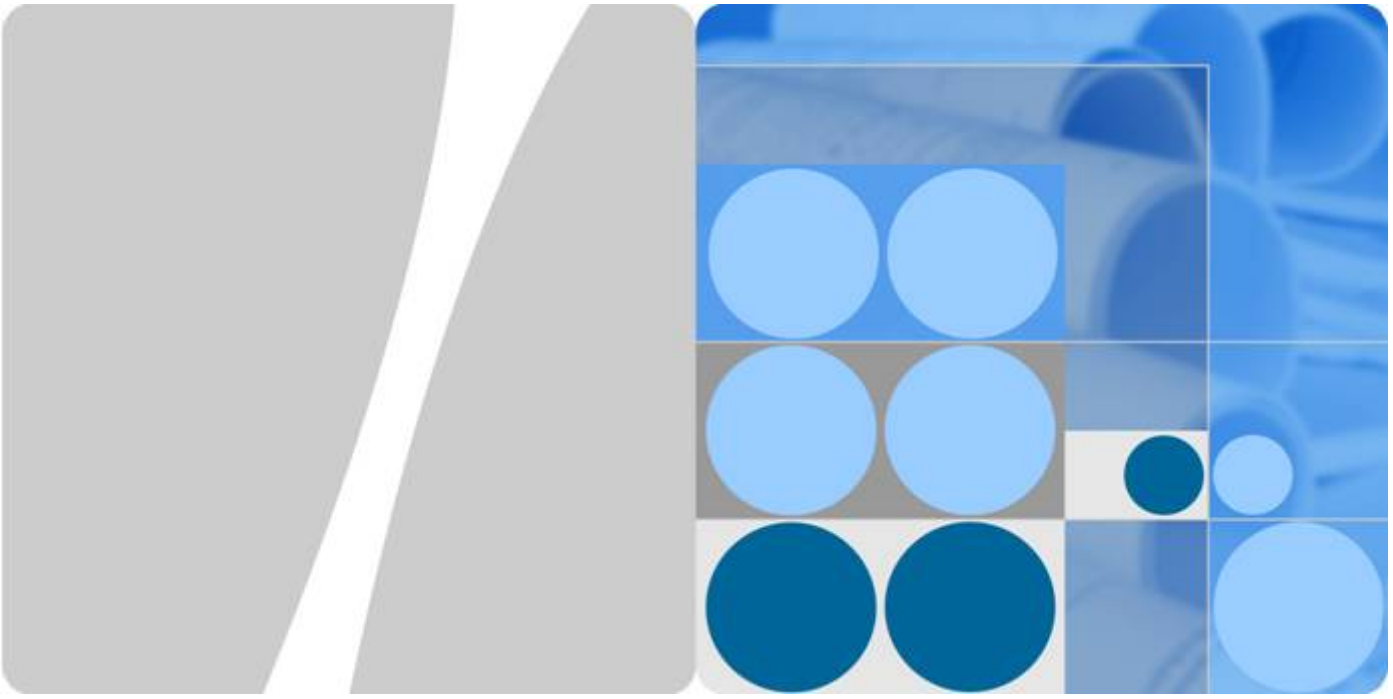


云容器引擎

用户指南

发布日期　2024-01-23

目 录

1 产品介绍

1.1 什么是云容器引擎

云容器引擎（Cloud Container Engine，简称CCE）提供高度可扩展的、高性能的企业级Kubernetes集群，支持运行Docker容器。借助云容器引擎，您可以在云上轻松部署、管理和扩展容器化应用程序。

为什么选择云容器引擎

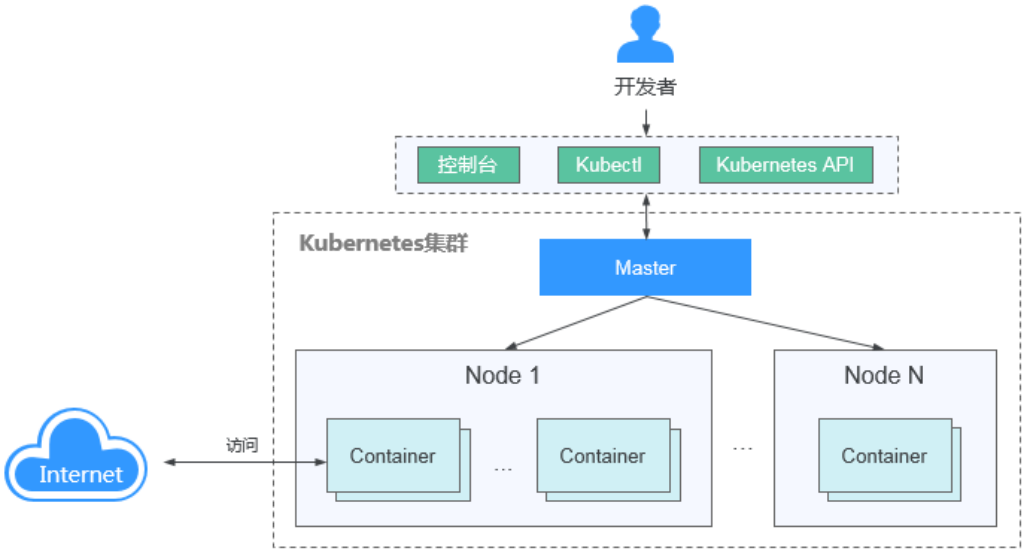
云容器引擎深度整合高性能的计算（ECS）、网络（VPC/EIP/ELB）、存储（EVS/OBS/SFS）等服务，支持多可用区（Available Zone，简称AZ）、多区域（Region）容灾等技术构建高可用**Kubernetes**集群。

更多选择理由，请参见**产品优势**和**应用场景**。

访问方式

您可以通过CCE控制台、Kubectl命令行、Kubernetes API使用云容器引擎服务。具体请参见**图1-1**。

图 1-1 使用云容器引擎



1.2 产品优势

云容器引擎的优势

云容器引擎是基于业界主流的Docker和Kubernetes开源技术构建的容器服务，提供众多契合企业大规模容器集群场景的功能，在系统可靠性、高性能、开源社区兼容性等多个方面具有独特的优势，满足企业在构建容器云方面的各种需求。

简单易用

- 通过WEB界面一键创建Kubernetes集群，支持管理虚拟机节点或裸金属节点，支持虚拟机与物理机混用场景。
- 一站式自动化部署和运维容器应用，整个生命周期都在容器服务内一站式完成。
- 通过Web界面轻松实现集群节点和工作负载的扩容和缩容，自由组合策略以应对多变的突发浪涌。
- 通过Web界面一键完成Kubernetes集群的升级。
- 深度集成Helm标准模板，真正实现开箱即用。

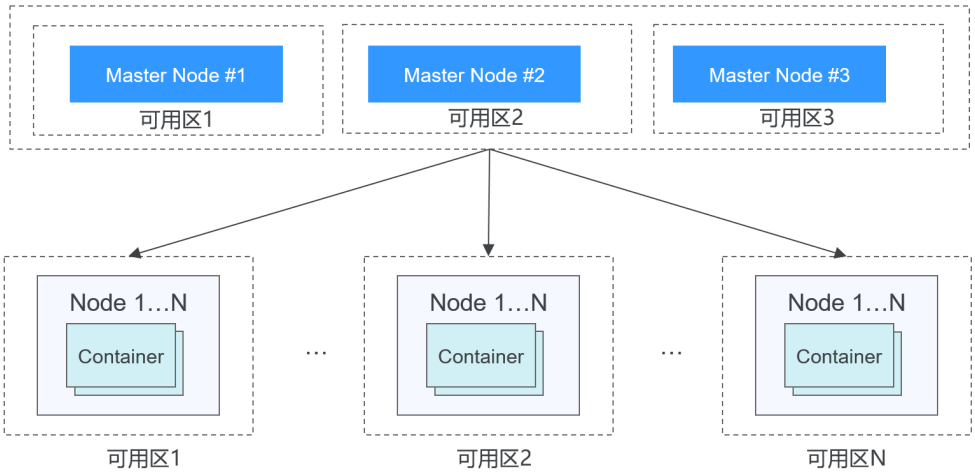
高性能

- 基于在计算、网络、存储、异构等方面多年的行业技术积累，提供业界领先的高性能云容器引擎，支撑您业务的高并发、大规模场景。
- 采用高性能裸金属NUMA架构和高速IB网卡，AI计算性能提升3-5倍以上。

安全可靠

- 高可靠：集群控制面支持3 Master HA高可用，3个Master节点可以处于不同可用区，保障您的业务高可用。集群内节点和工作负载支持跨可用区（AZ）部署，帮助您轻松构建多活业务架构，保证业务系统在主机故障、机房中断、自然灾害等情况下可持续运行，获得生产环境的高稳定性，实现业务系统零中断。

图 1-2 集群高可用



- 高安全：私有集群，完全由用户掌控，并深度整合IAM和Kubernetes RBAC能力，支持用户在界面为子用户设置不同的RBAC权限。

开放兼容

- 云容器引擎在Docker技术的基础上，为容器化的应用提供部署运行、资源调度、服务发现和动态伸缩等一系列完整功能，提高了大规模容器集群管理的便捷性。
- 云容器引擎基于业界主流的Kubernetes实现，完全兼容Kubernetes/Docker社区原生版本，与社区最新版本保持紧密同步，完全兼容Kubernetes API和Kubectl。

云容器引擎对比自建 Kubernetes 集群

表 1-1 云容器引擎和自建 kubernetes 集群对比

对比项	自建kubernetes集群	云容器引擎
易用性	自建kubernetes集群管理基础设施通常涉及安装、操作、扩展自己的集群管理软件、配置管理系统和监控解决方案，管理复杂。每次升级集群的过程都是巨大的调整，带来繁重的运维负担。	**简化集群管理，简单易用** 借助云容器引擎，您可以一键创建和升级Kubernetes容器集群，无需自行搭建Docker和Kubernetes集群。您可以通过云容器引擎自动化部署和一站式运维容器应用，使得应用的整个生命周期都在容器服务内高效完成。 您可以通过云容器引擎轻松使用深度集成的Helm标准模板，真正实现开箱即用。 您只需启动容器集群，并指定想要运行的任务，云容器引擎帮您完成所有的集群管理工作，让您可以集中精力开发容器化的应用程序。
可扩展性	自建kubernetes集群需要根据业务流量情况和健康情况人工确定容器服务的部署，可扩展性差。	**灵活集群托管，轻松实现扩缩容** 云容器引擎可以根据资源使用情况轻松实现集群节点和工作负载的自动扩容和缩容，并可以自由组合多种弹性策略，以应对业务高峰期的突发流量浪涌。

对比项	自建kubernetes集群	云容器引擎
可靠性	自建kubernetes集群多采用单控制节点，一旦出现故障，集群和业务将不可使用。	**服务高可用** 创建集群时若“高可用”选项配置为“是”，集群将创建3个Master节点，在单个控制节点发生故障时，集群仍然可用，从而保障您的业务高可用。
高效性	自建kubernetes集群需要自行搭建镜像仓库或使用第三方镜像仓库，镜像拉取方式多采用串行传输，效率低。	**镜像快速部署** 云容器引擎配合容器镜像服务，镜像拉取方式采用并行传输，确保高并发场景下能获得更快的下载速度，大幅提升容器交付效率。
成本	自建kubernetes集群需要投入资金构建、安装、运维、扩展自己的集群管理基础设施，成本开销大。	**云容器引擎成本低** 您只需支付用于存储和运行应用程序的基础设施资源（例如云服务器、云硬盘、弹性IP/带宽、负载均衡等）费用和容器集群控制节点费用。

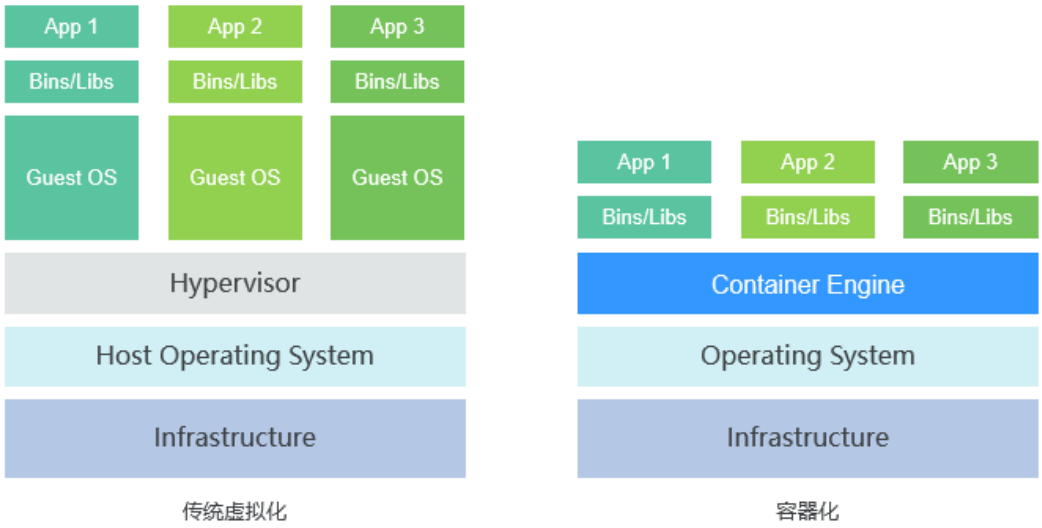
容器的优势

Docker使用Google公司推出的Go语言进行开发实现，基于Linux内核的cgroup，namespace，以及AUFS类的Union FS等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。

Docker在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等，极大的简化了容器的创建和维护。

传统虚拟机技术通过Hypervisor将宿主机的硬件资源（如内存、CPU、网络、磁盘等）进行了虚拟化分配，然后通过这些虚拟化的硬件资源组成了虚拟机，并在上面运行一个完整的操作系统，每个虚拟机需要运行自己的系统进程。而容器内的应用进程直接运行于宿主机操作系统内核，没有硬件资源虚拟化分配的过程，避免了额外的系统进程开销，因此使得Docker技术比虚拟机技术更为轻便、快捷。

图 1-3 传统虚拟化和容器化方式的对比



作为一种新兴的虚拟化方式，Docker跟虚拟机相比具有众多的优势：

更高效的利用系统资源

由于容器不需要进行硬件虚拟化分配以及运行完整操作系统等额外开销，Docker对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

更快速的启动时间

传统的虚拟机技术启动应用服务往往需要数分钟，而Docker容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。

一致的运行环境

开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些bug并未在开发过程中被发现。而Docker的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性。

持续交付和部署

对开发和运维（DevOps）人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。

使用Docker可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过Dockerfile来进行镜像构建，并结合持续集成（Continuous Integration）系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合持续部署（Continuous Delivery/Deployment） 系统进行自动部署。

而且使用Dockerfile使镜像构建透明化，不仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。

更轻松的迁移

由于Docker确保了执行环境的一致性，使得应用的迁移更加容易。Docker可以在很多平台上运行，无论是物理机、虚拟机，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

更轻松的维护和扩展

Docker使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker团队同各个开源项目团队一起维护了一大批高质量的官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

表 1-2 容器对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为MB	一般为GB
性能	接近原生	弱

特性	容器	虚拟机
系统支持量	单机支持上千个容器	一般几十个

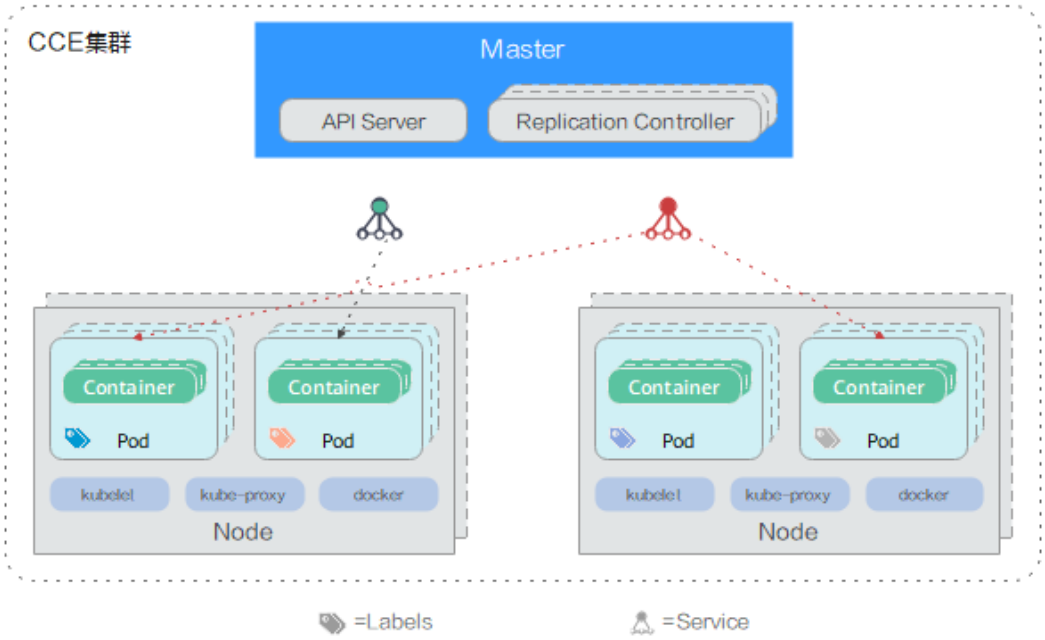
1.3 应用场景

1.3.1 基础设施与容器应用管理

应用场景

CCE集群支持管理X86资源池和ARM资源池，能方便的创建Kubernetes集群、部署您的容器化应用，以及方便的管理和维护。

图 1-4 CCE 集群



价值

通过容器化改造，使应用部署资源成本降低，提升应用的部署效率和升级效率，可以实现升级时业务不中断以及统一的自动化运维。

优势

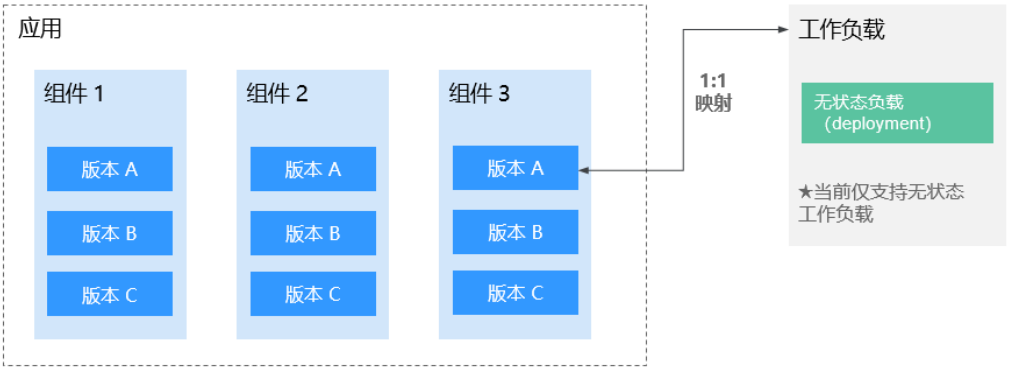
- 多种类型的容器部署

  支持部署无状态工作负载、有状态工作负载、守护进程集、普通任务、定时任务等。
- 应用升级

  支持替换升级、滚动升级（按比例、实例个数进行滚动升级）；支持升级回滚。

- 弹性伸缩
  支持节点和工作负载的弹性伸缩。

图 1-5 工作负载



1.3.2 秒级弹性伸缩

应用场景

- 电商客户遇到促销等活动期间，访问量激增，需及时、自动扩展云计算资源。
- 视频直播客户业务负载变化难以预测，需要根据CPU/内存使用率进行实时扩缩容。
- 游戏客户每天中午12点及晚上18:00-23:00间需求增长，需要定时扩容。

价值

云容器引擎可根据用户的业务需求预设策略自动调整计算资源，使云服务器或容器数量自动随业务负载增长而增加，随业务负载降低而减少，保证业务平稳健康运行，节省成本。

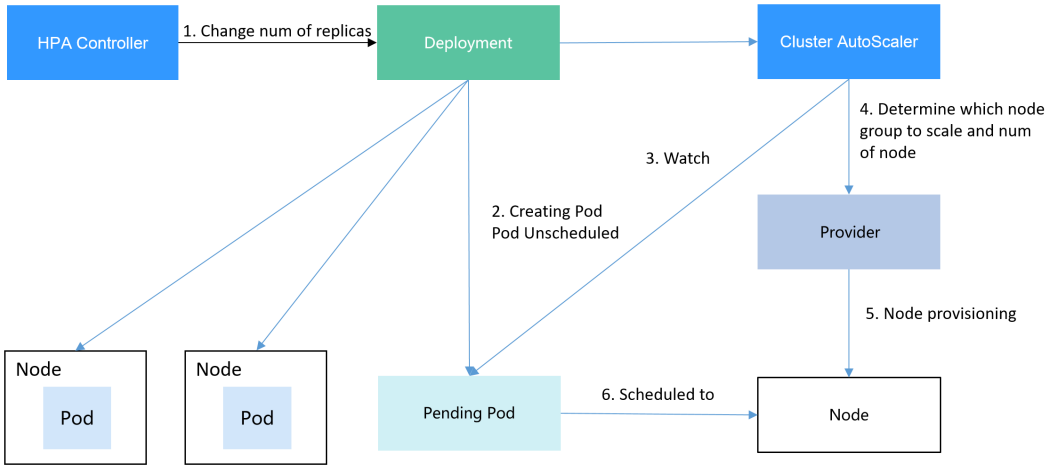
优势

- 自由灵活
  支持多种策略配置，业务流量达到扩容指标，秒级触发容器扩容操作。
- 高可用
  自动检测伸缩组中实例运行状况，启用新实例替换不健康实例，保证业务健康可用。
- 低成本
  只按照实际用量收取云服务器费用。

建议搭配使用

HPA（Horizontal Pod Autoscaling）+ CA（Cluster AutoScaling）

图 1-6 弹性伸缩场景



1.3.3 DevOps 持续交付

应用场景

当前IT行业发展日益快速，面对海量需求必须具备快速集成的能力。经过快速持续集成，才能保证不间断的补全用户体验，提升服务质量，为业务创新提供源源不断的动力。大量交付实践表明，不仅传统企业，甚至互联网企业都可能在持续集成方面存在研发效率低、工具落后、发布频率低等方面的问题，需要通过持续交付提高效率，降低发布风险。

价值

云容器引擎搭配容器镜像服务提供DevOps持续交付能力，能够基于代码源自动完成代码编译、镜像构建、灰度发布、容器化部署，实现一站式容器化交付流程，并可对接已有CI/CD，完成传统应用的容器化改造和部署。

优势

- 高效流程管理

  更优的流程交互设计，脚本编写量较传统CI/CD流水线减少80%以上，让CI/CD管理更高效。
- 灵活的集成方式

  提供丰富的接口便于与企业已有CI/CD系统进行集成，灵活适配企业的个性化诉求。
- 高性能

  全容器化架构设计，任务调度更灵活，执行效率更高。

建议搭配使用

容器镜像服务SWR + 对象存储服务OBS + 虚拟专用网络VPN

图 1-7 DevOps 持续交付场景

1.3.4 混合云架构

应用场景

- 多云部署、容灾备份

  为保证业务高可用，需要将业务同时部署在多个云的容器服务上，在某个云出现事故时，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上。
- 流量分发、弹性伸缩

  大型企业客户需要将业务同时部署在不同地域的云机房中，并能自动弹性扩容和缩容，以节约成本。
- 业务上云、数据库托管

对于金融、安全等行业用户，业务数据的敏感性要求将数据业务保留在本地的IDC中而将一般业务部署在云上，并需要进行统一管理。

- 开发与部署分离

  出于IP安全的考虑，用户希望将生产环境部署在云上，而将开发环境部署在本地的IDC。

价值

云容器引擎利用容器环境无关的特性，将容器服务实现网络互通和统一管理，应用和数据可在云上云下无缝迁移，并可统一运维多个云端资源，从而实现资源的灵活使用以及业务容灾等目的。

优势

- 云上容灾

  通过云容器引擎，可以将业务系统同时部署在多个云的容器服务上，统一流量分发，单云故障后能够自动将业务流量切换到其他云上，并能快速自动解决现网事故。

- 流量自动分发

  通过云容器引擎的统一流量分发机制，实现应用访问流量的地域亲和，降低业务访问时延，并需要能够将线下IDC中的业务在云上扩展，可根据业务流量峰值情况，自动弹性扩容和缩容。

- 计算与数据分离，能力共享

  通过云容器引擎，用户可以实现敏感业务数据与一般业务数据的分离，可以实现开发环境和生产环境分离，可以实现特殊计算能力与一般业务的分离，并能够实现弹性扩展和集群的统一管理，达到云上云下资源和能力的共享。

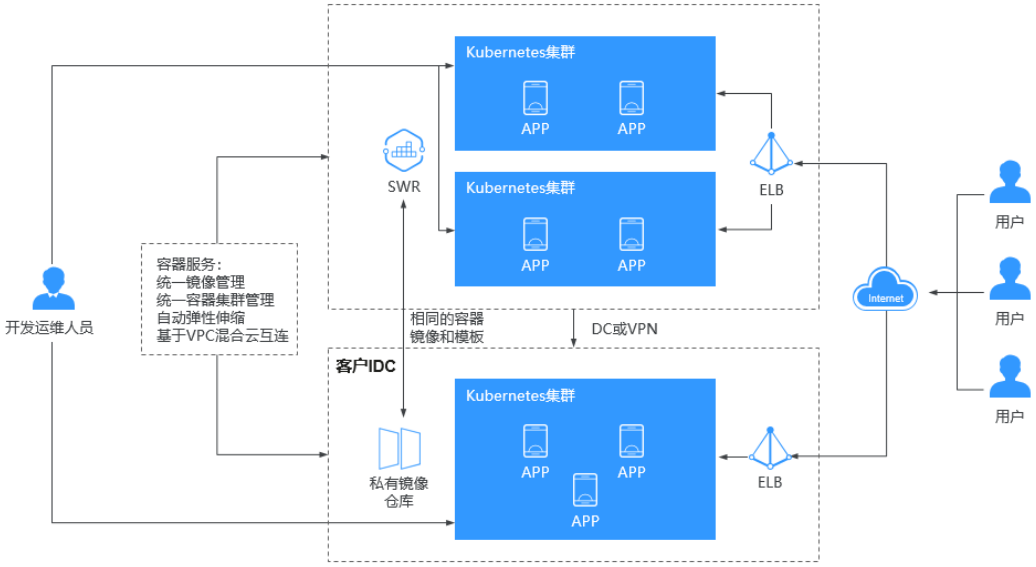
- 降低成本

  业务高峰时，利用云资源池快速扩容，用户不再需要根据流量峰值始终保持和维护大量资源，节约成本。

建议搭配使用

弹性云服务器ECS + 云专线DC + 虚拟专用网络VPN + 容器镜像服务SWR

图 1-8 混合云场景



1.4 约束与限制

本文主要为您介绍云容器引擎（CCE）集群使用过程中的一些限制。

集群/节点限制

- 集群一旦创建以后，不支持变更以下项：
  - 变更集群的控制节点数量，例如非高可用集群（控制节点数量为1）变更为高可用集群（控制节点数量为3）。
  - 变更控制节点可用区。
  - 变更集群的网络配置，如所在的虚拟私有云VPC、子网、容器网段、服务网段、IPv6、kubeproxy代理（转发）模式。
  - 变更网络模型，例如“容器隧道网络”更换为“VPC网络”。
- 由于ECS（节点）等CCE依赖的底层资源存在产品配额及库存限制，创建集群、扩容集群或者自动弹性扩容时，可能只有部分节点创建成功。
- ECS（节点）规格要求：CPU ≥ 2核且内存 ≥ 4GB。
- 通过搭建VPN方式访问CCE集群，需要注意VPN网络和集群所在的VPC网段、容器使用网段不能冲突。

网络限制

- 节点访问(NodePort)的使用约束：默认为VPC内网访问，如果需要通过公网访问该服务，请提前在集群的节点上绑定弹性IP。
- CCE中的负载均衡（LoadBalancer）访问类型使用弹性负载均衡 ELB提供网络访问，存在如下产品约束：
  - 自动创建的ELB实例建议不要被其他资源使用，否则会在删除时被占用，导致资源残留。
  - v1.15及之前版本集群使用的ELB实例请不要修改监听器名称，否则可能导致无法正常访问。

- 网络策略(NetworkPolicy)，存在如下产品约束：
  - 当前仅**容器隧道网络模型**的集群支持网络策略（NetworkPolicy）。网络策略可分为以下规则：
    - 入规则（Ingress）：所有版本均支持。
    - 出规则（Egress）：仅如下操作系统和集群版本支持设置Egress规则：

操作系统	集群版本	经验证的内核版本
CentOS	v1.23及以上	3.10.0-1062.18.1.el7.x86_64 3.10.0-1127.19.1.el7.x86_64 3.10.0-1160.25.1.el7.x86_64
EulerOS 2.5	v1.23及以上	3.10.0-862.14.1.5.h591.eulerosv2r7.x86_64 3.10.0-862.14.1.5.h687.eulerosv2r7.x86_64
EulerOS 2.9	v1.23及以上	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64 4.18.0-147.5.1.6.h766.eulerosv2r9.x86_64

  - 不支持对IPv6地址网络隔离。
  - 通过原地升级到支持Egress的集群版本，由于不会升级节点操作系统，会导致无法使用Egress，此种情况下，请重置节点。

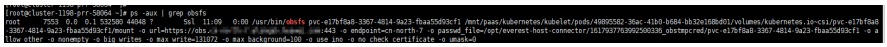
存储卷限制

- 云硬盘存储卷使用约束：
  - 云硬盘不支持跨可用区挂载，且不支持被多个工作负载、同一个工作负载的多个实例或多个任务使用。由于CCE集群各节点之间暂不支持共享盘的数据共享功能，多个节点挂载使用同一个云硬盘可能会出现读写冲突、数据缓存冲突等问题，所以创建无状态工作负载时，若使用了EVS云硬盘，建议工作负载只选择一个实例。
  - 1.19.10以下版本的集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，当新Pod被调度到另一个节点时，会导致之前Pod不能正常读写。

    1.19.10及以上版本集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，新Pod会因为无法挂载云硬盘导致无法成功启动。
- 文件存储卷使用约束：
  - 支持多个PV挂载同一个SFS或SFS Turbo，但有如下限制：
    - 多个不同的PVC/PV使用同一个底层SFS或SFS Turbo卷时，如果挂载至同一Pod使用，会因为PV的volumeHandle参数值相同导致无法为Pod挂载所有PVC，出现Pod无法启动的问题，请避免该使用场景。
    - PV中persistentVolumeReclaimPolicy参数需设置为Retain，否则可能存在一个PV删除时，级联删除底层卷，其他关联这个底层卷的PV会由于底层存储被删除导致使用出现异常。
    - 重复用底层存储时，建议在应用层做好多读多写的隔离保护，防止产生的数据覆盖和丢失。
- 对象存储卷使用约束如下：

- 使用并行文件系统和对象桶时，挂载点不支持修改属组和权限。
- CCE支持通过OBS SDK方式和PVC挂载方式使用OBS并行文件系统，其中PVC挂载方式是通过OBS服务提供的**obsfs工具**实现。每挂载一个并行文件系统对象存储卷，就会产生一个obsfs常驻进程。如下图所示：

  图 1-9 obsfs 常驻进程

  建议为每个obsfs进程预留1G的内存空间，例如4U8G的节点，则建议挂载obsfs并行文件系统的实例**不超过**8个。

  说明

  obsfs常驻进程是直接运行在节点上，如果消耗的内存超过了节点上限，则会导致节点异常。例如在4U8G的节点上，运行的挂载并行文件系统卷的实例超过100+，有极大概率会导致节点异常不可用。因此强烈建议控制单个节点上的挂载并行文件系统实例的数量。

- 本地持久卷使用约束：
  - 本地持久卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=2.1.23，推荐使用>=2.1.23版本。
  - 移除节点、删除节点、重置节点和缩容节点会导致与节点关联的本地持久存储卷类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。移除节点、删除节点、重置节点和缩容节点时使用了本地持久存储卷的Pod会从待删除、重置的节点上驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。节点重置完成后，Pod可能调度到重置好的节点上，此时Pod会一直处于creating状态，因为该PVC对应的底层逻辑卷已不存在。
  - 请勿在节点上手动删除对应的存储池或卸载数据盘，否则会导致数据丢失等异常情况。
  - 本地持久卷不支持被多个工作负载或多个任务同时挂载。
- 本地临时卷使用约束：
  - 本地临时卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=1.2.29。
  - 请勿在节点上手动删除对应的存储池或卸载数据盘，否则会导致数据丢失等异常情况。
  - 请确保节点上Pod不要挂载/var/lib/kubelet/pods/目录，否则可能会导致使用了临时存储卷的Pod无法正常删除。
- 快照与备份使用约束：
  - 快照功能**仅支持v1.15及以上版本**的集群，且需要安装基于CSI的everest插件才可以使用。
  - 基于快照创建的云硬盘，其子类型（普通IO/高IO/超高IO）、是否加密、磁盘模式（VBD/SCSI）、共享性(非共享/共享)、容量等都要与快照关联母盘保持一致，这些属性查询和设置出来后不能够修改。
  - 只有可用或正在使用状态的磁盘能创建快照，且单个磁盘最大支持创建7个快照。
  - 创建快照功能仅支持使用everest插件提供的存储类（StorageClass名称以csi开头）创建的PVC。使用Flexvolume存储类（StorageClass名为ssd、sas、sata）创建的PVC，无法创建快照。

- 加密磁盘的快照数据以加密方式存放，非加密磁盘的快照数据以非加密方式存放。

插件限制

CCE插件采用Helm模板方式部署，修改或升级插件请从插件配置页面或开放的插件管理API进行操作。请勿直接后台直接修改插件相关资源，以免插件异常或引入其他非预期问题。

CCE 集群配额限制

针对每个用户，云容器引擎的集群在每个地域分配了固定配额。

限制项	普通用户限制
单Region下集群总数	50
单集群最大节点数（集群管理规模）	可选择50节点、200节点、1000节点或2000节点多种管理规模。
单节点最大实例数	256
单个集群管理的最大Pod数	10万Pod

依赖底层云产品配额限制

限制大类	限制项	普通用户限制
计算	实例数	1000
	核心数	8000核
	RAM容量 (MB)	16384000
网络	一个用户创建虚拟私有云的数量	5
	一个用户创建子网的数量	100
	一个用户拥有的安全组数量	100
	一个用户拥有的安全组规则数量	5000
	一个路由表里拥有的路由数量	100
	一个虚拟私有云拥有路由数量	100
	一个区域下的对等连接数量	50
	一个用户拥有网络ACL数量	200
	一个用户创建二层连接网关的数量	5
负载均衡	弹性负载均衡	50
	弹性负载均衡监听器	100

限制大类	限制项	普通用户限制
	弹性负载均衡证书	120
	弹性负载均衡转发策略	500
	弹性负载均衡后端主机组	500
	弹性负载均衡后端服务器	500

1.5 权限管理

CCE权限管理是在统一身份认证服务（IAM）与Kubernetes的角色访问控制（RBAC）的能力基础上，打造的细粒度权限管理功能，支持基于IAM的细粒度权限控制和IAM Token认证，支持集群级别、命名空间级别的权限控制，帮助用户便捷灵活的对租户下的IAM用户、用户组设定不同的操作权限。

CCE的权限管理包括“集群权限”和“命名空间权限”两种能力，能够从集群和命名空间层面对用户组或用户进行细粒度授权，具体解释如下：

- **集群权限**：是基于IAM系统策略的授权，可以通过用户组功能实现IAM用户的授权。用户组是用户的集合，通过集群权限设置可以让某些用户组操作集群（如创建/删除集群、节点、节点池、模板、插件等），而让某些用户组仅能查看集群。

  集群权限涉及CCE非Kubernetes API，支持IAM细粒度策略、企业项目管理相关能力。
- **命名空间权限**：是基于Kubernetes RBAC能力的授权，通过权限设置可以让不同的用户或用户组拥有操作不同Kubernetes资源的权限（**如工作负载、任务、服务等Kubernetes原生资源**）。同时CCE基于开源能力进行了增强，可以支持基于IAM用户或用户组粒度进行RBAC授权、IAM token直接访问API进行RBAC认证鉴权。

  命名空间权限涉及CCE Kubernetes API，基于Kubernetes RBAC能力进行增强，支持对接IAM用户/用户组进行授权和认证鉴权，但与IAM细粒度策略独立，详见**Kubernetes RBAC**。

⚠ 注意

- 集群权限仅针对与集群相关的资源（如集群、节点等）有效，您必须确保同时配置了**命名空间权限**，才能有操作Kubernetes资源（如工作负载、任务、Service等）的权限。
- 任何用户创建v1.11.7-r2或以上版本集群后，CCE会自动为该用户添加该集群的所有命名空间的cluster-admin权限，也就是说该用户允许对集群以及所有命名空间中的全部资源进行完全控制。

集群权限（IAM 系统策略授权）

默认情况下，管理员创建的IAM用户没有任何权限，需要将其加入用户组，并给用户组授予策略或角色，才能使得用户组中的用户获得对应的权限，这一过程称为授权。授权后，用户就可以基于被授予的权限对云服务进行操作。

CCE部署时通过物理区域划分，为项目级服务。授权时，“作用范围”需要选择“区域级项目”，然后在指定区域对应的项目中设置相关权限，并且该权限仅对此项目生效；如果在“所有项目”中设置权限，则该权限在所有区域项目中都生效。访问CCE时，需要先切换至授权区域。

权限根据授权精细程度分为角色和策略。

- 角色：IAM最初提供的一种根据用户的工作职能定义权限的粗粒度授权机制。该机制以服务为粒度，提供有限的服务相关角色用于授权。由于云各服务之间存在业务依赖关系，因此给用户授予角色时，可能需要一并授予依赖的其他角色，才能正确完成业务。角色并不能满足用户对精细化授权的要求，无法完全达到企业对权限最小化的安全管控要求。
- 策略：IAM最新提供的一种细粒度授权的能力，可以精确到具体服务的操作、资源以及请求条件等。基于策略的授权是一种更加灵活的授权方式，能够满足企业对权限最小化的安全管控要求。例如：针对CCE服务，租户（Domain）能够控制用户仅能对某一类集群和节点资源进行指定的管理操作。

如表1-3所示，包括了CCE的所有系统权限。

表 1-3 CCE 系统权限

系统角色/策略名称	描述	类别	依赖关系
CCE Administrator	具有CCE集群及集群下所有资源（包含集群、节点、工作负载、任务、服务等）的读写权限。	系统角色	拥有该权限的用户必须同时拥有以下权限： **全局服务：** OBS Buckets Viewer、OBS Administrator。 **区域级项目：** Tenant Guest、Server Administrator、ELB Administrator、SFS Administrator、SWR Admin、APM FullAccess。 **说明** 如果同时拥有NAT Gateway Administrator权限，则可以在集群中使用NAT网关的相关功能。
CCE FullAccess	CCE服务集群相关资源的普通操作权限，不包括集群（启用Kubernetes RBAC鉴权）的命名空间权限，不包括委托授权、生成集群证书等管理员角色的特权操作。	策略	无
CCE ReadOnly Access	CCE服务集群相关资源的查看权限，不包括集群（启用Kubernetes RBAC鉴权）的命名空间权限。	策略	无

表 1-4 CCE 常用操作与系统权限的关系

操作	CCE ReadOnlyAccess	CCE FullAccess	CCE Administrator
创建集群	x	√	√
删除集群	x	√	√
更新集群，如后续允许集群支持RBAC，调度参数更新等	x	√	√
升级集群	x	√	√
唤醒集群	x	√	√
休眠集群	x	√	√
查询集群列表	√	√	√
查询集群详情	√	√	√
添加节点	x	√	√
删除节点/批量删除节点	x	√	√
更新节点，如更新节点名称	x	√	√
查询节点详情	√	√	√
查询节点列表	√	√	√
查询任务列表（集群层面的job）	√	√	√
删除任务/批量删除任务（集群层面的job）	x	√	√
查询任务详情（集群层面的job）	√	√	√
创建存储	x	√	√
删除存储	x	√	√
操作所有kubernetes资源。	√（需Kubernetes RBAC授权）	√（需Kubernetes RBAC授权）	√
ECS（弹性云服务器）服务的所有权限。	x	√	√

操作	CCE ReadOnlyAccess	CCE FullAccess	CCE Administrator
EVS（云硬盘）的所有权限。 可以将云硬盘挂载到云服务器，并可以随时扩容云硬盘容量	x	√	√
VPC（虚拟私有云）的所有权限。 创建的集群需要运行在虚拟私有云中，创建命名空间时，需要创建或关联VPC，创建在命名空间的容器都运行在VPC之内。	x	√	√
ECS（弹性云服务器）所有资源详情的查看权限。 CCE中的一个节点就是具有多个云硬盘的一台弹性云服务器	√	√	√
ECS（弹性云服务器）所有资源列表的查看权限。	√	√	√
EVS（云硬盘）所有资源详情的查看权限。可以将云硬盘挂载到云服务器，并可以随时扩容云硬盘容量	√	√	√
EVS（云硬盘）所有资源列表的查看权限。	√	√	√
VPC（虚拟私有云）所有资源详情的查看权限。 创建的集群需要运行在虚拟私有云中，创建命名空间时，需要创建或关联VPC，创建在命名空间的容器都运行在VPC之内	√	√	√
VPC（虚拟私有云）所有资源列表的查看权限。	√	√	√
ELB（弹性负载均衡）服务所有资源详情的查看权限。	x	x	√
ELB（弹性负载均衡）服务所有资源列表的查看权限。	x	x	√

操作	CCE ReadOnlyAccess	CCE FullAccess	CCE Administrator
SFS（弹性文件服务）服务所有资源详情的查看权限。	√	√	√
SFS（弹性文件服务）服务所有资源列表额查看权限。	√	√	√
AOM（应用运维管理）服务所有资源详情的查看权限。	√	√	√
AOM（应用运维管理）服务所有资源列表的查看权限。	√	√	√
AOM（应用运维管理）服务自动扩缩容规则的所有操作权限。	√	√	√

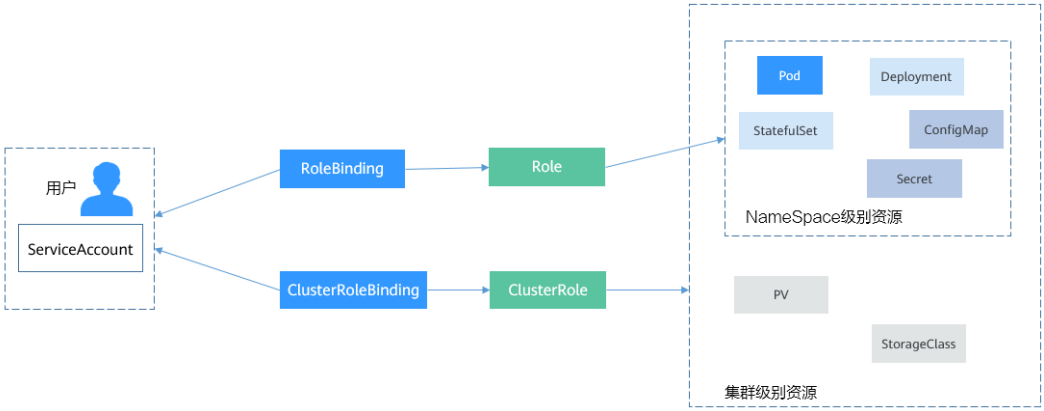
命名空间权限（kubernetes RBAC 授权）

命名空间权限是基于Kubernetes RBAC能力的授权，通过权限设置可以让不同的用户或用户组拥有操作不同Kubernetes资源的权限。Kubernetes RBAC API定义了四种类型：Role、ClusterRole、RoleBinding与ClusterRoleBinding，这四种类型之间的关系和简要说明如下：

- Role：角色，其实是定义一组对Kubernetes资源（命名空间级别）的访问规则。
- RoleBinding：角色绑定，定义了用户和角色的关系。
- ClusterRole：集群角色，其实是定义一组对Kubernetes资源（集群级别，包含全部命名空间）的访问规则。
- ClusterRoleBinding：集群角色绑定，定义了用户和集群角色的关系。

Role和ClusterRole指定了可以对哪些资源做哪些动作，RoleBinding和ClusterRoleBinding将角色绑定到特定的用户、用户组或ServiceAccount上。如下图所示。

图 1-10 角色绑定



在CCE控制台中可以授予用户或用户组命名空间权限，可以对某一个命名空间或全部命名空间授权，CCE控制台中默认提供如下ClusterRole。

- view（只读权限）：对全部或所选命名空间下大多数资源的只读权限。
- edit（开发权限）：对全部或所选命名空间下多数资源的读写权限。当配置在全部命名空间时能力与运维权限一致。
- admin（运维权限）：对全部命名空间下大多数资源的读写权限，对节点、存储卷，命名空间和配额管理的只读权限。
- cluster-admin（管理员权限）：对全部命名空间下所有资源的读写权限。
- drainage-editor：节点排水操作权限，可执行节点排水。
- drainage-viewer：节点排水只读权限，仅可查看节点排水状态，无法执行节点排水。

除了使用cluster-admin、admin、edit、view这4个最常用的clusterrole外，您还可以通过定义Role和RoleBinding来进一步对命名空间中不同类别资源（如Pod、Deployment、Service等）的增删改查权限进行配置，从而做到更加精细化的权限控制。

1.6 计费说明

计费项

云容器引擎（CCE）本身不收取任何费用，但在使用过程中会创建相关资源（如节点、带宽等），您需要为您使用的这些资源付费。CCE相关资源的计费项分为如下两部分：

1. **集群：**控制节点资源费用，按照每个集群的类型（虚拟机或裸金属、控制节点数）、集群规模（最大支持的节点数）的差异收取不同的费用。

   说明

   集群规模是指用户在集群下创建和购买的云主机或者裸金属服务器的数量。

2. **IaaS基础设施：**集群工作节点所使用的IaaS基础设施费用，包括集群创建使用过程中自动创建或手动加入的相关资源，如云服务器、云硬盘、弹性IP/带宽、负载均衡等，价格参照相应产品价格表。

计费模式

CCE支持按需计费计费模式。

- 按需计费：一种先使用后付费的方式，从“开通”开启计费到“删除”结束计费，按实际购买时长计费。这种购买方式比较灵活，您可以按需取用资源，随时开启和释放，无需提前购买大量资源。

  说明

  关于CCE集群休眠或节点关机后的收费说明：

  - **集群休眠：**集群休眠后，控制节点资源费用将停止收费，集群所属的云硬盘、绑定的弹性IP、带宽等资源按各自的计费方式（“按需付费”）进行收费。
  - **节点关机：**集群休眠后，集群中的工作节点（即ECS）并不会自动关机，如需关机可勾选“关机集群下所有节点”选项。您也可以在集群休眠后自行登录ECS控制台将节点关机。

须知

- 以集群作为计费量纲，根据集群类型和规模大小，按阶梯计费。
- 提供给客户进行续费与充值的时间，当您的按需资源欠费时提供宽限期和保留期。

1.7 基本概念

1.7.1 基本概念

云容器引擎（Cloud Container Engine，简称CCE）提供高度可扩展的、高性能的企业级Kubernetes集群，支持运行Docker容器。借助云容器引擎，您可以在云上轻松部署、管理和扩展容器化应用程序。

云容器引擎提供Kubernetes原生API，支持使用kubectl，且提供图形化控制台，让您能够拥有完整的端到端使用体验，使用云容器引擎前，建议您先了解相关的基本概念。

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、负载均衡等云资源。您可以理解为集群是“同一个子网中一个或多个弹性云服务器（又称：节点）”通过相关技术组合而成的计算机群体，为容器运行提供了计算资源池。

节点（Node）

每一个节点对应一台服务器（可以是虚拟机实例或者物理服务器），容器应用运行在节点上。节点上运行着Agent代理程序（kubelet），用于管理节点上运行的容器实例。集群中的节点数量可以伸缩。

节点池（NodePool）

节点池是集群中具有相同配置的一组节点，一个节点池包含一个节点或多个节点。

虚拟私有云（VPC）

虚拟私有云是通过逻辑方式进行网络隔离，提供安全、隔离的网络环境。您可以在VPC中定义与传统网络无差别的虚拟网络，同时提供弹性IP、安全组等高级网络服务。

安全组

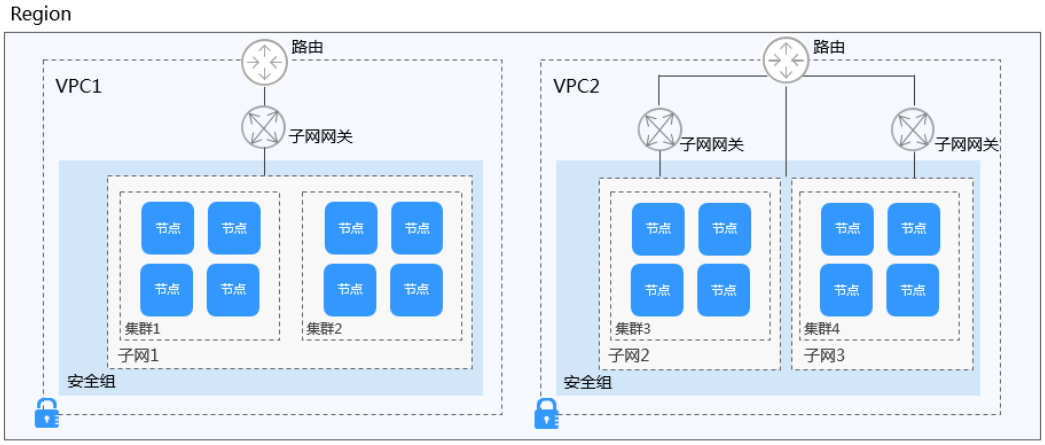
安全组是一个逻辑上的分组，为同一个VPC内具有相同安全保护需求并相互信任的弹性云服务器提供访问策略。安全组创建后，用户可以在安全组中定义各种访问规则，当弹性云服务器加入该安全组后，即受到这些访问规则的保护。

集群、虚拟私有云、安全组和节点的关系

如图1-11，同一个Region下可以有多个虚拟私有云（VPC）。虚拟私有云由一个个子网组成，子网与子网之间的网络交互通过子网网关完成，而集群就是建立在某个子网中。因此，存在以下三种场景：

- 不同集群可以创建在不同的虚拟私有云中。
- 不同集群可以创建在同一个子网中。
- 不同集群可以创建在不同的子网中。

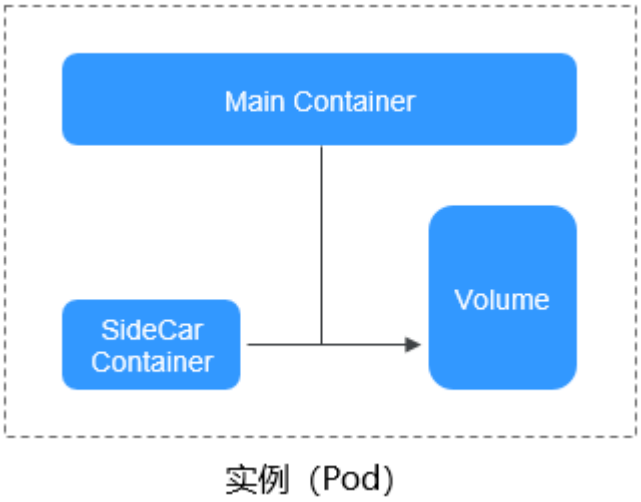
图 1-11 集群、VPC、安全组和节点的关系



实例（Pod）

实例（Pod）是 Kubernetes 部署应用或服务的最小的基本单位。一个Pod 封装多个应用容器（也可以只有一个容器）、存储资源、一个独立的网络 IP 以及管理控制容器运行方式的策略选项。

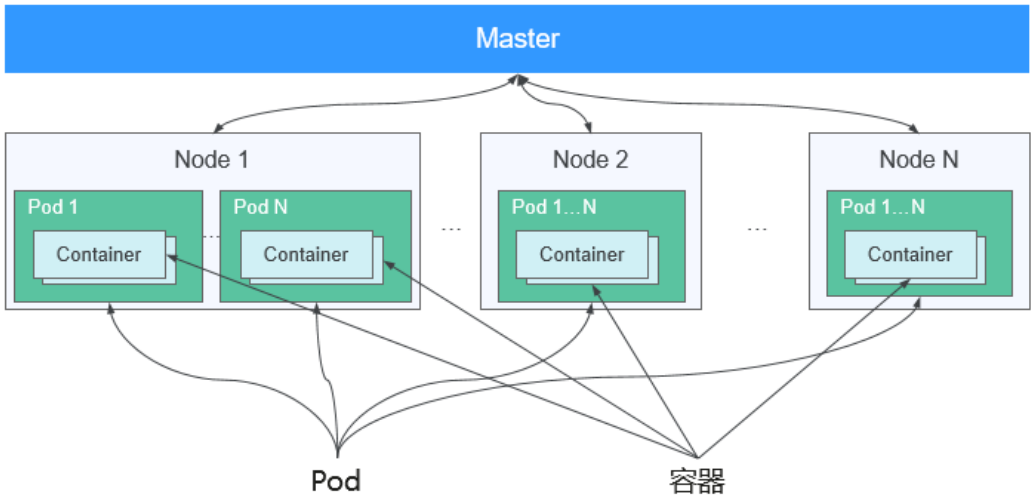
图 1-12 实例（Pod）



容器（Container）

一个通过 Docker 镜像创建的运行实例，一个节点可运行多个容器。容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。

图 1-13 实例 Pod、容器 Container、节点 Node 的关系



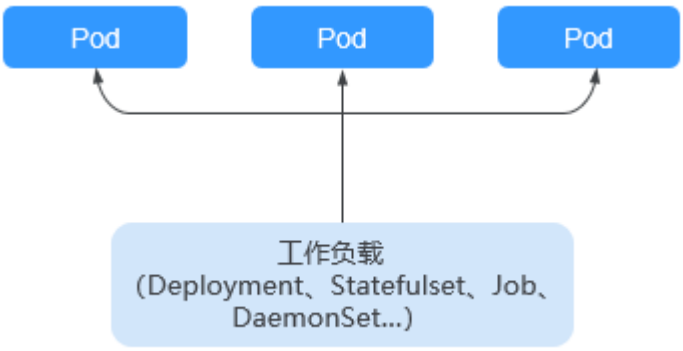
工作负载

工作负载是在Kubernetes上运行的应用程序。无论您的工作负载是单个组件还是协同工作的多个组件，您都可以在Kubernetes上的一组Pod中运行它。在Kubernetes中，工作负载是对一组Pod的抽象模型，用于描述业务的运行载体，包括Deployment、Statefulset、Daemonset、Job、CronJob等多种类型。

- **无状态工作负载：**即kubernetes中的“Deployment”，无状态工作负载支持弹性伸缩与滚动升级，适用于实例完全独立、功能相同的场景，如：nginx、wordpress等。

- **有状态工作负载：**即kubernetes中的“StatefulSet”，有状态工作负载支持实例有序部署和删除，支持持久化存储，适用于实例间存在互访的场景，如ETCD、mysql-HA等。
- **创建守护进程集：**即kubernetes中的“DaemonSet”，守护进程集确保全部（或者某些）节点都运行一个Pod实例，支持实例动态添加到新节点，适用于实例在每个节点上都需要运行的场景，如ceph、fluentd、Prometheus Node Exporter等。
- **普通任务：**即kubernetes中的“Job”，普通任务是一次性运行的短任务，部署完成后即可执行。使用场景为在创建工作负载前，执行普通任务，将镜像上传至镜像仓库。
- **定时任务：**即kubernetes中的“CronJob”，定时任务是按照指定时间周期运行的短任务。使用场景为在某个固定时间点，为所有运行中的节点做时间同步。

图 1-14 工作负载与 Pod 的关系

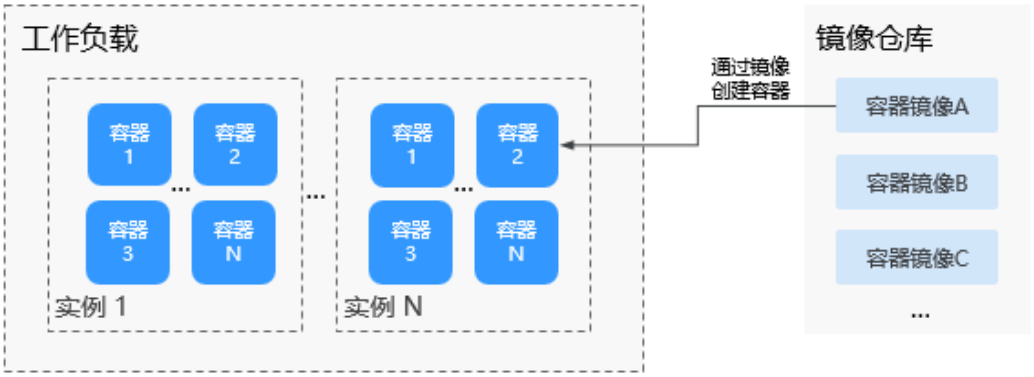


镜像（Image）

Docker镜像是一个模板，是容器应用打包的标准格式，用于创建Docker容器。或者说，Docker镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。在部署容器化应用时可以指定镜像，镜像可以来自于 Docker Hub、容器镜像服务或者用户的私有Registry。例如一个Docker镜像可以包含一个完整的Ubuntu操作系统环境，里面仅安装了用户需要的应用程序及其依赖文件。

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

图 1-15 镜像、容器、工作负载的关系



命名空间（Namespace）

命名空间是对一组资源和对象的抽象整合。在同一个集群内可创建不同的命名空间，不同命名空间中的数据彼此隔离。使得它们既可以共享同一个集群的服务，也能够互不干扰。例如：

- 可以将开发环境、测试环境的业务分别放在不同的命名空间。
- 常见的pods，services，replication controllers和deployments等都是属于某一个namespace的（默认是default），而node，persistentVolumes等则不属于任何namespace。

服务（Service）

Service是将运行在一组 Pods 上的应用程序公开为网络服务的抽象方法。

使用Kubernetes，您无需修改应用程序即可使用不熟悉的服务发现机制。 Kubernetes为Pods提供自己的IP地址和一组Pod的单个DNS名称，并且可以在它们之间进行负载平衡。

Kubernetes允许指定一个需要的类型的Service，类型 的取值以及行为如下：

- ClusterIP：集群内访问。通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 ServiceType。
- NodePort：节点访问。通过每个Node上的 IP 和静态端口（NodePort）暴露服务。NodePort服务会路由到ClusterIP服务，这个ClusterIP服务会自动创建。通过请求 <NodeIP>:<NodePort>，可以从集群的外部访问一个 NodePort 服务。
- LoadBalancer：负载均衡。使用云提供商的负载均衡器，可以向外部暴露服务。外部的负载均衡器可以路由到NodePort服务和ClusterIP服务。

七层负载均衡（Ingress）

Ingress是为进入集群的请求提供路由规则的集合，可以给service提供集群外部访问的URL、负载均衡、SSL终止、HTTP路由等。

网络策略（NetworkPolicy）

NetworkPolicy提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。

配置项（Configmap）

ConfigMap用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。ConfigMap跟secret很类似，但它可以更方便地处理不包含敏感信息的字符串。

密钥（Secret）

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

标签（Label）

标签其实就一对 key/value，被关联到对象上，比如Pod。标签的使用倾向于能够标示对象的特殊特点，并且对用户而言是有意义的，但是标签对内核系统是没有直接意义的。

选择器（LabelSelector）

Label selector是Kubernetes核心的分组机制，通过label selector客户端/用户能够识别一组有共同特征或属性的资源对象。

注解（Annotation）

Annotation与Label类似，也使用key/value键值对的形式进行定义。

Label具有严格的命名规则，它定义的是Kubernetes对象的元数据（Metadata），并且用于Label Selector。

Annotation则是用户任意定义的“附加”信息，以便于外部工具进行查找。

存储卷（PersistentVolume）

PersistentVolume（PV）是集群之中的一块网络存储。跟 Node 一样，也是集群的资源。

存储声明（PersistentVolumeClaim）

PV 是存储资源，而 PersistentVolumeClaim (PVC) 是对 PV 的请求。PVC 跟 Pod 类似：Pod 消费 Node 资源，而 PVC 消费 PV 资源；Pod 能够请求 CPU 和内存资源，而 PVC 请求特定大小和访问模式的数据卷。

弹性伸缩（HPA）

Horizontal Pod Autoscaling，简称HPA，是Kubernetes中实现POD水平自动伸缩的功能。Kubernetes集群可以通过Replication Controller的scale机制完成服务的扩容或缩容，实现具有伸缩性的服务。

亲和性与反亲和性

在应用没有容器化之前，原先一个虚机上会装多个组件，进程间会有通信。但在做容器化拆分的时候，往往直接按进程拆分容器，比如业务进程一个容器，监控日志处理或者本地数据放在另一个容器，并且有独立的生命周期。这时如果分布在网络中两个较远的点，请求经过多次转发，性能会很差。

- 亲和性：可以实现就近部署，增强网络能力实现通信上的就近路由，减少网络的损耗。如：应用A与应用B两个应用频繁交互，所以有必要利用亲和性让两个应用的尽可能的靠近，甚至在一个节点上，以减少因网络通信而带来的性能损耗。
- 反亲和性：主要是出于高可靠性考虑，尽量分散实例，某个节点故障的时候，对应用的影响只是 N 分之一或者只是一个实例。如：当应用采用多副本部署时，有必要采用反亲和性让各个应用实例打散分布在各个节点上，以提高HA。

节点亲和性（NodeAffinity）

通过选择标签的方式，可以限制pod被调度到特定的节点上。

节点反亲和性（NodeAntiAffinity）

通过选择标签的方式，可以限制pod不被调度到特定的节点上。

工作负载亲和性（PodAffinity）

指定工作负载部署在相同节点。用户可根据业务需求进行工作负载的就近部署，容器间通信就近路由，减少网络消耗。

工作负载反亲和性（PodAntiAffinity）

指定工作负载部署在不同节点。同个工作负载的多个实例反亲和部署，减少宕机影响；互相干扰的应用反亲和部署，避免干扰。

资源配额（Resource Quota）

资源配额（Resource Quotas）是用来限制用户资源用量的一种机制。

资源限制（Limit Range）

默认情况下，K8s中所有容器都没有任何CPU和内存限制。LimitRange(简称limits)用来给Namespace增加一个资源限制，包括最小、最大和默认资源。在pod创建时，强制执行使用limits的参数分配资源。

环境变量

环境变量是指容器运行环境中设定的一个变量，您可以在创建容器模板时设定不超过30个的环境变量。环境变量可以在工作负载部署后修改，为工作负载提供了极大的灵活性。

在CCE中设置环境变量与Dockerfile中的“ENV”效果相同。

模板（Chart）

Kubernetes集群可以通过**Helm**实现软件包管理，这里的Kubernetes软件包被称为模板（Chart）。Helm对于Kubernetes的关系类似于在Ubuntu系统中使用的apt命令，或是在CentOS系统中使用的yum命令，它能够快速查找、下载和安装模板（Chart）。

模板（Chart）是一种Helm的打包格式，它只是描述了一组相关的集群资源定义，而不是真正的容器镜像包。模板中仅仅包含了用于部署Kubernetes应用的一系列YAML文件，您可以在Helm模板中自定义应用程序的一些参数设置。在模板的实际安装过程中，Helm会根据模板中的YAML文件定义在集群中部署资源，相关的容器镜像并不会包含在模板包中，而是依旧从YAML中定义好的镜像仓库中进行拉取。

对于应用开发者而言，需要将容器镜像包发布到镜像仓库，并通过Helm的模板将安装应用时的依赖关系统一打包，预置一些关键参数，来降低应用的部署难度。

对于应用使用者而言，可以使用Helm查找模板（Chart）包并支持调整自定义参数。Helm会根据模板包中的YAML文件直接在集群中安装应用程序及其依赖，应用使用者不用编写复杂的应用部署文件，即可以实现简单的应用查找、安装、升级、回滚、卸载。

1.7.2 CCE 与原生 Kubernetes 名词对照

Kubernetes，简称K8s，是开源的容器集群管理系统，可以实现容器集群的自动化部署、自动扩缩容、维护等功能。它既是一款容器编排工具，也是全新的基于容器技术的分布式架构领先方案。在Docker技术的基础上，为容器化的应用提供部署运行、资源调度、服务发现和动态伸缩等功能，提高了大规模容器集群管理的便捷性。

本文主要为您介绍云容器引擎CCE与原生Kubernetes名词对照情况和简单解释。

表 1-5 CCE 与原生 Kubernetes 名词对照

云容器引擎CCE	原生Kubernetes
集群	Cluster
节点	Node
节点池	NodePool
容器	Container
镜像	Image
命名空间	Namespace
无状态工作负载	Deployment
有状态工作负载	StatefulSet
守护进程集	DaemonSet
普通任务	Job
定时任务	CronJob
实例（容器组）	Pod
服务（Service）	Service
虚拟集群IP	Cluster IP
节点端口	NodePort
负载均衡	LoadBalancer
七层负载均衡（路由）	Ingress
网络策略	NetworkPolicy
模板	Template

云容器引擎CCE	原生Kubernetes
配置项	ConfigMap
密钥	Secret
标签	Label
选择器	LabelSelector
注解	Annotation
存储卷	PersistentVolume
存储声明	PersistentVolumeClaim
弹性伸缩	HPA
节点亲和性	NodeAffinity
节点反亲和性	NodeAntiAffinity
工作负载亲和性	PodAffinity
工作负载反亲和性	PodAntiAffinity
触发器	Webhook
终端节点	Endpoint
资源配额	Resource Quota
资源限制	Limit Range

1.7.3 CCE Turbo 集群

没有容器化，就没有应用现代化。以容器为核心的云原生基础设施，不仅提供更高效的资源，还能把开发运维人员从资源的调配和运维中解放出来，聚焦于应用和业务创新。容器全面规模化应用的同时也对性能、弹性、调度能力提出了更高的要求。

CCE Turbo容器集群在计算、网络和调度上全方位加速，让容器真正成为企业应用创新的强劲引擎。

集群优势

- **计算加速**
  软硬协同基础设施，更少资源可实现更好的性能。
- **网络加速**
  云原生网络2.0，无损网络让应用间通信更流畅。
- **调度加速**
  智能混合调度，简单高效地管理应用与资源。
- **安全隔离**
  安全容器引擎，使应用拥有虚机级安全隔离。

CCE Turbo 集群与 CCE 集群对比

CCE支持多种类型的集群创建，以满足您各种业务需求，如下为CCE Turbo集群与CCE集群区别：

表 1-6 集群类型对比

维度	子维度	CCE Turbo集群	CCE集群
集群	定位	面向云原生2.0的新一代容器集群产品，计算、网络、调度全面加速	标准版本集群，提供商用级的容器集群服务
	节点形态	支持虚拟机	支持虚拟机和裸金属服务器混合
网络	网络模型	**云原生网络2.0**：面向大规模和高性能的场景。 组网规模最大支持2000节点	**云原生网络1.0**：面向性能和规模要求不高的场景。 ● 容器隧道网络模式 ● VPC网络模式
	网络性能	VPC网络和容器网络融合，性能无损耗	VPC网络叠加容器网络，性能有一定损耗
	容器网络隔离	Pod可直接关联安全组，基于安全组的隔离策略，支持集群内外部统一的安全隔离。	● 隧道网络模式：集群内部网络隔离策略，支持Networkpolicy。 ● VPC网络模式：不支持
安全	隔离性	● 虚拟机：普通容器，Cgroups隔离	普通容器，Cgroups隔离

1.7.4 区域与可用区

什么是区域、可用区?

用区域和可用区来描述数据中心的位置，您可以在特定的区域、可用区创建资源。

- 区域（Region）：从地理位置和网络时延维度划分，同一个Region内共享弹性计算、块存储、对象存储、VPC网络、弹性公网IP、镜像等公共服务。Region分为通用Region和专属Region，通用Region指面向公共租户提供通用云服务的Region；专属Region指只承载同一类业务或只面向特定租户提供业务服务的专用Region。
- 可用区（AZ，Availability Zone）：一个AZ是一个或多个物理数据中心的集合，有独立的风火水电，AZ内逻辑上再将计算、网络、存储等资源划分成多个集群。一个Region中的多个AZ间通过高速光纤相连，以满足用户跨AZ构建高可用性系统的需求。

目前，全球多个地域均已开放云服务，您可以根据需求选择适合自己的区域和可用区。

如何选择区域?

选择区域时，您需要考虑以下因素：

- 地理位置

  一般情况下，建议就近选择靠近您或者您的目标用户的区域，这样可以减少网络时延，提高访问速度。

如何选择可用区?

是否将资源放在同一可用区内，主要取决于您对容灾能力和网络时延的要求。

- 如果您的应用需要较高的容灾能力，建议您将资源部署在同一区域的不同可用区内。
- 如果您的应用要求实例之间的网络延时较低，则建议您将资源创建在同一可用区内。

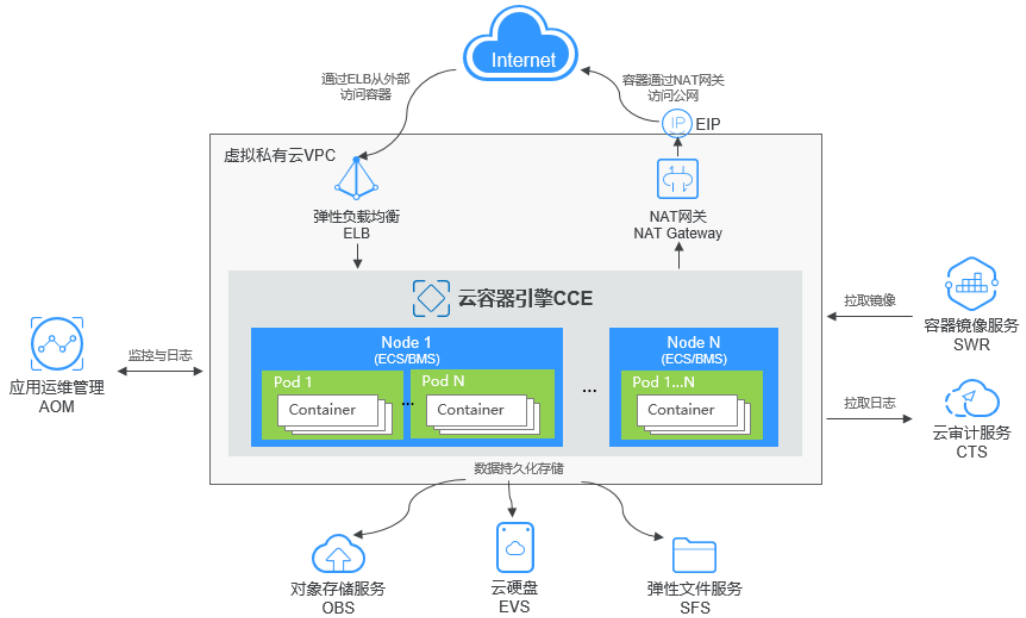
区域和终端节点

当您通过API使用资源时，您必须指定其区域终端节点。有关区域和终端节点的更多信息，请参阅**地区和终端节点**。

1.8 与其它云服务的关系

云容器引擎需要与其他云服务协同工作，云容器引擎需要获取如下云服务资源的权限。

图 1-16 云容器引擎与其他服务的关系示意图



云容器引擎与其他服务的关系

表 1-7 云容器引擎与其他服务的关系

服务名称	云容器引擎与其他服务的关系
弹性云服务器 ECS	在云容器引擎中具有多个云硬盘的一台弹性云服务器就是一个节点，您可以在创建节点时指定弹性云服务器的规格。
虚拟私有云 VPC	在云容器引擎中创建的集群需要运行在虚拟私有云中，您创建命名空间时，需要创建或关联VPC，创建在命名空间的容器都运行在VPC之内，从而保障网络安全。
弹性负载均衡 ELB	云容器引擎支持将创建的应用对接到弹性负载均衡，从而提高应用系统对外的服务能力，提高应用程序容错能力。
NAT网关	NAT网关能够为VPC内的容器实例提供网络地址转换（Network Address Translation）服务，SNAT功能通过绑定弹性公网IP，实现私有IP向公有IP的转换，可实现VPC内的容器实例共享弹性公网IP访问Internet。
容器镜像服务 SWR	容器镜像服务提供的镜像仓库是用于存储、管理docker容器镜像的场所，可以让使用人员轻松存储、管理、部署docker容器镜像。
云硬盘 EVS	可以将云硬盘挂载到云服务器，并可以随时扩容云硬盘容量。 在云容器引擎中一个节点就是具有多个云硬盘的一台弹性云服务器，您可以在创建节点时指定云硬盘的大小。
对象存储服务 OBS	对象存储服务是一个基于对象的海量存储服务，为客户提供海量、安全、高可靠、低成本的数据存储能力，包括：创建、修改、删除桶，上传、下载、删除对象等。 云容器引擎支持创建OBS对象存储卷并挂载到容器的某一路径下。
弹性文件服务 SFS	弹性文件服务提供托管的共享文件存储，符合标准文件协议（NFS），能够弹性伸缩至PB规模，具备可扩展的性能，为海量数据、高带宽型应用提供有力支持。 您可以使用弹性文件服务作为容器的持久化存储，在创建任务负载的时候挂载到容器上。
应用运维管理 AOM	云容器引擎对接了AOM，AOM会采集容器日志存储中的“.log”等格式日志文件，转储到AOM中，方便您查看和检索；并且云容器引擎基于AOM进行资源监控，为您提供弹性伸缩能力。
云审计服务 CTS	云审计服务提供云服务资源的操作记录，记录内容包括您从云管理控制台或者开放API发起的云服务资源操作请求以及每次请求的结果，供您查询、审计和回溯使用。

2 产品公告

2.1 集群节点高危操作

集群使用须知

- 在创建、删除、扩容和缩容集群的操作中，请不要在统一身份认证服务（IAM）中执行权限变更或修改的操作，可能会导致创建、删除、扩容和缩容集群执行失败。
- CCE节点的容器网络canal会使用一个网段作为容器网络的网段，该网段在创建集群时可配置，默认为172.16.0.0/16；docker服务也会默认创建一个docker0的网桥，docker0的默认地址为172.17.0.1。创建集群时，请确保集群中VPC（虚拟私有云）所在的网段不能和上述两个网段重复。使用VPC的对等连接功能时，对端的VPC也不能和上述两个网段重复。
- Kubernetes 1.15版本的集群，集群中节点的DNS服务器设置会使用虚拟私有云（VPC）子网下的DNS，不会添加K8S的CoreDNS地址，请务必确保子网下中的DNS address是存在的，且是可配置的。
- Kubernetes 1.17版本的集群，因为节点网络为单网络平面，对于多网络平面场景若您在ECS上新绑定了网卡，网卡绑定后，需要到节点上配置网卡信息并重启该网卡。
- 建议不要修改CCE创建的安全组、云硬盘等信息，会导致CCE集群功能异常（CCE创建的资源标记有“cce-”，如：cce-evs-jwh9pcl7-****）。
- 添加节点时，子网中的DNS服务器需要能解析出对应服务的域名，否则节点无法正常安装。

节点使用须知

节点的部分资源需要运行一些必要的Kubernetes系统组件和Kubernetes系统资源，使该节点可作为您的集群的一部分。 因此，您的节点资源总量与节点在Kubernetes中的可分配资源之间会存在差异。节点的规格越大，在节点上部署的容器可能会越多，所以Kubernetes自身需预留更多的资源。

为了保证节点的稳定性，CCE集群节点上会根据节点的规格预留一部分资源给Kubernetes的相关组件（kubelet、kube-proxy以及docker等）。

说明

建议不要在集群的节点上安装自己的软件，也不建议用户修改操作系统配置。可能会导致安装在节点上的Kubernetes组件异常，节点状态变成不可用，导致无法部署工作负载到此节点。

节点高危操作

通过CCE所创建的节点，在您登录到该节点后，执行以下操作时请务必谨慎，否则会导致该节点不可用。

表 2-1 导致节点不可用的高危操作

序号	高危操作	导致后果	误操作后解决方案
1	重装操作系统（使用原镜像或其它镜像）	节点不可用	删除节点，重新创建。
2	修改操作系统配置	节点不可用	尝试还原配置项或重新创建节点。
3	删除opt、/var/paas目录，删除数据盘	节点不可用	删除节点，重新创建。
4	节点磁盘格式化、分区	节点不可用	删除节点，重新创建。
5	修改安全组	集群功能异常/节点不可用	参照新建集群的安全组进行修复。

2.2 CCE 安全使用指引

节点安全加固说明

CCE服务的集群节点基于公有云IMS服务发布的公共镜像进行制作。CCE会对节点上安装的CCE软件进行安全加固，而OS相关的配置默认与公共镜像保持一致，用户应根据自身的安全诉求进行相应的安全加固。

容器安全使用建议

CCE给用户提供的是一个独享集群，不建议以共享集群方式为多租户使用。当用户以共享集群的方式提供给多租户使用时，请参考如下容器安全使用建议进行安全加固。

表 2-2 容器安全使用建议

分类	安全使用建议

权限最小化	1. 容器中通过sudo白名单的方式限制普通用户可以执行的docker命令（白名单）。
	2. 建议Kubernetes pod限制使用特权容器，即不要使用如下方式：
	`apiVersion: v1`
	`kind: Pod`
	`metadata:`
	`  name: hello-world`
	`spec:`
	`  containers:`
	`  - name: hello-world-container`
	`  # The container definition`
	`  # ...`
	`  securityContext:`
	`  privileged: true`

	默认不使用default serviceaccount下的Token。
	● 默认关闭default serviceaccount下的自动挂载开关。完成automountServiceAccountToken: false设置后，创建的工作负载将不会默认挂载token。
	注意
	每个命名空间都要按需设置；设置完毕后滚动重启相应命名空间下的pod。
	`apiVersion: v1`
	`kind: ServiceAccount`
	`metadata:`
	`  name: default`
	`automountServiceAccountToken: false`
	● 有选择性的在工作负载中挂载default serviceaccount下的Token：
	`...`
	`spec:`
	`  template:`
	`  spec:`
	`  serviceAccountName: default`
	`  automountServiceAccountToken: true`
	`  ...`

网络隔离	**集群内容器之间的网络隔离安全加固：** 通过networkpolicy策略实现访问控制。 **容器和集群外主机的网络隔离安全加固：** 通过VPC内的安全组实现访问控制。 **容器和集群管理面的网络隔离安全加固：** 由于CCE集群是私有集群，且工作负载有访问集群apiserver的场景，因此CCE未限制容器和Kubernetes管理面的网络通信，用户如通过限制Pod的QoS保证加固网络安全。
禁止挂载敏感主机目录	● 如非必须，在启动容器时请不要挂载主机上的/var/run/docker.sock文件到容器内。 ● 不要将主机上的hostPath目录挂载到容器中，如确有必要，则以只读的方式挂载，例如：/、/boot、/dev、/etc、/lib、/proc、/sys、/usr等。

2.3 集群节点操作系统补丁说明

CCE 集群

CCE集群中节点支持的操作系统及对应的内核版本如下：

表 2-3 集群节点操作系统内核说明

操作系统	集群版本	内核信息
EulerOS release 2.9	v1.27	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64
	v1.25	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64
	v1.23	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64
	v1.21	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64
	v1.19.10	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64
EulerOS release 2.5	v1.25	3.10.0-862.14.1.0.h197.eulerosv2r7.x86_64
	v1.23	3.10.0-862.14.1.0.h197.eulerosv2r7.x86_64
	v1.21	3.10.0-862.14.1.0.h197.eulerosv2r7.x86_64
	v1.19.10	3.10.0-862.14.1.0.h197.eulerosv2r7.x86_64
	v1.17.9	3.10.0-862.14.1.5.h470.eulerosv2r7.x86_64
	v1.15.11	3.10.0-862.14.1.5.h470.eulerosv2r7.x86_64
CentOS Linux release 7.7	v1.23	3.10.0-1062.12.1.el7.x86_64
	v1.21	3.10.0-1062.12.1.el7.x86_64
	v1.19.10	3.10.0-1062.12.1.el7.x86_64
	v1.17.9	3.10.0-1062.12.1.el7.x86_64
Ubuntu 22.04（仅支持VPC网络模型）	v1.25	5.15.0-46-generic
	v1.23	5.15.0-46-generic

后续将定期更新操作系统补丁发布以及验证结果，您可以根据需求进行操作系统更新。

2.4 漏洞公告

2.4.1 Kubernetes 安全漏洞公告（CVE-2022-3172）

漏洞详情

Kubernetes社区在 kube-apiserver 中发现了一个安全问题，该问题允许聚合 API Server将客户端流量重定向到任意 URL，这可能导致客户端执行意外操作以及将客户端的 API 服务器凭据转发给第三方。

表 2-4 漏洞信息

漏洞类型	CVE-ID	漏洞级别	披露/发现时间
SSRF	CVE-2022-3172	中	2022-09-09

漏洞影响

CCE受影响的版本：

- kube-apiserver <= v1.23.10

符合上述范围的CCE集群，且配置了聚合API Server的均受影响，尤其是将CCE集群在逻辑多租场景下使用风险较高。

判断方法

对于1.23及以下版本的CCE集群、CCE Turbo集群，配置kubectl连接集群，运行以下命令，确认是否运行聚合API Server：

```
kubectl get apiservices.apiregistration.k8s.io -o=jsonpath='{range .items[?(@.spec.service)]}{.metadata.name}{"\n"}{end}'
```

若返回值非空，说明存在聚合API Server。

漏洞修复方案

除了升级之外，当前没有直接可用的缓解措施。集群管理员应注意控制权限，防止非受信人员通过APIService接口部署和控制聚合API Server。

该漏洞已在v1.23.5-r0、v1.21.7-r0、v1.19.16-r4版本的CCE集群中修复。

相关链接

https://github.com/kubernetes/kubernetes/issues/112513

2.4.2 Linux Kernel openvswitch 模块权限提升漏洞预警（CVE-2022-2639）

漏洞详情

业界披露了Linux Kernel openvswitch模块权限提升漏洞（CVE-2022-2639）的漏洞细节。由于 openvswitch模块中reserve_sfa_size()函数在使用过程中存在缺陷，导致本地经过身份认证的攻击者可以利用漏洞提升至root权限。目前漏洞poc已公开，风险较高。

表 2-5 漏洞信息

漏洞类型	CVE-ID	漏洞级别	披露/发现时间
权限提升	CVE-2022-2639	高	2022-09-01

漏洞影响

1. 采用容器隧道网络的CCE集群，节点OS镜像使用了EulerOS 2.9。

2. 节点OS镜像使用了Ubuntu。

EulerOS 2.5 和CentOS 7.6的集群节点**不受该漏洞影响**。

漏洞修复方案

1. 容器内进程使用非root用户启动的进程可以通过为工作负载配置安全计算模式seccomp，建议配置RuntimeDefault模式或者禁用unshare等系统调用。具体配置方法可参考社区官方资料**使用 Seccomp 限制容器的系统调用**。
2. Ubuntu镜像自带openvswitch内核模块，可以通过将禁止加载openvswitch 内核模块来规避。操作如下：
   ```
   echo "blacklist openvswitch" >>/etc/modprobe.d/blacklist.conf
   ```
   然后重启节点，使上述设置生效。

相关链接

https://github.com/torvalds/linux/commit/cefa91b2332d7009bc0be5d951d6cbbf349f90f8

2.4.3 CRI-O 容器运行时引擎任意代码执行漏洞（CVE-2022-0811）

漏洞详情

crowdstrike安全团队披露CRI-O 1.19版本中存在一个安全漏洞，攻击者可以利用该漏洞绕过保护措施并在主机上设置任意内核参数。这将导致任何有权在使用CRI-O的Kubernetes集群上部署Pod的用户都可以滥用kernel.core_pattern内核参数，在集群中的任何节点上以root身份实现容器逃逸和执行任意代码。

该问题已被收录为CVE-2022-0811。

表 2-6 漏洞信息

漏洞类型	CVE-ID	漏洞级别	披露/发现时间
容器逃逸	CVE-2022-0811	高	2021-03-16

漏洞影响

该漏洞影响范围为使用了CRI-O的Kubernetes集群，CRI-O的版本大于1.19，涉及的补丁版本包括1.19.6、1.20.7、1.21.6、1.22.3、1.23.2、1.24.0。

CCE集群未使用CRI-O，因此不受此漏洞影响。

漏洞修复方案

1. 1.19、1.20版本CRI-O，将manage_ns_lifecycle设置为false, 由OCI运行时配置sysctl。
2. 创建PodSecurityPolicy，将所有sysctl指定为false。
3. 及时升级CRI-O版本。

相关链接

1. Red Hat社区漏洞公告：**https://access.redhat.com/security/cve/cve-2022-0811**
2. cr8escape: New Vulnerability in CRI-O Container Engine Discovered by CrowdStrike：**https://www.crowdstrike.com/blog/cr8escape-new-vulnerability-discovered-in-cri-o-container-engine-cve-2022-0811/**

2.4.4 linux 内核导致的容器逃逸漏洞公告（CVE-2022-0492）

漏洞详情

在某些场景下linux内核cgroup v1的release_agent特性存在可以被利用在容器内逃逸到OS上的安全问题，该问题已被收录为CVE-2022-0492。

表 2-7 漏洞信息

漏洞类型	CVE-ID	漏洞级别	披露/发现时间
容器逃逸	CVE-2022-0492	高	2021-02-07

漏洞影响

该漏洞为Linux内核权限校验漏洞，根因为没有针对性的检查设置release_agent文件的进程是否具有正确的权限。在受影响的OS节点上，工作负载使用了root用户运行进程（或者具有CAP_SYS_ADMIN权限），并且未配置seccomp时将受到漏洞影响。

CCE集群受该漏洞影响的范围如下：

1. x86场景EulerOS 2.5和CentOS镜像不受该漏洞影响。

2. 内核版本小于4.19.36-vhulk1907.1.0.h962.eulerosv2r8.aarch64的EulerOS arm版本。
3. 内核版本小于4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64的EulerOS x86版本。
4. 内核版本为4.15.0-136-generic以及以下内核版本的Ubuntu节点。

漏洞修复方案

1. EulerOS 2.9 版本镜像已提供修复版本，请尽快迁移到4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64版本节点。
2. 为工作负载配置seccomp，限制unshare系统调用，详情请参考Kuberenetes**社区文档**。
3. 限制容器内进程权限，最小化容器内的进程权限，如使用非root启动进程、通过capability机制细化进程权限等。

相关链接

1. 内核修复commit：**https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=24f6008564183aa120d07c03d9289519c2fe02af**
2. Red Hat社区漏洞公告：**https://access.redhat.com/security/cve/cve-2022-0492**

2.4.5 Linux 内核整数溢出漏洞（CVE-2022-0185）

漏洞详情

国外安全研究人员William Liu和Jamie Hill-Daniel发现Linux内核中包含一个整数溢出漏洞，可导致写操作越界。本地攻击者可以使用这一点导致拒绝服务(系统崩溃)或执行任意代码，在容器场景下拥有CAP_SYS_ADMIN权限的用户可导致容器逃逸到宿主机。目前已存在poc，但尚未发现已公开的利用代码。

表 2-8 漏洞信息

漏洞类型	CVE-ID	漏洞级别	披露/发现时间
资源管理错误	CVE-2022-0185	高	2022-01-27

漏洞影响

容器内用户拥有CAP_SYS_ADMIN权限，并且内核版本在5.1以及以上。在标准的docker环境下，由于使用了Docker seccomp filter，默认情况下不受该漏洞影响。在Kubernetes场景下，默认禁用了seccomp filter，在内核以及权限满足时受该漏洞影响。

CCE当前不受影响

判断方法

uname -a查看内核版本号

规避和消减措施

CCE集群节点不受该漏洞影响。对于自建的K8s集群，建议用户对工作负载：

1. 最小权限运行容器
2. 根据社区提供的配置方法配置**seccomp**

相关链接

https://blog.aquasec.com/cve-2022-0185-linux-kernel-container-escape-in-kubernetes

https://ubuntu.com/security/CVE-2022-0185

https://access.redhat.com/security/cve/CVE-2022-0185

https://www.openwall.com/lists/oss-security/2022/01/18/7

3 Kubernetes 基础知识

3.1 概述

Kubernetes是一个开源的容器编排部署管理平台，用于管理云平台上多个主机上的容器化应用。Kubernetes的目标是让部署容器化的应用简单并且高效（powerful），Kubernetes提供了应用部署、规划、更新、维护的一种机制。

Kubernetes一个核心的特点就是能够自主地管理容器来保证云平台上的容器按照用户的期望状态运行着（比如用户想让apache一直运行，用户不需要关心怎么去做，Kubernetes会自动去监控，然后去重启、新建，总之，让apache一直提供服务。），管理员可以加载一个微型服务，让规划器来找到合适的位置，同时，Kubernetes也系统提升工具以及人性化方面，让用户能够方便的部署自己的应用。

您可以通过CCE控制台、Kubectl命令行、Kubernetes API使用云容器引擎所提供的Kubernetes托管服务。在使用云容器引擎之前，你可以先行了解如下Kubernetes的相关概念，以便您更完整的使用云容器引擎的所有功能。

容器与 Kubernetes

- 容器
- Kubernetes

Pod、Label 和 Namespace

- Pod：Kubernetes中的最小调度对象
- 存活探针（Liveness Probe）
- Label：组织Pod的利器
- Namespace：资源分组

Pod 的编排与调度

- Deployment
- StatefulSet
- Job和CronJob
- DaemonSet

- 亲和与反亲和调度

配置管理

- ConfigMap
- Secret

Kubernetes 网络

- 容器网络
- Service
- Ingress
- 就绪探针（Readiness Probe）
- NetworkPolicy

持久化存储

- Volume
- PV、PVC和StorageClass

认证与授权

- ServiceAccount
- RBAC

弹性伸缩

- 弹性伸缩

3.2 容器与 Kubernetes

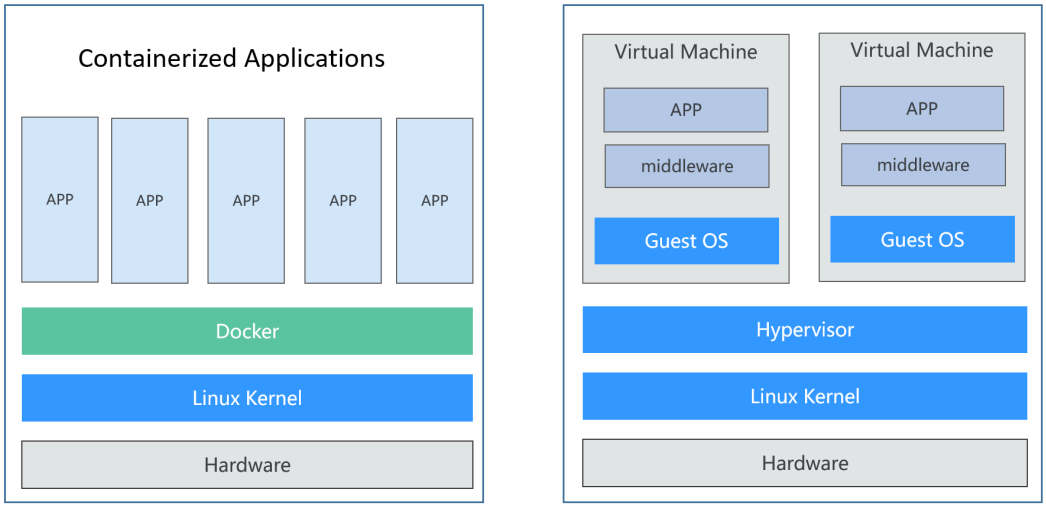
3.2.1 容器

容器与 Docker

容器技术起源于Linux，是一种内核虚拟化技术，提供轻量级的虚拟化，以便隔离进程和资源。尽管容器技术已经出现很久，却是随着Docker的出现而变得广为人知。Docker是第一个使容器能在不同机器之间移植的系统。它不仅简化了打包应用的流程，也简化了打包应用的库和依赖，甚至整个操作系统的文件系统能被打包成一个简单的可移植的包，这个包可以被用来在任何其他运行Docker的机器上使用。

容器和虚拟机具有相似的资源隔离和分配方式，容器虚拟化了操作系统而不是硬件，更加便携和高效。

图 3-1 容器 vs 虚拟机



相比于使用虚拟机，容器有如下优点：

- 更高效的利用系统资源

  由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，容器对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。
- 更快速的启动时间

  传统的虚拟机技术启动应用服务往往需要数分钟，而Docker容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间，大大节约了开发、测试、部署的时间。
- 一致的运行环境

  开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些问题并未在开发过程中被发现。而Docker的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性。
- 更轻松的迁移

  由于Docker确保了执行环境的一致性，使得应用的迁移更加容易。Docker可以在很多平台上运行，无论是物理机、虚拟机，其运行结果是一致的。因此可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。
- 更轻松的维护和扩展

  Docker使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker团队同各个开源项目团队一起维护了大批高质量的官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

Docker 容器典型使用流程

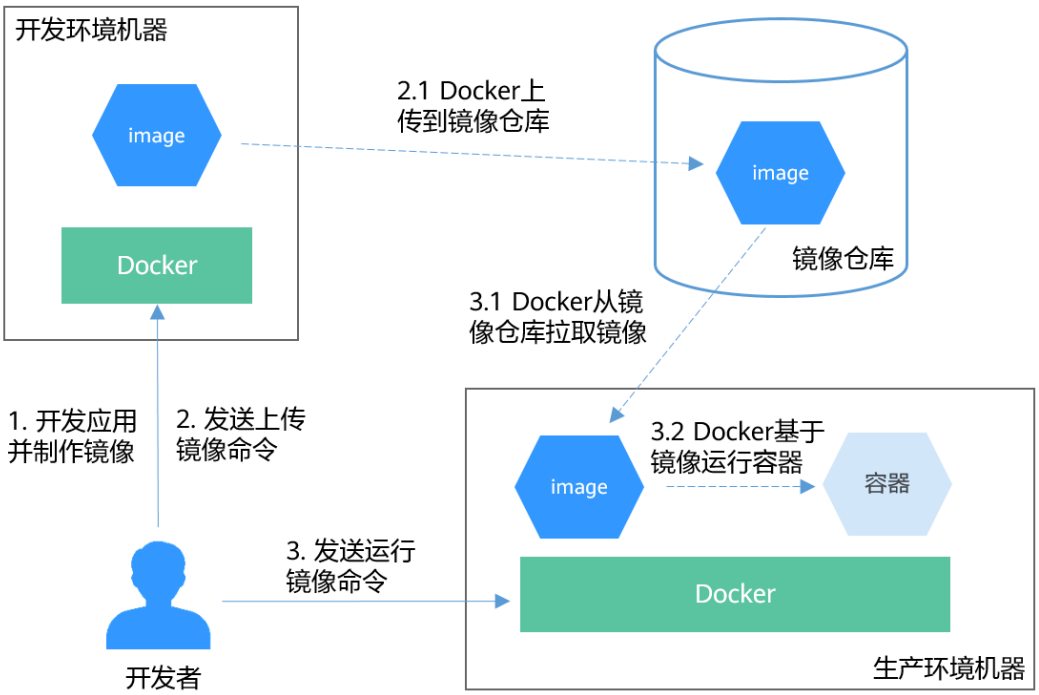
Docker容器有如下三个主要概念：

- **镜像**：Docker镜像里包含了已打包的应用程序及其所依赖的环境。它包含应用程序可用的文件系统和其他元数据，如镜像运行时的可执行文件路径。

- **镜像仓库**：Docker镜像仓库用于存放Docker镜像，以及促进不同人和不同电脑之间共享这些镜像。当编译镜像时，要么可以在编译它的电脑上运行，要么可以先上传镜像到一个镜像仓库，然后下载到另外一台电脑上并运行它。某些仓库是公开的，允许所有人从中拉取镜像，同时也有一些是私有的，仅部分人和机器可接入。
- **容器**：Docker容器通常是一个Linux容器，它基于Docker镜像被创建。一个运行中的容器是一个运行在Docker主机上的进程，但它和主机，以及所有运行在主机上的其他进程都是隔离的。这个进程也是资源受限的，意味着它只能访问和使用分配给它的资源（CPU、内存等）。

典型的使用流程如图3-2所示：

图 3-2 Docker 容器典型使用流程



1. 首先开发者在开发环境机器上开发应用并制作镜像。

   Docker执行命令，构建镜像并存储在机器上。
2. 开发者发送上传镜像命令。

   Docker收到命令后，将本地镜像上传到镜像仓库。
3. 开发者向生产环境机器发送运行镜像命令。

   生产环境机器收到命令后，Docker会从镜像仓库拉取镜像到机器上，然后基于镜像运行容器。

使用示例

下面使用Docker将基于Nginx镜像打包一个容器镜像，并基于容器镜像运行应用，然后推送到容器镜像仓库。

安装Docker

Docker几乎支持在所有操作系统上安装，用户可以根据需要选择要安装的Docker版本。

在Linux操作系统下，可以使用如下命令快速安装Docker。

```
curl -fsSL get.docker.com -o get-docker.sh
sh get-docker.sh
systemctl restart docker
```

Docker打包镜像

Docker提供了一种便捷的描述应用打包的方式，叫做Dockerfile，如下所示：

```
# 使用官方提供的Nginx镜像作为基础镜像
FROM nginx:alpine

# 执行一条命令修改Nginx镜像index.html的内容
RUN echo "hello world" > /usr/share/nginx/html/index.html

# 允许外界访问容器的80端口
EXPOSE 80
```

执行docker build命令打包镜像。

docker build -t *hello* **.**

其中-t表示给镜像加一个标签，也就是给镜像取名，这里镜像名为hello。**.** 表示在当前目录下执行该打包命令。

执行docker images命令查看镜像，可以看到hello镜像已经创建成功。您还可以看到一个Nginx镜像，这个镜像是从镜像仓库下载下来的，作为hello镜像的基础镜像使用。

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello               latest              d120ec16dcea        17 minutes ago      158MB
nginx               alpine              eeb27ee6b893        2 months ago        148MB
```

本地运行容器镜像

有了镜像后，您可以在本地执行docker run命令运行容器镜像。

```
# docker run -p 8080:80 hello
```

docker run命令会启动一个容器，命令中-p是将本地机器的8080端口映射到容器的80端口，即本地机器的8080端口的流量会映射到容器的80端口，当您在本地机器访问http://127.0.0.1:8080时，就会访问到容器中，此时浏览器中返回的内容应该就是“hello world”。

把镜像推送到镜像仓库

提供了容器镜像服务SWR，您也可以将镜像上传到SWR，下面演示如何将镜像推送到SWR。本文档后续的示例中将主要使用SWR作为示例。

首先登录SWR控制台，在左侧选择“我的镜像”，然后单击右侧“客户端上传镜像”，在弹出的窗口中单击“生成临时登录指令”，然后复制该指令在本地机器上执行，登录到SWR镜像仓库。

上传镜像前需要给镜像取一个完整的名称，如下所示：

```
# docker tag hello registry.eu-west-0.prod-cloud-ocb.orange-business.com/container/hello:v1
```

这里registry.eu-west-0.prod-cloud-ocb.orange-business.com是仓库地址，每个区域的地址不同，v1则是hello镜像分配的版本号。

- registry.eu-west-0.prod-cloud-ocb.orange-business.com是仓库地址，每个区域的地址不同。
- container是组织名，组织一般在SWR中创建，如果没有创建则首次上传的时候会自动创建，组织名在单个区域内全局唯一，需要选择合适的组织名称。
- v1则是hello镜像分配的版本号。

然后执行docker push命令就可以将镜像上传到SWR。

```
# docker push registry.eu-west-0.prod-cloud-ocb.orange-business.com/container/hello:v1
```

当需要使用该镜像时，使用docker pull命令拉取（下载）该命令即可。

```
# docker pull registry.eu-west-0.prod-cloud-ocb.orange-business.com/container/hello:v1
```

3.2.2 Kubernetes

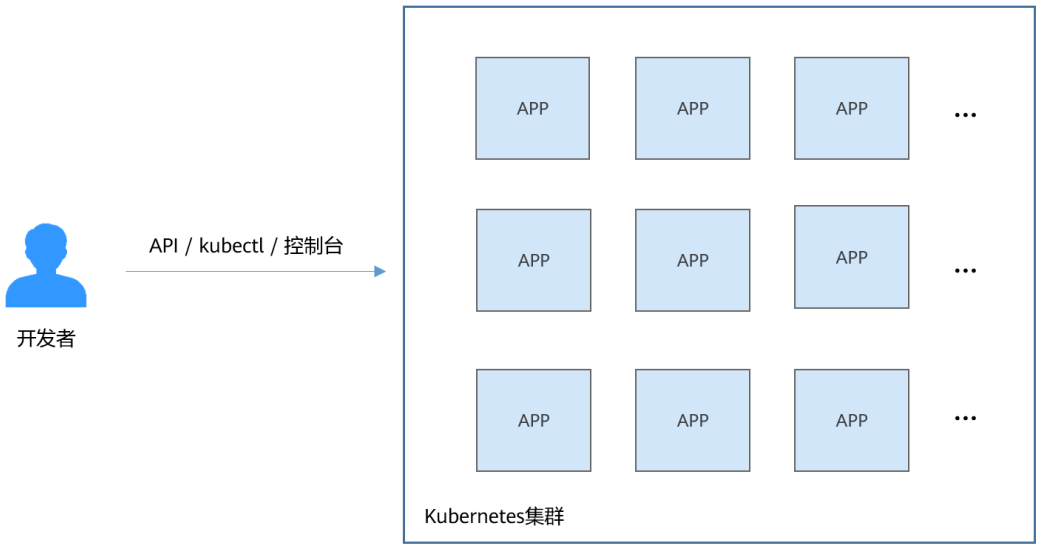
Kubernetes 是什么

Kubernetes是一个很容易地部署和管理容器化的应用软件系统，使用Kubernetes能够方便对容器进行调度和编排。

对应用开发者而言，可以把Kubernetes看成一个集群操作系统。Kubernetes提供服务发现、伸缩、负载均衡、自愈甚至选举等功能，让开发者从基础设施相关配置等解脱出来。

Kubernetes可以把大量的服务器看做一台巨大的服务器，在一台大服务器上面运行应用程序。无论Kubernetes的集群有多少台服务器，在Kubernetes上部署应用程序的方法永远一样。

图 3-3 在 Kubernetes 集群上运行应用程序

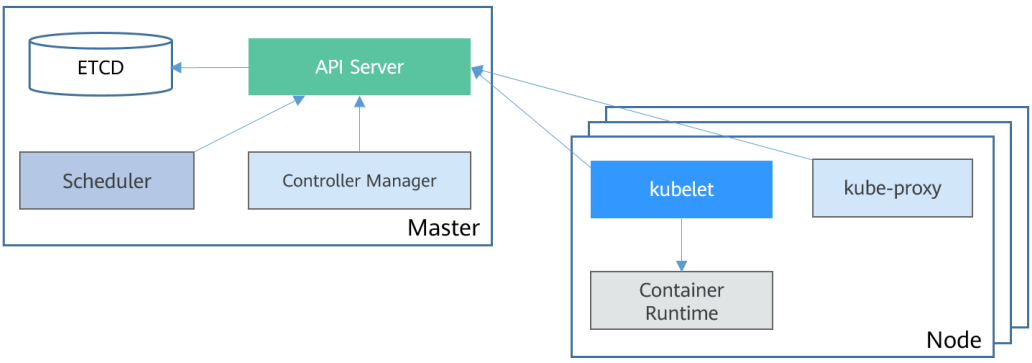


Kubernetes 集群架构

Kubernetes集群包含master节点和node节点，应用部署在node节点上，且可以通过配置选择应用部署在某些特定的节点上。

Kubernetes集群的架构如下所示：

图 3-4 Kubernetes 集群架构



Master节点

Master节点是集群的控制节点，由API Server、Scheduler、Controller Manager和ETCD四个组件构成。

- API Server：各组件互相通讯的中转站，接受外部请求，并将信息写到ETCD中。
- Controller Manager：执行集群级功能，例如复制组件，跟踪Node节点，处理节点故障等等。
- Scheduler：负责应用调度的组件，根据各种条件（如可用的资源、节点的亲和性等）将容器调度到Node上运行。
- ETCD：一个分布式数据存储组件，负责存储集群的配置信息。

在生产环境中，为了保障集群的高可用，通常会部署多个master，如CCE集群的高可用模式就是3个master节点。

Node节点

Node节点是集群的计算节点，即运行容器化应用的节点。

- kubelet：kubelet主要负责同Container Runtime打交道，并与API Server交互，管理节点上的容器。
- kube-proxy：应用组件间的访问代理，解决节点上应用的访问问题。
- Container Runtime：容器运行时，如Docker，最主要的功能是下载镜像和运行容器。

Kubernetes 的扩展性

Kubernetes开放了容器运行时接口（CRI）、容器网络接口（CNI）和容器存储接口（CSI），这些接口让Kubernetes的扩展性变得最大化，而Kubernetes本身则专注于容器调度。

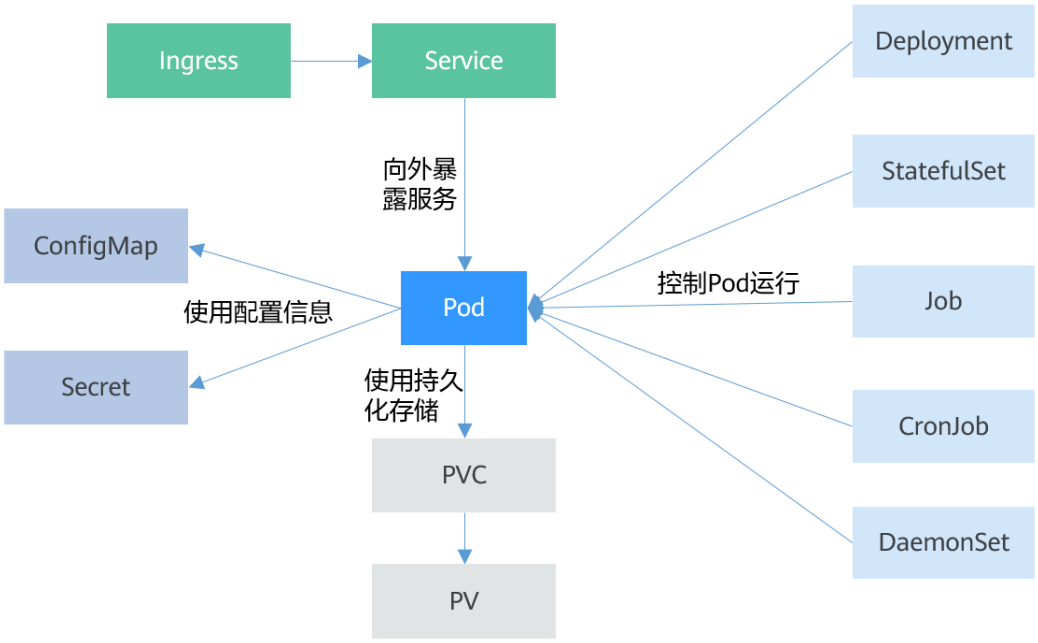
- CRI（Container Runtime Interface）：容器运行时接口，提供计算资源，CRI隔离了各个容器引擎之间的差异，而通过统一的接口与各个容器引擎之间进行互动。
- CNI（Container Network Interface）：容器网络接口，提供网络资源，通过CNI接口，Kubernetes可以支持不同网络环境。例如CCE就是开发的CNI插件支持Kubernetes集群运行在VPC网络中。

- CSI（Container Storage Interface）：容器存储接口，提供存储资源，通过CSI接口，Kubernetes可以支持各种类型的存储。例如CCE就可以方便的对接块存储（EVS）、文件存储（SFS）和对象存储（OBS）。

Kubernetes 中的基本对象

上面介绍Kubernetes集群的构成，下面将介绍Kubernetes中基本对象及它们之间的一些关系。

图 3-5 Kubernetes 基本对象



- Pod

  Pod是Kubernetes创建或部署的最小单位。一个Pod封装一个或多个容器（container）、存储资源（volume）、一个独立的网络IP以及管理控制容器运行方式的策略选项。

- Deployment

  Deployment是对Pod的服务化封装。一个Deployment可以包含一个或多个Pod，每个Pod的角色相同，所以系统会自动为Deployment的多个Pod分发请求。

- StatefulSet

  StatefulSet是用来管理有状态应用的对象。和Deployment相同的是，StatefulSet管理了基于相同容器定义的一组Pod。但和Deployment不同的是，StatefulSet为它们的每个Pod维护了一个固定的ID。这些Pod是基于相同的声明来创建的，但是不能相互替换，无论怎么调度，每个Pod都有一个永久不变的ID。

- Job

  Job是用来控制批处理型任务的对象。批处理业务与长期伺服业务（Deployment）的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出（Pod自动删除）。

- CronJob

CronJob是基于时间控制的Job，类似于Linux系统的crontab，在指定的时间周期运行指定的任务。

- DaemonSet

  DaemonSet是这样一种对象（守护进程），它在集群的每个节点上运行一个Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资源监控等，这类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes的kube-proxy。

- Service

  Service是用来解决Pod访问问题的。Service有一个固定IP地址，Service将访问流量转发给Pod，而且Service可以给这些Pod做负载均衡。

- Ingress

  Service是基于四层TCP和UDP协议转发的，Ingress可以基于七层的HTTP和HTTPS协议转发，可以通过域名和路径做到更细粒度的划分。

- ConfigMap

  ConfigMap是一种用于存储应用所需配置信息的资源类型，用于保存配置数据的键值对。通过ConfigMap可以方便的做到配置解耦，使得不同环境有不同的配置。

- Secret

  Secret是一种加密存储的资源对象，您可以将认证信息、证书、私钥等保存在Secret中，而不需要把这些敏感数据暴露到镜像或者Pod定义中，从而更加安全和灵活。

- PersistentVolume（PV）

  PV指持久化数据存储卷，主要定义的是一个持久化存储在宿主机上的目录，比如一个NFS的挂载目录。

- PersistentVolumeClaim（PVC）

  Kubernetes提供PVC专门用于持久化存储的申请，PVC可以让您无需关心底层存储资源如何创建、释放等动作，而只需要申明您需要何种类型的存储资源、多大的存储空间。

搭建 Kubernetes 集群

Kubernetes网站上有多种搭建Kubernetes集群的方法，例如minikube、kubeadm等。

如果不想自行搭建Kubernetes集群，可以在CCE服务中创建，本文后续内容都将在CCE中创建的集群上操作演示。

kubectl

kubectl是Kubernetes集群的命令行工具，您可以将kubectl安装在任意一台机器上，通过kubectl命令操作Kubernetes集群。

CCE集群的kubectl安装请参见**通过kubectl连接集群**。连接后您可以执行**kubectl cluster-info**查看集群的信息，如下所示。

```
# kubectl cluster-info
Kubernetes master is running at https://*.*.*.*:5443
CoreDNS is running at https://*.*.*.*:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

执行**kubectl get nodes**可以查看集群中的Node节点信息。

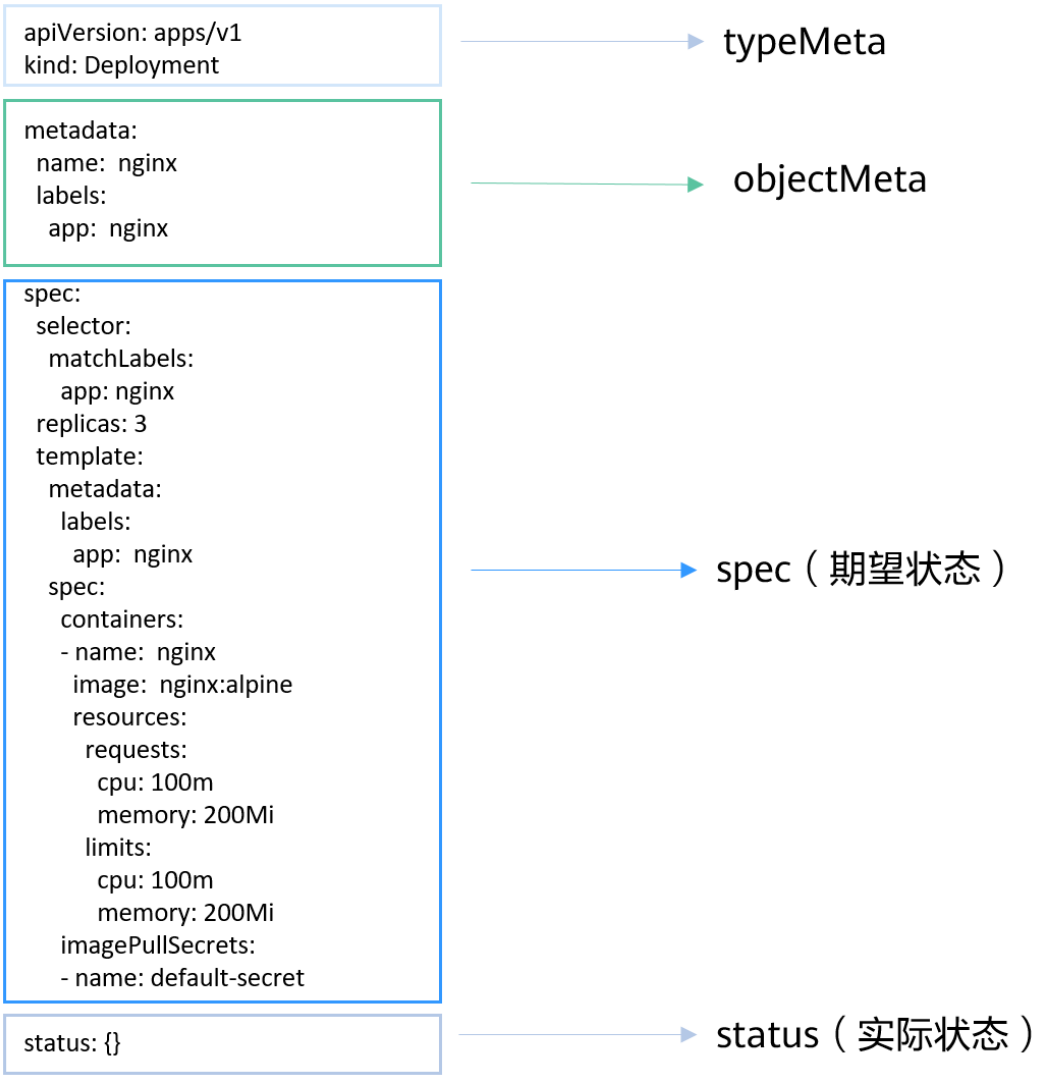
```
# kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
192.168.0.153   Ready    <none>   7m      v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.207   Ready    <none>   7m      v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.221   Ready    <none>   7m      v1.15.6-r1-20.3.0.2.B001-15.30.2
```

Kubernetes 对象的描述

kubernetes中资源可以使用YAML描述，也可以使用JSON。其内容可以分为如下四个部分：

- typeMeta：对象类型的元信息，声明对象使用哪个API版本，哪个类型的对象。
- objectMeta：对象的元信息，包括对象名称、使用的标签等。
- spec：对象的期望状态，例如对象使用什么镜像、有多少副本等。
- status：对象的实际状态，只能在对象创建后看到，创建对象时无需指定。

图 3-6 YAML 描述文件



在 Kubernetes 上运行应用

将图3-6中的内容去除status存为一个名为nginx-deployment.yaml的文件，如下所示：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  nginx
  labels:
    app:  nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app:  nginx
    spec:
      containers:
      - name:  nginx
        image:  nginx:alpine
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
          limits:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

使用kubectl连接集群后，执行如下命令：

```
# kubectl create -f nginx-deployment.yaml
deployment.apps/nginx created
```

命令执行后，Kubernetes集群中会创建3个Pod，使用如下命令可以查询到Deployment和Pod：

```
# kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   3/3     3            3           9s

# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-685898579b-qrt4d   1/1     Running   0          15s
nginx-685898579b-t9zd2   1/1     Running   0          15s
nginx-685898579b-w59jn   1/1     Running   0          15s
```

到此为止，您了解容器和Docker、Kubernetes集群、Kubernetes基本概念，并通过一个示例了解kubectl的最基本使用，本文后续将向您深入介绍Kubernetes对象的概念以及使用方法，并介绍对象之间的关系。

3.3 Pod、Label 和 Namespace

3.3.1 Pod：Kubernetes 中的最小调度对象

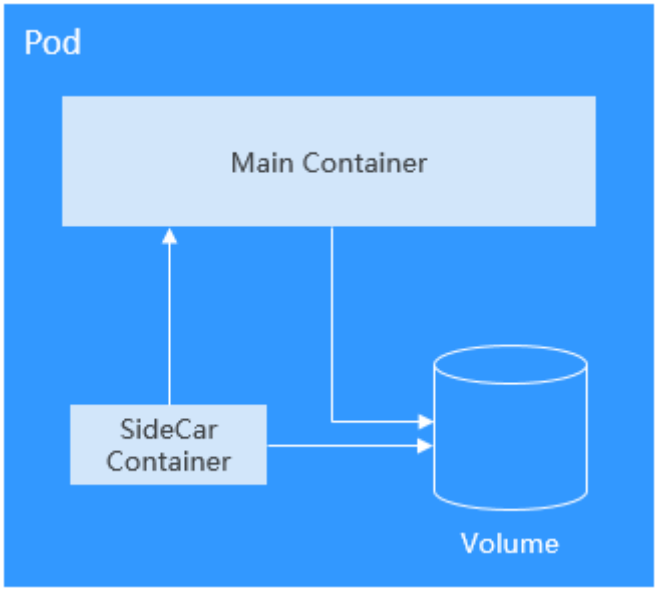
Pod

Pod是Kubernetes创建或部署的最小单位。一个Pod封装一个或多个容器（container）、存储资源（volume）、一个独立的网络IP以及管理控制容器运行方式的策略选项。

Pod使用主要分为两种方式：

- Pod中运行一个容器。这是Kubernetes最常见的用法，您可以将Pod视为单个封装的容器，但是Kubernetes是直接管理Pod而不是容器。
- Pod中运行多个需要耦合在一起工作、需要共享资源的容器。通常这种场景下应用包含一个主容器和几个辅助容器（SideCar Container），如图3-7所示，例如主容器为一个web服务器，从一个固定目录下对外提供文件服务，而辅助容器周期性的从外部下载文件存到这个固定目录下。

图 3-7 Pod



实际使用中很少直接创建Pod，而是使用Kubernetes中称为Controller的抽象层来管理Pod实例，例如Deployment和Job。Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力。通常，Controller会使用Pod Template来创建相应的Pod。

创建 Pod

kubernetes中资源可以使用YAML描述，也可以使用JSON，如下示例描述了一个名为nginx的Pod，这个Pod中包含一个名为container-0的容器，使用nginx:alpine镜像，使用的资源为100m core CPU、200Mi内存。

```
apiVersion: v1                  # Kubernetes的API Version
kind: Pod                       # Kubernetes的资源类型
metadata:
  name: nginx                   # Pod的名称
spec:                           # Pod的具体规格（specification）
  containers:
  - image: nginx:alpine         # 使用的镜像为 nginx:alpine
    name: container-0           # 容器的名称
```

```
  resources:                  # 申请容器所需的资源
    limits:
      cpu: 100m
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 200Mi
imagePullSecrets:             # 拉取镜像使用的证书，在CCE上必须为default-secret
- name: default-secret
```

如上面YAML的注释，YAML描述文件主要为如下部分：

- **metadata**：一些名称/标签/namespace等信息。
- **spec**：Pod实际的配置信息，包括使用什么镜像，volume等。

如果去查询Kubernetes的资源，您会看到还有一个**status**字段，status描述kubernetes资源的实际状态，创建时不需要配置。这个示例是一个最小集，其他参数定义后面会逐步介绍。

Pod定义好后就可以使用kubectl创建，如果上面YAML文件名称为nginx.yaml，则创建命令如下所示，-f表示使用文件方式创建。

```
$ kubectl create -f nginx.yaml
pod/nginx created
```

Pod创建完成后，可以使用kubectl get pods命令查询Pod的状态，如下所示。

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
nginx         1/1     Running   0          40s
```

可以看到此处nginx这个Pod的状态为Running，表示正在运行；READY为1/1，表示这个Pod中有1个容器，其中1个容器的状态为Ready。

可以使用kubectl get命令查询具体Pod的配置信息，如下所示，-o yaml表示以YAML格式返回，还可以使用-o json，以JSON格式返回。

```
$ kubectl get pod nginx -o yaml
```

您还可以使用kubectl describe命令查看Pod的详情。

```
$ kubectl describe pod nginx
```

删除pod时，Kubernetes终止Pod中所有容器。 Kubernetes向进程发送SIGTERM信号并等待一定的秒数（默认为30）让容器正常关闭。如果它没有在这个时间内关闭，Kubernetes会发送一个SIGKILL信号杀死该进程。

Pod的停止与删除有多种方法，比如按名称删除，如下所示。

```
$ kubectl delete po nginx
pod "nginx" deleted
```

同时删除多个Pod。

```
$ kubectl delete po pod1 pod2
```

删除所有Pod。

```
$ kubectl delete po --all
pod "nginx" deleted
```

根据Label删除Pod，**Label**详细内容将会在下一个章节介绍。

```
$ kubectl delete po -l app=nginx
pod "nginx" deleted
```

使用环境变量

环境变量是容器运行环境中设定的一个变量。

环境变量为应用提供极大的灵活性，您可以在应用程序中使用环境变量，在创建容器时为环境变量赋值，容器运行时读取环境变量的值，从而做到灵活的配置，而不是每次都重新编写应用程序制作镜像。

环境变量的使用方法如下所示，配置spec.containers.env字段即可。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
    containers:
    - image: nginx:alpine
      name: container-0
      resources:
        limits:
          cpu: 100m
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      env:                          # 环境变量
      - name: env_key
        value: env_value
    imagePullSecrets:
    - name: default-secret
```

执行如下命令查看容器中的环境变量，可以看到env_key这个环境变量，其值为env_value。

```
$ kubectl exec -it nginx -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=nginx
TERM=xterm
env_key=env_value
```

环境变量还可以引用ConfigMap和Secret，具体使用方法请参见**在环境变量中引用ConfigMap**和**在环境变量中引用Secret**。

容器启动命令

启动容器就是启动主进程，但有些时候，启动主进程前，需要一些准备工作。比如MySQL类的数据库，可能需要一些数据库配置、初始化的工作，这些工作要在最终的MySQL服务器运行之前做完。这些操作，可以在制作镜像时通过在Dockerfile文件中设置ENTRYPOINT或CMD来完成，如下所示的Dockerfile中设置了**ENTRYPOINT ["top", "-b"]**命令，其将会在容器启动时执行。

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
```

实际使用时，只需配置Pod的containers.command参数，该参数是list类型，第一个参数为执行命令，后面均为命令的参数。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
```

```
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    command:                    # 启动命令
    - top
    - "-b"
  imagePullSecrets:
   - name: default-secret
```

容器的生命周期

Kubernetes提供了**容器生命周期钩子**，在容器的生命周期的特定阶段执行调用，比如容器在停止前希望执行某项操作，就可以注册相应的钩子函数。目前提供的生命周期钩子函数如下所示。

- 启动后处理（PostStart）：容器启动后触发。
- 停止前处理（PreStop）：容器停止前触发。

实际使用时，只需配置Pod的lifecycle.postStart或lifecycle.preStop参数，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    lifecycle:
      postStart:                # 启动后处理
        exec:
          command:
          - "/postStart.sh"
      preStop:                  # 停止前处理
        exec:
          command:
          - "/preStop.sh"
  imagePullSecrets:
   - name: default-secret
```

3.3.2 存活探针（Liveness Probe）

存活探针

Kubernetes提供了自愈的能力，具体就是能感知到容器崩溃，然后能够重启这个容器。但是有时候例如Java程序内存泄漏了，程序无法正常工作，但是JVM进程却是一直运行的，对于这种应用本身业务出了问题的情况，Kubernetes提供了Liveness Probe机制，通过检测容器响应是否正常来决定是否重启，这是一种很好的健康检查机制。

毫无疑问，每个Pod最好都定义Liveness Probe，否则Kubernetes无法感知Pod是否正常运行。

Kubernetes支持如下三种探测机制。

- HTTP GET：向容器发送HTTP GET请求，如果Probe收到2xx或3xx，说明容器是健康的。
- TCP Socket：尝试与容器指定端口建立TCP连接，如果连接成功建立，说明容器是健康的。
- Exec：Probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明容器是健康的。

与存活探针对应的还有一个就绪探针（Readiness Probe），将在**就绪探针（Readiness Probe）**中会详细介绍。

HTTP GET

HTTP GET方式是最常见的探测方法，其具体机制是向容器发送HTTP GET请求，如果Probe收到2xx或3xx，说明容器是健康的，定义方法如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:          # liveness probe
      httpGet:              # HTTP GET定义
        path: /
        port: 80
  imagePullSecrets:
  - name: default-secret
```

创建这个Pod。

```
$ kubectl create -f liveness-http.yaml
pod/liveness-http created
```

如上，这个Probe往容器的80端口发送HTTP GET请求，如果请求不成功，Kubernetes会重启容器。

查看Pod详情。

```
$ kubectl describe po liveness-http
Name:               liveness-http
......
Containers:
  liveness:
    ......
    State:          Running
      Started:      Mon, 03 Aug 2020 03:08:55 +0000
    Ready:          True
    Restart Count:  0
    Liveness:       http-get http://:80/ delay=0s timeout=1s period=10s #success=1 #failure=3
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-vssmw (ro)
......
```

可以看到Pod当前状态是Running，Restart Count为0，说明没有重启。如果Restart Count不为0，则说明已经重启。

TCP Socket

TCP Socket尝试与容器指定端口建立TCP连接，如果连接成功建立，说明容器是健康的，定义方法如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:          # liveness probe
      tcpSocket:
        port: 80
  imagePullSecrets:
  - name: default-secret
```

Exec

Exec即执行具体命令，具体机制是Probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明健康，定义方法如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:          # liveness probe
      exec:               # Exec定义
        command:
        - cat
        - /tmp/healthy
  imagePullSecrets:
  - name: default-secret
```

上面定义在容器中执行**cat /tmp/healthy**命令，如果成功执行并返回0，则说明容器是健康的。上面定义中，30秒后命令会删除/tmp/healthy，这会导致Liveness Probe判定Pod处于不健康状态，然后会重启容器。

Liveness Probe 高级配置

上面liveness-http的describe命令回显中有如下行。

```
Liveness: http-get http://:80/ delay=0s timeout=1s period=10s #success=1 #failure=3
```

这一行表示Liveness Probe的具体参数配置，其含义如下：

- delay：延迟，delay=0s，表示在容器启动后立即开始探测，没有延迟时间
- timeout：超时，timeout=1s，表示容器必须在1s内进行响应，否则这次探测记作失败
- period：周期，period=10s，表示每10s探测一次容器
- success：成功，#success=1，表示连续1次成功后记作成功
- failure：失败，#failure=3，表示连续3次失败后会重启容器

以上存活探针表示：容器启动后立即进行探测，如果1s内容器没有给出回应则记作探测失败。每次间隔10s进行一次探测，在探测连续失败3次后重启容器。

这些是创建时默认设置的，您也可以手动配置，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10    # 容器启动后多久开始探测
      timeoutSeconds: 2          # 表示容器必须在2s内做出相应反馈给probe，否则视为探测失败
      periodSeconds: 30          # 探测周期，每30s探测一次
      successThreshold: 1        # 连续探测1次成功表示成功
      failureThreshold: 3       # 连续探测3次失败表示失败
```

initialDelaySeconds一般要设置大于0，这是由于很多情况下容器虽然启动成功，但应用就绪也需要一定的时间，需要等就绪时间之后才能返回成功，否则就会导致probe经常失败。

另外failureThreshold可以设置多次循环探测，这样在实际应用中健康检查的程序就不需要多次循环，这一点在开发应用时需要注意。

配置有效的 Liveness Probe

- **Liveness Probe应该检查什么**

  一个好的Liveness Probe应该检查应用内部所有关键部分是否健康，并使用一个专有的URL访问，例如/health，当访问/health 时执行这个功能，然后返回对应结果。这里要注意不能做鉴权，不然probe就会一直失败导致陷入重启的死循环。

  另外检查只能限制在应用内部，不能检查依赖外部的部分，例如当前端web server不能连接数据库时，这个就不能看成web server不健康。

- **Liveness Probe必须轻量**

  Liveness Probe不能占用过多的资源，且不能占用过长的时间，否则所有资源都在做健康检查，这就没有意义了。例如Java应用，就最好用HTTP GET方式，如果用Exec方式，JVM启动就占用了非常多的资源。

3.3.3 Label：组织 Pod 的利器

为什么需要 Label

当资源变得非常多的时候，如何分类管理就非常重要了，Kubernetes提供了一种机制来为资源分类，那就是Label（标签）。Label非常简单，但是却很强大，Kubernetes中几乎所有资源都可以用Label来组织。

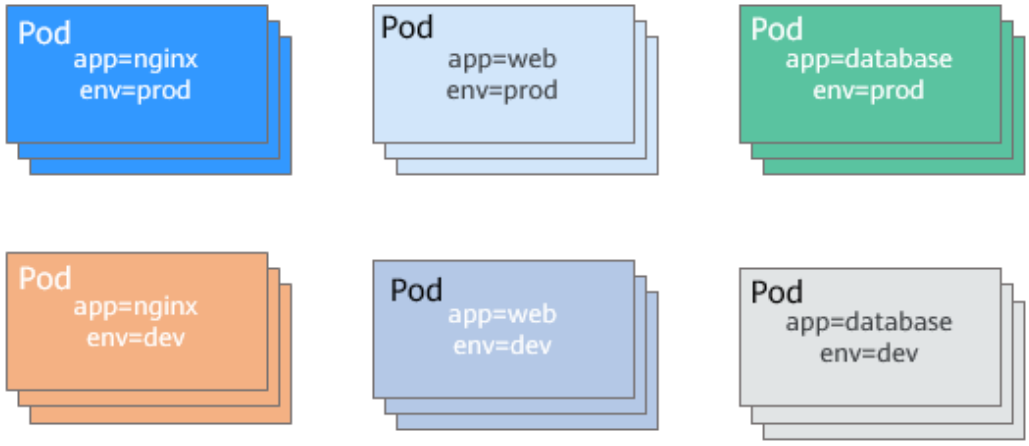
Label的具体形式是key-value的标记对，可以在创建资源的时候设置，也可以在后期添加和修改。

以Pod为例，当Pod变得多起来后，就显得杂乱且难以管理，如下图所示。

图 3-8 没有分类组织的 Pod

如果为Pod打上不同标签，那情况就完全不同了，如下图所示。

图 3-9 使用 Label 组织的 Pod



添加 Label

Label的形式为key-value形式，使用非常简单，如下，为Pod设置了app=nginx和env=prod两个Label。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:                  # 为Pod设置两个Label
    app: nginx
    env: prod
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
```

```
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 200Mi
imagePullSecrets:
- name: default-secret
```

Pod有了Label后，在查询Pod的时候带上--show-labels就可以看到Pod的Label。

```
$ kubectl get pod --show-labels
NAME            READY   STATUS    RESTARTS   AGE   LABELS
nginx           1/1     Running   0          50s   app=nginx,env=prod
```

还可以使用-L只查询某些固定的Label。

```
$ kubectl get pod -L app,env
NAME            READY   STATUS    RESTARTS   AGE   APP     ENV
nginx           1/1     Running   0          1m    nginx   prod
```

对已存在的Pod，可以直接使用kubectl label命令直接添加Label。

```
$ kubectl label pod nginx creation_method=manual
pod/nginx labeled

$ kubectl get pod --show-labels
NAME            READY   STATUS    RESTARTS   AGE   LABELS
nginx           1/1     Running   0          50s   app=nginx, creation_method=manual,env=prod
```

修改 Label

对于已存在的Label，如果要修改的话，需要在命令中带上--overwrite，如下所示。

```
$ kubectl label pod nginx env=debug --overwrite
pod/nginx labeled

$ kubectl get pod --show-labels
NAME            READY   STATUS    RESTARTS   AGE   LABELS
nginx           1/1     Running   0          50s   app=nginx,creation_method=manual,env=debug
```

3.3.4 Namespace：资源分组

为什么需要 Namespace

Label虽然好，但只用Label的话，那Label会非常多，有时候会有重叠，而且每次查询之类的动作都带一堆Label非常不方便。Kubernetes提供了Namespace来做资源组织和划分，使用多Namespace可以将包含很多组件的系统分成不同的组。Namespace也可以用来做多租户划分，这样多个团队可以共用一个集群，使用的资源用Namespace划分开。

不同的Namespace下面可以有相同的名字，Kubernetes中大部分资源可以用Namespace划分，不过有些资源不行，它们属于全局资源，不属于某一个Namespace，后面会逐步接触到。

通过如下命令可以查询到当前集群下的Namespace。

```
$ kubectl get ns
NAME              STATUS   AGE
default           Active   36m
kube-node-realease Active   36m
kube-public       Active   36m
kube-system       Active   36m
```

到目前为止都是在default Namespace下操作，当使用kubectl get而不指定Namespace时，默认为default Namespace。

看下kube-system下面有些什么东西。

```
$ kubectl get po --namespace=kube-system
NAME                                  READY   STATUS    RESTARTS   AGE
coredns-7689f8bdf-295rk               1/1     Running   0          9m11s
coredns-7689f8bdf-h7n68               1/1     Running   0          11m
everest-csi-controller-6d796fb9c5-v22df   2/2     Running   0          9m11s
everest-csi-driver-snzrr              1/1     Running   0          12m
everest-csi-driver-ttj28              1/1     Running   0          12m
everest-csi-driver-wtrk6              1/1     Running   0          12m
icagent-2kz8g                         1/1     Running   0          12m
icagent-hjz4h                         1/1     Running   0          12m
icagent-m4bbl                         1/1     Running   0          12m
```

可以看到kube-system有很多Pod，其中coredns是用于做服务发现、everest-csi是用于对接存储服务、icagent是用于对接云监控系统。

这些通用的、必须的应用放在kube-system这个命名空间中，能够做到与其他Pod之间隔离，在其他命名空间中不会看到kube-system这个命名空间中的东西，不会造成影响。

创建 Namespace

使用如下方式定义Namespace。

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

使用kubectl命令创建。

```
$ kubectl create -f custom-namespace.yaml
namespace/custom-namespace created
```

您还可以使用kubectl create namespace命令创建。

```
$ kubectl create namespace custom-namespace
namespace/custom-namespace created
```

在指定Namespace下创建资源。

```
$ kubectl create -f nginx.yaml -n custom-namespace
pod/nginx created
```

这样在default和custom-namespace下，就分别有一个名为nginx的Pod。

Namespace 的隔离说明

Namespace只能做到组织上划分，对运行的对象来说，它不能做到真正的隔离。举例来说，如果两个Namespace下的Pod知道对方的IP，而Kubernetes依赖的底层网络没有提供Namespace之间的网络隔离的话，那这两个Pod就可以互相访问。

3.4 Pod 的编排与调度

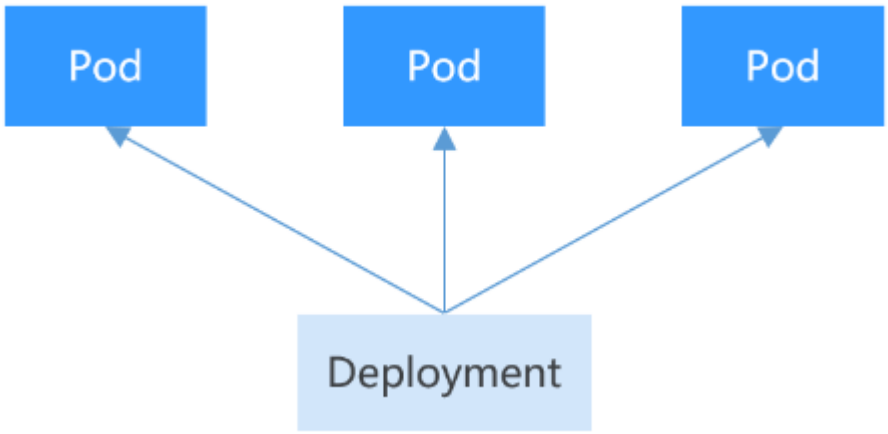
3.4.1 Deployment

Deployment

Pod是Kubernetes创建或部署的最小单位，但是Pod是被设计为相对短暂的一次性实体，Pod可以被驱逐（当节点资源不足时）、随着集群的节点崩溃而消失。

Kubernetes提供了Controller（控制器）来管理Pod，Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力，其中最为常用的就是Deployment。

图 3-10 Deployment



一个Deployment可以包含一个或多个Pod副本，每个Pod副本的角色相同，所以系统会自动为Deployment的多个Pod副本分发请求。

Deployment集成了上线部署、滚动升级、创建副本、恢复上线的功能，在某种程度上，Deployment实现无人值守的上线，大大降低了上线过程的复杂性和操作风险。

创建 Deployment

以下示例为创建一个名为nginx的Deployment负载，使用nginx:latest镜像创建两个Pod，每个Pod占用100m core CPU、200Mi内存。

```
apiVersion: apps/v1      # 注意这里与Pod的区别，Deployment是apps/v1而不是v1
kind: Deployment         # 资源类型为Deployment
metadata:
  name: nginx            # Deployment的名称
spec:
  replicas: 2            # Pod的数量，Deployment会确保一直有2个Pod运行
  selector:              # Label Selector
    matchLabels:
      app: nginx
  template:              # Pod的定义，用于创建Pod，也称为Pod template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

从这个定义中可以看到Deployment的名称为nginx，spec.replicas定义了Pod的数量，即这个Deployment控制2个Pod；spec.selector是Label Selector（标签选择器），表

示这个Deployment会选择Label为app=nginx的Pod；spec.template是Pod的定义，内容与**Pod**中的定义完全一致。

将上面Deployment的定义保存到deployment.yaml文件中，使用kubectl创建这个Deployment。

使用kubectl get查看Deployment和Pod，可以看到**READY**值为2/2，前一个2表示当前有2个Pod运行，后一个2表示期望有2个Pod，**AVAILABLE**为2表示有2个Pod是可用的。

```
$ kubectl create -f deployment.yaml
deployment.apps/nginx created

$ kubectl get deploy
NAME          READY     UP-TO-DATE   AVAILABLE   AGE
nginx         2/2       2            2           4m5s
```

Deployment 如何控制 Pod

继续查询Pod，如下所示。

```
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-7f98958cdf-tdmqk   1/1       Running   0          13s
nginx-7f98958cdf-txckx   1/1       Running   0          13s
```

如果删掉一个Pod，您会发现立马会有一个新的Pod被创建出来，如下所示，这就是前面所说的Deployment会确保有2个Pod在运行，如果删掉一个，Deployment会重新创建一个，如果某个Pod故障或有其他问题，Deployment会自动拉起这个Pod。

```
$ kubectl delete pod nginx-7f98958cdf-txckx

$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-7f98958cdf-tdmqk   1/1       Running   0          21s
nginx-7f98958cdf-tesqr   1/1       Running   0          21s
```

看到有如下两个名为nginx-7f98958cdf-tdmqk和nginx-7f98958cdf-tesqr的Pod， 其中nginx是直接使用Deployment的名称，-7f98958cdf-tdmqk和-7f98958cdf-tesqr是kubernetes随机生成的后缀。

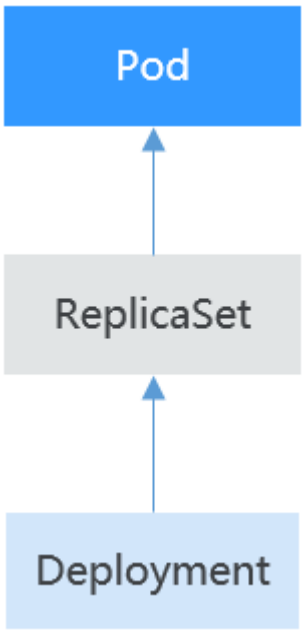
您也许会发现这两个后缀中前面一部分是相同的，都是7f98958cdf，这是因为Deployment不是直接控制Pod的，Deployment是通过一种名为ReplicaSet的控制器控制Pod，通过如下命令可以查询ReplicaSet，其中rs是ReplicaSet的缩写。

```
$ kubectl get rs
NAME              DESIRED   CURRENT   READY     AGE
nginx-7f98958cdf  2         2         2         1m
```

这个ReplicaSet的名称为nginx-7f98958cdf，后缀-7f98958cdf也是随机生成的。

Deployment控制Pod的方式如**图3-11**所示，Deployment控制ReplicaSet，ReplicaSet控制Pod。

图 3-11 Deployment 通过 ReplicaSet 控制 Pod



如果使用kubectl describe命令查看Deployment的详情，您就可以看到ReplicaSet，如下所示，可以看到有一行NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)，而且Events里面事件确是把ReplicaSet的实例扩容到2个。在实际使用中您也许不会直接操作ReplicaSet，但了解Deployment通过控制ReplicaSet来控制Pod会有助于您定位问题。

```
$ kubectl describe deploy nginx
Name:                   nginx
Namespace:              default
CreationTimestamp:      Sun, 16 Dec 2018 19:21:58 +0800
Labels:                 app=nginx

...

NewReplicaSet:   nginx-7f98958cdf (2/2 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  5m    deployment-controller  Scaled up replica set nginx-7f98958cdf to 2
```

升级

在实际应用中，升级是一个常见的场景，Deployment能够很方便的支撑应用升级。

Deployment可以设置不同的升级策略，有如下两种。

- RollingUpdate：滚动升级，即逐步创建新Pod再删除旧Pod，为默认策略。
- Recreate：替换升级，即先把当前Pod删掉再重新创建Pod。

Deployment的升级可以是声明式的，也就是说只需要修改Deployment的YAML定义即可，比如使用kubectl edit命令将上面Deployment中的镜像修改为nginx:alpine。修改完成后再查询ReplicaSet和Pod，发现创建了一个新的ReplicaSet，Pod也重新创建了。

```
$ kubectl edit deploy nginx

$ kubectl get rs
```

```
NAME              DESIRED   CURRENT   READY     AGE
nginx-6f9f58dffd  2         2         2         1m
nginx-7f98958cdf  0         0         0         48m

$ kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
nginx-6f9f58dffd-tdmqk  1/1       Running   0          21s
nginx-6f9f58dffd-tesqr  1/1       Running   0          21s
```

Deployment可以通过maxSurge和maxUnavailable两个参数控制升级过程中同时重新创建Pod的比例，这在很多时候是非常有用，配置如下所示。

```
spec:
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```

- maxSurge：与Deployment中spec.replicas相比，可以有多少个Pod存在，默认值是25%，比如spec.replicas为 4，那升级过程中就不能超过5个Pod存在，即按1个的步伐升级，实际升级过程中会换算成数字，且换算会向上取整。这个值也可以直接设置成数字。
- maxUnavailable：与Deployment中spec.replicas相比，可以有多少个Pod失效，也就是删除的比例，默认值是25%，比如spec.replicas为4，那升级过程中就至少有3个Pod存在，即删除Pod的步伐是1。同样这个值也可以设置成数字。

在前面的例子中，由于spec.replicas是2，如果maxSurge和maxUnavailable都为默认值25%，那实际升级过程中，maxSurge允许最多3个Pod存在（向上取整，2*1.25=2.5，取整为3），而maxUnavailable则不允许有Pod Unavailable（向上取整，2*0.75=1.5，取整为2），也就是说在升级过程中，一直会有2个Pod处于运行状态，每次新建一个Pod，等这个Pod创建成功后再删掉一个旧Pod，直至Pod全部为新Pod。

回滚

回滚也称为回退，即当发现升级出现问题时，让应用回到老的版本。Deployment可以非常方便的回滚到老版本。

例如上面升级的新版镜像有问题，可以执行kubectl rollout undo命令进行回滚。

```
$ kubectl rollout undo deployment nginx
deployment.apps/nginx rolled back
```

Deployment之所以能如此容易的做到回滚，是因为Deployment是通过ReplicaSet控制Pod的，升级后之前ReplicaSet都一直存在，Deployment回滚做的就是使用之前的ReplicaSet再次把Pod创建出来。Deployment中保存ReplicaSet的数量可以使用revisionHistoryLimit参数限制，默认值为10。

3.4.2 StatefulSet

StatefulSet

Deployment控制器下的Pod都有个共同特点，那就是每个Pod除了名称和IP地址不同，其余完全相同。需要的时候，Deployment可以通过Pod模板创建新的Pod；不需要的时候，Deployment就可以删除任意一个Pod。

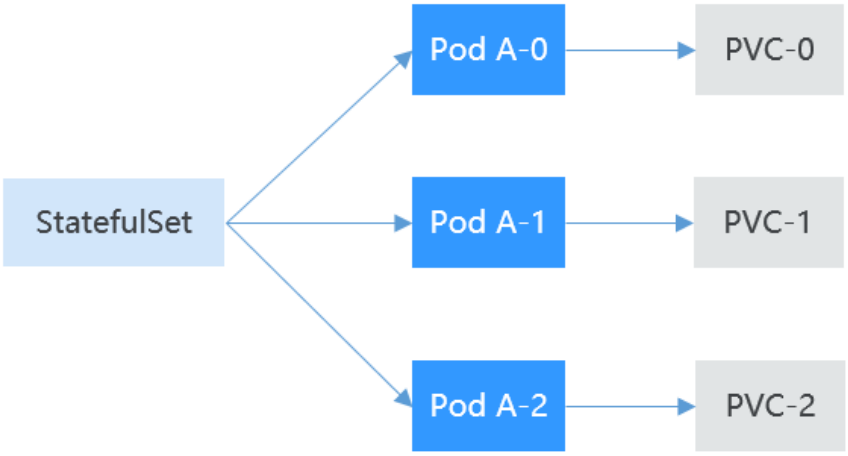
但是在某些场景下，这并不满足需求，比如有些分布式的场景，要求每个Pod都有自己单独的状态时，比如分布式数据库，每个Pod要求有单独的存储，这时Deployment就不能满足需求了。

详细分析下有状态应用的需求，分布式有状态的特点主要是应用中每个部分的角色不同（即分工不同），比如数据库有主备，Pod之间有依赖，对应到Kubernetes中就是对Pod有如下要求：

- Pod能够被别的Pod找到，这就要求Pod有固定的标识。
- 每个Pod有单独存储，Pod被删除恢复后，读取的数据必须还是以前那份，否则状态就会不一致。

Kubernetes提供了StatefulSet来解决这个问题，其具体如下：

1. StatefulSet给每个Pod提供固定名称，Pod名称增加从0-N的固定后缀，Pod重新调度后Pod名称和HostName不变。
2. StatefulSet通过Headless Service给每个Pod提供固定的访问域名，Service的概念会在后面章节中详细介绍。
3. StatefulSet通过创建固定标识的PVC保证Pod重新调度后还是能访问到相同的持久化数据。



下面将通过创建StatefulSet来体验StatefulSet的这些特性。

创建 Headless Service

如前所述，创建Statefulset需要一个Headless Service用于Pod访问，Service的概念会在Service中详细介绍，这里先介绍Headless Service的创建方法。

使用如下文件描述Headless Service，其中：

- spec.clusterIP：必须设置为None，表示Headless Service。
- spec.ports.port：Pod间通信端口号。
- spec.ports.name：Pod间通信端口名称。

```
apiVersion: v1
kind: Service      # 对象类型为Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Pod间通信的端口名称
      port: 80        # Pod间通信的端口号
  selector:
    app: nginx        # 选择标签为app:nginx的Pod
  clusterIP: None     # 必须设置为None，表示Headless Service
```

执行如下命令创建Headless Service。

```
# kubectl create -f headless.yaml
service/nginx created
```

创建完成后可以查询Service。

```
# kubectl get svc
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
nginx        ClusterIP   None          <none>         80/TCP     5s
```

创建 Statefulset

Statefulset的YAML定义与其他对象基本相同，主要有两个差异点：

- serviceName指定了Statefulset使用哪个Headless Service，需要填写Headless Service的名称。
- volumeClaimTemplates是用来申请持久化声明**PVC** ，这里定义了一个名为data的模板，它会为每个Pod创建一个PVC，storageClassName指定了持久化存储的类型，在**PV、PVC和StorageClass**会详细介绍；volumeMounts是为Pod挂载存储。如果不需要存储的话可以删除volumeClaimTemplates和volumeMounts字段。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                        # headless service的名称
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:                     # Pod挂载的存储
          - name:  data
            mountPath:  /usr/share/nginx/html     # 存储挂载到/usr/share/nginx/html
      imagePullSecrets:
        - name: default-secret
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
      - ReadWriteMany
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-nas                 # 持久化存储的类型
```

执行如下命令创建。

```
# kubectl create -f statefulset.yaml
statefulset.apps/nginx created
```

命令执行后，查询StatefulSet和Pod，可以看到Pod的名称后缀从0开始到2，逐个递增。

```
# kubectl get statefulset
NAME    READY   AGE
nginx   3/3     107s

# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          112s
nginx-1   1/1     Running   0          69s
nginx-2   1/1     Running   0          39s
```

此时如果手动删除nginx-1这个Pod，然后再次查询Pod，可以看到StatefulSet重新创建了一个名称相同的Pod，通过创建时间5s可以看出nginx-1是刚刚创建的。

```
# kubectl delete pod nginx-1
pod "nginx-1" deleted

# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          3m4s
nginx-1   1/1     Running   0          5s
nginx-2   1/1     Running   0          1m10s
```

进入容器查看容器的hostname，可以看到同样是nginx-0、nginx-1和nginx-2。

```
# kubectl exec nginx-0 -- sh -c 'hostname'
nginx-0
# kubectl exec nginx-1 -- sh -c 'hostname'
nginx-1
# kubectl exec nginx-2 -- sh -c 'hostname'
nginx-2
```

同时可以查看StatefulSet创建的PVC，可以看到这些PVC，都以“PVC名称-StatefulSet名称-编号”的方式命名，并且处于Bound状态。

```
# kubectl get pvc
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-nginx-0   Bound    pvc-f58bc1a9-6a52-4664-a587-a9a1c904ba29   1Gi        RWX            csi-nas        2m24s
data-nginx-1   Bound    pvc-066e3a3a-fd65-4e65-87cd-6c3fd0ae6485   1Gi        RWX            csi-nas        101s
data-nginx-2   Bound    pvc-a18cf1ce-708b-4e94-af83-766007250b0c   1Gi        RWX            csi-nas        71s
```

StatefulSet 的网络标识

StatefulSet创建后，可以看下Pod是有固定名称的，那Headless Service是如何起作用的呢，那就是使用DNS，为Pod提供固定的域名，这样Pod间就可以使用域名访问，即便Pod被重新创建而导致Pod的IP地址发生变化，这个域名也不会发生变化。

Headless Service创建后，每个Pod的IP都会有下面格式的域名。

<pod-name>.<svc-name>.<namespace>.svc.cluster.local

例如上面的三个Pod的域名就是：

- nginx-0.nginx.default.svc.cluster.local
- nginx-1.nginx.default.svc.cluster.local
- nginx-2.nginx.default.svc.cluster.local

实际访问时可以省略后面的**.<namespace>.svc.cluster.local**。

下面命令会使用tutum/dnsutils镜像创建一个Pod，进入这个Pod的容器，使用nslookup命令查看Pod对应的域名，可以发现能解析出Pod的IP地址。这里可以看到DNS服务器的地址是10.247.3.10，这是在创建CCE集群时默认安装CoreDNS插件，用于提供DNS服务，后续在**Kubernetes网络**会详细介绍CoreDNS的作用。

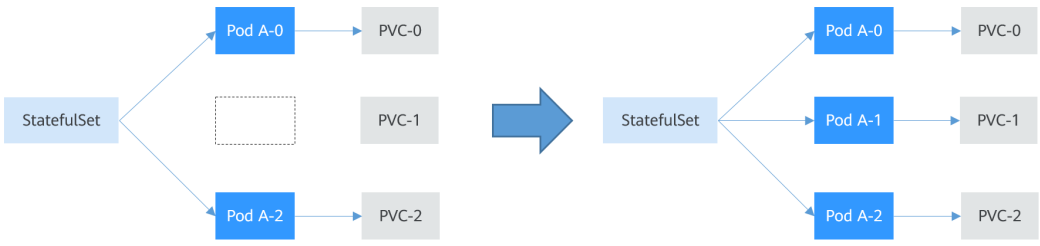
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # nslookup nginx-1.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-1.nginx.default.svc.cluster.local
Address: 172.16.0.18

/ # nslookup nginx-2.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-2.nginx.default.svc.cluster.local
Address: 172.16.0.19
```

此时如果手动删除这两个Pod，查询被StatefulSet重新创建的Pod的IP，然后使用nslookup命令解析Pod的域名，可以发现nginx-0.nginx和nginx-1.nginx仍然能解析到对应的Pod。这就保证了StatefulSet网络标识不变。

StatefulSet 存储状态

上面说了StatefulSet可以通过PVC做持久化存储，保证Pod重新调度后还是能访问到相同的持久化数据，在删除Pod时，PVC不会被删除。

图 3-12 StatefulSet 的 Pod 重建过程



Pod A-1 删除重建后，PVC-1也会重新绑定到Pod A-1上

下面将通过实际操作验证这一点是如何做到的，执行下面的命令，在nginx-1的目录/usr/share/nginx/html中写入一些内容，例如将index.html的内容修改为“hello world”。

```
# kubectl exec nginx-1 -- sh -c 'echo hello world > /usr/share/nginx/html/index.html'
```

修改完后，如果在Pod中访问“http://localhost”，那就会返回“hello world”。

```
# kubectl exec -it nginx-1 -- curl localhost
hello world
```

此时如果手动删除nginx-1这个Pod，然后再次查询Pod，可以看到StatefulSet重新创建了一个名称相同的Pod，通过创建时间4s可以看出nginx-1是刚刚创建的。

```
# kubectl delete pod nginx-1
pod "nginx-1" deleted

# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          14m
nginx-1   1/1     Running   0          4s
nginx-2   1/1     Running   0          13m
```

再次访问该Pod的index.html页面，会发现仍然返回“hello world”，这说明这个Pod仍然是访问相同的存储。

```
# kubectl exec -it nginx-1 -- curl localhost
hello world
```

3.4.3 Job 和 CronJob

Job 和 CronJob

Job和CronJob是负责批量处理短暂的一次性任务（short lived one-off tasks），即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

- Job：是Kubernetes用来控制批处理型任务的资源对象。批处理业务与长期伺服业务（Deployment、Statefulset）的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出（Pod自动删除）。
- CronJob：是基于时间的Job，就类似于Linux系统的crontab文件中的一行，在指定的时间周期运行指定的Job。

任务负载的这种用完即停止的特性特别适合一次性任务，比如持续集成。

创建 Job

以下是一个Job配置，其计算π到2000位并打印输出。Job结束需要运行50个Pod，这个示例中就是打印π 50次，并行运行5个Pod，Pod如果失败最多重试5次。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  completions: 50           # 运行的次数，即Job结束需要成功运行的Pod个数
  parallelism: 5            # 并行运行Pod的数量，默认为1
  backoffLimit: 5           # 表示失败Pod的重试最大次数，超过这个次数不会继续重试。
  activeDeadlineSeconds: 10  # 表示Pod超期时间，一旦达到这个时间，Job及其所有的Pod都会停止。
  template:                 # Pod定义
    spec:
      containers:
      - name: pi
        image: perl
        command:
        - perl
        - "-Mbignum=bpi"
        - "-wle"
        - print bpi(2000)
      restartPolicy: Never
```

根据completions和parallelism的设置，可以将Job划分为以下几种类型。

表 3-1 任务类型

Job类型	说明	使用示例
一次性Job	创建一个Pod直至其成功结束	数据库迁移
固定结束次数的Job	依次创建一个Pod运行直至completions个成功结束	处理工作队列的Pod
固定结束次数的并行Job	依次创建多个Pod运行直至completions个成功结束	多个Pod同时处理工作队列
并行Job	创建一个或多个Pod直至有一个成功结束	多个Pod同时处理工作队列

创建 CronJob

相比Job，CronJob就是一个加了定时的Job，CronJob执行时是在指定的时间创建出Job，然后由Job创建出Pod。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "0,15,30,45 * * * *"          # 定时相关配置
  jobTemplate:                           # Job的定义
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: main
            image: pi
```

CronJob的格式从前到后就是：

- Minute
- Hour
- Day of month
- Month
- Day of week

如 "0,15,30,45 * * * * " ，前面逗号隔开的是分钟，后面第一个* 表示每小时，第二个 * 表示每个月的哪天，第三个表示每月，第四个表示每周的哪天。

如果您想要每个月的第一天里面每半个小时执行一次，那就可以设置为" 0,30 * 1 * * "
如果您想每个星期天的3am执行一次任务，那就可以设置为 "0 3 * * 0"。

更详细的CronJob格式说明请参见**https://zh.wikipedia.org/wiki/Cron**。

3.4.4 DaemonSet

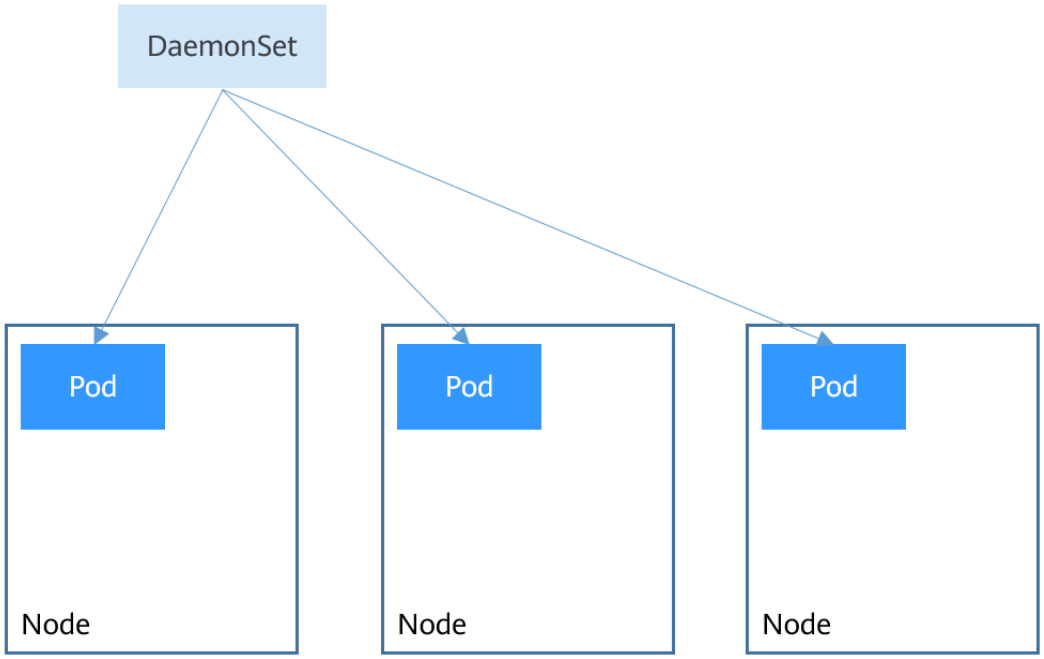
DaemonSet

DaemonSet是这样一种对象（守护进程），它在集群的每个节点上运行一个Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资源监控等，这

类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes的kube-proxy。

DaemonSet跟节点相关，如果节点异常，也不会在其他节点重新创建。

图 3-13 DaemonSet



创建 DaemonSet

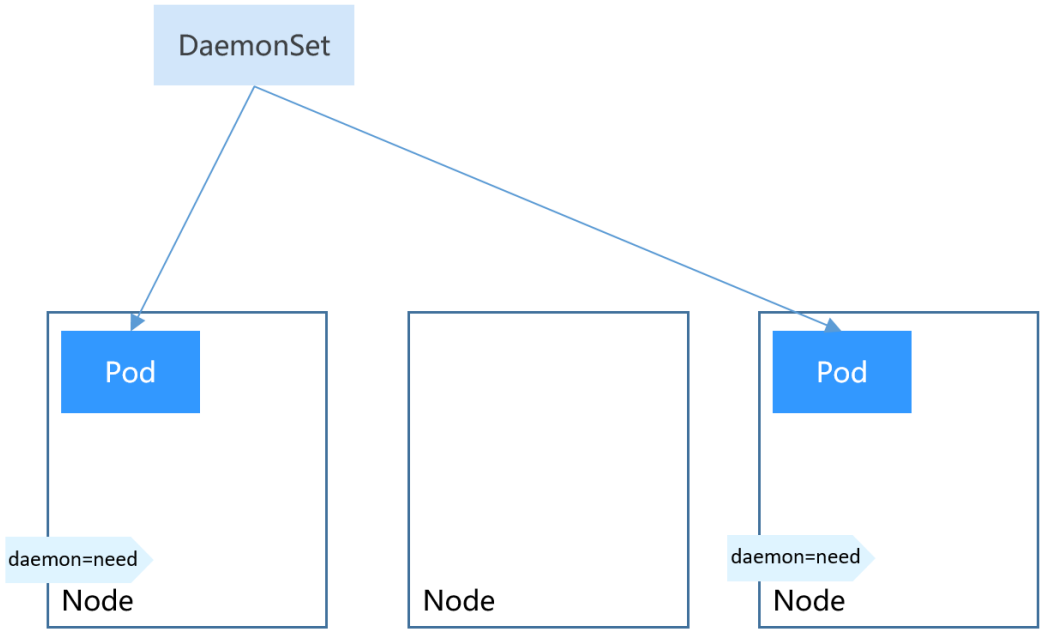
下面是一个DaemonSet的示例。

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
  labels:
    app: nginx-daemonset
spec:
  selector:
    matchLabels:
      app: nginx-daemonset
  template:
    metadata:
      labels:
        app: nginx-daemonset
    spec:
      nodeSelector:                # 节点选择，当节点拥有daemon=need时才在节点上创建Pod
        daemon: need
      containers:
      - name: nginx-daemonset
        image: nginx:alpine
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
      imagePullSecrets:
      - name: default-secret
```

这里可以看出没有Deployment或StatefulSet中的replicas参数，因为是每个节点固定一个。

Pod模板中有个nodeSelector，指定了只在有“daemon=need”的节点上才创建Pod，如下图所示，DaemonSet只在指定标签的节点上创建Pod。如果需要在每一个节点上创建Pod可以删除该标签。

图 3-14 DaemonSet 在指定标签的节点上创建 Pod



创建DaemonSet：

```
$ kubectl create -f daemonset.yaml
daemonset.apps/nginx-daemonset created
```

查询发现nginx-daemonset没有Pod创建。

```
$ kubectl get ds
NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   0         0         0       0            0           daemon=need     16s

$ kubectl get pods
No resources found in default namespace.
```

这是因为节点上没有daemon=need这个标签，使用如下命令可以查询节点的标签。

```
$ kubectl get node --show-labels
NAME            STATUS   ROLES    AGE   VERSION                        LABELS
192.168.0.212   Ready    <none>   83m   v1.15.6-r1-20.3.0.2.B001-15.30.2   beta.kubernetes.io/arch=amd64 ...
192.168.0.94    Ready    <none>   83m   v1.15.6-r1-20.3.0.2.B001-15.30.2   beta.kubernetes.io/arch=amd64 ...
192.168.0.97    Ready    <none>   83m   v1.15.6-r1-20.3.0.2.B001-15.30.2   beta.kubernetes.io/arch=amd64 ...
```

给192.168.0.212这个节点打上标签，然后再查询，发现已经创建了一个Pod，并且这个Pod是在192.168.0.212这个节点上。

```
$ kubectl label node 192.168.0.212 daemon=need
node/192.168.0.212 labeled

$ kubectl get ds
NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   1         1         0       1            0           daemon=need     116s
```

```
$ kubectl get pod -o wide
NAME                 READY   STATUS    RESTARTS   AGE   IP          NODE
nginx-daemonset-g9b7j   1/1     Running   0          18s   172.16.3.0   192.168.0.212
```

再给192.168.0.94这个节点打上标签，发现又创建了一个Pod：

```
$ kubectl label node 192.168.0.94 daemon=need
node/192.168.0.94 labeled

$ kubectl get ds
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   2         2         1       2            1           daemon=need     2m29s

$ kubectl get pod -o wide
NAME                 READY   STATUS              RESTARTS   AGE   IP          NODE
nginx-daemonset-6jjxz   0/1     ContainerCreating   0          8s    <none>      192.168.0.94
nginx-daemonset-g9b7j   1/1     Running             0          42s   172.16.3.0   192.168.0.212
```

如果修改掉192.168.0.94节点的标签，可以发现DaemonSet会删除这个节点上的Pod。

```
$ kubectl label node 192.168.0.94 daemon=no --overwrite
node/192.168.0.94 labeled

$ kubectl get ds
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   1         1         1       1            1           daemon=need     4m5s

$ kubectl get pod -o wide
NAME                 READY   STATUS    RESTARTS   AGE     IP          NODE
nginx-daemonset-g9b7j   1/1     Running   0          2m23s   172.16.3.0   192.168.0.212
```

3.4.5 亲和与反亲和调度

在**DaemonSet**中讲到使用nodeSelector选择Pod要部署的节点，其实Kubernetes还支持更精细、更灵活的调度机制，那就是亲和（affinity）与反亲和（anti-affinity）调度。

Kubernetes支持节点和Pod两个层级的亲和与反亲和。通过配置亲和与反亲和规则，可以允许您指定硬性限制或者偏好，例如将前台Pod和后台Pod部署在一起、某类应用部署到某些特定的节点、不同应用部署到不同的节点等等。

Node Affinity（节点亲和）

您肯定也猜到了亲和性规则的基础肯定也是标签，先来看CCE集群中节点上有些什么标签。

```
$ kubectl describe node 192.168.0.212
Name:               192.168.0.212
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    failure-domain.beta.kubernetes.io/is-baremetal=false
                    failure-domain.beta.kubernetes.io/region=eu-west-0
                    failure-domain.beta.kubernetes.io/zone=eu-west-0a
                    kubernetes.io/arch=amd64
                    kubernetes.io/availablezone=eu-west-0a
                    kubernetes.io/eniquota=12
                    kubernetes.io/hostname=192.168.0.212
                    kubernetes.io/os=linux
                    node.kubernetes.io/subnetid=fd43acad-33e7-48b2-a85a-24833f362e0e
                    os.architecture=amd64
                    os.name=EulerOS_2.0_SP5
                    os.version=3.10.0-862.14.1.5.h328.eulerosv2r7.x86_64
```

这些标签都是在创建节点的时候CCE会自动添加上的，下面介绍几个在调度中会用到比较多的标签。

- failure-domain.beta.kubernetes.io/region：表示节点所在的区域，如果上面这个节点标签值为eu-west-0，表示节点在法国-巴黎区域。
- failure-domain.beta.kubernetes.io/zone：表示节点所在的可用区（availability zone）。
- kubernetes.io/hostname：节点的hostname。

另外在**Label：组织Pod的利器**章节还介绍自定义标签，通常情况下，对于一个大型Kubernetes集群，肯定会根据业务需要定义很多标签。

在**DaemonSet**中介绍了nodeSelector，通过nodeSelector可以让Pod只部署在具有特定标签的节点上。如下所示，Pod只会部署在拥有gpu=true这个标签的节点上。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeSelector:              # 节点选择，当节点拥有gpu=true时才在节点上创建Pod
    gpu: true
...
```

通过节点亲和性规则配置，也可以做到同样的事情，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  gpu
  labels:
    app:  gpu
spec:
  selector:
    matchLabels:
      app: gpu
  replicas: 3
  template:
    metadata:
      labels:
        app:  gpu
    spec:
      containers:
      - image:  nginx:alpine
        name:  gpu
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
          limits:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: gpu
                operator: In
                values:
                - "true"
```

看起来这要复杂很多，但这种方式可以得到更强的表达能力，后面会进一步介绍。

这里affinity表示亲和，nodeAffinity表示节点亲和，requiredDuringSchedulingIgnoredDuringExecution非常长，不过可以将这个分作两段来看：

- 前半段requiredDuringScheduling表示下面定义的规则必须强制满足（require）。
- 后半段IgnoredDuringExecution表示不会影响已经在节点上运行的Pod，目前Kubernetes提供的规则都是以IgnoredDuringExecution结尾的，因为当前的节点亲缘性规则只会影响正在被调度的pod，最终，kubernetes也会支持RequiredDuringExecution，即去除节点上的某个标签，那些需要节点包含该标签的pod将会被剔除。

另外操作符operator的值为In，表示标签值需要在values的列表中，其他operator取值如下。

- NotIn：标签的值不在某个列表中
- Exists：某个标签存在
- DoesNotExist：某个标签不存在
- 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）

需要说明的是并没有nodeAntiAffinity（节点反亲和），因为NotIn和DoesNotExist可以提供相同的功能。

下面来验证这段规则是否生效，首先给192.168.0.212这个节点打上gpu=true的标签。

```
$ kubectl label node 192.168.0.212 gpu=true
node/192.168.0.212 labeled

$ kubectl get node -L gpu
NAME          STATUS   ROLES    AGE   VERSION                        GPU
192.168.0.212   Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
192.168.0.94    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.97    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
```

创建这个Deployment，可以发现所有的Pod都部署在了192.168.0.212这个节点上。

```
$ kubectl create -f affinity.yaml
deployment.apps/gpu created

$ kubectl get pod -o wide
NAME                   READY   STATUS    RESTARTS   AGE   IP           NODE
gpu-6df65c44cf-42xw4     1/1     Running   0          15s   172.16.0.37   192.168.0.212
gpu-6df65c44cf-jzjvs     1/1     Running   0          15s   172.16.0.36   192.168.0.212
gpu-6df65c44cf-zv5cl     1/1     Running   0          15s   172.16.0.38   192.168.0.212
```

节点优先选择规则

上面讲的requiredDuringSchedulingIgnoredDuringExecution是一种**强制**选择的规则，节点亲和还有一种优先选择规则，即preferredDuringSchedulingIgnoredDuringExecution，表示会根据规则**优先**选择哪些节点。

为演示这个效果，先为上面的集群添加一个节点，且这个节点跟另外三个节点不在同一个可用区，创建完之后查询节点的可用区标签，如下所示，新添加的节点在eu-west-0a这个可用区。

```
$ kubectl get node -L failure-domain.beta.kubernetes.io/zone,gpu
NAME          STATUS   ROLES    AGE     VERSION                        ZONE         GPU
192.168.0.100   Ready    <none>   7h23m   v1.15.6-r1-20.3.0.2.B001-15.30.2   eu-west-0a
192.168.0.212   Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   eu-west-0b   true
192.168.0.94    Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   eu-west-0b
192.168.0.97    Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   eu-west-0b
```

下面定义一个Deployment，要求Pod优先部署在可用区eu-west-0a的节点上，可以像下面这样定义，使用preferredDuringSchedulingIgnoredDuringExecution规则，给eu-west-0a设置权重（weight）为80，而gpu=true权重为20，这样Pod就优先部署在eu-west-0a的节点上。

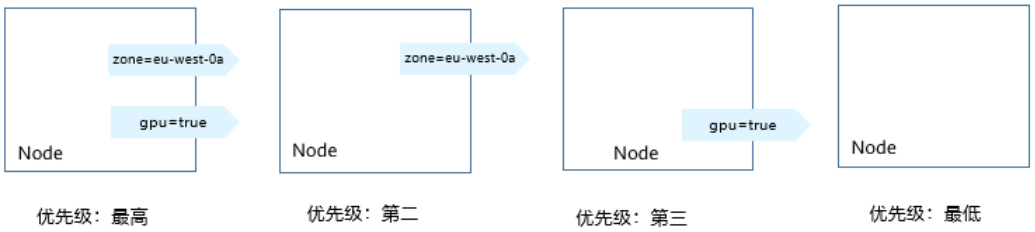
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  gpu
  labels:
    app:  gpu
spec:
  selector:
    matchLabels:
      app: gpu
  replicas: 10
  template:
    metadata:
      labels:
        app:  gpu
    spec:
      containers:
      - image:  nginx:alpine
        name:  gpu
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 80
            preference:
              matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                - eu-west-0a
          - weight: 20
            preference:
              matchExpressions:
              - key: gpu
                operator: In
                values:
                - "true"
```

来看实际部署后的情况，可以看到部署到192.168.0.212这个节点上的Pod有5个，而192.168.0.100上只有2个。

```
$ kubectl create -f affinity2.yaml
deployment.apps/gpu created

$ kubectl get po -o wide
NAME                   READY   STATUS    RESTARTS   AGE     IP            NODE
gpu-585455d466-5bmcz   1/1     Running   0          2m29s   172.16.0.44   192.168.0.212
gpu-585455d466-cg2l6   1/1     Running   0          2m29s   172.16.0.63   192.168.0.97
gpu-585455d466-f2bt2   1/1     Running   0          2m29s   172.16.0.79   192.168.0.100
gpu-585455d466-hdb5n   1/1     Running   0          2m29s   172.16.0.42   192.168.0.212
gpu-585455d466-hkgvz   1/1     Running   0          2m29s   172.16.0.43   192.168.0.212
gpu-585455d466-mngvn   1/1     Running   0          2m29s   172.16.0.48   192.168.0.97
gpu-585455d466-s26qs   1/1     Running   0          2m29s   172.16.0.62   192.168.0.97
gpu-585455d466-sxtzm   1/1     Running   0          2m29s   172.16.0.45   192.168.0.212
gpu-585455d466-t56cm   1/1     Running   0          2m29s   172.16.0.64   192.168.0.100
gpu-585455d466-t5w5x   1/1     Running   0          2m29s   172.16.0.41   192.168.0.212
```

上面这个例子中，对于节点排序优先级如下所示，有个两个标签的节点排序最高，只有eu-west-0a标签的节点排序第二（权重为80），只有gpu=true的节点排序第三，没有的节点排序最低。

图 3-15 优先级排序顺序



这里您看到Pod并没有调度到192.168.0.94这个节点上，这是因为这个节点上部署了很多其他Pod，资源使用较多，所以并没有往这个节点上调度，这也侧面说明preferredDuringSchedulingIgnoredDuringExecution是优先规则，而不是强制规则。

Pod Affinity（Pod 亲和）

节点亲和的规则只能影响Pod和节点之间的亲和，Kubernetes还支持Pod和Pod之间的亲和，例如将应用的前端和后端部署在一起，从而减少访问延迟。Pod亲和同样有requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution两种规则。

来看下面这个例子，假设有个应用的后端已经创建，且带有app=backend的标签。

```
$ kubectl get po -o wide
NAME                    READY   STATUS    RESTARTS   AGE     IP           NODE
backend-658f6cb858-dlrz8   1/1     Running   0          2m36s   172.16.0.67   192.168.0.100
```

将前端frontend的pod部署在backend一起时，可以做如下Pod亲和规则配置。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
```

```
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - topologyKey: kubernetes.io/hostname
        labelSelector:
          matchLabels:
            app: backend
```

创建frontend然后查看，可以看到frontend都创建到跟backend一样的节点上了。

```
$ kubectl create -f affinity3.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-dlrz8    1/1     Running   0          5m38s   172.16.0.67   192.168.0.100
frontend-67ff9b7b97-dsqzn   1/1     Running   0          6s      172.16.0.70   192.168.0.100
frontend-67ff9b7b97-hxm5t   1/1     Running   0          6s      172.16.0.71   192.168.0.100
frontend-67ff9b7b97-z8pdb   1/1     Running   0          6s      172.16.0.72   192.168.0.100
```

这里有个topologyKey字段，意思是先圈定topologyKey指定的范围，然后再选择下面规则定义的内容。这里每个节点上都有kubernetes.io/hostname，所以看不出topologyKey起到的作用。

如果backend有两个Pod，分别在不同的节点上。

```
$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-5bpd6    1/1     Running   0          23m     172.16.0.40   192.168.0.97
backend-658f6cb858-dlrz8    1/1     Running   0          2m36s   172.16.0.67   192.168.0.100
```

给192.168.0.97和192.168.0.94打一个perfer=true的标签。

```
$ kubectl label node 192.168.0.97 perfer=true
node/192.168.0.97 labeled
$ kubectl label node 192.168.0.94 perfer=true
node/192.168.0.94 labeled

$ kubectl get node -L perfer
NAME          STATUS   ROLES    AGE   VERSION                          PERFER
192.168.0.100   Ready    <none>   44m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.212   Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.94    Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
192.168.0.97    Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
```

将podAffinity的topologyKey定义为perfer。

```
    affinity:
      podAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - topologyKey: perfer
          labelSelector:
            matchLabels:
              app: backend
```

调度时，先圈定拥有perfer标签的节点，这里也就是192.168.0.97和192.168.0.94，然后再匹配app=backend标签的Pod，从而frontend就会全部部署在192.168.0.97上。

```
$ kubectl create -f affinity3.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-5bpd6    1/1     Running   0          26m     172.16.0.40   192.168.0.97
backend-658f6cb858-dlrz8    1/1     Running   0          5m38s   172.16.0.67   192.168.0.100
frontend-67ff9b7b97-dsqzn   1/1     Running   0          6s      172.16.0.70   192.168.0.97
frontend-67ff9b7b97-hxm5t   1/1     Running   0          6s      172.16.0.71   192.168.0.97
frontend-67ff9b7b97-z8pdb   1/1     Running   0          6s      172.16.0.72   192.168.0.97
```

Pod AntiAffinity（Pod 反亲和）

前面讲了Pod的亲和，通过亲和将Pod部署在一起，有时候需求却恰恰相反，需要将Pod分开部署，例如Pod之间部署在一起会影响性能的情况。

下面例子中定义了反亲和规则，这个规则表示Pod不能调度到拥有app=frontend标签Pod的节点上，也就是下面将frontend分别调度到不同的节点上（每个节点只有一个Pod）。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 5
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                app: frontend
```

创建并查看，可以看到每个节点上只有一个frontend的Pod，还有一个在Pending，因为在部署第5个时4个节点上都有了app=frontend的Pod，所以第5个一直是Pending。

```
$ kubectl create -f affinity4.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP            NODE
frontend-6f686d8d87-8dlsc   1/1     Running   0          18s   172.16.0.76   192.168.0.100
frontend-6f686d8d87-d6l8p   0/1     Pending   0          18s   <none>        <none>
frontend-6f686d8d87-hgcq2   1/1     Running   0          18s   172.16.0.54   192.168.0.97
frontend-6f686d8d87-q7cfq   1/1     Running   0          18s   172.16.0.47   192.168.0.212
frontend-6f686d8d87-xl8hx   1/1     Running   0          18s   172.16.0.23   192.168.0.94
```

3.5 配置管理

3.5.1 ConfigMap

ConfigMap是一种用于存储应用所需配置信息的资源类型，用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。

通过ConfigMap可以方便的做到配置解耦，使得不同环境有不同的配置。

创建 ConfigMap

下面示例创建了一个名为configmap-test的ConfigMap，ConfigMap的配置数据在data字段下定义。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-test
data:                   # 配置数据
  property_1: Hello
  property_2: World
```

在环境变量中引用 ConfigMap

ConfigMap最为常见的使用方式就是在环境变量和Volume中引用。

例如下面例子中，引用了configmap-test的property_1，将其作为环境变量EXAMPLE_PROPERTY_1的值，这样容器启动后里面EXAMPLE_PROPERTY_1的值就是property_1的值，即“Hello”。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:
    - name: EXAMPLE_PROPERTY_1
      valueFrom:
        configMapKeyRef:          # 引用ConfigMap
          name: configmap-test
          key: property_1
  imagePullSecrets:
  - name: default-secret
```

在 Volume 中引用 ConfigMap

在Volume中引用ConfigMap，就是通过文件的方式直接将ConfigMap的每条数据填入Volume，每条数据是一个文件，键就是文件名，键值就是文件内容。

如下示例中，创建一个名为vol-configmap的Volume，这个Volume引用名为“configmap-test”的ConfigMap，再将Volume挂载到容器的“/tmp”路径下。Pod创建成功后，在容器的“/tmp”路径下，就有两个文件property_1和property_2，它们的值分别为“Hello”和“World”。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    volumeMounts:
    - name: vol-configmap          # 挂载名为vol-configmap的Volume
      mountPath: "/tmp"
  imagePullSecrets:
  - name: default-secret
  volumes:
  - name: vol-configmap
    configMap:                     # 引用ConfigMap
      name: configmap-test
```

3.5.2 Secret

Secret是一种加密存储的资源对象，您可以将认证信息、证书、私钥等保存在Secret中，而不需要把这些敏感数据暴露到镜像或者Pod定义中，从而更加安全和灵活。

Secret与ConfigMap非常像，都是key-value键值对形式，使用方式也相同，不同的是Secret会加密存储，所以适用于存储敏感信息。

Base64 编码

Secret与ConfigMap相同，是以键值对形式保存数据，所不同的是在创建时，Secret的Value必须使用Base64编码。

对字符串进行Base64编码，可以直接使用“echo -n 要编码的内容 | base64”命令即可，示例如下：

```
root@ubuntu:~# echo -n "3306" | base64
MzMwNg==
```

创建 Secret

如下示例中定义的Secret中包含两条Key-Value。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  key1: aGVsbG8gd29ybGQ=   # "hello world" Base64编码后的值
  key2: MzMwNg==           # "3306" Base64编码后的值
```

在环境变量中引用 Secret

Secret最常见的用法是作为环境变量注入到容器中，如下示例。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
```

```
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:
    - name: key
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: key1
  imagePullSecrets:
  - name: default-secret
```

在 Volume 中引用 Secret

在Volume中引用Secret，就是通过文件的方式直接将Secret的每条数据填入Volume，每条数据是一个文件，键就是文件名，键值就是文件内容。

如下示例中，创建一个名为vol-secret的Volume，这个Volume引用名为“mysecret”的Secret，再将Volume挂载到容器的“/tmp”路径下。Pod创建成功后，在容器的“/tmp”路径下，就有两个文件key1和key2。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    volumeMounts:
    - name: vol-secret              # 挂载名为vol-secret的Volume
      mountPath: "/tmp"
  imagePullSecrets:
  - name: default-secret
  volumes:
  - name: vol-secret
    secret:                       # 引用Secret
      secretName: mysecret
```

进入Pod容器中，可以在/tmp目录下发现key1和key2两个文件，并看到文件中的值是base64解码后的值，分别为“hello world”和“3306”。

3.6 Kubernetes 网络

3.6.1 容器网络

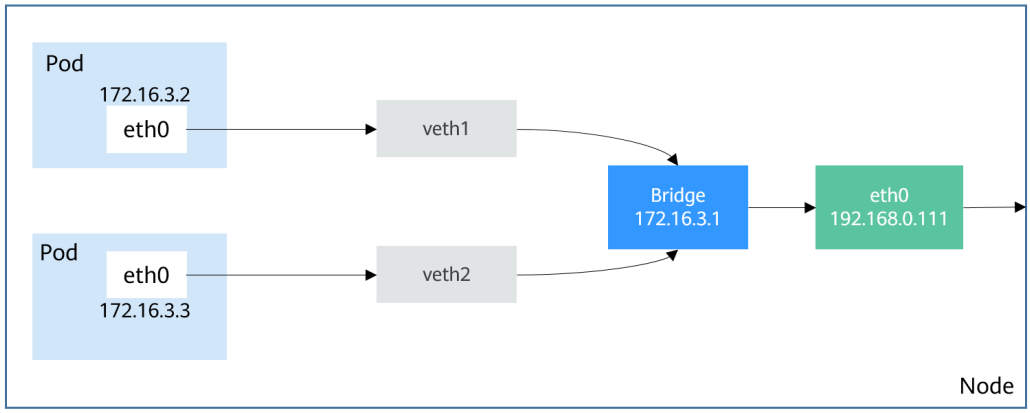
Kubernetes本身并不负责网络通信，Kubernetes提供了容器网络接口CNI（Container Network Interface），具体的网络通信交给CNI插件来负责，开源的CNI插件非常多，

像Flannel、Calico等，CCE也专门为Kubernetes定制了CNI插件，使得Kubernetes可以使用VPC网络。

Kubernetes虽然不负责网络，但要求集群中的Pod能够互相通信，且Pod必须通过非NAT网络连接，即收到的数据包的源IP就是发送数据包Pod的IP。同时Pod与节点之间的通信也是通过非NAT网络。但是Pod访问集群外部时源IP会被修改成节点的IP。

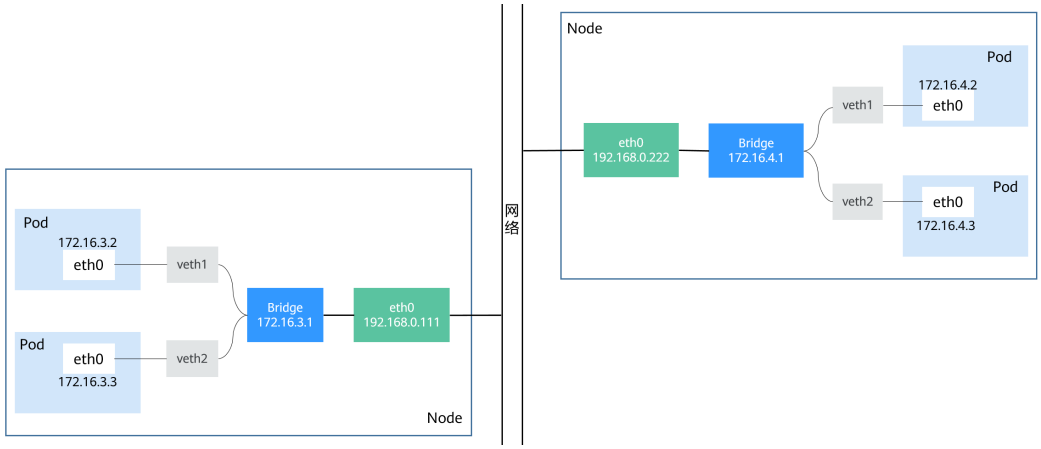
Pod内部是通过虚拟Ethernet接口对（Veth pair）与Pod外部连接，Veth pair就像一根网线，一端留在Pod内部，一端在Pod之外。而同一个节点上的Pod通过网桥（Linux Bridge）通信，如下图所示。

图 3-16 同一个节点中的 Pod 通信



不同节点间的网桥连接有很多种方式，这跟具体实现相关。但集群要求Pod的地址唯一，所以跨节点的网桥通常使用不同的地址段，以防止Pod的IP地址重复。

图 3-17 不同节点上的 Pod 通信



以上就是容器网络底层视图，后面将进一步介绍Kubernetes是如何在此基础上向用户提供访问方案，具体请参见Service和Ingress。

3.6.2 Service

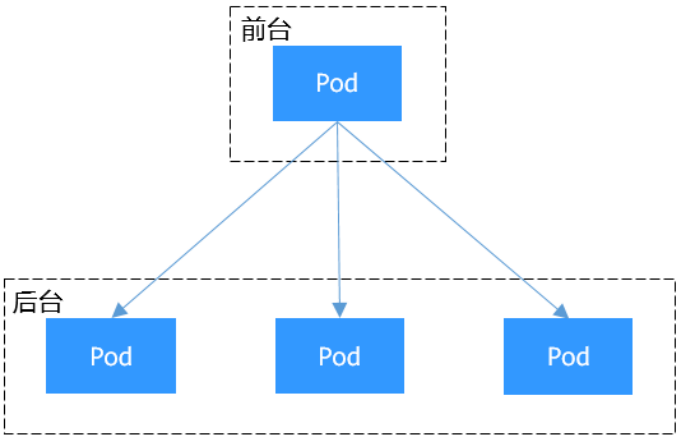
直接访问 Pod 的问题

Pod创建完成后，如何访问Pod呢？直接访问Pod会有如下几个问题：

- Pod会随时被Deployment这样的控制器删除重建，那访问Pod的结果就会变得不可预知。
- Pod的IP地址是在Pod启动后才被分配，在启动前并不知道Pod的IP地址。
- 应用往往都是由多个运行相同镜像的一组Pod组成，逐个访问Pod也变得不现实。

举个例子，假设有这样一个应用程序，使用Deployment创建了前台和后台，前台会调用后台做一些计算处理，如图3-18所示。后台运行了3个Pod，这些Pod是相互独立且可被替换的，当Pod出现状况被重建时，新建的Pod的IP地址是新IP，前台的Pod无法直接感知。

图 3-18 Pod 间访问

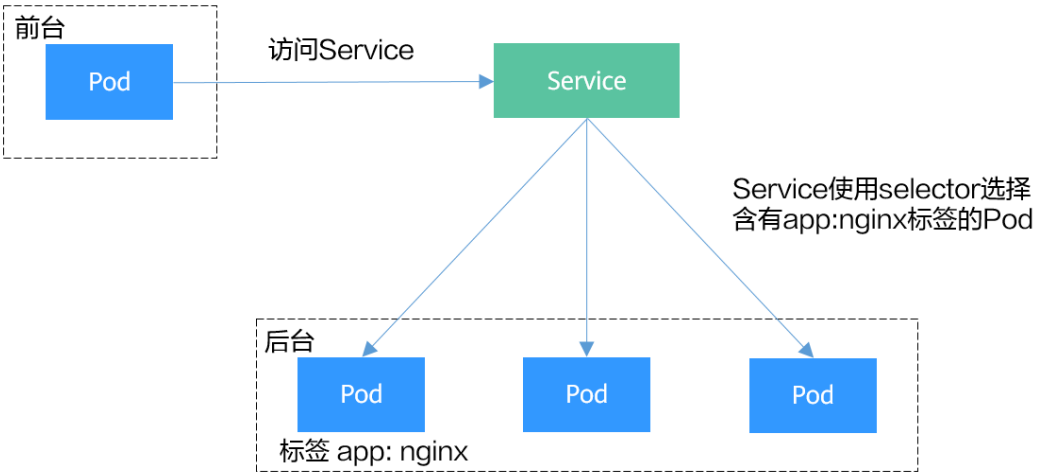


使用 Service 解决 Pod 的访问问题

Kubernetes中的Service对象就是用来解决上述Pod访问问题的。Service有一个固定IP地址（在创建CCE集群时有一个服务网段的设置，这个网段专门用于给Service分配IP地址），Service将访问它的流量转发给Pod，具体转发给哪些Pod通过Label来选择，而且Service可以给这些Pod做负载均衡。

那么对于上面的例子，为后台添加一个Service，通过Service来访问Pod，这样前台Pod就无需感知后台Pod的变化，如图3-19所示。

图 3-19 通过 Service 访问 Pod



创建后台 Pod

首先创建一个3副本的Deployment，即3个Pod，且Pod上带有标签“app: nginx”，具体如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

创建 Service

下面示例创建一个名为“nginx”的Service，通过selector选择到标签“app:nginx”的Pod，目标Pod的端口为80，Service对外暴露的端口为8080。

访问服务只需要通过“服务名称:对外暴露的端口”接口，对应本例即“nginx:8080”。这样，在其他Pod中，只需要通过“nginx:8080”就可以访问到“nginx”关联的Pod。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx        # Service的名称
spec:
  selector:          # Label Selector，选择包含app=nginx标签的Pod
    app: nginx
  ports:
  - name: service0
    targetPort: 80   # Pod的端口
    port: 8080       # Service对外暴露的端口
    protocol: TCP    # 转发协议类型，支持TCP和UDP
  type: ClusterIP    # Service的类型
```

将上面Service的定义保存到nginx-svc.yaml文件中，使用kubectl创建这个Service。

```
$ kubectl create -f nginx-svc.yaml
service/nginx created

$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.247.0.1       <none>        443/TCP    7h19m
nginx        ClusterIP   10.247.124.252   <none>        8080/TCP   5h48m
```

您可以看到Service有个Cluster IP，这个IP是固定不变的，除非Service被删除，所以您也可以使用ClusterIP在集群内部访问Service。

下面创建一个Pod并进入容器，使用ClusterIP访问Pod，可以看到能直接返回内容。

```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 10.247.124.252:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

使用 ServiceName 访问 Service

通过DNS进行域名解析后，可以使用“ServiceName:Port”访问Service，这也是Kubernetes中最常用的一种使用方式。在创建CCE集群的时候，会默认要求安装CoreDNS插件，在kube-system命名空间下可以查看到CoreDNS的Pod。

```
$ kubectl get po --namespace=kube-system
NAME                              READY   STATUS    RESTARTS   AGE
coredns-7689f8bdf-295rk           1/1     Running   0          9m11s
coredns-7689f8bdf-h7n68           1/1     Running   0          11m
```

CoreDNS安装成功后会成为DNS服务器，当创建Service后，CoreDNS会将Service的名称与IP记录起来，这样Pod就可以通过向CoreDNS查询Service的名称获得Service的IP地址。

访问时通过nginx.<namespace>.svc.cluster.local访问，其中nginx为Service的名称，<namespace>为命名空间名称，svc.cluster.local为域名后缀，在实际使用中，在同一个命名空间下可以省略<namespace>.svc.cluster.local，直接使用ServiceName即可。

例如上面创建的名为nginx的Service，直接通过“nginx:8080”就可以访问到Service，进而访问后台Pod。

使用ServiceName的方式有个主要的优点就是可以在开发应用程序时可以将ServiceName写在程序中，这样无需感知具体Service的IP地址。

下面创建一个Pod并进入容器，查询nginx域名的地址，可以发现是解析出nginx这个Service的IP地址10.247.124.252；同时访问Pod的域名，可以看到能直接返回内容。

```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx
Server:         10.247.3.10
Address:    10.247.3.10#53

Name:    nginx.default.svc.cluster.local
Address: 10.247.124.252

/ # curl nginx:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Service 是如何做到服务发现的

前面说到有了Service后，无论Pod如何变化，Service都能够发现到Pod。

如果调用kubectl describe命令查看Service的信息，您会看下如下信息。

```
$ kubectl describe svc nginx
Name:              nginx
......
```

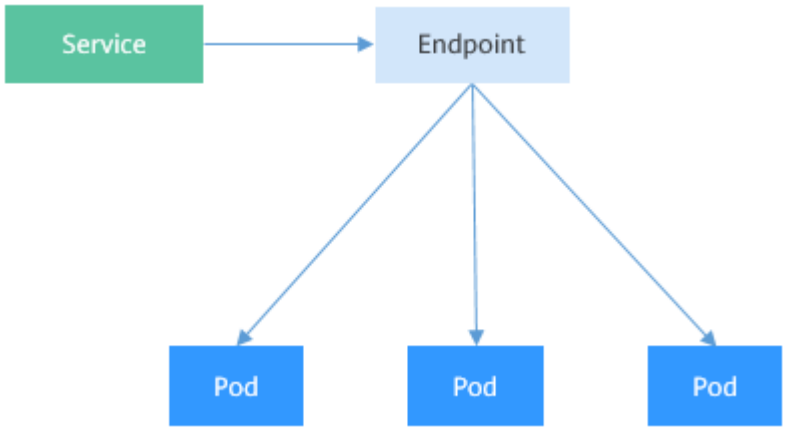
```
Endpoints:         172.16.2.132:80,172.16.3.6:80,172.16.3.7:80
......
```

可以看到一个Endpoints，Endpoints同样也是Kubernetes的一种资源对象，可以查询得到。Kubernetes正是通过Endpoints监控到Pod的IP，从而让Service能够发现Pod。

```
$ kubectl get endpoints
NAME         ENDPOINTS                                      AGE
kubernetes   192.168.0.127:5444                             7h19m
nginx        172.16.2.132:80,172.16.3.6:80,172.16.3.7:80    5h48m
```

这里的172.16.2.132:80是Pod的IP:Port，通过如下命令可以查看到Pod的IP，与上面的IP一致。

```
$ kubectl get po -o wide
NAME                    READY   STATUS    RESTARTS   AGE     IP            NODE
nginx-869759589d-dnknn   1/1     Running   0          5h40m   172.16.3.7    192.168.0.212
nginx-869759589d-fcxhh   1/1     Running   0          5h40m   172.16.3.6    192.168.0.212
nginx-869759589d-r69kh   1/1     Running   0          5h40m   172.16.2.132  192.168.0.94
```



如果删除一个Pod，Deployment会将Pod重建，新的Pod IP会发生变化。

```
$ kubectl delete po nginx-869759589d-dnknn
pod "nginx-869759589d-dnknn" deleted

$ kubectl get po -o wide
NAME                    READY   STATUS    RESTARTS   AGE     IP            NODE
nginx-869759589d-fcxhh   1/1     Running   0          5h41m   172.16.3.6    192.168.0.212
nginx-869759589d-r69kh   1/1     Running   0          5h41m   172.16.2.132  192.168.0.94
nginx-869759589d-w98wg   1/1     Running   0          7s      172.16.3.10   192.168.0.212
```

再次查看Endpoints，会发现Endpoints的内容随着Pod发生了变化。

```
$ kubectl get endpoints
NAME         ENDPOINTS                                      AGE
kubernetes   192.168.0.127:5444                             7h20m
nginx        172.16.2.132:80,172.16.3.10:80,172.16.3.6:80   5h49m
```

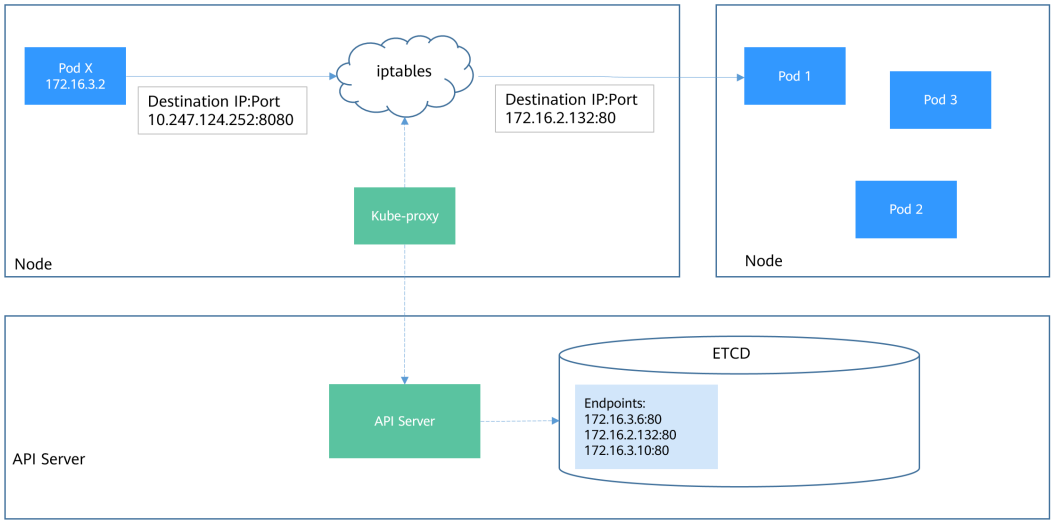
下面进一步了解这又是如何实现的。

在图3-4中介绍过Node节点上的kube-proxy，实际上Service相关的事情都由节点上的kube-proxy处理。在Service创建时Kubernetes会分配IP给Service，同时通过API Server通知所有kube-proxy有新Service创建了，kube-proxy收到通知后通过iptables记录Service和IP/端口对的关系，从而让Service在节点上可以被查询到。

下图是一个实际访问Service的图示，Pod X访问Service（10.247.124.252:8080），在往外发数据包时，在节点上根据iptables规则目的IP:Port被随机替换为Pod1的IP:Port，从而通过Service访问到实际的Pod。

除了记录Service和IP/端口对的关系，kube-proxy还会监控Service和Endpoint的变化，从而保证Pod重建后仍然能通过Service访问到Pod。

图 3-20 Pod X 访问 Service 的过程



Service 的类型与使用场景

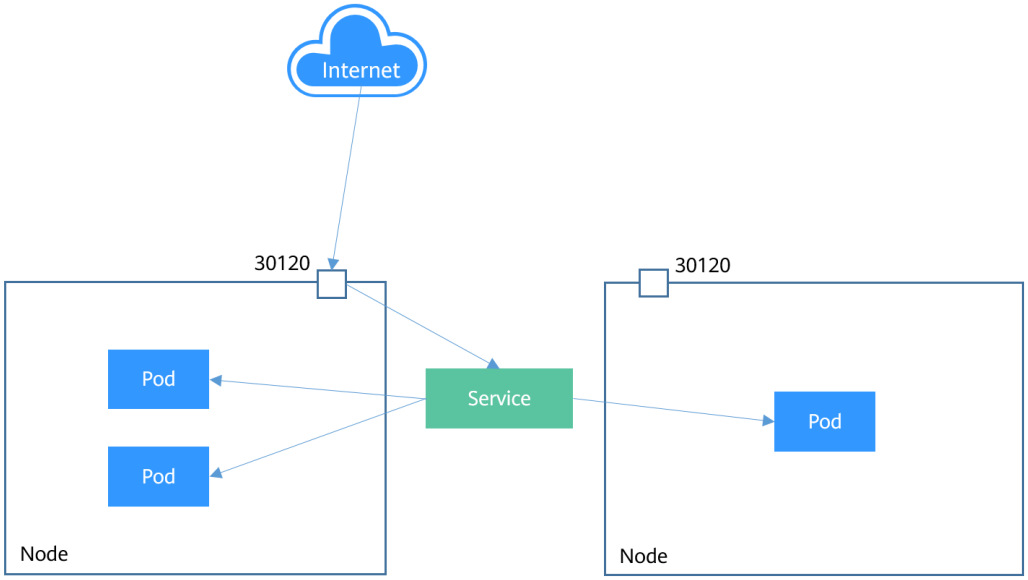
Service的类型除了ClusterIP还有NodePort、LoadBalancer和None，这几种类型的Service有着不同的用途。

- ClusterIP：用于在集群内部互相访问的场景，通过ClusterIP访问Service。
- NodePort：用于从集群外部访问的场景，通过节点上的端口访问Service，详细介绍请参见**NodePort类型的Service**。
- LoadBalancer：用于从集群外部访问的场景，其实是NodePort的扩展，通过一个特定的LoadBalancer访问Service，这个LoadBalancer将请求转发到节点的NodePort，而外部只需要访问LoadBalancer，详细介绍请参见**LoadBalancer类型的Service**。
- None：用于Pod间的互相发现，这种类型的Service又叫Headless Service，详细介绍请参见**Headless Service**。

NodePort 类型的 Service

NodePort类型的Service可以让Kubemetes集群每个节点上保留一个相同的端口， 外部访问连接首先访问节点IP:Port，然后将这些连接转发给服务对应的Pod。如下图所示。

图 3-21 NodePort Service



下面是一个创建NodePort类型的Service。创建完成后，可以通过节点的IP:Port访问到后台Pod。

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    nodePort: 30120
  selector:
    app: nginx
```

创建并查看，可以看到PORT这一列为8080:30120/TCP，说明Service的8080端口是映射到节点的30120端口。

```
$ kubectl create -f nodeport.yaml
service/nodeport-service created

$ kubectl get svc -o wide
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE    SELECTOR
kubernetes        ClusterIP   10.247.0.1       <none>        443/TCP          107m   <none>
nginx             ClusterIP   10.247.124.252   <none>        8080/TCP         16m    app=nginx
nodeport-service  NodePort    10.247.210.174   <none>        8080:30120/TCP   17s    app=nginx
```

此时，通过节点IP:端口访问Service可以访问到Pod，如下所示。

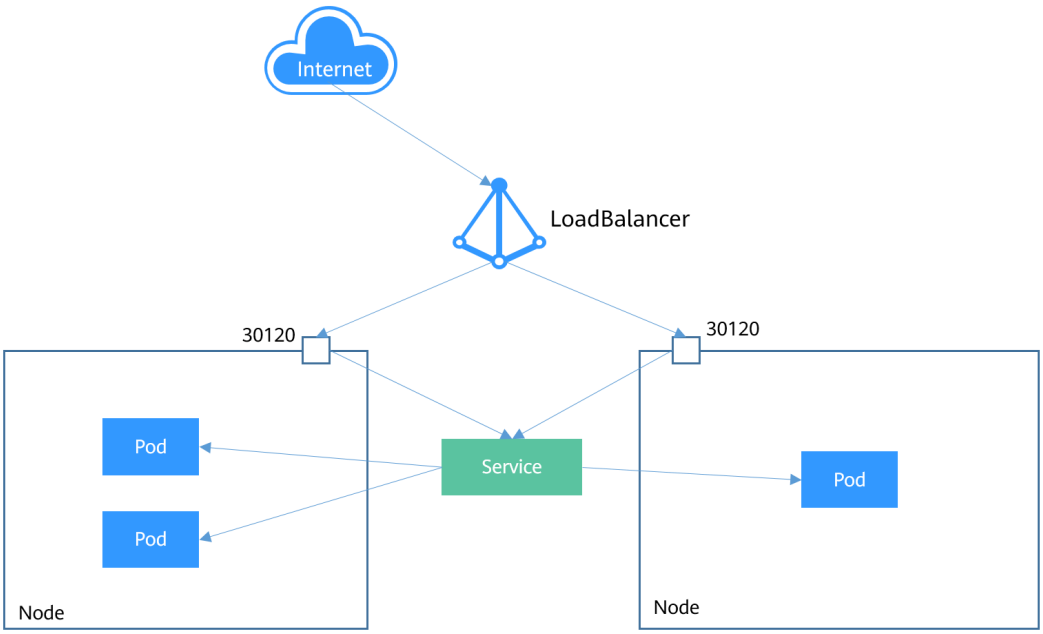
```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 192.168.0.212:30120
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
......
```

LoadBalancer 类型的 Service

LoadBalancer类型的Service其实是NodePort类型Service的扩展，通过一个特定的LoadBalancer访问Service，这个LoadBalancer将请求转发到节点的NodePort。

LoadBalancer本身不是属于Kubernetes的组件，这部分通常是由具体厂商（云服务提供商）提供，不同厂商的Kubernetes集群与LoadBalancer的对接实现各不相同，例如CCE对接了ELB。这就导致了创建LoadBalancer类型的Service有不同的实现。

图 3-22 LoadBalancer Service



下面是一个创建LoadBalancer类型的Service。创建完成后，可以通过ELB的IP:Port访问到后台Pod。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: 3c7caa5a-a641-4bff-801a-feace27424b6
  labels:
    app: nginx
  name: nginx
spec:
  loadBalancerIP: 10.78.42.242     #  ELB实例的IP地址
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30120
  selector:
    app: nginx
  type: LoadBalancer    # 类型为LoadBalancer
```

上面metadata.annotations里的参数配置是CCE的LoadBalancer类型Service需要配置的参数，表示这个Service绑定哪个ELB实例。CCE还支持创建LoadBalancer类型Service时新建ELB实例，详细的内容请参见**负载均衡（LoadBalancer）**。

Headless Service

前面讲的Service解决了Pod的内外部访问问题，但还有下面这些问题没解决。

- 同时访问所有Pod
- 一个Service内部的Pod互相访问

Headless Service正是解决这个问题的，Headless Service不会创建ClusterIP，并且查询会返回所有Pod的DNS记录，这样就可查询到所有Pod的IP地址。**StatefulSet**中StatefulSet正是使用Headless Service解决Pod间互相访问的问题。

```
apiVersion: v1
kind: Service       # 对象类型为Service
metadata:
  name: nginx-headless
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Pod间通信的端口名称
      port: 80        # Pod间通信的端口号
  selector:
    app: nginx        # 选择标签为app:nginx的Pod
  clusterIP: None     # 必须设置为None，表示Headless Service
```

执行如下命令创建Headless Service。

```
# kubectl create -f headless.yaml
service/nginx-headless created
```

创建完成后可以查询Service。

```
# kubectl get svc
NAME             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
nginx-headless   ClusterIP   None         <none>        80/TCP     5s
```

创建一个Pod来查询DNS，可以看到能返回所有Pod的记录，这就解决了访问所有Pod的问题了。

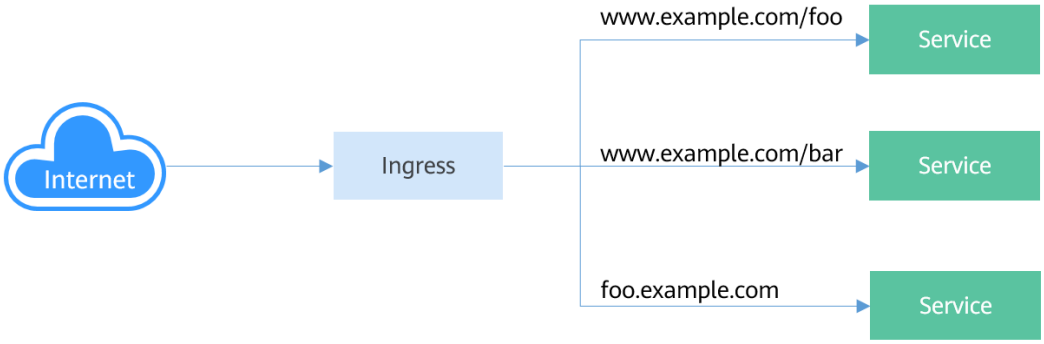
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # nslookup nginx-1.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-1.nginx.default.svc.cluster.local
Address: 172.16.0.18

/ # nslookup nginx-2.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-2.nginx.default.svc.cluster.local
Address: 172.16.0.19
```

3.6.3 Ingress

为什么需要 Ingress

Service是基于四层TCP和UDP协议转发的，而Ingress可以基于七层的HTTP和HTTPS协议转发，可以通过域名和路径做到更细粒度的划分，如下图所示。

图 3-23 Ingress-Service

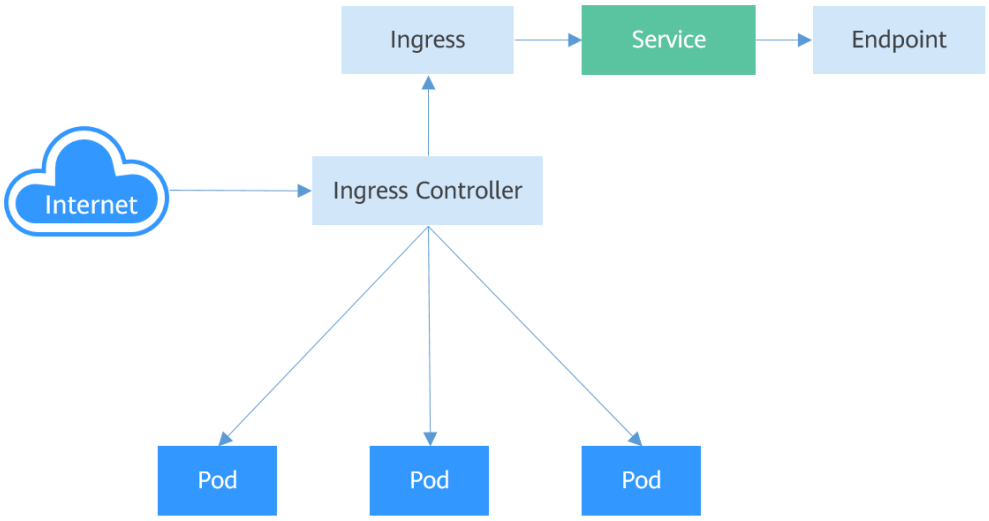


Ingress 工作机制

要想使用Ingress功能，必须在Kubernetes集群上安装Ingress Controller。Ingress Controller有很多种实现，最常见的就是Kubernetes官方维护的**NGINX Ingress Controller**；不同厂商通常有自己的实现，例如CCE使用弹性负载均衡服务ELB实现Ingress的七层负载均衡。

外部请求首先到达Ingress Controller，Ingress Controller根据Ingress的路由规则，查找到对应的Service，进而通过Endpoint查询到Pod的IP地址，然后将请求转发给Pod。

图 3-24 Ingress 工作机制



创建 Ingress

下面例子中，使用http协议，关联的后端Service为“nginx:8080”，使用ELB作为Ingress控制器（metadata.annotations字段都是指定使用哪个ELB实例），当访问“http://192.168.10.155:8080/test”时，流量转发“nginx:8080”对应的Service，从而将流量转发到对应Pod。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.port: '8080'
```

```
    kubernetes.io/elb.ip: 192.168.10.155
    kubernetes.io/elb.id: aa7cf5ec-7218-4c43-98d4-c36c0744667a
spec:
  rules:
  - host: ''
    http:
      paths:
      - backend:
          serviceName: nginx
          servicePort: 8080
        path: "/test"
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

Ingress中还可以设置外部域名，这样您就可以通过域名来访问到ELB，进而访问到后端服务。

说明

域名访问依赖于域名解析，需要您将域名解析指向ELB实例的IP地址，例如您可以使用云解析服务 DNS来实现域名解析。

```
spec:
  rules:
  - host: www.example.com       # 域名
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx
          servicePort: 80
```

路由到多个服务

Ingress可以同时路由到多个服务，配置如下所示。

- 当访问“http://foo.bar.com/foo”时，访问的是“s1:80”后端。
- 当访问“http://foo.bar.com/bar”时，访问的是“s2:80”后端。

```
spec:
  rules:
  - host: foo.bar.com          # host地址
    http:
      paths:
      - path: "/foo"
        backend:
          serviceName: s1
          servicePort: 80
      - path: "/bar"
        backend:
          serviceName: s2
          servicePort: 80
```

3.6.4 就绪探针（Readiness Probe）

一个新Pod创建后，Service就能立即选择到它，并会把请求转发给Pod，那问题就来了，通常一个Pod启动是需要时间的，如果Pod还没准备好（可能需要时间来加载配置或数据，或者可能需要执行一个预热程序之类），这时把请求转给Pod的话，Pod也无法处理，造成请求失败。

Kubernetes解决这个问题的方法就是给Pod加一个业务就绪探针Readiness Probe，当检测到Pod就绪后才允许Service将请求转给Pod。

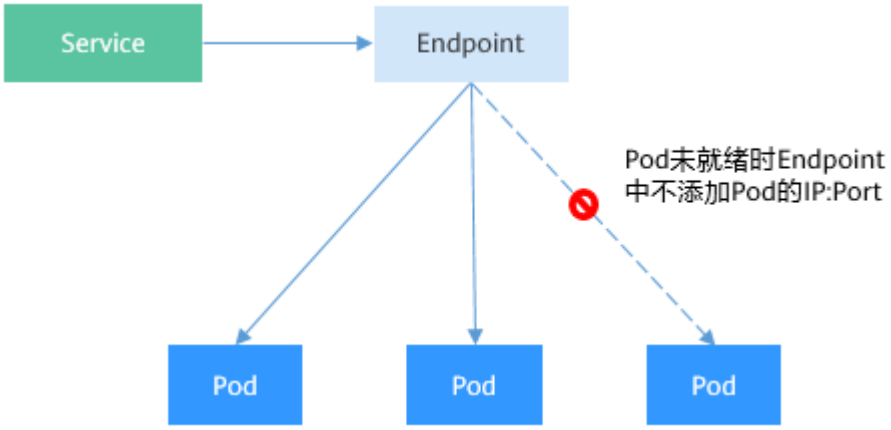
Readiness Probe同样是周期性的检测Pod，然后根据响应来判断Pod是否就绪，与**存活探针（Liveness Probe）**相同，就绪探针也支持如下三种类型。

- Exec：Probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明已经就绪。
- HTTP GET：往容器的IP:Port发送HTTP GET请求，如果Probe收到2xx或3xx，说明已经就绪。
- TCP Socket：尝试与容器建立TCP连接，如果能建立连接说明已经就绪。

Readiness Probe 的工作原理

通过Endpoints就可以实现Readiness Probe的效果，当Pod还未就绪时，将Pod的IP:Port从Endpoints中删除，Pod就绪后再加入到Endpoints中，如下图所示。

图 3-25 Readiness Probe 的实现原理



Exec

Exec方式与HTTP GET方式一致，如下所示，这个探针执行**ls /ready**命令，如果这个文件存在，则返回0，说明Pod就绪了，否则返回其他状态码。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:      # Readiness Probe
          exec:              # 定义 ls /ready 命令
            command:
```

```
        - ls
        - /ready
    imagePullSecrets:
    - name: default-secret
```

将上面Deployment的定义保存到deploy-read.yaml文件中，删除之前创建的Deployment，用deploy-read.yaml创建这个Deployment。

```
# kubectl delete deploy nginx
deployment.apps "nginx" deleted

# kubectl create -f deploy-read.yaml
deployment.apps/nginx created
```

这里由于nginx镜像不包含/ready这个文件，所以在创建完成后容器不在Ready状态，如下所示，注意READY这一列的值为0/1，表示容器没有Ready。

```
# kubectl get po
NAME                    READY     STATUS    RESTARTS   AGE
nginx-7955fd7786-686hp   0/1       Running   0          7s
nginx-7955fd7786-9tgwq   0/1       Running   0          7s
nginx-7955fd7786-bqsbj   0/1       Running   0          7s
```

创建Service。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    targetPort: 80
    port: 8080
    protocol: TCP
  type: ClusterIP
```

查看Service，发现Endpoints一行的值为空，表示没有Endpoints。

```
$ kubectl describe svc nginx
Name:              nginx
......
Endpoints:
......
```

如果此时给容器中创建一个/ready的文件，让Readiness Probe成功，则容器会处于Ready状态。再查看Pod和Endpoints，发现创建了/ready文件的容器已经Ready，Endpoints也已经添加。

```
# kubectl exec nginx-7955fd7786-686hp -- touch /ready

# kubectl get po -o wide
NAME                    READY     STATUS    RESTARTS   AGE       IP
nginx-7955fd7786-686hp   1/1       Running   0          10m       192.168.93.169
nginx-7955fd7786-9tgwq   0/1       Running   0          10m       192.168.166.130
nginx-7955fd7786-bqsbj   0/1       Running   0          10m       192.168.252.160

# kubectl get endpoints
NAME       ENDPOINTS           AGE
nginx      192.168.93.169:80   14d
```

HTTP GET

Readiness Probe的配置与**存活探针（livness probe）**一样，都是在Pod Template的containers里面，如下所示，这个Readiness Probe向Pod发送HTTP请求，当Probe收到2xx或3xx返回时，说明Pod已经就绪。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:           # readinessProbe
          httpGet:                # HTTP GET定义
            path: /read
            port: 80
      imagePullSecrets:
      - name: default-secret
```

TCP Socket

同样，TCP Socket类型的探针如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:           # readinessProbe
          tcpSocket:              # TCP Socket定义
            port: 80
      imagePullSecrets:
      - name: default-secret
```

Readiness Probe 高级配置

与Liveness Probe相同，Readiness Probe也有同样的高级配置选项，上面nginx Pod的describe命令回显有中有如下行。

```
Readiness: exec [ls /var/ready] delay=0s timeout=1s period=10s #success=1 #failure=3
```

这一行表示Readiness Probe的具体参数配置，其含义如下：

- delay=0s 表示容器启动后立即开始探测，没有延迟时间
- timeout=1s 表示容器必须在1s内做出相应反馈给probe，否则视为探测失败
- period=10s 表示每10s探测一次
- #success=1 探测连续1次成功表示成功
- #failure=3 探测连续3次失败表示失败

这些是创建时默认设置的，您也可以手动配置，如下所示。

```
    readinessProbe:      # Readiness Probe
      exec:              # 定义 ls /readiness/ready 命令
        command:
        - ls
        - /readiness/ready
      initialDelaySeconds: 10    # 容器启动后多久开始探测
      timeoutSeconds: 2          # 表示容器必须在2s内做出相应反馈给probe，否则视为探测失败
      periodSeconds: 30          # 探测周期，每30s探测一次
      successThreshold: 1        # 连续探测1次成功表示成功
      failureThreshold: 3        # 连续探测3次失败表示失败
```

3.6.5 NetworkPolicy

NetworkPolicy是Kubernetes设计用来限制Pod访问的对象，通过设置NetworkPolicy策略，可以允许Pod被哪些地址访问（即入规则）、或Pod访问哪些地址（即出规则）。这相当于从应用的层面构建了一道防火墙，进一步保证了网络安全。

NetworkPolicy支持的能力取决于集群的网络插件的能力，如CCE的集群只支持设置Pod的入规则。

默认情况下，如果命名空间中不存在任何策略，则所有进出该命名空间中的Pod的流量都被允许。

NetworkPolicy的规则可以选择如下3种：

- namespaceSelector：根据命名空间的标签选择，具有该标签的命名空间都可以访问。
- podSelector：根据Pod的标签选择，具有该标签的Pod都可以访问。
- ipBlock：根据网络选择，网段内的IP地址都可以访问。（CCE当前不支持此种方式）

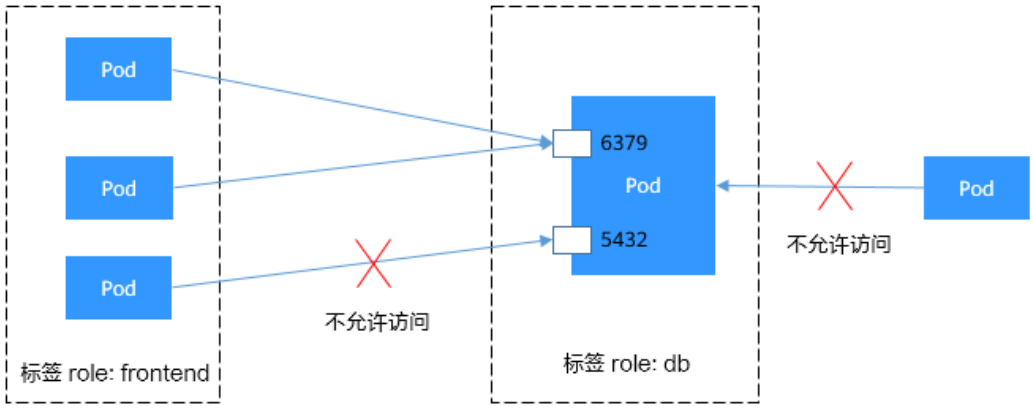
使用 podSelector 设置访问范围

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:                    # 表示入规则
```

```
- from:
  - podSelector:            # 只允许具有role=frontend标签的Pod访问
      matchLabels:
        role: frontend
  ports:                    # 只能使用TCP协议访问6379端口
  - protocol: TCP
    port: 6379
```

示意图如下所示。

图 3-26 podSelector

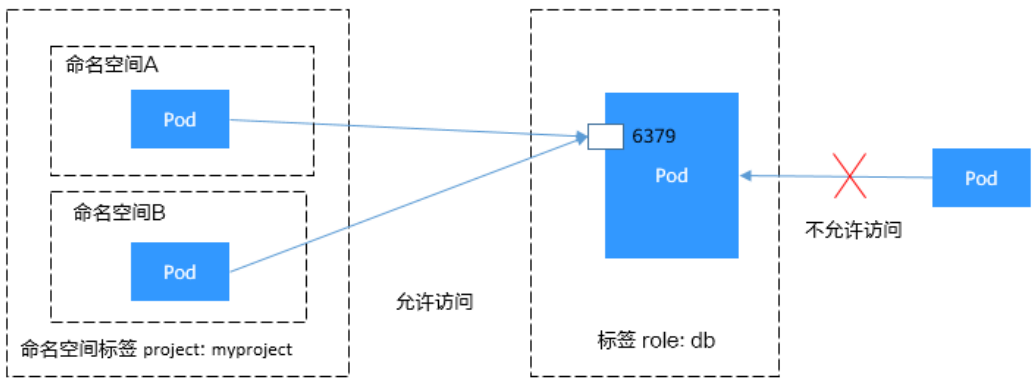


使用 namespaceSelector 设置访问范围

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:                  # 表示入规则
  - from:
    - namespaceSelector:        # 只允许具有project=myproject标签的命名空间中的Pod访问
        matchLabels:
          project: myproject
    ports:                  # 只能使用TCP协议访问6379端口
    - protocol: TCP
      port: 6379
```

示意图如下所示。

图 3-27 namespaceSelector



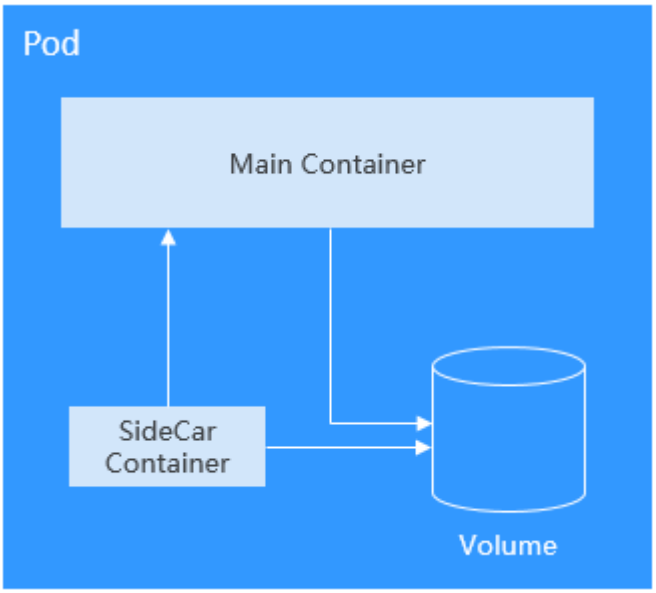
3.7 持久化存储

3.7.1 Volume

容器中的文件在磁盘上是临时存放的，当容器重建时，容器中的文件将会丢失，另外当在一个Pod中同时运行多个容器时，常常需要在这些容器之间共享文件，这也是容器不好解决的问题。 Kubernetes抽象出了Volume来解决这两个问题，也就是存储卷，Kubernetes的Volume是Pod的一部分，Volume不是单独的对象，不能独立创建，只能在Pod中定义。

Pod中的所有容器都可以访问Volume，但必须要挂载，且可以挂载到容器中任何目录。

实际中使用容器存储如下图所示，将容器的内容挂载到Volume中，通过Volume两个容器间实现了存储共享。



Volume的生命周期与挂载它的Pod相同，但是Volume里面的文件可能在Volume消失后仍然存在，这取决于Volume的类型。

Volume 的类型

Kubernetes的Volume有非常多的类型，在实际使用中使用最多的类型如下。

- emptyDir：一种简单的空目录，主要用于临时存储。
- hostPath：将主机某个目录挂载到容器中。
- ConfigMap、Secret：特殊类型，将Kubernetes特定的对象类型挂载到Pod，在**ConfigMap**和**Secret**章节介绍过如何将ConfigMap和Secret挂载到Volume中。
- persistentVolumeClaim：Kubernetes的持久化存储类型，详细介绍请参考**PV、PVC和StorageClass**中会详细介绍。

EmptyDir

EmptyDir是最简单的一种Volume类型，根据名字就能看出，这个Volume挂载后就是一个空目录，应用程序可以在里面读写文件，emptyDir Volume的生命周期与Pod相同，Pod删除后Volume的数据也同时删除掉。

emptyDir的一些用途：

- 缓存空间，例如基于磁盘的归并排序。
- 为耗时较长的计算任务提供检查点，以便任务能从崩溃前状态恢复执行。

emptyDir配置示例如下。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

emptyDir实际是将Volume的内容写在Pod所在节点的磁盘上，另外emptyDir也可以设置存储介质为内存，如下所示，medium设置为Memory。

```
volumes:
  - name: html
    emptyDir:
      medium: Memory
```

HostPath

HostPath是一种持久化存储，emptyDir里面的内容会随着Pod的删除而消失，但HostPath不会，如果对应的Pod删除，HostPath Volume里面的内容依然存在于节点的目录中，如果后续重新创建Pod并调度到同一个节点，挂载后依然可以读取到之前Pod写的内容。

HostPath存储的内容与节点相关，所以它不适合像数据库这类的应用，想象下如果数据库的Pod被调度到别的节点了，那读取的内容就完全不一样了。

记住永远不要使用HostPath存储跨Pod的数据，一定要把HostPath的使用范围限制在读取节点文件上，这是因为Pod被重建后不确定会调度哪个节点上，写文件可能会导致前后不一致。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: nginx:alpine
    name: hostpath-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
```

```
  hostPath:
    path: /data
```

3.7.2 PV、PVC 和 StorageClass

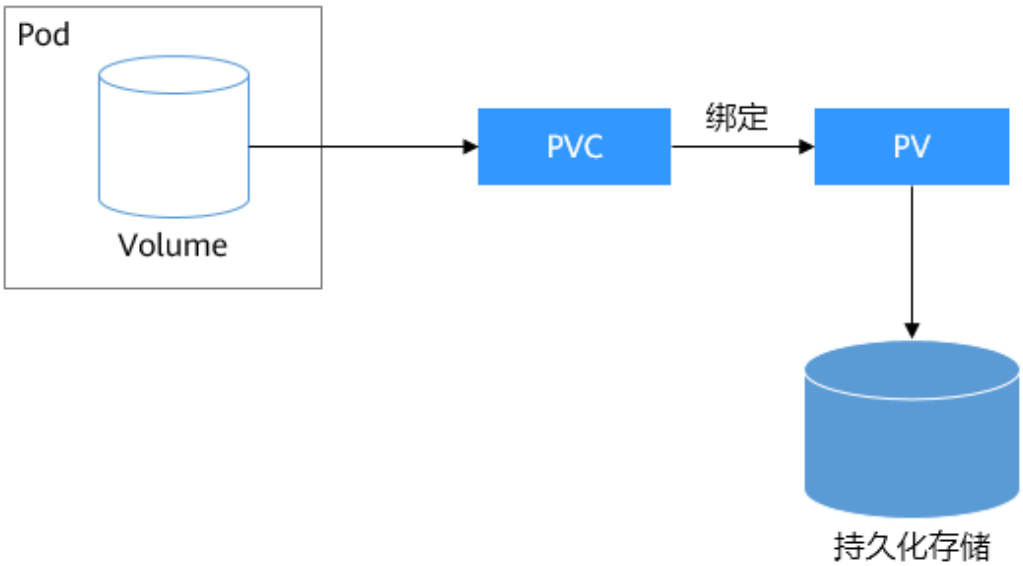
上一章节介绍的HostPath是一种持久化存储，但是HostPath的内容是存储在节点上，导致只适合读取。

如果要求Pod重新调度后仍然能使用之前读写过的数据，就只能使用网络存储了，网络存储种类非常多且有不同的使用方法，通常一个云服务提供商至少有块存储、文件存储、对象存储三种，如EVS、SFS和OBS。Kubernetes解决这个问题的方式是抽象了PV（PersistentVolume）和PVC（PersistentVolumeClaim）来解耦这个问题，从而让使用者不用关心具体的基础设施，当需要存储资源的时候，只要像CPU和内存一样，声明要多少即可。

- PV：PV描述的是持久化存储卷，主要定义的是一个持久化存储在宿主机上的目录，比如一个NFS的挂载目录。
- PVC：PVC描述的是Pod所希望使用的持久化存储的属性，比如，Volume存储的大小、可读写权限等等。

Kubernetes管理员设置好网络存储的类型，提供对应的PV描述符配置到Kubernetes，使用者需要存储的时候只需要创建PVC，然后在Pod中使用Volume关联PVC，即可让Pod使用到存储资源，它们之间的关系如下图所示。

图 3-28 PVC 绑定 PV



CSI

Kubernetes提供了CSI接口（Container Storage Interface，容器存储接口），基于CSI这套接口，可以开发定制出CSI插件，从而支持特定的存储，达到解耦的目的。例如在**Namespace：资源分组**中看到的kube-system命名空间下everest-csi-controller和everest-csi-driver就是CCE开发存储控制器和驱动。有了这些驱动就可以使用EVS、SFS、OBS存储。

```
$ kubectl get po --namespace=kube-system
NAME                                  READY   STATUS    RESTARTS   AGE
everest-csi-controller-6d796fb9c5-v22df   2/2     Running   0          9m11s
```

```
everest-csi-driver-snzrr                1/1     Running   0          12m
everest-csi-driver-ttj28                1/1     Running   0          12m
everest-csi-driver-wtrk6                1/1     Running   0          12m
```

PV

来看PV是如何描述持久化存储，例如在SFS中创建了一个文件存储，这个文件存储ID为68e4a4fd-d759-444b-8265-20dc66c8c502，挂载地址为sfs-nas01.eu-west-0a.prod-cloud-ocb.orange-business.com:/share-96314776。如果想在CCE中使用这个文件存储，则需要先创建一个PV来描述这个存储，如下所示。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  accessModes:
  - ReadWriteMany                    # 读写模式
  capacity:
    storage: 10Gi                    # 定义PV的大小
  csi:
    driver: nas.csi.everest.io       # 声明使用的驱动
    fsType: nfs                      # 存储类型
    volumeAttributes:
      everest.io/share-export-location: sfs-nas01.eu-west-0a.prod-cloud-ocb.orange-business.com:/
share-96314776   # 挂载地址
    volumeHandle: 68e4a4fd-d759-444b-8265-20dc66c8c502                                  # 存储ID
```

这里csi下面的内容就是CCE中特定的字段，在其他地方无法使用。

下面创建这个PV并查看。

```
$ kubectl create -f pv.yaml
persistentvolume/pv-example created

$ kubectl get pv
NAME              CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM    STORAGECLASS
REASON   AGE
pv-example        10Gi       RWX            Retain           Available                            4s
```

RECLAIM POLICY是指PV的回收策略，Retain表示PVC被释放后PV继续保留。STATUS值为Available，表示PV处于可用的状态。

PVC

PVC可以绑定一个PV，示例如下。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi              # 声明存储的大小
  volumeName: pv-example         # PV的名称
```

创建PVC并查看。

```
$ kubectl create -f pvc.yaml
persistentvolumeclaim/pvc-example created

$ kubectl get pvc
NAME          STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-example   Bound    pv-example   10Gi       RWX                           9s
```

这里可以看到状态是Bound，VOLUME是pv-example，表示PVC已经绑定了PV。

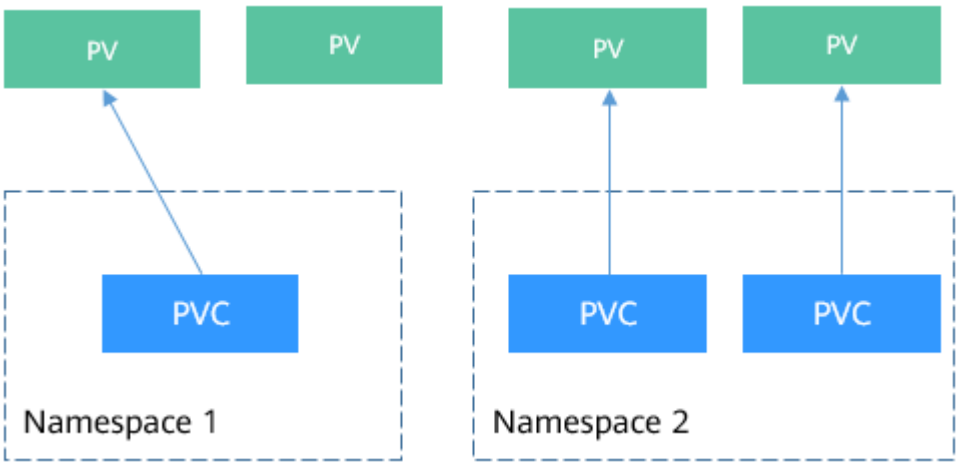
再来看下PV。

```
$ kubectl get pv
NAME         CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS
REASON   AGE
pv-example   10Gi       RWX            Retain           Bound    default/pvc-example                  50s
```

可以看到状态也变成了Bound，CLAIM是default/pvc-example，表示这个PV绑定了default命名空间下的pvc-example这个PVC。

这里一个比较有意思的地方是CLAIM是default/pvc-example，为什么要显示default呢，这是因为PV是集群级别的资源，并不属于某个命名空间，而PVC是命名空间级别的资源，PV可以与任何命名空间的PVC资源绑定。

图 3-29 PV 与 PVC



StorageClass

上节说的PV和PVC方法虽然能实现屏蔽底层存储，但是PV创建比较复杂（可以看到PV中csi字段的配置很麻烦），通常都是由集群管理员管理，这非常不方便。

Kubernetes解决这个问题的方法是提供动态配置PV的方法，可以自动创PV。管理员可以部署PV配置器（provisioner），然后定义对应的StorageClass，这样开发者在创建PVC的时候就可以选择需要创建存储的类型，PVC会把StorageClass传递给PV provisioner，由provisioner自动创建PV。如CCE就提供csi-disk、csi-nas、csi-obs等StorageClass，在声明PVC时加上StorageClassName，就可以自动创建PV，并自动创建底层的存储资源。

说明

下面是以CCE 1.15及以上版本集群使用方法举例，1.13以及之前版本集群上使用方法有差异。

执行如下命令即可查询CCE提供的默认StorageClass。您可以使用CCE提供的CSI插件自定义创建StorageClass，但从功能角度与CCE提供的默认StorageClass并无区别，这里不做过多描述。

```
# kubectl get sc
NAME                PROVISIONER                  AGE
csi-disk            everest-csi-provisioner      17d         # 云硬盘 StorageClass
csi-disk-topology   everest-csi-provisioner      17d         # 延迟绑定的云硬盘 StorageClass
```

```
csi-nas          everest-csi-provisioner       17d       # 文件存储 StorageClass
csi-obs          everest-csi-provisioner       17d       # 对象存储 StorageClass
csi-sfsturbo     everest-csi-provisioner       17d       # 极速文件存储 StorageClass
```

使用StorageClassName创建PVC。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name:  pvc-sfs-auto-example
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas       # StorageClass
```

创建PVC并查看PVC和PV。

```
$ kubectl create -f pvc2.yaml
persistentvolumeclaim/pvc-sfs-auto-example created

$ kubectl get pvc
NAME                  STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
pvc-sfs-auto-example   Bound    pvc-1f1c1812-f85f-41a6-a3b4-785d21063ff3   10Gi       RWX           csi-
nas        29s

$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                       STORAGECLASS   REASON   AGE
pvc-1f1c1812-f85f-41a6-a3b4-785d21063ff3   10Gi       RWO            Delete           Bound    default/pvc-sfs-
auto-example   csi-nas                 20s
```

这可以看到使用StorageClass后，不仅创建了PVC，而且创建了PV，并且将二者绑定了。

定义了StorageClass后，就可以减少创建并维护PV的工作，PV变成了自动创建，作为使用者，只需要在声明PVC时指定StorageClassName即可，这就大大减少工作量。

再次说明，StorageClassName的类型在不同厂商的产品上各不相同，这里只是使用了文件存储作为示例，其余存储类型请参见**存储概述**。

在 Pod 中使用 PVC

有了PVC后，在Pod中使用持久化存储就非常方便了，在Pod Template中的Volume直接关联PVC的名称，然后挂载到容器之中即可，如下所示。甚至在StatefulSet中还可以直接声明PVC，详情请参见**StatefulSet**。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
```

```
    volumeMounts:
    - mountPath: /tmp                    # 挂载路径
      name: pvc-sfs-example
  restartPolicy: Always
  volumes:
  - name: pvc-sfs-example
    persistentVolumeClaim:
      claimName:  pvc-example              # PVC的名称
```

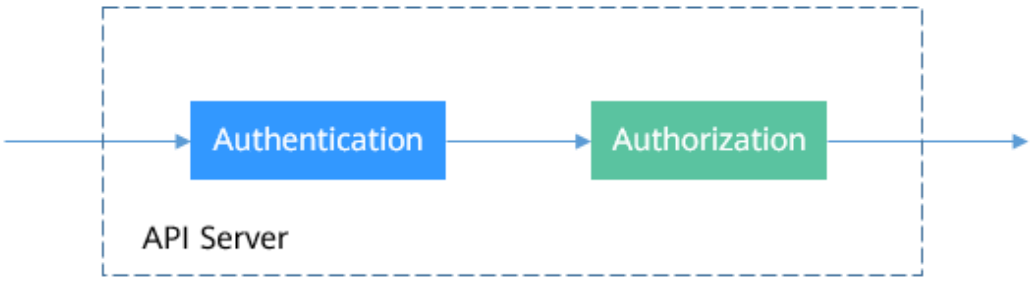
3.8 认证与授权

3.8.1 ServiceAccount

Kubernetes中所有的访问，无论外部内部，都会通过API Server处理，访问Kubernetes资源前需要经过认证与授权。

- Authentication：用于识别用户身份的认证，Kubernetes分外部服务账号和内部服务账号，采取不同的认证机制，具体请参见**认证与ServiceAccount**。
- Authorization：用于控制用户对资源访问的授权，对访问的授权目前主要使用RBAC机制，将在**RBAC**介绍。

图 3-30 API Server 的认证授权



认证与 ServiceAccount

Kubernetes的用户分为服务账户（ServiceAccount）和普通账户两种类型。

- 服务账户与Namespace绑定，关联一套凭证，存储在Secret中，Pod创建时挂载Secret，从而允许与API Server之间调用。
- Kubernetes中没有代表普通账户的对象，这类账户默认由外部服务独立管理，比如CCE的用户是由IAM管理的。

普通账号并不是这里要讨论的内容，这里主要关注ServiceAccount。

ServiceAccount同样是Kubernetes中的资源，与Pod、ConfigMap类似，且作用于独立的命名空间，也就是ServiceAccount是属于命名空间级别的，创建命名空间时会自动创建一个名为default的ServiceAccount。

使用下面命令可以查看ServiceAccount。

```
$ kubectl get sa
NAME     SECRETS   AGE
default  1         30d
```

同时Kubernetes还会为ServiceAccount自动创建一个Secret，使用下面命令可以查看到。

```
$ kubectl describe sa default
Name:                default
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   default-token-vssmw
Tokens:              default-token-vssmw
Events:              <none>
```

在Pod的定义文件中，可以用指定账户名称的方式将一个ServiceAccount赋值给一个Pod，如果不指定就会使用默认的ServiceAccount。当API Server接收到一个带有认证Token的请求时，API Server会用这个Token来验证发送请求的客户端所关联的ServiceAccount是否允许执行请求的操作。

创建 ServiceAccount

使用如下命令就可以创建ServiceAccount：

```
$ kubectl create serviceaccount sa-example
serviceaccount/sa-example created

$ kubectl get sa
NAME           SECRETS   AGE
default        1         30d
sa-example     1         2s
```

可以看到已经创建了与ServiceAccount相关联的Token。

```
$ kubectl describe sa sa-example
Name:                sa-example
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   sa-example-token-c7bqx
Tokens:              sa-example-token-c7bqx
Events:              <none>
```

查看Secret的内容，可以发现ca.crt、namespace和token三个数据。

```
$ kubectl describe secret sa-example-token-c7bqx
Name:         sa-example-token-c7bqx
...
Data
====
ca.crt:     1082 bytes
namespace:  7 bytes
token:      <Token的内容>
```

在 Pod 中使用 ServiceAccount

Pod中使用ServiceAccount非常方便，只需要指定ServiceAccount的名称即可。

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-example
spec:
  serviceAccountName: sa-example
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
```

```
    requests:
      cpu: 100m
      memory: 200Mi
imagePullSecrets:
- name: default-secret
```

创建并查看这个Pod，可以看到Pod挂载了sa-example-token-c7bqx，也就是sa-example这个ServiceAccount对应的Token，即Pod使用这个Token来做认证。

```
$ kubectl create -f sa-pod.yaml
pod/sa-example created

$ kubectl get pod
NAME                                READY   STATUS             RESTARTS   AGE
sa-example                           0/1     running            0          5s

$ kubectl describe pod sa-example
...
Containers:
  sa-example:
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from sa-example-token-c7bqx (ro)
```

进入Pod内部，还可以看到对应的文件，如下所示。

```
$ kubectl exec -it sa-example -- /bin/sh
/ # cd /run/secrets/kubernetes.io/serviceaccount
/run/secrets/kubernetes.io/serviceaccount # ls
ca.crt     namespace  token
```

如上，在容器应用中，就可以使用ca.crt和Token来访问API Server。

下面来验证认证是否能生效。在Kubernetes集群中，默认为API Server创建了一个名为kubernetes的Service，通过这个Service可以访问API Server。

```
$ kubectl get svc
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
kubernetes    ClusterIP   10.247.0.1       <none>        443/TCP         34
```

进入Pod，使用curl命令直接访问会得到如下返回信息，表示并没有权限。

```
$ kubectl exec -it sa-example -- /bin/sh
/ # curl https://kubernetes
curl: (60) SSL certificate problem: unable to get local issuer certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
```

使用ca.crt和Token做认证，先将ca.crt放到CURL_CA_BUNDLE这个环境变量中，curl命令使用CURL_CA_BUNDLE指定证书；再将Token的内容放到TOKEN中，然后带上TOKEN访问API Server。

```
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:serviceaccount:default:sa-example\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

  },
```

```
  "code": 403
}
```

可以看到，已经能够通过认证了，但是API Server返回的是cannot get path \"/\""，表示没有权限访问，这说明还需要得到授权后才能访问，授权机制将在RBAC中介绍。

3.8.2 RBAC

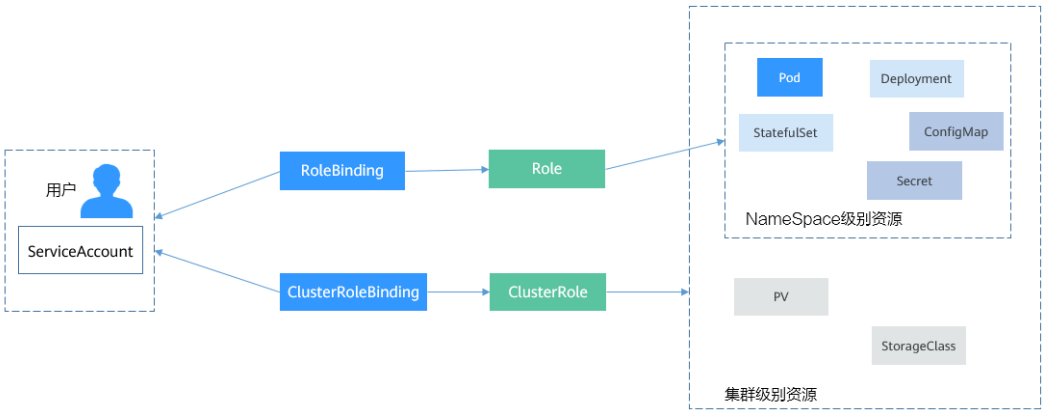
RBAC 资源

Kubernetes中完成授权工作的就是RBAC机制，RBAC授权规则是通过四种资源来进行配置。

- Role：角色，其实是定义一组对Kubernetes资源（命名空间级别）的访问规则。
- RoleBinding：角色绑定，定义了用户和角色的关系。
- ClusterRole：集群角色，其实是定义一组对Kubernetes资源（集群级别，包含全部命名空间）的访问规则。
- ClusterRoleBinding：集群角色绑定，定义了用户和集群角色的关系。

Role和ClusterRole指定了可以对哪些资源做哪些动作，RoleBinding和ClusterRoleBinding将角色绑定到特定的用户、用户组或ServiceAccount上。如下图所示。

图 3-31 角色绑定



创建 Role

Role的定义非常简单，指定namespace，然后就是rules规则。如下面示例中的规则就是允许对default命名空间下的Pod进行GET、LIST操作。

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default                    # 命名空间
  name: role-example
rules:
- apiGroups: [""]
  resources: ["pods"]                   # 可以访问pod
  verbs: ["get", "list"]                # 可以执行GET、LIST操作
```

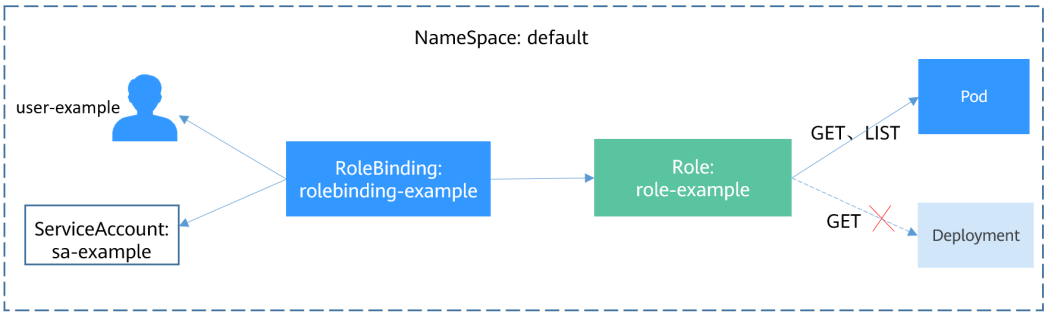
创建 RoleBinding

有了Role之后，就可以将Role与具体的用户绑定起来，实现这个的就是RoleBinding了。如下所示。

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rolebinding-example
  namespace: default
subjects:                          # 指定用户
- kind: User                        # 普通用户
  name: user-example
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount                # ServiceAccount
  name: sa-example
  namespace: default
roleRef:                           # 指定角色
  kind: Role
  name: role-example
  apiGroup: rbac.authorization.k8s.io
```

这里的subjects就是将Role与用户绑定起来，用户可以是外部普通用户，也可以是ServiceAccount，这两种用户类型在**ServiceAccount**有过介绍。绑定后的关系如下图所示。

图 3-32 RoleBinding 绑定 Role 和用户



下面来验证授权是否生效。

在前面一个章节**使用ServiceAccount**中，创建一个Pod，使用了sa-example这个ServiceAccount，而刚刚又给sa-example绑定了role-example这个角色，现在进入到Pod，使用curl命令通过API Server访问资源来验证权限是否生效。

使用sa-example对应的ca.crt和Token认证，查询default命名空间下所有Pod资源，对应**创建Role**中的LIST。

```
$ kubectl exec -it sa-example -- /bin/sh
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "10377013"
  },
  "items": [
    {
      "metadata": {
        "name": "sa-example",
```

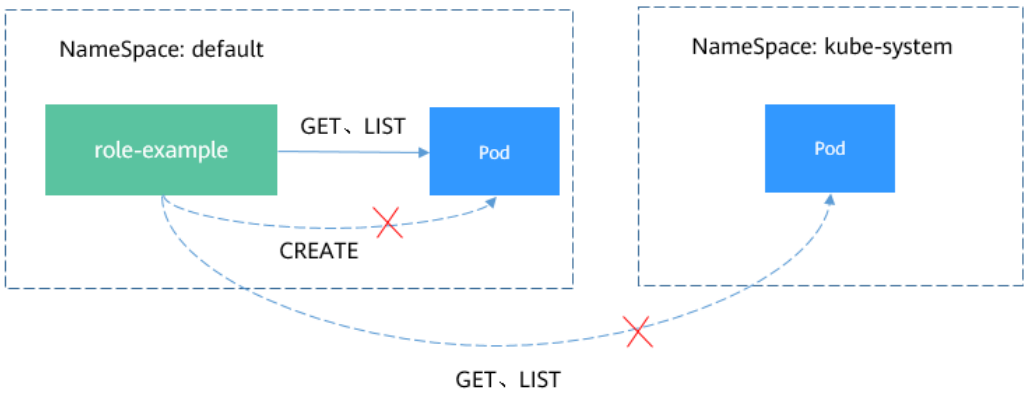
```
      "namespace": "default",
      "selfLink": "/api/v1/namespaces/default/pods/sa-example",
      "uid": "c969fb72-ad72-4111-a9f1-0a8b148e4a3f",
      "resourceVersion": "10362903",
      "creationTimestamp": "2020-07-15T06:19:26Z"
    },
    "spec": {
...
```

返回结果正常，说明sa-example是有LIST Pod的权限的。再查询Deployment，返回如下，说明没有访问Deployment的权限。

```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/deployments
...
  "status": "Failure",
  "message": "deployments is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list
resource \"deployments\" in API group \"\" in the namespace \"default\"",
...
```

Role和RoleBinding作用的范围是命名空间，能够做到一定程度的权限隔离，如下图所示，上面定义role-example就不能访问kube-system命名空间下的资源。

图 3-33 Role 和 RoleBinding 作用的范围是命名空间



在上面Pod中继续访问，返回如下，说明确实没有权限。

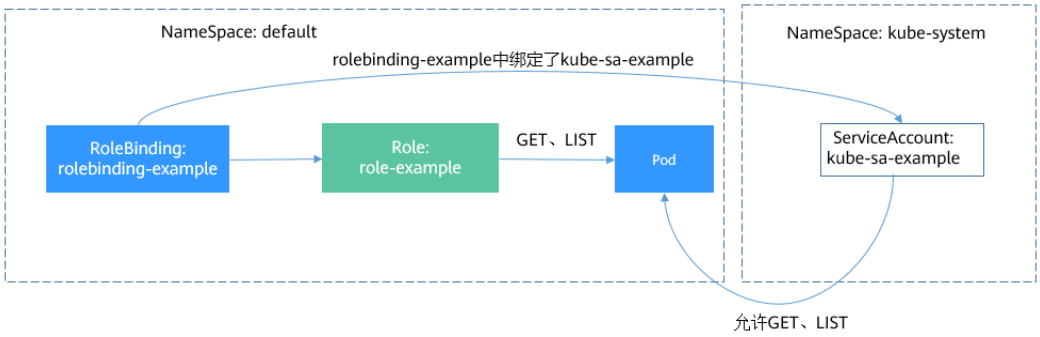
```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/kube-system/pods
...
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list resource
\"pods\" in API group \"\" in the namespace \"kube-system\"",
  "reason": "Forbidden",
...
```

在RoleBinding中，还可以绑定其他命名空间的ServiceAccount，只要在subjects字段下添加其他命名空间的ServiceAccount即可。

```
subjects:                          # 指定用户
- kind: ServiceAccount                   # ServiceAccount
  name: kube-sa-example
  namespace: kube-system
```

加入之后，kube-system下kube-sa-example这个ServiceAccount就可以GET、LIST命名空间default下的Pod了，如下图所示。

图 3-34 跨命名空间访问



ClusterRole 和 ClusterRoleBinding

相比Role和RoleBinding，ClusterRole和ClusterRoleBinding有如下几点不同：

- ClusterRole和ClusterRoleBinding不用定义namespace字段
- ClusterRole可以定义集群级别的资源

可以看出ClusterRole和ClusterRoleBinding控制的是集群级别的权限。

在Kubernetes中，默认定义了非常多的ClusterRole和ClusterRoleBinding，如下所示。

```
$ kubectl get clusterroles
NAME                                                                   AGE
admin                                                                  30d
cceaddon-prometheus-kube-state-metrics                                 6d3h
cluster-admin                                                          30d
coredns                                                                30d
custom-metrics-resource-reader                                         6d3h
custom-metrics-server-resources                                        6d3h
edit                                                                   30d
prometheus                                                             6d3h
system:aggregate-customedhorizontalpodautoscalers-admin                6d2h
system:aggregate-customedhorizontalpodautoscalers-edit                 6d2h
system:aggregate-customedhorizontalpodautoscalers-view                 6d2h
....
view                                                                   30d

$ kubectl get clusterrolebindings
NAME                                                   AGE
authenticated-access-network                           30d
authenticated-packageversion                           30d
auto-approve-csrs-for-group                            30d
auto-approve-renewals-for-nodes                        30d
auto-approve-renewals-for-nodes-server                 30d
cceaddon-prometheus-kube-state-metrics                 6d3h
cluster-admin                                          30d
cluster-creator                                        30d
coredns                                                30d
csrs-for-bootstrapping                                 30d
system:basic-user                                      30d
system:ccehpa-rolebinding                              6d2h
system:cluster-autoscaler                              6d1h
...
```

其中，最重要最常用的是如下四个ClusterRole。

- view：拥有查看命名空间资源的权限
- edit：拥有修改命名空间资源的权限

- admin：拥有命名空间全部权限
- cluster-admin：拥有集群的全部权限

使用**kubectl describe clusterrole**命令能够查看到各个规则的具体权限。

通常情况下，使用这四个ClusterRole与用户做绑定，就可以很好的做到权限隔离。这里的关键一点是理解到Role（规则、权限）与用户是分开的，只要通过Rolebinding来对这两者进行组合就能做到灵活的权限控制。

3.9 弹性伸缩

在**Pod的编排与调度**章节介绍了Deployment这类控制器来控制Pod的副本数量，通过调整replicas的大小就可以达到给应用手动扩缩容的目的。但是在某些实际场景下，手动调整一是繁琐，二是速度没有那么快，尤其是在应对流量洪峰需要快速弹性时无法做出快速反应。

Kubernetes支持Pod和集群节点的自动弹性伸缩，通过设置弹性伸缩规则，当外部条件（如CPU使用率）达到一定条件时，根据规则自动伸缩Pod和集群节点。

Prometheus 与 Metrics Server

想要做到自动弹性伸缩，先决条件就是能感知到各种运行数据，例如集群节点、Pod、容器的CPU、内存使用率等等。而这些数据的监控能力Kubernetes也没有自己实现，而是通过其他项目来扩展Kubernetes的能力。

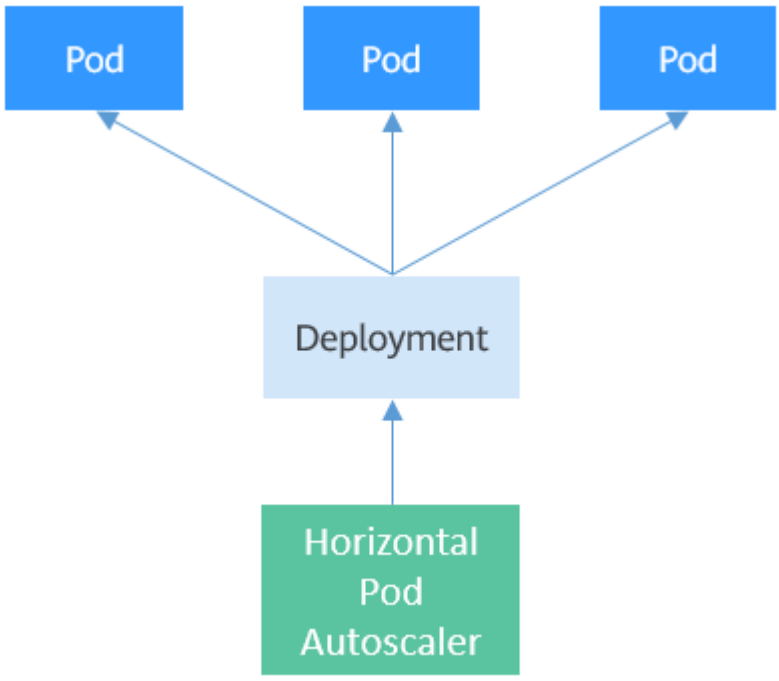
- **Prometheus**是一套开源的系统监控报警框架，能够采集丰富的Metrics（度量数据），目前已经基本是Kubernetes的标准监控方案。
- **Metrics Server**是Kubernetes集群范围资源使用数据的聚合器。Metrics Server从kubelet公开的Summary API中采集度量数据，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据，且对外提供一套标准的API。

使用HPA（Horizontal Pod Autoscaler）配合Metrics Server可以实现基于CPU和内存的自动弹性伸缩，再配合Prometheus还可以实现自定义监控指标的自动弹性伸缩。

HPA 工作机制

HPA（Horizontal Pod Autoscaler）是用来控制Pod水平伸缩的控制器，HPA周期性检查Pod的度量数据，计算满足HPA资源所配置的目标数值所需的副本数量，进而调整目标资源（如Deployment）的replicas字段。

图 3-35 HPA 工作机制



HPA可以配置单个和多个度量指标，配置单个度量指标时，只需要对Pod的当前度量数据求和，除以期望目标值，然后向上取整，就能得到期望的副本数。例如有一个Deployment控制有3个Pod，每个Pod的CPU使用率是70%、50%、90%，而HPA中配置的期望值是50%，计算期望副本数=（70 + 50 + 90）/50 = 4.2，向上取整得到5，即期望副本数就是5。

如果是配置多个度量指标，则会分别计算单个度量指标的期望副本数量，然后取其中最大值，就是最终的期望副本数量。

使用 HPA

下面通过示例演示HPA的使用。首先使用Nginx镜像创建一个4副本的Deployment。

```
$ kubectl get deploy
NAME               READY     UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   4/4       4            4           77s

$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-7cc6fd654c-5xzlt   1/1       Running   0          82s
nginx-deployment-7cc6fd654c-cwjzg   1/1       Running   0          82s
nginx-deployment-7cc6fd654c-dffkp   1/1       Running   0          82s
nginx-deployment-7cc6fd654c-j7mp8   1/1       Running   0          82s
```

创建一个HPA，期望CPU的利用率为70%，副本数的范围是1-10。

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: scale
  namespace: default
spec:
  maxReplicas: 10                  # 目标资源的最大副本数量
  minReplicas: 1                   # 目标资源的最小副本数量
  metrics:                         # 度量指标，期望CPU的利用率为70%
  - resource:
```

```
    name: cpu
    targetAverageUtilization: 70
  type: Resource
scaleTargetRef:                 # 目标资源
  apiVersion: apps/v1
  kind: Deployment
  name: nginx-deployment
```

创建后HPA查看。

```
$ kubectl create -f hpa.yaml
horizontalpodautoscaler.autoscaling/celue created

$ kubectl get hpa
NAME      REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
scale     Deployment/nginx-deployment   0%/70%    1         10        4          18s
```

可以看到，TARGETS的期望值是70%，而实际是0%，这就意味着HPA会做出缩容动作，期望副本数量=(0+0+0+0)/70=0，但是由于最小副本数为1，所以Pod数量会调整为1。等待一段时间，可以看到Pod数量变为1。

```
$ kubectl get pods
NAME                           READY    STATUS    RESTARTS   AGE
nginx-deployment-7cc6fd654c-5xzlt   1/1      Running   0          7m41s
```

查看HPA详情，可以在Events里面看到这样一条记录。这表示HPA在21秒前成功的执行了缩容动作，新的Pod数量为1，原因是所有度量数量都比目标值低。

```
$ kubectl describe hpa scale
...
Events:
  Type    Reason             Age   From                       Message
  ----    ------             ----  ----                       -------
  Normal  SuccessfulRescale  21s   horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

如果再查看Deployment的详情，可以在Events里面看到这样一条记录。这表示Deployment的副本数量被设置为1了，跟HPA中看到的一致。

```
$ kubectl describe deploy nginx-deployment
...
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  7m    deployment-controller  Scaled up replica set nginx-
deployment-7cc6fd654c to 4
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled down replica set nginx-
deployment-7cc6fd654c to 1
```

Cluster AutoScaler

HPA是针对Pod级别的，但是如果集群的资源不够了，那就只能对节点进行扩容了。集群节点的弹性伸缩本来是一件非常麻烦的事情，但是好在现在的集群大多都是构建在云上，云上可以直接调用接口添加删除节点，这就使得集群节点弹性伸缩变得非常方便。

Cluster Autoscaler是Kubernetes提供的集群节点弹性伸缩组件，根据Pod调度状态及资源使用情况对集群的节点进行自动扩容缩容。由于要调用云上接口实现弹性伸缩，这就使得在不同环境上的实现与使用各不相同，这里不详细介绍。

CCE的集群节点弹性伸缩请参见**创建节点伸缩策略**。

4 快速入门

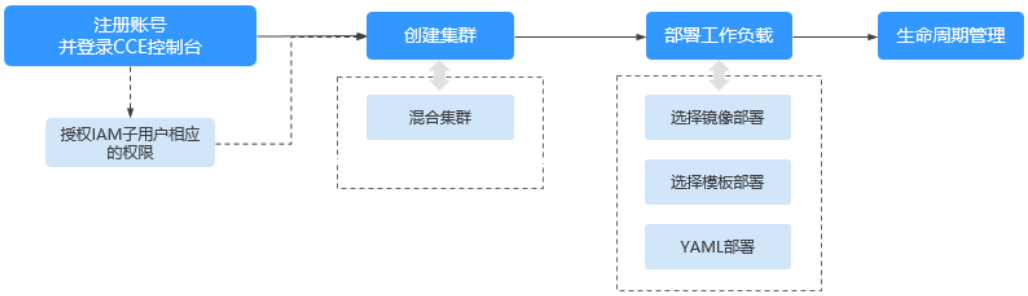
4.1 入门指引

本文旨在帮助您了解云容器引擎（Cloud Container Engine，简称CCE）的基本使用流程以及相关的常见问题，帮助您快速上手容器服务。

使用步骤

完整的云容器引擎使用流程包含以下步骤：

图 4-1 CCE 使用流程



步骤1 注册账号，并授予IAM用户相应的权限。

账号无需授权即可拥有所有权限，由账号创建的IAM子用户需要授予相应的权限才能使用CCE。

步骤2 创建集群。

如果您需要创建普通Kubernetes集群，请参见**快速创建Kubernetes集群**。

步骤3 通过镜像或编排模板创建工作负载（应用）。

- **镜像创建无状态工作负载（Nginx）**
- **部署有依赖关系的WordPress和MySQL**

步骤4 查看部署后工作负载的状态和日志信息，对工作负载进行相应的升级、伸缩和监控等。

----结束

常见问题

1. **我不懂kubernetes，是否可以使用CCE?**

   可以使用，CCE管理控制台操作简单，并提供新手入门指导文档，您可以快速了解并使用CCE。

2. **我不会制作镜像，是否可以使用CCE?**

   CCE除了提供“我的镜像”功能用于存储您自行创建的镜像外，您还可以基于开源镜像创建容器应用。详情请参考CCE快速入门**镜像创建无状态工作负载（Nginx）**。

3. **如何使用CCE创建工作负载?**

   创建工作负载非常简单，您只需要先创建一个集群，再创建工作负载即可。详细步骤请参考**镜像创建无状态工作负载（Nginx）**。

4. **如何创建一个可以在公网访问的工作负载?**

   云容器引擎为满足多种复杂场景下工作负载间的互相访问，提供了不同的访问方式，从而满足不同场景提供不同访问通道。

5. **我有多个工作负载（在同个集群中），它们之间需要互相访问，应该怎么办?**

   集群内访问表示工作负载暴露给同一集群内其他工作负载访问的方式，可以通过“集群内部域名”访问。

   集群内部域名格式为“<自定义的服务名称>.<工作负载所在命名空间>.svc.cluster.local:<端口号>”，例如“nginx.default.svc.cluster.local:80”。

4.2 准备工作

在使用云容器引擎前，您需要完成本文中的准备工作。

- **创建IAM用户**
- **获取资源权限**
- **（可选）创建虚拟私有云**
- **（可选）创建密钥对**

创建 IAM 用户

如果您需要多用户协同操作管理您账号下的资源，为了避免共享您的密码/访问密钥，您可以通过IAM创建用户，并授予用户对应权限。这些用户可以使用特别的登录链接和自己单独的用户账号访问，帮助您高效的管理资源，您还可以设置账号安全策略确保这些账号的安全，从而降低您的企业信息安全风险。

注册账号无需授权，由账号创建的IAM用户需要授予相应的权限才能使用CCE。

获取资源权限

由于CCE在运行中对计算、存储、网络以及监控等各类云服务资源都存在依赖关系，因此当您首次登录CCE控制台时，CCE将自动请求获取当前区域下的云资源权限，从而更好地为您提供服务。服务权限包括：

- 计算类服务

  CCE集群创建节点时会关联创建云服务器，因此需要获取访问弹性云服务器、裸金属服务器的权限。

- 存储类服务

  CCE支持为集群下节点和容器挂载存储，因此需要获取访问云硬盘、弹性文件、对象存储等服务的权限。

- 网络类服务

  CCE支持集群下容器发布为对外访问的服务，因此需要获取访问虚拟私有云、弹性负载均衡等服务的权限。

- 容器与监控类服务

  CCE集群下容器支持镜像拉取、监控和日志分析等功能，需要获取访问容器镜像、应用管理等服务的权限。

当您同意授权后，CCE将在IAM中创建名为“cce_admin_trust”委托，统一使用系统账号“op_svc_cce”对您的其他云服务资源进行操作，并且授予其Tenant Administrator权限。Tenant Administrator拥有除IAM管理外的全部云服务管理员权限，用于对CCE所依赖的其他云服务资源进行调用，且该授权仅在当前区域生效。

如果您在多个区域中使用CCE服务，则需在每个区域中分别申请云资源权限。您可前往“IAM控制台 > 委托”页签，单击“cce_admin_trust”查看各区域的授权记录。

说明

由于CCE对其他云服务有许多依赖，如果没有Tenant Administrator权限，可能会因为某个服务权限不足而影响CCE功能的正常使用。因此在使用CCE服务期间，请不要自行删除或者修改“cce_admin_trust”委托。

（可选）创建虚拟私有云

虚拟私有云为CCE集群提供一个隔离的、用户自主配置和管理的虚拟网络环境。

创建首个集群前，您必须先确保已存在虚拟私有云，否则无法创建集群。

若您已有虚拟私有云，可重复使用，无需重复创建。

步骤1 登录管理控制台。

步骤2 单击管理控制台左上角的，选择区域和项目。

步骤3 选择“网络 > 虚拟私有云”。

步骤4 单击“创建虚拟私有云”。

步骤5 在“创建虚拟私有云”页面，根据界面提示配置虚拟私有云参数。

创建虚拟私有云时会同时创建一个默认子网，您还可以单击“添加子网”创建多个子网。

步骤6 单击“立即创建”。

----结束

（可选）创建密钥对

云平台使用公共密钥密码术来保护您的云容器引擎节点的登录信息，密码或密钥对用于远程登录节点时的身份认证。

- 如果选择密钥登录方式，您需要在创建云容器引擎的集群节点时指定密钥对的名称，然后在SSH登录时提供私钥。
- 如果选择密码登录方式，可以跳过该任务。

说明

如果您计划在多个区域创建实例，则需要在每个区域中创建密钥对。

通过管理控制台创建密钥对

如果您尚未创建密钥对，可以通过管理控制台自行创建。步骤如下：

步骤1 登录管理控制台。

步骤2 单击管理控制台左上角的，选择区域和项目。

步骤3 选择“计算 > 弹性云服务器”。

步骤4 在左侧导航树中，选择“密钥对”。

步骤5 在“密钥对”页面，单击“创建密钥对”。

步骤6 输入密钥名称，单击“确定”。

步骤7 密钥名称由两部分组成：KeyPair-4位随机数字，使用一个容易记住的名称，如KeyPair-xxxx_ecs。

步骤8 您的浏览器会提示您下载或自动下载私钥文件。文件名是您为密钥对指定的名称，文件扩展名为“.pem”。请将私钥文件保存在安全位置。然后在系统弹出的提示框中单击“确定”。

说明

这是您保存私钥文件的唯一机会，请妥善保管。当您创建弹性云服务器时，您将需要提供密钥对的名称；每次SSH登录到弹性云服务器时，您将需要提供相应的私钥。

----结束

4.3 快速创建 Kubernetes 集群

背景信息

本章节将演示如何快速创建一个CCE集群，部分配置采用默认或最简配置。

创建集群

步骤1 登录CCE控制台。

- 如果您的账号还未创建过集群，会看见一个引导页面，请在CCE集群下单击“创建”。
- 如果您的账号已经创建过集群，请在左侧菜单栏选择集群管理，在右侧页面CCE集群下单击“创建”。

步骤2 在购买CCE集群页面的“服务选型”步骤中配置集群参数：

本例中大多数配置保留默认值，仅解释必要参数，参照**表4-1**设置服务选型参数。

表 4-1 创建集群参数配置

参数	参数说明
基础配置	
* 集群名称	新建集群的名称。集群名称长度范围为4-128个字符，以小写字母开头，由小写字母、数字、中划线（ - ）组成，且不能以中划线（ - ）结尾。
* 企业项目	该参数仅对开通企业项目的企业客户账号显示，不显示时请忽略。
* 集群版本	建议选择最新的版本。
* 集群规模	当前集群可以管理的最大**Node节点**规模。若选择50节点，表示当前集群最多可管理50个Node节点。
* 高可用	默认选择“是”。
网络配置	
* 网络模型	默认即可。
* 虚拟私有云	新建集群所在的虚拟私有云。 若没有可选虚拟私有云，单击“新建虚拟私有云”进行创建，完成创建后单击刷新按钮。
* 控制节点子网	集群Master节点所在的子网。
* 容器网段	按默认配置即可。
* IPv4 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。决定了Service资源的上限。 创建后不可修改。

步骤3 单击“下一步：插件配置”，配置默认插件即可。

步骤4 单击“下一步：规格确认”，显示集群资源清单，确认无误后，勾选“我已阅读并知晓上述使用说明”，单击“提交”。

等待集群创建成功，创建集群预计需要6-10分钟左右，请耐心等待。

创建成功后在集群管理下会显示一个运行中的集群，且集群节点数量为0。

----结束

创建节点

集群创建成功后，您还需要在集群中创建运行工作负载的节点。

步骤1 登录CCE控制台。

步骤2 单击创建的集群，进入集群控制台。

步骤3 在左侧菜单栏选择节点管理，单击右上角“创建节点”，在弹出的页面中配置节点的参数。

本例中大多数配置保留默认值，仅解释必要参数。

计算配置

- 可用区：默认即可。
- 节点类型：选择“虚拟机节点”。
- 节点规格：请根据业务需求选择相应的节点规格。
- 容器引擎：请根据业务需要选择相应的容器引擎。
- 操作系统：请选择节点对应的操作系统。
- 节点名称：自定义节点名称。
- 登录方式：
  - 选择“密钥对”：在选项框中选择用于登录本节点的密钥对，并单击勾选确认信息。

    密钥对用于远程登录节点时的身份认证。若没有密钥对，可单击选项框右侧的“创建密钥对”来新建。

存储配置

- 系统盘：按您的业务需求选择，缺省值为50GB。
- 数据盘：按您的业务需求选择，缺省值为100GB。

网络配置

- 虚拟私有云：使用默认，即创建集群时选择的子网。
- 节点子网：选择节点所在的子网。
- 节点IP：支持指定节点IP地址。
- 弹性公网IP：默认为“暂不使用”。支持“选择已有”和“自动创建”。

步骤4 在页面最下方选择节点的数量，单击“下一步: 规格确认”。

步骤5 查看节点规格无误后，阅读页面上的使用说明，勾选“我已阅读并知晓上述使用说明”，单击“提交”。

等待节点创建成功，添加节点预计需要6-10分钟左右，请耐心等待。

创建成功后在节点管理下会显示一个运行中的节点。

----结束

4.4 镜像创建无状态工作负载（Nginx）

您可以使用镜像快速创建一个可公网访问的单实例工作负载。本章节将指导您基于云容器引擎CCE快速部署Nginx容器应用，并管理该容器应用的全生命周期，以期让您具备将云容器引擎应用到实际项目中的能力。

前提条件

- 容器镜像服务中已包含Nginx。
- 您需要创建一个至少包含一个4核8G节点的集群，且该节点已绑定弹性IP。
- 集群是运行工作负载的逻辑分组，包含一组云服务器资源，每台云服务器即集群中的一个节点。
- 创建集群的方法，请参见**快速创建Kubernetes集群**。

Nginx 应用概述

Nginx是一款轻量级的Web服务器，您可通过CCE快速搭建nginx web服务器。

本章节将以创建Nginx应用为例，来创建一个工作负载，预计需要5分钟。

本章节执行完成后，可成功访问Nginx的网页，如下图。

图 4-2 本例结果

操作步骤

本章节将指导您通过容器镜像创建您的第一个容器工作负载。

步骤1 登录CCE控制台。

步骤2 单击集群进入集群控制台。

步骤3 在左侧菜单栏选择“工作负载”，单击右上角“创建负载”。

步骤4 填写以下参数，其它保持默认。

基本信息

- 负载类型：选择无状态负载。
- 负载名称：nginx。
- 命名空间：default。
- 实例数量：请设置为1。

容器配置

在基本信息中单击“选择镜像”，在弹出的窗口中选择“镜像中心”，并搜索“nginx”，选择nginx镜像。

服务配置

单击服务配置下的加号，创建服务（Service），用于从外部访问负载。本例将创建一个负载均衡类型的Service，请在右侧弹窗中配置如下参数。

- Service名称：输入应用发布的可被外部访问的名称，设置为：nginx。
- 访问类型：选择“负载均衡（LoadBalancer）”。
- 服务亲和：保持默认。
- 负载均衡器：如果已有负载均衡（ELB）实例，可以选择已有ELB，如果没有可选择“自动创建”，创建一个公网类型负载均衡器。

- 端口配置：
  - 对外协议：TCP。
  - 服务端口：设置为8080，该端口号将映射到容器端口。
  - 容器端口：容器中应用启动监听的端口，nginx镜像请设置为80，其他应用容器端口和应用本身的端口一致。

步骤5 单击右下角“创建工作负载”。

等待工作负载创建成功。

创建成功后在无状态负载下会显示一个运行中的工作负载。

----结束

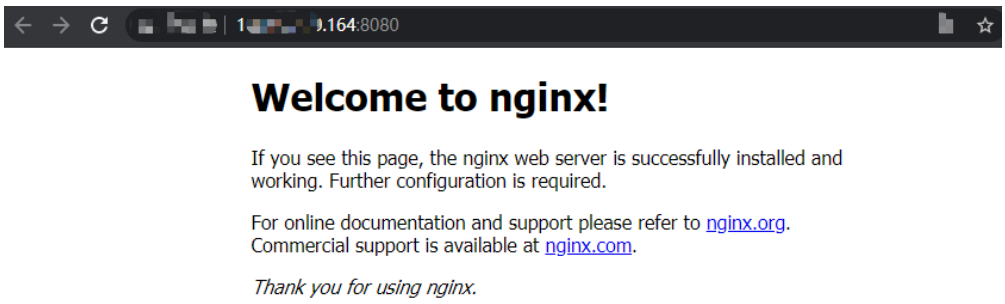
访问 Nginx

步骤1 获取Nginx的外部访问地址。

单击Nginx工作负载名称，进入工作负载详情页。在访问方式页签下可以看到nginx的IP地址，其中公网地址就是外部访问地址。

步骤2 在浏览器中输入“外部访问地址”，即可成功访问应用，如下图所示。

图 4-3 访问 nginx 应用



----结束

4.5 部署有依赖关系的 WordPress 和 MySQL

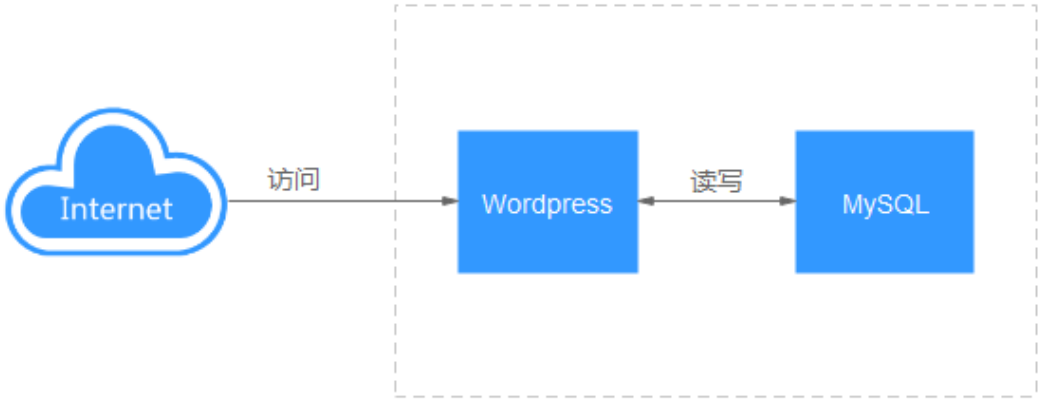
4.5.1 概述

WordPress是使用PHP语言和MySQL数据库开发的博客平台，并逐步演化成一款内容管理系统软件，您可以在支持PHP和MySQL数据库的服务器上架设属于自己的博客网站。WordPress拥有成千上万个各式插件和不计其数的主题模板样式，安装方式简单易用。

WordPress是使用PHP语言开发的博客平台。用户可以在支持PHP和MySQL数据库的服务上架设属于自己的网站，也可以把WordPress当作一个内容管理系统来使用。更多WordPress信息可以通过官方网站了解：**https://wordpress.org/**。

WordPress需配合MySQL一起使用，WordPress运行内容管理程序，MySQL作为数据库存储数据。在容器中运行通常会将WordPress和MySQL分别运行两个容器中，如下图所示。

图 4-4 WordPress



本例涉及到两个容器镜像。

- **WordPress**：本例选取wordpress:php7.3
- **MySQL**：本例选取mysql:5.7

在集群内部WordPress访问MySQL，Kubernetes提供一种叫服务（Service）的资源来解决负载的访问问题，本例中会为MySQL和WordPress分别创建一个Service，在后面的章节中您可以看到如何创建和配置。

4.5.2 步骤 1：创建 MySQL

WordPress需配合MySQL一起使用，WordPress运行内容管理程序，MySQL作为数据库存储数据。

前提条件

- 容器镜像服务中已包含WordPress和MySQL。
- 已创建一个包含4核8G节点的CCE集群。创建集群的方法，请参见**快速创建Kubernetes集群**。

创建 MySQL 负载

步骤1 登录CCE控制台。

步骤2 单击集群进入集群控制台。

步骤3 在左侧菜单栏选择“工作负载”，单击右上角“创建负载”。

步骤4 填写工作负载参数。

基本信息

- 负载类型：选择无状态负载。
- 负载名称：mysql。
- 命名空间：default。
- 实例数量：本例中修改数量为1。

容器配置

在基本信息中单击“选择镜像”，在弹出的窗口中选择“镜像中心”，并搜索“mysql”，选择mysql镜像。

选择镜像版本为“5.7”。

在环境变量下添加如下环境变量，此处一共需要设置四个环境变量。您可以在**MySQL**查看MySQL可以设置哪些环境变量。

- MYSQL_ROOT_PASSWORD：MySQL的root用户密码。
- MYSQL_DATABASE：镜像启动时要创建的数据库名称。
- MYSQL_USER：数据库用户名称。
- MYSQL_PASSWORD：数据库用户密码。

服务配置

单击服务配置下的加号，创建服务（Service），用于从Wordpress访问MySQL。

访问类型选择集群内访问（ClusterIP），服务名称设置为mysql，容器端口和服务端口都配置为3306，单击“确定”。

mysql镜像的默认访问端口默认为3306，所以容器端口的ID设置为3306，访问端口可以设置为其他端口号，但这里也设置成3306是为了方便使用。

这样在集群内部，通过**服务名称:访问端口**就可以访问MySQL负载，也就是**mysql:3306**。

步骤5 单击右下角“创建工作负载”。

等待工作负载创建成功。

创建成功后在无状态负载下会显示一个运行中的工作负载。

----结束

4.5.3 步骤 2：创建 WordPress

WordPress是使用PHP语言和MySQL数据库开发的博客平台，并逐步演化成一款内容管理系统软件，您可以在支持PHP和MySQL数据库的服务器上架设属于自己的博客网站。WordPress拥有成千上万个各式插件和不计其数的主题模板样式，安装方式简单易用。

本例主要演示如何使用镜像创建一个公开的WordPress网站。

前提条件

- 容器镜像服务中已包含WordPress和MySQL。
- 已创建一个包含4核8G节点的CCE集群。创建集群的方法，请参见**快速创建Kubernetes集群**。
- 已根据**步骤1：创建MySQL**部署MySQL数据库，本例中WordPress的数据将保存在该数据库中。

创建 WordPress 博客网站

步骤1 登录CCE控制台。

步骤2 单击集群进入集群控制台。

步骤3 在左侧菜单栏选择“工作负载”，单击右上角“创建负载”。

步骤4 填写工作负载参数。

基本信息

- 负载类型：选择无状态负载。
- 负载名称：wordpress。
- 命名空间：default。
- 实例数量：本例中实例数量设置为2。

容器配置

在基本信息中单击“选择镜像”，在弹出的窗口中选择“镜像中心”，并搜索“wordpress”，选择wordpress镜像。

选择镜像版本为“php7.3”。

在环境变量下添加如下环境变量，

此处一共需要设置四个环境变量，让WordPress知道MySQL数据库的信息。

- WORDPRESS_DB_HOST：数据库的访问地址。可以在mysql工作负载的访问方式中找到。可以使用集群内部域名mysql.default.svc.cluster.local:3306访问，其中.default.svc.cluster.local可以省略，即使用**mysql:3306**。
- WORDPRESS_DB_USER：访问数据的用户名，此处需要设置为**步骤1：创建MySQL**中MYSQL_USER一致，即使用这个用户去连接MySQL。
- WORDPRESS_DB_PASSWORD：访问数据库的密码，此处需要设置为**步骤1：创建MySQL**中MYSQL_PASSWORD一致。
- WORDPRESS_DB_NAME：访问数据库的名称，此处需要设置为**步骤1：创建MySQL**中MYSQL_DATABASE一致。

服务配置

单击服务配置下的加号，创建服务（Service），用于从外部访问负载。本例将创建一个负载均衡类型的Service，请在右侧弹窗中配置如下参数。

- Service名称：输入应用发布的可被外部访问的名称，设置为：wordpress。
- 访问类型：选择“负载均衡（LoadBalancer）”。
- 服务亲和：保持默认。
- 负载均衡器：如果已有负载均衡（ELB）实例，可以选择已有ELB，如果没有可选择“自动创建”，创建一个公网类型负载均衡器。
- 端口配置：
  - 对外协议：TCP。
  - 服务端口：设置为80，该端口号将映射到容器端口。
  - 容器端口：容器中应用启动监听的端口，wordpress镜像请设置为80，其他应用容器端口和应用本身的端口一致。

步骤5 单击右下角“创建工作负载”。

等待工作负载创建成功。

创建成功后在无状态负载下会显示一个运行中的工作负载。

----结束

访问 WordPress

步骤1 获取WordPress的外部访问地址。

单击WordPress工作负载名称，进入工作负载详情页。在访问方式页签下可以看到WordPress的IP地址，其中负载均衡IP就是外部访问地址。

步骤2 在浏览器中输入“外部访问地址”，即可成功访问应用。

访问到的WordPress应用如下图。

图 4-5 WordPress 应用-1

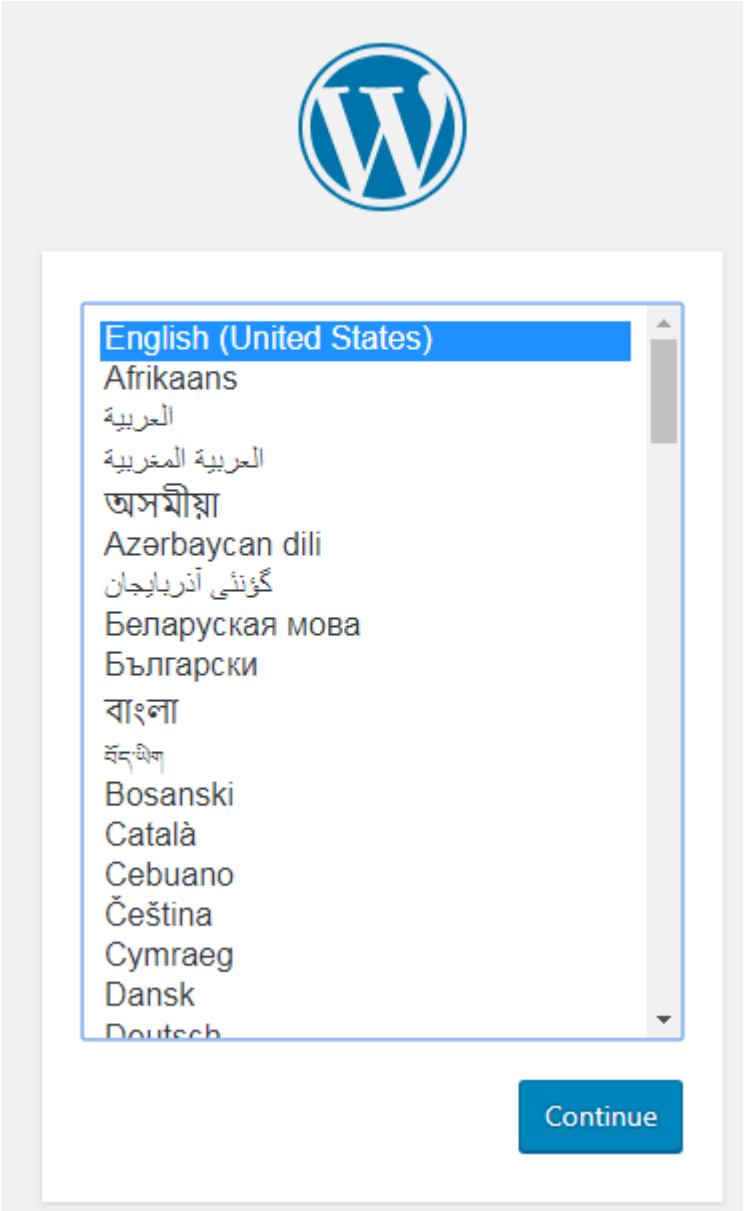
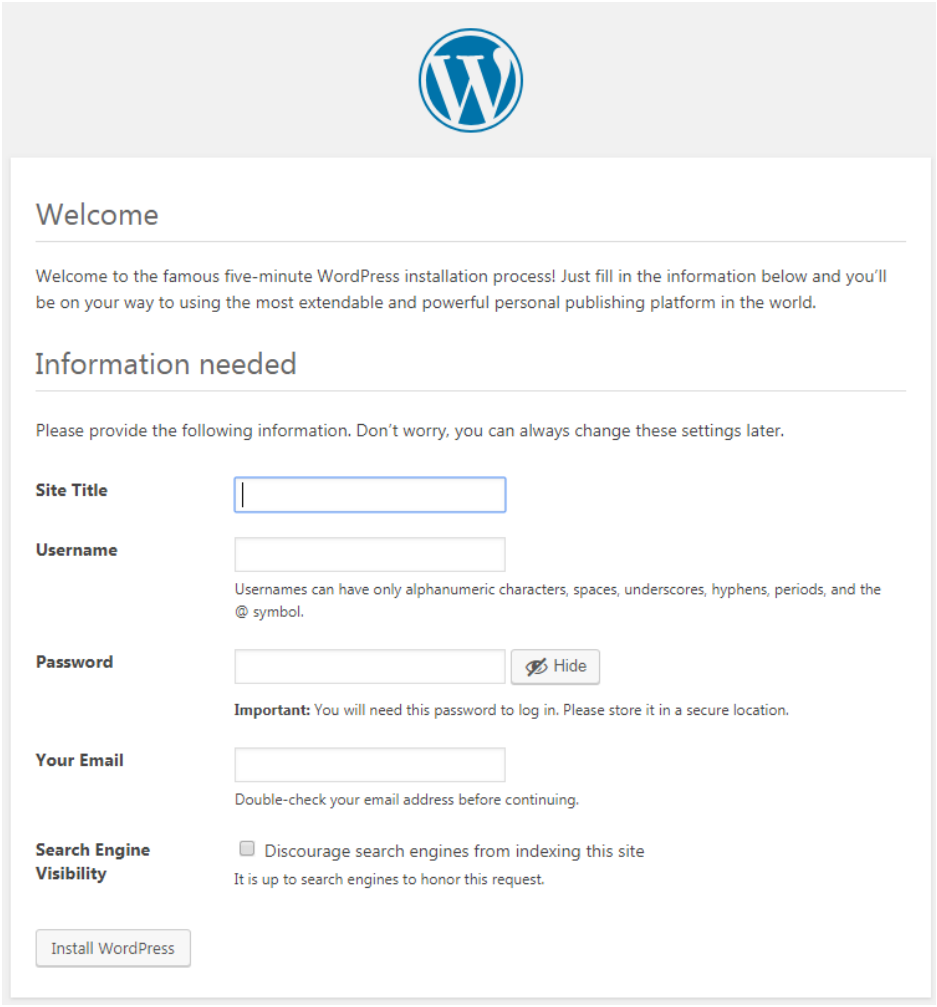


图 4-6 WordPress 应用-2



----结束

清除资源

您已经完成了入门的所有示例体验，基本了解了CCE的使用流程。节点运行过程中会产生费用，如果您不需要使用该集群，建议您参照以下步骤，删除节点，避免费用产生，如果您想继续体验CCE请继续保留集群节点资源。

步骤1 登录CCE控制台。

步骤2 单击左侧导航栏的“集群管理”。

步骤3 单击待删除集群后的，根据系统提示删除集群。

----结束

5 高危操作及解决方案

业务部署或运行过程中，用户可能会触发不同层面的高危操作，导致不同程度上的业务故障。为了能够更好地帮助用户预估及避免操作风险，本文将从集群/节点、网络与负载均衡、日志、云硬盘多个维度出发，为用户展示哪些高危操作会导致怎样的后果，以及为用户提供相应的误操作解决方案。

集群/节点

表 5-1 集群及节点高危操作

分类	高危操作	导致后果	误操作后解决方案
Master节点	修改集群内节点安全组	可能导致Master节点无法使用 **说明** 命名规则：集群名称-cce-control-随机数	参照新建集群的安全组进行修复，放通安全组。
	节点到期或被销毁	该Master节点不可用	不可恢复。
	重装操作系统	Master组件被删除	不可恢复。
	自行升级Master或者etcd组件版本	可能导致集群无法使用	回退到原始版本。
	删除或格式化节点/etc/kubernetes等核心目录数据	该Master节点不可用	不可恢复。
	更改节点IP	该Master节点不可用	改回原IP。
	自行修改核心组件（etcd、kube-apiserver、docker等）参数	可能导致Master节点不可用	按照推荐配置参数恢复，详情请参见**集群配置管理**。
	自行更换Master或etcd证书	可能导致集群不可用	不可恢复。

分类	高危操作	导致后果	误操作后解决方案
Node节点	修改集群内节点安全组	可能导致节点无法使用 **说明** 命名规则：集群名称-cce-node-随机数	参照新建集群的安全组进行修复，放通安全组。
	节点被删除	该节点不可用	不可恢复。
	重装操作系统	节点组件被删除，节点不可用	重置节点，具体请参见**重置节点**。
	升级节点内核	可能导致节点无法使用或网络异常 **说明** 节点运行依赖系统内核版本，如非必要，请不要使用**yum update**命令更新或重装节点的操作系统内核（使用原镜像或其它镜像重装均属高危操作）	重置节点，具体请参见**重置节点**。
	更改节点IP	节点不可用	改回原IP。
	自行修改核心组件（kubelet、kube-proxy等）参数	可能导致节点不可用、修改安全相关配置导致组件不安全等	按照推荐配置参数恢复，详情请参见**节点池配置管理**。
	修改操作系统配置	可能导致节点不可用	尝试还原配置项或重置节点，具体请参见**重置节点**。
	删除或修改/opt/cloud/cce、/var/paas目录，删除数据盘	节点不可用	重置节点，具体请参见**重置节点**。
	修改节点内目录权限、容器目录权限等	权限异常	不建议修改，请自行恢复。
	对节点进行磁盘格式化或分区，包括系统盘、Docker盘和kubelet盘	可能导致节点不可用	重置节点，具体请参见**重置节点**。
	在节点上安装自己的其他软件	导致安装在节点上的Kubernetes组件异常，节点状态变成不可用，无法部署工作负载到此节点	卸载已安装软件，尝试恢复或重置节点，具体请参见**重置节点**。
	修改NetworkManager的配置	节点不可用	重置节点，具体请参见**重置节点**。

分类	高危操作	导致后果	误操作后解决方案
	删除节点上的cce-pause等系统镜像	导致无法正常创建容器，且无法拉取系统镜像	请从其他正常节点拷贝该镜像恢复

网络

表 5-2 网络

高危操作	导致后果	误操作后解决方案
修改内核参数net.ipv4.ip_forward=0	网络不通	修改内核参数为net.ipv4.ip_forward=1
修改内核参数net.ipv4.tcp_tw_recycle=1	导致nat异常	修改内核参数net.ipv4.tcp_tw_recycle=0
修改内核参数net.ipv4.tcp_tw_reuse=1	导致网络异常	修改内核参数net.ipv4.tcp_tw_reuse=0
节点安全组配置未放通容器CIDR的53端口udp	集群内DNS无法正常工作	参照**新建集群**的安全组进行修复，放通安全组。
删除default-network的network-attachment-definitions的crd资源	容器网络不通，集群删除失败等	误删除该资源需要使用正确的配置重新创建default-network资源。

负载均衡

表 5-3 Service ELB

高危操作	导致后果	误操作后解决方案
通过ELB的控制台修改ELB的IPv4私有IP	• 基于IPv4私有IP进行私网流量转发功能会出现中断 • Service/Ingress的YAML中status字段下的IP变化	不建议修改，请自行恢复。
通过ELB的控制台解绑ELB的IPv4公网IP	解绑公网IP后，该弹性负载均衡器变更为私网类型，无法进行公网流量转发。	请自行恢复。

高危操作	导致后果	误操作后解决方案
通过ELB的控制台在CCE管理的ELB创建自定义的监听器	若ELB是创建Service/Ingress时自动创建的，在Service/Ingress删除时无法删除ELB的自定义监听器，会导致无法自动删除ELB。	通过Service/Ingress自动创建监听器，否则需要手动删除ELB。
通过ELB的控制台删除CCE自动创建的监听器	• 导致Service/Ingress访问不通。 • 在集群升级等需要重启控制节点的场景，所做修改会被CCE侧重置。	重新创建或更新Service/Ingress。
通过ELB的控制台修改CCE创建的监听器名称、访问控制、超时时间、描述等基本配置	如果监听器被删除，在集群升级等需要重启控制节点的场景，所做修改会被CCE侧重置。	不建议修改，请自行恢复。
通过ELB的控制台修改CCE创建的监听器后端服务器组，添加、删除后端服务器	• 导致Service/Ingress访问不通。 • 在集群升级等需要重启控制节点的场景，所做修改会被CCE侧重置： – 用户删除的后端服务器会恢复 – 用户添加的后端服务器会被移除	重新创建或更新Service/Ingress。
通过ELB的控制台更换CCE创建的监听器后端服务器组	• 导致Service/Ingress访问不通。 • 在集群升级等需要重启控制节点的场景，后端服务器组中的后端服务器会被CCE侧重置。	重新创建或更新Service/Ingress。
通过ELB的控制台修改CCE创建的监听器转发策略，添加、删除转发规则	• 导致Service/Ingress访问不通。 • 如果该转发规则由Ingress添加，在集群升级等需要重启控制节点的场景，所做修改会被CCE侧重置。	不建议修改，请自行恢复。
通过ELB的控制台修改CCE管理的ELB证书	在集群升级等需要重启控制节点的场景，后端服务器组中的后端服务器会被CCE侧重置。	通过Ingress的YAML来自动管理证书。

日志

表 5-4 日志

高危操作	导致后果	误操作后解决方案
删除宿主机/tmp/ccs-log-collector/pos目录	日志重复采集	无
删除宿主机/tmp/ccs-log-collector/buffer目录	日志丢失	无

云硬盘

表 5-5 云硬盘

高危操作	导致后果	误操作后解决方案	备注
控制台手动解除挂载EVS	Pod写入出现IO Error故障	删除节点上mount目录，重新调度Pod	Pod里面的文件记录了文件的采集位置
节点上umount磁盘挂载路径	Pod写入本地磁盘	重新mount对应目录到Pod中	Buffer里面是待消费的日志缓存文件
节点上直接操作EVS	Pod写入本地磁盘	无	无

6 集群

6.1 集群概述

6.1.1 集群基本信息

Kubernetes是一个开源的容器编排引擎，可用于容器化应用的自动化部署、 扩缩和管理。

对应用开发者而言，可以把Kubernetes看成一个集群操作系统。Kubernetes提供服务发现、伸缩、负载均衡、自愈甚至选举等功能，让开发者从基础设施相关配置中解脱出来。

集群的网络

集群的网络可以分成三个部分：

- 节点网络：为集群内节点分配IP地址。
- 容器网络：为集群内容器分配IP地址，负责容器的通信，当前支持多种容器网络模型，不同模型有不同的工作机制。
- 服务网络：服务（Service）是用来解决访问容器的Kubernetes对象，每个Service都有一个固定的IP地址。

在创建集群时，您需要为各个网络选择合适的网段，确保各网段之间不存在冲突，每个网段下有足够的IP地址可用。**集群创建后不支持修改容器网络模型**，您需要在创建前做好规划和选择。

强烈建议您在创建集群前详细了解集群的网络以及容器网络模型，具体请参见**容器网络模型**。

Master 节点数量与集群规模

在CCE中创建集群，Master节点可以是1个或3个，3个Master节点会按高可用部署，确保集群的可靠性。

Master节点的规格决定集群管理Node节点的规模，创建集群时可以选择集群管理规模，这个规模就是指的集群可以有多少个Node节点，例如50节点、200节点等。

集群生命周期

表 6-1 集群状态说明

状态	说明
创建中	集群正在创建，正在申请云资源
运行中	集群正常运行
休眠中	集群正在休眠中
唤醒中	集群正在唤醒中
升级中	集群正在升级中
不可用	当前集群不可用
删除中	集群正在删除中

6.1.2 Kubernetes 版本发布说明

6.1.2.1 Kubernetes 1.27 版本说明

云容器引擎（CCE）严格遵循社区一致性认证，现已支持创建Kubernetes 1.27集群。本文介绍Kubernetes 1.27版本相对于1.25版本所做的变更说明。

索引

主要特性

Kubernetes 1.27版本

- SeccompDefault特性已进入稳定阶段

  如需使用SeccompDefault特性，您需要为每个节点的kubelet启用--seccomp-default命令行标志。如果启用该特性，kubelet将为所有工作负载默认使用RuntimeDefault seccomp配置文件，该配置文件由容器运行时定义，而不是使用Unconfined（禁用seccomp）模式。

- Job可变调度指令

  该特性在Kubernetes 1.22版本中引入，当前已进入稳定阶段。在大多数情况下，并行作业Pod希望在一定的约束下运行，例如希望所有Pod在同一可用区。该特性允许在Job开始前修改调度指令。您可以使用suspend字段挂起Job，在Job挂起阶段，Pod模板中的调度部分（例如节点选择器、节点亲和性、反亲和性、容忍度）允许修改。详情请参见可变调度指令。

- Downward API HugePages已进入稳定阶段

在Kubernetes 1.20版本中，**Downward API**引入了*requests.hugepages-<pagesize>*和*limits.hugepages-<pagesize>*，HugePage可以和其他资源一样设置资源配额。

- Pod调度就绪态进入Beta阶段

  Pod创建后，Kubernetes调度程序会负责选择合适的节点运行pending状态的Pod。在实际使用时，一些Pod可能会由于资源不足长时间处于pending状态。这些Pod可能会影响集群中的其他组件运行（如Cluster Autoscaler）。通过指定/删除Pod的.spec.schedulingGates，您可以控制Pod何时准备好进行调度。详情请参见**Pod调度就绪态**。

- 通过Kubernetes API访问节点日志

  此功能当前处于Alpha阶段。集群管理员可以直接查询节点上的服务日志，可以帮助调试节点上运行的服务问题。如需使用此功能，请确保在该节点上启用了NodeLogQuery**特性门控**，并且kubelet配置选项enableSystemLogHandler和enableSystemLogQuery都设置为true。

- ReadWriteOncePod访问模式进入Beta阶段

  在Kubernetes 1.22版本中，PV和PVC提供了一种新的访问模式ReadWriteOncePod，该功能当前进入Beta阶段。卷可以被单个Pod以读写方式挂载。如果你想确保整个集群中只有一个Pod可以读取或写入该PVC，请使用ReadWriteOncePod访问模式，详情请参见**访问模式**。

- Pod拓扑分布约束中matchLabelKeys字段进入Beta阶段

  matchLabelKeys是一个Pod标签键的列表，用于选择需要计算分布方式的Pod集合。使用matchLabelKeys字段，您无需在变更Pod修订版本时更新pod.spec。控制器或Operator只需要将不同修订版的标签键设为不同的值。调度器将根据matchLabelKeys自动确定取值。详情请参见**Pod拓扑分布约束**。

- 快速标记SELinux卷标签功能进入Beta阶段

  默认情况下，容器运行时递归地将SELinux标签赋予所有Pod卷上的所有文件。 为了加快该过程，Kubernetes使用挂载可选项*-o context=<label>*可以立即改变卷的SELinux标签。详情请参见**快速标记SELinux卷标签**。

- VolumeManager重构进入Beta阶段

  重构的VolumeManager后，如果启用NewVolumeManagerReconstruction**特性门控**，将会在kubelet启动期间使用更有效的方式来获取已挂载卷。

- 服务器端字段校验和OpenAPI V3已进入稳定阶段

  Kubernetes 1.23中添加了对OpenAPI v3的支持，1.24版本中已进入Beta阶段，1.27已进入稳定阶段。

- 控制StatefulSet启动序号

  Kubernetes 1.26为StatefulSet引入了一个新的Alpha级别特性，可以控制Pod副本的序号。 从Kubernetes 1.27开始，此特性进入Beta阶段，序数可以从任意非负数开始。详情请参见**Kubernetes 1.27: StatefulSet 启动序号简化了迁移**。

- HorizontalPodAutoscaler ContainerResource类型指标进入Beta阶段

  Kubernetes 1.20在HorizontalPodAutoscaler (HPA) 中引入了**ContainerResource类型指标**。在 Kubernetes 1.27中，此特性进阶至Beta，相应的特性门控 (HPAContainerMetrics) 默认被启用。

- StatefulSet PVC自动删除进入Beta阶段

  Kubernetes v1.27提供一种新的策略机制，用于控制StatefulSets的PersistentVolumeClaims（PVCs）的生命周期。这种新的PVC保留策略允许用户指定当删除StatefulSet或者缩减StatefulSet中的副本时，是自动删除还是保留从StatefulSet规约模板生成的PVC。详情请参见**PersistentVolumeClaim保留**。

- 磁盘卷组快照

  磁盘卷组快照在Kubernetes 1.27中作为Alpha特性被引入。 此特性允许用户对多个卷进行快照，以保证在发生故障时数据的一致性。 它使用标签选择器来将多个PersistentVolumeClaims分组以进行快照。 这个新特性仅支持CSI卷驱动器。详情请参见**Kubernetes 1.27：介绍用于磁盘卷组快照的新API**。

- kubectl apply裁剪更安全、更高效

  在Kubernetes 1.5版本中，kubectl apply引入了--prune标志来删除不再需要的资源，允许kubectl apply自动清理从当前配置中删除的资源。然而，现有的--prune实现存在设计缺陷，会降低性能并导致意外行为。Kubernetes 1.27中，kubectl apply提供基于ApplySet的剪裁方式，当前处于Alpha阶段，详情请参见**使用配置文件对Kubernetes对象进行声明式管理**。

- 为NodePort Service分配端口时避免冲突

  在Kubernetes 1.27中，您可以启用新的**特性门控** ServiceNodePortStaticSubrange，为NodePort Service使用不同的端口分配策略，减少冲突的风险。当前该特性处于Alpha阶段。

- 原地调整Pod资源

  在Kubernetes 1.27中，允许用户调整分配给Pod的CPU和内存资源大小，而无需重新启动容器。当前该特性处于Alpha阶段，详情请参见**纵向弹性伸缩**。

- 加快Pod启动

  在Kubernetes 1.27中进行了一系列的参数调整，以提高Pod的启动速度，例如并行镜像拉取、提高Kubelet默认API每秒查询限值等。详情请参见**Kubernetes 1.27：关于加快Pod启动的进展**。

- KMS V2进入Beta阶段

  Kubernetes中的密钥管理KMS v2 API进入Beta阶段，对KMS加密提供程序的性能进行了重大改进。详情请参见**使用KMS驱动进行数据加密**。

Kubernetes 1.26版本

- 移除CRI v1alpha2

  Kubernetes 1.26版本不再支持CRI v1alpha2，请使用v1（要求containerd版本>= 1.5.0）。这意味着Kubernetes 1.26将不支持containerd 1.5.x 及更早的版本；需要升级到containerd 1.6.x或更高版本后，才能将该节点的kubelet升级到1.26。

  说明

  CCE目前使用的containerd版本为1.6.14，已满足要求。如存量的节点不满足containerd版本要求，请将节点重置为最新版本。

- 动态资源分配 Alpha API

  在Kubernetes 1.26版本，新增**动态资源分配**功能，用于Pod之间和Pod内部容器之间请求和共享资源，支持用户提供参数初始化资源。该功能尚处于alpha阶段，需要启用DynamicResourceAllocation特性门禁和resource.k8s.io/v1alpha1 API组，需要为要管理的特定资源安装驱动程序。更多信息，请参见**Kubernetes 1.26: 动态资源分配 Alpha API**。

- 节点非体面关闭进入Beta阶段

  在Kubernetes 1.26 中，节点非体面关闭特性是Beta版，默认被启用。当kubelet的节点关闭管理器可以检测到即将到来的节点关闭操作时，节点关闭才被认为是体面的。详情请参见**处理节点非体面关闭**。

- 支持在挂载时将Pod fsGroup传递给CSI驱动程序

将fsGroup委托给CSI驱动程序管理首先在Kubernetes 1.22中作为Alpha特性引入， 并在Kubernetes 1.25中进阶至Beta状态。该特性在Kubernetes 1.26已进入正式发布阶段，详情请参见**将卷权限和所有权更改委派给CSI驱动程序**。

- Pod调度就绪态

  Kubernetes 1.26引入了一个新的Pod特性schedulingGates，可以让调度器感知到何时可以进行Pod调度。详情请参见**Pod调度就绪态**。

- CPU Manager正式发布

  CPU管理器是kubelet的一部分，从Kubernetes 1.10**进阶至 Beta**，能够将独占CPU分配给容器。该特性在Kubernetes 1.26已进入稳定阶段，详情请参见**控制节点上的CPU管理策略**。

- Kubernetes中流量工程的进步

  优化内部节点本地流量和**支持EndpointSlice终止状况**升级为正式发布版本，**ProxyTerminatingEndpoints**功能升级为Beta版本。

- 支持跨命名空间存储数据源

  Kubernetes 1.26允许在源数据属于不同的命名空间时为PersistentVolumeClaim指定数据源。当前该特性处于Alpha阶段，详情请参见**跨命名空间数据源**。

- 可追溯的默认StorageClass进入Beta阶段

  Kubernetes 1.25引入了一个Alpha特性来更改默认StorageClass被分配到PersistentVolumeClaim (PVC) 的方式。 启用此特性后，您不再需要先创建默认StorageClass，再创建PVC来分配类。 此外，任何未分配StorageClass的PVC都可以在后续被更新。此特性在Kubernetes 1.26 中已进入Beta阶段，详情请参见**可追溯的默认StorageClass赋值**。

- PodDisruptionBudget支持指定不健康Pod的驱逐策略

  Kubernetes 1.26允许针对**PodDisruptionBudget** (PDB) 指定不健康Pod驱逐策略，这有助于在节点执行管理操作期间保持可用性。当前该特性处于Beta阶段，详情请参见**不健康的Pod驱逐策略**。

- 支持设置水平伸缩Pod控制器的数量

  kube-controller-manager支持flag --concurrent-horizontal-pod-autoscaler-syncs设置水平伸缩Pod控制器的worker数量。

弃用和移除

Kubernetes 1.27版本

- 在Kubernetes 1.27版本，针对卷扩展 GA 特性的以下特性门禁将被移除，且不得再在 --feature-gates 标志中引用。（**ExpandCSIVolumes，ExpandInUsePersistentVolumes，ExpandPersistentVolumes**）
- 在Kubernetes 1.27版本，移除--master-service-namespace 命令行参数。该参数支持指定在何处创建名为kubernetes的Service来表示API服务器。自v1.26版本已被弃用，1.27版本正式移除。
- 在Kubernetes 1.27版本，移除 ControllerManagerLeaderMigration 特性门禁。**Leader Migration** 提供了一种机制，让 HA 集群在升级多副本的控制平面时通过在 kube-controller-manager 和 cloud-controller-manager 这两个组件之间共享的资源锁，安全地迁移“特定于云平台”的控制器。特性自 v1.24 正式发布，被无条件启用， 在 v1.27 版本中此特性门禁选项将被移除。
- 在Kubernetes 1.27版本，移除 --enable-taint-manager 命令行参数。该参数支持的特性基于污点的驱逐已被默认启用， 且在标志被移除时也将继续被隐式启用。

- 在Kubernetes 1.27版本，移除--pod-eviction-timeout 命令行参数。弃用的命令行参数 --pod-eviction-timeout 将被从 kube-controller-manager 中移除。
- 在Kubernetes 1.27版本，移除 CSI Migration 特性门禁。**CSI migration** 程序允许从树内卷插件移动到树外 CSI 驱动程序。 CSI 迁移自 Kubernetes v1.16 起正式发布，关联的 CSIMigration 特性门禁将在 v1.27 中被移除。
- 在Kubernetes 1.27版本，移除 CSIInlineVolume 特性门禁。**CSI Ephemeral Volume** 特性允许在 Pod 规约中直接指定 CSI 卷作为临时使用场景。这些 CSI 卷可用于使用挂载的卷直接在 Pod 内注入任意状态，例如配置、Secret、身份、变量或类似信息。 此特性在 v1.25 中进阶至正式发布。因此，此特性门禁 CSIInlineVolume 将在 v1.27 版本中移除。
- 在Kubernetes 1.27版本，移除 EphemeralContainers 特性门禁。对于 Kubernetes v1.27，临时容器的 API 支持被无条件启用；EphemeralContainers 特性门禁将被移除。
- 在Kubernetes 1.27版本，移除 LocalStorageCapacityIsolation 特性门禁。**Local Ephemeral Storage Capacity Isolation** 特性在 v1.25 中进阶至正式发布。此特性支持 emptyDir 卷这类 Pod 之间本地临时存储的容量隔离， 因此可以硬性限制 Pod 对共享资源的消耗。如果本地临时存储的消耗超过了配置的限制，kubelet 将驱逐 Pod。 特性门禁 LocalStorageCapacityIsolation 将在 v1.27 版本中被移除。
- 在Kubernetes 1.27版本，移除 NetworkPolicyEndPort 特性门禁。Kubernetes v1.25 版本将 NetworkPolicy 中的 endPort 进阶至正式发布。 支持 endPort 字段的 NetworkPolicy 提供程序可用于指定一系列端口以应用 NetworkPolicy。
- 在Kubernetes 1.27版本，移除 StatefulSetMinReadySeconds 特性门禁。对于作为 StatefulSet 一部分的 Pod，只有当 Pod 至少在 **minReadySeconds** 中指定的持续期内可用（并通过检查）时，Kubernetes 才会将此 Pod 标记为只读。 该特性在 Kubernetes v1.25 中正式发布，StatefulSetMinReadySeconds 特性门禁将锁定为 true，并在 v1.27 版本中被移除。
- 在Kubernetes 1.27版本，移除 IdentifyPodOS 特性门禁。启用该特性门禁，你可以为 Pod 指定操作系统，此项特性支持自 v1.25 版本进入稳定。 IdentifyPodOS 特性门禁将在 Kubernetes v1.27 中被移除。
- 在Kubernetes 1.27版本，移除 DaemonSetUpdateSurge 特性门禁。Kubernetes v1.25 版本还稳定了对 DaemonSet Pod 的浪涌支持， 其实现是为了最大限度地减少部署期间 DaemonSet 的停机时间。 DaemonSetUpdateSurge 特性门禁将在 Kubernetes v1.27 中被移除。
- 在Kubernetes 1.27版本，移除 --container-runtime 命令行参数。kubelet 接受一个已弃用的命令行参数 --container-runtime， 并且在移除 dockershim 代码后，唯一有效的值将是 remote。 Kubernetes v1.27 将移除该参数，该参数自 v1.24 版本以来已被弃用。

Kubernetes 1.26版本

- 移除v2beta2版本的HorizontalPodAutoscaler API

  HorizontalPodAutoscaler的autoscaling/v2beta2 API版本将不再在1.26版本中提供，详情请参见**各发行版本中移除的API**。 用户应迁移至autoscaling/v2版本的API。
- 移除v1beta1版本的流量控制API组

  在Kubernetes 1.26版本后，开始不再提供flowcontrol.apiserver.k8s.io/v1beta1 API版本的FlowSchema和PriorityLevelConfiguration，详情请参见**各发行版本中移除的API**。但此API从Kubernetes 1.23版本开始，可以使用flowcontrol.apiserver.k8s.io/v1beta2；从Kubernetes 1.26版本开始，可以使用flowcontrol.apiserver.k8s.io/v1beta3。

- 存储驱动的弃用和移除，移除云服务厂商的in-tree卷驱动。
- 移除kube-proxy userspace模式

  在Kubernetes 1.26版本，Userspace代理模式已被移除，已弃用的Userspace代理模式不再受Linux或Windows支持。Linux用户应使用Iptables或IPVS，Windows用户应使用Kernelspace，现在使用--mode userspace会失败。

  – Windows winkernel kube-proxy不再支持Windows HNS v1 APIs。
- 弃用--prune-whitelist标志

  在Kubernetes 1.26版本，为了支持**Inclusive Naming Initiative**，--prune-whitelist标志将被**弃用**，并替换为--prune-allowlist，该标志在未来将彻底移除。
- 移除动态Kubelet配置

  DynamicKubeletConfig特性门控移除，通过API动态更新节点上的Kubelet配置。在Kubernetes 1.24版本中从Kubelet移除相关代码，在Kubernetes 1.26版本从APIServer移除相关代码，移除该逻辑有助于简化代码提升可靠性，推荐方式是修改Kubelet配置文件然后重启Kubelet。更多信息，请参见**在Kubernetes 1.26版本从APIServer移除相关代码**。
- 弃用kube-apiserver命令行参数

  在Kubernetes 1.26版本，正式标记**弃用 --master-service-namespace** 命令行参数，它对APIServer没有任何效果。
- 弃用kubectl run命令行参数

  在Kubernetes 1.26版本，kubectl run未使用的几个子命令将被标记为**弃用**，并在未来某个版本移除，包括--cascade、--filename、--force、--grace-period、--kustomize、--recursive、--timeout、--wait等这些子命令。
- 移除与日志相关的原有命令行参数

  在Kubernetes 1.26版本，将**移除**一些与日志相关的命令行参数，这些参数在之前的版本已被**弃用**。

CCE 对 Kubernetes 1.27 版本的增强

在版本维护周期中，CCE会对Kubernetes 1.27版本进行定期的更新，并提供功能增强。

关于CCE集群版本的更新说明，请参见**CCE集群版本发布说明**。

参考链接

关于Kubernetes 1.27与其他版本的性能对比和功能演进的更多信息，请参考：

- **Kubernetes v1.27 Release Notes**
- **Kubernetes v1.26 Release Notes**

6.1.2.2 Kubernetes 1.25 版本说明

云容器引擎（CCE）严格遵循社区一致性认证。本文介绍Kubernetes 1.25版本相对于1.23版本所做的变更说明。

索引

- **主要特性**

- **弃用和移除**
- **CCE对Kubernetes 1.25版本的增强**
- **参考链接**

主要特性

Kubernetes 1.25版本

- Pod Security Admission进入稳定阶段，并移除PodSecurityPolicy

  PodSecurityPolicy被废弃，并提供Pod Security Admission取代，具体的迁移方法可参见**从PodSecurityPolicy迁移到内置的PodSecurity准入控制器**。

- Ephemeral Containers进入稳定阶段

  临时容器是在现有的Pod中存在有限时间的容器。它对故障排除特别有用，特别是当需要检查另一个容器，但因为该容器已经崩溃或其镜像缺乏调试工具不能使用kubectl exec时。

- 对cgroups v2的支持进入稳定阶段

  Kubernetes支持cgroups v2，与cgroups v1相比提供了一些改进，详情请参见**cgroups v2**。

- SeccompDefault提升到Beta状态

  如果要开启该特性，需要给kubelet增加启动参数为--seccomp-default=true，这样会默认开启seccomp为RuntimeDefault，提升整个系统的安全。1.25集群将不再支持使用注解“seccomp.security.alpha.kubernetes.io/pod”和“container.seccomp.security.alpha.kubernetes.io/annotation”来使用seccomp，请使用pod或container中“securityContext.seccompProfile”字段替代，详情请参见**为Pod或容器配置安全上下文**。

  说明

  特性开启后可能应用所需的系统调用会被runtime限制，所以开启后应确保在测试环境调试，不会对应用造成影响。

- 网络策略中的EndPort进入稳定阶段

  Network Policy中的EndPort已进入稳定状态，该特性于1.21版本合入。主要是在NetworkPolicy新增EndPort，可以指定一个Port范围，避免声明每一个Port。

- 本地临时容器存储容量隔离进入稳定阶段

  本地临时存储容量隔离功能提供了对Pod之间本地临时存储容量隔离的支持，如EmptyDir。因此，如果一个Pod对本地临时存储容量的消耗超过该限制，就可以通过驱逐Pod来硬性限制其对共享资源的消耗。

- CRD验证表达式语言升级为Beta阶段

  CRD验证表达式语言已升级为 beta 版本， 这使得声明如何使用**通用表达式语言（CEL）**验证自定义资源成为可能。 请参考**验证规则**指导。

- 引入KMS v2 API

  在Kubernetes 1.25版本，引入KMS v2 alpha1 API以提升性能，实现轮替与可观察性改进。 此API使用AES-GCM替代了AES-CBC，通过DEK实现静态数据加密（Kubernetes Secrets），此过程中无需您额外操作，且支持通过AES-GCM和AES-CBC进行读取。 更多信息，请参考**使用 KMS provider进行数据加密指南**。

- Pod新增网络就绪状况

  Kubernetes 1.25引入了对kubelet所管理的新的Pod状况PodHasNetwork的Alpha支持， 该状况位于Pod的status字段中 。详情请参见**Pod网络就绪**。

- 应用滚动上线所用的两个特性进入稳定阶段
  - 在Kubernetes 1.25版本，StatefulSet的minReadySeconds进入稳定阶段，允许每个Pod等待一段预期时间来减缓StatefulSet的滚动上线。更多信息，请参见**最短就绪秒数**。
  - 在Kubernetes 1.25版本，DaemonSet的maxSurge进入稳定阶段，允许DaemonSet工作负载在滚动上线期间在一个节点上运行同一 Pod的多个实例，有助于将DaemonSet的停机时间降到最低。DaemonSet不允许maxSurge和hostPort同时使用，因为两个活跃的Pod无法共享同一节点的相同端口。更多信息，请参见**DaemonSet工作负载滚动上线**。
- 对使用用户命名空间运行Pod提供Alpha支持

  对使用user namespace运行Pod提供alpha支持，将Pod内的root用户映射到容器外的非零ID，使得从容器角度看是root身份运行，而从主机角度看是常规的非特权用户。目前尚处于内测阶段，需要开启特性门控UserNamespacesStatelessPodsSupport， 且要求容器运行时必须能够支持此功能。更多信息，请参见**对使用user namespace运行Pod提供alpha支持**。

Kubernetes 1.24版本

- 从kubelet中删除 Dockershim

  Dockershim自1.20版本被标废弃以来，在1.24版本正式从Kubelet代码中移除。如果还想使用Docker作为容器运行时的话，需要切换到cri-dockerd，或者使用其他支持CRI的运行时比如Containerd/CRI-O等。

  说明

  您需要注意排查是否有agent或者应用强依赖Docker Engine的，比如在代码中使用docker ps，docker run， docker inspect等，需要注意兼容多种runtime，以及切换到标准cri接口。
- Beta APIs默认关闭

  在社区移除一些长期Beta API的过程中发现，90%的集群管理员并没有关心Beta API默认开始，其实Beta特性是不推荐在生产环境中使用，但是因为默认的打开策略，导致这些API在生产环境中都被默认开启，这样会因为Beta特性的bug带来一些风险，以及升级的迁移的风险。所以在1.24版本开始，Beta API默认关闭，之前已经默认开启的Beta API会保持默认开启。
- 支持OpenAPI v3

  在Kubernetes 1.24版本后，OpenAPI V3默认开启。
- 存储容量跟踪特性进入稳定阶段

  在Kubernetes 1.24版本后，CSIStorageCapacity API支持显示当前可用的存储大小，确保Pod调度到足够存储容量的节点上，减少Volumes创建和挂载失败导致的Pod调度延迟，详细信息请参见**存储容量**。
- gRPC 探针升级到Beta阶段

  在Kubernetes 1.24版本后，gRPC探针进入Beta，默认可用特性门控参数GRPCContainerProbe，使用方式请参见**配置探针**。
- 特性门控LegacyServiceAccountTokenNoAutoGeneration默认启用

  LegacyServiceAccountTokenNoAutoGeneration特性门控进入beta状态，默认为开启状态，开启后将不再为Service Account自动生成Secret Token。如果需要使用永不过期的Token，需要自己新建Secrets并挂载，详情请参见**服务账号令牌Secret**。
- 避免 IP 分配给服务的冲突

Kubernetes 1.24引入了一项新功能，允许**为服务的静态IP地址分配软保留范围**。通过手动启用此功能，集群将从服务IP地址池中自动分配IP，从而降低冲突风险。

- 基于Go 1.18编译

  在Kubernetes 1.24版本后，Kubernetes基于Go 1.18编译，默认不再支持SHA-1哈希算法验证证书签名，例如SHA1WithRSA、ECDSAWithSHA1算法，推荐使用SHA256算法生成的证书进行认证。

- StatefulSet支持设置最大不可用副本数

  在Kubernetes 1.24版本后，StatefulSets支持可配置maxUnavailable参数，使得滚动更新时可以更快地停止Pods。

- 节点非体面关闭进入Alpha阶段

  在Kubernetes 1.24中，节点非体面关闭特性是Alpha版。当kubelet的节点关闭管理器可以检测到即将到来的节点关闭操作时，节点关闭才被认为是体面的。详情请参见**处理节点非体面关闭**。

弃用和移除

Kubernetes 1.25版本

- 清理iptables链的所有权

  Kubernetes通常创建iptables链来确保这些网络数据包到达， 这些iptables链及其名称属于Kubernetes内部实现的细节，仅供内部使用场景，目前有些组件依赖于这些内部实现细节，Kubernetes总体上不希望支持某些工具依赖这些内部实现细节。详细信息，请参见**Kubernetes的iptables链不是API**。

  在Kubernetes 1.25版本后，Kubelet通过IPTablesCleanup特性门控分阶段完成迁移，是为了不在NAT表中创建iptables链，例如KUBE-MARK-DROP、KUBE-MARK-MASQ、KUBE-POSTROUTING。关于清理iptables链所有权的信息，请参见**清理IPTables链的所有权**。

- 存储驱动的弃用和移除，移除云服务厂商的in-tree卷驱动。

Kubernetes 1.24版本

- 在Kubernetes 1.24版本后，Service.Spec.LoadBalancerIP被弃用，因为它无法用于双栈协议。请使用自定义annotation。
- 在Kubernetes 1.24版本后，kube-apiserver移除参数--address、--insecure-bind-address、--port、--insecure-port=0。
- 在Kubernetes 1.24版本后，kube-controller-manager和kube-scheduler移除启动参数--port=0和--address。
- 在Kubernetes 1.24版本后，kube-apiserver --audit-log-version和--audit-webhook-version仅支持audit.k8s.io/v1，Kubernetes 1.24移除audit.k8s.io/v1[alpha|beta]1，只能使用audit.k8s.io/v1。
- 在Kubernetes 1.24版本后，kubelet移除启动参数--network-plugin，仅当容器运行环境设置为Docker时，此特定于Docker的参数才有效，并会随着Dockershim一起删除。
- 在Kubernetes 1.24版本后，动态日志清理功能已经被废弃，并在Kubernetes 1.24版本移除。该功能引入了一个日志过滤器，可以应用于所有Kubernetes系统组件的日志，以防止各种类型的敏感信息通过日志泄漏。此功能可能导致日志阻塞，所以废弃，更多信息请参见**Dynamic log sanitization**和 **KEP-1753**。
- VolumeSnapshot v1beta1 CRD在Kubernetes 1.20版本中被废弃，在Kubernetes 1.24版本中移除，需改用v1版本。

- 在Kubernetes 1.24版本后，移除自1.11版本就废弃的service annotation tolerate-unready-endpoints，使用Service.spec.publishNotReadyAddresses代替。
- 在Kubernetes 1.24版本后，废弃metadata.clusterName字段，并将在下一个版本中删除。
- Kubernetes 1.24及以后的版本，去除了kube-proxy监听NodePort的逻辑，在NodePort与内核net.ipv4.ip_local_port_range范围有冲突的情况下，可能会导致偶发的TCP无法连接的情况，导致健康检查失败、业务异常等问题。升级前，请确保集群没有NodePort端口与任意节点net.ipv4.ip_local_port_range范围存在冲突。更多信息，请参见**Kubernetes社区PR**。

CCE 对 Kubernetes 1.25 版本的增强

在版本维护周期中，CCE会对Kubernetes 1.25版本进行定期的更新，并提供功能增强。

关于CCE集群版本的更新说明，请参见**CCE集群版本发布说明**。

参考链接

关于Kubernetes 1.25与其他版本的性能对比和功能演进的更多信息，请参考：

- **Kubernetes v1.25 Release Notes**
- **Kubernetes v1.24 Release Notes**

6.1.2.3 Kubernetes 1.23 版本说明

云容器引擎（CCE）严格遵循社区一致性认证。本文介绍CCE发布Kubernetes 1.23版本所做的变更说明。

资源变更与弃用

社区1.23 ReleaseNotes

- FlexVolume废弃，建议使用CSI。
- HorizontalPodAutoscaler v2版本GA，HorizontalPodAutoscaler API v2在1.23版本中逐渐稳定。不建议使用HorizontalPodAutoscaler v2beta2 API，建议使用新的v2版本API。
- **PodSecurity**支持beta，PodSecurity替代废弃的PodSecurityPolicy，PodSecurity是一个准入控制器，它根据设置实施级别的特定命名空间标签在命名空间中的Pod上实施Pod安全标准。在1.23中PodSecurity默认启用。

社区1.22 ReleaseNotes

- Ingress资源不再支持networking.k8s.io/v1beta1和extensions/v1beta1 API。如果使用旧版本API管理Ingress，会影响应用对外暴露服务，请尽快使用networking.k8s.io/v1替代。
- CustomResourceDefinition资源不再支持apiextensions.k8s.io/v1beta1 API。如果使用旧版本API创建自定义资源定义，会导致定义创建失败，进而影响调和（reconcile）该自定资源的控制器，请尽快使用apiextensions.k8s.io/v1替代。
- ClusterRole、ClusterRoleBinding、Role和RoleBinding资源不再支持rbac.authorization.k8s.io/v1beta1 API。如果使用旧版本API管理RBAC资源，会

影响应用的权限服务，甚至无法在集群内正常使用，请尽快使用rbac.authorization.k8s.io/v1替代。

- Kubernetes版本发布周期由一年4个版本变为一年3个版本。
- StatefulSets 支持minReadySeconds。
- 缩容时默认根据Pod uid排序随机选择删除Pod（LogarithmicScaleDown）。基于该特性，可以增强Pod被缩容的随机性，缓解由于Pod拓扑分布约束带来的问题。更多信息，请参见**KEP-2185**和**issues 96748**。
- **BoundServiceAccountTokenVolume**特性已稳定，该特性能够提升服务账号（ServiceAccount）Token的安全性，改变了Pod挂载Token的方式，Kubernetes 1.21及以上版本的集群中会默认开启。

参考链接

关于Kubernetes 1.23与其他版本的性能对比和功能演进的更多信息，请参考：

- **Kubernetes v1.23 Release Notes**
- **Kubernetes v1.22 Release Notes**

6.1.2.4 Kubernetes 1.21 版本说明

云容器引擎（CCE）严格遵循社区一致性认证。本文介绍CCE发布Kubernetes 1.21版本所做的变更说明。

资源变更与弃用

社区1.21 ReleaseNotes

- CronJob现在已达到稳定状态，版本号变为batch/v1。
- 不可变的Secret和ConfigMap现在已升级到稳定状态。向这些对象添加了一个新的不可变字段，以拒绝更改。此拒绝可保护集群免受可能无意中中断应用程序的更新。因为这些资源是不可变的，kubelet不会监视或轮询更改。这减少了kube-apiserver的负载，提高了可扩展性和性能。更多信息，请参见**Immutable ConfigMaps**。
- 优雅节点关闭现在已升级到测试状态。通过此更新，kubelet可以感知节点关闭，并可以优雅地终止该节点的Pod。在此更新之前，当节点关闭时，其Pod没有遵循预期的终止生命周期，这导致了工作负载问题。现在kubelet可以通过systemd检测即将关闭的系统，并通知正在运行的Pod，使它们优雅地终止。
- 具有多个容器的Pod现在可以使用kubectl.kubernetes.io/默认容器注释为kubectl命令预选容器。
- PodSecurityPolicy废弃，详情请参见**https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/**。
- **BoundServiceAccountTokenVolume**特性进入Beta，该特性能够提升服务账号（ServiceAccount）Token的安全性，改变了Pod挂载Token的方式，Kubernetes 1.21及以上版本的集群中会默认开启。

社区1.20 ReleaseNotes

- API优先级和公平性已达到测试状态，默认启用。这允许kube-apiserver按优先级对传入请求进行分类。更多信息，请参见**API Priority and Fairness**。

- 修复 exec probe timeouts不生效的BUG，在此修复之前，exec 探测器不考虑 timeoutSeconds 字段。相反，探测将无限期运行，甚至超过其配置的截止日期，直到返回结果。 通过此更改，如果未指定值，将使用默认值，默认值为1秒。如果探测时间超过一秒，可能会导致应用健康检查失败。请再升级时确定使用该特性的应用更新timeoutSeconds字段。新引入的 ExecProbeTimeout 特性门控所提供的修复使集群操作员能够恢复到以前的行为，但这种行为将在后续版本中锁定并删除。
- RuntimeClass已达到稳定状态。RuntimeClass资源提供了一种机制，用于支持集群中的多个运行时，并将有关该容器运行时的信息公开到控制平面。
- kubectl调试已达到测试状态。kubectl调试直接从kubectl提供对常见调试工作流的支持。
- Dockershim在1.20被标记为废弃，目前您可以继续在集群中使用Docker。该变动与集群所使用的容器镜像（Image）无关。您依然可以使用Docker构建您的镜像。更多信息，请参见**Dockershim Deprecation FAQ**。

参考链接

关于Kubernetes 1.21与其他版本的性能对比和功能演进的更多信息，请参考：

- **Kubernetes v1.21 Release Notes**
- **Kubernetes v1.20 Release Notes**

6.1.2.5 Kubernetes 1.19 版本说明

云容器引擎（CCE）严格遵循社区一致性认证。本文介绍CCE发布Kubernetes 1.19版本所做的变更说明。

资源变更与弃用

社区1.19 ReleaseNotes

- 增加对vSphere in-tree卷迁移至vSphere CSI驱动的支持。in-tree vSphere Volume插件将不再使用，并在将来的版本中删除。
- apiextensions.k8s.io/v1beta1已弃用，推荐使用apiextensions.k8s.io/v1。
- apiregistration.k8s.io/v1beta1已弃用，推荐使用apiregistration.k8s.io/v1。
- authentication.k8s.io/v1beta1、authorization.k8s.io/v1beta1已弃用，1.22将移除，推荐使用authentication.k8s.io/v1、authorization.k8s.io/v1。
- autoscaling/v2beta1已弃用，推荐使用autoscaling/v2beta2。
- coordination.k8s.io/v1beta1在1.19中已弃用，1.22将移除，推荐使用v1。
- Kube-apiserver: componentstatus API已弃用。
- Kubeadm：kubeadm config view命令已被弃用，并将在未来版本中删除，请使用kubectl get cm -o yaml -n kube-system kubeadm-config来直接获取kubeadm配置。
- Kubeadm：弃用kubeadm alpha kubelet config enable-dynamic命令。
- Kubeadm：kubeadm alpha certs renew命令--use-api参数已弃用。
- Kubernetes不再支持构建hyperkube镜像。
- Remove --export flag from kubectl get command - kubectl get中移除 --export参数。

- alpha特性“ResourceLimitsPriorityFunction”已完全删除。
- storage.k8s.io/v1beta1已弃用，推荐使用storage.k8s.io/v1。

社区1.18 ReleaseNotes

- kube-apiserver
  - apps/v1beta1 and apps/v1beta2下所有资源不再提供服务，使用apps/v1替代。
  - extensions/v1beta1下daemonsets，deployments，replicasets不再提供服务，使用apps/v1替代。
  - extensions/v1beta1下networkpolicies不再提供服务，使用networking.k8s.io/v1替代。
  - extensions/v1beta1下podsecuritypolicies不再提供服务，使用policy/v1beta1替代。
- kubelet
  - --redirect-container-streaming不推荐使用，v1.20会正式废弃。
  - 资源度量端点 /metrics/resource/v1alpha1以及此端点下的所有度量标准均已弃用。请转换为端点 /metrics/resource下的度量标准：
    - scrape_error --> scrape_error
    - node_cpu_usage_seconds_total --> node_cpu_usage_seconds
    - node_memory_working_set_bytes --> node_memory_working_set_bytes
    - container_cpu_usage_seconds_total --> container_cpu_usage_seconds
    - container_memory_working_set_bytes --> container_memory_working_set_bytes
    - scrape_error --> scrape_error
  - 在将来的发行版中，kubelet将不再根据CSI规范创建CSI NodePublishVolume目标目录。可能需要相应地更新CSI驱动程序，以正确创建和处理目标路径。
- kube-proxy
  - --healthz-port和--metrics-port参数不建议使用，请使用--healthz-bind-address和--metrics-bind-address。
  - 增加EndpointSliceProxying功能选项以控制kube-proxy中EndpointSlices的使用，默认情况下已禁用此功能。
- kubeadm
  - kubeadm upgrade node的--kubelet-version参数已弃用，将在后续版本中删除。
  - kubeadm alpha certs renew命令中--use-api参数已弃用。
  - kube-dns已弃用，在将来的版本中将不再受支持。
  - kubeadm-config ConfigMap中存在的ClusterStatus结构体已废弃，将在后续版本中删除。
- kubectl
  - --dry-run不建议使用boolean和unset values，新版本中server|client|none会被使用。

  - kubectl apply --server-dry-run已弃用，替换为--dry-run=server。
- add-ons

删除cluster-monitoring插件。

- kube-scheduler
  - scheduling_duration_seconds指标已弃用。
  - scheduling_algorithm_predicate_evaluation_seconds和scheduling_algorithm_priority_evaluation_seconds指标已弃用，使用framework_extension_point_duration_seconds[extension_point="Filter"]和framework_extension_point_duration_seconds[extension_point="Score"]替代。
  - 调度器策略AlwaysCheckAllPredicates已弃用。
- 其他变化
  - k8s.io/node-api组件不再更新。作为替代，可以使用位于k8s.io/api中的RuntimeClass类型和位于k8s.io/client-go中的generated clients。
  - 已从apiserver_request_total中删除“client”标签。

参考链接

关于Kubernetes 1.19与其他版本的性能对比和功能演进的更多信息，请参考：

- **Kubernetes v1.19.0 Release Notes**
- **Kubernetes v1.18.0 Release Notes**

6.1.2.6 （停止维护）Kubernetes 1.17 版本说明

云容器引擎（CCE）严格遵循社区一致性认证。本文介绍CCE发布Kubernetes 1.17版本所做的变更说明。

资源变更与弃用

- apps/v1beta1和apps/v1beta2下所有资源不再提供服务，使用apps/v1替代。
- extensions/v1beta1下daemonsets、deployments、replicasets不再提供服务，使用apps/v1替代。
- extensions/v1beta1下networkpolicies不再提供服务，使用networking.k8s.io/v1替代。
- extensions/v1beta1下podsecuritypolicies不再提供服务，使用policy/v1beta1替代。
- extensions/v1beta1 ingress v1.20版本不再提供服务，当前可使用networking.k8s.io/v1beta1。
- scheduling.k8s.io/v1beta1 and scheduling.k8s.io/v1alpha1下的PriorityClass计划在1.17不再提供服务，迁移至scheduling.k8s.io/v1。
- events.k8s.io/v1beta1中event series.state字段已废弃，将在1.18版本中移除。
- apiextensions.k8s.io/v1beta1下CustomResourceDefinition已废弃，将再1.19不在提供服务，使用apiextensions.k8s.io/v1。
- admissionregistration.k8s.io/v1beta1 MutatingWebhookConfiguration和ValidatingWebhookConfiguration已废弃，将在1.19不在提供服务，使用admissionregistration.k8s.io/v1替换。

- rbac.authorization.k8s.io/v1alpha1 and rbac.authorization.k8s.io/v1beta1被废弃，使用rbac.authorization.k8s.io/v1替代，v1.20会正式停止服务。
- storage.k8s.io/v1beta1 CSINode object废弃并会在未来版本中移除。

其他废弃和移除

- 移除OutOfDisk node condition，改为使用DiskPressure。
- scheduler.alpha.kubernetes.io/critical-pod annotation已被移除，如需要改为设置priorityClassName。
- beta.kubernetes.io/os和beta.kubernetes.io/arch在1.14版本中已经废弃，计划在1.18版本中移除。
- 禁止通过--node-labels设置kubernetes.io和k8s.io为前缀的标签，老版本中kubernetes.io/availablezone该label在1.17中移除，整改为failure-domain.beta.kubernetes.io/zone获取AZ信息。
- beta.kubernetes.io/instance-type被废弃，使用node.kubernetes.io/instance-type替代。
- 移除{kubelet_root_dir}/plugins路径。
- 移除内置集群角色system:csi-external-provisioner和system:csi-external-attacher。

参考链接

关于Kubernetes 1.17与其他版本的性能对比和功能演进的更多信息，请参考：

- **Kubernetes v1.17.0 Release Notes**
- **Kubernetes v1.16.0 Release Notes**

6.1.3 CCE 集群版本发布说明

v1.27 版本

须知

CCE从v1.27版本开始，集群的节点均仅支持Containerd容器引擎。

表 6-2 v1.27 补丁版本发布说明

CCE集群补丁版本号	Kubernetes社区版本	特性更新	优化增强	安全漏洞修复
v1.27.1-r0	v1.27.2	首次发布CCE v1.27集群，有关更多信息请参见Kubernetes 1.27版本说明。 • 节点池配置管理支持软驱逐和硬驱逐的设置。	-	-

v1.25 版本

须知

除EulerOS 2.5操作系统外，CCE v1.25集群的节点均默认采用Containerd容器引擎。

表 6-3 v1.25 补丁版本发布说明

CCE集群补丁版本号	Kubernetes社区版本	特性更新	优化增强	安全漏洞修复
v1.25.1-r0	v1.25.5	首次发布CCE v1.25集群，有关更多信息请参见Kubernetes 1.25版本说明。	-	-

v1.23 版本

表 6-4 v1.23 补丁版本发布说明

CCE集群补丁版本号	Kubernetes社区版本	特性更新	优化增强	安全漏洞修复
v1.23.8-r0	v1.23.11	-	• 增强docker版本升级时的可靠性。 • 优化集群节点时间同步能力。	修复部分安全问题。
v1.23.5-r0	v1.23.11	• 支持GPU节点的设备故障检测和隔离能力。 • 支持配置集群维度的自定义安全组。 • CCE集群支持选择Containerd容器运行时。	• 优化升级控制节点ETCD版本至社区版本3.5.6。 • 优化调度均衡性，工作负载实例数缩容时仍保持跨AZ分布均衡。 • 优化kube-apiserver在频繁更新CRD场景下的内存使用。	修复部分安全问题及以下CVE漏洞： • CVE-2022-3294 • CVE-2022-3162 • CVE-2022-3172 • CVE-2021-25749
v1.23.1-r0	v1.23.4	首次发布CCE v1.23集群，有关更多信息请参见Kubernetes 1.23版本说明。	-	-

v1.21 版本

表 6-5 v1.21 补丁版本发布说明

CCE集群补丁版本号	Kubernetes社区版本	特性更新	优化增强	安全漏洞修复
v1.21.10-r0	v1.21.14	-	• 增强docker版本升级时的可靠性。 • 优化集群节点时间同步能力。 • 优化节点重启后，docker运行时拉取镜像的稳定性。	修复部分安全问题。
v1.21.7-r0	v1.21.14	• 支持GPU节点的设备故障检测和隔离能力。 • 支持配置集群维度的自定义安全组。	优化ELB Service/Ingress在大量连接场景下的稳定性。	修复部分安全问题及以下CVE漏洞： • CVE-2022-3294 • CVE-2022-3162 • CVE-2022-3172
v1.21.1-r0	v1.21.7	首次发布CCE v1.21集群，有关更多信息请参见Kubernetes 1.21版本说明。	-	-

v1.19 版本

表 6-6 v1.19 补丁版本发布说明

CCE集群补丁版本号	Kubernetes社区版本	特性更新	优化增强	安全漏洞修复
v1.19.16-r20	v1.19.16	-	• 增强节点重启后，docker运行时拉取镜像的稳定性。	修复部分安全问题。
v1.19.16-r4	v1.19.16	• 支持GPU节点的设备故障检测和隔离能力。 • 支持配置集群维度的自定义安全组。	• 优化节点污点场景下负载调度的性能。 • 增强Containerd运行时绑核场景下长时间运行的稳定性。 • 优化ELB Service/Ingress在大量连接场景下的稳定性。 • 优化kube-apiserver在频繁更新crd场景下的内存使用。	修复部分安全问题及以下CVE漏洞： • CVE-2022-3294 • CVE-2022-3162 • CVE-2022-3172
v1.19.16-r0	v1.19.16	-	增强工作负载升级且节点伸缩状态下，负载均衡服务更新的稳定性。	修复部分安全问题及以下CVE漏洞： • CVE-2021-25741 • CVE-2021-25737
v1.19.10-r0	v1.19.10	首次发布CCE v1.19集群，有关更多信息请参见Kubernetes 1.19版本说明。	-	-

6.2 创建集群

6.2.1 CCE Turbo 集群与 CCE 集群的区别

CCE Turbo 集群与 CCE 集群对比

CCE支持多种类型的集群创建，以满足您各种业务需求，如下为CCE Turbo集群与CCE集群区别：

表 6-7 集群类型对比

维度	子维度	CCE Turbo集群	CCE集群
集群	定位	面向云原生2.0的新一代容器集群产品，计算、网络、调度全面加速	标准版本集群，提供商用级的容器集群服务
	节点形态	支持虚拟机	支持虚拟机
网络	网络模型	**云原生网络2.0**：面向大规模和高性能的场景。 组网规模最大支持2000节点	**云原生网络1.0**：面向性能和规模要求不高的场景。 ● 容器隧道网络模式 ● VPC网络模式
	网络性能	VPC网络和容器网络融合，性能无损耗	VPC网络叠加容器网络，性能有一定损耗
	容器网络隔离	Pod可直接关联安全组，基于安全组的隔离策略，支持集群内外部统一的安全隔离。	● 隧道网络模式：集群内部网络隔离策略，支持Networkpolicy。 ● VPC网络模式：不支持
安全	隔离性	● 物理机：安全容器，支持虚机级别的隔离 ● 虚拟机：普通容器，Cgroups隔离	普通容器，Cgroups隔离

6.2.2 创建集群

您可以通过云容器引擎控制台非常方便快速的创建Kubernetes集群。创建完成后，集群控制节点将由云容器引擎服务托管，您只需创建工作节点，帮助您降低集群运维成本，可实现简单高效的业务部署。

约束与限制

- 创建节点过程中会使用域名方式从OBS下载软件包，需要能够使用云上内网DNS解析OBS域名，否则会导致创建不成功。为此，节点所在子网的DNS服务器地址需要配置为内网DNS，从而使得节点使用内网DNS。在创建子网时DNS默认配置为内网DNS，如果您修改过子网的DNS，请务必**确保子网下的DNS服务器可以解析OBS服务域名**，否则需要将DNS改成内网DNS。
- 单Region下单用户可创建的集群总数限制为50个。

- 集群一旦创建以后，不支持变更以下项：
  - 变更集群类型。
  - 变更集群的控制节点数量。
  - 变更控制节点可用区。
  - 变更集群的网络配置，如所在的虚拟私有云VPC、子网、容器网段、服务网段、kube-proxy代理模式（即**服务转发模式**）。
  - 变更网络模型，例如“容器隧道网络”更换为“VPC网络”。

操作步骤

步骤1 登录CCE控制台。

步骤2 在“集群管理”页面选择需要创建的集群类型，单击“创建”。

步骤3 填写集群参数。

基础配置

- 集群名称：请输入集群名称，同一账户下集群不可重名。
- 企业项目：

  该参数仅对开通企业项目的企业客户账号显示。

  选择某企业项目（如：default）后，集群、集群下节点、集群安全组、节点安全组和自动创建的节点EIP（弹性公网IP）将创建到所选企业项目下。为方便管理资源，在集群创建成功后，建议不要修改集群下节点、集群安全组、节点安全组的企业项目。

  企业项目是一种云资源管理方式，企业项目管理服务提供统一的云资源按项目管理，以及项目内的资源管理、成员管理。
- 集群版本：选择集群使用的Kubernetes版本。
- 集群规模：集群支持管理的最大节点数量，请根据业务场景选择。
- 高可用：控制节点分布方式，默认随机分配，控制节点尽可能随机分布在不同可用区以提高容灾能力 。

  您还可以展开高级配置自定义控制节点分布方式，支持如下2种方式。
  - 随机分配：通过把控制节点随机创建在不同的可用区中实现容灾。
  - 自定义：自定义选择每台控制节点的位置。
    - 主机：通过把控制节点创建在相同可用区下的不同主机中实现容灾。
    - 自定义：用户自行决定每台控制节点所在的位置。

网络配置

集群网络涉及节点、容器和服务，强烈建议您详细了解集群的网络以及容器网络模型，具体请参见**网络概述**。

- 网络模型：CCE集群支持“VPC网络”和“容器隧道网络”，CCE Turbo集群支持选择“云原生网络2.0”。详情请参见**容器网络模型对比**。
- 虚拟私有云：选择集群所在的虚拟私有云VPC，如没有可选项可以单击右侧“新建虚拟私有云”创建。创建后不可修改。
- 控制节点子网：选择控制节点（即集群Master节点）所在子网，如没有可选项可以单击右侧“新建子网”创建。创建后控制节点子网不可修改。

- 容器网段（CCE集群设置）：设置容器使用的网段，决定了集群下容器的数量上限。
- 默认容器子网（CCE Turbo集群设置）：选择容器所在子网，如没有可选项可以单击右侧“新建子网”创建。容器子网决定了集群下容器的数量上限，创建后支持新增子网。
- 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。决定了Service资源的上限。 创建后不可修改。

高级配置

- 服务转发模式：支持IPVS和iptables两种转发模式，具体请参见**iptables与IPVS如何选择**。
- CPU管理策略：支持为工作负载实例设置独占CPU核的功能，详情请参见**CPU管理策略**。
- 证书认证：
  - 系统默认：默认开启X509认证模式，X509是一种非常通用的证书格式。
  - 自定义：您可以将自定义证书添加到集群中，用自定义证书进行认证。
    您需要分别上传自己的**CA根证书**、**客户端证书**和**客户端证书私钥**。

    注意

    - 请上传**小于1MB**的文件，CA根证书和客户端证书上传格式支持**.crt**或**.cer**格式，客户端证书私钥仅支持上传**未加密的证书私钥**。
    - 客户端证书有效期需要5年以上。
    - 上传的CA根证书既给认证代理使用，也用于配置kube-apiserver聚合层，**如不合法，集群将无法成功创建**。
    - 从1.25版本集群开始，Kubernetes不再支持使用SHA1WithRSA、ECDSAWithSHA1算法生成的证书认证，推荐使用SHA256算法生成的证书进行认证。
- 集群描述：支持200个英文字符。

步骤4 单击“下一步：插件配置”，配置插件。

域名解析：

- 使用域名解析服务：自动安装**coredns**插件，可为集群提供域名解析、连接云上 DNS 服务器等能力。

容器存储：自动安装**everest**插件，可为集群提供基于 CSI 的容器存储能力，支持对接云上云硬盘等存储服务。

故障检测：默认安装**npd**插件，安装后可为集群提供节点故障检测、隔离能力，帮助您及时识别节点问题。

业务日志

- ICAgent日志采集：
  由应用运维服务 AOM 提供的日志采集器，配置采集规则后可同时将日志上报至 AOM 服务和云日志服务 LTS。

您可以根据需要选择是否采集容器标准输出日志。

过载控制：过载控制开启后，将根据控制节点的资源压力，动态调整请求并发量，维护控制节点和集群的可靠性。详情请参见**集群过载控制**。

步骤5 参数填写完成后，单击“下一步：规格确认”，显示集群资源清单，确认无误后，单击“提交”。

集群创建预计需要6-10分钟，您可以单击“返回集群管理”进行其他操作或单击“查看集群事件列表”后查看集群详情。

----结束

相关操作

- 通过命令行工具连接集群：请参见**通过kubectl连接集群**。
- 添加节点：集群创建完成后，若您需要为集群添加节点，请参见**创建节点**。

6.2.3 iptables 与 IPVS 如何选择

kube-proxy是Kubernetes集群的关键组件，负责Service和其后端容器Pod之间进行负载均衡转发。

CCE当前支持iptables和IPVS两种服务转发模式，各有优缺点。

- IPVS：吞吐更高，速度更快的转发模式。适用于集群规模较大或Service数量较多的场景。
- iptables：社区传统的kube-proxy模式。适用于Service数量较少或客户端会出现大量并发短连接的场景。当集群中超过1000个Service时，可能会出现网络延迟的情况。

约束与限制

- IPVS模式集群下，Ingress和Service使用相同ELB实例时，无法在集群内的节点和容器中访问Ingress，因为kube-proxy会在ipvs-0的网桥上挂载LB类型的Service地址，Ingress对接的ELB的流量会被ipvs-0网桥劫持。建议Ingress和Service使用不同ELB实例。
- iptables模式下，集群内部CluterIP地址无法ping通；IPVS模式下，集群内部CluterIP地址可以正常ping通。

iptables

iptables是一个Linux内核功能，提供了大量的数据包处理和过滤方面的能力。它可以在核心数据包处理管线上用Hook挂接一系列的规则。iptables模式中kube-proxy 在NAT pre-routing Hook中实现它的NAT和负载均衡功能。

kube-proxy的用法是一种O(n) 算法，其中的n随集群规模同步增长，这里的集群规模，更明确的说就是服务和后端Pod的数量。

IPVS

IPVS（IP Virtual Server）是在Netfilter上层构建的，并作为Linux内核的一部分，实现传输层负载均衡。IPVS可以将基于TCP和UDP服务的请求定向到真实服务器，并使真实服务器的服务在单个IP地址上显示为虚拟服务。

IPVS模式下，kube-proxy使用IPVS负载均衡代替了iptables。这种模式同样有效，IPVS的设计就是用来为大量服务进行负载均衡的，它有一套优化过的API，使用优化的查找算法，而不是简单的从列表中查找规则。

kube-proxy在IPVS模式下，其连接过程的复杂度为O(1)。换句话说，多数情况下，IPVS的连接处理效率是和集群规模无关的。

IPVS包含了多种不同的负载均衡算法，例如轮询、最短期望延迟、最少连接以及各种哈希方法等。而iptables就只有一种随机平等的选择算法。

IPVS相较于iptables的优势大致如下：

1. 为大型集群提供了更好的可扩展性和性能
2. 支持比iptables更好的负载均衡算法
3. 支持服务器健康检查和连接重试等功能

6.3 连接集群

6.3.1 通过 kubectl 连接集群

操作场景

本文将以CCE集群为例，介绍如何通过kubectl连接CCE集群。

权限说明

kubectl访问CCE集群是通过集群上生成的配置文件（kubeconfig.json）进行认证，kubeconfig.json文件内包含用户信息，CCE根据用户信息的权限判断kubectl有权限访问哪些Kubernetes资源。即哪个用户获取的kubeconfig.json文件，kubeconfig.json就拥有哪个用户的信息，这样使用kubectl访问时就拥有这个用户的权限。

用户拥有的权限请参见**集群权限（IAM授权）与命名空间权限（Kubernetes RBAC授权）的关系**。

使用 kubectl 连接集群

若您需要从客户端计算机连接到Kubernetes集群，可使用Kubernetes命令行客户端kubectl，您可登录CCE控制台，单击待连接集群名称，在“集群信息”页面查看访问地址以及kubectl的连接步骤。

CCE支持“内网访问”和“公网访问”两种方式访问集群。

- 内网访问：访问集群的客户端机器需要位于集群所在的同一VPC内。
- 公网访问：访问集群的客户端机器需要具备访问公网的能力，并为集群绑定公网地址。

须知

通过“公网访问”方式访问集群，您需要在集群信息页中的“连接信息”版块为集群绑定公网地址。绑定公网集群的kube-apiserver将会暴露到互联网，存在被攻击的风险，建议对kube-apiserver所在节点的EIP配置DDoS高防服务。

您需要先下载kubectl以及配置文件，拷贝到您的客户端机器，完成配置后，即可以访问Kubernetes集群。使用kubectl连接集群的步骤如下：

步骤1 下载kubectl

您需要准备一台可访问公网的客户端计算机，并通过命令行方式安装kubectl。如果已经安装kubectl，则跳过此步骤，您可执行**kubectl version**命令判断是否已安装kubectl。

本文以Linux环境为例安装和配置kubectl，详情请参考**安装kubectl**。

1. 登录到您的客户端机器，下载kubectl。
   ```
   cd /home
   curl -LO https://dl.k8s.io/release/{v1.25.0}/bin/linux/amd64/kubectl
   ```
   其中*{v1.25.0}*为指定的版本号，请根据集群版本进行替换。
2. 安装kubectl。
   ```
   chmod +x kubectl
   mv -f kubectl /usr/local/bin
   ```

步骤2 获取kubectl配置文件

在集群信息页中的“连接信息”版块，单击kubectl后的“配置”按钮，查看kubectl的连接信息，并在弹出页面中下载配置文件。

说明

- kubectl配置文件（kubeconfig.json）用于对接认证集群，请您妥善保存该认证凭据，防止文件泄露后，集群有被攻击的风险。
- 当前集群默认不开启**域名双向认证说明**，可通过 kubectl config use-context externalTLSVerify 命令开启双向认证。对已经绑定了EIP的集群，如果在使用双向认证时出现认证不通过的情况（x509: certificate is valid），需要重新绑定EIP并重新下载kubeconfig.json。
- IAM用户下载的配置文件所拥有的Kubernetes权限与CCE控制台上IAM用户所拥有的权限一致。
- 如果Linux系统里面配置了KUBECONFIG环境变量，kubectl会优先加载KUBECONFIG环境变量，而不是$home/.kube/config，使用时请注意。

步骤3 配置kubectl

以Linux环境为例配置kubectl。

1. 登录到您的客户端机器，拷贝**步骤2**中下载的配置文件（kubeconfig.json）到您客户端机器的/home目录下。
2. 配置kubectl认证文件。
   ```
   cd /home
   mkdir -p $HOME/.kube
   mv -f kubeconfig.json $HOME/.kube/config
   ```
3. 根据使用场景，切换kubectl的访问模式。
   - VPC内网接入访问请执行：
     ```
     kubectl config use-context internal
     ```
   - 互联网接入访问请执行（集群需绑定公网地址）：
     ```
     kubectl config use-context external
     ```
   - 互联网接入访问如需开启双向认证请执行（集群需绑定公网地址）：
     ```
     kubectl config use-context externalTLSVerify
     ```
     关于集群双向认证的说明请参见**域名双向认证**。

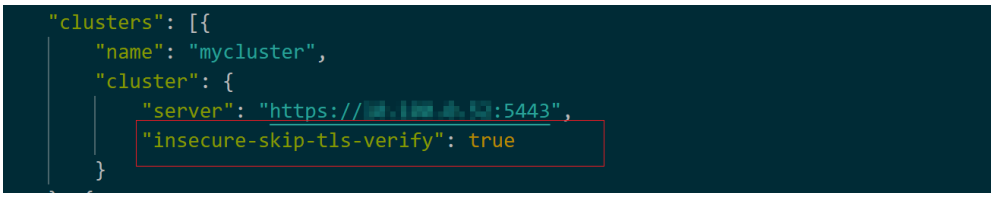
----结束

域名双向认证

CCE当前支持域名双向认证。

- 域名双向认证默认不开启，可通过 **kubectl config use-context externalTLSVerify** 命令切换到externalTLSVerify这个context开启使用。
- 集群绑定或解绑弹性IP、配置或更新自定义域名时，集群服务端证书将同步签入最新的集群访问地址（包括集群绑定的弹性IP、集群配置的所有自定义域名）。
- 异步同步集群通常耗时约5-10min，同步结果可以在操作记录中查看“同步证书”。
- 对已经绑定了EIP的集群，如果在使用双向认证时出现认证不通过的情况（x509: certificate is valid），需要重新绑定EIP并重新下载kubeconfig.json。
- 早期未支持域名双向认证时，kubeconfig.json中包含"insecure-skip-tls-verify": true字段，如图6-1所示。如果需要使用双向认证，您可以重新下载kubeconfig.json文件并配置开启域名双向认证。

图 6-1 未开启域名双向认证



常见问题

- **Error from server Forbidden**

  使用kubectl在创建或查询Kubernetes资源时，显示如下内容：

  ```
  # kubectl get deploy Error from server (Forbidden): deployments.apps is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "deployments" in API group "apps" in the namespace "default"
  ```

  原因是用户没有操作该Kubernetes资源的权限，请参见**命名空间权限（Kubernetes RBAC授权）**为用户授权。

- **The connection to the server localhost:8080 was refused**

  使用kubectl在创建或查询Kubernetes资源时，显示如下内容：

  ```
  The connection to the server localhost:8080 was refused - did you specify the right host or port?
  ```

  原因是由于该kubectl客户端未配置集群认证，请参见**步骤3**进行配置。

6.3.2 通过 X509 证书连接集群

操作场景

通过控制台获取集群证书，使用该证书可以访问Kubernetes集群。

操作步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“集群信息”，在右边“连接信息”下证书认证一栏，单击“下载”。

步骤3 在弹出的“证书获取”窗口中，根据系统提示选择证书的过期时间并下载集群X509证书。

须知

- 下载的证书包含client.key、client.crt、ca.crt三个文件，请妥善保管您的证书，不要泄露。
- 集群中容器之间互访不需要证书。

步骤4 使用集群证书调用Kubernetes原生API。

例如使用curl命令调用接口查看Pod信息，如下所示，其中*192.168.***.***:5443*为集群API Server地址。

```
curl --cacert ./ca.crt --cert ./client.crt --key ./client.key  https://192.168.***.***:5443/api/v1/namespaces/default/pods/
```

更多集群接口请参见**Kubernetes API**。

----结束

6.3.3 通过自定义域名访问集群

操作场景

集群服务端证书中签入的**主题备用名称（Subject Alternative Name，缩写SAN）**。SAN通常在TLS握手阶段被用于客户端校验服务端的合法性：服务端证书是否被客户端信任的CA所签发，且证书中的SAN是否与客户端实际访问的IP地址或DNS域名匹配。

当客户端无法直接访问集群内网私有IP地址或者公网弹性IP地址时，您可以将客户端可直接访问的IP地址或者DNS域名签入集群服务端证书，以支持客户端开启双向认证，提高安全性。典型场景例如DNAT访问、域名访问等特殊场景。

域名访问场景的典型使用方式如下：

- 客户端配置Host域名指定DNS域名地址，或者客户端主机配置/etc/hosts，添加响应域名映射。
- 云上内网使用，云解析服务DNS支持配置集群弹性IP与自定义域名的映射关系。后续更新弹性IP可以继续使用双向认证+自定义域名访问集群，无需重新下载kubeconfig.json配置文件。
- 自建DNS服务器，自行添加A记录。

约束与限制

仅支持1.19及以上版本集群。

添加自定义 SAN

步骤1 登录CCE控制台。

步骤2 在集群列表中单击集群，进入集群详情页。

步骤3 在连接信息的自定义SAN处单击，在弹出的窗口中添加IP地址或域名，然后单击“保存”。

说明

1. 当前操作将会短暂重启kube-apiserver并更新kubeconfig.json文件，请避免在此期间操作集群。
2. 请输入域名或IP，以英文逗号（,）分隔，最多128个。
3. 自定义域名如需绑定弹性公网，请确保已配置公网地址。

----结束

6.4 升级集群

6.4.1 升级概述

为了能够方便您使用稳定又可靠的Kubernetes版本，请您务必在维护周期结束之前升级您的Kubernetes集群。

CCE在发布Kubernetes最新版本后，会对该版本所做的变更进行说明。

您可以通过云容器引擎管理控制台，可视化升级集群的Kubernetes版本。

升级前，您需要在集群列表页面确认集群的Kubernetes版本，以及当前是否有新的版本可供升级。

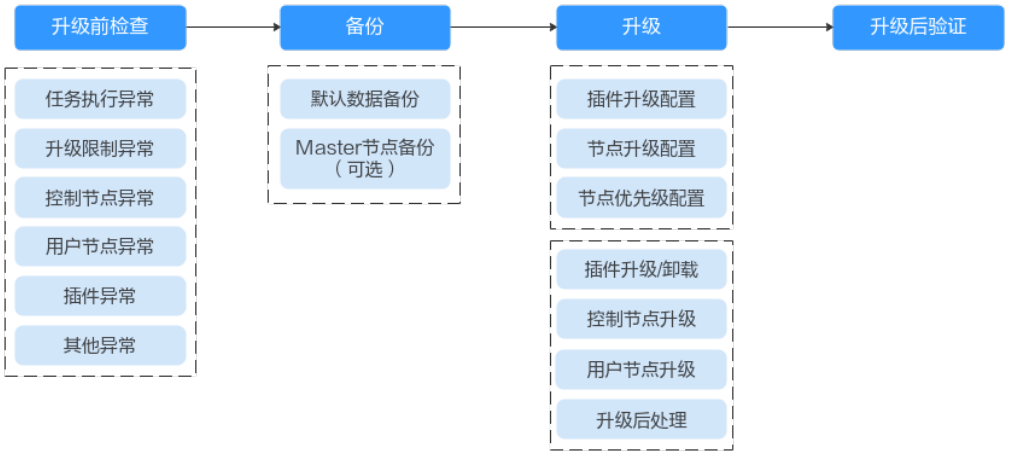
确认方法如下：

登录CCE控制台，查看待升级集群左下角是否存在“存在更新版本”提示，若存在则该集群支持升级，您可通过**CCE集群版本发布说明**查看最新版本的特性说明。若不存在，则表示该集群暂时没有可升级版本。

集群升级流程

集群升级流程包括升级前检查、备份、升级和升级后验证几个步骤，下面介绍集群升级过程中的相关流程。

图 6-2 集群升级流程



在确定集群的目标版本后，请您仔细阅读升级**注意事项**，避免升级时出现功能不兼容的问题。

1. **升级前检查**

   升级集群前，CCE会对集群内的节点、插件、工作负载兼容性等多方面进行检查，尽可能避免升级失败。如有检查异常项，请按控制台提示修复风险项。

2. **备份**

   升级时默认会对集群数据进行备份，您也可以根据需求进行Master节点整机备份，整机备份过程会使用云备份服务，将耗时约20分钟。

3. **升级**

   执行升级时，需要对升级参数进行配置，例如插件、节点滚动升级步长等。升级参数配置完成后，将进入正式升级流程，对插件、节点逐一进行升级。

4. **升级后验证**

   升级完成后，您需要手动进行业务验证，确保升级过程中对业务未造成影响。

集群升级路径

如下表格详细介绍了各集群版本能够升级到的目标版本，以及升级方式。

表 6-8 集群升级路径

源版本	目标版本	升级方式
v1.25	v1.27	原地升级
v1.23	v1.25	原地升级
v1.21	v1.25 v1.23	原地升级
v1.19	v1.23 v1.21	原地升级
v1.17	v1.19	原地升级
v1.15	v1.19	原地升级
v1.13	v1.15	滚动升级

升级方式

不同升级方式的优缺点如下：

表 6-9 升级方式的区别和优缺点

升级名称	方式	优点	缺点
原地升级	节点上升级Kubernetes组件、网络组件和CCE管理组件，升级过程中业务Pod和网络均不受影响；升级过程中，存量节点将全部打上SchedulingDisabled标签，升级完成后，用户能够正常使用存量节点。	用户无需迁移业务，可以基本上保证业务不断。	原地升级不会升级节点的操作系统，如果希望升级操作系统，可以在节点升级完成后，清空对应的节点数据，并执行节点重置，升级到新版本的操作系统。
滚动升级	节点上只升级Kubernetes组件以及网络部分组件，存量节点将全部打上SchedulingDisabled标签，仅保证原来运行的应用不受影响。 **须知** • **升级完成后，用户还需手动新建节点，并逐步释放老节点**，将应用逐步迁移到新节点上，用户控制升级步骤。	可以基本上保证业务不断。	• **升级完成后，用户需手动新建节点，并逐步释放老节点，将业务迁移至新节点。**该过程中，新建节点会存在额外的费用。待业务迁移至新节点后，老节点可以删除。 • 滚动升级完成后，如果需要继续向高版本升级，需要先完成老节点的重置，否则无法通过升级前检查。该过程可能会引起业务中断。

6.4.2 升级前须知

升级前，您可以在CCE控制台确认您的集群是否可以进行升级操作。确认方法请参见**升级概述**。

注意事项

升级集群前，您需要知晓以下事项：

- **集群升级操作不可回退，请务必慎重并选择合适的时间段进行升级，以减少升级对您的业务带来的影响。**为了您的数据安全，强烈建议您先备份数据然后再升级。
- 集群升级前，**请确认集群中未执行高危操作**，否则可能导致集群升级失败或升级后配置丢失。例如，常见的高危操作有本地修改集群节点的配置、通过ELB控制台修改CCE管理的监听器配置等。建议您通过CCE控制台修改相关配置，以便在升级时自动继承。

- 集群升级前，请查看集群状态是否均为健康状态。
- 集群升级前，请参考**Kubernetes版本发布说明**了解每个集群版本发布的特性以及差异，否则可能因为应用不兼容新集群版本而导致升级后异常。例如，您需要检查集群中是否使用了目标版本废弃的API，否则可能导致升级后调用接口失败，详情请参见**废弃API说明**。

集群升级时，以下几点注意事项可能会对您的业务存在影响，请您关注：

- 集群升级过程中，不建议对集群进行任何操作。尤其是，**请勿关机、重启或删除节点**，否则会导致升级失败。
- 集群升级过程中，已运行工作负载业务不会中断，但API Server访问会短暂中断，如果业务需要访问API Server可能会受到影响。
- 集群升级过程中，将会给节点打上“node.kubernetes.io/upgrade”（效果为“NoSchedule”）的污点，并在集群升级完成后移除该污点。请您不要在节点上手动添加相同键名的污点，即使污点效果不同，也可能会在升级后被系统误删除。

约束与限制

- 当前支持虚拟机节点的CCE集群升级。
- 当集群中存在使用私有镜像的节点时，暂不支持升级。
- 集群升级后，如**版本特性**中修复了容器引擎Containerd相关漏洞，存量节点需要手动重启Containerd才可以生效，对于存量Pod同样需要通过重启Pod才能生效。
- 若您将节点上的docker.sock文件通过HostPath方式挂载到Pod内，即docker in docker场景，在升级过程中Docker将会重启，而容器内的sock文件不会变化，可能导致您业务异常，推荐您使用挂载目录的方式挂载sock文件。
- 容器隧道网络集群升级至1.19.16-r4、1.21.7-r0、1.23.5-r0、1.25.1-r0及之后的版本时，会移除匹配目的地址是容器网段且源地址非容器网段的SNAT规则；如果您之前通过配置VPC路由实现集群外直接访问所有的Pod IP，升级后只支持直接访问对应节点上的Pod IP。

废弃 API 说明

随着Kubernetes API的演化，API会周期性地被重组或升级，老的API会被弃用并被最终删除。以下为各Kubernetes社区版本中被废弃的API，更多已废弃的API说明请参见**已弃用 API 的迁移指南**。

- **Kubernetes社区v1.25版本中废弃的API**
- **Kubernetes社区v1.22版本中废弃的API**
- **Kubernetes社区v1.16版本中废弃的API**

说明

当某API被废弃时，已经创建的资源对象不受影响，但新建或编辑该资源时将出现API版本被拦截的情况。

表 6-10 Kubernetes 社区 v1.25 版本中废弃的 API

资源名称	废弃API版本	替代API版本	变更说明
CronJob	batch/v1beta1	batch/v1 （该API从社区v1.21版本开始可用）	-
EndpointSlice	discovery.k8s.io/v1beta1	discovery.k8s.io/v1 （该API从社区v1.21版本开始可用）	此次更新需注意以下变更： ● 每个Endpoint中，**topology["kubernetes.io/hostname"]**字段已被弃用，请使用**nodeName**字段代替。 ● 每个Endpoint中，**topology["kubernetes.io/zone"]**字段已被弃用，请使用**zone**字段代替。 ● **topology**字段被替换为**deprecatedTopology**，并且在 v1 版本中不可写入。

资源名称	废弃API版本	替代API版本	变更说明
Event	events.k8s.io/v1beta1	events.k8s.io/v1 （该API从社区v1.19版本开始可用）	此次更新需注意以下变更： • **type** 字段只能设置为 **Normal** 和 **Warning** 之一。 • **involvedObject** 字段被更名为 **regarding**。 • **action**、**reason**、**reportingController** 和 **reportingInstance** 字段 在创建新的 **events.k8s.io/v1** 版本 Event 时都是必需的字段。 • 使用 **eventTime** 而不是已被弃用的 **firstTimestamp** 字段（该字段已被更名为 **deprecatedFirstTimestamp**，且不允许出现在新的 **events.k8s.io/v1** Event 对象中）。 • 使用 **series.lastObservedTime** 而不是已被弃用的 **lastTimestamp** 字段（该字段已被更名为 **deprecatedLastTimestamp**，且不允许出现在新的 **events.k8s.io/v1** Event 对象中）。 • 使用 **series.count** 而不是已被弃用的 **count** 字段（该字段已被更名为 **deprecatedCount**，且不允许出现在新的 **events.k8s.io/v1** Event 对象中）。 • 使用 **reportingController** 而不是已被弃用的 **source.component** 字段（该字段已被更名为 **deprecatedSource.component**，且不允许出现在新的 **events.k8s.io/v1** Event 对象中）。 • 使用 **reportingInstance** 而不是已被弃用的 **source.host** 字段（该字段已被更名为 **deprecatedSource.host**，且不允许出现在新的

资源名称	废弃API版本	替代API版本	变更说明
			events.k8s.io/v1 Event 对象中）。
HorizontalPodAutoscaler	autoscaling/v2beta1	autoscaling/v2 （该API从社区v1.23版本开始可用）	-
PodDisruptionBudget	policy/v1beta1	policy/v1 （该API从社区v1.21版本开始可用）	在 **policy/v1** 版本的 **PodDisruptionBudget** 中将 **spec.selector** 设置为空（{}）时会选择名字空间中的所有 Pod（在 **policy/v1beta1** 版本中，空的 **spec.selector** 不会选择任何 Pod）。如果 **spec.selector** 未设置，则在两个 API 版本下都不会选择任何 Pod。
PodSecurityPolicy	policy/v1beta1	-	从社区v1.25版本开始，PodSecurityPolicy资源不再提供**policy/v1beta1**版本的API，并且PodSecurityPolicy准入控制器也会被删除。 请使用**Pod Security Admission配置**替代。
RuntimeClass	node.k8s.io/v1beta1	node.k8s.io/v1 （该API从社区v1.20版本开始可用）	-

表 6-11 Kubernetes 社区 v1.22 版本中废弃的 API

资源名称	废弃API版本	替代API版本	变更说明
MutatingWebhookConfiguration ValidatingWebhookConfiguration	admissionregistration.k8s.io/v1beta1	admissionregistration.k8s.io/v1 （该API从社区v1.16版本开始可用）	• **webhooks[*].failurePolicy** 在 v1 版本中默认值从 **Ignore** 改为 **Fail**。 • **webhooks[*].matchPolicy** 在 v1 版本中默认值从 **Exact** 改为 **Equivalent**。 • **webhooks[*].timeoutSeconds** 在 v1 版本中默认值从 **30s** 改为 **10s**。 • **webhooks[*].sideEffects** 的默认值被删除，并且该字段变为必须指定； 在 v1 版本中可选的值只能是 **None** 和 **NoneOnDryRun** 之一。 • **webhooks[*].admissionReviewVersions** 的默认值被删除，在 v1 版本中此字段变为必须指定（**AdmissionReview** 的被支持版本包括 v1 和 v1beta1）。 • **webhooks[*].name** 必须在通过 **admissionregistration.k8s.io/v1** 创建的对象列表中唯一。

资源名称	废弃API版本	替代API版本	变更说明
CustomResourceDefinition	apiextensions.k8s.io/v1beta1	apiextensions/v1 （该API从社区v1.16版本开始可用）	• **spec.scope** 的默认值不再是 **Namespaced**，该字段必须显式指定。 • **spec.version** 在 v1 版本中被删除，应改用 **spec.versions**。 • **spec.validation** 在 v1 版本中被删除，应改用 **spec.versions[*].schema**。 • **spec.subresources** 在 v1 版本中被删除，应改用 **spec.versions[*].subresources**。 • **spec.additionalPrinterColumns** 在 v1 版本中被删除，应改用 **spec.versions[*].additionalPrinterColumns**。 • **spec.conversion.webhookClientConfig** 在 v1 版本中被移动到 **spec.conversion.webhook.clientConfig** 中。 • **spec.conversion.conversionReviewVersions** 在 v1 版本中被移动到 **spec.conversion.webhook.conversionReviewVersions** • **spec.versions[*].schema.openAPIV3Schema** 在创建 v1 版本的 **CustomResourceDefinition** 对象时变成必需字段，并且其取值必须是一个**结构化的 Schema**。 • **spec.preserveUnknownFields: true** 在创建 v1 版本的 **CustomResourceDefinition** 对象时不允许指定；该配置必须在 Schema 定义中使用 **x-kubernetes-preserve-unknown-fields: true** 来设置。 • 在 v1 版本中，**additionalPrinterColumns** 的条目中的 **JSONPath** 字段

资源名称	废弃API版本	替代API版本	变更说明
			被更名为 **jsonPath**（补丁 **#66531**）。
APIService	apiregistration/v1beta1	apiregistration.k8s.io/v1 （该API从社区v1.10版本开始可用）	-
TokenReview	authentication.k8s.io/v1beta1	authentication.k8s.io/v1 （该API从社区v1.6版本开始可用）	-
LocalSubjectAccessReview SelfSubjectAccessReview SubjectAccessReview SelfSubjectRulesReview	authorization.k8s.io/v1beta1	authorization.k8s.io/v1 （该API从社区v1.16版本开始可用）	**spec.group** 在 v1 版本中被更名为 **spec.groups**（补丁 **#32709**）

资源名称	废弃API版本	替代API版本	变更说明
CertificateSigningRequest	certificates.k8s.io/v1beta1	certificates.k8s.io/v1 （该API从社区v1.19版本开始可用）	**certificates.k8s.io/v1** 中需要额外注意的变更： • 对于请求证书的 API 客户端而言： – **spec.signerName** 现在变成必需字段（参阅 **已知的 Kubernetes 签署者**）， 并且通过 certificates.k8s.io/v1 API 不可以创建签署者为 **kubernetes.io/legacy-unknown** 的请求。 – **spec.usages** 现在变成必需字段，其中不可以包含重复的字符串值， 并且只能包含已知的用法字符串。 • 对于要批准或者签署证书的 API 客户端而言： – **status.conditions** 中不可以包含重复的类型。 – **status.conditions[*].status** 字段现在变为必需字段。 – **status.certificate** 必须是 PEM 编码的，而且其中只能包含 CERTIFICATE 数据块。
Lease	coordination.k8s.io/v1beta1	coordination.k8s.io/v1 （该API从社区v1.14版本开始可用）	-

资源名称	废弃API版本	替代API版本	变更说明
Ingress	networking.k8s.io/v1beta1 extensions/v1beta1	networking.k8s.io/v1 （该API从社区v1.19版本开始可用）	• **spec.backend** 字段被更名为 **spec.defaultBackend**。 • 后端的 **serviceName** 字段被更名为 **service.name**。 • 数值表示的后端 **servicePort** 字段被更名为 **service.port.number**。 • 字符串表示的后端 **servicePort** 字段被更名为 **service.port.name**。 • 对所有要指定的路径，**pathType** 都成为必需字段。 可选项为 **Prefix**、**Exact** 和 **ImplementationSpecific**。要匹配 v1beta1 版本中未定义路径类型时的行为，可使用 **ImplementationSpecific**。
IngressClass	networking.k8s.io/v1beta1	networking.k8s.io/v1 （该API从社区v1.19版本开始可用）	-
ClusterRole ClusterRoleBinding Role RoleBinding	rbac.authorization.k8s.io/v1beta1	rbac.authorization.k8s.io/v1 （该API从社区v1.8版本开始可用）	-
PriorityClass	scheduling.k8s.io/v1beta1	scheduling.k8s.io/v1 （该API从社区v1.14版本开始可用）	-

资源名称	废弃API版本	替代API版本	变更说明
CSIDriver CSINode StorageClass VolumeAttachment	storage.k8s.io/v1beta1	storage.k8s.io/v1	• CSIDriver从社区v1.19版本开始在**storage.k8s.io/v1**中提供。 • CSINode从社区v1.17版本开始在**storage.k8s.io/v1**中提供。 • StorageClass从社区v1.6 版本开始在**storage.k8s.io/v1**中提供。 • VolumeAttachment从社区v1.13版本开始在**storage.k8s.io/v1**中提供。

表 6-12 Kubernetes 社区 v1.16 版本中废弃的 API

资源名称	废弃API版本	替代API版本	变更说明
NetworkPolicy	extensions/v1beta1	networking.k8s.io/v1 （该API从社区v1.8版本开始可用）	-
DaemonSet	extensions/v1beta1 apps/v1beta2	apps/v1 （该API从社区v1.9版本开始可用）	• **spec.templateGeneration**字段被删除。 • **spec.selector** 现在变成必需字段，并且在对象创建之后不可变更； 可以将现有模板的标签作为选择算符以实现无缝迁移。 • **spec.updateStrategy.type**的默认值变为**RollingUpdate**（**extensions/v1beta1** API版本中的默认值是**OnDelete**）。

资源名称	废弃API版本	替代API版本	变更说明
Deployment	extensions/v1beta1 apps/v1beta1 apps/v1beta2	apps/v1 （该API从社区v1.9版本开始可用）	• **spec.rollbackTo** 字段被删除。 • **spec.selector** 字段现在变为必需字段，并且在 Deployment 创建之后不可变更； 可以使用现有的模板的标签作为选择算符以实现无缝迁移。 • **spec.progressDeadlineSeconds** 的默认值变为 600 秒（**extensions/v1beta1** 中的默认值是没有期限）。 • **spec.revisionHistoryLimit** 的默认值变为 10（**apps/v1beta1** API 版本中此字段默认值为 2，在**extensions/v1beta1** API 版本中的默认行为是保留所有历史记录）。 • **maxSurge** 和 **maxUnavailable** 的默认值变为 25%（在 **extensions/v1beta1** API 版本中，这些字段的默认值是 1）。
StatefulSet	apps/v1beta1 apps/v1beta2	apps/v1 （该API从社区v1.9版本开始可用）	• **spec.selector** 字段现在变为必需字段，并且在 StatefulSet 创建之后不可变更； 可以使用现有的模板的标签作为选择算符以实现无缝迁移。 • **spec.updateStrategy.type** 的默认值变为 **RollingUpdate**（**apps/v1beta1** API 版本中的默认值是 **OnDelete**）。
ReplicaSet	extensions/v1beta1 apps/v1beta1 apps/v1beta2	apps/v1 （该API从社区v1.9版本开始可用）	**spec.selector** 现在变成必需字段，并且在对象创建之后不可变更； 可以将现有模板的标签作为选择算符以实现无缝迁移。
PodSecurityPolicy	extensions/v1beta1	policy/v1beta1 （该API从社区v1.10版本开始可用）	**policy/v1beta1** API 版本的 PodSecurityPolicy 会在 v1.25 版本中移除。

版本差异说明

版本升级路径	版本差异	建议自检措施
v1.23升级至v1.25	在Kubernetes v1.25版本中，PodSecurityPolicy已被移除，并提供Pod安全性准入控制器（**Pod Security Admission配置**）作为PodSecurityPolicy的替代。	• 如果您需要将PodSecurityPolicy的相关能力迁移到Pod Security Admission中，需要参照以下步骤进行： 1. 确认集群为CCE v1.23的最新版本。 2. 迁移PodSecurityPolicy的相关能力迁移到Pod Security Admission，请参见**从PodSecurityPolicy迁移到Pod Security Admission**。 3. 确认迁移后功能正常，再升级为CCE v1.25版本。 • 如果您不再使用PodSecurityPolicy能力，则可以在删除集群中的PodSecurityPolicy后，直接升级为CCE v1.25版本。
v1.21升级至v1.23 v1.19升级至v1.23	社区较老版本的Nginx Ingress Controller来说（社区版本v0.49及以下，对应CCE插件版本v1.x.x），在创建Ingress时没有指定Ingress类别为nginx，即annotations中未添加kubernetes.io/ingress.class: nginx的情况，也可以被Nginx Ingress Controller纳管。但对于较新版本的Nginx Ingress Controller来说（社区版本v1.0.0及以上，对应CCE插件版本2.x.x），如果在创建Ingress时没有显示指定Ingress类别为nginx，该资源将被Nginx Ingress Controller忽略，Ingress规则失效，导致服务中断。	请参照**nginx-ingress插件升级检查**进行自检。
v1.19升级至v1.21	Kubernetes v1.21集群版本修复了exec probe timeouts不生效的BUG，在此修复之前，exec 探测器不考虑 timeoutSeconds 字段。相反，探测将无限期运行，甚至超过其配置的截止日期，直到返回结果。 若用户未配置，默认值为1秒。升级后此字段生效，如果探测时间超过1秒，可能会导致应用健康检查失败并频繁重启。	升级前检查您使用了exec probe的应用的probe timeouts是否合理。

版本升级路径	版本差异	建议自检措施
	CCE的v1.19及以上版本的kube-apiserver要求客户侧webhook server的证书必须配置Subject Alternative Names (SAN)字段。否则升级后kube-apiserver调用webhook server失败，容器无法正常启动。 根因：Go语言v1.15版本废弃了X.509 **CommonName**，CCE的v1.19版本的kube-apiserver编译的版本为v1.15，若客户的webhook证书没有Subject Alternative Names (SAN)，kube-apiserver不再默认将X509证书的CommonName字段作为hostname处理，最终导致认证失败。	升级前检查您自建webhook server的证书是否配置了SAN字段。 • 若无自建webhook server则不涉及。 • 若未配置，建议您配置使用SAN字段指定证书支持的IP及域名。
v1.15升级至v1.19	CCE v1.19版本的控制面与v1.15版本的Kubelet存在兼容性问题。若Master节点升级成功后，节点升级失败或待升级节点发生重启，则节点有极大概率为NotReady状态。 主要原因为升级失败的节点有大概率重启kubelet而触发节点注册流程，v1.15 kubelet默认注册标签（failure-domain.beta.kubernetes.io/is-baremetal和kubernetes.io/availablezone）被v1.19版本kube-apiserver视为非法标签。 v1.19版本中对应的合法标签为node.kubernetes.io/baremetal和failure-domain.beta.kubernetes.io/zone。	1. 正常升级流程不会触发此场景。 2. 在Master升级完成后尽量避免使用暂停升级功能，快速升级完Node节点。 3. 若Node节点升级失败且无法修复，请尽快驱逐此节点上的应用，请联系技术支持人员，跳过此节点升级，在整体升级完毕后，重置该节点。

版本升级路径	版本差异	建议自检措施

	CCE的v1.15版本集群及v1.19版本集群将docker的存储驱动文件系统由 xfs切换成ext4，可能会导致升级后的java应用Pod内的import包顺序异常，既而导致Pod异常。	升级前查看节点上docker配置文件/etc/docker/daemon.json。检查dm.fs配置项是否为xfs。
		• 若为ext4或存储驱动为overlay则不涉及。
		• 若为xfs则建议您在新版本集群预先部署应用，以测试应用与新版本集群是否兼容。
		{
		"storage-driver": "devicemapper",
		"storage-opts": [
		"dm.thinpooldev=/dev/mapper/vgpaas-thinpool",
		"dm.use_deferred_removal=true",
		"dm.fs=xfs",
		"dm.use_deferred_deletion=true"
		]
		}

	CCE的v1.19及以上版本的kube-apiserver要求客户侧webhook server的证书必须配置Subject Alternative Names (SAN)字段。否则升级后kube-apiserver调用webhook server失败，容器无法正常启动。 根因：Go语言v1.15版本废弃了X.509 **CommonName**，CCE的v1.19版本的kube-apiserver编译的版本为v1.15。CommonName字段作为hostname处理，最终导致认证失败。	升级前检查您自建webhook server的证书是否配置了SAN字段。 • 若无自建webhook server则不涉及。 • 若未配置，建议您配置使用SAN字段指定证书支持的IP及域名。 **须知** 为减弱此版本差异对集群升级的影响，v1.15升级至v1.19时，CCE会进行特殊处理，仍然会兼容支持证书不带SAN。但后续升级不再特殊处理，请尽快整改证书，以避免影响后续升级。
	v1.17.17版本及以后的集群CCE自动给用户创建了PSP规则，限制了不安全配置的Pod的创建，如securityContext配置了sysctl的net.core.somaxconn的Pod。	升级后请参考资料按需开放非安全系统配置，具体请参见**PodSecurityPolicy配置**。
	1.15版本集群原地升级，如果业务中存在initContainer或使用Istio的场景，则需要注意以下约束： 1.16及以上的kubelet统计**QosClass**和之前版本存在差异，1.15及以下版本仅统计spec.containers下的容器，1.16及以上的kubelet版本会同时统计spec.containers和spec.initContainers下的容器，升级前后会造成Pod的QosClass变化，从而造成Pod中容器重启。	建议参考**表6-13**在升级前修改业务容器的QosClass规避该问题。

版本升级路径	版本差异	建议自检措施
v1.13升级至v1.15	vpc集群升级后，由于网络组件的升级，master节点会额外占一个网段。在Master占用了网段后，无可用容器网段时，新建节点无法分配到网段，调度在该节点的pod会无法运行。	一般集群内节点数量快占满容器网段场景下会出现该问题。例如，容器网段为10.0.0.0/16，可用IP数量为65536，VPC网络IP分配是分配固定大小的网段（使用掩码实现，确定每个节点最多分配多少容器IP），例如上限为128，则此时集群最多支撑65536/128=512个节点，然后去掉Master节点数量为509，此时是1.13集群支持的节点数。集群升级后，在此基础上3台Master节点会各占用1个网段，最终结果就是506台节点。

表 6-13 1.15 版本升级前后 QosClass 变化

init容器（根据spec.initContainers计算）	业务容器（根据spec.containers计算）	Pod（根据spec.containers和spec.initContainers计算）	是否受影响
Guaranteed	Besteffort	Burstable	是
Guaranteed	Burstable	Burstable	否
Guaranteed	Guaranteed	Guaranteed	否
Besteffort	Besteffort	Besteffort	否
Besteffort	Burstable	Burstable	否
Besteffort	Guaranteed	Burstable	是
Burstable	Besteffort	Burstable	是
Burstable	Burstable	Burstable	否
Burstable	Guaranteed	Burstable	是

升级备份说明

目前集群升级备份方式如下：

- **集群ETCD数据库备份**：CCE服务会在集群升级流程中对etcd数据库进行备份，无需用户确认。
- **Master节点整机备份**：在升级界面对集群的Master节点进行整机备份，**需要用户手动确认**，备份过程会使用云备份服务，备份通常耗时在20分钟左右，若当前局点云备份任务排队较多时，备份时间可能同步延长，推荐用户使用进行整机备份。

6.4.3 原地升级

您可以通过云容器引擎管理控制台升级集群版本，以支持新特性的使用。

升级前，请先了解CCE各集群版本能够升级到的目标版本，以及升级方式和升级影响，详情请参见**升级概述**和**升级前须知**。

升级说明

- 集群的升级采用原地升级方式更新节点上的Kubernetes组件，升级后不会改变节点上的OS版本。
- 数据面节点升级时将采用分批升级的方式，默认会选择根据CPU、内存、PDB（Pod Disruption Budget，即**干扰预算**）等设置节点升级的优先级，您也可以根据您的业务需要自行设置优先级。

注意事项

- 集群升级过程中会自动升级插件到目标集群兼容的版本，升级过程中请不要卸载或者重装插件。
- 升级之前请确认所有的插件都处于运行状态，如果插件升级失败可以在插件问题修复后，重试升级。
- 升级时会检查插件运行状态，部分插件（如CoreDNS）需要至少两个节点才能维持正常状态，那此时升级就至少需要两个节点。

更多注意事项请参见**升级前须知**。

操作步骤

集群升级步骤包括：升级前检查、备份、配置与升级、升级后处理。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“集群升级”。

步骤3 进行升级前检查，单击“开始检查”并确认。如集群中存在异常项或风险项，请根据页面提示的检查结果进行处理，处理完成后需重新进行升级前检查。

- 异常项：请查看页面提示的解决方案并处理异常后，重新进行升级前检查。
- 风险项：表示该结果可能会影响集群升级结果，请您查看风险说明并确认您是否处于风险影响范围。如确认无风险，可单击该风险项后的“确认”按钮，手动跳过该风险项，然后重新进行升级前检查。

待升级前检查通过后，单击“下一步 备份”。

步骤4 （可选）手动进行集群备份。集群升级时会提供默认的数据备份，如需手动备份可单击“备份”按钮执行备份，如无需手动备份可直接单击“下一步 配置与升级”。

手动单击“备份”按钮会对集群的Master节点进行整机备份，备份过程会使用云备份服务，备份通常耗时在20分钟左右，若当前局点云备份任务排队较多时，备份时间可能同步延长。备份过程中集群将不允许升级。

步骤5 配置升级参数。

- **插件升级配置：**此处列出了您的集群中已安装的插件。在集群升级过程中系统会自动升级插件，以兼容升级后的集群版本，您可以单击插件右侧的“配置”重新定义插件参数。

说明

插件右侧如有红色标记，表明该插件将不能兼容升级后的集群版本，升级过程中会卸载并重装该插件，请您务必确认插件的配置参数。

- **节点升级配置：**您可以设置每批升级的最大节点数量。
- **节点优先级配置：**您可以自行定义节点升级的优先级顺序，若不选择，默认情况下系统会根据您节点的情况优选后分批升级。优先级设置时需要先选择节点池，再设置节点池中节点的升级批次，并按照您设置的节点池以及节点顺序进行升级。
  - 添加优先级：添加节点池的优先级，自行定义节点池升级的优先级顺序。
  - 添加节点优先级：添加节点池的优先级后，可以设置该节点池内节点升级的优先级顺序，升级时系统将按照您设置的顺序依次对节点进行升级，如不设置该优先级，系统将按照默认的策略执行。

步骤6 配置完成后，单击“升级”按钮，并确认升级操作后集群开始升级。您可以在页面下方查看版本升级的进程。

升级过程中，您可以单击右侧的“暂停”按钮，暂停集群版本的升级，若想继续升级，可单击“继续”。当版本更新进度条显示100%时，表示集群已完成升级。

说明

若在集群升级过程中出现升级失败的提示，请参照提示信息修复问题后重试。

步骤7 升级完成后，单击“下一步 升级后验证”，请根据页面提示的检查项进行升级后验证。确认所有检查项均正常后，可单击“完成”按钮，并确认完成升级后检查。

您可以在集群列表页面查看集群当前的Kubernetes版本，确认升级成功。

----结束

6.4.4 升级后验证

6.4.4.1 业务验证

检查项内容

集群升级完毕，由用户验证当前集群正在运行的业务是否正常。

检查步骤

业务不同，验证的方式也有所不同，建议您在升级前确认适合您业务的验证方式，并在升级前后均执行一遍。

常见的业务确认方式有：

- 业务界面可用
- 监控平台无异常告警与事件
- 关键应用进程无错误日志
- API拨测正常等

解决方案

若集群升级后您的在线业务有异常，请联系技术支持人员。

6.4.4.2 存量 Pod 检查

检查项内容

- 检查集群中是否的存在状态非预期的Pod
- 检查集群中的原先正常运行的Pod是否存在异常重启的情况

检查步骤

请您登录CCE控制台，在“资源->工作负载->容器组”处选择全部命名空间，单击“状态”，过滤并观察是否存在异常状态的Pod。

查看“重启次数”栏目，观察是否存在异常重启的Pod。

解决方案

若集群升级后您的集群有异常Pod，请联系技术支持人员。

6.4.4.3 存量节点与容器网络检查

检查项内容

- 检查存量节点是否运行正常
- 检查存量节点的网络是否运行正常
- 检查存量容器的网络是否运行正常

检查步骤

节点组件异常或节点网络异常，均会反映在节点状态上。

请登录CCE控制台，前往“资源->节点管理”处查看节点状态，检查是否有处于异常状态的节点，可通过状态栏过滤。

容器网络异常会反映在业务上，请检查您的业务是否运行正常。

解决方案

若节点状态异常，请联系技术支持人员。

若容器网络异常，并影响了您的业务，请联系技术支持人员，并同步确认当前异常的网络访问路径。

源端	目的端	目的端类型	可能故障
• 集群内Pod • 集群内节点 • 集群外但处于同VPC下的云服务器 • 集群所在VPC外	Service ELB 公网IP	集群流量负载均衡入口	未有记录
	Service ELB 私网IP	集群流量负载均衡入口	未有记录
	Ingress ELB 公网IP	集群流量负载均衡入口	未有记录
	Ingress ELB 私网IP	集群流量负载均衡入口	未有记录
	Service Node Port 公网IP	集群流量入口	kube proxy配置覆盖，该故障已在升级流程适配
	Service Node Port 私网IP	集群流量入口	未有记录
	Service Cluster IP	Service网络平面	未有记录
	非Service NodePort 节点端口	节点容器网络	未有记录
	跨节点Pod	容器网络平面	未有记录
	同节点Pod	容器网络平面	未有记录
	Service域名、Pod域名等，基于CoreDNS解析	域名解析	未有记录
	外部网站域名，基于CoreDNS hosts配置解析	域名解析	coredns插件升级后配置被覆盖，该故障已在插件升级流程适配
	外部网站域名，基于CoreDNS 上游服务器解析	域名解析	coredns插件升级后配置被覆盖，该故障已在插件升级流程适配
	外部网站域名，不通过CoreDNS解析	域名解析	未有记录

6.4.4.4 存量节点标签与污点检查

检查项内容

- 检查自定义的节点标签是否丢失
- 检查节点上是否存在异常新增的污点，影响工作负载调度

检查步骤

请登录CCE控制台，前往“资源->节点管理->节点”，勾选所有节点后，单击“标签与污点管理”，查看目前节点的标签与污点。

解决方案

集群升级过程中不会改变用户自定义的标签，若您发现标签丢失或异常新增，请联系技术支持人员。

若您发现节点新增污点（node.kubernetes.io/upgrade），该节点可能为升级过程中跳过的节点，请参照**跳过节点检查**处理。

若您发现节点新增其他污点，请联系技术人员支持。

6.4.4.5 新建节点检查

检查内容

检查集群是否可以正常创建节点。

检查步骤

请登录CCE控制台，前往“资源->节点管理”，单击“创建节点”。节点配置详情请参见**创建节点**。

解决方案

若集群升级后您的集群无法创建节点，请联系技术支持人员。

6.4.4.6 新建 Pod 检查

检查内容

- 检查集群升级后，存量节点是否能新建Pod。
- 检查集群升级后，新建节点是否能新建Pod。

检查步骤

基于**新建节点检查**创建了新节点后，通过创建DaemonSet类型工作负载，在每个节点上创建Pod。

请登录CCE控制台，前往“资源->工作负载->守护进程集”，单击右上角“创建负载”或“YAML创建”。创建DaemonSet的操作步骤详情请参见**创建守护进程集（DaemonSet）**。

建议您使用日常测试的镜像作为基础镜像。您可参照如下yaml部署最小应用Pod。

说明

该测试YAML将DaemonSet部署在default命名空间下，使用ngxin:perl为基础镜像，申请10m CPU，10Mi内存，限制100m CPU 50Mi内存。

```
apiVersion: apps/v1
kind: DaemonSet
```

```
metadata:
  name: post-upgrade-check
  namespace: default
spec:
  selector:
    matchLabels:
      app: post-upgrade-check
      version: v1
  template:
    metadata:
      labels:
        app: post-upgrade-check
        version: v1
    spec:
      containers:
        - name: container-1
          image: nginx:perl
          imagePullPolicy: IfNotPresent
          resources:
            requests:
              cpu: 10m
              memory: 10Mi
            limits:
              cpu: 100m
              memory: 50Mi
```

负载创建完毕后请检查该工作负载的Pod状态是否正常。

检查完毕后请登录CCE控制台，前往“资源->工作负载->守护进程集”，选择“post-upgrade-check”工作负载并单击“更多->删除”删除该测试用工作负载。

解决方案

若Pod无法新建，或状态异常，请联系技术支持人员，并说明异常发生的范围为新建节点还是存量节点。

6.4.4.7 跳过节点检查

检查项内容

集群升级后，需要检测集群内是否有跳过升级的节点，这些节点可能会影响正常使用。

检查步骤

系统会为您检查集群内是否存在跳过升级的节点，您可以根据诊断结果前往节点列表页进行确认。跳过的节点含有标签upgrade.cce.io/skipped=true。

解决方案

对于升级详情页面中跳过的节点，请在升级完毕后**重置节点**。

说明

重置节点会重置所有节点标签，可能影响工作负载调度，请在重置节点前检查并保留您手动为该节点打上的标签。

6.4.5 集群跨版本业务迁移

适用场景

本章介绍在CCE中如何将老版本集群的业务迁移到新版本集群。

适用于需要大幅度跨版本集群升级（如1.7.*或1.9.* 升级到1.17.*版本）的需求，可以接受新建新版本集群而进行业务迁移的升级方式。

前提条件

表 6-14 迁移前 Checklist

类别	描述
集群相关	Nodeip强相关：确认之前集群的节点IP（包括EIP），是否有作为其他的配置或者白名单之类的设置。
工作负载	记录工作负载数目，便于迁移后检查。
存储	1. 确认应用中存储，是否使用云，或者自己搭建存储。 2. 自动创建的存储需要在新集群中变成使用已有存储。
网络	1. 注意使用的负载均衡服务，以及Ingress。 2. 老版本的集群只支持经典型负载均衡服务，迁移到新集群中需要改成共享型负载均衡服务，对应负载均衡服务将会重新建立。
运维	私有配置：确认在之前集群中，是否在节点上配置内核参数或者系统配置。

操作步骤

步骤1 创建新集群

创建与老版本集群同规格同配置的集群，创建方法请参见**创建集群**。

步骤2 添加节点

添加同规格节点，并且在节点上配置之前的手动配置项，创建方法请参见**创建节点**。

步骤3 创建存储

在新集群中使用已有存储创建PVC，PVC名称不变，方法请参见**通过静态存储卷使用已有对象存储**或**通过静态存储卷使用已有极速文件存储**。

说明

切流方案仅支持OBS、SFS Turbo等共享存储。非共享存储切流需要将老集群内的工作负载暂停，将会导致服务不可用。

步骤4 创建工作负载

在新集群中创建工作负载，名称和规格参数保持不变，创建方法请参见**创建无状态负载（Deployment）**或**创建有状态负载（StatefulSet）**。

步骤5 重新挂载存储

在工作负载中重新挂载已有的存储，方法请参见**通过静态存储卷使用已有对象存储**或**通过静态存储卷使用已有极速文件存储**。

步骤6 创建服务

在新集群中创建Service，名称和规格参数保持不变，创建方法请参见**服务（Service）**。

步骤7 调测功能

全部创建完成后，请自行调测业务，调测无问题后切换流量。

步骤8 老集群删除

新集群全部功能ready，删除老集群，删除集群方法请参见**删除集群**。

----结束

6.4.6 升级前检查异常问题排查

6.4.6.1 升级前检查项

集群升级前系统将进行全面的升级前检查，当集群不满足升级前检查条件时将无法继续升级。为了能够更好地避免升级风险，本文提供全量的升级前检查项帮助您更好地避免可能存在的升级故障。

表 6-15 检查项列表

序号	检查项名称	检查项说明
1	**节点限制检查**	• 检查节点是否可用 • 检查节点操作系统是否支持升级 • 检查节点是否含有非预期的节点池标签 • 检查K8s节点名称是否与云服务器保持一致
2	**升级管控检查**	当前检查用户是否处于升级黑名单列表中。
3	**插件检查**	• 检查插件状态是否正常 • 检查插件是否支持目标版本
4	**Helm模板检查**	检查当前HelmRelease记录中是否含有目标集群版本不支持的K8s废弃API，可能导致升级后helm模板不可用。
5	**Master节点SSH联通性检查**	检查当前CCE是否能连接至您的Master节点。
6	**节点池检查**	检查节点池状态是否正常。
7	**安全组检查**	检查当前用户节点安全组是否允许Master节点使用ICMP协议访问节点。

序号	检查项名称	检查项说明
8	ARM节点限制检查	• 检查集群是否包含ARM架构的节点。
9	残留待迁移节点检查	检查节点是否需要迁移。
10	K8s废弃资源检查	检查集群是否存在对应版本已经废弃的资源。
11	兼容性风险检查	请您阅读版本兼容性差异，并确认不受影响。补丁升级不涉及版本兼容性差异。
12	节点CCE Agent版本检查	检测当前节点的CCE包管理组件cce-agent是否为最新版本。
13	节点CPU使用率检查	检查节点CPU使用情况，是否超过90%。
14	CRD检查	• 检查集群关键CRD "packageversions.version.cce.io"是否被删除。 • 检查集群关键CRD "network-attachment-definitions.k8s.cni.cncf.io"是否被删除。
15	节点磁盘检查	• 检查节点关键数据盘使用量是否满足升级要求 • 检查/tmp目录是否存在500MB可用空间
16	节点DNS检查	• 检查当前节点DNS配置是否能正常解析OBS地址 • 检查当前节点是否能访问存储升级组件包的OBS地址
17	节点关键目录文件权限检查	检查关键目录/var/paas下是否有异常属主和属组的文件。
18	节点Kubelet检查	检查节点kubelet服务是否运行正常。
19	节点内存检查	检查节点内存使用情况，是否超过90%。
20	节点时钟同步服务器检查	检查节点时钟同步服务器ntpd或chronyd是否运行正常。
21	节点OS检查	检查节点操作系统内核版本是否为CCE支持的版本。
22	节点CPU数量检查	检查Master节点的CPU数量是否大于2核。
23	节点Python命令检查	检查Node节点中Python命令是否可用。
24	ASM网格版本检查	• 检查集群是否使用ASM网格服务 • 检查当前ASM版本是否支持目标集群版本
25	节点Ready检查	检查集群内节点是否Ready。
26	节点journald检查	检查节点上的journald状态是否正常。
27	节点干扰ContainerdSock检查	检查节点上是否存在干扰的Containerd.Sock文件。该文件影响Euler操作系统下的容器运行时启动。

序号	检查项名称	检查项说明
28	内部错误	在升级前检查流程中是否出现内部错误。
29	节点挂载点检查	检查节点上是否存在不可访问的挂载点。
30	K8s节点污点检查	检查节点上是否存在集群升级需要使用到的污点。
31	everest插件版本限制检查	检查集群当前everest插件版本是否存在兼容性限制。
32	cce-hpa-controller插件限制检查	检查到目标cce-controller-hpa插件版本是否存在兼容性限制。
33	增强型CPU管理策略检查	检查当前集群版本和要升级的目标版本是否支持增强型CPU管理策略。
34	用户节点组件健康检查	检查用户节点的容器运行时组件和网络组件等是否健康。
35	控制节点组件健康检查	检查控制节点的Kubernetes组件、容器运行时组件、网络组件等是否健康。
36	K8s组件内存资源限制检查	检查K8s组件例如etcd、kube-controller-manager等组件是否资源超出限制。
37	K8s废弃API检查	系统会扫描过去一天的审计日志，检查用户是否调用目标K8s版本已废弃的API。 **说明** 由于审计日志的时间范围有限，该检查项仅作为辅助手段，集群中可能已使用即将废弃的API，但未在过去一天的审计日志中体现，请您充分排查。
38	CCE Turbo集群IPv6能力检查	如CCE Turbo集群启用IPv6，需检查目标集群版本是否支持IPv6。
39	节点NetworkManager检查	检查节点上的NetworkManager状态是否正常。
40	节点ID文件检查	检查节点的ID文件内容是否符合格式。
41	节点配置一致性检查	在升级CCE集群版本至v1.19及以上版本时，将对您的节点上的Kubenertes组件的配置进行检查，检查您是否后台修改过配置文件。
42	节点配置文件检查	检查节点上关键组件的配置文件是否存在。
43	CoreDNS配置一致性检查	检查当前CoreDNS关键配置Corefile是否同Helm Release记录存在差异，差异的部分可能在插件升级时被覆盖，**影响集群内部域名解析**。
44	节点Sudo检查	检查当前节点sudo命令，sudo相关文件是否正常。
45	节点关键命令检查	检查节点升级依赖的一些关键命令是否能正常执行。

序号	检查项名称	检查项说明
46	节点sock文件挂载检查	检查节点上的docker/containerd.sock文件通过HostPath方式挂载到Pod内，在升级过程中Docker/Containerd将会重启，而容器内的sock文件不会变化，可能导致您业务异常。
47	HTTPS类型负载均衡证书一致性检查	检查HTTPS类型负载均衡所使用的证书，是否在ELB服务侧被修改。
48	节点挂载检查	检查节点上默认挂载目录及软链接是否被手动挂载或修改。
49	节点paas用户登录权限检查	检查paas用户是否有登录权限。
50	ELB IPv4私网地址检查	检查集群内负载均衡类型的Service所关联的ELB实例是否包含IPv4私网IP。
51	检查历史升级记录是否满足升级条件	检查集群最初版本是否小于v1.11，且升级的目标版本大于v1.23。
52	检查集群管理平面网段是否与主干配置一致	检查集群管理平面网段是否与主干配置一致。
53	GPU插件检查	检查到本次升级涉及GPU插件，可能影响新建GPU节点时GPU驱动的安装。
54	节点系统参数检查	检查您节点上默认系统参数是否被修改。
55	残留packageversion检查	检查当前集群中是否存在残留的packageversion。
56	节点命令行检查	检查节点中是否存在升级所必须的命令。
57	节点交换区检查	检查集群节点上是否开启交换区。
58	nginx-ingress插件升级检查	检查nginx-ingress插件升级路径是否涉及兼容问题。

6.4.6.2 节点限制检查

检查项内容

当前检查项包括以下内容：

- 检查节点是否可用
- 检查节点操作系统是否支持升级
- 检查节点是否含有非预期的节点池标签
- 检查K8s节点名称是否与云服务器保持一致

解决方案

1. **检查到节点状态异常，请优先恢复**

   若检查发现节点不可用，请登录CCE控制台，单击集群名称进入集群控制台，前往“节点管理”页面并切换至“节点”页签查看节点状态，请确保节点处于“运行中”状态。节点处于“安装中”、“删除中”状态时，均不支持升级。

   若节点状态异常，修复节点后，重试检查任务。

2. **检查到节点容器引擎不支持升级**

   该异常通常发生在低版本集群升级到v1.27及以上集群。由于v1.27及以上的集群，仅支持containerd运行时。若您的节点的运行时非containerd，暂时无法升级。您可通过节点重置功能重置节点的运行时为containerd。

3. **检查到节点操作系统不支持升级**

   当前集群升级支持的节点操作系统范围如下表所示，若您的节点OS不在支持列表之内，暂时无法升级。您可将节点重置为列表中可用的操作系统。

   表 6-16 节点 OS 支持列表

操作系统	限制
EulerOS 2.x	目标版本为v1.27以下时，无限制 目标版本为v1.27及以上时，仅支持EulerOS 2.9、EulerOS 2.10
CentOS 7.x	无限制
Ubuntu	部分局点受限，若检查结果不支持升级，请联系技术支持人员 **说明** 目标版本为v1.27及以上时，仅支持Ubuntu 22.04。

4. **检查到节点属于默认节点池，但是含有普通节点池标签，将影响升级流程**

   由节点池迁移至默认节点池的节点，仍然会保留节点池标签"cce.cloud.com/cce-nodepool"，影响集群升级。请确认该节点上的负载调度是否依赖改标签：

   - 若无依赖，请删除该标签。
   - 若存在依赖，请修改负载调度策略，解除依赖后再删除该标签。

5. **检查到节点含有CNIProblem污点，请优先恢复**

   检查到节点含有key为node.cloudprovider.kubernetes.io/cni-problem，效果为不可调度（NoSchedule）的污点。该污点由NPD插件检查添加，建议您优先升级NPD插件至最新版本，再重新检查，若仍然有问题，请联系支持人员。

6. **检查到节点对应的k8s node不存在，该节点可能正在删除中，请稍后重试检查**

   请稍后重试检查。

6.4.6.3 升级管控检查

检查项内容

检查集群是否处于升级管控中。

解决方案

CCE基于以下几点原因，可能会暂时限制该集群的升级功能：

- 基于用户提供的信息，该集群被识别为核心重点保障的生产集群。
- 正在或即将进行其他运维任务，例如Master节点3AZ改造等。

请联系技术支持人员了解限制原因并申请解除升级限制。

6.4.6.4 插件检查

检查项内容

当前检查项包括以下内容：

- 检查插件状态是否正常
- 检查插件是否支持目标版本

解决方案

- **问题场景一：插件状态异常**

  请登录CCE控制台，前往“集群信息->运维->插件管理”处查看并处理处于异常状态的插件。
- **问题场景二：目标集群版本不支持当前插件版本**

  检查到该插件由于兼容性等问题无法随集群自动升级，请您登录CCE控制台，在“集群信息->运维->插件管理”处进行手动升级。
- **问题场景三：插件升级到最新版本后，仍不支持目标集群版本**

  请您登录CCE控制台，在“集群信息->运维->插件管理”处进行手动卸载，具体插件支持版本以及替换方案可查看**帮助文档**。
- **问题场景四：插件配置不满足在升级条件，请在插件升级页面升级插件之后重试**

  升级前检查出现以下报错：

  ```
  please upgrade addon [ ] in the page of addon managecheck and try again
  ```

  请您登录CCE控制台，在“集群信息->运维->插件管理”处手动升级插件。

6.4.6.5 Helm 模板检查

检查项内容

检查当前HelmRelease记录中是否含有目标集群版本不支持的K8s废弃API，可能导致升级后helm模板不可用。

解决方案

将HelmRelease记录中K8s废弃API转换为源版本和目标版本均兼容的API。

说明

该检查项解决方案已在升级流程中自动兼容处理，此检查不再限制。您无需关注并处理。

6.4.6.6 Master 节点 SSH 联通性检查

检查项内容

检查当前CCE是否能连接至您的Master节点。

解决方案

请联系技术支持人员排查。

6.4.6.7 节点池检查

检查项内容

检查节点池状态是否正常。

解决方案

问题场景：节点池状态异常

请登录CCE控制台，前往“集群信息->资源->节点管理->节点池”，查看问题节点池状态。若该节点池状态处于伸缩中，请等待节点池伸缩完毕。

6.4.6.8 安全组检查

检查项内容

检查当前用户节点安全组是否允许Master节点使用ICMP协议访问节点。

说明

仅VPC网络模型的集群执行该检查项，非VPC网络模型的集群将跳过该检查项。

解决方案

请登录VPC控制台，前往“访问控制->安全组”，在搜索框内输入集群名称，此时预期过滤出两个安全组：

- 安全组名称为“集群名称-node-xxx”，此安全组关联CCE用户节点。
- 安全组名称为“集群名称-control-xxx”，此安全组关联CCE控制节点。

单击node用户节点安全组，确保含有如下规则允许Master节点使用**ICMP协议**访问节点。

若不含有该规则请为Node安全组添加该放通规则，协议端口选择“基本协议/ICMP”，端口号为“全部”，源地址选择“安全组”并设置为Master安全组。

6.4.6.9 ARM 节点限制检查

检查项内容

当前检查项包括以下内容

- 检查集群是否包含ARM架构的节点。

解决方案

- **问题场景一： 集群包含ARM架构的Node节点**
  删除ARM架构的节点。

6.4.6.10 残留待迁移节点检查

检查项内容

检查节点是否需要迁移。

解决方案

由1.13滚动升级而来的1.15集群，需要将所有节点迁移（重置或新建替换）后，才允许再次进行升级。

解决方案一

请登录CCE控制台，单击集群名称进入集群控制台，前往“集群信息->资源->节点管理”，单击对应节点的“更多->重置节点”，详情请参见**重置节点**。节点重置完毕后，重试检查任务。

说明

重置节点会重置所有节点标签，可能影响工作负载调度，请在重置节点前检查并保留您手动为该节点打上的标签。

解决方案二

新建节点后，删除问题节点。

6.4.6.11 K8s 废弃资源检查

检查项内容

检查集群是否存在对应版本已经废弃的资源。

解决方案

问题场景1： 1.25及以上集群废弃了PodSecurityPolicy资源对象

在集群中通过**kubectl get psp -A**命令获取当前集群中已存在的PSP对象。

如果您并未使用这两个对象，可以直接单击确定按钮跳过该检查，若您存在使用该对象的情况，请参见**Pod安全配置**相关文档，将PSP相应功能升级至PodSecurity安全准入能力中。

问题场景2： 1.25及以上集群中的service存在废弃的annotation：tolerate-unready-endpoints

检查日志信息中所给出的service是否存在**"tolerate-unready-endpoints"**的annotation，如果存在则将其去掉，并在对应的service的spec中添加下列字段来替代该annotation:

```
publishNotReadyAddresses: true
```

6.4.6.12 兼容性风险检查

检查项内容

请您阅读版本兼容性差异，并确认不受影响。补丁升级不涉及版本兼容性差异。

版本兼容性差异

版本升级路径	版本差异	建议自检措施
v1.23升级至v1.25	在Kubernetes v1.25版本中，PodSecurityPolicy已被移除，并提供Pod安全性准入控制器（**Pod Security Admission配置**）作为PodSecurityPolicy的替代。	• 如果您需要将PodSecurityPolicy的相关能力迁移到Pod Security Admission中，需要参照以下步骤进行： 1. 确认集群为CCE v1.23的最新版本。 2. 迁移PodSecurityPolicy的相关能力迁移到Pod Security Admission，请参见**从PodSecurityPolicy迁移到Pod Security Admission**。 3. 确认迁移后功能正常，再升级为CCE v1.25版本。 • 如果您不再使用PodSecurityPolicy能力，则可以在删除集群中的PodSecurityPolicy后，直接升级为CCE v1.25版本。
v1.21升级至v1.23 v1.19升级至v1.23	社区较老版本的Nginx Ingress Controller来说（社区版本v0.49及以下，对应CCE插件版本v1.x.x），在创建Ingress时没有指定Ingress类别为nginx，即annotations中未添加kubernetes.io/ingress.class: nginx的情况，也可以被Nginx Ingress Controller纳管。但对于较新版本的Nginx Ingress Controller来说（社区版本v1.0.0及以上，对应CCE插件版本2.x.x），如果在创建Ingress时没有显示指定Ingress类别为nginx，该资源将被Nginx Ingress Controller忽略，Ingress规则失效，导致服务中断。	请参照**nginx-ingress插件升级检查**进行自检。

版本升级路径	版本差异	建议自检措施
v1.19升级至v1.21	Kubernetes v1.21集群版本修复了exec probe timeouts不生效的BUG，在此修复之前，exec 探测器不考虑 timeoutSeconds 字段。相反，探测将无限期运行，甚至超过其配置的截止日期，直到返回结果。 若用户未配置，默认值为1秒。升级后此字段生效，如果探测时间超过1秒，可能会导致应用健康检查失败并频繁重启。	升级前检查您使用了exec probe的应用的probe timeouts是否合理。
	CCE的v1.19及以上版本的kube-apiserver要求客户侧webhook server的证书必须配置Subject Alternative Names (SAN)字段。否则升级后kube-apiserver调用webhook server失败，容器无法正常启动。 根因：Go语言v1.15版本废弃了X.509 **CommonName**，CCE的v1.19版本的kube-apiserver编译的版本为v1.15，若客户的webhook证书没有Subject Alternative Names (SAN)，kube-apiserver不再默认将X509证书的CommonName字段作为hostname处理，最终导致认证失败。	升级前检查您自建webhook server的证书是否配置了SAN字段。 • 若无自建webhook server则不涉及。 • 若未配置，建议您配置使用SAN字段指定证书支持的IP及域名。
v1.15升级至v1.19	CCE v1.19版本的控制面与v1.15版本的Kubelet存在兼容性问题。若Master节点升级成功后，节点升级失败或待升级节点发生重启，则节点有极大概率为NotReady状态。 主要原因为升级失败的节点有大概率重启kubelet而触发节点注册流程，v1.15 kubelet默认注册标签（failure-domain.beta.kubernetes.io/is-baremetal和kubernetes.io/availablezone）被v1.19版本kube-apiserver视为非法标签。 v1.19版本中对应的合法标签为node.kubernetes.io/baremetal和failure-domain.beta.kubernetes.io/zone。	1. 正常升级流程不会触发此场景。 2. 在Master升级完成后尽量避免使用暂停升级功能，快速升级完Node节点。 3. 若Node节点升级失败且无法修复，请尽快驱逐此节点上的应用，请联系技术支持人员，跳过此节点升级，在整体升级完毕后，重置该节点。

版本升级路径	版本差异	建议自检措施

	CCE的v1.15版本集群及v1.19版本集群将docker的存储驱动文件系统由 xfs切换成ext4，可能会导致升级后的java应用Pod内的import包顺序异常，既而导致Pod异常。	升级前查看节点上docker配置文件/etc/docker/daemon.json。检查dm.fs配置项是否为xfs。
		• 若为ext4或存储驱动为overlay则不涉及。
		• 若为xfs则建议您在新版本集群预先部署应用，以测试应用与新版本集群是否兼容。
		{
		"storage-driver": "devicemapper",
		"storage-opts": [
		"dm.thinpooldev=/dev/mapper/vgpaas-thinpool",
		"dm.use_deferred_removal=true",
		"dm.fs=xfs",
		"dm.use_deferred_deletion=true"
		]
		}

	CCE的v1.19及以上版本的kube-apiserver要求客户侧webhook server的证书必须配置Subject Alternative Names (SAN)字段。否则升级后kube-apiserver调用webhook server失败，容器无法正常启动。 根因：Go语言v1.15版本废弃了X.509 **CommonName**，CCE的v1.19版本的kube-apiserver编译的版本为v1.15。CommonName字段作为hostname处理，最终导致认证失败。	升级前检查您自建webhook server的证书是否配置了SAN字段。 • 若无自建webhook server则不涉及。 • 若未配置，建议您配置使用SAN字段指定证书支持的IP及域名。 **须知** 为减弱此版本差异对集群升级的影响，v1.15升级至v1.19时，CCE会进行特殊处理，仍然会兼容支持证书不带SAN。但后续升级不再特殊处理，请尽快整改证书，以避免影响后续升级。
	v1.17.17版本及以后的集群CCE自动给用户创建了PSP规则，限制了不安全配置的Pod的创建，如securityContext配置了sysctl的net.core.somaxconn的Pod。	升级后请参考资料按需开放非安全系统配置，具体请参见**PodSecurityPolicy配置**。
	1.15版本集群原地升级，如果业务中存在initContainer或使用Istio的场景，则需要注意以下约束： 1.16及以上的kubelet统计**QosClass**和之前版本存在差异，1.15及以下版本仅统计spec.containers下的容器，1.16及以上的kubelet版本会同时统计spec.containers和spec.initContainers下的容器，升级前后会造成Pod的QosClass变化，从而造成Pod中容器重启。	建议参考**表6-13**在升级前修改业务容器的QosClass规避该问题。

版本升级路径	版本差异	建议自检措施
v1.13升级至v1.15	vpc集群升级后，由于网络组件的升级，master节点会额外占一个网段。在Master占用了网段后，无可用容器网段时，新建节点无法分配到网段，调度在该节点的pod会无法运行。	一般集群内节点数量快占满容器网段场景下会出现该问题。例如，容器网段为10.0.0.0/16，可用IP数量为65536，VPC网络IP分配是分配固定大小的网段（使用掩码实现，确定每个节点最多分配多少容器IP），例如上限为128，则此时集群最多支撑65536/128=512个节点，然后去掉Master节点数量为509，此时是1.13集群支持的节点数。集群升级后，在此基础上3台Master节点会各占用1个网段，最终结果就是506台节点。

6.4.6.13 节点 CCE Agent 版本检查

检查项内容

检测当前节点的CCE包管理组件cce-agent是否为最新版本。

解决方案

- **问题场景一： 错误信息为“you cce-agent no update, please restart it”。**

  该问题为cce-agent无需更新，但是没有重启，需要登录节点手动重启cce-agent。

  解决方式：登录节点执行：

  ```
  systemctl restart cce-agent
  ```

  执行完毕后，重新执行升级检查。

- **问题场景二： 错误信息为“your cce-agent is not the latest version”。**

  该问题为cce-agent不是最新版本，自动更新失败，通常由OBS地址失效或组件版本过低引起。

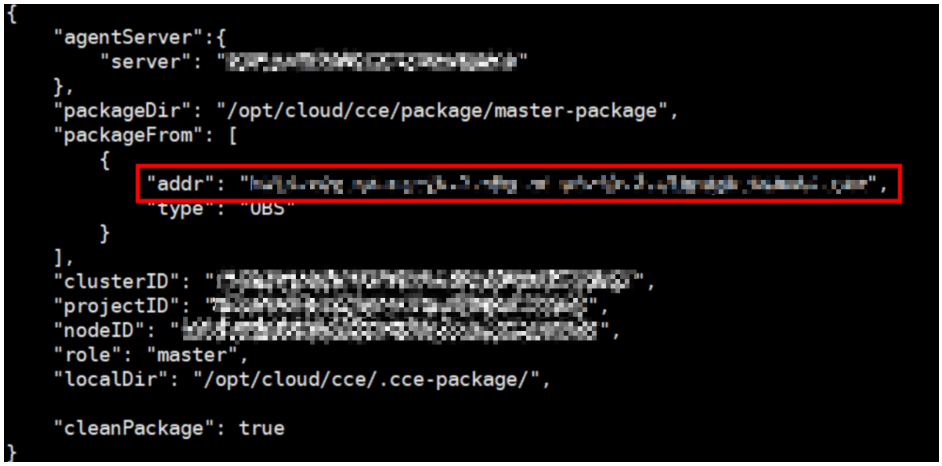
  解决方式：

  a. 登录检查通过的**正常节点**，获取cce-agent配置文件路径，查看有效OBS地址。

  ```
  cat `ps aux | grep cce-agent | grep -v grep | awk -F '-f ' '{print $2}'`
  ```

  配置文件内的OBS配置地址字段为packageFrom.addr

图 6-3 OBS 地址字段



b. 登录检查失败的**异常节点**，参考上一步重新获取OBS地址，检查是否一致。若不一致，请将异常节点的OBS地址修改为正确地址。

c. 通过以下命令下载最新的二进制文件。

- x86系统
  ```
  curl -k "https://{您获取的obs地址}/cluster-versions/base/cce-agent" > /tmp/cce-agent
  ```
- ARM系统
  ```
  curl -k "https://{您获取的obs地址}/cluster-versions/base/cce-agent-arm" > /tmp/cce-agent-arm
  ```

d. 替换原有的cce-agent二进制文件。

- x86系统
  ```
  mv -f /tmp/cce-agent /usr/local/bin/cce-agent
  chmod 750 /usr/local/bin/cce-agent
  chown root:root /usr/local/bin/cce-agent
  ```
- ARM系统
  ```
  mv -f /tmp/cce-agent-arm /usr/local/bin/cce-agent-arm
  chmod 750 /usr/local/bin/cce-agent-arm
  chown root:root /usr/local/bin/cce-agent-arm
  ```

e. 重启cce-agent服务。
```
systemctl restart cce-agent
```

若您对上述执行过程有疑问，请联系技术支持人员。

6.4.6.14 节点 CPU 使用率检查

检查项内容

检查节点CPU使用情况，是否超过90%。

解决方案

- **请在业务低峰时进行集群升级。**
- 请检查该节点的Pod部署数量是否过多，适当驱逐该节点上Pod到其他空闲节点。

6.4.6.15 CRD 检查

检查项内容

当前检查项包括以下内容：

- 检查集群关键CRD "packageversions.version.cce.io"是否被删除。
- 检查集群关键CRD "network-attachment-definitions.k8s.cni.cncf.io"是否被删除。

解决方案

如出现该检查项异常，请联系技术支持人员。

6.4.6.16 节点磁盘检查

检查项内容

当前检查项包括以下内容：

- 检查节点关键数据盘使用量是否满足升级要求
- 检查/tmp目录是否存在500MB可用空间

解决方案

节点升级过程中需要使用磁盘存储升级组件包，使用/tmp目录存储临时文件。

- **问题场景一：磁盘使用量是否满足升级要求**

  请执行以下检查命令，检查当前各关键磁盘的空间使用情况，删除整理确保各可用空间满足要求后，重试检查。若是master节点空间不足，请联系技术支持人员排查处理。

  - docker容器运行时磁盘分区（可用空间需满足 master:2G/node:1G）
    ```
    df -h /var/lib/docker
    ```
  - contianerd容器运行时磁盘分区（可用空间需满足 master:2G/node:1G）
    ```
    df -h /var/lib/containerd
    ```
  - kubelet磁盘分区（可用空间需满足 master:2G/node:1G）
    ```
    df -h /mnt/paas/kubernetes/kubelet
    ```
  - 系统盘（可用空间需满足 master:10G/node:2G）
    ```
    df -h /
    ```

- **问题场景二：/tmp目录空间不足**

  请执行以下检查命令，检查当前/tmp目录所在文件系统的空间使用情况，删除整理确保空间大于500MB后，重试检查。

  ```
  df -h /tmp
  ```

6.4.6.17 节点 DNS 检查

检查项内容

当前检查项包括以下内容：

- 检查当前节点DNS配置是否能正常解析OBS地址

- 检查当前节点是否能访问存储升级组件包的OBS地址

解决方案

节点升级过程中，需要从OBS拉取升级组件包。此项检查失败，请联系技术人员支持。

6.4.6.18 节点关键目录文件权限检查

检查项内容

检查关键目录/var/paas下是否有异常属主和属组的文件。

解决方案

- **问题场景一： 错误信息为“xx file permission has been changed!”。**

  解决方案： CCE使用/var/paas目录进行基本的节点管理活动并存储属主和属组均为paas的文件数据。

  当前集群升级流程会将/var/paas路径下的文件的属主和属组均重置为paas。

  请您排查当前业务Pod中是否将文件数据存储在/var/paas路径下，修改避免使用该路径，并移除该路径下的异常文件后重试检查，否则禁止升级。

- **问题场景二： 错误信息为“user paas must have at least read and execute permissions on the root directory”。**

  解决方案：节点根目录权限被修改导致paas用户没有根目录的读权限，这会导致升级时组件重启失败，建议将根目录权限修正为默认权限555。

6.4.6.19 节点 Kubelet 检查

检查项内容

检查节点kubelet服务是否运行正常。

解决方案

- **问题场景一：kubelet状态异常**

  kubelet异常时，节点显示不可用，修复节点后，重试检查任务。

- **问题场景二：cce-pause版本异常**

  检测到当前kubelet依赖的pause容器镜像版本非cce-pause:3.1，继续升级将会导致批量Pod重启，当前暂不支持升级，请联系技术支持人员。

6.4.6.20 节点内存检查

检查项内容

检查节点内存使用情况，是否超过90%。

解决方案

- **请在业务低峰时进行集群升级。**

- 请检查该节点的Pod部署数量是否过多，适当驱逐该节点上Pod到其他空闲节点。

6.4.6.21 节点时钟同步服务器检查

检查项内容

检查节点时钟同步服务器ntpd或chronyd是否运行正常。

解决方案

- **问题场景一：ntpd运行异常**

  请登录该节点，执行**systemctl status ntpd**命令查询ntpd服务运行状态。若回显状态异常，请执行**systemctl restart ntpd**命令后重新查询状态。

  以下为正常回显：

  图 6-4 ntpd 运行状态

  ```
  [root@[redacted] paas]# systemctl status ntpd
  ● ntpd.service - Network Time Service
     Loaded: loaded (/usr/lib/systemd/system/ntpd.service; enabled; vendor preset: disabled)
     Active: active (running) since Tue 2022-12-06 14:52:30 CST; 4 days ago
   Main PID: 8587 (ntpd)
      Tasks: 2
     Memory: 1.6M
     CGroup: /system.slice/ntpd.service
             └─8587 /usr/sbin/ntpd -u ntp:ntp -g -x
  ```

  若重启ntpd服务无法解决该问题，请联系技术支持人员。

- **问题场景二：chronyd运行异常**

  请登录该节点，执行**systemctl status chronyd**命令查询chronyd服务运行状态。若回显状态异常，请执行**systemctl restart chronyd**命令后重新查询状态。

  以下为正常回显：

  图 6-5 chronyd 运行状态

  ```
  root@[redacted]~# systemctl status chronyd
  ●  chrony.service - chrony, an NTP client/server
     Loaded: loaded (/lib/systemd/system/chrony.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2022-08-24 16:33:28 CST; 3 months 16 days ago
       Docs: man:chronyd(8)
             man:chronyc(1)
             man:chrony.conf(5)
    Process: 6492 ExecStartPost=/usr/lib/chrony/chrony-helper update-daemon (code=exited, status=0/SUCCESS)
    Process: 6461 ExecStart=/usr/lib/systemd/scripts/chronyd-starter.sh $DAEMON_OPTS (code=exited, status=0/SUCCESS)
   Main PID: 6488 (chronyd)
      Tasks: 1 (limit: 4915)
     CGroup: /system.slice/chrony.service
             └─6488 /usr/sbin/chronyd
  ```

  若重启chronyd服务无法解决该问题，请联系技术支持人员。

6.4.6.22 节点 OS 检查

检查项内容

检查节点操作系统内核版本是否为CCE支持的版本。

解决方案

CCE节点运行依赖创建时的初始标准内核版本，CCE基于该内核版本做了全面的兼容性测试，非标准的内核版本可能在节点升级中因兼容性问题导致节点升级失败，详情请参见**高危操作及解决方案**。

当前CCE不建议该类节点进行升级，建议您在升级前**重置节点**至标准内核版本。

6.4.6.23 节点 CPU 数量检查

检查项内容

检查Master节点的CPU数量是否大于2核。

解决方案

Master节点CPU数量为2核时，请联系技术支持人员，将该集群Master节点扩容至4核及以上。

6.4.6.24 节点 Python 命令检查

检查项内容

检查Node节点中Python命令是否可用。

检查方式

```
/usr/bin/python --version
echo $?
```

如果回显值不为0证明检查失败。

解决方案

安装Python之后再进行升级。

6.4.6.25 ASM 网格版本检查

检查项内容

当前检查项包括以下内容：

- 检查集群是否使用ASM网格服务
- 检查当前ASM版本是否支持目标集群版本

解决方案

- 先升级对应的ASM网格版本，再进行集群升级，ASM网格版本与集群版本适配规则如下表。

表 6-17 ASM 网格版本与集群版本适配规则

ASM网格版本	集群版本
1.3	v1.13、v1.15、v1.17、v1.19
1.6	v1.15、v1.17、v1.19、v1.21
1.8	v1.15、v1.17、v1.19、v1.21
1.13	v1.21、v1.23
1.15	v1.21、v1.23、v1.25

- 若不需要使用ASM网格，可删除ASM网格后再进行升级，升级后集群不能绑定与表中不匹配的ASM网格版本。例如，使用v1.21版本集群与1.8版本ASM网格，若要升级至v1.23版本集群时，请先升级ASM网格至1.13版本后再进行v1.23版本集群升级。

6.4.6.26 节点 Ready 检查

检查项内容

检查集群内节点是否Ready。

解决方案

- **问题场景一：节点状态显示不可用**

  请登录CCE控制台，单击集群名称进入集群控制台，前往“节点管理”，筛选出状态不可用的节点后，请参照控制台提供的“修复建议”修复该节点后重试检查。

- **问题场景二：节点状态与实际不符**

  节点状态与实际不符可能存在两种情况：

  a. 控制台“节点管理”处显示正常，但检查结果仍然提示该节点NotReady。请重试检查。

  b. 控制台“节点管理”处无该节点，但检查结果显示集群中仍然存在该节点。请联系技术人员支持。

6.4.6.27 节点 journald 检查

检查项内容

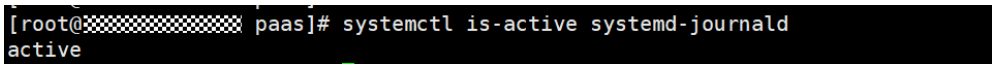
检查节点上的journald状态是否正常。

解决方案

请登录该节点，执行**systemctl is-active systemd-journald**命令查询journald服务运行状态。若回显状态异常，请执行**systemctl restart systemd-journald**命令后重新查询状态。

以下为正常回显：

图 6-6 journald 服务运行状态



若重启journald服务无法解决该问题，请联系技术支持人员。

6.4.6.28 节点干扰 ContainerdSock 检查

检查项内容

检查节点上是否存在干扰的Containerd.Sock文件。该文件影响Euler操作系统下的容器运行时启动。

解决方案

问题场景：节点使用的docker为定制的Euler-docker而非社区的docker

步骤1 登录相关节点。

步骤2 执行**rpm -qa | grep docker | grep euleros**命令，如果结果不为空，说明节点上使用的docker为Euler-docker

步骤3 执行**ls /run/containerd/containerd.sock**命令，若发现存在该文件则会导致docker启动失败。

步骤4 执行**rm -rf /run/containerd/containerd.sock**命令，然后重新进行集群升级检查。

----结束

6.4.6.29 内部错误

检查项内容

在升级前检查流程中是否出现内部错误。

解决方案

如遇该检查任务执行失败，请联系技术支持人员。

6.4.6.30 节点挂载点检查

检查项内容

检查节点上是否存在不可访问的挂载点。

解决方案

问题场景：节点上存在不可访问的挂载点

节点存在不可访问的挂载点，通常是由于该节点或节点上的Pod使用了网络存储nfs（常见的nfs类型有obsfs、sfs等），且节点与远端nfs服务器断连，导致挂载点失效，所有访问该挂载点的进程均会D住卡死。

步骤1 登录节点。

步骤2 节点上依次执行如下命令：

```
- df -h
- for dir in `df -h | grep -v "Mounted on" | awk "{print \\$NF}"`;do cd $dir; done && echo "ok"
```

步骤3 若返回ok则无问题。

否则，请另起一个终端执行如下命令，查询先前命令是否存在D状态：

```
- ps aux | grep "D "
```

步骤4 若发现进程存在D状态，则确认为该问题，目前只可以通过重置节点解决。请选择一个合适的时间重置节点后，重试升级。

说明

重置节点会重置所有节点标签，可能影响工作负载调度，请在重置节点前检查并保留您手动为该节点打上的标签。

----结束

6.4.6.31 K8s 节点污点检查

检查项内容

检查节点上是否存在集群升级需要使用到的污点。

表 6-18 检查污点列表

污点名称	污点影响
node.kubernetes.io/upgrade	NoSchedule

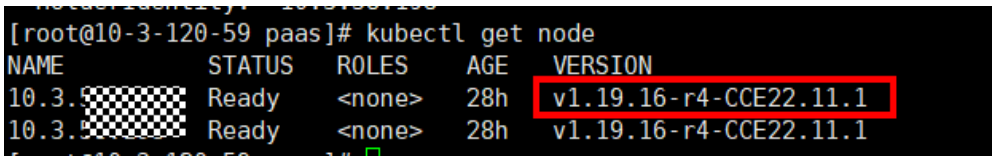
解决方案

问题场景一：该节点为集群升级过程中跳过的节点。

步骤1 配置Kubectl命令，具体请参见通过kubectl连接集群。

步骤2 查看对应节点kubelet版本，以下为正常回显：

图 6-7 kubelet 版本



若该节点的VERSION与其他节点不同，则该节点为升级过程中跳过的节点，请在合适的时间重置节点后，重试检查。

说明

重置节点会重置所有节点标签，可能影响工作负载调度，请在重置节点前检查并保留您手动为该节点打上的标签。

----结束

6.4.6.32 everest 插件版本限制检查

检查项内容

检查集群当前everest插件版本是否存在兼容性限制。

表 6-19 受限的 everest 插件版本

插件名称	涉及版本
everest	v1.0.2-v1.0.7 v1.1.1-v1.1.5

解决方案

检测到当前everest版本存在兼容性限制，无法随集群升级，请联系技术支持人员。

6.4.6.33 cce-hpa-controller 插件限制检查

检查项内容

检查到目标cce-controller-hpa插件版本是否存在兼容性限制。

解决方案

检测到目标cce-controller-hpa插件版本存在兼容性限制，需要集群安装能提供metric api的插件，例如metric-server。

6.4.6.34 增强型 CPU 管理策略检查

检查项内容

检查当前集群版本和要升级的目标版本是否支持增强型CPU管理策略。

解决方案

问题场景：当前集群版本使用增强型CPU管理策略功能，要升级的目标集群版本不支持增强型CPU管理策略功能。

升级到支持增强型CPU管理策略的集群版本，支持增强型CPU管理策略的集群版本如下表所示：

表 6-20 支持增强型 CPU 管理策略的集群版本列表

集群版本	是否支持增强型CPU管理策略功能
v1.17及以下版本	不支持
v1.19	不支持
v1.21	不支持
v1.23及以上版本	支持

6.4.6.35 用户节点组件健康检查

检查项内容

检查用户节点的容器运行时组件和网络组件等是否健康。

解决方案

检测到用户节点存在组件异常时，请登录节点查看异常组件状态并处理。

6.4.6.36 控制节点组件健康检查

检查项内容

检查控制节点的Kubernetes组件、容器运行时组件、网络组件等是否健康。

解决方案

检测到控制节点存在组件异常时，请联系技术支持人员进行处理。

6.4.6.37 K8s 组件内存资源限制检查

检查项内容

检查K8s组件例如etcd、kube-controller-manager等组件是否资源超出限制。

解决方案

- 方案一：适当减少K8s资源
- 方案二：**扩大集群规格**

6.4.6.38 K8s 废弃 API 检查

检查项内容

系统会扫描过去一天的审计日志，检查用户是否调用目标K8s版本已废弃的API。

说明

由于审计日志的时间范围有限，该检查项仅作为辅助手段，集群中可能已使用即将废弃的API，但未在过去一天的审计日志中体现，请您充分排查。

解决方案

检查说明

根据检查结果，检测到您的集群通过kubectl或其他应用调用了升级目标集群版本已废弃的API，您可在升级前进行整改，否则升级到目标版本后，该API将会被kube-apiserver拦截，影响您的使用。具体每个API废弃情况可参考**废弃API说明**。

案例介绍

社区v1.22版本集群废弃了extensions/v1beta1和networking.k8s.io/v1beta1 API 版本的 Ingress ，若您从v1.19或v1.21版本的CCE集群升级到v1.23版本，原有已创建的资源不受影响，但新建与编辑场景将会遇到v1beta1 API 版本被拦截的情况。

具体yaml配置结构变更可参考文档**通过Kubectl命令行创建ELB Ingress**。

6.4.6.39 CCE Turbo 集群 IPv6 能力检查

检查项内容

如CCE Turbo集群启用IPv6，需检查目标集群版本是否支持IPv6。

解决方案

CCE Turbo集群从v1.23版本开始支持IPv6，开放该特性的版本号如下：

- v1.23集群：1.23.8-r0及以上
- v1.25集群：1.25.3-r0及以上
- v1.25以上集群

如果升级前集群已开启IPv6，则目标集群的版本同样需要支持IPv6，请根据上述版本号选择合适的集群版本。

6.4.6.40 节点 NetworkManager 检查

检查项内容

检查节点上的NetworkManager状态是否正常。

解决方案

请登录该节点，执行**systemctl is-active NetworkManager**命令查询NetworkManager服务运行状态。若回显状态异常，请执行**systemctl restart NetworkManager**命令后重新查询状态。

若重启**NetworkManager**服务无法解决该问题，请联系技术支持人员。

6.4.6.41 节点 ID 文件检查

检查项内容

检查节点的ID文件内容是否符合格式。

解决方案

步骤1 在CCE控制台上的"节点管理"页面，单击异常节点名称进入ECS界面。

步骤2 复制节点ID，保存到本地。

步骤3 登录异常节点，备份文件 。

```
cp /var/lib/cloud/data/instance-id /tmp/instance-id
cp /var/paas/conf/server.conf /tmp/server.conf
```

步骤4 登录异常节点，将获取的节点ID写入文件。

```
echo "节点ID" > /var/lib/cloud/data/instance-id
echo "节点ID" > /var/paas/conf/server.conf
```

----结束

6.4.6.42 节点配置一致性检查

检查项内容

在升级CCE集群版本至v1.19及以上版本时，将对您的节点上的Kubenertes组件的配置进行检查，检查您是否后台修改过配置文件。

- /opt/cloud/cce/kubernetes/kubelet/kubelet
- /opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml
- /opt/cloud/cce/kubernetes/kube-proxy/kube-proxy
- /etc/containerd/default_runtime_spec.json
- /etc/sysconfig/docker
- /etc/default/docker
- /etc/docker/daemon.json

如您对这些文件的某些参数进行修改，有可能会导致集群升级失败或升级之后业务出现异常。如您确认该修改对业务无影响，可单击确认后继续进行升级操作。

说明

CCE采用标准镜像的脚本进行节点配置一致性检查，如您使用其它自定义镜像有可能导致检查失败。

当前可预期的修改将不会进行拦截，可预期修改的参数列表如下：

表 6-21 可预期修改的参数列表

组件	配置文件	参数	升级版本
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	cpuManagerPolicy	v1.19以上

组件	配置文件	参数	升级版本
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	maxPods	v1.19以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	kubeAPIQPS	v1.19以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	kubeAPIBurst	v1.19以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	podPidsLimit	v1.19以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	topologyManagerPolicy	v1.19以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	resolvConf	v1.19以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	eventRecordQPS	v1.21以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	topologyManagerScope	v1.21以上
kubelet	/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	allowedUnsafeSysctls	v1.19以上
docker	/etc/docker/daemon.json	dm.basesize	v1.19以上

解决方案

如您对这些文件的某些参数进行修改，有可能导致升级之后出现异常情况。如果您不能确认自行修改的参数是否会影响到升级，请联系技术人员确认。

6.4.6.43 节点配置文件检查

检查项内容

检查节点上关键组件的配置文件是否存在。

当前检查文件列表如下：

文件名	文件内容	备注
/opt/cloud/cce/kubernetes/kubelet/kubelet	kubelet命令行启动参数	-
/opt/cloud/cce/kubernetes/kubelet/kubelet_config.yaml	kubelet启动参数配置	-
/opt/cloud/cce/kubernetes/kube-proxy/kube-proxy	kube-proxy命令行启动参数	-

文件名	文件内容	备注
/etc/sysconfig/docker	docker配置文件	containerd运行时或Debain-Group机器不检查
/etc/default/docker	docker配置文件	containerd运行时或Centos-Group机器不检查

解决方案

请联系技术支持人员恢复配置文件后进行升级。

6.4.6.44 CoreDNS 配置一致性检查

检查项内容

检查当前CoreDNS关键配置Corefile是否同Helm Release记录存在差异，差异的部分可能在插件升级时被覆盖，**影响集群内部域名解析**。

解决方案

您可在明确差异配置后，单独升级CoreDNS插件。

步骤1 配置Kubectl命令，具体请参见**通过kubectl连接集群**。

步骤2 获取当前生效的Corefile。

```
kubectl get cm -nkube-system coredns -o jsonpath='{.data.Corefile}' > corefile_now.txt
cat corefile_now.txt
```

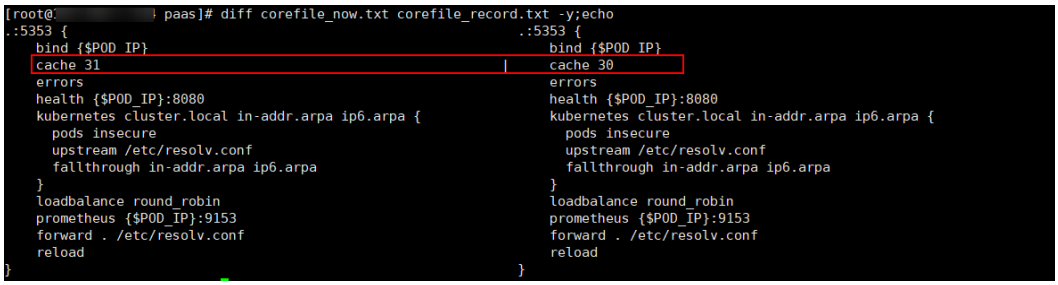
步骤3 获取Helm Release记录中的Corefile(依赖python 3)。

```
latest_release=`kubectl get secret -nkube-system -l owner=helm -l name=cceaddon-coredns --sort-by=.metadata.creationTimestamp | awk 'END{print $1}'`
kubectl get secret -nkube-system $latest_release -o jsonpath='{.data.release}' | base64 -d | base64 -d | gzip -d | python -m json.tool | python -c "
import json,sys,re,yaml;
manifests = json.load(sys.stdin)['manifest']
files = re.split('(?:^|\s*\n)---\s*',manifests)
for file in files:
    if 'coredns/templates/configmap.yaml' in file and 'Corefile' in file:
      corefile = yaml.safe_load(file)['data']['Corefile']
      print(corefile,end='')
      exit(0);
print('error')
exit(1);
" > corefile_record.txt
cat corefile_record.txt
```

步骤4 对比**步骤2**和**步骤3**的输出差异。

```
diff corefile_now.txt corefile_record.txt -y;
```

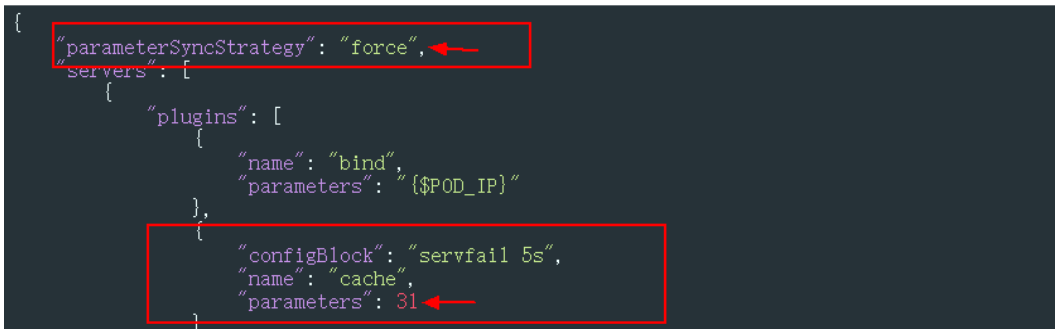
图 6-8 查看输出差异



步骤5 返回CCE控制台，单击集群名称进入集群控制台，前往“插件管理”，选择CoreDNS插件单击“升级”。

若要保留差异部分配置，您可采用以下方法之一：

- 将“parameterSyncStrategy”参数配置为“force”，您需手动填写差异配置，详情请参考CoreDNS。
- 将“parameterSyncStrategy”参数配置为“inherit”，差异配置将自动继承，由系统自动解析、识别与继承差异参数。



步骤6 单击“确定”，等待插件升级完毕，检查CoreDNS各实例均可用，且Corefile符合预期。

```
kubectl get cm -nkube-system coredns -o jsonpath='{.data.Corefile}'
```

步骤7 编辑CoreDNS插件配置的“parameterSyncStrategy”参数，重置为“ensureConsistent”，继续开启配置一致性校验。

同时建议您后续通过CCE插件管理的参数配置功能修改Corefile配置，避免产生差异。

----结束

6.4.6.45 节点 Sudo 检查

检查项内容

检查当前节点sudo命令，sudo相关文件是否正常。

解决方案

- 问题场景一：sudo命令执行失败

  集群原地升级过程中依赖sudo命令正常可用，请登录节点执行如下命令，排查sudo命令可用性。

  ```
  sudo echo hello
  ```

- 问题场景二：关键文件不可修改

  集群原地升级过程中会修改/etc/sudoers文件和/etc/sudoers.d/sudoerspaas文件，以获取sudo权限，更新节点上属主和属组为root的组件（例如docker、kubelet等）与相关配置文件。请登录节点执行如下命令，排查文件的可修改性。

  ```
  lsattr -l /etc/sudoers.d/sudoerspaas /etc/sudoers
  ```

  若回显中存在immutable，则说明文件被加了i锁不可修改，建议您去除i锁。

  ```
  chattr -i /etc/sudoers.d/sudoerspaas /etc/sudoers
  ```

6.4.6.46 节点关键命令检查

检查项内容

检查节点升级依赖的一些关键命令是否能正常执行。

解决方案

- 问题场景一：包管理器命令执行失败

  检查到包管理器命令rpm或dpkg命令执行失败，请登录节点排查下列命令的可用性。

  - rpm系统：
    ```
    rpm -qa
    ```
  - dpkg系统：
    ```
    dpkg -l
    ```
- 问题场景二：systemctl status命令执行失败

  检查到节点systemctl status命令不可用，将影响众多检查项，请登录节点排查下列命令的可用性。

  ```
  systemctl status kubelet
  ```

6.4.6.47 节点 sock 文件挂载检查

检查项内容

检查节点上的Pod是否直接挂载docker/containerd.sock文件。升级过程中Docker/Containerd将会重启，宿主机sock文件发生变化，但是容器内的sock文件不会随之变化，二者不匹配，导致您的业务无法访问Docker/Containerd。Pod重建后sock文件重新挂载，可恢复正常。

通常K8S集群用户基于如下场景在容器中使用上述sock文件：

1. 监控类应用，以DaemonSet形式部署，通过sock文件连接Docker/Containerd，获取节点容器状态信息。
2. 编译平台类应用，通过sock文件连接Docker/Containerd，创建程序编译用容器。

解决方案

- 问题场景一：检查到应用存在该异常，进行整改。

  推荐您使用挂载目录的方式挂载sock文件。例如，若宿主机sock文件路径为/var/run/docker.sock，您可参考下述配置进行整改。注意，该整改生效时会触发Pod重建。

  ```
  kind: Deployment
  apiVersion: apps/v1
  metadata:
    name: test
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: nginx
    template:
      metadata:
        app: nginx
      spec:
  ```

```
containers:
  - name: container-1
    image: 'nginx'
    imagePullPolicy: IfNotPresent
    volumeMounts:
      - name: sock-dir
        mountPath: /var/run
imagePullSecrets:
  - name: default-secret
volumes:
  - name: sock-dir
    hostPath:
      path: /var/run
```

- 问题场景二：检查到应用存在该异常，明确应用使用场景后，接受sock短暂不可访问风险，继续升级。

  请选择跳过该检查项异常后重新检查，在集群升级完成后删除存量Pod，触发Pod重建，访问将恢复。

- 问题场景三：部分老版本的CCE插件存在该异常

  请将老版本的CCE插件升级至最新版本。例如1.2.2以下的dolphin插件存在该问题，需升级至1.2.2及以上版本。

- 问题场景四：日志分析里面出现“failed to execute docker ps -aq”错误。

  该报错出现一般是因为容器引擎功能异常导致，请您提工单联系运维人员处理。

6.4.6.48 HTTPS 类型负载均衡证书一致性检查

检查项内容

检查HTTPS类型负载均衡所使用的证书，是否在ELB服务侧被修改。

解决方案

该问题的出现，一般是由于用户在CCE中创建HTTPS类型Ingress后，直接在ELB证书管理功能中修改了Ingress引用的证书，导致CCE集群中存储的证书内容与ELB侧不一致，进而导致升级后ELB侧证书被覆盖。

步骤1 请登录ELB服务控制台，在“弹性负载均衡 > 证书管理”界面找到该证书，在证书描述字段中找到对应的secret_id。

该secret_id即为集群中对应Secret的metadata.uid字段，可以根据该uid查询集群中Secert的名称。

您可以通过以下kubectl命令进行查询，其中*<secret_id>*请自行替换。

```
kubectl get secret --all-namespaces -o jsonpath='{range .items[*]}{"uid:"}{.metadata.uid}{" namespace:"}{.metadata.namespace}{" name:"}{.metadata.name}{"\n"}{end}' | grep <secret_id>
```

步骤2 仅v1.19.16-r2、v1.21.5-r0、v1.23.3-r0及以上版本的集群支持使用ELB服务中的证书，上述版本集群请参考**方案一**处理，其他版本集群请参考**方案二**处理。

- 方案一：您可以将Ingress使用的证书替换为ELB服务器证书，即可通过ELB控制台创建或编辑该证书。

  a. 请登录CCE控制台，前往“服务发现”页面并选择“路由”页签，找到使用该证书的路由，单击“更多 > 更新”。注意，这里可能有多个Ingess引用该证书，所涉及的Ingress都需要进行更新，可以根据Ingress的yaml文件的spec.tls中secertName字段判断是否引用该Secret中的证书。

  您可以通过以下kubectl命令进行查询引用该证书的Ingress，其中*<secret_name>*请自行替换。

```
kubectl get ingress --all-namespaces -o jsonpath='{range .items[*]}{"namespace:"}
{.metadata.namespace}{" name:"}{.metadata.name}{" tls:"}{.spec.tls[*]}{"\n"}{end}' | grep
<secret_name>
```

b. 在监听器配置中，选择服务器证书来源为“ELB服务器证书”，该证书可通过ELB控制台创建或编辑，单击“确定”。

c. 在“配置项与密钥”界面删除对应的Secert，删除前建议先备份。

- 方案二：您可以将Ingress使用的证书，覆写到集群对应的Secret资源中，避免在升级时出现ELB侧证书被更新。

  请登录CCE控制台，前往“配置项与密钥”页面找到该Secret并编辑，填入您正在使用的证书并保存。

----结束

6.4.6.49 节点挂载检查

检查项内容

检查节点上默认挂载目录及软链接是否被手动挂载或修改。

- 节点为非共享磁盘场景
  - CCE默认挂载/var/lib/docker或containerd、/mnt/paas/kubernetes/kubelet，检查/var、/var/lib、/mnt、/mnt/paas、/mnt/paas/kubernetes是否被用户挂载。
  - CCE默认创建链接/var/lib/kubelet -> /mnt/paas/kubernetes/kubelet，检查是否被用户修改。
- 节点为共享磁盘场景
  - CCE默认挂载/mnt/paas/，检查/mnt是否被用户挂载。
  - CCE默认创建软链接/var/lib/kubelet -> /mnt/paas/kubernetes/kubelet、/var/lib/docker或containerd-> /mnt/paas/runtime，检查是否被用户修改。

解决方案

如何确认是否共享磁盘

步骤1 根据检查信息，登录相应节点。

步骤2 执行**lsblk**命令，查看/mnt/paas挂载了vgpaas-share分区，若存在则是共享磁盘场景，若不存在，则是非共享磁盘场景。

图 6-9 查询是否为共享磁盘

```
[root@test-os-ugrade-35777 ~]# lsblk
NAME           MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda            253:0    0   50G  0 disk
└─vda1         253:1    0   50G  0 part /
vdb            253:16   0  100G  0 disk
└─vgpaas-share 252:0    0  100G  0 lvm  /mnt/paas
```

----结束

节点挂载检查异常如何解决

1. 取消手动修改的挂载点。
2. 取消默认软链接修改。

6.4.6.50 节点 paas 用户登录权限检查

检查项内容

检查paas用户是否有登录权限。

解决方案

执行以下命令查看paas用户是否有登录权限：

```
sudo grep "paas" /etc/passwd
```

如果paas用户权限中带有"nologin"或者"false"，说明paas用户没有登录权限，需要先恢复paas用户的登录权限命令。

执行以下命令恢复paas用户权限之后重新检查：

```
usermod -s /bin/bash paas
```

6.4.6.51 ELB IPv4 私网地址检查

检查项内容

检查集群内负载均衡类型的Service所关联的ELB实例是否包含IPv4私网IP。

解决方案

解决方案一：删除关联无IPv4私网地址ELB的负载均衡型Service。

解决方案二：为无IPv4私网IP地址的ELB绑定一个私网IP。步骤如下：

步骤1 查找负载均衡类型的Service所关联的ELB。

- 方法一：通过升级前检查的日志信息中，获取对应的ELB ID。然后前往ELB控制台通过ELB ID进行筛选。
  elbs (**ids: [*****]**) without ipv4 private ip, please bind private ip tothese elbs and try again
- 方法二：登录CCE控制台，前往“服务发现 > 服务”处查看服务，单击ELB名称，跳转到ELB界面。

步骤2 确认ELB实例是否包含IPv4私网IP。

步骤3 为无IPv4私网IP地址的ELB绑定一个私网IP。

1. 登录CCE控制台，单击目标ELB名称。
2. 在基本信息页面，单击“IPv4私有IP”旁的“绑定”。
3. 设置子网及IPv4地址，并单击“确定”。

----结束

6.4.6.52 检查历史升级记录是否满足升级条件

检查项内容

检查集群最初版本是否小于v1.11，且升级的目标版本大于v1.23。

解决方案

检查到该集群最初版本小于v1.11，连续升级风险较大，请联系技术支持人员。

6.4.6.53 检查集群管理平面网段是否与主干配置一致

检查项内容

检查集群管理平面网段是否与主干配置一致。

解决方案

检查到该集群中的管理平面网段与主干配置中的管理平面网段不一致，请联系技术支持人员。

6.4.6.54 GPU 插件检查

检查项内容

检查到本次升级涉及GPU插件，可能影响新建GPU节点时GPU驱动的安装。

解决方案

由于当前GPU插件的驱动配置由您自行配置，需要您验证两者的兼容性。建议您在测试环境验证安装升级目标版本的GPU插件，并配置当前GPU驱动后，测试创建节点是否正常使用。

您可以执行以下步骤确认GPU插件的升级目标版本与当前驱动配置。

步骤1 登录CCE控制台，前往“集群信息->运维->插件管理”处查看GPU插件。

说明

gpu-beta插件与gpu-device-plugin插件为同一插件。gpu-beta插件在2.0.0版本后，正式更名为gpu-device-plugin。

步骤2 单击该插件的“升级”按钮，查看插件目标版本及驱动配置。

步骤3 在测试环境验证安装升级目标版本的GPU插件，并配置当前GPU驱动后，测试创建节点是否正常使用。

如果两者不兼容，您可以尝试替换高版本驱动。如有需要，请联系技术支持人员。

----结束

6.4.6.55 节点系统参数检查

检查项内容

检查您节点上默认系统参数是否被修改。

解决方案

如您的bms节点上bond0网络的mtu值非默认值1500，将出现该检查异常。

非默认参数可能导致业务丢包，请改回默认值。

6.4.6.56 残留 packageversion 检查

检查项内容

检查当前集群中是否存在残留的packageversion。

解决方案

检查提示您的集群中存在残留的CRD资源10.12.1.109，该问题一般由于CCE早期版本节点删除后，对应的CRD资源未被清除导致。

您可以尝试手动执行以下步骤：

步骤1 备份残留的CRD资源。10.12.1.109 为示例资源，请根据报错中提示的资源进行替换。

```
kubectl get packageversion 10.12.1.109 -oyaml > /tmp/packageversion-109.bak
```

步骤2 清除残留的CRD资源。

```
kubectl delete packageversion 10.12.1.109
```

步骤3 上述步骤执行完成之后尝试重新检查。

----结束

6.4.6.57 节点命令行检查

检查项内容

检查节点中是否存在升级所必须的命令。

解决方案

该问题一般由于节点上缺少集群升级流程中使用到的关键命令，可能会导致集群升级失败。

报错信息如下：

```
__error_code#ErrorCommandNotExist#chage command is not exists#__
__error_code#ErrorCommandNotExist#chown command is not exists#__
__error_code#ErrorCommandNotExist#chmod command is not exists#__
__error_code#ErrorCommandNotExist#mkdir command is not exists#__
__error_code#ErrorCommandNotExist#in command is not exists#__
__error_code#ErrorCommandNotExist#touch command is not exists#__
__error_code#ErrorCommandNotExist#pidof command is not exists#__
```

以上报错代表您的节点上缺少了chage、chown、chmod、mkdir、in、touch、pidof等命令，请安装对应命令之后重新检查。

6.4.6.58 节点交换区检查

检查项内容

检查集群节点上是否开启交换区。

解决方案

CCE节点默认关闭swap交换区，请您确认手动开启交换区的原因，并确定关闭影响。确定后请执行**swapoff -a**命令关闭。

6.4.6.59 nginx-ingress 插件升级检查

检查项内容

检查集群中是否存在未指定Ingress类型（annotations中未添加kubernetes.io/ingress.class: nginx）的Nginx Ingress路由。

问题自检

针对Nginx类型的Ingress资源，查看对应Ingress的YAML，如Ingress的YAML中未指定Ingress类型，并确认该Ingress由Nginx Ingress Controller管理，则说明该Ingress资源存在风险。

步骤1 获取Ingress类别。

您可以通过如下命令获取Ingress类别：

```
kubectl get ingress <ingress-name> -oyaml | grep -E ' kubernetes.io/ingress.class: | ingressClassName:'
```

- 故障场景：如果上述命令输出为空，说明Ingress资源未指定类别。
- 正常场景：Ingress已通过annotations或ingressClassName指定其类别，即存在输出。

```
[root@ + + + + paas]# kubectl get ingress test -oyaml | grep -E ' kubernetes.io/ingress.class: | ingressClassName:' -B 1
Warning: extensions/v1beta1 Ingress is deprecated in v1.14+, unavailable in v1.22+; use networking.k8s.io/v1 Ingress
  annotations:
    kubernetes.io/ingress.class: nginx
--
spec:
  ingressClassName: nginx
```

步骤2 确认该Ingress被Nginx Ingress Controller纳管。如果使用ELB类型的Ingress则无此问题。

- 1.19集群可由通过managedFields机制确认。

```
kubectl get ingress <ingress-name> -oyaml | grep 'manager: nginx-ingress-controller'
```

```
[root@192-168-0-31 paas]# kubectl get ingress test -oyaml | grep 'manager: nginx-ingress-controller'
Warning: extensions/v1beta1 Ingress is deprecated in v1.14+, unavailable in v1.22+; use networking.k8s.io/v1 Ingress
    manager: nginx-ingress-controller
```

- 其他版本集群可通过Nginx Ingress Controller Pod的日志确认。

```
kubectl logs -nkube-system cceaddon-nginx-ingress-controller-545db6b4f7-bv74t | grep 'updating Ingress status'
```

```
[root@ paas]# kubectl logs -nkube-system   cceaddon-nginx-ingress-controller-545db6b4f7-bv74t | grep 'updating Ingress status'
 + + + + + + + +       8 status.go:281] "updating Ingress status" namespace="default" ingress="test" currentValue=[] newValue=[{IP: + + + + + Hostname: Ports:[]} {IP: + + + + Hostname: Ports:[]}]
```

若通过上述两种方式仍然无法确认，请联系技术支持人员。

----结束

解决方案

为Nginx类型的Ingress添加注解，方式如下：

```
kubectl annotate ingress <ingress-name> kubernetes.io/ingress.class=nginx
```

须知

ELB类型的Ingress无需添加该注解，请**确认**该Ingress被Nginx Ingress Controller纳管。

问题根因

NGINX Ingress控制器插件基于开源社区Nginx Ingress Controller的模板与镜像。

对于社区较老版本的Nginx Ingress Controller来说（社区版本v0.49及以下，对应CCE插件版本v1.x.x），在创建Ingress时没有指定Ingress类别为nginx，即annotations中未添加kubernetes.io/ingress.class: nginx的情况，也可以被Nginx Ingress Controller纳管。详情请参见**社区代码**。

但对于较新版本的Nginx Ingress Controller来说（社区版本v1.0.0及以上，对应CCE插件版本2.x.x），如果在创建Ingress时没有显示指定Ingress类别为nginx，该资源将被Nginx Ingress Controller忽略，Ingress规则失效，导致服务中断。详情请参见请参见**社区代码**。

社区相关PR链接为：**https://github.com/kubernetes/ingress-nginx/pull/7341**

目前有两种方式指定Ingress类别：

- 通过annotations指定，为Ingress资源添加annotations（kubernetes.io/ingress.class: nginx）。
- 通过spec指定，.spec.ingressClassName字段配置为nginx。但需要配套具有IngressClass资源。

示例如下：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  ingressClassName: nginx
  rules:
    ...
status:
  loadBalancer: {}
```

6.5 管理集群

6.5.1 集群配置管理

操作场景

CCE支持对集群配置参数进行管理，通过该功能您可以对核心组件进行深度配置。

约束与限制

本功能仅支持在**v1.15及以上版本**的集群中使用，v1.15以下版本不显示该功能。

操作步骤

步骤1 登录CCE控制台，在左侧导航栏中选择“集群管理”。

步骤2 单击集群后的  。

步骤3 在侧边栏滑出的“配置管理”窗口中，根据业务需求修改Kubernetes的参数值：

表 6-22 kube-apiserver 组件配置参数

参数	详情	取值
default-not-ready-toleration-seconds	表示节点处于NotReady状态下的容忍时间。 默认情况下，每个Pod会添加该容忍度。	默认：300s
default-unreachable-toleration-seconds	表示节点处于unreachable状态下的容忍时间。 默认情况下，每个Pod会添加该容忍度。	默认：300s
max-mutating-requests-inflight	最大mutating并发请求数。 当服务器超过此值时，它会拒绝请求。 0表示无限制。该参数与集群规模相关，不建议修改。	从v1.21版本开始不再支持手动配置，根据集群规格自动配置如下： ● 50和200节点：200 ● 1000节点：500 ● 2000节点：1000

参数	详情	取值
max-requests-inflight	最大non-mutating并发请求数。 当服务器超过此值时，它会拒绝请求。 0表示无限制。该参数与集群规模相关，不建议修改。	从v1.21版本开始不再支持手动配置，根据集群规格自动配置如下： • 50和200节点：400 • 1000节点：1000 • 2000节点：2000
service-node-port-range	NodePort端口范围，修改后需前往安全组页面同步修改节点安全组30000-32767的TCP/UDP端口范围，否则除默认端口外的其他端口将无法被外部访问。	默认： 30000-32767 取值范围： min>20105 max<32768
request-timeout	kube-apiserver组件的默认请求超时时间，请谨慎修改此参数，确保取值合理性，以避免频繁出现接口超时或其他异常。 该参数仅v1.19.16-r30、v1.21.10-r10、v1.23.8-r10、v1.25.3-r10及以上版本集群支持。	默认： 1m0s 取值范围： min>=1s max<=1h
feature-gates: ServerSideApply	kube-apiserver组件ServerSideApply特性开关，详情请参见**服务器端应用（Server-Side Apply）**。 该参数仅v1.19.16-r30、v1.21.10-r10、v1.23.8-r10、v1.25.3-r10及以上版本集群支持。	默认： true
support-overload	集群过载控制开关，开启后将根据控制节点的资源压力，动态调整请求并发量，维护控制节点和集群的可靠性。 该参数仅v1.23及以上版本集群支持。	• false：不启用过载控制 • true：启用过载控制

表 6-23 kube-scheduler 组件配置参数

参数	详情	取值
kube-api-qps	与kube-apiserver通信的qps	• 集群规格为1000节点以下时，默认值100 • 集群规格为1000节点及以上时，默认值200
kube-api-burst	与kube-apiserver通信的burst	• 集群规格为1000节点以下时，默认值100 • 集群规格为1000节点及以上时，默认值200
enable-gpu-share	是否开启GPU共享，该参数仅v1.23.7-r10、v1.25.3-r0及以上版本集群支持。 • 关闭GPU共享时，需保证集群中的Pod没有使用共享GPU能力（即Pod不存在cce.io/gpu-decision的annotation）。 • 开启GPU共享时，需保证集群中已使用GPU资源的Pod均存在cce.io/gpu-decision的annotation。	默认：true

表 6-24 kube-controller-manager 组件配置参数

参数	详情	取值
concurrent-deployment-syncs	deployment的并发处理数	默认：5
concurrent-endpoint-syncs	endpoint的并发处理数	默认：5
concurrent-gc-syncs	garbage collector的并发数	默认：20
concurrent-job-syncs	允许同时同步的作业对象的数量。	默认：5
concurrent-namespace-syncs	namespace的并发处理数	默认：10
concurrent-replicaset-syncs	replicaset的并发处理数	默认：5

参数	详情	取值
concurrent-resource-quota-syncs	resource quota的并发处理数	默认：5
concurrent-service-syncs	service的并发处理数	默认：10
concurrent-serviceaccount-token-syncs	serviceaccount-token的并发处理数	默认：5
concurrent-ttl-after-finished-syncs	ttl-after-finished的并发处理数	默认：5
concurrent-rc-syncs	rc的并发处理数 **说明** 该参数仅在v1.21至v1.23版本集群中使用。v1.25版本后，该参数弃用（正式弃用版本为v1.25.3-r0）。	默认：5
horizontal-pod-autoscaler-sync-period	集群弹性计算的周期	默认：15s
kube-api-qps	与kube-apiserver通信的qps	• 集群规格为1000节点以下时，默认值100 • 集群规格为1000节点及以上时，默认值200
kube-api-burst	与kube-apiserver通信的burst	• 集群规格为1000节点以下时，默认值100 • 集群规格为1000节点及以上时，默认值200
terminated-pod-gc-threshold	在Pod GC开始删除终止Pod之前，可以存在的terminated状态Pod数量。 如果<=0，则禁用终止的Pod GC。	默认：1000

表 6-25 网络组件配置参数（仅 CCE Turbo 集群支持）

参数	详情	取值
nic-minimum-target	集群级别的节点最少绑定容器网卡数	默认：10

参数	详情	取值
nic-maximum-target	集群级别的节点预热容器网卡上限检查值	默认：0
nic-warm-target	集群级别的节点动态预热容器网卡数	默认：2
nic-max-above-warm-target	集群级别的节点预热容器网卡回收阈值	默认：2

表 6-26 扩展控制器配置参数（仅 v1.21 及以上版本集群支持）

参数	详情	取值
enable-resource-quota	创建namespace时是否自动创建resourcequota对象。 • false：不自动创建resourcequota对象。 • true：自动创建resourcequota对象。resourcequota的默认取值请参见**设置资源配额及限制**。	默认：false

步骤4 单击“确定”，完成配置操作。

----结束

参考链接

- **kube-apiserver**
- **kube-controller-manager**
- **kube-scheduler**

6.5.2 集群过载控制

操作场景

过载控制开启后，将根据控制节点的资源压力，动态调整请求并发量，维护控制节点和集群的可靠性。

约束与限制

集群版本需为v1.23及以上。

开启集群过载控制

方式一：创建集群时开启

创建v1.23及以上集群时，可在创建集群过程中，开启过载控制选项。

方式二：已有集群中开启

步骤1 登录CCE控制台，进入一个已有的集群（集群版本为v1.23及以上）。

步骤2 在“集群信息”页面，查看控制节点信息，如集群未开启过载控制，此处将会出现相应提示。如需开启过载控制，您可单击“立即开启”。

----结束

关闭集群过载控制

步骤1 登录CCE控制台，进入一个已有的集群（集群版本为v1.23及以上）。

步骤2 在“集群信息”页面，单击右上角“配置管理”。

步骤3 在“kube-apiserver组件配置”中将support-overload参数设置为false。

步骤4 单击“确定”保存修改。

----结束

6.5.3 变更集群规格

操作场景

当前集群管理规模可支持管理的用户节点个数不能满足用户诉求，可通过“变更集群规格”功能来扩大使用的用户节点个数。

约束限制

- 集群v1.15及以上版本支持变更集群规格。
- 集群v1.15.11开始支持变更到2000节点，单控制节点的集群不允许变更到1000节点及以上。
- 变更集群规格目前只支持扩容到更大规格，不支持降低集群规格。
- 规格变更期间，控制节点存在开关机动作，集群将无法正常使用，请尽量在业务平稳期执行变更操作。
- 集群规格变更不会影响集群中已运行业务，但变更过程中管理面（Master节点）会有短暂中断，建议变更期间停止其他操作（如创建工作负载等）。
- 规格变更失败后集群将尽可能回退到正常状态，若回退异常可提交工单进行处理。

操作步骤

步骤1 登录CCE控制台，在左侧导航栏中选择“集群管理”。

步骤2 单击需要变更规格集群后的  。

步骤3 在弹出的页面中，根据实际需求选择新的“集群管理规模”。

步骤4 单击“下一步”进行规格确认，并单击“确定”。

您可以单击在左上角单击“操作记录”查看集群变更记录。状态从“执行中”变为“成功”，表示集群规格变更成功。

----结束

6.5.4 删除集群

注意事项

- 删除集群会删除集群下的节点（纳管节点不会被删除）、节点挂载的数据盘、工作负载与服务，相关业务将无法恢复。在执行操作前，请确保相关数据已完成备份或者迁移，删除完成后数据无法找回，请谨慎操作。

  部分不是在CCE中创建的资源不会删除：

  - 纳管的节点（仅删除在CCE中创建的节点）
  - Service和Ingress关联的ELB实例（仅删除自动创建的ELB实例）
  - 手动创建PV关联的云存储/导入的云存储（仅删除PVC自动创建的云存储）
- 在集群非运行状态（例如不可用状态）时删除集群，会残留存储、网络等关联资源，请妥善处理。

删除集群

须知

处于休眠状态的集群无法直接删除，请将集群唤醒后重试。

步骤1 登录CCE控制台，在左侧导航栏中选择“集群管理”。

步骤2 单击待删除集群后的 。

步骤3 在弹出的“删除集群”窗口中，根据系统提示，勾选删除集群时需要释放的资源。

- 删除集群下工作负载挂载的云存储。

  说明

  选择删除集群中存储卷绑定的底层云存储资源时，存在如下约束：

  - 底层存储依据存储卷指定的回收策略进行删除。例如，存储卷指定回收策略为Retain，则在删除集群后底层存储会保留。
  - 对象存储桶下存在大量文件（超过1000）时，请先手动清理桶内文件后再执行集群删除操作。
- 删除集群下负载均衡ELB等网络资源（仅删除自动创建的ELB资源）。

步骤4 单击“是”，开始执行删除集群操作。

删除集群需要花费1~3分钟，请耐心等候。

----结束

6.5.5 休眠与唤醒集群

操作场景

暂时不需要使用集群时，建议您将集群设置为休眠状态。

集群休眠后，将无法在此集群上创建和管理工作负载等资源。

休眠中的集群可以快速唤醒，正常使用。

约束与限制

集群唤醒过程中，可能会由于资源不足导致Master节点启动失败，从而导致集群唤醒失败，请过一段时间再次唤醒。

集群休眠

步骤1 登录CCE控制台，在左侧导航栏中选择“集群管理”。

步骤2 单击待休眠集群后的  。

步骤3 在弹出的集群休眠提示框中，查看风险提示，单击“是”，等待集群完成休眠。

集群休眠后，将暂停收取控制节点资源费用。集群所属的工作节点（ECS）、绑定的弹性IP、带宽等资源仍将按各自的计费方式进行收费。如需关机节点，请在集群休眠提示框中勾选“关机集群下所有节点”或参见**节点关机**。

大部分节点关机后不再收费，特殊ECS实例（包含本地硬盘，如磁盘增强型，超高I/O型等）关机后仍然正常收费。

----结束

集群唤醒

步骤1 登录CCE控制台，在左侧导航栏中选择“集群管理”。

步骤2 单击待唤醒集群后的  。

步骤3 当集群状态由“唤醒中”变为“运行中”时，即完成唤醒操作，唤醒集群预计需要3-5分钟。

说明

集群唤醒后，将继续收取集群管理费用。

----结束

7 节点

7.1 节点概述

简介

节点是容器集群组成的基本元素。节点取决于业务，既可以是虚拟机，也可以是物理机。每个节点都包含运行Pod所需要的基本组件，包括Kubelet、Kube-proxy 、Container Runtime等。

说明

CCE创建的Kubernetes集群包含master节点和node节点，本章讲述的节点特指**node节点**，node节点是集群的计算节点，即运行容器化应用的节点。

在云容器引擎CCE中，主要采用高性能的弹性云服务器ECS作为节点来构建高可用的Kubernetes集群。

支持的节点规格

不同区域支持的节点规格（flavor）不同，且节点规格存在新增、下线等情况，建议您在使用前登录CCE控制台，在创建节点界面查看您需要的节点规格是否支持。

Docker 容器底层文件存储系统说明

- 1.15.6及之前集群版本docker底层文件存储系统采用xfs格式。
- 1.15.11及之后版本集群新建节点或重置后docker底层文件存储系统全部采用ext4格式。

对于之前使用xfs格式容器应用，需要注意底层文件存储格式变动影响（不同文件系统格式文件排序存在差异：如部分java应用引用某个jar包，但目录中存在多个版本该jar包，在不指定版本时实际引用包由系统文件排序决定）。

查看当前节点使用的docker底层存储文件格式可采用docker info | grep "Backing Filesystem"确认。

节点 paas 用户/用户组说明

在CCE集群中创建节点时，默认会在节点上创建paas用户/用户组。节点上的CCE组件和CCE插件在非必要时会以非root用户（paas用户/用户组）运行，以实现运行权限最小化，如果修改paas用户/用户组可能会影响节点上CCE组件和业务Pod正常运行。

须知

CCE组件正常运行依赖paas用户/用户组，您需要注意以下几点要求：

- 请勿自行修改节点内目录权限、容器目录权限等。
- 请勿自行修改paas用户/用户组的GID和UID。
- 请勿在业务中直接使用paas用户/用户组设置业务文件的所属用户和组。

节点生命周期

生命周期是指节点从创建到删除（或释放）历经的各种状态。

表 7-1 节点生命周期状态说明

状态	状态属性	说明
运行中	稳定状态	节点正常运行状态，并且已连接上集群。 在这个状态的节点可以运行您的业务。
不可用	稳定状态	节点运行异常状态。 在这个状态下的实例，不能对外提供业务，需要**重置节点**。
创建中	中间状态	创建节点实例后，在节点状态进入运行中之前的状态。
安装中	中间状态	节点处于安装Kubernetes软件的过程中。
删除中	中间状态	节点处于正在被删除的状态。 如果长时间处于该状态，则说明出现异常。
关机	稳定状态	节点被正常停止。 在这个状态下的实例，不能对外提供业务，您可以在弹性云服务器列表页对其进行开机操作。
错误	稳定状态	节点处于异常状态。 在这个状态下的实例，不能对外提供业务，需要**重置节点**。

7.2 容器引擎

容器引擎介绍

容器引擎是Kubernetes最重要的组件之一，负责管理镜像和容器的生命周期。Kubelet通过Container Runtime Interface (CRI) 与容器引擎交互，以管理镜像和容器。

CCE当前支持用户选择Containerd和Docker容器引擎，**其中Containerd调用链更短，组件更少，更稳定，占用节点资源更少**。

表 7-2 容器引擎对比

对比	Containerd	Docker
调用链	kubelet --> CRI plugin（在containerd进程中） --> containerd	• Docker（Kubernetes 1.23及以下版本）：kubelet --> dockershim（在kubelet 进程中） --> docker --> containerd • Docker（Kubernetes 1.24及以上版本社区方案）：kubelet --> cri-dockerd（kubelet使用CRI接口对接cri-dockerd） --> docker --> containerd
命令	crictl	docker
Kubernetes CRI支持	原生支持	需通过dockershim或cri-dockerd提供CRI支持
Pod 启动延迟	低	高
kubelet CPU/内存占用	低	高
运行时CPU/内存占用	低	高

Containerd 和 Docker 组件常用命令对比

Containerd不支持dockerAPI和dockerCLI，但是可以通过cri-tool命令实现类似的功能。

表 7-3 镜像相关功能

序号	Docker命令	Containerd命令	备注
1	docker images [选项] [镜像名[:标签]]	crictl images [选项] [镜像名[:标签]]	列出本地镜像列表

序号	Docker命令	Containerd命令	备注
2	docker pull [选项] 镜像名[:标签|@DIGEST]	crictl pull [选项] 镜像名[:标签|@DIGEST]	拉取镜像
3	docker push	无	上传镜像
4	docker rmi [选项] 镜像...	crictl rmi [选项] 镜像ID…	删除本地镜像
5	docker inspect 镜像ID	crictl inspecti 镜像ID	检查镜像

表 7-4 容器相关功能

序号	Docker命令	Containerd命令	备注
1	docker ps [选项]	crictl ps [选项]	列出容器列表
2	docker create [选项]	crictl create [选项]	创建容器
3	docker start [选项] 容器ID...	crictl start [选项] 容器ID...	启动容器
4	docker stop [选项] 容器ID...	crictl stop [选项] 容器ID...	停止容器
5	docker rm [选项] 容器ID...	crictl rm [选项] 容器ID...	删除容器
6	docker attach [选项] 容器ID	crictl attach [选项] 容器ID	连接容器
7	docker exec [选项] 容器ID 启动命令 [参数...]	crictl exec [选项] 容器ID 启动命令[参数...]	进入容器
8	docker inspect [选项] 容器NAME|ID...	crictl inspect [选项] 容器ID...	查看容器详情
9	docker logs [选项] 容器ID	crictl logs [选项] 容器ID	查看容器日志
10	docker stats [选项] [容器ID...]	crictl stats [选项] [容器ID]	查看容器的资源使用情况
11	docker update [选项] 容器ID...	crictl update [选项] 容器ID...	更新容器资源限制

表 7-5 Pod 相关功能

序号	Docker命令	Containerd命令	备注
1	无	crictl pods [选项]	列出Pod列表

序号	Docker命令	Containerd命令	备注
2	无	crictl inspectp [选项] POD-ID...	查看Pod详情
3	无	crictl start [选项] POD-ID...	启动Pod
4	无	crictl runp [选项] POD-ID...	运行Pod
5	无	crictl stopp [选项] POD-ID...	停止Pod
6	无	crictl rmp [选项] POD-ID...	删除Pod

说明

Containerd创建并启动的容器会被kubelet立即删除，不支持暂停、恢复、重启、重命名、等待容器，Containerd不具备docker构建、导入、导出、比较、推送、查找、打标签镜像的能力，Containerd不支持拷贝文件，可通过修改containerd的配置文件实现登录镜像仓库。

调用链区别

- Docker（Kubernetes 1.23及以下版本）：

  kubelet --> docker shim （在kubelet 进程中） --> docker --> containerd
- Docker（Kubernetes 1.24及以上版本社区方案）：

  kubelet --> cri-dockerd （kubelet使用cri接口对接cri-dockerd） --> docker --> containerd
- Containerd：

  kubelet --> cri plugin（在containerd进程中） --> containerd

其中Docker虽增加了swarm cluster、docker build、docker API等功能，但也会引入一些bug，并且与Containerd相比，多了一层调用，因此**Containerd被认为更加节省资源且更安全**。

7.3 创建节点

前提条件

- 已创建至少一个集群。
- 您需要新建一个密钥对，用于远程登录节点时的身份认证。

约束与限制

- 集群节点规格要求：CPU必须2核及以上，内存必须4GB及以上。
- 为了保证节点的稳定性，CCE集群节点上会根据节点的规格预留一部分资源给Kubernetes的相关组件（kubelet、kube-proxy以及docker等）和Kubernetes系统资源，使该节点可作为您的集群的一部分。 因此，您的节点资源总量与节点在

Kubernetes中的可分配资源之间会存在差异。节点的规格越大，在节点上部署的容器可能会越多，所以Kubernetes自身需预留更多的资源，详情请参见**节点预留资源策略说明**。

- 节点的网络（如虚机网络、容器网络等）均被CCE接管，请勿自行添加删除网卡或改变路由。若自行修改可能导致服务不可用。例如，节点上名为的gw_11cbf51a@eth0网卡为容器网络网关，不可修改。
- 创建节点过程中会使用域名方式从OBS下载软件包，需要能够使用云上内网DNS解析OBS域名，否则会导致创建不成功。为此，节点所在子网的DNS服务器地址需要配置为内网DNS，从而使得节点使用内网DNS。在创建子网时DNS默认配置为内网DNS，如果您修改过子网的DNS，请务必**确保子网下的DNS服务器可以解析OBS服务域名**，否则需要将DNS改成内网DNS。
- 集群中的节点一旦创建后不可变更可用区。

操作步骤

集群创建完成后，您可以在集群中创建节点。

步骤1 登录CCE控制台。

步骤2 在左侧导航栏中选择“集群管理”，单击要创建节点的集群进入集群控制台。

步骤3 在集群控制台左侧导航栏中选择“节点管理”，单击右侧“创建节点”，在节点配置步骤中参照如下表格设置节点参数。

计算配置：

配置节点云服务器的规格与操作系统，为节点上的容器应用提供基本运行环境。

表 7-6 计算配置参数

参数	参数说明
可用区	节点云服务器所在的可用区，集群下节点创建在不同可用区下可以提高可靠性。创建后不可修改。 建议您选择“随机分配”，可根据选择的节点规格随机分配一个可以使用的可用区。 可用区是在同一区域下，电力、网络隔离的物理区域，可用区之间内网互通，不同可用区之间物理隔离。如果您需要提高工作负载的高可靠性，建议您将云服务器创建在不同的可用区。
节点类型	CCE集群： ● 弹性云服务器-虚拟机：基于弹性云服务器部署容器服务。 ● 弹性云服务器-物理机：基于擎天架构的服务器部署容器服务。 CCE Turbo集群： ● 弹性云服务器-虚拟机：基于弹性云服务器部署容器服务，仅支持可添加多张弹性网卡的机型。 ● 弹性云服务器-物理机：基于擎天架构的服务器部署容器服务。

参数	参数说明
容器引擎	CCE集群支持Docker，并在部分场景中支持Containerd。 ● 1.23及以上的VPC网络集群都支持Containerd，容器隧道网络集群从1.23.2-r0开始支持Containerd。 ● CCE Turbo引擎支持Docker和Containerd。
节点规格	请根据业务需求选择相应的节点规格。 不同的可用区可选择的节点规格类型可能不同，请根据实际情况选择。
操作系统	选择操作系统类型，不同类型节点支持的操作系统有所不同。 ● 公共镜像：请选择节点对应的操作系统。 **说明** ● 由于业务容器运行时共享节点的内核及底层调用，为保证兼容性，建议节点的操作系统选择与最终业务容器镜像相同或接近的Linux发行版本。
节点名称	节点云服务器使用的名称，批量创建时将作为云服务器名称的前缀。 系统会默认生成名称，支持修改。 节点名称以小写字母开头，支持小写字母、数字和中划线(-)，不能以中划线(-)结尾。
登录方式	● **密钥对** 选择用于登录本节点的密钥对，支持选择共享密钥。 密钥对用于远程登录节点时的身份认证。若没有密钥对，可单击选项框右侧的“创建密钥对”来新建。

存储配置：

配置节点云服务器上的存储资源，方便节点上的容器软件与容器应用使用。请根据实际场景设置磁盘大小。

表 7-7 存储配置参数

参数	参数说明
系统盘	节点云服务器使用的系统盘，供操作系统使用。您可以设置系统盘的规格为40GB-1024GB之间的数值，缺省值为50GB。 系统盘加密：系统盘加密功能可为您的数据提供强大的安全防护，加密磁盘生成的快照及通过这些快照创建的磁盘将自动继承加密功能。**目前仅在部分Region显示此选项，具体以界面为准。** ● 默认不加密。 ● 点选“加密”后，可在弹出的“加密设置”对话框中，选择已有的密钥，若没有可选的密钥，请单击后方的链接创建新密钥，完成创建后单击刷新按钮。

参数	参数说明
数据盘	**至少需要一块数据盘**，供容器运行时和Kubelet组件使用，**该数据盘不能被删除卸载，否则会导致节点不可用。** • 第一块数据盘：供容器运行时和Kubelet组件使用。您可以自行设置数据盘的规格为20GB-32768GB之间的数值，缺省值为100GB。 • 其他数据盘，您可以设置数据盘的规格为10GB-32768GB之间的数值，缺省值为100GB。 **说明** 节点规格为“磁盘增强型”或“超高I/O型”时，有一块数据盘可以是本地盘。 本地磁盘实例有宕机风险，不保证数据可靠性，建议您使用云硬盘存储您的业务数据。 **高级配置** 单击后方的“展开高级设置”可进行如下设置： • 数据盘空间分配：勾选“自定义容器引擎空间大小”后可定义容器引擎、镜像、临时存储在数据盘上占用的空间比例。容器引擎空间用于存放容器运行时工作目录、容器镜像数据以及镜像元数据；数据盘的剩余空间用于Pod配置文件、密钥及临时存储EmptyDir等。数据盘空间分配详细说明请参见**数据盘空间分配说明**。 • 数据盘加密：数据盘加密功能可为您的数据提供强大的安全防护，加密磁盘生成的快照及通过这些快照创建的磁盘将自动继承加密功能。**目前仅在部分Region显示此选项，具体以界面为准。** – 默认不加密。 – 勾选“加密”后，可选择已有的密钥。若没有可选的密钥，请单击后方的链接创建新密钥，完成创建后单击刷新按钮。 **添加多个数据盘** 最多可以添加4个，默认情况直接创建为裸盘，不做任何处理。您也可以展开高级配置，选择如下配置。 • 默认：默认情况直接创建为裸盘，不做任何处理。 • 挂载到指定目录：将数据盘挂载到指定目录。 • 作为持久存储卷：适用于对PV有性能要求的场景。持久存储卷的节点会添加上node.kubernetes.io/local-storage-persistent标签，取值为linear或striped。 • 作为临时存储卷：适用于对EmptyDir有性能要求的场景。 **说明** • 本地持久卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=2.1.23，推荐使用>=2.1.23版本。 • 本地临时卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=1.2.29。 **本地持久卷**和**本地临时卷**支持如下两种写入模式。 • 线性（linear）：线性逻辑卷是将一个或多个物理卷整合为一个逻辑卷，实际写入数据时会先往一个基本物理卷上写入，当存储空间占满时再往另一个基本物理卷写入。

参数	参数说明
	• 条带化（striped）：创建逻辑卷时指定条带化，当实际写入数据时会将连续数据分成大小相同的块，然后依次存储在多个物理卷上，实现数据的并发读写从而提高读写性能。条带化模式的存储池不支持扩容。多块存储卷才能选择条带化。

网络配置：

配置节点云服务器的网络资源，用于访问节点和容器应用。

表 7-8 网络配置参数

参数	参数说明
节点子网	节点子网默认使用创建集群时的子网配置，也可以选择其他子网。
节点IP	支持指定节点IP地址。默认随机分配。
弹性公网IP	未绑定弹性公网IP的云服务器无法直接访问外网，无法直接对外进行互相通信。 默认为“暂不使用”。支持“使用已有”和“自动创建”。

高级配置：

节点能力增强，可在此配置节点的标签、污点、启动命令等功能。

表 7-9 高级配置参数

参数	参数说明
K8s标签	设置附加到Kubernetes对象（比如Pod）上的键值对，填写键值对后，单击“确认添加”。最多可以添加20条标签。 使用该标签可区分不同节点，可结合工作负载的亲和能力实现容器Pod调度到指定节点的功能。详细请参见**Labels and Selectors**。

参数	参数说明
污点（Taints）	默认为空。支持给节点加污点来设置反亲和性，每个节点最多配置20条污点，每条污点包含以下3个参数： • Key：必须以字母或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符；另外可以使用DNS子域作为前缀。 • Value：必须以字符或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符。 • Effect：只可选NoSchedule，PreferNoSchedule或NoExecute。 污点的使用请参见**管理节点污点（Taint）**。 **说明** 对于1.19及以下版本集群，有可能会出现污点打上之前负载已经调度到节点上，如果需要避免这种情况，请选择1.19及以上集群。
最大实例数	节点最大可以正常运行的实例数(Pod)，该数量包含系统默认实例。 该设置的目的为防止节点因管理过多实例而负载过重，请根据您的业务需要进行设置。 节点最多能创建多少个Pod还受其他因素影响，具体请参见**节点可创建的最大Pod数量说明**。
云服务器组	云服务器组是对云服务器的一种逻辑划分，同一云服务器组中的云服务器遵从同一策略。 反亲和性策略：同一云服务器组中的云服务器分散地创建在不同主机上，提高业务的可靠性。 选择已创建的云服务器组，或单击“新建云服务器组”创建，创建完成后单击刷新按钮。
安装前执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装前执行，可能导致Kubernetes软件无法正常安装，需谨慎使用。
安装后执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装后执行，不影响Kubernetes软件安装。 **说明** 请不要在安装后执行脚本中使用**reboot**命令立即重启，如果需要重启，可以使用**shutdown -r 1**命令延迟1分钟重启。
委托	委托是由租户管理员在统一身份认证服务上创建的。通过委托，可以将云主机资源共享给其他账号，或委托更专业的人或团队来代为管理。 如果没有委托请单击右侧“新建委托”创建。

步骤4 完成以上配置后，您可以设置需要购买的节点数量，并单击“下一步：规格确认”，确认所设置的服务选型参数、规格等信息。

步骤5 单击“提交”，节点开始创建。

系统将自动跳转到节点列表页面，待节点状态为“运行中”，表示节点添加成功。添加节点预计需要6-10分钟左右，请耐心等待。

步骤6 单击“返回节点列表”，待状态为运行中，表示节点创建成功。

----结束

7.4 纳管节点

操作场景

CCE集群支持两种添加节点的方式：**创建节点**和纳管节点，纳管节点是指将“已有的ECS加入到CCE集群中”。

须知

- 纳管时，会将所选弹性云服务器的操作系统重置为CCE提供的标准镜像，以确保节点的稳定性，请选择操作系统及重置后的登录方式。
- 所选弹性云服务器挂载的系统盘、数据盘都会在纳管时清理LVM信息，包括卷组（VG）、逻辑卷（LV）、物理卷（PV），请确保信息已备份。
- 纳管过程中，请勿在弹性云服务器控制台对所选虚拟机做任何操作。

约束与限制

- 集群版本需1.15及以上。
- 原虚拟机节点创建时若已设置密码或密钥，纳管时您需要重新设置密码或密钥，原有的密码或密钥将会失效。
- 纳管节点时已分区的数据盘会被忽略，您需要保证节点至少有一个未分区且符合规格的数据盘。

前提条件

支持纳管符合如下条件的云服务器：

- 待纳管节点必须状态为“运行中”，未被其他集群所使用，且不携带 CCE 专属节点标签CCE-Dynamic-Provisioning-Node。
- 待纳管节点需与集群在同一虚拟私有云内（若集群版本低于1.13.10，纳管节点还需要与CCE集群在同一子网内）。
- 待纳管节点需挂载数据盘，可使用本地盘（磁盘增强型实例）或至少挂载一块20GiB及以上的数据盘，且不存在10GiB以下的数据盘。
- 待纳管节点规格要求：CPU必须2核及以上，内存必须4GiB及以上，网卡有且仅能有一个。
- 如果使用了企业项目，则待纳管节点需要和集群在同一企业项目下，不然在纳管时会识别不到资源，导致无法纳管。
- 批量纳管仅支持添加相同规格、相同可用区、相同数据盘配置的云服务器。

操作步骤

步骤1 登录CCE控制台，进入要纳管节点的集群。

步骤2 在左侧列表中选择节点管理，单击右上角纳管节点。

步骤3 配置节点参数。

计算配置

表 7-10 计算配置参数

参数	参数说明
节点规格	单击添加已有云服务器，选择要纳管的服务器。 可以选择多台云服务器批量纳管，但批量纳管仅支持添加相同规格、相同可用区、相同数据盘配置的云服务器。 如果云服务器有多块数据盘，需要选择其中一块作为供容器运行时和Kubelet组件使用。
容器引擎	CCE集群支持Docker，并在部分场景中支持Containerd。 • 1.23及以上的VPC网络集群都支持Containerd，容器隧道网络集群从1.23.2-r0开始支持Containerd。 • CCE Turbo引擎支持Docker和Containerd。
操作系统	公共镜像：请选择节点对应的操作系统。
登录方式	• **密钥对** 选择用于登录本节点的密钥对，支持选择共享密钥。 密钥对用于远程登录节点时的身份认证。若没有密钥对，可单击选项框右侧的“创建密钥对”来新建。

存储配置

配置节点云服务器上的存储资源，方便节点上的容器软件与容器应用使用。

表 7-11 存储配置参数

参数	参数说明
系统盘	直接使用云服务器的系统盘。
数据盘	**至少需要一块数据盘**，供容器运行时和Kubelet组件使用，**该数据盘不能被删除卸载，否则会导致节点不可用。** 单击后方的“展开高级设置”可设置自定义空间分配：勾选后可定义容器运行时在数据盘上占用的空间比例，容器运行时的空间用于存放容器运行时工作目录、容器镜像数据以及镜像元数据。数据盘空间分配详细说明请参见**数据盘空间分配说明**。 **其他数据盘**默认情况直接创建为裸盘，不做任何处理。您也可以展开高级配置，将磁盘挂载到指定目录。

高级配置

表 7-12 高级配置参数

参数	参数说明
K8s标签	单击“添加标签”可以设置附加到Kubernetes 对象（比如Pods）上的键值对，最多可以添加20条标签。 使用该标签可区分不同节点，可结合工作负载的亲和能力实现容器Pod调度到指定节点的功能。详细请参见**Labels and Selectors**。
污点（Taints）	默认为空。支持给节点加污点来设置反亲和性，每个节点最多配置20条污点，每条污点包含以下3个参数： • Key：必须以字母或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符；另外可以使用DNS子域作为前缀。 • Value：必须以字符或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符。 • Effect：只可选NoSchedule，PreferNoSchedule或NoExecute。 **须知** • 污点配置时需要配合Pod的toleration使用，否则可能导致扩容失败或者Pod无法调度到扩容节点。 • 节点池创建后可单击列表项的“编辑”修改配置，修改后将同步到节点池下的已有节点。
最大实例数	节点最大可以正常运行的实例数(Pod)，该数量包含系统默认实例。 该设置的目的为防止节点因管理过多实例而负载过重，请根据您的业务需要进行设置。
安装前执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装前执行，可能导致Kubernetes软件无法正常安装，需谨慎使用。
安装后执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装后执行，不影响Kubernetes软件安装。

步骤4 单击“下一步：规格确认”，并单击“提交”。

----结束

7.5 登录节点

约束与限制

- SSH方式登录时要求该节点（弹性云服务器 ECS）已绑定弹性公网IP。

- 只有运行中的弹性云服务器才允许用户登录。
- Linux操作系统用户名为cloud。

登录方式概述

登录节点（弹性云服务器 ECS）的方式有如下两种：

- **管理控制台远程登录（VNC方式）**

  未绑定弹性公网IP的弹性云服务器可通过管理控制台提供的远程登录方式直接登录。

- **SSH方式登录**

  仅适用于Linux弹性云服务器。您可以使用远程登录工具（例如PuTTY、Xshell、SecureCRT等）登录弹性云服务器。如果普通远程连接软件无法使用，您可以使用云服务器ECS管理控制台的管理终端连接实例，查看云服务器操作界面当时的状态。

  说明

  - 本地使用Windows操作系统登录Linux节点时，输入的镜像用户名（Auto-login username）为：cloud。
  - CCE控制台不提供针对节点的操作系统升级，也不建议您通过yum方式进行升级，如果您在节点上通过yum update升级了操作系统，会导致容器网络的组件不可用。

表 7-13 Linux 云服务器登录方式一览

是否绑定EIP	本地设备操作系统	连接方法
是	Windows	使用PuTTY、Xshell等远程登录工具。 • SSH密钥方式鉴权：**SSH密钥方式登录**
是	Linux	使用命令连接。 • SSH密钥方式鉴权：**SSH密钥方式登录**
是/否	Windows或者Linux	使用管理控制台远程登录方式**Linux云服务器远程登录（VNC方式）**。

7.6 管理节点

7.6.1 管理节点标签

节点标签可以给节点打上不同的标签，给节点定义不同的属性，通过这些标签可以快速的了解各个节点的特点。

节点标签使用场景

节点标签的主要使用场景有两类。

- 节点管理：通过节点标签管理节点，给节点分类。

- 工作负载与节点的亲和与反亲和：
  - 有的工作负载需要的CPU大，有的工作负载需要的内存大，有的工作负载需要IO大，可能会影响其他工作负载正常工作等等，此时建议给节点添加不同标签。在部署工作负载的时候，就可以选择相应标签的节点亲和部署，保证系统正常工作；反之，可以使用节点的反亲和部署。
  - 一个系统可以分为多个模块，每个模块由多个微服务组成，为保证后期运维的高效，可以将节点打上对应模块的标签，让各模块部署到各自的节点模块上，互不干扰，方便开发到各自节点上去维护。

节点固有标签

节点创建出来会存在一些固有的标签，并且是无法删除的，这些标签的含义请参见表7-14。

说明

系统自动添加的节点固有标签不建议手动修改，如果手动修改值与系统值产生冲突，将以系统值为准。

表 7-14 节点固有标签

键	说明
新：topology.kubernetes.io/region 旧：failure-domain.beta.kubernetes.io/region	表示节点当前所在区域。
新：topology.kubernetes.io/zone 旧：failure-domain.beta.kubernetes.io/zone	表示节点所在区域的可用区。
新：node.kubernetes.io/baremetal 旧：failure-domain.beta.kubernetes.io/is-baremetal	表示是否为裸金属节点。 例如：false，表示非裸金属节点
node.kubernetes.io/instance-type	节点实例规格。
kubernetes.io/arch	节点处理器架构。
kubernetes.io/hostname	节点名称。
kubernetes.io/os	节点操作系统类型。
node.kubernetes.io/subnetid	节点所在子网的ID。
os.architecture	表示节点处理器架构。 例如：amd64，表示AMD64位架构的处理器

键	说明
os.name	节点的操作系统名称。
os.version	操作系统节点内核版本。
node.kubernetes.io/container-engine	节点所用容器引擎。
accelerator	GPU节点标签。
cce.cloud.com/cce-nodepool	节点池节点专属标签。

添加/删除节点标签

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，勾选节点，单击左上方“标签与污点管理”。

步骤3 在弹出的窗口中，在“批量操作”下方单击“新增批量操作”，然后选择“添加/更新”或“删除”。

填写需要增加/删除标签的“键”和“值”，单击“确定”。

例如，填写的键为“deploy_qa”，值为“true”，就可以从逻辑概念表示该节点是用来部署QA（测试）环境使用。

步骤4 标签添加成功后，再次进入该界面，在节点数据下可查看到已经添加的标签。

----结束

7.6.2 管理节点污点（Taint）

污点（Taint）能够使节点排斥某些特定的Pod，从而避免Pod调度到该节点上。

污点

节点污点是与“效果”相关联的键值对。以下是可用的效果：

- NoSchedule：不能容忍此污点的 Pod 不会被调度到节点上；现有 Pod 不会从节点中逐出。
- PreferNoSchedule：Kubernetes 会尽量避免将不能容忍此污点的 Pod 安排到节点上。
- NoExecute：如果 Pod 已在节点上运行，则会将该 Pod 从节点中逐出；如果尚未在节点上运行，则不会将其安排到节点上。

使用 **kubectl taint node** *nodename* 命令可以给节点增加污点，如下所示。

```
$ kubectl get node
NAME            STATUS   ROLES    AGE    VERSION
192.168.10.170   Ready    <none>   73d    v1.19.8-r1-CCE21.4.1.B003
192.168.10.240   Ready    <none>   4h8m   v1.19.8-r1-CCE21.6.1.2.B001
$ kubectl taint node 192.168.10.240 key1=value1:NoSchedule
node/192.168.10.240 tainted
```

通过describe命名和get命令可以查看到污点的配置。

```
$ kubectl describe node 192.168.10.240
Name:               192.168.10.240
...
Taints:             key1=value1:NoSchedule
...
$ kubectl get node 192.168.10.240 -oyaml
apiVersion: v1
...
spec:
  providerID: 06a5ea3a-0482-11ec-8e1a-0255ac101dc2
  taints:
  - effect: NoSchedule
    key: key1
    value: value1
...
```

去除污点可以使用如下命令，在NoSchedule后加一个“-”。

```
$ kubectl taint node 192.168.10.240 key1=value1:NoSchedule-
node/192.168.10.240 untainted
$ kubectl describe node 192.168.10.240
Name:               192.168.10.240
...
Taints:             <none>
...
```

在CCE控制台上同样可以管理节点的污点，且可以批量操作。

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，勾选节点，单击左上方“标签与污点管理”。

步骤3 在弹出的窗口中，在“批量操作”下方单击“新增批量操作”，单击“添加/更新”，选择“污点(Taints)”。

填写需要增加污点的“键”和“值”，选择污点的效果，单击“确定”。

步骤4 污点添加成功后，再次进入该界面，在节点数据下可查看到已经添加的污点。

----结束

系统污点说明

当节点出现某些问题时，Kubernetes会自动给节点添加一个污点，当前内置的污点包括：

- node.kubernetes.io/not-ready：节点未准备好。这相当于节点状况 Ready 的值为 "False"。
- node.kubernetes.io/unreachable：节点控制器访问不到节点。这相当于节点状况 Ready 的值为 "Unknown"。
- node.kubernetes.io/memory-pressure：节点存在内存压力。
- node.kubernetes.io/disk-pressure：节点存在磁盘压力。
- node.kubernetes.io/pid-pressure：节点存在 PID 压力。
- node.kubernetes.io/network-unavailable：节点网络不可用。
- node.kubernetes.io/unschedulable：节点不可调度。
- node.cloudprovider.kubernetes.io/uninitialized：如果 kubelet 启动时指定了一个“外部”云平台驱动， 它将给当前节点添加一个污点将其标志为不可用。在 cloud-controller-manager 的一个控制器初始化这个节点后，kubelet 将删除这个污点。

节点调度设置

在CCE控制台上可以配置调度设置，登录CCE控制台进入节点，选择“节点管理”，在节点列表右侧单击“更多 > 禁止调度”。

弹出如下对话框，单击“是”，即可将节点设置为不可调度。

这个操作会给节点打上污点，使用kubectl可以查看污点的内容。

```
$ kubectl describe node 192.168.10.240
...
Taints:             node.kubernetes.io/unschedulable:NoSchedule
...
```

在CCE控制台相同位置再次设置，即可将去除污点，将节点设置为可调度。

容忍度（Toleration）

容忍度应用于Pod上，允许（但并不要求）Pod 调度到带有与之匹配的污点的节点上。

污点和容忍度相互配合，可以用来避免 Pod 被分配到不合适的节点上。 每个节点上都可以应用一个或多个污点，这表示对于那些不能容忍这些污点的 Pod，是不会被该节点接受的。

在 Pod 中设置容忍度示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoSchedule"
```

上面示例表示这个Pod容忍标签为key1=value1，效果为NoSchedule的污点，所以这个Pod能够调度到对应的节点上。

同样还可以按如下方式写，表示当节点有key1这个污点时，可以调度到节点。

```
tolerations:
- key: "key1"
  operator: "Exists"
  effect: "NoSchedule"
```

7.6.3 重置节点

操作场景

您可以通过重置节点修改节点的配置，比如修改节点操作系统、登录方式等。

重置节点会重装节点操作系统，并重新安装节点上Kubernetes软件。如果您在使用过程中修改了节点上的配置等操作导致节点不可用，可以通过重置节点进行修复。

约束与限制

- v1.13及以上版本的CCE集群、CCE Turbo集群支持重置节点。

注意事项

- 重置节点功能不会重置控制节点，仅能对工作节点进行重置操作，如果重置后节点仍然不可用，请删除该节点重新创建。
- **重置节点将对节点操作系统进行重置安装，节点上已运行的工作负载业务将会中断，请在业务低峰期操作。**
- **节点重置后系统盘和docker数据盘将会被清空，重置前请事先备份重要数据。**
- **用户节点如果有自行挂载了数据盘，重置完后会清除挂载信息，请事先备份重要数据，重置完成后请重新执行挂载行为，数据不会丢失。**
- 节点上的工作负载实例的IP会发生变化，但是不影响容器网络通信。
- 云硬盘必须有剩余配额。
- 操作过程中，后台会把当前节点设置为不可调度状态。
- 重置节点会导致与节点关联的本地持久卷类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。重置节点时使用了本地持久存储卷的Pod会从重置的节点上驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。节点重置完成后，Pod可能调度到重置好的节点上，此时Pod会一直处于creating状态，因为PVC对应的底层逻辑卷已经不存在了。

操作步骤

新UI支持批量重置节点。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧选择“节点管理”，在右侧列表中选择要重置的节点，可以选多个。单击“更多 > 重置节点”。

步骤3 在弹出的窗口中单击“下一步”。跳转到参数配置界面。

- 对于DefaultPool节点池下的节点，会跳转到配置参数界面，请参见步骤4进行配置。
- 对于用户创建的节点池下的节点，重置时不支持配置参数，直接使用节点池的配置镜像重置。

步骤4 配置节点参数。

计算配置

表 7-15 计算配置参数

参数	参数说明
节点规格	重置节点时不支持修改节点规格。

参数	参数说明
容器引擎	CCE集群支持Docker，并在部分场景中支持Containerd。 • 1.23及以上的VPC网络集群都支持Containerd，容器隧道网络集群从1.23.2-r0开始支持Containerd。 • CCE Turbo引擎支持Docker和Containerd。
操作系统	公共镜像：请选择节点对应的操作系统。
登录方式	• **密钥对** 选择用于登录本节点的密钥对，支持选择共享密钥。 密钥对用于远程登录节点时的身份认证。若没有密钥对，可单击选项框右侧的“创建密钥对”来新建。

存储配置

配置节点云服务器上的存储资源，方便节点上的容器软件与容器应用使用。

表 7-16 存储配置参数

参数	参数说明
系统盘	直接使用云服务器的系统盘。
数据盘	**至少需要一块数据盘**，供容器运行时和Kubelet组件使用，**该数据盘不能被删除卸载，否则会导致节点不可用。** 单击后方的“展开高级设置”可设置自定义空间分配：勾选后可定义容器运行时在数据盘上占用的空间比例，容器运行时的空间用于存放容器运行时工作目录、容器镜像数据以及镜像元数据。数据盘空间分配详细说明请参见**数据盘空间分配说明**。 **其他数据盘**默认情况直接创建为裸盘，不做任何处理。您也可以展开高级配置，将磁盘挂载到指定目录。

高级配置

表 7-17 高级配置参数

参数	参数说明
K8s标签	单击“添加标签”可以设置附加到Kubernetes 对象（比如 Pods ）上的键值对，最多可以添加20条标签。 使用该标签可区分不同节点，可结合工作负载的亲和能力实现容器Pod调度到指定节点的功能。详细请参见**Labels and Selectors**。

参数	参数说明
污点（Taints）	默认为空。支持给节点加污点来设置反亲和性，每个节点最多配置20条污点，每条污点包含以下3个参数： ● Key：必须以字母或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符；另外可以使用DNS子域作为前缀。 ● Value：必须以字符或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符。 ● Effect：只可选NoSchedule，PreferNoSchedule或NoExecute。 **须知** ● 污点配置时需要配合Pod的toleration使用，否则可能导致扩容失败或者Pod无法调度到扩容节点。 ● 节点池创建后可单击列表项的“编辑”修改配置，修改后将同步到节点池下的已有节点。
最大实例数	节点最大可以正常运行的实例数(Pod)，该数量包含系统默认实例。 该设置的目的为防止节点因管理过多实例而负载过重，请根据您的业务需要进行设置。
安装前执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装前执行，可能导致Kubernetes软件无法正常安装，需谨慎使用。
安装后执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装后执行，不影响Kubernetes软件安装。

步骤5 单击“下一步：规格确认”。

步骤6 单击“提交”。

----结束

7.6.4 移除节点

操作场景

在集群中移除节点会将该节点移出集群，然后重装节点的操作系统，并清理节点上的CCE组件。

移除不会删除节点对应的服务器。移除前请确认您的正常业务运行不受影响，请谨慎操作。

节点移出集群后会继续开机运行。

约束限制

- 当且仅当CCE集群状态为运行中或不可用时允许移除节点。

- 当且仅当CCE节点状态为运行中、不可用或错误时允许被移除。
- 为使CCE节点正常移除，且移除后能正常重装操作系统清理CCE组件，请确保服务器处于正常运行中状态。
- 若节点在CCE集群移除后重装操作系统失败，请手动完成失败节点的操作系统重装，并在重装后登录节点执行清理脚本完成CCE组件清理，具体步骤参见**重装操作系统失败如何处理**。
- 移除节点会导致与节点关联的**本地持久卷**类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。移除节点时使用了本地持久存储卷的Pod会从移除的节点上驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。

注意事项

- 移除节点会涉及Pod迁移，可能会影响业务，请在业务低峰期操作。
- 操作过程中可能存在非预期风险，请提前做好相关的数据备份。
- 操作过程中，后台会把当前节点设置为不可调度状态。
- 移除节点重装操作系统后将清理原有的LVM分区，通过LVM管理的数据将会清空，请提前做好相关的数据备份。

操作步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择节点管理，单击节点后的“更多 > 移除”。

步骤3 在弹出的“移除节点”对话框中，配置重装操作系统需要的登录信息，单击“是”，等待完成节点移除。

移除节点后，原有节点上的工作负载实例会自动迁移至其他可用节点。

----结束

重装操作系统失败如何处理

移除节点重装操作系统可能会失败，如果碰到这种情况，您可以执行如下步骤重装操作系统并清理节点上的CCE组件。

步骤1 登录服务器的管理控制台，完成操作系统的重装。

步骤2 登录服务器，执行如下命令完成CCE组件和LVM数据的清理。

将如下脚本写入**clean.sh**文件。

```
lsblk
vgs --noheadings | awk '{print $1}' | xargs vgremove -f
pvs --noheadings | awk '{print $1}' | xargs pvremove -f
lvs --noheadings | awk '{print $1}' | xargs -i lvremove -f --select {}
function init_data_disk() {
    all_devices=$(lsblk -o KNAME,TYPE | grep disk | grep -v nvme | awk '{print $1}' | awk '{ print "/dev/"$1}')
    for device in ${all_devices[@]}; do
        isRootDisk=$(lsblk -o KNAME,MOUNTPOINT $device 2>/dev/null| grep -E '[[:space:]]/$' | wc -l )
        if [[ ${isRootDisk} != 0 ]]; then
            continue
        fi
        dd if=/dev/urandom of=${device} bs=512 count=64
        return
    done
```

```
    exit 1
}
init_data_disk
lsblk
```

执行如下命令。

bash clean.sh

----结束

7.6.5 同步云服务器

操作场景

集群中的每一个节点对应一台云服务器，集群节点创建成功后，您仍可以根据需求，修改云服务器的名称或变更规格。由于规格变更对业务有影响，建议一台成功完成后再对下一台进行规格变更。

CCE节点的部分信息是独立于弹性云服务器ECS维护的，当您在ECS控制台修改云服务器的名称、弹性公网IP，以及变更规格后，需要通过“同步云服务器”功能将信息同步到CCE控制台相应节点中，同步后信息将保持一致。

约束与限制

- 支持同步数据：虚机状态、云服务器名称、CPU数量、Memory数量、云服务器规格、公网IP等。
- 不支持同步数据：操作系统、镜像ID等（此类参数禁止在ECS控制台变更）

操作步骤

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”。

步骤3 单击节点后的“更多 > 同步云服务器”。

同步完成后，页面右上角将会提示“同步云服务器任务下发成功”。

----结束

7.6.6 删除节点

操作场景

在CCE集群中删除节点会将该节点以及节点内运行的业务都销毁，删除前请确认您的正常业务运行不受影响，请谨慎操作。

约束与限制

- CCE正在使用的虚拟机节点不支持在ECS页面进行删除。
- 删除节点会导致与节点关联的**本地持久卷**类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。删除节点时使用了本地持久存储卷的Pod会从删除的节点上驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。

注意事项

- 删除节点会涉及Pod迁移，可能会影响业务，请在业务低峰期操作。
- 操作过程中可能存在非预期风险，请提前做好相关的数据备份。
- 操作过程中，后台会把当前节点设置为不可调度状态。
- 删除节点仅能移除工作节点，不会移除控制节点。

操作步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“节点管理”，在右侧单击节点后的“更多 > 删除”。

步骤3 在弹出的“删除节点”窗口中，单击“是”，等待完成节点删除。

说明

- 删除节点后，原有节点上的工作负载实例会自动迁移至其他可用节点。
- 节点上绑定的磁盘和EIP如果属于重要资源请先解绑，否则会被级联删除。

----结束

7.6.7 节点关机

操作场景

集群中的节点关机后，该节点以及节点内的业务将停止运行，节点关机前，请先确认您的正常业务运行将不受影响，请谨慎操作。

约束与限制

- 节点关机会涉及Pod迁移，可能会影响业务，请在业务低峰期操作。
- 操作过程中可能存在非预期风险，请提前做好相关的数据备份。
- 操作过程中，后台会把当前节点设置为不可调度状态。
- 节点关机仅能对工作节点关机，不会对控制节点关机。

操作方法

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“节点管理”，在右侧单击待关机节点的名称。

步骤3 页面跳转至弹性云服务器详情页中，单击右上角的“关机”，在弹出的关机窗口中单击“确定”，即可完成关机操作。

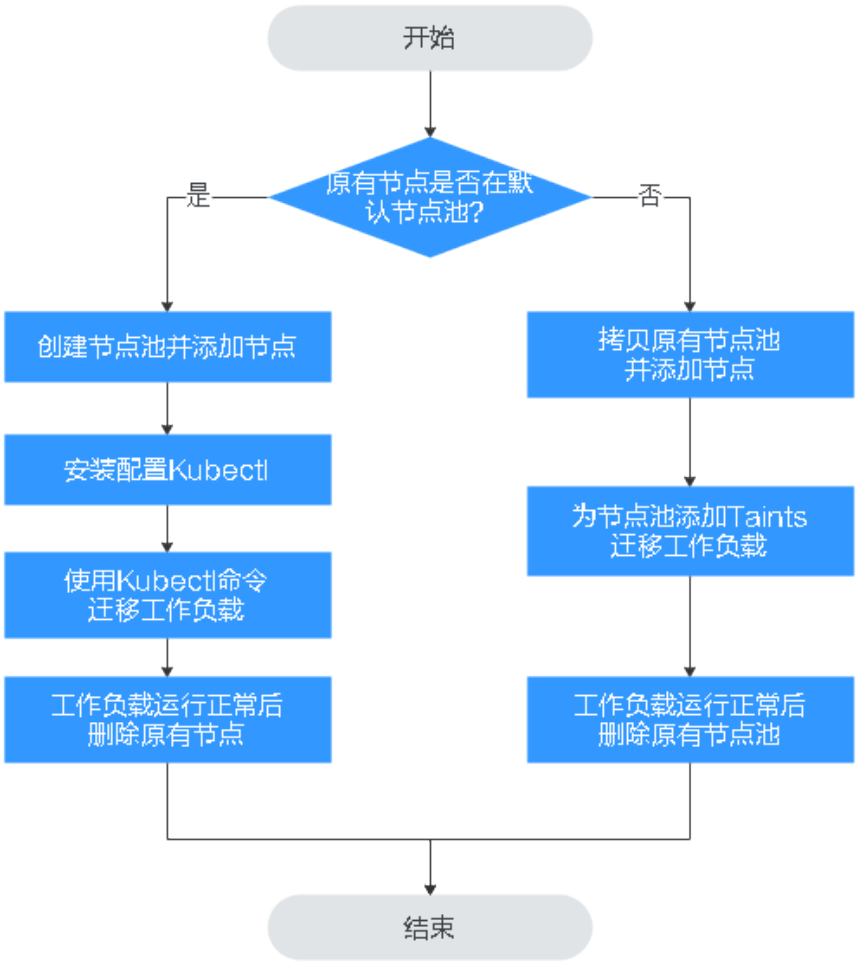
----结束

7.6.8 节点滚动升级

操作场景

节点滚动升级就是先创建新节点，然后将工作负载迁移到新的节点上，再删除老的节点。迁移流程如图7-1所示。

图 7-1 节点迁移流程



约束与限制

- 现有节点和工作负载待迁移的节点必须在同一集群。
- 当前仅支持在Kubernetes v1.13.10及以后集群版本执行此操作。
- 默认节点池DefaultPool不支持修改配置。

原有节点在默认节点池

步骤1 创建新的节点池。具体请参见**创建节点池**。

步骤2 单击节点池名称，单击“操作”区域的“节点列表”可查看新建节点的IP地址。

步骤3 安装配置kubectl。具体请参见**通过kubectl连接集群**。

步骤4 迁移工作负载。

1. 给需要迁移工作负载的节点打上Taint（污点）。

   kubectl taint node *[node]* **key=value:***[effect]*

   其中，*[node]*为待迁移工作负载所在节点的IP；*[effect]*取值为NoSchedule、PreferNoSchedule或NoExecute，此处必须设置为NoSchedule。

   - NoSchedule：一定不能被调度。

- PreferNoSchedule：尽量不要调度。
- NoExecute：不仅不会调度，还会驱逐Node上已有的Pod。

说明

若需要重新设置污点时，可执行**kubectl taint node** *[node]* **key:[***effect]*-命令去除污点。

2. 安全驱逐节点上的工作负载。

   kubectl drain *[node]*

   其中，*[node]*为待转移工作负载所在节点的IP。

3. 在左侧导航栏中选择“工作负载 > 无状态负载 Deployment”。在工作负载列表中，待迁移工作负载的状态由“运行中”变为“未就绪”。工作负载状态再次变为“运行中”，表示迁移成功。

说明

迁移工作负载时，若工作负载配置了节点亲和性，则工作负载会一直提示“未就绪”等异常情况。请单击工作负载名称进入到负载详情页，在选择“调度策略”页签，删除原节点的亲和性配置，配置新的节点亲和性和反亲和性策略，详情请参见**调度策略（亲和与反亲和）**。

工作负载迁移成功后，在工作负载详情页的“实例列表”页签，可查看到工作负载状已迁移到**步骤1**中所创建的节点上。

步骤5 删除原有节点。

工作负载迁移成功且运行正常后，即可删除原有节点。

----结束

原有节点不在默认节点池

步骤1 拷贝节点池并添加节点。具体请参见**拷贝节点池**。

步骤2 单击节点池名称操作列的“节点列表”，在节点列表中可查看到新建节点的IP地址。

步骤3 迁移工作负载。

1. 单击原节点池后的“编辑”配置污点参数。
2. 输入“污点(Taints)”的Key和Value值，Effect选项有NoSchedule、PreferNoSchedule或NoExecute，此处必须选择“NoExecute”，单击“确认添加”。
   - NoSchedule：一定不能被调度。
   - PreferNoSchedule：尽量不要调度。
   - NoExecute：不仅不会调度，还会驱逐Node上已有的Pod。

   说明

   若需要重新设置污点，需删除已配置污点。
3. 单击“确定”。
4. 在左侧导航栏中选择“工作负载 > 无状态负载 Deployment”。在工作负载列表中，待迁移工作负载的状态由“运行中”变为“未就绪”。工作负载状态再次变为“运行中”，表示迁移成功。

说明

迁移工作负载时，若工作负载配置了节点亲和性，则工作负载会一直提示“未就绪”等异常情况。请单击工作负载名称进入到负载详情页，在选择“调度策略”页签，删除原节点的亲和性配置，配置新的节点亲和性和反亲和性策略，详情请参见**调度策略（亲和与反亲和）**。

工作负载迁移成功后，在工作负载详情页的“实例列表”页签，可查看到工作负载状已迁移到**步骤1**中所创建的节点上。

步骤4 删除原有节点。

工作负载迁移成功且运行正常后，即可删除原有节点。

----结束

7.7 节点运维

7.7.1 节点预留资源策略说明

节点的部分资源需要运行一些必要的Kubernetes系统组件和Kubernetes系统资源，使该节点可作为您的集群的一部分。 因此，您的节点资源总量与节点在Kubernetes中的可分配资源之间会存在差异。节点的规格越大，在节点上部署的容器可能会越多，所以Kubernetes自身需预留更多的资源。

为了保证节点的稳定性，CCE集群节点上会根据节点的规格预留一部分资源给Kubernetes的相关组件（kubelet，kube-proxy以及docker等）。

CCE对用户节点可分配的资源计算法则如下：

Allocatable = Capacity - Reserved - Eviction Threshold

即，**节点资源可分配量=总量-预留值-驱逐阈值**。其中内存资源的驱逐阈值，固定为100MB。

说明

此处**总量 Capacity**为弹性云服务器除系统组件消耗外的可用内存，因此总量会略小于节点规格的内存值。

当节点上所有Pod消耗的内存上涨时，可能存在下列两种行为：

1. 当节点可用内存低于驱逐阈值时，将会触发kubelet驱逐Pod。关于Kubernetes中驱逐阈值的相关信息，请参见**节点压力驱逐**。
2. 如果节点在kubelet回收内存之前触发操作系统内存不足事件（OOM），系统会终止容器，但是与Pod驱逐不同，kubelet会根据Pod的RestartPolicy重新启动它。

CCE 对节点内存的预留规则 v1

说明

v1.21.4-r0和**v1.23.3-r0**以下版本集群中，节点内存的预留规则使用v1模型。对于**v1.21.4-r0**和**v1.23.3-r0**及以上版本集群，节点内存的预留规则优化为v2模型，请参见**CCE对节点内存的预留规则v2**。

CCE节点内存的总预留值等于**系统组件预留值**与**Kubelet管理Pod所需预留值**之和。

公式为：总预留值 = **系统组件预留值** + **Kubelet管理Pod所需预留值**

表 7-18 系统组件预留规则

内存总量范围	系统组件预留值
内存总量 <= 8GB	0MB
8GB < 内存总量 <= 16GB	((内存总量 – 8GB)*1024*10%)MB
16GB < 内存总量 <= 128GB	(8GB*1024*10% + (内存总量 – 16GB)*1024*6%)MB
内存总量 > 128GB	(8GB*1024*10% + 112GB*1024*6% + (内存总量 – 128GB)*1024*2%)MB

表 7-19 Kubelet 管理 Pod 所需预留规则

内存总量范围	Pod数量	Kubelet管理Pod所需预留值
内存总量 <= 2GB	-	内存总量 *25%
内存总量 > 2GB	0 < 节点的最大实例数 <= 16	700 MB
	16 < 节点的最大实例数 <= 32	(700 + (节点的最大实例数 - 16)*18.75)MB
	32 < 节点的最大实例数 <= 64	(1024 + (节点的最大实例数 - 32)*6.25)MB
	64 < 节点的最大实例数 <= 128	(1230 + (节点的最大实例数 - 64)*7.80)MB
	节点的最大实例数 > 128	(1740 + (节点的最大实例数 - 128)*11.20)MB

须知

对于小规格节点，需根据实际使用情况调整节点的最大实例数。或者在CCE控制台创建节点时，需考虑根据节点规格自适应调整节点的最大实例数参数。

CCE 对节点内存的预留规则 v2

对于**v1.21.4-r0**和**v1.23.3-r0**及以上版本集群，节点内存的预留规则优化为v2模型，且支持通过节点池配置管理参数（kube-reserved-mem和system-reserved-mem）动态调整，具体方法请参见**管理节点池**。

CCE节点内存v2模型的总预留值等于**OS侧预留值**与**CCE管理Pod所需预留值**之和。

其中OS侧预留包括基础预留和随节点内存规格变动的浮动预留；CCE侧预留包括基础预留和随节点Pod数量的浮动预留。

表 7-20 节点内存预留规则 v2

预留类型	基础/浮动	预留公式	预留对象
OS侧预留	基础预留	固定400MB	sshd、systemd-journald等操作系统服务组件占用
	浮动预留（随节点内存）	25MB/GB	内核占用
CCE侧预留	基础预留	固定500MB	节点空载时， kubelet、kube-proxy等容器引擎组件占用。
	浮动预留（随节点Pod数量）	Docker：20MB/Pod Containerd：5MB/Pod	Pod数量增加时，容器引擎组件的额外占用。 **说明** v2模型在计算节点默认预留内存时，随内存估计节点默认最大实例数，请参见表7-24。

CCE 对节点 CPU 的预留规则

表 7-21 节点 CPU 预留规则

CPU总量范围	CPU预留值
CPU总量 <= 1core	CPU总量 *6%
1core < CPU总量 <= 2core	1core*6% + (CPU总量– 1core)*1 %
2core < CPU总量 <= 4core	1core*6% + 1core*1% + (CPU总量– 2core)*0.5%
CPU总量 > 4core	1core*6% + 1core*1% + 2core*0.5% + (CPU总量–4core)*0.25%

CCE 对节点数据盘的预留规则

CCE使用LVM（Logical Volume Manager）进行磁盘管理，LVM会在磁盘上创建一个metadata区域用于存储逻辑卷和物理卷的信息，导致磁盘存在4MiB的不可用空间。因此节点实际可用的磁盘空间 = 磁盘大小 - 4MiB。

7.7.2 数据盘空间分配说明

本章节将详细介绍节点数据盘空间分配的情况，以便您根据业务实际情况配置数据盘大小。

设置数据盘空间分配

在创建节点时，您需要配置节点数据盘，您可单击“展开高级配置”，自定义节点数据盘的空间分配。

- **自定义容器引擎空间大小说明**：

  CCE 将数据盘空间默认划分为两块：一块用于存放容器引擎 (Docker/Containerd) 工作目录、容器镜像的数据和镜像元数据；另一块用于Kubelet组件和EmptyDir临时存储等。容器引擎空间的剩余容量将会影响镜像下载和容器的启动及运行。

  - 容器引擎和容器镜像空间（默认占90%）：用于容器运行时工作目录、存储容器镜像数据以及镜像元数据。
  - Kubelet组件和EmptyDir临时存储（默认占10%）：用于存储Pod配置文件、密钥以及临时存储EmptyDir等挂载数据。

- **自定义Pod容器空间大小**：即容器的basesize设置，每个工作负载下的容器组 Pod 占用的磁盘空间设置上限（包含容器镜像占用的空间）。合理的配置可避免容器组无节制使用磁盘空间导致业务异常。建议此值不超过容器引擎空间的 80%。该参数与节点操作系统和容器存储Rootfs相关，部分场景下不支持设置。

自定义容器引擎空间大小

对于容器引擎和Kubelet不共享磁盘空间的节点，数据盘根据容器存储Rootfs不同具有两种划分方式（以100G大小为例）：**Device Mapper类型**和**OverlayFS类型**。不同操作系统对应的容器存储Rootfs请参见**操作系统与容器存储Rootfs对应关系**。

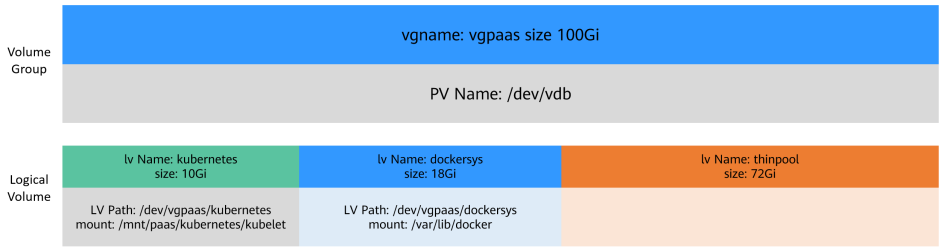
- **Device Mapper类型存储Rootfs**

  其中默认占90%的容器引擎和容器镜像空间又可分为以下两个部分：

  - 其中/var/lib/docker用于Docker工作目录，默认占比20%，其空间大小 = **数据盘空间 * 90% * 20%**
  - thinpool用于存储容器镜像数据、镜像元数据以及容器使用的磁盘空间，默认占比为80%，其空间大小 = **数据盘空间 * 90% * 80%**

    thinpool是动态挂载，在节点上使用**df -h**命令无法查看到，使用**lsblk**命令可以查看到。

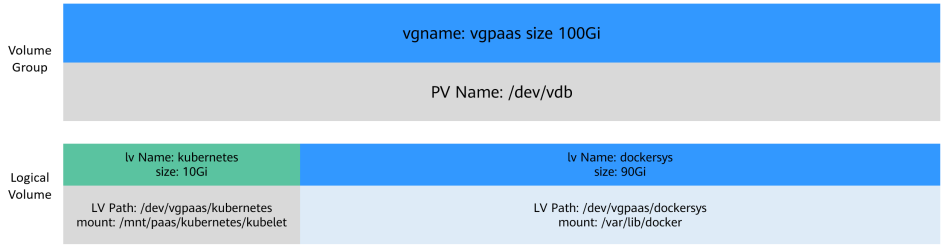
图 7-2 Device Mapper 类型容器引擎空间分配



- **OverlayFS类型存储Rootfs**

  相比Device Mapper存储引擎，没有单独划分thinpool，容器引擎和容器镜像空间（默认占90%）都在/var/lib/docker目录下。

图 7-3 OverlayFS 类型容器引擎空间分配



自定义 Pod 容器空间大小

自定义Pod容器空间（basesize）设置与节点操作系统和容器存储Rootfs相关（容器存储Rootfs请参见**操作系统与容器存储Rootfs对应关系**）：

- Device Mapper模式下支持自定义Pod容器空间（basesize）配置，默认值为10G。
- OverlayFS模式默认不限制Pod容器空间大小。

  说明

  使用EulerOS 2.9 的docker basesize设置时，若容器配置CAP_SYS_RESOURCE权限或privileged的特权，basesize限制Pod容器空间不起作用。

配置Pod容器空间（basesize）时，需要同时考虑节点的最大实例数配置。理想情况下，容器引擎空间需要大于容器使用的磁盘总空间，即：**容器引擎和容器镜像空间（默认占90%）> 容器数量 * Pod容器空间（basesize）**。否则，可能会引起节点分配的容器引擎空间不足，从而导致容器启动失败。

对于支持配置basesize的节点，尽管可以限制单个容器的主目录大小（开启时默认为10GB），但节点上的所有容器还是共用节点的thinpool磁盘空间，并不是完全隔离，当一些容器使用大量thinpool空间且总和达到节点thinpool空间上限时，也会影响其他容器正常运行。

另外，在容器的主目录中创删文件后，其占用的thinpool空间不会立即释放，因此即使basesize已经配置为10GB，而容器中不断创删文件时，占用的thinpool空间会不断增加一直到10GB为止，后续才会复用这10GB空间。如果节点上的容器数量*basesize > 节点thinpool空间大小，理论上有概率出现节点thinpool空间耗尽的场景。

操作系统与容器存储 Rootfs 对应关系

表 7-22 CCE 集群节点操作系统与容器引擎对应关系

操作系统	容器存储Rootfs	自定义Pod容器空间（basesize）
CentOS 7.x	v1.19.16以下版本集群使用Device Mapper v1.19.16及以上版本集群使用OverlayFS	Rootfs为Device Mapper且容器引擎为Docker时支持，默认值为10G。 Rootfs为OverlayFS时不支持。
EulerOS 2.5	Device Mapper	仅容器引擎为Docker时支持，默认值为10G。
EulerOS 2.9	OverlayFS	仅v1.19.16、v1.21.3、v1.23.3及以上的集群版本支持，默认值为不限制。 v1.19.16、v1.21.3、v1.23.3以前的集群版本不支持。
Ubuntu 22.04	OverlayFS	不支持。

表 7-23 CCE Turbo 集群节点操作系统与容器引擎对应关系

操作系统	容器存储Rootfs	自定义Pod容器空间（basesize）
CentOS 7.x	OverlayFS	不支持。
Ubuntu 22.04	OverlayFS	不支持。
EulerOS 2.9	弹性云服务器-虚拟机使用OverlayFS 弹性云服务器-物理机使用Device Mapper	Rootfs为OverlayFS且仅容器引擎为Docker时支持，默认值为不限制。 Rootfs为Device Mapper且容器引擎为Docker时支持，默认值为10G。

镜像回收策略说明

当容器引擎空间不足时，会触发镜像垃圾回收。

镜像垃圾回收策略只考虑两个因素：HighThresholdPercent 和 LowThresholdPercent。 磁盘使用率超过上限阈值（HighThresholdPercent，默认值为85%）将触发垃圾回收。 垃圾回收将删除最近最少使用的镜像，直到磁盘使用率满足下限阈值（LowThresholdPercent，默认值为80%）。

容器引擎空间大小配置建议

- 容器引擎空间需要大于容器使用的磁盘总空间，即：**容器引擎空间 > 容器数量 * Pod容器空间（basesize）**
- 容器业务的创删文件操作建议在容器挂载的本地存储（如emptyDir、hostPath）或云存储的目录中进行，这样不会占用thinpool空间。其中Emptydir使用的是kubelet空间，需要规划好kubelet空间的大小。
- 可将业务部署在使用OverlayFS存储模式的节点上（请参见**操作系统与容器存储Rootfs对应关系**），避免容器内创删文件后占用的磁盘空间不立即释放问题。

容器引擎和 Kubelet 共享磁盘空间说明

容器引擎和Kubelet共享磁盘空间即在节点上不再划分容器引擎（Docker/Containerd）和Kubelet组件的空间，二者共用磁盘空间。

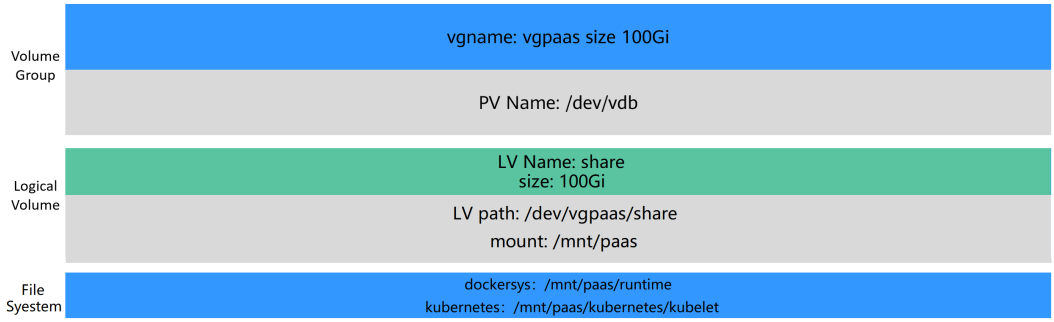
须知

- 容器引擎和Kubelet共享磁盘空间仅v1.21.10-r0、v1.23.8-r0、v1.25.3-r0及以上的集群支持。
- 容器存储Rootfs为OverlayFS类型时支持共享磁盘空间，Device Mapper类型不支持。
- 若您在集群中安装了npd插件，请将插件升级至1.18.10版本及以上，否则会产生误报警。
- 若您在集群中安装了log-agent插件，请将插件升级至1.3.0版本及以上，否则会影响日志采集。
- 若您在集群中安装了ICAgent，请将ICAgent升级至5.12.140版本及以上，否则会影响日志采集。。

对于共享磁盘空间的节点，容器存储Rootfs为**OverlayFS类型**。节点创建完成后，数据盘空间（以100G大小为例）不再划分容器引擎和容器镜像空间和Kubelet组件空间，均在/mnt/paas目录下，并通过两个文件系统区分：

- dockersys：/mnt/paas/runtime
- kubernetes：/mnt/paas/kubernetes/kubelet

图 7-4 共享数据盘空间分配



7.7.3 节点可创建的最大 Pod 数量说明

节点最大 Pod 数量计算方式

根据集群类型不同，节点可创建的最大Pod数量计算方式如下：

- 对于“容器隧道网络”的集群，仅取决于**节点最大实例数**。
- 对于“VPC网络”的集群，取决于**节点最大实例数**和**节点可分配容器IP数**中的最小值，建议节点最大实例数不要超过节点可分配容器IP数，否则当容器IP数不足时Pod实例可能无法调度。
- 对于“云原生2.0网络”的集群（CCE Turbo集群），取决于**节点最大实例数**和**CCE Turbo集群节点网卡数量**中的最小值。建议节点最大实例数不要超过节点网卡数，否则当节点规格可分配网卡不足时Pod实例可能无法正常调度。

节点可分配容器 IP 数

在创建CCE集群时，如果网络模型选择“VPC网络”，会让您选择每个节点可供分配的容器IP数量（alpha.cce/fixPoolMask）。Pod直接使用宿主机的网络（配置hostNetwork: true）时，不占用可分配容器IP，详情请参见**容器网络 vs 主机网络**。

该参数会影响节点上可以创建最大Pod的数量，因为每个Pod会占用一个IP（使用**容器网络**的情况），如果可用IP数量不够的话，就无法创建Pod。Pod直接使用宿主机的网络（配置hostNetwork: true）时，不占用可分配容器IP。

节点默认会占用掉3个容器IP地址（网络地址、网关地址、广播地址），因此节点上 **可分配给容器使用的IP数量 = 您选择的容器IP数量 - 3**。

节点最大实例数

在创建节点时，可以配置节点可以创建的最大实例数（maxPods）。该参数是kubelet的配置参数，决定kubelet最多可创建多少个Pod。

须知

对于默认节点池（DefaultPool）中的节点，节点创建完成后，最大实例数不支持修改。

对于自定义节点池中的节点，创建完成后可通过修改节点池配置中的max-pods参数，修改节点最大实例数。

根据节点规格不同，节点默认最大实例数如表7-24所示。

表 7-24 节点默认最大实例数

内存	节点默认最大实例数
4G	20
8G	40
16G	60
32G	80
64G及以上	110

节点网卡数量（仅 CCE Turbo 集群）

CCE Turbo集群ECS节点使用弹性辅助网卡，节点可以创建最大Pod数量与节点可使用网卡数量相关。

容器网络 vs 主机网络

创建Pod时，可以选择Pod使用容器网络或是宿主机网络。

- 容器网络：默认使用容器网络，Pod的网络由集群网络插件负责分配，**每个Pod分配一个IP地址，会占用容器网络的IP**。
- 主机网络：Pod直接使用宿主机的网络（Pod需要配置hostNetwork: true），会占用宿主机的端口，Pod的IP就是宿主机的IP，**不会占用容器网络的IP**。使用时需要考虑是否与宿主机上的端口冲突，因此一般情况下除非您知道需要某个特定应用占用宿主机上的特定端口时，不建议使用主机网络。

7.7.4 将节点容器引擎从 Docker 迁移到 Containerd

背景介绍

Kubernetes在1.24版本中移除了Dockershim，并从此不再默认支持Docker容器引擎。CCE 1.25集群中仍将继续维护Docker容器引擎，并计划在1.27版本中移除对Docker容器引擎的支持。如果您需要将容器引擎为Docker的节点迁移至Containerd节点，请参考本文。

前提条件

- 已创建至少一个集群，并且该集群支持Containerd节点。

- 您的集群中存在容器引擎为Docker的节点或节点池。

注意事项

- 理论上节点容器运行时的迁移会导致业务短暂中断，因此强烈建议您迁移的业务保证多实例高可用部署，并且建议先在测试环境试验迁移的影响，以最大限度避免可能存在的风险。
- Containerd不具备镜像构建功能，请勿在Containerd节点上使用Docker Build功能构建镜像。Docker和Containerd其他差异请参考**容器引擎**。

节点迁移步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧选择“节点管理”，并在节点列表中选择一个或多个需要重置的节点，单击“更多 > 重置节点”。

步骤3 在容器引擎中选择Containerd，其余参数可根据需要进行调整，也可以和创建时保持一致。

步骤4 当节点状态显示为安装中时，即表示正在重置节点。

待节点状态显示为运行中时，您即可检查节点容器运行时是否切换成功，页面中可以看到节点运行时版本已经切换为Containerd，并且登录节点可以执行**crictl**等Containerd相关命令查看节点上运行的容器信息。

----结束

节点池迁移步骤

您可使用**节点池拷贝**功能，拷贝原有的Docker节点池，并将新节点池的容器引擎选择为Containerd，其余配置和原Docker节点池保持一致。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧选择“节点管理”，切换至“节点池”页签，并在需要拷贝的Docker节点池“操作”栏中，单击“更多 > 拷贝”。

步骤3 在节点池配置页面中，选择容器引擎为Containerd，其余参数可根据需要进行调整，并完成节点池创建。

步骤4 将创建完的Containerd节点池扩容至原Docker节点池的数量，并逐个删除Docker节点池中的节点。

推荐使用滚动的方式迁移，即扩容部分Containerd节点，再删除部分Docker节点，直至新的Containerd节点池中节点数量和原Docker节点池中节点数量一致。

说明

若您在原有Docker节点或节点池上部署的负载设置了对应的节点亲和性，则需要将负载的节点亲和性策略配置为的新Containerd节点或节点池。

步骤5 迁移完成后，删除原有Docker节点池。

----结束

7.7.5 节点故障检测策略

节点故障检查功能依赖**node-problem-detector（简称：npd）**，npd是一款集群节点监控插件，插件实例会运行在每个节点上。本文介绍如何开启节点故障检测能力。

前提条件

集群中已安装**npd**插件。

开启节点故障检测

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧选择“节点管理”，检查集群中是否已安装npd插件，或将其升级至最新版本。npd安装成功后，可正常使用故障检测策略功能。

步骤3 npd运行正常时，单击“故障检测策略”，可查看当前故障检测项。关于NPD检查项列表请参见**NPD检查项**。

步骤4 当前节点检查结果异常时，将在节点列表处提示“指标异常”。

步骤5 您可单击“指标异常”，按照修复建议提示修复。

----结束

自定义检查项配置

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧选择“节点管理”，单击“故障检测策略”。

步骤3 在跳转的页面中查看当前检查项配置，单击检查项操作列的“编辑”，自定义检查项配置。

当前支持以下配置：

- 启用/停用：自定义某个检查项的开启或关闭。
- 目标节点配置：检查项默认运行在全部节点，用户可根据特殊场景需要自定义修改故障阈值。例如竞价实例中断回收检查只运行在竞价实例节点。
- 触发阈值配置：默认阈值匹配常见故障场景，用户可根据特殊场景需要自定义修改故障阈值。例如调整“连接跟踪表耗尽”触发阈值由90%调整至80%。
- 检查周期：默认检查周期为30秒，可根据用户场景需要自定义修改检查周期。
- 故障应对策略：故障产生后，可根据用户场景自定义修改故障应对策略，当前故障应对策略如下：

表 7-25 故障应对策略

故障应对策略	效果
提示异常	上报Kuberentes事件。

故障应对策略	效果
禁止调度	上报Kuberentes事件，并为节点添加NoSchedule污点。
驱逐节点负载	上报Kuberentes事件，并为节点添加NoExecute污点。该操作会驱逐节点上的负载，可能导致业务不连续，请谨慎选择。

----结束

NPD 检查项

说明

当前检查项仅1.16.0及以上版本支持。

NPD的检查项主要分为事件类检查项和状态类检查项。

- 事件类检查项

  对于事件类检查项，当问题发生时，NPD会向APIServer上报一条事件，事件类型分为Normal（正常事件）和Warning（异常事件）

  表 7-26 事件类检查项

故障检查项	功能	说明
OOMKilling	监听内核日志，检查OOM事件发生并上报 典型场景：容器内进程使用的内存超过了Limt，触发OOM并终止该进程	Warning类事件 监听对象：/dev/kmsg 匹配规则："Killed process \\d+ (.+) total-vm:\\d+kB, anon-rss:\\d+kB, file-rss:\\d+kB.*"
TaskHung	监听内核日志，检查taskHung事件发生并上报 典型场景：磁盘卡IO导致进程卡住	Warning类事件 监听对象：/dev/kmsg 匹配规则："task \\S+:\\w+ blocked for more than \\w+ seconds\\."
ReadonlyFilesystem	监听内核日志，检查系统内核是否有Remount root filesystem read-only错误 典型场景：用户从ECS侧误操作卸载节点数据盘，且应用程序对该数据盘的对应挂载点仍有持续写操作，触发内核产生IO错误将磁盘重挂载为只读磁盘。	Warning类事件 监听对象：/dev/kmsg 匹配规则："Remounting filesystem read-only"

- 状态类检查项

对于状态类检查项，当问题发生时，NPD会向APIServer上报一条事件，并同步修改节点状态，可配合**Node-problem-controller故障隔离**对节点进行隔离。

下列检查项中若未明确指出检查周期，则默认周期为30秒。

表 7-27 系统组件检查

故障检查项	功能	说明
容器网络组件异常 CNIProblem	检查CNI组件（容器网络组件）运行状态	无
容器运行时组件异常 CRIProblem	检查节点CRI组件（容器运行时组件）Docker和Containerd的运行状态	检查对象：Docker或Containerd
Kubelet频繁重启 FrequentKubeletRestart	通过定期回溯系统日志，检查关键组件Kubelet是否频繁重启	• 默认阈值：10分钟内重启10次 即在10分钟内组件重启10次表示频繁重启，将会产生故障告警。 • 监听对象：/run/log/journal目录下的日志
Docker频繁重启 FrequentDockerRestart	通过定期回溯系统日志，检查容器运行时Docker是否频繁重启	
Containerd频繁重启 FrequentContainerdRestart	通过定期回溯系统日志，检查容器运行时Containerd是否频繁重启	
Kubelet服务异常 KubeletProblem	检查关键组件Kubelet的运行状态	无
KubeProxy异常 KubeProxyProblem	检查关键组件KubeProxy的运行状态	无

表 7-28 系统指标

故障检查项	功能	说明
连接跟踪表耗尽 ConntrackFullProblem	检查连接跟踪表是否耗尽	• 默认阈值:90% • 使用量: nf_conntrack_count • 最大值: nf_conntrack_max
磁盘资源不足 DiskProblem	检查节点系统盘、CCE数据盘（包含CRI逻辑盘与Kubelet逻辑盘）的磁盘使用情况	• 默认阈值：90% • 数据来源： `df -h` 当前暂不支持额外的数据盘

故障检查项	功能	说明
文件句柄数不足 FDProblem	检查系统关键资源FD文件句柄数是否耗尽	• 默认阈值：90% • 使用量：/proc/sys/fs/file-nr中第1个值 • 最大值：/proc/sys/fs/file-nr中第3个值
节点内存资源不足 MemoryProblem	检查系统关键资源Memory内存资源是否耗尽	• 默认阈值：80% • 使用量：/proc/meminfo中MemTotal-MemAvailable • 最大值：/proc/meminfo中MemTotal
进程资源不足 PIDProblem	检查系统关键资源PID进程资源是否耗尽	• 默认阈值：90% • 使用量：/proc/loadavg中nr_threads • 最大值：/proc/sys/kernel/pid_max和/proc/sys/kernel/threads-max两者的较小值。

表 7-29 存储检查

故障检查项	功能	说明
磁盘只读 DiskReadonly	通过定期对节点系统盘、CCE数据盘（包含CRI逻辑盘与Kubelet逻辑盘）进行测试性写操作，检查关键磁盘的可用性	检测路径： • /mnt/paas/kubernetes/kubelet/ • /var/lib/docker/ • /var/lib/containerd/ • /var/paas/sys/log/cceaddon-npd/ 检测路径下会产生临时文件npd-disk-write-ping 当前暂不支持额外的数据盘
磁盘资源不足 DiskProblem	检查节点系统盘、CCE数据盘（包含cri逻辑盘与kubelet逻辑盘）的磁盘使用情况	• 默认阈值：90% • 数据来源：`df -h` 当前暂不支持额外的数据盘

故障检查项	功能	说明
节点emptydir存储池异常 EmptyDirVolumeGroupStatusError	检查节点上临时卷存储池是否正常 故障影响：依赖存储池的Pod无法正常写对应临时卷。临时卷由于IO错误被内核重挂载成只读文件系统。 典型场景：用户在创建节点时配置两个数据盘作为临时卷存储池，用户误操作删除了部分数据盘导致存储池异常。	• 检测周期：30秒 • 数据来源： `vgs -o vg_name, vg_attr` • 检测原理：检查VG（存储池）是否存在p状态，该状态表征部分PV（数据盘）丢失。 • 节点持久卷存储池异常调度联动：调度器可自动识别此异常状态并避免依赖存储池的Pod调度到该节点上。 • 例外场景：NPD无法检测所有PV（数据盘）丢失，导致VG（存储池）丢失的场景；此时依赖kubelet自动隔离该节点，其检测到VG（存储池）丢失并更新nodestatus.allocatable中对应资源为0，避免依赖存储池的Pod调度到该节点上。无法检测单个PV损坏；此时依赖ReadonlyFilesystem检测异常。
节点持久卷存储池异常 LocalPvVolumeGroupStatusError	检查节点上持久卷存储池是否正常 故障影响：依赖存储池的Pod无法正常写对应持久卷。持久卷由于IO错误被内核重挂载成只读文件系统。 典型场景：用户在创建节点时配置两个数据盘作为持久卷存储池，用户误操作删除了部分数据盘。	
挂载点异常 MountPointProblem	检查节点上的挂载点是否异常 异常定义：该挂载点不可访问（cd） 典型场景：节点挂载了nfs（网络文件系统，常见有obsfs、s3fs等），当由于网络或对端nfs服务器异常等原因导致连接异常时，所有访问该挂载点的进程均卡死。例如集群升级场景kubelet重启时扫描所有挂载点，当扫描到此异常挂载点会卡死，导致升级失败。	等效检查命令： `for dir in `df -h | grep -v "Mounted on" | awk "{print \\$NF}"`;do cd $dir; done && echo "ok"`

故障检查项	功能	说明
磁盘卡IO DiskHung	检查节点上所有磁盘是否存在卡IO，即IO读写无响应 卡IO定义：系统对磁盘的IO请求下发后未有响应，部分进程卡在D状态 典型场景：操作系统硬盘驱动异常或底层网络严重故障导致磁盘无法响应	• 检查对象：所有数据盘 • 数据来源：/proc/diskstat 等效查询命令：`iostat -xmt 1` • 阈值： – 平均利用率，ioutil >= 0.99 – 平均IO队列长度，avgqu-sz >=1 – 平均IO传输量，iops(w/s) +ioth(wMB/s) <= 1 **说明** 部分操作系统卡IO时无数据变化，此时计算CPU IO时间占用率，iowait > 0.8。
磁盘慢IO DiskSlow	检查节点上所有磁盘是否存在慢IO，即IO读写有响应但响应缓慢 典型场景：云硬盘由于网络波动导致慢IO。	• 检查对象：所有数据盘 • 数据来源：/proc/diskstat 等效查询命令 `iostat -xmt 1` • 默认阈值：平均IO时延，await >= 5000ms **说明** 卡IO场景下该检查项失效，原因为IO请求未有响应，await数据不会刷新。

表 7-30 其他检查

故障检查项	功能	说明
NTP异常 NTPProblem	检查节点时钟同步服务ntpd或chronyd是否正常运行，系统时间是否漂移	默认时钟偏移阈值：8000ms

故障检查项	功能	说明
进程D异常 ProcessD	检查节点是否存在D进程	默认阈值：连续3次存在10个异常进程 数据来源： • /proc/{PID}/stat • 等效命令：ps aux 例外场景：ProcessD忽略BMS节点下的SDI卡驱动依赖的常驻D进程heartbeat、update
进程Z异常 ProcessZ	检查节点是否存在Z进程	
ResolvConf配置文件异常 ResolvConfFileProblem	检查ResolvConf配置文件是否丢失 检查ResolvConf配置文件是否异常 异常定义：不包含任何上游域名解析服务器（nameserver）。	检查对象：/etc/resolv.conf
存在计划事件 ScheduledEvent	检查节点是否存在热迁移计划事件。热迁移计划事件通常由硬件故障触发，是IaaS层的一种自动故障修复手段。 典型场景：底层宿主机异常，例如风扇损坏、磁盘坏道等，导致其上虚机触发热迁移。	数据来源： • http://169.254.169.254/meta-data/latest/events/scheduled 该检查项为Alpha特性，默认不开启。

另外kubelet组件内置如下检查项，但是存在不足，您可通过集群升级或安装NPD进行补足。

表 7-31 Kubelet 内置检查项

故障检查项	功能	说明
PID资源不足 PIDPressure	检查PID是否充足	• 周期：10秒 • 阈值：90% • 缺点：社区1.23.1及以前版本，该检查项在pid使用量大于65535时失效，详见**issue 107107**。社区1.24及以前版本，该检查项未考虑thread-max。

故障检查项	功能	说明
内存资源不足 MemoryPressure	检查容器可分配空间（allocable）内存是否充足	• 周期：10秒 • 阈值：最大值-100MiB • 最大值（Allocable）：节点总内存-节点预留内存 • 缺点：该检测项没有从节点整体内存维度检查内存耗尽情况，只关注了容器部分(Allocable)。
磁盘资源不足 DiskPressure	检查kubelet盘和docker盘的磁盘使用量及inodes使用量	• 周期：10秒 • 阈值：90%

8 节点池

8.1 节点池概述

简介

为帮助您更好地管理Kubernetes集群内的节点，云容器引擎CCE引入节点池概念。节点池是集群中具有相同配置的一组节点，一个节点池包含一个节点或多个节点。

您可以在CCE控制台创建新的自定义节点池，借助节点池基本功能方便快捷地创建、管理和销毁节点，而不会影响整个集群。新节点池中所有节点的参数和类型都彼此相同，您无法在节点池中配置单个节点，任何配置更改都会影响节点池中的所有节点。

通过节点池功能您还可以实现节点的动态扩缩容：

- 当集群中出现因资源不足而无法调度的实例（Pod）时，自动触发扩容，为您减少人力成本。
- 当满足节点空闲等缩容条件时，自动触发缩容，为您节约资源成本。

本章节介绍节点池在云容器引擎（CCE）中的工作原理，以及如何创建和管理节点池。

节点池架构

通常情况下，节点池内的节点均具有如下相同属性：

- 节点操作系统。
- 节点规格。
- 节点登录方式。
- 节点容器运行时。
- 节点Kubernetes组件启动参数。
- 节点自定义启动脚本。
- 节点“K8s标签”及“污点”设置。

此外，CCE将同时围绕节点池扩展以下属性：

- 节点池级别操作系统。

- 节点池级别每节点的Pod数上限。

默认节点池 DefaultPool 说明

DefaultPool并非一个真正的节点池，只是将非自定义节点池中的节点做一个**归类**，所有非自定义节点池中创建的节点（直接在控制台创建的节点或调用API创建的节点）都会归类在DefaultPool中。DefaultPool不具备任何自定义节点池的功能，包括弹性伸缩、各项参数设置等，且不可编辑、删除或迁移，也不支持扩容、弹性伸缩。

应用场景

当业务需要使用大规模集群时，推荐您使用节点池进行节点管理，以提高大规模集群易用性。

下表介绍了多种大规模集群管理场景，并分别展示节点池在每种场景下发挥的作用：

表 8-1 节点池场景及作用

场景	作用
集群存在较多异构节点（机型配置不同）	通过节点池可规范节点分组管理。
集群需要频繁扩缩容节点	通过节点池可降低操作成本。
集群内应用程序调度规则复杂	通过节点池标签可快速指定业务调度规则。

功能点及注意事项

功能点	功能说明	注意事项
创建节点池	新增节点池。	单个集群不建议超过100个节点池。
删除节点池	删除节点池时会先删除节点池中的节点，原有节点上的工作负载实例会自动迁移至其他节点池的可用节点。	如果工作负载实例具有特定的节点选择器，且如果集群中的其他节点均不符合标准，则工作负载实例可能仍处于无法安排的状态。
节点池开启弹性伸缩	开启弹性伸缩后，节点池将根据集群负载情况自动创建或删除节点池内的节点。	节点池中的节点建议不要放置重要数据，以防止节点被弹性缩容，数据无法恢复。
节点池关闭弹性伸缩	关闭弹性伸缩后，节点池内节点数量不随集群负载情况自动调整。	/
调整节点池大小	支持直接调整节点池内节点个数。若减小节点数量，将从现有节点池内随机缩容节点。	开启弹性伸缩后，不建议手动调整节点池大小。

功能点	功能说明	注意事项
调整节点池配置	可修改节点池名称、节点个数，删除或新增K8s标签、污点，调整节点池磁盘配置、操作系统、容器引擎等配置。	删除或新增K8s标签和污点会对节点池内节点全部生效，可能会引起Pod重新调度，请谨慎变更。
移出节点池内节点	可以将同一个集群下某个节点池中的节点迁移到默认节点池（DefaultPool）中	暂不支持将默认节点池（DefaultPool）中的节点迁移到其他节点池中，也不支持将自定义节点池中的节点迁移到其他自定义节点池。
拷贝节点池	可以方便的拷贝现有节点池的配置，从而创建新的节点池。	/
配置Kubernetes参数	通过该功能您可以对核心组件进行深度配置。	• 本功能仅支持在v1.15及以上版本的集群中对节点池进行配置，v1.15以下版本不显示该功能。 • 默认节点池DefaultPool不支持修改该类配置。

将工作负载部署到特定节点池

在定义工作负载时，您可以间接的控制将其部署在哪个节点池上。

例如，您可以通过CCE控制台工作负载页面的“调度策略”设置工作负载与节点的亲和性，强制将该工作负载部署到特定节点池上，从而实现该工作负载仅在该节点池中的节点上运行的目的。如果您需要更好地控制工作负载实例的调度位置，您可以使用**调度策略（亲和与反亲和）**章节中关于工作负载与节点的亲和或反亲和策略相关说明。

您也可以为容器指定资源请求，工作负载将仅在满足资源请求的节点上运行。

例如，如果工作负载定义了需要包含四个CPU的容器，则工作负载将不会选择在具有两个CPU的节点上运行。

相关操作

您可以登录CCE控制台并参考以下文档， 进行节点池对应的操作：

- **创建节点池**
- **管理节点池**
- **创建无状态负载（Deployment）**
- **调度策略（亲和与反亲和）**

8.2 创建节点池

操作场景

本章介绍了如何添加运行节点池以及对节点池执行操作。要了解节点池的工作原理，请参阅**节点池概述**。

约束与限制

- 节点弹性伸缩功能需要安装autoscaler插件，具体安装与参数配置请参见**CCE Cluster Autoscaler**。

操作步骤

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击右上角“创建节点池”。

基础配置

表 8-2 基础配置

参数	参数说明
节点池名称	新建节点池的名称，默认按“集群名-nodepool-随机数”生成名称，可自定义。
节点数量	创建节点池时，创建节点的数量。

参数	参数说明
弹性伸缩	默认不开启。 开启弹性扩缩容功能要求安装**AutoScaler插件**。 单击  开启后，节点池将根据集群负载情况自动创建或删除节点池内的节点，参数设置如下： • 节点数上限和节点数下限：您可设置节点数的上限和下限，保证节点数在合理的范围内伸缩。 • 节点池优先级：请根据业务需要设置相应数值，该数值表示节点池之间进行**扩容的优先级**，数值越大优先级越高，如设置为4的节点池比设置为1的节点池优先启动扩容。若多个节点池的值设置相同，如都设置为2，表示这几个节点池之间不分优先级，系统将按最小资源浪费原则进行扩容。 **说明** 弹性扩容时CCE将按照如下策略来选择节点池进行扩容： 1. 通过预判算法判断节点池是否能满足让Pending的Pod正常调度的条件，包括节点资源大于Pod的request值、nodeSelect、nodeAffinity和taints等是否满足Pod正常调度的条件；另外还会过滤掉扩容失败（因为资源不足等原因）还处于15min冷却时间的节点池。 2. 有多个节点池满足条件时，判断节点池设置的优先级（优先级默认值为0，取值范围为0-100，其中100为最高，0为最低），选择优先级最高的节点池扩容。 3. 如果有多个节点池处于相同的优先级，或者都没有配置优先级时，通过最小浪费原则，根据节点池里设置的虚拟机规格，计算刚好能满足Pending的Pod正常调度，且浪费资源最少的节点池。 4. 如果还是有多个节点池的虚拟机规格都一样，只是AZ不同，那么会随机选择其中一个节点池触发扩容。 • 冷却时间：请设置时间，单位为分钟。弹性缩容冷却时间是指当前节点池扩容出的节点多长时间不能被缩容。 节点池中配置的缩容冷却时间和**autoscaler插件**中配置的缩容冷却时间之间的影响和关系如下： **节点池配置的缩容冷却时间** 弹性缩容冷却时间：当前节点池扩容出的节点多长时间不能被缩容，作用范围为节点池级别。 **autoscaler插件配置的缩容冷却时间** 扩容后缩容冷却时间：autoscaler触发扩容后（不可调度、指标、周期策略等场景）整个集群多长时间内不能被缩容，作用范围为集群级别。 节点删除后缩容冷却时间：autoscaler触发缩容后整个集群多长时间内不能继续缩容，作用范围为集群级别。 缩容失败后缩容冷却时间：autoscaler触发缩容失败后整个集群多长时间内不能继续缩容，作用范围为集群级别。 **说明** 节点池中的节点建议不要放置重要数据，以防止节点被弹性缩容，数据无法恢复。

计算配置：

配置节点云服务器的规格与操作系统，为节点上的容器应用提供基本运行环境。

表 8-3 计算配置参数

参数	参数说明
可用区	节点云服务器所在的可用区，集群下节点创建在不同可用区下可以提高可靠性。创建后不可修改。 建议您选择“随机分配”，可根据选择的节点规格随机分配一个可以使用的可用区。 可用区是在同一区域下，电力、网络隔离的物理区域，可用区之间内网互通，不同可用区之间物理隔离。如果您需要提高工作负载的高可靠性，建议您将云服务器创建在不同的可用区。
节点类型	CCE集群： ● 弹性云服务器-虚拟机：基于弹性云服务器部署容器服务。 ● 弹性云服务器-物理机：基于擎天架构的服务器部署容器服务。 CCE Turbo集群： ● 弹性云服务器-虚拟机：基于弹性云服务器部署容器服务，仅支持可添加多张弹性网卡的机型。 ● 弹性云服务器-物理机：基于擎天架构的服务器部署容器服务。
容器引擎	CCE集群支持Docker，并在部分场景中支持Containerd。 ● 1.23及以上的VPC网络集群都支持Containerd，容器隧道网络集群从1.23.2-r0开始支持Containerd。 ● CCE Turbo引擎支持Docker和Containerd。
节点规格	请根据业务需求选择相应的节点规格，不同区域/可用区支持的节点规格不同，请以CCE控制台呈现为准。
操作系统	选择操作系统类型，不同类型节点支持的操作系统有所不同。具体请参见**支持的节点规格**。 公共镜像：请选择节点对应的操作系统。
登录方式	● **密钥对** 选择用于登录本节点的密钥对，支持选择共享密钥。 密钥对用于远程登录节点时的身份认证。若没有密钥对，可单击选项框右侧的“创建密钥对”来新建。

存储配置：

配置节点云服务器上的存储资源，方便节点上的容器软件与容器应用使用。请根据实际场景设置磁盘大小。

表 8-4 存储配置参数

参数	参数说明
系统盘	节点云服务器使用的系统盘，供操作系统使用。您可以设置系统盘的规格为40GB-1024GB之间的数值，缺省值为50GB。 系统盘加密：系统盘加密功能可为您的数据提供强大的安全防护，加密磁盘生成的快照及通过这些快照创建的磁盘将自动继承加密功能。**目前仅在部分Region显示此选项，具体以界面为准。** ● 默认不加密。 ● 点选“加密”后，可在弹出的“加密设置”对话框中，选择已有的密钥，若没有可选的密钥，请单击后方的链接创建新密钥，完成创建后单击刷新按钮。

参数	参数说明
数据盘	**至少需要一块数据盘**，供容器运行时和Kubelet组件使用，**该数据盘不能被删除卸载，否则会导致节点不可用。** ● 第一块数据盘：供容器运行时和Kubelet组件使用。您可以自行设置数据盘的规格为20GB-32768GB之间的数值，缺省值为100GB。 ● 其他数据盘，您可以设置数据盘的规格为10GB-32768GB之间的数值，缺省值为100GB。 **说明** 节点规格为“磁盘增强型”或“超高I/O型”时，有一块数据盘可以是本地盘。 本地磁盘实例有宕机风险，不保证数据可靠性，建议您使用云硬盘存储您的业务数据。 **高级配置** 单击后方的“展开高级设置”可进行如下设置： ● 数据盘空间分配：勾选“自定义容器引擎空间大小”后可定义容器引擎、镜像、临时存储在数据盘上占用的空间比例。容器引擎空间用于存放容器运行时工作目录、容器镜像数据以及镜像元数据；数据盘的剩余空间用于Pod配置文件、密钥及临时存储EmptyDir等。数据盘空间分配详细说明请参见**数据盘空间分配说明**。 ● 数据盘加密：数据盘加密功能可为您的数据提供强大的安全防护，加密磁盘生成的快照及通过这些快照创建的磁盘将自动继承加密功能。**目前仅在部分Region显示此选项，具体以界面为准。** – 默认不加密。 – 勾选“加密”后，可选择已有的密钥。若没有可选的密钥，请单击后方的链接创建新密钥，完成创建后单击刷新按钮。 **添加多个数据盘** 最多可以添加4个，默认情况直接创建为裸盘，不做任何处理。您也可以展开高级配置，选择如下配置。 ● 默认：默认情况直接创建为裸盘，不做任何处理。 ● 挂载到指定目录：将数据盘挂载到指定目录。 ● 作为持久存储卷：适用于对PV有性能要求的场景。持久存储卷的节点会添加上node.kubernetes.io/local-storage-persistent标签，取值为linear或striped。 ● 作为临时存储卷：适用于对EmptyDir有性能要求的场景。 **说明** ● 本地持久卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=2.1.23，推荐使用>=2.1.23版本。 ● 本地临时卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=1.2.29。 **本地持久卷**和**本地临时卷**支持如下两种写入模式。 ● 线性（linear）：线性逻辑卷是将一个或多个物理卷整合为一个逻辑卷，实际写入数据时会先往一个基本物理卷上写入，当存储空间占满时再往另一个基本物理卷写入。

参数	参数说明
	• 条带化（striped）：创建逻辑卷时指定条带化，当实际写入数据时会将连续数据分成大小相同的块，然后依次存储在多个物理卷上，实现数据的并发读写从而提高读写性能。条带化模式的存储池不支持扩容。多块存储卷才能选择条带化。

网络配置：

配置节点云服务器的网络资源，用于访问节点和容器应用。

表 8-5 网络配置参数

参数	参数说明
节点子网	节点子网默认使用创建集群时的子网配置，也可以选择其他子网。
节点IP	支持随机分配。
关联安全组	指定节点池创建出来的节点使用哪个安全组。最多选择5个安全组。 创建集群时会默认创建一个节点安全组，名称为**{集群名}-cce-node-{随机ID}**，默认会使用该安全组。 节点安全组需要放通一些端口以保障节点通信，如选择其他安全组，需要放通这些端口。

高级配置：

节点能力增强，可在此配置节点的标签、污点、启动命令等功能。

表 8-6 高级配置参数

参数	参数说明
K8s标签	设置附加到Kubernetes对象（比如Pod）上的键值对，填写键值对后，单击“确认添加”。最多可以添加20条标签。 使用该标签可区分不同节点，可结合工作负载的亲和能力实现容器Pod调度到指定节点的功能。详细请参见**Labels and Selectors**。

参数	参数说明
污点（Taints）	默认为空。支持给节点加污点来设置反亲和性，每个节点最多配置20条污点，每条污点包含以下3个参数： • Key：必须以字母或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符；另外可以使用DNS子域作为前缀。 • Value：必须以字符或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符。 • Effect：只可选NoSchedule，PreferNoSchedule或NoExecute。 污点的使用请参见**管理节点污点（Taint）**。 **说明** 对于1.19及以下版本集群，有可能会出现污点打上之前负载已经调度到节点上，如果需要避免这种情况，请选择1.19及以上集群。
最大实例数	节点最大可以正常运行的实例数(Pod)，该数量包含系统默认实例。 该设置的目的为防止节点因管理过多实例而负载过重，请根据您的业务需要进行设置。 节点最多能创建多少个Pod还受其他因素影响，具体请参见**节点可创建的最大Pod数量说明**。
云服务器组	云服务器组是对云服务器的一种逻辑划分，同一云服务器组中的云服务器遵从同一策略。 反亲和性策略：同一云服务器组中的云服务器分散地创建在不同主机上，提高业务的可靠性。 选择已创建的云服务器组，或单击“新建云服务器组”创建，创建完成后单击刷新按钮。
安装前执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装前执行，可能导致Kubernetes软件无法正常安装，需谨慎使用。
安装后执行脚本	请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装后执行，不影响Kubernetes软件安装。 **说明** 请不要在安装后执行脚本中使用**reboot**命令立即重启，如果需要重启，可以使用**shutdown -r 1**命令延迟1分钟重启。
委托	委托是由租户管理员在统一身份认证服务上创建的。通过委托，可以将云主机资源共享给其他账号，或委托更专业的人或团队来代为管理。 如果没有委托请单击右侧“新建委托”创建。

步骤4 单击“下一步：规格确认”。

步骤5 单击“提交”。

----结束

8.3 管理节点池

8.3.1 更新节点池

约束与限制

- 修改节点池资源标签时，修改后的配置仅对新增节点生效，存量节点如需同步配置，需要手动重置存量节点。
- 修改K8s标签和污点数据会自动同步已有节点，无需重置节点。

更新节点池

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击节点池名称后的“更新”，在弹出的“更新节点池”页面中配置参数。

基础配置

表 8-7 基础配置

参数	参数说明
节点池名称	自定义节点池名称。
节点数量	请根据业务需求调整节点个数。
弹性伸缩	默认不开启。 单击  开启后，节点池将根据业务需求自动创建或删除节点池内的节点，参数设置如下： • 节点数上限和节点数下限：您可设置节点数的上限和下限，保证节点数在合理的范围内伸缩。 • 优先级：该数值表示节点池之间进行弹性扩容的优先级，数值越大优先级越高，如设置为4的节点池比设置为1的节点池优先启动扩容。若多个节点池的值设置相同，如都设置为2，表示这几个节点池之间不分优先级，系统将按最小资源浪费原则进行扩容。 更新优先级后，配置将在1分钟内生效。 • 冷却时间：请设置时间，单位为分钟。弹性缩容冷却时间是指当前节点池扩容出的节点多长时间不能被缩容。 为保证功能的正常使用，节点池开启弹性扩缩容功能后，请务必安装**AutoScaler插件**。

高级配置

表 8-8 高级配置

参数	参数说明
K8s标签	设置附加到Kubernetes对象（比如Pod）上的键值对，填写键值对后，单击“确认添加”。最多可以添加20条标签。 使用该标签可区分不同节点，可结合工作负载的亲和能力实现容器Pod调度到指定节点的功能。详细请参见**Labels and Selectors**。 **说明** 修改“K8s标签”后，节点池下的存量节点会同步更新。
污点（Taints）	默认为空。支持给节点加污点来设置反亲和性，每个节点最多配置20条污点，每条污点包含以下3个参数： • Key：必须以字母或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符；另外可以使用DNS子域作为前缀。 • Value：必须以字符或数字开头，可以包含字母、数字、连字符、下划线和点，最长63个字符。 • Effect：只可选NoSchedule，PreferNoSchedule或NoExecute。 污点的使用请参见**管理节点污点（Taint）**。 **说明** 修改“污点（Taints）”后，节点池下的存量节点会同步更新。
编辑密钥对	仅使用密钥对登录的节点池支持编辑，您可重新选择一个密钥对。 **说明** 编辑密钥对后，对新增的节点自动生效，存量节点需要手动重置节点后生效。

步骤4 配置完成后，单击“确定”。

节点池参数更新后，前往“节点管理”页面，可查看节点池所属节点存在更新，可通过重置节点同步节点配置，与节点池配置保持一致。

----结束

8.3.2 节点池配置管理

约束与限制

默认节点池DefaultPool不支持如下管理操作。

配置管理

为方便对CCE集群中的Kubernetes配置参数进行管理，CCE提供了配置管理功能，通过该功能您可以对核心组件进行深度配置，更多信息请参见**kubelet**。

仅支持在**v1.15及以上版本**的集群中对节点池进行配置，V1.15以下版本不显示该功能。

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击节点池名称后的“更多 > 配置管理”。

步骤4 在侧边栏滑出的“配置管理”窗口中，根据业务需求修改Kubernetes的参数值：

表 8-9 kubelet 组件配置

参数	参数说明	默认值	修改说明	备注
cpu-manager-policy	CPU管理策略配置，详情请参见**CPU调度**。 • none：关闭工作负载实例独占CPU的功能，优点是CPU共享池的可分配核数较多。 • static：开启工作负载实例独占CPU，适用于对CPU缓存和调度延迟敏感的场景。	none	-	-
kube-api-qps	与kube-apiserver通信的QPS	100	-	-
kube-api-burst	与kube-apiserver通信的burst	100	-	-
max-pods	kubelet管理的pod上限	• CCE集群：由**节点最大实例数**设置决定。 • CCE Turbo集群：由**节点网卡数量**决定。	-	-
pod-pids-limit	限制Pod中的进程数	-1	-	-
with-local-dns	是否使用本地IP作为该节点的ClusterDNS	false	-	-
event-qps	事件创建QPS限制	5	-	-

参数	参数说明	默认值	修改说明	备注
allowed-unsafe-sysctls	允许使用的不安全系统配置。 CCE从**1.17.17**集群版本开始，kube-apiserver开启了pod安全策略，需要在pod安全策略的allowedUnsafeSysctls中增加相应的配置才能生效（1.17.17以下版本的集群可不配置）。详情请参见**Pod安全策略开放非安全系统配置示例**。	[]	-	-
over-subscription-resource	节点超卖特性。 设置为true表示开启节点超卖特性。	false	-	-
colocation	节点混部特性。 设置为true表示开启节点混部特性。	false	-	-
kube-reserved-mem system-reserved-mem	节点内存预留。	随节点规格变动，具体请参见**节点预留资源策略说明**	-	kube-reserved-mem，system-reserved-mem之和小于内存的一半
topology-manager-policy	设置拓扑管理策略。 合法值包括： • restricted：kubelet仅接受在所请求资源上实现最佳NUMA对齐的Pod。 • best-effort：kubelet会优先选择在 CPU 和设备资源上实现NUMA对齐的Pod。 • none（默认）：不启用拓扑管理策略。 • single-numa-node：kubelet仅允许在 CPU和设备资源上对齐到同一NUMA节点的Pod。	none	-	**须知** 请谨慎修改，修改topology-manager-policy和topology-manager-scope会重启kubelet，并且以更改后的策略重新计算容器实例的资源分配，这有可能导致已经运行的容器实例重启甚至无法进行资源分配。

参数	参数说明	默认值	修改说明	备注
topology-manager-scope	设置拓扑管理策略的资源对齐粒度。合法值包括： • container（默认）：对齐粒度为容器级 • pod：对齐粒度为pod级	container		
resolv-conf	容器指定DNS解析配置文件	默认为空值	-	-
runtime-request-timeout	除长期运行的请求（pull、 logs、exec和attach）之外所有运行时请求的超时时长。	默认为2m0s	-	-
registry-pull-qps	每秒钟可以执行的镜像仓库拉取操作限值。	默认为5	取值范围为1~50	-
registry-burst	突发性镜像拉取的上限值，允许镜像拉取临时上升到所指定数量。	默认为10	取值范围为1~100，且取值必须大于等于registry-pull-qps的值。	-
serialize-image-pulls	开启时会通知kubelet每次仅拉取一个镜像。	默认为true	-	-
evictionHard: memory.available	硬驱逐信号memory.available门限	固定为100Mi	-	关于节点压力驱逐详情请参考**节点压力驱逐**。 **须知** 驱逐门限相关配置请**谨慎修改**，不合理的配置可能会导致节点频繁触发驱逐或节点已过载但未触发驱逐。 nodefs与imagefs分别对应kubelet及容器引擎使用的文件系统分区。
evictionHard: nodefs.available	硬驱逐信号nodefs.available门限	默认10%	范围1%~99%	
evictionHard: nodefs.inodesFree	硬驱逐信号nodefs.inodesFree门限	默认5%	范围1%~99%	

参数	参数说明	默认值	修改说明	备注
evictionHard: imagefs.available	硬驱逐信号imagefs.available门限	默认10%	范围1%~99%	
evictionHard: imagefs.inodesFree	硬驱逐信号imagefs.inodesFree门限	默认为空，不设置	范围1%~99%	
evictionHard: pid.available	硬驱逐信号pid.available门限	默认10%	范围1%~99%	
evictionSoft: memory.available	软驱逐信号memory.available门限	默认为空，不设置	范围100Mi~1000000Mi，需同时配置对应驱逐信号的evictionSoftGracePeriod来配置驱逐宽限期，此值要求大于对应硬驱逐信号的门限	
evictionSoft: nodefs.available	软驱逐信号nodefs.available门限	默认为空，不设置	范围1%~99%，若设置，需同时配置对应驱逐信号的evictionSoftGracePeriod来配置驱逐宽限期，此值要求大于对应硬驱逐信号的门限	

参数	参数说明	默认值	修改说明	备注
evictionSoft: nodefs.inodesFree	软驱逐信号nodefs.inodesFree门限	默认为空，不设置	范围1%~99%，需同时配置对应驱逐信号的evictionSoftGracePeriod来配置驱逐宽限期，此值要求大于对应硬驱逐信号的门限	
evictionSoft: imagefs.available	软驱逐信号imagefs.available门限	默认为空，不设置	范围1%~99%，需同时配置对应驱逐信号的evictionSoftGracePeriod来配置驱逐宽限期，此值要求大于对应硬驱逐信号的门限	
evictionSoft: imagefs.inodesFree	软驱逐信号imagefs.inodesFree门限	默认为空，不设置	范围1%~99%，需同时配置对应驱逐信号的evictionSoftGracePeriod来配置驱逐宽限期，此值要求大于对应硬驱逐信号的门限	

参数	参数说明	默认值	修改说明	备注
evictionSoft: pid.available	软驱逐信号 pid.available门限	默认为空，不设置	范围1%~99%，需同时配置对应驱逐信号的evictionSoftGracePeriod来配置驱逐宽限期，此值要求大于对应硬驱逐信号的门限	

表 8-10 kube-proxy 组件配置

参数	参数说明	默认值	修改说明
conntrack-min	系统中最大的连接跟踪表项数目。 可通过以下命令查询： `sysctl -w net.nf_conntrack_max`	131072	-
conntrack-tcp-timeout-close-wait	控制TCP连接在关闭状态下等待的时间。 可通过以下命令查询： `sysctl -w net.netfilter.nf_conntrack_tcp_timeout_close_wait`	1h0m0s	-

表 8-11 网络组件配置（仅 CCE Turbo 集群可见）

参数	参数说明	默认值	修改说明
nic-minimum-target	节点池级别的节点最少绑定容器网卡数	默认：10	-
nic-maximum-target	节点池级别的节点预热容器网卡上限检查值	默认：0	-
nic-warm-target	节点池级别的节点动态预热容器网卡数	默认：2	-

参数	参数说明	默认值	修改说明
nic-max-above-warm-target	节点池级别的节点预热容器网卡回收阈值	默认：2	-

表 8-12 节点池 Pod 安全组配置（仅 CCE Turbo 集群可见）

参数	参数说明	默认值	修改说明
security_groups_for_nodepool	• 节点池上Pod默认使用的安全组，可填写安全组 ID，不配置则使用集群容器网络的默认安全组，并且最多可同时指定5个安全组ID，中间以英文分号（;）分隔。 • 优先级低于SecurityGroup资源对象配置的安全组。	-	-

表 8-13 容器引擎 Docker 配置（仅使用 Docker 的节点池可见）

参数	参数说明	默认值	修改说明
native-umask	`--exec-opt native.umask	normal	不支持修改
docker-base-size	`--storage-opts dm.basesize	0	不支持修改
insecure-registry	不安全的镜像源地址	false	不支持修改
limitcore	容器core文件的大小限制，单位是Byte。 如果不设置大小限制，可设置为infinity。	5368709120	-
default-ulimit-nofile	容器内句柄数限制	{soft}:{hard}	该值大小不可超过节点内核参数nr_open的值，且不能是负数。 节点内核参数nr_open可通过以下命令获取： `sysctl -a | grep nr_open`

参数	参数说明	默认值	修改说明
image-pull-progress-timeout	如果超时之前镜像没有拉取成功，本次镜像拉取将会被取消。	默认为1m0s	该参数在v1.25.3-r0版本开始支持

表 8-14 容器引擎 Containerd 配置（仅使用 Containerd 的节点池可见）

参数	参数说明	默认值	修改说明
devmapper-base-size	单容器可用数据空间	-	不支持修改
limitcore	容器core文件的大小限制，单位是Byte。 如果不设置大小限制，可设置为infinity。	5368709120	-
default-ulimit-nofile	容器内句柄数限制	1048576	该值大小不可超过节点内核参数nr_open的值，且不能是负数。 节点内核参数nr_open可通过以下命令获取： `sysctl -a | grep nr_open`
image-pull-progress-timeout	如果超时之前镜像没有拉取成功，本次镜像拉取将会被取消。	默认为1m0s	该参数在v1.25.3-r0版本开始支持

步骤5 单击“确定”，完成配置操作。

----结束

8.3.3 拷贝节点池

通过CCE控制台可以方便的拷贝现有节点池的配置，从而创建新的节点池。

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击节点池名称后的“更多 > 拷贝”。

步骤4 在弹出的“拷贝节点池”窗口中，可以看到拷贝的节点池配置，您可以根据需要进行修改，配置项详情请参见**创建节点池**。确定配置后单击“下一步：规格确认”。

步骤5 在“规格确认”步骤中再次确认规格并单击“提交”，即可完成节点池的拷贝并创建新的节点池。

----结束

8.3.4 同步节点池

在节点池配置更新后，节点池中的已有节点无法自动同步部分配置，您可以手动同步节点配置。

须知

- 批量同步过程中请勿删除或重置节点，否则可能导致节点池配置同步失败。
- 该操作涉及重置节点，**节点上已运行的工作负载业务可能会由于单实例部署、可调度资源不足等原因产生中断**，请您合理评估升级风险，并挑选业务低峰期进行，或对关键业务应用设置PDB策略（Pod Disruption Budget，即**干扰预算**），升级过程中将严格根据PDB规则保障关键业务的可用性。
- 同步已有节点时，节点会被重置，系统盘和数据盘将会被清空，请在同步前备份重要数据。
- 仅部分节点池参数可通过重置节点同步，详细约束如下：
  - 修改节点池资源标签时，修改后的配置仅对新增节点生效，存量节点如需同步配置，需要手动重置存量节点。
  - 修改K8s标签和污点数据会自动同步已有节点，无需重置节点。

单个节点同步

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点”页签。

步骤3 节点池中的存量节点将提示“存在更新”。

步骤4 单击“存在更新”，在提示窗口中确认是否立即重置节点。

----结束

批量同步

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击节点池名称后的“更多 > 同步”。

步骤4 在弹出的“批量同步”窗口中，设置同步参数。

- 操作系统：该项无需设置，用于展示目标版本的镜像信息。
- 同步方式：当前支持节点重置方式进行同步。
- 每批最大同步节点数：节点升级时，允许节点不可用的最大数量。节点重置方式进行同步时节点将不可用，请合理设置该参数，尽量避免出现集群节点不可用数量过多导致Pod无法调度的情况。
- 节点列表：选择需要同步节点池配置的节点。

步骤5 单击“确定”，即可开始节点池的同步。

----结束

8.3.5 升级操作系统

当CCE发布新版本的操作系统镜像时，已有节点无法自动升级，您可以手动进行批量升级。

须知

该操作会通过重置节点的方式升级操作系统，**节点上已运行的工作负载业务可能会由于单实例部署、可调度资源不足等原因产生中断**，请您合理评估升级风险，并挑选业务低峰期进行，或对关键业务应用设置PDB策略（Pod Disruption Budget，即**干扰预算**），升级过程中将严格根据PDB规则保障关键业务的可用性。

约束与限制

- 使用私有镜像的节点暂不支持升级操作。
- 老版本的节点升级操作系统时可能存在兼容性问题，请手动重置节点完成操作系统升级。

默认节点池

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击默认节点池名称后的“升级”。

步骤4 在弹出的“升级操作系统”窗口中，设置升级参数。

- 目标操作系统：该项无需设置，用于展示目标版本的镜像信息。
- 升级方式：当前支持节点重置方式进行升级。
- 每批最大升级节点数：节点升级时，允许节点不可用的最大数量。节点重置方式进行同步时节点将不可用，请合理设置该参数，尽量避免出现集群节点不可用数量过多导致Pod无法调度的情况。
- 节点列表：选择需要升级的节点。
- 登录方式：
  - **密钥对**

    选择用于登录本节点的密钥对，支持选择共享密钥。

    密钥对用于远程登录节点时的身份认证。若没有密钥对，可单击选项框右侧的“创建密钥对”来新建。
- 安装前执行脚本：请输入脚本命令，大小限制为0~1000字符。

  脚本将在Kubernetes软件安装前执行，可能导致Kubernetes软件无法正常安装，需谨慎使用。
- 安装后执行脚本：请输入脚本命令，大小限制为0~1000字符。

  脚本将在Kubernetes软件安装后执行，不影响Kubernetes软件安装。

步骤5 单击“确定”，即可开始操作系统滚动升级。

----结束

非默认节点池

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击节点池名称后的“更多 > 同步”。

步骤4 在弹出的“批量同步”窗口中，设置同步参数。

- 操作系统：该项无需设置，用于展示目标版本的镜像信息。
- 同步方式：当前支持节点重置方式进行同步。
- 每批最大同步节点数：节点升级时，允许节点不可用的最大数量。节点重置方式进行同步时节点将不可用，请合理设置该参数，尽量避免出现集群节点不可用数量过多导致Pod无法调度的情况。
- 节点列表：选择需要同步节点池配置的节点。

步骤5 单击“确定”，即可开始节点池的同步。

----结束

8.3.6 迁移节点

您可以将同一个集群下某个节点池中的节点迁移到默认节点池（defaultpool）中，暂不支持将默认节点池（defaultpool）中的节点迁移到其他节点池中，也不支持将自定义节点池中的节点迁移到其他自定义节点池。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“节点管理”，并切换至“节点池”页签。

步骤3 单击待迁移的节点池名称后的“节点列表”。

步骤4 在需要迁移的节点的“操作”栏中，单击“更多 > 迁移”，迁移单个节点。

步骤5 在弹出的“迁移节点”窗口中进行确认。

说明

迁移完成后，节点原有资源标签、K8s标签、污点不受影响。

----结束

8.3.7 删除节点池

删除节点池，会先删除节点池中的节点，节点删除后，原有节点上的工作负载实例会自动迁移至其他节点池的可用节点。

注意事项

- 删除节点池会同时删除节点池下的全部节点，请及时备份数据，避免重要数据丢失。
- 删除节点会涉及Pod迁移，可能会影响业务，请在业务低峰期操作。如果Pod具有特定的节点选择器，且集群中的其他节点均不符合标准，则工作负载实例可能仍处于无法安排的状态。
- 删除过程中，系统会把当前节点池中的节点均设置为不可调度状态。

操作步骤

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签。

步骤3 单击节点池名称后的“更多 > 删除”。

步骤4 在弹出的“删除节点池”窗口中，请仔细阅读界面提示。

步骤5 确定要对节点池进行删除操作后，请在弹窗中单击“是”，即可完成节点池的删除。

----结束

9 工作负载

9.1 工作负载概述

工作负载是在Kubernetes上运行的应用程序。无论您的工作负载是单个组件还是协同工作的多个组件，您都可以在Kubernetes上的一组Pod中运行它。在Kubernetes中，工作负载是对一组Pod的抽象模型，用于描述业务的运行载体，包括Deployment、StatefulSet、DaemonSet、Job、CronJob等多种类型。

云容器引擎CCE提供基于Kubernetes原生类型的容器部署和管理能力，支持容器工作负载部署、配置、监控、扩容、升级、卸载、服务发现及负载均衡等生命周期管理。

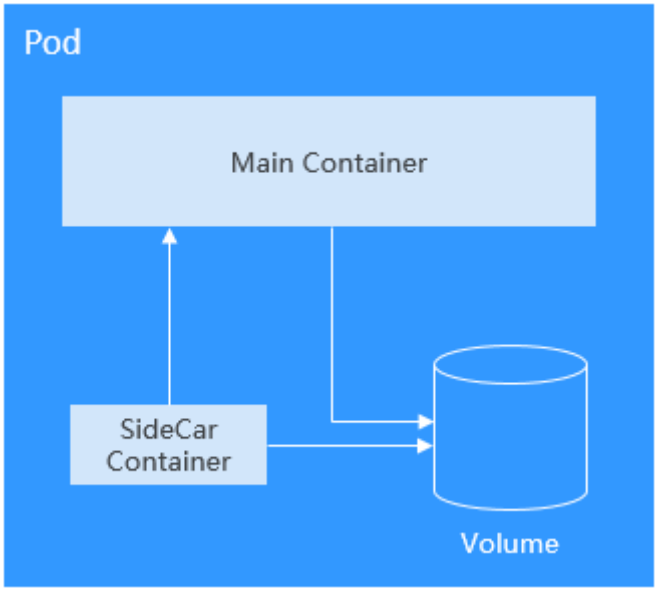
Pod

Pod是Kubernetes创建或部署的最小单位。一个Pod封装一个或多个容器（container）、存储资源（volume）、一个独立的网络IP以及管理控制容器运行方式的策略选项。

Pod使用主要分为两种方式：

- Pod中运行一个容器。这是Kubernetes最常见的用法，您可以将Pod视为单个封装的容器，但是Kubernetes是直接管理Pod而不是容器。
- Pod中运行多个需要耦合在一起工作、需要共享资源的容器。通常这种场景下应用包含一个主容器和几个辅助容器（SideCar Container），如图9-1所示，例如主容器为一个web服务器，从一个固定目录下对外提供文件服务，而辅助容器周期性的从外部下载文件存到这个固定目录下。

图 9-1 Pod

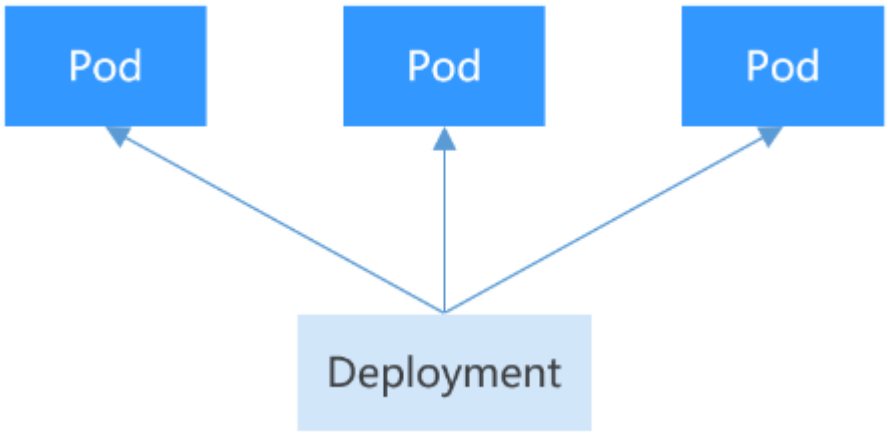


实际使用中很少直接创建Pod，而是使用Kubernetes中称为Controller的抽象层来管理Pod实例，例如Deployment和Job。Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力。通常，Controller会使用Pod Template来创建相应的Pod。

Deployment

Pod是Kubernetes创建或部署的最小单位，但是Pod是被设计为相对短暂的一次性实体，Pod可以被驱逐（当节点资源不足时）、随着集群的节点崩溃而消失。Kubernetes提供了Controller（控制器）来管理Pod，Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力，其中最为常用的就是Deployment。

图 9-2 Deployment



一个Deployment可以包含一个或多个Pod副本，每个Pod副本的角色相同，所以系统会自动为Deployment的多个Pod副本分发请求。

Deployment集成了上线部署、滚动升级、创建副本、恢复上线的功能，在某种程度上，Deployment实现无人值守的上线，大大降低了上线过程的复杂性和操作风险。

StatefulSet

Deployment控制器下的Pod都有个共同特点，那就是每个Pod除了名称和IP地址不同，其余完全相同。需要的时候，Deployment可以通过Pod模板创建新的Pod；不需要的时候，Deployment就可以删除任意一个Pod。

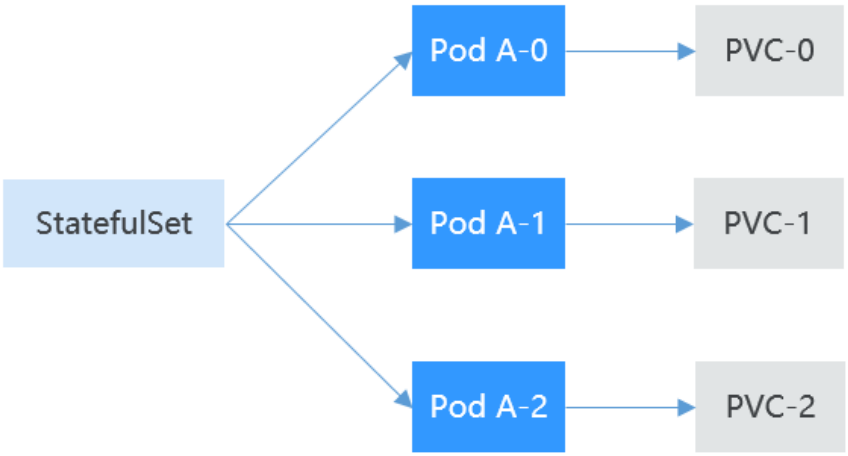
但是在某些场景下，这并不满足需求，比如有些分布式的场景，要求每个Pod都有自己单独的状态时，比如分布式数据库，每个Pod要求有单独的存储，这时Deployment就不能满足需求了。

详细分析下有状态应用的需求，分布式有状态的特点主要是应用中每个部分的角色不同（即分工不同），比如数据库有主备，Pod之间有依赖，对应到Kubernetes中就是对Pod有如下要求：

- Pod能够被别的Pod找到，这就要求Pod有固定的标识。
- 每个Pod有单独存储，Pod被删除恢复后，读取的数据必须还是以前那份，否则状态就会不一致。

Kubernetes提供了StatefulSet来解决这个问题，其具体如下：

1. StatefulSet给每个Pod提供固定名称，Pod名称增加从0-N的固定后缀，Pod重新调度后Pod名称和HostName不变。
2. StatefulSet通过Headless Service给每个Pod提供固定的访问域名，Service的概念会在后面章节中详细介绍。
3. StatefulSet通过创建固定标识的PVC保证Pod重新调度后还是能访问到相同的持久化数据。

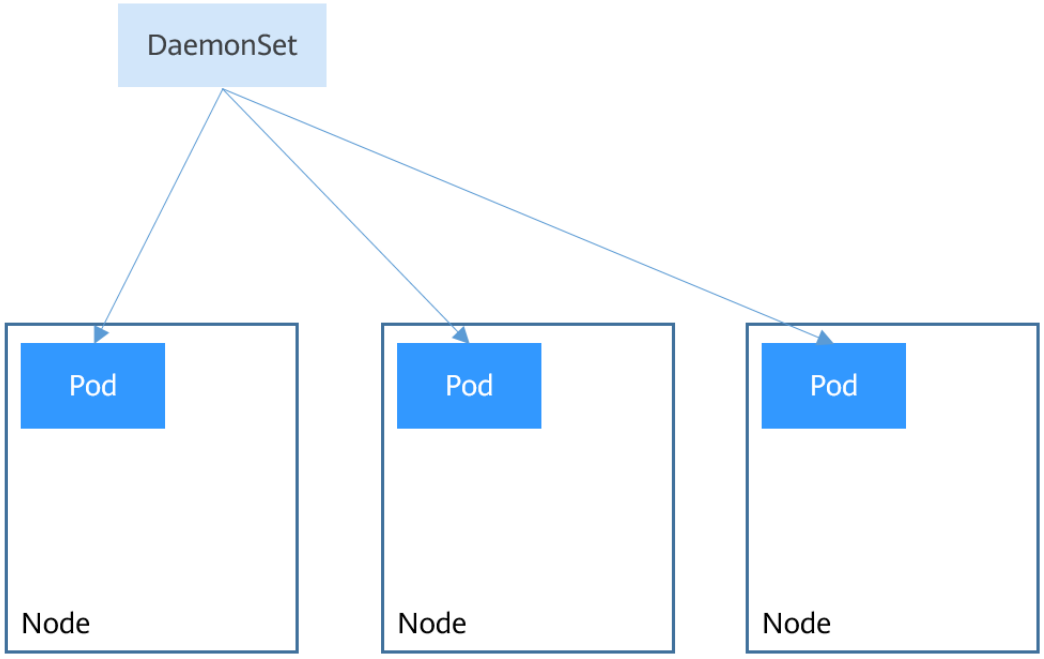


DaemonSet

DaemonSet是这样一种对象（守护进程），它在集群的每个节点上运行一个Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资源监控等，这类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes的kube-proxy。

DaemonSet跟节点相关，如果节点异常，也不会在其他节点重新创建。

图 9-3 DaemonSet



Job 和 CronJob

Job和CronJob是负责批量处理短暂的一次性任务（short lived one-off tasks），即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

- Job：是Kubernetes用来控制批处理型任务的资源对象。批处理业务与长期伺服业务（Deployment、Statefulset）的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出（Pod自动删除）。
- CronJob：是基于时间的Job，就类似于Linux系统的crontab文件中的一行，在指定的时间周期运行指定的Job。

任务负载的这种用完即停止的特性特别适合一次性任务，比如持续集成。

工作负载生命周期说明

表 9-1 状态说明

状态	说明
运行中	所有实例都处于运行中、或实例数为0时显示此状态。
未就绪	容器处于异常、负载下实例没有正常运行时显示此状态。
处理中	负载没有进入运行状态但也没有报错时显示此状态。
可用	当多实例无状态工作负载运行过程中部分实例异常，可用实例不为0，工作负载会处于可用状态。
执行完成	任务执行完成，仅普通任务存在该状态。

状态	说明
已停止	触发停止操作后，工作负载会处于停止状态，实例数变为0。v1.13之前的版本存在此状态。
删除中	触发删除操作后，工作负载会处于删除中状态。

9.2 创建工作负载

9.2.1 创建无状态负载（Deployment）

操作场景

在运行中始终不保存任何数据或状态的工作负载称为“无状态负载 Deployment”，例如nginx。您可以通过控制台或kubectl命令行创建无状态负载。

前提条件

- 在创建容器工作负载前，您需要存在一个可用集群。若没有可用集群，请参照**创建集群**中内容创建。
- 若工作负载需要被外网访问，请确保集群中至少有一个节点已绑定弹性IP，或已创建负载均衡实例。

  说明

  单个实例（Pod）内如果有多个容器，请确保容器使用的端口不冲突，否则部署会失败。

通过控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤3 配置工作负载的信息。

基本信息

- 负载类型：选择无状态工作负载Deployment。工作负载类型的介绍请参见**工作负载概述**。
- 负载名称：填写工作负载的名称。请输入1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。
- 命名空间：选择工作负载的命名空间，默认为default。您可以单击后面的“创建命名空间”，命名空间的详细介绍请参见**创建命名空间**。
- 实例数量：填写实例的数量，即工作负载Pod的数量。
- 时区同步：选择是否开启时区同步。开启后容器与节点使用相同时区（时区同步功能依赖容器中挂载的本地磁盘，请勿修改删除），时区同步详细介绍请参见**时区同步**。

容器配置

- 容器信息

  Pod中可以配置多个容器，您可以单击右侧“添加容器”为Pod配置多个容器。

  - 基本信息：配置容器的基本信息。

参数	说明
容器名称	为容器命名。
更新策略	镜像更新/拉取策略。可以勾选“总是拉取镜像”，表示每次都从镜像仓库拉取镜像；如不勾选则优使用节点已有的镜像，如果没有这个镜像再从镜像仓库拉取。
镜像名称	单击后方“选择镜像”，选择容器使用的镜像。 如果需要使用第三方镜像，请参见**使用第三方镜像**。
镜像版本	选择需要部署的镜像版本。
CPU配额	■ 申请：容器需要使用的最小CPU值，默认0.25Core。 ■ 限制：允许容器使用的CPU最大值。建议设容器配额的最高限额，避免容器资源超额导致系统故障。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
内存配额	■ 申请：容器需要使用的内存最小值，默认512MiB。 ■ 限制：允许容器使用的内存最大值。如果超过，容器会被终止。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
GPU配额（可选）	当集群中包含GPU节点时，才能设置GPU配额，且集群中需安装**gpu-device-plugin**插件。 ■ 不限制：表示不使用GPU。 ■ 独享：单个容器独享GPU。 ■ 共享：容器需要使用的GPU百分比，例如设置为10%，表示该容器需使用GPU资源的10%。 关于如何在集群中使用GPU，请参见**使用Kubernetes默认GPU调度**。
特权容器（可选）	特权容器是指容器里面的程序具有一定的特权。 若选中，容器将获得超级权限，例如可以操作宿主机上面的网络设备、修改内核参数等。

参数	说明
初始化容器（可选）	选择容器是否作为初始化（Init）容器。初始化（Init）容器不支持设置健康检查。 Init容器是一种特殊容器，可以在Pod中的其他应用容器启动之前运行。每个Pod中可以包含多个容器，同时Pod中也可以有一个或多个先于应用容器启动的Init容器，当所有的Init 容器运行完成时，Pod中的应用容器才会启动并运行。详细说明请参见**Init 容器**。

  - 生命周期（可选）：在容器的生命周期的特定阶段配置需要执行的操作，例如启动命令、启动后处理和停止前处理，详情请参见**设置容器生命周期**。
  - 健康检查（可选）：根据需求选择是否设置存活探针、就绪探针及启动探针，详情请参见**设置容器健康检查**。
  - 环境变量（可选）：支持通过键值对的形式为容器运行环境设置变量，可用于把外部信息传递给Pod中运行的容器，可以在应用部署后灵活修改，详情请参见**设置环境变量**。
  - 数据存储（可选）：在容器内挂载本地存储或云存储，不同类型的存储使用场景及挂载方式不同，详情请参见**存储**。

    说明

    负载实例数大于1时，不支持挂载云硬盘类型的存储。
  - 安全设置（可选）：对容器权限进行设置，保护系统和其他容器不受其影响。请输入用户ID，容器将以当前用户权限运行。
  - 容器日志（可选）：容器标准输出日志将默认上报至 AOM 服务，无需独立配置。您可以手动配置日志采集路径，详情请参见**使用ICAgent采集容器日志**。

    如需要关闭当前负载的标准输出，您可在**标签与注解**中添加键为kubernetes.AOM.log.stdout，值为[]的注解，即可关闭当前负载下全部容器的标准输出。该注解的使用方法请参见**表9-16**。

- 镜像访问凭证：用于访问镜像仓库的凭证，默认取值为default-secret，使用default-secret可访问SWR镜像仓库的镜像。default-secret详细说明请参见**default-secret**。
- GPU显卡（可选）：默认为不限制。当集群中存在GPU节点时，工作负载实例可以调度到指定GPU显卡类型的节点上。

服务配置（可选）

服务（Service）可为Pod提供外部访问。每个Service有一个固定IP地址，Service将访问流量转发给Pod，而且Service可以为这些Pod自动实现负载均衡。

您也可以在创建完工作负载之后再创建Service，不同类型的Service概念和使用方法请参见**服务概述**。

高级配置（可选）

- 升级策略：指定工作负载的升级方式及升级参数，支持滚动升级和替换升级，详情请参见**工作负载升级策略**。
- 调度策略：通过配置亲和与反亲和规则，可实现灵活的工作负载调度，支持节点亲和与Pod亲和/反亲和。详情请参见**调度策略（亲和与反亲和）**。

- 容忍策略：容忍策略与节点的污点能力配合使用，允许（不强制）负载调度到带有与之匹配的污点的节点上，也可用于控制负载所在的节点被标记污点后负载的驱逐策略，详情请参见**容忍策略**。
- 标签与注解：以键值对形式为工作负载Pod添加标签或注解，填写完成后需单击“确认添加”。关于标签与注解的作用及配置说明，请参见**标签与注解**。
- DNS配置：为工作负载单独配置DNS策略，详情请参见**工作负载DNS配置说明**。
- 网络配置：
  - Pod入/出口带宽限速：支持为Pod设置入/出口带宽限速，详情请参见**Pod互访QoS限速**。

步骤4 单击右下角“创建工作负载”。

----结束

通过 kubectl 命令行创建

本节以nginx工作负载为例，说明kubectl命令创建工作负载的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建一个名为nginx-deployment.yaml的描述文件。其中，nginx-deployment.yaml为自定义名称，您可以随意命名。

vi nginx-deployment.yaml

描述文件内容如下。此处仅为示例，deployment的详细说明请参见**kubernetes官方文档**。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx    #若使用“我的镜像”中的镜像，请在SWR中获取具体镜像地址。
        imagePullPolicy: Always
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

以上yaml字段解释如**表9-2**。

表 9-2 deployment 字段详解

字段名称	字段说明	必选/可选
apiVersion	表示API的版本号。 **说明** 请根据集群版本输入： • 1.17及以上版本的集群中无状态应用apiVersion格式为**apps/v1** • 1.15及以下版本的集群中无状态应用apiVersion格式为**extensions/v1beta1**	必选
kind	创建的对象类别。	必选
metadata	资源对象的元数据定义。	必选
name	deployment的名称。	必选
spec	用户对deployment的详细描述的主体部分都在spec中给出。	必选
replicas	实例数量。	必选
selector	定义Deployment可管理的容器实例。	必选
strategy	升级类型。当前支持两种升级方式，默认为滚动升级。 • RollingUpdate：滚动升级。 • ReplaceUpdate：替换升级。	可选
template	描述创建的容器实例详细信息。	必选
metadata	元数据。	必选
labels	metadata.labels定义容器标签。	可选
spec: containers	• image（必选）：容器镜像名称。 • imagePullPolicy（可选）：获取镜像的策略，可选值包括Always（每次都尝试重新下载镜像）、Never（仅使用本地镜像）、IfNotPresent（如果本地有该镜像，则使用本地镜像，本地不存在时下载镜像），默认为Always。 • name（必选）：容器名称。	必选
imagePullSecrets	Pull镜像时使用的secret名称。若使用私有镜像，该参数为必选。 • 需要Pull SWR容器镜像仓库的镜像时，参数值固定为default-secret。 • 当Pull第三方镜像仓库的镜像时，需设置为创建的secret名称。	可选

步骤3 创建deployment。

kubectl create -f nginx-deployment.yaml

回显如下表示已开始创建deployment。

```
deployment "nginx" created
```

步骤4 查看deployment状态。

kubectl get deployment

deployment状态显示为Running，表示deployment已创建成功。

```
NAME         READY    UP-TO-DATE   AVAILABLE   AGE
nginx        1/1      1            1           4m5s
```

参数解析：

- NAME：工作负载名称。
- READY：表示工作负载的可用状态，显示为“可用Pod个数/期望Pod个数”。
- UP-TO-DATE：指当前已经完成更新的副本数。
- AVAILABLE：可用的Pod个数。
- AGE：已经运行的时间。

步骤5 若工作负载（即deployment）需要被访问（集群内访问或节点访问），您需要设置访问方式，具体请参见网络创建对应服务。

----结束

9.2.2 创建有状态负载（StatefulSet）

操作场景

在运行过程中会保存数据或状态的工作负载称为“有状态工作负载（statefulset）”。例如MySQL，它需要存储产生的新数据。

因为容器可以在不同主机间迁移，所以在宿主机上并不会保存数据，这依赖于CCE提供的高可用存储卷，将存储卷挂载在容器上，从而实现有状态工作负载的数据持久化。

约束与限制

- 当您删除或扩缩有状态负载时，为保证数据安全，系统并不会删除它所关联的存储卷。
- 当您删除一个有状态负载时，为实现有状态负载中的Pod可以有序停止，请在删除之前将副本数缩容到0。
- 您需要在创建有状态负载的同时，创建一个Headless Service，用于解决有状态负载Pod互相访问的问题，详情请参见**Headless Service**。
- 节点不可用时，Pod状态变为“未就绪”，此时需要手工删除有状态工作负载的Pod，Pod实例才会迁移到正常节点上。

前提条件

- 在创建容器工作负载前，您需要存在一个可用集群。若没有请参照**创建集群**中内容创建。

- 若工作负载需要被外网访问，请确保集群中至少有一个节点已绑定弹性IP，或已创建负载均衡实例。

  说明

  单个实例（Pod）内如果有多个容器，请确保容器使用的端口不冲突，否则部署会失败。

通过控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤3 配置工作负载的信息。

基本信息

- 负载类型：选择有状态工作负载StatefulSet。工作负载类型的介绍请参见**工作负载概述**。
- 负载名称：填写工作负载的名称。请输入1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。
- 命名空间：选择工作负载的命名空间，默认为default。您可以单击后面的“创建命名空间”，命名空间的详细介绍请参见**创建命名空间**。
- 实例数量：填写实例的数量，即工作负载Pod的数量。
- 时区同步：选择是否开启时区同步。开启后容器与节点使用相同时区（时区同步功能依赖容器中挂载的本地磁盘，请勿修改删除），时区同步详细介绍请参见**时区同步**。

容器配置

- 容器信息

  Pod中可以配置多个容器，您可以单击右侧“添加容器”为Pod配置多个容器。

  – 基本信息：配置容器的基本信息。

参数	说明
容器名称	为容器命名。
更新策略	镜像更新/拉取策略。可以勾选“总是拉取镜像”，表示每次都从镜像仓库拉取镜像；如不勾选则优使用节点已有的镜像，如果没有这个镜像再从镜像仓库拉取。
镜像名称	单击后方“选择镜像”，选择容器使用的镜像。 如果需要使用第三方镜像，请参见**使用第三方镜像**。
镜像版本	选择需要部署的镜像版本。
CPU配额	▪ 申请：容器需要使用的最小CPU值，默认0.25Core。 ▪ 限制：允许容器使用的CPU最大值。建议设容器配额的最高限额，避免容器资源超额导致系统故障。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。

参数	说明
内存配额	■ 申请：容器需要使用的内存最小值，默认512MiB。 ■ 限制：允许容器使用的内存最大值。如果超过，容器会被终止。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
GPU配额（可选）	当集群中包含GPU节点时，才能设置GPU配额，且集群中需安装**gpu-device-plugin**插件。 ■ 不限制：表示不使用GPU。 ■ 独享：单个容器独享GPU。 ■ 共享：容器需要使用的GPU百分比，例如设置为10%，表示该容器需使用GPU资源的10%。 关于如何在集群中使用GPU，请参见**使用Kubernetes默认GPU调度**。
特权容器（可选）	特权容器是指容器里面的程序具有一定的特权。 若选中，容器将获得超级权限，例如可以操作宿主机上面的网络设备、修改内核参数等。
初始化容器（可选）	选择容器是否作为初始化（Init）容器。初始化（Init）容器不支持设置健康检查。 Init容器是一种特殊容器，可以在Pod中的其他应用容器启动之前运行。每个Pod中可以包含多个容器，同时Pod中也可以有一个或多个先于应用容器启动的Init容器，当所有的Init 容器运行完成时，Pod中的应用容器才会启动并运行。详细说明请参见**Init 容器**。

- 生命周期（可选）：在容器的生命周期的特定阶段配置需要执行的操作，例如启动命令、启动后处理和停止前处理，详情请参见**设置容器生命周期**。
- 健康检查（可选）：根据需求选择是否设置存活探针、就绪探针及启动探针，详情请参见**设置容器健康检查**。
- 环境变量（可选）：支持通过键值对的形式为容器运行环境设置变量，可用于把外部信息传递给Pod中运行的容器，可以在应用部署后灵活修改，详情请参见**设置环境变量**。
- 数据存储（可选）：在容器内挂载本地存储或云存储，不同类型的存储使用场景及挂载方式不同，详情请参见**存储**。

说明

- 有状态负载支持“动态挂载”云硬盘，详情请参见**有状态负载动态挂载云硬盘存储**及**有状态负载动态挂载本地持久卷**。

  动态挂载通过**volumeClaimTemplates**字段实现，并依赖于StorageClass动态创建能力。有状态工作负载通过volumeClaimTemplates字段为每一个Pod关联了一个独有的PVC，而这个PVC又会和对应的PV绑定。因此当Pod被重新调度后，仍然能够根据该PVC名称挂载原有的数据。

- 负载创建完成后，动态挂载的存储不支持更新。

  - 安全设置（可选）：对容器权限进行设置，保护系统和其他容器不受其影响。请输入用户ID，容器将以当前用户权限运行。
  - 容器日志（可选）：容器标准输出日志将默认上报至 AOM 服务，无需独立配置。您可以手动配置日志采集路径，详情请参见**使用ICAgent采集容器日志**。

    如需要关闭当前负载的标准输出，您可在**标签与注解**中添加键为kubernetes.AOM.log.stdout，值为[]的注解，即可关闭当前负载下全部容器的标准输出。该注解的使用方法请参见**表9-16**。

- 镜像访问凭证：用于访问镜像仓库的凭证，默认取值为default-secret，使用default-secret可访问SWR镜像仓库的镜像。default-secret详细说明请参见**default-secret**。
- GPU显卡（可选）：默认为不限制。当集群中存在GPU节点时，工作负载实例可以调度到指定GPU显卡类型的节点上。

实例间发现服务配置

Headless Service用于解决StatefulSet内Pod互相访问的问题，Headless Service给每个Pod提供固定的访问域名。具体请参见**Headless Service**。

服务配置（可选）

服务（Service）可为Pod提供外部访问。每个Service有一个固定IP地址，Service将访问流量转发给Pod，而且Service可以为这些Pod自动实现负载均衡。

您也可以在创建完工作负载之后再创建Service，不同类型的Service概念和使用方法请参见**服务概述**。

高级配置（可选）

- 升级策略：指定工作负载的升级方式及升级参数，支持滚动升级和替换升级，详情请参见**工作负载升级策略**。
- 实例管理策略（podManagementPolicy）：

  对于某些分布式系统来说，StatefulSet 的顺序性保证是不必要和/或者不应该的。这些系统仅仅要求唯一性和身份标志。

  - 有序策略：默认实例管理策略，有状态负载会逐个的、按顺序的进行部署、删除、伸缩实例, 只有前一个实例部署Ready或者删除完成后，有状态负载才会操作后一个实例。
  - 并行策略：支持有状态负载并行创建或者删除所有的实例，有状态负载发生变更时立刻在实例上生效。
- 调度策略：通过配置亲和与反亲和规则，可实现灵活的工作负载调度，支持节点亲和与Pod亲和/反亲和。详情请参见**调度策略（亲和与反亲和）**。

- 容忍策略：容忍策略与节点的污点能力配合使用，允许（不强制）负载调度到带有与之匹配的污点的节点上，也可用于控制负载所在的节点被标记污点后负载的驱逐策略，详情请参见**容忍策略**。
- 标签与注解：以键值对形式为工作负载Pod添加标签或注解，填写完成后需单击“确认添加”。关于标签与注解的作用及配置说明，请参见**标签与注解**。
- DNS配置：为工作负载单独配置DNS策略，详情请参见**工作负载DNS配置说明**。
- 网络配置：
  - Pod入/出口带宽限速：支持为Pod设置入/出口带宽限速，详情请参见**Pod互访QoS限速**。

步骤4 单击右下角“创建工作负载”。

----结束

通过 kubectl 命令行创建

本示例以nginx为例，并使用volumeClaimTemplates动态挂载云硬盘。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建一个名为nginx-statefulset.yaml的文件。

其中，nginx-statefulset.yaml为自定义名称，您可以随意命名。

vi nginx-statefulset.yaml

以下内容仅为示例，若需要了解statefulset的详细内容，请参考**kubernetes官方文档**。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-1
          image: nginx:latest
          imagePullPolicy: IfNotPresent
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
            limits:
              cpu: 250m
              memory: 512Mi
          volumeMounts:
            - name: test
              readOnly: false
              mountPath: /usr/share/nginx/html
              subPath: ''
      imagePullSecrets:
        - name: default-secret
      dnsPolicy: ClusterFirst
      volumes: []
```

```
  serviceName: nginx-svc
  replicas: 2
  volumeClaimTemplates:  #动态挂载云硬盘示例
    - apiVersion: v1
      kind: PersistentVolumeClaim
      metadata:
        name: test
        namespace: default
        annotations:
          everest.io/disk-volume-type: SAS  # 云硬盘的类型
        labels:
          failure-domain.beta.kubernetes.io/region: eu-west-0  #云硬盘所在的区域
          failure-domain.beta.kubernetes.io/zone:    #云硬盘所在的可用区，必须和工作负载部署的节点可用区一
致
      spec:
        accessModes:
          - ReadWriteOnce  # 云硬盘必须为ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        storageClassName: csi-disk #StorageClass的名称，云硬盘为csi-disk
  updateStrategy:
    type: RollingUpdate
```

vi nginx-headless.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  namespace: default
  labels:
    app: nginx
spec:
  selector:
    app: nginx
    version: v1
  clusterIP: None
  ports:
    - name: nginx
      targetPort: 80
      nodePort: 0
      port: 80
      protocol: TCP
  type: ClusterIP
```

步骤3 创建工作负载以及对应headless服务。

kubectl create -f nginx-statefulset.yaml

回显如下，表示有状态工作负载（stateful）已创建成功。

```
statefulset.apps/nginx created
```

kubectl create -f nginx-headless.yaml

回显如下，表示对应headless服务已创建成功。

```
service/nginx-svc created
```

步骤4 若工作负载需要被访问（集群内访问或节点访问），您需要设置访问方式，具体请参见网络创建对应服务。

----结束

9.2.3 创建守护进程集（DaemonSet）

操作场景

云容器引擎（CCE）提供多种类型的容器部署和管理能力，支持对容器工作负载的部署、配置、监控、扩容、升级、卸载、服务发现及负载均衡等特性。

其中守护进程集（DaemonSet）可以确保全部（或者某些）节点上仅运行一个Pod实例，当有节点加入集群时，也会为其新增一个 Pod 。 当有节点从集群移除时，这些Pod也会被回收。删除 DaemonSet 将会删除它创建的所有Pod。

使用DaemonSet的一些典型用法：

- 运行集群存储daemon，例如在每个节点上运行glusterd、ceph。
- 在每个节点上运行日志收集daemon，例如fluentd、logstash。
- 在每个节点上运行监控daemon，例如Prometheus Node Exporter、collectd、Datadog代理、New Relic代理，或Ganglia gmond。

一种简单的用法是为每种类型的守护进程在所有的节点上都启动一个DaemonSet。一个稍微复杂的用法是为同一种守护进程部署多个DaemonSet；每个具有不同的标志，并且对不同硬件类型具有不同的内存、CPU要求。

前提条件

在创建守护进程集前，您需要存在一个可用集群。若没有可用集群 ，请参照**创建集群**中内容创建。

通过控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤3 配置工作负载的信息。

基本信息

- 负载类型：选择守护进程DaemonSet。工作负载类型的介绍请参见**工作负载概述**。
- 负载名称：填写工作负载的名称。请输入1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。
- 命名空间：选择工作负载的命名空间，默认为default。您可以单击后面的“创建命名空间”，命名空间的详细介绍请参见**创建命名空间**。
- 时区同步：选择是否开启时区同步。开启后容器与节点使用相同时区（时区同步功能依赖容器中挂载的本地磁盘，请勿修改删除），时区同步详细介绍请参见**时区同步**。

容器配置

- 容器信息

  Pod中可以配置多个容器，您可以单击右侧“添加容器”为Pod配置多个容器。

  - 基本信息：配置容器的基本信息。

参数	说明
容器名称	为容器命名。
更新策略	镜像更新/拉取策略。可以勾选“总是拉取镜像”，表示每次都从镜像仓库拉取镜像；如不勾选则优使用节点已有的镜像，如果没有这个镜像再从镜像仓库拉取。
镜像名称	单击后方“选择镜像”，选择容器使用的镜像。 如果需要使用第三方镜像，请参见**使用第三方镜像**。
镜像版本	选择需要部署的镜像版本。
CPU配额	■ 申请：容器需要使用的最小CPU值，默认0.25Core。 ■ 限制：允许容器使用的CPU最大值。建议设容器配额的最高限额，避免容器资源超额导致系统故障。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
内存配额	■ 申请：容器需要使用的内存最小值，默认512MiB。 ■ 限制：允许容器使用的内存最大值。如果超过，容器会被终止。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
GPU配额（可选）	当集群中包含GPU节点时，才能设置GPU配额，且集群中需安装**gpu-device-plugin**插件。 ■ 不限制：表示不使用GPU。 ■ 独享：单个容器独享GPU。 ■ 共享：容器需要使用的GPU百分比，例如设置为10%，表示该容器需使用GPU资源的10%。 关于如何在集群中使用GPU，请参见**使用Kubernetes默认GPU调度**。
特权容器（可选）	特权容器是指容器里面的程序具有一定的特权。 若选中，容器将获得超级权限，例如可以操作宿主机上面的网络设备、修改内核参数等。
初始化容器（可选）	选择容器是否作为初始化（Init）容器。初始化（Init）容器不支持设置健康检查。 Init容器是一种特殊容器，可以在Pod中的其他应用容器启动之前运行。每个Pod中可以包含多个容器，同时Pod中也可以有一个或多个先于应用容器启动的Init容器，当所有的Init 容器运行完成时，Pod中的应用容器才会启动并运行。详细说明请参见**Init 容器**。

  - 生命周期（可选）：在容器的生命周期的特定阶段配置需要执行的操作，例如启动命令、启动后处理和停止前处理，详情请参见**设置容器生命周期**。
  - 健康检查（可选）：根据需求选择是否设置存活探针、就绪探针及启动探针，详情请参见**设置容器健康检查**。
  - 环境变量（可选）：支持通过键值对的形式为容器运行环境设置变量，可用于把外部信息传递给Pod中运行的容器，可以在应用部署后灵活修改，详情请参见**设置环境变量**。
  - 数据存储（可选）：在容器内挂载本地存储或云存储，不同类型的存储使用场景及挂载方式不同，详情请参见**存储**。
  - 安全设置（可选）：对容器权限进行设置，保护系统和其他容器不受其影响。请输入用户ID，容器将以当前用户权限运行。
  - 容器日志（可选）：容器标准输出日志将默认上报至 AOM 服务，无需独立配置。您可以手动配置日志采集路径，详情请参见**使用ICAgent采集容器日志**。

    如需要关闭当前负载的标准输出，您可在**标签与注解**中添加键为kubernetes.AOM.log.stdout，值为[]的注解，即可关闭当前负载下全部容器的标准输出。该注解的使用方法请参见**表9-16**。

- 镜像访问凭证：用于访问镜像仓库的凭证，默认取值为default-secret，使用default-secret可访问SWR镜像仓库的镜像。default-secret详细说明请参见**default-secret**。
- GPU显卡（可选）：默认为不限制。当集群中存在GPU节点时，工作负载实例可以调度到指定GPU显卡类型的节点上。

服务配置（可选）

服务（Service）可为Pod提供外部访问。每个Service有一个固定IP地址，Service将访问流量转发给Pod，而且Service可以为这些Pod自动实现负载均衡。

您也可以在创建完工作负载之后再创建Service，不同类型的Service概念和使用方法请参见**服务概述**。

高级配置（可选）

- 升级策略：指定工作负载的升级方式及升级参数，支持滚动升级和替换升级，详情请参见**工作负载升级策略**。
- 调度策略：通过配置亲和与反亲和规则，可实现灵活的工作负载调度，支持节点亲和与Pod亲和/反亲和。详情请参见**调度策略（亲和与反亲和）**。
- 容忍策略：容忍策略与节点的污点能力配合使用，允许（不强制）负载调度到带有与之匹配的污点的节点上，也可用于控制负载所在的节点被标记污点后负载的驱逐策略，详情请参见**容忍策略**。
- 标签与注解：以键值对形式为工作负载Pod添加标签或注解，填写完成后需单击“确认添加”。关于标签与注解的作用及配置说明，请参见**标签与注解**。
- DNS配置：为工作负载单独配置DNS策略，详情请参见**工作负载DNS配置说明**。
- 网络配置：
  - Pod入/出口带宽限速：支持为Pod设置入/出口带宽限速，详情请参见**Pod互访QoS限速**。

步骤4 单击右下角“创建工作负载”。

----结束

通过 kubectl 命令行创建

本节以nginx工作负载为例，说明kubectl命令创建工作负载的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建一个名为nginx-daemonset.yaml的描述文件。其中，nginx-daemonset.yaml为自定义名称，您可以随意命名。

vi nginx-daemonset.yaml

描述文件内容如下。此处仅为示例，daemonset的详细说明请参见**kubernetes官方文档**。

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
  labels:
    app: nginx-daemonset
spec:
  selector:
    matchLabels:
      app: nginx-daemonset
  template:
    metadata:
      labels:
        app: nginx-daemonset
    spec:
      nodeSelector:                # 节点选择，当节点拥有daemon=need时才在节点上创建Pod
        daemon: need
      containers:
      - name: nginx-daemonset
        image: nginx:alpine
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
      imagePullSecrets:
      - name: default-secret
```

这里可以看出没有Deployment或StatefulSet中的replicas参数，因为是每个节点固定一个。

Pod模板中有个nodeSelector，指定了只在有“daemon=need”的节点上才创建Pod，DaemonSet只在指定标签的节点上创建Pod。如果需要在每一个节点上创建Pod可以删除该标签。

步骤3 创建daemonset。

kubectl create -f nginx-daemonset.yaml

回显如下表示已开始创建daemonset。

```
daemonset.apps/nginx-daemonset created
```

步骤4 查看daemonset状态。

kubectl get ds

```
$ kubectl get ds
NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   1         1         0       1            0           daemon=need     116s
```

步骤5 若工作负载需要被访问（集群内访问或节点访问），您需要设置访问方式，具体请参见网络创建对应服务。

----结束

9.2.4 创建普通任务（Job）

操作场景

普通任务是一次性运行的短任务，部署完成后即可执行。正常退出（exit 0）后，任务即执行完成。

普通任务是用来控制批处理型任务的资源对象。批处理业务与长期伺服业务（Deployment、Statefulset）的主要区别是：

批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同，即：

- 单Pod型任务有一个Pod成功就标志完成。
- 定数成功型任务保证有N个任务全部成功。
- 工作队列型任务根据应用确认的全局成功而标志成功。

前提条件

已创建资源，具体操作请参见**创建节点**。若已有集群和节点资源，无需重复操作。

通过控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤3 配置工作负载的信息。

基本信息

- 负载类型：选择任务Job。工作负载类型的介绍请参见**工作负载概述**。
- 负载名称：填写工作负载的名称。请输入1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。
- 命名空间：选择工作负载的命名空间，默认为default。您可以单击后面的“创建命名空间”，命名空间的详细介绍请参见**创建命名空间**。
- 实例数量：填写实例的数量，即工作负载Pod的数量。

容器配置

- 容器信息

  Pod中可以配置多个容器，您可以单击右侧“添加容器”为Pod配置多个容器。

  - 基本信息：配置容器的基本信息。

参数	说明
容器名称	为容器命名。

参数	说明
更新策略	镜像更新/拉取策略。可以勾选“总是拉取镜像”，表示每次都从镜像仓库拉取镜像；如不勾选则优使用节点已有的镜像，如果没有这个镜像再从镜像仓库拉取。
镜像名称	单击后方“选择镜像”，选择容器使用的镜像。 如果需要使用第三方镜像，请参见**使用第三方镜像**。
镜像版本	选择需要部署的镜像版本。
CPU配额	■ 申请：容器需要使用的最小CPU值，默认0.25Core。 ■ 限制：允许容器使用的CPU最大值。建议设容器配额的最高限额，避免容器资源超额导致系统故障。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
内存配额	■ 申请：容器需要使用的内存最小值，默认512MiB。 ■ 限制：允许容器使用的内存最大值。如果超过，容器会被终止。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
GPU配额（可选）	当集群中包含GPU节点时，才能设置GPU配额，且集群中需安装**gpu-device-plugin**插件。 ■ 不限制：表示不使用GPU。 ■ 独享：单个容器独享GPU。 ■ 共享：容器需要使用的GPU百分比，例如设置为10%，表示该容器需使用GPU资源的10%。 关于如何在集群中使用GPU，请参见**使用Kubernetes默认GPU调度**。
特权容器（可选）	特权容器是指容器里面的程序具有一定的特权。 若选中，容器将获得超级权限，例如可以操作宿主机上面的网络设备、修改内核参数等。
初始化容器（可选）	选择容器是否作为初始化（Init）容器。初始化（Init）容器不支持设置健康检查。 Init容器是一种特殊容器，可以在Pod中的其他应用容器启动之前运行。每个Pod中可以包含多个容器，同时Pod中也可以有一个或多个先于应用容器启动的Init容器，当所有的Init 容器运行完成时，Pod中的应用容器才会启动并运行。详细说明请参见**Init 容器**。

- 生命周期（可选）：在容器的生命周期的特定阶段配置需要执行的操作，例如启动命令、启动后处理和停止前处理，详情请参见**设置容器生命周期**。

- 环境变量（可选）：支持通过键值对的形式为容器运行环境设置变量，可用于把外部信息传递给Pod中运行的容器，可以在应用部署后灵活修改，详情请参见**设置环境变量**。
- 数据存储（可选）：在容器内挂载本地存储或云存储，不同类型的存储使用场景及挂载方式不同，详情请参见**存储**。

  说明

  负载实例数大于1时，不支持挂载云硬盘类型的存储。
- 容器日志（可选）：容器标准输出日志将默认上报至 AOM 服务，无需独立配置。您可以手动配置日志采集路径，详情请参见**使用ICAgent采集容器日志**。

  如需要关闭当前负载的标准输出，您可在**标签与注解**中添加键为kubernetes.AOM.log.stdout，值为[]的注解，即可关闭当前负载下全部容器的标准输出。该注解的使用方法请参见**表9-16**。

- 镜像访问凭证：用于访问镜像仓库的凭证，默认取值为default-secret，使用default-secret可访问SWR镜像仓库的镜像。default-secret详细说明请参见**default-secret**。
- GPU显卡（可选）：默认为不限制。当集群中存在GPU节点时，工作负载实例可以调度到指定GPU显卡类型的节点上。

高级配置（可选）

- 标签与注解：以键值对形式为工作负载Pod添加标签或注解，填写完成后需单击“确认添加”。关于标签与注解的作用及配置说明，请参见**标签与注解**。
- 任务设置：
  - 并行数：任务负载执行过程中允许同时创建的最大实例数，并行数应不大于实例数。
  - 超时时间（秒）：当任务执行超出该时间时，任务将会被标识为执行失败，任务下的所有实例都会被删除。为空时表示不设置超时时间。
- 网络配置：
  - Pod入/出口带宽限速：支持为Pod设置入/出口带宽限速，详情请参见**Pod互访QoS限速**。

步骤4 单击右下角“创建工作负载”。

----结束

使用 kubectl 创建 Job

Job的配置参数如下所示。

- spec.template格式与Pod相同。
- RestartPolicy仅支持Never或OnFailure。
- 单个Pod时，默认Pod成功运行后Job即结束。
- .spec.completions表示Job结束需要成功运行的Pod个数，默认为1。
- .spec.parallelism表示并行运行的Pod的个数，默认为1。
- spec.backoffLimit表示失败Pod的重试最大次数，超过这个次数不会继续重试。
- .spec.activeDeadlineSeconds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activeDeadlineSeconds优先级高于backoffLimit，即到达activeDeadlineSeconds的Job会忽略backoffLimit的设置。

根据.spec.completions和.spec.Parallelism的设置，可以将Job划分为以下几种类型。

表 9-3 任务类型

Job类型	说明	使用示例
一次性Job	创建一个Pod直至其成功结束	数据库迁移
固定结束次数的Job	依次创建一个Pod运行直至completions个成功结束	处理工作队列的Pod
固定结束次数的并行Job	依次创建多个Pod运行直至completions个成功结束	多个Pod同时处理工作队列
并行Job	创建一个或多个Pod直至有一个成功结束	多个Pod同时处理工作队列

以下是一个Job配置示例，保存在myjob.yaml中，其计算π到2000位并打印输出。

```
apiVersion: batch/v1
kind: Job
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
      imagePullSecrets:
        - name: default-secret
```

说明：

- apiVersion: batch/v1 是当前job的Version
- kind: Job：指定当前资源的类型时Job
- restartPolicy: Never：是指当前的重启策略。对于Job，只能设置为Never或者OnFailure。对于其他controller（比如Deployment）可以设置为Always。

运行该任务，如下：

步骤1 启动这个job。

```
[root@k8s-master k8s]# kubectl apply -f myjob.yaml
job.batch/myjob created
```

步骤2 查看这个job。

kubectl get job

```
[root@k8s-master k8s]# kubectl get job
NAME    COMPLETIONS   DURATION   AGE
myjob   50/50         23s        3m45s
```

completions为 50/50 表示成功运行了这个job。

步骤3 查看pod的状态。

kubectl get pod

```
[root@k8s-master k8s]# kubectl get pod
NAME          READY   STATUS      RESTARTS   AGE
myjob-29qlw   0/1     Completed   0          4m5s
...
```

状态为Completed表示这个job已经运行完成。

步骤4 查看这个pod的日志。

kubectl logs

```
# kubectl logs myjob-29qlw
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019893809525720106548586327886593615338182796823030195203530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588907509838175463746493931925506040092770167113900984882401285836160356370766010471018194295559619894676783744944825537977472684710404753464620804668425906949129331367702898915210475216205696602405803815019351125338243003558764024749647326391419927260426992279678235478163600934172164121992458631503028618297455570674983850549458858692699569092721079750930295532116534498720275596023648066549911988183479775356636980742654252786255181841757467289097777279380008164706001614524919217321721477235014144197356854816136115735255213347574184946843852332390739414333454776241686251898356948556209921922218427255025425688767179049460165346680498862723279178608578438382796797668145410095388378636095068006422512520511739298489608412848862694560424196528502221066118630674427862203919494504712371378696095636437191728746776465757396241389086583264599581339047802759015
```

----结束

相关操作

普通任务创建完成后，您还可执行表9-4中操作。

表 9-4 其他操作

操作	操作说明
编辑YAML	单击任务名称后的“更多 > 编辑YAML”，可编辑当前任务对应的YAML文件。
删除普通任务	1. 选择待删除的任务，单击操作列的“更多 > 删除”。 2. 单击“是”。 任务删除后将无法恢复，请谨慎操作。

9.2.5 创建定时任务（CronJob）

操作场景

定时任务是按照指定时间周期运行的短任务。使用场景为在某个固定时间点，为所有运行中的节点做时间同步。

定时任务是基于时间的Job，就类似于Linux系统的crontab，在指定的时间周期运行指定的Job，即：

- 在给定时间点只运行一次。
- 在给定时间点周期性地运行。

CronJob的典型用法如下所示：

- 在给定的时间点调度Job运行。
- 创建周期性运行的Job，例如数据库备份、发送邮件。

前提条件

已创建资源，具体操作请参见**创建节点**。

通过控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤3 配置工作负载的信息。

基本信息

- 负载类型：选择定时任务CronJob。工作负载类型的介绍请参见**工作负载概述**。
- 负载名称：填写工作负载的名称。请输入1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。
- 命名空间：选择工作负载的命名空间，默认为default。您可以单击后面的“创建命名空间”，命名空间的详细介绍请参见**创建命名空间**。

容器配置

- 容器信息

  Pod中可以配置多个容器，您可以单击右侧“添加容器”为Pod配置多个容器。

  - 基本信息：配置容器的基本信息。

参数	说明
容器名称	为容器命名。
更新策略	镜像更新/拉取策略。可以勾选“总是拉取镜像”，表示每次都从镜像仓库拉取镜像；如不勾选则优使用节点已有的镜像，如果没有这个镜像再从镜像仓库拉取。
镜像名称	单击后方“选择镜像”，选择容器使用的镜像。 如果需要使用第三方镜像，请参见**使用第三方镜像**。

参数	说明
镜像版本	选择需要部署的镜像版本。
CPU配额	■ 申请：容器需要使用的最小CPU值，默认0.25Core。 ■ 限制：允许容器使用的CPU最大值。建议设容器配额的最高限额，避免容器资源超额导致系统故障。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
内存配额	■ 申请：容器需要使用的内存最小值，默认512MiB。 ■ 限制：允许容器使用的内存最大值。如果超过，容器会被终止。 如不填写申请值和限制值，表示不限制配额。申请值和限制值的配置说明及建议请参见**设置容器规格**。
GPU配额（可选）	当集群中包含GPU节点时，才能设置GPU配额，且集群中需安装**gpu-device-plugin**插件。 ■ 不限制：表示不使用GPU。 ■ 独享：单个容器独享GPU。 ■ 共享：容器需要使用的GPU百分比，例如设置为10%，表示该容器需使用GPU资源的10%。 关于如何在集群中使用GPU，请参见**使用Kubernetes默认GPU调度**。
特权容器（可选）	特权容器是指容器里面的程序具有一定的特权。 若选中，容器将获得超级权限，例如可以操作宿主机上面的网络设备、修改内核参数等。
初始化容器（可选）	选择容器是否作为初始化（Init）容器。初始化（Init）容器不支持设置健康检查。 Init容器是一种特殊容器，可以在Pod中的其他应用容器启动之前运行。每个Pod中可以包含多个容器，同时Pod中也可以有一个或多个先于应用容器启动的Init容器，当所有的Init 容器运行完成时，Pod中的应用容器才会启动并运行。详细说明请参见**Init 容器**。

  - 生命周期（可选）：在容器的生命周期的特定阶段配置需要执行的操作，例如启动命令、启动后处理和停止前处理，详情请参见**设置容器生命周期**。
  - 环境变量（可选）：支持通过键值对的形式为容器运行环境设置变量，可用于把外部信息传递给Pod中运行的容器，可以在应用部署后灵活修改，详情请参见**设置环境变量**。

- 镜像访问凭证：用于访问镜像仓库的凭证，默认取值为default-secret，使用default-secret可访问SWR镜像仓库的镜像。default-secret详细说明请参见**default-secret**。

- GPU显卡（可选）：默认为不限制。当集群中存在GPU节点时，工作负载实例可以调度到指定GPU显卡类型的节点上。

定时规则

- 并发策略：支持如下三种模式。
  - Forbid：在前一个任务未完成时，不创建新任务。
  - Allow：定时任务不断新建Job，会抢占集群资源。
  - Replace：已到新任务创建时间点，但前一个任务还未完成，新的任务会取代前一个任务。
- 定时规则：指定新建定时任务在何时执行，YAML中的定时规则通过CRON表达式实现。
  - 以固定周期执行定时任务，支持的周期单位为分钟、小时、日、月。例如，每30分钟执行一次任务，对应的CRON表达式为“*/30 * * * *”，执行时间将从单位范围内的0值开始计算，如**00:00:00**、**00:30:00**、**01:00:00**、**...**。
  - 以固定时间（按月）执行定时任务。例如，在每个月1日的0时0分执行任务，对应的CRON表达式为“0 0 1 */1 *”，执行时间为******-01-01 00:00:00**、******-02-01 00:00:00**、**...**。
  - 以固定时间（按周）执行定时任务。例如，在每周一的0时0分执行任务，对应的CRON表达式为“0 0 * * 1”，执行时间为******-**-01 周一 00:00:00**、******-**-08 周一 00:00:00**、**...**。
  - 自定义CRON表达式：关于CRON表达式的用法，可参考**CRON**。

    说明

    - 以固定时间（按月）执行定时任务时，在某月的天数不存在的情况下，任务将不会在该月执行。例如设置天数为30，而2月份没有30号，任务将跳过该月份，在3月30号继续执行。
    - 由于CRON表达式的定义，这里的固定周期并非严格意义的周期。将从0开始按周期对其时间单位范围（例如单位为分钟时，则范围为0~59）进行划分，无法整除时最后一个周期会被重置。因此仅在周期能够平均划分其时间单位范围时，才能表示准确的周期。

      举个例子，周期单位为小时，因为“/2、/3、/4、/6、/8和/12”可将24小时整除，所以可以表示准确的周期；而使用其他周期时，在新的一天开始时，最后一个周期将会被重置。比如CRON式为“* */12 * * *”时为准确的周期，每天的执行时间为**00:00:00**和**12:00:00**；而CRON式为“* */13 * * *”时，每天的执行时间为**00:00:00**和**13:00:00**，在第二天0时，虽然没到13个小时的周期还是会被刷新。
- 任务记录：可以设置保留执行成功或执行失败的任务个数，设置为0表示不保留。

高级配置（可选）

- 标签与注解：以键值对形式为工作负载Pod添加标签或注解，填写完成后需单击“确认添加”。关于标签与注解的作用及配置说明，请参见**标签与注解**。
- 网络配置：
  - Pod入/出口带宽限速：支持为Pod设置入/出口带宽限速，详情请参见**Pod互访QoS限速**。

步骤4 单击右下角“创建工作负载”。

----结束

使用 kubectl 创建 CronJob

CronJob的配置参数如下所示：

- .spec.schedule指定任务运行时间与周期，参数格式请参见**Cron**，例如“0 * * * * ”或“@hourly”。
- .spec.jobTemplate指定需要运行的任务，格式与**使用kubectl创建Job**相同。
- .spec.startingDeadlineSeconds指定任务开始的截止期限。
- .spec.concurrencyPolicy指定任务的并发策略，支持Allow、Forbid和Replace三个选项。
  - Allow（默认）：允许并发运行Job。
  - Forbid：禁止并发运行，如果前一个还没有完成，则直接跳过下一个。
  - Replace：取消当前正在运行的Job，用一个新的来替换。

下面是一个CronJob的示例，保存在cronjob.yaml文件中。

说明

在v1.21及以上集群中，CronJob的apiVersion为**batch/v1**。

在v1.21以下集群中，CronJob的apiVersion为**batch/v1beta1**。

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
          imagePullSecrets:
            - name: default-secret
```

运行该任务，如下：

步骤1 创建CronJob。

kubectl create -f cronjob.yaml

命令行终端显示如下信息：

```
cronjob.batch/hello created
```

步骤2 执行如下命令，查看执行情况。

kubectl get cronjob

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        <none>          9s
```

kubectl get jobs

```
NAME              COMPLETIONS   DURATION   AGE
hello-1597387980  1/1           27s        45s
```

kubectl get pod

```
NAME                     READY   STATUS      RESTARTS   AGE
hello-1597387980-tjv8f   0/1     Completed   0          114s
hello-1597388040-lckg9   0/1     Completed   0          39s
```

kubectl logs hello-1597387980-tjv8f

```
Fri Aug 14 06:56:31 UTC 2020
Hello from the Kubernetes cluster
```

kubectl delete cronjob hello

```
cronjob.batch "hello" deleted
```

须知

删除CronJob时，对应的普通任务及相关的Pod都会被删除。

----结束

相关操作

定时任务创建完成后，您还可执行表9-5中操作。

表 9-5 其他操作

操作	操作说明
编辑YAML	单击定时任务名称后的“更多 > 编辑YAML”，可修改当前任务对应的YAML文件。
停止定时任务	1. 选择待停止的任务，单击操作列的“停止”。 2. 单击“是”。
删除定时任务	1. 选择待删除的任务，单击操作列的“更多 > 删除”。 2. 单击“是”。 任务删除后将无法恢复，请谨慎操作。

9.3 容器设置

9.3.1 时区同步

创建工作负载时，支持设置容器使用节点相同的时区。您可以在创建工作负载时打开时区同步配置。

时区同步功能依赖容器中挂载的本地磁盘（HostPath），如下所示，开启时区同步后，Pod中会通过HostPath方式，将节点的“/etc/localtime”挂载到容器的“/etc/localtime”，从而使得节点和容器使用相同的时区配置文件。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: test
  namespace: default
```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      volumes:
        - name: vol-162979628557461404
          hostPath:
            path: /etc/localtime
            type: ''
      containers:
        - name: container-0
          image: 'nginx:alpine'
          volumeMounts:
            - name: vol-162979628557461404
              readOnly: true
              mountPath: /etc/localtime
          imagePullPolicy: IfNotPresent
      imagePullSecrets:
        - name: default-secret
```

9.3.2 配置镜像拉取策略

创建工作负载会从镜像仓库拉取容器镜像到节点上，当前Pod重启、升级时也会拉取镜像。

默认情况下容器镜像拉取策略imagePullPolicy是**IfNotPresent**，表示如果节点上有这个镜像就直接使用节点已有镜像，如果没有这个镜像就会从镜像仓库拉取。

容器镜像拉取策略还可以设置为**Always**，表示无论节点上是否有这个镜像，都会从镜像仓库拉取，并覆盖节点上的镜像。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    imagePullPolicy: Always
  imagePullSecrets:
  - name: default-secret
```

在CCE控制台也可以设置镜像拉取策略，在创建工作负载时设置。勾选表示总是拉取镜像（Always），不勾选就是IfNotPresent。

须知

建议您在制作镜像时，每次制作一个新的镜像都使用一个新的Tag，如果不更新Tag只更新镜像，当拉取策略选择为IfNotPresent时，CCE会认为当前节点已经存在这个Tag的镜像，不会重新拉取。

9.3.3 使用第三方镜像

操作场景

CCE支持拉取第三方镜像仓库的镜像来创建工作负载。

通常第三方镜像仓库必须经过认证（账号密码）才能访问，而CCE中容器拉取镜像是使用密钥认证方式，这就要求在拉取镜像前先创建镜像仓库的密钥。

前提条件

使用第三方镜像时，请确保工作负载运行的节点可访问公网。

通过界面操作

步骤1 创建第三方镜像仓库的密钥。

单击集群名称进入集群，在左侧导航栏选择“配置项与密钥”，在右侧选择“密钥”页签，单击右上角“创建密钥”，密钥类型必须选择为kubernetes.io/dockerconfigjson。详细操作请参见**创建密钥**。

此处的“用户名”和“密码”请填写第三方镜像仓库的账号密码。

步骤2 创建工作负载时，可以在“镜像名称”中直接填写私有镜像地址，填写的格式为domainname/namespace/imagename:tag，并在“镜像访问凭证”中选择**步骤1**中创建的密钥。

步骤3 填写其他参数后，单击“创建工作负载”。

----结束

使用 kubectl 创建第三方镜像仓库的密钥

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 通过kubectl创建认证密钥，该密钥类型为kubernetes.io/dockerconfigjson类型。

```
kubectl create secret docker-registry myregistrykey -n default --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

其中，*myregistrykey*为密钥名称，*default*为密钥所在的命名空间，其余参数如下所示。

- DOCKER_REGISTRY_SERVER：第三方镜像仓库的地址，如“www.3rdregistry.com”或“10.10.10.10:443”。
- DOCKER_USER：第三方镜像仓库的账号。
- DOCKER_PASSWORD：第三方镜像仓库的密码。
- DOCKER_EMAIL：第三方镜像仓库的邮箱。

步骤3 创建工作负载时使用第三方镜像，具体步骤请参见如下。

kubernetes.io/dockerconfigjson类型的密钥作为私有镜像获取的认证方式，以Pod为例，创建的myregistrykey作为镜像的认证方式。

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: foo
  namespace: default
spec:
  containers:
    - name: foo
      image: www.3rdregistry.com/janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey               #使用上面创建的密钥
```

----结束

9.3.4 设置容器规格

操作场景

CCE支持在创建工作负载时为添加的容器设置资源的需求量和限制，最常见的可设定资源是 CPU 和内存（ RAM ）大小。此外Kubernetes还支持其他类型的资源，可通过YAML设置。

申请与限制

在CPU配额和内存配额设置中，**申请**与**限制**的含义如下：

- 申请（ Request ）：根据申请值调度该实例到满足条件的节点去部署工作负载。
- 限制（ Limit ）：根据限制值限制工作负载使用的资源。

如果实例运行所在的节点具有足够的可用资源，实例可以使用超出申请的资源量，但不能超过限制的资源量。

例如，如果您将实例的内存申请值为1GiB、限制值为2GiB，而该实例被调度到一个具有8GiB CPU的节点上，且该节点上没有其他实例运行，那么该实例在负载压力较大的情况下可使用超过1GiB的内存，但内存使用量不得超过2GiB。若容器中的进程尝试使用超过2GiB的资源时，系统内核将会尝试将进程终止，出现内存不足（ OOM ）错误。

说明

创建工作负载时，建议设置CPU和内存的资源上下限。同一个节点上部署的工作负载，对于未设置资源上下限的工作负载，如果其异常资源泄露会导致其它工作负载分配不到资源而异常。未设置资源上下限的工作负载，工作负载监控信息也会不准确。

配置说明

在实际生产业务中，建议申请和限制比例为1:1.5左右，对于一些敏感业务建议设置成1:1。如果申请值过小而限制值过大，容易导致节点超分严重。如果遇到业务高峰或流量高峰，容易把节点内存或者CPU耗尽，导致节点不可用的情况发生。

- CPU配额：CPU资源单位为核，可以通过数量或带单位后缀（ m ）的整数表达，例如数量表达式0.1核等价于表达式100m，但Kubernetes不允许设置精度小于1m的CPU资源。

表 9-6 CPU 配额说明

参数	说明
CPU申请	容器使用的最小CPU需求，作为容器调度时资源分配的判断依赖。只有当节点上可分配CPU总量 ≥ 容器CPU申请数时，才允许将容器调度到该节点。

参数	说明
CPU限制	容器能使用的CPU最大值。

建议配置方法：

节点的实际可用分配CPU量 ≥ 当前实例所有容器CPU限制值之和 ≥ 当前实例所有容器CPU申请值之和，节点的实际可用分配CPU量请在“资源管理 > 节点管理”中对应节点的“可分配资源”列下查看“CPU: ** Core”。

- 内存配额：内存资源默认单位为字节，或者也可以使用带单位后缀的整数来表达，例如100Mi。但需要注意单位大小写。

表 9-7 内存配额说明

参数	说明
内存申请	容器使用的最小内存需求，作为容器调度时资源分配的判断依赖。只有当节点上可分配内存总量 ≥ 容器内存申请数时，才允许将容器调度到该节点。
内存限制	容器能使用的内存最大值。当内存使用率超出设置的内存限制值时，该实例可能会被重启进而影响工作负载的正常使用。

建议配置方法：

节点的实际可用分配内存量 ≥ 当前节点所有容器内存限制值之和 ≥ 当前节点所有容器内存申请值之和，节点的实际可用分配内存量请在“资源管理 > 节点管理”中对应节点的“可分配资源”列下查看“内存: ** GiB”。

说明

可分配资源：可分配量按照实例申请值(Request)计算，表示实例在该节点上可请求的资源上限，不代表节点实际可用资源（请参见**CPU和内存配额使用示例**）。 计算公式为：

- 可分配CPU = CPU总量 - 所有实例的CPU申请值 - 其他资源CPU预留值
- 可分配内存 = 内存总量 - 所有实例的内存申请值 - 其他资源内存预留值

CPU 和内存配额使用示例

假设集群中可调度的节点资源总量为4Core 8GiB，且已经在集群中部署了两个实例，其中实例1存在CPU和内存资源超分（即限制值>申请值），而实例2不存在资源超分。两个实例的规格设置如下：

实例	CPU申请	CPU限制	内存申请	内存限制
实例1	1Core	2Core	1GiB	4GiB
实例2	2Core	2Core	2GiB	2GiB

那么节点上CPU和内存的资源使用情况如下：

- CPU可分配资源=4Core-（实例1申请的1Core+实例2申请的2Core）=1Core
- 内存可分配资源=8GiB-（实例1申请的1GB+实例2申请的2GiB）=5GiB

此时节点还剩余1Core 5GiB的资源可供下一个新增的实例调度。

如果实例1处于业务高峰、负载压力较大时，会尝试在限制值范围内使用更多的CPU和内存，因此实际可用的资源将会小于1Core 5GiB。

9.3.5 设置容器生命周期

操作场景

CCE提供了回调函数，在容器的生命周期的特定阶段执行调用，比如容器在停止前希望执行某项操作，就可以注册相应的钩子函数。

目前提供的生命周期回调函数如下所示：

- **启动命令：**容器将会以该启动命令启动，请参见**启动命令**。
- **启动后处理：**容器启动后触发，请参见**启动后处理**。
- **停止前处理：**容器停止前触发。设置停止前处理，确保升级或实例删除时可提前将实例中运行的业务排水。详细请参见**停止前处理**。

启动命令

在默认情况下，镜像启动时会运行默认命令，如果想运行特定命令或重写镜像默认值，需要进行相应设置。

Docker的镜像拥有存储镜像信息的相关元数据，如果不设置生命周期命令和参数，容器运行时将运行镜像制作时提供的默认的命令和参数，Docker将这两个字段定义为ENTRYPOINT和 CMD。

如果在创建工作负载时填写了容器的运行命令和参数，将会覆盖镜像构建时的默认命令ENTRYPOINT、CMD，规则如下：

表 9-8 容器如何执行命令和参数

镜像 ENTRYPOINT	镜像CMD	容器运行命令	容器运行参数	最终执行
[touch]	[/root/test]	未设置	未设置	[touch /root/test]
[touch]	[/root/test]	[mkdir]	未设置	[mkdir]
[touch]	[/root/test]	未设置	[/opt/test]	[touch /opt/test]
[touch]	[/root/test]	[mkdir]	[/opt/test]	[mkdir /opt/test]

步骤1 登录CCE控制台，在创建工作负载时，配置容器信息，选择“生命周期”。

步骤2 在“启动命令”页签，输入运行命令和运行参数。

表 9-9 容器启动命令

命令方式	操作步骤
运行命令	输入可执行的命令，例如“/run/server”。 若运行命令有多个，需分行书写。 **说明** 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。
运行参数	输入控制容器运行命令参数，例如--port=8080。 若参数有多个，多个参数以换行分隔。

----结束

启动后处理

步骤1 登录CCE控制台，在创建工作负载时，配置容器信息，选择“生命周期”。

步骤2 在“启动后处理”页签，设置启动后处理的参数。

表 9-10 启动后处理-参数说明

参数	说明

命令行方式	在容器中执行指定的命令，配置为需要执行的命令。命令的格式为Command Args[1] Args[2]…（Command为系统命令或者用户自定义可执行程序，如果未指定路径则在默认路径下寻找可执行程序），如果需要执行多条命令，建议采用将命令写入脚本执行的方式。**不支持后台执行和异步执行的命令。**
	如需要执行的命令如下：
	`exec:`
	`  command:`
	`  - /install.sh`
	`  - install_agent`
	请在执行脚本中填写: /install install_agent。这条命令表示容器创建成功后将执行install.sh。

HTTP请求方式	发起一个HTTP调用请求。配置参数如下： ● 路径：请求的URL路径，可选项。 ● 端口：请求的端口，必选项。 ● 主机地址：请求的IP地址，可选项，默认为实例IP。

----结束

停止前处理

步骤1 登录CCE控制台，在创建工作负载时，配置容器信息，选择“生命周期”。

步骤2 在“停止前处理”页签，设置停止前处理的命令。

表 9-11 停止前处理

参数	说明
命令行方式	在容器中执行指定的命令，配置为需要执行的命令。命令的格式为Command Args[1] Args[2]…（ Command为系统命令或者用户自定义可执行程序，如果未指定路径则在默认路径下寻找可执行程序），如果需要执行多条命令，建议采用将命令写入脚本执行的方式。 如需要执行的命令如下： exec: command: - /uninstall.sh - uninstall_agent 请在执行脚本中填写: /uninstall uninstall_agent。这条命令表示容器结束前将执行uninstall.sh。
HTTP请求方式	发起一个HTTP调用请求。配置参数如下： ● 路径：请求的URL路径，可选项。 ● 端口：请求的端口，必选项。 ● 主机地址：请求的IP地址，可选项，默认为实例IP。

----结束

YAML 样例

本节以nginx为例，说明kubectl命令设置容器生命周期的方法。

在以下配置文件中，您可以看到postStart命令在容器目录/bin/bash下写了个**install.sh**命令。 preStop执行**uninstall.sh**命令。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        command:
        - sleep 3600                    #启动命令
        imagePullPolicy: Always
        lifecycle:
          postStart:
            exec:
              command:
              - /bin/bash
              - install.sh              #启动后命令
          preStop:
            exec:
              command:
              - /bin/bash
```

```
      - uninstall.sh          #停止前命令
  name: nginx
imagePullSecrets:
- name: default-secret
```

9.3.6 设置容器健康检查

操作场景

健康检查是指容器运行过程中，根据用户需要，定时检查容器健康状况。若不配置健康检查，如果容器内应用程序异常，Pod将无法感知，也不会自动重启去恢复。最终导致虽然Pod状态显示正常，但Pod中的应用程序异常的情况。

Kubernetes提供了三种健康检查的探针：

- **存活探针：**livenessProbe，用于检测容器是否正常，类似于执行ps命令检查进程是否存在。如果容器的存活检查失败，集群会对该容器执行重启操作；若容器的存活检查成功则不执行任何操作。
- **就绪探针：**readinessProbe，用于检查用户业务是否就绪，如果未就绪，则不转发流量到当前实例。一些程序的启动时间可能很长，比如要加载磁盘数据或者要依赖外部的某个模块启动完成才能提供服务。这时候程序进程在，但是并不能对外提供服务。这种场景下该检查方式就非常有用。如果容器的就绪检查失败，集群会屏蔽请求访问该容器；若检查成功，则会开放对该容器的访问。
- **启动探针**：startupProbe，用于探测应用程序容器什么时候启动了。 如果配置了这类探测器，就可以控制容器在启动成功后再进行存活性和就绪检查， 确保这些存活、就绪探针不会影响应用程序的启动。 这可以用于对启动慢的容器进行存活性检测，避免它们在启动运行之前就被终止。

检查方式

- **HTTP 请求检查**

  HTTP 请求方式针对的是提供HTTP/HTTPS服务的容器，集群周期性地对该容器发起HTTP/HTTPS GET请求，如果HTTP/HTTPS response返回码属于200~399范围，则证明探测成功，否则探测失败。使用HTTP请求探测必须指定容器监听的端口和HTTP/HTTPS的请求路径。

  例如：提供HTTP服务的容器，HTTP检查路径为：/health-check；端口为：80；主机地址可不填，默认为容器实例IP，此处以172.16.0.186为例。那么集群会周期性地对容器发起如下请求：GET http://172.16.0.186:80/health-check。您也可以为HTTP请求添加一个或多个请求头部，例如设置请求头名称为Custom-Header，对应的值为example。

- **TCP 端口检查**

  对于提供TCP通信服务的容器，集群周期性地对该容器建立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。

  例如：有一个nginx容器，它的服务端口是80，对该容器配置了TCP端口探测，指定探测端口为80，那么集群会周期性地对该容器的80端口发起TCP连接，如果连接成功则证明检查成功，否则检查失败。

- **执行命令检查**

  命令检查是一种强大的检查方式，该方式要求用户指定一个容器内的可执行命令，集群会周期性地在容器内执行该命令，如果命令的返回结果是0则检查成功，否则检查失败。

对于上面提到的TCP端口检查和HTTP请求检查，都可以通过执行命令检查的方式来替代：

- 对于TCP端口探测，可以使用程序对容器的端口尝试connect，如果connect成功，脚本返回0，否则返回-1。
- 对于HTTP请求探测，可以使用脚本命令来对容器尝试使用wget命令进行探测。

  wget http://127.0.0.1:80/health-check

  并检查response 的返回码，如果返回码在200~399 的范围，脚本返回0，否则返回-1。如下图：

须知

- 必须把要执行的程序放在容器的镜像里面，否则会因找不到程序而执行失败。
- 如果执行的命令是一个shell脚本，由于集群在执行容器里的程序时，不在终端环境下，因此不能直接指定脚本为执行命令，需要加上脚本解析器。比如脚本是**/data/scripts/health_check.sh**，那么使用执行命令检查时，指定的程序应该是**sh /data/scripts/health_check.sh**。究其原因是集群在执行容器里的程序时，不在终端环境下。

公共参数说明

表 9-12 公共参数说明

参数	参数说明
检测周期（periodSeconds）	探针检测周期，单位为秒。 例如，设置为30，表示每30秒检测一次。
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，将状态转变为成功所需要的最小连续成功次数。例如，设置为1时，表明健康检查失败后，健康检查需要连续成功1次，才认为工作负载状态正常。 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。

参数	参数说明
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。

YAML 示例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      exec:
        command:
          - cat
          - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
    startupProbe:
      httpGet:
        path: /healthz
        port: 80
      failureThreshold: 30
      periodSeconds: 10
```

9.3.7 设置环境变量

操作场景

环境变量是指容器运行环境中设定的一个变量，环境变量可以在工作负载部署后修改，为工作负载提供极大的灵活性。

CCE中设置的环境变量与Dockerfile中的“ENV”效果相同。

须知

容器启动后，容器中的内容不应修改。如果修改配置项（例如将容器应用的密码、证书、环境变量配置到容器中），当容器重启（例如节点异常重新调度Pod）后，会导致配置丢失，业务异常。

配置信息应通过入参等方式导入容器中，以免重启后配置丢失。

环境变量支持如下几种方式设置。

- **自定义**：手动填写环境变量名称及对应的参数值。
- **配置项导入**：将配置项中所有键值都导入为环境变量。
- **配置项键值导入**：将配置项中某个键的值导入作为某个环境变量的值。
- **密钥导入**：将密钥中所有键值都导入为环境变量。
- **密钥键值导入**：将密钥中某个键的值导入作为某个环境变量的值。
- **变量/变量引用**：用Pod定义的字段作为环境变量的值。
- **资源引用**：用容器定义的资源申请值或限制值作为环境变量的值。

添加环境变量

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤3 在创建工作负载时，在“容器配置”中选择“环境变量”页签。

步骤4 设置环境变量。

----结束

YAML 样例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: env-example
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: env-example
  template:
    metadata:
      labels:
        app: env-example
    spec:
      containers:
        - name: container-1
          image: nginx:alpine
          imagePullPolicy: Always
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
            limits:
              cpu: 250m
              memory: 512Mi
          env:
```

```
        - name: key                    # 自定义
          value: value
        - name: key1                   # 配置项键值导入
          valueFrom:
            configMapKeyRef:
              name: configmap-example
              key: key1
        - name: key2                   # 密钥键值导入
          valueFrom:
            secretKeyRef:
              name: secret-example
              key: key2
        - name: key3                   # 变量引用，用Pod定义的字段作为环境变量的值
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
        - name: key4                   # 资源引用，用Container定义的字段作为环境变量的值
          valueFrom:
            resourceFieldRef:
              containerName: container1
              resource: limits.cpu
              divisor: 1
      envFrom:
        - configMapRef:                # 配置项导入
            name: configmap-example
        - secretRef:                   # 密钥导入
            name: secret-example
    imagePullSecrets:
      - name: default-secret
```

环境变量查看

如果configmap-example和secret-example的内容如下。

```
$ kubectl get configmap configmap-example -oyaml
apiVersion: v1
data:
  configmap_key: configmap_value
kind: ConfigMap
...

$ kubectl get secret secret-example -oyaml
apiVersion: v1
data:
  secret_key: c2VjcmV0X3ZhbHVl              # c2VjcmV0X3ZhbHVl为secret_value的base64编码
kind: Secret
...
```

则进入Pod中查看的环境变量结果如下。

```
$ kubectl get pod
NAME                       READY   STATUS    RESTARTS   AGE
env-example-695b759569-lx9jp   1/1     Running   0          17m

$ kubectl exec env-example-695b759569-lx9jp  -- printenv
/ # env
key=value                           # 自定义环境变量
key1=configmap_value                # 配置项键值导入
key2=secret_value                   # 密钥键值导入
key3=env-example-695b759569-lx9jp     # Pod的metadata.name
key4=1                              # container1这个容器的limits.cpu，单位为Core，向上取整
configmap_key=configmap_value       # 配置项导入，原配置项中的键值直接会导入结果
secret_key=secret_value             # 密钥导入，原密钥中的键值直接会导入结果
```

9.3.8 工作负载升级策略

在实际应用中，升级是一个常见的场景，Deployment、StatefulSet和DaemonSet都能够很方便的支撑应用升级。

设置不同的升级策略，有如下两种。

- RollingUpdate：滚动升级，即逐步创建新Pod再删除旧Pod，为默认策略。
- Recreate：替换升级，即先把当前Pod删掉再重新创建Pod。

升级参数说明

参数	说明	限制
最大浪涌（maxSurge）	与spec.replicas相比，可以有多少个Pod存在，默认值是25%。 比如spec.replicas为 4，那升级过程中就不能超过5个Pod存在，即按1个的步长升级，实际升级过程中会换算成数字，且换算会向上取整。这个值也可以直接设置成数字。	仅Deployment、DaemonSet支持配置。
最大无效实例数（maxUnavailable）	与spec.replicas相比，可以有多少个Pod失效，也就是删除的比例，默认值是25%。 比如spec.replicas为4，那升级过程中就至少有3个Pod存在，即删除Pod的步长是1。同样这个值也可以设置成数字。	仅Deployment、DaemonSet支持配置。
实例可用最短时间（minReadySeconds）	指定新创建的 Pod 在没有任意容器崩溃情况下的最小就绪时间， 只有超出这个时间 Pod 才被视为可用。默认值为 0（Pod 在准备就绪后立即将被视为可用）。	-
最大保留版本数（revisionHistoryLimit）	用来设定出于回滚目的所要保留的旧 ReplicaSet 数量。 这些旧 ReplicaSet 会消耗 etcd 中的资源，并占用 kubectl get rs 的输出。 每个 Deployment 修订版本的配置都存储在其 ReplicaSets 中；因此，一旦删除了旧的 ReplicaSet， 将失去回滚到 Deployment 的对应修订版本的能力。 默认情况下，系统保留 10 个旧 ReplicaSet，但其理想值取决于新 Deployment 的频率和稳定性。	-

参数	说明	限制
升级最大时长（progressDeadlineSeconds）	指定系统在报告 Deployment 进展失败 之前等待 Deployment 取得进展的秒数。 这类报告会在资源状态中体现为 Type=Progressing、Status=False、Reason=ProgressDeadlineExceeded。 Deployment 控制器将持续重试 Deployment。 将来，一旦实现了自动回滚，Deployment 控制器将在探测到这样的条件时立即回滚 Deployment。 如果指定，则此字段值需要大于 .spec.minReadySeconds 取值。	-
缩容时间窗（terminationGracePeriodSeconds）	优雅删除时间，默认为30秒，删除Pod时发送SIGTERM终止信号，然后等待容器中的应用程序终止执行，如果在terminationGracePeriodSeconds时间内未能终止，则发送SIGKILL的系统信号强行终止。	-

升级示例

Deployment的升级可以是声明式的，也就是说只需要修改Deployment的YAML定义即可，比如使用kubectl edit命令将上面Deployment中的镜像修改为nginx:alpine。修改完成后再查询ReplicaSet和Pod，发现创建了一个新的ReplicaSet，Pod也重新创建了。

```
$ kubectl edit deploy nginx

$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-6f9f58dffd   2         2         2         1m
nginx-7f98958cdf   0         0         0         48m

$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-6f9f58dffd-tdmqk   1/1       Running   0          1m
nginx-6f9f58dffd-tesqr   1/1       Running   0          1m
```

Deployment可以通过maxSurge和maxUnavailable两个参数控制升级过程中同时重新创建Pod的比例，这在很多时候是非常有用，配置如下所示。

```
spec:
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```

在前面的例子中，由于spec.replicas是2，如果maxSurge和maxUnavailable都为默认值25%，那实际升级过程中，maxSurge允许最多3个Pod存在（向上取整，2*1.25=2.5，取整为3），而maxUnavailable则不允许有Pod Unavailable（向上取整，2*0.75=1.5，取整为2），也就是说在升级过程中，一直会有2个Pod处于运行状态，每次新建一个Pod，等这个Pod创建成功后再删掉一个旧Pod，直至Pod全部为新Pod。

回滚

回滚也称为回退，即当发现升级出现问题时，让应用回到老的版本。Deployment可以非常方便的回滚到老版本。

例如上面升级的新版镜像有问题，可以执行kubectl rollout undo命令进行回滚。

```
$ kubectl rollout undo deployment nginx
deployment.apps/nginx rolled back
```

Deployment之所以能如此容易的做到回滚，是因为Deployment是通过ReplicaSet控制Pod的，升级后之前ReplicaSet都一直存在，Deployment回滚做的就是使用之前的ReplicaSet再次把Pod创建出来。Deployment中保存ReplicaSet的数量可以使用revisionHistoryLimit参数限制，默认值为10。

9.3.9 调度策略（亲和与反亲和）

Kubernetes支持节点亲和与Pod亲和/反亲和。通过配置亲和与反亲和规则，可以允许您指定硬性限制或者偏好，例如将前台Pod和后台Pod部署在一起、某类应用部署到某些特定的节点、不同应用部署到不同的节点等等。

Kubernetes的亲和功能由节点和工作负载两种类型组成：

- **节点亲和（nodeAffinity）**：类似于Pod中的nodeSelector字段，使用nodeSelector字段只会将Pod调度到指定标签的节点上，这与节点亲和类似，但节点亲和性的表达能力更强，并且允许指定优先选择的软约束。两种类型的节点亲和如下：
  - requiredDuringSchedulingIgnoredDuringExecution：**必须满足**的硬约束，即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，详情请参见**节点亲和（nodeAffinity）**。
  - preferredDuringSchedulingIgnoredDuringExecution：**尽量满足**的软约束，即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，详情请参见**节点优先选择规则**。
- **工作负载亲和（podAffinity）**/**工作负载反亲和（podAntiAffinity）**：基于已经在节点上运行的Pod标签来约束Pod可以调度到的节点，而不是基于节点上的标签。与节点亲和类似，工作负载亲和与反亲和也有requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution两种类型。

  说明

  工作负载亲和性和反亲和性需要一定的计算时间，因此在大规模集群中会显著降低调度的速度。在包含数百个节点的集群中，不建议使用这类设置。

您可以通过控制台创建上述亲和策略，详情请参见**通过控制台配置调度策略**。

通过控制台配置调度策略

步骤1 登录CCE控制台。

步骤2 在创建工作负载时，在“高级设置”中找到“调度策略”。

表 9-13 节点亲和性设置

参数名	参数描述
必须满足	即硬约束，设置必须要满足的条件，对应于requiredDuringSchedulingIgnoredDuringExecution。 添加多条“必须满足”规则时，只需要满足一条规则就会进行调度。
尽量满足	即软约束，设置尽量满足的条件，对应于preferredDuringSchedulingIgnoredDuringExecution。 添加多条“尽量满足”规则时，满足其中一条或者都不满足也会进行调度。

步骤3 在“节点亲和性”、“工作负载亲和性”、“工作负载反亲和性”下单击 添加调度策略。在弹出的窗口中可以直接添加策略，您也可以单击“指定节点”或“指定可用区”通过控制台快速选择需要调度的节点或可用区。

“指定节点”和“指定可用区”本质也是通过标签实现，只是通过控制台提供了更为便捷的操作，无需手动填写节点标签和标签值。指定节点使用的是 kubernetes.io/hostname 标签，指定可用区使用的是 failure-domain.beta.kubernetes.io/zone 标签。

表 9-14 调度策略设置参数说明

参数名	参数描述
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	可以设置六种匹配关系（In、NotIn、Exists、DoesNotExist、Gt、Lt）。 ● In：亲和/反亲和对象的标签在标签值列表（values字段）中。 ● NotIn：亲和/反亲和对象的标签不在标签值列表（values字段）中。 ● Exists：亲和/反亲和对象存在指定标签名。 ● DoesNotExist：亲和/反亲和对象不存在指定标签名。 ● Gt：仅在节点亲和性中设置，调度节点的标签值大于列表值（字符串比较）。 ● Lt：仅在节点亲和性中设置，调度节点的标签值小于列表值（字符串比较）。
标签值	请填写标签值。
命名空间	仅支持在工作负载亲和/工作负载反亲和调度策略中使用。 指定调度策略生效的命名空间。

参数名	参数描述
拓扑域	仅支持在工作负载亲和/工作负载反亲和调度策略中使用。 先圈定拓扑域（topologyKey）指定的范围，然后再选择策略定义的内容。
权重	仅支持在“尽量满足”策略中添加。

----结束

节点亲和（nodeAffinity）

工作负载节点亲和性规则通过节点标签实现。CCE集群中节点在创建时会自动添加一些标签，您可通过kubectl describe node命令查看，示例如下：

```
$ kubectl describe node 192.168.0.212
Name:               192.168.0.212
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    failure-domain.beta.kubernetes.io/is-baremetal=false
                    failure-domain.beta.kubernetes.io/region=******
                    failure-domain.beta.kubernetes.io/zone=******
                    kubernetes.io/arch=amd64
                    kubernetes.io/availablezone=******
                    kubernetes.io/eniquota=12
                    kubernetes.io/hostname=192.168.0.212
                    kubernetes.io/os=linux
                    node.kubernetes.io/subnetid=fd43acad-33e7-48b2-a85a-24833f362e0e
                    os.architecture=amd64
                    os.name=EulerOS_2.0_SP5
                    os.version=3.10.0-862.14.1.5.h328.eulerosv2r7.x86_64
```

在工作负载调度中，常用的节点标签如下：

- failure-domain.beta.kubernetes.io/region：表示节点所在的区域。
- failure-domain.beta.kubernetes.io/zone：表示节点所在的可用区（availability zone）。
- kubernetes.io/hostname：节点的hostname。

在创建工作负载时，Kubernetes提供了nodeSelector字段，设置该字段后可以让Pod只部署在具有特定标签的节点上。如下所示，Pod只会部署在拥有gpu=true这个标签的节点上。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeSelector:              # 节点选择，当节点拥有gpu=true标签时才在节点上创建Pod
    gpu: true
...
```

通过节点亲和性规则配置，也可以做到同样的事情。相比nodeSelector的方式，节点亲和性规则看起来要复杂很多，但这种方式可以得到更强的表达能力，您可以使用spec.affinity.nodeAffinity字段设置节点亲和性。节点亲和性以下有两种规则：

- requiredDuringSchedulingIgnoredDuringExecution：表示必须满足指定的规则才能将Pod调度到节点。

- preferredDuringSchedulingIgnoredDuringExecution：表示将Pod调度到尽量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。

说明

在上述节点亲和规则中，前半段requiredDuringScheduling或preferredDuringScheduling表示下面定义的规则必须强制满足（require）才会调度Pod到节点上。而后半段IgnoredDuringExecution表示如果节点标签在Kubernetes调度Pod后发生了变更，Pod仍将继续运行不会重新调度。

设置节点亲和性示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  gpu
  labels:
    app:  gpu
spec:
  selector:
    matchLabels:
      app: gpu
  replicas: 3
  template:
    metadata:
      labels:
        app:  gpu
    spec:
      containers:
      - image:  nginx:alpine
        name:  gpu
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
          limits:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: gpu
                operator: In
                values:
                - "true"
```

本示例中，调度的节点**必须**包含一个键名为gpu的标签，且操作符operator的值为In，表示标签值需要在values的列表中，即节点gpu标签的键值为true。其他operator取值请参见**操作符取值说明**。需要说明的是并没有nodeAntiAffinity（节点反亲和），因为NotIn和DoesNotExist操作符可以提供相同的功能。

下面来验证这段规则是否生效，假设某集群有如下三个节点。

```
$ kubectl get node
NAME          STATUS   ROLES    AGE   VERSION
192.168.0.212   Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.94    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.97    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
```

首先给192.168.0.212这个节点打上gpu=true的标签。

```
$ kubectl label node 192.168.0.212 gpu=true
node/192.168.0.212 labeled
```

```
$ kubectl get node -L gpu
NAME           STATUS   ROLES    AGE   VERSION                          GPU
192.168.0.212   Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
192.168.0.94    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.97    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
```

创建这个Deployment，可以发现所有的Pod都部署在了192.168.0.212这个节点上。

```
$ kubectl create -f affinity.yaml
deployment.apps/gpu created

$ kubectl get pod -o wide
NAME                   READY   STATUS    RESTARTS   AGE   IP           NODE
gpu-6df65c44cf-42xw4     1/1     Running   0          15s   172.16.0.37   192.168.0.212
gpu-6df65c44cf-jzjvs     1/1     Running   0          15s   172.16.0.36   192.168.0.212
gpu-6df65c44cf-zv5cl     1/1     Running   0          15s   172.16.0.38   192.168.0.212
```

节点优先选择规则

上面讲的requiredDuringSchedulingIgnoredDuringExecution是一种**强制**选择的规则，节点亲和还有一种优先选择规则，即preferredDuringSchedulingIgnoredDuringExecution，表示会根据规则**优先**选择哪些节点。

为演示这个效果，先为上面的集群添加一个SAS磁盘的节点，并打上DISK=SAS的标签，为另外三个节点打上DISK=SSD的标签。

```
$ kubectl get node -L DISK,gpu
NAME           STATUS   ROLES    AGE     VERSION                          DISK     GPU
192.168.0.100   Ready    <none>   7h23m   v1.15.6-r1-20.3.0.2.B001-15.30.2   SAS
192.168.0.212   Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   SSD      true
192.168.0.94    Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   SSD
192.168.0.97    Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   SSD
```

下面定义一个Deployment，要求Pod优先部署在SSD磁盘的节点上，可以像下面这样定义，使用preferredDuringSchedulingIgnoredDuringExecution规则，给SSD设置权重（weight）为80，而gpu=true权重为20，这样Pod就优先部署在SSD的节点上。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  gpu
  labels:
    app:  gpu
spec:
  selector:
    matchLabels:
      app: gpu
  replicas: 10
  template:
    metadata:
      labels:
        app:  gpu
    spec:
      containers:
      - image:  nginx:alpine
        name:  gpu
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
```

```
      nodeAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 80
          preference:
            matchExpressions:
            - key: DISK
              operator: In
              values:
              - SSD
        - weight: 20
          preference:
            matchExpressions:
            - key: gpu
              operator: In
              values:
              - "true"
```

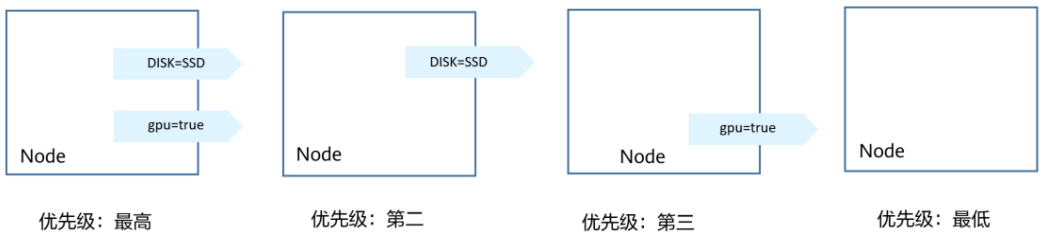
来看实际部署后的情况，可以看到部署到192.168.0.212（标签为DISK=SSD、gpu=true）这个节点上的Pod有5个，192.168.0.97（标签为DISK=SSD）上有3个，而192.168.0.100（标签为DISK=SAS）上只有2个。

这里您看到Pod并没有调度到192.168.0.94（标签为DISK=SSD）这个节点上，这是因为这个节点上部署了很多其他Pod，资源使用较多，所以并没有往这个节点上调度，这也侧面说明preferredDuringSchedulingIgnoredDuringExecution是优先规则，而不是强制规则。

```
$ kubectl create -f affinity2.yaml
deployment.apps/gpu created

$ kubectl get po -o wide
NAME                   READY   STATUS    RESTARTS   AGE     IP            NODE
gpu-585455d466-5bmcz   1/1     Running   0          2m29s   172.16.0.44   192.168.0.212
gpu-585455d466-cg2l6   1/1     Running   0          2m29s   172.16.0.63   192.168.0.97
gpu-585455d466-f2bt2   1/1     Running   0          2m29s   172.16.0.79   192.168.0.100
gpu-585455d466-hdb5n   1/1     Running   0          2m29s   172.16.0.42   192.168.0.212
gpu-585455d466-hkgvz   1/1     Running   0          2m29s   172.16.0.43   192.168.0.212
gpu-585455d466-mngvn   1/1     Running   0          2m29s   172.16.0.48   192.168.0.97
gpu-585455d466-s26qs   1/1     Running   0          2m29s   172.16.0.62   192.168.0.97
gpu-585455d466-sxtzm   1/1     Running   0          2m29s   172.16.0.45   192.168.0.212
gpu-585455d466-t56cm   1/1     Running   0          2m29s   172.16.0.64   192.168.0.100
gpu-585455d466-t5w5x   1/1     Running   0          2m29s   172.16.0.41   192.168.0.212
```

上面这个例子中，对于节点排序优先级如下所示，有个两个标签的节点排序最高，只有SSD标签的节点排序第二（权重为80），只有gpu=true的节点排序第三，没有的节点排序最低。

图 9-4 优先级排序顺序



工作负载亲和（podAffinity）

节点亲和的规则只能影响Pod和节点之间的亲和，Kubernetes还支持Pod和Pod之间的亲和，例如将应用的前端和后端部署在一起，从而减少访问延迟。Pod亲和同样有requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution两种规则。

说明

对于工作负载亲和来说，使用requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution规则时， topologyKey字段不允许为空。

来看下面这个例子，假设有个应用的后端已经创建，且带有app=backend的标签。

```
$ kubectl get po -o wide
NAME                       READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-dlrz8   1/1     Running   0          2m36s   172.16.0.67   192.168.0.100
```

将前端frontend的pod部署在backend一起时，可以做如下Pod亲和规则配置。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - backend
```

创建frontend然后查看，可以看到frontend都创建到跟backend一样的节点上了。

```
$ kubectl create -f affinity3.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                       READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-dlrz8   1/1     Running   0          5m38s   172.16.0.67   192.168.0.100
frontend-67ff9b7b97-dsqzn  1/1     Running   0          6s      172.16.0.70   192.168.0.100
frontend-67ff9b7b97-hxm5t  1/1     Running   0          6s      172.16.0.71   192.168.0.100
frontend-67ff9b7b97-z8pdb  1/1     Running   0          6s      172.16.0.72   192.168.0.100
```

这里有个**topologyKey**字段（用于划分拓扑域），意思是先圈定topologyKey指定的范围，当节点上的标签键、值均相同时会被认为同一拓扑域，然后再选择下面规则定义的内容。这里每个节点上都有kubernetes.io/hostname，所以看不出topologyKey起到的作用。

如果backend有两个Pod，分别在不同的节点上。

```
$ kubectl get po -o wide
NAME                  READY   STATUS    RESTARTS   AGE     IP           NODE
backend-658f6cb858-5bpd6   1/1     Running   0          23m     172.16.0.40   192.168.0.97
backend-658f6cb858-dlrz8   1/1     Running   0          2m36s   172.16.0.67   192.168.0.100
```

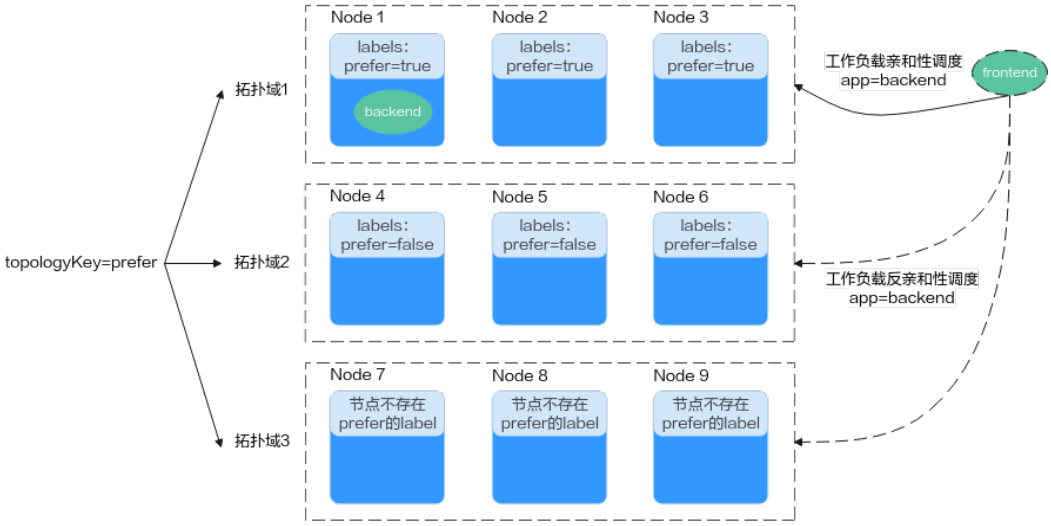
给192.168.0.97和192.168.0.94打上prefer=true的标签。

```
$ kubectl label node 192.168.0.97 prefer=true
node/192.168.0.97 labeled
$ kubectl label node 192.168.0.94 prefer=true
node/192.168.0.94 labeled

$ kubectl get node -L prefer
NAME          STATUS   ROLES    AGE   VERSION                         PREFER
192.168.0.100   Ready    <none>   44m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.212   Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.94    Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
192.168.0.97    Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
```

将podAffinity的topologyKey定义为prefer，则节点拓扑域的划分如图9-5所示。

```
    affinity:
      podAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - topologyKey: prefer
          labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - backend
```

图 9-5 拓扑域示意图



调度时，会根据prefer标签划分节点拓扑域，本示例中192.168.0.97和192.168.0.94被划作同一拓扑域。如果当拓扑域中运行着app=backend的Pod，即使该拓扑域中并非所有节点均运行了app=backend的Pod（本例该拓扑域中仅192.168.0.97节点上存在app=backend的Pod），frontend同样会部署在此拓扑域中（这里的192.168.0.97或192.168.0.94）。

```
$ kubectl create -f affinity3.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
```

```
NAME                        READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-5bpd6    1/1     Running   0          26m     172.16.0.40   192.168.0.97
backend-658f6cb858-dlrz8    1/1     Running   0          5m38s   172.16.0.67   192.168.0.100
frontend-67ff9b7b97-dsqzn   1/1     Running   0          6s      172.16.0.70   192.168.0.97
frontend-67ff9b7b97-hxm5t   1/1     Running   0          6s      172.16.0.71   192.168.0.97
frontend-67ff9b7b97-z8pdb   1/1     Running   0          6s      172.16.0.72   192.168.0.97
```

工作负载反亲和（podAntiAffinity）

前面讲了Pod的亲和，通过亲和将Pod部署在一起，有时候需求却恰恰相反，需要将Pod分开部署，例如Pod之间部署在一起会影响性能的情况。

说明

对于工作负载反亲和来说，使用requiredDuringSchedulingIgnoredDuringExecution规则时，Kubernetes默认的准入控制器 LimitPodHardAntiAffinityTopology要求topologyKey字段只能是kubernetes.io/hostname。如果您希望使用其他定制拓扑逻辑，可以更改或者禁用该准入控制器。

下面例子中定义了反亲和规则，这个规则表示根据kubernetes.io/hostname标签划分节点拓扑域，且如果该拓扑域中的某个节点上已经存在带有app=frontend标签的Pod，那么拥有相同标签的Pod将不能被调度到该拓扑域内的其他节点上。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 5
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname     #节点拓扑域
            labelSelector:     #Pod标签匹配规则
              matchExpressions:
              - key: app
                operator: In
                values:
                - frontend
```

创建并查看部署效果，示例中根据kubernetes.io/hostname标签划分节点拓扑域，在拥有kubernetes.io/hostname标签的节点中，每个节点的标签值均不同，因此一个拓扑域中只有一个节点。当一个拓扑域中（此处为一个节点）已经存在frontend标签的

Pod时，该拓扑域不会被继续调度具有相同标签的Pod。本例中只有4个节点，因此还有一个Pod处于Pending状态无法调度。

```
$ kubectl create -f affinity4.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP           NODE
frontend-6f686d8d87-8dlsc   1/1     Running   0          18s   172.16.0.76   192.168.0.100
frontend-6f686d8d87-d6l8p   0/1     Pending   0          18s   <none>       <none>
frontend-6f686d8d87-hgcq2   1/1     Running   0          18s   172.16.0.54   192.168.0.97
frontend-6f686d8d87-q7cfq   1/1     Running   0          18s   172.16.0.47   192.168.0.212
frontend-6f686d8d87-xl8hx   1/1     Running   0          18s   172.16.0.23   192.168.0.94
```

操作符取值说明

您可以使用操作符（operator字段）来设置使用规则的逻辑关系，operator取值如下：

- In：亲和/反亲和对象的标签在标签值列表（values字段）中。
- NotIn：亲和/反亲和对象的标签不在标签值列表（values字段）中。
- Exists：亲和/反亲和对象存在指定标签名。
- DoesNotExist：亲和/反亲和对象不存在指定标签名。
- Gt：仅在节点亲和性中设置，调度节点的标签值大于列表值（字符串比较）。
- Lt：仅在节点亲和性中设置，调度节点的标签值小于列表值（字符串比较）。

9.3.10 容忍策略

容忍度（Toleration） 允许调度器将Pod调度至带有对应污点的节点上。 容忍度需要和**节点污点**相互配合，每个节点上都可以拥有一个或多个污点，对于未设置容忍度的Pod，调度器会根据节点上的污点效果进行选择性调度，可以用来避免Pod被分配到不合适的节点上。

污点可以指定多种效果，对应的容忍策略对Pod运行影响如下：

污点效果	Pod未设置对污点的容忍策略	Pod已设置对污点的容忍策略
NoExecute	• 已运行在该节点的Pod会立刻被驱逐。 • 未运行的Pod不会被调度到该节点。	• 未指定容忍时间窗（tolerationSeconds）：Pod可以在这个节点上一直运行。 • 已指定容忍时间窗（tolerationSeconds）：在容忍时间窗内，Pod还会在拥有污点的节点上运行，超出时间后会被驱逐。
PreferNoSchedule	• 已运行在该节点的Pod不会被驱逐。 • 未运行的Pod**尽量**不调度到该节点。	Pod可以在这个节点上一直运行。

污点效果	Pod未设置对污点的容忍策略	Pod已设置对污点的容忍策略
NoSchedule	• 已运行在该节点的Pod不会被驱逐。 • 未运行的Pod不会被调度到该节点。	Pod可以在这个节点上一直运行。

通过控制台配置容忍策略

步骤1 登录CCE控制台。

步骤2 在创建工作负载时，在“高级设置”中找到“容忍策略”。

步骤3 添加污点容忍策略。

表 9-15 容忍策略设置参数说明

参数名	参数描述
污点键	节点的污点键。
操作符	• Equal：设置此操作符表示**准确匹配**指定污点键（必填）和污点值的节点。如果不填写污点值，则表示可以与所有污点键相同的污点匹配。 • Exists：设置此操作符表示**匹配存在**指定污点键的节点，此时容忍度不能指定污点值。若不填写污点键则可以容忍全部污点。
污点值	操作符为Equal时需要填写污点值。
污点策略	• 全部：表示匹配所有污点效果。 • NoSchedule：表示匹配污点效果为NoSchedule的污点。 • PreferNoSchedule：表示匹配污点效果为PreferNoSchedule的污点。 • NoExecute：表示匹配污点效果为NoExecute的污点。
容忍时间窗	即tolerationSeconds参数，当污点策略为NoExecute时支持配置。 在容忍时间窗内，Pod还会在拥有污点的节点上运行，超出时间后会被驱逐。

----结束

默认容忍策略说明

Kubernetes会自动给Pod添加针对**node.kubernetes.io/not-ready**和**node.kubernetes.io/unreachable**污点的容忍度，且配置容忍时间窗（tolerationSeconds）为300s。这些默认容忍度策略表示当Pod运行的节点被打上这两个污点之一时，可以在5分钟内依旧保持运行在该节点上。

说明

DaemonSet中的Pod被创建时，针对以上污点自动添加的容忍度将不会指定容忍时间窗，即表示节点存在上述污点时，DaemonSet中的Pod一直不会被驱逐。

```
tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
  - key: node.kubernetes.io/unreachable
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
```

9.3.11 标签与注解

Pod 注解

CCE提供一些使用Pod的高级功能，这些功能使用时可以通过给YAML添加注解Annotation实现。具体的Annotation如下表所示。

表 9-16 Pod Annotation

注解	说明	默认值

kubernetes.AOM.log.stdout	容器标准输出采集参数，不配置默认将全部容器的标准输出上报至AOM，可配置采集指定容器或全部不采集。
	示例：
	• 全部不采集：
	`kubernetes.AOM.log.stdout: '[]'`
	• 采集container-1和container-2容器：
	`kubernetes.AOM.log.stdout: '["container-1","container-2"]'`
		-

metrics.alpha.kubernetes.io/custom-endpoints	AOM监控指标上报参数，可将指定指标上报是AOM服务。 具体使用请参见**使用AOM监控自定义指标**。	-
kubernetes.io/ingress-bandwidth	Pod的入口带宽 具体使用请参见**Pod互访QoS限速**。	-
kubernetes.io/egress-bandwidth	Pod的出口带宽 具体使用请参见**Pod互访QoS限速**。	-

Pod 标签

在控制台创建工作负载时，会默认为Pod添加如下标签，其中app的值为工作负载名称。

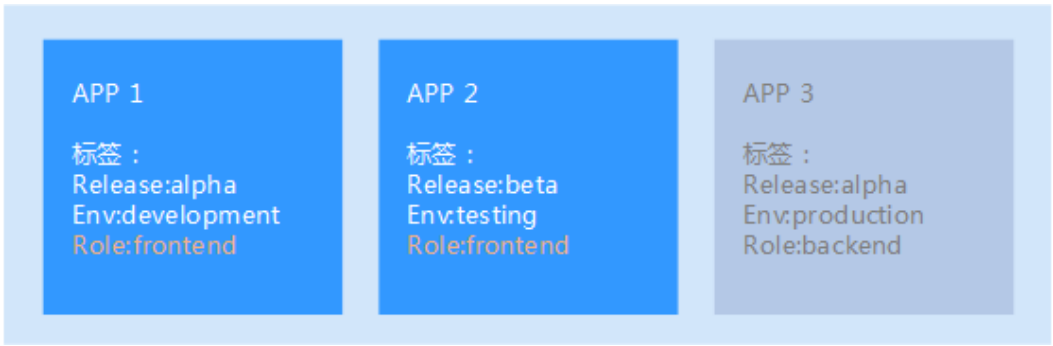
YAML示例如下：

```
...
spec:
  selector:
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      ...
```

您也可以根据需要为Pod添加其他标签，可用于设置工作负载亲和性与反亲和性调度。如下图，假设为工作负载（例如名称为APP1、APP2、APP3）定义了3个Pod标签：release、env、role。不同工作负载定义了不同的取值，分别为：

- APP 1：[release:alpha;env:development;role:frontend]
- APP 2：[release:beta;env:testing;role:frontend]
- APP 3：[release:alpha;env:production;role:backend]

图 9-6 标签案例



例如，设置工作负载亲和性的“key/value”值为“role/backend”，则会选择APP3进行亲和性调度，详情请参见**工作负载亲和（podAffinity）**。

9.4 登录容器

操作场景

如果在使用容器的过程中遇到非预期的问题，您可登录容器进行调试。

使用 kubectl 命令登录容器

步骤1 使用kubectl连接集群，详情请参见**通过kubectl连接集群**。

步骤2 执行以下命令，查看已创建的Pod。

```
kubectl get pod
```

示例输出如下：

```
NAME                          READY   STATUS    RESTARTS      AGE
nginx-59d89cb66f-mhljr        1/1     Running   0             11m
```

步骤3 查询该Pod中的容器名称。

```
kubectl get po nginx-59d89cb66f-mhljr -o jsonpath='{range .spec.containers[*]}{.name}{end}{"\n"}'
```

示例输出如下：

```
container-1
```

步骤4 执行以下命令，登录到nginx-59d89cb66f-mhljr这个Pod中名为container-1的容器。

```
kubectl exec -it nginx-59d89cb66f-mhljr -c container-1 -- /bin/sh
```

步骤5 如需退出容器，可执行**exit**命令。

----结束

9.5 管理工作负载和任务

操作场景

工作负载创建后，您可以对其执行升级、编辑YAML、日志、监控、回退、删除等操作。

表 9-17 工作负载/任务管理

操作	描述
监控	可以通过CCE控制台查看工作负载和容器组的CPU和内存占用情况，以确定需要的资源规格。
日志	可查看工作负载的日志信息。
升级	可以通过更换镜像或镜像版本实现无状态工作负载、有状态工作负载、守护进程集的快速升级，业务无中断。
编辑YAML	可通过在线YAML编辑窗对无状态工作负载、有状态工作负载、守护进程集和容器组的YAML文件进行修改和下载。普通任务和定时任务的YAML文件仅支持查看、复制和下载。
回退	无状态工作负载可以进行回退操作，仅无状态工作负载可用。
重新部署	工作负载可以进行重新部署操作，重新部署后将重启负载下的全部容器组Pod。
关闭/开启升级	无状态工作负载可以进行关闭/开启升级操作，仅无状态工作负载可用。

操作	描述
标签管理	标签是以key/value键值对的形式附加在工作负载上的。添加标签后，可通过标签对工作负载进行管理和选择。任务或定时任务无法使用标签管理功能。
删除	若工作负载无需再使用，您可以将工作负载或任务删除。工作负载或任务删除后，将无法恢复，请谨慎操作。
事件	查看具体实例的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间。
停止/启动	停止/启动一个定时任务，该功能仅定时任务可用。

监控

您可以通过CCE控制台查看工作负载和容器组的CPU和内存占用情况，以确定需要的资源规格。本文以无状态工作负载为例说明如何使用监控功能。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击已创建工作负载后的“监控”。在监控页面，可查看工作负载的CPU利用率和物理内存使用率。

步骤3 单击工作负载名称，可在“实例列表”中单击某个实例的“监控”按钮，查看相应实例的CPU使用率、内存使用率。

----结束

日志

您可以通过“日志”功能查看无状态工作负载、有状态工作负载、守护进程集、普通任务的日志信息。本文以无状态工作负载为例说明如何查看日志。

须知

查看日志前请将浏览器与后端服务器时间调成一致。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击工作负载后的“日志”。

在弹出的“日志”窗口中可以查看容器日志信息。

说明

当前显示的日志内容为容器标准输出日志，不具备持久化和高阶运维能力，如需使用更完善的日志能力，可使用**日志管理**功能。如工作负载开启了AOM采集标准输出的功能（默认开启），可前往AOM查阅更多的负载日志，详情请参见**使用ICAgent采集容器日志**。

----结束

升级

您可以通过CCE控制台实现无状态工作负载、有状态工作负载、守护进程集的快速升级。

本文以无状态工作负载为例说明如何进行升级。

若需要更换镜像或镜像版本，您需要提前将镜像上传到容器镜像服务。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击待升级工作负载后的“升级”。

说明

- 暂不支持批量升级多个工作负载。
- 有状态工作负载升级时，若升级类型为替换升级，需要用户手动删除实例后才能升级成功，否则界面会始终显示“处理中”。

步骤3 请根据业务需求进行工作负载的升级，参数设置方法与创建工作负载时一致。

步骤4 更新完成后，单击“升级工作负载”，并手动确认YAML文件差异后提交升级。

----结束

编辑 YAML

可通过在线YAML编辑窗对无状态工作负载、有状态工作负载、守护进程集和容器组的YAML文件进行修改和下载。普通任务和定时任务的YAML文件仅支持查看、复制和下载。本文以无状态工作负载为例说明如何在线编辑YAML。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击工作负载后的“更多 > 编辑YAML”，在弹出的“编辑YAML”窗中可对当前工作负载的YAML文件进行修改。

步骤3 单击“确定”，完成修改。

步骤4 （可选）在“编辑YAML”窗中，单击“下载”，可下载该YAML文件。

----结束

回退（仅无状态工作负载可用）

所有无状态工作负载的发布历史记录都保留在系统中，您可以回退到指定的版本。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击待回退工作负载后的“更多 > 回退”。

步骤3 切换至“版本记录”页签，并选择回退版本，单击“回退到此版本”，并手动确认YAML文件差异后单击“确定”。

----结束

重新部署

重新部署将重启负载下的全部容器组Pod。本文以无状态工作负载为例说明如何重新部署工作负载。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击工作负载后的“更多 > 重新部署”。

步骤3 在弹出的提示框中单击“是”，即可完成工作负载的重新部署。

----结束

关闭/开启升级（仅无状态工作负载可用）

无状态工作负载可以进行“关闭/开启升级”操作。

- 关闭升级后，对负载进行的升级操作可以正常下发，但不会被应用到实例。
  如果您正在滚动升级的过程中，滚动升级会在关闭升级命令下发后停止，出现新旧实例共存的状态。
- 开启升级后，负载可以正常升级和回退，负载下的实例会与负载当前的最新信息进行一次同步，如果有不一致的，则会自动按照负载的最新信息进行升级。

须知

工作负载状态在关闭升级时无法执行回退操作。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击工作负载后方操作栏中的“更多 > 关闭/开启升级”。

步骤3 在弹出的信息提示框中，单击“是”。

----结束

标签管理

标签是以key/value键值对的形式附加在工作负载上的。添加标签后，可通过标签对工作负载进行管理和选择。您可以给多个工作负载打标签，也可以给指定的某个工作负载打标签。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击工作负载后方操作栏中的“更多 > 标签管理”。

步骤3 单击“添加”，输入键和值后单击“确定”。

说明

标签格式要求如下：以字母和数字开头或结尾，由字母、数字、连接符（-）、下划线（_）、点号（.）组成且63字符以内。

----结束

删除工作负载/任务

若工作负载无需再使用，您可以将工作负载或任务删除。工作负载或任务删除后，将无法恢复，请谨慎操作。本文以无状态工作负载为例说明如何使用删除功能。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 单击待删除工作负载后的“更多 > 删除”，删除工作负载。

请仔细阅读系统提示，删除操作无法恢复，请谨慎操作。

步骤3 单击“是”。

说明

- 若Pod所在节点不可用或者关机，负载无法删除时可以在详情页面实例列表选择强制删除。
- 请确保要删除的存储没有被其他负载使用，导入和存在快照的存储只做解关联操作。

----结束

事件

本文以无状态工作负载为例说明如何使用事件功能。任务或定时任务中的事件功能可直接单击工作负载操作栏中的“事件”按钮查看。

步骤1 登录CCE控制台，进入一个已有的集群，在左侧导航栏中选择“工作负载”。

步骤2 选择“无状态负载”页签，单击工作负载名称，可在“实例列表”中单击某个实例的“事件”按钮，查看该工作负载或具体实例的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间。

说明

事件保存时间为1小时，1小时后自动清除数据。

----结束

10 调度

10.1 调度概述

CCE支持不同类型的资源调度及任务调度等，可提升应用的性能和集群整体资源的利用率。本文介绍CPU资源调度、GPU/NPU异构资源调度、Volcano调度的主要功能。

CPU 调度

CCE提供CPU管理策略为应用分配完整的CPU物理核，提升应用性能，减少应用的调度延迟。

功能	描述	参考文档
CPU管理策略	当节点上运行了很多 CPU 密集的 Pod 时，工作负载可能会迁移到不同的 CPU 核。许多应用对这种迁移不敏感，因此无需任何干预即可正常工作。有些应用对CPU敏感，对于CPU敏感型应用，您可以利用Kubernetes中提供的CPU管理策略为应用分配独占核，提升应用性能，减少应用的调度延迟。	CPU管理策略

GPU 调度

CCE为集群中的GPU异构资源提供调度能力，支持在容器中使用GPU显卡。

功能	描述	参考文档
Kubernetes默认GPU调度	Kubernetes默认GPU调度可以指定Pod申请GPU的数量，支持申请设置为小于1的数量，实现多个Pod共享使用GPU。	使用Kubernetes默认GPU调度

volcano 调度

Volcano是一个基于Kubernetes的批处理平台，提供了机器学习、深度学习、生物信息学、基因组学及其他大数据应用所需要而Kubernetes当前缺失的一系列特性，提供了高性能任务调度引擎、高性能异构芯片管理、高性能任务运行管理等通用计算能力。

功能	描述	参考文档
NUMA亲和性调度	Volcano可解决调度程序NUMA拓扑感知的限制，实现以下目标： • 避免将Pod调度到NUMA拓扑不匹配的节点。 • 将Pod调度到NUMA拓扑的最佳节点。	NUMA亲和性调度

10.2 CPU 调度

10.2.1 CPU 管理策略

使用场景

默认情况下，kubelet使用 **CFS 配额** 来执行Pod的CPU约束。 当节点上运行了很多CPU密集的Pod时，工作负载可能会迁移到不同的CPU核， 这取决于调度时Pod是否被扼制，以及哪些CPU核是可用的。许多应用对这种迁移不敏感，因此无需任何干预即可正常工作。有些应用对CPU敏感，CPU敏感型应用有如下特点。

- 对CPU throttling 敏感
- 对上下文切换敏感
- 对处理器缓存未命中敏感
- 对跨Socket内存访问敏感
- 期望运行在同一物理CPU的超线程

如果您的应用有以上其中一个特点，可以利用Kubernetes中提供的CPU管理策略为应用分配独占的CPU核（即CPU绑核），提升应用性能，减少应用的调度延迟。CPU manager会优先在一个Socket上分配资源，也会优先分配完整的物理核，避免一些干扰。

开启 CPU 管理策略

CPU 管理策略通过kubelet参数*--cpu-manager-policy*来指定。Kubernetes默认支持两种策略：

- 关闭（none）：默认策略，显式地启用现有的默认CPU亲和方案，不提供操作系统调度器默认行为之外的亲和性策略。
- 开启（static）：针对CPU申请值设置为整数的**Guaranteed Pods**，它允许该类Pod中的容器访问节点上的独占CPU资源（绑核）。

在创建集群时的高级配置中可以配置CPU管理策略。

另外在节点池中也可以配置CPU管理策略，配置后会自动修改节点的上kubelet参数 --cpu-manager-policy。登录CCE控制台，单击集群名称进入集群，在左侧选择“节点

管理”，在右侧选择“节点池”页签，单击节点池名称后的“更多 > 配置管理”，将cpu-manager-policy的值修改为static即可。

为 Pod 设置独占 CPU

Pod设置独占CPU（即CPU绑核）有如下几点要求：

- 节点上开启静态（static）CPU管理策略，具体方法请参见**开启CPU管理策略**。
- Pod的定义里都要设置requests和limits参数，requests和limits必须为整数，且数值一致。
- 如果有init container需要设置独占CPU，init container的requests参数建议与业务容器设置的requests参数一致（避免业务容器未继承init container的CPU分配结果，导致CPU manager多预留一部分CPU）。更多信息请参见**App Containers can't inherit Init Containers CPUs - CPU Manager Static Policy**。

在使用时您可以利用**调度策略（亲和与反亲和）**将如上配置的Pod调度到开启静态（static）CPU管理策略的节点上，这样就能够达到独占CPU的效果。

设置独占CPU的YAML示例如下：

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    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
```

10.3 GPU 调度

10.3.1 使用 Kubernetes 默认 GPU 调度

CCE支持在容器中使用GPU资源。

前提条件

- 创建GPU类型节点，具体请参见**创建节点**。
- 安装gpu-device-plugin（原gpu-beta）插件，安装时注意要选择节点上GPU对应的驱动，具体请参见**gpu-device-plugin**。

- gpu-device-plugin（原gpu-beta）插件会把驱动的目录挂载到/usr/local/nvidia/lib64，在容器中使用GPU资源需要将/usr/local/nvidia/lib64追加到LD_LIBRARY_PATH环境变量中。

  通常可以通过如下三种方式追加。

  a. 制作镜像的Dockerfile中配置LD_LIBRARY_PATH。（推荐）
  ```
  ENV LD_LIBRARY_PATH /usr/local/nvidia/lib64:$LD_LIBRARY_PATH
  ```
  b. 镜像的启动命令中配置LD_LIBRARY_PATH。
  ```
  /bin/bash -c "export LD_LIBRARY_PATH=/usr/local/nvidia/lib64:$LD_LIBRARY_PATH && ..."
  ```
  c. 创建工作负载时定义LD_LIBRARY_PATH环境变量（需确保容器内未配置该变量，不然会被覆盖）。
  ```
  env:
    - name: LD_LIBRARY_PATH
      value: /usr/local/nvidia/lib64
  ```

使用 GPU

创建工作负载申请GPU资源，可按如下方法配置，指定显卡的数量。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gpu-test
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gpu-test
  template:
    metadata:
      labels:
        app: gpu-test
    spec:
      containers:
      - image: nginx:perl
        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
```

通过**nvidia.com/gpu**指定申请GPU的数量，支持申请设置为小于1的数量，比如**nvidia.com/gpu: 0.5**，这样可以多个Pod共享使用GPU。GPU数量小于1时，不支持跨GPU分配，如0.5 GPU只会分配到一张卡上。

说明

使用**nvidia.com/gpu**参数指定GPU数量时，requests和limits值需要保持一致。

指定**nvidia.com/gpu**后，在调度时不会将负载调度到没有GPU的节点。如果缺乏GPU资源，会报类似如下的Kubernetes事件。

- 0/2 nodes are available: 2 Insufficient nvidia.com/gpu.
- 0/4 nodes are available: 1 InsufficientResourceOnSingleGPU, 3 Insufficient nvidia.com/gpu.

在CCE控制台使用GPU资源，只需在创建负载时，勾选GPU配额，并指定使用的比例即可。

GPU 节点标签

创建GPU节点后，CCE会给节点打上对应标签，如下所示，不同类型的GPU节点有不同标签。

```
$ kubectl get node -L accelerator
NAME          STATUS   ROLES    AGE     VERSION                                 ACCELERATOR
10.100.2.179   Ready    <none>   8m43s   v1.19.10-r0-CCE21.11.1.B006-21.11.1.B006   nvidia-t4
```

在使用GPU时，可以根据标签让Pod与节点亲和，从而让Pod选择正确的节点，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gpu-test
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gpu-test
  template:
    metadata:
      labels:
        app: gpu-test
    spec:
      nodeSelector:
        accelerator: nvidia-t4
      containers:
      - image: nginx:perl
        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
```

10.4 volcano 调度

10.4.1 NUMA 亲和性调度

背景信息

当节点运行许多CPU绑定的Pod时，工作负载可以迁移到不同的CPU核心，这取决于Pod是否被限制以及调度时哪些CPU核心可用。许多工作负载对此迁移不敏感，因此在没有任何干预的情况下工作正常。但是，在CPU缓存亲和性和调度延迟显著影响工作负载性能的工作负载中，kubelet允许替代CPU管理策略来确定节点上的一些放置首选项。

CPU Manager和拓扑管理器都是kubelet组件，但有以下限制：

- K8s默认调度器不感知NUMA拓扑。因此，可能会调度到不满足NUMA拓扑要求的节点上，然后工作负载实例启动失败。这对于Tensorflow作业来说是不可接受的。如果节点上有任何工作进程或ps失败，则作业将失败。
- 管理器是节点级的，导致无法匹配整个集群中NUMA拓扑的最佳节点。

更多资料请查看社区NUMA亲和性插件指导链接：**https://github.com/volcano-sh/volcano/blob/master/docs/design/numa-aware.md**

volcano的目标是解决调度程序NUMA拓扑感知的限制，以便实现以下目标：

- 避免将Pod调度到NUMA拓扑不匹配的节点。
- 将Pod调度到NUMA拓扑的最佳节点。

支持范围

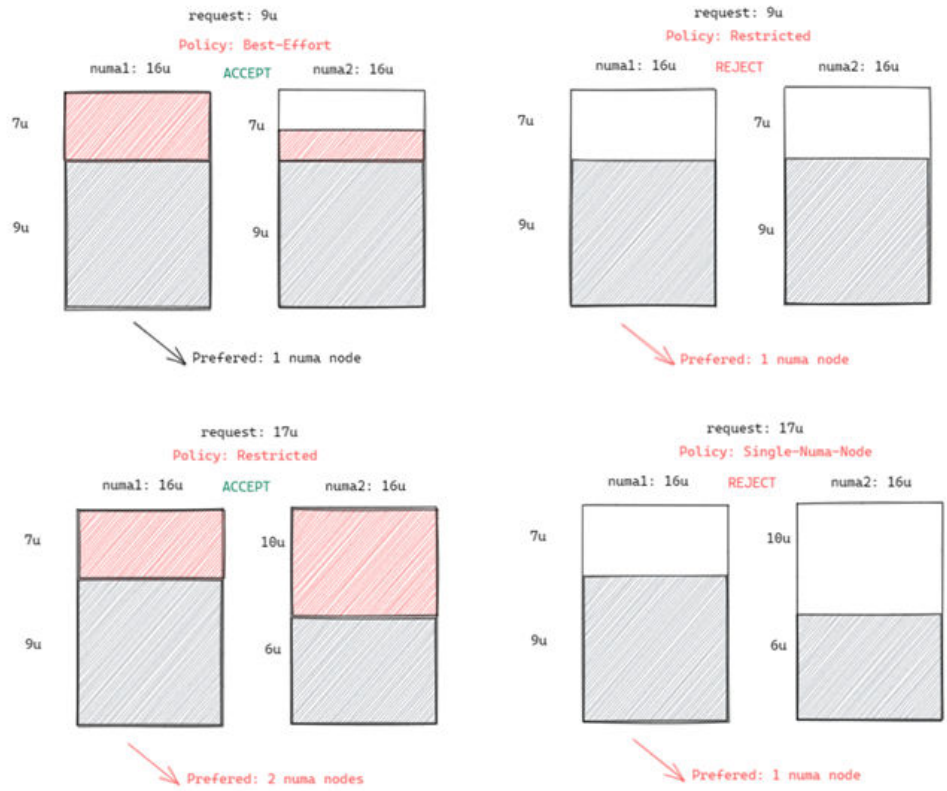
- 支持CPU资源拓扑调度
- 支持Pod级拓扑策略

调度预测

对于具有拓扑策略的Pod，需要预测匹配的节点列表。

policy	action
none	1. 无过滤行为
best-effort	1. 过滤拓扑策略为“best-effort”的节点
restricted	1. 过滤拓扑策略为“restricted”的节点 2. 过滤cpu拓扑满足“restricted”策略要求的节点
single-numa-node	1. 过滤拓扑策略为“single-numa-node”的节点 2. 过滤cpu拓扑满足“single-numa-node”策略要求的节点

图 10-1 NUMA 调度策略对比



调度优先级

不管是什么拓扑策略，都是希望把Pod调度到当时最优的节点上，这里通过给每一个节点进行打分的机制来排序筛选最优节点。

原则：尽可能把pod调度到需要跨numa节点最少的工作节点上。

打分公式如下：

score = weight * (100 - 100 * numaNodeNum / maxNumaNodeNum)

参数说明：

- **weight**：NUMA Aware Plugin的权重。
- **numaNodeNum**：表示工作节点上运行该pod需要numa节点的个数。
- **maxNumaNodeNum**：表示所有工作节点中该pod的最大numa节点个数。

开启 volcano 支持 NUMA 亲和性调度

步骤1 开启CPU管理策略，具体请参考 **开启CPU管理策略**。

步骤2 配置CPU拓扑策略。

1. 登录CCE控制台，单击集群名称进入集群，在左侧选择“节点管理”，在右侧选择“节点池”页签，单击节点池名称后的“更多 > 配置管理”。
2. 在“全量配置”中将kubelet的**topology-manager-policy**的值修改为需要的CPU拓扑策略即可。

有效拓扑策略为“none”、“best-effort”、“restricted”、“single-numa-node”，具体策略对应的调度行为请参见**调度预测**。

步骤3 开启numa-aware插件功能和resource_exporter功能。

volcano 1.7.1及以上版本

1. 登录CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件管理”，在右侧找到volcano，单击“编辑”，并在“参数配置”中设置volcano调度器配置参数。

```
{
  "ca_cert": "",
  "default_scheduler_conf": {
    "actions": "allocate, backfill",
    "tiers": [
      {
        "plugins": [
          {
            "name": "priority"
          },
          {
            "name": "gang"
          },
          {
            "name": "conformance"
          }
        ]
      },
      {
        "plugins": [
          {
            "name": "drf"
          },
          {
            "name": "predicates"
          },
          {
            "name": "nodeorder"
          }
        ]
      },
      {
        "plugins": [
          {
            "name": "cce-gpu-topology-predicate"
          },
          {
            "name": "cce-gpu-topology-priority"
          },
          {
            "name": "cce-gpu"
          },
          {
            // add this also enable resource_exporter
            "name": "numa-aware",
            // the weight of the NUMA Aware Plugin
            "arguments": {
              "weight": "10"
            }
          }
        ]
      },
      {
        "plugins": [
          {
            "name": "nodelocalvolume"
          },
          {
```

```
                "name": "nodeemptydirvolume"
            },
            {
                "name": "nodeCSIscheduling"
            },
            {
                "name": "networkresource"
            }
        ]
      }
    ]
  },
  "server_cert": "",
  "server_key": ""
}
```

volcano 1.7.1以下版本

1. volcano插件开启resource_exporter_enable参数，用于收集节点numa拓扑信息。

```
{
  "plugins": {
    "eas_service": {
      "availability_zone_id": "",
      "driver_id": "",
      "enable": "false",
      "endpoint": "",
      "flavor_id": "",
      "network_type": "",
      "network_virtual_subnet_id": "",
      "pool_id": "",
      "project_id": "",
      "secret_name": "eas-service-secret"
    }
  },
  "resource_exporter_enable": "true"
}
```

   开启后可以查看当前节点的numa拓扑信息。

```
kubectl get numatopo
NAME             AGE
node-1           4h8m
node-2           4h8m
node-3           4h8m
```

2. 启用volcano numa-aware算法插件。

   kubectl edit cm -n kube-system volcano-scheduler-configmap

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: volcano-scheduler-configmap
  namespace: kube-system
data:
  default-scheduler.conf: |-
    actions: "allocate, backfill"
    tiers:
    - plugins:
      - name: priority
      - name: gang
      - name: conformance
    - plugins:
      - name: overcommit
      - name: drf
      - name: predicates
      - name: nodeorder
    - plugins:
      - name: cce-gpu-topology-predicate
      - name: cce-gpu-topology-priority
      - name: cce-gpu
    - plugins:
      - name: nodelocalvolume
```

```
    - name: nodeemptydirvolume
    - name: nodeCSIscheduling
    - name: networkresource
      arguments:
        NetworkType: vpc-router
    - name: numa-aware # add it to enable numa-aware plugin
      arguments:
        weight: 10 # the weight of the NUMA Aware Plugin
```

----结束

使用 volcano 支持 NUMA 亲和性调度

步骤1 在无状态工作负载中配置NUMA亲和性，示例如下：

```
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
    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
```

步骤2 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
```

步骤3 确认NUMA使用情况。

```
# 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
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-8fcd-76c1aa64e6fd":
{"container-1":"1-9"}},"checksum":318470969}
```

----结束

11 网络

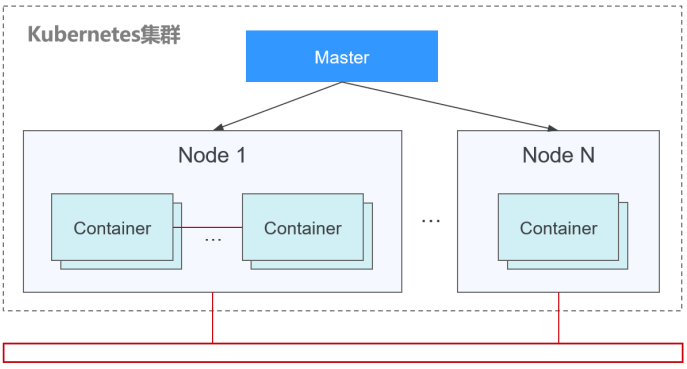
11.1 网络概述

关于集群的网络，可以从如下两个角度进行了解：

- 集群网络是什么样的：集群由多个节点构成，集群中又运行着Pod（容器），每个Pod都需要访问，节点与节点、节点与Pod、Pod与Pod都需要访问。那集群中包含有哪些网络，各自的用处是什么，具体请参见**集群网络构成**。
- 集群中的Pod是如何访问的：访问Pod就是访问容器，也就是访问用户的业务，Kubernetes提供**Service**和**Ingress**来解决Pod的访问问题。本章节根据用户使用场景总结了常见的**网络访问场景**，让您能够在不同使用场景下选择合适的使用方法。

集群网络构成

集群中节点都位于VPC中，节点使用VPC的网络，容器的网络是使用专门的网络插件来管理。



- **节点网络**

  节点网络为集群内主机（节点，图中的Node）分配IP地址，您需要选择VPC中的子网用于CCE集群的节点网络。子网的可用IP数量决定了集群中可以创建节点数量的上限（包括Master节点和Node节点），集群中可创建节点数量还受容器网络的影响，在容器网络模型中会进一步说明。

- **容器网络**

为集群内容器分配IP地址。CCE继承Kubernetes的IP-Per-Pod-Per-Network的容器网络模型，即每个Pod在每个网络平面下都拥有一个独立的IP地址，Pod内所有容器共享同一个网络命名空间，集群内所有Pod都在一个直接连通的扁平网络中，无需NAT可直接通过Pod的IP地址访问。Kubernetes只提供了如何为Pod提供网络的机制，并不直接负责配置Pod网络；Pod网络的具体配置操作交由具体的容器网络插件实现。容器网络插件负责为Pod配置网络并管理容器IP地址。

当前CCE支持如下容器网络模型。

- 容器隧道网络：容器隧道网络在节点网络基础上通过隧道封装构建的独立于节点网络平面的容器网络平面，CCE集群容器隧道网络使用的封装协议为VXLAN，后端虚拟交换机采用的是openvswitch，VXLAN是将以太网报文封装成UDP报文进行隧道传输。
- VPC网络：VPC网络采用VPC路由方式与底层网络深度整合，适用于高性能场景，节点数量受限于虚拟私有云VPC的路由配额。每个节点将会被分配固定大小的IP地址段。VPC网络由于没有隧道封装的消耗，容器网络性能相对于容器隧道网络有一定优势。VPC网络集群由于VPC路由中配置有容器网段与节点IP的路由，可以支持集群外直接访问容器实例等特殊场景。
- 云原生2.0网络：云原生网络2.0是自研的新一代容器网络模型，深度整合了虚拟私有云VPC的弹性网卡（Elastic Network Interface，简称ENI）和辅助弹性网卡（Sub Network Interface，简称Sub-ENI）的能力，直接从VPC网段内分配容器IP地址，支持ELB直通容器，绑定安全组，绑定弹性公网IP，享有高性能。

不同容器网络模型，容器网络的性能、组网规模、适用场景各不相同，在**容器网络模型对比**章节，将会详细介绍不同容器网络模型的功能特性，了解这些有助于您选择容器网络模型。

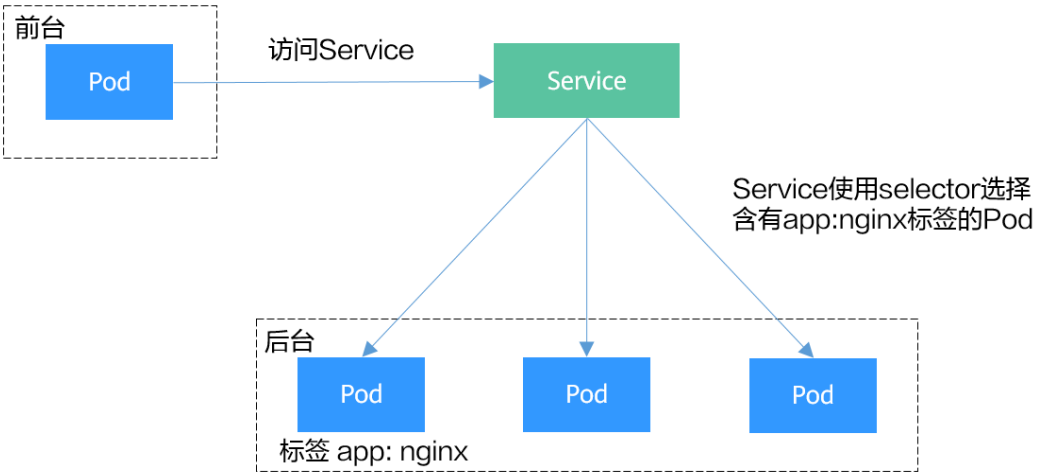
- **服务网络**

  服务（Service）是Kubernetes内的概念，每个Service都有一个固定的IP地址，在CCE上创建集群时，可以指定Service的地址段（即服务网段）。服务网段不能和节点网段、容器网段重叠。服务网段只在集群内使用，不能在集群外使用。

Service

Service是用来解决Pod访问问题的。每个Service有一个固定IP地址，Service将访问流量转发给Pod，而且Service可以给这些Pod做负载均衡。

图 11-1 通过 Service 访问 Pod



根据创建Service的类型不同，可分成如下模式：

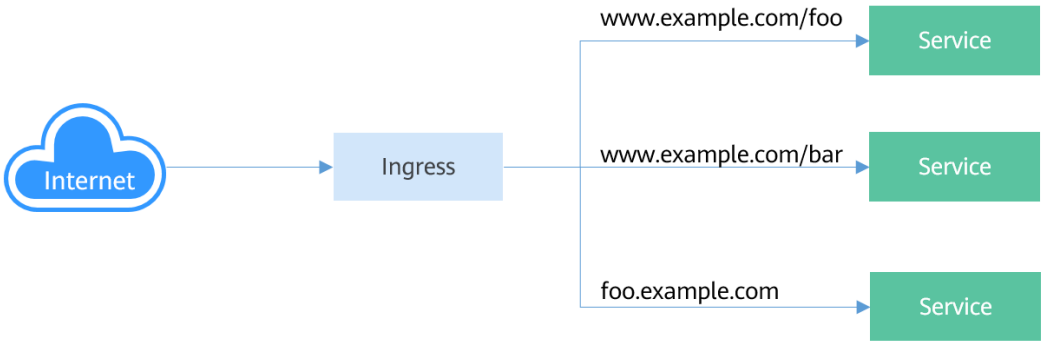
- ClusterIP：用于在集群内部互相访问的场景，通过ClusterIP访问Service。
- NodePort：用于从集群外部访问的场景，通过节点上的端口访问Service。
- LoadBalancer：用于从集群外部访问的场景，其实是NodePort的扩展，通过一个特定的LoadBalancer访问Service，这个LoadBalancer将请求转发到节点的NodePort，而外部只需要访问LoadBalancer。

Service的详细介绍请参见**服务概述**。

Ingress

Service是基于四层TCP和UDP协议转发的，而Ingress可以基于七层的HTTP和HTTPS协议转发，可以通过域名和路径做到更细粒度的划分，如下图所示。

图 11-2 Ingress-Service



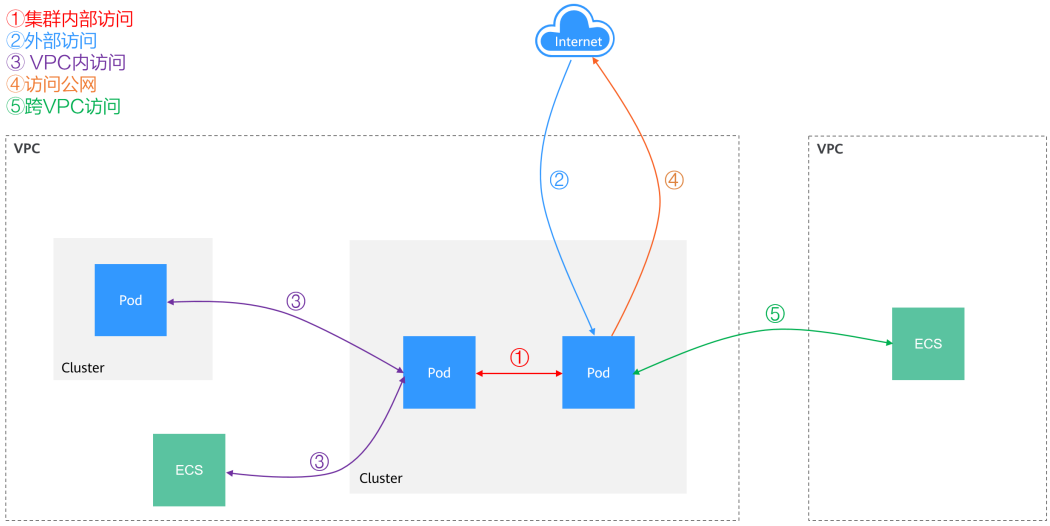
Ingress的详细介绍请参见**路由概述**。

网络访问场景

工作负载网络访问可以分为如下几种场景。

- 从集群内部访问工作负载：创建ClusterIP类型的Service，通过Service访问工作负载。
- 从集群外部访问工作负载：从集群外部访问工作负载推荐使用Service（NodePort类型或LoadBalancer类型）或Ingress访问。
  - 通过公网访问工作负载：需要节点或LoadBalancer绑定公网IP。
  - 通过内网访问工作负载：通过节点或LoadBalancer的内网IP即可访问工作负载。如果跨VPC需要通过对等连接等手段打通不同VPC网络。
- 工作负载访问外部网络：
  - 工作负载访问内网：负载访问内网地址，在不同容器网络模型下有不同的表现，需要注意在对端安全组放通容器网段。
  - 工作负载访问公网：访问公网有几种方法可以实现，一是让容器所在节点绑定公网IP，另一个是通过NAT网关配置SNAT规则，具体请参见**从容器访问公网**。

图 11-3 网络访问示意图



11.2 容器网络模型

11.2.1 容器网络模型对比

容器网络为集群内Pod分配IP地址并提供网络服务，CCE支持如下几种网络模型，您可在创建集群时进行选择。

- **容器隧道网络**
- **VPC网络**
- **云原生网络2.0**

网络模型对比

表11-1主要介绍CCE所支持的网络模型，您可根据实际业务需求进行选择。

⚠ 注意

集群创建成功后，网络模型不可更改，请谨慎选择。

表 11-1 网络模型对比

对比维度	容器隧道网络	VPC网络	云原生网络2.0
适用场景	• 一般容器业务场景。 • 对网络时延、带宽要求不是特别高的场景。	• 对网络时延、带宽要求高。 • 容器与虚机IP互通，使用了微服务注册框架，如Dubbo、CSE等。	• 对网络时延、带宽要求高，高性能场景。 • 容器与虚机IP互通，使用了微服务注册框架的，如Dubbo、CSE等。

对比维度	容器隧道网络	VPC网络	云原生网络2.0
核心技术	OVS	IPVlan，VPC路由	VPC弹性网卡/弹性辅助网卡
适用集群	CCE集群	CCE集群	CCE Turbo集群
网络隔离	Pod支持Kubernetes原生NetworkPolicy	否	Pod支持使用安全组隔离
ELB直通容器	否	否	是
IP地址管理	• 容器网段单独分配 • 节点维度划分地址段，动态分配（地址段分配后可动态增加）	• 容器网段单独分配 • 节点维度划分地址段，静态分配（节点创建完成后，地址段分配即固定，不可更改）	容器网段从VPC子网划分，无需单独分配
网络性能	基于vxlan隧道封装，有一定性能损耗。	无隧道封装，跨节点通过VPC 路由器转发，性能好，媲美主机网络。	容器网络与VPC网络融合，性能无损耗
组网规模	最大可支持2000节点	最大可支持2000节点，受限于VPC路由表能力。 VPC网络模式下，集群每添加一个节点，会在VPC的路由表中添加一条路由，因此集群本身规模受VPC路由表上限限制。	最大可支持2000节点

须知

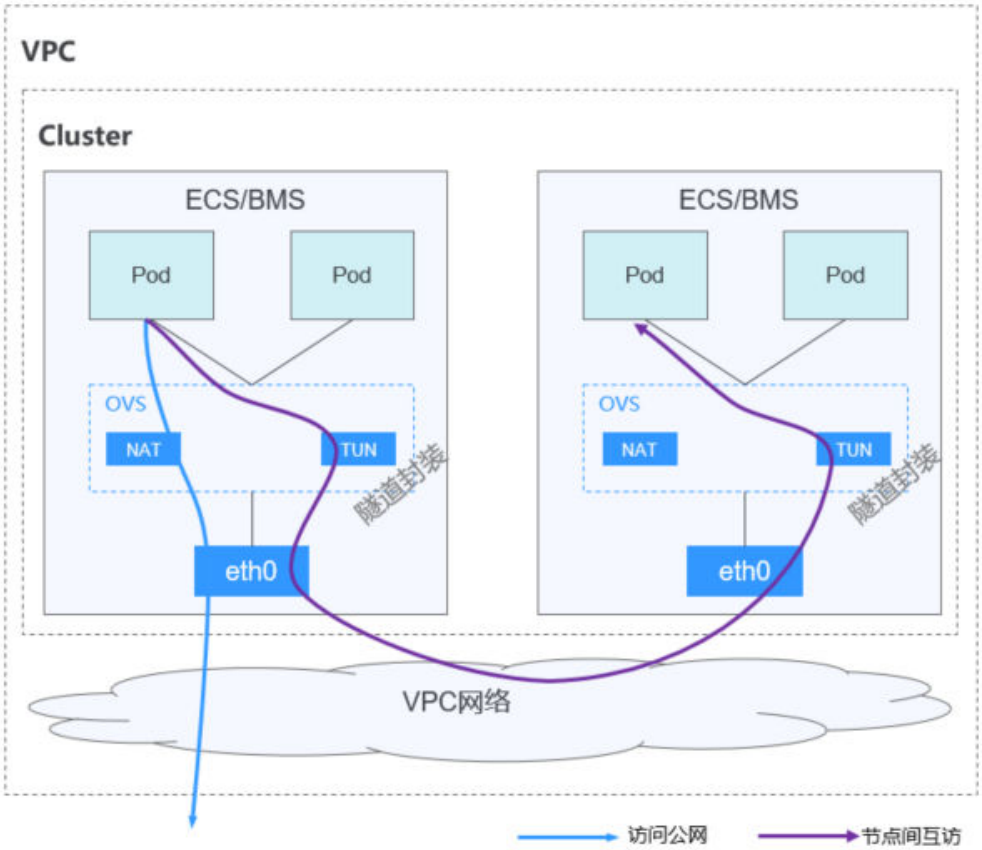
1. VPC路由网络集群实际支持规模受限于VPC的路由表路由条目配额，创建前请提前评估集群规模。
2. 云原生网络2.0集群实际支持规模受限于网络平面选择的VPC子网网段大小，创建前请提前评估集群规模。
3. VPC路由网络默认支持容器与同一VPC的虚拟机直接互访，与其他VPC的主机在配置对等连接策略后可以支持直接互访。此外，云专线/VPN等混合组网场景在合理规划后可以支持对端直接与容器互访。
4. 请勿在创建集群后修改VPC侧的主网段掩码大小，若强行修改会导致VPC集群新建节点的部分网络功能异常。

11.2.2 容器隧道网络

容器隧道网络模型

容器隧道网络在节点网络基础上通过隧道封装构建的独立于节点网络平面的容器网络平面，CCE集群容器隧道网络使用的封装协议为VXLAN，后端虚拟交换机采用的是openvswitch，VXLAN是将以太网报文封装成UDP报文进行隧道传输。容器隧道网络具有付出少量隧道封装性能损耗，即可获得通用性强、互通性强、高级特性支持全面（例如NetworkPolicy网络隔离）的优势，可以满足大多数性能要求不高的场景。

图 11-4 容器隧道网络



说明如下：

- 节点内Pod间通信：同节点的Pod间通信直接通过本节点的ovs网桥直接转发。
- 跨节点Pod间通信：所有跨节点Pod间的通信通过ovs隧道网桥进行封装后，转发到对端节点上。

优缺点

优点

- 容器网络和节点网络解耦，不受VPC配额规格、响应速度的限制（如VPC路由条目数、弹性网卡数、创建速度限制）
- 支持网络隔离，具体请参见**网络策略（NetworkPolicy）**

- 支持带宽限制
- 支持大规模组网

缺点

- 由于隧道封装，网络问题排查难度较大，整体性能较低
- Pod无法直接利用EIP、安全组等能力
- 不支持外部网络与容器IP直通

应用场景

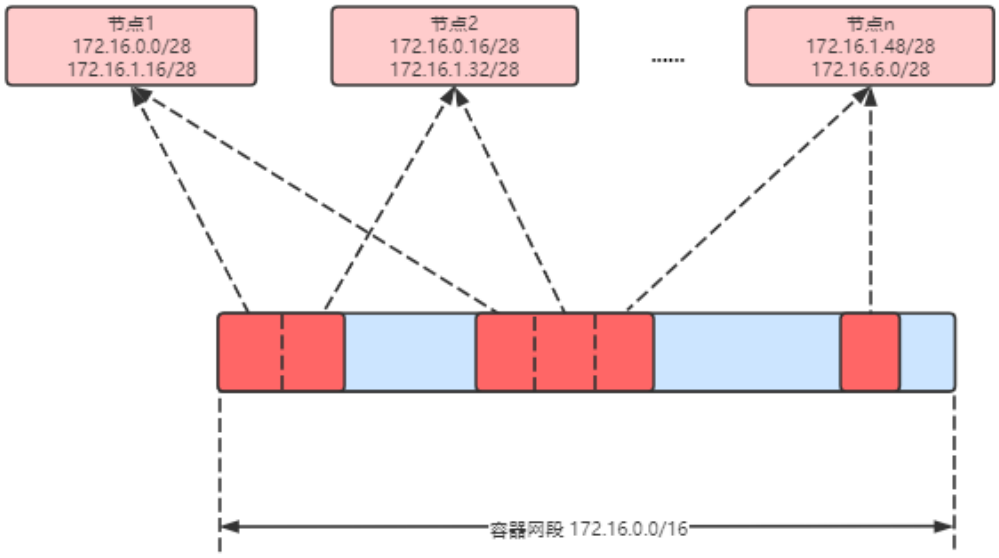
- 对性能要求不高：由于需要额外的VXLAN隧道封装，相对于另外两种容器网络模式，性能存在一定的损耗。大概有5%-15%的性能损失。所以容器隧道网络适用于对性能要求不是特别高的业务场景，比如：web应用、访问量不大的数据中台、后台服务等。
- 大规模组网：相比VPC路由网络受限于VPC路由条目配额的限制，容器隧道网络没有网络基础设施的任何限制；同时容器隧道网络把广播域控制到了节点级别，容器隧道网络最大可支持2000节点规模。

容器 IP 地址管理

容器隧道网络按如下规则分配容器IP：

- 容器网段需单独分配，与节点网段无关
- 节点维度划分地址段，集群的所有节点从容器网段中分配一个或多个固定大小（默认16）的IP网段
- 当节点上的IP地址使用完后，可再次申请分配一个新的IP网段
- 容器网段依次循环分配IP网段给新增节点或存量节点
- 调度到节点上的Pod依次循环从分配给节点的一个或多个IP网段内分配IP地址

图 11-5 容器隧道网络 IP 地址分配



按如上IP分配，容器隧道网络的集群最多能创建节点数量 = 容器网段IP数量 ÷ 节点从容器网段中一次分配的IP网段大小（默认为16）

比如容器网段为172.16.0.0/16，则IP数量为65536，一次分配16，则最多可创建节点数量为65536/16=4096。这是一种极端情况，如果创建4096个节点，则每个节点最多只能创建16个Pod，因为给每个节点只分配了16个IP的网段。另外集群能创建多少节点，还受节点网络和集群规模的影响。

网段规划建议

在**集群网络构成**中介绍集群中网络地址可分为节点网络、容器网络、服务网络三块，在规划网络地址时需要从如下方面考虑：

- **三个网段不能重叠**，否则会导致冲突。
- 保证**每个网段有足够的IP地址可用**。
  - 节点网段的IP地址要与集群规模相匹配，否则会因为IP地址不足导致无法创建节点。
  - 容器网段的IP地址要与业务规模相匹配，否则会因为IP地址不足导致无法创建Pod。每个节点上可以创建多少Pod还与其他参数设置相关。

容器隧道网络访问示例

创建一个容器隧道网络的集群。在集群中创建一个Deployment。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: example
  namespace: default
spec:
  replicas: 4
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
```

创建后查看Pod。

```
$ kubectl get pod -owide
NAME                  READY   STATUS    RESTARTS   AGE     IP          NODE           NOMINATED NODE
READINESS GATES
example-5bdc5699b7-5rvq4   1/1     Running   0          3m28s   10.0.0.20   192.168.0.42   <none>
<none>
example-5bdc5699b7-984j9   1/1     Running   0          3m28s   10.0.0.21   192.168.0.42   <none>
<none>
example-5bdc5699b7-lfxkm   1/1     Running   0          3m28s   10.0.0.22   192.168.0.42   <none>
```

```
<none>
example-5bdc5699b7-wjcmg   1/1     Running   0          3m28s   10.0.0.52   192.168.0.64   <none>
<none>
```

此时如果在集群同VPC下集群外部直接访问Pod的IP，会发现访问不通，这就是容器隧道网络的特性，不支持外部网络与容器IP直通。

而在集群内部节点或Pod内，都能正常访问Pod，如下进入到容器中直接访问Pod能够正常访问。

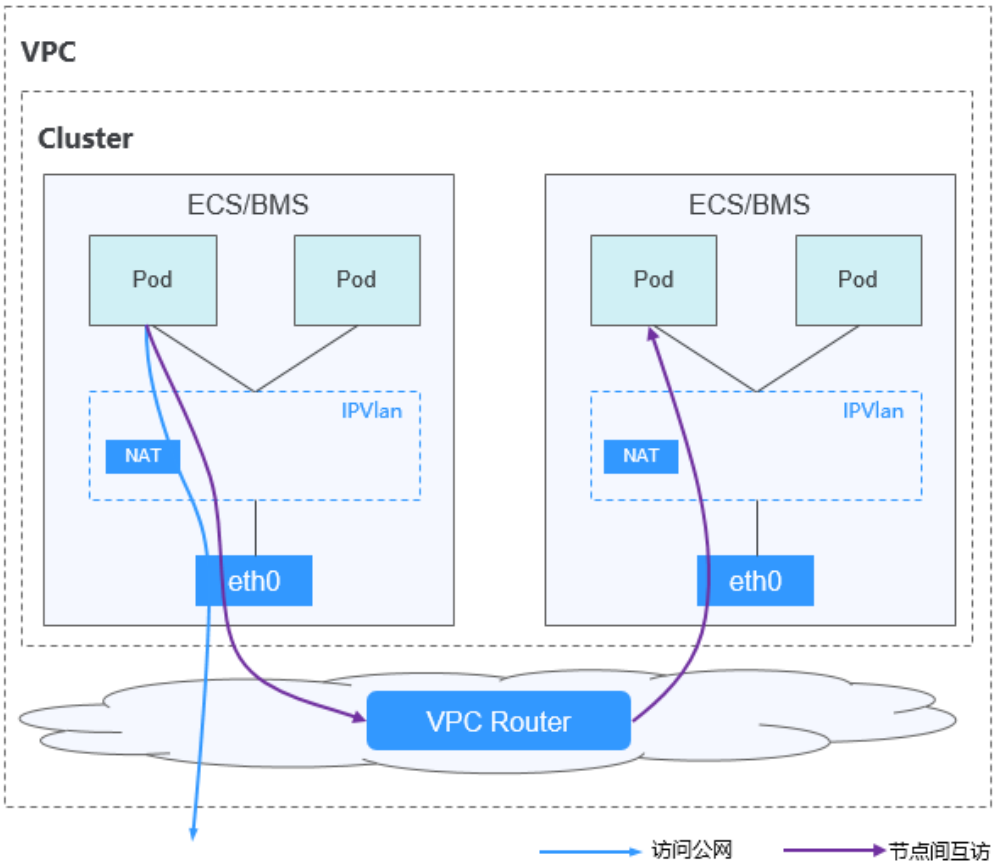
```
$ kubectl exec -it example-5bdc5699b7-5rvq4 -- curl 10.0.0.21
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

11.2.3 VPC 网络

VPC 网络模型

VPC网络采用VPC路由方式与底层网络深度整合，适用于高性能场景，节点数量受限于虚拟私有云VPC的路由配额。每个节点将会被分配固定大小的IP地址段。VPC网络由于没有隧道封装的消耗，容器网络性能相对于容器隧道网络有一定优势。VPC网络集群由于VPC路由中配置有容器网段与节点IP的路由，可以支持同一VPC内的云服务器从集群外直接访问容器实例等特殊场景。

图 11-6 VPC 网络



说明如下：

- 节点内Pod间通信：ipvlan子接口分配给节点上的Pod，同节点的Pod间通信直接通过ipvlan直接转发。
- 跨节点Pod间通信：所有跨节点Pod间的通信通过默认路由到默认网关，借助VPC的路由转发能力，转发到对端节点上。

优缺点

优点

- 由于没有隧道封装，网络问题易排查、性能较高
- 同VPC内，集群外部网络可与容器IP直通

缺点

- 节点数量受限于虚拟私有云VPC的路由配额
- 每个节点将会被分配固定大小的IP地址段，存在一定的容器网段IP地址浪费
- Pod无法直接利用EIP、安全组等能力

应用场景

- 性能要求高：由于没有额外的隧道封装，相比于容器隧道网络模式，性能接近于VPC网络的性能，所以适用于对性能要求较高的业务场景，比如：AI计算、大数据计算等。

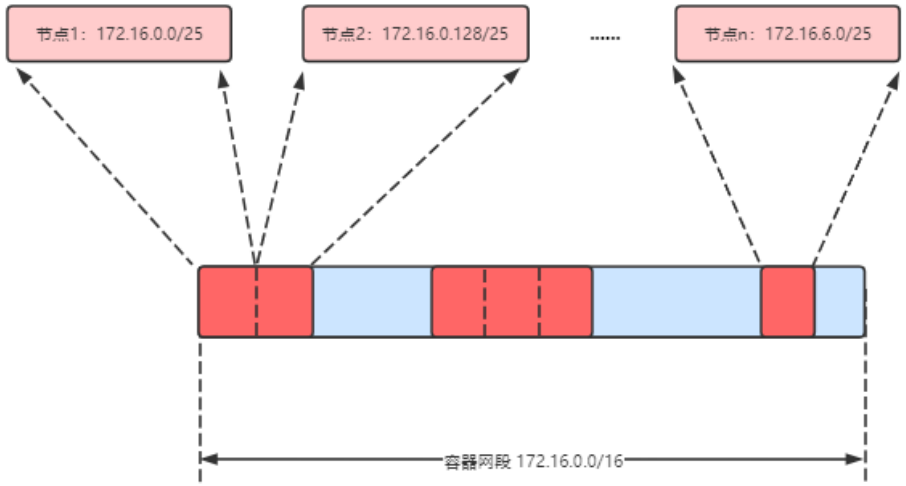
- 中小规模组网：由于VPC路由网络受限于VPC路由表条目配额的限制，当前默认最大只支持200个节点，如果有更大规模的需求，可提升VPC路由表条目配额。

容器 IP 地址管理

VPC网络按如下规则分配容器IP：

- 容器网段需单独分配
- 节点维度划分地址段，集群的所有节点从容器网段中分配一个固定大小（用户自己配置）的IP网段
- 容器网段依次循环分配IP网段给新增节点
- 调度到节点上的Pod依次循环从分配给节点的IP网段内分配IP地址

图 11-7 VPC 网络 IP 地址管理



按如上IP分配，VPC网络的集群最多能创建节点数量 = 容器网段IP数量 ÷ 节点从容器网段中分配IP网段的IP数量

比如容器网段为172.16.0.0/16，则IP数量为65536，节点分配容器网段掩码为25，也就是每个节点容器IP数量为128，则最多可创建节点数量为65536/128=512。另外，集群能创建多少节点，还受节点网络和集群规模的影响。

网段规划建议

在**集群网络构成**中介绍集群中网络地址可分为节点网络、容器网络、服务网络三块，在规划网络地址时需要从如下方面考虑：

- **三个网段不能重叠**，否则会导致冲突。
- 保证**每个网段有足够的IP地址可用**。
  - 节点网段的IP地址要与集群规模相匹配，否则会因为IP地址不足导致无法创建节点。
  - 容器网段的IP地址要与业务规模相匹配，否则会因为IP地址不足导致无法创建Pod。每个节点上可以创建多少Pod还与其他参数设置相关。

例如集群规模为200节点，容器网络模型为VPC网络。

则此时选择节点子网的可用IP数量需要超过200，否则会因为IP地址不足导致无法创建节点。

容器网段为10.0.0.0/16，可用IP数量为65536，如**容器IP地址管理**中所述，VPC网络IP分配是分配固定大小的网段（使用掩码实现，确定每个节点最多分配多少容器IP），例如上限为128，则此时集群最多支撑65536/128=512个节点，然后去掉Master节点数量，最终结果就是509个。

VPC 网络访问示例

创建一个VPC网络的集群。集群有一个Node节点。

```
$ kubectl get node
NAME          STATUS   ROLES    AGE   VERSION
192.168.0.99   Ready    <none>   9d    v1.17.17-r0-CCE21.6.1.B004-17.37.5
```

查看VPC的路由表，会看到如下一条路由，目的地址172.16.0.0/25是分配给节点的容器网段，下一跳指向对应的节点，当访问容器IP时，VPC路由就会转发给下一跳的节点。这印证了前面说的VPC网络模型使用VPC的路由。

在集群中创建一个Deployment。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: example
  namespace: default
spec:
  replicas: 4
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
      imagePullSecrets:
        - name: default-secret
```

然后查看Pod。

```
$ kubectl get pod -owide
NAME                       READY   STATUS    RESTARTS   AGE   IP           NODE          NOMINATED NODE
READINESS GATES
example-86b9779494-l8qrw   1/1     Running   0          14s   172.16.0.6   192.168.0.99   <none>
<none>
example-86b9779494-svs8t   1/1     Running   0          14s   172.16.0.7   192.168.0.99   <none>
<none>
example-86b9779494-x8kl5   1/1     Running   0          14s   172.16.0.5   192.168.0.99   <none>
<none>
example-86b9779494-zt627   1/1     Running   0          14s   172.16.0.8   192.168.0.99   <none>
<none>
```

此时如果使用同一VPC内的云服务器从集群外直接访问Pod的IP，会发现可以访问，这就是VPC网络的特性。在同一VPC下，支持从集群外部通过IP地址直接访问容器。

而在集群内部节点或Pod内，都能正常访问Pod。如下进入到容器中直接访问Pod能够正常访问。

```
$ kubectl exec -it example-86b9779494-l8qrw -- curl 172.16.0.7
<!DOCTYPE html>
<html>
```

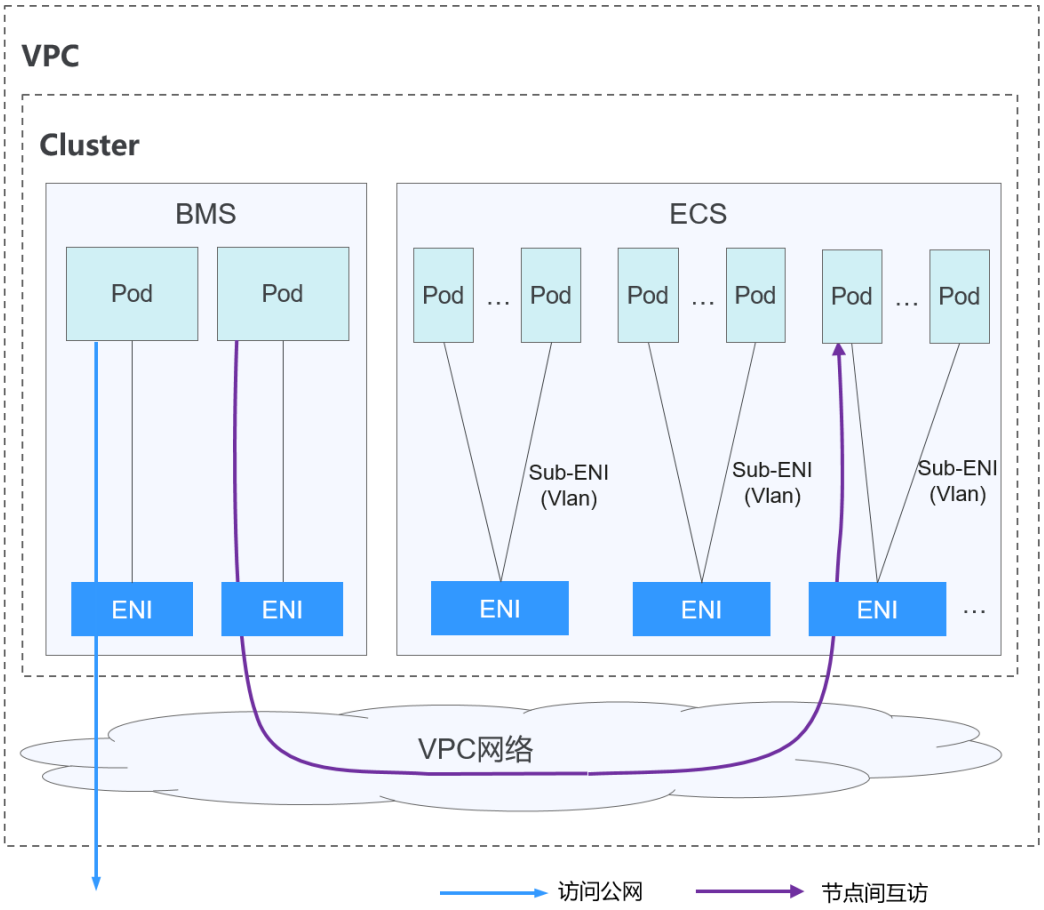
```
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

11.2.4 云原生网络 2.0

云原生网络 2.0 网络模型

云原生网络2.0是自研的新一代容器网络模型，深度整合了虚拟私有云VPC的弹性网卡（Elastic Network Interface，简称ENI）和辅助弹性网卡（Sub Network Interface，简称Sub-ENI）的能力，直接从VPC网段内分配容器IP地址，支持ELB直通容器，绑定安全组，绑定弹性公网IP，享有高性能。

图 11-8 云原生网络 2.0



说明如下：

- 节点内Pod间通信：直接通过VPC的弹性网卡/弹性辅助网卡进行流量转发。
- 跨节点Pod间通信：直接通过VPC的弹性网卡/弹性辅助网卡进行流量转发。

优缺点

优点

- 容器网络直接使用的VPC，网络问题易排查、性能最高。
- 支持VPC内的外部网络与容器IP直通。
- Pod可直接利用VPC提供的负载均衡、安全组、弹性公网IP等能力。

缺点

由于容器网络直接使用的VPC，消耗VPC的地址空间，创建集群前需要合理规划好容器网段。

适用场景

- 性能要求高，需要使用VPC其他网络能力的场景：由于云原生网络2.0直接使用的VPC网络，性能与VPC网络的性能几乎一致，所以适用于对带宽、时延要求极高的业务场景。

- 大规模组网：云原生网络2.0当前最大可支持2000个ECS节点，10万个容器。

网段规划建议

在**集群网络构成**中介绍集群中网络地址可分为节点网络、容器网络、服务网络三块，在规划网络地址时需要从如下方面考虑：

- **三个网段不能重叠**，否则会导致冲突。且集群所在VPC下所有子网（包括扩展网段子网）不能和容器网段、服务网段冲突。
- 保证**每个网段有足够的IP地址可用**。
  - 节点网段的IP地址要与集群规模相匹配，否则会因为IP地址不足导致无法创建节点。
  - 容器网段的IP地址要与业务规模相匹配，否则会因为IP地址不足导致无法创建Pod。

云原生网络2.0模型下，由于容器网段与节点网段共同使用VPC下的网络地址，建议容器子网与节点子网不要使用同一个子网，否则容易出现IP资源不足导致容器或节点创建失败的情况。

另外云原生网络2.0模型下容器网段支持在创建集群后增加子网，扩展可用IP数量，此时需要注意增加的子网不要与容器网段其他子网存在网络冲突。

云原生网络 2.0 访问示例

创建一个CCE Turbo集群，集群包含3个ECS节点。

进入其中一个节点，可以看到节点有一个主网卡和扩展网卡，这两个网卡都是弹性网卡，其中扩展网卡是属于容器网络网段，用于给Pod挂载辅助弹性网卡Sub-ENI。

在集群中创建一个Deployment。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: example
  namespace: default
spec:
  replicas: 6
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
```

创建后查询Pod。

```
$ kubectl get pod -owide
NAME                  READY   STATUS    RESTARTS   AGE   IP          NODE        NOMINATED NODE
```

```
READINESS GATES
example-5bdc5699b7-54v7g   1/1     Running   0          7s    10.1.18.2     10.1.0.167   <none>           <none>
example-5bdc5699b7-6dzx5   1/1     Running   0          7s    10.1.18.216   10.1.0.186   <none>           <none>
example-5bdc5699b7-gq7xs   1/1     Running   0          7s    10.1.16.63    10.1.0.144   <none>           <none>
example-5bdc5699b7-h9rvb   1/1     Running   0          7s    10.1.16.125   10.1.0.167   <none>           <none>
example-5bdc5699b7-s9fts   1/1     Running   0          7s    10.1.16.89    10.1.0.144   <none>           <none>
example-5bdc5699b7-swq6q   1/1     Running   0          7s    10.1.17.111   10.1.0.167   <none>
<none>
```

这里Pod的IP都是Sub-ENI，挂载在节点的ENI上（扩展网卡）。

例如10.1.0.167节点对应的扩展网卡是10.1.17.172。在弹性网卡控制台上可以看到10.1.17.172这块扩展网卡挂载3个Sub-ENI，正是Pod的IP。

在VPC中直接访问Pod的IP，能够正常访问。

11.3 服务（Service）

11.3.1 服务概述

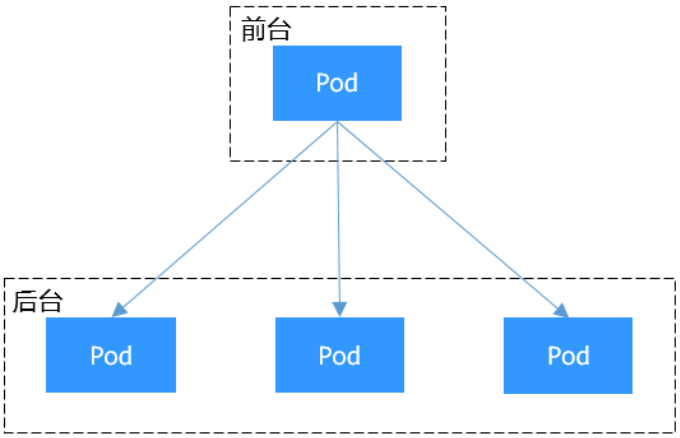
直接访问 Pod 的问题

Pod创建完成后，如何访问Pod呢？直接访问Pod会有如下几个问题：

- Pod会随时被Deployment这样的控制器删除重建，那访问Pod的结果就会变得不可预知。
- Pod的IP地址是在Pod启动后才被分配，在启动前并不知道Pod的IP地址。
- 应用往往都是由多个运行相同镜像的一组Pod组成，逐个访问Pod也变得不现实。

举个例子，假设有这样一个应用程序，使用Deployment创建了前台和后台，前台会调用后台做一些计算处理，如图11-9所示。后台运行了3个Pod，这些Pod是相互独立且可被替换的，当Pod出现状况被重建时，新建的Pod的IP地址是新IP，前台的Pod无法直接感知。

图 11-9 Pod 间访问



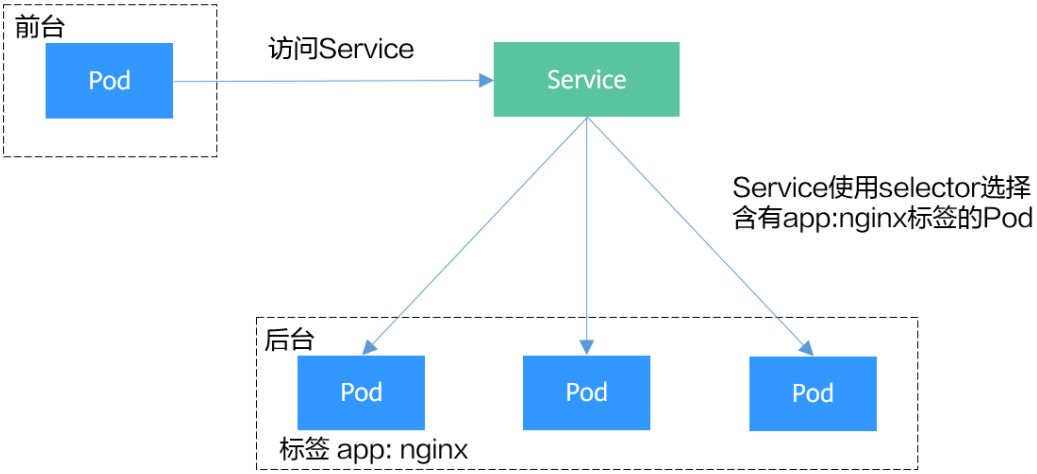
使用 Service 解决 Pod 的访问问题

Kubernetes中的Service对象就是用来解决上述Pod访问问题的。Service有一个固定IP地址（在创建CCE集群时有一个服务网段的设置，这个网段专门用于给Service分配IP地

址），Service将访问它的流量转发给Pod，具体转发给哪些Pod通过Label来选择，而且Service可以给这些Pod做负载均衡。

那么对于上面的例子，为后台添加一个Service，通过Service来访问Pod，这样前台Pod就无需感知后台Pod的变化，如图11-10所示。

图 11-10 通过 Service 访问 Pod



Service 的类型

Kubernetes允许指定一个需要的类型的Service，类型的取值以及行为如下：

- **集群内访问(ClusterIP)**

  集群内访问表示工作负载暴露给同一集群内其他工作负载访问的方式，可以通过“集群内部域名”访问。

- **节点访问(NodePort)**

  节点访问（NodePort）是指在每个节点的IP上开放一个静态端口，通过静态端口对外暴露服务。节点访问（NodePort）会路由到ClusterIP服务，这个ClusterIP服务会自动创建。通过请求<NodeIP>:<NodePort>，可以从集群的外部访问一个NodePort服务。

- **负载均衡(LoadBalancer)**

  负载均衡（LoadBalancer）可以通过弹性负载均衡从公网访问到工作负载，与弹性IP方式相比提供了高可靠的保障。集群外访问推荐使用负载均衡类型。

externalTrafficPolicy（服务亲和）

NodePort类型及LoadBalancer类型的Service接收请求时，会先访问到节点，然后转到Service，再由Service选择一个Pod转发到该Pod，但Service选择的Pod不一定在接收请求的节点上。默认情况下，从任意节点IP+服务端口都能访问到后端工作负载，当Pod不在接收请求的节点上时，请求会再跳转到Pod所在的节点，带来一定性能损失。

Service有一个配置参数externalTrafficPolicy，如下所示。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  externalTrafficPolicy: Local
```

```
ports:
- name: service
  nodePort: 30000
  port: 80
  protocol: TCP
  targetPort: 80
selector:
  app: nginx
type: NodePort
```

当externalTrafficPolicy取值为**Local**时，通过节点IP:服务端口的请求只会转发给本节点上的Pod，如果节点没有Pod的话请求会挂起。

当externalTrafficPolicy取值为**Cluster**时，请求会在集群内转发，从任意节点IP+服务端口都能访问到后端工作负载。

如不设置externalTrafficPolicy，默认取值为**Cluster**。

在CCE 控制台创建NodePort类型Service时也可以配置该参数。

总结externalTrafficPolicy的两个取值：

- Cluster（集群级别）：集群下所有节点的IP+访问端口均可以访问到此服务关联的负载，服务访问会因路由跳转导致一定性能损失，且无法获取到客户端源IP。
- Local（节点级别）：只有通过负载所在节点的IP+访问端口才可以访问此服务关联的负载，服务访问没有因路由跳转导致的性能损失，且可以获取到客户端源IP。该场景下可能出现集群内无法访问Service的情况，详情请参见**集群内无法访问Service的说明**。

集群内无法访问 Service 的说明

当Service设置了服务亲和为节点级别，即externalTrafficPolicy取值为Local时，在使用中可能会碰到从集群内部（节点上或容器中）访问不通的情况，回显类似如下内容：

```
upstream connect error or disconnect/reset before headers. reset reason: connection failure
或：
curl: (7) Failed to connect to 192.168.10.36 port 900: Connection refused
```

在集群中访问ELB地址时出现无法访问的场景较为常见，这是由于Kubernetes在创建Service时，kube-proxy会把ELB的访问地址作为外部IP（即External-IP，如下方回显所示）添加到iptables或IPVS中。如果客户端从集群内部发起访问ELB地址的请求，该地址会被认为是服务的外部IP，被kube-proxy直接转发，而不再经过集群外部的ELB。

```
# kubectl get svc nginx
NAME    TYPE          CLUSTER-IP      EXTERNAL-IP                PORT(S)       AGE
nginx   LoadBalancer  10.247.76.156   123.**.**.**,192.168.0.133  80:32146/TCP  37s
```

当externalTrafficPolicy的取值为Local时，在不同容器网络模型和服务转发模式下访问不通的场景如下：

说明

- 多实例的工作负载需要保证所有实例均可正常访问，否则可能出现概率性访问不通的情况。
- CCE Turbo集群（云原生2.0网络模型）不支持设置服务亲和为节点级别。

服务端发布服务类型	访问类型	客户端请求发起位置	容器隧道集群（IPVS）	VPC集群（IPVS）	容器隧道集群（IPTABLES）	VPC集群（IPTABLES）
节点访问类型 Service	公网/私网	与服务Pod同节点	访问服务端所在节点IP +NodePort — 正常访问 访问非服务端所在节点IP +NodePort — 无法访问	访问服务端所在节点IP +NodePort — 正常访问 访问非服务端所在节点IP +NodePort — 无法访问	访问服务端所在节点IP +NodePort — 正常访问 访问非服务端所在节点IP +NodePort — 无法访问	访问服务端所在节点IP +NodePort — 正常访问 访问非服务端所在节点IP +NodePort — 无法访问
		与服务Pod不同节点	访问服务端所在节点IP +NodePort — 通 访问非服务端所在节点IP +NodePort — 无法访问	访问服务端所在节点IP +NodePort — 通 访问非服务端所在节点IP +NodePort — 无法访问	正常访问	正常访问
		与服务Pod同节点的其他容器	访问服务端所在节点IP +NodePort — 正常访问 访问非服务端所在节点IP +NodePort — 无法访问	无法访问	访问服务端所在节点IP +NodePort — 正常访问 访问非服务端所在节点IP +NodePort — 无法访问	无法访问

服务端发布服务类型	访问类型	客户端请求发起位置	容器隧道集群（IPVS）	VPC集群（IPVS）	容器隧道集群（IPTABLES）	VPC集群（IPTABLES）
		与服务Pod不同节点的其他容器	访问服务端所在节点IP+NodePort — 正常访问 访问非服务端所在节点IP+NodePort — 无法访问	访问服务端所在节点IP+NodePort — 正常访问 访问非服务端所在节点IP+NodePort — 无法访问	访问服务端所在节点IP+NodePort — 正常访问 访问非服务端所在节点IP+NodePort — 无法访问	访问服务端所在节点IP+NodePort — 正常访问 访问非服务端所在节点IP+NodePort — 无法访问
独享型负载均衡类型Service	私网	与服务Pod同节点	无法访问	无法访问	无法访问	无法访问
		与服务Pod同节点的其他容器	无法访问	无法访问	无法访问	无法访问
nginx-ingress插件对接独享型ELB（Local）	私网	与cceaddon-nginx-ingress-controller Pod同节点	无法访问	无法访问	无法访问	无法访问
		与cceaddon-nginx-ingress-controller Pod同节点的其他容器	无法访问	无法访问	无法访问	无法访问

解决这个问题通常有如下办法：

- （**推荐**）在集群内部访问使用Service的ClusterIP或服务域名访问。
- 将Service的externalTrafficPolicy设置为Cluster，即集群级别服务亲和。不过需要注意这会影响源地址保持。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
```

```
    kubernetes.io/elb.class: union
    kubernetes.io/elb.autocreate: '{"type":"public","bandwidth_name":"cce-
bandwidth","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","
eip_type":"5_bgp","name":"james"}'
  labels:
    app: nginx
  name: nginx
spec:
  externalTrafficPolicy: Cluster
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

11.3.2 集群内访问（ClusterIP）

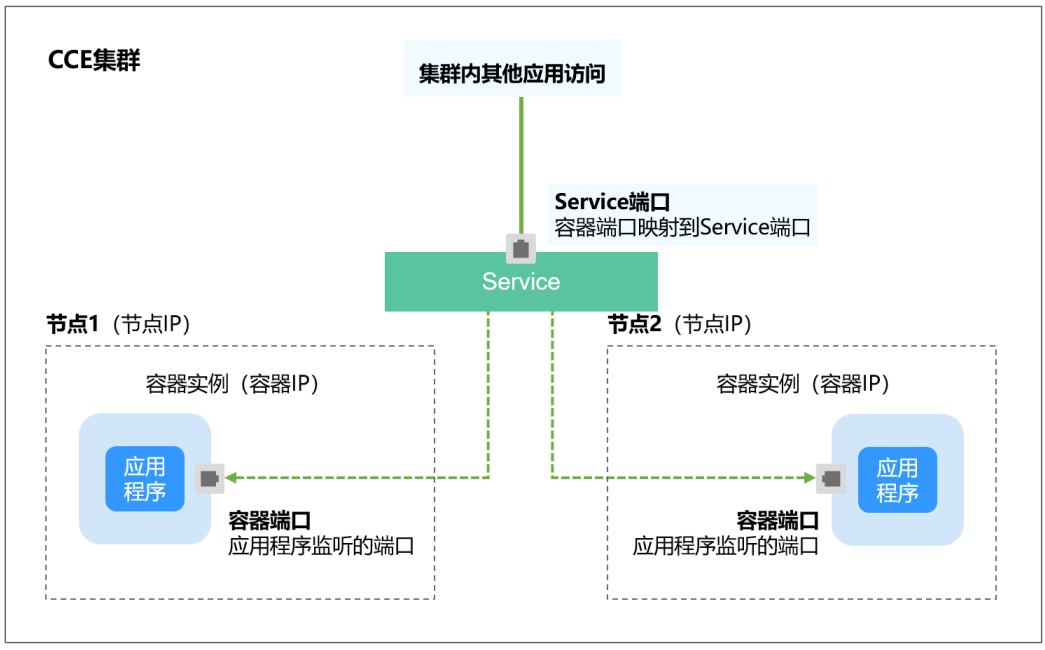
操作场景

集群内访问表示工作负载暴露给同一集群内其他工作负载访问的方式，可以通过“集群内部域名”访问。

集群内部域名格式为“<服务名称>.<工作负载所在命名空间>.svc.cluster.local:<端口号>”，例如“nginx.default.svc.cluster.local:80”。

访问通道、容器端口与访问端口映射如图11-11所示。

图 11-11 集群内访问



创建 ClusterIP 类型 Service

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“服务发现”，在右上角单击“创建服务”。

步骤3 设置集群内访问参数。

- **Service名称：**自定义服务名称，可与工作负载名称保持一致。
- **访问类型**：选择“集群内访问 ClusterIP”。
- **命名空间：**工作负载所在命名空间。
- **选择器：**添加标签，Service根据标签选择Pod，填写后单击“确认添加”。也可以引用已有工作负载的标签，单击“引用负载标签”，在弹出的窗口中选择负载，然后单击“确定”。
- **端口配置：**
  - 协议：请根据业务的协议类型选择。
  - 服务端口：Service使用的端口，端口范围为1-65535。
  - 容器端口：工作负载程序实际监听的端口，需用户确定。例如nginx默认使用80端口。

步骤4 单击“确定”，创建Service。

----结束

通过 kubectl 命令行创建

您可以通过kubectl命令行设置Service访问方式。本节以nginx为例，说明kubectl命令实现集群内访问的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建并编辑nginx-deployment.yaml和nginx-clusterip-svc.yaml文件。

其中，nginx-deployment.yaml和nginx-clusterip-svc.yaml为自定义名称，您可以随意命名。

vi nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

vi nginx-clusterip-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx-clusterip
spec:
  ports:
```

```
  - name: service0
    port: 8080              # 访问Service的端口
    protocol: TCP           # 访问Service的协议，支持TCP和UDP
    targetPort: 80          # Service访问目标容器的端口，此端口与容器中运行的应用强相关，如本例中nginx镜像默认使用80端口
  selector:                 # 标签选择器，Service通过标签选择Pod，将访问Service的流量转发给Pod，此处选择带有 app:nginx 标签的Pod
    app: nginx
  type: ClusterIP           # Service的类型，ClusterIP表示在集群内访问
```

步骤3 创建工作负载。

kubectl create -f nginx-deployment.yaml

回显如下，表示工作负载已经创建。

```
deployment "nginx" created
```

kubectl get po

回显如下，工作负载状态为Running，表示工作负载已处于运行中状态。

```
NAME                     READY     STATUS            RESTARTS   AGE
nginx-2601814895-znhbr   1/1       Running           0          15s
```

步骤4 创建服务。

kubectl create -f nginx-clusterip-svc.yaml

回显如下，表示服务已开始创建。

```
service "nginx-clusterip" created
```

kubectl get svc

回显如下，表示服务已创建成功，CLUSTER-IP已生成。

```
# kubectl get svc
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes      ClusterIP   10.247.0.1     <none>        443/TCP    4d6h
nginx-clusterip   ClusterIP   10.247.74.52   <none>        8080/TCP   14m
```

步骤5 访问Service。

在集群内的容器或节点上都能够访问Service。

创建一个Pod并进入到容器内，使用curl命令访问Service的IP:Port或域名，如下所示。

其中域名后缀可以省略，在同个命名空间内可以直接使用nginx-clusterip:8080访问，跨命名空间可以使用nginx-clusterip.default:8080访问。

```
# kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 10.247.74.52:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

```
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ # curl nginx-clusterip.default.svc.cluster.local:8080
...
<h1>Welcome to nginx!</h1>
...
/ # curl nginx-clusterip.default:8080
...
<h1>Welcome to nginx!</h1>
...
/ # curl nginx-clusterip:8080
...
<h1>Welcome to nginx!</h1>
...
```

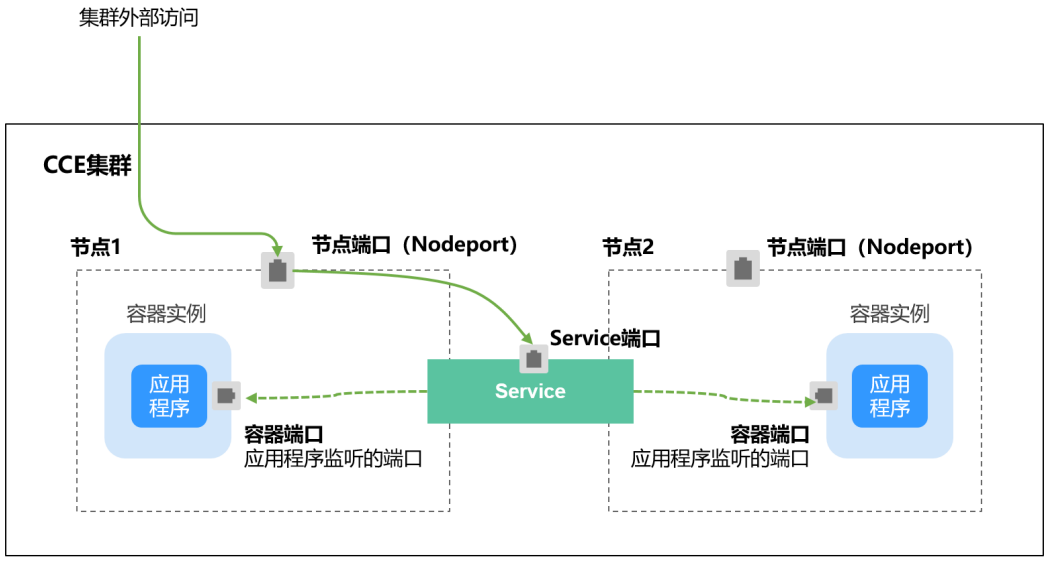
----结束

11.3.3 节点访问（NodePort）

操作场景

节点访问（NodePort）是指在每个节点的IP上开放一个静态端口，通过静态端口对外暴露服务。创建NodePort服务时，Kubernetes会自动创建一个集群内部IP地址（ClusterIP），集群外部的客户端通过访问 <NodeIP>:<NodePort>，流量会通过NodePort服务对应的ClusterIP转发到对应的Pod。

图 11-12 NodePort 访问



约束与限制

- “节点访问（NodePort）”默认为VPC内网访问，如果需要使用弹性IP通过公网访问该服务，请提前在集群的节点上绑定弹性IP。

- 创建Service后，如果服务亲和从集群级别切换为节点级别，连接跟踪表将不会被清理，建议用户创建Service后不要修改服务亲和属性，如需修改请重新创建Service。
- CCE Turbo集群仅支持集群级别服务亲和。
- VPC网络模式下，当某容器A通过NodePort类型服务发布时，且服务亲和设置为节点级别（即externalTrafficPolicy为local），部署在同节点的容器B将无法通过节点IP+NodePort访问容器A。
- v1.21.7及以上的集群创建的NodePort类型服务时，节点上的NodePort端口默认不会用netstat显示：如果集群转发模式为iptables，可使用**iptables -t nat -L**查看端口；如果集群转发模式为IPVS，可使用**ipvsadm -Ln**查看端口。

创建 NodePort 类型 Service

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“服务发现”，在右上角单击“创建服务”。

步骤3 设置集群内访问参数。

- **Service名称：**自定义服务名称，可与工作负载名称保持一致。
- **访问类型**：选择“节点访问 NodePort”。
- **命名空间：**工作负载所在命名空间。
- **服务亲和：**详情请参见**externalTrafficPolicy（服务亲和）**。
  - 集群级别：集群下所有节点的IP+节点端口均可以访问到此服务关联的负载，服务访问会因路由跳转导致一定性能损失，且无法获取到客户端源IP。
  - 节点级别：只有通过负载所在节点的IP+节点端口才可以访问此服务关联的负载，服务访问没有因路由跳转导致的性能损失，且可以获取到客户端源IP。
- **选择器：**添加标签，Service根据标签选择Pod，填写后单击“确认添加”。也可以引用已有工作负载的标签，单击“引用负载标签”，在弹出的窗口中选择负载，然后单击“确定”。
- **端口配置：**
  - 协议：请根据业务的协议类型选择。
  - 服务端口：Service使用的端口，端口范围为1-65535。
  - 容器端口：工作负载程序实际监听的端口，需用户确定。例如nginx默认使用80端口。
  - 节点端口：即NodePort，建议选择“自动生成”；也可以指定端口，默认范围为30000-32767。

步骤4 单击“确定”，创建Service。

----结束

kubectl 命令行创建

您可以通过kubectl命令行设置Service访问方式。本节以nginx为例，说明kubectl命令实现节点访问的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建并编辑nginx-deployment.yaml以及nginx-nodeport-svc.yaml文件。

其中，nginx-deployment.yaml和nginx-nodeport-svc.yaml为自定义名称，您可以随意命名。

vi nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

vi nginx-nodeport-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx-nodeport
spec:
  ports:
  - name: service
    nodePort: 30000     # 节点端口，取值范围为30000-32767
    port: 8080          # 访问Service的端口
    protocol: TCP       # 访问Service的协议，支持TCP和UDP
    targetPort: 80      # Service访问目标容器的端口，此端口与容器中运行的应用强相关，如本例中nginx镜像默认使用80端口
  selector:             # 标签选择器，Service通过标签选择Pod，将访问Service的流量转发给Pod，此处选择带有app:nginx 标签的Pod
    app: nginx
  type: NodePort        # Service的类型，NodePort表示在通过节点端口访问
```

步骤3 创建工作负载。

kubectl create -f nginx-deployment.yaml

回显如下，表示工作负载已创建完成。

```
deployment "nginx" created
```

kubectl get po

回显如下，工作负载状态为Running，表示工作负载已处于运行状态。

```
NAME                     READY     STATUS            RESTARTS   AGE
nginx-2601814895-qhxqv   1/1       Running           0          9s
```

步骤4 创建服务。

kubectl create -f nginx-nodeport-svc.yaml

回显如下，表示服务开始创建。

```
service "nginx-nodeport" created
```

kubectl get svc

回显如下，表示服务已创建完成。

```
# kubectl get svc
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes       ClusterIP   10.247.0.1     <none>        443/TCP          4d8h
nginx-nodeport   NodePort    10.247.30.40   <none>        8080:30000/TCP   18s
```

步骤5 访问Service。

默认情况下，NodePort类型Service可以通过**任意节点IP:节点端口**访问。

在集群同VPC下或集群容器内都可以访问，如果给节点绑定公网IP，也可以使用公网IP访问。如下所示，在集群上创建一个容器，从容器中使用节点IP:节点端口访问。

```
# kubectl get node -owide
NAME          STATUS   ROLES    AGE    INTERNAL-IP    EXTERNAL-IP   OS-IMAGE               KERNEL-
VERSION               CONTAINER-RUNTIME
10.100.0.136   Ready    <none>   152m   10.100.0.136   <none>        CentOS Linux 7 (Core)
3.10.0-1160.25.1.el7.x86_64   docker://18.9.0
10.100.0.5     Ready    <none>   152m   10.100.0.5     <none>        CentOS Linux 7 (Core)
3.10.0-1160.25.1.el7.x86_64   docker://18.9.0
# kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 10.100.0.136:30000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ #
```

----结束

11.3.4 负载均衡（LoadBalancer）

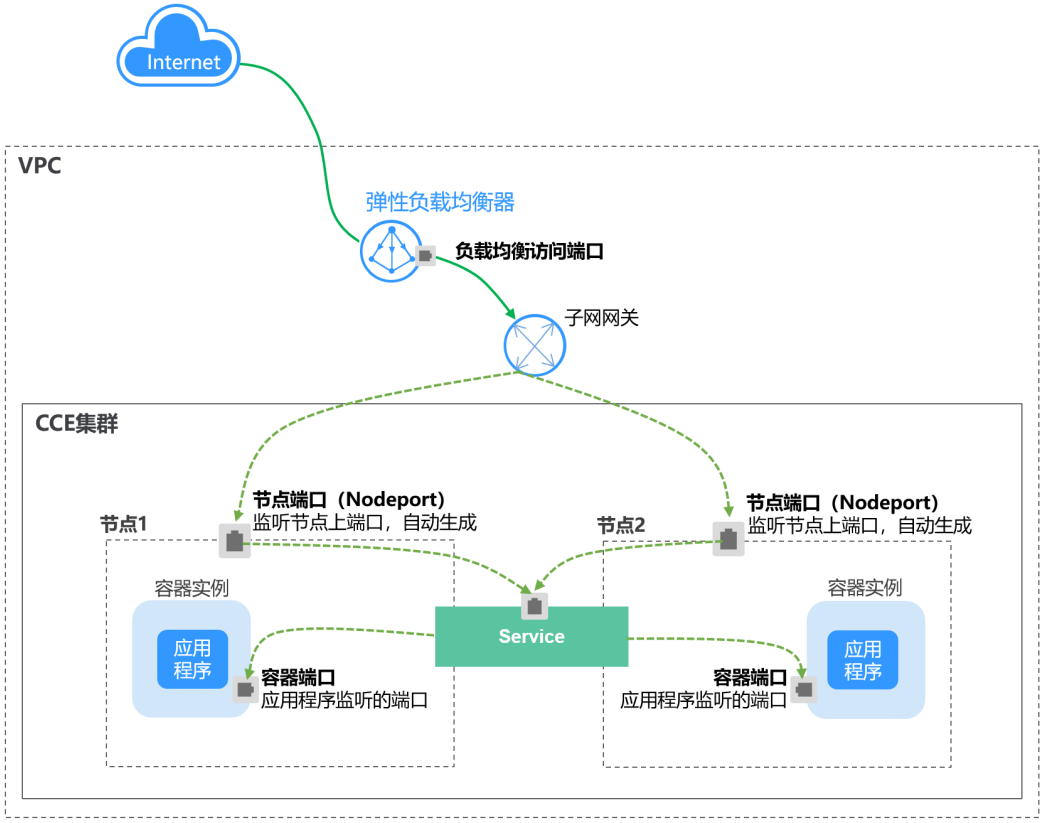
11.3.4.1 创建负载均衡类型的服务

操作场景

负载均衡（LoadBalancer）类型的服务可以通过弹性负载均衡（ELB）从公网访问到工作负载，与弹性IP方式相比提供了高可靠的保障。负载均衡访问方式由公网弹性负载均衡服务地址以及设置的访问端口组成，例如“10.117.117.117:80”。

在访问负载均衡类型的服务时，从ELB过来的流量会先访问到节点，然后通过Service转发到Pod。

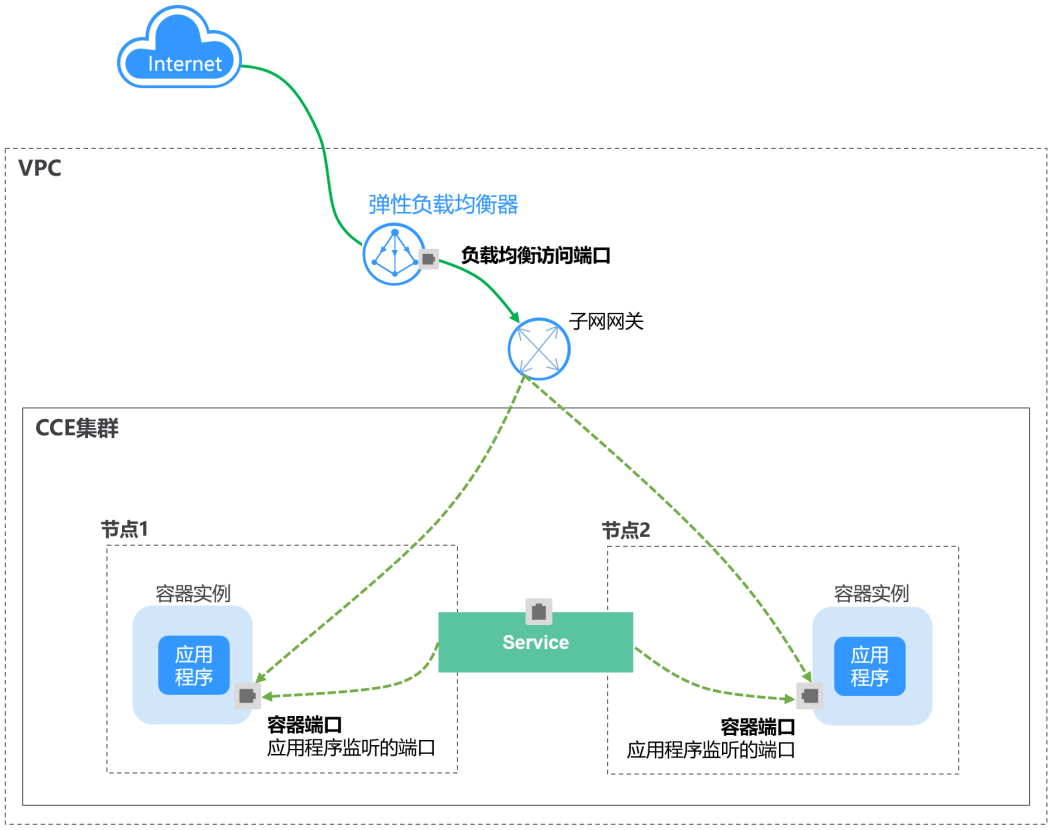
图 11-13 负载均衡（LoadBalancer）



在使用**CCE Turbo集群 + 独享型ELB实例**时，支持ELB直通Pod，使部署在容器中的业务时延降低、性能无损耗。

从集群外部访问时，从ELB直接转发到Pod；集群内部访问可通过Service转发到Pod。

图 11-14 ELB 直通容器



约束与限制

- CCE中的负载均衡（LoadBalancer）访问类型使用弹性负载均衡 ELB提供网络访问，存在如下产品约束：
  - 自动创建的ELB实例建议不要被其他资源使用，否则会在删除时被占用，导致资源残留。
  - v1.15及之前版本集群使用的ELB实例请不要修改监听器名称，否则可能导致无法正常访问。
- 创建Service后，如果服务亲和从集群级别切换为节点级别，连接跟踪表将不会被清理，建议用户创建Service后不要修改服务亲和属性，如需修改请重新创建Service。
- 当服务亲和设置为节点级别（即**externalTrafficPolicy**为local）时，集群内部可能使用ELB地址访问不通，具体情况请参见**集群内无法访问Service的说明**。
- CCE Turbo集群仅支持集群级别服务亲和。
- 独享型ELB仅支持1.17及以上集群。
- 独享型ELB规格必须支持网络型（TCP/UDP），且网络类型必须支持私网（有私有IP地址）。如果需要Service支持HTTP，则独享型ELB规格需要为网络型（TCP/UDP）和应用型（HTTP/HTTPS）。
- 使用CCE集群时，如果LoadBalancer类型Service的服务亲和类型为集群级别（Cluster），当请求进入到集群时，会使用SNAT分发到各个节点的节点端口（nodeport），不能超过节点可用的nodeport数量，而服务亲和为节点级别（Local）则无此约束。使用CCE Turbo集群时，如果是使用共享型ELB依然有此约束，而独享型ELB无此约束，建议使用CCE Turbo时配合使用独享型ELB。

- 集群服务转发模式为IPVS时，不支持配置节点的IP作为Service的externalIP，会导致节点不可用。
- IPVS模式集群下，Ingress和Service使用相同ELB实例时，无法在集群内的节点和容器中访问Ingress，因为kube-proxy会在ipvs-0的网桥上挂载LB类型的Service地址，Ingress对接的ELB的流量会被ipvs-0网桥劫持。建议Ingress和Service使用不同ELB实例。

创建 LoadBalancer 类型 Service

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“服务发现”，在右上角单击“创建服务”。

步骤3 设置参数。

- **Service名称：**自定义服务名称，可与工作负载名称保持一致。
- **访问类型**：选择“负载均衡 LoadBalancer”。
- **命名空间：**工作负载所在命名空间。
- **服务亲和：**详情请参见**externalTrafficPolicy（服务亲和）**。
  - 集群级别：集群下所有节点的IP+访问端口均可以访问到此服务关联的负载，服务访问会因路由跳转导致一定性能损失，且无法获取到客户端源IP。
  - 节点级别：只有通过负载所在节点的IP+访问端口才可以访问此服务关联的负载，服务访问没有因路由跳转导致的性能损失，且可以获取到客户端源IP。
- **选择器：**添加标签，Service根据标签选择Pod，填写后单击“确认添加”。也可以引用已有工作负载的标签，单击“引用负载标签”，在弹出的窗口中选择负载，然后单击“确定”。
- **负载均衡器：**

  选择对接的ELB实例，仅支持与集群在同一个VPC下的ELB实例。如果没有可选的ELB实例，请单击“创建负载均衡器”跳转到ELB控制台创建。

  CCE控制台支持自动创建ELB实例，在下拉框选择“自动创建”，并填写以下参数：
  - 实例名称：请填写ELB名称。
  - 公网访问：开启后，将创建 5 Mbit/s 带宽的弹性公网 IP。
  - 可用区、子网和规格（仅独享型ELB实例支持选择）：设置独享型ELB的可用区、子网和规格。当前仅支持自动创建网络型（TCP/UDP）独享型ELB实例。

  您可以单击负载均衡配置的“编辑”图标配置ELB实例的参数，在弹出窗口中配置ELB实例的参数。
  - 分配策略：可选择加权轮询算法、加权最少连接或源IP算法。

说明

- 加权轮询算法：根据后端服务器的权重，按顺序依次将请求分发给不同的服务器。它用相应的权重表示服务器的处理性能，按照权重的高低以及轮询方式将请求分配给各服务器，相同权重的服务器处理相同数目的连接数。常用于短连接服务，例如HTTP等服务。
- 加权最少连接：最少连接是通过当前活跃的连接数来估计服务器负载情况的一种动态调度算法。加权最少连接就是在最少连接数的基础上，根据服务器的不同处理能力，给每个服务器分配不同的权重，使其能够接受相应权值数的服务请求。常用于长连接服务，例如数据库连接等服务。
- 源IP算法：将请求的源IP地址进行Hash运算，得到一个具体的数值，同时对后端服务器进行编号，按照运算结果将请求分发到对应编号的服务器上。这可以使得对不同源IP的访问进行负载分发，同时使得同一个客户端IP的请求始终被派发至某特定的服务器。该方式适合负载均衡无cookie功能的TCP协议。

- 会话保持类型：默认不启用，可选择“源IP地址”。基于源IP地址的简单会话保持，即来自同一IP地址的访问请求转发到同一台后端服务器上。

说明

当**分配策略**使用源IP算法时，不支持设置会话保持。

- **健康检查**：设置负载均衡的健康检查配置。
  - 全局检查：全局检查仅支持使用相同协议的端口，无法对多个使用不同协议的端口生效，建议使用“自定义检查”。
  - 自定义检查：在**端口配置**中对多种不同协议的端口设置健康检查。关于自定义检查的YAML定义，请参见**指定多个端口配置健康检查**。

表 11-2 健康检查参数

参数	说明
协议	当**端口配置**协议为TCP时，支持TCP和HTTP协议；当**端口配置**协议为UDP时，支持UDP协议。 – 检查路径（仅HTTP健康检查协议支持）：指定健康检查的URL地址。检查路径只能以/开头，长度范围为1-80。
端口	健康检查默认使用业务端口（Service的NodePort和容器端口）作为健康检查的端口；您也可以重新指定端口用于健康检查，重新指定端口会为服务增加一个名为cce-healthz的服务端口配置。 – 节点端口：使用共享型负载均衡或不关联ENI实例时，节点端口作为健康检查的检查端口；如不指定将随机一个端口。取值范围为30000-32767。 – 容器端口：使用独享型负载均衡关联ENI实例时，容器端口作为健康检查的检查端口。取值范围为1-65535。
检查周期（秒）	每次健康检查响应的最大间隔时间，取值范围为1-50。
超时时间（秒）	每次健康检查响应的最大超时时间，取值范围为1-50。
最大重试次数	健康检查最大的重试次数，取值范围为1-10。

- **端口配置：**
  - 协议：请根据业务的协议类型选择。
  - 服务端口：Service使用的端口，端口范围为1-65535。
  - 容器端口：工作负载程序实际监听的端口，需用户确定。例如nginx默认使用80端口。
  - 健康检查：**健康检查**选项设置为“自定义检查”时，可以为不同协议的端口配置健康检查，参数说明请参见**表11-2**。

  说明

  在创建LoadBalancer类型Service时，会自动生成一个随机节点端口号（NodePort）。
- **注解**：LoadBalancer类型Service有一些CCE定制的高级功能，通过注解annotations实现，具体注解的内容请参见**使用Annotation配置负载均衡**。

步骤4 单击“确定”，创建Service。

----结束

通过 kubectl 命令行创建-使用已有 ELB

您可以在创建工作负载时通过kubectl命令行设置Service访问方式。本节以nginx为例，说明kubectl命令实现负载均衡（LoadBalancer）访问的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建并编辑nginx-deployment.yaml以及nginx-elb-svc.yaml文件。

其中，nginx-deployment.yaml和nginx-elb-svc.yaml为自定义名称，您可以随意命名。

vi nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

vi nginx-elb-svc.yaml

说明

若需要开启会话保持，需要满足如下条件：

- 工作负载协议为TCP。
- 工作负载的各实例已设置反亲和部署，即所有的实例都部署在不同节点上。具体请参见**调度策略（亲和与反亲和）**。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: <your_elb_id>                    # ELB ID，替换为实际值
    kubernetes.io/elb.class: union                 # 负载均衡器类型
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN                 # 负载均衡器算法
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP          # 会话保持类型为源IP
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout": "30"}'     # 会话保持时间（分钟）
    kubernetes.io/elb.health-check-flag: 'on'                   # 开启ELB健康检查功能
    kubernetes.io/elb.health-check-option: '{
      "protocol":"TCP",
      "delay":"5",
      "timeout":"10",
      "max_retries":"3"
    }'
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    port: 80     #访问Service的端口，也是负载均衡上的监听器端口。
    protocol: TCP
    targetPort: 80  #Service访问目标容器的端口，此端口与容器中运行的应用强相关
    nodePort: 31128  #节点的端口号，如不指定，将在30000-32767范围内随机生成一个端口号
  type: LoadBalancer
```

上述示例通过Annotation（注解）实现负载均衡的一些高级功能，例如会话保持、健康检查等，对应的说明请参见**表11-3**。

除本示例中的功能外，如需了解更多高级功能相关注解及示例，请参见**使用Annotation配置负载均衡**。

表 11-3 annotations 参数

参数	是否必填	参数类型	描述
kubernetes.io/elb.id	是	String	为负载均衡实例的ID。 在关联已有ELB时：必填。 **获取方法：** 在控制台的“服务列表”中，单击“网络 > 弹性负载均衡 ELB”，单击ELB的名称，在ELB详情页的“基本信息”页签下找到“ID”字段复制即可。 **说明** 系统优先根据kubernetes.io/elb.id注解对接ELB，若此字段未指定，则会根据spec.loadBalancerIP字段（非必填，且仅1.23及以前版本可用）对接ELB。 请尽量不要使用spec.loadBalancerIP字段对接ELB，该字段在将来的集群版本中会被Kubernetes官方废弃，详情请参见**Deprecation**。
kubernetes.io/elb.class	是	String	请根据不同的应用场景和功能需求选择合适的负载均衡器类型。 ● union：共享型负载均衡。 ● performance：独享型负载均衡，仅支持1.17及以上集群。 **说明** 负载均衡类型的服务对接已有的独享型ELB时，该独享型ELB必须支持网络型（TCP/UDP）规格。
kubernetes.io/elb.lb-algorithm	否	String	后端云服务器组的负载均衡算法，默认值为“ROUND_ROBIN”。 取值范围： ● ROUND_ROBIN：加权轮询算法。 ● LEAST_CONNECTIONS：加权最少连接算法。 ● SOURCE_IP：源IP算法。 **说明** 当该字段的取值为SOURCE_IP时，后端云服务器组绑定的后端云服务器的权重设置（weight字段）无效，且不支持开启会话保持。

参数	是否必填	参数类型	描述
kubernetes.io/elb.session-affinity-mode	否	String	支持基于源IP地址的简单会话保持，即来自同一IP地址的访问请求转发到同一台后端服务器上。 • 不启用：不填写该参数。 • 开启会话保持：需增加该参数，取值“SOURCE_IP”，表示基于源IP地址。 **说明** 当kubernetes.io/elb.lb-algorithm设置为“SOURCE_IP”（源IP算法）时，不支持开启会话保持。
kubernetes.io/elb.session-affinity-option	否	表11-4 Object	ELB会话保持配置选项，可设置会话保持的超时时间。
kubernetes.io/elb.health-check-flag	否	String	是否开启ELB健康检查功能。 • 开启：空值或"on" • 关闭："off" 开启时需同时填写**kubernetes.io/elb.health-check-option**字段。
kubernetes.io/elb.health-check-option	否	表11-5 Object	ELB健康检查配置选项。

表 11-4 elb.session-affinity-option 字段数据结构说明

参数	是否必填	参数类型	描述
persistence_timeout	是	String	当elb.session-affinity-mode是“SOURCE_IP”时生效，设置会话保持的超时时间（分钟）。 默认值为："60"，取值范围：1-60。

表 11-5 elb.health-check-option 字段数据结构说明

参数	是否必填	参数类型	描述
delay	否	String	开始健康检查的初始等待时间（秒）。 默认值：5，取值范围：1-50

参数	是否必填	参数类型	描述
timeout	否	String	健康检查的超时时间（秒）。 默认值：10，取值范围1-50
max_retries	否	String	健康检查的最大重试次数。 默认值：3，取值范围1-10
protocol	否	String	健康检查的协议。 取值范围：“TCP”或者“HTTP”
path	否	String	健康检查的URL，协议是“HTTP”时配置。 默认值：“/” 取值范围：1-10000字符

步骤3 创建工作负载。

kubectl create -f nginx-deployment.yaml

回显如下，表示工作负载已创建完成。

```
deployment/nginx created
```

kubectl get pod

回显如下，工作负载状态为Running状态，表示工作负载已运行中。

```
NAME                     READY   STATUS          RESTARTS   AGE
nginx-2601814895-c1xhw   1/1     Running         0          6s
```

步骤4 创建服务。

kubectl create -f nginx-elb-svc.yaml

回显如下，表示服务已创建。

```
service/nginx created
```

kubectl get svc

回显如下，表示工作负载访问方式已设置成功，工作负载可访问。

```
NAME         TYPE          CLUSTER-IP       EXTERNAL-IP    PORT(S)        AGE
kubernetes   ClusterIP     10.247.0.1       <none>         443/TCP        3d
nginx        LoadBalancer  10.247.130.196   10.78.42.242   80:31540/TCP   51s
```

步骤5 在浏览器中输入访问地址，例如输入10.78.42.242:80。10.78.42.242为负载均衡实例IP地址，80为对应界面上的访问端口。

可成功访问nginx。

图 11-15 通过负载均衡访问 nginx

----结束

通过 kubectl 命令行创建-自动创建 ELB

您可以在创建工作负载时通过kubectl命令行设置Service访问方式。本节以nginx为例，说明kubectl命令实现负载均衡（ LoadBalancer)访问的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建并编辑nginx-deployment.yaml以及nginx-elb-svc.yaml文件。

其中，nginx-deployment.yaml和nginx-elb-svc.yaml为自定义名称，您可以随意命名。

vi nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

vi nginx-elb-svc.yaml

说明

若需要开启会话保持，需要满足如下条件：

- 工作负载协议为TCP。
- 工作负载的各实例已设置反亲和部署，即所有的实例都部署在不同节点上。具体请参见**调度策略（亲和与反亲和）**。

共享型负载均衡（公网访问）Service示例：

```
apiVersion: v1
kind: Service
```

```
metadata:
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.autocreate: '{
      "type": "public",
      "bandwidth_name": "cce-bandwidth-1551163379627",
      "bandwidth_chargemode": "bandwidth",
      "bandwidth_size": 5,
      "bandwidth_sharetype": "PER",
      "eip_type": "5_bgp"
    }'
    kubernetes.io/elb.enterpriseID: '0'       # 负载均衡所属企业项目ID
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN                 # 负载均衡器算法
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP          # 会话保持类型为源IP
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout": "30"}'     # 会话保持时间（分钟）
    kubernetes.io/elb.health-check-flag: 'on'                   # 开启ELB健康检查功能
    kubernetes.io/elb.health-check-option: '{
      "protocol":"TCP",
      "delay":"5",
      "timeout":"10",
      "max_retries":"3"
    }'
  labels:
    app: nginx
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

独享型负载均衡（公网访问）Service示例 - 仅支持1.17及以上集群：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.autocreate: '{
      "type": "public",
      "bandwidth_name": "cce-bandwidth-1626694478577",
      "bandwidth_chargemode": "bandwidth",
      "bandwidth_size": 5,
      "bandwidth_sharetype": "PER",
      "eip_type": "5_bgp",
      "available_zone": [
        ""
      ],
      "l4_flavor_name": "L4_flavor.elb.s1.small"
    }'
    kubernetes.io/elb.enterpriseID: '0'       # 负载均衡所属企业项目ID
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN                 # 负载均衡器算法
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP          # 会话保持类型为源IP
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout": "30"}'     # 会话保持时间（分钟）
    kubernetes.io/elb.health-check-flag: 'on'                   # 开启ELB健康检查功能
    kubernetes.io/elb.health-check-option: '{
      "protocol":"TCP",
      "delay":"5",
      "timeout":"10",
      "max_retries":"3"
    }'
spec:
  selector:
```

```
  app: nginx
ports:
- name: cce-service-0
  targetPort: 80
  nodePort: 0
  port: 80
  protocol: TCP
type: LoadBalancer
```

上述示例通过Annotation（注解）实现负载均衡的一些高级功能，例如会话保持、健康检查等，对应的说明请参见**表11-6**。

除本示例中的功能外，如需了解更多高级功能相关注解及示例，请参见**使用Annotation配置负载均衡**。

表 11-6 annotations 参数

参数	是否必填	参数类型	描述
kubernetes.io/elb.class	是	String	请根据不同的应用场景和功能需求选择合适的负载均衡器类型。 • union：共享型负载均衡。 • performance：独享型负载均衡，仅支持1.17及以上集群。
kubernetes.io/elb.autocreate	是	**elb.autocreate** object	自动创建service关联的ELB **示例：** • 公网自动创建： 值为 '{"type":"public","bandwidth_name":"cce-bandwidth-1551163379627","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_type":"5_bgp","name":"james"}' • 私网自动创建： 值为 '{"type":"inner", "name": "A-location-d-test"}'
kubernetes.io/elb.subnet-id	-	String	为集群所在子网的ID，取值范围：1-100字符。 • Kubernetes v1.11.7-r0及以下版本的集群自动创建时为必填参数。 • Kubernetes v1.11.7-r0以上版本的集群：可不填。

参数	是否必填	参数类型	描述
kubernetes.io/elb.enterpriseID	否	String	**v1.15及以上版本的集群支持此字段，v1.15以下版本默认创建到default项目下。** 为ELB企业项目ID，选择后可以直接创建在具体的ELB企业项目下。 该字段不传（或传为字符串'0'），则将资源绑定给默认企业项目。 **获取方法：** 登录企业项目管理控制台，在左侧导航栏中选择“项目管理”，在企业项目列表中单击要加入的企业项目名称，进入企业项目详情页，找到“ID”字段复制即可。
kubernetes.io/elb.lb-algorithm	否	String	后端云服务器组的负载均衡算法，默认值为“ROUND_ROBIN”。 取值范围： • ROUND_ROBIN：加权轮询算法。 • LEAST_CONNECTIONS：加权最少连接算法。 • SOURCE_IP：源IP算法。 **说明** 当该字段的取值为SOURCE_IP时，后端云服务器组绑定的后端云服务器的权重设置（weight字段）无效，且不支持开启会话保持。
kubernetes.io/elb.session-affinity-mode	否	String	支持基于源IP地址的简单会话保持，即来自同一IP地址的访问请求转发到同一台后端服务器上。 • 不启用：不填写该参数。 • 开启会话保持：需增加该参数，取值“SOURCE_IP”，表示基于源IP地址。 **说明** 当kubernetes.io/elb.lb-algorithm设置为“SOURCE_IP”（源IP算法）时，不支持开启会话保持。
kubernetes.io/elb.session-affinity-option	否	表11-4 Object	ELB会话保持配置选项，可设置会话保持的超时时间。

参数	是否必填	参数类型	描述
kubernetes.io/elb.health-check-flag	否	String	是否开启ELB健康检查功能。 ● 开启：“（空值）”或“on” ● 关闭：“off” 开启时需同时填写**kubernetes.io/elb.health-check-option**字段。
kubernetes.io/elb.health-check-option	否	**表11-5** Object	ELB健康检查配置选项。

表 11-7 elb.autocreate 字段数据结构说明

参数	是否必填	参数类型	描述
name	否	String	自动创建的负载均衡的名称。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。 默认名称：cce-lb+service.UID
type	否	String	负载均衡实例网络类型，公网或者私网。 ● public：公网型负载均衡 ● inner：私网型负载均衡 默认类型：inner
bandwidth_name	公网型负载均衡必填	String	带宽的名称，默认值为：cce-bandwidth-******。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。
bandwidth_chargemode	否	String	带宽模式。 ● bandwidth：按带宽 ● traffic：按流量 默认类型：bandwidth

参数	是否必填	参数类型	描述
bandwidth_size	公网型负载均衡必填	Integer	带宽大小，默认1Mbit/s~2000Mbit/s，请根据Region带宽支持范围设置。 调整带宽时的最小单位会根据带宽范围不同存在差异。 ● 小于等于300Mbit/s：默认最小单位为1Mbit/s。 ● 300Mbit/s~1000Mbit/s：默认最小单位为50Mbit/s。 ● 大于1000Mbit/s：默认最小单位为500Mbit/s。
bandwidth_sharetype	公网型负载均衡必填	String	带宽共享方式。 ● PER：独享带宽
eip_type	公网型负载均衡必填	String	弹性公网IP类型。 ● 5_bgp：全动态BGP 具体类型以各区域配置为准，详情请参见弹性公网IP控制台。
available_zone	是	Array of strings	负载均衡所在可用区。 独享型负载均衡器独有字段。
l4_flavor_name	是	String	四层负载均衡实例规格名称。 独享型负载均衡器独有字段。
l7_flavor_name	否	String	七层负载均衡实例规格名称。 独享型负载均衡器独有字段，必须与l4_flavor_name对应规格的类型一致，即都为弹性规格或都为固定规格。
elb_virsubnet_ids	否	Array of strings	负载均衡后端所在子网，不填默认集群子网。不同实例规格将占用不同数量子网IP，不建议使用其他资源（如集群，节点等）的子网网段。 独享型负载均衡器独有字段。 示例： `"elb_virsubnet_ids": [ "14567f27-8ae4-42b8-ae47-9f847a4690dd" ]`

步骤3 创建工作负载。

kubectl create -f nginx-deployment.yaml

回显如下，表示工作负载已开始创建。

```
deployment/nginx created
```

kubectl get pod

回显如下，工作负载状态为Running状态，表示工作负载已运行中。

```
NAME                READY   STATUS    RESTARTS   AGE
nginx-2601814895-c1xhw   1/1     Running   0          6s
```

步骤4 创建服务。

kubectl create -f nginx-elb-svc.yaml

回显如下，表示服务已创建。

```
service/nginx created
```

kubectl get svc

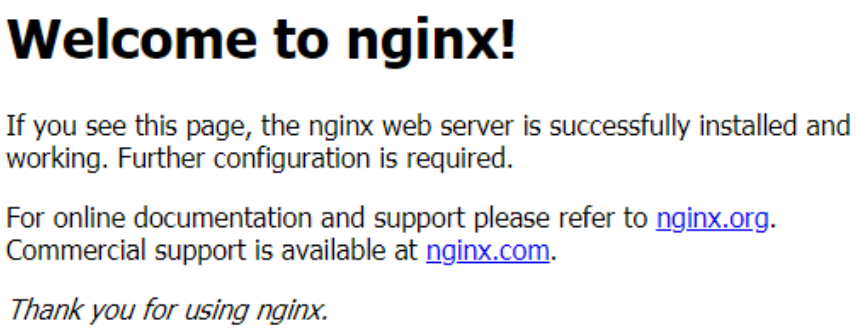
回显如下，表示工作负载访问方式已设置成功，工作负载可访问。

```
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)        AGE
kubernetes   ClusterIP      10.247.0.1       <none>          443/TCP        3d
nginx        LoadBalancer   10.247.130.196   10.78.42.242    80:31540/TCP   51s
```

步骤5 在浏览器中输入访问地址，例如输入10.XXX.XXX.XXX:80。10.XXX.XXX.XXX为负载均衡实例IP地址，80为对应界面上的访问端口。

可成功访问nginx。

图 11-16 通过负载均衡访问 nginx



----结束

11.3.4.2 使用 Annotation 配置负载均衡

通过在YAML中添加注解Annotation（注解），您可以实现CCE提供的一些高级功能。本文介绍在创建LoadBalancer类型的Service时可供使用的Annotation。

- **对接ELB**
- **会话保持**
- **健康检查**
- **使用HTTP协议**
- **动态调整后端云服务器权重**
- **pass-through能力**
- **主机网络**
- **设置超时时间**

对接 ELB

表 11-8 对接 ELB 注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.class	String	请根据不同的应用场景和功能需求选择合适的负载均衡器类型。 • union：共享型负载均衡。 • performance：独享型负载均衡，仅支持1.17及以上集群。	v1.9及以上
kubernetes.io/elb.id	String	**仅关联已有ELB的场景**：必填。 为负载均衡实例的ID。 **获取方法：** 在控制台的“服务列表”中，单击“网络 > 弹性负载均衡 ELB”，单击ELB的名称，在ELB详情页的“基本信息”页签下找到“ID”字段复制即可。 **说明** 系统优先根据kubernetes.io/elb.id注解对接ELB，若此字段未指定，则会根据spec.loadBalancerIP字段（非必填，且仅1.23及以前版本可用）对接ELB。 请尽量不要使用spec.loadBalancerIP字段对接ELB，该字段在将来的集群版本中会被Kubernetes官方废弃，详情请参见Deprecation。	v1.9及以上
kubernetes.io/elb.autocreate	表 11-16	**仅自动创建ELB的场景**：必填。 **示例：** • 公网自动创建： 值为 '{"type":"public","bandwidth_name":"cce-bandwidth-1551163379627","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_type":"5_bgp","name":"james"}' • 私网自动创建： 值为 '{"type":"inner", "name": "A-location-d-test"}'	v1.9及以上

参数	类型	描述	支持的集群版本
kubernetes.io/elb.enterpriseID	String	**仅自动创建ELB的场景**：选填。 **v1.15及以上版本的集群支持此字段，v1.15以下版本默认创建到default项目下。** 为ELB企业项目ID，选择后可以直接创建在具体的ELB企业项目下。 该字段不传（或传为字符串'0'），则将资源绑定给默认企业项目。 **获取方法：** 登录企业项目管理控制台，在左侧导航栏中选择“项目管理”，在企业项目列表中单击要加入的企业项目名称，进入企业项目详情页，找到“ID”字段复制即可。	v1.15及以上
kubernetes.io/elb.subnet-id	String	**仅自动创建ELB的场景**：选填。 为集群所在子网的ID，取值范围：1-100字符。 ● Kubernetes v1.11.7-r0及以下版本的集群自动创建时：必填 ● Kubernetes v1.11.7-r0以上版本的集群：可不填。	v1.11.7-r0以下必填 v1.11.7-r0以上该字段废弃
kubernetes.io/elb.lb-algorithm	String	后端云服务器组的负载均衡算法，默认值为“ROUND_ROBIN”。 取值范围： ● ROUND_ROBIN：加权轮询算法。 ● LEAST_CONNECTIONS：加权最少连接算法。 ● SOURCE_IP：源IP算法。 **说明** 当该字段的取值为SOURCE_IP时，后端云服务器组绑定的后端云服务器的权重设置（weight字段）无效，且不支持开启会话保持。	v1.9及以上

上述注解的使用方法如下：

- 关联已有ELB场景：详情请参见**通过kubectl命令行创建-使用已有ELB**

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: <your_elb_id>                    # ELB ID，替换为实际值
    kubernetes.io/elb.class: performance                    # 负载均衡器类型
```

```
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN                # 负载均衡器算法
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  type: LoadBalancer
```

- 自动创建ELB场景：详情请参见**通过kubectl命令行创建-自动创建ELB**

  共享型负载均衡：

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.autocreate: '{
      "type": "public",
      "bandwidth_name": "cce-bandwidth-1551163379627",
      "bandwidth_chargemode": "bandwidth",
      "bandwidth_size": 5,
      "bandwidth_sharetype": "PER",
      "eip_type": "5_bgp"
    }'
    kubernetes.io/elb.enterpriseID: '0'              # 负载均衡所属企业项目ID
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # 负载均衡器算法
  labels:
    app: nginx
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

  独享型负载均衡：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.autocreate: '{
      "type": "public",
      "bandwidth_name": "cce-bandwidth-1626694478577",
      "bandwidth_chargemode": "bandwidth",
      "bandwidth_size": 5,
      "bandwidth_sharetype": "PER",
      "eip_type": "5_bgp",
      "available_zone": [
        ""
      ],
      "l4_flavor_name": "L4_flavor.elb.s1.small"
    }'
    kubernetes.io/elb.enterpriseID: '0'              # 负载均衡所属企业项目ID
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # 负载均衡器算法
spec:
  selector:
    app: nginx
  ports:
  - name: cce-service-0
```

```
  targetPort: 80
  nodePort: 0
  port: 80
  protocol: TCP
type: LoadBalancer
```

会话保持

表 11-9 会话保持注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.session-affinity-mode	String	支持基于源IP地址的简单会话保持，即来自同一IP地址的访问请求转发到同一台后端服务器上。 • 不启用：不填写该参数。 • 开启会话保持：需增加该参数，取值“SOURCE_IP”，表示基于源IP地址。 **说明** 当kubernetes.io/elb.lb-algorithm设置为“SOURCE_IP”（源IP算法）时，不支持开启会话保持。	v1.9及以上
kubernetes.io/elb.session-affinity-option	表 11-19	ELB会话保持配置选项，可设置会话保持的超时时间。	v1.9及以上

上述注解的使用方法如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: <your_elb_id>                    # ELB ID，替换为实际值
    kubernetes.io/elb.class: union                 # 负载均衡器类型
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP          # 会话保持类型为源IP
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout": "30"}'     # 会话保持时间（分钟）
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  type: LoadBalancer
```

健康检查

表 11-10 健康检查注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.health-check-flag	String	是否开启ELB健康检查功能。 • 开启：“（空值）”或“on” • 关闭：“off” 开启时需同时填写**kubernetes.io/elb.health-check-option**字段。	v1.9及以上
kubernetes.io/elb.health-check-option	**表 11-17**	ELB健康检查配置选项。	v1.9及以上
kubernetes.io/elb.health-check-options	**表 11-18**	ELB健康检查配置选项。支持Service每个端口单独配置，且可以只配置部分端口。 **说明** 不允许同时配置 "kubernetes.io/elb.health-check-option" 和 "kubernetes.io/elb.health-check-options"。	v1.19.16-r5及以上 v1.21.8-r0及以上 v1.23.6-r0及以上 v1.25.2-r0及以上

- kubernetes.io/elb.health-check-option的使用方法如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: <your_elb_id>                    # ELB ID，替换为实际值
    kubernetes.io/elb.class: union                 # 负载均衡器类型
    kubernetes.io/elb.health-check-flag: 'on'                # 开启ELB健康检查功能
    kubernetes.io/elb.health-check-option: '{
      "protocol":"TCP",
      "delay":"5",
      "timeout":"10",
      "max_retries":"3"
    }'
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  type: LoadBalancer
```

- kubernetes.io/elb.health-check-options的使用方法请参见**指定多个端口配置健康检查**。

使用 HTTP 协议

表 11-11 使用 HTTP 协议注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.protocol-port	String	Service使用7层能力配置端口。	v1.19.16及以上
kubernetes.io/elb.cert-id	String	Service使用7层能力配置HTTPS证书。	v1.19.16及以上

具体使用场景和说明请参见**Service使用HTTP协议**。

动态调整后端云服务器权重

表 11-12 动态调整后端云服务器权重注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.adaptive-weight	String	根据Pod动态调整ELB后端云服务器的权重。每个Pod收到的负载请求更加均衡。 ● 开启：true ● 关闭：false 该参数仅1.21及以上集群适用，且ELB直通Pod场景下无效。	v1.21及以上

上述注解的使用方法如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: <your_elb_id>                    # ELB ID，替换为实际值
    kubernetes.io/elb.class: union                 # 负载均衡器类型
    kubernetes.io/elb.adaptive-weight: 'true'                # 开启动态调整后端云服务器权重功能
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    port: 80
    protocol: TCP
```

```
  targetPort: 80
type: LoadBalancer
```

pass-through 能力

表 11-13 pass-through 注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.pass-through	String	集群内访问Service是否经过ELB。	v1.19及以上

具体使用场景和说明请参见**LoadBalancer类型Service使用pass-through能力**。

主机网络

表 11-14 主机网络注解

参数	类型	描述	支持的集群版本
kubernetes.io/hws-hostNetwork	String	如果Pod使用hostNetwork主机网络，使用该注解后ELB会将请求转发至主机网络。 取值范围： • 开启：true • 关闭：false，默认为false	v1.9及以上

上述注解的使用方法如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: <your_elb_id>                    # ELB ID，替换为实际值
    kubernetes.io/elb.class: union                 # 负载均衡器类型
    kubernetes.io/hws-hostNetwork: 'true'                  # ELB会将请求转发至主机网络
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  type: LoadBalancer
```

设置超时时间

表 11-15 设置超时时间注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.keepalive_timeout	Integer	客户端连接空闲超时时间，在超过keepalive_timeout时长一直没有请求，负载均衡会暂时中断当前连接，直到下一次请求时重新建立新的连接。 取值： • 若为TCP协议，取值范围为（10-4000s）默认值为300s。 • 若为HTTP/HTTPS/TERMINATED_HTTPS监听器，取值范围为（0-4000s）默认值为60s。 • UDP监听器不支持此字段。	v1.19.16-r30、v1.21.10-r10、v1.23.8-r10、v1.25.3-r10及以上

具体使用场景和说明请参见**负载均衡类型的服务设置超时时间**。

数据结构说明

表 11-16 elb.autocreate 字段数据结构说明

参数	是否必填	参数类型	描述
name	否	String	自动创建的负载均衡的名称。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。 默认名称：cce-lb+service.UID
type	否	String	负载均衡实例网络类型，公网或者私网。 • public：公网型负载均衡 • inner：私网型负载均衡 默认类型：inner
bandwidth_name	公网型负载均衡必填	String	带宽的名称，默认值为：cce-bandwidth-******。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。

参数	是否必填	参数类型	描述
bandwidth_chargemode	否	String	带宽模式。 ● bandwidth：按带宽 ● traffic：按流量 默认类型：bandwidth
bandwidth_size	公网型负载均衡必填	Integer	带宽大小，默认1Mbit/s~2000Mbit/s，请根据Region带宽支持范围设置。 调整带宽时的最小单位会根据带宽范围不同存在差异。 ● 小于等于300Mbit/s：默认最小单位为1Mbit/s。 ● 300Mbit/s~1000Mbit/s：默认最小单位为50Mbit/s。 ● 大于1000Mbit/s：默认最小单位为500Mbit/s。
bandwidth_sharetype	公网型负载均衡必填	String	带宽共享方式。 ● PER：独享带宽
eip_type	公网型负载均衡必填	String	弹性公网IP类型。 ● 5_bgp：全动态BGP 具体类型以各区域配置为准，详情请参见弹性公网IP控制台。
available_zone	是	Array of strings	负载均衡所在可用区。 独享型负载均衡器独有字段。
l4_flavor_name	是	String	四层负载均衡实例规格名称。 独享型负载均衡器独有字段。
l7_flavor_name	否	String	七层负载均衡实例规格名称。 独享型负载均衡器独有字段，必须与l4_flavor_name对应规格的类型一致，即都为弹性规格或都为固定规格。
elb_virsubnet_ids	否	Array of strings	负载均衡后端所在子网，不填默认集群子网。不同实例规格将占用不同数量子网IP，不建议使用其他资源（如集群，节点等）的子网网段。 独享型负载均衡器独有字段。 示例： `"elb_virsubnet_ids": [ "14567f27-8ae4-42b8-ae47-9f847a4690dd" ]`

表 11-17 elb.health-check-option 字段数据结构说明

参数	是否必填	参数类型	描述
delay	否	String	开始健康检查的初始等待时间（秒）。 默认值：5，取值范围：1-50
timeout	否	String	健康检查的超时时间（秒）。 默认值：10，取值范围1-50
max_retries	否	String	健康检查的最大重试次数。 默认值：3，取值范围1-10
protocol	否	String	健康检查的协议。 取值范围：“TCP”或者“HTTP”
path	否	String	健康检查的URL，协议是“HTTP”时配置。 默认值：“/” 取值范围：1-10000字符

表 11-18 elb.health-check-options 字段数据结构说明

参数	是否必填	参数类型	描述
target_service_port	是	String	spec.ports添加健康检查的目标端口，由协议、端口号组成，如：TCP:80
monitor_port	否	String	重新指定的健康检查端口，不指定时默认使用业务端口。 **说明** 请确保该端口在Pod所在节点已被监听，否则会影响健康检查结果。
delay	否	String	开始健康检查的初始等待时间（秒） 默认值：5，取值范围：1-50
timeout	否	String	健康检查的超时时间（秒） 默认值：10，取值范围1-50
max_retries	否	String	健康检查的最大重试次数 默认值：3，取值范围1-10
protocol	否	String	健康检查的协议 默认值：取关联服务的协议 取值范围：“TCP”、“UDP”或者“HTTP”

参数	是否必填	参数类型	描述
path	否	String	健康检查的URL，协议是“HTTP”时需要配置 默认值：“/” 取值范围：1-10000字符

表 11-19 elb.session-affinity-option 字段数据结构说明

参数	是否必填	参数类型	描述
persistence_timeout	是	String	当elb.session-affinity-mode是“SOURCE_IP”时生效，设置会话保持的超时时间（分钟）。 默认值为："60"，取值范围：1-60。

11.3.4.3 Service 使用 HTTP 协议

约束与限制

- Service使用HTTP协议仅v1.19.16及以上版本集群支持。
- 请勿将Ingress与使用HTTP的Service对接同一个ELB下的同一个监听器，否则将产生端口冲突。
- Service支持使用ELB的7层能力，共享型和独享型ELB都支持对接。

  独享型ELB实例有如下限制：
  - 对接已有的独享型ELB实例时，需要**独享型ELB实例支持4层和7层的flavor**，否则会功能不可用。
  - 使用自动创建的ELB实例时，暂不支持使用CCE控制台自动创建7层独享型ELB实例，请通过YAML进行创建，并注意**使用独享型ELB实例的4层和7层能力**（即在kubernetes.io/elb.autocreate的annotation中指定4层和7层flavor）。

Service 使用 HTTP 协议

使用ELB的7层能力时，需要添加如下annotation：

- **kubernetes.io/elb.protocol-port**: "https:443,http:80"

  protocol-port的取值需要和service的spec.ports字段中的端口对应，格式为protocol:port，port中的端口会匹配service.spec.ports中端口，并将该端口发布成对应的protocol协议。
- **kubernetes.io/elb.cert-id**: "17e3b4f4bc40471c86741dc3aa211379"

  cert-id内容为ELB证书管理的证书ID，当protocol-port指定了https协议，ELB监听器的证书会设置为cert-id证书，当发布多个HTTPS的服务，会使用同一份证书。

配置示例如下，其中spec.ports中两个端口与kubernetes.io/elb.protocol-port中对应，443端口、80端口分别发布成HTTPS、HTTP协议。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    #自动创建ELB实例时，需要同时指定4层和7层flavor
    kubernetes.io/elb.autocreate: '
      {
          "type": "public",
          "bandwidth_name": "cce-bandwidth-1634816602057",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              ""
          ],
          "l7_flavor_name": "L7_flavor.elb.s2.small"
      }'
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.protocol-port: "https:443,http:80"
    kubernetes.io/elb.cert-id: "17e3b4f4bc40471c86741dc3aa211379"
  labels:
    app: nginx
    name: test
  name: test
  namespace: default
spec:
  ports:
  - name: cce-service-0
    port: 443
    protocol: TCP
    targetPort: 80
  - name: cce-service-1
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
    version: v1
  sessionAffinity: None
  type: LoadBalancer
```

使用上面的示例创建Service，在新建的ELB实例中可以看到创建了443端口和80端口的监听器。

11.3.4.4 指定多个端口配置健康检查

LoadBalancer Service的健康检查相关注解字段由"kubernetes.io/elb.health-check-option"升级为"kubernetes.io/elb.health-check-options"，支持Service每个端口单独配置，且可以只配置部分端口。如无需单独配置端口协议，原有注解字段依旧可用无需修改。

约束与限制

- 该特性存在集群版本限制，仅在以下版本中生效：
  - v1.19集群：v1.19.16-r5及以上版本
  - v1.21集群：v1.21.8-r0及以上版本
  - v1.23集群：v1.23.6-r0及以上版本
  - v1.25集群：v1.25.2-r0及以上版本
- 不允许同时配置 "kubernetes.io/elb.health-check-option" 和 "kubernetes.io/elb.health-check-options"。

- target_service_port字段必须配置，且不能重复。
- TCP端口只能配置健康检查协议为TCP、HTTP，UDP端口必须配置健康检查协议为UDP。

操作步骤

使用"kubernetes.io/elb.health-check-options"注解的示例如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  labels:
    app: nginx
    version: v1
  annotations:
    kubernetes.io/elb.class: union          # 负载均衡类型
    kubernetes.io/elb.id: <your_elb_id>    # ELB ID，替换为实际值
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN  # 负载均衡器算法
    kubernetes.io/elb.health-check-flag: 'on'    # 开启ELB健康检查功能
    kubernetes.io/elb.health-check-options: '[
    {
        "protocol": "TCP",
        "delay": "5",
        "timeout": "10",
        "max_retries": "3",
        "target_service_port": "TCP:1",
        "monitor_port": "22"
    },
    {
        "protocol": "HTTP",
        "delay": "5",
        "timeout": "10",
        "max_retries": "3",
        "path": "/",
        "target_service_port": "TCP:2",
        "monitor_port": "22"
    }
    ]'
spec:
  selector:
    app: nginx
    version: v1
  externalTrafficPolicy: Cluster
  ports:
    - name: cce-service-0
      targetPort: 1
      nodePort: 0
      port: 1
      protocol: TCP
    - name: cce-service-1
      targetPort: 2
      nodePort: 0
      port: 2
      protocol: TCP
  type: LoadBalancer
  loadBalancerIP: **.**.**.**
```

表 11-20 elb.health-check-options 字段数据结构说明

参数	是否必填	参数类型	描述
target_service_port	是	String	spec.ports添加健康检查的目标端口，由协议、端口号组成，如：TCP:80
monitor_port	否	String	重新指定的健康检查端口，不指定时默认使用业务端口。 **说明** 请确保该端口在Pod所在节点已被监听，否则会影响健康检查结果。
delay	否	String	开始健康检查的初始等待时间（秒） 默认值：5，取值范围：1-50
timeout	否	String	健康检查的超时时间（秒） 默认值：10，取值范围1-50
max_retries	否	String	健康检查的最大重试次数 默认值：3，取值范围1-10
protocol	否	String	健康检查的协议 默认值：取关联服务的协议 取值范围：“TCP”、“UDP”或者“HTTP”
path	否	String	健康检查的URL，协议是“HTTP”时需要配置 默认值：“/” 取值范围：1-10000字符

11.3.4.5 负载均衡类型的服务设置超时时间

LoadBalancer Service支持设置连接空闲超时时间，即没有收到客户端请求的情况下保持连接的最长时间。如果在这个时间内没有新的请求， 负载均衡会暂时中断当前连接，直到下一次请求时重新建立新的连接。

约束与限制

- 该特性存在集群版本限制，仅在以下版本中生效：
  - v1.19集群：v1.19.16-r30及以上版本
  - v1.21集群：v1.21.10-r10及以上版本
  - v1.23集群：v1.23.8-r10及以上版本
  - v1.25集群：v1.25.3-r10及以上版本
- 仅在使用独享型ELB时，负载均衡类型的服务支持设置超时时间。
- 更新Service时，如果删除超时时间配置，不会修改已有监听器的超时时间配置。

操作步骤

当前支持通过注解的方式设置客户端连接空闲超时时间，示例如下：

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: <your_elb_id>    #本示例中使用已有的独享型ELB，请替换为您的独享型ELB ID
    kubernetes.io/elb.class: performance  # ELB类型
    kubernetes.io/elb.keepalive_timeout: '300'  # 客户端连接空闲超时时间
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

表 11-21 annotation 关键参数说明

参数	是否必填	参数类型	描述
kubernetes.io/elb.keepalive_timeout	否	Integer	客户端连接空闲超时时间，在超过keepalive_timeout时长一直没有请求，负载均衡会暂时中断当前连接，直到下一次请求时重新建立新的连接。 取值： ● 若为TCP协议，取值范围为10-4000s，默认值为300s。 ● 若为HTTP/HTTPS/TERMINATED_HTTPS监听器，取值范围为（0-4000s）默认值为60s。 ● 若为UDP协议，取值范围为10-4000s，默认值为300s。

11.3.4.6 LoadBalancer 类型 Service 使用 pass-through 能力

应用现状

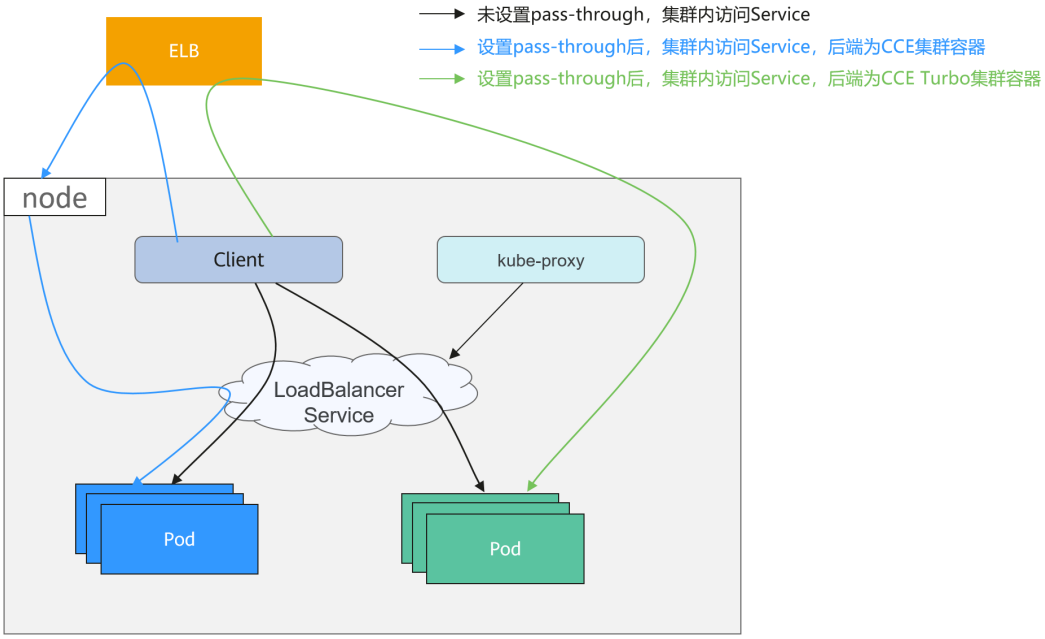
Kubernetes集群可以将运行在一组Pod上的应用程序发布为服务，提供统一的四层访问入口。对于Loadbalancer类型的service，kube-proxy默认会将Service的status中LoadbalanceIP地址配置到节点本地的转发规则中，集群内部访问ELB的地址，流量就会在集群内部转发，而不会经过ELB转发。

集群内部转发功能是kube-proxy组件负责，kube-proxy有iptables和IPVS两种转发模式，iptables是一种简单的轮询转发，IPVS虽有多种转发模式，但也需要修改kube-proxy的启动参数，不能像ELB那样灵活配置转发策略，且无法利用ELB的健康检查能力。

解决方案

CCE服务支持pass-through能力，通过Loadbalance类型Service配置kubernetes.io/elb.pass-through的annotation实现集群内部访问Service的ELB地址时绕出集群，并通过ELB的转发最终转发到后端的Pod。

图 11-17 pass-through 访问示例



- 对于CCE集群：

  集群内部客户端访问LB类型Service时，访问请求默认是通过集群服务转发规则（iptables或IPVS）转发到后端的容器实例。

  当LB类型Service配置elb.pass-through后，集群内部客户端访问Service地址时会先访问到ELB，再通过ELB的负载均衡能力先访问到节点，然后通过集群服务转发规则（iptables或IPVS）转发到后端的容器实例。

约束限制

- 独享型负载均衡配置pass-through后，在工作负载同节点和同节点容器内无法通过Service访问。
- 1.15及以下老版本集群暂不支持该能力。
- IPVS网络模式下，对接同一个ELB的Service需保持pass-through设置情况一致。

操作步骤

下文以nginx镜像创建无状态工作负载，并创建一个具有pass-through的Service。

步骤1 使用nginx镜像创建无状态负载。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

步骤2 Loadbalance类型的Service，并设置kubernetes.io/elb.pass-through为true。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.pass-through: "true"
    kubernetes.io/elb.class: union
    kubernetes.io/elb.autocreate: '{"type":"public","bandwidth_name":"cce-
bandwidth","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_ty
pe":"5_bgp","name":"james"}'
  labels:
    app: nginx
  name: nginx
spec:
  externalTrafficPolicy: Local
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

此处是自动创建一个名为james的共享型ELB实例，需要通过kubernetes.io/elb.subnet-id指定ELB所在VPC子网，必须和集群在同一个VPC。

----结束

配置验证

查看上面创建的Service对应的ELB，名称为james，可以看到ELB的连接数为0。

使用kubectl连接集群，进入到某一个nginx容器中，然后访问ELB的地址。可以看到能够正常访问。

```
# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
nginx-7c4c5cc6b5-vpncx   1/1     Running   0          9m47s
nginx-7c4c5cc6b5-xj5wl   1/1     Running   0          9m47s
# kubectl exec -it nginx-7c4c5cc6b5-vpncx -- /bin/sh
# curl 120.46.141.192
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

```
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

稍微等待一段时间看ELB的监控数据，可以看到ELB有一个新建访问连接，这就证明了这次访问经过ELB，与预期一致。

11.3.4.7 健康检查使用 UDP 协议的安全组规则说明

操作场景

当负载均衡协议为UDP时，健康检查也采用的UDP协议，您需要打开其后端服务器的ICMP协议安全组规则。

操作步骤

步骤1 登录弹性云服务器控制台，找到集群中的节点对应的节点，单击云服务器名称，进入详情页面，记录安全组名称。

步骤2 登录虚拟私有云控制台，在左侧导航栏单击“访问控制 > 安全组”，在界面右侧的安全组列表中单击步骤1获取的安全组名称。

步骤3 在打开的页面中单击“入方向规则”页签，单击“添加规则”，为云服务器添加入方向规则，单击“确定”。

说明

- 您只需为工作负载所在集群下的任意一个节点更改安全组规则，请添加规则即可，不要修改原有的安全组规则。
- 安全组需放通网段100.125.0.0/16流量。

----结束

11.3.5 Headless Service

前面讲的Service解决了Pod的内外部访问问题，但还有下面这些问题没解决。

- 同时访问所有Pod
- 一个Service内部的Pod互相访问

Headless Service正是解决这个问题的，Headless Service不会创建ClusterIP，并且查询会返回所有Pod的DNS记录，这样就可查询到所有Pod的IP地址。**有状态负载StatefulSet**正是使用Headless Service解决Pod间互相访问的问题。

```
apiVersion: v1
kind: Service       # 对象类型为Service
metadata:
  name: nginx-headless
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Pod间通信的端口名称
      port: 80        # Pod间通信的端口号
  selector:
    app: nginx        # 选择标签为app:nginx的Pod
  clusterIP: None     # 必须设置为None，表示Headless Service
```

执行如下命令创建Headless Service。

```
# kubectl create -f headless.yaml
service/nginx-headless created
```

创建完成后可以查询Service。

```
# kubectl get svc
NAME             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx-headless   ClusterIP   None         <none>        80/TCP    5s
```

创建一个Pod来查询DNS，可以看到能返回所有Pod的记录，这就解决了访问所有Pod的问题了。

```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # nslookup nginx-1.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-1.nginx.default.svc.cluster.local
Address: 172.16.0.18

/ # nslookup nginx-2.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-2.nginx.default.svc.cluster.local
Address: 172.16.0.19
```

11.4 路由（Ingress）

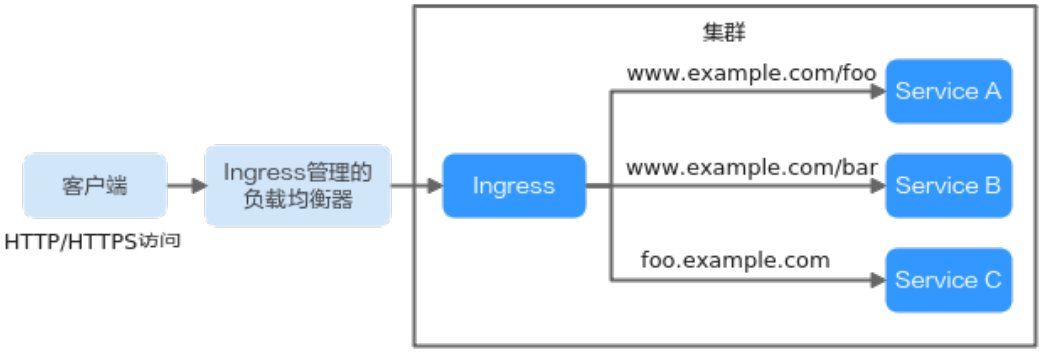
11.4.1 路由概述

为什么需要 Ingress

Service基于TCP和UDP协议进行访问转发，为集群提供了四层负载均衡的能力。但是在实际场景中，Service无法满足应用层中存在着大量的HTTP/HTTPS访问需求。因此，Kubernetes集群提供了另一种基于HTTP协议的访问方式——Ingress。

Ingress是Kubernetes集群中一种独立的资源，制定了集群外部访问流量的转发规则。如图11-18所示，用户可根据域名和路径对转发规则进行自定义，完成对访问流量的细粒度划分。

图 11-18 Ingress 示意图



下面对Ingress的相关定义进行介绍：

- Ingress资源：一组基于域名或URL把请求转发到指定Service实例的访问规则，是Kubernetes的一种资源对象，通过接口服务实现增、删、改、查的操作。
- Ingress Controller：请求转发的执行器，用以实时监控资源对象Ingress、Service、End-point、Secret（主要是TLS证书和Key）、Node、ConfigMap的变化，解析Ingress定义的规则并负责将请求转发到相应的后端Service。

Ingress Controller在不同厂商之间的实现方式不同，根据负载均衡器种类的不同，可以将其分成ELB型和Nginx型。CCE支持上述两种Ingress Controller类型，其中ELB Ingress Controller基于弹性负载均衡服务（ELB）实现流量转发；而Nginx Ingress Controller使用Kubernetes社区维护的模板与镜像，通过Nginx组件完成流量转发。

Ingress 特性对比

表 11-22 Ingress 特性对比

特性	ELB Ingress Controller	Nginx Ingress Controller
运维	免运维	自行安装、升级、维护
性能	一个Ingress支持一个ELB实例	多个Ingress只支持一个ELB实例
	使用企业级LB，高性能高可用，升级、故障等场景不影响业务转发	性能依赖pod的资源配置
	支持配置动态加载	• 非后端端点变更需要Reload进程，对长连接有损。 • 端点变更使用Lua实现热更新。 • Lua插件变更需要Reload进程。
组件部署	Master节点，不占用工作节点	Worker节点，需要Nginx组件运行成本

特性	ELB Ingress Controller	Nginx Ingress Controller
路由重定向	不支持	支持
SSL配置	支持	支持
代理HTTPS协议的后端服务	支持	支持，可通过backend-protocol: "HTTPS"注解实现

由于ELB Ingress和社区开源的Nginx Ingress在原理上存在本质区别，因此支持的Service类型不同，详情请参见**Ingress支持的Service类型**。

ELB Ingress Controller部署在master节点，所有策略配置和转发行为均在ELB侧完成。集群外部的ELB只能通过VPC的IP对接集群内部节点，因此ELB Ingress只支持NodePort类型的Service。

Nginx Ingress Controller运行在集群中，作为服务通过NodePort对外暴露，流量经过Nginx-ingress转发到集群内其他业务，流量转发行为及转发对象均在集群内部，因此支持ClusterIP和NodePort类型的Service。

综上，ELB Ingress使用企业级LB进行流量转发，拥有高性能和高稳定性的优点，而Nginx Ingress Controller部署在集群节点上，牺牲了一定的集群资源但可配置性相对更好。

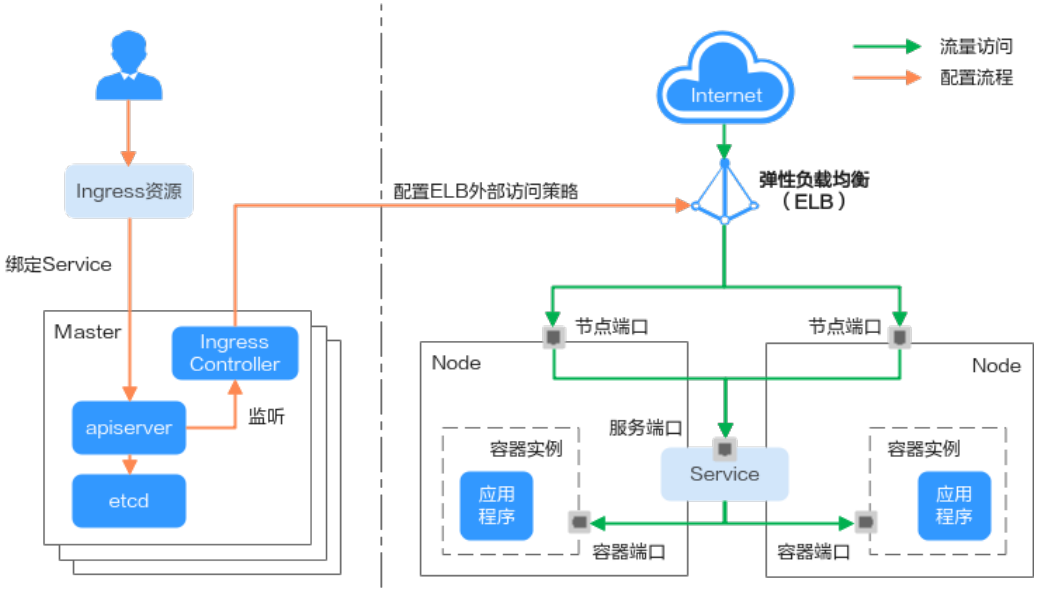
ELB Ingress Controller 工作原理

CCE自研的ELB Ingress Controller基于弹性负载均衡服务ELB实现公网和内网（同一VPC内）的七层网络访问，通过不同的URL将访问流量分发到对应的服务。

ELB Ingress Controller部署于Master节点上，与集群所在VPC下的弹性负载均衡器绑定，支持在同一个ELB实例（同一IP）下进行不同域名、端口和转发策略的设置。ELB Ingress Controller的工作原理如**图11-19**，实现步骤如下：

1. 用户创建Ingress资源，在Ingress中配置流量访问规则，包括负载均衡器、URL、SSL以及访问的后端Service端口等。
2. Ingress Controller监听到Ingress资源发生变化时，就会根据其中定义的流量访问规则，在ELB侧重新配置监听器以及后端服务器路由。
3. 当用户进行访问时，流量根据ELB中配置的转发策略转发到对应的后端Service端口，然后再经过Service二次转发访问到关联的各个工作负载。

图 11-19 ELB Ingress Controller 工作原理



Nginx Ingress Controller 工作原理

Nginx型的Ingress使用弹性负载均衡（ELB）作为流量入口，并在集群中部署**nginx-ingress插件**来对流量进行负载均衡及访问控制。

说明

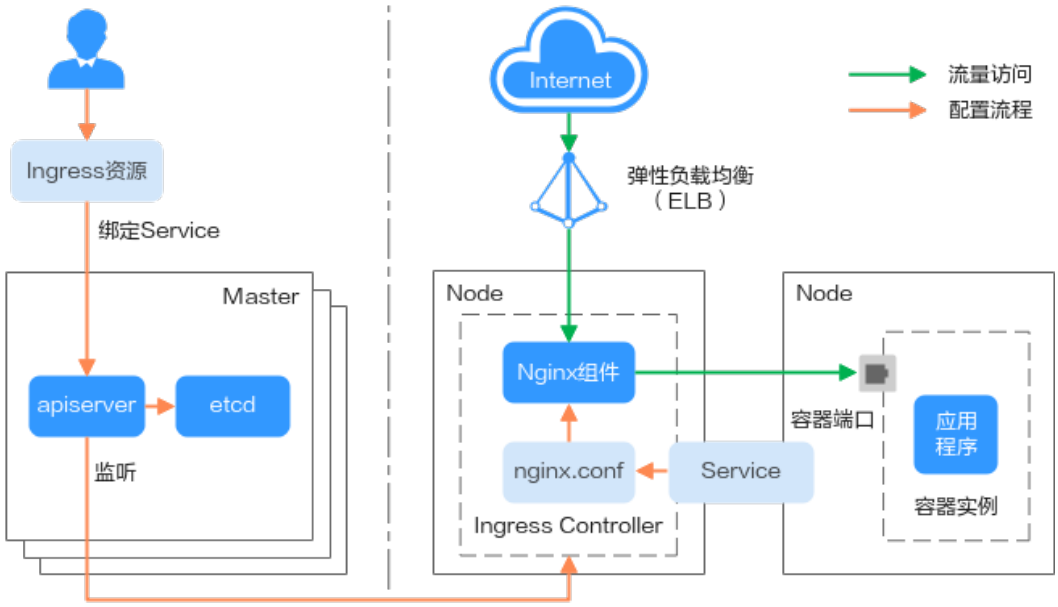
nginx-ingress插件直接使用社区模板与镜像，CCE不提供额外维护，不建议用于商用场景。

开源社区地址：**https://github.com/kubernetes/ingress-nginx**

Nginx型的Ingress Controller通过pod部署在工作节点上，因此引入了相应的运维成本和Nginx组件运行成本，其工作原理如**图11-20**，实现步骤如下：

1. 当用户更新Ingress资源后，Ingress Controller就会将其中定义的转发规则写入到Nginx的配置文件（nginx.conf）中。
2. 内置的Nginx组件进行reload，加载更新后的配置文件，完成Nginx转发规则的修改和更新。
3. 在流量访问集群时，首先被已创建的负载均衡实例转发到集群内部的Nginx组件，然后Nginx组件再根据转发规则将其转发至对应的各个工作负载。

图 11-20 Nginx Ingress Controller 工作原理



Ingress 支持的 Service 类型

ELB Ingress支持的Service类型如表11-23所示。

表 11-23 ELB Ingress 支持的 Service 类型

集群类型	ELB类型	集群内访问（ClusterIP）	节点访问（NodePort）
CCE集群	共享型负载均衡	不支持	支持
	独享型负载均衡	不支持（集群内访问服务关联实例未绑定eni网卡，独享型负载均衡无法对接）	支持
CCE Turbo集群	共享型负载均衡	不支持	支持
	独享型负载均衡	支持	不支持（节点访问服务关联实例已绑定eni网卡，独享型负载均衡无法对接）

Nginx Ingress支持的Service类型如表11-24所示。

表 11-24 Nginx Ingress 支持的 Service 类型

集群类型	ELB类型	集群内访问（ClusterIP）	节点访问（NodePort）
CCE集群	共享型负载均衡	支持	支持

集群类型	ELB类型	集群内访问（ClusterIP）	节点访问（NodePort）
	独享型负载均衡	支持	支持
CCE Turbo集群	共享型负载均衡	支持	支持
	独享型负载均衡	支持	支持

11.4.2 ELB Ingress 管理

11.4.2.1 通过控制台创建 ELB Ingress

前提条件

- Ingress为后端工作负载提供网络访问，因此集群中需提前部署可用的工作负载。若您无可用工作负载，可参考**创建无状态负载（Deployment）**、**创建有状态负载（StatefulSet）**或**创建守护进程集（DaemonSet）**部署工作负载。
- 为上述工作负载配置Service，ELB Ingress支持的Service类型请参见**Ingress支持的Service类型**。

注意事项

- 建议其他资源不要使用Ingress自动创建的ELB实例，否则在删除Ingress时，ELB实例会被占用，导致资源残留。
- 添加Ingress后请在CCE页面对所选ELB实例进行配置升级和维护，不可在ELB页面对配置进行更改，否则可能导致Ingress服务异常。
- Ingress转发策略中注册的URL需与后端应用提供访问的URL一致，否则将返回404错误。
- IPVS模式集群下，Ingress和Service使用相同ELB实例时，无法在集群内的节点和容器中访问Ingress，因为kube-proxy会在ipvs-0的网桥上挂载LB类型的Service地址，Ingress对接的ELB的流量会被ipvs-0网桥劫持。建议Ingress和Service使用不同ELB实例。
- 独享型ELB规格必须支持应用型（HTTP/HTTPS），且网络类型必须支持私网（有私有IP地址）。
- 同集群使用多个Ingress对接同一个ELB端口时，监听器的配置项（例如监听器关联的证书、监听器HTTP2属性等）均以第一个Ingress配置为准。

添加 ELB Ingress

本节以nginx作为工作负载并添加ELB Ingress为例进行说明。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 选择左侧导航栏的“服务发现”，在右侧选择“路由”页签，单击右上角“创建路由”。

步骤3 设置Ingress参数。

- **名称：**自定义Ingress名称，例如ingress-demo。
- **对接Nginx：**此选项只有在安装了**NGNIX Ingress Controller**插件后才会显示。如显示了“对接Nginx”，则说明您安装了nginx-ingress插件，创建ELB Ingress时不能打开该项开关，如果打开则是使用Nginx Ingress Controller，具体请参见**通过控制台创建Nginx Ingress**。
- **负载均衡器：**

  选择对接的ELB实例，仅支持与集群在同一个VPC下的ELB实例。如果没有可选的ELB实例，请单击“创建负载均衡器”跳转到ELB控制台创建。

  独享型ELB规格需要支持应用型（HTTP/HTTPS），且网络类型必须支持私网。

  CCE控制台支持自动创建ELB实例，在下拉框选择“自动创建”，并填写以下参数：

  - 实例名称：请填写ELB名称。
  - 公网访问：开启后，将创建 5 Mbit/s 带宽的弹性公网 IP。
  - 可用区、子网和规格（仅独享型ELB实例支持选择）：设置独享型ELB的可用区、子网和规格。当前仅支持自动创建应用型（HTTP/HTTPS）独享型ELB实例。
- **监听器配置：**Ingress为负载均衡器配置监听器，监听器对负载均衡器上的请求进行监听，并分发流量。配置完成后ELB实例侧将会创建对应的监听器，名称默认为k8s_<协议类型>_<端口号>，例如“k8s_HTTP_80”。
  - 对外协议：支持HTTP和HTTPS。
  - 对外端口：开放在负载均衡服务地址的端口，可任意指定。
  - 证书来源：支持TLS密钥和ELB服务器证书。
  - 服务器证书：负载均衡器创建HTTPS协议监听时需要绑定证书，以支持HTTPS数据传输加密认证。
    - TLS密钥：创建密钥证书的方法请参见**创建密钥**。
    - ELB服务器证书：使用在ELB服务中创建的证书。

    说明

    同一个ELB实例的同一个端口配置HTTPS时，一个监听器只支持配置一个密钥证书。若使用两个不同的密钥证书将两个Ingress添加到同一个ELB下的同一个监听器，ELB侧实际只生效最先添加的证书。
  - SNI：SNI（Server Name Indication）是TLS的扩展协议，在该协议下允许同一个IP地址和端口号下对外提供多个基于TLS的访问域名，且不同的域名可以使用不同的安全证书。开启SNI后，允许客户端在发起TLS握手请求时就提交请求的域名信息。负载均衡收到TLS请求后，会根据请求的域名去查找证书：若找到域名对应的证书，则返回该证书认证鉴权；否则，返回缺省证书（服务器证书）认证鉴权。

    说明

    - 当选择HTTPS协议时，才支持配置“SNI”选项。
    - 该功能仅支持1.15.11及以上版本的集群。
    - 用于SNI的证书需要指定域名，每个证书只能指定一个域名。支持泛域名证书。
  - 安全策略：安全策略包含HTTPS可选的TLS协议版本和配套的加密算法套件。

    关于安全策略的详细说明，请参见ELB用户指南。

说明

- 选择HTTPS协议时，才支持配置“安全策略”选项。
- 该功能仅支持1.17.9及以上版本的集群。

- **转发策略配置：**请求的访问地址与转发规则匹配时（转发规则由域名、URL组成，例如：10.XXX.XXX.XXX:80/helloworld），此请求将被转发到对应的目标Service处理。单击 + 按钮可添加多条转发策略。
  - 域名：实际访问的域名地址。请确保所填写的域名已注册并备案，一旦配置了域名规则后，必须使用域名访问。
  - URL匹配规则：
    - 前缀匹配：例如映射URL为/healthz，只要符合此前缀的URL均可访问。例如/healthz/v1，/healthz/v2。
    - 精确匹配：表示只有URL完全匹配时，访问才能生效。例如映射URL为/healthz，则必须为此URL才能访问。
    - 正则匹配：按正则表达式方式匹配URL。例如正则表达式为**/[A-Za-z0-9_.-]+/test**。只要符合此规则的URL均可访问，例如/abcA9/test，/v1-Ab/test。正则匹配规则支持POSIX与Perl两种标准。
  - URL：需要注册的访问路径，例如：/healthz。

    说明

    此处添加的访问路径要求后端应用内存在相同的路径，否则转发无法生效。

    例如，Nginx应用默认的Web访问路径为“/usr/share/nginx/html”，在为Ingress转发策略添加“/test”路径时，需要应用的Web访问路径下也包含相同路径，即“/usr/share/nginx/html/test”，否则将返回404。
  - 目标服务名称：请选择已有Service或新建Service。页面列表中的查询结果已自动过滤不符合要求的Service。
  - 目标服务访问端口：可选择目标Service的访问端口。
  - 负载均衡配置：
    - 分配策略：可选择加权轮询算法、加权最少连接或源IP算法。

      说明

      - 加权轮询算法：根据后端服务器的权重，按顺序依次将请求分发给不同的服务器。它用相应的权重表示服务器的处理性能，按照权重的高低以及轮询方式将请求分配给各服务器，相同权重的服务器处理相同数目的连接数。常用于短连接服务，例如HTTP等服务。
      - 加权最少连接：最少连接是通过当前活跃的连接数来估计服务器负载情况的一种动态调度算法。加权最少连接就是在最少连接数的基础上，根据服务器的不同处理能力，给每个服务器分配不同的权重，使其能够接受相应权值数的服务请求。常用于长连接服务，例如数据库连接等服务。
      - 源IP算法：将请求的源IP地址进行Hash运算，得到一个具体的数值，同时对后端服务器进行编号，按照运算结果将请求分发到对应编号的服务器上。这可以使得对不同源IP的访问进行负载分发，同时使得同一个客户端IP的请求始终被派发至某特定的服务器。该方式适合负载均衡无cookie功能的TCP协议。

- 会话保持类型：默认不启用。支持以下类型：
  - 负载均衡器cookie：同时需填写“会话保持时间”，范围为1-1440分钟。
  - 应用程序cookie：仅共享型ELB支持设置。同时需填写“cookie名称”，长度范围为1-64。

  说明

  当**分配策略**使用源IP算法时，不支持设置会话保持。

- 健康检查：设置负载均衡的健康检查配置，启用时支持以下配置。

参数	说明
协议	当目标服务端口配置协议为TCP时，支持TCP和HTTP协议；当目标服务端口配置协议为UDP时，支持UDP协议。 ○ 检查路径（仅HTTP健康检查协议支持）：指定健康检查的URL地址。检查路径只能以/开头，长度范围为1-80。
端口	健康检查默认使用业务端口（Service的NodePort和容器端口）作为健康检查的端口；您也可以重新指定端口用于健康检查，重新指定端口会为服务增加一个名为cce-healthz的服务端口配置。 ○ 节点端口：使用共享型负载均衡或不关联ENI实例时，节点端口作为健康检查的检查端口；如不指定将随机一个端口。取值范围为30000-32767。 ○ 容器端口：使用独享型负载均衡关联ENI实例时，容器端口作为健康检查的检查端口。取值范围为1-65535。
检查周期（秒）	每次健康检查响应的最大间隔时间，取值范围为1-50。
超时时间（秒）	每次健康检查响应的最大超时时间，取值范围为1-50。
最大重试次数	健康检查最大的重试次数，取值范围为1-10。

  - 操作：可单击“删除”按钮删除该配置。

- **注解**：Ingress有一些CCE定制的高级功能，通过注解annotations实现。在使用kubectl创建时，会用到注解，具体请参见**添加Ingress-自动创建ELB**和**添加Ingress-对接已有ELB**。

步骤4 配置完成后，单击“确定”。创建完成后，在Ingress列表可查看到已添加的Ingress。

在ELB控制台可查看通过CCE自动创建的ELB，名称默认为“cce-lb-ingress.UID”。单击ELB名称进入详情页，在“监听器”页签下即可查看Ingress对应的路由设置，包括URL、监听器端口以及对应的后端服务器组端口。

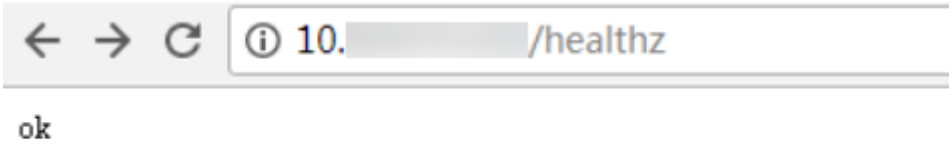
须知

Ingress创建后请在CCE页面对所选ELB实例进行配置升级和维护，不要在ELB控制台对ELB实例进行维护，否则可能导致Ingress服务异常。

步骤5 访问工作负载（例如名称为defaultbackend）的“/healthz”接口。

1. 获取工作负载“/healthz”接口的访问地址。访问地址由负载均衡实例IP、对外端口、映射URL组成，例如：10.**.**.**:80/healthz。
2. 在浏览器中输入“/healthz”接口的访问地址，如：http://10.**.**.**:80/healthz，即可成功访问工作负载，如图11-21。

图 11-21 访问 defaultbackend“/healthz”接口



----结束

11.4.2.2 通过 Kubectl 命令行创建 ELB Ingress

操作场景

本节以**Nginx工作负载**为例，说明通过kubectl命令添加ELB Ingress的方法。

- 如您在同一VPC下没有可用的ELB，CCE支持在添加Ingress时自动创建ELB，请参考**添加Ingress-自动创建ELB**。
- 如您已在同一VPC下提前创建了一个可用的ELB，则可参考**添加Ingress-对接已有ELB**。

前提条件

- Ingress为后端工作负载提供网络访问，因此集群中需提前部署可用的工作负载。若您无可用工作负载，可参考**创建无状态负载（Deployment）**、**创建有状态负载（StatefulSet）**或**创建守护进程集（DaemonSet）**部署示例nginx工作负载。
- 为上述工作负载配置Service，ELB Ingress支持的Service类型请参见**Ingress支持的Service类型**。
- 独享型ELB规格必须支持应用型（HTTP/HTTPS），且网络类型必须支持私网（有私有IP地址）。

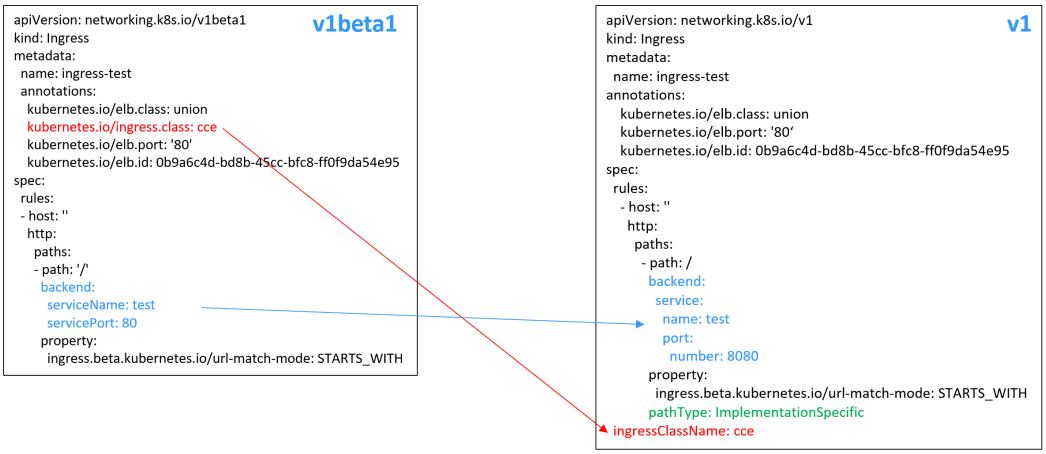
networking.k8s.io/v1 版本 Ingress 说明

CCE在v1.23版本集群开始Ingress切换到**networking.k8s.io/v1**版本。

v1版本参数相较v1beta1参数有如下区别。

- ingress类型由annotations中**kubernetes.io/ingress.class**变为使用**spec.ingressClassName**字段。

- **backend**的写法变化。
- 每个路径下必须指定路径类型**pathType**，支持如下类型。
  - ImplementationSpecific: 对于这种路径类型，匹配方法取决于具体Ingress Controller的实现。在CCE中会使用ingress.beta.kubernetes.io/url-match-mode指定的匹配方式，这与v1beta1方式相同。
  - Exact：精确匹配 URL 路径，且区分大小写。
  - Prefix：基于以 / 分隔的 URL 路径前缀匹配。匹配区分大小写，并且对路径中的元素逐个匹配。 路径元素指的是由 / 分隔符分隔的路径中的标签列表。



添加 Ingress-自动创建 ELB

下面介绍如何通过kubectl命令在添加Ingress时自动创建ELB。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建名为“**ingress-test.yaml**”的YAML文件，此处文件名可自定义。

vi ingress-test.yaml

说明

CCE在1.23版本集群开始Ingress切换到networking.k8s.io/v1版本，之前版本集群使用networking.k8s.io/v1beta1。v1版本与v1beta1版本的区别请参见**networking.k8s.io/v1版本Ingress说明**。

共享型负载均衡（公网访问）示例 -1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.autocreate:
      '{
          "type":"public",
          "bandwidth_name":"cce-bandwidth-******",
          "bandwidth_chargemode":"bandwidth",
          "bandwidth_size":5,
          "bandwidth_sharetype":"PER",
          "eip_type":"5_bgp"
        }'
spec:
  rules:
```

```
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: <your_service_name>  #替换为您的目标服务名称
            port:
              number: <your_service_port>  #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce    # 表示使用ELB Ingress
```

共享型负载均衡（公网访问）示例 - 1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/ingress.class: cce    # 表示使用ELB Ingress
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.autocreate:
      '{
          "type":"public",
          "bandwidth_name":"cce-bandwidth-******",
          "bandwidth_chargemode":"bandwidth",
          "bandwidth_size":5,
          "bandwidth_sharetype":"PER",
          "eip_type":"5_bgp"
        }'
spec:
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: <your_service_port>  #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

独享型负载均衡（公网访问）示例 - 1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.autocreate:
      '{
          "type": "public",
          "bandwidth_name": "cce-bandwidth-******",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              "eu-west-0a"
          ],
          "elb_virsubnet_ids":["b4bf8152-6c36-4c3b-9f74-2229f8e640c9"],
          "l7_flavor_name": "L7_flavor.elb.s1.small"
      }'
spec:
  rules:
  - host: ''
```

```
  http:
    paths:
    - path: '/'
      backend:
        service:
          name: <your_service_name>  #替换为您的目标服务名称
          port:
            number: <your_service_port>  #替换为您的目标服务端口
      property:
        ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
      pathType: ImplementationSpecific
ingressClassName: cce
```

独享型负载均衡（公网访问）示例 - 1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.autocreate:
      '{
          "type": "public",
          "bandwidth_name": "cce-bandwidth-******",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              "eu-west-0a"
          ],
          "elb_virsubnet_ids":["b4bf8152-6c36-4c3b-9f74-2229f8e640c9"],
          "l7_flavor_name": "L7_flavor.elb.s1.small"
      }'
spec:
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: <your_service_port>  #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

表 11-25 关键参数说明

参数	是否必填	参数类型	描述
kubernetes.io/elb.class	是	String	请根据不同的应用场景和功能需求选择合适的负载均衡器类型。 ● union：共享型负载均衡。 ● performance：独享型负载均衡，仅支持1.17及以上集群。
kubernetes.io/ingress.class	是 （仅1.21及以下集群）	String	cce：表示使用自研ELBIngress。 通过API接口创建Ingress时必须增加该参数。

参数	是否必填	参数类型	描述
ingressClassName	是 （仅1.23及以上集群）	String	cce：表示使用自研ELBIngress。 通过API接口创建Ingress时必须增加该参数。
kubernetes.io/elb.port	是	Integer	界面上的对外端口，为注册到负载均衡服务地址上的端口。 取值范围：1~65535。 **说明** 部分端口为高危端口，默认被屏蔽，如21端口。
kubernetes.io/elb.subnet-id	-	String	为集群所在子网的ID，取值范围：1~100字符。 ● Kubernetes v1.11.7-r0及以下版本的集群自动创建时：必填。 ● Kubernetes v1.11.7-r0以上版本的集群：可不填，默认为""。
kubernetes.io/elb.enterpriseID	否	String	**Kubernetes v1.15及以上版本的集群支持此字段；Kubernetes v1.15以下版本默认创建到default项目下。** 企业项目ID，选择后可以直接创建在具体的企业项目下。 取值范围：1~100字符。 **获取方法：** 登录企业项目管理控制台，在左侧导航栏中选择“项目管理”，在企业项目列表中单击要加入的企业项目名称，进入企业项目详情页，找到“ID”字段复制即可。
kubernetes.io/elb.autocreate	是	elb.autocreate object	自动创建Ingress关联的ELB，详细字段说明参见**表11-26**。 **示例：** ● 公网自动创建： 值为 '{"type":"public","bandwidth_name":"cce-bandwidth-******","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_type":"5_bgp","name":"james"}' ● 私网自动创建： 值为 '{"type":"inner", "name": "A-location-d-test"}'

参数	是否必填	参数类型	描述
host	否	String	为服务访问域名配置，默认为""，表示域名全匹配。请确保所填写的域名已注册并备案，一旦配置了域名规则后，必须使用域名访问。
path	是	String	为路由路径，用户自定义设置。所有外部访问请求需要匹配host和path。 **说明** 此处添加的访问路径要求后端应用内存在相同的路径，否则转发无法生效。 例如，Nginx应用默认的Web访问路径为“/usr/share/nginx/html”，在为Ingress转发策略添加“/test”路径时，需要应用的Web访问路径下也包含相同路径，即“/usr/share/nginx/html/test”，否则将返回404。
ingress.beta.kubernetes.io/url-match-mode	否	String	路由匹配策略。 默认值为“STARTS_WITH”(前缀匹配)。 取值范围： ● EQUAL_TO：精确匹配 ● STARTS_WITH：前缀匹配 ● REGEX：正则匹配

参数	是否必填	参数类型	描述
pathType	是	String	路径类型，该字段仅v1.23及以上集群支持。 ● ImplementationSpecific: 匹配方法取决于具体Ingress Controller的实现。在CCE中会使用ingress.beta.kubernetes.io/url-match-mode指定的匹配方式。 ● Exact：精确匹配 URL 路径，且区分大小写。 ● Prefix：前缀匹配，且区分大小写。该方式是将URL路径通过“/”分隔成多个元素 ，并且对元素进行逐个匹配。 如果URL中的每个元素均和路径匹配，则说明该URL的子路径均可以正常路由。 **说明** – Prefix匹配时每个元素均需精确匹配，如果URL的最后一个元素是请求路径中最后一个元素的子字符串，则不会匹配 。例如：/foo/bar匹配/foo/bar/baz，但不匹配/foo/barbaz。 – 通过“/”分隔元素时，若URL或请求路径以“/”结尾，将会忽略结尾的“/”。例如：/foo/bar会匹配/foo/bar/。 关于Ingress路径匹配示例，请参见示例。

表 11-26 elb.autocreate 字段数据结构说明

参数	是否必填	参数类型	描述
name	否	String	自动创建的负载均衡的名称。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。 默认名称：cce-lb+service.UID
type	否	String	负载均衡实例网络类型，公网或者私网。 ● public：公网型负载均衡 ● inner：私网型负载均衡 默认类型：inner

参数	是否必填	参数类型	描述
bandwidth_name	公网型负载均衡必填	String	带宽的名称，默认值为：cce-bandwidth-******。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。
bandwidth_chargemode	否	String	带宽模式。 ● bandwidth：按带宽 ● traffic：按流量 默认类型：bandwidth
bandwidth_size	公网型负载均衡必填	Integer	带宽大小，默认1Mbit/s~2000Mbit/s，请根据Region带宽支持范围设置。 调整带宽时的最小单位会根据带宽范围不同存在差异。 ● 小于等于300Mbit/s：默认最小单位为1Mbit/s。 ● 300Mbit/s~1000Mbit/s：默认最小单位为50Mbit/s。 ● 大于1000Mbit/s：默认最小单位为500Mbit/s。
bandwidth_sharetype	公网型负载均衡必填	String	带宽共享方式。 ● PER：独享带宽
eip_type	公网型负载均衡必填	String	弹性公网IP类型。 ● 5_bgp：全动态BGP 具体类型以各区域配置为准，详情请参见弹性公网IP控制台。
available_zone	是	Array of strings	负载均衡所在可用区。 独享型负载均衡器独有字段。
l4_flavor_name	是	String	四层负载均衡实例规格名称。 独享型负载均衡器独有字段。
l7_flavor_name	否	String	七层负载均衡实例规格名称。 独享型负载均衡器独有字段，必须与l4_flavor_name对应规格的类型一致，即都为弹性规格或都为固定规格。

参数	是否必填	参数类型	描述
elb_virsubnet_ids	否	Array of strings	负载均衡后端所在子网，不填默认集群子网。不同实例规格将占用不同数量子网IP，不建议使用其他资源（如集群，节点等）的子网网段。 独享型负载均衡器独有字段。 示例： "elb_virsubnet_ids": ["14567f27-8ae4-42b8-ae47-9f847a4690dd"]

步骤3 创建Ingress。

kubectl create -f ingress-test.yaml

回显如下，表示Ingress服务已创建。

```
ingress/ingress-test created
```

kubectl get ingress

回显如下，表示Ingress服务创建成功，工作负载可访问。

```
NAME           HOSTS     ADDRESS         PORTS   AGE
ingress-test   *         121.**.**.**    80      10s
```

步骤4 访问工作负载（例如**Nginx工作负载**），在浏览器中输入访问地址http://121.**.**.**:80进行验证。

其中，121.**.**.**为统一负载均衡实例的IP地址。

----结束

添加 Ingress-对接已有 ELB

CCE支持在添加Ingress时选择对接已有的ELB。

说明

- 对接已有独享型ELB规格必须支持应用型（HTTP/HTTPS），且网络类型必须支持私网（有私有IP）。

以1.23及以上版本集群为例，YAML文件配置如下：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.id: <your_elb_id>  #替换为您已有的ELB ID
    kubernetes.io/elb.ip: <your_elb_ip>  #替换为您已有的ELB IP
    kubernetes.io/elb.class: performance  #ELB类型
    kubernetes.io/elb.port: '80'
spec:
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
```

```
        service:
          name: <your_service_name>  #替换为您的目标服务名称
          port:
            number: 8080             #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```

以1.21及以下版本集群为例，YAML文件配置如下：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.id: <your_elb_id>  #替换为您已有的ELB ID
    kubernetes.io/elb.ip: <your_elb_ip>  #替换为您已有的ELB IP
    kubernetes.io/elb.class: performance  #ELB类型
    kubernetes.io/elb.port: '80
    kubernetes.io/ingress.class: cce
spec:
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

表 11-27 关键参数说明

参数	是否必填	参数类型	描述
kubernetes.io/elb.id	是	String	为负载均衡实例的ID，取值范围：1-100字符。 **获取方法：** 在控制台的“服务列表”中，单击“网络 > 弹性负载均衡 ELB”，单击ELB的名称，在ELB详情页的“基本信息”页签下找到“ID”字段复制即可。
kubernetes.io/elb.ip	否	String	为负载均衡实例的服务地址，公网ELB配置为公网IP，私网ELB配置为私网IP。
kubernetes.io/elb.class	是	String	负载均衡器类型。 ● union：共享型负载均衡。 ● performance：独享型负载均衡，仅支持1.17及以上集群。 **说明** ELB Ingress对接已有的独享型ELB时，该独享型ELB必须支持应用型（HTTP/HTTPS）规格。

11.4.2.3 使用 Annotation 配置 ELB Ingress

通过在YAML中添加注解Annotation（注解），您可以实现更多的Ingress高级功能。本文介绍在创建ELB类型的Ingress时可供使用的Annotation。

- **对接ELB**
- **使用HTTP/2**
- **对接HTTPS协议的后端服务**
- **配置Ingress超时时间**

对接 ELB

表 11-28 对接 ELB 注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.class	String	请根据不同的应用场景和功能需求选择合适的负载均衡器类型。 取值如下： • union：共享型负载均衡。 • performance：独享型负载均衡，仅支持1.17及以上集群。	v1.9及以上
kubernetes.io/ingress.class	String	• cce：表示使用自研ELB Ingress。 • nginx：表示使用Nginx Ingress。 通过API接口创建Ingress时必须增加该参数。 v1.23及以上集群使用ingressClassName参数代替，详情请参见**通过Kubectl命令行创建ELB Ingress**。	仅v1.21及以下集群
kubernetes.io/elb.port	Integer	界面上的对外端口，为注册到负载均衡服务地址上的端口。 取值范围：1~65535。 **说明** 部分端口为高危端口，默认被屏蔽，如21端口。	v1.9及以上
kubernetes.io/elb.id	String	**仅关联已有ELB的场景**：必填。 为负载均衡实例的ID。 **获取方法：** 在控制台的“服务列表”中，单击“网络 > 弹性负载均衡 ELB”，单击ELB的名称，在ELB详情页的“基本信息”页签下找到“ID”字段复制即可。	v1.9及以上

参数	类型	描述	支持的集群版本
kubernetes.io/elb.ip	String	**仅关联已有ELB的场景**：必填。 为负载均衡实例的服务地址，公网ELB配置为公网IP，私网ELB配置为私网IP。	v1.9及以上
kubernetes.io/elb.autocreate	表11-32 Object	**仅自动创建ELB的场景**：必填。 **示例：** ● 公网自动创建： 值为 '{"type":"public","bandwidth_name":"cce-bandwidth-1551163379627","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_type":"5_bgp","name":"james"}' ● 私网自动创建： 值为 '{"type":"inner", "name": "A-location-d-test"}'	v1.9及以上
kubernetes.io/elb.enterpriseID	String	**仅自动创建ELB的场景**：选填。 **v1.15及以上版本的集群支持此字段，v1.15以下版本默认创建到default项目下。** 为ELB企业项目ID，选择后可以直接创建在具体的ELB企业项目下。 该字段不传（或传为字符串'0'），则将资源绑定给默认企业项目。 **获取方法：** 登录企业项目管理控制台，在左侧导航栏中选择“项目管理”，在企业项目列表中单击要加入的企业项目名称，进入企业项目详情页，找到“ID”字段复制即可。	v1.15及以上
kubernetes.io/elb.subnet-id	String	**仅自动创建ELB的场景**：选填。 为集群所在子网的ID，取值范围：1-100字符。 ● Kubernetes v1.11.7-r0及以下版本的集群自动创建时：必填 ● Kubernetes v1.11.7-r0以上版本的集群：可不填。	v1.11.7-r0以下必填 v1.11.7-r0以上该字段废弃

上述注解的使用方法如下：

- 关联已有ELB场景：详情请参见**添加Ingress-对接已有ELB**
- 自动创建ELB场景：详情请参见**添加Ingress-自动创建ELB**

使用 HTTP/2

表 11-29 使用 HTTP/2 注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.http2-enable	String	表示HTTP/2功能的开启状态。开启后，可提升客户端与LB间的访问性能，但LB与后端服务器间仍采用HTTP1.X协议。**v1.19.16-r0、v1.21.3-r0及以上版本的集群支持此字段**。 取值范围： ● true：开启HTTP/2功能； ● false：关闭HTTP/2功能（默认为关闭状态）。 注意：**只有当监听器的协议为HTTPS时，才支持开启或关闭HTTP/2功能**。当监听器的协议为HTTP时，该字段无效，默认将其设置为false。	v1.19.16-r0、v1.21.3-r0及以上

具体使用场景和说明请参见**ELB Ingress使用HTTP/2**。

对接 HTTPS 协议的后端服务

表 11-30 对接 HTTPS 协议的后端服务注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.pool-protocol	String	对接HTTPS协议的后端服务，取值为'https'。	v1.23.8、v1.25.3及以上

具体使用场景和说明请参见**ELB Ingress对接HTTPS协议的后端服务**。

配置 Ingress 超时时间

表 11-31 配置 Ingress 重定向规则注解

参数	类型	描述	支持的集群版本
kubernetes.io/elb.keepalive_timeout	Integer	客户端连接空闲超时时间，在超过keepalive_timeout时长一直没有请求，负载均衡会暂时中断当前连接，直到下一次请求时重新建立新的连接。 取值： ● 若为TCP协议，取值范围为（10-4000s）默认值为300s。 ● 若为HTTP/HTTPS协议，取值范围为（0-4000s）默认值为60s。 UDP监听器不支持此字段。	v1.19.16-r30、v1.21.10-r10、v1.23.8-r10、v1.25.3-r10及以上
kubernetes.io/elb.client_timeout	Integer	等待客户端请求超时时间，包括两种情况： ● 读取整个客户端请求头的超时时长：如果客户端未在超时时长内发送完整个请求头，则请求将被中断 ● 两个连续body体的数据包到达LB的时间间隔，超出client_timeout将会断开连接。 取值范围为1-300s，默认值为60s。 使用说明：仅协议为HTTP/HTTPS的监听器支持该字段。 最小值：1 最大值：300 缺省值：60	v1.19.16-r30、v1.21.10-r10、v1.23.8-r10、v1.25.3-r10及以上
kubernetes.io/elb.member_timeout	Integer	等待后端服务器响应超时时间。请求转发后端服务器后，在等待超时member_timeout时长没有响应，负载均衡将终止等待，并返回 HTTP504错误码。 取值：1-300s，默认为60s。 使用说明：仅支持协议为HTTP/HTTPS的监听器。 最小值：1 最大值：300 缺省值：60	v1.19.16-r30、v1.21.10-r10、v1.23.8-r10、v1.25.3-r10及以上

具体使用场景和说明请参见**ELB Ingress设置超时时间**。

数据结构

表 11-32 elb.autocreate 字段数据结构说明

参数	是否必填	参数类型	描述
name	否	String	自动创建的负载均衡的名称。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。 默认名称：cce-lb+service.UID
type	否	String	负载均衡实例网络类型，公网或者私网。 • public：公网型负载均衡 • inner：私网型负载均衡 默认类型：inner
bandwidth_name	公网型负载均衡必填	String	带宽的名称，默认值为：cce-bandwidth-******。 取值范围：只能由中文、英文字母、数字、下划线、中划线、点组成，且长度范围为1-64个字符。
bandwidth_chargemode	否	String	带宽模式。 • bandwidth：按带宽 • traffic：按流量 默认类型：bandwidth
bandwidth_size	公网型负载均衡必填	Integer	带宽大小，默认1Mbit/s~2000Mbit/s，请根据Region带宽支持范围设置。 调整带宽时的最小单位会根据带宽范围不同存在差异。 • 小于等于300Mbit/s：默认最小单位为1Mbit/s。 • 300Mbit/s~1000Mbit/s：默认最小单位为50Mbit/s。 • 大于1000Mbit/s：默认最小单位为500Mbit/s。
bandwidth_sharetype	公网型负载均衡必填	String	带宽共享方式。 • PER：独享带宽

参数	是否必填	参数类型	描述
eip_type	公网型负载均衡必填	String	弹性公网IP类型。 • 5_bgp：全动态BGP 具体类型以各区域配置为准，详情请参见弹性公网IP控制台。
available_zone	是	Array of strings	负载均衡所在可用区。 独享型负载均衡器独有字段。
l4_flavor_name	是	String	四层负载均衡实例规格名称。 独享型负载均衡器独有字段。
l7_flavor_name	否	String	七层负载均衡实例规格名称。 独享型负载均衡器独有字段，必须与l4_flavor_name对应规格的类型一致，即都为弹性规格或都为固定规格。
elb_virsubnet_ids	否	Array of strings	负载均衡后端所在子网，不填默认集群子网。不同实例规格将占用不同数量子网IP，不建议使用其他资源（如集群，节点等）的子网网段。 独享型负载均衡器独有字段。 示例： "elb_virsubnet_ids": ["14567f27-8ae4-42b8-ae47-9f847a4690dd"]

11.4.2.4 ELB Ingress 配置 HTTPS 证书

Ingress支持配置TLS证书，以HTTPS协议的方式对外提供安全服务。

当前支持使用配置在集群中的TLS类型的密钥证书，以及ELB服务中的证书。

说明

同一个ELB实例的同一个端口配置HTTPS时，需要选择一样的证书。

使用 TLS 类型的密钥证书

步骤1 请参见通过kubectl连接集群，使用kubectl连接集群。

步骤2 Ingress支持使用kubernetes.io/tls和IngressTLS两种TLS密钥类型，此处以IngressTLS类型为例，详情请参见创建密钥。kubernetes.io/tls类型的密钥示例及说明请参见TLS Secret。

执行如下命令，创建名为“**ingress-test-secret.yaml**”的YAML文件，此处文件名可自定义。

vi ingress-test-secret.yaml

YAML文件配置如下：

```
apiVersion: v1
data:
  tls.crt: LS0******tLS0tCg==
  tls.key: LS0tL******0tLS0K
kind: Secret
metadata:
  annotations:
    description: test for ingressTLS secrets
  name: ingress-test-secret
  namespace: default
type: IngressTLS
```

说明

此处tls.crt和tls.key为示例，请获取真实的证书和密钥进行替换。tls.crt和tls.key的值为Base64编码后的内容。

步骤3 创建密钥。

kubectl create -f ingress-test-secret.yaml

回显如下，表明密钥已创建。

```
secret/ingress-test-secret created
```

查看已创建的密钥。

kubectl get secrets

回显如下，表明密钥创建成功。

```
NAME                  TYPE                DATA   AGE
ingress-test-secret   IngressTLS          2      13s
```

步骤4 创建名为“**ingress-test.yaml**”的YAML文件，此处文件名可自定义。

vi ingress-test.yaml

说明

默认安全策略选择（kubernetes.io/elb.tls-ciphers-policy）仅在1.17.17及以上版本的集群中支持。

以自动创建关联ELB为例，YAML文件配置如下：

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.autocreate:
      '{
          "type": "public",
          "bandwidth_name": "cce-bandwidth-******",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              "eu-west-0a"
          ],
          "elb_virsubnet_ids":["b4bf8152-6c36-4c3b-9f74-2229f8e640c9"],
```

```
        "l7_flavor_name": "L7_flavor.elb.s1.small"
    }'
  kubernetes.io/elb.tls-ciphers-policy: tls-1-2
spec:
  tls:
  - secretName: ingress-test-secret
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.autocreate:
      '{
          "type": "public",
          "bandwidth_name": "cce-bandwidth-******",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              "eu-west-0a"
          ],
          "elb_virsubnet_ids":["b4bf8152-6c36-4c3b-9f74-2229f8e640c9"],
          "l7_flavor_name": "L7_flavor.elb.s1.small"
      }'
    kubernetes.io/elb.tls-ciphers-policy: tls-1-2
spec:
  tls:
  - secretName: ingress-test-secret
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: <your_service_name>  #替换为您的目标服务名称
            port:
              number: 8080              #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```

表 11-33 关键参数说明

参数	是否必填	参数类型	描述
kubernetes.io/elb.tls-ciphers-policy	否	String	默认值为“tls-1-2”，为监听器使用的默认安全策略，仅在HTTPS协议下生效。 取值范围： • tls-1-0 • tls-1-1 • tls-1-2 • tls-1-2-strict 各安全策略使用的加密套件列表详细参见**表11-34**。
tls	否	Array of strings	HTTPS协议时，需添加此字段用于指定密钥证书。 该字段支持添加多项独立的域名和证书，详见**ELB Ingress配置服务器名称指示（SNI）**。
secretName	否	String	HTTPS协议时添加，配置为创建的密钥证书名称。

表 11-34 tls_ciphers_policy 取值说明

安全策略	支持的TLS版本类型	使用的加密套件列表
tls-1-0	TLS 1.2 TLS 1.1 TLS 1.0	ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:AES128-GCM-SHA256:AES256-GCM-SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:AES128-SHA256:AES256-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES128-SHA:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:AES128-SHA:AES256-SHA
tls-1-1	TLS 1.2 TLS 1.1	
tls-1-2	TLS 1.2	

安全策略	支持的TLS版本类型	使用的加密套件列表
tls-1-2-strict	TLS 1.2	ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:AES128-GCM-SHA256:AES256-GCM-SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:AES128-SHA256:AES256-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384

步骤5 创建Ingress。

kubectl create -f ingress-test.yaml

回显如下，表示Ingress服务已创建。

```
ingress/ingress-test created
```

查看已创建的Ingress。

kubectl get ingress

回显如下，表示Ingress服务创建成功，工作负载可访问。

```
NAME           HOSTS     ADDRESS         PORTS   AGE
ingress-test   *         121.**.**.**    80      10s
```

步骤6 访问工作负载（例如**Nginx工作负载**），在浏览器中输入安全访问地址https://121.**.**.**:443进行验证。

其中，121.**.**.**为统一负载均衡实例的IP地址。

----结束

使用 ELB 服务中的证书

使用ELB服务中的证书，可以通过指定kubernetes.io/elb.tls-certificate-ids这个annotations实现。

说明

1. 当同时指定annotation中已有证书和IngressTLS时，使用ELB服务中的证书。
2. CCE不校验ELB服务中的证书是否有效，只校验证书是否存在。
3. 仅v1.19.16-r2、v1.21.5-r0、v1.23.3-r0及以上版本的集群支持使用ELB服务中的证书。

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.id: 0b9a6c4d-bd8b-45cc-bfc8-ff0f9da54e95
    kubernetes.io/elb.class: union
```

```
  kubernetes.io/elb.tls-certificate-ids:
058cc023690d48a3867ad69dbe9cd6e5,b98382b1f01c473286653afd1ed9ab63
spec:
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.id: 0b9a6c4d-bd8b-45cc-bfc8-ff0f9da54e95
    kubernetes.io/elb.class: union
    kubernetes.io/elb.tls-certificate-ids:
058cc023690d48a3867ad69dbe9cd6e5,b98382b1f01c473286653afd1ed9ab63
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: '/'
            backend:
              service:
                name: <your_service_name>  #替换为您的目标服务名称
                port:
                  number: 8080              #替换为您的目标服务端口
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: cce
```

11.4.2.5 ELB Ingress 配置服务器名称指示（SNI）

SNI允许同一个IP地址和端口号下对外提供多个基于TLS的访问域名，且不同的域名可以使用不同的安全证书。

说明

- 该功能仅支持1.15.11及以上版本的集群。
- 当使用HTTPS协议时，才支持配置SNI。
- 用于SNI的证书需要指定域名，每个证书只能指定一个域名。支持泛域名证书。
- 安全策略选择（kubernetes.io/elb.tls-ciphers-policy）仅在1.17.11及以上版本的集群中支持。

满足以上条件时可进行SNI配置，以自动创建关联ELB为例，yaml文件配置如下，本例中**sni-test-secret-1**、**sni-test-secret-2**为SNI证书，该证书指定的域名必须与证书中的域名一致。

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
```

```
    kubernetes.io/elb.class: performance
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.autocreate:
      '{
          "type": "public",
          "bandwidth_name": "cce-bandwidth-******",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              "eu-west-0a"
          ],
          "elb_virsubnet_ids":["b4bf8152-6c36-4c3b-9f74-2229f8e640c9"],
          "l7_flavor_name": "L7_flavor.elb.s1.small"
      }'
    kubernetes.io/elb.tls-ciphers-policy: tls-1-2
spec:
  tls:
  - secretName: ingress-test-secret
  - hosts:
      - example.top  #签发证书时指定域名为example.top
    secretName: sni-test-secret-1
  - hosts:
      - example.com  #签发证书时指定域名为example.com
    secretName: sni-test-secret-2
  rules:
  - host: example.com
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.autocreate:
      '{
          "type": "public",
          "bandwidth_name": "cce-bandwidth-******",
          "bandwidth_chargemode": "bandwidth",
          "bandwidth_size": 5,
          "bandwidth_sharetype": "PER",
          "eip_type": "5_bgp",
          "available_zone": [
              "eu-west-0a"
          ],
          "elb_virsubnet_ids":["b4bf8152-6c36-4c3b-9f74-2229f8e640c9"],
          "l7_flavor_name": "L7_flavor.elb.s1.small"
      }'
    kubernetes.io/elb.tls-ciphers-policy: tls-1-2
spec:
  tls:
  - secretName: ingress-test-secret
  - hosts:
      - example.top  #签发证书时指定域名为example.top
    secretName: sni-test-secret-1
  - hosts:
      - example.com  #签发证书时指定域名为example.com
    secretName: sni-test-secret-2
```

```
rules:
- host: example.com
  http:
    paths:
    - path: '/'
      backend:
        service:
          name: <your_service_name>  #替换为您的目标服务名称
          port:
            number: 8080             #替换为您的目标服务端口
      property:
        ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
      pathType: ImplementationSpecific
ingressClassName: cce
```

11.4.2.6 ELB Ingress 路由到多个服务

Ingress可通过不同的匹配策略同时路由到多个后端服务，YAML文件中的**spec**字段设置如下。通过访问“www.example.com/foo”、“www.example.com/bar”、“foo.example.com/”即可分别路由到三个不同的后端Service。

须知

Ingress转发策略中注册的URL需与后端应用提供访问的URL一致，否则将返回404错误。

```
...
spec:
  rules:
  - host: 'www.example.com'
    http:
      paths:
      - path: '/foo'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
      - path: '/bar'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
  - host: 'foo.example.com'
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

11.4.2.7 ELB Ingress 使用 HTTP/2

Ingress支持HTTP/2的方式暴露服务，在默认情况下，客户端与负载均衡之间采用HTTP1.X协议，若需开启HTTP2功能，可在annotation字段中加入如下配置：

```
kubernetes.io/elb.http2-enable: 'true'
```

以关联已有ELB为例，yaml配置文件如下。

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.id: <your_elb_id>  #替换为您已有的ELB ID
    kubernetes.io/elb.ip: <your_elb_ip>  #替换为您已有的ELB IP
    kubernetes.io/elb.port: '443'
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.http2-enable: 'true' # 开启HTTP/2功能
spec:
  tls:
  - secretName: ingress-test-secret
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: 80                   #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/elb.id: <your_elb_id>  #替换为您已有的ELB ID
    kubernetes.io/elb.ip: <your_elb_ip>  #替换为您已有的ELB IP
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.http2-enable: 'true' # 开启HTTP/2功能
spec:
  tls:
  - secretName: ingress-test-secret
  rules:
  - host: ''
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: <your_service_name>  #替换为您的目标服务名称
            port:
              number: 8080             #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```

表6 HTTP/2参数说明

参数	是否必填	参数类型	描述
kubernetes.io/elb.http2-enable	否	Bool	表示HTTP/2功能的开启状态。开启后，可提升客户端与LB间的访问性能，但LB与后端服务器间仍采用HTTP1.X协议。**v1.19.16-r0、v1.21.3-r0及以上版本的集群支持此字段**。 取值范围： ● true：开启HTTP/2功能； ● false：关闭HTTP/2功能（默认为关闭状态）。 注意：**只有当监听器的协议为HTTPS时，才支持开启或关闭HTTP/2功能**。当监听器的协议为HTTP时，该字段无效，默认将其设置为false。

11.4.2.8 ELB Ingress 对接 HTTPS 协议的后端服务

Ingress可以对接不同协议的后端服务，在默认情况下Ingress的后端代理通道是HTTP协议的，若需要建立HTTPS协议的通道，可在annotation字段中加入如下配置：

```
kubernetes.io/elb.pool-protocol: https
```

约束与限制

- 仅支持v1.23.8和v1.25.3及以上集群版本使用该能力。
- 仅使用独享型ELB时，Ingress支持对接HTTPS协议的后端服务。
- 对接HTTPS协议的后端服务时，Ingress的对外协议也需要选择HTTPS。

对接 HTTPS 协议的后端服务

Ingress配置示例如下：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.port: '443'
    kubernetes.io/elb.id: <your_elb_id>    #本示例中使用已有的独享型ELB，请替换为您的独享型ELB ID
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.pool-protocol: https  # 对接HTTPS协议的后端服务
    kubernetes.io/elb.tls-ciphers-policy: tls-1-2
spec:
  tls:
    - secretName: ingress-test-secret
  rules:
    - host: ''
      http:
        paths:
          - path: '/'
            backend:
              service:
                name: <your_service_name>  #替换为您的目标服务名称
                port:
```

```
            number: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```

11.4.2.9 ELB Ingress 设置超时时间

ELB Ingress支持设置以下超时时间：

- 客户端连接空闲超时时间：没有收到客户端请求的情况下保持连接的最长时间。如果在这个时间内没有新的请求， 负载均衡会暂时中断当前连接，直到下一次请求时重新建立新的连接。
- 等待客户端请求超时时间：如果在规定的时间内客户端没有发送完请求头，或body体数据发送间隔超过一定时间，负载均衡会自动关闭连接。
- 等待后端服务器响应超时时间：向后端服务器发送请求后，如果在一定时间内没有收到响应，负载均衡将返回504错误码。

约束与限制

- 该特性存在集群版本限制，仅在以下版本中生效：
  - v1.19集群：v1.19.16-r30及以上版本
  - v1.21集群：v1.21.10-r10及以上版本
  - v1.23集群：v1.23.8-r10及以上版本
  - v1.25集群：v1.25.3-r10及以上版本
- 仅在使用独享型ELB时，Ingress支持设置超时时间。
- 更新Ingress时，如果删除超时时间配置，不会修改已有监听器的超时时间配置。

设置超时时间

Ingress配置示例如下：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test
  namespace: default
  annotations:
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.id: <your_elb_id>    #本示例中使用已有的独享型ELB，请替换为您的独享型ELB ID
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.keepalive_timeout: '300'  # 客户端连接空闲超时时间
    kubernetes.io/elb.client_timeout: '60'      # 等待客户端请求超时时间
    kubernetes.io/elb.member_timeout: '60'      # 等待后端服务器响应超时时间
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: /
            backend:
              service:
                name: test
                port:
                  number: 80
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: cce
```

表 11-35 annotation 关键参数说明

参数	是否必填	参数类型	描述
kubernetes.io/elb.keepalive_timeout	否	Integer	客户端连接空闲超时时间，在超过keepalive_timeout时长一直没有请求，负载均衡会暂时中断当前连接，直到下一次请求时重新建立新的连接。 取值范围为0-4000s，默认值为60s。
kubernetes.io/elb.client_timeout	否	Integer	等待客户端请求超时时间，包括两种情况： • 读取整个客户端请求头的超时时长：如果客户端未在超时时长内发送完整个请求头，则请求将被中断。 • 两个连续body体的数据包到达LB的时间间隔，超出client_timeout将会断开连接。 取值范围为1-300s，默认值为60s。
kubernetes.io/elb.member_timeout	否	Integer	等待后端服务器响应超时时间。请求转发后端服务器后，等待超过member_timeout时长没有响应，负载均衡将终止等待，并返回 HTTP504错误码。 取值范围为1-300s，默认值为60s。

11.4.3 Nginx Ingress 管理

11.4.3.1 通过控制台创建 Nginx Ingress

前提条件

- Ingress为后端工作负载提供网络访问，因此集群中需提前部署可用的工作负载。若您无可用工作负载，可参考**创建无状态负载（Deployment）**、**创建有状态负载（StatefulSet）**或**创建守护进程集（DaemonSet）**部署工作负载。
- 为上述工作负载配置ClusterIP类型或NodePort类型的Service，可参考**集群内访问（ClusterIP）**或**节点访问（NodePort）**配置示例Service。
- 添加Nginx Ingress时，需在集群中提前安装nginx-ingress插件，具体操作可参考**安装插件**。

注意事项

- **不建议在ELB服务页面修改ELB实例的任何配置，否则将导致服务异常。**如果您已经误操作，请卸载Nginx Ingress插件后重装。
- Ingress转发策略中注册的URL需与后端应用提供访问的URL一致，否则将返回404错误。

- 负载均衡实例需与当前集群处于相同VPC 且为相同公网或私网类型。
- 负载均衡实例需要拥有至少两个监听器配额，且端口80和443没有被监听器占用。

添加 Nginx Ingress

本节以Nginx作为工作负载并添加Nginx Ingress为例进行说明。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 选择左侧导航栏的“服务发现”，在右侧选择“路由”页签，单击右上角“创建路由”。

步骤3 设置Ingress参数。

- **名称：**自定义Ingress名称，例如nginx-ingress-demo。
- **命名空间：**选择需要添加Ingress的命名空间。
- **对接Nginx：**集群中已安装**NGNIX Ingress Controller**插件后显示此选项，未安装nginx-ingress模板时本选项不显示。

  单击  开启后将对接nginx-ingress提供7层访问，可配置如下参数。

  TLS配置：nginx-ingress支持HTTP和HTTPS，安装nginx-ingress时预留的监听端口，默认HTTP为80，HTTPS为443。使用HTTPS需要配置相关证书。
  - 服务器证书：创建HTTPS协议监听时需要绑定TLS类型的密钥证书，以支持HTTPS数据传输加密认证，创建密钥的方法请参见**创建密钥**。
  - SNI：SNI（Server Name Indication）是TLS的扩展协议，在该协议下允许同一个IP地址和端口号下对外提供多个基于TLS的访问域名，且不同的域名可以使用不同的安全证书。开启SNI后，允许客户端在发起TLS握手请求时就提交请求的域名信息。负载均衡收到TLS请求后，会根据请求的域名去查找证书：若找到域名对应的证书，则返回该证书认证鉴权；否则，返回缺省证书（服务器证书）认证鉴权。
- **转发策略配置：**请求的访问地址与转发规则匹配时（转发规则由域名、URL组成），此请求将被转发到对应的目标Service处理。单击“添加转发策略”按钮可添加多条转发策略。
  - 域名：实际访问的域名地址。请确保所填写的域名已注册并备案，在Ingress创建完成后，将域名与自动创建的负载均衡实例的IP（即Ingress访问地址的IP部分）绑定。一旦配置了域名规则，则必须使用域名访问。
  - URL匹配规则：
    - 默认：默认为前缀匹配。
    - 前缀匹配：例如映射URL为/healthz，只要符合此前缀的URL均可访问。例如/healthz/v1，/healthz/v2。
    - 精确匹配：表示只有URL完全匹配时，访问才能生效。例如映射URL为/healthz，则必须为此URL才能访问。
  - URL：需要注册的访问路径，例如：/healthz。

说明

- Nginx Ingress的访问路径匹配规则是基于“/”符号分隔的路径前缀匹配，并区分大小写。只要访问路径以“/”符号分隔后的子路径匹配此前缀，均可正常访问，但如果该前缀仅是子路径中的部分字符串，则不会匹配。例如URL设置为/healthz，则匹配/healthz/v1，但不匹配/healthzv1。
- 此处添加的访问路径要求后端应用内存在相同的路径，否则转发无法生效。
  例如，Nginx应用默认的Web访问路径为“/usr/share/nginx/html”，在为Ingress转发策略添加“/test”路径时，需要应用的Web访问路径下也包含相同路径，即“/usr/share/nginx/html/test”，否则将返回404。

  - 目标服务名称：请选择已有Service或新建Service。页面列表中的查询结果已自动过滤不符合要求的Service。
  - 目标服务访问端口：可选择目标Service的访问端口。
  - 操作：可单击“删除”按钮删除该配置。
- **注解**：以“key: value”形式设置，可通过**Annotations**查询nginx-ingress支持的配置。

步骤4 配置完成后，单击“确定”。

创建完成后，在Ingress列表可查看到已添加的Ingress。

----结束

11.4.3.2 通过 Kubectl 命令行创建 Nginx Ingress

操作场景

本节以**Nginx工作负载**为例，说明kubectl命令添加Nginx Ingress的方法。

前提条件

- 集群必须已安装nginx-ingress插件，具体操作可参考**安装插件**。
- Ingress为后端工作负载提供网络访问，因此集群中需提前部署可用的工作负载。若您无可用工作负载，可参考**创建无状态负载（Deployment）**、**创建有状态负载（StatefulSet）**或**创建守护进程集（DaemonSet）**部署工作负载。
- 为上述工作负载配置ClusterIP类型或NodePort类型的Service，可参考**集群内访问（ClusterIP）**或**节点访问（NodePort）**配置示例Service。

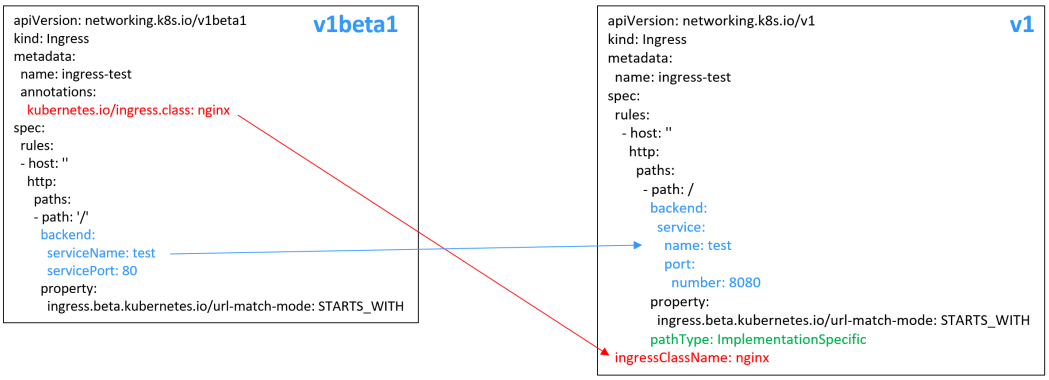
networking.k8s.io/v1 版本 Ingress 说明

CCE在v1.23版本集群开始Ingress切换到**networking.k8s.io/v1**版本。

v1版本参数相较v1beta1参数有如下区别。

- ingress类型由annotations中**kubernetes.io/ingress.class**变为使用**spec.ingressClassName**字段。
- **backend**的写法变化。
- 每个路径下必须指定路径类型**pathType**，支持如下类型。
  - ImplementationSpecific: 对于这种路径类型，匹配方法取决于具体Ingress Controller的实现。在CCE中会使用ingress.beta.kubernetes.io/url-match-mode指定的匹配方式，这与v1beta1方式相同。

- Exact：精确匹配 URL 路径，且区分大小写。
- Prefix：基于以 / 分隔的 URL 路径前缀匹配。匹配区分大小写，并且对路径中的元素逐个匹配。 路径元素指的是由 / 分隔符分隔的路径中的标签列表。



添加 Nginx Ingress

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 创建名为“**ingress-test.yaml**”的YAML文件，此处文件名可自定义。

vi ingress-test.yaml

说明

CCE在1.23版本集群开始Ingress切换到networking.k8s.io/v1版本，之前版本集群使用networking.k8s.io/v1beta1。v1版本与v1beta1版本的区别请参见**networking.k8s.io/v1版本Ingress说明**。

以HTTP协议访问为例，YAML文件配置如下。

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: /
            backend:
              service:
                name: <your_service_name>  #替换为您的目标服务名称
                port:
                  number: <your_service_port>  #替换为您的目标服务端口
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: nginx  # 表示使用Nginx Ingress
```

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx  # 表示使用Nginx Ingress
spec:
```

```
rules:
  - host: ''
    http:
      paths:
        - path: '/'
          backend:
            serviceName: <your_service_name>  #替换为您的目标服务名称
            servicePort: <your_service_port>  #替换为您的目标服务端口
```

表 11-36 关键参数说明

参数	是否必填	参数类型	描述
kubernetes.io/ingress.class	是（仅1.21及以下集群）	String	nginx：表示使用NginxIngress，未安装nginx-ingress插件时无法使用。 通过API接口创建Ingress时必须增加该参数。
ingressClassName	是 （仅1.23及以上集群）	String	nginx：表示使用NginxIngress，未安装nginx-ingress插件时无法使用。 通过API接口创建Ingress时必须增加该参数。
host	否	String	为服务访问域名配置，默认为""，表示域名全匹配。请确保所填写的域名已注册并备案，一旦配置了域名规则后，必须使用域名访问。
path	是	String	为路由路径，用户自定义设置。所有外部访问请求需要匹配host和path。 **说明** ● Nginx Ingress的访问路径匹配规则是基于“/”符号分隔的路径前缀匹配，并区分大小写。只要访问路径以“/”符号分隔后的子路径匹配此前缀，均可正常访问，但如果该前缀仅是子路径中的部分字符串，则不会匹配。例如URL设置为/healthz，则匹配/healthz/v1，但不匹配/healthzv1。 ● 此处添加的访问路径要求后端应用内存在相同的路径，否则转发无法生效。例如，Nginx应用默认的Web访问路径为“/usr/share/nginx/html”，在为Ingress转发策略添加“/test”路径时，需要应用的Web访问路径下也包含相同路径，即“/usr/share/nginx/html/test”，否则将返回404。
ingress.beta.kubernetes.io/url-match-mode	否	String	路由匹配策略。 默认值为“STARTS_WITH”(前缀匹配)。 取值范围： ● EQUAL_TO：精确匹配 ● STARTS_WITH：前缀匹配

参数	是否必填	参数类型	描述
pathType	是	String	路径类型，该字段仅v1.23及以上集群支持。 • ImplementationSpecific: 匹配方法取决于具体Ingress Controller的实现。在CCE中会使用ingress.beta.kubernetes.io/url-match-mode指定的匹配方式。 • Exact：精确匹配 URL 路径，且区分大小写。 • Prefix：前缀匹配，且区分大小写。该方式是将URL路径通过“/”分隔成多个元素 ，并且对元素进行逐个匹配。 如果URL中的每个元素均和路径匹配，则说明该URL的子路径均可以正常路由。 **说明** – Prefix匹配时每个元素均需精确匹配，如果URL的最后一个元素是请求路径中最后一个元素的子字符串，则不会匹配 。例如：/foo/bar匹配/foo/bar/baz，但不匹配/foo/barbaz。 – 通过“/”分隔元素时，若URL或请求路径以“/”结尾，将会忽略结尾的“/”。例如：/foo/bar会匹配/foo/bar/。 关于Ingress路径匹配示例，请参见**示例**。

步骤3 创建Ingress。

kubectl create -f ingress-test.yaml

回显如下，表示Ingress服务已创建。

```
ingress/ingress-test created
```

查看已创建的Ingress。

kubectl get ingress

回显如下，表示Ingress服务创建成功，工作负载可访问。

```
NAME           HOSTS     ADDRESS         PORTS   AGE
ingress-test   *         121.**.**.**    80      10s
```

步骤4 访问工作负载（例如**Nginx工作负载**），在浏览器中输入访问地址“http://121.**.**.**:80”进行验证。

其中，“121.**.**.**”为统一负载均衡实例的IP地址。

----结束

11.4.3.3 Nginx Ingress 配置 HTTPS 证书

Ingress支持配置HTTPS证书以提供安全服务。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 Ingress支持使用kubernetes.io/tls和IngressTLS两种TLS密钥类型，此处以IngressTLS类型为例，详情请参见**创建密钥**。kubernetes.io/tls类型的密钥示例及说明请参见**TLS Secret**。

执行如下命令，创建名为“**ingress-test-secret.yaml**”的YAML文件，此处文件名可自定义。

vi ingress-test-secret.yaml

YAML文件配置如下：

```
apiVersion: v1
data:
  tls.crt: LS0******tLS0tCg==
  tls.key: LS0tL******0tLS0K
kind: Secret
metadata:
  annotations:
    description: test for ingressTLS secrets
  name: ingress-test-secret
  namespace: default
type: IngressTLS
```

说明

此处tls.crt和tls.key为示例，请获取真实的证书和密钥进行替换。tls.crt和tls.key的值为Base64编码后的内容。

步骤3 创建密钥。

kubectl create -f ingress-test-secret.yaml

回显如下，表明密钥已创建。

```
secret/ingress-test-secret created
```

查看已创建的密钥。

kubectl get secrets

回显如下，表明密钥创建成功。

```
NAME                  TYPE                DATA   AGE
ingress-test-secret   IngressTLS          2      13s
```

步骤4 创建名为“**ingress-test.yaml**”的YAML文件，此处文件名可自定义。

vi ingress-test.yaml

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
spec:
  tls:
  - hosts:
    - foo.bar.com
    secretName: ingress-test-secret  #替换为您的TLS密钥证书
  rules:
```

```
  - host: foo.bar.com
    http:
      paths:
        - path: /
          backend:
            service:
              name: <your_service_name>  #替换为您的目标服务名称
              port:
                number: <your_service_port>  #替换为您的目标服务端口
          property:
            ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
          pathType: ImplementationSpecific
  ingressClassName: nginx
```

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  tls:
  - hosts:
    - foo.bar.com
    secretName: ingress-test-secret   #替换为您的TLS密钥证书
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: '/'
        backend:
          serviceName: <your_service_name>  #替换为您的目标服务名称
          servicePort: <your_service_port>  #替换为您的目标服务端口
  ingressClassName: nginx
```

步骤5 创建Ingress。

kubectl create -f ingress-test.yaml

回显如下，表示Ingress服务已创建。

```
ingress/ingress-test created
```

查看已创建的Ingress。

kubectl get ingress

回显如下，表示Ingress服务创建成功，工作负载可访问。

```
NAME           HOSTS     ADDRESS         PORTS   AGE
ingress-test   *         121.**.**.**    80      10s
```

步骤6 访问工作负载（例如**Nginx工作负载**），在浏览器中输入安全访问地址https://121.**.**.**:443进行验证。

其中，121.**.**.**为统一负载均衡实例的IP地址。

----结束

11.4.3.4 Nginx Ingress 配置 URL 重写规则

在一些使用场景中后端服务提供访问的URL与Ingress规则中指定的路径不同，而Ingress会将访问路径直接转发到后端相同路径，如果不进行URL重写配置，所有访问都将返回404。例如，Ingress规则中的访问路径设置为/app/demo，而后端服务提供的访问路径为/demo，在实际访问Ingress时会直接转发到后端服务的/app/demo路径，与后端实际提供的访问路径（/demo）不匹配，导致404的情况发生。

此时，您可以通过Rewrite方法实现URL重写，即使用“nginx.ingress.kubernetes.io/rewrite-target”注解可以实现不同路径的重写规则。

配置重写规则

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  rules:
    - host: 'rewrite.bar.com'
      http:
        paths:
          - path: '/something(/|$)(.*)'
            backend:
              service:
                name: <your_service_name>  #替换为您的目标服务名称
                port:
                  number: <your_service_port>  #替换为您的目标服务端口
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: nginx
```

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  rules:
    - host: 'rewrite.bar.com'
      http:
        paths:
          - path: '/something(/|$)(.*)'
            backend:
              serviceName: <your_service_name>  #替换为您的目标服务名称
              servicePort: <your_service_port>  #替换为您的目标服务端口
```

说明

只要有一个Ingress使用了rewrite-target，则所有Ingress定义下同一个host下所有path都会正则大小写敏感，包括没有使用rewrite-target的Ingress。

以上示例中，占位符$2表示将第二个括号即(.*)中匹配到的所有字符填写到“nginx.ingress.kubernetes.io/rewrite-target”注解中。

例如，上面的Ingress定义将导致以下重写：

- rewrite.bar.com/something 重写为 rewrite.bar.com/
- rewrite.bar.com/something/ 重写为 rewrite.bar.com/
- rewrite.bar.com/something/new 重写为 rewrite.bar.com/new

nginx-ingress-controller容器中，“/etc/nginx”路径下的nginx.conf文件可查看所有Ingress配置。以上示例中的重写规则将生成一条Rewrite指令，并写入到nginx.conf的location字段中，如下所示：

```
## start server rewrite.bar.com
    server {
        server_name rewrite.bar.com ;
        ...
        location ~* "^/something(/|$)(.*)" {
            set $namespace      "default";
            set $ingress_name   "ingress-test";
            set $service_name   "<your_service_name>";
            set $service_port   "80";
            ...
            rewrite "(?i)/something(/|$)(.*)" /$2 break;
            ...
        }
    }
    ## end server rewrite.bar.com
```

上面的Rewrite指令基本语法结构为：

```
rewrite regex replacement [flag];
```

- regex：匹配URI的正则表达式。在上述例子中，“(?i)/something(/|$)(.*)”即为匹配URI的正则表达式，其中“(?i)”表示不区分大小写。
- replacement：重写内容。在上述例子中，“/$2”即为重写内容，表示把路径重写为第二个括号“(.*)”中匹配到的所有字符。
- flag：表示重写形式的标记，包括：
  - last：表示本条规则匹配完成后继续向下匹配。
  - break：表示本条规则匹配完成后停止匹配。
  - redirect：表示临时重定向，返回状态码302。
  - permanent：表示永久重定向，返回状态码301。

高级 Rewrite 配置

对于一些复杂高级的Rewrite需求，可以通过如下注解来实现，其本质也是修改Nginx的配置文件（nginx.conf），可以实现上面提到的“nginx.ingress.kubernetes.io/rewrite-target”注解的功能，但是自定义程度更高，适合更加复杂的Rewrite需求。

- nginx.ingress.kubernetes.io/server-snippet：在nginx.conf的“server”字段中添加自定义配置。
- nginx.ingress.kubernetes.io/configuration-snippet：在nginx.conf的“location”字段中添加自定义配置。

通过以上两个注解可以在nginx.conf中的“server”或“location”字段中插入Rewrite指令，完成URL的重写，示例如下：

```
annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/configuration-snippet: |
      rewrite ^/stylesheets/(.*)$ /something/stylesheets/$1 redirect;  # 添加 /something 前缀
      rewrite ^/images/(.*)$ /something/images/$1 redirect;  # 添加 /something 前缀
```

如上两条规则在访问URL中添加了“/something”路径，即：

- 当用户访问rewrite.bar.com/stylesheets/new.css时，重写为rewrite.bar.com/something/stylesheets/new.css
- 当用户访问rewrite.bar.com/images/new.jpg时，重写为rewrite.bar.com/something/images/new.jpg

HTTP 重定向到 HTTPS

默认情况下，若Ingress使用了TLS，通过HTTP访问时，请求将会被重定向（状态码为308）到HTTPS。您也可以通过以下注解强制重定向至HTTPS。

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: 'true'
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: /
            backend:
              service:
                name: <your_service_name>  #替换为您的目标服务名称
                port:
                  number: <your_service_port>  #替换为您的目标服务端口
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: nginx
```

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: 'true'
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: /
            backend:
              serviceName: <your_service_name>  #替换为您的目标服务名称
              servicePort: <your_service_port>  #替换为您的目标服务端口
```

11.4.3.5 Nginx Ingress 对接 HTTPS 协议的后端服务

Ingress可以代理不同协议的后端服务，在默认情况下Ingress的后端代理通道是HTTP协议的，若需要建立HTTPS协议的通道，可在annotation字段中加入如下配置：

```
nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
```

Ingress配置示例如下：

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
spec:
  tls:
```

```
  - secretName: ingress-test-secret  #替换为您的TLS密钥证书
 rules:
  - host: ''
    http:
      paths:
        - path: '/'
          backend:
            service:
              name: <your_service_name>  #替换为您的目标服务名称
              port:
                number: <your_service_port>  #替换为您的目标服务端口
          property:
            ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
          pathType: ImplementationSpecific
 ingressClassName: nginx
```

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
spec:
  tls:
    - secretName: ingress-test-secret  #替换为您的TLS密钥证书
  rules:
    - host: ''
      http:
        paths:
          - path: '/'
            backend:
              serviceName: <your_service_name>  #替换为您的目标服务名称
              servicePort: <your_service_port>  #替换为您的目标服务端口
```

11.4.3.6 Nginx Ingress 使用一致性哈希负载均衡

原生的Nginx支持多种负载均衡规则，其中常用的有加权轮询、IP hash等。Nginx Ingress在原生的Nginx能力基础上，支持使用一致性哈希方法进行负载均衡。

Nginx默认支持的IP hash方法使用的是线性的hash空间，根据IP的hash运算值来选取后端的目标服务器。但是这种方法在添加删除节点时，所有IP值都需要重新进行hash运算，然后重新路由，这样的话就会导致大面积的会话丢失或缓存失效，因此Nginx Ingress引入了一致性哈希来解决这一问题。

一致性哈希是一种特殊的哈希算法，通过构建环状的hash空间来替代普通的线性hash空间，在增删节点时仅需要将路由的目标按顺时针原则向下迁移，而其他路由无需改变，可以尽可能地减少重新路由，有效解决动态增删节点带来的负载均衡问题。

通过配置一致性哈希规则，在增加一台服务器时，新的服务器会尽量分担其他所有服务器的压力；同样，在减少一台服务器时，其他所有服务器也可以尽量分担它的资源，可以有效减少集群局部节点的压力，防止由于某一节点宕机带来的集群雪崩效应。

配置一致性哈希规则

Nginx Ingress可以通过“nginx.ingress.kubernetes.io/upstream-hash-by”注解实现一致性哈希规则的配置，如下所示：

1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: ingress-test
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/upstream-hash-by: "$request_uri"  #按照请求uri进行hash
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: '/'
            backend:
              service:
                name: <your_service_name>  #替换为您的目标服务名称
                port:
                  number: <your_service_port>  #替换为您的目标服务端口
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: nginx
```

1.21及以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/upstream-hash-by: "$request_uri"  #按照请求uri进行hash
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: '/'
            backend:
              serviceName: <your_service_name>  #替换为您的目标服务名称
              servicePort: <your_service_port>  #替换为您的目标服务端口
```

注解“nginx.ingress.kubernetes.io/upstream-hash-by”的参数值支持nginx参数、文本值或任意组合，例如：

- nginx.ingress.kubernetes.io/upstream-hash-by: "$request_uri"代表按照请求uri进行hash。
- nginx.ingress.kubernetes.io/upstream-hash-by: "$request_uri$host"代表按照请求uri和域名进行hash。
- nginx.ingress.kubernetes.io/upstream-hash-by: "${request_uri}-text-value"代表按照请求uri和文本值进行hash。

相关文档

Custom NGINX upstream hashing

11.4.3.7 使用 Annotation 配置 Nginx Ingress

CCE的Nginx Ingress插件使用社区模板与镜像，Nginx Ingress默认的其他参数无法满足业务需求时，也可通过添加注解Annotation（注解）的方式自定义参数，例如默认后端、超时时间、请求body体大小等。

本文介绍在创建Nginx类型的Ingress时常用的Annotation。

说明

- 注解的键值只能是字符串，其他类型（如布尔值或数值）必须使用引号，例如"true"、"false"、"100"。
- Nginx Ingress支持社区的原生注解，详情请参考**Annotations**。

- **Ingress类型**
- **配置URL重写规则**
- **对接HTTPS协议的后端服务**
- **创建一致性哈希负载均衡规则**
- **自定义超时时长**
- **自定义Body体大小**
- **相关文档**

Ingress 类型

表 11-37 Ingress 类型注解

参数	类型	描述	支持的集群版本
kubernetes.io/ingress.class	String	• nginx：表示使用Nginx Ingress。 • cce：表示使用自研ELB Ingress。 通过API接口创建Ingress时必须增加该参数。 v1.23及以上集群使用ingressClassName参数代替，详情请参见**通过Kubectl命令行创建Nginx Ingress**。	仅v1.21及以下集群

上述注解的使用方法详情请参见**通过Kubectl命令行创建Nginx Ingress**。

配置 URL 重写规则

表 11-38 URL 重写规则注解

参数	类型	描述
nginx.ingress.kubernetes.io/rewrite-target	String	重定向流量的目标URI。
nginx.ingress.kubernetes.io/ssl-redirect	Bool	是否只能通过SSL访问（当Ingress包含证书时默认为true）。
nginx.ingress.kubernetes.io/force-ssl-redirect	Bool	是否强制重定向到HTTPS，即使Ingress未启用TLS。通过HTTP访问时，请求将会被强制重定向（状态码为308）到HTTPS。

具体使用场景和说明请参见**Nginx Ingress配置URL重写规则**。

对接 HTTPS 协议的后端服务

表 11-39 对接 HTTPS 协议的后端服务注解

参数	类型	描述
nginx.ingress.kubernetes.io/backend-protocol	String	参数值为'HTTPS'，表示使用HTTPS协议转发请求到后端业务容器。

具体使用场景和说明请参见**Nginx Ingress对接HTTPS协议的后端服务**。

创建一致性哈希负载均衡规则

表 11-40 一致性哈希负载均衡注解

参数	类型	描述
nginx.ingress.kubernetes.io/upstream-hash-by	String	为后端启用一致性哈希进行负载均衡，参数值支持nginx参数、文本值或任意组合，例如： • nginx.ingress.kubernetes.io/upstream-hash-by: "$request_uri"代表按照请求uri进行hash。 • nginx.ingress.kubernetes.io/upstream-hash-by: "$request_uri$host"代表按照请求uri和域名进行hash。 • nginx.ingress.kubernetes.io/upstream-hash-by: "${request_uri}-text-value"代表按照请求uri和文本值进行hash。

具体使用场景和说明请参见**Nginx Ingress使用一致性哈希负载均衡**。

自定义超时时长

表 11-41 自定义超时时长注解

参数	类型	描述
nginx.ingress.kubernetes.io/proxy-connect-timeout	String	自定义连接超时时长，设置超时值时无需填写单位，默认单位为秒。 例如： `nginx.ingress.kubernetes.io/proxy-connect-timeout: '120'`

自定义 Body 体大小

表 11-42 自定义 Body 体大小注解

参数	类型	描述
nginx.ingress.kubernetes.io/proxy-body-size	String	当请求中的Body体大小超过允许的最大值时，将向客户端返回413错误，您可通过该参数调整Body体的限制大小。 例如： nginx.ingress.kubernetes.io/proxy-body-size: 8m

相关文档

更多关于Nginx Ingress支持的注解参数，请参见**Annotations**。

11.5 DNS

11.5.1 DNS 概述

CoreDNS 介绍

创建集群时会安装**CoreDNS插件**，CoreDNS是用来做集群内部域名解析。

在kube-system命名空间下可以查看到CoreDNS的Pod。

```
$ kubectl get po --namespace=kube-system
NAME                                READY   STATUS    RESTARTS   AGE
coredns-7689f8bdf-295rk             1/1     Running   0          9m11s
coredns-7689f8bdf-h7n68             1/1     Running   0          11m
```

CoreDNS安装成功后会成为DNS服务器，当创建Service后，CoreDNS会将Service的名称与IP记录起来，这样Pod就可以通过向CoreDNS查询Service的名称获得Service的IP地址。

访问时通过nginx.<namespace>.svc.cluster.local访问，其中nginx为Service的名称，<namespace>为命名空间名称，svc.cluster.local为域名后缀，在实际使用中，在同一个命名空间下可以省略<namespace>.svc.cluster.local，直接使用ServiceName即可。

使用ServiceName的方式有个主要的优点就是可以在开发应用程序时可以将ServiceName写在程序中，这样无需感知具体Service的IP地址。

CoreDNS插件安装后也有一个Service，在kube-system命名空间下，如下所示。

```
$ kubectl get svc -n kube-system
NAME              TYPE        CLUSTER-IP     EXTERNAL-IP                PORT(S)                  AGE
coredns           ClusterIP   10.247.3.10    <none>                     53/UDP,53/TCP,8080/TCP   13d
```

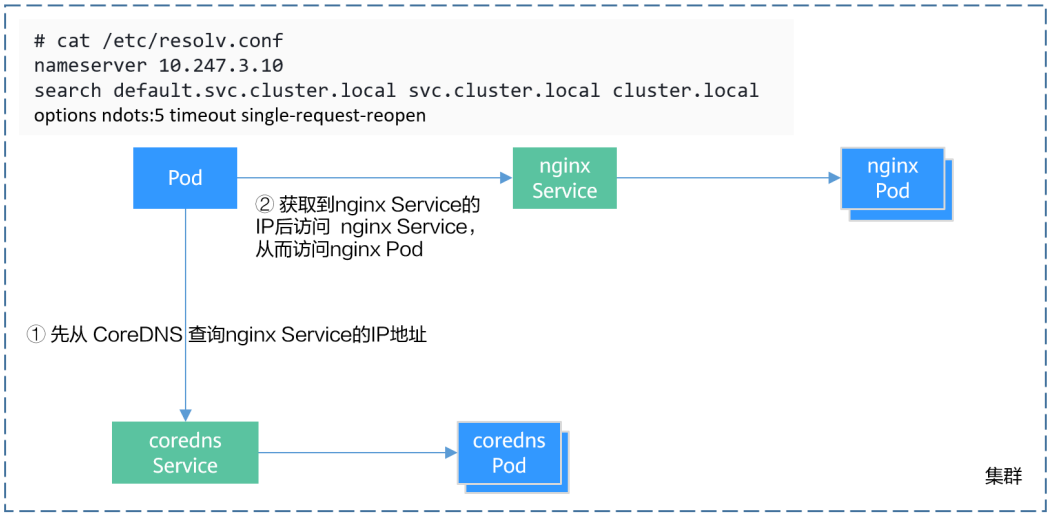
默认情况下，其他Pod创建后，会将coredns Service的地址作为域名解析服务器的地址写在Pod的 /etc/resolv.conf 文件中，创建一个Pod，查看/etc/resolv.conf文件，如下所示。

```
$ kubectl exec test01-6cbbf97b78-krj6h -it -- /bin/sh
/ # cat /etc/resolv.conf
nameserver 10.247.3.10
```

```
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5 timeout single-request-reopen
```

在Pod中访问nginx Pod的ServiceName:Port，会先从CoreDNS中解析出nginx Service的IP地址，然后再访问nginx Service的IP地址，从而访问到nginx Pod。

图 11-22 集群内域名解析示例图



Kubernetes 中的域名解析逻辑

DNS策略可以在每个pod基础上进行设置，目前，Kubernetes支持**Default、ClusterFirst、ClusterFirstWithHostNet和None**四种DNS策略，具体请参见**https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/**。这些策略在pod-specific的**dnsPolicy**字段中指定。

- **“Default”**：如果dnsPolicy被设置为“Default”，则名称解析配置将从pod运行的节点继承。 自定义上游域名服务器和存根域不能够与这个策略一起使用。
- **“ClusterFirst”**：如果dnsPolicy被设置为“ClusterFirst”，任何与配置的集群域后缀不匹配的DNS查询（例如，www.kubernetes.io）将转发到从该节点继承的上游名称服务器。集群管理员可能配置了额外的存根域和上游DNS服务器。
- **“ClusterFirstWithHostNet”**：对于使用hostNetwork运行的Pod，您应该明确设置其DNS策略“ClusterFirstWithHostNet”。
- **“None”**：它允许Pod忽略Kubernetes环境中的DNS设置。应使用dnsConfigPod规范中的字段提供所有DNS设置 。

说明

- Kubernetes 1.10及以上版本，支持Default、ClusterFirst、ClusterFirstWithHostNet和None四种策略；低于Kubernetes 1.10版本，仅支持default、ClusterFirst和ClusterFirstWithHostNet三种。
- “Default”不是默认的DNS策略。如果dnsPolicy的Flag没有特别指明，则默认使用“**ClusterFirst**”。

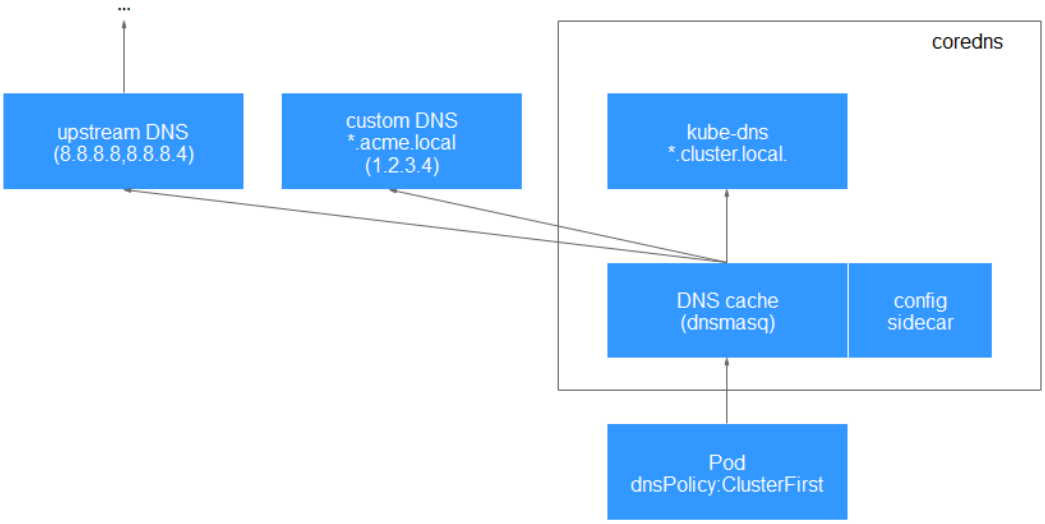
路由请求流程：

未配置存根域：没有匹配上配置的集群域名后缀的任何请求，例如“www.kubernetes.io”，将会被转发到继承自节点的上游域名服务器。

已配置存根域：如果配置了存根域和上游DNS服务器，DNS查询将基于下面的流程对请求进行路由：

1. 查询首先被发送到coredns中的DNS缓存层。
2. 从缓存层，检查请求的后缀，并根据下面的情况转发到对应的DNS上：
   - 具有集群后缀的名字（例如“.cluster.local”）：请求被发送到coredns。
   - 具有存根域后缀的名字（例如“.acme.local”）：请求被发送到配置的自定义DNS解析器（例如：监听在 1.2.3.4）。
   - 未能匹配上后缀的名字（例如“widget.com”）：请求被转发到上游DNS。

图 11-23 路由请求流程



相关操作

您还可以在工作负载中进行DNS配置，具体请参见**工作负载DNS配置说明**。

您还可以使用CoreDNS实现自定义域名解析，具体请参见**使用CoreDNS实现自定义域名解析**。

11.5.2 工作负载 DNS 配置说明

Kubernetes集群内置DNS插件Kube-DNS/CoreDNS，为集群内的工作负载提供域名解析服务。业务在高并发调用场景下，如果使用到域名解析服务，可能会触及到Kube-DNS/CoreDNS的性能瓶颈，导致DNS请求概率失败，影响用户业务正常运行。在Kubernetes使用的过程中，发现有些场景下工作负载的域名解析存在冗余的DNS查询，使得高并发场景更容易触及DNS的性能瓶颈。根据业务使用场景，对工作负载的DNS配置进行优化，能够在一定程度上减少DNS请求概率失败的问题。

更多DNS相关信息请参见**CoreDNS**。

DNS 配置项说明

在Linux系统的节点或者容器里执行cat /etc/resolv.conf命令，能够查看到DNS配置，以Kubernetes集群的容器DNS配置为例：

```
nameserver 10.247.x.x
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

配置项说明：

- nameserver：容器解析域名时查询的DNS服务器的IP地址列表。如果设置为10.247.x.x说明DNS对接到Kube-DNS/CoreDNS，如果是其他IP地址，则表示采用云上DNS或者用户自建的DNS。
- search：定义域名的搜索域列表，当访问的域名不能被DNS解析时，会把该域名与搜索域列表中的域依次进行组合，并重新向DNS发起请求，直到域名被正确解析或者尝试完搜索域列表为止。对于CCE集群来说，容器的搜索域列表配置3个域，当解析一个不存在的域名时，会产生8次DNS查询，因为对于每个域名需要查询两次，分别是IPv4和IPv6。
- options：定义域名解析配置文件的其他选项，常见的有timeout、ndots等等。

  Kubernetes集群容器的域名解析文件设置为options ndots:5，该参数的含义是当域名的“.”个数小于ndots的值，会先把域名与search搜索域列表进行组合后进行DNS查询，如果均没有被正确解析，再以域名本身去进行DNS查询。当域名的“.”个数大于或者等于ndots的值，会先对域名本身进行DNS查询，如果没有被正确解析，再把域名与search搜索域列表依次进行组合后进行DNS查询。

  如查询www.***.com域名时，由于该域名的“.”个数为2，小于ndots的值，所以DNS查询请求的顺序依次为：www.***.default.svc.cluster.local、www.***.com.svc.cluster.local、 www.***.com.cluster.local和 www.***.com，需要发起至少7次DNS查询请求才能解析出该域名的IP。可以看出，这种配置在访问外部域名时，存在大量冗余的DNS查询，存在优化点。

说明

完整的Linux域名解析文件配置项说明可以参考文档：**http://man7.org/linux/man-pages/man5/resolv.conf.5.html**。

通过控制台进行工作负载 DNS 配置

Kubernetes为应用提供了与DNS相关的配置选项，通过对应用进行DNS配置，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。以下步骤以nginx应用为例，介绍如何通过控制台为工作负载添加DNS配置。

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧选择“工作负载”，在右上角单击“创建负载”。

步骤2 设置工作负载基本参数，详情请参见**创建工作负载**。

步骤3 在“高级配置”中，选择“DNS配置”页签，并按需填写以下参数。

- DNS策略：控制台中提供的DNS策略与YAML中的dnsPolicy字段对应，详情请参见**表11-43**。
  - 追加域名解析配置：即dnsPolicy字段设置为ClusterFirst，此时容器中既能够解析service注册的集群内部域名，也能够解析发布到互联网上的外部域名。
  - 替换域名解析配置：即dnsPolicy字段设置为None，此时必须填写“IP地址”和“搜索域”参数。容器将仅使用自定义的IP地址和搜索域配置进行域名解析。
  - 继承Pod所在节点域名解析配置：即dnsPolicy字段设置为Default，此时容器将使用Pod所在节点的域名解析配置，无法解析集群内部域名。
- 可选对象：即**dnsConfig字段**中的options参数。每个对象可以具有name属性（必需）和value属性（可选），填写完成后需单击“确认添加”。
  - timeout：超时时间 (s)。

  - ndots：域名中必须出现的"."的个数。如果域名中的"."的个数不小于ndots，则该域名为一个全限定域名，操作系统会直接查询；如果域名中的"."的个数小于ndots，操作系统会在搜索域中进行查询。
- IP地址：即**dnsConfig字段**中的nameservers参数，您可对自定义的域名配置域名服务器，值为一个或一组DNS IP地址。
- 搜索域：即**dnsConfig字段**中的searches参数，表示域名查询时的DNS搜索域列表，此属性是可选的。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。

步骤4 单击“创建工作负载”。

----结束

通过工作负载 YAML 进行 DNS 配置

您也可以通过YAML的方式创建工作负载，以nginx应用为例，其YAML文件中的DNS配置示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-1
          image: nginx:latest
          imagePullPolicy: IfNotPresent
      imagePullSecrets:
        - name: default-secret
      dnsPolicy: None
      dnsConfig:
        options:
          - name: ndots
            value: '5'
          - name: timeout
            value: '3'
        nameservers:
          - 10.2.3.4
        searches:
          - my.dns.search.suffix
```

- **dnsPolicy字段说明：**

  dnsPolicy字段是应用设置的DNS策略，默认值为“ClusterFirst”。dnsPolicy当前支持四种参数值：

表 11-43 dnsPolicy 字段说明

参数	说明
ClusterFirst（默认值）	即在默认DNS配置中追加自定义的域名解析配置。应用会默认对接CoreDNS（CCE集群的CoreDNS默认级联云上DNS），自定义填写的dnsConfig会追加到默认DNS参数中。这种场景下，容器既能够解析service注册的集群内部域名，也能够解析发布到互联网上的外部域名。由于该配置下，域名解析文件设置了search搜索域列表和ndots: 5，因此当访问外部域名和集群内部长域名（如kubernetes.default.svc.cluster.local）时，大部分域名都会优先遍历search搜索域列表，导致至少有6次无效的DNS查询，只有访问集群内部短域名（如kubernetes）时，才不存在无效的DNS查询。
ClusterFirstWithHostNet	对于配置**主机网络（hostNetwork）**的应用，默认对接Pod所在节点域名解析配置，即kubelet的“--resolv-conf”参数指向的域名解析文件（CCE集群在该配置下对接云上DNS）。如需对接集群的Kube-DNS/CoreDNS，dnsPolicy字段需设置为ClusterFirstWithHostNet，此时容器的域名解析文件配置与“ClusterFirst”一致，也存在无效的DNS查询。 ... spec: containers: - image: nginx:latest imagePullPolicy: IfNotPresent name: container-1 restartPolicy: Always **hostNetwork: true** dnsPolicy: **ClusterFirstWithHostNet**
Default	即继承Pod所在节点域名解析配置，并在此基础上追加自定义的域名解析配置。容器的域名解析文件使用kubelet的“--resolv-conf”参数指向的域名解析文件（CCE集群在该配置下对接云上DNS），没有配置search搜索域列表和options。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。
None	即替换默认的域名解析配置，完全使用自定义的域名解析配置。设置为None之后，必须设置dnsConfig字段，此时容器的域名解析文件将完全通过dnsConfig的配置来生成。

说明

此处如果dnsPolicy字段未被指定，其默认值为ClusterFirst，而不是Default。

- **dnsConfig字段说明：**

  dnsConfig为应用设置DNS参数，设置的参数将合并到基于dnsPolicy策略生成的域名解析文件中。当dnsPolicy为“None”，应用的域名解析文件完全由dnsConfig指定；当dnsPolicy不为“None”时，会在基于dnsPolicy生成的域名解析文件的基础上，追加dnsConfig中配置的dns参数。

表 11-44 dnsConfig 字段说明

参数	说明
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。
nameservers	DNS的IP地址列表。当应用的dnsPolicy设置为“None”时，列表必须至少包含一个IP地址，否则此属性是可选的。列出的DNS的IP列表将合并到基于dnsPolicy生成的域名解析文件的nameserver字段中，并删除重复的地址。
searches	域名查询时的DNS搜索域列表，此属性是可选的。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。

工作负载的 DNS 配置实践

前面介绍了Linux系统域名解析文件以及Kubernetes为应用提供的DNS相关配置项，下面将举例介绍应用如何进行DNS配置。

- **场景1 对接Kubernetes内置的Kube-DNS/CoreDNS**

  场景说明：

  这种方式适用于应用中的域名解析只涉及集群内部域名，或者集群内部域名+外部域名两种方式，应用默认采用这种配置。

  示例：

  ```
  apiVersion: v1
  kind: Pod
  metadata:
    namespace: default
    name: dns-example
  spec:
    containers:
    - name: test
      image: nginx:alpine
    dnsPolicy: ClusterFirst
    imagePullSecrets:
      - name: default-secret
  ```

  该配置下容器的域名解析文件将如下所示：

  ```
  nameserver 10.247.3.10
  search default.svc.cluster.local svc.cluster.local cluster.local
  options ndots:5
  ```

- **场景2 直接对接云DNS**

  场景说明：

  这种方式适用于应用只访问注册到互联网的外部域名，该场景不能解析集群内部域名。

  示例：

  ```
  apiVersion: v1
  kind: Pod
  metadata:
  ```

```
  namespace: default
  name: dns-example
spec:
  containers:
  - name: test
    image: nginx:alpine
  dnsPolicy: Default  #使用kubelet的“--resolv-conf”参数指向的域名解析文件（CCE集群在该配置下对接云DNS）
  imagePullSecrets:
    - name: default-secret
```

该配置下容器的域名解析文件将如下所示：

```
nameserver 100.125.x.x
```

- **场景3 主机网络模式的应用对接Kube-DNS/CoreDNS**

  场景说明：

  对于配置主机网络模式的应用，默认对接云DNS，如果应用需要对接Kube-DNS/CoreDNS，需将dnsPolicy设置为“ClusterFirstWithHostNet”。

  示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
  containers:
  - name: nginx
    image: nginx:alpine
    ports:
    - containerPort: 80
  imagePullSecrets:
    - name: default-secret
```

该配置下容器的域名解析文件将如下所示：

```
nameserver 10.247.3.10
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

- **场景4 自定义应用的域名配置**

  场景说明：

  用户可以完全自定义配置应用的域名解析文件，这种方式非常灵活，dnsPolicy和dnsConfig配合使用，几乎能够满足所有使用场景，如对接用户自建DNS的场景、串联多个DNS的场景以及优化DNS配置选项的场景等等。

  示例1：对接用户自建DNS

  该配置下，dnsPolicy为“None”，应用的域名解析文件完全根据dnsConfig配置生成。

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
  - name: test
    image: nginx:alpine
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
    - 10.2.3.4  #用户自建DNS的IP地址
    searches:
    - ns1.svc.cluster.local
    - my.dns.search.suffix
```

```
  options:
  - name: ndots
    value: "2"
  - name: timeout
    value: "3"
 imagePullSecrets:
  - name: default-secret
```

该配置下容器的域名解析文件将如下所示：

```
nameserver 10.2.3.4
search ns1.svc.cluster.local my.dns.search.suffix
options timeout:3 ndots:2
```

示例2：修改域名解析文件的ndots选项，减少无效的DNS查询

该配置下，dnsPolicy不为“None”，会在基于dnsPolicy生成的域名解析文件的基础上，追加dnsConfig中配置的dns参数。

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
  - name: test
    image: nginx:alpine
  dnsPolicy: "ClusterFirst"
  dnsConfig:
    options:
    - name: ndots
      value: "2" #该配置会将基于ClusterFirst策略生成的域名解析文件的ndots:5参数改写为ndots:2
  imagePullSecrets:
    - name: default-secret
```

该配置下容器的域名解析文件将如下所示：

```
nameserver 10.247.3.10
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:2
```

示例3：串联使用多个DNS

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
  - name: test
    image: nginx:alpine
  dnsPolicy: ClusterFirst  # 追加域名解析配置，集群中会默认对接CoreDNS
  dnsConfig:
    nameservers:
    - 10.2.3.4 # 用户自建DNS的IP地址
  imagePullSecrets:
    - name: default-secret
```

该配置下容器的域名解析文件将如下所示：

```
nameserver 10.247.3.10 10.2.3.4
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

11.5.3 使用 CoreDNS 实现自定义域名解析

应用现状

在使用CCE时，可能会有解析自定义内部域名的需求，例如：

- 存量代码配置了用固定域名调用内部其他服务，如果要切换到Kubernetes Service方式，修改配置工作量大。
- 在集群外自建了一个其他服务，需要将集群中的数据通过固定域名发送到这个服务。

解决方案

使用CoreDNS有以下几种自定义域名解析的方案。

- **为CoreDNS配置存根域**：可以直接在控制台添加，简单易操作。
- **使用 CoreDNS Hosts 插件配置任意域名解析**：简单直观，可以添加任意解析记录，类似在本地/etc/hosts中添加解析记录。
- **使用 CoreDNS Rewrite 插件指向域名到集群内服务**：相当于给Kubernetes中的Service名称取了个别名，无需提前知道解析记录的IP地址。
- **使用 CoreDNS Forward 插件将自建 DNS 设为上游 DNS**：自建DNS中，可以管理大量的解析记录，解析记录专门管理，增删记录无需修改CoreDNS配置。

注意事项

CoreDNS修改配置需额外谨慎，因为CoreDNS负责集群的域名解析任务，修改不当可能会导致集群解析出现异常。请做好修改前后的测试验证。

为 CoreDNS 配置存根域

集群管理员可以修改CoreDNS Corefile的ConfigMap以更改服务发现的工作方式。

若集群管理员有一个位于10.150.0.1的Consul域名解析服务器，并且所有Consul的域名都带有.consul.local的后缀。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下添加存根域。格式为一个键值对，键为DNS后缀域名，值为一个或一组DNS IP地址，如 'consul.local --10.XXX.XXX.XXX'。

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置项与密钥”，在“kube-system”命名空间下，查看coredns配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
```

```
    reload
    ready {$POD_IP}:8081
}
consul.local:5353 {
    bind {$POD_IP}
    errors
    cache 30
    forward . 10.150.0.1
}
```

----结束

修改 CoreDNS Hosts 配置

在CoreDNS中修改hosts后，可以不用单独在每个Pod中配置hosts添加解析记录。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下编辑高级配置，在plugins字段添加以下内容。

```
{
  "configBlock": "192.168.1.1 www.example.com\nfallthrough",
  "name": "hosts"
}
```

须知

此处配置不能遗漏fallthrough字段，fallthrough表示当在hosts找不到要解析的域名时，会将解析任务传递给CoreDNS的下一个插件。如果不写fallthrough的话，任务就此结束，不会继续解析，会导致集群内部域名解析失败的情况。

hosts的详细配置请参见**https://coredns.io/plugins/hosts/**。

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置项与密钥”，在“kube-system”命名空间下，查看coredns配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    hosts {
      192.168.1.1 www.example.com
      fallthrough
    }
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
```

```
    ready {$POD_IP}:8081
}
```

----结束

添加 CoreDNS Rewrite 配置指向域名到集群内服务

使用 CoreDNS 的 Rewrite 插件，将指定域名解析到某个 Service 的域名。例如，将访问example.com域名的请求重新指向到example.default.svc.cluster.local域名，即指向到default命名空间下的example服务。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下编辑高级配置，在plugins字段添加以下内容。

```
{
  "name": "rewrite",
  "parameters": "name example.com example.default.svc.cluster.local"
}
```

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置项与密钥”，在“kube-system”命名空间下，查看coredns配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    rewrite name example.com example.default.svc.cluster.local
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
```

----结束

使用 CoreDNS 级联自建 DNS

CoreDNS默认使用节点的/etc/resolv.conf文件进行解析，您也可以修改成外部DNS的解析地址。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下编辑高级配置，在plugins字段修改以下内容。

```
{
  "configBlock": "policy random",
```

```
    "name": "forward",
    "parameters": ". 192.168.1.1"
}
```

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置项与密钥”，在“kube-system”命名空间下，查看coredns配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . 192.168.1.1 {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
```

----结束

11.6 容器网络配置

11.6.1 主机网络（hostNetwork）

背景信息

Kubernetes支持Pod直接使用主机（节点）的网络，当Pod配置为hostNetwork: true时，在此Pod中运行的应用程序可以直接看到Pod所在主机的网络接口。

配置说明

Pod使用主机网络只需要在配置中添加hostNetwork: true即可，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      hostNetwork: true
      containers:
      - image: nginx:alpine
        name: nginx
```

```
    imagePullSecrets:
    - name: default-secret
```

部署后可以看到Pod的IP与节点的IP相同，说明Pod直接使用了主机网络。

```
$ kubectl get pod -owide
NAME                 READY   STATUS    RESTARTS   AGE     IP          NODE        NOMINATED NODE
READINESS GATES
nginx-6fdf99c8b-6wwft   1/1     Running   0          3m41s   10.1.0.55   10.1.0.55   <none>          <none>
```

hostNetwork 使用注意事项

Pod直接使用主机的网络会占用宿主机的端口，Pod的IP就是宿主机的IP，使用时需要考虑是否与主机上的端口冲突，因此一般情况下除非您知道需要某个特定应用占用宿主机上的特定端口时，不建议使用主机网络。

由于使用主机网络，访问Pod就是访问节点，要**注意放通节点安全组端口**，否则会出现访问不通的情况。

另外由于占用主机端口，使用Deployment部署hostNetwork类型Pod时，要注意**Pod的副本数不要超过节点数量**，否则会导致一个节点上调度了多个Pod，Pod启动时端口冲突无法创建。例如上面例子中的nginx，如果服务数为2，并部署在只有1个节点的集群上，就会有一个Pod无法创建，查询Pod日志会发现是由于端口占用导致nginx无法启动。

⚠ 注意

请避免在同一个节点上调度多个使用主机网络的Pod，否则在创建ClusterIP类型的Service访问Pod时，会出现访问ClusterIP不通的情况。

```
$ kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   1/2     2            1           67m
$ kubectl get pod
NAME                 READY   STATUS             RESTARTS   AGE
nginx-6fdf99c8b-6wwft   1/1     Running            0          67m
nginx-6fdf99c8b-rglm7   0/1     CrashLoopBackOff   13         44m
$ kubectl logs nginx-6fdf99c8b-rglm7
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/05/11 07:18:11 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to [::]:80 failed (98: Address in use)
nginx: [emerg] bind() to [::]:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to [::]:80 failed (98: Address in use)
nginx: [emerg] bind() to [::]:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to [::]:80 failed (98: Address in use)
nginx: [emerg] bind() to [::]:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to [::]:80 failed (98: Address in use)
nginx: [emerg] bind() to [::]:80 failed (98: Address in use)
```

```
2022/05/11 07:18:11 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: bind() to [::]:80 failed (98: Address in use)
nginx: [emerg] bind() to [::]:80 failed (98: Address in use)
2022/05/11 07:18:11 [emerg] 1#1: still could not bind()
nginx: [emerg] still could not bind()
```

11.6.2 Pod 互访 QoS 限速

操作场景

部署在同一节点上的不同业务容器之间存在带宽抢占，容易造成业务抖动。您可以通过对Pod间互访进行QoS限速来解决这个问题。

约束与限制

Pod互访限速设置需遵循以下约束：

约束类别	容器隧道网络模式	VPC网络模式	云原生2.0网络模式
支持的版本	所有版本都支持	v1.19.10以上集群版本	v1.19.10以上集群版本
支持的运行时类型	仅支持普通容器（容器运行时为runC） 容器隧道网络模式不支持安全容器	仅支持普通容器（容器运行时为runC） 不支持安全容器（容器运行时为Kata）	仅支持普通容器（容器运行时为runC） 不支持安全容器（容器运行时为Kata）
支持的Pod类型	仅支持非HostNetwork类型Pod		
支持的场景	支持Pod间互访、Pod访问Node、Pod访问Service的场景限速		
限制的场景	无	无	• 不支持Pod访问100.64.0.0/10和214.0.0.0/8外部云服务网段的限速场景 • 不支持健康检查的流量限速场景
限速值上限	机型带宽上限和34G两者之间的最小值，超过34G将设置为34G	机型带宽上限和4.3G两者之间的最小值	机型带宽上限和4.3G两者之间的最小值
限速值下限	支持K级别以上的限速	目前仅支持兆（M）级别以上的限速	

通过控制台设置

通过控制台创建工作负载时，您可在创建工作负载页面的“高级配置 > 网络配置”中设置Pod入/出口带宽限速。

通过 kubectl 命令行设置

您可以通过对工作负载添加annotations指定出口带宽和入口带宽，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test
  namespace: default
  labels:
    app: test
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
      annotations:
        kubernetes.io/ingress-bandwidth: 100M
        kubernetes.io/egress-bandwidth: 100M
    spec:
      containers:
        - name: container-1
          image: nginx:alpine
          imagePullPolicy: IfNotPresent
      imagePullSecrets:
        - name: default-secret
```

- kubernetes.io/ingress-bandwidth：Pod的入口带宽
- kubernetes.io/egress-bandwidth：Pod的出口带宽

如果不设置这两个参数，则表示不限制带宽。

说明

修改Pod出/入口带宽限速后，需要重启容器才可生效。由于独立创建的Pod（不通过工作负载管理）修改annotations后不会触发容器重启，因此带宽限制不会生效，您可以重新创建Pod或手动触发容器重启。

11.6.3 容器隧道网络配置

11.6.3.1 网络策略（NetworkPolicy）

NetworkPolicy是Kubernetes设计用来限制Pod访问的对象，相当于从应用的层面构建了一道防火墙，进一步保证了网络安全。NetworkPolicy支持的能力取决于集群的网络插件的能力。

默认情况下，如果命名空间中不存在任何策略，则所有进出该命名空间中的Pod的流量都被允许。

NetworkPolicy的规则可以选择如下3种：

- namespaceSelector：根据命名空间的标签选择，具有该标签的命名空间都可以访问。
- podSelector：根据Pod的标签选择，具有该标签的Pod都可以访问。
- ipBlock：根据网络选择，网段内的IP地址都可以访问。（仅Egress支持IPBlock）

约束与限制

- 当前仅**容器隧道网络模型**的集群支持网络策略（NetworkPolicy）。网络策略可分为以下规则：
  - 入规则（Ingress）：所有版本均支持。
  - 出规则（Egress）：仅如下操作系统和集群版本支持设置Egress规则：

操作系统	集群版本	经验证的内核版本
CentOS	v1.23及以上	3.10.0-1062.18.1.el7.x86_64 3.10.0-1127.19.1.el7.x86_64 3.10.0-1160.25.1.el7.x86_64
EulerOS 2.5	v1.23及以上	3.10.0-862.14.1.5.h591.eulerosv2r7.x86_64 3.10.0-862.14.1.5.h687.eulerosv2r7.x86_64
EulerOS 2.9	v1.23及以上	4.18.0-147.5.1.6.h541.eulerosv2r9.x86_64 4.18.0-147.5.1.6.h766.eulerosv2r9.x86_64

- 不支持对IPv6地址网络隔离。
- 通过原地升级到支持Egress的集群版本，由于不会升级节点操作系统，会导致无法使用Egress，此种情况下，请重置节点。

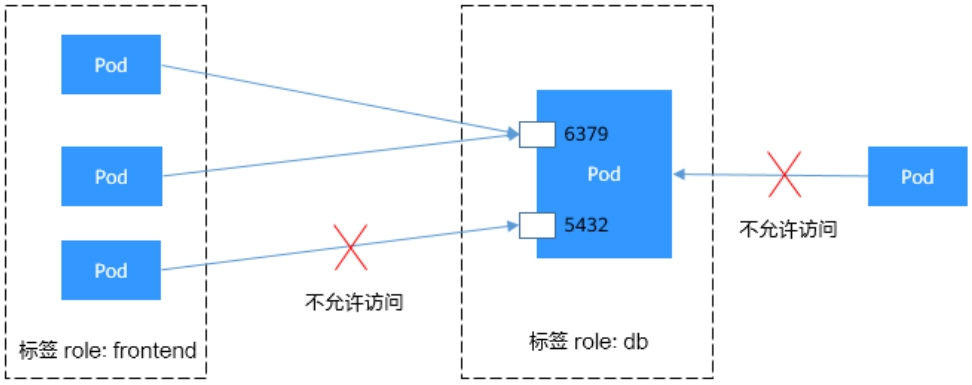
使用 Ingress 规则

- **使用podSelector设置访问范围**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:                  # 规则对具有role=db标签的Pod生效
    matchLabels:
      role: db
  ingress:                    # 表示入规则
  - from:
    - podSelector:              # 只允许具有role=frontend标签的Pod访问
        matchLabels:
          role: frontend
    ports:                    # 只能使用TCP协议访问6379端口
    - protocol: TCP
      port: 6379
```

示意图如下所示。

图 11-24 podSelector

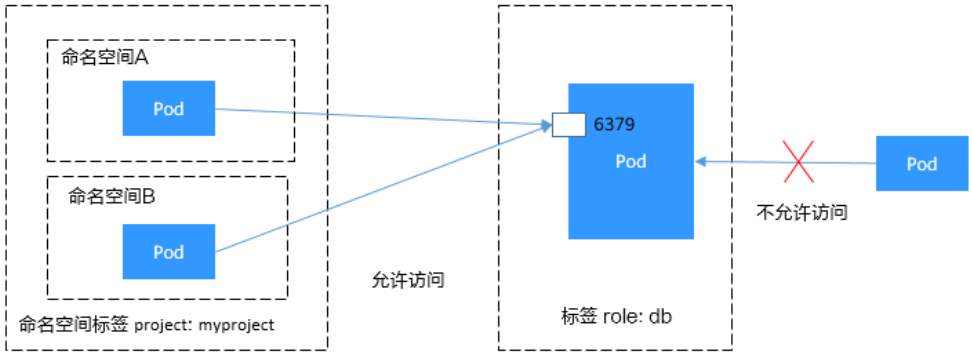


- **使用namespaceSelector设置访问范围**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector:                  # 规则对具有role=db标签的Pod生效
    matchLabels:
      role: db
  ingress:                      # 表示入规则
  - from:
    - namespaceSelector:        # 只允许具有project=myproject标签的命名空间中的Pod访问
        matchLabels:
          project: myproject
    ports:                      # 只能使用TCP协议访问6379端口
    - protocol: TCP
      port: 6379
```

示意图如下所示。

图 11-25 namespaceSelector



使用 Egress 规则

Egress不仅支持podSelector和namespaceSelector，还支持ipBlock。

说明

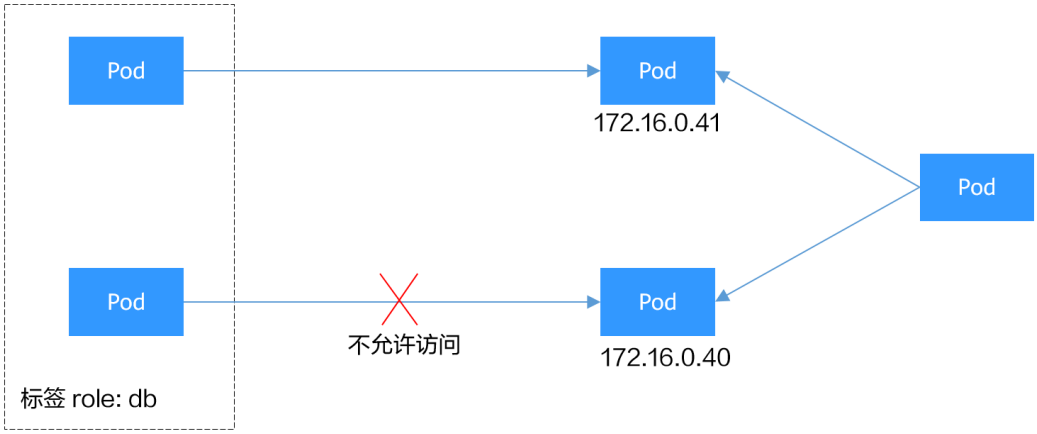
仅1.23及以上版本集群支持Egress规则，且当前仅支持EulerOS 2.5、EulerOS 2.9和CentOS 7.X。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: deny-client-a-via-except-cidr-egress-rule
  namespace: default
spec:
  policyTypes:                  # 使用Egress必须指定policyType
    - Egress
  podSelector:                  # 规则对具有role=db标签的Pod生效
    matchLabels:
      role: db
  egress:                       # 表示出规则
  - to:
    - ipBlock:
        cidr: 172.16.0.16/16    # 允许此网段被访问
        except:
        - 172.16.0.40/32        # 不允许此网段被访问，except需在cidr网段内
```

示意图如下所示。

图 11-26 ipBlock

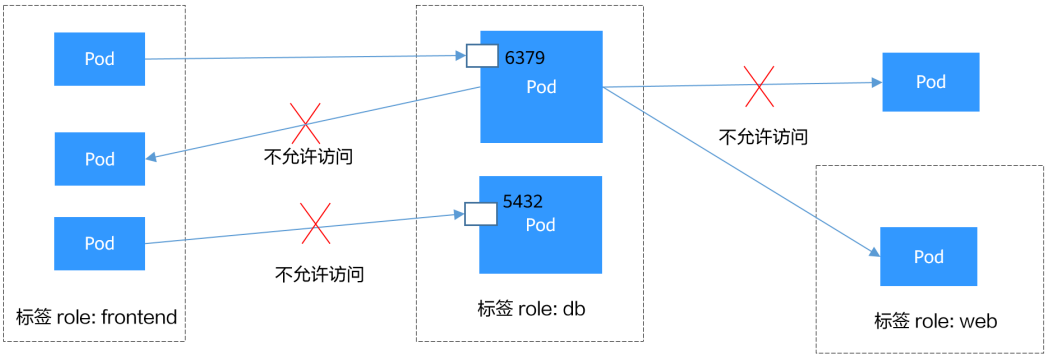


Ingress和Egress可以定义在同一个规则中。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  policyTypes:
  - Ingress
  - Egress
  podSelector:                  # 规则对具有role=db标签的Pod生效
    matchLabels:
      role: db
  ingress:                      # 表示入规则
  - from:
    - podSelector:              # 只允许具有role=frontend标签的Pod访问
        matchLabels:
          role: frontend
    ports:                      # 只能使用TCP协议访问6379端口
    - protocol: TCP
      port: 6379
  egress:                       # 表示出规则
  - to:
    - podSelector:              # 只允许访问具有role=web标签的Pod
        matchLabels:
          role: web
```

示意图如下所示。

图 11-27 同时使用 Ingress 和 Egress



在控制台创建网络策略

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“服务发现”，在右侧选择“网络策略”页签，单击右上角“创建网络策略”。

- 策略名称：自定义输入NetworkPolicy名称。
- 命名空间：选择网络策略所在命名空间。
- 选择器：输入标签选择要关联的Pod，然后单击添加。您也可以单击“引用负载标签”直接引用已有负载的标签。
- 入方向规则：单击+添加入方向规则，参数设置请参见表11-45。

表 11-45 添加入方向规则

参数	参数说明
协议端口	请选择对应的协议类型和端口，目前支持TCP和UDP协议。
源对象命名空间	选择允许哪个命名空间的对象访问。不填写表示和当前策略属于同一命名空间。
源对象Pod标签	允许带有这个标签的Pod访问，不填写表示命名空间下全部Pod。

步骤3 设置完成后，单击“确定”。

----结束

11.6.4 云原生网络 2.0 配置

11.6.4.1 安全组策略（SecurityGroup）

云原生网络2.0网络模式下，Pod使用的是VPC的弹性网卡/辅助弹性网卡，可直接绑定安全组，绑定弹性公网IP。为方便用户在CCE内直接为Pod关联安全组，CCE新增了一个名为SecurityGroup的自定义资源对象。通过SecurityGroup资源对象，用户可对工作负载实现自定义的安全隔离诉求。

约束与限制

- 1个工作负载最多可绑定5个安全组。

通过界面创建

步骤1 登录CCE控制台，单击集群名称，进入集群。

步骤2 在左侧选择“工作负载”，单击工作负载名称。

步骤3 在“安全组策略”页签下，单击“创建”。

步骤4 根据界面提示，配置参数， 具体如表11-46所示。

表 11-46 配置参数

参数名称	描述	示例
安全组策略名称	输入安全组策略名称。 请输入1-63个字符，以小写字母开头，由小写字母、数字、连接符（-）组成，且不能以连接符（-）结尾。	security-group
关联安全组	选中的安全组将绑定到选中的工作负载的弹性网卡/辅助弹性网卡上，在下拉框中最多可以选择5条，安全组必选，不可缺省。 如将绑定的安全组未创建，可单击“创建安全组”，完成创建后单击刷新按钮。 **须知** • 最多可选择5个安全组。 • 鼠标悬浮在安全组名称旁的 图标上，可查看安全组的详细信息。	64566556-bd6f-48fb-b2c6-df8f44617953 5451f1b0-bd6f-48fb-b2c6-df8f44617953

步骤5 参数配置后，单击“确定”。

创建完成后页面将自动返回到安全组策略列表页，可以看到新添加的安全组策略已在列表中。

----结束

通过 kubectl 命令行创建

步骤1 请参见通过kubectl连接集群，使用kubectl连接集群。

步骤2 创建一个名为securitygroup-demo.yaml的描述文件。

vi securitygroup-demo.yaml

例如，用户创建如下的SecurityGroup资源对象，给所有的app：nginx工作负载绑定上提前已经创建的64566556-bd6f-48fb-b2c6-df8f44617953，5451f1b0-bd6f-48fb-b2c6-df8f44617953的两个安全组。示例如下：

```
apiVersion: crd.yangtse.cni/v1
kind: SecurityGroup
metadata:
  name: demo
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: nginx
  securityGroups:
  - id: 64566556-bd6f-48fb-b2c6-df8f44617953
  - id: 5451f1b0-bd6f-48fb-b2c6-df8f44617953
```

以上yaml参数说明如**表11-47**。

表 11-47 参数说明

字段名称	字段说明	必选/可选
apiVersion	表示API的版本号，版本号为crd.yangtse.cni/v1。	必选
kind	创建的对象类别。	必选
metadata	资源对象的元数据定义。	必选
name	SecurityGroup的名称。	必选
namespace	工作空间名称。	必选
spec	用户对SecurityGroup的详细描述的主体部分都在spec中给出。	必选
podSelector	定义SecurityGroup中需要关联安全组的工作负载。	必选
securityGroups	id为安全组的ID。	必选

步骤3 执行以下命令，创建SecurityGroup。

kubectl create -f securitygroup-demo.yaml

回显如下表示已开始创建SecurityGroup

```
securitygroup.crd.yangtse.cni/demo created
```

步骤4 执行以下命令，查看SecurityGroup。

kubectl get sg

回显信息中有创建的SecurityGroup名称为demo，表示SecurityGroup已创建成功。

```
NAME                POD-SELECTOR                    AGE
all-no              map[matchLabels:map[app:nginx]]   4h1m
s001test            map[matchLabels:map[app:nginx]]   19m
demo                map[matchLabels:map[app:nginx]]   2m9s
```

----结束

11.6.4.2 网络配置（NetworkAttachmentDefinition）

操作场景

CCE Turbo集群支持以命名空间粒度设置容器所在的容器子网及安全组，该功能通过集群中的一种**CRD**资源NetworkAttachmentDefinition实现。在为某一命名空间配置NetworkAttachmentDefinition后，该命名空间的Pod支持以下功能：

- 容器绑定子网：Pod IP会被约束在特定的网段中，不同命名空间之间可实现网络隔离。
- 容器绑定安全组：支持为同一命名空间的Pod设置安全组规则，实现自定义访问策略。

约束与限制

- 网络配置（NetworkAttachmentDefinition）仅在CCE Turbo集群中可用，且集群为1.23.8-r0、1.25.3-r0及以上版本。
- 仅默认配置 default-network 支持开启网卡预热，用户自定义配置中的容器子网不支持网卡预热。未开启网卡预热时，工作负载实例的创建速度会减慢，因此不适用于高性能Pod创建场景。
- 如需删除NetworkAttachmentDefinition，请先删除对应的命名空间下使用该配置创建的Pod（带有名为“cni.yangtse.io/network-status”的annotation），详情请参见**删除网络配置**。

通过控制台配置

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“系统配置”，选择“网络配置”页签。

说明

集群中存在默认配置default-network，对所有未配置容器子网的命名空间生效，“集群信息”页面的网络信息中的“默认容器子网”即为default-network中的容器子网。该配置不可删除。

步骤3 单击右上角“创建网络配置”，在弹窗中配置子网和命名空间等信息。

- 名称：自定义名称，最长支持253个字符。default-network、default、mgnt0、mgnt1四个名称为系统预留，请勿使用。
- 命名空间：请选择命名空间。不同配置之间的命名空间不可重复。若无命名空间可选请单击后方的“创建命名空间”进行创建。
- 容器子网：请选择子网。若无子网可选请单击后方的“创建子网”进行创建，创建完成后单击刷新按钮。最多可选择 20 个子网。
- 关联安全组：默认为容器ENI安全组，您也可以选择单击后方的“创建安全组”进行创建，创建完成后单击刷新按钮。最多可选择5个安全组。

步骤4 完成基本配置后单击“创建”，创建完成后页面自动返回到网络配置列表，可以看到新添加的容器子网配置已在列表中。

----结束

通过 kubectl 命令行配置

本节说明通过kubectl命令创建NetworkAttachmentDefinition的方法。

步骤1 请参见**通过kubectl连接集群**，使用kubectl连接集群。

步骤2 修改networkattachment-test.yaml。

vi networkattachment-test.yaml

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    yangtse.io/project-id: 05e38**
  name: example
  namespace: kube-system
spec:
  config:
  '{
    "type":"eni-neutron",
    "args":{
      "securityGroups":"41891**",
      "subnets":[
        {
          "subnetID":"27d95**"
        }
      ]
    },
    "selector":{
      "namespaceSelector":{
        "matchLabels":{
          "kubernetes.io/metadata.name":"default"
        }
      }
    }
  }'
```

表 11-48 关键参数说明

参数	是否必填	参数类型	描述
apiVersion	是	String	表示API的版本号。固定为k8s.cni.cncf.io/v1。
kind	是	String	创建的对象类别。固定为NetworkAttachmentDefinition。
yangtse.io/project-id	是	String	项目ID。
name	是	String	配置名称。
namespace	是	String	配置资源所在命名空间，固定为 kube-system。
config	是	**表2 config字段数据结构说明** Object	配置内容，为json格式的字符串。

表 11-49 config 字段数据结构说明

参数	是否必填	参数类型	描述
type	是	String	固定为eni-neutron。
args	否	**表3 args字段数据结构说明** Object	配置参数。
selector	否	**表11-51**Object	选择该配置所作用的命名空间。

表 11-50 args 字段数据结构说明

参数	是否必填	参数类型	描述
securityGroups	否	String	安全组ID。若未对安全组存在规划，请选择和default-network中的安全组一致。 获取方式： 登录虚拟私有云控制台，在左侧导航栏选择“访问控制 > 安全组”，单击安全组名称，在“基本信息”页签下找到“ID”字段复制即可。
subnets	是	Array of subnetID Objects	容器子网ID列表，至少需填写一个，不可以为空，格式如下： [{"subnetID":"27d95**"},{"subnetID":"827bb**"},{"subnetID":"bdd6b**"}] 同一VPC下非集群的子网ID。 获取方式： 登录虚拟私有云控制台，在左侧导航栏选择“虚拟私有云 > 子网”，单击子网名称，在“基本信息”页签下找到“子网ID”字段复制即可。

表 11-51 selector 字段数据结构说明

参数	是否必填	参数类型	描述

namespaceSelector	否	matchLabels Object	该选择器为Kubernetes标准的选择器，需填写命名空间标签，格式如下：
			"matchLabels":{
			"kubernetes.io/metadata.name":"default"
			}
			不同配置之间的命名空间不可重合。

步骤3 创建NetworkAttachmentDefinition。

kubectl create -f networkattachment-test.yaml

回显如下，表示NetworkAttachmentDefinition已创建。

```
networkattachmentdefinition.k8s.cni.cncf.io/example created
```

----结束

删除网络配置

您可以查看新添加网络配置的YAML，也可以对新添加的配置进行“删除”操作。

说明

在删除网络配置时，需先删除该配置所对应的容器，否则将删除失败。

1. 执行以下命令筛选集群中使用该配置的Pod（其中example为示例配置名称，请自行替换）：
   ```
   kubectl get po -A -o=jsonpath="{.items[?(@.metadata.annotations.cni\.yangtse\.io/network-status=='[{\"name\":\"example\"}]')]['metadata.namespace', 'metadata.name']}"
   ```
   返回结果中包含了该配置关联的Pod名字及命名空间。
2. 删除创建该Pod的Owner，其Owner可能为Deployment、StatefulSet、DaemonSet或Job类型的工作负载。

11.7 集群网络配置

11.7.1 切换节点子网

操作场景

本文介绍如何为集群中的节点切换子网。

约束限制

- 仅支持切换与集群同VPC下的子网，不支持切换节点安全组。

操作步骤

步骤1 登录ECS控制台。

步骤2 单击目标云服务器“操作”列下的“更多 > 网络设置 > 切换VPC”。

步骤3 设置切换VPC参数。

- 虚拟私有云：必须选择集群相同的VPC。
- 子网：选择需要切换的子网。
- 私有IP地址：根据需求选择自动分配或使用已有IP。
- 安全组：必须选择集群节点安全组，否则节点将不可用。

步骤4 单击“确定”，等待云服务器切换完成。

步骤5 前往CCE控制台，重置该节点。可按照默认参数配置，详情请参见**重置节点**。

----结束

11.7.2 扩展集群容器网段

操作场景

当创建CCE集群时设置的容器网段（CCE Turbo集群中为容器子网）太小，无法满足业务扩容需求时，您通过扩展集群容器网段的方法来解决。本文介绍如何为集群添加容器网段。

约束与限制

- 仅支持v1.19及以上版本的CCE集群和CCE Turbo集群，且暂不支持“容器隧道网络”模型的集群。
- 容器网段/容器子网添加后无法删除，请谨慎操作。

为 CCE 集群添加容器网段

步骤1 登录CCE控制台，单击CCE集群名称，进入集群。

步骤2 在“集群信息”页面，找到“网络信息”版块，并单击“添加容器网段”。

步骤3 设置需要添加的容器网段，您可单击 + 一次性添加多个容器网段。

说明

新增的容器网段不能与服务网段、VPC网段及已有的容器网段冲突。

步骤4 单击“确定”。

----结束

为 CCE Turbo 集群添加容器子网

步骤1 登录CCE控制台，单击CCE Turbo集群名称，进入集群。

步骤2 在“集群信息”页面，找到“网络信息”版块，并单击“添加容器子网”。

步骤3 选择同一VPC下的容器子网，您可一次性添加多个容器子网。如没有其他可用的容器子网，可前往VPC控制台创建。

步骤4 单击“确定”。

----结束

11.8 容器如何访问 VPC 内部网络

前面章节介绍了使用Service和Ingress访问容器，本节将介绍如何从容器访问内部网络（**VPC内集群外**），包括VPC内访问和跨VPC访问。

VPC 内访问

根据集群容器网络模型不同，从容器访问内部网络有不同表现。

- **容器隧道网络**

容器隧道网络在节点网络基础上通过隧道封装网络数据包，容器访问同VPC下其他资源时，只要节点能访问通，容器就能访问通。如果访问不通，需要确认对端资源的安全组配置是否能够允许容器所在节点访问。

- **云原生网络2.0**

  云原生网络2.0模型下，容器直接从VPC网段内分配IP地址，容器网段是节点所在VPC的子网，容器与VPC内其他地址天然能够互通。如果访问不通，需要确认对端资源的安全组配置是否能够允许容器网段访问。

- **VPC网络**

  VPC网络使用了VPC路由功能来转发容器的流量，容器网段与节点VPC不在同一个网段，容器访问同VPC下其他资源时，**需要对端资源的安全组能够允许容器网段访问**。

  例如集群节点所在网段为192.168.10.0/24，容器网段为172.16.0.0/16。

  VPC下（集群外）有一个地址为192.168.10.52的ECS，其安全组规则仅允许集群节点的IP网段访问。

  此时如果从容器中ping 192.168.10.52，会发现无法ping通。

  ```
  kubectl exec test01-6cbbf97b78-krj6h  -it -- /bin/sh
  / # ping 192.168.10.25
  PING 192.168.10.25 (192.168.10.25): 56 data bytes
  ^C
  --- 192.168.10.25 ping statistics ---
  104 packets transmitted, 0 packets received, 100% packet loss
  ```

  在安全组放通容器网段172.16.0.0/16访问。

  此时再从容器中ping 192.168.10.52，会发现可以ping通。

  ```
  $ kubectl exec test01-6cbbf97b78-krj6h  -it -- /bin/sh
  / # ping 192.168.10.25
  PING 192.168.10.25 (192.168.10.25): 56 data bytes
  64 bytes from 192.168.10.25: seq=0 ttl=64 time=1.412 ms
  64 bytes from 192.168.10.25: seq=1 ttl=64 time=1.400 ms
  64 bytes from 192.168.10.25: seq=2 ttl=64 time=1.299 ms
  64 bytes from 192.168.10.25: seq=3 ttl=64 time=1.283 ms
  ^C
  --- 192.168.10.25 ping statistics ---
  4 packets transmitted, 4 packets received, 0% packet loss
  ```

跨 VPC 访问

跨VPC访问通常采用对等连接等方法打通VPC。

- 容器隧道网络只需将节点网络与对端VPC打通，容器自然就能访问对端VPC。
- 云原生网络2.0与容器隧道网络类似，将容器所在子网网段与对端VPC打通即可。
- VPC网络由于容器网段独立，除了要打通VPC网段，还要打通容器网段。

  例如有如下两个VPC。

  - vpc-demo：网段为192.168.0.0/16，集群在vpc-demo内，容器网段为10.0.0.0/16。
  - vpc-demo2：网段为10.1.0.0/16。

  创建一个名为peering-demo的对等连接（本端为vpc-demo，对端为vpc-demo2），注意对端VPC的路由添加容器网段。

  这样配置后，在vpc-demo2中就能够访问容器网段10.0.0.0/16。具体访问时要关注安全组配置，打通端口配置。

访问其他云服务

与CCE进行内网通信的与服务常见服务有：RDS、DCS、Kafka、RabbitMQ、ModelArts等。

访问其他云服务除了上面所说的**VPC内访问**和**跨VPC访问**的网络配置外，还**需要关注所访问的云服务是否允许外部访问**，如DCS的Redis实例，需要添加白名单才允许访问。通常这些云服务会允许同VPC下IP访问，但是VPC网络模型下容器网段与VPC网段不同，需要特殊处理，将容器网段加入到白名单中。

容器访问内网不通的定位方法

如前所述，从容器中访问内部网络不通的情况可以按如下路径排查：

1. 查看要访问的对端服务器安全组规则，确认是否允许容器访问。
   - 容器隧道网络模型需要放通容器所在节点的IP地址
   - VPC网络模型需要放通容器网段
   - 云原生网络2.0需要放通容器所在子网网段
2. 查看要访问的对端服务器是否设置了白名单，如DCS的Redis实例，需要添加白名单才允许访问。添加容器和节点网段到白名单后可解决问题。
3. 查看要访问的对端服务器上是否安装了容器引擎，是否存在与CCE中容器网段冲突的情况。如果有网络冲突，会导致无法访问。

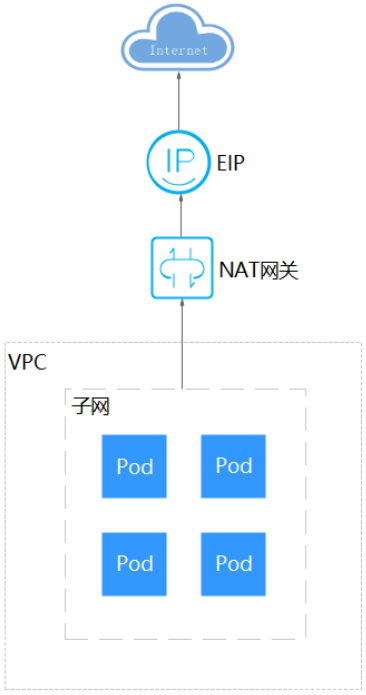
11.9 从容器访问公网

容器访问公网有如下方法可以实现。

- 给容器所在节点绑定弹性公网IP。
- 通过NAT网关配置SNAT规则，通过NAT网关访问公网。

下面将详细讲解通过NAT网关访问公网的方法，NAT网关能够为VPC内的容器实例提供网络地址转换（Network Address Translation）服务，SNAT功能通过绑定弹性公网IP，实现私有IP向公有IP的转换，可实现VPC内的容器实例共享弹性公网IP访问Internet。其原理如图11-28所示。通过NAT网关的SNAT功能，即使VPC内的容器实例不配置弹性公网IP也可以直接访问Internet，提供超大并发数的连接服务，适用于请求量大、连接数多的服务。

图 11-28 SNAT



您可以通过如下步骤实现容器实例访问Internet。

步骤1 创建弹性公网IP。

1. 登录管理控制台。
2. 在管理控制台左上角单击，选择区域和项目。
3. 在控制台首页，单击左上角的，在展开的列表中单击“网络 > 弹性公网IP”。
4. 在“弹性公网IP”界面，单击“创建弹性公网IP”。
5. 根据界面提示配置参数。

   说明

   此处“区域”需选择容器实例所在区域。

步骤2 创建NAT网关。

1. 登录管理控制台。
2. 在管理控制台左上角单击，选择区域和项目。
3. 在控制台首页，单击左上角的，在展开的列表中单击“网络 > NAT网关”。
4. 在NAT网关页面，单击右上角的“创建公网NAT网关”。
5. 根据界面提示配置参数。

   说明

   此处需选择集群相同的VPC。

步骤3 配置SNAT规则，为子网绑定弹性公网IP。

1. 登录管理控制台。
2. 在管理控制台左上角单击 ，选择区域和项目。
3. 在控制台首页，单击左上角的 ，在展开的列表中单击“网络 >NAT网关”。
4. 在NAT网关页面，单击需要添加SNAT规则的NAT网关名称。
5. 在SNAT规则页签中，单击“添加SNAT规则”。
6. 根据界面提示配置参数。

说明

SNAT规则是按网段生效，因为不同容器网络模型通信方式不同，此处子网需按如下规则选择。

- 容器隧道网络、VPC网络：需要选择节点所在子网，即创建节点时选择的子网。
- 云原生2.0网络：需要选择容器所在子网，即创建集群时选择的容器子网。

对于存在多个网段的情况，可以创建多个SNAT规则或选择自定义网段，只要网段能包含容器子网（云原生2.0网络）或节点子网即可。

SNAT规则配置完成后，您就可以从容器中访问公网了，从容器中能够ping通公网。

----结束

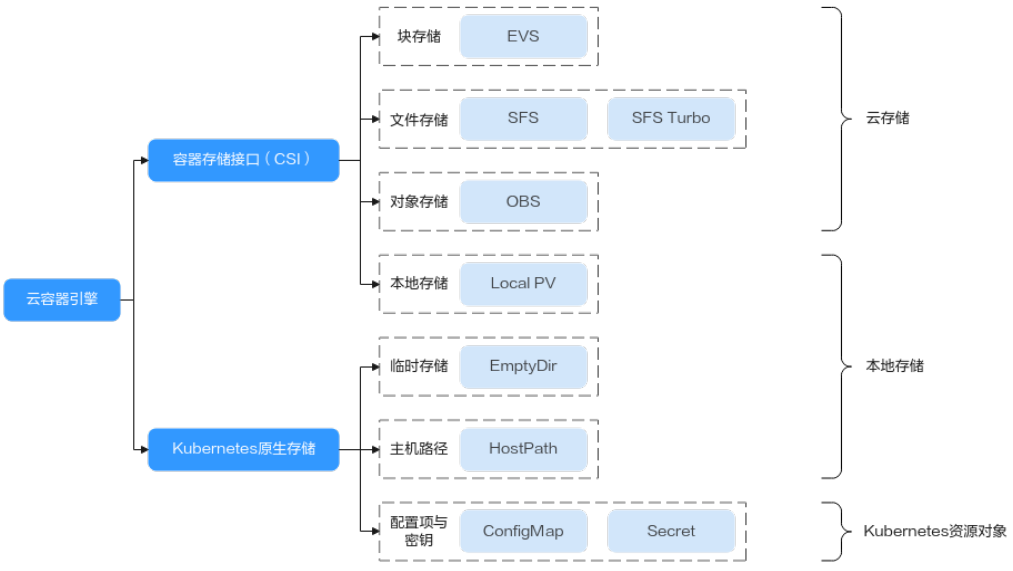
12 存储

12.1 存储概述

存储概览

CCE的容器存储功能基于Kubernetes容器存储接口（CSI）实现，深度融合多种类型的云存储并全面覆盖不同的应用场景，而且完全兼容Kubernetes原生的存储服务，例如EmptyDir、HostPath、Secret、ConfigMap等本地存储。

图 12-1 容器存储概览类型



CCE支持工作负载Pod绑定多种类型的云存储，每种存储卷的主要特点及应用场景如下表：

表 12-1 云存储对比

对比维度	云硬盘EVS	文件存储SFS	极速文件存储 SFS Turbo	对象存储OBS
概念	云硬盘（Elastic Volume Service）可以为云服务器提供高可靠、高性能、规格丰富并且可弹性扩展的块存储服务，可满足不同场景的业务需求，适用于分布式文件系统、开发测试、数据仓库以及高性能计算等场景。	SFS为用户提供一个完全托管的共享文件存储，能够弹性伸缩至PB规模，具备高可用性和持久性，为海量数据、高带宽型应用提供有力支持。适用于多种应用场景，包括HPC、媒体处理、文件共享、内容管理和Web服务等。	SFS Turbo为用户提供一个完全托管的共享文件存储，能够弹性伸缩至320TB规模，具备高可用性和持久性，为海量的小文件、低延迟高IOPS型应用提供有力支持。适用于多种应用场景，包括高性能网站、日志存储、压缩解压、DevOps、企业办公、容器应用等。	对象存储服务（Object Storage Service，OBS）提供海量、安全、高可靠、低成本的数据存储能力，可供用户存储任意类型和大小的数据。适合企业备份/归档、视频点播、视频监控等多种数据存储场景。
存储数据的逻辑	存放的是二进制数据，无法直接存放文件，如果需要存放文件，需要先格式化文件系统后使用。	存放的是文件，会以文件和文件夹的层次结构来整理和呈现数据。	存放的是文件，会以文件和文件夹的层次结构来整理和呈现数据。	存放的是对象，可以直接存放文件，文件会自动产生对应的系统元数据，用户也可以自定义文件的元数据。
访问方式	只能在ECS/BMS中挂载使用，不能被操作系统应用直接访问，需要格式化成文件系统进行访问。	在ECS/BMS中通过网络协议挂载使用。需要指定网络地址进行访问，也可以将网络地址映射为本地目录后进行访问。	提供标准的文件访问协议NFS（仅支持NFSv3），用户可以将现有应用和工具与SFS Turbo无缝集成。	可以通过互联网或专线访问。需要指定桶地址进行访问，使用的是HTTP和HTTPS等传输协议。
静态数据卷	支持，请参见**通过静态存储卷使用已有云硬盘**。	支持，请参见**通过静态存储卷使用已有文件存储**。	支持，请参见**通过静态存储卷使用已有极速文件存储**。	支持，请参见**通过静态存储卷使用已有对象存储**。
动态数据卷	支持，请参见**通过动态存储卷使用云硬盘**。	支持，请参见**通过动态存储卷使用文件存储**。	不支持	支持，请参见**通过动态存储卷使用对象存储**。
主要特点	非共享存储，每个云盘只能在单个节点挂载。	共享存储，可提供高性能、高吞吐存储服务。	高性能、高带宽、共享存储。	共享存储，用户态文件系统。

对比维度	云硬盘EVS	文件存储SFS	极速文件存储 SFS Turbo	对象存储OBS
应用场景	HPC高性能计算、企业核心集群应用、企业应用系统和开发测试等。 **说明** 高性能计算：主要是高速率、高IOPS的需求，用于作为高性能存储，比如工业设计、能源勘探等。	HPC高性能计算、媒体处理、内容管理和Web服务、大数据和分析应用程序等。 **说明** 高性能计算：主要是高带宽的需求，用于共享文件存储，比如基因测序、图片渲染等。	高性能网站、日志存储、DevOps、企业办公等。	大数据分析、静态网站托管、在线视频点播、基因测序、智能视频监控、备份归档、企业云盘（网盘）等。
容量	TB级别	SFS 1.0：PB级别	通用型：TB级别	EB级别
时延	1~2ms	SFS 1.0：3~20ms	通用型：1~5ms	10ms
IOPS/TPS	单盘 33K	SFS 1.0：2K	通用型：最大达100K	千万级
带宽	MB/s级别	SFS 1.0：GB/s级别	通用型：最大为GB/s级别	TB/s级别

企业项目支持说明

说明

该功能需要everest插件升级到1.2.33及以上版本。

- 自动创建存储：

  CCE支持使用存储类创建云硬盘和对象存储类型PVC时指定企业项目，将创建的存储资源（云硬盘和对象存储）归属于指定的企业项目下，**企业项目可选为集群所属的企业项目或default企业项目**。

  若不指定企业项目，则创建的存储资源默认使用存储类StorageClass中指定的企业项目。

  - 对于自定义的StorageClass，可以在StorageClass中指定企业项目，详见**指定StorageClass的企业项目**。StorageClass中如不指定的企业项目，则默认为default企业项目。
  - 对于CCE提供的 csi-disk 和 csi-obs 存储类，所创建的存储资源属于default企业项目。

- 使用已有存储：

  使用PV创建PVC时，因为存储资源在创建时已经指定了企业项目，如果PVC中指定企业项目，则务必确保在PVC和PV中指定的everest.io/enterprise-project-id保持一致，否则两者无法正常绑定。

相关文档

- 存储基础知识
- 云硬盘存储（EVS）
- 文件存储（SFS）
- 极速文件存储（SFS Turbo）
- 对象存储（OBS）

12.2 存储基础知识

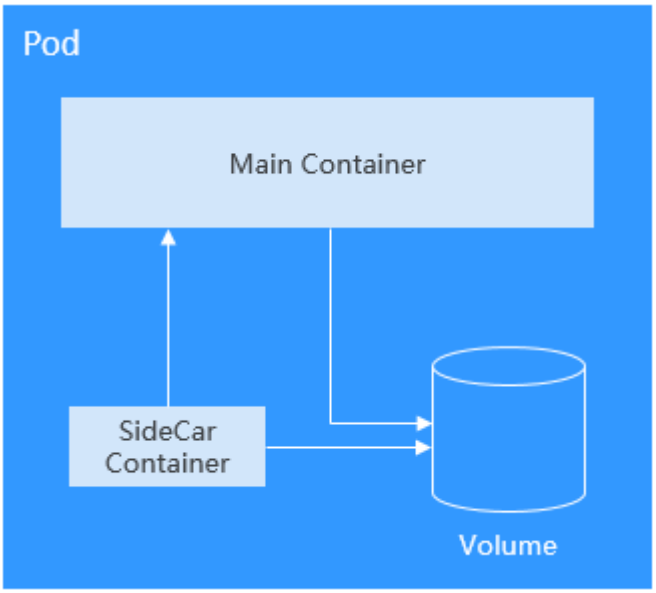
Volume（卷）

容器中的文件在磁盘上是临时存放的，这给容器中运行的较重要的应用程序带来如下两个问题：

1. 当容器重建时，容器中的文件将会丢失。
2. 当在一个Pod中同时运行多个容器时，容器间需要共享文件。

Kubernetes抽象出了Volume（卷）来解决以上两个问题。Kubernetes的Volume是Pod的一部分，Volume不是单独的对象，不能独立创建，只能在Pod中定义。Pod中的所有容器都可以使用Volume，但需要将Volume挂载到容器中的目录下。

实际中使用容器存储如下图所示，可将同一个Volume挂载到不同的容器中，实现不同容器间的存储共享。



存储卷的基本使用原则如下：

- 一个Pod可以挂载多个Volume。虽然单Pod可以挂载多个Volume，但是并不建议给一个Pod挂载过多卷。
- 一个Pod可以挂载多种类型的Volume。
- 每个被Pod挂载的Volume卷，可以在不同的容器间共享。
- Kubernetes环境推荐使用PVC和PV方式挂载Volume。

说明

卷（Volume）的生命周期与挂载它的Pod相同，即Pod被删除的时候，Volume也一起被删除。但是Volume里面的文件可能在Volume消失后仍然存在，这取决于Volume的类型。

Kubernetes提供了非常丰富的Volume类型，主要可分为In-Tree和Out-of-Tree两个大类：

卷（Volume）分类	描述
In-Tree	In-Tree卷是通过Kubernetes代码仓库维护的，与Kubernetes二进制文件一起构建、编译、发布，当前Kubernetes已不再接受这种模式的卷类型。 例如HostPath、EmptyDir、Secret和ConfigMap等Kubernetes原生支持的卷都属于这个类型。 而PVC（PersistentVolumeClaim）可以说是一种特殊的In-Tree卷，Kubernetes使用这种类型的卷从In-Tree模式向Out-of-Tree模式进行转换，这种类型的卷允许使用者在不同的存储供应商环境中“申请”使用底层存储创建的PV（PersistentVolume）。
Out-of-Tree	Out-of-Tree卷包括容器存储接口（CSI）和FlexVolume（已弃用），存储供应商只需遵循一定的规范即可创建自定义存储插件，创建可供Kubernetes使用的PV，而无需将插件源码添加到Kubernetes代码仓库。例如SFS、OBS等云存储都是通过在集群中安装存储驱动的形式使用的，需要在集群中创建对应的PV，然后使用PVC挂载到Pod中。

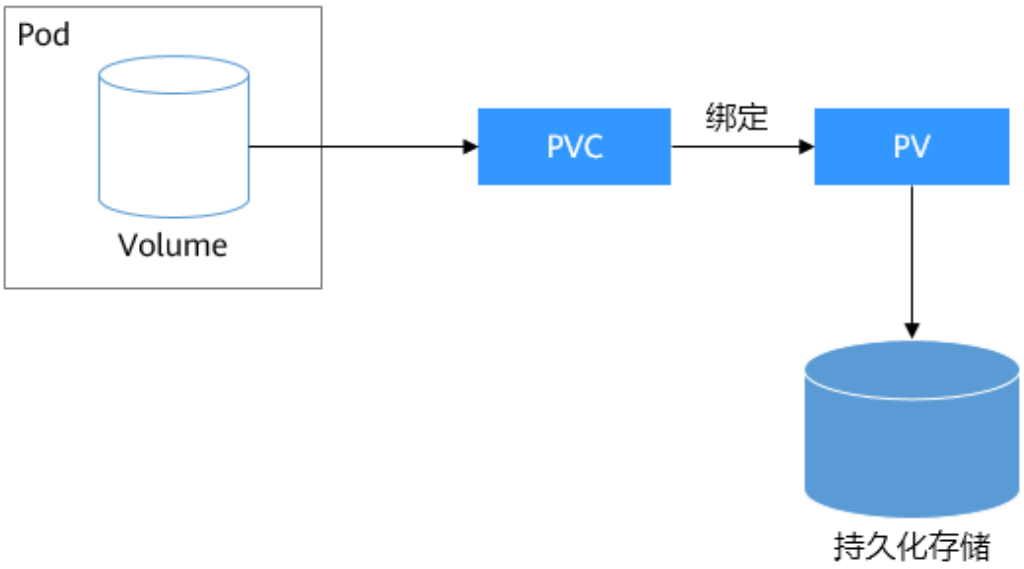
PV 与 PVC

Kubernetes抽象了PV（PersistentVolume）和PVC（PersistentVolumeClaim）来定义和使用存储，从而让使用者不用关心具体的基础设施，当需要存储资源的时候，只要像CPU和内存一样，声明要多少即可。

- PV：PV是PersistentVolume的缩写，译为持久化存储卷，描述的是一个集群里的持久化存储卷，它和节点一样，属于集群级别资源，其对象作用范围是整个Kubernetes集群。PV可以有自己的独立生命周期，不依附于Pod。
- PVC：PVC是PersistentVolumeClaim的缩写，译为持久化存储卷声明，描述的是负载对存储的申领。为应用配置存储时，需要声明一个存储需求（即PVC），Kubernetes会通过最佳匹配的方式选择一个满足需求的PV，并与PVC绑定。PVC与PV是一一对应关系，在创建PVC时，需描述请求的持久化存储的属性，比如，存储的大小、可读写权限等等。

在Pod中可以使用Volume关联PVC，即可让Pod使用到存储资源，它们之间的关系如下图所示。

图 12-2 PVC 绑定 PV



CSI

CSI（Container Storage Interface，容器存储接口）是容器标准存储接口规范，也是Kubernetes社区推荐的存储插件实现方案。everest是CCE基于CSI开发的自研存储插件，能够为容器提供不同类型的持久化存储功能。

存储卷访问模式

存储卷只能以底层存储资源所支持的方式挂载到宿主系统上。例如，文件存储可以支持多个节点读写，云硬盘只能被一个节点读写。

- ReadWriteOnce：存储卷可以被一个节点以读写方式挂载。
- ReadWriteMany：存储卷可以被多个节点以读写方式挂载。

表 12-2 存储卷支持的访问模式

存储类型	ReadWriteOnce	ReadWriteMany
云硬盘EVS	√	×
文件存储SFS	×	√
对象存储OBS	×	√
极速文件存储SFS Turbo	×	√

存储卷挂载方式

通常在使用存储卷时，可以通过以下方式挂载：

可以使用PV描述已有的存储资源，然后通过创建PVC在Pod中使用存储资源。也可以使用动态创建的方式，在PVC中指定**存储类（StorageClass）**，利用StorageClass中的Provisioner自动创建PV来绑定PVC。

表 12-3 挂载存储卷的方式

挂载方式	说明	支持的存储卷类型	其他限制
静态创建存储卷（使用已有存储）	即使用已有的存储（例如云硬盘、文件存储等）创建好PV，并通过PVC在工作负载中挂载。Kubernetes会将PVC和匹配的PV进行绑定，这样就实现了工作负载访问存储服务的能力。	所有存储卷均支持	无
动态创建存储卷（自动创建存储）	即在PVC中指定**存储类（StorageClass）**，由存储Provisioner根据需求创建底层存储介质，实现PV的自动化创建并直接绑定至PVC。	云硬盘存储、对象存储、文件存储、本地持久卷	无
动态挂载（VolumeClaimTemplate）	动态挂载能力通过卷申领模板（**volumeClaimTemplates**字段）实现，并依赖于StorageClass的动态创建PV能力。动态挂载可以为每一个Pod关联一个独有的PVC及PV，当Pod被重新调度后，仍然能够根据该PVC名称挂载原有的数据。	仅云硬盘存储、本地持久卷支持	仅有状态工作负载支持

PV 回收策略

PV回收策略用于指定删除PVC时，底层卷的回收策略，支持设定Delete、Retain回收策略。

- Delete：删除PVC的动作会将PV对象从Kubernetes中移除，同时也会从外部基础设施中移除所关联的底层存储资产。
- Retain：当PVC对象被删除时，PV对象与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，且不能直接再次被PVC绑定使用。

  您可以通过以下步骤来手动删除回收：

  a. 手动删除PersistentVolume对象。
  b. 根据情况，手动清除所关联的底层存储资源上的数据。
  c. 手动删除所关联的底层存储资源。

  如果您希望重用该底层存储资源，可以重新创建新的PersistentVolume对象。

CCE还支持一种删除PVC时不删除底层存储资源的使用方法，当前仅支持使用YAML创建：PV回收策略设置为Delete，并添加annotations“everest.io/reclaim-policy: retain-volume-only”。这样在删除PVC时，PV会被删除，但底层存储资源会保留。

以云硬盘为例，YAML示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test
  namespace: default
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/disk-volume-type: SAS
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
  volumeName: pv-evs-test

---
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
    everest.io/reclaim-policy: retain-volume-only
  name: pv-evs-test
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 10Gi
  csi:
    driver: disk.csi.everest.io
    fsType: ext4
    volumeHandle: 2af98016-6082-4ad6-bedc-1a9c673aef20
    volumeAttributes:
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
      everest.io/disk-mode: SCSI
      everest.io/disk-volume-type: SAS
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-disk
```

相关文档

- 更多关于Kubernetes存储的信息，请参见**Storage**。
- 更多关于CCE容器存储的信息，请参见**存储概述**。

12.3 云硬盘存储（EVS）

12.3.1 云硬盘概述

为满足数据持久化的需求，CCE支持将云硬盘（EVS）创建的存储卷挂载到容器的某一路径下，当容器在同一可用区内迁移时，挂载的云硬盘将一同迁移。通过云硬盘，可以将存储系统的远端文件目录挂载到容器中，数据卷中的数据将被永久保存，即使删除了容器，数据卷中的数据依然保存在存储系统中。

云硬盘性能规格

云硬盘性能的主要指标包括：

- IOPS：云硬盘每秒进行读写的操作次数。
- 吞吐量：云硬盘每秒成功传送的数据量，即读取和写入的数据量。
- IO读写时延：云硬盘连续两次进行读写操作所需要的最小时间间隔。

表 12-4 云硬盘性能规格

参数	超高IO	高IO	普通IO
云硬盘最大容量（GiB）	• 系统盘：1024 • 数据盘：32768	• 系统盘：1024 • 数据盘：32768	• 系统盘：1024 • 数据盘：32768
最大IOPS	50000	5000	2200
最大吞吐量（MiB/s）	350	150	50
IOPS突发上限	16000	5000	2200
云硬盘IOPS性能计算公式	IOPS = min (50000, 1800 + 50 × 容量)	IOPS = min (5000, 1800 + 8 × 容量)	IOPS = min (2200, 500 + 2 × 容量)
云硬盘吞吐量性能计算公式（MiB/s）	吞吐量 = min (350, 120 + 0.5 × 容量)	吞吐量 = min (150, 100 + 0.15 × 容量)	吞吐量 = 50
单队列访问时延（ms）	1	1~ 3	5~ 10
API名称	SSD	SAS	SATA

使用场景

根据使用场景不同，云硬盘类型的存储支持以下挂载方式：

- **通过静态存储卷使用已有云硬盘**：即静态创建的方式，需要先使用已有的云硬盘创建PV，然后通过PVC在工作负载中挂载存储。适用于已有可用的底层存储的场景。
- **通过动态存储卷使用云硬盘**：即动态创建的方式，无需预先创建云硬盘，在创建PVC时通过指定存储类（StorageClass），即可自动创建云硬盘和对应的PV对象。适用于无可用的底层存储，需要新创建的场景。
- **有状态负载动态挂载云硬盘存储**：仅有状态工作负载支持，可以为每一个Pod关联一个独有的PVC及PV，当Pod被重新调度后，仍然能够根据该PVC名称挂载原有的数据。适用于多实例的有状态工作负载。

12.3.2 通过静态存储卷使用已有云硬盘

CCE支持使用已有的云硬盘创建存储卷（PersistentVolume）。创建成功后，通过创建相应的PersistentVolumeClaim绑定当前PersistentVolume使用。适用于已有底层存储的场景。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 您已经创建好一块云硬盘，并且云硬盘满足以下条件：
  - 已有的云硬盘不可以是系统盘、专属盘或共享盘。
  - 云硬盘模式需选择SCSI（购买云硬盘时默认为VBD模式）。
  - 云硬盘的状态可用，且未被其他资源使用。
  - 云硬盘的可用区需要与集群节点的可用区相同，否则无法挂载将导致实例启动失败。
  - 若云硬盘加密，所使用的密钥状态需可用。
  - 仅支持选择集群所属企业项目和default企业项目下的云硬盘。
  - 不支持使用已进行分区或者非ext4文件系统的云硬盘。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。

约束与限制

- 云硬盘不支持跨可用区挂载，且不支持被多个工作负载、同一个工作负载的多个实例或多个任务使用。由于CCE集群各节点之间暂不支持共享盘的数据共享功能，多个节点挂载使用同一个云硬盘可能会出现读写冲突、数据缓存冲突等问题，所以创建无状态工作负载时，若使用了EVS云硬盘，建议工作负载只选择一个实例。
- 1.19.10以下版本的集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，当新Pod被调度到另一个节点时，会导致之前Pod不能正常读写。

  1.19.10及以上版本集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，新Pod会因为无法挂载云硬盘导致无法成功启动。

通过控制台使用已有云硬盘

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 静态创建存储卷声明和存储卷。

1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击右上角“创建存储卷声明”，在弹出的窗口中填写存储卷声明参数。

参数	描述
存储卷声明类型	本文中选择“云硬盘”。
PVC名称	输入PVC的名称，同一命名空间下的PVC名称需唯一。

参数	描述
创建方式	– 已有底层存储的场景下，根据是否已经创建存储卷可选择“新建存储卷”或“已有存储卷”来静态创建PVC。 – 无可用底层存储的场景下，可选择“动态创建”，具体操作请参见**通过动态存储卷使用云硬盘**。 本文示例中选择“新建存储卷”，可通过控制台同时创建PV及PVC。
关联存储卷[a]	选择集群中已有的PV卷，需要提前创建PV，请参考**相关操作**中的“创建存储卷”操作。 本文示例中无需选择。
云硬盘[b]	单击“选择云硬盘”，您可以在新页面中勾选满足要求的云硬盘，并单击“确定”。
PV名称[b]	输入PV名称，同一集群内的PV名称需唯一。
访问模式[b]	云硬盘类型的存储卷仅支持ReadWriteOnce，表示存储卷可以被一个节点以读写方式挂载，详情请参见**存储卷访问模式**。
回收策略[b]	您可以选择Delete或Retain，用于指定删除PVC时底层存储的回收策略，详情请参见**PV回收策略**。

说明

a：创建方式选择“已有存储卷”时可设置。

b：创建方式选择“新建存储卷”时可设置。

2. 单击“创建”，将同时为您创建存储卷声明及存储卷。

   您可以在左侧导航栏中选择“容器存储”，在“存储卷声明”和“存储卷”页签下查看已经创建的存储卷声明和存储卷。

步骤3 创建应用。

1. 在左侧导航栏中选择“工作负载”，在右侧选择“有状态负载”页签。
2. 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 已有存储卷声明 (PVC)”。

   本文主要为您介绍存储卷的挂载使用，如**表12-5**，其他参数详情请参见**工作负载**。

表 12-5 存储卷挂载

参数	参数说明
存储卷声明（PVC）	选择已有的云硬盘存储卷。 云硬盘存储卷无法被多个工作负载重复挂载。

参数	参数说明
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下 ，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	- 只读：只能读容器路径中的数据卷。 - 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

本例中将该盘挂载到容器中/data路径下，在该路径下生成的容器数据会存储到云硬盘中。

说明

由于云硬盘为非共享模式，工作负载下多个实例无法同时挂载，会导致实例启动异常。因此挂载云硬盘时，工作负载实例数需为1。

3. 其余信息都配置完成后，单击“创建工作负载”。

   工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

通过 kubectl 命令行使用已有云硬盘

步骤1 使用kubectl连接集群。

步骤2 创建PV。当您的集群中已存在创建完成的PV时，可跳过本步骤。

1. 创建pv-evs.yaml文件。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
    everest.io/reclaim-policy: retain-volume-only        # 可选字段，删除PV时可保留底层存储卷
  name: pv-evs    # PV的名称
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
```

```
  - ReadWriteOnce    # 访问模式，云硬盘必须为ReadWriteOnce
 capacity:
   storage: 10Gi      # 云硬盘的容量，单位为Gi，取值范围 1-32768
 csi:
   driver: disk.csi.everest.io    # 挂载依赖的存储驱动
   fsType: ext4
   volumeHandle: <your_volume_id>   #云硬盘的volumeID
   volumeAttributes:
     everest.io/disk-mode: SCSI          # 云硬盘的磁盘模式，仅支持SCSI
     everest.io/disk-volume-type: SAS    # 云硬盘的类型
     storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
     everest.io/crypt-key-id: <your_key_id>    # 可选字段，加密密钥ID，使用加密盘的时候填写
     everest.io/enterprise-project-id: <your_project_id>  # 可选字段，企业项目ID，如果指定企业项目，
则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
 persistentVolumeReclaimPolicy: Delete    # 回收策略
 storageClassName: csi-disk             # 存储类名称，云硬盘必须为csi-disk
```

表 12-6 关键参数说明

参数	是否必选	描述
everest.io/reclaim-policy: retain-volume-only	否	可选字段 目前仅支持配置“retain-volume-only” everest插件版本需 >= 1.2.9且回收策略为Delete时生效。如果回收策略是Delete且当前值设置为“retain-volume-only”删除PVC回收逻辑为：删除PV，保留底层存储卷。
failure-domain.beta.kubernetes.io/region	是	集群所在的region。 Region对应的值请参见**地区和终端节点**。
failure-domain.beta.kubernetes.io/zone	是	创建云硬盘所在的可用区，必须和工作负载规划的可用区保持一致。 zone对应的值请参见**地区和终端节点**。
volumeHandle	是	云硬盘的volumeID。 **获取方法：**在云服务器控制台，单击左侧栏目树中的“云硬盘 > 磁盘”，单击要对接的云硬盘名称进入详情页，在“概览信息”页签下单击“ID”后的复制图标即可获取云硬盘的volumeID。
everest.io/disk-volume-type	是	云硬盘类型，全大写。 - SATA：普通I/O - SAS：高I/O - SSD：超高I/O
everest.io/crypt-key-id	否	当云硬盘是加密卷时为必填，填写创建云硬盘时选择的加密密钥ID。 **获取方法**：在云服务器控制台，单击左侧栏目树中的“云硬盘 > 磁盘”，单击要对接的云硬盘名称进入详情页，在“概览信息”页签下找到“配置信息”，复制密钥ID值即可。

参数	是否必选	描述
everest.io/enterprise-project-id	否	可选字段 云硬盘的企业项目ID。如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。 **获取方法**：在云服务器控制台，单击左侧栏目树中的“云硬盘 > 磁盘”，单击要对接的云硬盘名称进入详情页，在“概览信息”页签下找到“管理信息”中的企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可获取云硬盘所属的企业项目的ID。
persistentVolumeReclaimPolicy	是	集群版本号>=1.19.10且everest插件版本>=1.2.9时正式开放回收策略支持。 支持Delete、Retain回收策略，详情请参见**PV回收策略**。如果数据安全性要求较高，建议使用**Retain**以免误删数据。 **Delete**: – Delete且不设置everest.io/reclaim-policy：删除PVC，PV资源与云硬盘均被删除。 – Delete且设置everest.io/reclaim-policy=retain-volume-only：删除PVC，PV资源被删除，云硬盘资源会保留。 **Retain**：删除PVC，PV资源与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，不能直接再次被PVC绑定使用。
storageClassName	是	云硬盘存储对应的存储类名称为csi-disk。

2. 执行以下命令，创建PV。
```
kubectl apply -f pv-evs.yaml
```

步骤3 创建PVC。

1. 创建pvc-evs.yaml文件。
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS    # 云硬盘的类型
    everest.io/crypt-key-id: <your_key_id>    # 可选字段，加密密钥ID，使用加密盘的时候填写
    everest.io/enterprise-project-id: <your_project_id> # 可选字段，企业项目ID，如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
  - ReadWriteOnce             # 云硬盘必须为ReadWriteOnce
```

```
resources:
  requests:
    storage: 10Gi            # 云硬盘大小，取值范围 1-32768，必须和已有PV的storage大小保持一致。
storageClassName: csi-disk   # StorageClass类型为云硬盘
volumeName: pv-evs           # PV的名称
```

表 12-7 关键参数说明

参数	是否必选	描述
failure-domain.beta.kubernetes.io/region	是	集群所在的region。 Region对应的值请参见**地区和终端节点**。
failure-domain.beta.kubernetes.io/zone	是	创建云硬盘所在的可用区，必须和工作负载规划的可用区保持一致。 zone对应的值请参见**地区和终端节点**。
storage	是	PVC申请容量，单位为Gi。 必须和已有PV的storage大小保持一致。
volumeName	是	PV的名称，必须与**1**中PV的名称一致。
storageClassName	是	存储类名称，必须与**1**中PV的存储类一致。 云硬盘存储对应的存储类名称为csi-disk。

2. 执行以下命令，创建PVC。

```
kubectl apply -f pvc-evs.yaml
```

步骤4 创建应用。

1. 创建web-evs.yaml文件，本示例中将云硬盘挂载至/data路径。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-evs
  namespace: default
spec:
  replicas: 1            # 使用云硬盘的工作负载副本数必须是1
  selector:
    matchLabels:
      app: web-evs
  serviceName: web-evs   # Headless Service名称
  template:
    metadata:
      labels:
        app: web-evs
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: pvc-disk     #卷名称，需与volumes字段中的卷名称对应
          mountPath: /data  #存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: pvc-disk     #卷名称，可自定义
          persistentVolumeClaim:
            claimName: pvc-evs     #已创建的PVC名称
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: web-evs   # Headless Service名称
  namespace: default
  labels:
    app: web-evs
spec:
  selector:
    app: web-evs
  clusterIP: None
  ports:
    - name: web-evs
      targetPort: 80
      nodePort: 0
      port: 80
      protocol: TCP
  type: ClusterIP
```

2. 执行以下命令，创建一个挂载云硬盘存储的应用。

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

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

验证数据持久化

步骤1 查看部署的应用及云硬盘文件。

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-evs
```

预期输出如下：

```
web-evs-0               1/1     Running   0             38s
```

2. 执行以下命令，查看云硬盘是否挂载至/data路径。

```
kubectl exec web-evs-0 -- df | grep data
```

预期输出如下：

```
/dev/sdc            10255636     36888  10202364   0% /data
```

3. 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-evs-0 -- ls /data
```

预期输出如下：

```
lost+found
```

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-evs-0 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-evs-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

步骤4 执行以下命令，删除名称为web-evs-0的Pod。

```
kubectl delete pod web-evs-0
```

预期输出如下：

```
pod "web-evs-0" deleted
```

步骤5 删除后，StatefulSet控制器会自动重新创建一个同名副本。执行以下命令，验证/data路径下的文件是否更改。

```
kubectl exec web-evs-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

static文件仍然存在，则说明云硬盘中的数据可持久化保存。

----结束

相关操作

您还可以执行表12-8中的操作。

表 12-8 其他操作

操作	说明	操作步骤
创建存储卷	通过CCE控制台单独创建PV。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷”页签。单击右上角“创建存储卷”，在弹出的窗口中填写存储卷声明参数。 • 存储卷类型：选择“云硬盘”。 • 云硬盘：单击“选择云硬盘”，在新页面中勾选满足要求的云硬盘，并单击“确定”。 • PV名称：输入PV名称，同一集群内的PV名称需唯一。 • 访问模式：仅支持ReadWriteOnce，表示存储卷可以被一个节点以读写方式挂载，详情请参见存储卷访问模式。 • 回收策略：Delete或Retain，详情请参见PV回收策略。 2. 单击“创建”。
扩容云硬盘存储卷	通过CCE控制台快速扩容已挂载的云硬盘。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 扩容”。 2. 输入新增容量，并单击“确定”。
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。

操作	说明	操作步骤
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.3.3 通过动态存储卷使用云硬盘

CCE支持指定存储类（StorageClass），自动创建云硬盘类型的底层存储和对应的存储卷，适用于无可用的底层存储，需要新创建的场景。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。

约束与限制

- 云硬盘不支持跨可用区挂载，且不支持被多个工作负载、同一个工作负载的多个实例或多个任务使用。由于CCE集群各节点之间暂不支持共享盘的数据共享功能，多个节点挂载使用同一个云硬盘可能会出现读写冲突、数据缓存冲突等问题，所以创建无状态工作负载时，若使用了EVS云硬盘，建议工作负载只选择一个实例。
- 1.19.10以下版本的集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，当新Pod被调度到另一个节点时，会导致之前Pod不能正常读写。

  1.19.10及以上版本集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，新Pod会因为无法挂载云硬盘导致无法成功启动。

通过控制台自动创建云硬盘存储

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 动态创建存储卷声明和存储卷。

1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击右上角“创建存储卷声明”，在弹出的窗口中填写存储卷声明参数。

参数	描述
存储卷声明类型	本文中选择“云硬盘”。
PVC名称	输入PVC的名称，同一命名空间下的PVC名称需唯一。
创建方式	- 无可用底层存储的场景下，可选择“动态创建”，通过控制台级联创建存储卷声明PVC、存储卷PV和底层存储。 - 已有底层存储的场景下，根据是否已经创建PV可选择“新建存储卷”或“已有存储卷”，静态创建PVC，具体操作请参见**通过静态存储卷使用已有云硬盘**。 本文中选择“动态创建”。

参数	描述
存储类	云硬盘对应的存储类为csi-disk。
可用区	选择云硬盘的可用区，需要与集群节点的可用区相同。 **说明** 云硬盘只能挂载到同一可用区的节点上，创建后不支持更换可用区，请谨慎选择。
云硬盘类型	选择云硬盘类型。
访问模式	云硬盘类型的存储卷仅支持ReadWriteOnce，表示存储卷可以被一个节点以读写方式挂载，详情请参见**存储卷访问模式**。
容量（GiB）	申请的存储卷容量大小。
加密	勾选底层存储是否加密，勾选后需要选择使用的加密密钥。使用前请确认云硬盘所在区域（Region）是否支持硬盘加密能力。
企业项目	仅支持default、集群所在企业项目或存储类指定的企业项目。

2. 单击“创建”。

   您可以在左侧导航栏中选择“容器存储”，在“存储卷声明”和“存储卷”页签下查看已经创建的存储卷声明和存储卷。

步骤3 创建应用。

1. 在左侧导航栏中选择“工作负载”，在右侧选择“有状态负载”页签。
2. 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 已有存储卷声明 (PVC)”。

   本文主要为您介绍存储卷的挂载使用，如**表12-9**，其他参数详情请参见**工作负载**。

表 12-9 存储卷挂载

参数	参数说明
存储卷声明（PVC）	选择已有的云硬盘存储卷。 云硬盘存储卷无法被多个工作负载重复挂载。
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“**/var/run**”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。

参数	参数说明
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	- 只读：只能读容器路径中的数据卷。 - 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

本例中将该盘挂载到容器中/data路径下，在该路径下生成的容器数据会存储到云硬盘中。

说明

由于云硬盘为非共享模式，工作负载下多个实例无法同时挂载，会导致实例启动异常。因此挂载云硬盘时，工作负载实例数需为1。

3. 其余信息都配置完成后，单击“创建工作负载”。

   工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

使用 kubectl 自动创建云硬盘存储

步骤1 使用kubectl连接集群。

步骤2 使用StorageClass动态创建PVC及PV。

1. 创建pvc-evs-auto.yaml文件。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-auto
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS    # 云硬盘的类型
    everest.io/crypt-key-id: <your_key_id>    # 可选字段，加密密钥ID，使用加密盘的时候填写
    everest.io/enterprise-project-id: <your_project_id> # 可选字段，企业项目ID，如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>      # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
  - ReadWriteOnce              # 云硬盘必须为ReadWriteOnce
  resources:
    requests:
      storage: 10Gi            # 云硬盘大小，取值范围 1-32768
  storageClassName: csi-disk    # StorageClass类型为云硬盘
```

表 12-10 关键参数说明

参数	是否必选	描述
failure-domain.beta.kubernetes.io/region	是	集群所在的region。 Region对应的值请参见**地区和终端节点**。
failure-domain.beta.kubernetes.io/zone	是	创建云硬盘所在的可用区，必须和工作负载规划的可用区保持一致。 zone对应的值请参见**地区和终端节点**。
everest.io/disk-volume-type	是	云硬盘类型，全大写。 – SATA：普通I/O – SAS：高I/O – SSD：超高I/O
everest.io/crypt-key-id	否	当云硬盘是加密卷时为必填，填写创建云硬盘时选择的加密密钥ID，可使用自定义密钥或名为“evs/default”的云硬盘默认密钥。 **获取方法**：在数据加密控制台，找到需要加密的密钥，复制密钥ID值即可。
everest.io/enterprise-project-id	否	可选字段 云硬盘的企业项目ID。如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。 **获取方法**：在云服务器控制台，单击左侧栏目树中的“云硬盘 > 磁盘”，单击要对接的云硬盘名称进入详情页，在“概览信息”页签下找到“管理信息”中的企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可获取云硬盘所属的企业项目的ID。
storage	是	PVC申请容量，单位为Gi，取值范围为1-32768。
storageClassName	是	云硬盘存储对应的存储类名称为csi-disk。

2. 执行以下命令，创建PVC。
```
kubectl apply -f pvc-evs-auto.yaml
```

步骤3 创建应用。

1. 创建web-evs-auto.yaml文件，本示例中将云硬盘挂载至/data路径。
```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-evs-auto
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web-evs-auto
```

```
  serviceName: web-evs-auto   # Headless Service名称
  template:
    metadata:
      labels:
        app: web-evs-auto
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: pvc-disk    #卷名称，需与volumes字段中的卷名称对应
          mountPath: /data  #存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: pvc-disk    #卷名称，可自定义
          persistentVolumeClaim:
            claimName: pvc-evs-auto    #已创建的PVC名称
---
apiVersion: v1
kind: Service
metadata:
  name: web-evs-auto   # Headless Service名称
  namespace: default
  labels:
    app: web-evs-auto
spec:
  selector:
    app: web-evs-auto
  clusterIP: None
  ports:
    - name: web-evs-auto
      targetPort: 80
      nodePort: 0
      port: 80
      protocol: TCP
  type: ClusterIP
```

2. 执行以下命令，创建一个挂载云硬盘存储的应用。

```
kubectl apply -f web-evs-auto.yaml
```

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

验证数据持久化

步骤1 查看部署的应用及云硬盘文件。

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-evs-auto
```

预期输出如下：

```
web-evs-auto-0                1/1     Running   0             38s
```

2. 执行以下命令，查看云硬盘是否挂载至/data路径。

```
kubectl exec web-evs-auto-0 -- df | grep data
```

预期输出如下：

```
/dev/sdc            10255636     36888  10202364   0% /data
```

3. 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-evs-auto-0 -- ls /data
```

预期输出如下：

```
lost+found
```

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-evs-auto-0 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-evs-auto-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

步骤4 执行以下命令，删除名称为web-evs-auto-0的Pod。

```
kubectl delete pod web-evs-auto-0
```

预期输出如下：

```
pod "web-evs-auto-0" deleted
```

步骤5 删除后，StatefulSet控制器会自动重新创建一个同名副本。执行以下命令，验证/data路径下的文件是否更改。

```
kubectl exec web-evs-auto-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

static文件仍然存在，则说明云硬盘中的数据可持久化保存。

----结束

相关操作

您还可以执行表12-11中的操作。

表 12-11 其他操作

操作	说明	操作步骤
扩容云硬盘存储卷	通过CCE控制台快速扩容已挂载的云硬盘。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 扩容”。 2. 输入新增容量，并单击“确定”。
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.3.4 有状态负载动态挂载云硬盘存储

使用场景

动态挂载仅可在创建**有状态负载**时使用，通过卷声明模板（**volumeClaimTemplates**字段）实现，并依赖于StorageClass的动态创建PV能力。在多实例的有状态负载中，动态挂载可以为每一个Pod关联一个独有的PVC及PV，当Pod被重新调度后，仍然能够根据该PVC名称挂载原有的数据。而在无状态工作负载的普通挂载方式中，当存储支持多点挂载（ReadWriteMany）时，工作负载下的多个Pod会被挂载到同一个底层存储中。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。

通过控制台动态挂载云硬盘存储

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“工作负载”，在右侧选择“有状态负载”页签。

步骤3 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 动态挂载 (VolumeClaimTemplate)”。

步骤4 单击“创建存储卷声明”，在弹出窗口中填写存储卷声明参数。

参数填写完成后，单击“创建”。

参数	描述
存储卷声明类型	本文中选择“云硬盘”。
PVC名称	输入PVC的名称。创建后将根据实例数自动增加后缀，格式为<自定义PVC名称>-<序号>，例如example-0。
创建方式	可选择“动态创建”，通过控制台级联创建存储卷声明PVC、存储卷PV和底层存储。
存储类	云硬盘对应的存储类为csi-disk。
可用区	选择云硬盘的可用区，需要与集群节点的可用区相同。 **说明** 云硬盘只能挂载到同一可用区的节点上，创建后不支持更换可用区，请谨慎选择。
云硬盘类型	选择云硬盘类型。
访问模式	云硬盘类型的存储卷仅支持ReadWriteOnce，表示存储卷可以被一个节点以读写方式挂载，详情请参见**存储卷访问模式**。
容量（GiB）	申请的存储卷容量大小。
加密	勾选底层存储是否加密，勾选后需要选择使用的加密密钥。仅云硬盘和文件存储支持加密。

参数	描述
企业项目	仅支持default、集群所在企业项目或存储类指定的企业项目。

步骤5 填写挂载路径。

表 12-12 存储卷挂载

参数	参数说明
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下 ，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	• 只读：只能读容器路径中的数据卷。 • 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

本例中将该盘挂载到容器中/data路径下，在该路径下生成的容器数据会存储到云硬盘中。

步骤6 本文主要为您介绍存储卷的动态挂载使用，其他参数详情请参见**创建有状态负载（StatefulSet）**。其余信息都配置完成后，单击“创建工作负载”。

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

通过 kubectl 命令行使用已有存储

步骤1 使用kubectl连接集群。

步骤2 创建statefulset-evs.yaml文件，本示例中将云硬盘挂载至/data路径。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: statefulset-evs
  namespace: default
spec:
  selector:
    matchLabels:
```

```
    app: statefulset-evs
  template:
    metadata:
      labels:
        app: statefulset-evs
    spec:
      containers:
        - name: container-1
          image: nginx:latest
          volumeMounts:
            - name: pvc-disk          # 需与volumeClaimTemplates字段中的名称对应
              mountPath: /data        # 存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
  serviceName: statefulset-evs        # Headless Service名称
  replicas: 2
  volumeClaimTemplates:
    - apiVersion: v1
      kind: PersistentVolumeClaim
      metadata:
        name: pvc-disk
        namespace: default
        annotations:
          everest.io/disk-volume-type: SAS    # 云硬盘的类型
          everest.io/crypt-key-id: <your_key_id>    # 可选字段，加密密钥ID，使用加密盘的时候填写
          everest.io/enterprise-project-id: <your_project_id> # 可选字段，企业项目ID，如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
        labels:
          failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
          failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
      spec:
        accessModes:
          - ReadWriteOnce             # 云硬盘必须为ReadWriteOnce
        resources:
          requests:
            storage: 10Gi             # 云硬盘大小，取值范围 1-32768
        storageClassName: csi-disk      # StorageClass类型为云硬盘
---
apiVersion: v1
kind: Service
metadata:
  name: statefulset-evs   # Headless Service名称
  namespace: default
  labels:
    app: statefulset-evs
spec:
  selector:
    app: statefulset-evs
  clusterIP: None
  ports:
    - name: statefulset-evs
      targetPort: 80
      nodePort: 0
      port: 80
      protocol: TCP
  type: ClusterIP
```

表 12-13 关键参数说明

参数	是否必选	描述
failure-domain.beta.kubernetes.io/region	是	集群所在的region。 Region对应的值请参见**地区和终端节点**。

参数	是否必选	描述
failure-domain.beta.kubernetes.io/zone	是	创建云硬盘所在的可用区，必须和工作负载规划的可用区保持一致。 zone对应的值请参见**地区和终端节点**。
everest.io/disk-volume-type	是	云硬盘类型，全大写。 ● SATA：普通I/O ● SAS：高I/O ● SSD：超高I/O
everest.io/crypt-key-id	否	当云硬盘是加密卷时为必填，填写创建云硬盘时选择的加密密钥ID。 **获取方法**：在云服务器控制台，单击左侧栏目树中的“云硬盘 > 磁盘”，单击要对接的云硬盘名称进入详情页，在“概览信息”页签下找到“配置信息”，复制密钥ID值即可。
everest.io/enterprise-project-id	否	可选字段 云硬盘的企业项目ID。如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。 **获取方法**：在云服务器控制台，单击左侧栏目树中的“云硬盘 > 磁盘”，单击要对接的云硬盘名称进入详情页，在“概览信息”页签下找到“管理信息”中的企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可获取云硬盘所属的企业项目的ID。
storage	是	PVC申请容量，单位为Gi，取值范围为1-32768。
storageClassName	是	云硬盘存储对应的存储类名称为csi-disk。

步骤3 执行以下命令，创建一个挂载云硬盘存储的应用。

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

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

验证数据持久化

步骤1 查看部署的应用及云硬盘文件。

1. 执行以下命令，查看已创建的Pod。

   ```
   kubectl get pod | grep statefulset-evs
   ```

   预期输出如下：

   ```
   statefulset-evs-0      1/1     Running   0          45s
   statefulset-evs-1      1/1     Running   0          28s
   ```

2. 执行以下命令，查看云硬盘是否挂载至/data路径。

```
kubectl exec statefulset-evs-0 -- df | grep data
```

预期输出如下：

```
/dev/sdd            10255636     36888  10202364   0% /data
```

3. 执行以下命令，查看/data路径下的文件。

```
kubectl exec statefulset-evs-0 -- ls /data
```

预期输出如下：

```
lost+found
```

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec statefulset-evs-0 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec statefulset-evs-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

步骤4 执行以下命令，删除名称为web-evs-auto-0的Pod。

```
kubectl delete pod statefulset-evs-0
```

预期输出如下：

```
pod "statefulset-evs-0" deleted
```

步骤5 删除后，StatefulSet控制器会自动重新创建一个同名副本。执行以下命令，验证/data路径下的文件是否更改。

```
kubectl exec statefulset-evs-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

static文件仍然存在，则说明云硬盘中的数据可持久化保存。

----结束

相关操作

您还可以执行表12-14中的操作。

表 12-14 其他操作

操作	说明	操作步骤
扩容云硬盘存储卷	通过CCE控制台快速扩容已挂载的云硬盘。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 扩容”。 2. 输入新增容量，并单击“确定”。
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。

操作	说明	操作步骤
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.3.5 快照与备份

CCE通过云硬盘EVS服务为您提供快照功能，云硬盘快照简称快照，指云硬盘数据在某个时刻的完整拷贝或镜像，是一种重要的数据容灾手段，当数据丢失时，可通过快照将数据完整的恢复到快照时间点。

您可以创建快照，从而快速保存指定时刻云硬盘的数据。同时，您还可以通过快照创建新的云硬盘，这样云硬盘在初始状态就具有快照中的数据。

使用须知

- 快照功能**仅支持v1.15及以上版本**的集群，且需要安装基于CSI的everest插件才可以使用。
- 基于快照创建的云硬盘，其子类型（普通IO/高IO/超高IO）、是否加密、磁盘模式（VBD/SCSI）、共享性(非共享/共享)、容量等都要与快照关联母盘保持一致，这些属性查询和设置出来后不能够修改。
- 只有可用或正在使用状态的磁盘能创建快照，且单个磁盘最大支持创建7个快照。
- 创建快照功能仅支持使用everest插件提供的存储类（StorageClass名称以csi开头）创建的PVC。使用Flexvolume存储类（StorageClass名为ssd、sas、sata）创建的PVC，无法创建快照。
- 加密磁盘的快照数据以加密方式存放，非加密磁盘的快照数据以非加密方式存放。

使用场景

快照功能可以帮助您实现以下需求：

- **日常备份数据**

  通过对云硬盘定期创建快照，实现数据的日常备份，可以应对由于误操作、病毒以及黑客攻击等导致数据丢失或不一致的情况。

- **快速恢复数据**

  更换操作系统、应用软件升级或业务数据迁移等重大操作前，您可以创建一份或多份快照，一旦升级或迁移过程中出现问题，可以通过快照及时将业务恢复到快照创建点的数据状态。

  例如，当由于云服务器 A的系统盘 A发生故障而无法正常开机时，由于系统盘 A已经故障，因此也无法将快照数据回滚至系统盘A。此时您可以使用系统盘 A已有的快照新创建一块云硬盘 B并挂载至正常运行的云服务器 B上，从而云服务器 B能够通过云硬盘 B读取原系统盘 A的数据。

  说明

  当前CCE提供的快照能力与K8s社区CSI快照功能一致：只支持基于快照创建新云硬盘，不支持将快照回滚到源云硬盘。

- **快速部署多个业务**

  通过同一个快照可以快速创建出多个具有相同数据的云硬盘，从而可以同时为多种业务提供数据资源。例如数据挖掘、报表查询和开发测试等业务。这种方式既保护了原始数据，又能通过快照创建的新云硬盘快速部署其他业务，满足企业对业务数据的多元化需求。

创建快照

使用控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“容器存储”，在右侧选择“快照与备份”页签。

步骤3 单击右上角“创建快照”，在弹出的窗口中设置相关参数。

- 快照名称：填写快照的名称。
- 选择存储：选择要创建快照的PVC，仅能选择云硬盘类型PVC。

步骤4 单击“创建”。

----结束

使用YAML创建

```
kind: VolumeSnapshot
apiVersion: snapshot.storage.k8s.io/v1beta1
metadata:
  finalizers:
    - snapshot.storage.kubernetes.io/volumesnapshot-as-source-protection
    - snapshot.storage.kubernetes.io/volumesnapshot-bound-protection
  name: cce-disksnap-test   # 快照名称
  namespace: default
spec:
  source:
    persistentVolumeClaimName: pvc-evs-test     # PVC的名称，仅能选择云硬盘类型PVC
  volumeSnapshotClassName: csi-disk-snapclass
```

使用快照创建 PVC

通过快照创建云硬盘PVC时，磁盘类型、磁盘模式、加密属性需和快照源云硬盘保持一致。

使用控制台创建

步骤1 登录CCE控制台。

步骤2 单击集群名称进入集群，在左侧选择“容器存储”，在右侧选择“快照与备份”页签。

步骤3 找到需要创建PVC的快照，单击“创建存储卷声明”，并在弹出窗口中设置PVC参数。

- PVC名称：请输入PVC名称。

步骤4 单击“创建”。

----结束

使用YAML创建

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: pvc-test
  namespace: default
  annotations:
    everest.io/disk-volume-type: SSD     # 云硬盘类型，需要与快照源云硬盘保持一致
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为云硬盘所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为云硬盘所在的可用区
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
  dataSource:
    name: cce-disksnap-test             # 快照的名称
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
```

12.4 文件存储（SFS）

12.4.1 文件存储概述

文件存储介绍

CCE支持将弹性文件存储（SFS）创建的存储卷挂载到容器的某一路径下，以满足数据持久化需求，SFS存储卷适用于多读多写的持久化存储，适用大容量扩展以及成本敏感型的业务场景，包括媒体处理、内容管理、大数据分析和分析工作负载程序等。SFS容量型文件系统不适合海量小文件业务，推荐使用SFS Turbo文件系统。

SFS为用户提供一个完全托管的共享文件存储，能够弹性伸缩至PB规模，具备高可用性和持久性，为海量数据、高带宽型应用提供有力支持。

- **符合标准文件协议：**用户可以将文件系统挂载给服务器，像使用本地文件目录一样。
- **数据共享：**多台服务器可挂载相同的文件系统，数据可以共享操作和访问。
- **私有网络：**数据访问必须在数据中心内部网络中。
- **容量与性能：**单文件系统容量较高（PB级），性能极佳（IO读写时延ms级）。
- **应用场景：**适用于多读多写（ReadWriteMany）场景下的各种工作负载（Deployment/StatefulSet）和普通任务（Job）使用，主要面向高性能计算、媒体处理、内容管理和Web服务、大数据和分析应用程序等场景。

使用场景

根据使用场景不同，文件存储支持以下挂载方式：

- **通过静态存储卷使用已有文件存储**：即静态创建的方式，需要先使用已有的文件存储创建PV，然后通过PVC在工作负载中挂载存储。适用于已有可用的底层存储的场景。
- **通过动态存储卷使用文件存储**：即动态创建的方式，无需预先创建文件存储，在创建PVC时通过指定存储类（StorageClass），即可自动创建文件存储和对应的PV对象。适用于无可用的底层存储，需要新创建的场景。

12.4.2 通过静态存储卷使用已有文件存储

文件存储（SFS）是一种可共享访问，并提供按需扩展的高性能文件系统（NAS），适用大容量扩展以及成本敏感型的业务场景。本文介绍如何使用已有的文件存储静态创建PV和PVC，并在工作负载中实现数据持久化与共享性。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。
- 您已经创建好一个文件存储，并且文件存储与集群在同一个VPC内。

约束与限制

- 支持多个PV挂载同一个SFS或SFS Turbo，但有如下限制：
  - 多个不同的PVC/PV使用同一个底层SFS或SFS Turbo卷时，如果挂载至同一Pod使用，会因为PV的volumeHandle参数值相同导致无法为Pod挂载所有PVC，出现Pod无法启动的问题，请避免该使用场景。
  - PV中persistentVolumeReclaimPolicy参数需设置为Retain，否则可能存在一个PV删除时，级联删除底层卷，其他关联这个底层卷的PV会由于底层存储被删除导致使用出现异常。
  - 重复用底层存储时，建议在应用层做好多读多写的隔离保护，防止产生的数据覆盖和丢失。

通过 kubectl 命令行使用已有文件存储

步骤1 使用kubectl连接集群。

步骤2 创建PV。

1. 创建pv-sfs.yaml文件。

   SFS容量型：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
    everest.io/reclaim-policy: retain-volume-only      # 可选字段，删除PV，保留底层存储卷
  name: pv-sfs    # PV的名称
spec:
  accessModes:
  - ReadWriteMany      # 访问模式，文件存储必须为ReadWriteMany
  capacity:
    storage: 1Gi       # 文件存储容量大小
  csi:
    driver: nas.csi.everest.io    # 挂载依赖的存储驱动
    fsType: nfs
    volumeHandle: <your_volume_id>   # SFS容量型文件存储的ID
    volumeAttributes:
      everest.io/share-export-location: <your_location>  #文件存储的共享路径
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
  persistentVolumeReclaimPolicy: Retain    # 回收策略
  storageClassName: csi-nas              # 存储类名称：csi-nas表示使用SFS容量型
  mountOptions: []                       # 挂载参数
```

表 12-15 关键参数说明

参数	是否必选	描述
everest.io/reclaim-policy: retain-volume-only	否	可选字段 目前仅支持配置“retain-volume-only” everest插件版本需 >= 1.2.9且回收策略为Delete时生效。如果回收策略是Delete且当前值设置为“retain-volume-only”删除PVC回收逻辑为：删除PV，保留底层存储卷。
volumeHandle	是	– 使用SFS容量型文件存储：填写文件存储的ID。 获取方法：在CCE控制台，单击顶部的“服务列表 > 存储 > 弹性文件服务”，并选择SFS容量型。在列表中单击对应的弹性文件存储名称，在详情页中复制“ID”后的内容即可。
everest.io/share-export-location	是	文件存储的共享路径。 – SFS容量型获取方法：在CCE控制台，单击顶部的“服务列表 > 存储 > 弹性文件服务”，在弹性文件服务列表中可以看到“挂载地址”列，即为文件存储的共享路径。
mountOptions	是	挂载参数。 不设置时默认配置为如下配置，具体说明请参见**设置文件存储挂载参数**。 mountOptions: - vers=3 - timeo=600 - nolock - hard
persistentVolumeReclaimPolicy	是	集群版本号>=1.19.10且everest插件版本>=1.2.9时正式开放回收策略支持。 支持Delete、Retain回收策略，详情请参见**PV回收策略**。多个PV使用同一个文件存储时建议使用**Retain**，避免级联删除底层卷。 **Delete**: – Delete且不设置everest.io/reclaim-policy：删除PVC，PV资源与文件存储均被删除。 – Delete且设置everest.io/reclaim-policy=retain-volume-only：删除PVC，PV资源被删除，文件存储资源会保留。 **Retain**：删除PVC，PV资源与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，不能直接再次被PVC绑定使用。

参数	是否必选	描述
storage	是	PVC申请容量，单位为Gi。 对文件存储来说，此处仅为校验需要（不能为空和0），设置的大小不起作用，此处设定为固定值1Gi。

2. 执行以下命令，创建PV。
```
kubectl apply -f pv-sfs.yaml
```

步骤3 创建PVC。

1. 创建pvc-sfs.yaml文件。
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-sfs
  namespace: default
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany                 # 文件存储必须为ReadWriteMany
  resources:
    requests:
      storage: 1Gi                # 文件存储大小
  storageClassName: csi-nas       # 存储类名称，必须与PV的存储类一致。
  volumeName: pv-sfs              # PV的名称
```

表 12-16 关键参数说明

参数	是否必选	描述
storage	是	PVC申请容量，单位为Gi。 必须和已有PV的storage大小保持一致。
volumeName	是	PV的名称，必须与1中PV的名称一致。

2. 执行以下命令，创建PVC。
```
kubectl apply -f pvc-sfs.yaml
```

步骤4 创建应用。

1. 创建web-demo.yaml文件，本示例中将文件存储挂载至/data路径。
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-demo
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-demo
  template:
    metadata:
      labels:
        app: web-demo
    spec:
```

```
containers:
- name: container-1
  image: nginx:latest
  volumeMounts:
  - name: pvc-sfs-volume    #卷名称，需与volumes字段中的卷名称对应
    mountPath: /data  #存储卷挂载的位置
imagePullSecrets:
  - name: default-secret
volumes:
  - name: pvc-sfs-volume    #卷名称，可自定义
    persistentVolumeClaim:
      claimName: pvc-sfs    #已创建的PVC名称
```

2. 执行以下命令，创建一个挂载文件存储的应用。

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

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化及共享性**中的步骤进行验证。

----结束

验证数据持久化及共享性

步骤1 查看部署的应用及文件。

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下：

```
web-demo-846b489584-mjhm9   1/1     Running   0          46s
web-demo-846b489584-wvv5s   1/1     Running   0          46s
```

2. 依次执行以下命令，查看Pod的/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
kubectl exec web-demo-846b489584-wvv5s -- ls /data
```

两个Pod均无返回结果，说明/data路径下无文件。

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-demo-846b489584-mjhm9 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
```

预期输出如下：

```
static
```

步骤4 验证数据持久化

1. 执行以下命令，删除名称为web-demo-846b489584-mjhm9的Pod。

```
kubectl delete pod web-demo-846b489584-mjhm9
```

预期输出如下：

```
pod "web-demo-846b489584-mjhm9" deleted
```

删除后，Deployment控制器会自动重新创建一个副本。

2. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下，web-demo-846b489584-d4d4j为新建的Pod：

```
web-demo-846b489584-d4d4j   1/1     Running   0          110s
web-demo-846b489584-wvv5s   1/1     Running   0          7m50s
```

3. 执行以下命令，验证新建的Pod中/data路径下的文件是否更改。

```
kubectl exec web-demo-846b489584-d4d4j -- ls /data
```

预期输出如下：

```
static
```

static文件仍然存在，则说明数据可持久化保存。

步骤5 验证数据共享性

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep web-demo
   ```
   预期输出如下：
   ```
   web-demo-846b489584-d4d4j   1/1     Running   0          7m
   web-demo-846b489584-wvv5s   1/1     Running   0          13m
   ```
2. 执行以下命令，在任意一个Pod的/data路径下创建share文件。本例中选择名为web-demo-846b489584-d4d4j的Pod。
   ```
   kubectl exec web-demo-846b489584-d4d4j --  touch /data/share
   ```
   并查看该Pod中/data路径下的文件。
   ```
   kubectl exec web-demo-846b489584-d4d4j -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
3. 由于写入share文件的操作未在名为web-demo-846b489584-wvv5s的Pod中执行，在该Pod中查看/data路径下是否存在文件以验证数据共享性。
   ```
   kubectl exec web-demo-846b489584-wvv5s -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
   如果在任意一个Pod中的/data路径下创建文件，其他Pod下的/data路径下均存在此文件，则说明两个Pod共享一个存储卷。

----结束

相关操作

您还可以执行表12-17中的操作。

表 12-17 其他操作

操作	说明	操作步骤
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.4.3 通过动态存储卷使用文件存储

本文介绍如何通过存储类动态创建PV和PVC，并在工作负载中实现数据持久化与共享性。

使用 kubectl 自动创建文件存储

步骤1 使用kubectl连接集群。

步骤2 使用StorageClass动态创建PVC及PV。

1. 创建pvc-sfs-auto.yaml文件。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-sfs-auto
  namespace: default
  annotations:
    everest.io/crypt-key-id: <your_key_id>      # 可选字段，密钥的id，使用该密钥加密文件存储
    everest.io/crypt-alias: sfs/default         # 可选字段，密钥的名称，创建加密卷时必须提供该字段
    everest.io/crypt-domain-id: <your_domain_id>  # 可选字段，指定加密卷所属租户的ID，创建加密卷时必须提供该字段
spec:
  accessModes:
    - ReadWriteMany              # 文件存储必须为ReadWriteMany
  resources:
    requests:
      storage: 1Gi               # 文件存储大小
  storageClassName: csi-nas      # StorageClass类型为文件存储
```

表 12-18 关键参数说明

参数	是否必选	描述
storage	是	PVC申请容量，单位为Gi。 对文件存储来说，此处仅为校验需要（不能为空和0），设置的大小不起作用，此处设定为固定值1Gi。
everest.io/crypt-key-id	否	当文件存储是加密卷时为必填，填写创建文件存储时选择的加密密钥ID，可使用自定义密钥或名为“sfs/default”的云硬盘默认密钥。 **获取方法**：在数据加密控制台，找到需要加密的密钥，复制密钥ID值即可。
everest.io/crypt-alias	否	密钥的名称，创建加密卷时必须提供该字段。 **获取方法**：在数据加密控制台，找到需要加密的密钥，复制密钥名称即可。
everest.io/crypt-domain-id	否	指定加密卷所属租户的ID，创建加密卷时必须提供该字段。 **获取方法**：在云服务器控制台，鼠标悬浮至右上角的用户名称并单击“我的凭证”，复制账号ID即可。

2. 执行以下命令，创建PVC。
```
kubectl apply -f pvc-sfs-auto.yaml
```

步骤3 创建应用。

1. 创建web-demo.yaml文件，本示例中将文件存储挂载至/data路径。
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-demo
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-demo
  template:
    metadata:
      labels:
        app: web-demo
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: pvc-sfs-volume    #卷名称，需与volumes字段中的卷名称对应
          mountPath: /data  #存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: pvc-sfs-volume    #卷名称，可自定义
          persistentVolumeClaim:
            claimName: pvc-sfs-auto    #已创建的PVC名称
```
2. 执行以下命令，创建一个挂载文件存储的应用。
```
kubectl apply -f web-demo.yaml
```
工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化及共享性**中的步骤进行验证。

----结束

验证数据持久化及共享性

步骤1 查看部署的应用及文件。

1. 执行以下命令，查看已创建的Pod。
```
kubectl get pod | grep web-demo
```
预期输出如下：
```
web-demo-846b489584-mjhm9   1/1     Running   0          46s
web-demo-846b489584-wvv5s   1/1     Running   0          46s
```
2. 依次执行以下命令，查看Pod的/data路径下的文件。
```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
kubectl exec web-demo-846b489584-wvv5s -- ls /data
```
两个Pod均无返回结果，说明/data路径下无文件。

步骤2 执行以下命令，在/data路径下创建static文件。
```
kubectl exec web-demo-846b489584-mjhm9 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。
```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
```
预期输出如下：
```
static
```

步骤4 验证数据持久化

1. 执行以下命令，删除名称为web-demo-846b489584-mjhm9的Pod。
   ```
   kubectl delete pod web-demo-846b489584-mjhm9
   ```
   预期输出如下：
   ```
   pod "web-demo-846b489584-mjhm9" deleted
   ```
   删除后，Deployment控制器会自动重新创建一个副本。
2. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep web-demo
   ```
   预期输出如下，web-demo-846b489584-d4d4j为新建的Pod：
   ```
   web-demo-846b489584-d4d4j   1/1     Running   0          110s
   web-demo-846b489584-wvv5s   1/1     Running   0          7m50s
   ```
3. 执行以下命令，验证新建的Pod中/data路径下的文件是否更改。
   ```
   kubectl exec web-demo-846b489584-d4d4j -- ls /data
   ```
   预期输出如下：
   ```
   static
   ```
   static文件仍然存在，则说明数据可持久化保存。

步骤5 验证数据共享性

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep web-demo
   ```
   预期输出如下：
   ```
   web-demo-846b489584-d4d4j   1/1     Running   0          7m
   web-demo-846b489584-wvv5s   1/1     Running   0          13m
   ```
2. 执行以下命令，在任意一个Pod的/data路径下创建share文件。本例中选择名为web-demo-846b489584-d4d4j的Pod。
   ```
   kubectl exec web-demo-846b489584-d4d4j --  touch /data/share
   ```
   并查看该Pod中/data路径下的文件。
   ```
   kubectl exec web-demo-846b489584-d4d4j -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
3. 由于写入share文件的操作未在名为web-demo-846b489584-wvv5s的Pod中执行，在该Pod中查看/data路径下是否存在文件以验证数据共享性。
   ```
   kubectl exec web-demo-846b489584-wvv5s -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
   如果在任意一个Pod中的/data路径下创建文件，其他Pod下的/data路径下均存在此文件，则说明两个Pod共享一个存储卷。

----结束

相关操作

您还可以执行表12-19中的操作。

表 12-19 其他操作

操作	说明	操作步骤
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.4.4 设置文件存储挂载参数

本章节主要介绍如何设置文件存储的挂载参数。您可以在PV中设置挂载参数，然后通过PVC绑定PV，也可以在StorageClass中设置挂载参数，然后使用StorageClass创建PVC，动态创建出的PV会默认带有StorageClass中设置的挂载参数。

前提条件

everest插件版本要求**1.2.8及以上**版本。插件主要负责将挂载参数识别并传递给底层存储，指定参数有否有效依赖于底层存储是否支持。

文件存储挂载参数

CCE的存储插件everest在挂载文件存储时默认设置了如**表12-20**所示的参数。

表 12-20 文件存储挂载参数

参数	参数值	描述
keep-original-ownership	无需填写	表示是否保留文件挂载点的ownership，使用该参数时，要求everest插件版本为1.2.63或2.1.2以上。 • 默认为不添加该参数，此时挂载文件存储时将会默认把挂载点的ownership修改为root:root。 • 如添加该参数，挂载文件存储时将保持文件系统原有的ownership。
vers	3	文件系统版本，目前只支持NFSv3。取值：3
nolock	无需填写	选择是否使用NLM协议在服务器上锁文件。当选择nolock选项时，锁对于同一主机的应用有效，对不同主机不受锁的影响。
timeo	600	NFS客户端重传请求前的等待时间(单位为0.1秒)。建议值：600。

参数	参数值	描述
hard/soft	无需填写	挂载方式类型。 ● 取值为hard，即使用硬连接方式，若NFS请求超时，则客户端一直重新请求直至成功。 ● 取值为soft，即软挂载方式挂载系统，若NFS请求超时，则客户端向调用程序返回错误。 默认为hard。

在 PV 中设置挂载参数

在PV中设置挂载参数可以通过mountOptions字段实现，如下所示，mountOptions支持挂载的字段请参见**文件存储挂载参数**。

步骤1 使用kubectl连接集群，详情请参见**通过kubectl连接集群**。

步骤2 在PV中设置挂载参数，示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
    everest.io/reclaim-policy: retain-volume-only      # 可选字段，删除PV，保留底层存储卷
  name: pv-sfs
spec:
  accessModes:
  - ReadWriteMany      # 访问模式，文件存储必须为ReadWriteMany
  capacity:
    storage: 1Gi       # 文件存储容量大小
  csi:
    driver: nas.csi.everest.io    # 挂载依赖的存储驱动
    fsType: nfs
    volumeHandle: <your_volume_id>  # SFS容量型文件存储的ID
    volumeAttributes:
      everest.io/share-export-location: <your_location> # 文件存储的共享路径
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
  persistentVolumeReclaimPolicy: Retain    # 回收策略
  storageClassName: csi-nas              # 存储类名称
  mountOptions:                          # 挂载参数
  - vers=3
  - nolock
  - timeo=600
  - hard
```

步骤3 PV创建后，可以创建PVC关联PV，然后在工作负载的容器中挂载，具体操作步骤请参见**通过静态存储卷使用已有文件存储**。

步骤4 验证挂载参数是否生效。

本例中将PVC挂载至使用nginx:latest镜像的工作负载，并通过**mount -l**命令查看挂载参数是否生效。

1. 查看已挂载文件存储的Pod，本文中的示例工作负载名称为web-sfs。

   ```
   kubectl get pod | grep web-sfs
   ```

   回显如下：

   ```
   web-sfs-***  1/1     Running   0          23m
   ```

2. 执行以下命令查看挂载参数，其中web-sfs-***为示例Pod。

```
kubectl exec -it web-sfs-*** -- mount -l | grep nfs
```

若回显中的挂载信息与设置的挂载参数一致，说明挂载参数设置成功。

```
<您的共享路径> on /data type nfs
(rw,relatime,vers=3,rsize=1048576,wsize=1048576,namlen=255,hard,nolock,noresvport,proto=tcp,
timeo=600,retrans=2,sec=sys,mountaddr=**.**.**.**,mountvers=3,mountport=2050,mountproto=tcp
,local_lock=all,addr=**.**.**.**)
```

----结束

在 StorageClass 中设置挂载参数

在StorageClass中设置挂载参数同样可以通过mountOptions字段实现，如下所示，mountOptions支持挂载的字段请参见**文件存储挂载参数**。

步骤1 使用kubectl连接集群，详情请参见**通过kubectl连接集群**。

步骤2 创建自定义的StorageClass，示例如下：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-sfs-mount-option
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: nas.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/share-access-to: <your_vpc_id>   # 集群所在VPC的ID
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:                          # 挂载参数
- vers=3
- nolock
- timeo=600
- hard
```

步骤3 StorageClass设置好后，就可以使用这个StorageClass创建PVC，动态创建出的PV会默认带有StorageClass中设置的挂载参数，具体操作步骤请参见**通过动态存储卷使用文件存储**。

步骤4 验证挂载参数是否生效。

本例中将PVC挂载至使用nginx:latest镜像的工作负载，并通过**mount -l**命令查看挂载参数是否生效。

1. 查看已挂载文件存储的Pod，本文中的示例工作负载名称为web-sfs。

```
kubectl get pod | grep web-sfs
```

回显如下：

```
web-sfs-***   1/1     Running   0          23m
```

2. 执行以下命令查看挂载参数，其中web-sfs-***为示例Pod。

```
kubectl exec -it web-sfs-*** -- mount -l | grep nfs
```

若回显中的挂载信息与设置的挂载参数一致，说明挂载参数设置成功。

```
<您的共享路径> on /data type nfs
(rw,relatime,vers=3,rsize=1048576,wsize=1048576,namlen=255,hard,nolock,noresvport,proto=tcp,
timeo=600,retrans=2,sec=sys,mountaddr=**.**.**.**,mountvers=3,mountport=2050,mountproto=tcp
,local_lock=all,addr=**.**.**.**)
```

----结束

12.5 极速文件存储（SFS Turbo）

12.5.1 极速文件存储概述

极速文件存储介绍

CCE支持将极速文件存储（SFS Turbo）创建的存储卷挂载到容器的某一路径下，以满足数据持久化的需求。极速文件存储具有按需申请，快速供给，弹性扩展，方便灵活等特点，适用于海量小文件业务，例如DevOps、容器微服务、企业办公等应用场景。

SFS Turbo为用户提供一个完全托管的共享文件存储，能够弹性伸缩至320TB规模，具备高可用性和持久性，为海量的小文件、低延迟高IOPS型应用提供有力支持。

- **符合标准文件协议：**用户可以将文件系统挂载给服务器，像使用本地文件目录一样。
- **数据共享：**多台服务器可挂载相同的文件系统，数据可以共享操作和访问。
- **私有网络：**数据访问必须在数据中心内部网络中。
- **安全隔离：**直接使用云上现有IaaS服务构建独享的云文件存储，为租户提供数据隔离保护和IOPS性能保障。
- **应用场景：**适用于多读多写（ReadWriteMany）场景下的各种工作负载（Deployment/StatefulSet）、守护进程集（DaemonSet）和普通任务（Job）使用，主要面向高性能网站、日志存储、DevOps、企业办公等场景。

使用场景

极速文件存储支持以下挂载方式：

- **通过静态存储卷使用已有极速文件存储**：即静态创建的方式，需要先使用已有的文件存储创建PV，然后通过PVC在工作负载中挂载存储。
- **SFS Turbo动态创建子目录并挂载**：SFS Turbo支持动态创建子目录并挂载到容器，实现共享使用SFS Turbo，从而更加经济合理的利用SFS Turbo存储容量。

12.5.2 通过静态存储卷使用已有极速文件存储

极速文件存储（SFS Turbo）是一种具备高可用性和持久性的共享文件系统，适合海量的小文件、低延迟高IOPS的应用。本文介绍如何使用已有的极速文件存储静态创建PV和PVC，并在工作负载中实现数据持久化与共享性。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。
- 您已经创建好一个状态可用的SFS Turbo，并且SFS Turbo与集群在同一个VPC内。

约束与限制

- 支持多个PV挂载同一个SFS或SFS Turbo，但有如下限制：
  - 多个不同的PVC/PV使用同一个底层SFS或SFS Turbo卷时，如果挂载至同一Pod使用，会因为PV的volumeHandle参数值相同导致无法为Pod挂载所有PVC，出现Pod无法启动的问题，请避免该使用场景。

- PV中persistentVolumeReclaimPolicy参数需设置为Retain，否则可能存在一个PV删除时，级联删除底层卷，其他关联这个底层卷的PV会由于底层存储被删除导致使用出现异常。
- 重复用底层存储时，建议在应用层做好多读多写的隔离保护，防止产生的数据覆盖和丢失。

通过控制台使用已有极速文件存储

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 静态创建存储卷声明和存储卷。

1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击右上角“创建存储卷声明”，在弹出的窗口中填写存储卷声明参数。

参数	描述
存储卷声明类型	本文中选择“极速文件存储”。
PVC名称	输入PVC的名称，同一命名空间下的PVC名称需唯一。
创建方式	根据是否已经创建PV可选择“新建存储卷”或“已有存储卷”来静态创建PVC。 本文示例中选择“新建存储卷”，可通过控制台同时创建PV及PVC。
关联存储卷[a]	选择集群中已有的PV卷，需要提前创建PV，请参考**相关操作**中的“创建存储卷”操作。 本文示例中无需选择。
极速文件存储[b]	单击“选择极速文件存储”，您可以在新页面中勾选满足要求的极速文件存储，并单击“确定”。
PV名称[b]	输入PV名称，同一集群内的PV名称需唯一。
访问模式[b]	极速文件存储类型的存储卷仅支持ReadWriteMany，表示存储卷可以被多个节点以读写方式挂载，详情请参见**存储卷访问模式**。
回收策略[b]	仅支持Retain，表示删除PVC时PV不会被同时删除，详情请参见**PV回收策略**。
挂载参数[b]	输入挂载参数键值对，详情请参见**设置极速文件存储挂载参数**。

说明

a：创建方式选择“已有存储卷”时可设置。

b：创建方式选择“新建存储卷”时可设置。

2. 单击“创建”，将同时为您创建存储卷声明和存储卷。

   您可以在左侧导航栏中选择“容器存储”，在“存储卷声明”和“存储卷”页签下查看已经创建的存储卷声明和存储卷。

步骤3 创建应用。

1. 在左侧导航栏中选择“工作负载”，在右侧选择“无状态负载”页签。
2. 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 已有存储卷声明 (PVC)”。

   本文主要为您介绍存储卷的挂载使用，如**表12-21**，其他参数详情请参见**工作负载**。

   表 12-21 存储卷挂载

参数	参数说明
存储卷声明（PVC）	选择已有的极速文件存储卷。
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“**/var/run**”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	- 只读：只能读容器路径中的数据卷。 - 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

   本例中将该存储卷挂载到容器中/data路径下，在该路径下生成的容器数据会存储到极速文件存储中。

3. 其余信息都配置完成后，单击“创建工作负载”。

   工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化及共享性**中的步骤进行验证。

----结束

通过 kubectl 命令行使用已有文件存储

步骤1 使用kubectl连接集群。

步骤2 创建PV。

1. 创建pv-sfsturbo.yaml文件。

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
  name: pv-sfsturbo    # PV的名称
spec:
  accessModes:
  - ReadWriteMany      # 访问模式，极速文件存储必须为ReadWriteMany
  capacity:
    storage: 500Gi     # 极速文件存储容量大小
  csi:
    driver: sfsturbo.csi.everest.io    # 挂载依赖的存储驱动
    fsType: nfs
    volumeHandle: <your_volume_id>   # 极速文件存储的ID
    volumeAttributes:
      everest.io/share-export-location: <your_location>   # 极速文件存储的共享路径
      everest.io/enterprise-project-id: <your_project_id>  # 极速文件存储的项目ID
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
  persistentVolumeReclaimPolicy: Retain    # 回收策略
  storageClassName: csi-sfsturbo          # SFS Turbo存储类名称
  mountOptions: []                        # 挂载参数
```

表 12-22 关键参数说明

参数	是否必选	描述
volumeHandle	是	填写极速文件存储的ID。 获取方法：在CCE控制台，单击顶部的“服务列表 > 存储 > 弹性文件服务”，并选择SFS Turbo。在列表中单击对应的SFS Turbo文件存储名称，在详情页中复制“ID”后的内容即可。
everest.io/share-export-location	是	极速文件存储的共享路径。 获取方法：在CCE控制台，单击顶部的“服务列表 > 存储 > 弹性文件服务”，选择SFS Turbo，在弹性文件服务列表中可以看到“挂载地址”列，即为文件存储的共享路径。
everest.io/enterprise-project-id	否	极速文件存储的项目ID。 获取方法：在弹性文件服务控制台，单击左侧栏目树中的“SFS Turbo”，单击要对接的SFS Turbo名称进入详情页，在“基本信息”页签下找到企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可。
mountOptions	否	挂载参数。 不设置时默认配置为如下配置，具体说明请参见**设置极速文件存储挂载参数**。 mountOptions: - vers=3 - timeo=600 - nolock - hard

参数	是否必选	描述
persistentVolumeReclaimPolicy	是	集群版本号>=1.19.10且everest插件版本>=1.2.9时正式开放回收策略支持。 仅支持Retain回收策略，详情请参见**PV回收策略**。 **Retain**：删除PVC，PV资源与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，不能直接再次被PVC绑定使用。
storage	是	PVC申请容量，单位为Gi。
storageClassName	是	极速文件存储对应的存储类名称为csi-sfsturbo。

2. 执行以下命令，创建PV。
```
kubectl apply -f pv-sfsturbo.yaml
```

步骤3 创建PVC。

1. 创建pvc-sfsturbo.yaml文件。
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-sfsturbo
  namespace: default
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/enterprise-project-id: <your_project_id>  # 极速文件存储的项目ID
spec:
  accessModes:
  - ReadWriteMany                 # 极速文件存储必须为ReadWriteMany
  resources:
    requests:
      storage: 500Gi              # 极速文件存储大小
  storageClassName: csi-sfsturbo      # SFS Turbo存储类名称，必须与PV的存储类一致
  volumeName: pv-sfsturbo     # PV的名称
```

表 12-23 关键参数说明

参数	是否必选	描述
everest.io/enterprise-project-id	否	极速文件存储的项目ID。 获取方法：在弹性文件服务控制台，单击左侧栏目树中的“SFS Turbo”，单击要对接的SFS Turbo名称进入详情页，在“基本信息”页签下找到企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可。
storage	是	PVC申请容量，单位为Gi。 必须和已有PV的storage大小保持一致。

参数	是否必选	描述
storageClassName	是	存储类名称，必须与1中PV的存储类一致。 极速文件存储对应的存储类名称为csi-sfsturbo。
volumeName	是	PV的名称，必须与1中PV名称一致。

2. 执行以下命令，创建PVC。
```
kubectl apply -f pvc-sfsturbo.yaml
```

步骤4 创建应用。

1. 创建web-demo.yaml文件，本示例中将极速文件存储挂载至/data路径。
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-demo
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-demo
  template:
    metadata:
      labels:
        app: web-demo
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: pvc-sfsturbo-volume    #卷名称，需与volumes字段中的卷名称对应
          mountPath: /data  #存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: pvc-sfsturbo-volume    #卷名称，可自定义
          persistentVolumeClaim:
            claimName: pvc-sfsturbo    #已创建的PVC名称
```
2. 执行以下命令，创建一个挂载极速文件存储的应用。
```
kubectl apply -f web-demo.yaml
```
工作负载创建成功后，您可以尝试**验证数据持久化及共享性**。

----结束

验证数据持久化及共享性

步骤1 查看部署的应用及文件。

1. 执行以下命令，查看已创建的Pod。
```
kubectl get pod | grep web-demo
```
预期输出如下：
```
web-demo-846b489584-mjhm9   1/1     Running   0          46s
web-demo-846b489584-wvv5s   1/1     Running   0          46s
```
2. 依次执行以下命令，查看Pod的/data路径下的文件。
```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
kubectl exec web-demo-846b489584-wvv5s -- ls /data
```
两个Pod均无返回结果，说明/data路径下无文件。

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-demo-846b489584-mjhm9 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
```

预期输出如下：

```
static
```

步骤4 验证数据持久化

1. 执行以下命令，删除名称为web-demo-846b489584-mjhm9的Pod。
   ```
   kubectl delete pod web-demo-846b489584-mjhm9
   ```
   预期输出如下：
   ```
   pod "web-demo-846b489584-mjhm9" deleted
   ```
   删除后，Deployment控制器会自动重新创建一个副本。
2. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep web-demo
   ```
   预期输出如下，web-demo-846b489584-d4d4j为新建的Pod：
   ```
   web-demo-846b489584-d4d4j   1/1     Running   0          110s
   web-demo-846b489584-wvv5s   1/1     Running   0          7m50s
   ```
3. 执行以下命令，验证新建的Pod中/data路径下的文件是否更改。
   ```
   kubectl exec web-demo-846b489584-d4d4j -- ls /data
   ```
   预期输出如下：
   ```
   static
   ```
   static文件仍然存在，则说明数据可持久化保存。

步骤5 验证数据共享性

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep web-demo
   ```
   预期输出如下：
   ```
   web-demo-846b489584-d4d4j   1/1     Running   0          7m
   web-demo-846b489584-wvv5s   1/1     Running   0          13m
   ```
2. 执行以下命令，在任意一个Pod的/data路径下创建share文件。本例中选择名为web-demo-846b489584-d4d4j的Pod。
   ```
   kubectl exec web-demo-846b489584-d4d4j --  touch /data/share
   ```
   并查看该Pod中/data路径下的文件。
   ```
   kubectl exec web-demo-846b489584-d4d4j -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
3. 由于写入share文件的操作未在名为web-demo-846b489584-wvv5s的Pod中执行，在该Pod中查看/data路径下是否存在文件以验证数据共享性。
   ```
   kubectl exec web-demo-846b489584-wvv5s -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
   如果在任意一个Pod中的/data路径下创建文件，其他Pod下的/data路径下均存在此文件，则说明两个Pod共享一个存储卷。

----结束

相关操作

您还可以执行表12-24中的基本操作。

表 12-24 其他操作

操作	说明	操作步骤
创建存储卷	通过CCE控制台单独创建PV。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷”页签。单击右上角“创建存储卷”，在弹出的窗口中填写存储卷声明参数。 • 存储卷类型：选择“极速文件存储”。 • 极速文件存储：单击“选择极速文件存储”，在新页面中勾选满足要求的极速文件存储，并单击“确定”。 • PV名称：输入PV名称，同一集群内的PV名称需唯一。 • 访问模式：仅支持ReadWriteMany，表示存储卷可以被多个节点以读写方式挂载，详情请参见**存储卷访问模式**。 • 回收策略：仅支持Retain，详情请参见**PV回收策略**。 • 挂载参数：输入挂载参数键值对，详情请参见**设置极速文件存储挂载参数**。 2. 单击“创建”。
扩容极速文件存储存储卷	通过CCE控制台快速扩容已挂载的极速文件存储。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 扩容”。 2. 输入新增容量，并单击“确定”。
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.5.3 设置极速文件存储挂载参数

本章节主要介绍如何设置极速文件存储的挂载参数。极速文件存储仅支持在PV中设置挂载参数，然后通过创建PVC绑定PV。

前提条件

everest插件版本要求**1.2.8及以上**版本。插件主要负责将挂载参数识别并传递给底层存储，指定参数有否有效依赖于底层存储是否支持。

极速文件存储挂载参数

CCE的存储插件everest在挂载极速文件存储时默认设置了如**表12-25**所示的参数。

表 12-25 极速文件存储挂载参数

参数	参数值	描述
vers	3	文件系统版本，目前只支持NFSv3。取值：3
nolock	无需填写	选择是否使用NLM协议在服务器上锁文件。当选择nolock选项时，锁对于同一主机的应用有效，对不同主机不受锁的影响。
timeo	600	NFS客户端重传请求前的等待时间(单位为0.1秒)。建议值：600。
hard/soft	无需填写	挂载方式类型。 • 取值为hard，即使用硬连接方式，若NFS请求超时，则客户端一直重新请求直至成功。 • 取值为soft，即软挂载方式挂载系统，若NFS请求超时，则客户端向调用程序返回错误。 默认为hard。

在 PV 中设置挂载参数

在PV中设置挂载参数可以通过mountOptions字段实现，如下所示，mountOptions支持挂载的字段请参见**极速文件存储挂载参数**。

步骤1 使用kubectl连接集群，详情请参见**通过kubectl连接集群**。

步骤2 在PV中设置挂载参数，示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
  name: pv-sfsturbo    # PV的名称
spec:
  accessModes:
  - ReadWriteMany      # 访问模式，极速文件存储必须为ReadWriteMany
  capacity:
    storage: 500Gi       # 极速文件存储容量大小
  csi:
    driver: sfsturbo.csi.everest.io    # 挂载依赖的存储驱动
    fsType: nfs
    volumeHandle: {your_volume_id}  # 极速文件存储的ID
    volumeAttributes:
      everest.io/share-export-location: {your_location}  # 极速文件存储的共享路径
      everest.io/enterprise-project-id: {your_project_id} # 极速文件存储的项目ID
```

```
    storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
persistentVolumeReclaimPolicy: Retain     # 回收策略
storageClassName: csi-sfsturbo            # SFS Turbo存储类名称
mountOptions:                             # 挂载参数
- vers=3
- nolock
- timeo=600
- hard
```

步骤3 PV创建后，可以创建PVC关联PV，然后在工作负载的容器中挂载，具体操作步骤请参见**通过静态存储卷使用已有极速文件存储**。

步骤4 验证挂载参数是否生效。

本例中将PVC挂载至使用nginx:latest镜像的工作负载，并通过**mount -l**命令查看挂载参数是否生效。

1. 查看已挂载文件存储的Pod，本文中的示例工作负载名称为web-sfsturbo。
   ```
   kubectl get pod | grep web-sfsturbo
   ```
   回显如下：
   ```
   web-sfsturbo-***  1/1     Running   0          23m
   ```
2. 执行以下命令查看挂载参数，其中web-sfsturbo-***为示例Pod。
   ```
   kubectl exec -it web-sfsturbo-*** -- mount -l | grep nfs
   ```
   若回显中的挂载信息与设置的挂载参数一致，说明挂载参数设置成功。
   ```
   {您的挂载地址} on /data type nfs
   (rw,relatime,vers=3,rsize=1048576,wsize=1048576,namlen=255,hard,nolock,noresvport,proto=tcp,
   timeo=600,retrans=2,sec=sys,mountaddr=**.**.**.**,mountvers=3,mountport=20048,mountproto=tc
   p,local_lock=all,addr=**.**.**.**)
   ```

----结束

12.5.4 SFS Turbo 动态创建子目录并挂载

背景信息

SFS Turbo容量最小500G，且不是按使用量计费。SFS Turbo挂载时默认将根目录挂载到容器，而通常情况下负载不需要这么大容量，造成浪费。

everest插件支持一种在SFS Turbo下动态创建子目录的方法，能够在SFS Turbo下动态创建子目录并挂载到容器，这种方法能够共享使用SFS Turbo，从而更加经济合理的利用SFS Turbo存储容量。

约束与限制

- 仅支持1.15+集群。
- 集群必须使用everest插件，插件版本要求1.1.13+。
- 使用everest 1.2.69之前或2.1.11之前的版本时，使用子目录功能时不能同时并发创建超过10个PVC。推荐使用everest 1.2.69及以上或2.1.11及以上的版本。
- Ubuntu操作系统的节点使用Docker容器引擎时不支持该功能。

创建 subpath 类型 SFS Turbo 存储卷

⚠ 注意

subpath模式的卷请勿通过前端进行“扩容”、“解关联”、“删除”等操作。

步骤1 创建SFS Turbo资源，选择网络时，请选择与集群相同的VPC与子网。

步骤2 新建一个StorageClass的YAML文件，例如sfsturbo-subpath-sc.yaml。

配置示例：

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  name: sfsturbo-subpath-sc
mountOptions:
- lock
parameters:
  csi.storage.k8s.io/csi-driver-name: sfsturbo.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/archive-on-delete: "true"
  everest.io/share-access-to: 7ca2dba2-1234-1234-1234-626371a8fb3a
  everest.io/share-expand-type: bandwidth
  everest.io/share-export-location: 192.168.1.1:/sfsturbo/
  everest.io/share-source: sfs-turbo
  everest.io/share-volume-type: STANDARD
  everest.io/volume-as: subpath
  everest.io/volume-id: 0d773f2e-1234-1234-1234-de6a35074696
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

其中：

- name：storageclass的名称。
- mountOptions：选填字段；mount挂载参数。
  - everest 1.2.8以下，1.1.13以上版本仅开放对nolock参数配置，mount操作默认使用nolock参数，无需配置。nolock=false时，使用lock参数。
  - everest 1.2.8及以上版本支持更多参数，默认使用如下所示配置。**此处不能配置为nolock=true，会导致挂载失败。**

    ```
    mountOptions:
    - vers=3
    - timeo=600
    - nolock
    - hard
    ```
- everest.io/volume-as：该参数需设置为“subpath”来使用subpath模式。
- everest.io/share-access-to：选填字段。subpath模式下，填写SFS Turbo资源的所在VPC的ID。
- everest.io/share-expand-type：选填字段。若SFS Turbo资源存储类型为增强版（标准型增强版、性能型增强版），设置为bandwidth。
- everest.io/share-export-location：挂载目录配置。由SFS Turbo共享路径和子目录组成，共享路径可至SFS Turbo服务页面查询，子路径由用户自定义，后续指定该StorageClass创建的PVC均位于该子目录下。
- everest.io/share-volume-type：选填字段。填写SFS Turbo的类型。标准型为STANDARD，性能型为PERFORMANCE。对于增强型需配合“everest.io/share-expand-type”字段使用，everest.io/share-expand-type设置为“bandwidth”。
- everest.io/zone：选填字段。指定SFS Turbo资源所在的可用区。
- everest.io/volume-id：SFS Turbo资源的卷ID，可至SFS Turbo界面查询。
- everest.io/archive-on-delete：若该参数设置为“true”，在回收策略为“Delete”时，删除PVC会将PV的原文档进行归档，归档目录的命名规则“archived-$pv名称.时间戳”。该参数设置为“false”时，会将PV对应的SFS Turbo子目录删除。默认设置为“true”，即删除PVC时进行归档。

步骤3 执行**kubectl create -f sfsturbo-subpath-sc.yaml**。

步骤4 新建一个PVC的YAML文件，sfs-turbo-test.yaml。

配置示例：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sfs-turbo-test
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
  storageClassName: sfsturbo-subpath-sc
  volumeMode: Filesystem
```

其中：

- name: PVC的名称。
- storageClassName：SC的名称。
- storage：subpath模式下，该参数无实际意义，容量受限于SFS Turbo资源的总容量，若SFS Turbo资源总容量不足，请及时到SFS Turbo界面扩容。

步骤5 执行**kubectl create -f sfs-turbo-test.yaml**。

----结束

说明

对subpath类型的SFS Turbo扩容时，没有实际的扩容意义。该操作不会对SFS Turbo资源进行实际的扩容，需要用户自行保证SFS Turbo的总容量不被耗尽。

创建 Deployment 挂载已有数据卷

步骤1 新建一个Deployment的YAML文件，例如deployment-test.yaml。

配置示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-turbo-subpath-example
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-turbo-subpath-example
  template:
    metadata:
      labels:
        app: test-turbo-subpath-example
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        volumeMounts:
        - mountPath: /tmp
```

```
      name: pvc-sfs-turbo-example
  restartPolicy: Always
  imagePullSecrets:
  - name: default-secret
  volumes:
  - name: pvc-sfs-turbo-example
    persistentVolumeClaim:
      claimName: sfs-turbo-test
```

其中：

- name：创建的工作负载名称。
- image：工作负载的镜像。
- mountPath：容器内挂载路径，示例中挂载到“/tmp”路径。
- claimName：已有的PVC名称。

步骤2 创建Deployment负载。

kubectl create -f deployment-test.yaml

----结束

StatefulSet 动态创建 subpath 模式的数据卷

步骤1 新建一个StatefulSet的YAML文件，例如statefulset-test.yaml。

配置示例：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: test-turbo-subpath
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test-turbo-subpath
  template:
    metadata:
      labels:
        app: test-turbo-subpath
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
        pod.alpha.kubernetes.io/initialized: 'true'
    spec:
      containers:
        - name: container-0
          image: 'nginx:latest'
          resources: {}
          volumeMounts:
            - name: sfs-turbo-160024548582479676
              mountPath: /tmp
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: default-secret
```

```
    affinity: {}
    schedulerName: default-scheduler
volumeClaimTemplates:
  - metadata:
      name: sfs-turbo-160024548582479676
      namespace: default
      annotations: {}
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
      storageClassName: sfsturbo-subpath-sc
serviceName: wwww
podManagementPolicy: OrderedReady
updateStrategy:
  type: RollingUpdate
revisionHistoryLimit: 10
```

其中：

- name：创建的工作负载名称。
- image：工作负载的镜像。
- mountPath：容器内挂载路径，示例中挂载到“/tmp”路径。
- “spec.template.spec.containers.volumeMounts.name ”和“spec.volumeClaimTemplates.metadata.name”有映射关系，必须保持一致。
- storageClassName: 填写自建的SC名称。

步骤2 创建StatefulSet负载。

kubectl create -f statefulset-test.yaml

----结束

12.6 对象存储（OBS）

12.6.1 对象存储概述

对象存储介绍

对象存储服务（Object Storage Service，OBS）提供海量、安全、高可靠、低成本的数据存储能力，可供用户存储任意类型和大小的数据。适合企业备份/归档、视频点播、视频监控等多种数据存储场景。

- **标准接口：**具备标准Http Restful API接口，用户必须通过编程或第三方工具访问对象存储。
- **数据共享：**服务器、嵌入式设备、IOT设备等所有调用相同路径，均可访问共享的对象存储数据。
- **公共/私有网络：**对象存储数据允许在公网访问，满足互联网应用需求。
- **容量与性能：**容量无限制，性能较高（IO读写时延10ms级）。
- **应用场景：**适用于（基于OBS界面、OBS工具、OBS SDK等）的一次上传共享多读（ReadOnlyMany）的各种工作负载（Deployment/StatefulSet）和普通任务（Job）使用，主要面向大数据分析、静态网站托管、在线视频点播、基因测序、智能视频监控、备份归档、企业云盘（网盘）等场景。

对象存储规格

对象存储提供了多种存储类别，从而满足客户业务对存储性能、成本的不同诉求。

- 并行文件系统（**推荐使用**）：并行文件系统（Parallel File System）是OBS提供的一种经过优化的高性能文件系统，提供毫秒级别访问时延，以及TB/s级别带宽和百万级别的IOPS，能够快速处理高性能计算（HPC）工作负载。相较于对象桶，并行文件系统在稳定性、性能上更具优势。
- 对象桶（**不推荐使用**）：
  - 标准存储：访问时延低和吞吐量高，因而适用于有大量热点文件（平均一个月多次）或小文件（小于1MB），且需要频繁访问数据的业务场景，例如：大数据、移动应用、热点视频、社交图片等场景。
  - 低频访问存储：适用于不频繁访问（平均一年少于12次）但在需要时也要求快速访问数据的业务场景，例如：文件同步/共享、企业备份等场景。与标准存储相比，低频访问存储有相同的数据持久性、吞吐量以及访问时延，且成本较低，但是可用性略低于标准存储。

使用场景

根据使用场景不同，对象存储支持以下挂载方式：

- **通过静态存储卷使用已有对象存储**：即静态创建的方式，需要先使用已有的对象存储创建PV，然后通过PVC在工作负载中挂载存储。适用于已有可用的底层存储的场景。
- **通过动态存储卷使用对象存储**：即动态创建的方式，无需预先创建对象存储，在创建PVC时通过指定存储类（StorageClass），即可自动创建对象存储和对应的PV对象。适用于无可用的底层存储，需要新创建的场景。

12.6.2 通过静态存储卷使用已有对象存储

本文介绍如何使用已有的对象存储静态创建PV和PVC，并在工作负载中实现数据持久化与共享性。

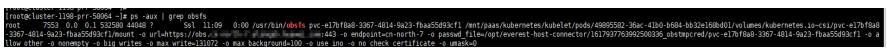
前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。
- 您已经创建好一个对象存储，仅支持选择与集群同一区域的并行文件系统类型的对象存储。

约束与限制

- 使用并行文件系统和对象桶时，挂载点不支持修改属组和权限。
- CCE支持通过OBS SDK方式和PVC挂载方式使用OBS并行文件系统，其中PVC挂载方式是通过OBS服务提供的**obsfs工具**实现。每挂载一个并行文件系统对象存储卷，就会产生一个obsfs常驻进程。如下图所示：

图 12-3 obsfs 常驻进程



建议为每个obsfs进程预留1G的内存空间，例如4U8G的节点，则建议挂载obsfs并行文件系统的实例**不超过**8个。

说明

obsfs常驻进程是直接运行在节点上，如果消耗的内存超过了节点上限，则会导致节点异常。例如在4U8G的节点上，运行的挂载并行文件系统卷的实例超过100+，有极大概率会导致节点异常不可用。因此强烈建议控制单个节点上的挂载并行文件系统实例的数量。

- 支持多个PV挂载同一个对象存储，但有如下限制：
  - 多个不同的PVC/PV使用同一个底层对象存储卷时，如果挂载至同一Pod使用，会因为PV的volumeHandle参数值相同导致无法挂载，请避免该使用场景。
  - PV中persistentVolumeReclaimPolicy参数需设置为Retain，否则可能存在一个PV删除时，级联删除底层卷，其他关联这个底层卷的PV会由于底层存储被删除导致使用出现异常。
  - 重复用底层存储时，建议在应用层做好多读多写的隔离保护，防止产生的数据覆盖和丢失。

通过控制台使用已有对象存储

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 静态创建存储卷声明和存储卷。

1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击右上角“创建存储卷声明”，在弹出的窗口中填写存储卷声明参数。

参数	描述
存储卷声明类型	本文中选择“对象存储”。
PVC名称	输入PVC的名称，同一命名空间下的PVC名称需唯一。
创建方式	- 已有底层存储的场景下，根据是否已经创建PV可选择“新建存储卷”或“已有存储卷”来静态创建PVC。 - 无可用底层存储的场景下，可选择“动态创建”，具体操作请参见**通过动态存储卷使用对象存储**。 本文示例中选择“新建存储卷”，可通过控制台同时创建PV及PVC。
关联存储卷[a]	选择集群中已有的PV卷，需要提前创建PV，请参考**相关操作**中的“创建存储卷”操作。 本文示例中无需选择。
对象存储[b]	单击“选择对象存储”，您可以在新页面中勾选满足要求的对象存储，并单击“确定”。 **说明** 当前仅支持选择并行文件系统。
PV名称[b]	输入PV名称，同一集群内的PV名称需唯一。
访问模式[b]	对象存储类型的存储卷仅支持ReadWriteMany，表示存储卷可以被多个节点以读写方式挂载，详情请参见**存储卷访问模式**。

参数	描述
回收策略[b]	您可以选择Delete或Retain，用于指定删除PVC时底层存储的回收策略，详情请参见**PV回收策略**。 **说明** 多个PV使用同一个对象存储时建议使用**Retain**，避免级联删除底层卷。
密钥[b]	自定义密钥：如果您需要为不同OBS存储分配不同的用户权限时，可通过选择不同的Secret实现更灵活的权限控制（推荐使用）。具体使用请参见**对象存储卷挂载设置自定义访问密钥（AK/SK）**。 仅支持选择带有 secret.kubernetes.io/used-by = csi 标签的密钥，密钥类型为cfe/secure-opaque。如果无可用密钥，可单击“创建密钥”进行创建： – 名称：请输入密钥名称。 – 命名空间：密钥所在的命名空间。 – 访问密钥（AK/SK）：上传.csv格式的密钥文件，详情请参见**获取访问密钥**。
挂载参数[b]	输入挂载参数键值对，详情请参见**设置对象存储挂载参数**。

说明

a：创建方式选择“已有存储卷”时可设置。

b：创建方式选择“新建存储卷”时可设置。

2. 单击“创建”，将同时为您创建存储卷声明和存储卷。

   您可以在左侧导航栏中选择“容器存储”，在“存储卷声明”和“存储卷”页签下查看已经创建的存储卷声明和存储卷。

步骤3 创建应用。

1. 在左侧导航栏中选择“工作负载”，在右侧选择“无状态负载”页签。
2. 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 已有存储卷声明 (PVC)”。

   本文主要为您介绍存储卷的挂载使用，如**表12-26**，其他参数详情请参见**工作负载**。

表 12-26 存储卷挂载

参数	参数说明
存储卷声明（PVC）	选择已有的对象存储卷。

参数	参数说明
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	- 只读：只能读容器路径中的数据卷。 - 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

本例中将该存储卷挂载到容器中/data路径下，在该路径下生成的容器数据会存储到对象存储中。

3. 其余信息都配置完成后，单击“创建工作负载”。

   工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化及共享性**中的步骤进行验证。

----结束

通过 kubectl 命令行使用已有对象存储

步骤1 使用kubectl连接集群。

步骤2 创建PV。

1. 创建pv-obs.yaml文件。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
    everest.io/reclaim-policy: retain-volume-only      # 可选字段，删除PV，保留底层存储卷
  name: pv-obs       # PV的名称
spec:
  accessModes:
  - ReadWriteMany    # 访问模式，对象存储必须为ReadWriteMany
  capacity:
    storage: 1Gi     # 对象存储容量大小
  csi:
    driver: obs.csi.everest.io        # 挂载依赖的存储驱动
    fsType: obsfs                     # 实例类型
    volumeHandle: <your_volume_id>    # 对象存储的名称
    volumeAttributes:
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
      everest.io/obs-volume-type: STANDARD
      everest.io/region: <your_region>                    # 对象存储的区域
```

```
    everest.io/enterprise-project-id: <your_project_id>     # 可选字段，企业项目ID，如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
  nodePublishSecretRef:           # 设置对象存储的自定义密钥
    name: <your_secret_name>       # 自定义密钥的名称
    namespace: <your_namespace>    # 自定义密钥的命名空间
persistentVolumeReclaimPolicy: Retain    # 回收策略
storageClassName: csi-obs              # 存储类名称
mountOptions: []                       # 挂载参数
```

表 12-27 关键参数说明

参数	是否必填	描述
everest.io/reclaim-policy: retain-volume-only	否	可选字段 目前仅支持配置“retain-volume-only” everest插件版本需 >= 1.2.9且回收策略为Delete时生效。如果回收策略是Delete且当前值设置为“retain-volume-only”删除PVC回收逻辑为：删除PV，保留底层存储卷。
fsType	是	实例类型，支持“obsfs”与“s3fs”。 – obsfs：并行文件系统，配套使用obsfs挂载，推荐使用。 – s3fs：对象桶，配套使用s3fs挂载。
volumeHandle	是	对象存储的名称。
everest.io/obs-volume-type	是	对象存储类型。 – fsType设置为s3fs时，支持STANDARD（标准桶）、WARM（低频访问桶）。 – fsType设置为obsfs时，该字段不起作用。
everest.io/region	是	OBS存储区域。 Region对应的值请参见**地区和终端节点**。
everest.io/enterprise-project-id	否	可选字段 对象存储的企业项目ID。如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。 **获取方法**：在对象存储服务控制台，单击左侧栏目树中的“桶列表”或“并行文件系统”，单击要对接的对象存储名称进入详情页，在“概览 > 基本信息”页签下找到企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可获取对象存储所属的企业项目的ID。

参数	是否必填	描述
nodePublishSecretRef	否	对象存储卷挂载支持设置自定义访问密钥（AK/SK），您可以使用AK/SK创建一个Secret，然后挂载到PV。详细说明请参见**对象存储卷挂载设置自定义访问密钥（AK/SK）**。 示例如下： nodePublishSecretRef: name: secret-demo namespace: default
mountOptions	否	挂载参数，具体请参见**设置对象存储挂载参数**。
persistentVolumeReclaimPolicy	是	集群版本号>=1.19.10且everest插件版本>=1.2.9时正式开放回收策略支持。 支持Delete、Retain回收策略，详情请参见**PV回收策略**。多个PV使用同一个对象存储时建议使用**Retain**，避免级联删除底层卷。 **Delete**: – Delete且不设置everest.io/reclaim-policy：删除PVC，PV资源与存储均被删除。 – Delete且设置everest.io/reclaim-policy=retain-volume-only：删除PVC，PV资源被删除，存储资源会保留。 **Retain**：删除PVC，PV资源与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，不能直接再次被PVC绑定使用。
storage	是	存储容量，单位为Gi。 对对象存储来说，此处仅为校验需要（不能为空和0），设置的大小不起作用，此处设定为固定值1Gi。
storageClassName	是	对象存储对应的存储类名称为csi-obs。

2. 执行以下命令，创建PV。
```
kubectl apply -f pv-obs.yaml
```

步骤3 创建PVC。

1. 创建pvc-obs.yaml文件。
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-obs
  namespace: default
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/obs-volume-type: STANDARD
    csi.storage.k8s.io/fstype: obsfs
    csi.storage.k8s.io/node-publish-secret-name: <your_secret_name>  # 自定义密钥的名称
    csi.storage.k8s.io/node-publish-secret-namespace: <your_namespace>       # 自定义密钥的命名空间
    everest.io/enterprise-project-id: <your_project_id>    # 可选字段，企业项目ID，如果指定企业项目，
```

```
则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
spec:
  accessModes:
  - ReadWriteMany                # 对象存储必须为ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs      # 存储类名称，必须与PV的存储类一致。
  volumeName: pv-obs    # PV的名称
```

表 12-28 关键参数说明

参数	是否必填	描述
csi.storage.k8s.io/node-publish-secret-name	否	PV中指定的自定义密钥的名称。
csi.storage.k8s.io/node-publish-secret-namespace	否	PV中指定的自定义密钥的命名空间。
everest.io/enterprise-project-id	否	对象存储的项目ID。 获取方法：在对象存储服务控制台，单击左侧栏目树中的“桶列表”或“并行文件系统”，单击要对接的对象存储名称进入详情页，在“概览 > 基本信息”页签下找到企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可获取对象存储所属的企业项目的ID。
storage	是	PVC申请容量，单位为Gi。 对于对象存储来说，此处仅为校验需要（不能为空和0），设置的大小不起作用，此处设定为固定值1Gi。
storageClassName	是	存储类名称，必须与1中PV的存储类一致。 对象存储对应的存储类名称为csi-obs。
volumeName	是	PV的名称，必须与1中PV名称一致。

2. 执行以下命令，创建PVC。
```
kubectl apply -f pvc-obs.yaml
```

步骤4 创建应用。

1. 创建web-demo.yaml文件，本示例中将对象存储挂载至/data路径。
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-demo
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-demo
  template:
    metadata:
      labels:
```

```
    app: web-demo
spec:
  containers:
  - name: container-1
    image: nginx:latest
    volumeMounts:
    - name: pvc-obs-volume    #卷名称，需与volumes字段中的卷名称对应
      mountPath: /data  #存储卷挂载的位置
  imagePullSecrets:
    - name: default-secret
  volumes:
    - name: pvc-obs-volume    #卷名称，可自定义
      persistentVolumeClaim:
        claimName: pvc-obs    #已创建的PVC名称
```

2. 执行以下命令，创建一个挂载对象存储的应用。

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

工作负载创建成功后，您可以尝试**验证数据持久化及共享性**。

----结束

验证数据持久化及共享性

步骤1 查看部署的应用及文件。

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下：

```
web-demo-846b489584-mjhm9   1/1     Running   0          46s
web-demo-846b489584-wvv5s   1/1     Running   0          46s
```

2. 依次执行以下命令，查看Pod的/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
kubectl exec web-demo-846b489584-wvv5s -- ls /data
```

两个Pod均无返回结果，说明/data路径下无文件。

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-demo-846b489584-mjhm9 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
```

预期输出如下：

```
static
```

步骤4 验证数据持久化

1. 执行以下命令，删除名称为web-demo-846b489584-mjhm9的Pod。

```
kubectl delete pod web-demo-846b489584-mjhm9
```

预期输出如下：

```
pod "web-demo-846b489584-mjhm9" deleted
```

删除后，Deployment控制器会自动重新创建一个副本。

2. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下，web-demo-846b489584-d4d4j为新建的Pod：

```
web-demo-846b489584-d4d4j   1/1     Running   0          110s
web-demo-846b489584-wvv5s   1/1     Running   0          7m50s
```

3. 执行以下命令，验证新建的Pod中/data路径下的文件是否更改。

```
kubectl exec web-demo-846b489584-d4d4j -- ls /data
```

预期输出如下：

```
static
```

static文件仍然存在，则说明数据可持久化保存。

步骤5 验证数据共享性

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep web-demo
   ```
   预期输出如下：
   ```
   web-demo-846b489584-d4d4j   1/1     Running   0          7m
   web-demo-846b489584-wvv5s   1/1     Running   0          13m
   ```
2. 执行以下命令，在任意一个Pod的/data路径下创建share文件。本例中选择名为web-demo-846b489584-d4d4j的Pod。
   ```
   kubectl exec web-demo-846b489584-d4d4j --  touch /data/share
   ```
   并查看该Pod中/data路径下的文件。
   ```
   kubectl exec web-demo-846b489584-d4d4j -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
3. 由于写入share文件的操作未在名为web-demo-846b489584-wvv5s的Pod中执行，在该Pod中查看/data路径下是否存在文件以验证数据共享性。
   ```
   kubectl exec web-demo-846b489584-wvv5s -- ls /data
   ```
   预期输出如下：
   ```
   share
   static
   ```
   如果在任意一个Pod中的/data路径下创建文件，其他Pod下的/data路径下均存在此文件，则说明两个Pod共享一个存储卷。

----结束

相关操作

您还可以执行**表12-29**中的操作。

表 12-29 其他操作

操作	说明	操作步骤
创建存储卷	通过CCE控制台单独创建PV。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷”页签。单击右上角“创建存储卷”，在弹出的窗口中填写存储卷声明参数。 • 存储卷类型：选择“对象存储”。 • 对象存储：单击“选择对象存储”，在新页面中勾选满足要求的对象存储，并单击“确定”。 • PV名称：输入PV名称，同一集群内的PV名称需唯一。 • 访问模式：仅支持ReadWriteMany，表示存储卷可以被多个节点以读写方式挂载，详情请参见**存储卷访问模式**。 • 回收策略：Delete或Retain，详情请参见**PV回收策略**。 **说明** 多个PV使用同一个底层存储时建议使用**Retain**，避免级联删除底层卷。 • 自定义密钥：如果您需要为不同OBS存储分配不同的用户权限时，可通过选择不同的Secret实现更灵活的权限控制（推荐使用）。具体使用请参见**对象存储卷挂载设置自定义访问密钥（AK/SK）**。 仅支持选择带有 secret.kubernetes.io/used-by = csi 标签的密钥，密钥类型为 cfe/secure-opaque。如果无可用密钥，可单击“创建密钥”进行创建。 • 挂载参数：输入挂载参数键值对，详情请参见**设置对象存储挂载参数**。 2. 单击“创建”。
更新访问密钥	通过CCE控制台更新对象存储的访问密钥。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 更新访问密钥”。 2. 上传.csv格式的密钥文件，详情请参见**获取访问密钥**。单击“确定”。 **说明** 更新全局访问密钥后，租户下所有挂载使用全局访问密钥的对象存储的负载实例需要重启后才能正常访问。
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。

操作	说明	操作步骤
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

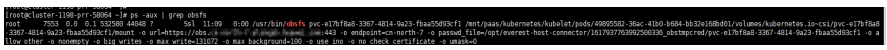
12.6.3 通过动态存储卷使用对象存储

本文介绍如何自动创建对象存储，适用于无可用的底层存储卷，需要新创建的场景。

约束与限制

- 使用并行文件系统和对象桶时，挂载点不支持修改属组和权限。
- CCE支持通过OBS SDK方式和PVC挂载方式使用OBS并行文件系统，其中PVC挂载方式是通过OBS服务提供的**obsfs工具**实现。每挂载一个并行文件系统对象存储卷，就会产生一个obsfs常驻进程。如下图所示：

  图 12-4 obsfs 常驻进程

  建议为每个obsfs进程预留1G的内存空间，例如4U8G的节点，则建议挂载obsfs并行文件系统的实例**不超过**8个。

  说明

  obsfs常驻进程是直接运行在节点上，如果消耗的内存超过了节点上限，则会导致节点异常。例如在4U8G的节点上，运行的挂载并行文件系统卷的实例超过100+，有极大概率会导致节点异常不可用。因此强烈建议控制单个节点上的挂载并行文件系统实例的数量。
- OBS限制单用户创建100个桶，当动态创建的PVC数量较多时，容易导致桶数量超过限制，OBS桶无法创建。此种场景下建议直接调用OBS的API或SDK使用OBS，不在工作负载中挂载OBS桶。

通过控制台自动创建对象存储

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 动态创建存储卷声明和存储卷。

1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击右上角“创建存储卷声明”，在弹出的窗口中填写存储卷声明参数。

参数	描述
存储卷声明类型	本文中选择“对象存储”。
PVC名称	输入PVC的名称，同一命名空间下的PVC名称需唯一。

参数	描述
创建方式	– 无可用底层存储的场景下，可选择“动态创建”，通过控制台级联创建存储卷声明PVC、存储卷PV和底层存储。 – 已有底层存储的场景下，根据是否已经创建PV可选择“新建存储卷”或“已有存储卷”，静态创建PVC，具体操作请参见**通过静态存储卷使用已有对象存储**。 本文中选择“动态创建”。
存储类	对象存储对应的存储类为csi-obs。
实例类型	– 并行文件系统：一种对象存储服务提供的高性能文件系统，提供毫秒级别访问时延，以及TB/s级别带宽和百万级别的IOPS。**推荐您使用并行文件系统**。 – 对象桶：桶（Bucket）是OBS中存储对象的容器，桶中的所有对象都处于同一逻辑层级。
对象存储类型	选择“对象桶”时，支持选择以下类别： – 标准存储：适用于有大量热点文件或小文件，且需要频繁访问（平均一个月多次）并快速获取数据的业务场景。 – 低频访问存储：适用于不频繁访问（平均一年少于12次），但需要快速获取数据的业务场景。
访问模式	对象存储类型的存储卷仅支持ReadWriteMany，表示存储卷可以被多个节点以读写方式挂载，详情请参见**存储卷访问模式**。
密钥	自定义密钥：如果您需要为不同OBS存储分配不同的用户权限时，可通过选择不同的Secret实现更灵活的权限控制（推荐使用）。具体使用请参见**对象存储卷挂载设置自定义访问密钥（AK/SK）**。 仅支持选择带有 secret.kubernetes.io/used-by = csi 标签的密钥，密钥类型为cfe/secure-opaque。如果无可用密钥，可单击“创建密钥”进行创建： – 名称：请输入密钥名称。 – 命名空间：密钥所在的命名空间。 – 访问密钥（AK/SK）：上传.csv格式的密钥文件，详情请参见**获取访问密钥**。
企业项目	仅支持default、集群所在企业项目或存储类指定的企业项目。

2. 单击“创建”，将同时为您创建存储卷声明和存储卷。

   您可以在左侧导航栏中选择“容器存储”，在“存储卷声明”和“存储卷”页签下查看已经创建的存储卷声明和存储卷。

步骤3 创建应用。

1. 在左侧导航栏中选择“工作负载”，在右侧选择“无状态负载”页签。
2. 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 已有存储卷声明 (PVC)”。

本文主要为您介绍存储卷的挂载使用，如**表12-30**，其他参数详情请参见**工作负载**。

表 12-30 存储卷挂载

参数	参数说明
存储卷声明（PVC）	选择已有的对象存储卷。
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	- 只读：只能读容器路径中的数据卷。 - 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

本例中将该存储卷挂载到容器中/data路径下，在该路径下生成的容器数据会存储到对象存储中。

3. 其余信息都配置完成后，单击“创建工作负载”。

   工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化及共享性**中的步骤进行验证。

----结束

使用 kubectl 自动创建对象存储

步骤1 使用kubectl连接集群。

步骤2 使用StorageClass动态创建PVC及PV。

1. 创建pvc-obs-auto.yaml文件。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-obs-auto
  namespace: default
  annotations:
    everest.io/obs-volume-type: STANDARD    # 对象存储类型
    csi.storage.k8s.io/fstype: obsfs        # 实例类型
    csi.storage.k8s.io/node-publish-secret-name: <your_secret_name>       # 自定义密钥的名称
    csi.storage.k8s.io/node-publish-secret-namespace: <your_namespace>    # 自定义密钥的命名空间
```

```
  everest.io/enterprise-project-id: <your_project_id>     # 可选字段，企业项目ID，如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
spec:
  accessModes:
    - ReadWriteMany              # 对象存储必须为ReadWriteMany
  resources:
    requests:
      storage: 1Gi               # 对象存储大小
  storageClassName: csi-obs      # StorageClass类型为对象存储
```

表 12-31 关键参数说明

参数	是否必选	描述
everest.io/obs-volume-type	是	对象存储类型。 – fsType设置为s3fs时，支持STANDARD（标准桶）、WARM（低频访问桶）。 – fsType设置为obsfs时，该字段不起作用。
csi.storage.k8s.io/fstype	是	实例类型，支持“obsfs”与“s3fs”。 – obsfs：并行文件系统，配套使用obsfs挂载，推荐使用。 – s3fs：对象桶，配套使用s3fs挂载。
csi.storage.k8s.io/node-publish-secret-name	否	自定义密钥的名称。 如果您需要为不同OBS存储分配不同的用户权限时，可通过选择不同的Secret实现更灵活的权限控制（推荐使用）。具体使用请参见**对象存储卷挂载设置自定义访问密钥（AK/SK）**。
csi.storage.k8s.io/node-publish-secret-namespace	否	自定义密钥的命名空间。
everest.io/enterprise-project-id	否	对象存储的项目ID。 **获取方法**： **获取方法**：在对象存储服务控制台，单击左侧栏目树中的“桶列表”或“并行文件系统”，单击要对接的对象存储名称进入详情页，在“概览 > 基本信息”页签下找到企业项目，单击并进入对应的企业项目控制台，复制对应的ID值即可获取对象存储所属的企业项目的ID。
storage	是	PVC申请容量，单位为Gi。 对对象存储来说，此处仅为校验需要（不能为空和0），设置的大小不起作用，此处设定为固定值1Gi。
storageClassName	是	存储类名称，对象存储对应的存储类名称为csi-obs。

2. 执行以下命令，创建PVC。

```
kubectl apply -f pvc-obs-auto.yaml
```

步骤3 创建应用。

1. 创建web-demo.yaml文件，本示例中将对象存储挂载至/data路径。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-demo
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-demo
  template:
    metadata:
      labels:
        app: web-demo
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: pvc-obs-volume    #卷名称，需与volumes字段中的卷名称对应
          mountPath: /data  #存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: pvc-obs-volume    #卷名称，可自定义
          persistentVolumeClaim:
            claimName: pvc-obs-auto    #已创建的PVC名称
```

2. 执行以下命令，创建一个挂载对象存储的应用。

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

工作负载创建成功后，您可以尝试**验证数据持久化及共享性**。

----结束

验证数据持久化及共享性

步骤1 查看部署的应用及文件。

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下：

```
web-demo-846b489584-mjhm9   1/1     Running   0          46s
web-demo-846b489584-wvv5s   1/1     Running   0          46s
```

2. 依次执行以下命令，查看Pod的/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
kubectl exec web-demo-846b489584-wvv5s -- ls /data
```

两个Pod均无返回结果，说明/data路径下无文件。

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-demo-846b489584-mjhm9 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-demo-846b489584-mjhm9 -- ls /data
```

预期输出如下：

```
static
```

步骤4 验证数据持久化

1. 执行以下命令，删除名称为web-demo-846b489584-mjhm9的Pod。

```
kubectl delete pod web-demo-846b489584-mjhm9
```

预期输出如下：

```
pod "web-demo-846b489584-mjhm9" deleted
```

删除后，Deployment控制器会自动重新创建一个副本。

2. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下，web-demo-846b489584-d4d4j为新建的Pod：

```
web-demo-846b489584-d4d4j   1/1     Running   0          110s
web-demo-846b489584-wvv5s   1/1     Running   0          7m50s
```

3. 执行以下命令，验证新建的Pod中/data路径下的文件是否更改。

```
kubectl exec web-demo-846b489584-d4d4j -- ls /data
```

预期输出如下：

```
static
```

static文件仍然存在，则说明数据可持久化保存。

步骤5 验证数据共享性

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-demo
```

预期输出如下：

```
web-demo-846b489584-d4d4j   1/1     Running   0          7m
web-demo-846b489584-wvv5s   1/1     Running   0          13m
```

2. 执行以下命令，在任意一个Pod的/data路径下创建share文件。本例中选择名为web-demo-846b489584-d4d4j的Pod。

```
kubectl exec web-demo-846b489584-d4d4j --  touch /data/share
```

并查看该Pod中/data路径下的文件。

```
kubectl exec web-demo-846b489584-d4d4j -- ls /data
```

预期输出如下：

```
share
static
```

3. 由于写入share文件的操作未在名为web-demo-846b489584-wvv5s的Pod中执行，在该Pod中查看/data路径下是否存在文件以验证数据共享性。

```
kubectl exec web-demo-846b489584-wvv5s -- ls /data
```

预期输出如下：

```
share
static
```

如果在任意一个Pod中的/data路径下创建文件，其他Pod下的/data路径下均存在此文件，则说明两个Pod共享一个存储卷。

----结束

相关操作

您还可以执行**表12-32**中的操作。

表 12-32 其他操作

操作	说明	操作步骤
更新访问密钥	通过CCE控制台更新对象存储的访问密钥。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 更新访问密钥”。 2. 上传.csv格式的密钥文件，详情请参见**获取访问密钥**。单击“确定”。 **说明** 更新全局访问密钥后，租户下所有挂载使用全局访问密钥的对象存储的负载实例需要重启后才能正常访问。
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.6.4 设置对象存储挂载参数

本章节主要介绍如何设置对象存储的挂载参数。您可以在PV中设置挂载参数，然后通过PVC绑定PV，也可以在StorageClass中设置挂载参数，然后使用StorageClass创建PVC，动态创建出的PV会默认带有StorageClass中设置的挂载参数。

前提条件

everest插件版本要求**1.2.8及以上**版本。插件主要负责将挂载参数识别并传递给底层存储，指定参数有否有效依赖于底层存储是否支持。

对象存储挂载参数

CCE的存储插件everest在挂载对象存储时默认设置了**表12-33**和**表12-34**的参数，其中**表12-33**中的参数不可取消。

表 12-33 默认使用且不可取消的挂载参数

参数	参数值	描述
use_ino	无需填写	使用该选项，由obsfs分配inode编号。读写模式下自动开启。

参数	参数值	描述
big_writes	无需填写	配置后可更改写缓存最大值大小
nonempty	无需填写	允许挂载目录非空
allow_other	无需填写	允许其他用户访问并行文件系统
no_check_certificate	无需填写	不校验服务端证书
enable_noobj_cache	无需填写	为不存在的对象启用缓存条目，可提高性能。对象桶读写模式下自动使用。 **从everest 1.2.40版本开始不再默认设置enable_noobj_cache参数。**
sigv2	无需填写	签名版本。对象桶自动使用。

表 12-34 默认使用且可修改的挂载参数

参数	参数值	描述
max_write	131072	仅配置big_writes的情况下才生效，推荐使用128KB。
ssl_verify_hostname	0	不根据主机名验证SSL证书。
max_background	100	可配置后台最大等待请求数。并行文件系统自动使用。
public_bucket	1	设置为1时匿名挂载公共桶。对象桶读写模式下自动使用。
umask	无需填写	配置文件权限的掩码。

在 PV 中设置挂载参数

在PV中设置挂载参数可以通过mountOptions字段实现，如下所示，mountOptions支持挂载的字段请参见**对象存储挂载参数**。

步骤1 使用kubectl连接集群，详情请参见**通过kubectl连接集群**。

步骤2 在PV中设置挂载参数，示例如下：

```
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
    everest.io/reclaim-policy: retain-volume-only      # 可选字段，删除PV，保留底层存储卷
  name: pv-obs       # PV的名称
spec:
  accessModes:
  - ReadWriteMany    # 访问模式，对象存储必须为ReadWriteMany
  capacity:
    storage: 1Gi     # 对象存储容量大小
  csi:
    driver: obs.csi.everest.io        # 挂载依赖的存储驱动
    fsType: obsfs                     # 实例类型
    volumeHandle: <your_volume_id>    # 对象存储的名称
    volumeAttributes:
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
      everest.io/obs-volume-type: STANDARD
      everest.io/region: <your_region>                    # 对象存储的区域
      everest.io/enterprise-project-id: <your_project_id>    # 可选字段，企业项目ID，如果指定企业项目，则创建PVC时也需要指定相同的企业项目，否则PVC无法绑定PV。
    nodePublishSecretRef:           # 设置对象存储的自定义密钥
      name: <your_secret_name>       # 自定义密钥的名称
      namespace: <your_namespace>    # 自定义密钥的命名空间
  persistentVolumeReclaimPolicy: Retain    # 回收策略
  storageClassName: csi-obs              # 存储类名称
  mountOptions:                          # 挂载参数
  - umask=0027
```

步骤3 PV创建后，可以创建PVC关联PV，然后在工作负载的容器中挂载，具体操作步骤请参见**通过静态存储卷使用已有对象存储**。

步骤4 验证挂载参数是否生效。

本例中将PVC挂载至使用nginx:latest镜像的工作负载，可以登录到运行挂载对象存储卷的Pod所在节点上通过进程详情观察。

执行以下命令：

- 对象桶：ps -ef | grep s3fs

  ```
  root     22142     1  0 Jun03 ?        00:00:00 /usr/bin/s3fs {your_obs_name} /mnt/paas/kubernetes/kubelet/pods/{pod_uid}/volumes/kubernetes.io~csi/{your_pv_name}/mount -o url=https://{endpoint}:443 -o endpoint={region} -o passwd_file=/opt/everest-host-connector/***_obstmpcred/{your_obs_name} -o nonempty -o big_writes -o sigv2 -o allow_other -o no_check_certificate -o ssl_verify_hostname=0 -o umask=0027 -o max_write=131072 -o multipart_size=20
  ```

- 并行文件系统：ps -ef | grep obsfs

  ```
  root      1355     1  0 Jun03 ?        00:03:16 /usr/bin/obsfs {your_obs_name} /mnt/paas/kubernetes/kubelet/pods/{pod_uid}/volumes/kubernetes.io~csi/{your_pv_name}/mount -o url=https://{endpoint}:443 -o endpoint={region} -o passwd_file=/opt/everest-host-connector/***_obstmpcred/{your_obs_name} -o allow_other -o nonempty -o big_writes -o use_ino -o no_check_certificate -o ssl_verify_hostname=0 -o max_background=100 -o umask=0027 -o max_write=131072
  ```

----结束

在 StorageClass 中设置挂载参数

在StorageClass中设置挂载参数同样可以通过mountOptions字段实现，如下所示，mountOptions支持挂载的字段请参见**对象存储挂载参数**。

步骤1 使用kubectl连接集群，详情请参见**通过kubectl连接集群**。

步骤2 创建自定义的StorageClass，示例如下：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-obs-mount-option
provisioner: everest-csi-provisioner
```

```
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: s3fs
  everest.io/obs-volume-type: STANDARD
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:                    # 挂载参数
- umask=0027
```

步骤3 StorageClass设置好后，就可以使用这个StorageClass创建PVC，动态创建出的PV会默认带有StorageClass中设置的挂载参数，具体操作步骤请参见**通过动态存储卷使用对象存储**。

步骤4 验证挂载参数是否生效。

本例中将PVC挂载至使用nginx:latest镜像的工作负载，可以登录到运行挂载对象存储卷的Pod所在节点上通过进程详情观察。

执行以下命令：

- 对象桶：ps -ef | grep s3fs

  root 22142 1 0 Jun03 ? 00:00:00 /usr/bin/s3fs *{your_obs_name}* /mnt/paas/kubernetes/kubelet/pods/*{pod_uid}*/volumes/kubernetes.io~csi/*{your_pv_name}*/mount -o url=https://*{endpoint}*:443 -o endpoint=*{region}* -o passwd_file=/opt/everest-host-connector/***_obstmpcred/*{your_obs_name}* -o nonempty -o big_writes -o sigv2 -o allow_other -o no_check_certificate -o ssl_verify_hostname=0 -o **umask=0027** -o max_write=131072 -o multipart_size=20

- 并行文件系统：ps -ef | grep obsfs

  root 1355 1 0 Jun03 ? 00:03:16 /usr/bin/obsfs *{your_obs_name}* /mnt/paas/kubernetes/kubelet/pods/*{pod_uid}*/volumes/kubernetes.io~csi/*{your_pv_name}*/mount -o url=https://*{endpoint}*:443 -o endpoint=*{region}* -o passwd_file=/opt/everest-host-connector/***_obstmpcred/*{your_obs_name}* -o allow_other -o nonempty -o big_writes -o use_ino -o no_check_certificate -o ssl_verify_hostname=0 -o max_background=100 -o **umask=0027** -o max_write=131072

----结束

12.6.5 对象存储卷挂载设置自定义访问密钥（AK/SK）

背景信息

everest在1.2.8及以上版本提供了设置自定义访问密钥的能力，这样可以让IAM用户使用自己的访问密钥挂载对象存储卷，从而可以对OBS进行访问权限控制。

前提条件

- everest要求1.2.8及以上版本。
- 集群要求1.15.11及以上版本。

约束与限制

- 对象存储卷使用自定义访问密钥（AK/SK）时，对应的AK/SK不允许删除或禁用，否则业务容器将无法访问已挂载的对象存储。

关闭自动挂载访问密钥

老版本控制台会要求您上传AK/SK，对象存储卷挂载时默认使用您上传的访问密钥，相当于所有IAM用户（即子用户）都使用的是同一个访问密钥挂载的对象桶，对桶的权限都是一样的，导致无法对IAM用户使用对象存储桶进行权限控制。

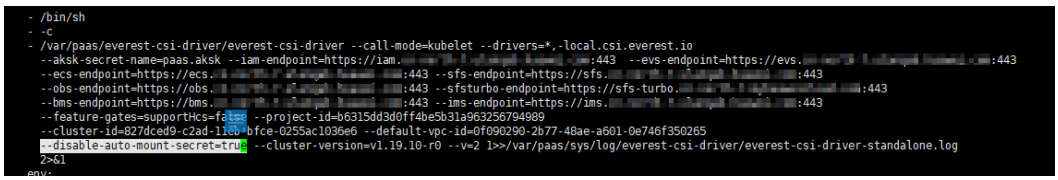
如果您之前上传过AK/SK，为防止IAM用户越权，建议关闭自动挂载访问密钥，即需要在everest插件中将**disable_auto_mount_secret**参数打开，这样使用对象存储时就不会自动使用在控制台上传的访问密钥。

说明

- 设置disable-auto-mount-secret时要求当前集群中无对象存储卷，否则挂载了该对象卷的工作负载扩容或重启的时候会由于必须指定访问密钥而导致挂卷失败。
- disable-auto-mount-secret设置为true后，则创建PV和PVC时必须指定挂载访问密钥，否则会导致对象卷挂载失败。

kubectl edit ds everest-csi-driver -nkube-system

搜索disable-auto-mount-secret，并将值设置为true。



执行 **:wq** 保存退出，等待实例重启完毕即可。

获取访问密钥

步骤1 登录控制台。

步骤2 鼠标指向界面右上角的登录用户名，在下拉列表中单击“我的凭证”。

步骤3 在左侧导航栏单击“访问密钥”。

步骤4 单击“新增访问密钥”，进入“新增访问密钥”页面。

步骤5 单击“确定”，下载访问密钥。

----结束

使用访问密钥创建 Secret

步骤1 获取访问密钥。

步骤2 对访问密钥进行base64编码（假设上文获取到的ak为“xxx”，sk为“yyy”）。

echo -n xxx|base64

echo -n yyy|base64

记录编码后的AK和SK。

步骤3 新建一个secret的yaml，如test-user.yaml。

```
apiVersion: v1
data:
  access.key: WE5WWVhVNU*****
  secret.key: Nnk4emJyZ0*****
kind: Secret
metadata:
  name: test-user
  namespace: default
  labels:
    secret.kubernetes.io/used-by: csi
type: cfe/secure-opaque
```

其中：

参数	描述
access.key	base64编码后的ak。
secret.key	base64编码后的sk。
name	secret的名称
namespace	secret的命名空间
secret.kubernetes.io/used-by: csi	带上这个标签才能在控制台上创建OBS PV/PVC时可见。
type	密钥类型，该值必须为cfe/secure-opaque 使用该类型，用户输入的数据会自动加密。

步骤4 创建Secret。

kubectl create -f test-user.yaml

----结束

静态创建对象存储卷时指定挂载 Secret

使用访问密钥创建Secret后，在创建PV时只需要关联上Secret，就可以使用Secret中的访问密钥（AK/SK）挂载对象存储卷。

步骤1 登录OBS控制台，创建对象存储桶，记录桶名称和存储类型，以并行文件系统为例。

步骤2 新建一个pv的yaml文件，如pv-example.yaml。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-obs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  csi:
    nodePublishSecretRef:
      name: test-user
      namespace: default
    driver: obs.csi.everest.io
    fsType: obsfs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region: eu-west-0
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: obs-normal-static-pv
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-obs
```

参数	描述
nodePublishSecretRef	挂载时指定的密钥，其中 ● name：指定secret的名字 ● namespace：指定secret的命令空间
fsType	文件类型，支持“obsfs”与“s3fs”，取值为s3fs时创建是obs对象桶，配套使用s3fs挂载；取值为obsfs时创建的是obs并行文件系统，配套使用obsfs挂载，推荐使用。
volumeHandle	对象存储的桶名称。

步骤3 创建PV。

kubectl create -f pv-example.yaml

PV创建完成后，就可以创建PVC关联PV。

步骤4 新建一个PVC的yaml文件，如pvc-example.yaml。

PVC yaml文件配置示例：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/node-publish-secret-name: test-user
    csi.storage.k8s.io/node-publish-secret-namespace: default
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/obs-volume-type: STANDARD
    csi.storage.k8s.io/fstype: obsfs
  name: obs-secret
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs
  volumeName: pv-obs-example
```

参数	描述
csi.storage.k8s.io/node-publish-secret-name	指定secret的名字
csi.storage.k8s.io/node-publish-secret-namespace	指定secret的命令空间

步骤5 创建PVC。

kubectl create -f pvc-example.yaml

PVC创建后，就可以创建工作负载挂载PVC使用存储。

----结束

动态创建对象存储卷时指定挂载密钥

动态创建对象存储卷时，可通过如下方法指定挂载密钥。

步骤1 新建一个pvc的yaml文件，如pvc-example.yaml。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/node-publish-secret-name: test-user
    csi.storage.k8s.io/node-publish-secret-namespace: default
    everest.io/obs-volume-type: STANDARD
    csi.storage.k8s.io/fstype: obsfs
  name: obs-secret
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs
```

参数	描述
csi.storage.k8s.io/node-publish-secret-name	指定secret的名字
csi.storage.k8s.io/node-publish-secret-namespace	指定secret的命令空间

步骤2 创建PVC。

kubectl create -f pvc-example.yaml

PVC创建后，就可以创建工作负载挂载PVC使用存储。

----结束

配置验证

根据上述步骤，使用IAM用户的密钥挂载对象存储卷。假设工作负载名称为obs-secret，容器内挂载目录是/temp，IAM用户权限为CCE ReadOnlyAccess和Tenant Guest。

1. 查询工作负载实例名称。

   kubectl get po | grep obs-secret

   期望输出：

   ```
   obs-secret-5cd558f76f-vxslv         1/1     Running   0          3m22s
   ```

2. 查询挂载目录下对象，查询正常。

   kubectl exec obs-secret-5cd558f76f-vxslv -- ls -l /temp/

3. 尝试在挂载目录内写入数据，写入失败。

   kubectl exec obs-secret-5cd558f76f-vxslv -- touch /temp/test

   期望输出：

   ```
   touch: setting times of '/temp/test': No such file or directory
   command terminated with exit code 1
   ```

4. 参考桶策略配置，给挂载桶的子用户设置读写权限。

5. 再次尝试在挂载目录内写入数据，写入成功。

   kubectl exec obs-secret-5cd558f76f-vxslv -- touch /temp/test

6. 查看容器内挂载目录，验证数据写入成功。

   kubectl exec obs-secret-5cd558f76f-vxslv -- ls -l /temp/

   期望输出：

   ```
   -rwxrwxrwx 1 root root 0 Jun  7 01:52 test
   ```

12.7 本地持久卷（Local PV）

12.7.1 本地持久卷概述

本地持久卷介绍

CCE支持使用LVM将节点上的数据卷组成存储池（VolumeGroup），然后划分LV给容器挂载使用。使用本地持久卷作为存储介质的PV的类型可称之为Local PV。

与HostPath卷相比，本地持久卷能够以持久和可移植的方式使用，而且本地持久卷的PV会存在节点亲和性配置，其挂载的Pod会自动根据该亲和性配置进行调度，无需手动将Pod调度到特定节点。

挂载方式

本地持久卷仅支持以下挂载方式：

- **通过动态存储卷使用本地持久卷**：即动态创建的方式，在创建PVC时通过指定存储类（StorageClass），即可自动创建对象存储和对应的PV对象。
- **有状态负载动态挂载本地持久卷**：仅有状态工作负载支持，可以为每一个Pod关联一个独有的PVC及PV，当Pod被重新调度后，仍然能够根据该PVC名称挂载原有的数据。适用于多实例的有状态工作负载。

说明

本地持久卷不支持通过静态PV使用，即不支持先手动创建PV然后通过PVC在工作负载中挂载的方式使用。

约束与限制

- 本地持久卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=2.1.23，推荐使用>=2.1.23版本。
- **移除节点**、**删除节点**、**重置节点**和**缩容节点**会导致与节点关联的本地持久存储卷类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。移除节点、删除节点、重置节点和缩容节点时使用了本地持久存储卷的Pod会从待删除、重置的节点上驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。节点重置完成后，Pod可能调度到重置好的节点上，此时Pod会一直处于creating状态，因为该PVC对应的底层逻辑卷已不存在。
- 请勿在节点上手动删除对应的存储池或卸载数据盘，否则会导致数据丢失等异常情况。

- 本地持久卷不支持被多个工作负载或多个任务同时挂载。

12.7.2 在存储池中导入持久卷

CCE支持使用LVM将节点上的数据卷组成存储池（VolumeGroup），然后划分LV给容器挂载使用。在创建本地持久卷前，需将节点数据盘导入存储池。

约束与限制

- 本地持久卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=2.1.23，推荐使用>=2.1.23版本。
- 节点上的第一块数据盘（供容器运行时和Kubelet组件使用）不支持导入为存储池。
- 条带化模式的存储池不支持扩容，条带化存储池扩容后可能造成碎片空间，无法使用。
- 存储池不支持缩容和删除。
- 如果删除节点上存储池的磁盘，会导致存储池异常。

导入存储池

创建节点时导入

在创建节点时，在存储配置中可以为节点添加数据盘，选择“作为持久存储卷”导入存储池，详情请参见**创建节点**。

手动导入

如果创建节点时没有导入持久存储卷，或当前存储卷容量不够，可以进行手动导入。

步骤1 前往ECS控制台为节点添加SCSI类型的磁盘。

步骤2 登录CCE控制台，单击集群名称进入集群。

步骤3 在左侧导航栏中选择“容器存储”，并切换至“存储池”页签。

步骤4 查看已添加磁盘的节点，选择“导入持久卷”，导入时可以选择写入模式。

说明

如存储池列表中未找到手动挂载的磁盘，请耐心等待1分钟后刷新列表。

- 线性：线性逻辑卷是将一个或多个物理卷整合为一个逻辑卷，实际写入数据时会先往一个基本物理卷上写入，当存储空间占满时再往另一个基本物理卷写入。
- 条带化：创建逻辑卷时指定条带化，当实际写入数据时会将连续数据分成大小相同的块，然后依次存储在多个物理卷上，实现数据的并发读写从而提高读写性能。多块卷才能选择条带化。

----结束

12.7.3 通过动态存储卷使用本地持久卷

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。

- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。
- 您已经将一块节点数据盘导入本地持久卷存储池，详情请参见**在存储池中导入持久卷**。

约束与限制

- 本地持久卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本>=2.1.23，推荐使用>=2.1.23版本。
- **移除节点**、**删除节点**、**重置节点**和**缩容节点**会导致与节点关联的本地持久存储卷类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。移除节点、删除节点、重置节点和缩容节点时使用了本地持久存储卷的Pod会从待删除、重置的节点上驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。节点重置完成后，Pod可能调度到重置好的节点上，此时Pod会一直处于creating状态，因为该PVC对应的底层逻辑卷已不存在。
- 请勿在节点上手动删除对应的存储池或卸载数据盘，否则会导致数据丢失等异常情况。
- 本地持久卷不支持被多个工作负载或多个任务同时挂载。

通过控制台自动创建本地持久卷

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 动态创建PVC及PV。

1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”页签。单击右上角“创建存储卷声明”，在弹出的窗口中填写存储卷声明参数。

参数	描述
存储卷声明类型	本文中选择“本地持久卷”。
PVC名称	输入PVC的名称，同一命名空间下的PVC名称需唯一。
创建方式	仅可选择“动态创建”，通过控制台级联创建存储卷声明PVC、存储卷PV和底层存储。
存储类	本地持久卷对应的存储类为csi-local-topology。
访问模式	本地持久卷类型的存储卷仅支持ReadWriteOnce，表示存储卷可以被一个节点以读写方式挂载，详情请参见**存储卷访问模式**。
存储池	查看已导入的存储池，如需将新的数据卷导入存储池，请参见**在存储池中导入持久卷**。
容量（GiB）	申请的存储卷容量大小。

2. 单击“创建”，将同时为您创建PVC和PV。

   您可以在左侧导航栏中选择“容器存储”，在“存储卷声明”和“存储卷”页签下查看已经创建的PVC和PV。

说明

本地存储卷存储类（名为csi-local-topology）的卷绑定模式为延迟绑定（即volumeBindingMode参数值为WaitForFirstConsumer）。该模式会延迟PV的创建和绑定，只有在创建工作负载时声明使用该PVC，对应的PV才会创建并绑定。

步骤3 创建应用。

1. 在左侧导航栏中选择“工作负载”，在右侧选择“无状态负载”页签。
2. 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 已有存储卷声明 (PVC)”。

   本文主要为您介绍存储卷的挂载使用，如**表12-35**，其他参数详情请参见**工作负载**。

   表 12-35 存储卷挂载

参数	参数说明
存储卷声明（PVC）	选择已有的本地持久卷。 本地持久卷无法被多个工作负载重复挂载。
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“**/var/run**”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	- 只读：只能读容器路径中的数据卷。 - 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

   本例中将该存储卷挂载到容器中/data路径下，在该路径下生成的容器数据会存储到本地持久存储中。

3. 其余信息都配置完成后，单击“创建工作负载”。

   工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

使用 kubectl 自动创建本地持久卷

步骤1 使用kubectl连接集群。

步骤2 使用StorageClass动态创建PVC及PV。

1. 创建pvc-local.yaml文件。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-local
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce             # 本地持久卷必须为ReadWriteOnce
  resources:
    requests:
      storage: 10Gi             # 本地持久卷大小
  storageClassName: csi-local-topology    # StorageClass类型为本地持久卷
```

表 12-36 关键参数说明

参数	是否必选	描述
storage	是	PVC申请容量，单位为Gi。
storageClassName	是	存储类名称，本地持久卷对应的存储类名称为csi-local-topology。

2. 执行以下命令，创建PVC。

```
kubectl apply -f pvc-local.yaml
```

步骤3 创建应用。

1. 创建web-local.yaml文件，本示例中将本地持久卷挂载至/data路径。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-local
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web-local
  serviceName: web-local   # Headless Service名称
  template:
    metadata:
      labels:
        app: web-local
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: pvc-disk    #卷名称，需与volumes字段中的卷名称对应
          mountPath: /data  #存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: pvc-disk    #卷名称，可自定义
          persistentVolumeClaim:
            claimName: pvc-local    #已创建的PVC名称
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: web-local   # Headless Service名称
  namespace: default
  labels:
    app: web-local
spec:
  selector:
    app: web-local
  clusterIP: None
  ports:
    - name: web-local
      targetPort: 80
      nodePort: 0
      port: 80
      protocol: TCP
  type: ClusterIP
```

2. 执行以下命令，创建一个挂载本地持久存储的应用。

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

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

验证数据持久化

步骤1 查看部署的应用及本地文件。

1. 执行以下命令，查看已创建的Pod。

```
kubectl get pod | grep web-local
```

预期输出如下：

```
web-local-0                1/1     Running   0             38s
```

2. 执行以下命令，查看本地持久卷是否挂载至/data路径。

```
kubectl exec web-local-0 -- df | grep data
```

预期输出如下：

```
/dev/mapper/vg--everest--localvolume--persistent-pvc-local          10255636     36888  10202364
0% /data
```

3. 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-local-0 -- ls /data
```

预期输出如下：

```
lost+found
```

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec web-local-0 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec web-local-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

步骤4 执行以下命令，删除名称为web-local-0的Pod。

```
kubectl delete pod web-local-0
```

预期输出如下：

```
pod "web-local-0" deleted
```

步骤5 删除后，StatefulSet控制器会自动重新创建一个同名副本。执行以下命令，验证/data路径下的文件是否更改。

```
kubectl exec web-local-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

static文件仍然存在，则说明本地持久存储中的数据可持久化保存。

----结束

相关操作

您还可以执行**表12-37**中的操作。

表 12-37 其他操作

操作	说明	操作步骤
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.7.4 有状态负载动态挂载本地持久卷

使用场景

动态挂载仅可在创建**有状态负载**时使用，通过卷声明模板（**volumeClaimTemplates**字段）实现，并依赖于StorageClass的动态创建PV能力。在多实例的有状态负载中，动态挂载可以为每一个Pod关联一个独有的PVC及PV，当Pod被重新调度后，仍然能够根据该PVC名称挂载原有的数据。而在无状态工作负载的普通挂载方式中，当存储支持多点挂载（ReadWriteMany）时，工作负载下的多个Pod会被挂载到同一个底层存储中。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。
- 您已经将一块节点数据盘导入本地持久卷存储池，详情请参见**在存储池中导入持久卷**。

通过控制台动态挂载本地持久卷

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“工作负载”，在右侧选择“有状态负载”页签。

步骤3 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 动态挂载 (VolumeClaimTemplate)”。

步骤4 单击“创建存储卷声明”，在弹出窗口中填写卷声明模板参数。

参数填写完成后，单击“创建”。

参数	描述
存储卷声明类型	本文中选择“本地持久卷”。
PVC名称	输入PVC的名称。创建后将根据实例数自动增加后缀，格式为<自定义PVC名称>-<序号>，例如example-0。
创建方式	仅可选择“动态创建”，通过控制台级联创建存储卷声明PVC、存储卷PV和底层存储。
存储类	本地持久卷对应的存储类为csi-local-topology。
访问模式	本地持久卷类型的存储卷仅支持ReadWriteOnce，表示存储卷可以被一个节点以读写方式挂载，详情请参见**存储卷访问模式**。
存储池	查看已导入的存储池，如需将新的数据卷导入存储池，请参见**在存储池中导入持久卷**。
容量（GiB）	申请的存储卷容量大小。

步骤5 填写挂载路径。

表 12-38 存储卷挂载

参数	参数说明
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“**/var/run**”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。

参数	参数说明
权限	• 只读：只能读容器路径中的数据卷。 • 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

本例中将该存储卷挂载到容器中/data路径下，在该路径下生成的容器数据会存储到本地持久卷中。

步骤6 本文主要为您介绍存储卷的动态挂载使用，其他参数详情请参见**创建有状态负载（StatefulSet）**。其余信息都配置完成后，单击“创建工作负载”。

工作负载创建成功后，容器挂载目录下的数据将会持久化保持，您可以参考**验证数据持久化**中的步骤进行验证。

----结束

通过 kubectl 命令行使用已有存储

步骤1 使用kubectl连接集群。

步骤2 创建statefulset-local.yaml文件，本示例中将本地持久卷挂载至/data路径。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: statefulset-local
  namespace: default
spec:
  selector:
    matchLabels:
      app: statefulset-local
  template:
    metadata:
      labels:
        app: statefulset-local
    spec:
      containers:
        - name: container-1
          image: nginx:latest
          volumeMounts:
            - name: pvc-local          # 需与volumeClaimTemplates字段中的名称对应
              mountPath: /data         # 存储卷挂载的位置
      imagePullSecrets:
        - name: default-secret
  serviceName: statefulset-local       # Headless Service名称
  replicas: 2
  volumeClaimTemplates:
    - apiVersion: v1
      kind: PersistentVolumeClaim
      metadata:
        name: pvc-local
        namespace: default
      spec:
        accessModes:
          - ReadWriteOnce              # 本地持久卷必须为ReadWriteOnce
        resources:
          requests:
            storage: 10Gi              # 存储卷容量大小
        storageClassName: csi-local-topology      # StorageClass类型为本地持久卷
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: statefulset-local   # Headless Service名称
  namespace: default
  labels:
    app: statefulset-local
spec:
  selector:
    app: statefulset-local
  clusterIP: None
  ports:
    - name: statefulset-local
      targetPort: 80
      nodePort: 0
      port: 80
      protocol: TCP
  type: ClusterIP
```

表 12-39 关键参数说明

参数	是否必选	描述
storage	是	PVC申请容量，单位为Gi。
storageClassName	是	本地持久卷对应的存储类名称为csi-local-topology。

步骤3 执行以下命令，创建一个挂载本地持久卷存储的应用。

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

工作负载创建成功后，您可以尝试**验证数据持久化**。

----结束

验证数据持久化

步骤1 查看部署的应用及文件。

1. 执行以下命令，查看已创建的Pod。

   ```
   kubectl get pod | grep statefulset-local
   ```

   预期输出如下：

   ```
   statefulset-local-0         1/1     Running   0          45s
   statefulset-local-1         1/1     Running   0          28s
   ```

2. 执行以下命令，查看本地持久卷是否挂载至/data路径。

   ```
   kubectl exec statefulset-local-0 -- df | grep data
   ```

   预期输出如下：

   ```
   /dev/mapper/vg--everest--localvolume--persistent-pvc-local          10255636     36888  10202364   0% /data
   ```

3. 执行以下命令，查看/data路径下的文件。

   ```
   kubectl exec statefulset-local-0 -- ls /data
   ```

   预期输出如下：

   ```
   lost+found
   ```

步骤2 执行以下命令，在/data路径下创建static文件。

```
kubectl exec statefulset-local-0 --  touch /data/static
```

步骤3 执行以下命令，查看/data路径下的文件。

```
kubectl exec statefulset-local-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

步骤4 执行以下命令，删除名称为web-local-auto-0的Pod。

```
kubectl delete pod statefulset-local-0
```

预期输出如下：

```
pod "statefulset-local-0" deleted
```

步骤5 删除后，StatefulSet控制器会自动重新创建一个同名副本。执行以下命令，验证/data路径下的文件是否更改。

```
kubectl exec statefulset-local-0 -- ls /data
```

预期输出如下：

```
lost+found
static
```

static文件仍然存在，则说明本地持久卷中的数据可持久化保存。

----结束

相关操作

您还可以执行表12-40中的操作。

表 12-40 其他操作

操作	说明	操作步骤
事件	查看PVC或PV的事件名称、事件类型、发生次数、Kubernetes事件、首次和最近发生的时间，便于定位问题。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“事件”，即可查看1小时内的事件（事件保存时间为1小时）。
查看YAML	可对PVC或PV的YAML文件进行查看、复制和下载。	1. 在左侧导航栏选择“容器存储”，在右侧选择“存储卷声明”或“存储卷”页签。 2. 单击目标实例操作列的“查看YAML”，即可查看或下载YAML。

12.8 临时存储卷（EmptyDir）

12.8.1 临时存储卷概述

临时卷介绍

当有些应用程序需要额外的存储，但并不关心数据在重启后是否仍然可用。 例如，缓存服务经常受限于内存大小，而且可以将不常用的数据转移到比内存慢的存储中，对

总体性能的影响并不大。另有些应用程序需要以文件形式注入的只读数据，比如配置数据或密钥。

Kubernetes中的**临时卷**（Ephemeral Volume），就是为此类场景设计的。临时卷会遵从Pod的生命周期，与 Pod一起创建和删除。

Kubernetes中常用的临时卷：

- **EmptyDir**：Pod启动时为空，存储空间来自本地的kubelet根目录（通常是根磁盘）或内存。EmptyDir是从**节点临时存储**中分配的，如果来自其他来源（如日志文件或镜像分层数据）的数据占满了临时存储，可能会发生存储容量不足的问题。
- **ConfigMap**：将ConfigMap类型的Kubernetes数据以数据卷的形式挂载到Pod中。
- **Secret**：将Secret类型的Kubernetes数据以数据卷的形式挂载到Pod中。

EmptyDir 的类型

CCE提供了如下两种EmptyDir类型：

- **临时路径**：Kubernetes原生的EmptyDir类型，生命周期与容器实例相同，并支持指定内存作为存储介质。容器实例消亡时，EmptyDir会被删除，数据会永久丢失。
- **本地临时卷**：本地临时存储卷将节点的本地数据盘通过LVM组成**存储池**（VolumeGroup），然后划分LV作为EmptyDir的存储介质给容器挂载使用，相比原生EmptyDir默认的存储介质类型性能更好。

约束与限制

- 本地临时卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本 >=1.2.29。
- 请勿在节点上手动删除对应的存储池或卸载数据盘，否则会导致数据丢失等异常情况。
- 请确保节点上Pod不要挂载/var/lib/kubelet/pods/目录，否则可能会导致使用了临时存储卷的Pod无法正常删除。

12.8.2 在存储池中导入临时卷

CCE支持使用LVM将节点上的数据卷组成存储池（VolumeGroup），然后划分LV给容器挂载使用。在创建本地临时卷前，需将节点数据盘导入存储池。

约束与限制

- 本地临时卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本 >=1.2.29。
- 节点上的第一块数据盘（供容器运行时和Kubelet组件使用）不支持导入为存储池。
- 条带化模式的存储池不支持扩容，条带化存储池扩容后可能造成碎片空间，无法使用。
- 存储池不支持缩容和删除。
- 如果删除节点上存储池的磁盘，会导致存储池异常。

导入存储池

创建节点时导入

在创建节点时，在存储配置中可以为节点添加数据盘，选择“作为临时存储卷”导入存储池，详情请参见**创建节点**。

手动导入

如果创建节点时没有导入临时存储卷，或当前存储卷容量不够，可以进行手动导入。

步骤1 前往ECS控制台为节点添加SCSI类型的磁盘。

步骤2 登录CCE控制台，单击集群名称进入集群。

步骤3 在左侧导航栏中选择“容器存储”，并切换至“存储池”页签。

步骤4 查看已添加磁盘的节点，选择“导入临时卷”，导入时可以选择写入模式。

说明

如存储池列表中未找到手动挂载的磁盘，请耐心等待1分钟后刷新列表。

- 线性：线性逻辑卷是将一个或多个物理卷整合为一个逻辑卷，实际写入数据时会先往一个基本物理卷上写入，当存储空间占满时再往另一个基本物理卷写入。
- 条带化：创建逻辑卷时指定条带化，当实际写入数据时会将连续数据分成大小相同的块，然后依次存储在多个物理卷上，实现数据的并发读写从而提高读写性能。多块卷才能选择条带化。

----结束

12.8.3 使用本地临时卷

本地临时卷（Local Ephemeral Volume）存储在临时卷**存储池**，相比原生EmptyDir默认的存储介质类型性能要更好，且支持扩容。

前提条件

- 您已经创建好一个集群，并且在该集群中安装CSI插件（**everest**）。
- 如果您需要通过命令行创建，需要使用kubectl连接到集群，详情请参见**通过kubectl连接集群**。
- 如需使用本地临时卷，您需要将一块节点数据盘导入本地临时卷存储池，详情请参见**在存储池中导入临时卷**。

约束与限制

- 本地临时卷仅在集群版本 >= v1.21.2-r0 时支持，且需要everest插件版本 >=1.2.29。
- 请勿在节点上手动删除对应的存储池或卸载数据盘，否则会导致数据丢失等异常情况。
- 请确保节点上Pod不要挂载/var/lib/kubelet/pods/目录，否则可能会导致使用了临时存储卷的Pod无法正常删除。

通过控制台使用本地临时卷

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“工作负载”，在右侧选择“无状态负载”页签。

步骤3 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 本地临时卷(EmptyDir)”。

步骤4 本文主要为您介绍存储卷的挂载使用，如**表12-41**，其他参数详情请参见**工作负载**。

表 12-41 本地临时卷挂载

参数	参数说明
容量	申请的存储卷容量大小。
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“**/var/run**”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	• 只读：只能读容器路径中的数据卷。 • 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

步骤5 其余工作负载参数都配置完成后，单击“创建工作负载”。

----结束

通过 kubectl 使用本地临时卷

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑nginx-emptydir.yaml文件。

vi nginx-emptydir.yaml

YAML文件内容如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-emptydir
  namespace: default
spec:
  replicas: 2
  selector:
```

```
  matchLabels:
    app: nginx-emptydir
template:
  metadata:
    labels:
      app: nginx-emptydir
  spec:
    containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
          - name: vol-emptydir          # 卷名称，需与volumes字段中的卷名称对应
            mountPath: /tmp             # emptyDir挂载路径
    imagePullSecrets:
      - name: default-secret
    volumes:
      - name: vol-emptydir              # 卷名称，可自定义
        emptyDir:
          medium: LocalVolume           # emptyDir磁盘介质设置为LocalVolume，表示使用本地临时卷
          sizeLimit: 1Gi                # 卷容量大小
```

步骤3 创建工作负载。

kubectl apply -f nginx-emptydir.yaml

----结束

12.8.4 使用临时路径

临时路径是Kubernetes原生的EmptyDir类型，生命周期与容器实例相同，并支持指定内存作为存储介质。容器实例消亡时，EmptyDir会被删除，数据会永久丢失。

通过控制台使用临时路径

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“工作负载”，在右侧选择“无状态负载”页签。

步骤3 单击页面右上角“创建负载”，在“容器配置”中选择“数据存储”页签，并单击“添加存储卷 > 临时路径(EmptyDir)”。

步骤4 本文主要为您介绍存储卷的挂载使用，如**表12-42**，其他参数详情请参见**工作负载**。

表 12-42 临时路径挂载

参数	参数说明
存储介质	开启内存： • 开启后可以使用内存提高运行速度，但存储容量受内存大小限制。适用于数据量少，读写效率要求高的场景。 • 未开启时默认存储在硬盘上，适用于数据量大，读写效率要求低的场景。 **说明** • 开启内存后请注意内存大小，如果存储容量超出内存大小会发生OOM事件。 • 使用内存时的EmptyDir的大小为Pod规格限制值的100%。 • 不使用内存的EmptyDir不会占用系统内存。

参数	参数说明
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下 ，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	• 只读：只能读容器路径中的数据卷。 • 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

步骤5 其余工作负载参数都配置完成后，单击“创建工作负载”。

----结束

通过 kubectl 使用临时路径

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑nginx-emptydir.yaml文件。

vi nginx-emptydir.yaml

YAML文件内容如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-emptydir
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-emptydir
  template:
    metadata:
      labels:
        app: nginx-emptydir
    spec:
      containers:
        - name: container-1
          image: nginx:latest
          volumeMounts:
            - name: vol-emptydir     # 卷名称，需与volumes字段中的卷名称对应
              mountPath: /tmp        # emptyDir挂载路径
      imagePullSecrets:
        - name: default-secret
      volumes:
        - name: vol-emptydir         # 卷名称，可自定义
```

```
      emptyDir:
        medium: Memory          # emptyDir磁盘介质：设置为Memory时，表示开启内存；设置为空时为原生默认的存储介质类型
        sizeLimit: 1Gi          # 卷容量大小
```

步骤3 创建工作负载。

kubectl apply -f nginx-emptydir.yaml

----结束

12.9 主机路径（HostPath）

主机路径（HostPath）可以将容器所在宿主机的文件目录挂载到容器指定的挂载点中，如容器需要访问/etc/hosts则可以使用HostPath映射/etc/hosts等场景。

须知

- HostPath卷存在许多安全风险，最佳做法是尽可能避免使用HostPath。 当必须使用HostPath卷时，它的范围应仅限于所需的文件或目录，并以只读方式挂载。
- 当挂载HostPath卷的Pod删除后，HostPath中的数据依然会保留。

通过控制台使用主机路径

主机路径(HostPath)挂载表示将主机上的路径挂载到指定的容器路径。通常用于：“容器工作负载程序生成的日志文件需要永久保存”或者“需要访问宿主机上Docker引擎内部数据结构的容器工作负载”。

步骤1 登录CCE控制台。

步骤2 在创建工作负载时，在“容器配置”中找到“数据存储”页签，单击“添加存储卷 > 主机路径(HostPath)”。

步骤3 设置添加本地磁盘参数，如表12-43。

表 12-43 卷类型选择主机路径挂载

参数	参数说明
存储类型	主机路径(HostPath)。

参数	参数说明
主机路径	输入主机路径，如/etc/hosts。 **说明** 请注意“主机路径”不能设置为根目录“/”，否则将导致挂载失败。挂载路径一般设置为： • /opt/xxxx（但不能为/opt/cloud） • /mnt/xxxx（但不能为/mnt/paas） • /tmp/xxx • /var/xxx（但不能为/var/lib、/var/script、/var/paas等关键目录） • /xxxx（但不能和系统目录冲突，例如bin、lib、home、root、boot、dev、etc、lost+found、mnt、proc、sbin、srv、tmp、var、media、opt、selinux、sys、usr等） 注意不能设置为/home/paas、/var/paas、/var/lib、/var/script、/mnt/paas、/opt/cloud，否则会导致系统或节点安装失败。
挂载路径	请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“**/var/run**”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。
子路径	请输入子路径，如：tmp，表示容器中挂载路径下的数据会存储在卷的tmp文件夹中。 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume。不填写时默认为根。
权限	• 只读：只能读容器路径中的数据卷。 • 读写：可修改容器路径中的数据卷，容器迁移时新写入的数据不会随之迁移，会造成数据丢失。

步骤4 其余信息都配置完成后，单击“创建工作负载”。

----结束

使用 kubectl 使用主机路径

步骤1 使用kubectl连接集群。

步骤2 创建并编辑nginx-hostpath.yaml文件。

vi nginx-hostpath.yaml

YAML文件内容如下，将节点上的/data目录挂载至容器中的/data目录下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-hostpath
  namespace: default
spec:
```

```
replicas: 2
selector:
  matchLabels:
    app: nginx-hostpath
template:
  metadata:
    labels:
      app: nginx-hostpath
  spec:
    containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
          - name: vol-hostpath        # 卷名称，需与volumes字段中的卷名称对应
            mountPath: /data          # 容器中的挂载路径
    imagePullSecrets:
      - name: default-secret
    volumes:
      - name: vol-hostpath            # 卷名称，可自定义
        hostPath:
          path: /data                 # 宿主机节点上的目录位置
```

步骤3 创建工作负载。

kubectl apply -f nginx-hostpath.yaml

----结束

12.10 存储类（StorageClass）

存储类介绍

StorageClass可译为存储类，描述了集群中的存储类型“分类”，可以表示为一个创建PV存储卷的配置模板。在创建PVC/PV均需要指定StorageClass。

作为使用者，只需要在声明PVC时指定StorageClassName即可自动创建PV及底层存储，大大减少了创建并维护PV的工作量。

除了使用CCE提供的**默认存储类**外，您也可以根据需求自定义存储类。

- **自定义存储类应用场景**
- **自定义StorageClass**
- **指定默认StorageClass**
- **指定StorageClass的企业项目**

CCE 默认存储类

目前CCE默认提供csi-disk、csi-nas、csi-obs等StorageClass，在声明PVC时使用对应StorageClassName，就可以自动创建对应类型PV，并自动创建底层的存储资源。

执行如下kubectl命令即可查询CCE提供的默认StorageClass。您可以使用CCE提供的CSI插件自定义创建StorageClass。

```
# kubectl get sc
NAME                PROVISIONER                  AGE
csi-disk            everest-csi-provisioner      17d      # 云硬盘
csi-disk-topology   everest-csi-provisioner      17d      # 延迟创建的云硬盘
csi-nas             everest-csi-provisioner      17d      # 文件存储 1.0
csi-obs             everest-csi-provisioner      17d      # 对象存储
csi-sfsturbo        everest-csi-provisioner      17d      # 极速文件存储
```

每个StorageClass都包含了动态制备PersistentVolume时会使用到的默认参数。如以下云硬盘存储类的示例：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

参数	描述
provisioner	存储资源提供商，CCE均由everest插件提供，此处只能填写everest-csi-provisioner。
parameters	存储参数，不同类型的存储支持的参数不同。
reclaimPolicy	用来指定创建PV的persistentVolumeReclaimPolicy字段值，支持Delete和Retain。如果StorageClass 对象被创建时没有指定reclaimPolicy，它将默认为Delete。 ● Delete：表示动态创建的PV，在销毁的时候也会自动销毁。 ● Retain：表示动态创建的PV，不会自动销毁。
allowVolumeExpansion	定义由此存储类创建的PV是否支持动态扩容，默认为false。是否能动态扩容是由底层存储插件来实现的，这里只是一个开关。
volumeBindingMode	表示卷绑定模式，即动态创建PV的时间，分为立即创建和延迟创建。 ● Immediate：创建PVC时完成PV绑定和动态创建。 ● WaitForFirstConsumer：延迟PV的绑定和创建，当在工作负载中使用该PVC时才执行PV创建和绑定流程。
mountOptions	该字段需要底层存储支持，如果不支持挂载选项，却指定了挂载选项，会导致创建PV操作失败。

自定义存储类应用场景

在CCE中使用存储时，最常见的方法是创建PVC时通过指定StorageClassName定义要创建存储的类型，如下所示，使用PVC申请一个SAS（高I/O）类型云硬盘/块存储。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS
spec:
  accessModes:
  - ReadWriteOnce
```

```
resources:
  requests:
    storage: 10Gi
storageClassName: csi-disk
```

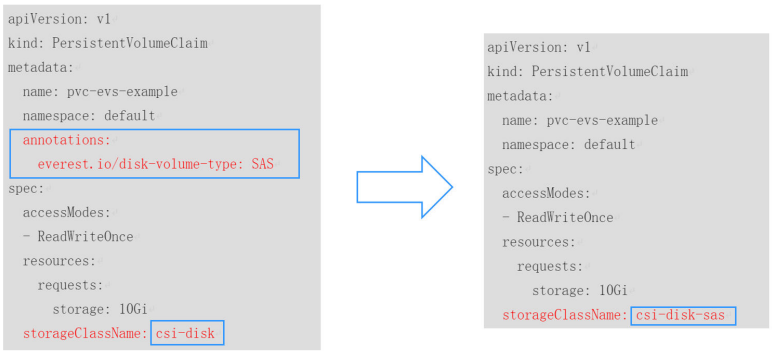
可以看到在CCE中如果需要指定云硬盘的类型，是通过everest.io/disk-volume-type字段指定，这里SAS是云硬盘的类型。

以上是较为基础的StorageClass使用方法，在实际应用中，也可以使用StorageClass进行更为简便的操作：

应用场景	解决方案	操作步骤
在使用annotations指定存储配置时，配置较为繁琐。例如此处使用everest.io/disk-volume-type字段指定云硬盘的类型。	在StorageClass的parameters字段中定义PVC的annotations，编写YAML时只需要指定StorageClassName。 例如，将SAS、SSD类型云硬盘分别定义一个StorageClass，比如定义一个名为csi-disk-sas的StorageClass，这个StorageClass创建SAS类型的存储，	自定义StorageClass
当用户从自建Kubernetes或其他Kubernetes服务迁移到CCE，原先的应用YAML中使用的StorageClass与CCE中使用的不同，导致使用存储时需要修改大量YAML文件或Helm Chart包，非常繁琐且容易出错。	在CCE集权中创建与原有应用YAML中相同名称的StorageClass，迁移后无需再修改应用YAML中的StorageClassName。 例如，迁移前使用的云硬盘存储类为disk-standard，在迁移CCE集群后，可以复制CCE集群中csi-disk存储类的YAML，将其名称修改为disk-standard后重新创建。	
在YAML中必须指定StorageClassName才能使用存储，不指定StorageClass时无法正常创建。	在集群中设置默认的StorageClass，则YAML中无需指定StorageClassName也能创建存储。	指定默认StorageClass

自定义 StorageClass

本文以自定义云硬盘类型的StorageClass为例，将SAS、SSD类型云硬盘分别定义一个StorageClass，比如定义一个名为csi-disk-sas的StorageClass，这个StorageClass创建SAS类型的存储，则前后使用的差异如下图所示，编写YAML时只需要指定StorageClassName。



未使用自定义StorageClass的写法　　使用自定义StorageClass的写法

- 自定义高I/O类型StorageClass，使用YAML描述如下，这里取名为csi-disk-sas，指定云硬盘类型为SAS，即高I/O。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-sas                          # 高IO StorageClass名字，用户可自定义
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS            # 云硬盘高I/O类型，用户不可自定义
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true                    # true表示允许扩容
```

- 超高I/O类型StorageClass，这里取名为csi-disk-ssd，指定云硬盘类型为SSD，即超高I/O。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd                          # 超高I/O StorageClass名字，用户可自定义
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD            # 云硬盘超高I/O类型，用户不可自定义
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

reclaimPolicy：底层云存储的回收策略，支持Delete、Retain回收策略。

- **Delete**：删除PVC，PV资源与云硬盘均被删除。
- **Retain**：删除PVC，PV资源与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，不能直接再次被PVC绑定使用。

如果数据安全性要求较高，建议使用**Retain**以免误删数据。

定义完之后，使用kubectl create命令创建。

```
# kubectl create -f sas.yaml
storageclass.storage.k8s.io/csi-disk-sas created
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
```

再次查询StorageClass，回显如下：

```
# kubectl get sc
NAME              PROVISIONER                  AGE
```

```
csi-disk             everest-csi-provisioner         17d
csi-disk-sas         everest-csi-provisioner         2m28s
csi-disk-ssd         everest-csi-provisioner         16s
csi-disk-topology    everest-csi-provisioner         17d
csi-nas              everest-csi-provisioner         17d
csi-obs              everest-csi-provisioner         17d
csi-sfsturbo         everest-csi-provisioner         17d
```

指定默认 StorageClass

您还可以指定某个StorageClass作为默认StorageClass，这样在创建PVC时不指定StorageClassName就会使用默认StorageClass创建。

例如将csi-disk-ssd指定为默认StorageClass，则可以按如下方式设置。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"   # 指定集群中默认的StorageClass，一个集群中只能有一个默认的StorageClass
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

先删除之前创建的csi-disk-ssd，再使用kubectl create命令重新创建，然后再查询StorageClass，显示如下。

```
# kubectl delete sc csi-disk-ssd
storageclass.storage.k8s.io "csi-disk-ssd" deleted
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
# kubectl get sc
NAME                    PROVISIONER                     AGE
csi-disk                everest-csi-provisioner         17d
csi-disk-sas            everest-csi-provisioner         114m
csi-disk-ssd (default)  everest-csi-provisioner         9s
csi-disk-topology       everest-csi-provisioner         17d
csi-nas                 everest-csi-provisioner         17d
csi-obs                 everest-csi-provisioner         17d
csi-sfsturbo            everest-csi-provisioner         17d
```

指定 StorageClass 的企业项目

CCE支持使用存储类创建云硬盘和对象存储类型PVC时指定企业项目，将创建的存储资源（云硬盘和对象存储）归属于指定的企业项目下，**企业项目可选为集群所属的企业项目或default企业项目**。

若不指定企业项目，则创建的存储资源默认使用存储类StorageClass中指定的企业项目，CCE提供的 csi-disk 和 csi-obs 存储类，所创建的存储资源属于default企业项目。

如果您希望通过StorageClass创建的存储资源能与集群在同一个企业项目，则可以自定义StorageClass，并指定企业项目ID，如下所示。

说明

该功能需要everest插件升级到1.2.33及以上版本。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk-epid      # 自定义名称
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/enterprise-project-id: 86bfc701-9d9e-4871-a318-6385aa368183  # 指定企业项目id
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

配置验证

- 使用csi-disk-sas创建PVC。

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  sas-disk
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
    storageClassName: csi-disk-sas
  ```

  创建并查看详情，如下所示，可以发现能够创建，且StorageClass显示为csi-disk-sas

  ```
  # kubectl create -f sas-disk.yaml
  persistentvolumeclaim/sas-disk created
  # kubectl get pvc
  NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
  STORAGECLASS   AGE
  sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            csi-disk-sas
  24s
  # kubectl get pv
  NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM                  STORAGECLASS   REASON   AGE
  pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            Delete           Bound       default/
  sas-disk         csi-disk-sas            30s
  ```

  在CCE控制台界面上查看PVC详情，在“PV详情”页签下可以看到磁盘类型是高I/O。

- 不指定StorageClassName，使用默认配置，如下所示，并未指定storageClassName。

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  ssd-disk
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
  ```

  创建并查看，可以看到PVC ssd-disk的StorageClass为csi-disk-ssd，说明默认使用了csi-disk-ssd。

  ```
  # kubectl create -f ssd-disk.yaml
  persistentvolumeclaim/ssd-disk created
  # kubectl get pvc
  NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
  STORAGECLASS   AGE
  ```

```
sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            csi-disk-sas
16m
ssd-disk   Bound    pvc-4d2b059c-0d6c-44af-9994-f74d01c78731   10Gi       RWO            csi-disk-ssd
10s
# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                    STORAGECLASS   REASON   AGE
pvc-4d2b059c-0d6c-44af-9994-f74d01c78731   10Gi       RWO            Delete           Bound
default/ssd-disk         csi-disk-ssd            15s
pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            Delete           Bound       default/
sas-disk         csi-disk-sas            17m
```

在CCE控制台界面上查看PVC详情，在“PV详情”页签下可以看到磁盘类型是超高I/O。

13 可观测性

13.1 日志管理

13.1.1 日志概述

CCE支持配置工作负载日志策略，便于日志的统一收集、管理和分析，以及按周期防爆处理。

- ICAgent日志采集：

  默认情况下，ICAgent会采集容器的标准输出，您无需做任何设置。

  您还可以在创建工作负载的时候设置容器日志存储路径，ICAgent会采集该路径下日志。

  容器日志可以选择主机路径和容器路径两种模式。

  - 主机路径：HostPath模式，将主机路径挂载到指定的容器路径（挂载路径）。用户可以在节点的主机路径中查看到容器输出在挂载路径中的日志信息。
  - 容器路径：EmptyDir模式，将节点的临时路径挂载到指定的路径（挂载路径）。临时路径中存在的但暂未被采集器上报到AOM的日志数据在Pod实例删除后会消失。

13.1.2 使用 ICAgent 采集容器日志

CCE配合AOM收集工作负载的日志，在创建节点时会默认安装AOM的ICAgent（在集群kube-system命名空间下名为icagent的DaemonSet），ICAgent负责收集工作负载的日志并上报到AOM，您可以在CCE控制台和AOM控制台查看工作负载的日志。

约束与限制

ICAgent只采集*.log、*.trace和*.out类型的文本日志文件。

使用 ICAgent 采集日志

步骤1 在CCE中创建工作负载时，在配置容器信息时可以设置容器日志。

步骤2 单击┼添加日志策略。

步骤3 存储类型有“主机路径”和“容器路径”两种类型可供选择：

表 13-1 配置日志策略

参数	参数说明
存储类型	• 主机路径：HostPath模式，将主机路径挂载到指定的容器路径（挂载路径）。用户可以在节点的主机路径中查看到容器输出在挂载路径中的日志信息。 • 容器路径：EmptyDir模式，将节点的临时路径挂载到指定的路径（挂载路径）。临时路径中存在的但暂未被采集器上报到AOM的日志数据在Pod实例删除后会消失。
主机路径	请输入主机的路径，如：/var/paas/sys/log/nginx
挂载路径	请输入数据存储要挂载到容器上的路径，如：/tmp **须知** • 请不要挂载到系统目录下，如“/”、“**/var/run**”等，否则会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 • 挂载高危目录的情况下 ，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。 • AOM只采集最近修改过的前20个日志文件，且默认采集两级子目录。 • AOM只采集挂载路径下的“.log”、“.trace”、“.out”文本日志文件。 • 容器中挂载点的权限设置方法，请参见**为Pod或容器配置安全性上下文**。
主机扩展路径	仅“主机路径”类型需要填写 通过实例的ID或者容器的名称扩展主机路径，实现同一个主机路径下区分来自不同容器的挂载。 会在原先的“卷目录/子目录”中增加一个三级目录。使用户更方便获取单个Pod输出的文件。 • None：不配置拓展路径。 • PodUID：Pod的ID。 • PodName：Pod的名称。 • PodUID/ContainerName：Pod的ID/容器名称。 • PodName/ContainerName：Pod名称/容器名称。

参数	参数说明
采集路径	设置采集路径可以更精确的指定采集内容，当前支持以下设置方式： ● 不设置则默认采集当前路径下.log .trace .out文件 ● 设置**表示递归采集5层目录下的.log .trace .out文件 ● 设置*表示模糊匹配 例子： 采集路径为/tmp/**/test*.log表示采集/tmp目录及其1-5层子目录下的全部以test开头的.log文件。 **注意** 使用采集路径功能请确认您的采集器ICAgent版本为5.12.22或以上版本。
日志转储	此处日志转储是指日志的本地绕接。 ● 设置：AOM每分钟扫描一次日志文件，当某个日志文件超过50MB时会对其转储（转储时会在该日志文件所在的目录下生成一个新的zip文件。对于一个日志文件，AOM只保留最近生成的20个zip文件，当zip文件超过20个时，时间较早的zip文件会被删除）。 ● 不设置：若您在下拉列表框中选择“不设置”，则AOM不会对日志文件进行转储。 **说明** ● AOM的日志绕接能力是使用copytruncate方式实现的，如果选择了设置，请务必保证您写日志文件的方式是append（追加模式），否则可能出现文件空洞问题。 ● 当前主流的日志组件例如Log4j、Logback等均已经具备日志文件的绕接能力，如果您的日志文件已经实现了绕接能力，则无需设置。否则可能出现冲突。 ● 建议您的业务自己实现绕接，可以更灵活的控制绕接文件的大小和个数。

步骤4 单击“确定”，并完成创建工作负载。

----结束

YAML 示例

您可以通过在YAML定义的方式设置容器日志存储路径。

如下所示，使用EmptyDir挂载到容器的“/var/log/nginx”路径下，这样ICAgent就会采集容器“/var/log/nginx”路径下的日志。其中policy字段是CCE自定义的字段，能够让ICAgent识别并采集日志。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: testlog
  namespace: default
spec:
  selector:
    matchLabels:
      app: testlog
  template:
    replicas: 1
    metadata:
```

```
    labels:
      app: testlog
  spec:
    containers:
      - image: 'nginx:alpine'
        name: container-0
        resources:
          requests:
            cpu: 250m
            memory: 512Mi
          limits:
            cpu: 250m
            memory: 512Mi
        volumeMounts:
          - name: vol-log
            mountPath: /var/log/nginx
            policy:
              logs:
                rotate: ''
    volumes:
      - emptyDir: {}
        name: vol-log
    imagePullSecrets:
      - name: default-secret
```

使用HostPath方法如下所示，相比EmptyDir就是volume的类型变成hostPath，且需要配置hostPath在主机上的路径。下面示例中将主机上“/tmp/log”挂载到容器的“/var/log/nginx”路径下，这样ICAgent就会采集容器“/var/log/nginx”路径下的日志，且日志还会在主机的“/tmp/log”路径下存储。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: testlog
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: testlog
  template:
    metadata:
      labels:
        app: testlog
    spec:
      containers:
        - image: 'nginx:alpine'
          name: container-0
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
            limits:
              cpu: 250m
              memory: 512Mi
          volumeMounts:
            - name: vol-log
              mountPath: /var/log/nginx
              readOnly: false
              extendPathMode: PodUID
              policy:
                logs:
                  rotate: Hourly
                  annotations:
                    pathPattern: '**'

      volumes:
        - hostPath:
            path: /tmp/log
```

```
    name: vol-log
imagePullSecrets:
  - name: default-secret
```

表 13-2 关键参数解释

参数	解释	说明
extendPathMode	主机扩展路径	通过实例的ID或者容器的名称扩展主机路径，实现同一个主机路径下区分来自不同容器的挂载。 会在原先的“卷目录/子目录”中增加一个三级目录。使用户更方便获取单个Pod输出的文件。 ● None：不配置拓展路径。 ● PodUID：Pod的ID。 ● PodName：Pod的名称。 ● PodUID/ContainerName：Pod的ID/容器名称。 ● PodName/ContainerName：Pod名称/容器名称。
policy.logs.rotate	日志转储	此处日志转储是指日志的本地绕接。 ● 设置：AOM每分钟扫描一次日志文件，当某个日志文件超过50MB时，会立即对其转储（转储时会在该日志文件所在的目录下生成一个新的zip文件。对于一个日志文件，AOM只保留最近生成的20个zip文件，当zip文件超过20个时，时间较早的zip文件会被删除），转储完成后AOM会将该日志文件清空。 ● 不设置：若您在下拉列表框中选择“不设置”，则AOM不会对日志文件进行转储。 **说明** ● AOM的日志绕接能力是使用copytruncate方式实现的，如果选择了设置，请务必保证您写日志文件的方式是append（追加模式），否则可能出现文件空洞问题。 ● 当前主流的日志组件例如Log4j、Logback等均已经具备日志文件的绕接能力，如果您的日志文件已经实现了绕接能力，则无需设置。否则可能出现冲突。 ● 建议您的业务自己实现绕接，可以更灵活的控制绕接文件的大小和个数。
policy.logs.annotations.pathPattern	采集路径	设置采集路径可以更精确的指定采集内容，当前支持以下设置方式： ● 不设置则默认采集当前路径下.log .trace .out文件 ● 设置**表示递归采集5层目录下的.log .trace .out文件 ● 设置*表示模糊匹配 例子： 采集路径为/tmp/**/test*.log表示采集/tmp目录及其1-5层子目录下的全部以test开头的.log文件。 **注意** 使用采集路径功能请确认您的采集器ICAgent版本为5.12.22或以上版本。

查看日志

日志采集路径配置和工作负载创建完成后，若已配置的路径下存在日志文件，则ICAgent会从已配置的路径中采集日志文件，采集大概需要1分钟，请您耐心等待。

待采集完成后，进入工作负载详情页，单击右上角的“日志”按钮查看日志详情。

您还可以在AOM控制台查看日志。

另外您还可以使用kubectl logs命令查看容器的标准输出，具体如下所示。

```
# 查看指定pod的日志
kubectl logs <pod_name>
kubectl logs -f <pod_name> #类似tail -f的方式查看

# 查看指定pod中指定容器的日志
kubectl logs <pod_name> -c <container_name>

kubectl logs pod_name -c container_name -n namespace (一次性查看)
kubectl logs -f <pod_name> -n namespace (tail -f方式实时查看)
```

13.2 监控管理

13.2.1 监控概述

CCE配合AOM对集群进行全方位的监控，在创建节点时会默认安装AOM的ICAgent（在集群kube-system命名空间下名为icagent的DaemonSet），ICAgent默认采集集群底层资源以及运行在集群上负载的监控数据；另外，ICAgent还能采集负载的自定义指标监控数据。

- 资源监控指标

  资源基础监控包含CPU/内存/磁盘等，具体请参见**资源监控指标**。您可以在CCE控制台从集群、节点、工作负载等维度查看这些监控指标数据，也可以在AOM中查看。

- 自定义指标

  ICAgent采集应用程序中的自定义指标并上传到AOM，具体使用方法请参见**使用AOM监控自定义指标**。

资源监控指标

在CCE控制台，可以查看如下指标。

- **查看集群监控数据**
- **查看节点监控数据**
- **查看工作负载的监控数据**
- **查看容器实例Pod的监控数据**

在AOM控制台，可以查看主机指标和容器实例的指标。

查看集群监控数据

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏单击“集群信息”，在右侧可看到集群所有节点（不含控制节点）近一小时的CPU指标和内存指标，以及控制节点的状态、所在可用区、CPU使用率和内存使用率等信息。

表 13-3 集群监控指标

监控指标	指标含义
CPU分配率	分配给工作负载使用的CPU占比。 CPU分配率 = 集群下运行的Pod CPU配额申请值（Request）之和 / 集群下所有节点（不含控制节点）的CPU可分配量之和
内存分配率	分配给工作负载使用的内存占比。 内存分配率 = 集群下运行的Pod 内存配额申请值（Request）之和 / 集群下所有节点（不含控制节点）的内存可分配量之和
CPU使用率	集群CPU使用率。 CPU使用率 = 集群下所有节点（不含控制节点）上实际使用的CPU使用率的平均值
内存使用率	集群内存使用率。 内存使用率 = 集群下所有节点（不含控制节点）上实际使用的内存使用率的平均值

说明

节点资源（CPU或内存）可分配量=总量-预留值-驱逐阈值。详情请参见**节点预留资源策略说明**。

----结束

查看节点监控数据

除了在集群监控界面查看所有节点监控数据外，您还可以查看单个节点的监控数据。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“节点管理”，在右侧节点所在行单击“监控”即可查看节点监控数据。

步骤3 您可以自定义调整数据的统计方式及时间范围。监控数据来源于AOM，可查看节点的监控数据包括CPU、内存、磁盘、网络、GPU等。

表 13-4 节点监控指标

监控指标	指标含义
CPU使用率	节点CPU使用率。 CPU使用率 = CPU内核占用 / CPU总核数
CPU内核占用	已实际使用的CPU核个数。
物理内存使用率	节点物理内存使用率。 内存使用率 =（物理内存容量 - 可用物理内存）/ 物理内存容量
可用物理内存	节点尚未被使用的物理内存。
磁盘使用率	节点数据盘上文件系统的磁盘使用率，根据文件分区分别计算。数据盘分区详情请参见**数据盘空间分配说明**。 磁盘使用率 =（磁盘容量 - 可用磁盘空间）/ 磁盘容量
可用磁盘空间	还未经使用的磁盘空间，单位为GiB。
下行速率	一般指从网络下载数据到节点的速度，单位为KB/s。
上行速率	一般指从节点上传网络的速度，单位为KB/s。
GPU使用率	节点GPU使用率。
显存使用率	已使用的显存占显存容量的百分比。 显存使用率 = 显存使用量 / 显存容量
显存使用量	已使用的显存大小，单位为GiB。

----结束

查看工作负载的监控数据

工作负载的监控数据可以在工作负载详情的监控页面下查看。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，在右侧工作负载所在行单击“监控”即可查看工作负载监控数据。

步骤3 您可以自定义调整数据的统计方式及时间范围。监控数据来源于AOM，可查看工作负载的监控数据包括CPU、内存、网络、GPU等。

说明

如果工作负载有多个实例，监控数据可能根据“统计方式”而不同。例如，当选择“最大/最小值”时，各监控数据的取值为该工作负载下所有实例的最大/最小值。选择“平均值”时，各监控数据的取值为该工作负载下所有实例的平均值。

表 13-5 工作负载监控指标

监控指标	指标含义
CPU使用率	工作负载的CPU使用率。 CPU使用率 = CPU内核占用 / 所有业务容器CPU核数限制值之和（未配置限制值时采用节点总量）
CPU内核占用	已实际使用的CPU核个数。
物理内存使用率	工作负载的物理内存使用率。 内存使用率 = 物理内存使用量 / 所有业务容器CPU核数限制值之和（未配置限制值时采用节点总量）
物理内存使用量	已实际使用的物理内存。
磁盘读取速率	每秒从磁盘读出的数据量，单位为KB/s。
磁盘写入速率	每秒写入磁盘的数据量，单位为KB/s。
下行速率	一般指从网络下载数据的速度，单位为KB/s。
上行速率	一般指从节点上传网络的速度，单位为KB/s。
GPU使用率	工作负载GPU使用率。
显存使用率	已使用的显存占显存容量的百分比。 显存使用率 = 显存使用量 / 显存容量
显存使用量	已使用的显存大小，单位为GiB。

----结束

查看容器实例 Pod 的监控数据

在工作负载详情页面的实例列表页签中可以查看Pod的监控数据。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，在右侧单击工作负载名称，查看实例列表。

步骤3 在实例所在行单击“监控”即可查看某个实例的监控数据。

步骤4 您可以自定义调整数据的统计方式及时间范围。监控数据来源于AOM，可查看实例的监控数据包括CPU、内存、网络、GPU等。

说明

如果单个实例下存在多个容器，监控数据可能根据“统计方式”而不同。例如，当选择“最大/最小值”时，各监控数据的取值为该实例下所有容器的最大/最小值。选择“平均值”时，各监控数据的取值为该实例下所有容器的平均值。

表 13-6 实例监控指标

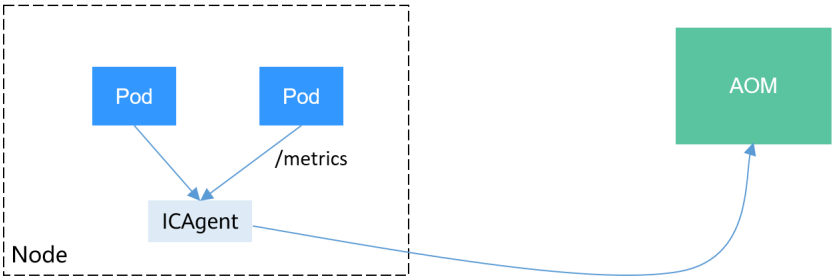
监控指标	指标含义
CPU使用率	Pod的CPU使用率。 CPU使用率 = Pod实际使用的CPU核数 / Pod中所有业务容器CPU核数限制值之和（未配置限制值时采用节点总量）
CPU内核占用	已实际使用的CPU核个数。
物理内存使用率	Pod的物理内存使用率。 内存使用率 = Pod实际使用的物理内存 / Pod中所有业务容器物理内存限制值之和（未配置限制值时采用节点总量）
物理内存使用量	已实际使用的物理内存。
磁盘读取速率	每秒从磁盘读出的数据量，单位为KB/s。
磁盘写入速率	每秒写入磁盘的数据量，单位为KB/s。
下行速率	一般指从网络下载数据的速度，单位为KB/s。
上行速率	一般指从节点上传网络的速度，单位为KB/s。
GPU使用率	Pod的GPU使用率。
显存使用率	已使用的显存占显存容量的百分比。 显存使用率 = 显存使用量 / 显存容量
显存使用量	Pod已使用的显存大小，单位为GiB。

----结束

13.2.2 使用 AOM 监控自定义指标

CCE支持上传自定义指标到AOM，节点上的ICAgent会定期调用负载中配置的监控指标接口读取监控数据，然后上传到AOM上。

图 13-1 ICAgent 采集监控指标



负载的自定义指标接口可以在创建时配置。本文将通过一个Nginx应用的示例演示如何上报自定义监控指标到AOM，步骤如下：

1. **准备应用**

您需要准备一个应用镜像，该应用需要提供监控指标接口供ICAgent采集，且监控数据需要**满足Prometheus的规范**。

2. **部署应用并转换指标**

   在集群中使用该应用镜像部署工作负载，将自动上报自定义监控指标。

3. **配置验证**

   前往AOM查看自定义指标是否采集成功。

约束与限制

- ICAgent兼容**Prometheus**的监控数据规范，Pod提供的自定义指标必须满足Prometheus的监控数据规范才能够被ICAgent采集，参见**Prometheus监控数据采集说明**。
- ICAgent仅支持上报**Gauge指标类型**的指标。
- ICAgent调用自定义指标的接口周期为1分钟，不支持修改。

Prometheus 监控数据采集说明

Prometheus通过周期性的调用应用程序的监控指标接口（默认为“/metrics”）获取监控数据，应用程序需要提供监控指标接口供Prometheus调用，且监控数据需要满足Prometheus的规范，如下所示。

```
# TYPE nginx_connections_active gauge
nginx_connections_active 2
# TYPE nginx_connections_reading gauge
nginx_connections_reading 0
```

Prometheus提供了各种语言的客户端，客户端具体请参见**Prometheus CLIENT LIBRARIES**，开发Exporter具体方法请参见**WRITING EXPORTERS**。Prometheus社区提供丰富的第三方exporter可以直接使用，具体请参见**EXPORTERS AND INTEGRATIONS**。

准备应用

自行开发的应用程序需要提供监控指标接口供ICAgent采集，且监控数据需要满足Prometheus的规范，详情请参见**Prometheus监控数据采集说明**。

本文以Nginx为例采集监控数据，Nginx本身有个名叫ngx_http_stub_status_module的模块，这个模块提供了基本的监控功能，通过在nginx.conf的配置可以提供一个对外访问Nginx监控数据的接口。

步骤1 登录一台可连接公网的Linux虚拟机，且要求可执行Docker命令。

步骤2 创建一个nginx.conf文件，如下所示，在http下添加server配置即可让nginx提供对外访问的监控数据的接口。

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
```

```
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;
    sendfile        on;
    #tcp_nopush     on;
    keepalive_timeout  65;
    #gzip  on;
    include /etc/nginx/conf.d/*.conf;

    server {
      listen 8080;
      server_name  localhost;
      location /stub_status {
         stub_status on;
         access_log off;
      }
    }
}
```

步骤3 使用该配置制作一个镜像，创建Dockerfile文件。

```
vi Dockerfile
```

Dockerfile文件内容如下所示：

```
FROM nginx:1.21.5-alpine
ADD nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

步骤4 使用上面Dockerfile构建镜像并上传到SWR镜像仓库，镜像名称为nginx:exporter。

1. 在左侧导航栏选择“我的镜像”，单击右侧“客户端上传”，在弹出的页面中单击“生成临时登录指令”，单击 复制登录指令。
2. 在集群节点上执行上一步复制的登录指令，登录成功会显示“Login Succeeded”。
3. 执行如下命令构建镜像，镜像名称为nginx，版本为exporter。
   ```
   docker build -t nginx:exporter .
   ```
4. 为镜像打标签并上传至镜像仓库，其中镜像仓库地址和组织名称请根据实际情况修改。
   ```
   docker tag nginx:exporter {swr-address}/{group}/nginx:exporter
   docker push {swr-address}/{group}/nginx:exporter
   ```

步骤5 查看应用指标。

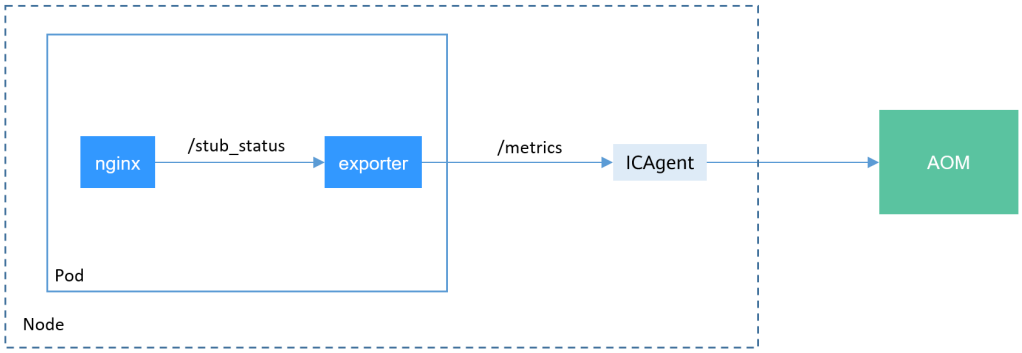
1. 使用nginx:exporter创建工作负载。
2. **登录到容器中**，并通过http://<ip_address>:8080/stub_status获取到nginx的监控数据，其中<ip_address>为容器的IP地址，监控数据如下所示。
   ```
   # curl http://127.0.0.1:8080/stub_status
   Active connections: 3
   server accepts handled requests
    146269 146269 212
   Reading: 0 Writing: 1 Waiting: 2
   ```

----结束

部署应用并转换指标

如上所述的nginx:exporter提供的监控数据，其数据格式并不满足Prometheus的要求，需要将其转换成Prometheus需要的格式，可以使用**nginx-prometheus-exporter**来转换Nginx的指标，如下所示。

图 13-2 使用 exporter 转换数据格式



使用nginx:exporter和nginx-prometheus-exporter部署到同一个Pod，如下所示。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx-exporter
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-exporter
  template:
    metadata:
      labels:
        app: nginx-exporter
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"prometheus","path":"/
metrics","port":"9113","names":""}]'
    spec:
      containers:
        - name: container-0
          image: 'nginx:exporter'  # 替换为您上传到SWR的镜像地址
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
        - name: container-1
          image: 'nginx/nginx-prometheus-exporter:0.9.0'
          command:
            - nginx-prometheus-exporter
          args:
            - '-nginx.scrape-uri=http://127.0.0.1:8080/stub_status'
      imagePullSecrets:
        - name: default-secret
```

说明

nginx/nginx-prometheus-exporter:0.9.0需要从公网拉取，需要集群节点绑定公网IP。

nginx-prometheus-exporter需要一个启动命令，nginx-prometheus-exporter -nginx.scrape-uri=http://127.0.0.1:8080/stub_status，用于获取nginx的监控数据。

另外Pod需要添加一个annotations，metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"prometheus","path":"**/metrics**","port":"**9113**","names":""}]'。

配置验证

应用部署后，可以通过访问Nginx构造一些访问数据，然后在AOM中查看是否能够获取到相应的监控数据。

步骤1 获取Nginx Pod名称。

```
$ kubectl get pod
NAME                              READY   STATUS    RESTARTS   AGE
nginx-exporter-78859765db-6j8sw   2/2     Running   0          4m
```

步骤2 登录容器执行命令访问Nginx。

```
$ kubectl exec -it nginx-exporter-78859765db-6j8sw -- /bin/sh
Defaulting container name to container-0.
Use 'kubectl describe pod/nginx-exporter-78859765db-6j8sw -n default' to see all of the containers in this pod.
/ # curl http://localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ #
```

步骤3 登录AOM，在左侧目录选择“监控 > 指标浏览”，查看Nginx相关的监控指标，如“nginx_connections_active”。

----结束

13.3 云审计日志

13.3.1 云审计服务支持的 CCE 操作列表

CCE通过云审计服务（Cloud Trace Service，简称CTS）为您提供云服务资源的操作记录，记录内容包括您从云管理控制台或者开放API发起的云服务资源操作请求以及每次请求的结果，供您查询、审计和回溯使用。

表 13-7 云审计服务支持的 CCE 操作列表

操作名称	资源类型	事件名称
创建用户委托	集群	createUserAgencies

操作名称	资源类型	事件名称
创建集群	集群	createCluster
更新集群描述	集群	updateCluster
升级集群	集群	clusterUpgrade
删除集群	集群	claimCluster/deleteCluster
下载集群证书	集群	getClusterCertByUID
绑定、解绑eip	集群	operateMasterEIP
集群休眠唤醒、节点纳管重置（V2）	集群	operateCluster
集群休眠（V3）	集群	hibernateCluster
集群唤醒（V3）	集群	awakeCluster
集群规格变更	集群	resizeCluster
修改集群配置	集群	updateConfiguration
创建节点池	节点池	createNodePool
更新节点池	节点池	updateNodePool
删除节点池	节点池	claimNodePool
迁移节点池	节点池	migrateNodepool
修改节点池配置	节点池	updateConfiguration
创建节点	节点	createNode
删除集群下所有节点	节点	deleteAllHosts
删除单个节点	节点	deleteOneHost/claimOneHost
更新节点描述	节点	updateNode
创建插件实例	插件实例	createAddonInstance
删除插件实例	插件实例	deleteAddonInstance
上传模板	模板	uploadChart
更新模板	模板	updateChart
删除模板	模板	deleteChart
创建模板实例	模板实例	createRelease
升级模板实例	模板实例	updateRelease
删除模板实例	模板实例	deleteRelease
创建ConfigMap	Kubernetes资源	createConfigmaps

操作名称	资源类型	事件名称
创建DaemonSet	Kubernetes资源	createDaemonsets
创建Deployment	Kubernetes资源	createDeployments
创建Event	Kubernetes资源	createEvents
创建Ingress	Kubernetes资源	createIngresses
创建Job	Kubernetes资源	createJobs
创建namespace	Kubernetes资源	createNamespaces
创建Node	Kubernetes资源	createNodes
创建PersistentVolumeClaim	Kubernetes资源	createPersistentvolumeclaims
创建Pod	Kubernetes资源	createPods
创建ReplicaSet	Kubernetes资源	createReplicasets
创建ResourceQuota	Kubernetes资源	createResourcequotas
创建密钥	Kubernetes资源	createSecrets
创建服务	Kubernetes资源	createServices
创建StatefulSet	Kubernetes资源	createStatefulsets
创建卷	Kubernetes资源	createVolumes
删除ConfigMap	Kubernetes资源	deleteConfigmaps
删除DaemonSet	Kubernetes资源	deleteDaemonsets
删除Deployment	Kubernetes资源	deleteDeployments
删除Event	Kubernetes资源	deleteEvents
删除Ingress	Kubernetes资源	deleteIngresses
删除Job	Kubernetes资源	deleteJobs
删除Namespace	Kubernetes资源	deleteNamespaces
删除Node	Kubernetes资源	deleteNodes
删除Pod	Kubernetes资源	deletePods
删除ReplicaSet	Kubernetes资源	deleteReplicasets
删除ResourceQuota	Kubernetes资源	deleteResourcequotas
删除Secret	Kubernetes资源	deleteSecrets
删除Service	Kubernetes资源	deleteServices
删除StatefulSet	Kubernetes资源	deleteStatefulsets

操作名称	资源类型	事件名称
删除卷	Kubernetes资源	deleteVolumes
替换指定的ConfigMap	Kubernetes资源	updateConfigmaps
替换指定的DaemonSet	Kubernetes资源	updateDaemonsets
替换指定的Deployment	Kubernetes资源	updateDeployments
替换指定的Event	Kubernetes资源	updateEvents
替换指定的Ingress	Kubernetes资源	updateIngresses
替换指定的Job	Kubernetes资源	updateJobs
替换指定的Namespace	Kubernetes资源	updateNamespaces
替换指定的Node	Kubernetes资源	updateNodes
替换指定的PersistentVolumeClaim	Kubernetes资源	updatePersistentvolumeclaims
替换指定的Pod	Kubernetes资源	updatePods
替换指定的Replicaset	Kubernetes资源	updateReplicasets
替换指定的ResourceQuota	Kubernetes资源	updateResourcequotas
替换指定的Secret	Kubernetes资源	updateSecrets
替换指定的Service	Kubernetes资源	updateServices
替换指定的Statefulset	Kubernetes资源	updateStatefulsets
替换指定的状态	Kubernetes资源	updateStatus
上传组件模板	Kubernetes资源	uploadChart
更新组件模板	Kubernetes资源	updateChart
删除组件模板	Kubernetes资源	deleteChart
创建模板应用	Kubernetes资源	createRelease
更新模板应用	Kubernetes资源	updateRelease
删除模板应用	Kubernetes资源	deleteRelease

13.3.2 查询审计事件

操作场景

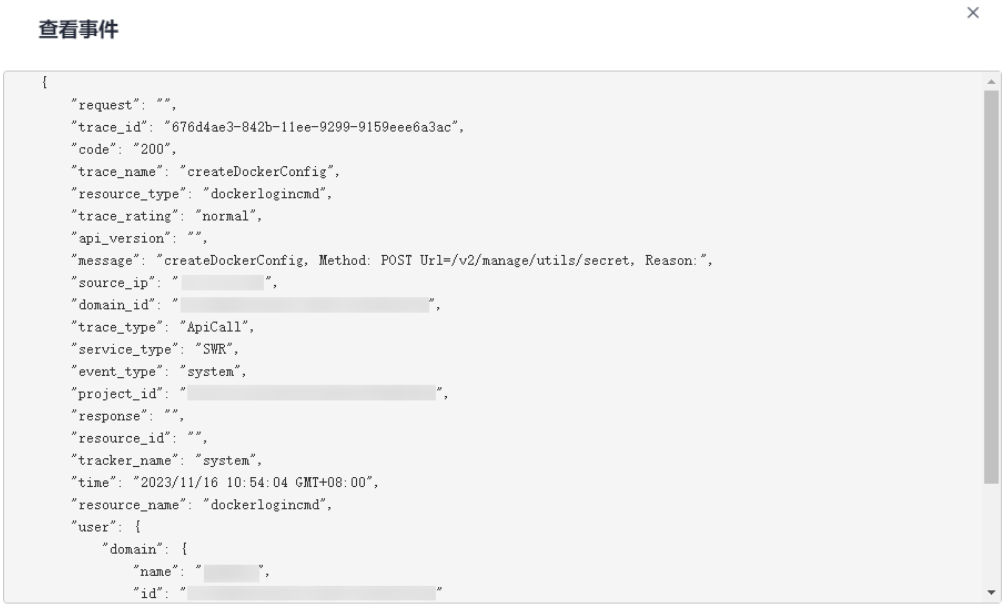
用户进入云审计服务创建管理类追踪器后，系统开始记录云服务资源的操作。云审计服务管理控制台会保存最近7天的操作记录。

本节介绍如何在云审计服务管理控制台查看或导出最近7天的操作记录：

- **在事件列表查看审计事件**

在事件列表查看审计事件

1. 登录管理控制台。
2. 单击左上角 ☰，选择“管理与部署 > 云审计服务 CTS”，进入云审计服务页面。
3. 单击左侧导航树的“事件列表”，进入事件列表信息页面。
4. 事件列表支持通过筛选来查询对应的操作事件。当前事件列表支持四个维度的组合查询，详细信息如下：
   - 事件类型、事件来源、资源类型和筛选类型，在下拉框中选择查询条件。
     - 筛选类型按资源ID筛选时，还需手动输入某个具体的资源ID。
     - 筛选类型按事件名称筛选时，还需选择某个具体的事件名称。
     - 筛选类型按资源名称筛选时，还需选择或手动输入某个具体的资源名称。
   - 操作用户：在下拉框中选择某一具体的操作用户，此操作用户指用户级别，而非租户级别。
   - 事件级别：可选项为“所有事件级别”、“Normal”、“Warning”、“Incident”，只可选择其中一项。
   - 时间范围：可选择查询最近7天内任意时间段的操作事件。
   - 单击“导出”按钮，云审计服务会将查询结果以CSV格式的表格文件导出，该CSV文件包含了本次查询结果的所有事件，且最多导出5000条信息。
5. 选择完查询条件后，单击“查询”。
6. 在事件列表页面，您还可以导出操作记录文件和刷新列表。
   - 单击“导出”按钮，云审计服务会将查询结果以CSV格式的表格文件导出，该CSV文件包含了本次查询结果的所有事件，且最多导出5000条信息。
   - 单击 ↻ 按钮，可以获取到事件操作记录的最新信息。
7. 在需要查看的事件左侧，单击 ⌄ 展开该记录的详细信息。
8. 在需要查看的记录右侧，单击“查看事件”，会弹出一个窗口显示该操作事件结构的详细信息。



9. 关于事件结构的关键字段详解，请参见“云审计服务事件参考 > 事件结构”章节和“云审计服务事件参考 > 事件样例”章节。

14 命名空间

14.1 创建命名空间

操作场景

命名空间（Namespace）是对一组资源和对象的抽象整合。在同一个集群内可创建不同的命名空间，不同命名空间中的数据彼此隔离。使得它们既可以共享同一个集群的服务，也能够互不干扰。

例如可以将开发环境、测试环境的业务分别放在不同的命名空间。

前提条件

至少已创建一个集群。

约束与限制

每个命名空间下，创建的服务数量不能超过6000个。此处的服务对应kubernetes的service资源，即工作负载所添加的服务。

命名空间类别

命名空间按创建类型分为两大类：集群默认创建的、用户创建的。

- 集群默认创建的：集群在启动时会默认创建default、kube-public、kube-system、kube-node-lease命名空间。
  - default：所有未指定Namespace的对象都会被分配在default命名空间。
  - kube-public：此命名空间下的资源可以被所有人访问（包括未认证用户），用来部署公共插件、容器模板等。
  - kube-system：所有由Kubernetes系统创建的资源都处于这个命名空间。
  - kube-node-lease：每个节点在该命名空间中都有一个关联的“Lease”对象，该对象由节点定期更新。NodeStatus和NodeLease都被视为来自节点的心跳，在v1.13之前的版本中，节点的心跳只有NodeStatus，NodeLease特性从v1.13开始引入。NodeLease比NodeStatus更轻量级，该特性在集群规模扩展性和性能上有明显提升。

- 用户创建的：用户可以按照需要创建命名空间，例如开发环境、联调环境和测试环境分别创建对应的命名空间。或者按照不同的业务创建对应的命名空间，例如系统若分为登录和游戏服务，可以分别创建对应命名空间。

创建命名空间

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“命名空间”，在右上角单击“创建命名空间”。

步骤3 参照表14-1设置命名空间参数。

表 14-1 命名空间基本信息

参数	参数说明
名称	新建命名空间的名称，命名必须唯一。
描述	输入对命名空间的描述信息。
配额管理	资源配额可以限制命名空间下的资源使用，进而支持以命名空间为粒度的资源划分。 **须知** **建议根据需要在命名空间中设置资源配额，避免因资源过载导致集群或节点异常。** 例如：在集群中每个节点可以创建的实例（Pod）数默认为110个，如果您创建的是50节点规格的集群，则最多可以创建5500个实例。因此，您可以在命名空间中自行设置资源配额以确保所有命名空间内的实例总数不超过5500个，以避免资源过载。 请输入整型数值，不输入表示不限制该资源的使用。 若您需要限制CPU或内存的配额，则创建工作负载时必须指定CPU或内存请求值。

步骤4 配置完成后，单击“确定”。

----结束

使用 kubectl 创建 Namespace

使用如下方式定义Namespace。

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

使用kubectl命令创建。

```
$ kubectl create -f custom-namespace.yaml
namespace/custom-namespace created
```

您还可以使用kubectl create namespace命令创建。

```
$ kubectl create namespace custom-namespace
namespace/custom-namespace created
```

14.2 管理命名空间

使用命名空间

- 创建工作负载时，您可以选择对应的命名空间，实现资源或租户的隔离。
- 查询工作负载时，选择对应的命名空间，查看对应命名空间下的所有工作负载。

命名空间使用实践

- **按照不同环境划分命名空间**

  一般情况下，工作负载发布会经历开发环境、联调环境、测试环境，最后到生产环境的过程。这个过程中不同环境部署的工作负载相同，只是在逻辑上进行了定义。分为两种做法：

  - 分别创建不同集群。

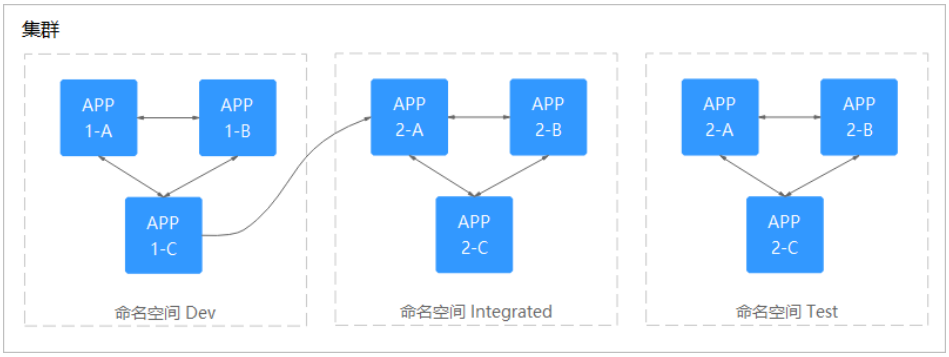
    不同集群之间，资源不能共享。同时，不同环境中的服务互访需要通过负载均衡才能实现。

  - 不同环境创建对应命名空间。

    同个命名空间下，通过服务名称（Service name）可直接访问。跨命名空间的可以通过服务名称、命名空间名称访问。

    例如下图，开发环境/联调环境/测试环境分别创建了命名空间。

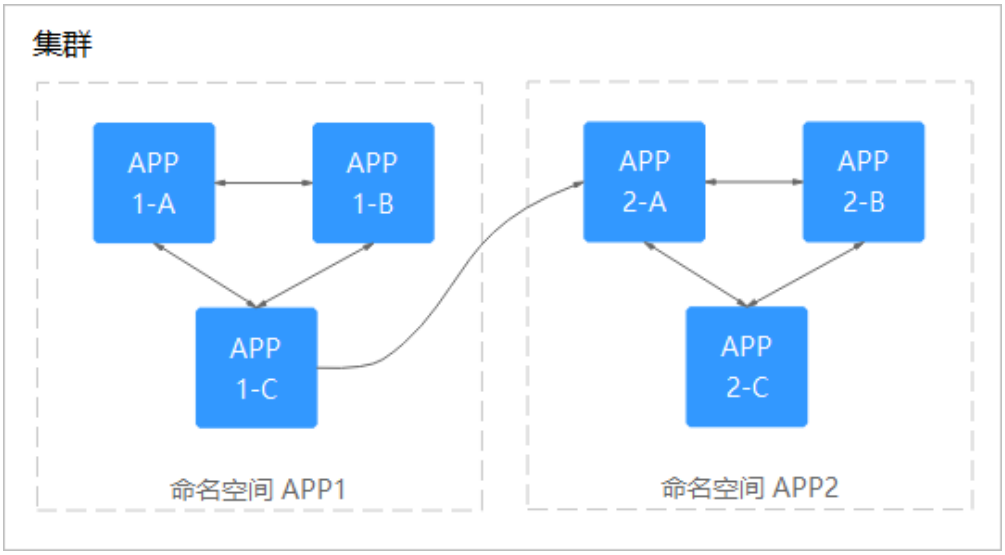
    图 14-1 不同环境创建对应命名空间

- **按照应用划分命名空间**

  对于同个环境中，应用数量较多的情况，建议进一步按照工作负载类型划分命名空间。例如下图中，按照APP1和APP2划分不同命名空间，将不同工作负载在逻辑上当做一个工作负载组进行管理。且同一个命名空间内的工作负载可以通过服务名称访问，不同命名空间下的通过服务名称、命名空间名称访问。

图 14-2 按照工作负载划分命名空间



管理命名空间标签

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧选择“命名空间”。

步骤2 单击目标命名空间操作列的“更多 > 标签管理”。

步骤3 弹出的窗口中将展示命名空间已有的标签，您可根据需要进行修改。

- 添加标签：单击“添加”，填写需要增加标签的“键”和“值”，单击“确定”。
  例如，填写的键为“project”，值为“cicd”，就可以从逻辑概念表示该命名空间是用来部署CICD环境使用。
- 删除标签：单击需要删除标签前的⊖，并单击“确定”。

步骤4 标签修改成功后，再次进入该界面，在“标签”列下可查看到已经修改的标签。

----结束

删除命名空间

删除命名空间会删除该命名空间下所有的资源（如工作负载，短任务、配置项等），请谨慎操作。

步骤1 登录CCE控制台，进入集群。

步骤2 在左侧导航栏中选择“命名空间”，选中待删除的命名空间，单击“更多 > 删除”。

根据系统提示进行删除操作。系统内置的命名空间不支持删除。

----结束

14.3 设置资源配额及限制

通过设置命名空间级别的资源配额，实现多团队或多用户在共享集群资源的情况下限制团队、用户可以使用的资源总量，包括限制命名空间下创建某一类型对象的数量以及对象消耗计算资源（CPU、内存）的总量。

背景信息

默认情况下，运行中的Pod可以无限制的使用Node节点上的CPU和内存，这意味着任意一个Pod都可以无节制地使用集群的计算资源，某个命名空间的Pod可能会耗尽集群的所有资源。

kubernetes在一个物理集群上提供了多个虚拟集群，这些虚拟集群被称为命名空间。命名空间可用于多种工作用途，满足多用户的使用需求，通过为每个命名空间配置资源额度可以有效限制资源滥用，从而保证集群的可靠性。

您可为命名空间配置包括CPU、内存、Pod数量等资源的额度，更多信息请参见**Resource Quotas**。

其中，不同的集群规模对应的Pod数量推荐值如下：

集群规模	Pod数量推荐值
50节点	2500 Pod实例
200节点	1W Pod实例
1000节点	3W Pod实例
2000节点	5W Pod实例

从1.21版本集群开始，如果在**集群配置管理**中开启了enable-resource-quota参数，则创建命名空间将会同时创建默认的资源配额**Resource Quotas**，根据集群规格不同，各个资源的配额如**表14-2**所示。您可以根据实际需求修改。

表 14-2 默认资源配额

集群规模	Pod	Deployment	Secret	ConfigMap	Service
50节点	2000	1000	1000	1000	1000
200节点	2000	1000	1000	1000	1000
1000节点	5000	2000	2000	2000	2000
2000节点	5000	2000	2000	2000	2000

约束与限制

在Kubernetes中，外部用户及内部组件频繁的数据更新操作采用乐观并行的控制方法。通过定义资源版本（resourceVersion）实现乐观锁，资源版本字段包含在对象的元数据（metadata）中。这个字段标识了对象的内部版本号，且对象被修改时，该字段将随之修改。kube-apiserver可以通过该字段判断对象是否已经被修改。当包含resourceVersion的更新请求到达apiserver，服务器端将对比请求数据与服务器中数据的资源版本号，如果不一致，则表明在本次更新提交时，服务端对象已被修改，此时apiserver将返回冲突错误（409）。客户端需重新获取服务端数据，重新修改后再次提交到服务器端；而资源配额对每个命名空间的资源消耗总量提供限制，并且会记录集

群中的资源信息，因此开启资源配额后，在大规模并发场景下创建资源冲突概率会变高，会影响批创资源性能。

操作步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“命名空间”。

步骤3 单击对应命名空间后的“管理配额”。

系统级别的命名空间kube-system、kube-public默认不支持设置资源配额。

步骤4 设置资源配额，然后单击“确定”。

须知

- 命名空间设置了CPU或内存资源配额后，创建工作负载时，必须指定CPU或内存的请求值（request）和约束值（limit），否则CCE将拒绝创建实例。若设置资源配额值为0，则不限制该资源的使用。
- 配额累计使用量包含CCE系统默认创建的资源，如default命名空间下系统默认创建的kubernetes服务（该服务可通过后端kubectl工具查看）等，故建议命名空间下的资源配额略大于实际期望值以去除系统默认创建资源的影响。

----结束

15 配置项与密钥

15.1 创建配置项

操作场景

配置项（ConfigMap）是一种用于存储工作负载所需配置信息的资源类型，内容由用户决定。配置项创建完成后，可在容器工作负载中作为文件或者环境变量使用。

配置项允许您将配置文件从容器镜像中解耦，从而增强容器工作负载的可移植性。

配置项价值如下：

- 使用配置项功能可以帮您管理不同环境、不同业务的配置。
- 方便您部署相同工作负载的不同环境，配置文件支持多版本，方便您进行更新和回滚工作负载。
- 方便您快速将您的配置以文件的形式导入到容器中。

约束与限制

- ConfigMap资源文件大小不得超过2MB。
- **静态Pod**中不可使用ConfigMap。

操作步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“配置项与密钥”，在右上角单击“创建配置项”。

步骤3 填写参数。

表 15-1 新建配置参数说明

参数	参数说明
名称	新建的配置项名称，同一个命名空间里命名必须唯一。
命名空间	新建配置项所在的命名空间。若不选择，默认为default。

参数	参数说明
描述	配置项的描述信息。
配置数据	配置项的数据。 键值对形式，单击 ┼ 添加。其中值支持String、JSON和YAML格式。
标签	配置项的标签。键值对形式，输入键值对后单击“添加”。

步骤4 配置完成后，单击“确定”。

工作负载配置列表中会出现新创建的工作负载配置。

----结束

使用 kubectl 创建配置项

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑cce-configmap.yaml文件。

vi cce-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cce-configmap
data:
  SPECIAL_LEVEL: Hello
  SPECIAL_TYPE: CCE
```

表 15-2 关键参数说明

参数	说明
apiVersion	固定值为v1。
kind	固定值为ConfigMap。
metadata.name	配置项名称，可自定义。
data	配置项的数据，需填写键值对形式。

步骤3 创建配置项。

kubectl create -f cce-configmap.yaml

查看已创建的配置项。

kubectl get cm

```
NAME              DATA      AGE
cce-configmap     3         7m
```

----结束

相关操作

配置项创建完成后，您还可以执行表15-3中的操作。

表 15-3 其他操作

操作	说明
编辑YAML	单击配置项名称后的“编辑YAML”，可编辑当前配置项的YAML文件。
更新配置	1. 选择需要更新的配置项名称，单击“更新”。 2. 根据表15-1更改信息。 3. 单击“确定”。
删除配置	选择要删除的配置项，单击“删除”。 根据系统提示删除配置。

15.2 使用配置项

配置项创建后，可在工作负载环境变量、命令行参数和数据卷三个场景使用。

- 通过配置项设置工作负载环境变量
- 通过配置项设置命令行参数
- 使用配置项挂载到工作负载数据卷

本节以下面这个ConfigMap为例，具体介绍ConfigMap的用法。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cce-configmap
data:
  SPECIAL_LEVEL: Hello
  SPECIAL_TYPE: CCE
```

须知

- 在工作负载里使用ConfigMap时，需要工作负载和ConfigMap处于同一集群和命名空间中。
- 以数据卷挂载使用ConfigMap时，当ConfigMap被更新，Kubernetes会同时更新数据卷中的数据。

  对于以subPath形式挂载的ConfigMap数据卷，当ConfigMap被更新时，Kubernetes无法自动更新数据卷中的数据。
- 以环境变量方式使用ConfigMap时，当ConfigMap被更新，数据不会被自动更新。更新这些数据需要重新启动Pod。

通过配置项设置工作负载环境变量

使用控制台方式

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，单击右上角“创建负载”。

在创建工作负载时，在“容器配置”中找到“环境变量”，单击。

- **配置项导入**：选择一个配置项，将配置项中所有键值都导入为环境变量。
- **配置项键值导入**：将配置项中某个键的值导入作为某个环境变量的值。
  - 变量名称：工作负载中的环境变量名称，可自定义，默认为配置项中选择的键名。
  - 变量/变量引用：选择一个配置项及需要导入的键名，将其对应的值导入为工作负载环境变量。

  例如将cce-configmap这个配置项中“SPECIAL_LEVEL”的值“Hello”导入，作为工作负载环境变量“SPECIAL_LEVEL”的值，导入后容器中有一个名为“SPECIAL_LEVEL”的环境变量，其值为“Hello”。

步骤3 配置其他工作负载参数后，单击“创建工作负载”。

等待工作负载正常运行后，您可登录容器执行以下语句，查看该配置项是否已被设置为工作负载的环境变量。

```
printenv SPECIAL_LEVEL
```

示例输出如下：

```
Hello
```

----结束

使用kubectl方式

步骤1 请参见通过kubectl连接集群配置kubectl命令。

步骤2 创建并编辑nginx-configmap.yaml文件。

vi nginx-configmap.yaml

YAML文件内容如下：

- **配置项导入**：如果要将一个配置项中所有数据都添加到环境变量中，可以使用envFrom参数，配置项中的Key会成为工作负载中的环境变量名称。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-configmap
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-configmap
  template:
    metadata:
      labels:
        app: nginx-configmap
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        envFrom:                    # 使用envFrom来指定环境变量引用的配置项
        - configMapRef:
            name: cce-configmap     # 引用的配置项名称
      imagePullSecrets:
      - name: default-secret
```

- **配置项键值导入**：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参数单独引用ConfigMap中的Key/Value。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-configmap
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-configmap
  template:
    metadata:
      labels:
        app: nginx-configmap
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        env:                                # 设置工作负载中的环境变量
        - name: SPECIAL_LEVEL               # 工作负载中的环境变量名称
          valueFrom:                        # 使用valueFrom来指定环境变量引用配置项
            configMapKeyRef:
              name: cce-configmap           # 引用的配置项名称
              key: SPECIAL_LEVEL            # 引用的配置项中的key
        - name: SPECIAL_TYPE                # 添加多个环境变量参数，可同时导入多个环境变量
          valueFrom:
            configMapKeyRef:
              name: cce-configmap
              key: SPECIAL_TYPE
      imagePullSecrets:
      - name: default-secret
```

步骤3 创建工作负载。

kubectl apply -f nginx-configmap.yaml

步骤4 创建完成后，查看Pod中的环境变量。

1. 执行以下命令，查看已创建的Pod。

   ```
   kubectl get pod | grep nginx-configmap
   ```

   预期输出如下：

   ```
   nginx-configmap-***   1/1     Running   0          2m18s
   ```

2. 执行以下命令，查看该Pod中的环境变量。

   ```
   kubectl exec nginx-configmap-*** -- printenv SPECIAL_LEVEL SPECIAL_TYPE
   ```

   预期输出如下：

   ```
   Hello
   CCE
   ```

   说明该配置项已被设置为工作负载的环境变量。

----结束

通过配置项设置命令行参数

您可以使用配置项作为环境变量来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。

使用控制台方式

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，单击右上角“创建负载”。

在创建工作负载时，在“容器配置”中找到“环境变量”，单击 。本例中以“配置项导入”为例。

- **配置项导入**：选择一个配置项，将配置项中所有键值都导入为环境变量。

步骤3 在“容器配置”中找到“生命周期”，在右侧选择“启动后处理”页签，并填写以下参数。

- 处理方式：命令行脚本。
- 执行命令：以下命令需分三行填写，其中*SPECIAL_LEVEL*和*SPECIAL_TYPE*为工作负载中的环境变量名，即cce-configmap配置项中的键名。

```
/bin/bash
-c
echo $SPECIAL_LEVEL $SPECIAL_TYPE > /usr/share/nginx/html/index.html
```

步骤4 配置其他工作负载参数后，单击“创建工作负载”。

等待工作负载正常运行后，您可**登录容器**执行以下语句，查看该配置项是否已被设置为工作负载的环境变量。

```
cat /usr/share/nginx/html/index.html
```

示例输出如下：

```
Hello CCE
```

----结束

使用kubectl方式

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑nginx-configmap.yaml文件。

vi nginx-configmap.yaml

如下面的示例所示，在工作负载中导入了cce-configmap配置项，其中*SPECIAL_LEVEL*和*SPECIAL_TYPE*为工作负载中的环境变量名，即cce-configmap配置项中的键名。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-configmap
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-configmap
  template:
    metadata:
      labels:
        app: nginx-configmap
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        lifecycle:
          postStart:
            exec:
              command: [ "/bin/sh", "-c", "echo $SPECIAL_LEVEL $SPECIAL_TYPE > /usr/share/nginx/html/
index.html" ]
        envFrom:                    # 使用envFrom来指定环境变量引用的配置项
        - configMapRef:
            name: cce-configmap     # 引用的配置项名称
      imagePullSecrets:
        - name: default-secret
```

步骤3 创建工作负载。

kubectl apply -f nginx-configmap.yaml

步骤4 等待工作负载正常运行后，容器中的/usr/share/nginx/html/index.html文件将被输入如下内容。

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep nginx-configmap
   ```
   预期输出如下：
   ```
   nginx-configmap-***   1/1     Running   0          2m18s
   ```
2. 执行以下命令，查看该Pod中的环境变量。
   ```
   kubectl exec nginx-configmap-*** -- cat /usr/share/nginx/html/index.html
   ```
   预期输出如下：
   ```
   Hello CCE
   ```

----结束

使用配置项挂载到工作负载数据卷

配置项(ConfigMap)挂载是将配置项中的数据挂载到指定的容器路径。平台提供工作负载代码和配置文件的分离，“配置项挂载”用于处理工作负载配置参数。用户需要提前创建工作负载配置，操作步骤请参见**创建配置项**。

使用控制台方式

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，单击右上角“创建负载”。

在创建工作负载时，在“容器配置”中找到“数据存储”，选择“添加存储卷 > 配置项(ConfigMap)”。

步骤3 配置参数，如**表15-4**。

表 15-4 配置项挂载

参数	参数说明
配置项	选择对应的配置项名称。 配置项需要提前创建，具体请参见**创建配置项**。

参数	参数说明
添加容器挂载	配置如下参数： 1. 挂载路径：请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。 2. 子路径：请输入子路径，如：tmp。 – 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume，不填写时默认为根。 – 子路径可以填写ConfigMap/Secret的键值，子路径若填写为不存在的键值则数据导入不会生效。 – 通过子路径导入的数据不会随ConfigMap/Secret的更新而动态更新。 3. 设置权限：只读。只能读容器路径中的数据卷。 单击+可增加多条设置。

----结束

使用kubectl方式

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑nginx-configmap.yaml文件。

vi nginx-configmap.yaml

如下面的示例所示，配置项挂载完成后，最终会在容器中的/etc/config目录下生成以配置项中的key为文件名， value为文件内容的配置文件。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-configmap
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-configmap
  template:
    metadata:
      labels:
        app: nginx-configmap
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config          # 挂载到/etc/config目录下
          readOnly: true
    volumes:
```

```
  - name: config-volume
    configMap:
      name: cce-configmap              # 引用的配置项名称
```

步骤3 创建工作负载。

kubectl apply -f nginx-configmap.yaml

步骤4 等待工作负载正常运行后，在/etc/config目录下会生成SPECIAL_LEVEL和SPECIAL_TYPE两个文件，且文件的内容分别为Hello和CCE。

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep nginx-configmap
   ```
   预期输出如下：
   ```
   nginx-configmap-***   1/1     Running   0          2m18s
   ```
2. 执行以下命令，查看该Pod中的SPECIAL_LEVEL或SPECIAL_TYPE文件。
   ```
   kubectl exec nginx-configmap-*** -- /etc/config/SPECIAL_LEVEL
   ```
   预期输出如下：
   ```
   Hello
   ```

----结束

15.3 创建密钥

操作场景

密钥（Secret）是一种用于存储工作负载所需要认证信息、密钥的敏感信息等的资源类型，内容由用户决定。资源创建完成后，可在容器工作负载中作为文件或者环境变量使用。

约束与限制

静态Pod中不可使用Secret。

操作步骤

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“配置项与密钥”，选择“密钥”页签，在右上角单击“创建密钥”。

步骤3 填写参数。

表 15-5 基本信息说明

参数	参数说明
名称	新建的密钥的名称，同一个命名空间内命名必须唯一。
命名空间	新建密钥所在的命名空间，默认为default。
描述	密钥的描述信息。

参数	参数说明
密钥类型	新建的密钥类型。 • Opaque：一般密钥类型。 • kubernetes.io/dockerconfigjson：存放拉取私有仓库镜像所需的认证信息。 • kubernetes.io/tls：Kubernetes的TLS密钥类型，用于存放7层负载均衡服务所需的证书。kubernetes.io/tls类型的密钥示例及说明请参见**TLS Secret**。 • IngressTLS：CCE提供的TLS密钥类型，用于存放7层负载均衡服务所需的证书。 • 其他：若需要创建其他类型的密钥，请手动输入密钥类型。
密钥数据	工作负载密钥的数据可以在容器中使用。 • 当密钥为Opaque类型时，单击+，在弹出的窗口中输入键值对，并且可以勾选“自动Base64转码”。 • 当密钥为kubernetes.io/dockerconfigjson类型时，输入私有镜像仓库的账号和密码。 • 当密钥为kubernetes.io/tls或IngressTLS类型时，上传证书文件和私钥文件。 **说明** – 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。
密钥标签	密钥的标签。键值对形式，输入键值对后单击“添加”。

步骤4 配置完成后，单击“确定”。

密钥列表中会出现新创建的密钥。

----结束

Secret 资源文件配置示例

本章节主要介绍Secret类型的资源描述文件的配置示例。

- Opaque类型

  定义的Secret文件secret.yaml内容如下。其中data字段以键值对的形式填写，value需要用Base64编码，Base64编码方法请参见**如何进行Base64编码**。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret          # secret的名称
  namespace: default      #命名空间，默认为default
data:
  <your_key>: <your_value>  #填写键值对，其中value需要用Base64编码
type: Opaque
```

- kubernetes.io/dockerconfigjson类型

  定义的Secret文件secret.yaml内容如下。其中.dockerconfigjson需要用Base64，Base64编码方法请参见**如何进行Base64编码**。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret          # secret的名称
  namespace: default      #命名空间，默认为default
data:
  .dockerconfigjson: eyJh*****    #Base64编码后的内容
type: kubernetes.io/dockerconfigjson
```

获取.dockerconfigjson内容的步骤如下：

a. 获取镜像仓库的登录信息：

- 镜像仓库地址：本文中以address为例，请根据实际信息替换。
- 用户名：本文中以username为例，请根据实际信息替换。
- 密码：本文中以password为例，请根据实际信息替换。

b. 使用Base64将键值对username:password进行编码，获取编码后的内容填入3中。

```
echo -n "username:password" | base64
```

回显如下：

```
dXNlcm5hbWU6cGFzc3dvcmQ=
```

c. 使用Base64对以下JSON内容进行编码。

```
echo -n '{"auths":{"address":
{"username":"username","password":"password","auth":"dXNlcm5hbWU6cGFzc3dvcmQ="}}}'
| base64
```

回显如下：

```
eyJhdXRocyI6eyJhZGRyZXNzIjp7InVzZXJuYW1lIjoidXNlcm5hbWUiLCJwYXNzd29yZCI6InBhc3N3b3JkIiwiYXV0aCI6ImRYTmxjbTVoYldVNmNHRnpjM2R2Y21RPSJ9fX0=
```

编码后的内容即为.dockerconfigjson内容。

- kubernetes.io/tls类型

  其中tls.crt和tls.key需要用Base64，Base64编码方法请参见**如何进行Base64编码**。

```
kind: Secret
apiVersion: v1
metadata:
  name: mysecret          # secret的名称
  namespace: default      #命名空间，默认为default
data:
  tls.crt: LS0tLS1CRU*****FURS0tLS0t  #证书内容，需要Base64编码
  tls.key: LS0tLS1CRU*****VZLS0tLS0=  #私钥内容，需要Base64编码
type: kubernetes.io/tls
```

- IngressTLS类型

  其中tls.crt和tls.key需要用Base64，Base64编码方法请参见**如何进行Base64编码**。

```
kind: Secret
apiVersion: v1
metadata:
  name: mysecret          # secret的名称
  namespace: default      #命名空间，默认为default
data:
  tls.crt: LS0tLS1CRU*****FURS0tLS0t  #证书内容，需要Base64编码
  tls.key: LS0tLS1CRU*****VZLS0tLS0=  #私钥内容，需要Base64编码
type: IngressTLS
```

使用 kubectl 创建密钥

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 通过Base64编码，创建并编辑cce-secret.yaml文件。

```
# echo -n "待编码内容" | base64
******
```

vi cce-secret.yaml

Opaque类型的YAML示例如下，其余类型请参见**Secret资源文件配置示例**：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  <your_key>: <your_value>  #填写键值对，其中value需要用Base64编码
```

步骤3 创建密钥。

kubectl create -f cce-secret.yaml

创建完成后可以查询到密钥。

kubectl get secret -n default

----结束

相关操作

密钥创建完成后，您还可以执行**表15-6**中的操作。

说明

密钥列表中包含系统密钥资源，系统密钥资源不可更新，也不能删除，只能查看。

表 15-6 其他操作

操作	说明
编辑YAML	单击密钥名称后的“编辑YAML”，可编辑当前密钥的YAML文件。
更新密钥	1. 选择需要更新的密钥名称，单击“更新”。 2. 根据**表15-5**更改信息。 3. 单击“确定”。
删除密钥	选择要删除的密钥，单击“删除”。 根据系统提示删除密钥。
批量删除密钥	1. 勾选需要删除的密钥名称。 2. 单击页面左上角的“批量删除”，删除选中的密钥。 3. 根据系统提示删除密钥。

如何进行 Base64 编码

对字符串进行Base64编码，可以直接使用“echo -n 要编码的内容 | base64”命令即可，示例如下：

```
root@ubuntu:~# echo -n "待编码内容" | base64
******
```

15.4 使用密钥

密钥创建后，可在工作负载环境变量和数据卷两个场景使用。

须知

请勿对以下CCE系统使用的密钥做任何操作，详情请参见**集群系统密钥说明**。

- 请不要操作kube-system下的secrets。
- 请不要操作任何命名空间下的default-secret、paas.elb。其中，default-secret用于SWR的私有镜像拉取，paas.elb用于该命名空间下的服务对接ELB。

- **使用密钥设置工作负载的环境变量**
- **使用密钥配置工作负载的数据卷**

本节以下面这个Secret为例，具体介绍Secret的用法。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: ******   #需要用Base64编码
  password: ******   #需要用Base64编码
```

须知

- 在Pod里使用密钥时，需要Pod和密钥处于同一集群和命名空间中。
- 当Secret 被更新时，Kubernetes会同时更新数据卷中的数据。

  但对于以**subPath**形式挂载的Secret数据卷，当Secret 被更新时，Kubernetes无法自动更新数据卷中的数据。

使用密钥设置工作负载的环境变量

使用控制台方式

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，单击右上角“创建负载”。

在创建工作负载时，在“容器配置”中找到“环境变量”，单击。

- **密钥导入**：选择一个密钥，将密钥中所有键值都导入为环境变量。
- **密钥项键值导入**：将密钥中某个键的值导入作为某个环境变量的值。
  - 变量名称：工作负载中的环境变量名称，可自定义，默认为密钥中选择的键名。

- 变量/变量引用：选择一个密钥及需要导入的键名，将其对应的值导入为工作负载环境变量。

例如将mysecret这个密钥中“username”的值导入，作为工作负载环境变量“username”的值，导入后容器中将会有一个名为“username”的环境变量。

步骤3 配置其他工作负载参数后，单击“创建工作负载”。

等待工作负载正常运行后，您可**登录容器**执行以下语句，查看该密钥是否已被设置为工作负载的环境变量。

```
printenv username
```

如输出与Secret中的内容一致，则说明该密钥已被设置为工作负载的环境变量。

----结束

使用kubectl方式

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑nginx-secret.yaml文件。

vi nginx-secret.yaml

YAML文件内容如下：

- **密钥导入**：如果要将一个密钥中所有数据都添加到环境变量中，可以使用envFrom参数，密钥中的Key会成为工作负载中的环境变量名称。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-secret
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-secret
  template:
    metadata:
      labels:
        app: nginx-secret
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        envFrom:                # 使用envFrom来指定环境变量引用的密钥
        - secretRef:
            name: mysecret      # 引用的密钥名称
      imagePullSecrets:
      - name: default-secret
```

- **密钥键值导入**：您可以在创建工作负载时将密钥设置为环境变量，使用valueFrom参数单独引用Secret中的Key/Value。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-secret
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-secret
  template:
    metadata:
      labels:
        app: nginx-secret
```

```
spec:
  containers:
  - name: container-1
    image: nginx:latest
    env:                                # 设置工作负载中的环境变量
    - name: SECRET_USERNAME             # 工作负载中的环境变量名称
      valueFrom:                        # 使用valueFrom来指定环境变量引用的密钥
        secretKeyRef:
          name: mysecret                # 引用的密钥名称
          key: username                 # 引用的密钥中的key
    - name: SECRET_PASSWORD             # 添加多个环境变量参数，可同时导入多个环境变量
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: password
  imagePullSecrets:
  - name: default-secret
```

步骤3 创建工作负载。

kubectl apply -f nginx-secret.yaml

步骤4 创建完成后，查看Pod中的环境变量。

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep nginx-secret
   ```
   预期输出如下：
   ```
   nginx-secret-***   1/1     Running   0          2m18s
   ```
2. 执行以下命令，查看该Pod中的环境变量。
   ```
   kubectl exec nginx-secret-*** -- printenv SPECIAL_USERNAME SPECIAL_PASSWORD
   ```
   如输出与Secret中的内容一致，则说明该密钥已被设置为工作负载的环境变量。

----结束

使用密钥配置工作负载的数据卷

密钥(Secret)挂载将密钥中的数据挂载到指定的容器路径，密钥内容由用户决定。用户需要提前创建密钥，操作步骤请参见**创建密钥**。

使用控制台方式

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏选择“工作负载”，在右侧选择“无状态负载”页签。单击右上角“创建负载”。

在创建工作负载时，在“容器配置”中找到“数据存储”，选择“添加存储卷 > 密钥(Secret)”。

步骤3 配置参数，如**表15-7**。

表 15-7 密钥挂载

参数	参数说明
密钥	选择对应的密钥名称。 密钥需要提前创建，具体请参见**创建密钥**。

参数	参数说明
添加容器挂载	配置如下参数： 1. 挂载路径：请输入挂载路径，如：/tmp。 数据存储挂载到容器上的路径。请不要挂载在系统目录下，如“/”、“/var/run”等，会导致容器异常。建议挂载在空目录下，若目录不为空，请确保目录下无影响容器启动的文件，否则文件会被替换，导致容器启动异常，工作负载创建失败。 **须知** 挂载高危目录的情况下，建议使用低权限账号启动，否则可能会造成宿主机高危文件被破坏。 2. 子路径：请输入子路径，如：tmp。 – 使用子路径挂载本地磁盘，实现在单一Pod中重复使用同一个Volume，不填写时默认为根。 – 子路径可以填写ConfigMap/Secret的键值，子路径若填写为不存在的键值则数据导入不会生效。 – 通过子路径导入的数据不会随ConfigMap/Secret的更新而动态更新。 3. 设置权限：只读。只能读容器路径中的数据卷。 单击+可增加多条设置。

----结束

使用kubectl方式

步骤1 请参见**通过kubectl连接集群**配置kubectl命令。

步骤2 创建并编辑nginx-secret.yaml文件。

vi nginx-secret.yaml

如下面的示例所示，mysecret密钥的username和password以文件方式保存在/etc/foo目录下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-secret
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-secret
  template:
    metadata:
      labels:
        app: nginx-secret
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
       - name: foo
          mountPath: /etc/foo        # 挂载到/etc/foo目录下
          readOnly: true
    volumes:
```

```
  - name: foo
    secret:
      secretName: mysecret      # 引用的密钥名称
```

您还可以使用items字段控制Secret键的映射路径，例如，将username存放在容器中的/etc/foo/my-group/my-username目录下。

说明

- 使用items字段指定Secret键的映射路径后，没有被指定的键将不会被以文件形式创建。例如，下面的例子中的password键未被指定，则该文件将不会被创建。
- 如果要使用Secret中全部的键，那么必须将全部的键都列在items字段中。
- items字段中列出的所有键必须存在于相应的Secret 中。否则，该卷不被创建。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-secret
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-secret
  template:
    metadata:
      labels:
        app: nginx-secret
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
       - name: foo
          mountPath: /etc/foo          # 挂载到/etc/foo目录下
          readOnly: true
    volumes:
    - name: foo
      secret:
        secretName: mysecret      # 引用的密钥名称
        items:
        - key: username      # 引用的密钥中的键名
          path: my-group/my-username    # Secret键的映射路径
```

步骤3 创建工作负载。

kubectl apply -f nginx-secret.yaml

步骤4 等待工作负载正常运行后，在/etc/foo目录下会生成username和password两个文件。

1. 执行以下命令，查看已创建的Pod。
   ```
   kubectl get pod | grep nginx-secret
   ```
   预期输出如下：
   ```
   nginx-secret-***   1/1     Running   0          2m18s
   ```
2. 执行以下命令，查看该Pod中的username或password文件。
   ```
   kubectl exec nginx-secret-*** -- /etc/foo/username
   ```
   预期输出与Secret中的内容一致。

----结束

15.5 集群系统密钥说明

CCE默认会在每个命名空间下创建如下密钥。

- default-secret
- paas.elb
- default-token-xxxxx（xxxxx为随机数）

下面将详细介绍这个几个密钥的用途。

default-secret

default-secret的类型为kubernetes.io/dockerconfigjson，其data内容是登录SWR镜像仓库的凭据，用于从SWR拉取镜像。在CCE中创建工作负载时如果需要从SWR拉取镜像，需要配置imagePullSecrets的取值为default-secret，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
  imagePullSecrets:
  - name: default-secret
```

default-secret的data数据会定期更新，且当前的data内容会在一定时间后会过期失效。您可以使用describe命令在default-secret的中查看到具体的过期时间，如下所示。

须知

在使用时请直接使用default-secret，而不要拷贝secret内容重新创建，因为secret里面的凭据会过期，从而导致无法拉取镜像。

```
$ kubectl describe secret default-secret
Name:         default-secret
Namespace:    default
Labels:       secret-generated-by=cce
Annotations:  temporary-ak-sk-expires-at: 2021-11-26 20:55:31.380909 +0000 UTC

Type:  kubernetes.io/dockerconfigjson

Data
====
.dockerconfigjson:  347 bytes
```

paas.elb

paas.elb的data内容是临时AK/SK数据，用于创建Service和Ingress时创建ELB，paas.elb的data数据同样会定期更新，且在一定时间后会过期失效。

实际使用中您不会直接使用paas.elb，但请不要删除paas.elb，否则会导致创建ELB失败。

default-token-xxxxx

Kubernetes为每个命名空间默认创建一个名为default的ServiceAccount，default-token-xxxxx为这个ServiceAccount的密钥，xxxxx是随机数。

```
$ kubectl get sa
NAME     SECRETS   AGE
default  1         30d
$ kubectl describe sa default
Name:                default
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   default-token-xxxxx
Tokens:              default-token-xxxxx
Events:              <none>
```

16 弹性伸缩

16.1 弹性伸缩概述

弹性伸缩是根据业务需求和策略，经济地自动调整弹性计算资源的管理服务。

背景介绍

随着Kubernetes已经成为云原生应用编排、管理的事实标准，越来越多的应用选择向Kubernetes迁移，用户也越来越关心在Kubernetes上应用如何快速扩容面对业务高峰，以及如何在业务低谷时快速缩容节约资源与成本。

在Kubernetes的集群中，“弹性伸缩”一般涉及到扩缩容Pod个数以及Node个数。Pod代表应用的实例数（每个Pod包含一个或多个容器），当业务高峰的时候需要扩容应用的实例个数。所有的Pod都是运行在某一个节点（虚机或裸机）上，当集群中没有足够多的节点来调度新扩容的Pod，那么就需要为集群增加节点，从而保证业务能够正常提供服务。

弹性伸缩在CCE上的使用场景非常广泛，典型的场景包含在线业务弹性、大规模计算训练、深度学习GPU或共享GPU的训练与推理、定时周期性负载变化等。

CCE 弹性伸缩

CCE的弹性伸缩能力分为如下两个维度：

- **工作负载弹性伸缩**：即调度层弹性，主要是负责修改负载的调度容量变化。例如，HPA是典型的调度层弹性组件，通过HPA可以调整应用的副本数，调整的副本数会改变当前负载占用的调度容量，从而实现调度层的伸缩。
- **节点弹性伸缩**：即资源层弹性，主要是集群的容量规划不能满足集群调度容量时，会通过弹出ECS资源的方式进行调度容量的补充。

组件介绍

工作负载弹性组件介绍

表 16-1 工作负载弹性组件

类型	组件名称	组件介绍	参考文档
HPA	Kubernetes Metrics Server	Kubernetes内置组件，实现Pod水平自动伸缩的功能，即Horizontal Pod Autoscaling。在kubernetes社区HPA功能的基础上，增加了应用级别的冷却时间窗和扩缩容阈值等功能。	HPA策略

节点弹性伸缩组件介绍

表 16-2 节点弹性组件

组件名称	组件介绍	适用场景	参考文档
CCE Cluster Autoscaler	Kubernetes社区开源组件，节点水平伸缩组件，提供了独有的调度、弹性优化、成本优化的功能。	全场景支持，适合在线业务、深度学习、大规模成本算力交付等。	节点自动伸缩

16.2 工作负载弹性伸缩

16.2.1 工作负载伸缩原理

HPA 工作原理

HPA（Horizontal Pod Autoscaler）是用来控制Pod水平伸缩的控制器，HPA周期性检查Pod的度量数据，计算满足HPA资源所配置的目标数值所需的副本数量，进而调整目标资源（如Deployment）的replicas字段。

想要做到自动弹性伸缩，先决条件就是能感知到各种运行数据，例如集群节点、Pod、容器的CPU、内存使用率等等。而这些数据的监控能力Kubernetes也没有自己实现，而是通过其他项目来扩展Kubernetes的能力，CCE提供Metrics Server插件来实现该能力：

- Metrics Server是Kubernetes集群范围资源使用数据的聚合器。Metrics Server从kubelet公开的Summary API中采集度量数据，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据，且对外提供一套标准的API。

使用HPA（Horizontal Pod Autoscaler）配合Metrics Server可以实现基于CPU和内存的自动弹性伸缩。

HPA的核心有如下2个部分：

- 监控数据来源

  最早社区只提供基于CPU和Mem的HPA，随着应用越来越多搬迁到K8s上，开发者已经不满足于CPU和Memory，开发者需要应用自身的业务指标，或者是一些接

入层的监控信息，例如：Load Balancer的QPS、网站的实时在线人数等。社区经过思考之后，定义了一套标准的Metrics API，通过聚合API对外提供服务。

- metrics.k8s.io： 主要提供Pod和Node的CPU和Memory相关的监控指标。
- custom.metrics.k8s.io： 主要提供Kubernetes Object相关的自定义监控指标。
- external.metrics.k8s.io：指标来源外部，与任何的Kubernetes资源的指标无关。

- 扩缩容决策算法

  HPA controller根据当前指标和期望指标来计算缩放比例，计算公式如下：

  desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

  例如当前的指标值是200m，目标值是100m，那么按照公式计算期望的实例数就会翻倍。那么在实际过程中，可能会遇到实例数值反复伸缩，导致集群震荡。为了保证稳定性，HPA controller从以下几个方面进行优化：

  - 冷却时间：在1.11版本以及之前的版本，社区引入了horizontal-pod-autoscaler-downscale-stabilization-window和horizontal-pod-autoScaler-upscale-stabilization-window这两个启动参数代表缩容冷却时间和扩容冷却时间，这样保证在冷却时间内，跳过扩缩容。1.14版本之后引入延迟队列，保存一段时间内每一次检测的决策建议，然后根据当前所有有效的决策建议来进行决策，从而保证期望的副本数尽量小的发生变更，保证稳定性。
  - 忍受度：可以看成一个缓冲区，当实例变化范围在忍受范围之内的话，保持原有的实例数不变。

    首先定义ratio = currentMetricValue / desiredMetricValue

    当|ratio – 1.0| <= tolerance时，则会忽略，跳过scale。

    当|ratio – 1.0| > tolerance时， 就会根据之前的公式计算期望值。

    当前社区版本中默认值为0.1

HPA是基于指标阈值进行伸缩的，常见的指标主要是 CPU、内存，也可以通过自定义指标，例如QPS、连接数等进行伸缩。但是存在一个问题：基于指标的伸缩存在一定的时延，这个时延主要包含：采集时延(分钟级) + 判断时延(分钟级) + 伸缩时延(分钟级)。这个分钟级的时延，可能会导致应用CPU飚高，响应时间变慢。为了解决这个问题，CCE提供了定时策略，对于一些有周期性变化的应用，提前扩容资源，而业务低谷时，定时回收资源。

16.2.2 HPA 策略

HPA策略即Horizontal Pod Autoscaling，是Kubernetes中实现POD水平自动伸缩的功能。该策略在Kubernetes社区HPA功能的基础上，增加了应用级别的冷却时间窗和扩缩容阈值等功能。

前提条件

使用HPA需要安装能够提供Metrics API的插件，您可根据集群版本和实际需求选择其中之一：

- **Kubernetes Metrics Server**：提供基础资源使用指标，例如容器CPU和内存使用率。所有集群版本均可安装。

约束与限制

- HPA策略：仅支持1.13及以上版本的集群创建。
- 1.19.10以下版本的集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，当新Pod被调度到另一个节点时，会导致之前Pod不能正常读写。

  1.19.10及以上版本集群中，如果使用HPA策略对挂载了EVS卷的负载进行扩容，新Pod会因为无法挂载云硬盘导致无法成功启动。

创建 HPA 策略

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“负载伸缩”，在右上角单击“创建HPA策略”。

步骤3 填写HPA策略参数。

表 16-3 HPA 策略参数配置

参数	参数说明
策略名称	新建策略的名称，请自定义。
命名空间	请选择工作负载所在的命名空间。
关联工作负载	请选择要设置HPA策略的工作负载。
实例范围	请输入最小实例数和最大实例数。 策略触发时，工作负载实例将在此范围内伸缩。
冷却时间	请输入缩容和扩容的冷却时间，单位为分钟，**缩容扩容冷却时间不能小于1分钟**。 **该设置仅在1.15到1.23版本的集群中显示。** 策略成功触发后，在此缩容/扩容冷却时间内，不会再次触发缩容/扩容，目的是等待伸缩动作完成后在系统稳定且集群正常的情况下进行下一次策略匹配。
伸缩配置	**该设置仅在1.25及以上版本的集群中显示。** ● 系统默认：采用社区推荐的默认行为进行负载伸缩，详情请参见**社区默认行为说明**。 ● 自定义：自定义扩/缩容配置的稳定窗口、步长、优先级等策略，实现更灵活的配置。未配置的参数将采用社区推荐的默认值。 – 禁止扩/缩容：选择是否禁止扩容或缩容。 – 稳定窗口：需要伸缩时，会在一段时间（设定的稳定窗口值）内持续检测，如在该时间段内始终需要进行伸缩（不满足设定的指标期望值）才进行伸缩，避免短时间的指标抖动造成异常。 – 步长策略：扩/缩容的步长，可设置一定时间内扩/缩容Pod数量或百分比。在存在多条策略时，可以选择使Pod数量最多或最少的策略。

参数	参数说明
系统策略	• 指标：可选择“CPU利用率”或“内存利用率”。 **说明** 利用率 = 工作负载容器组（Pod）的实际使用量 / 申请量 • 期望值：请输入期望资源平均利用率。 期望值表示所选指标的期望值，通过向上取整（当前指标值 / 期望值 × 当前实例数）来计算需要伸缩的实例数。 **说明** HPA在计算扩容、缩容实例数时，会选择最近5分钟内实例数的最大值。 • 容忍范围：指标处于范围内时不会触发伸缩，期望值必须在容忍范围之间。 当指标值大于缩容阈值且小于扩容阈值时，不会触发扩容或缩容。**阈值仅在1.15及以上版本的集群中支持**。
自定义策略（仅在1.15及以上版本的集群中支持）	**说明** 使用自定义策略时，集群中需要安装支持采集自定义指标的插件（例如Prometheus），且工作负载需正常上报并采集自定义指标。 • 自定义指标名称：自定义指标的名称，输入时可根据联想值进行选择。 • 指标来源：在下拉框中选择对象类型，可选择“Pod”。 • 期望值：Pod支持指标为平均值。通过向上取整（当前指标值 / 期望值 × 当前实例数）来计算需要伸缩的实例数。 **说明** HPA在计算扩容、缩容实例数时，会选择最近5分钟内实例数的最大值。 • 容忍范围：指标处于范围内时不会触发伸缩，期望值必须在容忍范围之间。

步骤4 设置完成后，单击“创建”。

----结束

16.2.3 管理工作负载伸缩策略

操作场景

HPA策略创建完成后，可对创建的策略进行更新、克隆、编辑YAML以及删除等操作。

查看 HPA 策略

您可以查看HPA策略的规则、状态和事件，参照界面中的报错提示有针对性的解决异常事件。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中单击“负载伸缩”，在“HPA策略”页签下，单击要查看的HPA策略前方的 ⌄ 。

步骤3 在展开的区域中，可以看到规则、状态页签。单击策略操作区域的“事件”，可以看到事件页签。若策略异常，请参照界面中的报错提示进行定位处理。

说明

您还可以在工作负载详情页中查看已创建的HPA策略：

1. 登录CCE控制台，单击集群名称进入集群。
2. 在左侧导航栏中单击“工作负载”，单击工作负载名称查看详情。
3. 在该工作负载详情页的“弹性伸缩”页签下可以看到HPA策略，您在“负载伸缩”页面配置的伸缩策略也会在这里显示。

表 16-4 事件类型及名称

事件类型	事件名称	描述
正常	SuccessfulRescale	扩缩容成功
异常	InvalidTargetRange	无效的TargetRange
	InvalidSelector	无效选择器
	FailedGetObjectMetric	获取对象失败数
	FailedGetPodsMetric	获取Pod列表失败指标
	FailedGetResourceMetric	获取资源失败数
	FailedGetExternalMetric	获取外部指标失败
	InvalidMetricSourceType	无效的指标来源类型
	FailedConvertHPA	转换HPA失败
	FailedGetScale	获取比例尺失败
	FailedComputeMetricsReplicas	计算指标副本数失败
	FailedGetScaleWindow	获取ScaleWindow失败
	FailedRescale	扩缩容失败

----结束

编辑 HPA 策略

以HPA策略为例。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中单击“负载伸缩”，单击策略后方“操作”栏中的“更多 > 编辑”。

步骤3 在打开的“编辑HPA策略”页面中，参考表16-3更新策略参数。

步骤4 单击“确定”完成策略更新。

----结束

编辑 YAML（HPA 策略）

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中单击“负载伸缩”，单击HPA策略后方“操作”栏中的“编辑YAML”。

步骤3 在弹出的“编辑YAML”窗口中，可以对YAML进行修改和下载。

----结束

删除 HPA 策略

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中单击“负载伸缩”，单击策略后方栏中的“更多 > 删除”。

步骤3 在弹出的窗口中，单击“是”完成删除操作。

----结束

16.3 节点弹性伸缩

16.3.1 节点伸缩原理

HPA是针对Pod级别的，可以根据负载指标动态调整副本数量，但是如果集群的资源不足，新的副本无法运行的情况下，就只能对集群进行扩容。

Autoscaler是Kubernetes提供的集群节点弹性伸缩组件，根据Pod调度状态及资源使用情况对集群的节点进行自动扩容缩容，同时支持多可用区、多实例规格、指标触发和周期触发等多种伸缩模式，满足不同的节点伸缩场景。

前提条件

使用节点伸缩功能前，需要安装**autoscaler**插件，插件版本要求1.13.8及以上。

Autoscaler 工作原理

Autoscaler的主要流程包括两部分：

- ScaleUp流程： Autoscaler会每隔10s检查一次所有未调度的Pod，根据用户设置的策略，选择出一个符合要求的节点池进行扩容。

  说明

  Autoscaler检测未调度Pod进行扩容时，使用的是与Kubernetes社区版本一致的调度算法进行模拟调度计算，若应用调度采用非内置kube-scheduler调度器或其他非Kubernetes社区调度策略，此类应用使用Autoscaler扩容时可能因调度算法不一致出现无法扩容或多扩风险。

- ScaleDown流程：Autoscaler每隔10s会扫描一次所有的Node，如果该Node上所有的Pod Requests少于用户定义的缩容百分比时，Autoscaler会模拟将该节点上的Pod是否能迁移到其他节点，如果可以的话，当满足不被需要的时间窗以后，该节点就会被移除。

  当集群节点处于一段时间空闲状态时（默认10min），会触发集群缩容操作（即节点会被自动删除）。当节点存在以下几种状态的Pod时，不可缩容：

  - Pod有设置Pod Disruption Budget（即**干扰预算**），当移除Pod不满足对应条件时，节点不会缩容。

- Pod由于一些限制，如亲和、反亲和等，无法调度到其他节点，节点不会缩容。
- Pod拥有cluster-autoscaler.kubernetes.io/safe-to-evict: 'false'这个annotations时，节点不缩容。
- 节点上存在kube-system命名空间下的Pod（除kube-system命名空间下由DaemonSet创建的Pod），节点不缩容。
- 节点上如果有非controller（Deployment/ReplicaSet/Job/StatefulSet）创建的Pod，节点不缩容。

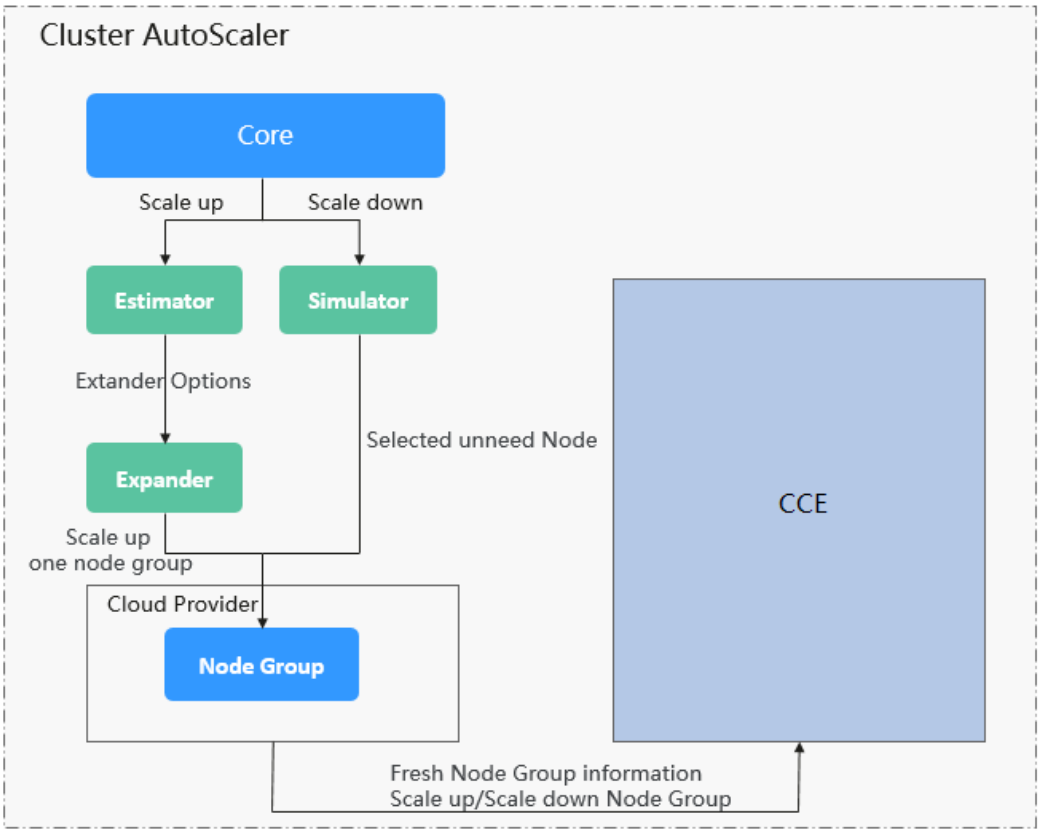
说明

当节点符合缩容条件时，Autoscaler将预先给节点打上DeletionCandidateOfClusterAutoscaler污点，限制Pod调度到该节点上。当autoscaler插件被卸载后，如果节点上依然存在该污点请您手动进行删除。

AutoScaler 架构

AutoScaler架构如图16-1所示，主要由以下几个核心模块组成：

图 16-1 AutoScaler 架构图



说明如下：

- Estimator： 负责扩容场景下，评估满足当前不可调度Pod时，每个节点池需要扩容的节点数量。
- Simulator： 负责缩容场景下，找到满足缩容条件的节点。

- Expander： 负责在扩容场景下，根据用户设置的不同的策略来，从Estimator选出的节点池中，选出一个最佳的选择。当前Expander有多种策略：

表 16-5 CCE 支持的 Expander 策略

策略	策略说明	使用场景	模拟样例
Random	随机选择一个可调度节点池中执行本次扩容。	此策略通常作为其他更复杂策略的基础退避策略。仅当其他优选策略无法决策时，作为备选策略使用，通常不建议直接配置使用。	假设集群中节点池1和节点池2启用了弹性伸缩，且均未达到扩容上限。当工作负载扩容副本数时，扩容策略如下： 1. Pending Pods触发autoscaler决策扩容流程。 2. autoscaler模拟调度阶段，评估节点池1和节点池2中扩容的节点均可调度。 3. autoscaler决策优选节点池，将在节点池1和节点池2范围中随机选择一个节点池执行扩容。
most-pods	组合型策略，优先级排序为：most-pods > random。 优先选择扩容后能调度最多Pods的节点池。如果存在多个节点池满足条件，则基于random策略进一步决策。	此策略基于最多可调度Pods数量作为优选依据。	假设集群中节点池1和节点池2启用了弹性伸缩，且均未达到扩容上限。当工作负载扩容副本数时，扩容策略如下： 1. Pending Pods触发autoscaler决策扩容流程。 2. autoscaler模拟调度阶段，评估节点池1和节点池2中扩容的节点均可调度部分Pending Pods。 3. autoscaler决策优选节点池，评估节点池1扩容后可调度工作负载新增的20个Pods，而节点池2扩容仅可调度工作负载新增的10个Pods，因此优选节点池1执行本次扩容。

策略	策略说明	使用场景	模拟样例
least-waste	组合型策略，优先级排序为：least-waste > random。 评估本次扩容的节点池整体CPU或MEM资源分配率，优先选择具有最小浪费的CPU或者Mem资源的节点池。如果存在多个节点池满足条件，则基于random策略进一步决策。	此策略将CPU或者内存资源最小浪费分数作为优选依据。 其中最小浪费分数wastedScore定义公式如下： • wastedCPU = (待扩容节点的CPU总量 - 待调度Pods的CPU总量) / 待扩容节点的CPU总量 • wastedMemory = (待扩容节点的MEM总量 - 待调度Pods的MEM总量) / 待扩容节点的MEM总量 • wastedScore = wastedCPU + wastedMemory	假设集群中节点池1和节点池2启用了弹性伸缩，且均未达到扩容上限。当工作负载扩容副本数时，扩容策略如下： 1. Pending Pods触发autoscaler决策扩容流程。 2. autoscaler模拟调度阶段，评估节点池1和节点池2中扩容的节点均可调度部分Pending Pods。 3. autoscaler决策优选节点池，评估节点池1扩容后最小浪费分数小于节点池2，因此优选节点池1执行本次扩容。
priority	组合策略，优先级排序为：priority > least-waste > random。 基于节点池/伸缩组优先级配置增强的least-waste策略。如果存在多个节点池满足条件，则基于least-waste策略进一步决策。	priority可通过Console/API主动配置管理节点池/伸缩组优先级，least-waste则在通用场景下降低资源浪费比例。此策略通用性较好，当前作为默认优选策略。	假设集群中节点池1和节点池2启用了弹性伸缩，且均未达到扩容上限。当工作负载扩容副本数时，扩容策略如下： 1. Pending Pods触发autoscaler决策扩容流程。 2. autoscaler模拟调度阶段，评估节点池1和节点池2中扩容的节点均可调度部分Pending Pods。 3. autoscaler决策优选节点池，评估节点池1优先级高于节点池2，因此优选节点池1执行本次扩容。

16.3.2 创建节点伸缩策略

CCE的自动伸缩能力是通过节点自动伸缩组件**autoscaler**实现的，可以按需弹出节点实例，支持多可用区、多实例规格、多种伸缩模式，满足不同的节点伸缩场景。

当节点伸缩中创建的策略和autoscaler插件中的配置同时生效时（比如不可调度和指标规则同时满足时），将优先执行不可调度扩容。

- 若不可调度执行成功，则会跳过指标规则逻辑，进入下一次循环。

- 若不可调度执行失败，将执行指标规则逻辑。

前提条件

使用节点伸缩功能前，集群中需要安装**autoscaler**插件，插件版本要求1.13.8及以上。

约束限制

- 弹性伸缩的策略作用在节点池，当节点池中节点为0时，且按CPU/内存弹性伸缩时，不会触发节点伸缩。
- 缩容节点会导致与节点关联的**本地持久卷**类型的PVC/PV数据丢失，无法恢复，且PVC/PV无法再正常使用。缩容节点时使用了本地持久存储卷的Pod会从缩容的节点上被驱逐，并重新创建Pod，Pod会一直处于pending状态，因为Pod使用的PVC带有节点标签，由于冲突无法调度成功。
- 使用autoscaler插件时，部分污点/注解可能会影响弹性伸缩功能，因此集群中应避免使用以下污点/注解：
  - **节点避免使用ignore-taint.cluster-autoscaler.kubernetes.io的污点**：该污点作用于节点。由于autoscaler原生支持异常扩容保护策略，会定期评估集群的可用节点比例，非Ready分类节点数统计比例超过45%比例会触发保护机制；而集群中任何存在该污点的节点都将从自动缩放器模板节点中过滤掉，记录到非Ready分类的节点中，进而影响集群的扩缩容。
  - **Pod避免使用cluster-autoscaler.kubernetes.io/enable-ds-eviction的注解**：该注解作用于Pod，控制DaemonSet Pod是否可以被autoscaler驱逐。详情请参见**Kubernetes原生的标签、注解和污点**。

操作步骤

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“节点伸缩”，进入创建节点伸缩策略页面。

- 若插件名称后方显示“未安装”，请单击插件后方的“安装”，根据业务需求配置插件参数后单击“安装”，等待插件安装完成。
- 若插件名称后方显示“已安装”，则说明插件已安装成功。

步骤3 单击右上角“创建节点伸缩策略”，参照如下参数设置策略。

- 策略名称：新建策略的名称，请自定义。
- 关联节点池：选择要关联的节点池。您可以关联多个节点池，以使用相同的伸缩策略。
- 执行规则：单击“添加规则”，在弹出的添加规则窗口中设置如下参数：

  规则名称：请输入规则名称，可自定义。

  规则类型：可选择“指标触发”或“周期触发”，两种类型区别如下：
  - **指标触发：**

    触发条件：请选择“CPU分配率”或“内存分配率”，输入百分比的值。该百分比应大于autoscaler插件中配置的缩容百分比。

说明

- 分配率 = 节点池容器组（Pod）资源申请量 / 节点池Pod可用资源量 (Node Allocatable) 。
- **如果多条规则同时满足条件，会有如下两种执行的情况：**

  如果同时配置了“CPU分配率”和“内存分配率”的规则，两种或多种规则同时满足扩容条件时，执行扩容节点数更多的规则。

  如果同时配置了“CPU分配率”和“周期触发”的规则，当达到“周期触发”的时间值时CPU也满足扩容条件时，较早执行的A规则会将节点池状态置为伸缩中状态，导致B规则无法正常执行。待A规则执行完毕，节点池状态恢复正常后，B规则也不会执行。
- 配置了“CPU分配率”和“内存分配率”的规则后，策略的检测周期会因autoscaler每次循环的处理逻辑而变动。只要一次检测出满足条件就会触发扩容（还需要满足冷却时间、节点池状态等约束条件）。

– **周期触发：**

触发时间：可选择每天、每周、每月或每年的具体时间点。

执行动作：与上述“触发条件”或“触发时间”相对应，达到触发条件或触发时间值后所要执行的动作。

您可以单击“添加规则”，设置多条节点伸缩策略。您最多可以添加1条CPU使用率指标规则、1条内存使用率指标规则，且规则总数小于等于10条。

步骤4 设置完成后，单击“确定”。

----结束

缩容说明

节点伸缩策略中不能直接设置缩容策略，您可以在安装**autoscaler插件**时进行设置。

节点缩容仅支持资源分配率缩容机制，当集群下CPU和内存分配率低于您设置的门限（在安装/编辑autoscaler插件时设置）时将对节点池下节点启动缩容（该功能可以关闭）。

YAML 样例

节点伸缩策略Yaml样例如下：

```
apiVersion: autoscaling.cce.io/v1alpha1
kind: HorizontalNodeAutoscaler
metadata:
  creationTimestamp: "2020-02-13T12:47:49Z"
  generation: 1
  name: xxxx
  namespace: kube-system
  resourceVersion: "11433270"
  selfLink: /apis/autoscaling.cce.io/v1alpha1/namespaces/kube-system/horizontalnodeautoscalers/xxxx
  uid: c2bd1e1d-60aa-47b5-938c-6bf3fadbe91f
spec:
  disable: false
  rules:
  - action:
      type: ScaleUp
      unit: Node
      value: 1
    cronTrigger:
      schedule: 47 20 * * *
```

```
  disable: false
  ruleName: cronrule
  type: Cron
- action:
    type: ScaleUp
    unit: Node
    value: 2
  disable: false
  metricTrigger:
    metricName: Cpu
    metricOperation: '>'
    metricValue: "40"
    unit: Percent
  ruleName: metricrule
  type: Metric
targetNodepoolIds:
- 7d48eca7-3419-11ea-bc29-0255ac1001a8
```

表 16-6 关键参数说明

参数	参数类型	描述
spec.disable	Bool	伸缩策略开关，会对策略中的所有规则生效
spec.rules	Array	伸缩策略中的所有规则
spec.rules[x].ruleName	String	规则名称
spec.rules[x].type	String	规则类型，当前支持“Cron”和“Metric”两种类型
spec.rules[x].disable	Bool	规则开关，当前仅支持“false”
spec.rules[x].action.type	String	规则操作类型，当前仅支持“ScaleUp”
spec.rules[x].action.unit	String	规则操作单位，当前仅支持“Node”
spec.rules[x].action.value	Integer	规则操作数值
spec.rules[x].cronTrigger	/	可选，仅在周期规则中有效
spec.rules[x].cronTrigger.schedule	String	周期规则的cron表达式
spec.rules[x].metricTrigger	/	可选，仅在指标规则中有效
spec.rules[x].metricTrigger.metricName	String	指标规则对应的指标，当前支持“Cpu”和“Memory”两种类型
spec.rules[x].metricTrigger.metricOperation	String	指标规则的比较符，当前仅支持“>”
spec.rules[x].metricTrigger.metricValue	String	指标规则的阈值，支持1-100之间的所有整数，需以字符串表示
spec.rules[x].metricTrigger.Unit	String	指标规则阈值的单位，当前仅支持“%”

参数	参数类型	描述
spec.targetNodepoolIds	Array	伸缩策略关联的所有节点池
spec.targetNodepoolIds[x]	String	伸缩策略关联节点池的uid

16.3.3 管理节点伸缩策略

操作场景

节点伸缩策略创建完成后，可对创建的策略进行删除、编辑、停用、启用、克隆等操作。

查看节点伸缩策略

您可以查看节点伸缩策略的关联节点池、执行规则和伸缩历史，参照界面中的提示有针对性的解决异常问题。

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“节点伸缩”，单击要查看的策略前方的 。

步骤3 在展开的区域中，可以看到该策略的关联节点池、执行规则和伸缩历史页签，若策略异常，请参照界面中的报错提示进行定位处理。

说明

您还可以在节点池管理中关闭或开启弹性扩缩容：

1. 登录CCE控制台，单击集群名称进入集群。
2. 在左侧导航栏中单击“节点管理”并切换至“节点池”页签。
3. 单击要操作的节点池的“更新节点池”按钮，在弹出的窗口中可设置“弹性伸缩配置”的节点数上下限和冷却时间。

----结束

删除节点伸缩策略

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“节点伸缩”，单击要删除的策略后方的“更多 > 删除”。

步骤3 在弹出的“删除节点伸缩策略”窗口中，确认是否删除。

步骤4 单击“是”按钮即完成删除操作。

----结束

编辑节点伸缩策略

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“节点伸缩”，单击要编辑的策略后方的“编辑”。

步骤3 在打开的“编辑节点伸缩策略”页面中，参照表16-6更新策略参数。

步骤4 完成设置后，单击“确定”按钮完成编辑操作。

----结束

克隆节点伸缩策略

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“节点伸缩”，单击要克隆的策略后方的“更多 > 克隆”。

步骤3 在打开的“拷贝节点伸缩策略”页面中，可以看到部分参数已经克隆过来，请按照业务需求补充或修改其他策略参数。

步骤4 单击“确定”完成策略克隆。

----结束

停用/启用节点伸缩策略

步骤1 在CCE控制台，单击集群名称进入集群。

步骤2 单击左侧导航栏的“节点伸缩”，单击策略后方的“停用”，若策略为停用状态时，则单击策略后方的“启用”。

步骤3 在弹出的“停用节点策略”或“启用节点策略”窗口中，确认是否进行停用或启用操作。

----结束

16.4 使用 HPA+CA 实现工作负载和节点联动弹性伸缩

应用场景

企业应用的流量大小不是每时每刻都一样，有高峰，有低谷，如果每时每刻都要保持能够扛住高峰流量的机器数目，那么成本会很高。通常解决这个问题的办法就是根据流量大小或资源占用率自动调节机器的数量，也就是弹性伸缩。

当使用Pod/容器部署应用时，通常会设置容器的申请/限制值来确定可使用的资源上限，以避免在流量高峰期无限制地占用节点资源。然而，这种方法可能会存在资源瓶颈，达到资源使用上限后可能会导致应用出现异常。为了解决这个问题，可以通过伸缩Pod的数量来分摊每个应用实例的压力。如果增加Pod数量后，节点资源使用率上升到一定程度，继续扩容出来的Pod无法调度，则可以根据节点资源使用率继续伸缩节点数量。

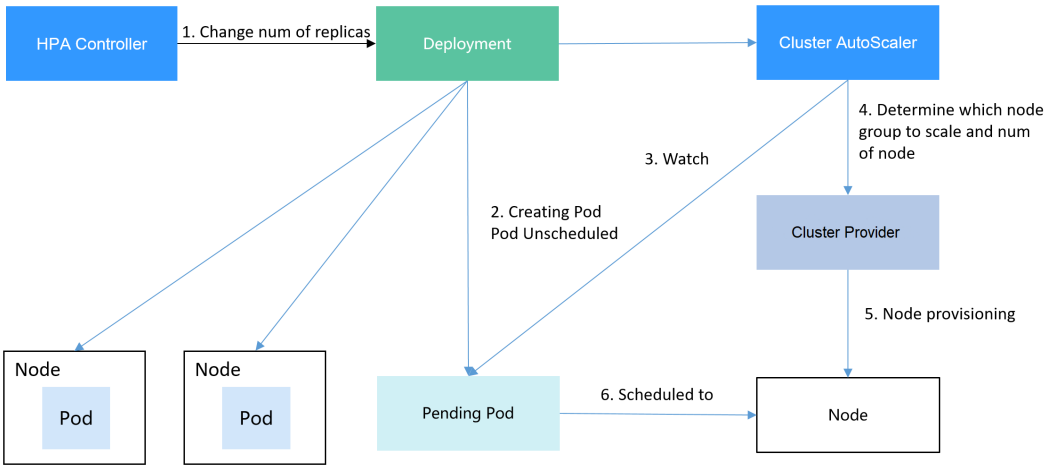
解决方案

CCE中弹性伸缩最主要的就是使用HPA（Horizontal Pod Autoscaling）和CA（Cluster AutoScaling）两种弹性伸缩策略，HPA负责工作负载弹性伸缩，也就是应用层面的弹性伸缩，CA负责节点弹性伸缩，也就是资源层面的弹性伸缩。

通常情况下，两者需要配合使用，因为HPA需要集群有足够的资源才能扩容成功，当集群资源不够时需要CA扩容节点，使得集群有足够资源；而当HPA缩容后集群会有大量空余资源，这时需要CA缩容节点释放资源，才不至于造成浪费。

如图16-2所示，HPA根据监控指标进行扩容，当集群资源不够时，新创建的Pod会处于Pending状态，CA会检查所有Pending状态的Pod，根据用户配置的扩缩容策略，选择出一个最合适的节点池，在这个节点池扩容。

图 16-2 HPA + CA 工作流程



使用HPA+CA可以很容易做到弹性伸缩，且节点和Pod的伸缩过程可以非常方便的观察到，使用HPA+CA做弹性伸缩能够满足大部分业务场景需求。

本文将通过一个示例介绍HPA+CA两种策略配合使用下弹性伸缩的过程，从而帮助您更好的理解和使用弹性伸缩。

准备工作

步骤1 创建一个有1个节点的集群，节点规格为2U4G及以上，并在创建节点时为节点添加弹性公网IP，以便从外部访问。如创建节点时未绑定弹性公网IP，您也可以前往ECS控制台为该节点进行手动绑定。

步骤2 给集群安装插件。

- autoscaler：节点伸缩插件。
- metrics-server：是Kubernetes集群范围资源使用数据的聚合器，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据。

步骤3 登录集群节点，准备一个算力密集型的应用。当用户请求时，需要先计算出结果后才返回给用户结果，如下所示。

1. 创建一个名为index.php的PHP文件，文件内容是在用户请求时先循环开方1000000次，然后再返回“OK!”。

   ```
   vi index.php
   ```

   文件内容如下：

   ```
   <?php
     $x = 0.0001;
     for ($i = 0; $i <= 1000000; $i++) {
       $x += sqrt($x);
     }
     echo "OK!";
   ?>
   ```

2. 编写Dockerfile制作镜像。

   ```
   vi Dockerfile
   ```

   Dockerfile内容如下：

```
FROM php:5-apache
COPY index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

3. 执行如下命令构建镜像，镜像名称为hpa-example，版本为latest。
   ```
   docker build -t hpa-example:latest .
   ```
4. （可选）登录SWR管理控制台，选择左侧导航栏的“组织管理”，单击页面右上角的“创建组织”，创建一个组织。

   如已有组织可跳过此步骤。
5. 在左侧导航栏选择“我的镜像”，单击右侧“客户端上传”，在弹出的页面中单击“生成临时登录指令”，单击 复制登录指令。
6. 在集群节点上执行上一步复制的登录指令，登录成功会显示“Login Succeeded”。
7. 为hpa-example镜像添加标签。

   docker tag [镜像名称1:版本名称1] [镜像仓库地址]/[组织名称]/[镜像名称2:版本名称2]
   - **[镜像名称1:版本名称1]**：请替换为您本地所要上传的实际镜像的名称和版本名称。
   - **[镜像仓库地址]**：可在SWR控制台上查询，**登录指令**中末尾的域名即为镜像仓库地址。
   - **[组织名称]**：请替换为**已创建的组织名称**。
   - **[镜像名称2:版本名称2]**：请替换为SWR镜像仓库中需要显示的镜像名称和镜像版本。

   示例：

   docker tag hpa-example:latest {*Image repository address*}/group/hpa-example:latest
8. 上传镜像至镜像仓库。

   docker push [镜像仓库地址]/[组织名称]/[镜像名称2:版本名称2]

   示例：

   docker push {*Image repository address*}/group/hpa-example:latest

   终端显示如下信息，表明上传镜像成功。

   ```
   6d6b9812c8ae: Pushed
   ...
   fe4c16cbf7a4: Pushed
   latest: digest: sha256:eb7e3bbd*** size: **
   ```

   返回容器镜像服务控制台，在“我的镜像”页面，执行刷新操作后可查看到对应的镜像信息。

----结束

创建节点池和节点伸缩策略

步骤1 登录CCE控制台，进入已创建的集群，在左侧单击“节点管理”，选择“节点池”页签并单击右上角“创建节点池”。

步骤2 填写节点池配置，添加2U4G的节点，并打开弹性扩缩容开关。

- 节点数量：设置为1，表示创建节点池时默认创建的节点数为1。
- 弹性伸缩：开启，表示节点池将根据集群负载情况自动创建或删除节点池内的节点。

- 节点数上限：设置为5，表示节点池中节点数的最大值。
- 节点规格：2核 | 4GiB

其余参数设置可使用默认值。

步骤3 在集群控制台左侧单击“插件管理”，单击autoscaler插件下的“编辑”按钮，修改autoscaler插件配置，将自动缩容开关打开，并配置缩容相关参数。例如节点资源使用率小于50%时进行缩容扫描，启动缩容。

上面配置的节点池弹性伸缩，会根据Pod的Pending状态进行扩容，根据节点的资源使用率进行缩容。

步骤4 在集群控制台左侧单击“节点伸缩”，单击页面右上角的“创建节点伸缩策略”。这里的节点伸缩策略可以根据CPU/内存分配率扩容、还可以按照时间定期扩容节点数量。

节点伸缩示例如下，这里配置当集群CPU分配率大于70%时，增加一个节点。CA策略需要关联节点池，可以关联多个节点池，当需要对节点扩缩容时，在节点池中根据最小浪费规则挑选合适规格的节点扩缩容。

----结束

创建工作负载

使用构建的hpa-example镜像创建无状态工作负载，副本数为1，镜像地址与上传到SWR仓库的组织有关，需要替换为实际取值。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hpa-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hpa-example
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
      - name: container-1
        image: 'hpa-example:latest' # 替换为您上传到SWR的镜像地址
        resources:
          limits:                   # limits与requests建议取值保持一致，避免扩缩容过程中出现震荡
            cpu: 500m
            memory: 200Mi
          requests:
            cpu: 500m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

然后再为这个负载创建一个Nodeport类型的Service，以便能从外部访问。

```
kind: Service
apiVersion: v1
metadata:
  name: hpa-example
```

```
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 31144
  selector:
    app: hpa-example
  type: NodePort
```

创建 HPA 策略

创建HPA策略，如下所示，该策略关联了名为hpa-example的负载，期望CPU使用率为50%。

另外有两条注解annotations，一条是CPU的阈值范围，最低30，最高70，表示CPU使用率在30%到70%之间时，不会扩缩容，防止小幅度波动造成影响。另一条是扩缩容时间窗，表示策略成功触发后，在缩容/扩容冷却时间内，不会再次触发缩容/扩容，以防止短期波动造成影响。

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-policy
  annotations:
    extendedhpa.metrics: '[{"type":"Resource","name":"cpu","targetType":"Utilization","targetRange":{"low":"30","high":"70"}}]'
    extendedhpa.option: '{"downscaleWindow":"5m","upscaleWindow":"3m"}'
spec:
  scaleTargetRef:
    kind: Deployment
    name: hpa-example
    apiVersion: apps/v1
  minReplicas: 1
  maxReplicas: 100
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

观察弹性伸缩过程

步骤1 首先查看集群节点情况，如下所示，有两个节点。

```
# kubectl get node
NAME           STATUS   ROLES    AGE     VERSION
192.168.0.183   Ready    <none>   2m20s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   55m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

查看HPA策略，可以看到目标负载的指标（CPU使用率）为0%

```
# kubectl get hpa hpa-policy
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%    1         100       1          4m
```

步骤2 通过如下命令访问负载，如下所示，其中{ip:port}为负载的访问地址，可以在负载的详情页中查询。

while true;do wget -q -O- http://*{ip:port}*; done

说明

如果此处不显示公网IP地址，则说明集群节点没有弹性公网IP，请创建弹性公网IP并绑定到节点，创建完后需要同步节点信息。

观察负载的伸缩过程。

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          4m
hpa-policy   Deployment/hpa-example   190%/50%   1         100       1          4m23s
hpa-policy   Deployment/hpa-example   190%/50%   1         100       4          4m31s
hpa-policy   Deployment/hpa-example   200%/50%   1         100       4          5m16s
hpa-policy   Deployment/hpa-example   200%/50%   1         100       4          6m16s
hpa-policy   Deployment/hpa-example   85%/50%    1         100       4          7m16s
hpa-policy   Deployment/hpa-example   81%/50%    1         100       4          8m16s
hpa-policy   Deployment/hpa-example   81%/50%    1         100       7          8m31s
hpa-policy   Deployment/hpa-example   57%/50%    1         100       7          9m16s
hpa-policy   Deployment/hpa-example   51%/50%    1         100       7          10m
hpa-policy   Deployment/hpa-example   58%/50%    1         100       7          11m
```

可以看到4m23s时负载的CPU使用率为190%，超过了目标值，此时触发了负载弹性伸缩，将负载扩容为4个副本/Pod，随后的几分钟内，CPU使用并未下降，直到到7m16s时CPU使用率才开始下降，这是因为新创建的Pod并不一定创建成功，可能是因为资源不足Pod处于Pending状态，这段时间内在扩容节点。

到7m16s时CPU使用率开始下降，说明Pod创建成功，开始分担请求流量，到8分钟时下降到81%，还是高于目标值，且高于70%，说明还会再次扩容，到9m16s时再次扩容到7个Pod，这时CPU使用率降为51%，在30%-70%的范围内，不会再次伸缩，可以观察到此后Pod数量一直稳定在7个。

观察负载和HPA策略的详情，从事件中可以看到负载的扩容的过程和策略生效的时间，如下所示。

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 7
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization (percentage of request) above target
```

此时查看节点数量，发现节点多了两个，也就是在刚才过程中节点扩容了两个。

```
# kubectl get node
NAME           STATUS   ROLES    AGE     VERSION
192.168.0.120   Ready    <none>   3m5s    v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.136   Ready    <none>   6m58s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.183   Ready    <none>   18m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   71m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

在控制台也可以看到伸缩历史，这里可以看到CA策略执行了一次，当集群中CPU分配率大于70%，将节点池中节点数量从2扩容到3。另一个节点是autoscaler默认的根据Pod的Pending状态扩容而来，在HPA初期。

这里节点扩容过程具体是这样：

1. Pod数量变为4后，由于没有资源，Pod处于Pending状态，触发了autoscaler默认的扩容策略，将节点数增加一个。
2. 第二次节点扩容是因为集群中CPU分配率大于70%，触发了CA策略，从而将节点数增加一个，从控制台上伸缩历史可以看出来。根据分配率扩容，可以保证集群一直处于资源充足的状态。

步骤3 停止访问负载，观察负载Pod数量。

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   50%/50%    1         100       7          12m
hpa-policy   Deployment/hpa-example   21%/50%    1         100       7          13m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          14m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          23m
```

可以看到从13m开始CPU使用率为21%，18m时Pod数量缩为3个，到23m时Pod数量缩为1个。

观察负载和HPA策略的详情，从事件中可以看到负载的扩容的过程和策略生效的时间，如下所示。

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 7
  Normal  ScalingReplicaSet  6m28s  deployment-controller  Scaled down replica set hpa-
example-79dd795485 to 3
  Normal  ScalingReplicaSet  72s    deployment-controller  Scaled down replica set hpa-
example-79dd795485 to 1
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  6m45s  horizontal-pod-autoscaler  New size: 3; reason: All metrics below target
  Normal  SuccessfulRescale  90s    horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

在控制台同样可以看到HPA策略生效历史，再继续等待，会看到节点也会被缩容一个。

这里为何没有被缩容掉两个节点，是因为节点池中这两个节点都存在kube-system namespace下的Pod（且不是DaemonSets创建的Pod）。

----结束

总结

通过上述内容可以看到，使用HPA+CA可以很容易做到弹性伸缩，且节点和Pod的伸缩过程可以非常方便的观察到，使用HPA+CA做弹性伸缩能够满足大部分业务场景需求。

17 插件

17.1 插件概述

CCE提供了多种类型的插件，用于管理集群的扩展功能，以支持选择性扩展满足特性需求的功能。

须知

CCE插件采用Helm模板方式部署，修改或升级插件请从插件配置页面或开放的插件管理API进行操作。勿直接后台直接修改插件相关资源，以免插件异常或引入其他非预期问题。

表 17-1 插件列表

插件名称	插件简介
CoreDNS	CoreDNS插件是一款通过链式插件的方式为Kubernetes提供域名解析服务的DNS服务器。
CCE Container Storage (FlexVolume)	storage-driver插件是用于对接块存储、文件存储、对象存储等Iaas存储服务的FlexVolume驱动。
CCE Container Storage (Everest)	everest是一个云原生容器存储系统，基于CSI为Kubernetes v1.15.6及以上版本集群对接云存储服务的能力。
npd	node-problem-detector（简称：npd）是一款监控集群节点异常事件的插件，以及对接第三方监控平台功能的组件。它是一个在每个节点上运行的守护程序，可从不同的守护进程中搜集节点问题并将其报告给apiserver。node-problem-detector可以作为DaemonSet运行， 也可以独立运行。
CCE Cluster Autoscaler	集群自动扩缩容插件autoscaler，是根据pod调度状态及资源使用情况对集群的工作节点进行自动扩容缩容的插件。

插件名称	插件简介
Kubernetes Metrics Server	Metrics-Server是集群核心资源监控数据的聚合器。
gpu-device-plugin	gpu-device-plugin（原gpu-beta）插件是支持在容器中使用GPU显卡的设备管理插件，仅支持Nvidia驱动。
Volcano Scheduler	volcano提供了高性能任务调度引擎、高性能异构芯片管理、高性能任务运行管理等通用计算能力，通过接入AI、大数据、基因、渲染等诸多行业计算框架服务终端用户。
NGNIX Ingress Controller	nginx-ingress为Service提供了可直接被集群外部访问的虚拟主机、负载均衡、SSL代理、HTTP路由等应用层转发功能。

插件生命周期

生命周期是指插件从安装到卸载历经的各种状态。

表 17-2 插件生命周期状态说明

状态	状态属性	说明
运行中	稳定状态	插件正常运行状态，所有插件实例均正常部署，插件可正常使用。
部分就绪	稳定状态	插件正常运行状态，部分插件实例未正常部署。此状态下，插件功能可能无法正常使用。
不可用	稳定状态	插件异常状态，所有插件实例均未正常部署。
安装中	中间状态	插件正处于部署状态。 如遇到插件配置错误或资源不足所有实例均无法调度等情况，系统会在10分钟后将该插件置为“不可用”状态。
安装失败	稳定状态	插件安装失败，需要卸载后重新安装。
升级中	中间状态	插件正处于更新状态。
升级失败	稳定状态	插件升级失败，可重试升级或卸载后重新安装。
回滚中	中间状态	插件正在回滚中。
回滚失败	稳定状态	插件回滚失败，可重试回滚或卸载后重新安装。
删除中	中间状态	插件处于正在被删除的状态。 如果长时间处于该状态，则说明出现异常。
删除失败	稳定状态	插件删除失败，可重试卸载。
未知状态	稳定状态	插件模板实例不存在。

说明

当插件处于“安装中”或“删除中”等中间状态时，不可进行编辑、卸载等相关操作。

插件相关操作

您可以在“插件管理”执行表17-3中的操作。

表 17-3 插件相关操作

操作	说明	操作步骤
安装	安装指定的插件。	1. 登录CCE控制台，单击集群名称进入集群，在左侧导航栏选择“插件管理”。 2. 单击需要安装插件下的“安装”。 由于不同插件支持的配置参数不同，详细步骤请参见插件章节。 3. 设置完插件参数后，单击“确定”。
升级	将插件升级至新版。	1. 登录CCE控制台，单击集群名称进入集群，在左侧导航栏选择“插件管理”。 2. 如存在可升级的插件，该插件将提供“升级”按钮。 单击“升级”。由于不同插件支持的配置参数不同，详细步骤请参见插件章节。 3. 设置完插件参数后，单击“确定”。
编辑	编辑插件参数。	1. 登录CCE控制台，单击集群名称进入集群，在左侧导航栏选择“插件管理”。 2. 单击需要编辑插件下的“编辑”。 由于不同插件支持的配置参数不同，详细步骤请参见插件章节。 3. 设置完插件参数后，单击“确定”。
卸载	将插件从集群中卸载。	1. 登录CCE控制台，单击集群名称进入集群，在左侧导航栏选择“插件管理”。 2. 单击需要编辑插件下的“卸载”。 3. 在弹出的确认窗口中单击“是”。 卸载操作无法恢复，请谨慎操作。

17.2 CoreDNS

插件简介

CoreDNS插件是一款通过链式插件的方式为Kubernetes提供域名解析服务的DNS服务器。

CoreDNS是由CNCF孵化的开源软件，用于Cloud-Native环境下的DNS服务器和服务发现解决方案。CoreDNS实现了插件链式架构，能够按需组合插件，运行效率高、配置

灵活。在Kubernetes集群中使用CoreDNS能够自动发现集群内的服务，并为这些服务提供域名解析。同时，通过级联云上DNS服务器，还能够为集群内的工作负载提供外部域名的解析服务。

该插件为系统资源插件，在创建集群时默认安装。

目前CoreDNS已经成为社区Kubernetes集群推荐的DNS服务器解决方案。

CoreDNS官网：**https://coredns.io/**

开源社区地址：**https://github.com/coredns/coredns**

说明

DNS详细使用方法请参见**DNS**。

约束与限制

CoreDNS正常运行或升级时，请确保集群中的可用节点数大于等于插件的实例数，且所有实例都处于运行状态，否则将导致插件异常或升级失败。

安装插件

本插件为系统默认安装，若因特殊情况卸载后，可参照如下步骤重新安装。

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“插件管理”，在右侧找到**CoreDNS**，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-4 插件规格配置

参数	参数说明
插件规格	并发域名解析能力，请根据业务需求选择插件规格。 如选择“自定义qps”，CoreDNS所能提供的域名解析QPS与CPU消耗成正相关，请根据业务需求，合理调整插件实例数和容器CPU/内存配额。
实例数	选择上方插件规格后，显示插件中的实例数。 选择“自定义qps”规格时，您可根据需求调整插件实例数。
多可用区部署	• 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 • 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	选择插件规格后，显示插件容器的CPU和内存配额。 选择“自定义qps”规格时，您可根据需求调整插件实例的容器规格。

步骤3 设置插件支持的“参数配置”。

表 17-5 插件参数配置

参数	参数说明
存根域设置	对自定义的域名配置域名服务器，格式为一个键值对，键为DNS后缀域名，值为一个或一组DNS IP地址，如 'acme.local --1.2.3.4,6.7.8.9'。 详情请参见**为CoreDNS配置存根域**。

参数	参数说明

高级配置	• parameterSyncStrategy：插件升级时是否配置一致性检查。
	– ensureConsistent：表示启用配置一致性检查，如果集群中记录的配置和实际生效配置不一致，插件将无法升级。
	– force：表示升级时忽略配置一致性检查。请您自行确保当前生效配置和原配置一致。插件升级完毕后，需恢复parameterSyncStrategy值为ensureConsistent，重新启用配置一致性检查。
	– inherit：表示升级时自动继承差异配置。插件升级完毕后，需恢复parameterSyncStrategy值为ensureConsistent，重新启用配置一致性检查。
	• stub_domains：存根域，您可对自定义的域名配置域名服务器，格式为一个键值对，键为DNS后缀域名，值为一个或一组DNS IP地址。
	• upstream_nameservers：上游域名服务器地址。
	• servers: CoreDNS 1.23.1插件版本开始开放servers配置，用户可对servers做定制化配置，详情请参见**dns-custom-nameservers**。
	其中plugins为CoreDNS中各组件配置。一般场景建议保持默认配置，以免出现配置错误而导致CoreDNS整体不可用。每个plugin组件可包含"name"、"parameters"(可选)、"configBlock"(可选)配置，对应生成的Corefile配置文件中格式如下：
	`$name $parameters { $configBlock }`
	常用plugin的说明请参见**表17-6**。更多配置详情请参见**Plugins**。
	示例：
	`{ "servers": [ { "plugins": [ { "name": "bind", "parameters": "{$POD_IP}" }, { "name": "cache", "parameters": 30 }, { "name": "errors" }, { "name": "health", "parameters": "{$POD_IP}:8080" }, { "name": "ready", "{$POD_IP}:8081" }, { "configBlock": "pods insecure\nfallthrough in-addr.arpa`

参数	参数说明

	ip6.arpa",
	"name": "kubernetes",
	"parameters": "cluster.local in-addr.arpa ip6.arpa"
	},
	{
	"name": "loadbalance",
	"parameters": "round_robin"
	},
	{
	"name": "prometheus",
	"parameters": "{$POD_IP}:9153"
	},
	{
	"configBlock": "policy random",
	"name": "forward",
	"parameters": ". /etc/resolv.conf"
	},
	{
	"name": "reload"
	}
	],
	"port": 5353,
	"zones": [
	{
	"zone": "."
	}
	]
	}
	],
	"stub_domains": {
	"acme.local": [
	"1.2.3.4",
	"6.7.8.9"
	]
	},
	"upstream_nameservers": ["8.8.8.8", "8.8.4.4"]
	}

表 17-6 CoreDNS 主 zone 默认 plugin 配置说明

plugin名称	描述
bind	CoreDNS侦听的hostIP配置，建议保持当前默认值{$POD_IP}。详情请参见**bind**。
cache	启用DNS缓存。详情请参见**cache**。
errors	错误信息到标准输出。详情请参见**errors**。
health	CoreDNS健康检查配置，当前侦听{$POD_IP}:8080，请保持此默认值，否则导致coredns健康检查失败而不断重启。详情请参见**health**。
ready	检查后端服务是否准备好接收流量，当前侦听{$POD_IP}:8081。如果后端服务没有准备好，CoreDNS将会暂停 DNS 解析服务，直到后端服务准备好为止。详情请参见**ready**。

plugin名称	描述
kubernetes	CoreDNS Kubernetes插件，提供集群内服务解析能力。详情请参见**kubernetes**。
loadbalance	轮转式 DNS 负载均衡器， 在应答中随机分配 A、AAAA 和 MX 记录的顺序。详情请参见**loadbalance**。
prometheus	CoreDNS自身metrics数据接口, 默认zone侦听{$POD_IP}:9153，请保持此默认值，否则普罗无法采集coredns metrics数据。详情请参见**prometheus**。
forward	不在 Kubernetes 集群域内的任何查询都将转发到默认的解析器 (/etc/resolv.conf)。详情请参见**forward**。
reload	允许自动重新加载已更改的 Corefile。 编辑 ConfigMap 配置后，请等待两分钟，以使更改生效。详情请参见**reload**。

步骤4 完成以上配置后，单击“安装”。

----结束

组件说明

表 17-7 coredns 组件

容器组件	说明	资源类型
coredns	该容器为提供集群域名解析服务的DNS服务器。	Deployment

Kubernetes 中的域名解析逻辑

DNS策略可以在每个pod基础上进行设置，目前，Kubernetes支持**Default**、**ClusterFirst**、**ClusterFirstWithHostNet**和**None**四种DNS策略，具体请参见**https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/**。这些策略在pod-specific的**dnsPolicy**字段中指定。

- **“Default”**：如果dnsPolicy被设置为“Default”，则名称解析配置将从pod运行的节点继承。 自定义上游域名服务器和存根域不能够与这个策略一起使用。
- **“ClusterFirst”**：如果dnsPolicy被设置为“ClusterFirst”，任何与配置的集群域后缀不匹配的DNS查询（例如，www.kubernetes.io）将转发到从该节点继承的上游名称服务器。集群管理员可能配置了额外的存根域和上游DNS服务器。
- **“ClusterFirstWithHostNet”**：对于使用hostNetwork运行的Pod，您应该明确设置其DNS策略“ClusterFirstWithHostNet”。
- **“None”**：它允许Pod忽略Kubernetes环境中的DNS设置。应使用dnsConfigPod规范中的字段提供所有DNS设置 。

说明

- Kubernetes 1.10及以上版本，支持Default、ClusterFirst、ClusterFirstWithHostNet和None四种策略；低于Kubernetes 1.10版本，仅支持default、ClusterFirst和ClusterFirstWithHostNet三种。
- “Default”不是默认的DNS策略。如果dnsPolicy的Flag没有特别指明，则默认使用“**ClusterFirst**”。

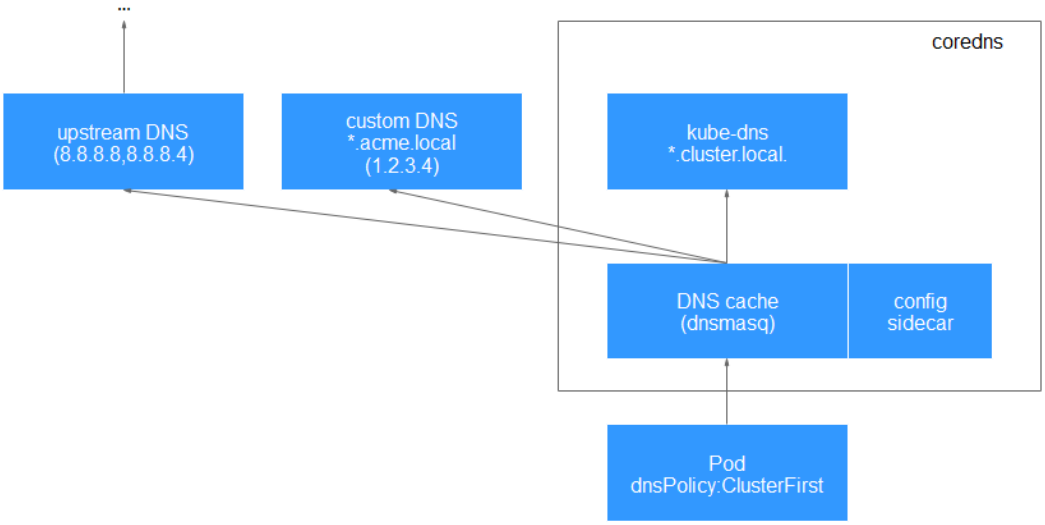
路由请求流程：

未配置存根域：没有匹配上配置的集群域名后缀的任何请求，例如“www.kubernetes.io”，将会被转发到继承自节点的上游域名服务器。

已配置存根域：如果配置了存根域和上游DNS服务器，DNS查询将基于下面的流程对请求进行路由：

1. 查询首先被发送到coredns中的DNS缓存层。
2. 从缓存层，检查请求的后缀，并根据下面的情况转发到对应的DNS上：
   - 具有集群后缀的名字（例如“.cluster.local”）：请求被发送到coredns。
   - 具有存根域后缀的名字（例如“.acme.local”）：请求被发送到配置的自定义DNS解析器（例如：监听在 1.2.3.4）。
   - 未能匹配上后缀的名字（例如“widget.com”）：请求被转发到上游DNS。

图 17-1 路由请求流程



17.3 CCE Container Storage (Everest)

插件简介

everest是一个云原生容器存储系统，基于CSI（即Container Storage Interface）为Kubernetes v1.15.6及以上版本集群对接云存储服务的能力。

该插件为系统资源插件，Kubernetes 1.15及以上版本的集群在创建时默认安装。

约束与限制

- 集群版本由v1.13升级到v1.15后，v1.13版本集群中的Flexvolume容器存储插件（**storage-driver**）能力将由v1.15的CSI插件（everest，插件版本v1.1.6及以上）接管，接管后原有功能保持不变。
- 插件版本为1.2.0的everest优化了使用OBS存储时的**密钥认证功能**，低于该版本的everest插件在升级完成后，需要重启集群中使用OBS存储的全部工作负载，否则工作负载使用存储的能力将受影响！
- **v1.15及以上版本**的集群默认安装本插件，v1.13及以下版本集群创建时默认安装**storage-driver**插件。

安装插件

本插件为系统默认安装，若因特殊情况卸载后，可参照如下步骤重新安装。

步骤1 登录CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件管理”，在右侧找到CCE Container Storage (Everest)，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-8 插件规格配置

参数	参数说明
实例数	插件实例的副本数量。 实例数为1时插件不具备高可用能力，当插件实例所在节点异常时可能导致插件功能无法正常使用，请谨慎选择。
多可用区部署	• 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 • 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	everest插件包含everest-csi-controller和everest-csi-driver两个组件，详情请参见**组件说明**。 选择插件规格后，显示插件组件的CPU和内存申请值。详情请参见**表17-9**。 非典型场景下，限制值一般估算公式如下： • everest-csi-controller： – CPU限制值：200及以下节点规模设置为250m；1000节点规模设置为350m；2000节点规模设置为500m。 – 内存限制值 =（200Mi + 节点数 * 1Mi + PVC数 * 0.2Mi）* 1.2 • everest-csi-driver： – CPU限制值：200及以下节点规模设置为300m；1000节点规模设置为500m；2000节点规模设置为800m。 – 内存限制值 = 200及以下节点规模设置为300Mi；1000节点规模设置为600Mi；2000节点规模设置为900Mi。

表 17-9 典型场景组件限制值建议

配置场景			everest-csi-controller组件		everest-csi-driver组件	
节点数量	PV/PVC数量	插件实例数	CPU（限制值同申请值）	内存（限制值同申请值）	CPU（限制值同申请值）	内存（限制值同申请值）
50	1000	2	250m	600Mi	300m	300Mi
200	1000	2	250m	1Gi	300m	300Mi
1000	1000	2	350m	2Gi	500m	600Mi
1000	5000	2	450m	3Gi	500m	600Mi
2000	5000	2	550m	4Gi	800m	900Mi
2000	10000	2	650m	5Gi	800m	900Mi

步骤3 设置插件支持的“参数配置”。

表 17-10 插件参数配置

参数	参数说明
csi_attacher_worker_threads	everest插件中同时处理挂EVS卷的worker数，默认值为“60”。
csi_attacher_detach_worker_threads	everest插件中同时处理卸载EVS卷的worker数，默认值均为“60”。
volume_attaching_flow_ctrl	everest插件在1分钟内可以挂载EVS卷的最大数量，此参数的默认值“0”表示everest插件不做挂卷限制，此时挂卷性能由底层存储资源决定。
cluster_id	集群ID。
default_vpc_id	集群所在VPC的ID。
disable_auto_mount_secret	挂载对象桶/并行文件系统时，是否允许使用默认的AKSK，默认为false。
enable_node_attacher	是否开启agent侧attacher，开启后由attacher负责处理VolumeAttachment。
flow_control	默认为空。用户无需填写。
over_subscription	本地存储池（local_storage）的超分比。默认为80，若本地存储池为100G，可以超分为180G使用。
project_id	集群所属项目ID。

说明

everest 1.2.26以上版本针对大批量挂EVS卷的性能做了优化，用户可配置如下3个参数：

- csi_attacher_worker_threads
- csi_attacher_detach_worker_threads
- volume_attaching_flow_ctrl

上述三个参数由于存在关联性且与集群所在局点的底层存储资源限制有关，当您对大批量挂卷的性能有要求（大于500EVS卷/分钟）时，请联系客服，在指导下进行配置，否则可能会因为参数配置不合理导致出现everest插件运行不正常的情况。

步骤4 单击“安装”。

----结束

组件说明

表 17-11 everest 组件

容器组件	说明	资源类型
everest-csi-controller	此容器负责存储卷的创建、删除、快照、扩容、attach/detach等功能。若集群版本大于等于1.19，且插件版本为1.2.x，everest-csi-controller组件的Pod还会默认带有一个everest-localvolume-manager容器，此容器负责管理节点上的lvm存储池及localpv的创建。	Deployment
everest-csi-driver	此容器负责PV的挂载、卸载、文件系统resize等功能。若插件版本为1.2.x，且集群所在区域支持node-attacher，everest-csi-driver组件的Pod还会带有一个everest-node-attacher的容器，此容器负责分布式attach EVS，该配置项在部分Region开放。	DaemonSet

17.4 npd

插件简介

node-problem-detector（简称：npd）是一款监控集群节点异常事件的插件，以及对接第三方监控平台功能的组件。它是一个在每个节点上运行的守护程序，可从不同的守护进程中搜集节点问题并将其报告给apiserver。node-problem-detector可以作为DaemonSet运行， 也可以独立运行。

有关社区开源项目node-problem-detector的详细信息，请参见**node-problem-detector**。

约束与限制

- 使用NPD插件时，不可对节点磁盘进行格式化或分区。
- 节点上每个NPD进程标准占用30mCPU，100MB内存。

权限说明

NPD插件为监控内核日志，需要读取宿主机/dev/kmsg设备，为此需要开启容器特权，详见**privileged**。

同时CCE根据最小化权限原则进行了风险消减，NPD运行限制只拥有以下特权：

- cap_dac_read_search，为访问/run/log/journal
- cap_sys_admin，为访问/dev/kmsg

安装插件

步骤1 登录CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件管理”，在右侧找到**npd**，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-12 npd 插件规格配置

参数	参数说明
插件规格	该插件可配置“自定义”规格。
实例数	选择“自定义”规格时，您可根据需求调整插件实例数。
多可用区部署	• 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 • 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	选择“自定义”规格时，您可根据需求调整插件实例的容器规格。

步骤3 设置插件支持的“参数配置”。

仅v1.16.0及以上版本支持配置。

表 17-13 npd 插件参数配置

参数	参数说明
common.image.pullPolicy	镜像拉取策略，默认为IfNotPresent。
feature_gates	特性门控。
npc.maxTaintedNode	单个故障在多个节点间发生时，至多多少节点允许被npc添加污点，避免雪崩效应。 支持int格式和百分比格式。
npc.nodeAffinity	Controller的节点亲和性配置。

说明

此处仅描述部分参数，更多参数请参考开源项目node-problem-detector的详细信息。

步骤4 单击“安装”。

----结束

组件说明

表 17-14 npd 组件

容器组件	说明	资源类型
node-problem-controller	根据故障探测结果提供基础故障隔离能力。	Deployment
node-problem-detector	提供节点故障探测能力。	DaemonSet

NPD 检查项

说明

当前检查项仅1.16.0及以上版本支持。

NPD的检查项主要分为事件类检查项和状态类检查项。

- 事件类检查项

  对于事件类检查项，当问题发生时，NPD会向APIServer上报一条事件，事件类型分为Normal（正常事件）和Warning（异常事件）

  表 17-15 事件类检查项

故障检查项	功能	说明
OOMKilling	监听内核日志，检查OOM事件发生并上报 典型场景：容器内进程使用的内存超过了Limt，触发OOM并终止该进程	Warning类事件 监听对象：/dev/kmsg 匹配规则："Killed process \\d+ (.+) total-vm:\\d+kB, anon-rss:\\d+kB, file-rss:\\d+kB.*"
TaskHung	监听内核日志，检查taskHung事件发生并上报 典型场景：磁盘卡IO导致进程卡住	Warning类事件 监听对象：/dev/kmsg 匹配规则："task \\S+:\\w+ blocked for more than \\w+ seconds\\."

故障检查项	功能	说明
ReadonlyFilesystem	监听内核日志，检查系统内核是否有Remount root filesystem read-only错误 典型场景：用户从ECS侧误操作卸载节点数据盘，且应用程序对该数据盘的对应挂载点仍有持续写操作，触发内核产生IO错误将磁盘重挂载为只读磁盘。	Warning类事件 监听对象：/dev/kmsg 匹配规则："Remounting filesystem read-only"

- 状态类检查项

  对于状态类检查项，当问题发生时，NPD会向APIServer上报一条事件，并同步修改节点状态，可配合**Node-problem-controller故障隔离**对节点进行隔离。

  下列检查项中若未明确指出检查周期，则默认周期为30秒。

  表 17-16 系统组件检查

故障检查项	功能	说明
容器网络组件异常 CNIProblem	检查CNI组件（容器网络组件）运行状态	无
容器运行时组件异常 CRIProblem	检查节点CRI组件（容器运行时组件）Docker和Containerd的运行状态	检查对象：Docker或Containerd
Kubelet频繁重启 FrequentKubeletRestart	通过定期回溯系统日志，检查关键组件Kubelet是否频繁重启	• 默认阈值：10分钟内重启10次 即在10分钟内组件重启10次表示频繁重启，将会产生故障告警。 • 监听对象：/run/log/journal目录下的日志
Docker频繁重启 FrequentDockerRestart	通过定期回溯系统日志，检查容器运行时Docker是否频繁重启	
Containerd频繁重启 FrequentContainerdRestart	通过定期回溯系统日志，检查容器运行时Containerd是否频繁重启	
Kubelet服务异常 KubeletProblem	检查关键组件Kubelet的运行状态	无
KubeProxy异常 KubeProxyProblem	检查关键组件KubeProxy的运行状态	无

表 17-17 系统指标

故障检查项	功能	说明
连接跟踪表耗尽 ConntrackFullProblem	检查连接跟踪表是否耗尽	• 默认阈值:90% • 使用量:nf_conntrack_count • 最大值:nf_conntrack_max
磁盘资源不足 DiskProblem	检查节点系统盘、CCE数据盘（包含CRI逻辑盘与Kubelet逻辑盘）的磁盘使用情况	• 默认阈值：90% • 数据来源:`df -h` 当前暂不支持额外的数据盘
文件句柄数不足 FDProblem	检查系统关键资源FD文件句柄数是否耗尽	• 默认阈值：90% • 使用量：/proc/sys/fs/file-nr中第1个值 • 最大值：/proc/sys/fs/file-nr中第3个值
节点内存资源不足 MemoryProblem	检查系统关键资源Memory内存资源是否耗尽	• 默认阈值：80% • 使用量：/proc/meminfo中MemTotal-MemAvailable • 最大值：/proc/meminfo中MemTotal
进程资源不足 PIDProblem	检查系统关键资源PID进程资源是否耗尽	• 默认阈值：90% • 使用量：/proc/loadavg中nr_threads • 最大值：/proc/sys/kernel/pid_max和/proc/sys/kernel/threads-max两者的较小值。

表 17-18 存储检查

故障检查项	功能	说明
磁盘只读 DiskReadonly	通过定期对节点系统盘、CCE数据盘（包含CRI逻辑盘与Kubelet逻辑盘）进行测试性写操作，检查关键磁盘的可用性	检测路径： ● /mnt/paas/kubernetes/kubelet/ ● /var/lib/docker/ ● /var/lib/containerd/ ● /var/paas/sys/log/cceaddon-npd/ 检测路径下会产生临时文件npd-disk-write-ping 当前暂不支持额外的数据盘
磁盘资源不足 DiskProblem	检查节点系统盘、CCE数据盘（包含cri逻辑盘与kubelet逻辑盘）的磁盘使用情况	● 默认阈值：90% ● 数据来源：`df -h` 当前暂不支持额外的数据盘
节点emptydir存储池异常 EmptyDirVolumeGroupStatusError	检查节点上临时卷存储池是否正常 故障影响：依赖存储池的Pod无法正常写对应临时卷。临时卷由于IO错误被内核重挂载成只读文件系统。 典型场景：用户在创建节点时配置两个数据盘作为临时卷存储池，用户误操作删除了部分数据盘导致存储池异常。	● 检测周期：30秒 ● 数据来源：`vgs -o vg_name, vg_attr` ● 检测原理：检查VG（存储池）是否存在p状态，该状态表征部分PV（数据盘）丢失。 ● 节点持久卷存储池异常调度联动：调度器可自动识别此异常状态并避免依赖存储池的Pod调度到该节点上。 ● 例外场景：NPD无法检测所有PV（数据盘）丢失，导致VG（存储池）丢失的场景；此时依赖kubelet自动隔离该节点，其检测到VG（存储池）丢失并更新nodestatus.allocatable中对应资源为0，避免依赖存储池的Pod调度到该节点上。无法检测单个PV损坏；此时依赖ReadonlyFilesystem检测异常。
节点持久卷存储池异常 LocalPvVolumeGroupStatusError	检查节点上持久卷存储池是否正常 故障影响：依赖存储池的Pod无法正常写对应持久卷。持久卷由于IO错误被内核重挂载成只读文件系统。 典型场景：用户在创建节点时配置两个数据盘作为持久卷存储池，用户误操作删除了部分数据盘。	

故障检查项	功能	说明
挂载点异常 MountPointProblem	检查节点上的挂载点是否异常 异常定义：该挂载点不可访问（cd） 典型场景：节点挂载了nfs（网络文件系统，常见有obsfs、s3fs等），当由于网络或对端nfs服务器异常等原因导致连接异常时，所有访问该挂载点的进程均卡死。例如集群升级场景kubelet重启时扫描所有挂载点，当扫描到此异常挂载点会卡死，导致升级失败。	等效检查命令： `for dir in `df -h | grep -v "Mounted on" | awk "{print \\$NF}"`;do cd $dir; done && echo "ok"`
磁盘卡IO DiskHung	检查节点上所有磁盘是否存在卡IO，即IO读写无响应 卡IO定义：系统对磁盘的IO请求下发后未有响应，部分进程卡在D状态 典型场景：操作系统硬盘驱动异常或底层网络严重故障导致磁盘无法响应	• 检查对象：所有数据盘 • 数据来源：/proc/diskstat 等效查询命令：`iostat -xmt 1` • 阈值： – 平均利用率，ioutil >= 0.99 – 平均IO队列长度，avgqu-sz >=1 – 平均IO传输量，iops(w/s) +ioth(wMB/s) <= 1 **说明** 部分操作系统卡IO时无数据变化，此时计算CPU IO时间占用率，iowait > 0.8。
磁盘慢IO DiskSlow	检查节点上所有磁盘是否存在慢IO，即IO读写有响应但响应缓慢 典型场景：云硬盘由于网络波动导致慢IO。	• 检查对象：所有数据盘 • 数据来源：/proc/diskstat 等效查询命令 `iostat -xmt 1` • 默认阈值：平均IO时延，await >= 5000ms **说明** 卡IO场景下该检查项失效，原因为IO请求未有响应，await数据不会刷新。

表 17-19 其他检查

故障检查项	功能	说明
NTP异常 NTPProblem	检查节点时钟同步服务ntpd或chronyd是否正常运行，系统时间是否漂移	默认时钟偏移阈值：8000ms
进程D异常 ProcessD	检查节点是否存在D进程	默认阈值：连续3次存在10个异常进程 数据来源： ● /proc/{PID}/stat ● 等效命令：ps aux 例外场景：ProcessD忽略BMS节点下的SDI卡驱动依赖的常驻D进程heartbeat、update
进程Z异常 ProcessZ	检查节点是否存在Z进程	
ResolvConf配置文件异常 ResolvConfFileProblem	检查ResolvConf配置文件是否丢失 检查ResolvConf配置文件是否异常 异常定义：不包含任何上游域名解析服务器（nameserver）。	检查对象：/etc/resolv.conf
存在计划事件 ScheduledEvent	检查节点是否存在热迁移计划事件。热迁移计划事件通常由硬件故障触发，是IaaS层的一种自动故障修复手段。 典型场景：底层宿主机异常，例如风扇损坏、磁盘坏道等，导致其上虚机触发热迁移。	数据来源： ● http://169.254.169.254/meta-data/latest/events/scheduled 该检查项为Alpha特性，默认不开启。

另外kubelet组件内置如下检查项，但是存在不足，您可通过集群升级或安装NPD进行补足。

表 17-20 Kubelet 内置检查项

故障检查项	功能	说明
PID资源不足 PIDPressure	检查PID是否充足	• 周期：10秒 • 阈值：90% • 缺点：社区1.23.1及以前版本，该检查项在pid使用量大于65535时失效，详见**issue 107107**。社区1.24及以前版本，该检查项未考虑thread-max。
内存资源不足 MemoryPressure	检查容器可分配空间（allocable）内存是否充足	• 周期：10秒 • 阈值：最大值-100MiB • 最大值（Allocable）：节点总内存-节点预留内存 • 缺点：该检测项没有从节点整体内存维度检查内存耗尽情况，只关注了容器部分(Allocable)。
磁盘资源不足 DiskPressure	检查kubelet盘和docker盘的磁盘使用量及inodes使用量	• 周期：10秒 • 阈值：90%

Node-problem-controller 故障隔离

说明

故障隔离仅1.16.0及以上版本的插件支持。

默认情况下，若多个节点发生故障，NPC至多为10%的节点添加污点，可通过参数npc.maxTaintedNode提高数量限制。

开源NPD插件提供了故障探测能力，但未提供基础故障隔离能力。对此，CCE在开源NPD的基础上，增强了Node-problem-controller（节点故障控制器组件），该组件参照Kubernetes**节点控制器**实现，针对NPD探测上报的故障，自动为节点添加污点以进行基本的节点故障隔离。

表 17-21 参数说明

参数	说明	默认值
npc.enable	是否启用npc 1.18.0及以上版本不再支持该参数	true

参数	说明	默认值
npc.maxTaintedNode	单个故障在多个节点间发生时，至多多少节点允许被npc添加污点，避免雪崩效应 支持int格式和百分比格式	10% 值域： • int格式，数值范围为1到无穷大 • 百分比格式，数值范围为1%到100%，与集群节点数量乘积计算后最小值为1。
npc.nodeAffinity	Controller的节点亲和性配置	N/A

17.5 CCE Cluster Autoscaler

插件简介

CCE集群弹性引擎（autoscaler）是Kubernetes中非常重要的一个Controller，它提供了微服务的弹性能力，并且和Serverless密切相关。

弹性伸缩是很好理解的一个概念，当微服务负载高（CPU/内存使用率过高）时水平扩容，增加pod的数量以降低负载，当负载降低时减少pod的数量，减少资源的消耗，通过这种方式使得微服务始终稳定在一个理想的状态。

云容器引擎简化了Kubernetes集群的创建、升级和手动扩缩容，而集群中应用的负载本身是会随着时间动态变化的，为了更好的平衡资源使用率以及性能，Kubernetes引入了autoscaler插件，它可以根据部署的应用所请求的资源量自动伸缩集群中节点数量，详情请了解**创建节点伸缩策略**。

开源社区地址：https://github.com/kubernetes/autoscaler

插件说明

autoscaler可分成扩容和缩容两个方面：

- **自动扩容**

  集群的自动扩容有以下两种方式实现：

  - 当集群中的Pod由于工作节点资源不足而无法调度时，会触发集群扩容，扩容节点与所在节点池资源配额一致。

    此时需要满足以下条件时才会执行自动扩容：

    - 节点上的资源不足。
    - Pod的调度配置中不能包含节点亲和的策略（即Pod若已经设置亲和某个节点，则不会自动扩容节点），节点亲和策略设置方法请参见**调度策略（亲和与反亲和）**。
  - 当集群满足节点伸缩策略时，也会触发集群扩容，详情请参见**创建节点伸缩策略**。

> **说明**
>
> 当前该插件使用的是最小浪费策略，即若Pod创建需要3核，此时有4核、8核两种规格，优先创建规格为4核的节点。

- **自动缩容**

  当集群节点处于一段时间空闲状态时（默认10min），会触发集群缩容操作（即节点会被自动删除）。当节点存在以下几种状态的Pod时，不可缩容：

  - Pod有设置Pod Disruption Budget（即**干扰预算**），当移除Pod不满足对应条件时，节点不会缩容。
  - Pod由于一些限制，如亲和、反亲和等，无法调度到其他节点，节点不会缩容。
  - Pod拥有cluster-autoscaler.kubernetes.io/safe-to-evict: 'false'这个annotations时，节点不缩容。
  - 节点上存在kube-system命名空间下的Pod（除kube-system命名空间下由DaemonSet创建的Pod），节点不缩容。
  - 节点上如果有非controller（Deployment/ReplicaSet/Job/StatefulSet）创建的Pod，节点不缩容。

  > **说明**
  >
  > 当节点符合缩容条件时，Autoscaler将预先给节点打上DeletionCandidateOfClusterAutoscaler污点，限制Pod调度到该节点上。当autoscaler插件被卸载后，如果节点上依然存在该污点请您手动进行删除。

约束与限制

- 安装时请确保有足够的资源安装本插件。
- 默认节点池不支持弹性扩缩容，详情请参见**默认节点池DefaultPool说明**。
- 使用autoscaler插件时，部分污点/注解可能会影响弹性伸缩功能，因此集群中应避免使用以下污点/注解：
  - **节点避免使用ignore-taint.cluster-autoscaler.kubernetes.io的污点**：该污点作用于节点。由于autoscaler原生支持异常扩容保护策略，会定期评估集群的可用节点比例，非Ready分类节点数统计比例超过45%比例会触发保护机制；而集群中任何存在该污点的节点都将从自动缩放器模板节点中过滤掉，记录到非Ready分类的节点中，进而影响集群的扩缩容。
  - **Pod避免使用cluster-autoscaler.kubernetes.io/enable-ds-eviction的注解**：该注解作用于Pod，控制DaemonSet Pod是否可以被autoscaler驱逐。详情请参见**Kubernetes原生的标签、注解和污点**。

安装插件

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“ 插件管理”，在右侧找到CCE Cluster Autoscaler，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-22 插件规格配置

参数	参数说明
插件规格	插件部署可选择如下几种规格。 **说明** autoscaler插件高可用或自定义部署时，插件实例间存在反亲和策略，会分别部署在不同的节点上，因此集群中的可用节点必须大于等于插件实例数才可保证插件高可用。 ● 单实例：以单实例部署插件。 ● 高可用50：50节点集群规模多实例部署，实例数为2，具有高可用能力。 ● 高可用200：200节点集群规模多实例部署，实例数为2，具有高可用能力，每个实例使用资源比高可用50的实例更多。 ● 自定义：根据需要自定义实例数量和实例规格。
实例数	选择上方插件规格后，显示插件中的实例数。 选择“自定义”规格时，您可根据需求调整插件实例数。
多可用区部署	● 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 ● 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	选择插件规格后，显示插件容器的CPU和内存配额。 选择“自定义”规格时，您可根据需求调整插件实例的容器规格。

步骤3 设置插件支持的“参数配置”。

表 17-23 插件参数配置

参数	说明
扩缩容配置	可以勾选自动扩缩容。 • **当集群下负载实例无法调度时自动扩容（从节点池）** 即当出现Pod处于Pending状态无法调度时，集群会自动扩容节点。若Pod已经设置亲和某个节点，则不会自动扩容节点。该功能一般与HPA策略配合使用，具体请参见**使用HPA+CA实现工作负载和节点联动弹性伸缩**。 如不勾选，则只能通过**节点伸缩策略**进行扩缩容。 • **自动缩容节点** – 空置时间（min）：当集群节点处于一段时间的空闲状态时，会触发集群缩容操作，删除节点，默认10min。 – 缩容阈值：当集群节点资源（Request值）低于多少百分比时，进行集群缩容扫描（默认0.5，即50%，cpu和mem都要满足的条件下才会缩容）。 – 冷却时间： 扩容执行后多久能再次判断是否缩容，默认10min。 **说明** 集群中如果同时存在自动扩容和自动缩容的场景，建议配置“扩容执行后多久能再次判断是否缩容”为0min，避免由于部分节点池持续扩容或者扩容失败重试而阻塞整体缩容节点行为，导致非预期的节点资源浪费。 节点删除后多久能再次判断是否缩容：删除节点后能再次启动缩容评估的时间间隔，默认10min。 缩容失败后多久能再次判断是否缩容：缩容失败后能再次启动缩容评估的时间间隔，默认3min。节点池中配置的缩容冷却时间和此处配置的缩容冷却时间之间的影响和关系请参见**缩容冷却时间说明**。 – 缩容并发数：最多支持多少个空闲节点同时缩容，默认10。 缩容并发数只针对完全空闲节点，完全空闲节点可实现并发缩容。非完全空闲节点则只能逐个缩容。 **说明** 节点在缩容的时候，若节点上的Pod不需要驱逐（DaemonSet的Pod认为不需要驱逐），则认为该节点为完全空闲节点，否则认为该节点为非完全空闲。 – 检查间隔：节点被判定不可移除后能再次启动检查的时间间隔，默认5min。
节点总数	集群可管理的节点数目的最大值，扩容时不会让集群下节点数超过此值。
CPU总数（核）	集群中所有节点 CPU 核数之和的最大值，扩容时不会让集群下节点CPU核数之和超过此值。
内存总数（GB）	集群中所有节点内存之和的最大值，扩容时不会让集群下节点内存之和超过此值。

步骤4 配置完成后，单击“安装”。

----结束

组件说明

表 17-24 autoscaler 组件

容器组件	说明	资源类型
autoscaler	该容器为Kubernetes集群提供自动扩缩容节点的能力。	Deployment

缩容冷却时间说明

节点池中配置的缩容冷却时间和autoscaler插件中配置的缩容冷却时间之间的影响和关系如下：

节点池配置的缩容冷却时间

弹性缩容冷却时间：当前节点池扩容出的节点多长时间不能被缩容，作用范围为节点池级别。

autoscaler插件配置的缩容冷却时间

扩容后缩容冷却时间：autoscaler触发扩容后（不可调度、指标、周期策略）整个集群多长时间内不能被缩容，作用范围为集群级别。

节点删除后缩容冷却时间：autoscaler触发缩容后整个集群多长时间内不能继续缩容，作用范围为集群级别。

缩容失败后缩容冷却时间：autoscaler触发缩容失败后整个集群多长时间内不能继续缩容，作用范围为集群级别。

17.6 NGNIX Ingress Controller

插件简介

Kubernetes通过kube-proxy服务实现了Service的对外发布及负载均衡，它的各种方式都是基于传输层实现的。在实际的互联网应用场景中，不仅要实现单纯的转发，还有更加细致的策略需求，如果使用真正的负载均衡器更会增加操作的灵活性和转发性能。

基于以上需求，Kubernetes引入了资源对象Ingress，Ingress为Service提供了可直接被集群外部访问的虚拟主机、负载均衡、SSL代理、HTTP路由等应用层转发功能。

Kubernetes官方发布了基于Nginx的Ingress控制器，CCE的nginx-ingress插件直接使用社区模板与镜像。Nginx Ingress控制器会将Ingress生成一段Nginx的配置，并将Nginx配置通过ConfigMap进行储存，这个配置会通过Kubernetes API写到Nginx的Pod中，然后完成Nginx的配置修改和更新，详细工作原理请参见**工作原理**。

开源社区地址：**https://github.com/kubernetes/ingress-nginx**

说明

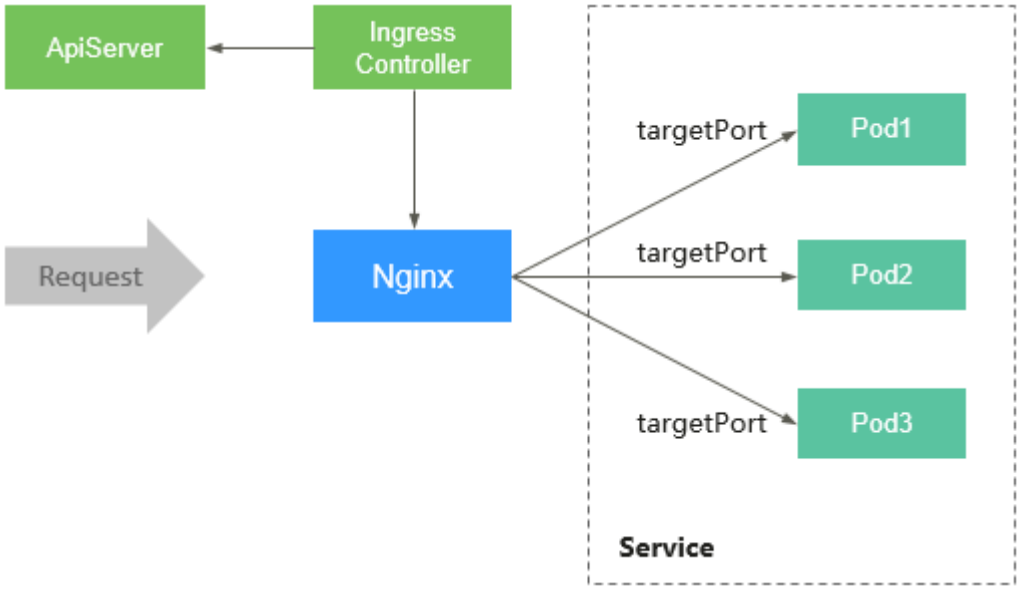
- 安装该插件时，您可以通过“定义nginx配置”添加配置，此处的设置将会全局生效，该参数直接通过配置nginx.conf生成，将影响管理的全部Ingress，相关参数可通过**configmap**查找，如果您配置的参数不包含在**configmap**所列出的选项中将不会生效。
- 请勿手动修改和删除CCE自动创建的ELB和监听器，否则将出现工作负载异常；若您已经误修改或删除，请卸载Nginx Ingress插件后重装。

工作原理

Nginx Ingress由资源对象Ingress、Ingress控制器、Nginx三部分组成，Ingress控制器用以将Ingress资源实例组装成Nginx配置文件（nginx.conf），并重新加载Nginx使变更的配置生效。当它监听到Service中Pod变化时通过动态变更的方式实现Nginx上游服务器组配置的变更，无须重新加载Nginx进程。工作原理如**图17-2**所示。

- Ingress：一组基于域名或URL把请求转发到指定Service实例的访问规则，是Kubernetes的一种资源对象，Ingress实例被存储在对象存储服务etcd中，通过接口服务被实现增、删、改、查的操作。
- Ingress控制器（Ingress Controller）：用以实时监控资源对象Ingress、Service、End-point、Secret（主要是TLS证书和Key）、Node、ConfigMap的变化，自动对Nginx进行相应的操作。
- Nginx：实现具体的应用层负载均衡及访问控制。

图 17-2 Nginx Ingress 工作原理



使用约束

- 仅支持在1.15及以上版本的CCE集群中安装此插件。
- 通过API接口创建的Ingress在注解中必须添加kubernetes.io/ingress.class: "nginx"。
- 独享型ELB规格必须支持网络型（TCP/UDP），且网络类型必须支持私网（有私有IP地址）。

- 运行nginx-ingress-controller的节点以及该节点上运行的容器，无法访问Nginx Ingress，请将工作负载Pod与nginx-ingress-controller进行反亲和部署，具体操作步骤请参见**工作负载与nginx-ingress-controller反亲和部署**。

前提条件

在创建容器工作负载前，您需要存在一个可用集群。若没有可用集群 ，请参照**创建集群**中的步骤创建。

安装插件

说明

- 社区v1.2.0版本nginx-ingress-controller（对应CCE nginx-ingress插件2.1.0版本）已修复**CVE-2021-25746**漏洞，新增**规则**禁用一些存在越权风险的Anntotations值。
- 社区v1.2.0版本nginx-ingress-controller（对应CCE nginx-ingress插件2.1.0版本）已修复**CVE-2021-25745**漏洞，新增**规则**禁用一些存在越权风险的访问路径。

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“ 插件管理”，在右侧找到NGNIX Ingress Controller，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-25 插件规格配置

参数	参数说明
插件规格	该插件可配置“自定义”规格。
实例数	选择“自定义”规格时，您可根据需求调整插件实例数。
多可用区部署	• 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 • 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	选择“自定义”规格时，您可根据需求调整插件实例的容器规格。

步骤3 设置插件支持的“参数配置”。

- **负载均衡器**：支持对接共享型或独享型负载均衡实例，如果无可用实例，请先创建。负载均衡器需要拥有至少两个监听器配额，且端口 80 和 443 没有被监听器占用。
- **nginx配置参数**：配置nginx.conf文件，将影响管理的全部Ingress，相关参数可通过**configmap**查找，如果您配置的参数不包含在**configmap**所列出的选项中将不会生效。

  此处以设置keep-alive-requests参数为例，设置保持活动连接的最大请求数为100。

  ```
  {
    "keep-alive-requests": "100"
  }
  ```

- **默认404服务**：默认使用插件自带的404服务。支持自定义404服务，填写“命名空间/服务名称”，如果服务不存在，插件会安装失败。

步骤4 单击“安装”。

----结束

组件说明

表 17-26 nginx-ingress 组件

容器组件	说明	资源类型
nginx-ingress	基于Nginx的Ingress控制器，为集群提供灵活的路由转发能力。	Deployment

工作负载与 nginx-ingress-controller 反亲和部署

运行nginx-ingress-controller的节点以及该节点上运行的容器，无法访问Nginx Ingress，为避免这个问题发生，需要将工作负载与nginx-ingress-controller反亲和部署，即工作负载Pod无法调度至nginx-ingress-controller运行的节点。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:aplpine
        imagePullPolicy: IfNotPresent
        name: nginx
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:    #通过nginx-ingress-controller的标签实现反亲和
                - key: app
                  operator: In
                  values:
                    - nginx-ingress
                - key: component
                  operator: In
                  values:
                    - controller
              namespaces:
                - kube-system
              topologyKey: kubernetes.io/hostname
```

17.7 Kubernetes Metrics Server

从Kubernetes 1.8开始，Kubernetes通过Metrics API提供资源使用指标，例如容器CPU和内存使用率。这些度量可以由用户直接访问（例如，通过使用kubectl top命令），或者由集群中的控制器（例如，Horizontal Pod Autoscaler）使用来进行决策，具体的组件为Metrics-Server，用来替换之前的heapster，heapster从1.11开始逐渐被废弃。

Metrics Server是集群核心资源监控数据的聚合器，您可以在CCE控制台快速安装本插件。

安装本插件后，可在“弹性伸缩”页面的“工作负载伸缩”页签下，创建HPA策略，具体请参见**HPA策略**。

社区官方项目及文档：**https://github.com/kubernetes-sigs/metrics-server**。

安装插件

步骤1 登录CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件管理”，在右侧找到Kubernetes Metrics Server，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-27 插件规格配置

参数	参数说明
插件规格	该插件可配置“单实例”、“高可用”或“自定义”规格。
实例数	选择上方插件规格后，显示插件中的实例数。 选择“自定义”规格时，您可根据需求调整插件实例数。
多可用区部署	• 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 • 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	选择插件规格后，显示插件容器的CPU和内存配额。 选择“自定义”规格时，您可根据需求调整插件实例的容器规格。

步骤3 单击“安装”。

----结束

组件说明

表 17-28 metrics-server 组件

容器组件	说明	资源类型
metrics-server	集群核心资源监控数据的聚合器，用于收集和聚合集群中通过Metrics API提供的资源使用指标。	Deployment

17.8 gpu-device-plugin

插件简介

gpu-device-plugin插件是支持在容器中使用GPU显卡的设备管理插件，集群中使用GPU节点时必须安装本插件。

约束与限制

- 下载的驱动必须是后缀为“.run”的文件。
- 仅支持Nvidia Tesla驱动，不支持GRID驱动。
- 安装或重装插件时，需要保证驱动下载链接正确且可正常访问，插件对链接有效性不做额外校验。
- 插件仅提供驱动的下载及安装脚本执行功能，插件的状态仅代表插件本身功能正常，与驱动是否安装成功无关。
- 对于GPU驱动版本与您业务应用的兼容性（GPU驱动版本与CDUA库版本的兼容性），CCE不保证两者之间兼容性，请您自行验证。
- 对于已经安装GPU驱动的自定义操作系统镜像，CCE无法保证其提供的GPU驱动与CCE其他GPU组件兼容（例如监控组件等）。

安装插件

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“插件管理”，在右侧找到gpu-device-plugin，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-29 插件规格配置

参数	参数说明
插件规格	该插件可配置“默认”或“自定义”规格。
容器	选择插件规格后，显示插件容器的CPU和内存配额。 选择“自定义”规格时，您可根据需求调整插件实例的容器规格。

步骤3 设置插件支持的“参数配置”。

- Nvidia驱动：填写Nvidia驱动的下载链接，集群下全部GPU节点将使用相同的驱动。

  须知

  - 如果下载链接为公网地址，如nvidia官网地址（ https://us.download.nvidia.com/tesla/470.103.01/NVIDIA-Linux-x86_64-470.103.01.run ），各GPU节点均需要绑定EIP。获取驱动链接方法请参考**获取驱动链接-公网地址**。
  - 若下载链接为OBS上的链接，无需绑定EIP 。获取驱动链接方法请参考**获取驱动链接-OBS地址**。
  - 请确保Nvidia驱动版本与GPU节点适配。
  - 更改驱动版本后，需要重启节点才能生效。

- 驱动选择：若您不希望集群中的所有GPU节点使用相同的驱动，CCE支持以节点池为单位安装不同的GPU驱动。

  说明

  - 插件将根据节点池指定的驱动版本进行安装，仅对节点池新建节点生效。
  - 新建节点更新驱动版本后，需重启节点生效。非新建节点不支持更新驱动版本。

步骤4 单击“安装”，安装插件的任务即可提交成功。

说明

卸载插件会清理节点上的GPU驱动，将会导致新调度的GPU Pod无法正常运行，但已运行的GPU Pod不会受到影响。

----结束

验证插件

插件安装完成后，在GPU节点及调度了GPU资源的容器中执行nvidia-smi命令，验证GPU设备及驱动的可用性。

- GPU节点：

```
# 插件版本为2.0.0以下时，执行以下命令：
cd /opt/cloud/cce/nvidia/bin && ./nvidia-smi

# 插件版本为2.0.0及以上时，驱动安装路径更改，需执行以下命令：
cd /usr/local/nvidia/bin && ./nvidia-smi
```

- 容器：

```
cd /usr/local/nvidia/bin && ./nvidia-smi
```

若能正常返回GPU信息，说明设备可用，插件安装成功。

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.118.02   Driver Version: 440.118.02   CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| 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 V100-SXM2...  Off  | 00000000:21:01.0 Off |                    0 |
| N/A   31C    P0    23W / 300W |      0MiB / 16160MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

获取驱动链接-公网地址

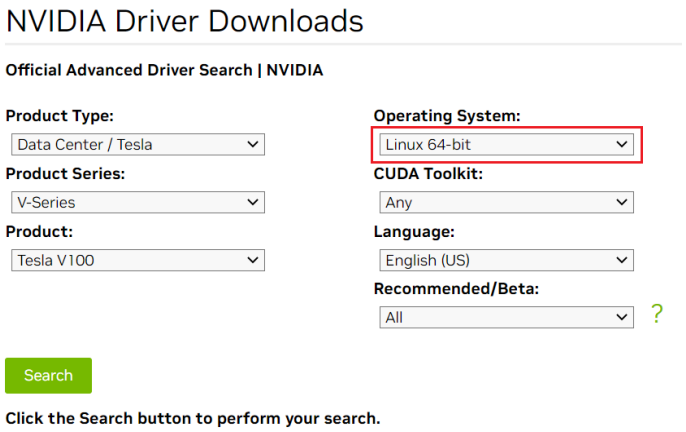
步骤1 登录CCE控制台。

步骤2 创建节点，在节点规格处选择要创建的GPU节点，选中后下方显示的信息中可以看到节点的GPU显卡型号。

步骤3 登录到*https://www.nvidia.com/Download/Find.aspx?lang=cn*网站。

步骤4 如图17-3所示，在“NVIDIA驱动程序下载”框内选择对应的驱动信息。其中“操作系统”必须选**Linux 64-bit**。

图 17-3 参数选择



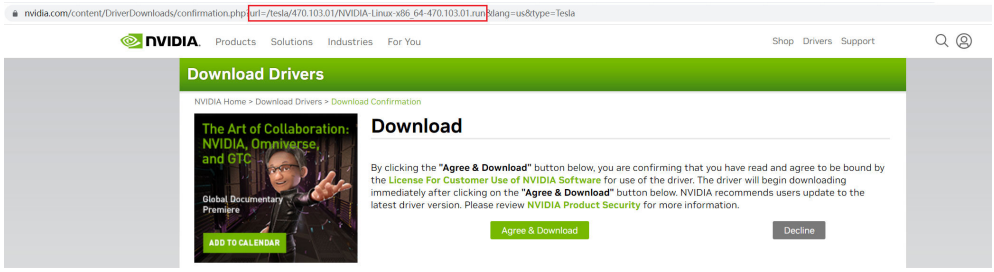
步骤5 驱动信息确认完毕，单击“搜索”按钮，会跳转到驱动信息展示页面，该页面会显示驱动的版本信息如图17-4，单击“下载”到下载页面。

图 17-4 驱动信息

步骤6 获取驱动软件链接方式分两种：

- 方式一：如图17-5，在浏览器的链接中找到路径为*url=/tesla/470.103.01/NVIDIA-Linux-x86_64-470.103.01.run*的路径，补齐全路径**https://us.download.nvidia.com/tesla/470.103.01/NVIDIA-Linux-x86_64-470.103.01.run**该方式节点需要绑定EIP 。
- 方式二：如图17-5，单击“下载”按钮下载驱动，然后上传到OBS，获取软件的链接，该方式节点不需要绑定EIP。

图 17-5 获取链接



----结束

获取驱动链接-OBS 地址

步骤1 将驱动上传到对象存储服务OBS中，并将驱动文件设置为公共读。

说明

节点重启时会重新下载驱动进行安装，请保证驱动的OBS桶链接长期有效。

步骤2 在桶列表单击待操作的桶，进入“概览”页面。

步骤3 在左侧导航栏，单击“对象”。

步骤4 单击目标对象名称，在对象详情页复制驱动链接。

----结束

组件说明

表 17-30 gpu 插件组件

容器组件	说明	资源类型
nvidia-driver-installer	该容器运行在GPU节点上，负责安装NVIDIA驱动。	Daemon Set

17.9 Volcano Scheduler

插件简介

volcano是一个基于Kubernetes的批处理平台，提供了机器学习、深度学习、生物信息学、基因组学及其他大数据应用所需要而Kubernetes当前缺失的一系列特性。

volcano提供了高性能任务调度引擎、高性能异构芯片管理、高性能任务运行管理等通用计算能力，通过接入AI、大数据、基因、渲染等诸多行业计算框架服务终端用户。

volcano针对计算型应用提供了作业调度、作业管理、队列管理等多项功能，主要特性包括：

- 丰富的计算框架支持：通过CRD提供了批量计算任务的通用API，通过提供丰富的插件及作业生命周期高级管理，支持TensorFlow，MPI，Spark等计算框架容器化运行在Kubernetes上。
- 高级调度：面向批量计算、高性能计算场景提供丰富的高级调度能力，包括成组调度，优先级抢占、装箱、资源预留、任务拓扑关系等。
- 队列管理：支持分队列调度，提供队列优先级、多级队列等复杂任务调度能力。

目前volcano项目已经在Github开源，项目开源地址：**https://github.com/volcano-sh/volcano**

本文介绍如何在CCE集群中安装及配置volcano插件，具体使用方法请参见**volcano调度**。

说明

在使用volcano作为调度器时，建议将集群中所有工作负载都使用volcano进行调度，以避免多调度器同时工作导致的一些调度资源冲突问题。

安装插件

步骤1 登录CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件管理”，在右侧找到Volcano Scheduler，单击“安装”。

步骤2 在安装插件页面，设置“规格配置”。

表 17-31 插件规格配置

参数	参数说明
插件规格	该插件可配置“单实例”、“高可用”或“自定义”规格。

参数	参数说明
实例数	选择上方插件规格后，显示插件中的实例数。 选择“自定义”规格时，您可根据需求调整插件实例数。
多可用区部署	• 优先模式：优先将插件的Deployment实例调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将调度到单可用区。 • 强制模式：插件Deployment实例强制调度到不同可用区的节点上，如集群下节点不满足多可用区，插件实例将无法全部运行。
容器	选择插件规格后，显示插件容器的CPU和内存配额。 选择“自定义”规格时，volcano-controller和volcano-scheduler的建议值如下： • 小于100个节点，可使用默认配置，即CPU的申请值为500m，限制值为2000m；内存的申请值为500Mi，限制值为2000Mi。 • 高于100个节点，每增加100个节点（10000个Pod），建议CPU的申请值增加500m，内存的申请值增加1000Mi；CPU的限制值建议比申请值多1500m，内存的限制值建议比申请值多1000Mi。 **说明** 申请值推荐计算公式： – CPU申请值：计算“目标节点数 * 目标Pod规模”的值，并在**表17-32**中根据“集群节点数 * Pod规模”的计算值进行插值查找，向上取最接近规格的申请值及限制值。 例如2000节点和2w个Pod的场景下，“目标节点数 * 目标Pod规模”等于4000w，向上取最接近的规格为700/7w（“集群节点数 * Pod规模”等于4900w），因此建议CPU申请值为4000m，限制值为5500m。 – 内存申请值：建议每1000个节点分配2.4G内存，每1w个Pod分配1G内存，二者叠加进行计算。（该计算方法相比**表17-32**中的建议值会存在一定的误差，通过查表或计算均可） 即：内存申请值 = 目标节点数/1000 * 2.4G + 目标Pod规模/1w * 1G。 例如2000节点和2w个Pod的场景下，内存申请值 = 2 * 2.4G + 2 * 1G = 6.8G

表 17-32 volcano-controller 和 volcano-scheduler 的建议值

集群节点数/Pod规模	CPU Request(m)	CPU Limit(m)	Memory Request(Mi)	Memory Limit(Mi)
50/5k	500	2000	500	2000
100/1w	1000	2500	1500	2500
200/2w	1500	3000	2500	3500
300/3w	2000	3500	3500	4500

集群节点数/Pod规模	CPU Request(m)	CPU Limit(m)	Memory Request(Mi)	Memory Limit(Mi)
400/4w	2500	4000	4500	5500
500/5w	3000	4500	5500	6500
600/6w	3500	5000	6500	7500
700/7w	4000	5500	7500	8500

步骤3 设置插件支持的“参数配置”。

配置volcano默认调度器配置参数，详情请参见**表17-34**。

```
colocation_enable: ''
default_scheduler_conf:
  actions: 'allocate, backfill'
  tiers:
    - plugins:
        - name: 'priority'
        - name: 'gang'
        - name: 'conformance'
        - name: 'lifecycle'
          arguments:
            lifecycle.MaxGrade: 10
            lifecycle.MaxScore: 200.0
            lifecycle.SaturatedTresh: 1.0
            lifecycle.WindowSize: 10
    - plugins:
        - name: 'drf'
        - name: 'predicates'
        - name: 'nodeorder'
    - plugins:
        - name: 'cce-gpu-topology-predicate'
        - name: 'cce-gpu-topology-priority'
        - name: 'cce-gpu'
    - plugins:
        - name: 'nodelocalvolume'
        - name: 'nodeemptydirvolume'
        - name: 'nodeCSIscheduling'
        - name: 'networkresource'
tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 60
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 60
```

表 17-33 volcano 高级配置参数说明

插件	功能	参数说明	用法演示
default_scheduler_conf	负责Pod调度的组件配置，由一系列action和plugin组成。具有高度的可扩展性，您可以根据需要实现自己的action和plugin。	主要包括actions和tiers两部分： • actions：定义调度器需要执行的action类型及顺序。 • tiers：配置plugin列表。	-
actions	定义了调度各环节中需要执行的动作，action的配置顺序就是scheduler的执行顺序。详情请参见**Actions**。 调度器会遍历所有的待调度Job，按照定义的次序依次执行enqueue、allocate、preempt、backfill等动作，为每个Job找到一个最合适的节点。	支持的参数值： • enqueue：负责通过一系列的过滤算法筛选出符合要求的待调度任务并将它们送入待调度队列。经过这个action，任务的状态将由pending变为inqueue。 • allocate：负责通过一系列的预选和优选算法筛选出最适合的节点。 • preempt：负责根据优先级规则为同一队列中高优先级任务执行抢占调度。 • backfill：负责将处于pending状态的任务尽可能的调度下去以保证节点资源的最大化利用。	actions: 'allocate, backfill' **说明** 配置action时，preempt和enqueue不可同时使用。
plugins	根据不同场景提供了action中算法的具体实现细节，详情请参见**Plugins**。	支持的参数值请参见**表 17-34**。	-
tolerations	插件实例对节点污点的容忍度设置。	默认配置下，插件实例可以运行在拥有“node.kubernetes.io/not-ready”或“node.kubernetes.io/unreachable”污点，且污点效果值为NoExecute的节点上，但会在60秒后被驱逐。	tolerations: - effect: NoExecute key: node.kubernetes.io/not-ready operator: Exists tolerationSeconds: 60 - effect: NoExecute key: node.kubernetes.io/unreachable operator: Exists tolerationSeconds: 60

表 17-34 支持的 Plugins 列表

插件	功能	参数说明	用法演示
binpack	将Pod调度到资源使用较高的节点（尽量不往空白节点分配），以减少资源碎片。	arguments参数： • binpack.weight：binpack插件本身在所有插件打分中的权重。 • binpack.cpu：CPU资源在所有资源中的权重，默认是1。 • binpack.memory：内存资源在所有资源中的权重，默认是1。 • binpack.resources：Pod请求的其他自定义资源类型，例如nvidia.com/gpu。可添加多个并用英文逗号隔开。 • binpack.resources.*<your_resource>*：自定义资源在所有资源中的权重，可添加多个类型的资源，其中*<your_resource>*为binpack.resources参数中定义的资源类型。例如binpack.resources.nvidia.com/gpu。	- 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
conformance	跳过关键Pod（比如在kube-system命名空间的Pod），防止这些Pod被驱逐。	-	- plugins: - name: 'priority' - name: 'gang' enablePreemptable: false - **name: 'conformance'**

插件	功能	参数说明	用法演示
lifecycle	通过统计业务伸缩的规律，将有相近生命周期的Pod优先调度到同一节点，配合autoscaler的水平扩缩容能力，快速缩容释放资源，节约成本并提高资源利用率。 1. 统计业务负载中Pod的生命周期，将有相近生命周期的Pod调度到同一节点 2. 对配置了自动扩缩容策略的集群，通过调整节点的缩容注解，优先缩容使用率低的节点	arguments参数： ● lifecycle.WindowSize：为int型整数，不小于1，默认为10。 记录副本数变更的次数，负载变化规律、周期性明显时可适当调低；变化不规律，副本数频繁变化需要调大。若过大会导致学习周期变长，记录事件过多。 ● lifecycle.MaxGrade：为int型整数，不小于3，默认为3。 副本分档数，如设为3，代表分为高中低档。负载变化规律、周期性明显时可适当调低；变化不规律，需要调大。若过小会导致预测的生命周期不够准确。 ● lifecycle.MaxScore：为float64浮点数，不小于50.0，默认为200.0。 lifecycle插件的最大得分，等效于插件权重。 ● lifecycle.SaturatedTresh：为float64浮点数，小于0.5时取值为0.5；大于1时取值为1，默认为0.8。 用于判断节点利用率是否过高的阈值，当超过该阈值，调度器会优先调度作业至其他节点。	- plugins: - name: priority - name: gang enablePreemptable: false - name: conformance - **name: lifecycle** arguments: lifecycle.MaxGrade: 10 lifecycle.MaxScore: 200.0 lifecycle.SaturatedTresh: 1.0 lifecycle.WindowSize: 10 **说明** ● 对不希望被缩容的节点，需要手动标记长周期节点，为节点添加volcano.sh/long-lifecycle-node: true的annotation。对未标记节点，lifecycle插件将根据节点上负载的生命周期自动标记。 ● MaxScore默认值200.0相当于其他插件权重的两倍，当lifecycle插件效果不明显或与其他插件冲突时，需要关闭其他插件，或将MaxScore调大。 ● 调度器重启后，lifecycle插件需要重新记录负载的变化状况，需要统计数个周期后才能达到最优调度效果。

插件	功能	参数说明	用法演示
gang	将一组Pod看做一个整体进行资源分配。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。 **说明** 使用gang调度策略时，当集群剩余的资源大于等于Job的最小运行数量的1/2、但小于Job的最小运行数量时，不会触发autoscaler扩容。	• enablePreemptable： – true：表示开启抢占。 – false：表示不开启抢占。 • enableJobStarving： – true: 表示按照Job的minAvailable进行抢占。 – false：表示按照Job的replicas进行抢占。 **说明** – Kubernetes原生工作负载（如Deployment）的minAvailable默认值为1，建议配置enableJobStarving: false。 – AI大数据场景，创建vcjob时可指定minAvailable值，推荐配置enableJobStarving: true。 – Volcano 1.11.5之前的版本enableJobStarving默认为true，1.11.5之后的版本默认配置为false。	- plugins: - name: priority **- name: gang** **enablePreemptable: false** **enableJobStarving: false** - name: conformance
priority	根据自定义的负载优先级进行调度。	-	- plugins: **- name: priority** - name: gang enablePreemptable: false - name: conformance
overcommit	将集群的资源放到一定倍数后调度，提高负载入队效率。负载都是deployment的时候，建议去掉此插件或者设置扩大因子为2.0。 **说明** 该插件在volcano 1.6.5及以上版本中支持使用。	arguments参数： • overcommit-factor：扩大因子，默认是1.2。	- plugins: **- name: overcommit** arguments: overcommit-factor: 2.0

插件	功能	参数说明	用法演示

drf	DRF调度算法（Dominant Resource Fairness）可以根据作业使用的主导资源份额进行调度，资源份额较小的作业将具有更高优先级。	-	- plugins:
			  - **name: 'drf'**
			  - name: 'predicates'
			  - name: 'nodeorder'

predicates	预选节点的常用算法，包括节点亲和、Pod亲和、污点容忍、Node重复，volume limits，volume zone匹配等一系列基础算法。	-	- plugins:
			  - name: 'drf'
			  - **name: 'predicates'**
			  - name: 'nodeorder'

插件	功能	参数说明	用法演示

nodeorder	优选节点的常用算法，通过模拟分配从各个维度为节点打分，找到最适合当前作业的节点。	打分参数：
		● nodeaffinity.weight：节点亲和性优先调度，默认值是1。
		● podaffinity.weight：Pod亲和性优先调度，默认值是1。
		● leastrequested.weight：资源分配最少的节点优先，默认值是1。
		● balancedresource.weight：节点上面的不同资源分配平衡的优先，默认值是1。
		● mostrequested.weight：资源分配最多的节点优先，默认值是0。
		● tainttoleration.weight：污点容忍高的优先调度，默认值是1。
		● imagelocality.weight：节点上面有Pod需要镜像的优先调度，默认值是1。
		● selectorspread.weight：把Pod均匀调度到不同的节点上，默认值是0。
		● podtopologyspread.weight：Pod拓扑调度，默认值是2。
			- plugins:
			  - **name: nodeorder**
			  arguments:
			  leastrequested.weight: 1
			  mostrequested.weight: 0
			  nodeaffinity.weight: 1
			  podaffinity.weight: 1
			  balancedresource.weight: 1
			  tainttoleration.weight: 1
			  imagelocality.weight: 1
			  volumebinding.weight: 1
			  podtopologyspread.weight: 2

cce-gpu-topology-predicate	GPU拓扑调度预选算法	-	- plugins:
			  - **name: 'cce-gpu-topology-predicate'**
			  - name: 'cce-gpu-topology-priority'
			  - name: 'cce-gpu'

cce-gpu-topology-priority	GPU拓扑调度优选算法	-	- plugins:
			  - name: 'cce-gpu-topology-predicate'
			  - **name: 'cce-gpu-topology-priority'**
			  - name: 'cce-gpu'

插件	功能	参数说明	用法演示

cce-gpu	结合CCE的GPU插件支持GPU资源分配，支持小数GPU配置。	-	- plugins:
			  - name: 'cce-gpu-topology-predicate'
			  - name: 'cce-gpu-topology-priority'
			  - **name: 'cce-gpu'**

numa-aware	NUMA亲和性调度。	arguments参数：
		● weight：插件的权重。
			- plugins:
			  - name: 'nodelocalvolume'
			  - name: 'nodeemptydirvolume'
			  - name: 'nodeCSIscheduling'
			  - name: 'networkresource'
			  arguments:
			  NetworkType: vpc-router
			  - **name: numa-aware**
			  arguments:
			  weight: 10

networkresource	支持预选过滤ENI需求节点，参数由CCE传递，不需要手动配置。	arguments参数：
		● NetworkType：网络类型（eni或者vpc-router类型）。
			- plugins:
			  - name: 'nodelocalvolume'
			  - name: 'nodeemptydirvolume'
			  - name: 'nodeCSIscheduling'
			  - **name: 'networkresource'**
			  arguments:
			  NetworkType: vpc-router

nodelocalvolume	支持预选过滤不符合local volume需求的节点。	-	- plugins:
			  - **name: 'nodelocalvolume'**
			  - name: 'nodeemptydirvolume'
			  - name: 'nodeCSIscheduling'
			  - name: 'networkresource'

nodeemptydirvolume	支持预选过滤不符合emptydir需求的节点。	-	- plugins:
			  - name: 'nodelocalvolume'
			  - **name: 'nodeemptydirvolume'**
			  - name: 'nodeCSIscheduling'
			  - name: 'networkresource'

nodeCSIscheduling	支持预选过滤everest组件异常的节点。	-	- plugins:
			  - name: 'nodelocalvolume'
			  - name: 'nodeemptydirvolume'
			  - **name: 'nodeCSIscheduling'**
			  - name: 'networkresource'

步骤4 单击“安装”。

----结束

组件说明

表 17-35 volcano 组件

容器组件	说明	资源类型
volcano-scheduler	负责Pod调度。	Deployment

容器组件	说明	资源类型
volcano-controller	负责CRD资源的同步。	Deployment
volcano-admission	Webhook server端，负责Pod、Job等资源的校验和更改。	Deployment
volcano-agent	云原生混部agent，负责节点QoS保障、CPU Burst和动态资源超卖等。	DaemonSet
resource-exporter	负责上报节点的NUMA拓扑信息。	DaemonSet

在控制台中修改 volcano-scheduler 配置

volcano-scheduler是负责Pod调度的组件，它由一系列action和plugin组成。action定义了调度各环节中需要执行的动作；plugin根据不同场景提供了action 中算法的具体实现细节。volcano-scheduler具有高度的可扩展性，您可以根据需要实现自己的action和plugin。

volcano允许用户在安装，升级，编辑时，编写volcano调度器配置信息，并将配置内容同步到volcano-scheduler-configmap里。

当前小节介绍如何使用自定义配置，以便用户让volcano-scheduler能更适合自己的场景。

说明

仅volcano 1.7.1及以上版本支持该功能。在新版插件界面上合并了原plugins.eas_service和resource_exporter_enable等选项，以新选项default_scheduler_conf代替。

您可登录CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件管理”，在右侧找到**volcano**，单击“安装”或“升级”，并在“参数配置”中设置volcano调度器配置参数。

- 使用resource_exporter配置，示例如下：

```
{
  "ca_cert": "",
  "default_scheduler_conf": {
    "actions": "allocate, backfill",
    "tiers": [
      {
        "plugins": [
          {
            "name": "priority"
          },
          {
            "name": "gang"
          },
          {
            "name": "conformance"
          }
        ]
      },
      {
        "plugins": [
          {
            "name": "drf"
          },
```

```
            {
               "name": "predicates"
            },
            {
               "name": "nodeorder"
            }
         ]
      },
      {
         "plugins": [
            {
               "name": "cce-gpu-topology-predicate"
            },
            {
               "name": "cce-gpu-topology-priority"
            },
            {
               "name": "cce-gpu"
            },
            {
               "name": "numa-aware" # add this also enable resource_exporter
            }
         ]
      },
      {
         "plugins": [
            {
               "name": "nodelocalvolume"
            },
            {
               "name": "nodeemptydirvolume"
            },
            {
               "name": "nodeCSIscheduling"
            },
            {
               "name": "networkresource"
            }
         ]
      }
   ]
  },
  "server_cert": "",
  "server_key": ""
}
```

开启后可以同时使用volcano-scheduler的numa-aware插件功能和resource_exporter功能。

- 使用eas_service配置，示例如下：

```
{
  "ca_cert": "",
  "default_scheduler_conf": {
     "actions": "allocate, backfill",
     "tiers": [
        {
           "plugins": [
              {
                 "name": "priority"
              },
              {
                 "name": "gang"
              },
              {
                 "name": "conformance"
              }
           ]
        },
        {
           "plugins": [
```

```
                {
                    "name": "drf"
                },
                {
                    "name": "predicates"
                },
                {
                    "name": "nodeorder"
                }
            ]
        },
        {
            "plugins": [
                {
                    "name": "cce-gpu-topology-predicate"
                },
                {
                    "name": "cce-gpu-topology-priority"
                },
                {
                    "name": "cce-gpu"
                },
                {
                    "name": "eas",
                    "custom": {
                        "availability_zone_id": "",
                        "driver_id": "",
                        "endpoint": "",
                        "flavor_id": "",
                        "network_type": "",
                        "network_virtual_subnet_id": "",
                        "pool_id": "",
                        "project_id": "",
                        "secret_name": "eas-service-secret"
                    }
                }
            ]
        },
        {
            "plugins": [
                {
                    "name": "nodelocalvolume"
                },
                {
                    "name": "nodeemptydirvolume"
                },
                {
                    "name": "nodeCSIscheduling"
                },
                {
                    "name": "networkresource"
                }
            ]
        }
    ]
  },
  "server_cert": "",
  "server_key": ""
}
```


- 使用ief配置，示例如下：


```
{
  "ca_cert": "",
  "default_scheduler_conf": {
    "actions": "allocate, backfill",
    "tiers": [
      {
        "plugins": [
          {
            "name": "priority"
```

```
                },
                {
                    "name": "gang"
                },
                {
                    "name": "conformance"
                }
            ]
        },
        {
            "plugins": [
                {
                    "name": "drf"
                },
                {
                    "name": "predicates"
                },
                {
                    "name": "nodeorder"
                }
            ]
        },
        {
            "plugins": [
                {
                    "name": "cce-gpu-topology-predicate"
                },
                {
                    "name": "cce-gpu-topology-priority"
                },
                {
                    "name": "cce-gpu"
                },
                {
                    "name": "ief",
                    "enableBestNode": true
                }
            ]
        },
        {
            "plugins": [
                {
                    "name": "nodelocalvolume"
                },
                {
                    "name": "nodeemptydirvolume"
                },
                {
                    "name": "nodeCSIscheduling"
                },
                {
                    "name": "networkresource"
                }
            ]
        }
    ]
  },
  "server_cert": "",
  "server_key": ""
}
```


保留原 volcano-scheduler-configmap 配置

假如在某场景下希望插件升级后时沿用原配置，可参考以下步骤：

步骤1 查看原volcano-scheduler-configmap配置，并备份。

示例如下：

```
# kubectl edit cm volcano-scheduler-configmap -n kube-system
apiVersion: v1
data:
  default-scheduler.conf: |-
    actions: "enqueue, allocate, backfill"
    tiers:
    - plugins:
      - name: priority
      - name: gang
      - name: conformance
    - plugins:
      - name: drf
      - name: predicates
      - name: nodeorder
      - name: binpack
        arguments:
          binpack.cpu: 100
          binpack.weight: 10
          binpack.resources: nvidia.com/gpu
          binpack.resources.nvidia.com/gpu: 10000
    - plugins:
      - name: cce-gpu-topology-predicate
      - name: cce-gpu-topology-priority
      - name: cce-gpu
    - plugins:
      - name: nodelocalvolume
      - name: nodeemptydirvolume
      - name: nodeCSIscheduling
      - name: networkresource
```

步骤2 在控制台“参数配置”中填写自定义修改的内容：

```
{
    "ca_cert": "",
    "default_scheduler_conf": {
        "actions": "enqueue, allocate, backfill",
        "tiers": [
            {
                "plugins": [
                    {
                        "name": "priority"
                    },
                    {
                        "name": "gang"
                    },
                    {
                        "name": "conformance"
                    }
                ]
            },
            {
                "plugins": [
                    {
                        "name": "drf"
                    },
                    {
                        "name": "predicates"
                    },
                    {
                        "name": "nodeorder"
                    },
                    {
                        "name": "binpack",
                        "arguments": {
                            "binpack.cpu": 100,
                            "binpack.weight": 10,
                            "binpack.resources": "nvidia.com/gpu",
                            "binpack.resources.nvidia.com/gpu": 10000
                        }
                    }
```

```
            ]
        },
        {
            "plugins": [
                {
                    "name": "cce-gpu-topology-predicate"
                },
                {
                    "name": "cce-gpu-topology-priority"
                },
                {
                    "name": "cce-gpu"
                }
            ]
        },
        {
            "plugins": [
                {
                    "name": "nodelocalvolume"
                },
                {
                    "name": "nodeemptydirvolume"
                },
                {
                    "name": "nodeCSIscheduling"
                },
                {
                    "name": "networkresource"
                }
            ]
        }
    ]
  },
  "server_cert": "",
  "server_key": ""
}
```

说明

使用该功能时会覆盖原volcano-scheduler-configmap中内容，所以升级时务必检查是否在volcano-scheduler-configmap做过修改。如果是，需要把修改内容同步到升级界面里。

----结束

volcano 插件卸载说明

卸载插件时，会将volcano的自定义资源（表17-36）全部清理，已创建的相关资源也将同步删除，重新安装插件不会继承和恢复卸载之前的任务信息。如集群中还存在正在使用的volcano自定义资源，请谨慎卸载插件。

表 17-36 volcano 自定义资源

名称	API组	API版本	资源级别
Command	bus.volcano.sh	v1alpha1	Namespaced
Job	batch.volcano.sh	v1alpha1	Namespaced
Numatopology	nodeinfo.volcano.sh	v1alpha1	Cluster
PodGroup	scheduling.volcano.sh	v1beta1	Namespaced

名称	API组	API版本	资源级别
Queue	scheduling.volcano.sh	v1beta1	Cluster

17.10 CCE Container Storage (FlexVolume)

插件简介

storage-driver是一款云存储驱动插件，北向遵循标准容器平台存储驱动接口。实现Kubernetes Flex Volume标准接口，提供容器使用EVS块存储、SFS文件存储、OBS 对象存储、SFS Turbo 极速文件存储的能力。通过安装升级云存储插件可以实现云存储功能的快速安装和更新升级。

该插件为系统资源插件，kubernetes 1.13及以下版本的集群在创建时默认安装。

约束与限制

- 在CCE所创的集群中，Kubernetes v1.15.11版本是Flexvolume插件（storage-driver）被CSI插件（everest）兼容接管的过渡版本，v1.17及之后版本的集群中将彻底去除对Flexvolume插件（storage-driver）的支持，请您将Flexvolume插件的使用切换到CSI插件上。
- CSI插件（everest）兼容接管Flexvolume插件（storage-driver）后，原有功能保持不变，但请注意不要新建Flexvolume插件（storage-driver）的存储，否则将导致部分存储功能异常。
- 本插件仅支持在**v1.13及以下版本**的集群中安装，v1.15及以上版本的集群在创建时默认安装everest插件。

  说明

  在**v1.13及以下版本**的集群中，当存储功能有升级或者BUG修复时，用户无需升级集群或新建集群来升级存储功能，仅需安装或升级storage-driver插件。

安装插件

本插件为系统默认安装，若因特殊情况卸载后，可参照如下步骤重新安装。

未安装storage-driver插件的集群，可参考如下步骤进行安装：

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“插件管理”，在右侧找到CCE Container Storage (Flexvolume)，单击“安装”。

步骤2 云存储插件暂未开放可配置参数，直接单击“安装”。

----结束

18 模板（Helm Chart）

18.1 概述

CCE提供了管理Helm Chart（模板）的控制台，能够帮助您方便的使用模板部署应用，并在控制台上管理应用。

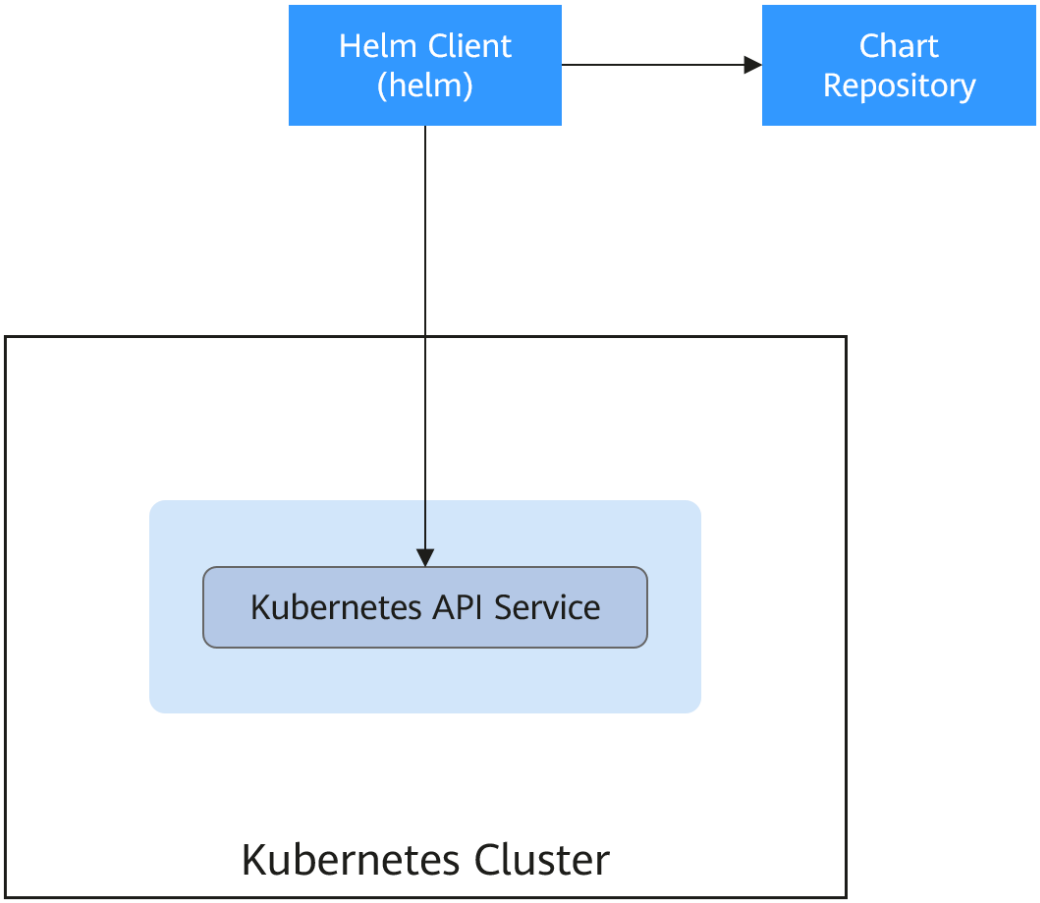
Helm

Helm是Kubernetes的包管理器，主要用来管理Charts。Helm Chart是用来封装Kubernetes原生应用程序的一系列YAML文件。可以在您部署应用的时候自定义应用程序的一些Metadata，以便于应用程序的分发。对于应用发布者而言，可以通过Helm打包应用、管理应用依赖关系、管理应用版本并发布应用到软件仓库。对于使用者而言，使用Helm后不用需要编写复杂的应用部署文件，可以以简单的方式在Kubernetes上查找、安装、升级、回滚、卸载应用程序。

Helm和Kubernetes之间的关系可以如下类比：

- Helm <–> Kubernetes
- Apt <–> Ubuntu
- Yum <–> CentOS
- Pip <–> Python

Helm的整体架构如下图：



Kubernetes的应用编排存在着一些问题，Helm可以用来解决这些问题，如下：

- 管理、编辑与更新大量的Kubernetes配置文件。
- 部署一个含有大量配置文件的复杂Kubernetes应用。
- 分享和复用Kubernetes配置和应用。
- 参数化配置模板支持多个环境。
- 管理应用的发布：回滚、diff和查看发布历史。
- 控制一个部署周期中的某一些环节。
- 发布后的测试验证。

18.2 通过模板部署应用

在CCE控制台上，您可以上传Helm模板包，然后在控制台安装部署，并对部署的实例进行管理。

约束与限制

- 单个用户可以上传模板的个数有限制，请以各个Region控制台界面中提示的实际值为准。
- 模板若存在多个版本，则消耗对应数量的模板配额。

- 由于模板的操作权限同时具有较高的集群操作权限，因此租户应当谨慎授予用户对于模板生命周期管理的权限，包括上传模板的权限，以及创建、删除和更新模板实例的权限。

模板包规范

以下以redis为例，在准备redis模板包时根据模板包规范制作模板包。

- **命名要求**

  模板包命名格式为：**{name}-{version}**.tgz，其中**{version}**为版本号，格式为“主版本号.次版本号.修订号”，如redis-0.4.2.tgz。

  说明

  模板名称{name}的长度不能超过64个字符。

  版本号需遵循**语义化版本**规则。

  - 主版本号、次版本号为必选，修订号为可选。
  - 主版本号、次版本号、修订号的数值为整数，均需要≥0，且≤99。

- **目录结构**

  模板包的目录结构如下所示：

```
redis/
  templates/
  values.yaml
  README.md
  Chart.yaml
  .helmignore
```

  目录说明如**表18-1**所示，带*的为必选项：

  表 18-1 模板包目录说明

参数	参数说明
* templates	用于存放所有的template（模板）文件。
* values.yaml	用于描述template文件所需的配置参数。 **须知** 定义template文件配置参数时，请注意此处定义的“镜像地址”务必和容器镜像仓库中对应的镜像地址保持一致。否则创建工作负载会异常，提示镜像拉取失败。 镜像地址获取方法如下：在CCE控制台，单击左侧导航栏的“镜像仓库”，进入容器镜像服务控制台。在“我的镜像 > 自有镜像”中，单击已上传镜像的名称，在“镜像版本 ”页签的“下载指令”栏中即可获取镜像地址，单击 按钮即可复制该指令。
README.md	一个markdown文件，包括： • 描述Chart提供的工作负载或服务。 • 运行Chart的前提。 • 解释values.yaml文件中的配置。 • 安装和配置Chart的相关信息。
* Chart.yaml	模板的基本信息说明。 注：Helm v3版本apiVersion从v1切换到了v2。

参数	参数说明
.helmignore	设定在工作负载安装时不需要读取templates的某些文件或数据。

上传模板

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“模板管理”，在右上角单击“上传模板”。

步骤2 单击“添加文件”，选中待上传的工作负载包后，单击“上传”。

说明

由于上传模板时创建OBS桶的命名规则由cce-charts-{region}-**{domain_name}**变为cce-charts-{region}-**{domain_id}**，其中旧命名规则中的domain_name系统会做base64转化并取前63位，如果您在现有命名规则的OBS桶中找不到模板，请在旧命名规则的桶中进行查找。

----结束

创建模板实例

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“模板管理”。

步骤2 在“我的模板”页签中，单击目标模板下的“安装”。

步骤3 参照表18-2设置安装工作负载参数。

表 18-2 安装工作负载参数说明

参数	参数说明
实例名称	新建模板实例名称，命名必须唯一。
命名空间	指定部署的命名空间。
选择版本	选择模板的版本。
配置文件	用户可以导入values.yaml文件，导入后可替换模板包中的values.yaml文件；也可直接在配置框中在线编辑模板参数。 **说明** 此处导入的values.yaml文件需符合yaml规范，即KEY:VALUE格式。对于文件中的字段不做任何限制。 导入的value.yaml的key值必须与所选的模板包的values.yaml保持一致，否则不会生效。即key不能修改。 1. 单击“添加文件”。 2. 选择对应的values.yaml文件，单击“打开”。

步骤4 配置完成后，单击“安装”。

在“模板实例”页签下可以查看模板实例的安装情况。

----结束

升级模板工作负载

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“模板管理”，在右侧选择“模板实例”页签。

步骤2 单击待升级工作负载后的“升级”，设置升级模板工作负载的参数。

步骤3 选择对应的模板版本。

步骤4 参照界面提示修改模板参数。单击“升级”，再单击“提交”。

步骤5 单击“返回模板实例列表”，模板状态为“升级成功”时，表明工作负载升级成功。

----结束

回退模板工作负载

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“模板管理”，在右侧选择“模板实例”页签。

步骤2 单击待回退工作负载后的“回退”，选择要回退的工作负载版本，单击“回退”。

模板工作负载列表中，状态为“回退成功”时，表明工作负载回退成功。

----结束

卸载模板工作负载

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“模板管理”，在右侧选择“模板实例”页签。

步骤2 单击待卸载模板实例后的“更多 > 卸载”，确认待卸载模板实例后，单击“是”。模板实例卸载后不能恢复，请谨慎操作。

----结束

18.3 Helm v2 与 Helm v3 的差异及适配方案

随着Helm v2 发布最终版本Helm 2.17.0，Helm v3 现在已是 Helm 开发者社区支持的唯一标准。为便于管理，建议用户尽快将模板切换至**Helm v3格式**。

当前社区从Helm v2演进到Helm v3，主要有以下变化：

1. **移除tiller**

   Helm v3 使用更加简单和灵活的架构，移除了 tiller，直接通过kubeconfig连接apiserver，简化安全模块，降低了用户的使用壁垒。

2. **改进了升级策略，采用三路策略合并补丁**

   Helm v2 使用双路策略合并补丁。在升级过程中，会对比最近一次发布的chart manifest和本次发布的chart manifest的差异，来决定哪些更改会应用到Kubernetes资源中。如果更改是集群外带的（比如通过kubectl edit），则修改不会被Helm识别和考虑。结果就是资源不会回滚到之前的状态。

   Helm v3 使用三路策略来合并补丁，Helm在生成一个补丁时，会考虑之前老的manifest的活动状态。因此，Helm在使用老的chart manifest生成新补丁时会考虑当前活动状态，并将其与之前老的 manifest 进行比对，并再比对新的 manifest 是否有改动，并进行自动补全，以此来生成最终的更新补丁。

详情及示例请见Helm官方文档：**https://v3.helm.sh/docs/faq/changes_since_helm2**

3. **默认存储驱动程序更改为secrets**

   Helm v2 默认情况下使用 ConfigMaps 存储发行信息，而在 Helm v3 中默认使用 Secrets。

4. **发布名称限制在namespace范围内**

   Helm v2 只使用tiller 的namespace 作为release信息的存储，这样全集群的release名字都不能重复。Helm v3只会在release安装的所在namespace记录对应的信息，这样release名称可在不同命名空间重用。应用和release命名空间一致。

5. **校验方式改变**

   Helm v3 对模板格式的校验更加严格，如Helm v3 将chart.yaml的apiVersion从v1切换到v2，针对Helm v2的chart.yaml，强要求指定apiVersion为v1。可安装Helm v3客户端后，通过执行helm lint命令校验模板格式是否符合v3规范。

   适配方案：根据Helm官方文档 **https://helm.sh/docs/topics/charts/**适配Helm v3模板，apiVersion字段必填。

6. **废弃crd-install**

   Helm v3删除了crd-install hook， 并用chart中的crds目录替换。需要注意的是，crds目录中的资源只有在release安装时会部署，升级时不会更新，删除时不会卸载crds目录中的资源。若crd已存在，则重复安装不会报错。

   适配方案：根据Helm官方文档 **https://helm.sh/docs/chart_best_practices/custom_resource_definitions/**, 当前可使用crds目录或者将crd定义单独放入chart。考虑到目前不支持使用Helm升级或删除CRD，推荐分隔chart，将CRD定义放入chart中，然后将所有使用该CRD的资源放到另一个 chart中进行管理。

7. **未通过helm创建的资源不强制update，releaes默认不强制升级**

   Helm v3强制升级逻辑变化，不再是升级失败后走删除重建，而是直接走put更新逻辑。因此当前CCE release升级默认使用非强制更新逻辑，无法通过Patch更新的资源将导致release升级失败。若环境存在同名资源且无Helm V3的归属标记app.kubernetes.io/managed-by: Helm，则会提示资源冲突。

   适配方案：删除相关资源，并通过Helm创建。

8. **Release history数量限制更新**

   为避免release 历史版本无限增加，当前release升级默认只保留最近10个历史版本。

更多变化和详细说明请参见Helm官方文档

- Helm v2与Helm v3的区别：**https://v3.helm.sh/docs/faq/changes_since_helm2**
- Helm v2如何迁移到Helm v3：**https://helm.sh/docs/topics/v2_v3_migration**

18.4 通过 Helm v2 客户端部署应用

前提条件

在CCE中创建的Kubernetes集群已对接kubectl，具体请参见**使用kubectl连接集群**。

安装 Helm v2

本文以Helm v2.17.0为例进行演示。

如需选择其他合适的版本，请访问**https://github.com/helm/helm/releases**。

步骤1 在连接集群的虚拟机上下载Helm客户端。

```
wget https://get.helm.sh/helm-v2.17.0-linux-amd64.tar.gz
```

步骤2 解压Helm包。

```
tar -xzvf helm-v2.17.0-linux-amd64.tar.gz
```

步骤3 将helm拷贝到系统path路径下，以下为/usr/local/bin/helm。

```
mv linux-amd64/helm /usr/local/bin/helm
```

步骤4 因为Kubernetes APIServer开启了RBAC访问控制，所以需创建tiller使用的service account:tiller并给其分配cluster-admin这个集群内置的ClusterRole。按如下创建tiller的资源账户。

vim tiller-rbac.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

步骤5 部署tiller资源账户。

```
kubectl apply -f tiller-rbac.yaml
```

步骤6 初始化Helm, 部署tiller的Pod。

```
helm init --service-account tiller --skip-refresh
```

步骤7 查看状态。

```
kubectl get pod -n kube-system -l app=helm
```

回显如下

```
NAME                            READY   STATUS    RESTARTS   AGE
tiller-deploy-7b56c8dfb7-fxk5g   1/1     Running   1          23h
```

步骤8 查看helm版本。

```
# helm version
Client: &version.Version{SemVer:"v2.17.0", GitCommit:"a690bad98af45b015bd3da1a41f6218b1a451dbe",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.17.0", GitCommit:"a690bad98af45b015bd3da1a41f6218b1a451dbe",
GitTreeState:"clean"}
```

----结束

安装 Helm 模板 chart 包

CCE提供的模板不能满足要求时，可下载模板的chart包进行安装。

在https://github.com/helm/charts的stable目录中查找您需要的chart包，下载后将chart包上传至节点。

1. 下载并解压已获取的chart包，一般chart包格式为.zip。
   ```
   unzip chart.zip
   ```
2. 安装Helm模板。
   ```
   helm install aerospike/
   ```
3. 安装完成后，执行**helm list**查看已经安装的模板实例状态。

常见问题

- **执行Helm version时，提示如下错误信息：**
  ```
  Client:
  &version.Version{SemVer:"v2.17.0",
  GitCommit:"a690bad98af45b015bd3da1a41f6218b1a451dbe", GitTreeState:"clean"}
  E0718 11:46:10.132102    7023 portforward.go:332] an error occurred
  forwarding 41458 -> 44134: error forwarding port 44134 to pod
  d566b78f997eea6c4b1c0322b34ce8052c6c2001e8edff243647748464cd7919, uid : unable
  to do port forwarding: socat not found.
  Error: cannot connect to Tiller
  ```
  出现上述问题，说明未安装socat，请执行如下命令安装socat。
  ```
  yum install socat -y
  ```
- **在操作系统为EulerOS 2.9的节点执行yum install socat –y，如报如下错误：**

  No match for argument: socat

  Error: Unable to find a match: socat

  说明镜像未自带socat镜像，请手动下载rpm包，执行以下命令安装，其中rpm包名请根据实际情况进行替换：
  ```
  rpm -i socat-1.7.3.2-8.oe1.x86_64.rpm
  ```
- **socat已安装，执行Helm version时，提示如下错误信息：**
  ```
  test@local:~/k8s/helm/test$ helm version
  Client: &version.Version{SemVer:"v3.3.0",
  GitCommit:"021cb0ac1a1b2f888144ef5a67b8dab6c2d45be6", GitTreeState:"clean"}
  Error: cannot connect to Tiller
  ```
  Helm模板从“.Kube/config”中读取配置证书和kubernetes进行通讯，出现上述错误信息说明kubectl配置有误，请重新对接kubectl，具体请参见**使用kubectl连接集群**。
- **对接云存储后，存储未创建成功。**

  出现上述问题可能是创建的pvc中annotation字段导致的，请修改模板名称后再次进行安装。
- **如果kubectl没有配置好，helm install时会出现如下报错：**
  ```
  [root@prometheus-57046 ~]# helm install prometheus/ --generate-name
  WARNING: This chart is deprecated
  Error: Kubernetes cluster unreachable: Get "http://localhost:8080/version?timeout=32s": dial tcp
  [::1]:8080: connect: connection refused
  ```
  解决办法：给节点配置kubeconfig，配置方法请参见**使用kubectl连接集群**。

18.5 通过 Helm v3 客户端部署应用

前提条件

在CCE中创建的Kubernetes集群已对接kubectl，具体请参见**使用kubectl连接集群**。

安装 Helm v3

本文以Helm v3.3.0为例进行演示。

如需选择其他合适的版本，请访问**https://github.com/helm/helm/releases**。

步骤1 在连接集群的虚拟机上下载Helm客户端。

```
wget https://get.helm.sh/helm-v3.3.0-linux-amd64.tar.gz
```

步骤2 解压Helm包。

```
tar -xzvf helm-v3.3.0-linux-amd64.tar.gz
```

步骤3 将Helm拷贝到系统path路径下，以下为/usr/local/bin/helm。

```
mv linux-amd64/helm /usr/local/bin/helm
```

步骤4 查看Helm版本。

```
helm version
version.BuildInfo{Version:"v3.3.0", GitCommit:"e29ce2a54e96cd02ccfce88bee4f58bb6e2a28b6",
GitTreeState:"clean", GoVersion:"go1.13.4"}
```

----结束

安装 Helm 模板包

您可以使用Helm安装模板包（Chart），在使用Helm命令安装模板包前，您可能需要了解三大概念帮助您更好地使用Helm。

- 模板包（Chart）：模板包中含有Kubernetes应用的资源定义以及大量的配置文件。
- 仓库（Repository）：仓库是用于存放共享模板包的地方，您可以从仓库中下载模板包至本地安装，也可以选择直接在线安装。
- 实例（Release）：实例是Helm在Kubernetes集群中安装模板包后的运行结果。一个模板包通常可以在一个集群中安装多次，每次安装都会创建一个新的实例。以MySQL模板包为例，如果您想在集群中运行两个数据库，可以安装该模板包两次，每一个数据库都会拥有自己的release 和release name。

更多关于Helm命令的使用方法请参见**使用Helm**。

步骤1 从Helm官方推荐的仓库**Artifact Hub**中查找模板包，并配置Helm仓库。

```
helm repo add {repo_name} {repo_addr}
```

例如，以**WordPress模板包**为例：

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

步骤2 使用**helm install**命令安装模板包。

- 默认安装：最简单的使用方法只需传入两个参数。

  ```
  helm install {release_name} {chart_name}
  ```

  例如，以安装WordPress为例，**步骤1**添加的仓库中WordPress的模板包为bitnami/wordpress，并将实例自定义命名为my-wordpress。

```
helm install my-wordpress bitnami/wordpress
```

- 自定义安装参数：上述方法只会使用Chart中的默认配置，但很多时候需要根据安装环境自定义配置。使用**helm show values *{chart_name}***命令可以查看模板可配置的选项。例如，查看WordPress的可配置项：

  ```
  helm show values bitnami/wordpress
  ```

  通过命令行的方式，可以对指定的参数项进行覆盖：

  ```
  helm install my-wordpress bitnami/wordpress \
      --set mariadb.primary.persistence.enabled=true \
      --set mariadb.primary.persistence.storageClass=csi-disk \
      --set mariadb.primary.persistence.size=10Gi \
      --set persistence.enabled=false
  ```

步骤3 查看已安装的模板实例。

```
helm list
```

----结束

常见问题

- **执行Helm version时，提示如下错误信息：**

  ```
  Client:
  &version.Version{SemVer:"v3.3.0",
  GitCommit:"012cb0ac1a1b2f888144ef5a67b8dab6c2d45be6", GitTreeState:"clean"}
  E0718 11:46:10.132102    7023 portforward.go:332] an error occurred
  forwarding 41458 -> 44134: error forwarding port 44134 to pod
  d566b78f997eea6c4b1c0322b34ce8052c6c2001e8edff243647748464cd7919, uid : unable
  to do port forwarding: socat not found.
  Error: cannot connect to Tiller
  ```

  出现上述问题，说明未安装socat，请执行如下命令安装socat。

  ```
  yum install socat -y
  ```

- **在操作系统为EulerOS 2.9的节点执行yum install socat –y，如报如下错误：**

  ```
  No match for argument: socat
  Error: Unable to find a match: socat
  ```

  说明节点镜像未自带socat镜像，请手动下载rpm包后，执行以下命令安装，其中rpm包名请根据实际情况进行替换：

  ```
  rpm -i socat-1.7.3.2-8.oe1.x86_64.rpm
  ```

- **socat已安装，执行Helm version时，提示如下错误信息：**

  ```
  $ helm version
  Client: &version.Version{SemVer:"v3.3.0",
  GitCommit:"021cb0ac1a1b2f888144ef5a67b8dab6c2d45be6", GitTreeState:"clean"}
  Error: cannot connect to Tiller
  ```

  Helm模板从节点上的“.Kube/config”路径中读取配置证书和Kubernetes进行通讯，出现上述错误信息说明kubectl配置有误，请重新对接kubectl，具体请参见**使用kubectl连接集群**。

- **对接云存储后，存储未创建成功。**

  出现上述问题可能是创建的PVC中annotation字段导致的，请修改模板名称后再次进行安装。

- **如果kubectl没有配置好，helm install时会出现如下报错：**

  ```
  # helm install prometheus/ --generate-name
  WARNING: This chart is deprecated
  Error: Kubernetes cluster unreachable: Get "http://localhost:8080/version?timeout=32s": dial tcp
  [::1]:8080: connect: connection refused
  ```

  解决办法：给节点配置kubeconfig，配置方法请参见**使用kubectl连接集群**。

18.6 Helm v2 Release 转换成 Helm v3 Release

背景介绍

当前CCE已全面支持Helm v3版本，用户可通过本指南将已创建的v2 release转换成v3 release，从而更好地使用v3的特性。因Helm v3底层相对于Helm v2来说，一些功能已被弃用或重构，因此转换会有一定风险，需转换前进行模拟转换。

该指南参考社区文档：**https://github.com/helm/helm-2to3**

注意事项：

- Helm v2 release信息存储在configmap中，Helm v3 release信息存储在secrets中。
- 若用户通过前端console操作，在获取实例、更新实例等操作中CCE会自动尝试转换v2模板实例到v3模板实例。若用户仅在后台操作实例，需通过该指南进行转换操作。

转换流程（不使用 Helm v3 客户端）

步骤1 在CCE节点上下载helm 2to3 转换插件。

```
wget https://github.com/helm/helm-2to3/releases/download/v0.10.2/helm-2to3_0.10.2_linux_amd64.tar.gz
```

步骤2 解压插件包。

```
tar -xzvf helm-2to3_0.10.2_linux_amd64.tar.gz
```

步骤3 模拟转换。

以test-convert实例为例，执行以下命令进行转换的模拟。若出现以下提示，说明模拟转换成功。

```
# ./2to3 convert --dry-run --tiller-out-cluster -s configmaps test-convert
NOTE: This is in dry-run mode, the following actions will not be executed.
Run without --dry-run to take the actions described below:
Release "test-convert" will be converted from Helm v2 to Helm v3.
[Helm 3] Release "test-convert" will be created.
[Helm 3] ReleaseVersion "test-convert.v1" will be created.
```

步骤4 执行正式转换。若出现以下提示，说明转换成功。

```
# ./2to3 convert --tiller-out-cluster -s configmaps test-convert
Release "test-convert" will be converted from Helm v2 to Helm v3.
[Helm 3] Release "test-convert" will be created.
[Helm 3] ReleaseVersion "test-convert.v1" will be created.
[Helm 3] ReleaseVersion "test-convert.v1" created.
[Helm 3] Release "test-convert" created.
Release "test-convert" was converted successfully from Helm v2 to Helm v3.
Note: The v2 release information still remains and should be removed to avoid conflicts with the migrated v3 release.
v2 release information should only be removed using `helm 2to3` cleanup and when all releases have been migrated over.
```

步骤5 转换完成后进行v2 release资源的清理，同样先进行模拟清理，成功后正式清理v2 release资源。

模拟清理：

```
# ./2to3 cleanup --dry-run --tiller-out-cluster -s configmaps --name test-convert
NOTE: This is in dry-run mode, the following actions will not be executed.
```

```
Run without --dry-run to take the actions described below:
WARNING: "Release 'test-convert' Data" will be removed.

[Cleanup/confirm] Are you sure you want to cleanup Helm v2 data? [y/N]: y
Helm v2 data will be cleaned up.
[Helm 2] Release 'test-convert' will be deleted.
[Helm 2] ReleaseVersion "test-convert.v1" will be deleted.
```

正式清理：

```
# ./2to3 cleanup --tiller-out-cluster -s configmaps --name test-convert
WARNING: "Release 'test-convert' Data" will be removed.

[Cleanup/confirm] Are you sure you want to cleanup Helm v2 data? [y/N]: y
Helm v2 data will be cleaned up.
[Helm 2] Release 'test-convert' will be deleted.
[Helm 2] ReleaseVersion "test-convert.v1" will be deleted.
[Helm 2] ReleaseVersion "test-convert.v1" d
```

----结束

转换流程（使用 Helm v3 客户端）

步骤1 安装Helm v3客户端，参见**安装Helm v3**。

步骤2 安装转换插件。

```
# helm plugin install https://github.com/helm/helm-2to3
Downloading and installing helm-2to3 v0.10.2 ...
https://github.com/helm/helm-2to3/releases/download/v0.10.2/helm-2to3_0.10.2_linux_amd64.tar.gz
Installed plugin: 2to3
```

步骤3 查看已安装的插件，确认插件已安装。

```
# helm plugin list
NAME VERSION DESCRIPTION
2to3  0.10.2      migrate and cleanup Helm v2 configuration and releases in-place to Helm v3
```

步骤4 模拟转换。

以test-convert实例为例，执行以下命令进行转换的模拟。若出现以下相关提示，说明模拟转换成功。

```
# helm 2to3 convert --dry-run --tiller-out-cluster -s configmaps test-convert
NOTE: This is in dry-run mode, the following actions will not be executed.
Run without --dry-run to take the actions described below:
Release "test-convert" will be converted from Helm v2 to Helm v3.
[Helm 3] Release "test-convert" will be created.
[Helm 3] ReleaseVersion "test-convert.v1" will be created.
```

步骤5 执行正式转换。若出现以下提示，说明转换成功。

```
# helm 2to3 convert --tiller-out-cluster -s configmaps test-convert
Release "test-convert" will be converted from Helm v2 to Helm v3.
[Helm 3] Release "test-convert" will be created.
[Helm 3] ReleaseVersion "test-convert.v1" will be created.
[Helm 3] ReleaseVersion "test-convert.v1" created.
[Helm 3] Release "test-convert" created.
Release "test-convert" was converted successfully from Helm v2 to Helm v3.
Note: The v2 release information still remains and should be removed to avoid conflicts with the migrated v3 release.
v2 release information should only be removed using `helm 2to3` cleanup and when all releases have been migrated over.
```

步骤6 正式转换成功后，用户可通过helm list查看已转换成功的模板实例。

```
# helm list
NAME            NAMESPACE   REVISION UPDATED                         STATUS      CHART           APP VERSION
test-convert    default     1        2022-08-29 06:56:28.166918487 +0000 UTC    deployed    test-helmold-1
```

步骤7 转换完成后进行V2 release资源的清理，同样先进行模拟清理，成功后正式清理V2 release资源。

模拟清理：

```
# helm 2to3 cleanup --dry-run --tiller-out-cluster -s configmaps --name test-convert
NOTE: This is in dry-run mode, the following actions will not be executed.
Run without --dry-run to take the actions described below:
WARNING: "Release 'test-convert' Data" will be removed.

[Cleanup/confirm] Are you sure you want to cleanup Helm v2 data? [y/N]: y
Helm v2 data will be cleaned up.
[Helm 2] Release 'test-convert' will be deleted.
[Helm 2] ReleaseVersion "test-convert.v1" will be deleted.
```

正式清理：

```
# helm 2to3 cleanup --tiller-out-cluster -s configmaps --name test-convert
WARNING: "Release 'test-convert' Data" will be removed.

[Cleanup/confirm] Are you sure you want to cleanup Helm v2 data? [y/N]: y
Helm v2 data will be cleaned up.
[Helm 2] Release 'test-convert' will be deleted.
[Helm 2] ReleaseVersion "test-convert.v1" will be deleted.
[Helm 2] ReleaseVersion "test-convert.v1" deleted.
[Helm 2] Release 'test-convert' deleted.
Helm v2 data was cleaned up successfully.
```

----结束

19 权限

19.1 CCE 权限概述

CCE权限管理是在统一身份认证服务（IAM）与Kubernetes的角色访问控制（RBAC）的能力基础上，打造的细粒度权限管理功能，支持基于IAM的细粒度权限控制和IAM Token认证，支持集群级别、命名空间级别的权限控制，帮助用户便捷灵活的对租户下的IAM用户、用户组设定不同的操作权限。

如果您需要对CCE集群及相关资源进行精细的权限管理，例如限制不同部门的员工拥有部门内资源的细粒度权限，您可以使用CCE权限管理提供的增强能力进行多维度的权限管理。

本章节将介绍CCE权限管理机制及其涉及到的基本概念。如果当前账号已经能满足您的要求，您可以跳过本章节，不影响您使用CCE服务的其它功能。

CCE 支持的权限管理能力

CCE的权限管理包括“集群权限”和“命名空间权限”两种能力，能够从集群和命名空间层面对用户组或用户进行细粒度授权，具体解释如下：

- **集群权限：**是基于IAM系统策略的授权，可以通过用户组功能实现IAM用户的授权。用户组是用户的集合，通过集群权限设置可以让某些用户组操作集群（如创建/删除集群、节点、节点池、模板、插件等），而让某些用户组仅能查看集群。

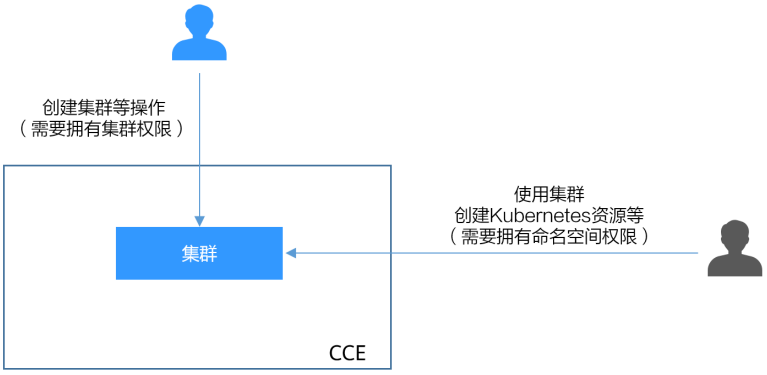
  集群权限涉及CCE非Kubernetes API，支持IAM细粒度策略相关能力。

- **命名空间权限：**是基于Kubernetes RBAC（Role-Based Access Control，基于角色的访问控制）能力的授权，通过权限设置可以让不同的用户或用户组拥有操作不同Kubernetes资源的权限。同时CCE基于开源能力进行了增强，可以支持基于IAM用户或用户组粒度进行RBAC授权、IAM token直接访问API进行RBAC认证鉴权。

  命名空间权限涉及CCE Kubernetes API，基于Kubernetes RBAC能力进行增强，支持对接IAM用户/用户组进行授权和认证鉴权，但与IAM细粒度策略独立。

CCE的权限可以从使用的阶段分为两个阶段来看，第一个阶段是创建和管理集群的权限，也就是拥有创建/删除集群、节点等资源的权限。第二个阶段是使用集群Kubernetes资源（如工作负载、Service等）的权限。

图 19-1 权限示例图



清楚了集群权限和命名空间权限后，您就可以通过这两步授权，做到精细化的权限控制。

集群权限（IAM 授权）与命名空间权限（Kubernetes RBAC 授权）的关系

拥有不同集群权限（IAM授权）的用户，其拥有的命名空间权限（Kubernetes RBAC授权）不同。**表19-1**给出了不同用户拥有的命名空间权限详情。

表 19-1 不同用户拥有的命名空间权限

用户类型	1.13及以上版本的集群
拥有Tenant Administrator权限的用户	全部命名空间权限
拥有CCE Administrator权限的IAM用户	全部命名空间权限
拥有CCE FullAccess或者CCE ReadOnlyAccess权限的IAM用户	按Kubernetes RBAC授权
拥有Tenant Guest权限的IAM用户	按Kubernetes RBAC授权

kubectl 权限说明

您可以通过**kubectl访问集群**的Kubernetes资源，那kubectl拥有哪些Kubernetes资源的权限呢?

kubectl访问CCE集群是通过集群上生成的配置文件（kubeconfig.json）进行认证，kubeconfig.json文件内包含用户信息，CCE根据用户信息的权限判断kubectl有权限访问哪些Kubernetes资源。即哪个用户获取的kubeconfig.json文件，kubeconfig.json就拥有哪个用户的信息，这样使用kubectl访问时就拥有这个用户的权限。而用户拥有的权限就是**表19-1**所示的权限。

19.2 集群权限（IAM 授权）

CCE集群权限是基于IAM**系统策略**和**自定义策略**的授权，可以通过用户组功能实现IAM用户的授权。

⚠ 注意

- 集群权限仅针对与集群相关的资源（如集群、节点等）有效，您必须确保同时配置了**命名空间权限**，才能有操作Kubernetes资源（如工作负载、Service等）的权限。
- 使用CCE控制台查看集群时，显示情况依赖于命名空间权限的设置情况，如果没有设置命名空间权限，则无法查看集群下的资源，详情请参见**CCE控制台的权限依赖**。

前提条件

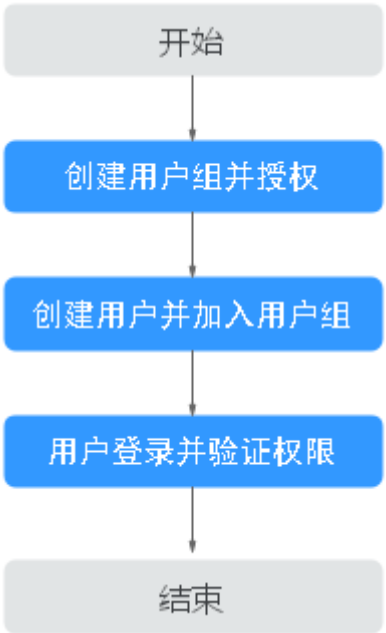
- 拥有Security Administrator（IAM除切换角色外所有权限）权限的用户（如账号默认拥有此权限），才能看见CCE控制台权限管理页面当前用户组及用户组所拥有的权限。

配置说明

CCE控制台“权限管理 > 集群权限”页面中创建用户组和具体权限设置均是跳转到IAM控制台进行具体操作，设置完后在集群权限页面能看到用户组所拥有的权限。本章节描述操作直接以IAM中操作为主，不重复介绍在CCE控制台如何跳转。

示例流程

图 19-2 给用户授予 CCE 权限流程



1. 创建用户组并授权。
   在IAM控制台创建用户组，并授予CCE权限，例如CCEReadOnlyAccess。

   说明

   CCE服务按区域部署，在IAM控制台授予CCE权限时请选择“区域级项目”。

2. 创建用户并加入用户组。
   在IAM控制台创建用户，并将其加入1中创建的用户组。
3. 用户登录并验证权限。
   新创建的用户登录控制台，切换至授权区域，验证权限：
   - 在“服务列表”中选择云容器引擎，进入CCE主界面尝试购买集群，如果无法无法成功操作（假设当前权限仅包含CCEReadOnlyAccess），表示“CCEReadOnlyAccess”已生效。
   - 在“服务列表”中选择除云容器引擎外（假设当前策略仅包含CCEReadOnlyAccess）的任一服务，若提示权限不足，表示“CCEReadOnlyAccess”已生效。

系统角色

角色是IAM最初提供的一种根据用户的工作职能定义权限的粗粒度授权机制。该机制以服务为粒度，提供有限的服务相关角色用于授权。角色并不能满足用户对精细化授权的要求，无法完全达到企业对权限最小化的安全管控要求。

IAM中预置的CCE系统角色为**CCEAdministrator**，给用户组授予该系统角色权限时，必须同时勾选该角色依赖的其他策略才会生效，例如Tenant Guest、Server Administrator、ELB Administrator、OBS Administrator、SFS Administrator、SWR Admin、APM FullAccess。

系统策略

IAM中预置的CCE系统策略当前包含**CCEFullAccess**和**CCEReadOnlyAccess**两种策略：

- **CCE FullAccess**：系统策略，CCE服务集群相关资源的普通操作权限，不包括集群（启用Kubernetes RBAC鉴权）的命名空间权限，不包括委托授权、生成集群证书等管理员角色的特权操作。
- **CCE ReadOnlyAccess**：系统策略，CCE服务集群相关资源的只读权限，不包括集群（启用Kubernetes RBAC鉴权）的命名空间权限。

自定义策略

如果系统预置的CCE策略，不满足您的授权要求，可以创建自定义策略。

目前支持以下两种方式创建自定义策略：

- 可视化视图创建自定义策略：无需了解策略语法，按可视化视图导航栏选择云服务、操作、资源、条件等策略内容，可自动生成策略。
- JSON视图创建自定义策略：可以在选择策略模板后，根据具体需求编辑策略内容；也可以直接在编辑框内编写JSON格式的策略内容。

本章为您介绍常用的CCE自定义策略样例。

CCE自定义策略样例：

- 示例1：创建一个名称为“test”的集群

```
{
    "Version": "1.1",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
            "cce:cluster:create"
        ]
    }
  ]
}
```

- 示例2：拒绝用户删除节点

  拒绝策略需要同时配合其他策略使用，否则没有实际作用。用户被授予的策略中，一个授权项的作用如果同时存在Allow和Deny，则遵循**Deny优先原则**。

  如果您给用户授予CCEFullAccess的系统策略，但不希望用户拥有CCEFullAccess中定义的删除节点权限（cce:node:delete），您可以创建一条相同Action的自定义策略，并将自定义策略的Effect设置为Deny，然后同时将CCEFullAccess和拒绝策略授予用户，根据Deny优先原则，则用户可以对CCE执行除了删除节点外的所有操作。拒绝策略示例如下：

  ```
  {
    "Version": "1.1",
    "Statement": [
      {
        "Effect": "Deny",
        "Action": [
          "cce:node:delete"
        ]
      }
    ]
  }
  ```

- 示例3：多个授权项策略

  一个自定义策略中可以包含多个授权项，且除了可以包含本服务的授权项外，还可以包含其他服务的授权项，可以包含的其他服务必须跟本服务同属性，即都是项目级服务或都是全局级服务。多个授权语句策略描述如下：

  ```
  {
    "Version": "1.1",
    "Statement": [
      {
        "Action": [
          "ecs:cloudServers:resize",
          "ecs:cloudServers:delete",
          "ecs:cloudServers:delete",
          "ims:images:list",
          "ims:serverImages:create"
        ],
        "Effect": "Allow"
      }
    ]
  }
  ```

CCE 集群权限与企业项目

CCE支持以集群为粒度，基于企业项目维度进行资源管理以及权限分配。

如下事项需特别注意：

- IAM项目是基于资源的物理隔离进行管理，而企业项目则是提供资源的全局逻辑分组，更符合企业实际场景，并且支持基于企业项目维度的IAM策略管理，因此推荐您使用企业项目。
- IAM项目与企业项目共存时，IAM将优先匹配IAM项目策略、未决则匹配企业项目策略。
- CCE集群基于已有基础资源（VPC）创建集群、节点时，请确保IAM用户在已有资源的企业项目下有相关权限，否则可能导致集群或者节点创建失败。

- 当资源不支持企业项目时，为企业项目授予该资源的权限将不会生效。

是否支持企业项目	资源名称	说明
支持企业项目的资源	cluster	集群
	node	节点
	nodepool	节点池
	job	任务
	tag	集群标签
	addonInstance	插件实例
	release	Helm版本
	storage	存储资源
不支持企业项目的资源	quota	集群配额
	chart	模板
	addonTemplate	插件模板

CCE 集群权限与 IAM RBAC

CCE兼容IAM传统的系统角色进行权限管理，建议您切换使用IAM的细粒度策略，避免设置过于复杂或不必要的权限管理场景。

CCE当前支持的角色如下：

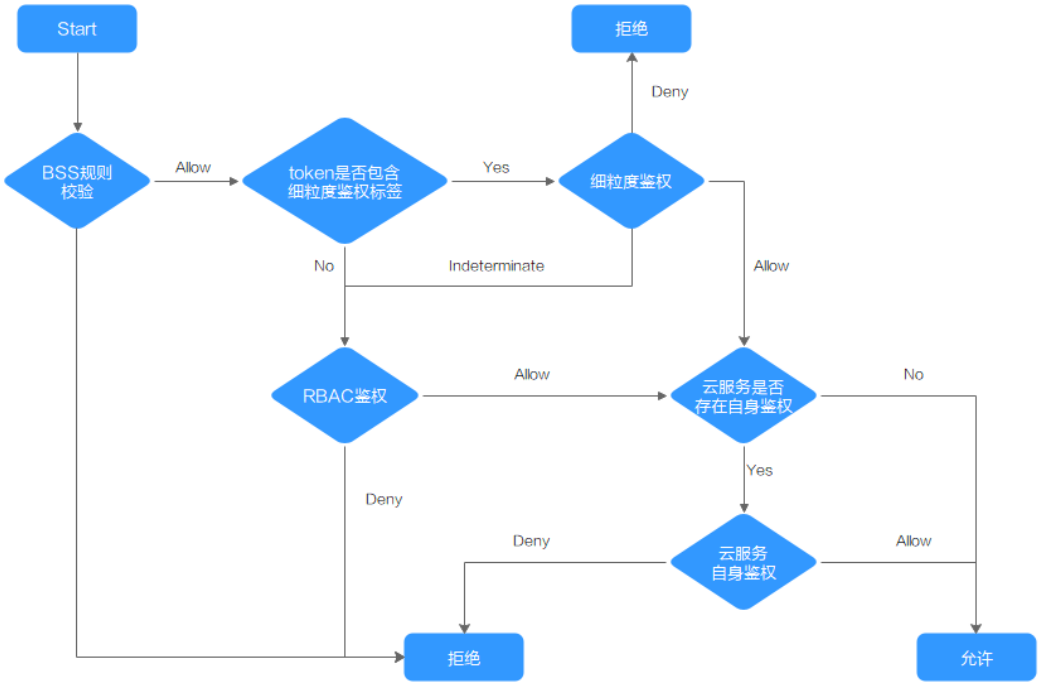
- IAM的基础角色：
  - te_admin（Tenant Administrator）：可以调用除IAM外所有服务的所有API。
  - readonly（Tenant Guest）：可以调用除IAM外所有服务的只读权限的API。
- CCE的自定义管理员角色：CCE Administrator。

说明

- Tenant Administrator、Tenant Guest是特殊的IAM系统角色，当配置任意系统或自定义策略后，Tenant Administrator、Tenant Guest将以系统策略形式生效，用于兼容IAM RBAC和ABAC场景。
- 如果用户有Tenant Administrator或者CCE Administrator的系统角色，则此用户拥有Kubernetes RBAC的cluster-admin权限，在集群创建后不可移除。

  如果用户为集群创建者，则默认被授权Kubernetes RBAC的cluster-admin权限，此项权限可以在集群创建后被手动移除:
  - 方式1：权限管理 - 命名空间权限 - 移除cluster-creator。
  - 方式2：通过API或者kubectl删除资源，ClusterRoleBinding：cluster-creator。

RBAC与IAM策略共存时，CCE开放API或Console操作的后端鉴权逻辑如下：



⚠ 注意

CCE部分接口由于涉及命名空间权限或关键操作，需要特殊权限：

clusterCert获取集群K8s kubeconfig: cceadm/teadmin

19.3 命名空间权限（Kubernetes RBAC 授权）

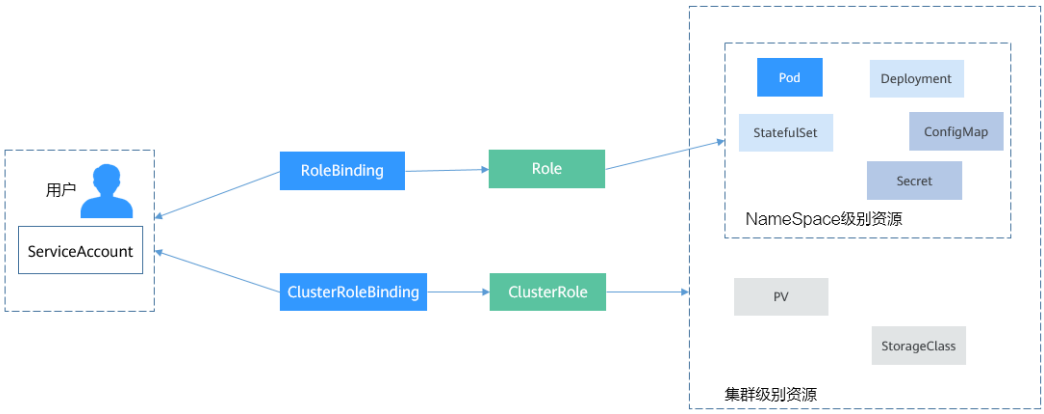
命名空间权限（kubernetes RBAC 授权）

命名空间权限是基于Kubernetes RBAC能力的授权，通过权限设置可以让不同的用户或用户组拥有操作不同Kubernetes资源的权限。Kubernetes RBAC API定义了四种类型：Role、ClusterRole、RoleBinding与ClusterRoleBinding，这四种类型之间的关系和简要说明如下：

- Role：角色，其实是定义一组对Kubernetes资源（命名空间级别）的访问规则。
- RoleBinding：角色绑定，定义了用户和角色的关系。
- ClusterRole：集群角色，其实是定义一组对Kubernetes资源（集群级别，包含全部命名空间）的访问规则。
- ClusterRoleBinding：集群角色绑定，定义了用户和集群角色的关系。

Role和ClusterRole指定了可以对哪些资源做哪些动作，RoleBinding和ClusterRoleBinding将角色绑定到特定的用户、用户组或ServiceAccount上。如下图所示。

图 19-3 角色绑定



在CCE控制台可以授予用户或用户组命名空间权限，可以对某一个命名空间或全部命名空间授权，CCE控制台默认提供如下ClusterRole。

- view（只读权限）：对全部或所选命名空间下大多数资源的只读权限。
- edit（开发权限）：对全部或所选命名空间下多数资源的读写权限。当配置在全部命名空间时能力与运维权限一致。
- admin（运维权限）：对全部命名空间下大多数资源的读写权限，对节点、存储卷，命名空间和配额管理的只读权限。
- cluster-admin（管理员权限）：对全部命名空间下所有资源的读写权限。

集群权限（IAM 授权）与命名空间权限（Kubernetes RBAC 授权）的关系

拥有不同集群权限（IAM授权）的用户，其拥有的命名空间权限（Kubernetes RBAC授权）不同。表19-2给出了不同用户拥有的命名空间权限详情。

表 19-2 不同用户拥有的命名空间权限

用户类型	1.13及以上版本的集群
拥有Tenant Administrator权限的用户	全部命名空间权限
拥有CCE Administrator权限的IAM用户	全部命名空间权限
拥有CCE FullAccess或者CCE ReadOnlyAccess权限的IAM用户	按Kubernetes RBAC授权
拥有Tenant Guest权限的IAM用户	按Kubernetes RBAC授权

注意事项

- 任何用户创建集群后，CCE会自动为该用户添加该集群的所有命名空间的cluster-admin权限，也就是说该用户允许对集群以及所有命名空间中的全部资源进行完全控制。联邦用户由于每次登录注销都会改变用户ID，所以权限用户会显示已删除，此情况下请勿删除该权限，否则会导致鉴权失败。此种情况下建议在CCE为某个用户组创建cluster-admin权限，将联邦用户加入此用户组。

配置命名空间权限（控制台）

CCE中的命名空间权限是基于Kubernetes RBAC能力的授权，通过权限设置可以让不同的用户或用户组拥有操作不同Kubernetes资源的权限。

步骤1 登录CCE控制台，在左侧导航栏中选择“权限管理”。

步骤2 在右边下拉列表中选择要添加权限的集群。

步骤3 在右上角单击“添加权限”，进入添加授权页面。

步骤4 在添加权限页面，确认集群名称，选择该集群下要授权使用的命名空间，例如选择“全部命名空间”，选择要授权的用户或用户组，再选择具体权限。

说明

对于没有IAM权限的用户，给其他用户和用户组配置权限时，无法选择用户和用户组，此时支持填写用户ID或用户组ID进行配置。

其中自定义权限可以根据需要自定义，选择自定义权限后，在自定义权限一行右侧单击新建自定义权限，在弹出的窗口中填写名称并选择规则。创建完成后，在添加权限的自定义权限下拉框中可以选择。

步骤5 单击“确定”。

----结束

自定义命名空间权限（kubectl）

说明

kubectl访问CCE集群是通过集群上生成的配置文件（kubeconfig.json）进行认证，kubeconfig.json文件内包含用户信息，CCE根据用户信息的权限判断kubectl有权限访问哪些Kubernetes资源。即哪个用户获取的kubeconfig.json文件，kubeconfig.json就拥有哪个用户的信息，这样使用kubectl访问时就拥有这个用户的权限。而用户拥有的权限就是**集群权限（IAM授权）与命名空间权限（Kubernetes RBAC授权）的关系**所示的权限。

除了使用cluster-admin、admin、edit、view这4个最常用的clusterrole外，您还可以通过定义Role和RoleBinding来进一步对命名空间中不同类别资源（如Pod、Deployment、Service等）的增删改查权限进行配置，从而做到更加精细化的权限控制。

Role的定义非常简单，指定namespace，然后就是rules规则。如下面示例中的规则就是允许对default命名空间下的Pod进行GET、LIST操作。

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default                    # 命名空间
  name: role-example
rules:
- apiGroups: [""]
  resources: ["pods"]                   # 可以访问pod
  verbs: ["get", "list"]                # 可以执行GET、LIST操作
```

- apiGroups表示资源所在的API分组。
- resources表示可以操作哪些资源：pods表示可以操作Pod，其他Kubernetes的资源如deployments、configmaps等都可以操作
- verbs表示可以执行的操作：get表示查询一个Pod，list表示查询所有Pod。您还可以使用create（创建），update（更新），delete（删除）等操作词。

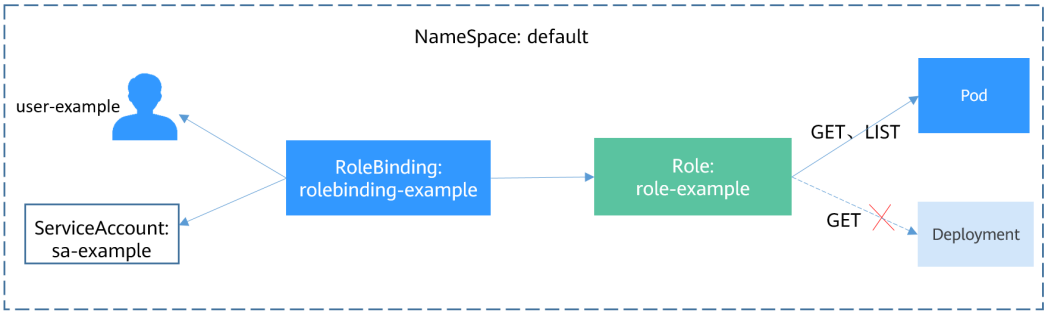
详细的类型和操作请参见**使用 RBAC 鉴权**。

有了Role之后，就可以将Role与具体的用户绑定起来，实现这个的就是RoleBinding了。如下所示。

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: RoleBinding-example
  namespace: default
  annotations:
    CCE.com/IAM: 'true'
roleRef:
  kind: Role
  name: role-example
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: User
  name: 0c97ac3cb280f4d91fa7c0096739e1f8    # user-example的用户ID
  apiGroup: rbac.authorization.k8s.io
```

这里的subjects就是将Role与IAM用户绑定起来，从而使得IAM用户获取role-example这个Role里面定义的权限，如下图所示。

图 19-4 RoleBinding 绑定 Role 和用户



subjects下用户的类型还可以是用户组，这样配置可以对用户组下所有用户生效。

```
subjects:
- kind: Group
  name: 0c96fad22880f32a3f84c009862af6f7    # 用户组ID
  apiGroup: rbac.authorization.k8s.io
```

使用IAM用户user-example连接集群，获取Pod信息，发现可获取到Pod的信息。

```
# kubectl get pod
NAME                                  READY   STATUS    RESTARTS   AGE
deployment-389584-2-6f6bd4c574-2n9rk   1/1     Running   0          4d7h
deployment-389584-2-6f6bd4c574-7s5qw   1/1     Running   0          4d7h
deployment-3895841-746b97b455-86g77    1/1     Running   0          4d7h
deployment-3895841-746b97b455-twvpn    1/1     Running   0          4d7h
nginx-658dff48ff-7rkph                 1/1     Running   0          4d9h
nginx-658dff48ff-njdhj                 1/1     Running   0          4d9h
# kubectl get pod nginx-658dff48ff-7rkph
NAME                     READY   STATUS    RESTARTS   AGE
nginx-658dff48ff-7rkph   1/1     Running   0          4d9h
```

然后查看Deployment和Service，发现没有权限；再查询kube-system命名空间下的Pod信息，发现也没有权限。这就说明IAM用户user-example仅拥有defaul这个命名空间下GET和LIST Pod的权限，与前面定义的没有偏差。

```
# kubectl get deploy
Error from server (Forbidden): deployments.apps is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8"
```

```
cannot list resource "deployments" in API group "apps" in the namespace "default"
# kubectl get svc
Error from server (Forbidden): services is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list
resource "services" in API group "" in the namespace "default"
# kubectl get pod --namespace=kube-system
Error from server (Forbidden): pods is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list
resource "pods" in API group "" in the namespace "kube-system"
```

示例：授予集群管理员权限（cluster-admin）

集群全部权限可以使用cluster-admin权限，cluster-admin包含集群所有资源的权限。

如果使用kubectl查看可以看到创建了一个ClusterRoleBinding，将cluster-admin和cce-role-group这个用户组绑定了起来。

```
# kubectl get clusterrolebinding
NAME                                                    ROLE                        AGE
clusterrole_cluster-admin_group0c96fad22880f32a3f84c009862af6f7   ClusterRole/cluster-admin      61s

# kubectl get clusterrolebinding clusterrole_cluster-admin_group0c96fad22880f32a3f84c009862af6f7 -oyaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    CCE.com/IAM: "true"
  creationTimestamp: "2021-06-23T09:15:22Z"
  name: clusterrole_cluster-admin_group0c96fad22880f32a3f84c009862af6f7
  resourceVersion: "36659058"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/clusterrole_cluster-
admin_group0c96fad22880f32a3f84c009862af6f7
  uid: d6cd43e9-b4ca-4b56-bc52-e36346fc1320
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: 0c96fad22880f32a3f84c009862af6f7
```

使用被授予用户连接集群，如果能正常查询PV、StorageClass的信息，则说明权限配置正常。

```
# kubectl get pv
No resources found
# kubectl get sc
NAME                PROVISIONER                     RECLAIMPOLICY    VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION    AGE
csi-disk            everest-csi-provisioner         Delete           Immediate              true                   75d
csi-disk-topology   everest-csi-provisioner         Delete           WaitForFirstConsumer   true                   75d
csi-nas             everest-csi-provisioner         Delete           Immediate              true                   75d
csi-obs             everest-csi-provisioner         Delete           Immediate              false                  75d
```

示例：授予命名空间运维权限（admin）

admin权限拥有命名空间大多数资源的读写权限，您可以授予用户/用户组全部命名空间admin权限。

如果使用kubectl查看可以看到创建了一个RoleBinding，将admin和cce-role-group这个用户组绑定了起来。

```
# kubectl get rolebinding
NAME                                            ROLE                AGE
clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7   ClusterRole/admin   18s
# kubectl get rolebinding clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7 -oyaml
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: RoleBinding
metadata:
  annotations:
    CCE.com/IAM: "true"
  creationTimestamp: "2021-06-24T01:30:08Z"
  name: clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7
  resourceVersion: "36963685"
  selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/default/rolebindings/
clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7
  uid: 6c6f46a6-8584-47da-83f5-9eef1f7b75d6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: 0c96fad22880f32a3f84c009862af6f7
```

使用被授予用户连接集群，如果能正常查询PV、StorageClass的信息，但无法创建命名空间，则说明权限配置正常。

```
# kubectl get pv
No resources found
# kubectl get sc
NAME                PROVISIONER                     RECLAIMPOLICY    VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION    AGE
csi-disk            everest-csi-provisioner         Delete           Immediate              true                   75d
csi-disk-topology   everest-csi-provisioner         Delete           WaitForFirstConsumer   true                   75d
csi-nas             everest-csi-provisioner         Delete           Immediate              true                   75d
csi-obs             everest-csi-provisioner         Delete           Immediate              false                  75d
# kubectl apply -f namespaces.yaml
Error from server (Forbidden): namespaces is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot
create resource "namespaces" in API group "" at the cluster scope
```

示例：授予命名空间开发权限（edit）

edit权限拥有命名空间大多数资源的读写权限，您可以授予用户/用户组全部命名空间edit权限。

如果使用kubectl查看可以看到创建了一个RoleBinding，将edit和cce-role-group这个用户组绑定了起来，且权限范围是default这个命名空间。

```
# kubectl get rolebinding
NAME                                            ROLE                AGE
clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7   ClusterRole/admin   18s
# kubectl get rolebinding clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7 -oyaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  annotations:
    CCE.com/IAM: "true"
  creationTimestamp: "2021-06-24T01:30:08Z"
  name: clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7
  namespace: default
  resourceVersion: "36963685"
  selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/default/rolebindings/
clusterrole_admin_group0c96fad22880f32a3f84c009862af6f7
  uid: 6c6f46a6-8584-47da-83f5-9eef1f7b75d6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: edit
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: 0c96fad22880f32a3f84c009862af6f7
```

使用被授予用户连接集群，您会发现可以查询和创建default命名空间的资源，但无法查询kube-system命名空间资源，也无法查询集群级别的资源。

```
# kubectl get pod
NAME                 READY   STATUS    RESTARTS   AGE
test-568d96f4f8-brdrp   1/1     Running   0          33m
test-568d96f4f8-cgjqp   1/1     Running   0          33m
# kubectl get pod -nkube-system
Error from server (Forbidden): pods is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "pods" in API group "" in the namespace "kube-system"
# kubectl get pv
Error from server (Forbidden): persistentvolumes is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "persistentvolumes" in API group "" at the cluster scope
```

示例：授予命名空间只读权限（view）

view权限拥有命名空间查看权限，您可以给某个或全部命名空间授权。

如果使用kubectl查看可以看到创建了一个RoleBinding，将view和cce-role-group这个用户组绑定了起来，且权限范围是default这个命名空间。

```
# kubectl get rolebinding
NAME                                                ROLE             AGE
clusterrole_view_group0c96fad22880f32a3f84c009862af6f7   ClusterRole/view   7s

# kubectl get rolebinding clusterrole_view_group0c96fad22880f32a3f84c009862af6f7 -oyaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  annotations:
    CCE.com/IAM: "true"
  creationTimestamp: "2021-06-24T01:36:53Z"
  name: clusterrole_view_group0c96fad22880f32a3f84c009862af6f7
  namespace: default
  resourceVersion: "36965800"
  selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/default/rolebindings/
clusterrole_view_group0c96fad22880f32a3f84c009862af6f7
  uid: b86e2507-e735-494c-be55-c41a0c4ef0dd
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: 0c96fad22880f32a3f84c009862af6f7
```

使用被授予用户连接集群，您会发现可以查询default命名空间的资源，但无法创建资源。

```
# kubectl get pod
NAME                 READY   STATUS    RESTARTS   AGE
test-568d96f4f8-brdrp   1/1     Running   0          40m
test-568d96f4f8-cgjqp   1/1     Running   0          40m
# kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
Error from server (Forbidden): pods is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot create resource "pods" in API group "" in the namespace "default"
```

示例：授予某类 Kubernetes 资源权限

上面几个示例都是集群全部资源（cluster-admin）、命名空间全部资源（admin、view），也可以对某类Kubernetes资源授权，如Pod、Deployment、Service这些资源，具体请参见**自定义命名空间权限（kubectl）**。

19.4 示例：某部门权限设计及配置

概述

随着容器技术的快速发展，原有的分布式任务调度模式正在被基于Kubernetes的技术架构所取代。云容器引擎（Cloud Container Engine，简称CCE）是高度可扩展的、高性能的企业级Kubernetes集群，支持社区原生应用和工具。借助云容器引擎，您可以在云上轻松部署、管理和扩展容器化应用程序，快速高效的将微服务部署在云端。

为方便企业中的管理人员对集群中的资源权限进行管理，CCE后台提供了多种维度的细粒度权限策略和管理方式。CCE的权限管理包括“集群权限”和“命名空间权限”两种能力，分别从集群和命名空间两个层面对用户组或用户进行细粒度授权，具体解释如下：

- **集群权限：**是基于IAM系统策略的授权，可以让用户组拥有“集群管理”、“节点管理”、“节点池管理”、“模板市场”、“插件管理”权限。
- **命名空间权限：**是基于Kubernetes RBAC能力的授权，可以让用户或用户组拥有Kubernetes资源的权限，如“工作负载”、“网络管理”、“存储管理”、“命名空间”等的权限。

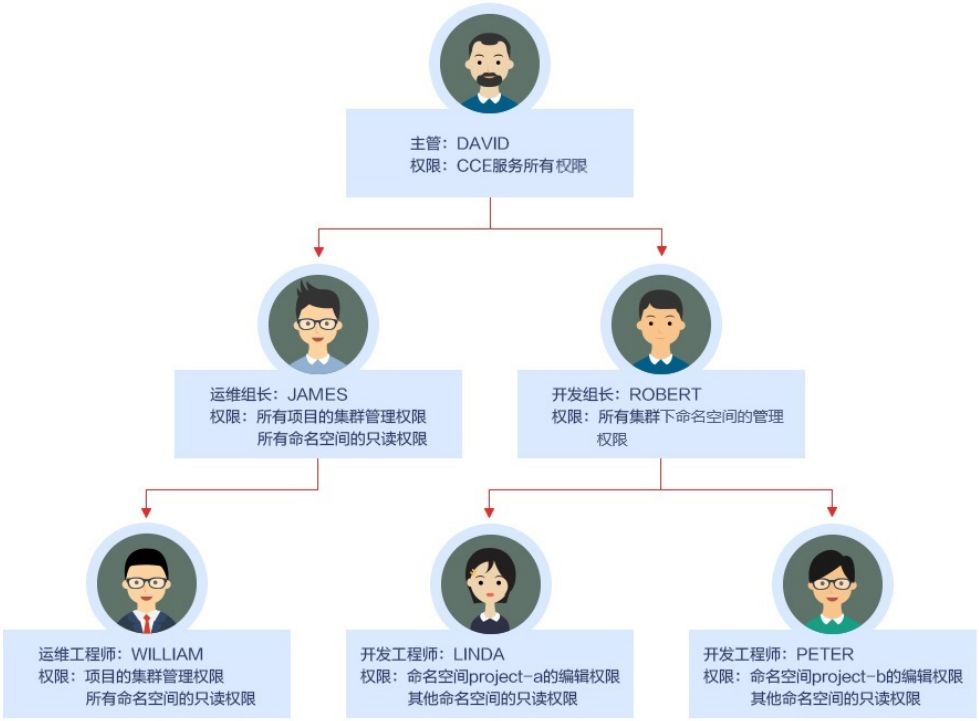
基于IAM系统策略的“集群权限”与基于Kubernetes RBAC能力的“命名空间权限”，两者是完全独立的，互不影响，但要配合使用。同时，为用户组设置的权限将作用于用户组下的全部用户。当给用户或用户组添加多个权限时，多个权限会同时生效（取并集）。

权限设计

下面以一个公司为例进行介绍。

通常一个公司中有多个部门或项目，每个部门又有多个成员，所以在配置权限前需要先进行详细设计，并在设置权限之前提前为每个成员创建用户名，便于后续对用户进行用户组归属和权限设置。

下图为某公司某部门的组织架构图和相关人员的权限设计，本文将按照该设计对每个角色的权限设置进行演示：



主管：DAVID

用户“DAVID”为该公司某部门的主管，根据权限设计需要为其配置CCE服务的所有权限（包括集群权限和命名空间权限），因此需要在统一身份认证服务 IAM中单独为DAVID创建用户组“cce-admin”，并配置所有项目的权限：“CCE Administrator”，这样主管DAVID的权限就配置好了。

说明

CCE Administrator：CCE的管理员权限，拥有该服务的所有权限，不需要再赋予其他权限。

CCE FullAccess、CCE ReadOnlyAccess：CCE的集群管理权限，仅针对与集群相关的资源（如集群、节点）有效，您必须确保同时配置了“命名空间权限”，才能有操作Kubernetes资源（如工作负载、Service等）的权限。

运维组长：JAMES

用户“JAMES”为该部门的运维组长，需要设置所有项目的集群权限和所有命名空间的只读权限。

在统一身份认证服务 IAM中先为用户“JAMES”单独创建并加入用户组“cce-sre”，然后为用户组“cce-sre”配置所有项目的集群权限：“CCE FullAccess”，用户组“cce-sre”便拥有了所有项目的集群管理权限，接下来还需要为其设置命名空间的只读权限。

为所有组长和工程师添加所有集群和命名空间的只读权限

在统一身份认证服务 IAM中再创建一个只读用户组“read_only”，然后将相关用户都添加到此用户组中。

- 两个开发工程师虽然不需要配置集群的管理权限，但也需要查看CCE控制台，因此需要有集群的只读权限才能满足需求。

- 运维工程师需要某区域集群的管理权限，为方便管理，这里先为其赋予集群的只读权限。
- 运维组长已经拥有了所有集群的管理权限，为方便管理，也可以将其添加到“read_only”用户组中，为其赋予集群的只读权限。

将JAMES、ROBERT、WILLIAM、LINDA、PETER五个用户都添加到用户组“read_only”中。

接下来为用户组“read_only”赋予集群的只读权限。

然后返回CCE控制台，为这五个用户所在的用户组“read_only”增加命名空间的只读权限，单击左侧栏目树中的“权限管理”，为用户组“read_only”逐个赋予所有集群的只读权限。

设置完成后，运维组长“JAMES”就拥有了所有项目的集群管理权限和所有命名空间的只读权限，而开发组长“ROBERT”、运维工程师“WILLIAM”以及两位开发工程师“LINDA”和“PRTER”则拥有了所有集群和命名空间的只读权限。

开发组长：ROBERT

用户“ROBERT”作为开发组的组长，虽然在上一步中已经为其设置了所有集群和命名空间的只读权限，但显然还不够，还需要为其设置所有命名空间的管理权限。

因此需要再单独为其赋予所有集群下全部命名空间的管理员权限。

运维工程师：WILLIAM

运维工程师“WILLIAM”虽然也有了所有集群和命名空间的只读权限，但还需要在统一身份认证服务 IAM中为其设置区域的集群管理权限，因此单独为其创建一个用户组“cce-sre-b4”，然后配置区域项目的“CCE FullAccess”。

由于之前已经为其设置过所有命名空间的只读权限，所以运维工程师“WILLIAM”现在就拥有了区域的集群管理权限和所有命名空间的只读权限。

开发工程师：LINDA、PETER

“LINDA”和“PETER”是开发工程师，由于前面已经在用户组“read-only”中为两位工程师配置了集群和命名空间的只读权限，这里只需要再另外配置相应命名空间的编辑权限即可。

至此，该部门的所有权限就设置完成了。

19.5 CCE 控制台的权限依赖

CCE对其他云服务有诸多依赖关系，因此在您开启IAM系统策略授权后，在CCE Console控制台的各项功能需要配置相应的服务权限后才能正常查看或使用，详细说明如下：

- 依赖服务的权限配置均基于您已设置了IAM系统策略授权的CCE FullAccess或CCE ReadOnlyAccess策略权限。
- 集群显示情况依赖于命名空间权限的设置情况，如果没有设置命名空间权限，则无法查看集群下的资源。

- 如果您设置了全部命名空间的view权限，则可以查看到对应集群的全部命名空间下的资源，但密钥（Secret）除外，密钥（Secret）需要在命名空间权限下设置admin或者edit权限才能查看。
- HPA策略需要配置命名空间cluster-admin权限下才能生效。
- 如果您设置的是单一命名空间的view权限，则看到的只能是指定命名空间下的资源。

依赖服务的权限设置

如果IAM用户需要在CCE Console控制台拥有相应功能的查看或使用权限，请确认已经对该用户所在的用户组设置了CCE Administrator、CCE FullAccess或CCE ReadOnlyAccess策略的集群权限，再按如下表19-3增加依赖服务的角色或策略。

说明

CCE支持细粒度的权限设置，但有如下限制说明：

- AOM不支持资源级别细粒度：当通过IAM集群资源细粒度设置特定资源操作权限之后，IAM用户在CCE控制台的总览界面查看集群监控时，将显示非细粒度关联集群的监控信息。

表 19-3 CCE Console 中依赖服务的角色或策略

Console控制台功能	依赖服务	需配置角色/策略
集群信息总览	应用运维管理 AOM	• IAM用户设置了CCE Administrator权限后，需要增加AOM FullAccess权限后才能访问总览中的数据图表。 • 支持设置了IAM ReadOnlyAccess和CCE FullAccess或CCE ReadOnlyAccess权限的IAM用户直接访问总览中的数据图表。

Console控制台功能	依赖服务	需配置角色/策略
工作负载	弹性负载均衡 ELB 应用性能管理 APM 应用运维管理 AOM NAT网关 NAT 对象存储服务 OBS 弹性文件服务 SFS	正常创建工作负载时不依赖其他服务的权限。 ● 如果需要创建ELB类型的服务，需要设置ELB FullAccess或者ELB Administrator权限，以及VPC Administrator权限。 ● 如果需要使用Java探针，需要设置AOM FullAccess和APM FullAccess权限。 ● 如果需要NAT网关类型的服务，需要设置NAT Gateway Administrator权限。 ● 如果使用对象存储，需要全局设置OBS Administrator权限。 **说明** 由于缓存的存在，对用户、用户组以及企业项目授予OBS相关的RBAC策略后，大概需要等待13分钟RBAC策略才能生效；授予OBS相关的系统策略后，大概需要等待5分钟系统策略能生效。 ● 如果使用文件存储，需要设置SFS FullAccess权限。
集群管理	应用运维管理 AOM	● 如果需要弹性扩容权限，需要设置AOM FullAccess权限。
节点管理	弹性云服务器 ECS	当IAM用户权限为CCE Administrator时，如果创建和删除节点，需要配置ECS FullAccess或ECS Administrator权限，以及VPC Administrator权限。
服务发现	弹性负载均衡 ELB NAT网关 NAT	正常创建时不依赖其他服务的权限。 ● 如果需要创建ELB类型的服务，需要设置ELB FullAccess或者ELB Administrator权限，以及VPC Administrator权限。 ● 如果需要NAT网关类型的服务，需要设置NAT Administrator权限。

Console控制台功能	依赖服务	需配置角色/策略
容器存储	对象存储服务 OBS 弹性文件服务 SFS 极速文件存储 SFS Turbo	• 如果使用对象存储，需要全局设置OBS Administrator权限。 **说明** 由于缓存的存在，对用户、用户组以及企业项目授予OBS相关的RBAC策略后，大概需要等待13分钟RBAC策略才能生效；授予OBS相关的系统策略后，大概需要等待5分钟系统策略能生效。 • 如果使用文件存储，需要设置SFS FullAccess权限。 • 如果使用极速文件存储，需要设置SFS Turbo Admin权限 导入存储的功能需要设置CCE Administrator权限。
命名空间	/	无需其他依赖权限。
模板市场	/	当前仅支持账号、设置了CCE Administrator权限的IAM用户访问。
插件管理	/	支持账号、设置了CCE Administrator、CCE FullAccess或CCE ReadOnlyAccess等权限的IAM用户访问本功能。
权限管理	/	• 支持账号访问。 • 支持设置了CCE Administrator和Security Administrator（全局级策略）权限的IAM用户访问。 • 支持设置了CCE FullAccess或CCE ReadOnlyAccess权限的IAM用户访问，同时还需要拥有命名空间的**管理员权限（cluster-admin）**。
配置项与密钥	/	• 配置项（ConfigMap）无需其他依赖权限。 • 密钥（Secret）需要在命名空间权限下设置cluster-admin、admin或者edit权限才能查看，依赖服务需要添加DEW KeypairFullAccess或者DEW KeypairReadOnlyAccess权限。
帮助中心	/	无需其他依赖权限。
其他服务跳转	容器镜像服务 SWR 应用运维管理 AOM	为便于您快速进入CCE相关服务的控制台，在CCE控制台增加了其他服务的跳转链接，CCE默认没有这些服务的全部权限，如果IAM用户需要查看或使用其功能，请按照该服务的权限策略说明设置相应的权限策略。

19.6 Pod 安全配置

19.6.1 PodSecurityPolicy 配置

Pod安全策略（Pod Security Policy）是集群级别的资源，它能够控制Pod规约中与安全性相关的各个方面。 **PodSecurityPolicy**对象定义了一组Pod运行时必须遵循的条件及相关字段的默认值，只有Pod满足这些条件才会被系统接受。

v1.17.17版本的集群默认启用Pod安全策略准入控制组件，并创建名为**psp-global**的全局默认安全策略，您可根据自身业务需要修改全局策略（请勿直接删除默认策略），也可新建自己的Pod安全策略并绑定RBAC配置。

说明

- 除全局默认安全策略外，系统为kube-system命名空间下的系统组件配置了独立的Pod安全策略，修改psp-global配置不影响kube-system下Pod创建。
- PodSecurityPolicy在Kubernetes v1.21版本中被弃用，并在Kubernetes v1.25中被移除。您可以Pod安全性准入控制器（Pod Security Admission）作为PodSecurityPolicy的替代，详情请参见**Pod Security Admission配置**。

修改全局默认 Pod 安全策略

修改全局默认Pod安全策略前，请确保已创建CCE集群，并且通过kubectl连接集群成功。

步骤1 执行如下命令：

kubectl edit psp psp-global

步骤2 修改所需的参数，如**表19-4**。

表 19-4 Pod 安全策略配置

配置项	描述
privileged	启动特权容器。
hostPID hostIPC	使用主机命名空间。
hostNetwork hostPorts	使用主机网络和端口。
volumes	允许使用的挂载卷类型。
allowedHostPaths	允许hostPath类型挂载卷在主机上挂载的路径，通过pathPrefix字段声明允许挂载的主机路径前缀组。
allowedFlexVolumes	允许使用的指定FlexVolume驱动。
fsGroup	配置Pod中挂载卷使用的辅组ID。

配置项	描述
readOnlyRootFilesystem	约束启动Pod使用只读的root文件系统。
runAsUser runAsGroup supplementalGroups	指定Pod中容器启动的用户ID以及主组和辅组ID。
allowPrivilegeEscalation defaultAllowPrivilegeEscalation	约束Pod中是否允许配置allowPrivilegeEscalation=true，该配置会控制Setuid的使用，同时控制程序是否可以使用额外的特权系统调用。
defaultAddCapabilities requiredDropCapabilities allowedCapabilities	控制Pod中使用的Linux Capabilities。
seLinux	控制Pod使用seLinux配置。
allowedProcMountTypes	控制Pod允许使用的ProcMountTypes。
annotations	配置Pod中容器使用的AppArmor或Seccomp。
forbiddenSysctls allowedUnsafeSysctls	控制Pod中容器使用的Sysctl配置。

----结束

Pod 安全策略开放非安全系统配置示例

节点池管理中可以为相应的节点池配置allowed-unsafe-sysctls，CCE从**1.17.17**集群版本开始，需要在Pod安全策略的allowedUnsafeSysctls字段中增加相应的配置才能生效，配置详情请参考**表19-4**。

除修改全局Pod安全策略外，也可增加新的Pod安全策略，如开放net.core.somaxconn非安全系统配置，新增Pod安全策略示例参考如下：

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
  - net.core.somaxconn
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  fsGroup:
    rule: RunAsAny
  hostIPC: true
  hostNetwork: true
  hostPID: true
  hostPorts:
  - max: 65535
    min: 0
  privileged: true
```

```
  runAsGroup:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - '*'
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: sysctl-psp
rules:
  - apiGroups:
      - "*"
    resources:
      - podsecuritypolicies
    resourceNames:
      - sysctl-psp
    verbs:
      - use

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: sysctl-psp
roleRef:
  kind: ClusterRole
  name: sysctl-psp
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
```

恢复原始 Pod 安全策略

如果您已经修改默认Pod安全策略后，想恢复原始Pod安全策略，请执行以下操作。

步骤1 创建一个名为policy.yaml的描述文件。其中，policy.yaml为自定义名称，您可以随意命名。

vi policy.yaml

描述文件内容如下。

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-global
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
    - '*'
  volumes:
    - '*'
  hostNetwork: true
  hostPorts:
    - min: 0
      max: 65535
  hostIPC: true
```

```
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'

---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-global
rules:
  - apiGroups:
      - "*"
    resources:
      - podsecuritypolicies
    resourceNames:
      - psp-global
    verbs:
      - use

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: psp-global
roleRef:
  kind: ClusterRole
  name: psp-global
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
```

步骤2 执行如下命令：

kubectl apply -f policy.yaml

----结束

19.6.2 Pod Security Admission 配置

在使用**Pod Security Admission**前，需要先了解Kubernetes的**Pod安全性标准（Security Standards）**。Pod安全性标准（Security Standards）为 Pod 定义了不同的安全性策略级别。这些标准能够让你以一种清晰、一致的方式定义如何限制Pod行为。而Pod Security Admission则是这些安全性标准的控制器，用于在创建Pod时执行定义好的安全限制。

Pod安全性标准定义了三种安全性策略级别：

表 19-5 Pod 安全性策略级别

策略级别（level）	描述
privileged	不受限制，通常适用于特权较高、受信任的用户所管理的系统级或基础设施级负载，例如CNI、存储驱动等。

策略级别（level）	描述
baseline	限制较弱但防止已知的特权提升（Privilege Escalation），通常适用于部署常用的非关键性应用负载，该策略将禁止使用hostNetwork、hostPID等能力。
restricted	严格限制，遵循Pod防护的最佳实践。

Pod Security Admission配置是命名空间级别的，控制器将会对该命名空间下Pod或容器中的安全上下文（Security Context）以及其他参数进行限制。其中，privileged策略将不会对Pod和Container配置中的securityContext字段有任何校验，而Baseline和Restricted则会对securityContext字段有不同的取值要求，具体规范请参见**Pod安全性标准（Security Standards）**。

关于如何在Pod或容器中设置Security Context，请参见**为Pod或容器配置Security Context**。

Pod Security Admission 标签

Kubernetes为Pod Security Admission定义了三种标签，如**表19-6**，您可以在某个命名空间中设置这些标签来定义需要使用的Pod安全性标准级别，但请勿在kube-system等系统命名空间修改Pod安全性标准级别，否则可能导致系统命名空间下Pod故障。

表 19-6 Pod Security Admission 标签

隔离模式（mode）	生效对象	描述
enforce	Pod	违反指定策略会导致Pod无法创建。
audit	工作负载（例如Deployment、Job等）	违反指定策略会在审计日志（audit log）中添加新的审计事件，Pod可以被创建。
warn	工作负载（例如Deployment、Job等）	违反指定策略会返回用户可见的告警信息，Pod可以被创建。

说明

Pod通常是通过创建Deployment或Job这类工作负载对象来间接创建的。在使用Pod Security Admission时，audit或warn模式的隔离都将在工作负载级别生效，而enforce模式并不会应用到工作负载，仅在Pod上生效。

使用命名空间标签进行 Pod Security Admission 配置

您可以在不同的隔离模式中应用不同的策略，由于Pod安全性准入能力是在命名空间（Namespace）级别实现的，因此假设某个Namespace配置如下：

```
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: my-baseline-namespace
  labels:
    pod-security.kubernetes.io/enforce: privileged
    pod-security.kubernetes.io/enforce-version: v1.25
    pod-security.kubernetes.io/audit: baseline
    pod-security.kubernetes.io/audit-version: v1.25
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: v1.25

    # 标签有以下两种格式：
    # pod-security.kubernetes.io/<MODE>: <LEVEL>
    # pod-security.kubernetes.io/<MODE>-version: <VERSION>
    # audit和warn模式的作用主要在于提供相应信息供用户排查负载违反了哪些安全行为
```

命名空间的标签用来表示不同的模式所应用的安全策略级别，存在以下两种格式：

- pod-security.kubernetes.io/<MODE>: <LEVEL>
  - <MODE>：必须是enforce、audit或warn之一，关于标签详情请参见**表19-6**。
  - <LEVEL>：必须是privileged、baseline或restricted之一，关于安全性策略级别详情请参见**表19-5**。
- pod-security.kubernetes.io/<MODE>-version: <VERSION>

  该标签为可选，可以将安全性策略锁定到Kubernetes版本号。
  - <MODE>：必须是enforce、audit或warn之一，关于标签详情请参见**表19-6**。
  - <VERSION>：Kubernetes版本号。例如 v1.25，也可以使用latest。

若在上述Namespace中部署Pod，则会有以下安全性限制：

1. 设置了enforce隔离模式对应的策略为privileged，将会跳过enforce阶段的校验。
2. 设置了audit隔离模式对应的策略为baseline，将会校验baseline策略相关限制，即如果Pod或Container违反了该策略，审计日志中将添加相应事件。
3. 设置了warn隔离模式对应的策略为restricted，将会校验restricted策略相关限制，即如果Pod或Container违反了该策略，用户将会在创建Pod时收到告警信息。

从 PodSecurityPolicy 迁移到 Pod Security Admission

如您在1.25之前版本的集群中使用了PodSecurityPolicy，且需要在1.25及以后版本集群中继续使用Pod Security Admission来替代PodSecurityPolicy的用户，请参见**从PodSecurityPolicy迁移到内置的Pod Security Admission**。

须知

1. 由于Pod Security Admission仅支持三种隔离模式，因此灵活性相比于PodSecurityPolicy较差，部分场景下需要用户自行定义验证准入Webhook来实施更精准的策略。
2. 由于PodSecurityPolicy具有变更能力，而Pod Security Admission并不具备该能力，因此之前依赖该能力的用户需要自行定义变更准入Webhook或修改Pod中的securityContext字段。
3. PodSecurityPolicy允许为不同的服务账号（Service Account）绑定不同策略（Kubernetes社区不建议使用该能力）。如果您有使用该能力的诉求，在迁移至Pod Security Admission后，需要自行定义第三方Webhook。
4. 请勿将Pod Security Admission能力应用于kube-system、kube-public和kube-node-lease等一些CCE组件部署的Namespace中，否则会导致CCE组件、插件功能异常。

参考文档

- **Pod安全性准入**
- **从PodSecurityPolicy映射到Pod安全性标准**
- **使用命名空间标签来实施Pod安全性标准**
- **通过配置内置准入控制器实施Pod安全标准**

19.7 ServiceAccount Token 安全性提升说明

Kubernetes 1.21以前版本的集群中，Pod中获取Token的形式是通过挂载ServiceAccount的Secret来获取Token，这种方式获得的Token是永久的。该方式在1.21及以上的版本中不再推荐使用，并且根据社区版本迭代策略，在1.25及以上版本的集群中，ServiceAccount将不会自动创建对应的Secret。

Kubernetes 1.21及以上版本的集群中，直接使用**TokenRequest** API获得Token，并使用投射卷（Projected Volume）挂载到Pod中。使用这种方法获得的Token具有固定的生命周期（默认有效期为1小时），在到达有效期之前，Kubelet会刷新该Token，保证Pod始终拥有有效的Token，并且当挂载的Pod被删除时这些Token将自动失效。该方式通过**BoundServiceAccountTokenVolume**特性实现，能够提升服务账号（ServiceAccount）Token的安全性，Kubernetes 1.21及以上版本的集群中会默认开启。

为了帮助用户平滑过渡，社区默认将Token有效时间延长为1年，1年后Token失效，不具备证书reload能力的client将无法访问APIServer，建议使用低版本Client的用户尽快升级至高版本，否则业务将存在故障风险。

如果用户使用版本过低的Kubernetes客户端（Client），由于低版本Client并不具备证书轮转能力，会存在证书轮转失效的风险。K8s社区默认具有证书轮转能力的Client版本如下：

- Go: >= v0.15.7
- Python: >= v12.0.0
- Java: >= v9.0.0
- Javascript: >= v0.10.3

- Ruby: master branch
- Haskell: v0.3.0.0
- C#: >= 7.0.5

官方说明请参见：**https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/1205-bound-service-account-tokens**

说明

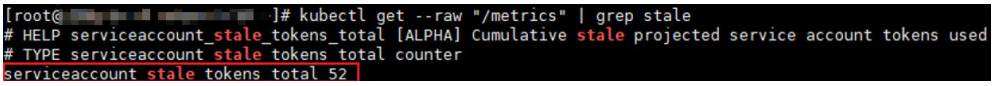
如果您在业务中需要一个永不过期的Token，您也可以选择**手动管理ServiceAccount的Secret**。尽管存在手动创建永久ServiceAccount Token的机制，但还是推荐使用**TokenRequest**的方式使用短期的Token，以提高安全性。

排查方案

CCE提供以下排查方式供用户参考（CCE 1.21及以上版本的集群均涉及）：

1. 通过kubectl连接集群，并通过**kubectl get --raw "/metrics" | grep stale**查询，可以看到一个名为serviceaccount_stale_tokens_total的指标。

   如果该值大于0，则表示当前集群可能存在某些负载正在使用过低的client-go版本情况，此时请您排查自己部署的应用中是否有该情况出现。如果存在，则尽快将client-go版本升级至社区指定的版本之上（至少不低于CCE集群的两个大版本，如部署在1.23集群上的应用需要使用1.19版本以上的Kubernetes依赖库）。

20 常见问题

20.1 高频常见问题

集群管理

- CCE集群创建失败的原因与解决方法?
- 集群的管理规模和控制节点的数量有关系吗?
- 当集群状态为“不可用”时，如何排查解决?

节点及节点池

- 集群可用，但节点状态为“不可用”？
- 容器使用SCSI类型云硬盘偶现IO卡住

工作负载

- 工作负载异常：实例调度失败
- 工作负载异常：实例拉取镜像失败
- 工作负载异常：启动容器失败
- 工作负载异常：结束中，解决Terminating状态的Pod删不掉的问题
- CCE集群中工作负载镜像的拉取策略?

网络管理

为什么访问部署的应用时浏览器返回404错误码?

节点无法连接互联网（公网），如何排查定位?

解析外部域名很慢或超时，如何优化配置?

20.2 计费类

20.2.1 云容器引擎 CCE 如何定价/收费?

计费项

云容器引擎（CCE）本身不收取任何费用，但在使用过程中会创建相关资源（如节点、带宽等），您需要为您使用的这些资源付费。CCE相关资源的计费项分为如下两部分：

1. **集群：**控制节点资源费用，按照每个集群的类型（虚拟机或裸金属、控制节点数）、集群规模（最大支持的节点数）的差异收取不同的费用。

   说明

   集群规模是指用户在集群下创建和购买的云主机或者裸金属服务器的数量。

2. **IaaS基础设施：**集群工作节点所使用的IaaS基础设施费用，包括集群创建使用过程中自动创建或手动加入的相关资源，如云服务器、云硬盘、弹性IP/带宽、负载均衡等，价格参照相应产品价格表。

计费模式

CCE支持按需计费费模式。

- 按需计费：一种先使用后付费的方式，从“开通”开启计费到“删除”结束计费，按实际购买时长计费。这种购买方式比较灵活，您可以按需取用资源，随时开启和释放，无需提前购买大量资源。

  说明

  关于CCE集群休眠或节点关机后的收费说明：

  - **集群休眠：**集群休眠后，控制节点资源费用将停止收费，集群所属的云硬盘、绑定的弹性IP、带宽等资源按各自的计费方式（“按需付费”）进行收费。
  - **节点关机：**集群休眠后，集群中的工作节点（即ECS）并不会自动关机，如需关机可勾选“关机集群下所有节点”选项。您也可以在集群休眠后自行登录ECS控制台将节点关机。

须知

- 以集群作为计费量纲，根据集群类型和规模大小，按阶梯计费。
- 提供给客户进行续费与充值的时间，当您的按需资源欠费时提供宽限期和保留期。

20.2.2 CCE 是否支持余额不足提醒?

用户可在费用中心总览页面“可用额度”区域单击“设置”，设置“可用额度预警”后的开关，即可开通或关闭可用额度预警功能。单击“修改”，可以对预警阈值进行修改。

- 开通后，当可用额度（含现金余额、信用余额、通用代金券、现金券）的总金额低于预警阈值时， 会每天给联系人发送短信和邮件提醒，最多连续提醒3天。
- 您可到消息中心“消息接收设置 > 财务信息 > 账户余额预警”中修改预警提醒的联系人信息。

20.2.3 CCE 是否支持账户余额变动提醒?

系统会以邮件、短信形式给客户发送账户余额变动通知，包括账户余额调整、充值到账、客户在线充值等。

20.3 集群

20.3.1 集群创建

20.3.1.1 CCE 集群创建失败的原因与解决方法?

概述

本文主要介绍在CCE集群创建失败时，如何查找失败的原因，并解决问题。

详细信息

集群创建失败的原因包括：

1. ntpd没安装或者安装失败、k8s组件预校验不过、磁盘分区错误等，目前只能尝试重新创建，定位方法请参见**定位失败原因**。

定位失败原因

您可以参考以下步骤，通过集群日志查看集群创建失败的报错信息，然后根据相应的解决方法解决问题：

步骤1 登录CCE控制台，单击集群列表上方的“操作记录”查看具体的报错信息。

步骤2 单击“操作记录”窗口中失败状态的报错信息。

步骤3 根据上一步获取的失败报错信息自行解决后，尝试重新创建集群。

----结束

20.3.1.2 集群的管理规模和控制节点的数量有关系吗?

集群管理规模是指：当前集群支持管理的最大节点数。若选择50节点，表示当前集群最多可管理50个节点。

针对不同的集群规模，控制节点的规格不同，但数量不受管理规模的影响。

集群的多控制节点模式开启后将创建三个控制节点，在单个控制节点发生故障后集群可以继续使用，不影响业务功能。

20.3.1.3 使用 CCE 需要关注哪些配额限制?

云容器引擎CCE配额**只限制了集群个数**，但是用户使用CCE时也会使用其他云服务，包括：弹性云服务器、云硬盘、虚拟私有云、弹性负载均衡、容器镜像服务等。

什么是配额?

为防止资源滥用，平台限定了各服务资源的配额，对用户的资源数量和容量做了限制。如您最多可以创建多少台弹性云服务器、多少块云硬盘。

如果当前资源配额限制无法满足使用需要，您可以申请扩大配额。

20.3.2 集群运行

20.3.2.1 当集群状态为“不可用”时，如何排查解决?

当集群状态显示为“不可用”时，请参照如下方式来排查解决。

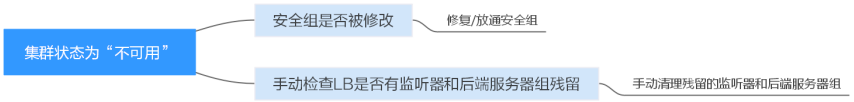
排查思路

以下排查思路根据原因的出现概率进行排序，建议您从高频率原因往低频率原因排查，从而帮助您快速找到问题的原因。

如果解决完某个可能原因仍未解决问题，请继续排查其他可能原因。

- **排查项一：安全组是否被修改**
- **排查项二：手动检查LB是否有监听器和后端服务器组残留**

图 20-1 排查思路



排查项一：安全组是否被修改

步骤1 登录控制台，选择“服务列表 > 网络 > 虚拟私有云 VPC”，单击左侧导航栏的“访问控制 > 安全组”，找到集群控制节点的安全组。

控制节点安全组名称为：集群名称-cce-**control**-编号。

步骤2 单击安全组名称，进入详情页面，请确保集群控制节点的安全组规则的正确性。

安全的详细说明请参见**集群安全组规则配置**

----结束

排查项二：手动检查 LB 是否有监听器和后端服务器组残留

模拟异常状态:

创删负载均衡（LoadBalancer，简称LB）类型service的任务执行时发生集群异常，恢复后会出现service删除成功，但是LB的监听器和后端服务器组残留。

步骤1 预创建CCE集群，在集群内使用nginx官方镜像创建工作负载、预置lb、各类型service、ingress等资源。

步骤2 保持集群正常运行，nginx负载处于稳态。

步骤3 持续间隔每20s创建删除10个lb类型的service。

步骤4 集群出现注入异常：如etcd实例不可用、集群休眠等问题。

----结束

问题原因：

异常注入时正在进行创建或删除过程中的lb-service被删除了，但是elb内有监听器和后端服务器组残留。

解决方案：

可以手动清理残留的监听器和后端服务器组。

步骤1 登录控制台，单击服务列表中“网络 > 弹性负载均衡 ELB”。

步骤2 在负载均衡器列表中，单击对应的ELB名称进入详情页，在“监听器”页签下找到残留的监听器，单击后方的删除图标进行删除操作。

步骤3 在“后端服务器组”页签下找到残留的后端服务器组，单击后方的删除图标进行删除操作。

----结束

20.3.2.2 集群删除之后相关数据能否再次找回?

集群删除之后，部署在集群上的工作负载也会同步删除，无法恢复，请慎重删除集群。

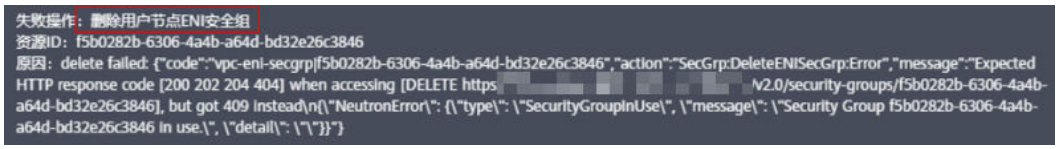
20.3.3 集群删除

20.3.3.1 集群删除失败：弹性网卡残留

CCE在删除集群时，会连接集群的kube-apiserver查询集群对接的周边资源信息，当CCE集群的状态为不可用，冻结，休眠等状态时，删除集群有可能会出现查询资源失败而导致集群删除失败的情况。

故障现象

删除集群失败。



问题根因

该场景引起的原因是连接集群的kube-apiserver查询集群对接的弹性网卡/弹性辅助网卡失败导致无法删除弹性网卡，CCE创建的用于弹性网卡/弹性辅助网卡的安全组由于弹性网卡残留删除时报错了409，最终导致了集群删除失败。

操作步骤

步骤1 复制报错信息中的资源ID，进入到VPC服务的安全组界面，根据ID过滤安全组。

步骤2 单击进入安全组详情界面，选择关联实例页签。

导致安全组残留的原因是关联了弹性网卡实例，辅助弹性网卡实例，单击其他页签，可以看到有残留的弹性网卡，将残留的弹性网卡（辅助弹性网卡会自动删除）删除。

步骤3 在弹性网卡界面将上一步查询到的网卡删除。

可以用ID过滤需要删除的弹性网卡，也可以通过集群ID的名称过滤需要删除的弹性网卡。

步骤4 清理完成后，到安全组确认clusterName-cce-eni-xxx的安全组已经没有关联的实例了，然后到CCE控制台就能正常删除集群了。

----结束

20.3.3.2 冻结或不可用的集群删除后如何清除残留资源

处于非运行状态（例如冻结、不可用状态）中的集群，由于无法获取集群中的PVC、Service、Ingress等资源，因此删除集群之后可能会残留网络及存储等资源，您需要前往资源所属服务手动删除。

弹性负载均衡资源

步骤1 前往弹性负载均衡控制台。

步骤2 通过集群使用的VPC ID进行过滤，得到该虚拟私有云下所有的弹性负载均衡实例。

步骤3 查看负载均衡实例下的监听器详情，描述中包含集群ID、Service ID等信息，说明该监听器由此集群创建。

步骤4 您可以根据上述信息将集群下残留的弹性负载均衡相关资源删除。

----结束

云硬盘资源

通过PVC动态创建方式创建的云硬盘名称格式为“pvc-{uid}”，且接口中的MetaData字段包含集群ID信息，您可以通过集群ID筛选出该集群中自动创建的云硬盘，根据需要进行删除。

步骤1 前往云硬盘控制台。

步骤2 通过名称“pvc-{uid}”进行过滤，得到所有由CCE自动创建的云硬盘实例。

步骤3 通过F12进入浏览器开发人员工具，查看detail接口中的MetaData字段包含集群ID信息，说明该云硬盘由此集群创建。

步骤4 您可以根据上述信息将集群下残留的云硬盘资源删除。

说明

删除后将无法恢复数据，请谨慎操作。

----结束

弹性文件服务资源

通过PVC动态创建方式创建的弹性文件服务容量型实例名称格式为“pvc-{uid}”，且接口中的MetaData字段包含集群ID信息，您可以通过集群ID筛选出该集群中自动创建的弹性文件服务容量型实例，根据需要进行删除。

步骤1 前往弹性文件服务控制台。

步骤2 通过名称“pvc-{uid}”进行过滤，得到所有由CCE自动创建的弹性文件实例。

步骤3 通过F12进入浏览器开发人员工具，查看detail接口中的MetaData字段包含集群ID信息，说明该弹性文件实例由此集群创建。

步骤4 您可以根据上述信息将集群下残留的弹性文件资源删除。

说明

删除后将无法恢复数据，请谨慎操作。

----结束

20.3.4 集群升级

20.3.4.1 CCE 集群升级时，升级集群插件失败如何排查解决?

概述

本文主要介绍在CCE在升级集群时，如何查找插件升级失败的原因，并解决问题。

操作步骤

步骤1 插件升级失败后，请优先进行重试。若重试不成功，则根据后续步骤排查问题。

步骤2 在升级界面显示失败后，请退出集群升级页面，前往“插件管理”界面查看插件的详细状态。针对异常插件，单击插件名称查看详情。

步骤3 在插件运行实例的详情界面，单击“事件”查看异常实例的信息。

步骤4 根据具体的异常信息进行相应处理，比如尝试删除未启动的实例让其重启等。

步骤5 处理成功后，插件状态会变为运行中，需要保证所有插件状态都处于运行中。

步骤6 此时进入集群升级界面，再次单击“重试”按钮即可。

----结束

20.4 节点

20.4.1 节点创建

20.4.1.1 CCE 集群新增节点时的问题与排查方法?

注意事项

- 同一集群下的节点镜像保证一致，后续新建/添加/纳管节点时需注意。
- 新建节点时，数据盘如需分配用户空间，分配目录注意不要设置关键目录，例如：如需放到home下，建议设置为/home/test，不要直接写到/home/下。

  说明

  请注意“挂载路径”不能设置为根目录“/”，否则将导致挂载失败。挂载路径一般设置为：

  - /opt/xxxx（但不能为/opt/cloud）
  - /mnt/xxxx（但不能为/mnt/paas）
  - /tmp/xxx
  - /var/xxx（但不能为/var/lib、/var/script、/var/paas等关键目录）
  - /xxxx（但不能和系统目录冲突，例如bin、lib、home、root、boot、dev、etc、lost+found、mnt、proc、sbin、srv、tmp、var、media、opt、selinux、sys、usr等）

  注意不能设置为/home/paas、/var/paas、/var/lib、/var/script、/mnt/paas、/opt/cloud，否则会导致系统或节点安装失败。

排查项一：提示子网配额不足

问题现象：

CCE集群中新增节点时无法添加新的节点，提示子网配额不足。

原因分析：

例：

VPC网段为：192.168.66.0/24

子网网段为：192.168.66.0/24

当前192.168.66.0/24子网内私有IP已占用251个。

解决方法：

步骤1 如需扩容需先扩容VPC。

登录控制台，在服务列表中单击“虚拟私有云 VPC”，在虚拟私有云列表中找到需要扩容的VPC，单击“操作”栏中的“编辑网段”。

步骤2 修改子网掩码为16位，单击“确定”按钮。

步骤3 单击VPC名称，在“基本信息”页签下单击右侧子网后的数字，在子网页面中创建新的子网规划。

步骤4 返回CCE新增节点页面，选择新的子网即可创建。

说明

1. 扩容后原VPC内子网192.168.66.0/24网段正常使用不受影响。
   创建CCE时选择新的子网网段即可，新子网网段上限也为251个私有IP，如仍无法满足业务需求，可继续新增子网。
2. 同VPC下不同子网间内网也是可以互信通信的。

----结束

排查项二：提示弹性 IP 不足

问题现象：

在CCE集群中新增节点时，在“弹性公网IP”处选择“自动创建”，但创建节点失败，提示弹性IP不足。

解决方法：

您可以有两种方法解决弹性IP不足的问题。

- **方法一：**解绑已绑定弹性IP的虚拟机，再重新添加节点。
  a. 登录控制台。
  b. 选择“计算> 弹性云服务 ECS”。
  c. 在弹性云服务器列表中，找到待解绑云服务器，单击云服务器名称。
  d. 在打开的弹性云服务器详情页中，单击“弹性公网IP”页签，在公网IP列表中单击待解绑IP后的“解绑”，为该云服务器解绑弹性IP，单击“确定”。
  e. 返回CCE控制台新增节点页面中，选择“使用已有”重新执行新增节点的操作。
- **方法二：**提高弹性IP的配额。

排查项三：节点安全组是否被修改或删除

问题现象：

在CCE集群中新增节点时创建失败。

解决方法：

您可单击集群名称，查看“集群信息”页面。在“网络信息”中单击“节点默认安全组”后的按钮，检查集群的节点默认安全组是否被删除，且安全组规则需要满足**集群安全组规则配置**。

如果您的账号下含有多个集群，需要统一管理节点的网络安全策略，您也可以指定自定义的安全组。

20.4.2 节点运行

20.4.2.1 集群可用，但节点状态为“不可用”？

当集群状态为“可用”，而集群中部分节点状态为“不可用”时，请参照如下方式来排查解决。

节点不可用检测机制说明

Kubernetes 节点发送的心跳确定每个节点的可用性，并在检测到故障时采取行动。检测的机制和间隔时间详细说明请参见**心跳**。

排查思路

以下排查思路根据原因的出现概率进行排序，建议您从高频率原因往低频率原因排查，从而帮助您快速找到问题的原因。

如果解决完某个可能原因仍未解决问题，请继续排查其他可能原因。

- **排查项一：节点负载过高**
- **排查项二：弹性云服务器是否删除或故障**
- **排查项三：弹性云服务器能否登录**
- **排查项四：安全组是否被修改**
- **排查项五：检查安全组规则中是否包含Master和Node互通的安全组策略**
- **排查项六：检查磁盘是否异常**
- **排查项七：内部组件是否正常**
- **排查项八：DNS地址配置错误**
- **排查项九：检查节点中的vdb盘是否被删除**
- **排查项十：排查Docker服务是否正常**

图 20-2 节点状态不可用排查思路

排查项一：节点负载过高

问题描述：

集群中节点连接异常，多个节点报写入错误，业务未受影响。

问题定位：

步骤1 登录CCE控制台，进入集群，在不可用节点所在行单击“监控”。

步骤2 单击“监控”页签顶部的“查看更多”，前往运维管理页面查看历史监控记录。

当节点cpu和内存负载过高时，会导致节点网络时延过高，或系统OOM，最终展示为不可用。

----结束

解决方案：

1. 建议迁移业务，减少节点中的工作负载数量，并对工作负载设置资源上限，降低节点CPU或内存等资源负载。

2. 将集群中对应的cce节点进行数据清理。
3. 限制每个容器的CPU和内存限制配额值。
4. 对集群进行节点扩容。
5. 您也可以重启节点，请至ECS控制台对节点进行重启。
6. 增加节点，将高内存使用的业务容器分开部署。
7. 重置节点。

节点恢复为可用后，工作负载即可恢复正常。

排查项二：弹性云服务器是否删除或故障

步骤1 确认集群是否可用。

登录CCE控制台，确定集群是否可用。

- 若集群非可用状态，如错误等，请参见**当集群状态为“不可用”时，如何排查解决？**。
- 若集群状态为“运行中”，而集群中部分节点状态为“不可用”，请执行**步骤2**。

步骤2 登录ECS控制台，查看对应的弹性云服务器状态。

- 若弹性云服务器状态为“已删除”：请在CCE中删除对应节点，再重新创建节点。
- 若弹性云服务器状态为“关机”或“冻结”：请先恢复弹性云服务器，约3分钟后集群节点可自行恢复。
- 若弹性云服务器出现故障：请先重启弹性云服务器，恢复故障。
- 若弹性云服务器状态为“可用”：请参考**排查项七：内部组件是否正常**登录弹性云服务器进行本地故障排查。

----结束

排查项三：弹性云服务器能否登录

步骤1 登录ECS控制台。

步骤2 确认界面显示的节点名称与虚机内的节点名称是否一致，并且密码或者密钥能否登录。

如果节点名称不一致，并且密码和密钥均不能登录，说明是ECS创建虚机时的cloudinit初始化问题，临时规避可以尝试重启节点，之后再提单给ECS确认问题根因。

----结束

排查项四：安全组是否被修改

登录VPC控制台，在左侧栏目树中单击“访问控制 > 安全组”，找到集群控制节点的安全组。

控制节点安全组名称为：集群名称-cce-**control**-编号。您可以通过**集群名称**查找安全组，再进一步在名称中区分“-cce-control-”字样，即为本集群安全组。

排查安全组中规则是否被修改，安全的详细说明请参见**集群安全组规则配置**

排查项五：检查安全组规则中是否包含 Master 和 Node 互通的安全组策略

请检查安全组规则中是否包含Master和Node互通的安全组策略。

已有集群添加节点时，如果子网对应的VPC新增了扩展网段且子网是扩展网段，要在控制节点安全组（即集群名称-cce-control-随机数）中添加如下三条安全组规则，以保证集群添加的节点功能可用（新建集群时如果VPC已经新增了扩展网段则不涉及此场景）。

安全的详细说明请参见**集群安全组规则配置**

排查项六：检查磁盘是否异常

新建节点会给节点绑定一个100G的docker专用数据盘。若数据盘卸载或损坏，会导致docker服务异常，最终导致节点不可用。

请检查节点挂载的数据盘是否已被卸载。若已卸载请重新挂载数据盘，再重启节点，节点可恢复。

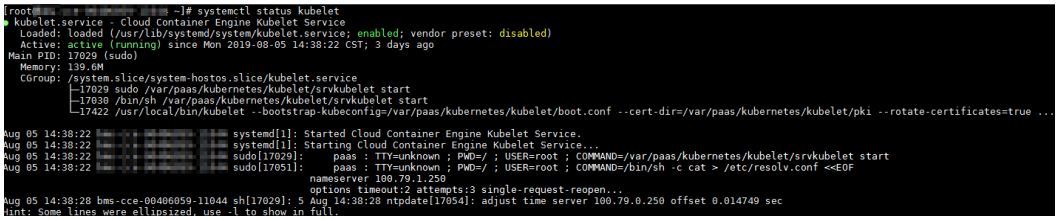
排查项七：内部组件是否正常

步骤1 登录不可用节点对应的弹性云服务器。

步骤2 执行以下命令判断paas组件是否正常。

systemctl status kubelet

执行成功，可查看到各组件的状态为Active，如下图：



若服务的组件状态不是Active，执行如下命令：

重启命令根据出错组件指定，如canal组件出错，则命令为：systemctl restart canal

重启后再查看状态：systemctl status canal

步骤3 若执行失败，请执行如下命令，查看monitrc进程的运行状态。

ps -ef | grep monitrc

若存在此进程，请终止此进程，进程终止后会自动重新拉起。

kill -s 9 `ps -ef | grep monitrc | grep -v grep | awk '{print $2}'`

----结束

排查项八：DNS 地址配置错误

步骤1 登录节点，在日志/var/log/cloud-init-output.log中查看是否有域名解析失败相关的报错。

cat /var/log/cloud-init-output.log | grep resolv

如果回显包含如下内容则说明无法解析该域名。

Could not resolve host: Unknown error

步骤2 在节点上ping上一步无法解析的域名，确认节点上能否解析此域名。

- 如果不能，则说明DNS无法解析该地址。请确认/etc/resolv.conf文件中的DNS地址与配置在VPC的子网上的DNS地址是否一致，通常是由于此DNS地址配置错误，导致无法解析此域名。请修改VPC子网DNS为正确配置，然后重置节点。
- 如果能，则说明DNS地址配置没有问题，请排查其他问题。

----结束

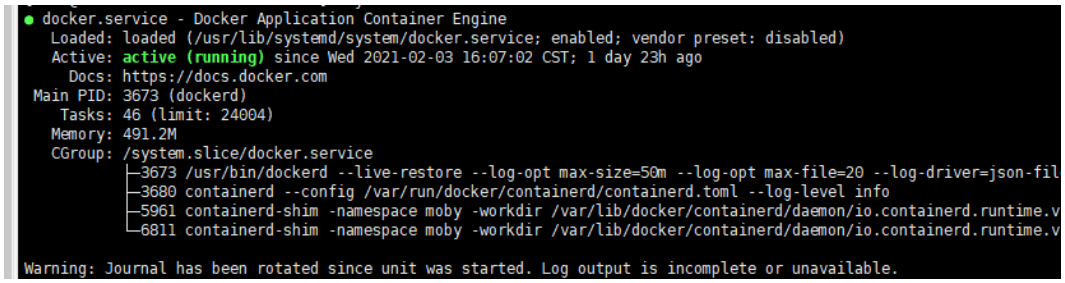
排查项九：检查节点中的 vdb 盘是否被删除

如果节点中的vdb盘被删除，可参考**此章节内容**恢复节点。

排查项十：排查 Docker 服务是否正常

步骤1 执行以下命令确认docker服务是否正在运行：

```
systemctl status docker
```



若执行失败或服务状态非active，请确认docker运行失败原因，必要时可提交工单联系技术支持。

步骤2 执行以下命令检查当前节点上所有容器数量：

```
docker ps -a | wc -l
```

若命令卡死、执行时间过长或异常容器数过多（1000以上），请确认外部是否存在重复不断地创删负载现象，在大量容器频繁创删过程中有可能出现大量异常容器且难以及时清理。

在此场景下可考虑停止重复创删负载或采用更多的节点去分摊负载，一般等待一段时间后节点会恢复正常，必要情况可执行docker rm {container_id}手动清理异常容器。

----结束

20.4.2.2 如何重置 CCE 集群中节点的密码?

问题背景

在CCE中创建节点时，您选择了使用密钥对或者密码作为登录方式，当密钥对或密码丢失时，您可以登录ECS控制台对节点进行密码重置操作，重置密码后即可使用密码登录CCE服务中的节点。

操作步骤

步骤1 登录ECS控制台。

步骤2 在左侧弹性云服务器列表中，选择待操作节点对应的云服务器，单击后方操作列中的“更多 > 关机”。

步骤3 待云服务器关机后，单击待操作节点后方操作列中的“更多 > 重置密码”，按照界面提示进行操作即可重置密码。

步骤4 密码重置完成后，单击待操作节点后方操作列中的“更多 > 开机”，单击后方的“远程登录”即可通过密码登录该节点。

----结束

20.4.2.3 如何收集 CCE 集群中节点的日志?

CCE节点日志文件如下表所示。

表 20-1 节点日志列表

日志名称	路径
kubelet日志	• v1.21及以上版本集群：/var/log/cce/kubernetes/kubelet.log • v1.19及以下版本集群：/var/paas/sys/log/kubernetes/kubelet.log
kube-proxy日志	• v1.21及以上版本集群：/var/log/cce/kubernetes/kube-proxy.log • v1.19及以下版本集群：/var/paas/sys/log/kubernetes/kube-proxy.log
yangtse日志（网络）	• v1.21及以上版本集群：/var/log/cce/yangtse • v1.19及以下版本集群：/var/paas/sys/log/yangtse
canal日志	• v1.21及以上版本集群：/var/log/cce/canal • v1.19及以下版本集群：/var/paas/sys/log/canal
系统日志	/var/log/messages

表 20-2 插件日志列表

插件日志名称	路径
everest插件日志	• 2.1.41及以上版本插件： – everest-csi-driver：/var/log/cce/kubernetes – everest-csi-controller：/var/paas/sys/log/kubernetes • 2.1.41以下版本插件： – everest-csi-driver：/var/log/cce/everest-csi-driver – everest-csi-controller：/var/paas/sys/log/everest-csi-controller
npd插件日志	• 1.18.16及以上版本插件：/var/paas/sys/log/kubernetes • 1.18.16以下版本插件：/var/paas/sys/log/cceaddon-npd
cce-hpa-controller插件日志	• 1.3.12及以上版本插件：/var/paas/sys/log/kubernetes • 1.3.12以下版本插件：/var/paas/sys/log/ccehpa-controller

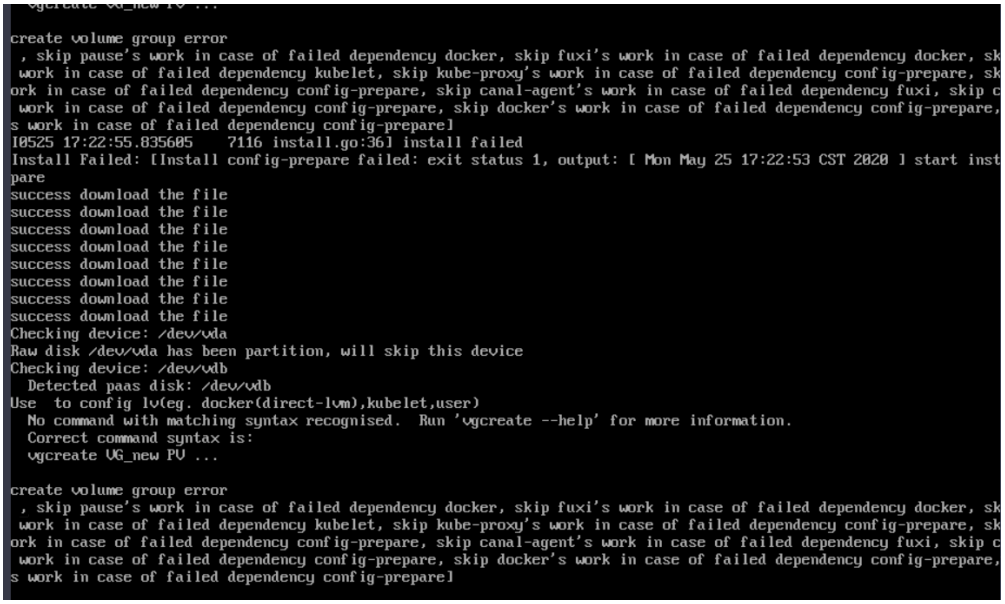
20.4.2.4 Node 节点 vdb 盘受损，通过重置节点仍无法恢复节点?

问题现象

客户node节点vdb盘受损，通过重置节点，无法恢复节点。

问题过程：

- 在一个正常的node节点上，删除lv，删除vg，节点不可用。
- 重置异常节点，重置过程中，报语法错误，而且节点不可用。
  如下图：



问题定位

node节点中vg被删除或者损坏无法识别，为了避免重置的时候误格式化用户的数据盘，需要先手动恢复vg，这样重置的时候就不会去格式化其余的数据盘。

解决方案

步骤1 登录节点。

步骤2 重新创建PV和VG，但是创建时报错：

```
root@host1:~# pvcreate /dev/vdb
Device /dev/vdb excluded by a filter
```

这是由于添加的磁盘是在另一个虚拟机中新建的，已经存在了分区表，当前虚拟机并不能识别磁盘的分区表，运行parted命令重做分区表，中途需要输入三次命令。

```
root@host1:~# parted /dev/vdb
GNU Parted 3.2
Using /dev/vdb
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) mklabel msdos
Warning: The existing disk label on /dev/vdb will be destroyed and all data on this disk will be lost. Do you want to continue?
Yes/No? yes
(parted) quit
Information: You may need to update /etc/fstab.
```

再次运行pvcreate，当询问是否擦除dos签名时，输入**y**，就可以将磁盘创建为PV。

```
root@host1:~# pvcreate /dev/vdb
WARNING: dos signature detected on /dev/vdb at offset 510. Wipe it? [y/n]: y
Wiping dos signature on /dev/vdb.
Physical volume "/dev/vdb" successfully created
```

步骤3 创建VG。

判断该节点的docker盘，如果是/dev/vdb和/dev/vdc两个盘，则执行下面的命令：

```
root@host1:~# vgcreate vgpaas /dev/vdb /dev/vdc
```

如果只有/dev/vdb盘，则执行下面的命令：

```
root@host1:~# vgcreate vgpaas /dev/vdb
```

创建完成后，重置节点即可恢复。

----结束

20.4.2.5 容器使用 SCSI 类型云硬盘偶现 IO 卡住

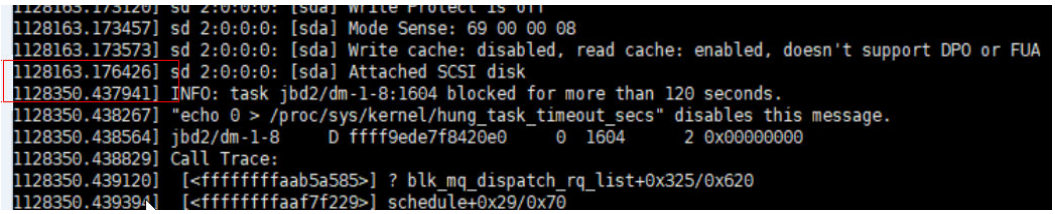
问题描述

容器使用SCSI类型的云硬盘存储，在CentOS节点上创建和删除容器触发磁盘频繁挂载卸载的场景，有概率会出现系统盘读写瞬时冲高，然后系统卡住的问题，影响节点正常工作。

出现该问题时，可在dmesg日志中观察到：

```
Attached SCSI disk
task jdb2/xxx blocked for more than 120 seconds.
```

如下图红框所示：



问题原理

BUS 0上热插PCI设备后，Linux内核会多次遍历挂载在BUS 0上的所有PCI-Bridge，且PCI-Bridge在被更新期间无法正常工作。在此期间，若设备使用的PCI-Bridge被更新，由于内核缺陷，该设备会认为PCI-Bridge异常，设备进入故障模式进而无法正常工作。如果此时前端正要写PCI配置空间让后端处理磁盘IO，那么这个写配置空间操作就可能会被剔除，导致后端接收不到通知去处理IO环上的新增请求，最终表现为前端IO卡住。

该问题由linux内核缺陷引起，详见Linux发行版缺陷列表：**https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=51c48b310183ab6ba5419edfc6a8de889cc04521**。

影响范围

对CentOS Linux内核3.10.0-1127.el7之前的版本有影响。

解决方法

通过**重置节点**将内核升级至高版本。

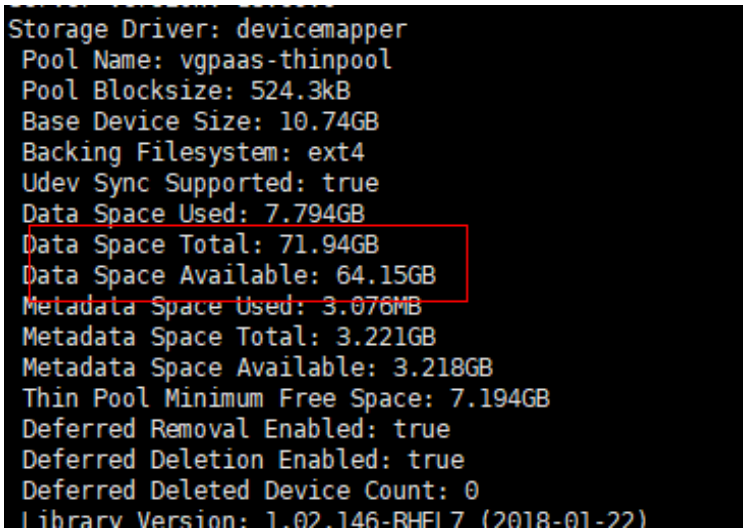
20.4.2.6 thinpool 磁盘空间耗尽导致容器或节点异常时，如何解决?

问题描述

当节点上的thinpool磁盘空间接近写满时，概率性出现以下异常：

在容器内创建文件或目录失败、容器内文件系统只读、节点被标记disk-pressure污点及节点不可用状态等。

用户可手动在节点上执行docker info查看当前thinpool空间使用及剩余量信息，从而定位该问题。如下图：



问题原理

docker devicemapper模式下，尽管可以通过配置basesize参数限制单个容器的主目录大小（默认为10GB），但节点上的所有容器还是共用节点的thinpool磁盘空间，并不是完全隔离，当一些容器使用大量thinpool空间且总和达到节点thinpool空间上限时，也会影响其他容器正常运行。

另外，在容器的主目录中创删文件后，其占用的thinpool空间不会立即释放，因此即使basesize已经配置为10GB，而容器中不断创删文件时，占用的thinpool空间会不断增加一直到10GB为止，后续才会复用这10GB空间。如果节点上的**业务容器数*basesize > 节点thinpool空间大小**，理论上有概率出现节点thinpool空间耗尽的场景。

解决方案

当节点已出现thinpool空间耗尽时，可将部分业务迁移至其他节点实现业务快速恢复。但对于此类问题，建议采用以下方案从根因上解决问题：

方案1：

合理规划业务分布及数据面磁盘空间，避免和减少出现**业务容器数*basesize > 节点thinpool空间大小**场景。如需对thinpool空间进行扩容，请参考以下步骤：

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。

```
# lsblk
NAME                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                   8:0    0   50G  0 disk
└─sda1                8:1    0   50G  0 part /
sdb                   8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker                # 容器引擎使用的空间
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
```

  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

- Devicemapper，单独划分了thinpool存储镜像相关数据。

```
# lsblk
NAME                        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                           8:0    0   50G  0 disk
└─sda1                        8:1    0   50G  0 part /
sdb                           8:16   0  200G  0 disk
├─vgpaas-dockersys          253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta     253:1    0    3G  0 lvm
│ └─vgpaas-thinpool         253:3    0   67G  0 lvm                # thinpool空间
│   ...
├─vgpaas-thinpool_tdata     253:2    0   67G  0 lvm
│ └─vgpaas-thinpool         253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes         253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到thinpool盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/thinpool
```

- 在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----结束

方案2：

容器业务的创删文件操作建议在容器挂载的本地存储（如emptyDir、hostPath）或云存储的目录中进行，这样不会占用thinpool空间。

方案3：

使用overlayfs存储模式的操作系统，可将业务部署在此类节点上，避免容器内创删文件后占用的磁盘空间不立即释放问题。

20.4.2.7 GPU 节点使用 nvidia 驱动启动容器排查思路

集群中的节点是否有资源调度失败的事件?

问题现象：

节点运行正常且有GPU资源，但报如下失败信息：

0/9 nodes are aviable: 9 insufficient nvida.com/gpu

排查思路：

1. 确认节点标签是否已经打上nvidia资源。

2. 查看nvidia驱动运行是否正常。

   到插件运行所在的节点上，查看驱动的安装日志，路径如下所示：

   ```
   /opt/cloud/cce/nvidia/nvidia_installer.log
   ```

   查看nvidia容器标准输出日志：

   过滤容器id

   ```
   docker ps –a | grep nvidia
   ```

   查看日志

   ```
   docker logs 容器id
   ```

业务上报 nvidia 版本和 cuda 版本不匹配?

容器中查看cuda的版本，执行如下命令：

```
cat /usr/local/cuda/version.txt
```

然后查看容器所在节点的nvidia驱动版本支持的cuda版本范围，是否包含容器中的cuda版本。

相关链接

工作负载异常：GPU节点部署服务报错

20.4.3 规格配置变更

20.4.3.1 如何变更 CCE 集群中的节点规格?

操作方法

注意

如果需要变更规格的节点是纳管到集群中的，可将节点从CCE集群中移除后再变更节点规格，避免影响业务。

步骤1 登录CCE控制台，进入集群，在左侧选择“节点管理”，在右侧单击节点名称，跳转到弹性云服务器详情页。

步骤2 在弹性云服务器详情页中，单击右上角的“关机”，关机完成后单击“更多 > 变更规格”。

步骤3 在“云服务器变更规格”页面中根据业务需求选择相应的规格，单击“提交”完成节点规格的变更，返回弹性云服务器列表页，将该云服务器执行“开机”操作。

步骤4 登录CCE控制台，进入集群，在节点管理列表中找到该节点，并单击操作栏中的“同步云服务器”，同步后即可看到节点规格已与弹性云服务器中变更的规格一致。

----结束

常见问题

配置了CPU管理策略绑核的节点，在变更规格后，可能会无法重新拉起或创建工作负载。如发生此种情况请参见**CCE节点变更规格后，为什么无法重新拉起或创建工作负载?** 解决。

20.4.3.2 CCE 节点变更规格后，为什么无法重新拉起或创建工作负载?

问题背景

kubelet启动参数中默认将CPU Manager的策略设置为static，允许为节点上具有某些资源特征的pod赋予增强的CPU亲和性和独占性。用户如果直接在ECS控制台对CCE节点变更规格，会由于变更前后CPU信息不匹配，导致节点上的负载无法重新拉起，也无法创建新负载。

更多信息请参见**Kubernetes控制节点上的CPU管理策略**。

影响范围

开启了CPU管理策略的集群。

解决方案

步骤1 登录CCE节点（弹性云服务器）并删除**cpu_manager_state**文件。

删除命令示例如下：

```
rm -rf /mnt/paas/kubernetes/kubelet/cpu_manager_state
```

步骤2 重启节点或重启kubelet，重启kubelet的方法如下：

```
systemctl restart kubelet
```

步骤3 此时重新拉起或创建工作负载，已可成功执行。

----结束

20.5 节点池

20.5.1 节点池一直在扩容中，但“操作记录”里却没有创建节点的记录?

问题现象

节点池的状态一直处于“扩容中”，但是“操作记录”里面没有看到有对应创建节点的记录。

原因排查：

检查如下问题并修复：

- 查看节点池配置的规格是否资源不足。
- 租户的ECS或内存配额是否不足。
- 如果一次创建节点太多，可能会出现租户的ECS容量校验不过的情况发生。

解决方案：

- 若ECS节点资源不足，使用其他规格节点替代。
- 若ECS或内存配额不足，请扩大配额。
- 若ECS容量校验不通过，请重新校验。

20.6 工作负载

20.6.1 工作负载异常

20.6.1.1 工作负载状态异常定位方法

工作负载状态异常时，建议先查看Pod的事件以便于确定导致异常的初步原因，再参照**表20-3**中的内容针对性解决问题。

表 20-3 排查思路列表

事件信息	实例状态	处理措施
实例调度失败	Pending	请参考**工作负载异常：实例调度失败**
拉取镜像失败 重新拉取镜像失败	FailedPullImage ImagePullBackOff	请参考**工作负载异常：实例拉取镜像失败**
启动容器失败 重新启动容器失败	CreateContainerError CrashLoopBackOff	请参考**工作负载异常：启动容器失败**
实例状态为“Evicted”，pod不断被驱逐	Evicted	请参考**工作负载异常：实例驱逐异常（Evicted）**
实例挂卷失败	Pending	请参考**工作负载异常：存储卷无法挂载或挂载超时**
实例状态一直为“创建中”	Creating	请参考**工作负载异常：一直处于创建中**
实例状态一直为“结束中”	Terminating	请参考**工作负载异常：结束中，解决Terminating状态的Pod删不掉的问题**
实例状态为“已停止”	Stopped	请参考**工作负载异常：已停止**

Pod 事件查看方法

Pod的事件可以使用kubectl describe pod *{pod-name}*命令查看，或在CCE控制台，工作负载详情页面中查看。

```
$ kubectl describe pod prepare-58bd7bdf9-fthrp
...
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  49s   default-scheduler  0/2 nodes are available: 2 Insufficient cpu.
  Warning  FailedScheduling  49s   default-scheduler  0/2 nodes are available: 2 Insufficient cpu.
```

20.6.1.2 工作负载异常：实例调度失败

问题定位

当Pod状态为“Pending”，事件中出现“实例调度失败”的信息时，可根据具体事件信息确定具体问题原因。事件查看方法请参见**工作负载状态异常定位方法**。

排查思路

根据具体事件信息确定具体问题原因，如**表20-4**所示。

表 20-4 实例调度失败

事件信息	问题原因与解决方案
no nodes available to schedule pods.	集群中没有可用的节点。 **排查项一：集群内是否无可用节点**
0/2 nodes are available: 2 Insufficient cpu. 0/2 nodes are available: 2 Insufficient memory.	节点资源（CPU、内存）不足。 **排查项二：节点资源（CPU、内存等）是否充足**
0/2 nodes are available: 1 node(s) didn't match node selector, 1 node(s) didn't match pod affinity rules, 1 node(s) didn't match pod affinity/anti-affinity.	节点与Pod亲和性配置互斥，没有满足Pod要求的节点。 **排查项三：检查工作负载的亲和性配置**
0/2 nodes are available: 2 node(s) had volume node affinity conflict.	Pod挂载云硬盘存储卷与节点不在同一个可用区。 **排查项四：挂载的存储卷与节点是否处于同一可用区**
0/1 nodes are available: 1 node(s) had taints that the pod didn't tolerate.	节点存在污点Tanits，而Pod不能容忍这些污点，所以不可调度。 **排查项五：检查Pod污点容忍情况**
0/7 nodes are available: 7 Insufficient ephemeral-storage.	节点临时存储不足。 **排查项六：检查临时卷使用量**
0/1 nodes are available: 1 everest driver not found at node	节点上everest-csi-driver不在running状态。 **排查项七：检查everest插件是否工作正常**
Failed to create pod sandbox: ... Create more free space in thin pool or use dm.min_free_space option to change behavior	节点thinpool空间不足。 **排查项八：检查节点thinpool空间是否充足**
0/1 nodes are available: 1 Too many pods.	该节点调度的Pod超出上限。 **检查项九：检查节点上调度的Pod是否过多**

排查项一：集群内是否无可用节点

登录CCE控制台，检查节点状态是否为可用。或使用如下命令查看节点状态是否为Ready。

```
$ kubectl get node
NAME          STATUS   ROLES    AGE   VERSION
192.168.0.37   Ready    <none>   21d   v1.19.10-r1.0.0-source-121-gb9675686c54267
192.168.0.71   Ready    <none>   21d   v1.19.10-r1.0.0-source-121-gb9675686c54267
```

如果状态都为不可用（Not Ready），则说明集群中无可用节点。

解决方案：

- 新增节点，若工作负载未设置亲和策略，pod将自动迁移至新增的可用节点，确保业务正常。
- 排查不可用节点问题并修复，排查修复方法请参见**集群可用，但节点状态为“不可用”？**。
- 重置不可用的节点。

排查项二：节点资源（CPU、内存等）是否充足

0/2 nodes are available: 2 Insufficient cpu. CPU不足。

0/2 nodes are available: 2 Insufficient memory. 内存不足。

当“实例资源的申请量”超过了“实例所在节点的可分配资源总量”时，节点无法满足实例所需资源要求导致调度失败。

如果节点可分配资源小于Pod的申请量，则节点无法满足实例所需资源要求导致调度失败。

解决方案：

资源不足的情况主要解决办法是扩容，建议在集群中增加节点数量。

排查项三：检查工作负载的亲和性配置

当亲和性配置出现如下互斥情况时，也会导致实例调度失败：

例如：

workload1、workload2设置了工作负载间的反亲和，如workload1部署在Node1，workload2部署在Node2。

workload3部署上线时，既希望与workload2亲和，又希望可以部署在不同节点如Node1上，这就造成了工作负载亲和与节点亲和间的互斥，导致最终工作负载部署失败。

0/2 nodes are available: 1 node(s) didn't match **node selector**, 1 node(s) didn't match **pod affinity rules**, 1 node(s) didn't match **pod affinity/anti-affinity**.

- **node selector** 表示节点亲和不满足。
- **pod affinity rules** 表示Pod亲和不满足。
- **pod affinity/anti-affinity** 表示Pod亲和/反亲和不满足。

解决方案：

- 在设置“工作负载间的亲和性”和“工作负载和节点的亲和性”时，需确保不要出现互斥情况，否则工作负载会部署失败。
- 若工作负载配置了节点亲和性，需确保亲和的节点标签中supportContainer设置为true，否则会导致pod无法调动到节点上，查看事件提示如下错误信息：
  ```
  No nodes are available that match all of the following predicates: MatchNode Selector, NodeNotSupportsContainer
  ```
  节点标签为false时将会调度失败。

排查项四：挂载的存储卷与节点是否处于同一可用区

0/2 nodes are available: 2 node(s) had volume node affinity conflict. 存储卷与节点之间存在亲和性冲突，导致无法调度。

这是因为云硬盘不能跨可用区挂载到节点。例如云硬盘存储卷在可用区1，节点在可用区2，则会导致无法调度。

CCE中创建云硬盘存储卷，默认带有亲和性设置，如下所示：

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pvc-c29bfac7-efa3-40e6-b8d6-229d8a5372ac
spec:
  ...
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: failure-domain.beta.kubernetes.io/zone
              operator: In
              values:
                -
```

解决方案：

重新创建存储卷，可用区选择与节点同一分区，或重新创建工作负载，存储卷选择自动分配。

排查项五：检查 Pod 污点容忍情况

0/1 nodes are available: 1 node(s) had taints that the pod didn't tolerate. 是因为节点打上了污点，不允许Pod调度到节点上。

查看节点的上污点的情况。如下则说明节点上存在污点。

```
$ kubectl describe node 192.168.0.37
Name:               192.168.0.37
...
Taints:             key1=value1:NoSchedule
...
```

在某些情况下，系统会自动给节点添加一个污点。当前内置的污点包括：

- node.kubernetes.io/not-ready：节点未准备好。
- node.kubernetes.io/unreachable：节点控制器访问不到节点。
- node.kubernetes.io/memory-pressure：节点存在内存压力。
- node.kubernetes.io/disk-pressure：节点存在磁盘压力，此情况下您可通过**节点磁盘空间不足**的方案进行解决。
- node.kubernetes.io/pid-pressure：节点的 PID 压力。
- node.kubernetes.io/network-unavailable：节点网络不可用。
- node.kubernetes.io/unschedulable：节点不可调度。
- node.cloudprovider.kubernetes.io/uninitialized：如果kubelet启动时指定了一个“外部”云平台驱动， 它将给当前节点添加一个污点将其标志为不可用。在cloud-controller-manager初始化这个节点后，kubelet将删除这个污点。

解决方案：

要想把Pod调度到这个节点上，有两种方法：

- 若该污点为用户自行添加，可考虑删除节点上的污点。若该污点为**系统自动添加**，解决相应问题后污点会自动删除。
- Pod的定义中容忍这个污点，如下所示。详细内容请参见**污点和容忍**。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:alpine
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoSchedule"
```

排查项六：检查临时卷使用量

0/7 nodes are available: 7 Insufficient ephemeral-storage. 节点临时存储不足。

检查Pod是否限制了临时卷的大小，如下所示，当应用程序需要使用的量超过节点已有容量时会导致无法调度，修改临时卷限制或扩容节点磁盘可解决此问题。

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
    volumeMounts:
    - name: ephemeral
      mountPath: "/tmp"
  volumes:
    - name: ephemeral
      emptyDir: {}
```

您可以通过**kubectl describe node**命令查询节点临时卷的容量（Capacity）和可使用量（Allocatable），并可查询节点已分配的临时卷申请值和限制值。

返回示例如下：

```
...
Capacity:
  cpu:                4
  ephemeral-storage:  61607776Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  localssd:           0
  localvolume:        0
  memory:             7614352Ki
  pods:               40
Allocatable:
  cpu:                3920m
  ephemeral-storage:  56777726268
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  localssd:           0
  localvolume:        0
```

```
  memory:             6180752Ki
  pods:               40
...
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests      Limits
  --------           --------      ------
  cpu                1605m (40%)   6530m (166%)
  memory             2625Mi (43%)  5612Mi (92%)
  ephemeral-storage  0 (0%)        0 (0%)
  hugepages-1Gi      0 (0%)        0 (0%)
  hugepages-2Mi      0 (0%)        0 (0%)
  localssd           0             0
  localvolume        0             0
Events:              <none>
```

排查项七：检查 everest 插件是否工作正常

0/1 nodes are available: 1 everest driver not found at node。集群everest插件的everest-csi-driver 在节点上未正常启动。

检查kube-system命名空间下名为everest-csi-driver的守护进程，查看对应Pod是否正常启动，若未正常启动，删除该Pod，守护进程会重新拉起该Pod。

排查项八：检查节点 thinpool 空间是否充足

节点在创建时会绑定一个供kubelet及容器引擎使用的专用数据盘。若数据盘空间不足，将导致实例无法正常创建。

方案一：清理镜像

您可以执行以下步骤清理未使用的镜像：

- 使用containerd容器引擎的节点：
  a. 查看节点上的本地镜像。
     ```
     crictl images -v
     ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
     ```
     crictl rmi {镜像ID}
     ```
- 使用docker容器引擎的节点：
  a. 查看节点上的本地镜像。
     ```
     docker images
     ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
     ```
     docker rmi {镜像ID}
     ```

说明

请勿删除cce-pause等系统镜像，否则可能导致无法正常创建容器。

方案二：扩容磁盘

扩容磁盘的操作步骤如下：

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。

```
# lsblk
NAME              MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                 8:0    0   50G  0 disk
└─sda1              8:1    0   50G  0 part /
sdb                 8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker              # 容器引擎使用的空间
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
```

  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

- Devicemapper，单独划分了thinpool存储镜像相关数据。

```
# lsblk
NAME                          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                             8:0    0   50G  0 disk
└─sda1                          8:1    0   50G  0 part /
sdb                             8:16   0  200G  0 disk
├─vgpaas-dockersys              253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta         253:1    0    3G  0 lvm
│ └─vgpaas-thinpool             253:3    0   67G  0 lvm                # thinpool空间
│   ...
├─vgpaas-thinpool_tdata          253:2    0   67G  0 lvm
│ └─vgpaas-thinpool             253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes             253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到thinpool盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/thinpool
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----结束

检查项九：检查节点上调度的 Pod 是否过多

0/1 nodes are available: 1 Too many pods.表示节点上调度的Pod过多，超出可调度的最大实例数。

创建节点时，在“高级配置”中可选择设置“最大实例数”参数，设置节点上可以正常运行的容器 Pod 的数目上限。该数值的默认值随节点规格浮动，您也可以手动设置。

您可以在“节点管理”页面，查看节点的“容器组(已分配/总额度)”参数列，检查节点已调度的容器是否达到上限。若已达到上限，可通过添加节点或修改最大实例数的方式解决。

您可通过以下方式修改“最大实例数”参数：

- 默认节点池中的节点：通过重置节点时修改“最大实例数”。
- 自定义节点池中的节点：可修改节点池配置参数中的max-pods参数。

20.6.1.3 工作负载异常：实例拉取镜像失败

问题定位

当工作负载状态显示“实例未就绪：Back-off pulling image "xxxxx"”，该状态下工作负载实例K8s事件名称为“实例拉取镜像失败”或“重新拉取镜像失败”。查看K8s事件的方法请参见**Pod事件查看方法**。

排查思路

根据具体事件信息确定具体问题原因，如**表20-5**所示。

表 20-5 实例拉取镜像失败

事件信息	问题原因与解决方案
Failed to pull image "xxx": rpc error: code = Unknown desc = Error response from daemon: Get xxx: denied: You may not login yet	没有登录镜像仓库，无法拉取镜像。 **排查项一：kubectl创建工作负载时未指定imagePullSecret**
Failed to pull image "nginx:v1.1": rpc error: code = Unknown desc = Error response from daemon: Get https://registry-1.docker.io/v2/: dial tcp: lookup registry-1.docker.io: no such host	镜像地址配置有误找不到镜像导致失败。 **排查项二：填写的镜像地址错误（使用第三方镜像时）** **排查项三：使用错误的密钥（使用第三方镜像时）**
Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a sandbox for pod "nginx-6dc48bf8b6-l8xrw": Error response from daemon: mkdir xxxxx: no space left on device	磁盘空间不足。 **排查项四：节点磁盘空间不足**
Failed to pull image "xxx": rpc error: code = Unknown desc = error pulling image configuration: xxx x509: certificate signed by unknown authority	从第三方仓库下载镜像时，第三方仓库使用了非知名或者不安全的证书. **排查项五： 远程镜像仓库使用非知名或不安全的证书**
Failed to pull image "XXX": rpc error: code = Unknown desc = context canceled	镜像体积过大。 **排查项六： 镜像过大导致失败**
Failed to pull image "docker.io/bitnami/nginx:1.22.0-debian-11-r3": rpc error: code = Unknown desc = Error response from daemon: Get https://registry-1.docker.io/v2/: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)	**排查项七：无法连接镜像仓库**

事件信息	问题原因与解决方案
ERROR: toomanyrequests: Too Many Requests. 或 you have reached your pull rate limit, you may increase the limit by authenticating an upgrading	由于拉取镜像次数达到上限而被限速。 **排查项八：拉取公共镜像达上限**

排查项一：kubectl 创建工作负载时未指定 imagePullSecret

当工作负载状态异常并显示“实例拉取镜像失败”的K8s事件时，请排查yaml文件中是否存在**imagePullSecrets**字段。

排查事项：

- 当Pull SWR容器镜像仓库的镜像时，name参数值需固定为default-secret。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

- Pull第三方镜像仓库的镜像时，需设置为创建的secret名称。

  kubectl创建工作负载拉取第三方镜像时，需指定的imagePullSecret字段，name表示pull镜像时的secret名称。

排查项二：填写的镜像地址错误（使用第三方镜像时）

CCE支持拉取第三方镜像仓库中的镜像来创建工作负载。

在填写第三方镜像的地址时，请参照要求的格式来填写。镜像地址格式为：ip:port/path/name:version或name:version，若没标注版本号则默认版本号为latest。

- 若是私有仓库，请填写ip:port/path/name:version。
- 若是docker开源仓库，请填写name:version，例如nginx:latest。

镜像地址配置有误找不到镜像导致失败，Kubernetes Event中提示如下信息：

```
Failed to pull image "nginx:v1.1": rpc error: code = Unknown desc = Error response from daemon: Get https://registry-1.docker.io/v2/: dial tcp: lookup registry-1.docker.io: no such host
```

解决方案：

可编辑yaml修改镜像地址，也可在工作负载详情页面更新升级页签单击更换镜像。

排查项三：使用错误的密钥（使用第三方镜像时）

通常第三方镜像仓库都必须经过认证（账号密码）才可以访问，而CCE中容器拉取镜像是使用密钥认证方式，这就要求在拉取镜像前必须先创建镜像仓库的密钥。

解决方案：

若您的密钥错误将会导致镜像拉取失败，请重新获取密钥。

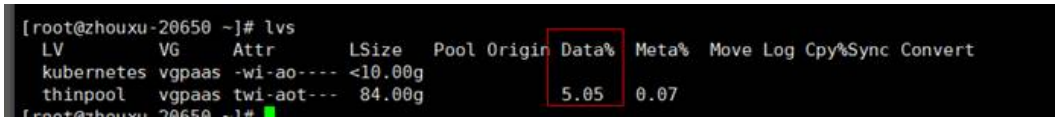
排查项四：节点磁盘空间不足

当k8s事件中包含以下信息，表明节点上用于存储镜像的磁盘空间已满，会导致重新拉取镜像失败。您可以通过清理镜像或扩容磁盘解决该问题。

```
Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a sandbox for pod "nginx-6dc48bf8b6-l8xrw": Error response from daemon: mkdir xxxxx: no space left on device
```

您可以执行以下命令，确认节点上存储镜像的磁盘空间：

```
lvs
```



方案一：清理镜像

您可以执行以下步骤清理未使用的镜像：

- 使用containerd容器引擎的节点：
  a. 查看节点上的本地镜像。
     ```
     crictl images -v
     ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
     ```
     crictl rmi {镜像ID}
     ```
- 使用docker容器引擎的节点：
  a. 查看节点上的本地镜像。
     ```
     docker images
     ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
     ```
     docker rmi {镜像ID}
     ```

说明

请勿删除cce-pause等系统镜像，否则可能导致无法正常创建容器。

方案二：扩容磁盘

扩容磁盘的操作步骤如下：

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。

```
# lsblk
NAME               MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                8:0    0   50G  0 disk
└─sda1               8:1    0   50G  0 part /
sdb                8:16   0  200G  0 disk
├─vgpaas-dockersys 253:0    0   90G  0 lvm  /var/lib/docker              # 容器引擎使用的空间
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
```

  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

- Devicemapper，单独划分了thinpool存储镜像相关数据。

```
# lsblk
NAME                          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                           8:0    0   50G  0 disk
└─sda1                          8:1    0   50G  0 part /
sdb                           8:16   0  200G  0 disk
├─vgpaas-dockersys              253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta           253:1    0    3G  0 lvm
│ └─vgpaas-thinpool               253:3    0   67G  0 lvm                # thinpool空间
│   ...
├─vgpaas-thinpool_tdata           253:2    0   67G  0 lvm
│ └─vgpaas-thinpool               253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes               253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到thinpool盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/thinpool
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----结束

排查项五： 远程镜像仓库使用非知名或不安全的证书

从第三方仓库下载镜像时，若第三方仓库使用了非知名或者不安全的证书，节点上会拉取镜像失败，Pod事件列表中有“实例拉取镜像失败”事件，报错原因为"x509: certificate signed by unknown authority"。

说明

当前EulerOS 2.9镜像中有进行安全增强，移除系统中部分非安全或过期知名证书配置，部分第三方镜像在其他类型节点上未报错，在EulerOS 2.9系统报此错误属正常现象，也可通过下述解决方案进行处理。

解决方案：

步骤1 确认报错unknown authority的第三方镜像服务器地址和端口。

从"实例拉取镜像失败"事件信息中能够直接看到报错的第三方镜像服务器地址和端口，如上图中错误信息为：

```
Failed to pull image "bitnami/redis-cluster:latest": rpc error: code = Unknown desc = error pulling image configuration: Get https://production.cloudflare.docker.com/registry-v2/docker/registry/v2/blobs/sha256/e8/e83853f03a2e792614e7c1e6de75d63e2d6d633b4e7c39b9d700792ee50f7b56/data?verify=1636972064-AQbl5RActnudDZV%2F3EShZwnqOe8%3D: x509: certificate signed by unknown authority
```

对应的第三方镜像服务器地址为 *production.cloudflare.docker.com*，端口为https默认端口*443*。

步骤2 在需要下载第三方镜像的节点上加载第三方镜像服务器的根证书。

EulerOS, CentOS节点执行如下命令，{server_url}:{server_port}需替换成步骤1中地址和端口，如 production.cloudflare.docker.com:443。

若节点的容器引擎为containerd，最后一步“systemctl restart docker”命令替换为"systemctl restart containerd"。

```
openssl s_client -showcerts -connect ***{server_url}:{server_port}*** < /dev/null | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > /etc/pki/ca-trust/source/anchors/tmp_ca.crt
update-ca-trust
systemctl restart docker
```

ubuntu节点执行如下命令。

```
openssl s_client -showcerts -connect ***{server_url}:{server_port***} < /dev/null | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > /usr/local/share/ca-certificates/tmp_ca.crt
update-ca-trust
systemctl restart docker
```

----结束

排查项六： 镜像过大导致失败

Pod事件列表中有“实例拉取镜像失败”事件，报错原因如下。这可能是镜像较大导致的情况。

```
Failed to pull image "XXX": rpc error: code = Unknown desc = context canceled
```

登录节点使用docker pull命令手动下拉镜像，镜像下拉成功。

问题根因：

kubernetes默认的image-pull-progress-deadline是1分钟, 如果1分钟内镜像下载没有任何进度更新, 下载动作就会取消。在节点性能较差或镜像较大时，可能出现镜像无法成功下载，负载启动失败的现象。

解决方案：

- (推荐)方法一：登录节点使用docker pull命令手动下拉镜像，确认负载的镜像拉取策略imagePullPolicy为IfNotPresent（默认策略配置）。此时创建负载会使用已拉取到本地的镜像。
- 方法二：修改kubelet配置参数。

  1.15及以上集群使用如下命令:

  ```
  vi /opt/cloud/cce/kubernetes/kubelet/kubelet
  ```

  1.15以下集群使用如下命令：

  ```
  vi /var/paas/kubernetes/kubelet/kubelet
  ```

  在DAEMON_ARGS参数末尾追加配置 --image-pull-progress-deadline=30m ，30m为30分钟，可根据需求修改为合适时间。追加配置和前项配置之间由空格分开。

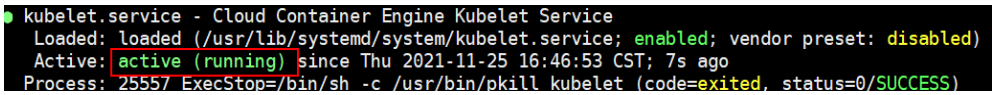
  

  重启kubelet:

  ```
  systemctl restart kubelet
  ```

等待片刻，确定kubelet状态为running

```
systemctl status kubelet
```



负载正常启动，镜像下拉成功。

排查项七：无法连接镜像仓库

问题现象

创建工作负载时报如下错误。

```
Failed to pull image "docker.io/bitnami/nginx:1.22.0-debian-11-r3": rpc error: code = Unknown desc = Error response from daemon: Get https://registry-1.docker.io/v2/: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)
```

问题原因

无法连接镜像仓库，网络不通。SWR仅支持直接拉取Docker官方的镜像，其他仓库的镜像需要连接公网。

解决方案：

- 方案一：给需要下载镜像的节点绑定公网IP。
- 方案二：先将镜像上传到SWR，然后从SWR拉取镜像。

排查项八：拉取公共镜像达上限

问题现象

创建工作负载时报如下错误。

```
ERROR: toomanyrequests: Too Many Requests.
```

或

```
you have reached your pull rate limit, you may increase the limit by authenticating an upgrading: https://www.docker.com/increase-rate-limits.
```

问题原因

DockerHub对用户拉取容器镜像请求设定了上限，详情请参见**Understanding Docker Hub Rate Limiting**。

解决方案：

将常用的镜像上传到SWR，然后从SWR拉取镜像。

20.6.1.4 工作负载异常：启动容器失败

问题定位

工作负载详情中，若事件中提示“启动容器失败”，请按照如下方式来初步排查原因：

步骤1 登录异常工作负载所在的节点。

步骤2 查看工作负载实例非正常退出的容器ID。

```
docker ps -a | grep $podName
```

步骤3 查看退出容器的错误日志。

```
docker logs $containerID
```

根据日志提示修复工作负载本身的问题。

步骤4 查看操作系统的错误日志。

```
cat /var/log/messages | grep $containerID | grep oom
```

根据日志判断是否触发了系统OOM。

----结束

排查思路

根据具体事件信息确定具体问题原因，如表20-6所示。

表 20-6 容器启动失败

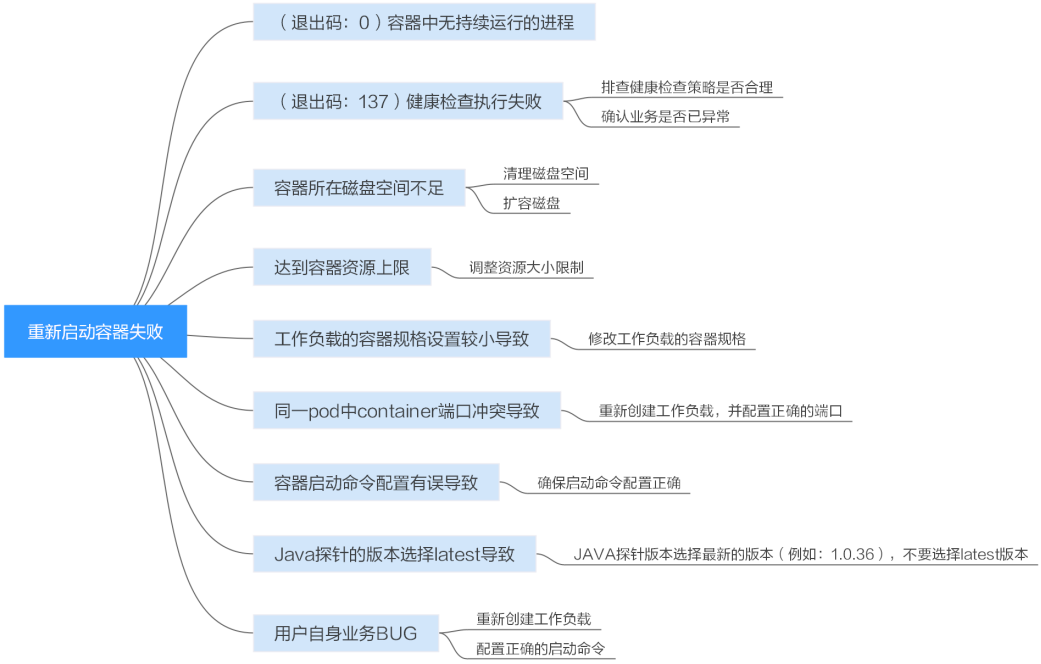
日志或事件信息	问题原因与解决方案
日志中存在exit(0)	容器中无进程。 请调试容器是否能正常运行。 **排查项一：（退出码：0）容器中无持续运行的进程**
事件信息：Liveness probe failed: Get http… 日志中存在exit(137)	健康检查执行失败。 **排查项二：（退出码：137）健康检查执行失败**
事件信息： Thin Pool has 15991 free data blocks which is less than minimum required 16383 free data blocks. Create more free space in thin pool or use dm.min_free_space option to change behavior	磁盘空间不足，需要清理磁盘空间。 **排查项三：容器所在磁盘空间不足**
日志中存在OOM字眼	内存不足。 **排查项四：达到容器资源上限** **排查项五：工作负载的容器规格设置较小导致**
Address already in use	Pod中容器端口冲突 **排查项六：同一pod中container端口冲突导致**

除上述可能原因外，还可能存在如下原因，请根据顺序排查。

- **排查项七：容器启动命令配置有误导致**

- **排查项八：用户自身业务BUG**
- 在ARM架构的节点上创建工作负载时未使用正确的镜像版本，使用正确的镜像版本即可解决该问题。

图 20-3 重新启动容器失败排查思路



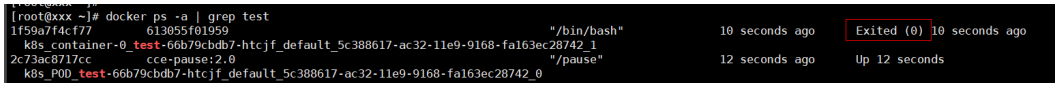
排查项一：（退出码：0）容器中无持续运行的进程

步骤1 登录异常工作负载所在的节点。

步骤2 查看容器状态。

```
docker ps -a | grep $podName
```

如下图所示：



当容器中无持续运行的进程时，会出现exit(0)的状态码，此时说明容器中无进程。

----结束

排查项二：（退出码：137）健康检查执行失败

工作负载配置的健康检查会定时检查业务，异常情况下pod会报实例不健康的事件且pod一直重启失败。

工作负载若配置liveness型（工作负载存活探针）健康检查，当健康检查失败次数超过阈值时，会重启实例中的容器。在工作负载详情页面查看事件，若K8s事件中出现“Liveness probe failed: Get http…”时，表示健康检查失败。

解决方案：

请在工作负载详情页中，切换至“容器管理”页签，核查容器的“健康检查”配置信息，排查健康检查策略是否合理或业务是否已异常。

排查项三：容器所在磁盘空间不足

如下磁盘为创建节点时选择的docker专用盘分出来的thinpool盘，以root用户执行**lvs**命令可以查看当前磁盘的使用量。

```
Thin Pool has 15991 free data blocks which is less than minimum required 16383 free data blocks. Create more free space in thin pool or use dm.min_free_space option to change behavior
```

解决方案：

方案一：清理镜像

您可以执行以下步骤清理未使用的镜像：

- 使用containerd容器引擎的节点：
  a. 查看节点上的本地镜像。
     ```
     crictl images -v
     ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
     ```
     crictl rmi {镜像ID}
     ```
- 使用docker容器引擎的节点：
  a. 查看节点上的本地镜像。
     ```
     docker images
     ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
     ```
     docker rmi {镜像ID}
     ```

说明

请勿删除cce-pause等系统镜像，否则可能导致无法正常创建容器。

方案二：扩容磁盘

扩容磁盘的操作步骤如下：

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。
  ```
  # lsblk
  NAME                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
  sda                   8:0    0   50G  0 disk
  └─sda1                8:1    0   50G  0 part /
  sdb                   8:16   0  200G  0 disk
  ├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker                # 容器引擎使用的空间
  └─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
  ```
  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。
  ```
  pvresize /dev/sdb
  lvextend -l+100%FREE -n vgpaas/dockersys
  resize2fs /dev/vgpaas/dockersys
  ```

- Devicemapper，单独划分了thinpool存储镜像相关数据。

```
# lsblk
NAME                          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                           8:0    0   50G  0 disk
└─sda1                          8:1    0   50G  0 part /
sdb                           8:16   0  200G  0 disk
├─vgpaas-dockersys              253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta           253:1    0    3G  0 lvm
│ └─vgpaas-thinpool             253:3    0   67G  0 lvm                # thinpool空间
│   ...
├─vgpaas-thinpool_tdata           253:2    0   67G  0 lvm
│ └─vgpaas-thinpool             253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes             253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到thinpool盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/thinpool
```

  - 在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----结束

排查项四：达到容器资源上限

事件详情中有OOM字样。并且，在日志中也会有记录：

```
cat /var/log/messages | grep 96feb0a425d6 | grep oom
```

创建工作负载时，设置的限制资源若小于实际所需资源，会触发系统OOM，并导致容器异常退出。

排查项五：工作负载的容器规格设置较小导致

工作负载的容器规格设置较小导致，若创建工作负载时，设置的限制资源少于实际所需资源，会导致启动容器失败。

排查项六：同一 pod 中 container 端口冲突导致

步骤1 登录异常工作负载所在的节点。

步骤2 查看工作负载实例非正常退出的容器ID。

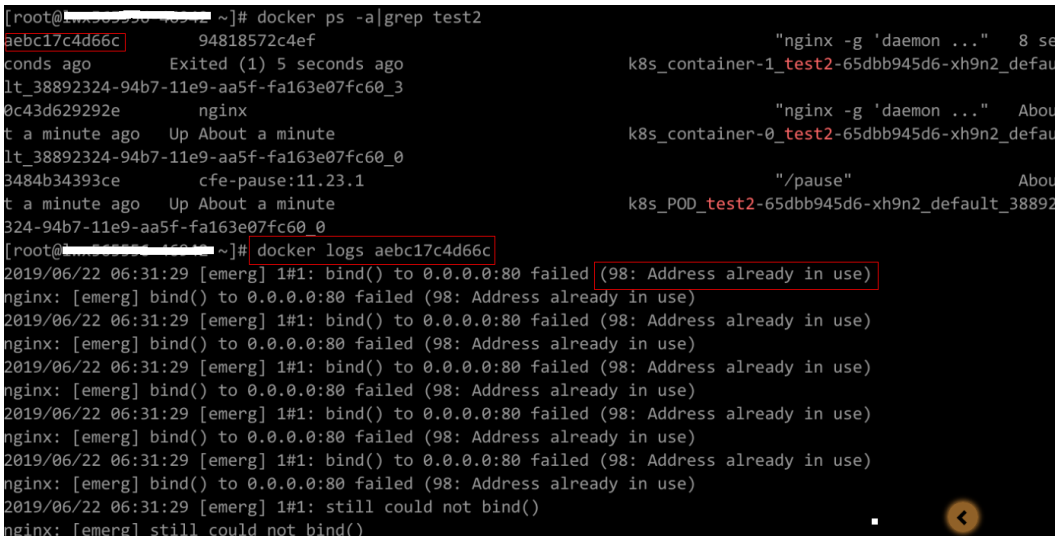
docker ps -a | grep *$podName*

步骤3 查看退出容器的错误日志。

docker logs *$containerID*

根据日志提示修复工作负载本身的问题。如下图所示，即同一Pod中的container端口冲突导致容器启动失败。

图 20-4 container 冲突导致容器启动失败



----结束

解决方案：

重新创建工作负载，并配置正确的端口，确保端口不冲突。

排查项七：容器启动命令配置有误导致

错误信息如下图所示：

解决方案：

请在工作负载详情页中，切换至“容器管理”页签，核查容器的“生命周期 > 启动命令”配置信息，确保启动命令配置正确。

排查项八：用户自身业务 BUG

请检查工作负载启动命令是否正确执行，或工作负载本身bug导致容器不断重启。

步骤1 登录异常工作负载所在的节点。

步骤2 查看工作负载实例非正常退出的容器ID。

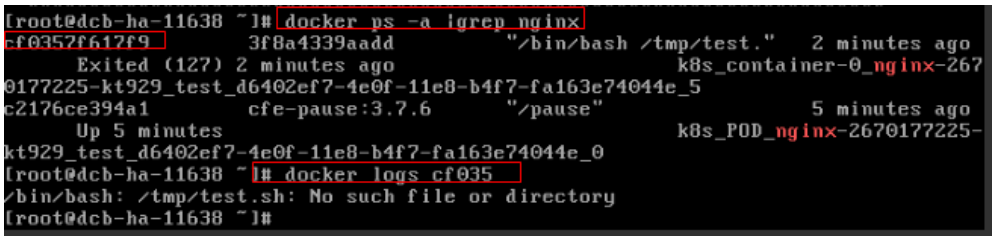
```
docker ps -a | grep $podName
```

步骤3 查看退出容器的错误日志。

```
docker logs $containerID
```

注意：这里的containerID为已退出的容器的ID

图 20-5 容器启动命令配置不正确



如上图所示，容器配置的启动命令不正确导致容器启动失败。其他错误请根据日志提示修复工作负载本身的BUG问题。

----结束

解决方案：

重新创建工作负载，并配置正确的启动命令。

20.6.1.5 工作负载异常：实例驱逐异常（Evicted）

Eviction 介绍

Eviction，即驱逐的意思，意思是当节点出现异常时，为了保证工作负载的可用性，kubernetes将有相应的机制驱逐该节点上的Pod。

目前kubernetes中存在两种eviction机制，分别由**kube-controller-manager**和**kubelet**实现。

- **kube-controller-manager实现的eviction**

  kube-controller-manager主要由多个控制器构成，而eviction的功能主要由node controller这个控制器实现。该Eviction会周期性检查所有节点状态，当节点处于NotReady状态超过一段时间后，驱逐该节点上所有pod。

  kube-controller-manager提供了以下启动参数控制eviction：

  - **pod-eviction-timeout**：即当节点宕机该时间间隔后，开始eviction机制，驱赶宕机节点上的Pod，默认为5min。
  - **node-eviction-rate**：驱赶速率，即驱赶Node的速率，由令牌桶流控算法实现，默认为0.1，即每秒驱赶0.1个节点，注意这里不是驱赶Pod的速率，而是驱赶节点的速率。相当于每隔10s，清空一个节点。
  - **secondary-node-eviction-rate**：二级驱赶速率，当集群中宕机节点过多时，相应的驱赶速率也降低，默认为0.01。
  - **unhealthy-zone-threshold**：不健康zone阈值，会影响什么时候开启二级驱赶速率，默认为0.55，即当该zone中节点宕机数目超过55%，而认为该zone不健康。
  - **large-cluster-size-threshold**：大集群阈值，当该zone的节点多于该阈值时，则认为该zone是一个大集群。大集群节点宕机数目超过55%时，则将驱赶速率降为0.01，假如是小集群，则将速率直接降为0。

- **kubelet的eviction机制**

  如果节点处于资源压力，那么kubelet就会执行驱逐策略。驱逐会考虑Pod的优先级，资源使用和资源申请。当优先级相同时，资源使用/资源申请最大的Pod会被首先驱逐。

kube-controller-manager的eviction机制是粗粒度的，即驱赶一个节点上的所有pod，而kubelet则是细粒度的，它驱赶的是节点上的某些Pod，驱赶哪些Pod与Pod的Qos机制有关。该Eviction会周期性检查本节点内存、磁盘等资源，当资源不足时，按照优先级驱逐部分pod。

驱逐阈值分为软驱逐阈值（Soft Eviction Thresholds）和强制驱逐（Hard Eviction Thresholds）两种机制，如下：

- **软驱逐阈值：**当node的内存/磁盘空间达到一定的阈值后，kubelet不会马上回收资源，如果改善到低于阈值就不进行驱逐，若这段时间一直高于阈值就进行驱逐。
- **强制驱逐：**强制驱逐机制则简单的多，一旦达到阈值，直接把pod从本地驱逐。

kubelet提供了以下参数控制eviction：

- **eviction-soft：**软驱逐阈值设置，具有一系列阈值，比如memory.available<1.5Gi时，它不会立即执行pod eviction，而会等待eviction-soft-grace-period时间，假如该时间过后，依然还是达到了eviction-soft，则触发一次pod eviction。
- **eviction-soft-grace-period：**默认为90秒，当eviction-soft时，终止Pod的grace的时间，即软驱逐宽限期，软驱逐信号与驱逐处理之间的时间差。
- **eviction-max-pod-grace-period：**最大驱逐pod宽限期，停止信号与kill之间的时间差。
- **eviction-pressure-transition-period：**默认为5分钟，驱逐压力过渡时间，超过阈值时，节点会被设置为memory pressure或者disk pressure，然后开启pod eviction。
- **eviction-minimum-reclaim：**表示每一次eviction必须至少回收多少资源。
- **eviction-hard：**强制驱逐设置，也具有一系列的阈值，比如memory.available<1Gi，即当节点可用内存低于1Gi时，会立即触发一次pod eviction。

问题定位

若节点故障时，实例未被驱逐，请先按照如下方法进行问题定位。

使用如下命令发现很多pod的状态为Evicted：

```
kubectl get pods
```

在节点的kubelet日志中会记录Evicted相关内容，搜索方法可参考如下命令：

```
cat /var/paas/sys/log/kubernetes/kubelet.log | grep -i Evicted -C3
```

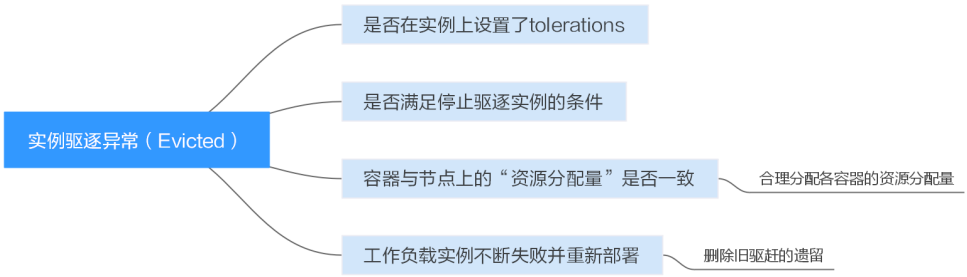
排查思路

以下排查思路根据原因的出现概率进行排序，建议您从高频率原因往低频率原因排查，从而帮助您快速找到问题的原因。

如果解决完某个可能原因仍未解决问题，请继续排查其他可能原因。

- **排查项一：是否在实例上设置了tolerations**
- **排查项二：是否满足停止驱逐实例的条件**
- **排查项三：容器与节点上的“资源分配量”是否一致**
- **排查项四：工作负载实例不断失败并重新部署**

图 20-6 实例驱逐异常排查思路



排查项一：是否在实例上设置了 tolerations

通过kubectl工具或单击对应工作负载后的“更多 > 编辑YAML”，检查工作负载上是不是设置了容忍度，具体请参见**污点和容忍度**。

排查项二：是否满足停止驱逐实例的条件

若属于小规格的集群（集群节点数小于50个节点），如果故障的节点大于总节点数的55%，实例的驱逐将会被暂停。此情况下Kubernetes将不再尝试驱逐故障节点的工作负载，具体请参见**节点驱逐速率限制**。

排查项三：容器与节点上的“资源分配量”是否一致

容器被驱逐后还会频繁调度到原节点。

问题原因：

节点驱逐容器是根据节点的“资源使用率”进行判断；容器的调度规则是根据节点上的“资源分配量”进行判断。由于判断标准不同，所以可能会出现被驱逐后又再次被调度到原节点的情况。

解决方案：

遇到此类问题时，请合理分配各容器的资源分配量即可解决。

排查项四：工作负载实例不断失败并重新部署

工作负载实例出现不断失败，不断重新部署的情况。

问题分析：

pod驱逐后，如果新调度到的节点也有驱逐情况，就会再次被驱逐；甚至出现pod不断被驱逐的情况。

如果是由kube-controller-manager触发的驱逐，会留下一个状态为Terminating的pod；直到容器所在节点状态恢复后，pod才会自动删除。如果节点已经删除或者其他原因导致的无法恢复，可以使用“强制删除”删除pod。

如果是由kubelet触发的驱逐，会留下一个状态为Evicted的pod，此pod只是方便后期定位的记录，可以直接删除。

解决方案：

使用如下命令删除旧驱赶的遗留：

```
kubectl get pods <namespace> | grep Evicted | awk '{print $1}' | xargs kubectl delete pod <namespace>
```

*<namespace>*为命名空间名称，请根据需要指定。

参考

Kubelet does not delete evicted pods

20.6.1.6 工作负载异常：存储卷无法挂载或挂载超时

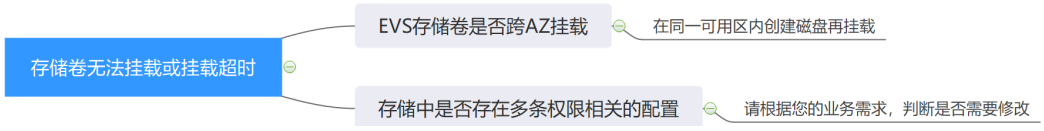
排查思路

以下排查思路根据原因的出现概率进行排序，建议您从高频率原因往低频率原因排查，从而帮助您快速找到问题的原因。

如果解决完某个可能原因仍未解决问题，请继续排查其他可能原因。

- **排查项一：EVS存储卷是否跨AZ挂载**
- **排查项二：存储中是否同时存在多条权限相关的配置**
- **排查项三：带云硬盘卷的Deployment的副本数大于1**
- **排查项四：EVS磁盘文件系统损坏**

图 20-7 存储卷无法挂载或挂载超时排查思路



排查项一：EVS 存储卷是否跨 AZ 挂载

问题描述：

客户在有状态工作负载上挂载EVS存储卷，但无法挂载卷并超时。

问题定位：

经查询确认，该节点在**可用区1**，而要挂载的磁盘在**可用区2**，导致无法挂载而超时。

解决方案：

在同一可用区内创建磁盘再挂载后即可正常。

排查项二：存储中是否同时存在多条权限相关的配置

如果您挂载的存储中内容太多，同时又配置了以下几条配置，最终会由于逐个修改文件权限，而导致挂载时间过长。

问题定位：

- Securitycontext字段中是否包含runAsuser/fsGroup。securityContext是kubernetes中的字段，即安全上下文，它用于定义Pod或Container的权限和访问控制设置。
- 启动命令中是否包含ls、chmod、chown等查询或修改文件权限的操作。

解决建议：

请根据您的业务需求，判断是否需要修改。

排查项三：带云硬盘卷的 Deployment 的副本数大于 1

问题描述：

创建Pod失败，并报“添加存储失败”的事件，事件信息如下。

```
Multi-Attach error for volume "pvc-62a7a7d9-9dc8-42a2-8366-0f5ef9db5b60" Volume is already used by pod(s) testttt-7b774658cb-lc98h
```

问题定位：

查看Deployment的副本数是否大于1。

Deployment中使用EVS存储卷时，副本数只能为1。若用户在后台指定Deployment的实例数为2以上，此时CCE并不会限制Deployment的创建。但若这些实例Pod被调度到不同的节点，则会有部分Pod因为其要使用的EVS无法被挂载到节点，导致Pod无法启动成功。

解决方案：

使用EVS的Deployment的副本数指定为1，或使用其他类型存储卷。

排查项四：EVS 磁盘文件系统损坏

问题描述：

创建Pod失败，出现类似信息，磁盘文件系统损坏。

```
MountVolume.MountDevice failed for volume "pvc-08178474-c58c-4820-a828-14437d46ba6f" : rpc error: code = Internal desc = [09060def-afd0-11ec-9664-fa163eef47d0] /dev/sda has file system, but it is detected to be damaged
```

解决方案：

在EVS中对磁盘进行备份，然后执行如下命令修复文件系统。

fsck -y {盘符}

20.6.1.7 工作负载异常：一直处于创建中

问题描述

节点上的工作负载一直处于创建中。

排查思路

以下排查思路根据原因的出现概率进行排序，建议您从高频率原因往低频率原因排查，从而帮助您快速找到问题的原因。

如果解决完某个可能原因仍未解决问题，请继续排查其他可能原因。

- **排查项一：cce-pause镜像是否被误删除**
- **排查项二：集群开启CPU管理策略后变更节点规格**

排查项一：cce-pause 镜像是否被误删除

问题现象

创建工作负载时报如下错误，显示无法创建sandbox，原因是拉取cce-pause:3.1镜像失败。

```
Failed to create pod sandbox: rpc error: code = Unknown desc = failed to get sandbox image "cce-pause:3.1": failed to pull image "cce-pause:3.1": failed to pull and unpack image "docker.io/library/cce-pause:3.1": failed to resolve reference "docker.io/library/cce-pause:3.1": pulling from host **** failed with status code [manifests 3.1]: 400 Bad Request
```

问题原因

该镜像为创建节点时添加的系统镜像，如果手动误删除该镜像，会导致工作负载Pod一直无法创建。

解决方案：

步骤1 登录该问题节点。

步骤2 手动解压节点上的cce-pause镜像安装包。

```
tar -xzvf /opt/cloud/cce/package/node-package/pause-*.tgz
```

步骤3 导入镜像。

- Docker节点：
  ```
  docker load ./pause/package/image/cce-pause-3.1.tar
  ```
- Containerd节点：
  ```
  ctr -n k8s.io image import ./pause/package/image/cce-pause-3.1.tar
  ```

步骤4 镜像导入成功后，即可正常创建工作负载。

----结束

排查项二：集群开启 CPU 管理策略后变更节点规格

集群开启CPU管理策略（绑核）时，kubelet启动参数中会将CPU Manager的策略设置为static，允许为节点上具有某些资源特征的pod赋予增强的CPU亲和性和独占性。用户如果直接在ECS控制台对CCE节点变更规格，会由于变更前后CPU信息不匹配，导致节点上的负载无法重新拉起，也无法创建新负载。

步骤1 登录CCE节点（弹性云服务器）并删除cpu_manager_state文件。

删除命令示例如下：

```
rm -rf /mnt/paas/kubernetes/kubelet/cpu_manager_state
```

步骤2 重启节点或重启kubelet，重启kubelet的方法如下：

```
systemctl restart kubelet
```

此时重新拉起或创建工作负载，已可成功执行。

解决方式链接：**CCE节点变更规格后，为什么无法重新拉起或创建工作负载?**

----结束

20.6.1.8 工作负载异常：结束中，解决 Terminating 状态的 Pod 删不掉的问题

问题描述

在节点处于“不可用”状态时，CCE会迁移节点上的容器实例，并将节点上运行的pod置为“Terminating”状态。

待节点恢复后，处于“Terminating”状态的pod会自动删除。

偶现部分pod（实例）一直处于“Terminating ”状态：

```
#kubectl get pod -n aos
NAME                          READY     STATUS        RESTARTS   AGE
aos-apiserver-5f8f5b5585-s9l92    1/1       Terminating    0          3d1h
aos-cmdbserver-789bf5b497-6rwrg   1/1       Running        0          3d1h
aos-controller-545d78bs8d-vm6j9   1/1       Running        3          3d1h
```

通过kubectl delete pods <podname> -n <namespace> 命令始终无法将其删除：

```
kubectl delete pods aos-apiserver-5f8f5b5585-s9l92 -n aos
```

解决方法

无论各种方式生成的pod，均可以使用如下命令强制删除：

```
kubectl delete pods <pod> --grace-period=0 --force
```

因此对于上面的pod，只要执行如下命令即可删除：

```
kubectl delete pods aos-apiserver-5f8f5b5585-s9l92 --grace-period=0 --force
```

20.6.1.9 工作负载异常：已停止

问题现象

工作负载的状态为“已停止”。

问题原因：

工作负载的yaml的中metadata.enable字段为false，导致工作负载被停止，Pod被删除导致工作负载处于已停止状态，如下图所示：

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: test
  namespace: default
  selfLink: /apis/apps/v1/namespaces/default/deployments/test
  uid: b130db9f-9306-11e9-a2a9-fa163eaff9f7
  resourceVersion: '7314771'
  generation: 1
  creationTimestamp: '2019-06-20T02:54:16Z'
  labels:
    appgroup: ''
  annotations:
    deployment.kubernetes.io/revision: '1'
    description: ''
  enable: false
spec:
```

解决方案

将enable字段删除或者将false修改为true。

20.6.1.10 工作负载异常：GPU 节点部署服务报错

问题现象

客户在CCE集群的GPU节点上部署服务出现如下问题：

1. 容器无法查看显存。
2. 部署了7个GPU服务，有2个是能正常访问的，其他启动时都有报错。
   - 2个是能正常访问的CUDA版本分别是10.1和10.0
   - 其他服务CUDA版本也在这2个范围内
3. 在GPU服务容器中发现一些新增的文件core.*，在以前的部署中没有出现过。

问题定位

1. GPU插件的驱动版本较低，客户单独下载驱动安装后正常。
2. 客户工作负载中未声明需要gpu资源。

建议方案

节点安装了gpu-beta（gpu-device-plugin）插件后，会自动安装nvidia-smi命令行工具。引起部署GPU服务报错通常是由于nvidia驱动安装失败，请排查nvidia驱动是否下载成功。

- GPU节点：
```
# 插件版本为2.0.0以下时，执行以下命令：
cd /opt/cloud/cce/nvidia/bin && ./nvidia-smi

# 插件版本为2.0.0及以上时，驱动安装路径更改，需执行以下命令：
cd /usr/local/nvidia/bin && ./nvidia-smi
```
- 容器：
```
cd /usr/local/nvidia/bin && ./nvidia-smi
```

若能正常返回GPU信息，说明设备可用，插件安装成功。

如果驱动地址填写错误，需要将插件卸载后重新安装，并配置正确的地址。

说明

nvidia驱动建议放在OBS桶里，并设置为公共读。

相关链接

- **GPU节点使用nvidia驱动启动容器排查思路**

20.6.1.11 实例网络空间更新，报 sandbox 相关错，如何处理?

问题现象

实例一直处于创建中，报错sandbox相关错，如下所示。

```
Failed create pod sandbox: rpc error: code = Unknown desc = [ failed to setup sandbox .....
```

解决方案

排查方法如下：

1.13版本的集群：

说明

此方法仅支持1.13版本集群。

1. sandbox错误一般为节点上容器组件启动有异常，systemctl status canal查看节点的容器组件，重启组件命令如：systemctl restart canal。
2. 节点上容器组件全部正常，network: failed to find plugin "loopback" in path [/opt/cni/bin]，cni的文件loopback缺失导致，可以从其他局点或者当前局点（建议和集群版本配套，可忽略小版本）拷贝一份完整的loopback文件，放入 /opt/cni/bin/下，然后重启canal组件。

1.13版本之前的集群：

1. sandbox错误一般为节点上容器组件启动有异常，su paas -c '/var/paas/monit/bin/monit summary' 查看节点的容器组件，重启组件命令如：su paas -c '/var/paas/monit/bin/monit restart canal'。
2. 节点上容器组件全部正常，network: failed to find plugin "loopback" in path [/opt/cni/bin]，cni的文件loopback缺失导致，可以从其他局点或者当前局点（建议和集群版本配套，可忽略小版本）拷贝一份完整的loopback文件，放入 /opt/cni/bin/下，然后重启canal组件。

20.6.2 容器设置

20.6.2.1 在什么场景下设置工作负载生命周期中的“停止前处理”？

服务的业务处理时间较长，在升级时，需要先等Pod中的业务处理完，才能kill该Pod，以保证业务不中断的场景。

20.6.2.2 在同一个命名空间内访问指定容器的 FQDN 是什么?

问题背景

客户询问在创建负载时指定部署的容器名称、pod名称、namespace名称，在同一个命名空间内访问该容器的FQDN是什么?

全限定域名：FQDN，即Fully Qualified Domain Name，同时带有主机名和域名的名称。（通过符号“.”）

例如：主机名是bigserver，域名是mycompany.com，那么FQDN就是：bigserver.mycompany.com。

问题建议

方案一：发布服务使用域名发现，需要提前预制好主机名和命名空间，服务发现使用域名的方式，注册的服务的域名为：服务名.命名空间.svc.cluster.local 。这种使用有限制，注册中心部署必须容器化部署。

方案二：容器部署使用主机网络部署，然后亲和到集群的某一个节点，这样可以明确知道容器的服务地址（就是节点的地址），注册的地址为：服务所在节点IP，这种方案可以满足注册中心利用VM部署，缺陷是使用主机网络效率没有容器网络高。

20.6.2.3 健康检查探针（Liveness、Readiness）偶现检查失败?

健康检查探针偶现检测失败，是由于容器内的业务故障所导致，您需要优先定位自身业务问题。

常见情况有：

- 业务处理时间长，导致返回超时。
- tomcat建链和等到耗费时间太长（连接数、线程数等），导致返回超时。
- 容器所在节点，磁盘IO等性能达到瓶颈，导致业务处理超时。

20.6.2.4 如何设置容器 umask 值?

问题描述

tailf /dev/null的方式启动容器，然后手动执行启动脚本的方式得到的目录的权限是700，而不加tailf由Kubernetes自行启动的方式得到的目录权限却是751。

解决方案

这个问题是因为两种方式设置的umask值不一样，所以创建出来的目录权限不相同。

umask值用于为用户新创建的文件和目录设置缺省权限。如果umask的值设置过小，会使群组用户或其他用户的权限过大，给系统带来安全威胁。因此设置所有用户默认的umask值为0077，即用户创建的目录默认权限为700，文件的默认权限为600。

可以在启动脚本里面增加如下内容实现创建出来的目录权限为700：

1. 分别在**/etc/bashrc**文件和**/etc/profile.d/**目录下的所有文件中加入“umask 0077”。
2. 执行如下命令：
   ```
   echo "umask 0077" >> $FILE
   ```

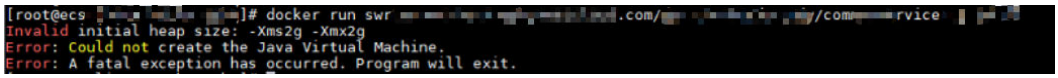
   说明

   FILE为具体的文件名，例如：echo “umask 0077” >> /etc/bashrc
3. 设置**/etc/bashrc**文件和**/etc/profile.d/**目录下所有文件的属主为：root，群组为：root。
4. 执行如下命令：
   ```
   chown root.root $FILE
   ```

20.6.2.5 Dockerfile 中 ENTRYPOINT 指定 JVM 启动堆内存参数后部署容器启动报错?

问题描述

Dockerfile中ENTRYPOINT指定JVM启动堆内存参数后部署容器启动报错，报错信息为：invalid initial heap size，如下图：



解答

请检查ENTRYPOINT设置，下方的设置是错误的：

```
ENTRYPOINT ["java","-Xms2g -Xmx2g","-jar","xxx.jar"]
```

如下两种办法可以解决该问题：

- （**推荐**）将容器启动命令写在“工作负载 > 更新升级 > 容器设置 > 生命周期 > 启动命令”这里，容器能正常启动。
- 将ENTRYPOINT启动命令修改为如下格式：
  ```
  ENTRYPOINT exec java -Xmx2g -Xms2g -jar xxxx.jar
  ```

20.6.2.6 CCE 启动实例失败时的重试机制是怎样的?

CCE是基于原生Kubernetes的云容器引擎服务，完全兼容Kubernetes社区原生版本，与社区最新版本保持紧密同步，完全兼容Kubernetes API和Kubectl。

在Kubernetes中，Pod的spec中包含一个restartPolicy字段，其取值包括：Always、OnFailure和Never，默认值为：Always。

- Always：当容器失效时，由kubelet自动重启该容器。
- OnFailure：当容器终止运行且退出不为0时（正常退出），由kubelet自动重启该容器。
- Never：不论容器运行状态如何，kubelet都不会重启该容器。

restartPolicy适用于Pod中的所有容器。

restartPolicy仅针对同一节点上kubelet的容器重启动作。当Pod中的容器退出时，kubelet 会按指数回退方式计算重启的延迟（10s、20s、40s...），其最长延迟为5分钟。 一旦某容器执行了10分钟并且没有出现问题，kubelet对该容器的重启回退计时器执行重置操作。

每种控制器对Pod的重启策略要求如下：

- Replication Controller（RC）和DaemonSet：必须设置为Always，需要保证该容器的持续运行。
- Job：OnFailure或Never，确保容器执行完成后不再重启。

20.6.3 调度策略

20.6.3.1 如何让多个 Pod 均匀部署到各个节点上?

Kubernetes中kube-scheduler组件负责Pod的调度，对每一个新创建的 Pod 或者是未被调度的 Pod，kube-scheduler 会选择一个最优的节点去运行这个 Pod。kube-scheduler 给一个 Pod 做调度选择包含过滤和打分两个步骤。过滤阶段会将所有满足 Pod 调度需求的节点选出来，在打分阶段 kube-scheduler 会给每一个可调度节点进行优先级打分，最后kube-scheduler 会将 Pod 调度到得分最高的节点上，如果存在多个得分最高的节点，kube-scheduler 会从中随机选取一个。

打分优先级中节点调度均衡（BalancedResourceAllocation）只是其中一项，还有其他打分项会导致分布不均匀。详细的调度说明请参见**Kubernetes 调度器**和**调度策略**。

想要让多个Pod尽可能的均匀分布在各个节点上，可以考虑使用工作负载反亲和特性，让Pod之间尽量“互斥”，这样就能尽量均匀的分布在各节点上。

示例如下：

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      affinity:
        podAntiAffinity:                  # 工作负载反亲和
          preferredDuringSchedulingIgnoredDuringExecution:    # 尽量满足如下条件
            - podAffinityTerm:
                labelSelector:                      # 选择Pod的标签，与工作负载本身反亲和
                  matchExpressions:
                    - key: app
                      operator: In
                      values:
                        - nginx
                namespaces:
                  - default
                topologyKey: kubernetes.io/hostname     # 在节点上起作用
      imagePullSecrets:
        - name: default-secret
```

20.6.3.2 如何避免节点上的某个容器被驱逐?

问题背景

在工作负载调度时可能会发生一个节点上的两个容器之间互相争资源的情况，最终导致kubelet将其全部驱逐。那么能不能设定策略让其中一个服务一直保留？如何设定?

问题建议

Kubelet会按照下面的标准对Pod的驱逐行为进行评判：

- 根据服务质量：即**BestEffort**、**Burstable**、**Guaranteed**。
- 根据Pod调度请求的被耗尽资源的消耗量。

接下来，Pod按照下面的顺序进行驱逐（QOS）：

BestEffort -> Burstable -> Guaranteed

- BestEffort类型的Pod：系统用完了全部内存时，该类型Pod会最先被终止。
- Burstable类型的Pod：系统用完了全部内存，且没有BestEffort容器可以终止时，该类型Pod会被终止。

- Guaranteed类型的Pod：系统用完了全部内存、且没有Burstable与BestEffort容器可以终止时，该类型的Pod会被终止。

说明

- 如果Pod进程因使用超过预先设定的限制值而非Node资源紧张情况，系统倾向于在其原所在的机器上重启该容器或本机或其他重新创建一个Pod。
- 如果资源充足，可将QoS Pod类型均设置为Guaranteed。用计算资源换业务性能和稳定性，减少排查问题时间和成本。
- 如果想更好的提高资源利用率，业务服务可以设置为Guaranteed，而其他服务根据重要程度可分别设置为Burstable或BestEffort，例如filebeat。

20.6.3.3 为什么 Pod 在节点不是均匀分布?

Kubernetes中kube-scheduler组件负责Pod的调度，对每一个新创建的 Pod 或者是未被调度的Pod，kube-scheduler 会选择一个最优的节点去运行这个 Pod。kube-scheduler 给一个 Pod 做调度选择包含过滤和打分两个步骤。过滤阶段会将所有满足 Pod 调度需求的节点选出来，在打分阶段 kube-scheduler 会给每一个可调度节点进行优先级打分，最后kube-scheduler 会将 Pod 调度到得分最高的节点上，如果存在多个得分最高的节点，kube-scheduler 会从中随机选取一个。

打分优先级中节点调度均衡（BalancedResourceAllocation）只是其中一项，还有其他打分项会导致分布不均匀。详细调度说明请参见**Kubernetes 调度器**和**调度策略**。

20.6.3.4 如何驱逐节点上的所有 Pod?

您可使用**kubectl drain**命令从节点安全地逐出所有Pod。

说明

默认情况下，**kubectl drain**命令会保留某些系统级Pod不被驱逐，例如everest-csi-driver。

步骤1 使用kubectl连接集群。

步骤2 查看集群中的节点。

```
kubectl get node
```

步骤3 选择一个节点，查看节点上存在的所有Pod。

```
kubectl get pod --all-namespaces -owide --field-selector spec.nodeName=192.168.0.160
```

驱逐前该节点上的Pod如下：

```
NAMESPACE     NAME                                  READY   STATUS    RESTARTS   AGE     IP
NODE          NOMINATED NODE   READINESS GATES
default       nginx-5bcc57c74b-lgcvh                1/1     Running   0          7m25s   10.0.0.140
192.168.0.160   <none>           <none>
kube-system   coredns-6fcd88c4c-97p6s               1/1     Running   0          3h16m   10.0.0.138
192.168.0.160   <none>           <none>
kube-system   everest-csi-controller-56796f47cc-99dtm   1/1     Running   0          3h16m   10.0.0.139
192.168.0.160   <none>           <none>
kube-system   everest-csi-driver-dpfzl              2/2     Running   2          12d     192.168.0.160
192.168.0.160   <none>           <none>
kube-system   icagent-tpfpv                         1/1     Running   1          12d     192.168.0.160
192.168.0.160   <none>           <none>
```

步骤4 驱逐该节点上的所有Pod。

```
kubectl drain 192.168.0.160
```

如果节点上存在绑定了本地存储的Pod或是一些守护进程集管理的Pod，将提示“error: unable to drain node "192.168.0.160", aborting command...”。驱逐命令将不会生效，您可在上述命令后面添加如下参数进行强制驱逐：

- --delete-emptydir-data：强制驱逐节点上绑定了本地存储的Pod，例如coredns。
- --ignore-daemonsets：忽略节点上的守护进程集Pod，例如everest-csi-driver。

示例中节点上存在绑定本地存储的Pod和守护进程集Pod，因此驱逐命令如下：

```
kubectl drain 192.168.0.160 --delete-emptydir-data --ignore-daemonsets
```

步骤5 驱逐成功后，该节点被自动标记为不可调度，即该节点将会被打上**node.kubernetes.io/unschedulable = : NoSchedule**的污点。

驱逐后该节点上的Pod如下，节点上仅保留了不可驱逐的系统级Pod。

```
NAMESPACE     NAME                       READY   STATUS    RESTARTS   AGE   IP              NODE
NOMINATED NODE   READINESS GATES
kube-system   everest-csi-driver-dpfzl   2/2     Running   2          12d   192.168.0.160   192.168.0.160
<none>           <none>
kube-system   icagent-tpfpv              1/1     Running   1          12d   192.168.0.160   192.168.0.160
<none>           <none>
```

----结束

相关操作

kubectl的drain、cordon和uncordon操作：

- drain：从节点安全地逐出所有Pod，并将该节点标记为不可调度。
- cordon：将节点标记为不可调度，即该节点将会被打上**node.kubernetes.io/unschedulable = : NoSchedule**的污点。
- uncordon：将节点标记为可调度。

更多说明请参考**kubectl文档**。

20.6.4 其他

20.6.4.1 定时任务停止一段时间后，为何无法重新启动?

定时任务在运行过程中，如果被暂停，再次被开启时，会根据最后一次的成功时间跟当前的时间计算时间差，然后与定时的周期*100作对比，如果时间差大于单次周期时长*100，后期的定时任务就不会被触发，详情请参考**CronJob限制**。

例如，假设一个CronJob被设置为从08:30:00开始每隔1分钟创建一个新的Job，且startingDeadlineSeconds字段未被设置。如果CronJob控制器从08:29:00到10:21:00终止运行，则该Job将不会启动，因为从08:29:00到10:21:00超过了100分钟，即错过的调度次数超过了100（示例中一个调度周期为1分钟）。

但如果设置了startingDeadlineSeconds字段，则控制器会统计从startingDeadlineSeconds设置的值到现在的时间，计算期间错过了多少次Job。 例如，如果startingDeadlineSeconds是 200，则控制器会统计在过去200秒中错过了多少次Job。此时如果CronJob控制器同样在08:29:00到10:21:00时间段终止运行，则Job仍将从10:22:00开始，因为最近200秒中仅错过了3个调度（示例中一个调度周期为1分钟）。

解决方法

如果想要解决这个问题，可以在定时任务的CronJob中配置参数：startingDeadlineSeconds。该参数只能使用kubectl命令，或者通过API接口进行创建或修改。

YAML示例如下：

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  startingDeadlineSeconds: 200
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello
          restartPolicy: OnFailure
```

如果重新创建CronJob，也可以临时规避这个限制。

20.6.4.2 创建有状态负载时，实例间发现服务是指什么？

云容器引擎的实例间发现服务，在原生Kubernetes中称之为Headless Service。Headless Service也是一种Service，但是会在YAML中定义spec.clusterIP: None，也就是不需要Cluster IP的Service。

Headless Service 和普通 Service 的区别

- 普通Service：

  一个Service可能对应多个EndPoint（Pod），client访问的是Cluster IP，通过iptables或IPVS规则转到Real Server，从而达到负载均衡的效果。例如：Service有2个EndPoint，但是DNS查询时只会返回Service的地址，具体client访问的是哪个Real Server，是由iptables或IPVS规则来决定的，客户端无法自行选择访问指定的EndPoint。

- Headless Service：

  访问Headless Service时，DNS查询会如实的返回每个真实的EndPoint（Pod的IP地址）。对应到每一个EndPoints，即每一个Pod，都会有对应的DNS域名；这样Pod之间就可以互相访问，达到实例间发现和访问的效果。

Headless Service 使用场景

当某个工作负载的多个Pod之间没有任何区别时，可以使用普通Service，利用集群kube-proxy实现Service的负载均衡，例如常见的无状态应用Nginx。

但是某些应用场景下，工作负载的各个实例间存在不同的角色区别，比如Redis集群，每个Redis实例都是不同的，它们之间存在主从关系，并且需要相互通信。这种情况下，使用普通Service无法通过Cluster IP来保证访问到某个指定的实例，因此需要设置Headless Service直接访问Pod的真实IP地址，实现Pod间互相访问。

Headless Service一般结合**StatefulSet**来部署有状态的应用，比如Redis集群、MySQL集群等。

20.6.4.3 CCE 容器拉取私有镜像时报错“Auth is empty”

问题描述

在CCE的控制台界面中为已经创建的工作负载更换镜像，选择我上传的镜像，容器在拉取镜像时报错“Auth is empty, only accept X-Auth-Token or Authorization”。

```
Failed to pull image "IP地址:端口号/magicdoom/tidb-operator:latest": rpc error: code = Unknown desc = Error response from daemon: Get https://IP地址:端口号/v2/magicdoom/tidb-operator/manifests/latest: error parsing HTTP 400 response body: json: cannot unmarshal number into Go struct field Error.code of type errcode.ErrorCode: "{\"errors\":[{\"code\":400,\"message\":\"Auth is empty, only accept X-Auth-Token or Authorization.\"}]}"
```

解答

您可以通过CCE控制台界面选择私有镜像创建应用，此时CCE会自动带上该secret，升级时不会出现该问题。

您通过API创建应用时，在deployment中带入该secret也可以在升级时避免该问题。

```
imagePullSecrets:
- name: default-secret
```

20.6.4.4 为什么 Pod 调度不到某个节点上?

步骤1 请排查节点和docker是否正常，排查方法请参见**排查项七：内部组件是否正常**。

步骤2 如果节点和docker正常，而pod调度不到节点上，请确认pod是否做了亲和，排查方法请参见**排查项三：检查工作负载的亲和性配置**。

步骤3 如果节点上的资源不足，导致节点调度不上，请扩容或者新增节点。

----结束

20.6.4.5 CCE 集群中工作负载镜像的拉取策略?

容器在启动运行前，需要镜像。镜像的存储位置可能会在本地，也可能会在远程镜像仓库中。

Kubernetes配置文件中的imagePullPolicy属性是用于描述镜像的拉取策略的，如下：

- Always：总是拉取镜像。
  ```
  imagePullPolicy: Always
  ```
- IfNotPresent：本地有则使用本地镜像，不拉取。
  ```
  imagePullPolicy: IfNotPresent
  ```
- Never：只使用本地镜像，从不拉取，即使本地没有。
  ```
  imagePullPolicy: Never
  ```

说明如下：

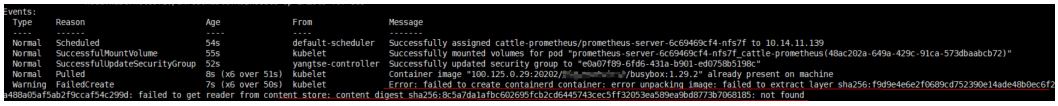
1. 如果设置为Always ，则每次容器启动或者重启时，都会从远程仓库拉取镜像。
   如果省略imagePullPolicy，策略默认为Always。
2. 如果设置为IfNotPreset，有下面两种情况：

a. 当本地不存在所需的镜像时，会从远程仓库中拉取。
b. 如果需要的镜像和本地镜像内容相同，只不过重新打了tag。此tag镜像本地不存在，而远程仓库存在此tag镜像。这种情况下，Kubernetes并不会拉取新的镜像。

20.6.4.6 下载镜像缺少层如何解决

故障现象

在使用containerd容器引擎场景下，拉取镜像到节点时，概率性缺少镜像层，导致工作负载容器创建失败。



问题根因

docker v1.10 之前支持mediaType 为 application/octet-stream 的layer，而containerd不支持application/octet-stream，导致没有拉取。

解决方法

有如下两种方式可解决该问题。

- 使用高版本Docker（ >= docker v1.11 ）重新打包镜像。
- 手动下载镜像
  a. 登录节点。
  b. 执行如下命令手动下载镜像。
     ctr -n k8s.io images pull --user u:p images
  c. 使用新下载的镜像重新创建工作负载。

20.7 网络管理

20.7.1 网络规划

20.7.1.1 集群与虚拟私有云、子网的关系是怎样的?

“虚拟私有云”类似家庭生活中路由器管理192.168.0.0/16的私有局域网，是为用户在云上构建的一个私有网络，是弹性云服务器、负载均衡、中间件等工作的基本网络环境。根据实际业务需要可以设置不同规模的网络，一般可为10.0.0.0/8~24，172.16.0.0/12~24，192.168.0.0/16~24，其中最大的网络10.0.0.0/8的A类地址网络。

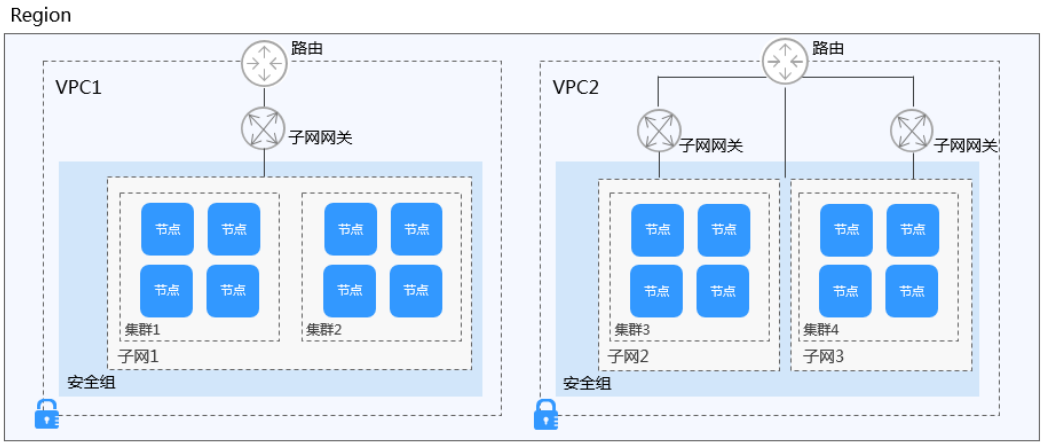
子网是虚拟私有云中的一个子集，可以将虚拟私有云划分为一个个子网，每个子网之间可以通过安全组控制其之间能否互通，保证子网之间可以相互隔离，用户可以将不同业务部署在不同的子网内。

集群是同一个VPC中一个或多个弹性云服务器或裸金属服务器（又称：节点）通过相关技术组合而成的计算机群体，为容器运行提供了计算资源池。

如图20-8，同一个region下可以有多个虚拟私有云（图中以VPC表示）。虚拟私有云由一个个子网组成，子网与子网之间的网络交互通过子网网关完成，而集群就是建立在某个子网中。因此，存在以下三种场景：

- 不同集群可以创建在不同的虚拟私有云中。
- 不同集群可以创建在同一个子网中。
- 不同集群可以创建在不同的子网中。

图 20-8 集群与 VPC、Subnet 的关系



20.7.1.2 集群安全组规则配置

CCE作为通用的容器平台，安全组规则的设置适用于通用场景。集群在创建时将会自动为Master节点和Node节点分别创建一个安全组，其中Master节点的安全组名称是：**{集群名}-cce-control-{随机ID}**；Node节点的安全组名称是：**{集群名}-cce-node-{随机ID}**。使用CCE Turbo集群时会额外创建一个ENI的安全组，名为**{集群名}-cce-eni-{随机ID}**。

用户可根据安全需求，登录CCE控制台，单击服务列表中的“网络 > 虚拟私有云 VPC”，在网络控制台单击“访问控制 > 安全组”，找到集群对应的安全组规则进行修改和加固。

不同网络模型的默认安全组规则如下：

- **VPC网络模型安全组规则**
- **容器隧道网络模型安全组规则**
- **云原生网络2.0（CCE Turbo集群）安全组规则**

须知

安全组规则的**修改和删除可能会影响集群的正常运行**，请谨慎操作。如需修改安全组规则，请尽量避免对CCE运行依赖的端口规则进行修改。

VPC 网络模型安全组规则

Node节点安全组

集群自动创建的Node节点安全组名称为**{集群名}-cce-node-{随机ID}**，默认端口说明请参见**表20-7**。

表 20-7 VPC 网络模型 Node 节点安全组默认端口说明

方向	端口	默认源地址	说明	是否可修改	修改建议
入方向规则	UDP：全部	VPC网段	Node节点之间互访、Node节点与Master节点互访。	不可修改	不涉及
	TCP：全部				
	ICMP：全部	Master节点安全组	Master节点访问Node节点。	不可修改	不涉及
	TCP：30000-32767	所有IP地址（0.0.0.0/0）	集群NodePort服务默认访问端口范围。	可修改	端口需对VPC网段、容器网段和ELB的网段放通。
	UDP：30000-32767				
	全部	容器网段	节点与容器互访。	不可修改	不涉及
	全部	Node节点安全组	Node节点之间互访。	不可修改	不涉及
	TCP：22	所有IP地址（0.0.0.0/0）	允许SSH远程连接Linux弹性云服务器。	**建议修改**	不涉及
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通，通常情况下不建议修改。	可修改	如需加固出方向规则，请注意指定端口需要放通，详情请参见**安全组出方向规则加固建议**。

Master节点安全组

集群自动创建的Master节点安全组名称为**{集群名}-cce-control-{随机ID}**，默认端口说明请参见**表20-8**。

表 20-8 VPC 网络模型 Master 节点安全组默认端口说明

方向	端口	默认源地址	说明	是否支持修改	修改建议
入方向规则	TCP：5444	VPC网段	kube-apiserver服务端口，提供K8s资源的生命周期管理。	不可修改	不涉及
	TCP：5444	容器网段			
	TCP：9443	VPC网段	Node节点网络插件访问Master节点。	不可修改	不涉及
	TCP：5443	所有IP地址（0.0.0.0/0）	Master的kube-apiserver的监听端口。	**建议修改**	端口需保留对VPC网段、容器网段和托管网格控制面网段放通。
	TCP：8445	VPC网段	Node节点存储插件访问Master节点。	不可修改	不涉及
	全部	本安全组	属于本安全组的源地址需全部放通。	不可修改	不涉及
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通。	不可修改	不涉及

容器隧道网络模型安全组规则

Node节点安全组

集群自动创建的Node节点安全组名称为**{集群名}-cce-node-{随机ID}**，默认端口说明请参见**表20-9**。

表 20-9 容器隧道网络模型 Node 节点安全组默认端口说明

方向	端口	默认源地址	说明	是否可修改	修改建议
入方向规则	UDP：4789	所有IP地址（0.0.0.0/0）	容器间网络互访。	不可修改	不涉及
	TCP：10250	Master节点网段	Master节点主动访问Node节点的kubelet（如执行kubectl exec {pod}）。	不可修改	不涉及

方向	端口	默认源地址	说明	是否可修改	修改建议
	TCP：30000-32767	所有IP地址（0.0.0.0/0）	集群NodePort服务默认访问端口范围。	可修改	端口需对VPC网段、容器网段和ELB的网段放通。
	UDP：30000-32767				
	TCP：22	所有IP地址（0.0.0.0/0）	允许SSH远程连接Linux弹性云服务器。	**建议修改**	不涉及
	全部	本安全组	属于本安全组的源地址需全部放通。	不可修改	不涉及
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通，通常情况下不建议修改。	可修改	如需加固出方向规则，请注意指定端口需要放通，详情请参见**安全组出方向规则加固建议**。

Master节点安全组

集群自动创建的Master节点安全组名称为**{集群名}-cce-control-{随机ID}**，默认端口说明请参见**表20-10**。

表 20-10 容器隧道网络模型 Master 节点安全组默认端口说明

方向	端口	默认源地址	说明	是否支持修改	修改建议
入方向规则	UDP：4789	所有IP地址（0.0.0.0/0）	容器间网络互访。	不可修改	不涉及
	TCP：5444	VPC网段	kube-apiserver服务端口，提供K8s资源的生命周期管理。	不可修改	不涉及
	TCP：5444	容器网段			
	TCP：9443	VPC网段	Node节点网络插件访问Master节点。	不可修改	不涉及

方向	端口	默认源地址	说明	是否支持修改	修改建议
	TCP：5443	所有IP地址（0.0.0.0/0）	master的kube-apiserver的监听端口。	**建议修改**	端口需保留对VPC网段、容器网段和托管网格控制面网段放通。
	TCP：8445	VPC网段	Node节点存储插件访问Master节点。	不可修改	不涉及
	全部	本安全组	属于本安全组的源地址需全部放通。	不可修改	不涉及
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通。	不可修改	不涉及

云原生网络 2.0（CCE Turbo 集群）安全组规则

Node节点安全组

集群自动创建的Node节点安全组名称为**{集群名}-cce-node-{随机ID}**，默认端口说明请参见**表20-11**。

表 20-11 CCE Turbo 集群 Node 节点安全组默认端口说明

方向	端口	默认源地址	说明	是否可修改	修改建议
入方向规则	TCP：10250	Master节点网段	Master节点主动访问Node节点的kubelet（如执行kubectl exec {pod}）。	不可修改	不涉及
	TCP：30000-32767	所有IP地址（0.0.0.0/0）	集群NodePort服务默认访问端口范围。	可修改	端口需对VPC网段、容器网段和ELB的网段放通。
	UDP：30000-32767				
	TCP：22	所有IP地址（0.0.0.0/0）	允许SSH远程连接Linux弹性云服务器。	**建议修改**	不涉及

方向	端口	默认源地址	说明	是否可修改	修改建议
	全部	本安全组	属于本安全组的源地址需全部放通。	不可修改	不涉及
	全部	控制节点子网网段	属于控制节点子网网段的源地址需全部放通。	不可修改	不涉及
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通，通常情况下不建议修改。	可修改	如需加固出方向规则，请注意指定端口需要放通，详情请参见**安全组出方向规则加固建议**。

Master节点安全组

集群自动创建的Master节点安全组名称为**{集群名}-cce-control-{随机ID}**，默认端口说明请参见**表20-12**。

表 20-12 CCE Turbo 集群 Master 节点安全组默认端口说明

方向	端口	默认源地址	说明	是否支持修改	修改建议
入方向规则	TCP：5444	所有IP地址（0.0.0.0/0）	kube-apiserver服务端口，提供K8s资源的生命周期管理。	不可修改	不涉及
	TCP：5444	VPC网段		不可修改	不涉及
	TCP：9443	VPC网段	Node节点网络插件访问Master节点。	不可修改	不涉及
	TCP：5443	所有IP地址（0.0.0.0/0）	master的kube-apiserver的监听端口。	**建议修改**	端口需保留对VPC网段、容器网段和托管网格控制面网段放通。
	TCP：8445	VPC网段	Node节点存储插件访问Master节点。	不可修改	不涉及
	全部	本安全组	属于本安全组的源地址需全部放通。	不可修改	不涉及
	全部	控制节点子网网段	属于控制节点子网网段的源地址需全部放通。	不可修改	不涉及

方向	端口	默认源地址	说明	是否支持修改	修改建议
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通。	不可修改	不涉及

ENI安全组

CCE Turbo集群会额外创建名为**{集群名}-cce-eni-{随机ID}**的安全组，默认端口说明请参见**表20-13**。

表 20-13 ENI 安全组默认端口说明

方向	端口	默认源地址	说明	是否可修改	修改建议
入方向规则	全部	本安全组	属于本安全组的源地址需全部放通。	不可修改	不涉及
		VPC网段	属于VPC网段的源地址需全部放通。	不可修改	不涉及
出方向规则	全部	所有IP地址（0.0.0.0/0）	默认全部放通。	不可修改	不涉及

安全组出方向规则加固建议

对于**出方向规则**，CCE创建的安全组默认全部放通，通常情况下不建议修改。如需加固出方向规则，请注意如下端口需要放通。

表 20-14 Node 节点安全组出方向规则最小范围

端口	放通地址段	说明
UDP：53	子网的DNS服务器	用于域名解析。
UDP：4789（仅容器隧道网络模型的集群需要）	所有IP地址	容器间网络互访。
TCP：5443	Master节点网段	master的kube-apiserver的监听端口。

端口	放通地址段	说明
TCP：5444	VPC网段、容器网段	kube-apiserver服务端口，提供K8s资源的生命周期管理。
TCP：6443	Master节点网段	-
TCP：8445	VPC网段	Node节点存储插件访问Master节点。
TCP：9443	VPC网段	Node节点网络插件访问Master节点。

20.7.2 网络异常

20.7.2.1 工作负载网络异常时，如何定位排查?

排查思路

以下排查思路根据原因的出现概率进行排序，建议您从高频率原因往低频率原因排查，从而帮助您快速找到问题的原因。

如果解决完某个可能原因仍未解决问题，请继续排查其他可能原因。

- **排查项一：容器+容器端口**
- **排查项二：节点IP+节点端口**
- **排查项三：负载均衡IP+端口**
- **排查项四：NAT网关+端口**
- **排查项五：检查容器所在节点安全组是否放通**

排查项一：容器+容器端口

在CCE控制台界面或者使用kubectl命令查找pod的IP，然后登录到集群内的节点或容器中，使用curl命令等方法手动调用接口，查看结果是否符合预期。

如果容器IP+端口不能访问，建议登录到业务容器内使用“127.0.0.1+端口”进行排查。

常见问题：

1. 容器端口配置错误（容器内未监听访问端口）。
2. URL不存在（容器内无相关路径）。
3. 服务异常（容器内的业务BUG）。
4. 检查集群网络内核组件是否异常（容器隧道网络模型：openswitch内核组件；VPC网络模型：ipvlan内核组件）。

排查项二：节点 IP+节点端口

只有发布为节点访问（NodePort）或负载均衡（LoadBalancer ）的服务才能通过节点IP+节点端口进行访问。

- **节点访问（NodePort）类型：**

  节点的访问端口就是节点对外发布的端口。

- **负载均衡（LoadBalancer ）类型：**

  负载均衡的节点端口通过“编辑YAML”可以查看。

如下图所示：

nodePort: 30637为节点对外暴露的端口。**targetPort: 80**为Pod对外暴露的端口。**port: 123**为服务对外暴露的端口，负载均衡类型的服务同时使用该端口配置ELB的监听器。

```
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 123
      targetPort: 80
      nodePort: 30637
```

找到节点端口（nodePort）后，使用容器所在节点的IP地址+端口进行访问，并查看结果是否符合预期。

常见问题：

1. 节点的入方向对业务端口未放通。
2. 节点配置了自定义路由，并且配置错误。
3. pod的label与service的label不匹配（kubectl或API创建）。

排查项三：负载均衡 IP+端口

如果使用负载均衡IP+端口不能访问，但节点IP+端口可以访问。

请排查：

- 相关端口或URL的后端服务器组是否符合预期。
- 节点上的安全组是否对ELB暴露了相关的协议或端口。
- 四层ELB的健康检查是否开启（未开启的话，请开启）。
- 七层ELB的访问方式中使用的证书是否过期。

常见问题：

1. 发布四层ELB时，如果客户在界面未开启健康检查，ELB可能会将流量转发到异常的节点。
2. UDP协议的访问，需要放通节点的ICMP协议。
3. pod的label与service的label不匹配（kubectl或API创建）。

排查项四：NAT 网关+端口

配置在NAT后端的服务器，通常不配置EIP，不然可能会出现网络丢包等异常。

排查项五：检查容器所在节点安全组是否放通

用户可单击服务列表中的“网络 > 虚拟私有云 VPC”，在网络控制台单击“访问控制 > 安全组”，找到CCE集群对应的安全组规则进行修改和加固。

- CCE集群：

  Node节点的安全组名称是：**{集群名}-cce-node-{随机字符}**。

请排查：

- 从集群外访问集群内负载时，来访者的IP地址、端口、协议需在集群安全组的入方向规则中开放。
- 集群内的工作负载访问外部时，访问的地址、端口、协议需在集群安全组的出方向规则中开放。

更多安全组配置信息请参见**集群安全组规则配置**。

20.7.2.2 为什么访问部署的应用时浏览器返回 404 错误码?

CCE服务本身在浏览器中访问应用时不会返回任何的错误码，请优先排查自身业务。

404 Not Found

如果404的返回如下图所示，说明这个返回码是ELB返回的，说明ELB找不到相关的转发策略。请排查相关的转发规则等。

图 20-9 404:ALB

如果404的返回如下图所示，说明这个返回码是由nginx（客户业务）返回，请排查客户自身业务问题。

图 20-10 404:nginx/1.**.*

20.7.2.3 为什么容器无法连接互联网?

当容器无法连接互联网时，首先需要排查容器所在节点能否连接互联网。其次，需要查看容器的网络配置是否正确，例如DNS配置是否可以正常解析域名。

排查项一：节点能否连接互联网

步骤1 登录ECS控制台。

步骤2 查看节点对应的弹性云服务器是否已绑定弹性IP或者配置NAT网关。

若弹性IP一栏有IP地址，表示已绑定弹性IP；若没有，请为弹性云服务器绑定弹性IP。

----结束

排查项二：节点是否配置网络 ACL

步骤1 登录VPC控制台。

步骤2 单击左侧导航栏的“访问控制 > 网络ACL”。

步骤3 排查节点所在集群的子网是否配置了网络ACL，并限制了外部访问。

----结束

排查项三：检查容器 DNS 配置

在容器中执行**cat /etc/resolv.conf**命令，查看DNS配置。示例如下：

```
nameserver 10.247.x.x
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

若nameserver设置为10.247.x.x说明DNS对接到集群的CoreDNS，需要确保集群CoreDNS工作负载运行正常。如果是其他IP地址，则表示采用云上DNS或者用户自建的DNS，请您自行确保解析正常。

20.7.2.4 节点无法连接互联网（公网），如何排查定位?

当节点无法连接互联网时，请参照如下方法排查。

排查项一：节点是否绑定弹性 IP

登录ECS控制台，查看节点对应的弹性云服务器是否已绑定弹性IP。

若弹性IP一栏有IP地址，表示已绑定弹性IP。若没有，请为弹性云服务器绑定弹性IP。

排查项二：节点是否配置网络 ACL

登录VPC控制台，单击左侧导航栏的“访问控制 > 网络ACL”。排查节点所在集群的子网是否配置了网络ACL，并限制了外部访问。

20.7.2.5 NGINX Ingress 控制器插件升级导致集群内 Nginx 类型的 Ingress 路由访问异常

问题现象

集群中存在未指定Ingress类型（annotations中未添加kubernetes.io/ingress.class: nginx）的Nginx Ingress路由，NGINX Ingress控制器插件从1.x版本升级至2.x版本后，服务中断。

问题自检

针对Nginx类型的Ingress资源，查看对应Ingress的YAML，如Ingress的YAML中未指定Ingress类型，并确认该Ingress由Nginx Ingress Controller管理，则说明该Ingress资源存在风险。

步骤1 获取Ingress类别。

您可以通过如下命令获取Ingress类别：

```
kubectl get ingress <ingress-name> -oyaml | grep -E ' kubernetes.io/ingress.class: | ingressClassName:'
```

- 故障场景：如果上述命令输出为空，说明Ingress资源未指定类别。
- 正常场景：Ingress已通过annotations或ingressClassName指定其类别，即存在输出。

```
[root@ + + + +  paas]# kubectl get ingress test -oyaml | grep -E ' kubernetes.io/ingress.class: | ingressClassName:' -B 1
Warning: extensions/v1beta1 Ingress is deprecated in v1.14+, unavailable in v1.22+; use networking.k8s.io/v1 Ingress
  annotations:
    kubernetes.io/ingress.class: nginx
--
spec:
  ingressClassName: nginx
```

步骤2 确认该Ingress被Nginx Ingress Controller纳管。如果使用ELB类型的Ingress则无此问题。

- 1.19集群可由通过managedFields机制确认。

  ```
  kubectl get ingress <ingress-name> -oyaml | grep 'manager: nginx-ingress-controller'
  ```

  ```
  [root@192-168-0-31 paas]# kubectl get ingress test -oyaml | grep 'manager: nginx-ingress-controller'
  Warning: extensions/v1beta1 Ingress is deprecated in v1.14+, unavailable in v1.22+; use networking.k8s.io/v1 Ingress
      manager: nginx-ingress-controller
  ```

- 其他版本集群可通过Nginx Ingress Controller Pod的日志确认。

  ```
  kubectl logs -nkube-system cceaddon-nginx-ingress-controller-545db6b4f7-bv74t | grep 'updating Ingress status'
  ```

  ```
  [root@ + + + +  paas]# kubectl logs -nkube-system   cceaddon-nginx-ingress-controller-545db6b4f7-bv74t | grep 'updating Ingress status'
  + + + + + + + +       8 status.go:281] "updating Ingress status" namespace="default" ingress="test" currentValue=[] newValue=[{IP: + + + +  Hostname: Ports:[]} {IP: + + + +  Hostname: Ports:[]}]
  ```

若通过上述两种方式仍然无法确认，请联系技术支持人员。

----结束

解决方案

为Nginx类型的Ingress添加注解，方式如下：

```
kubectl annotate ingress <ingress-name> kubernetes.io/ingress.class=nginx
```

须知

ELB类型的Ingress无需添加该注解，请**确认**该Ingress被Nginx Ingress Controller纳管。

问题根因

NGINX Ingress控制器插件基于开源社区Nginx Ingress Controller的模板与镜像。

对于社区较老版本的Nginx Ingress Controller来说（社区版本v0.49及以下，对应CCE插件版本v1.x.x），在创建Ingress时没有指定Ingress类别为nginx，即annotations中未添加kubernetes.io/ingress.class: nginx的情况，也可以被Nginx Ingress Controller纳管。详情请参见**社区代码**。

但对于较新版本的Nginx Ingress Controller来说（社区版本v1.0.0及以上，对应CCE插件版本2.x.x），如果在创建Ingress时没有显示指定Ingress类别为nginx，该资源将被Nginx Ingress Controller忽略，Ingress规则失效，导致服务中断。详情请参见请参见**社区代码**。

社区相关PR链接为：**https://github.com/kubernetes/ingress-nginx/pull/7341**

目前有两种方式指定Ingress类别：

- 通过annotations指定，为Ingress资源添加annotations（kubernetes.io/ingress.class: nginx）。
- 通过spec指定，.spec.ingressClassName字段配置为nginx。但需要配套具有IngressClass资源。

示例如下：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  ingressClassName: nginx
  rules:
    ...
status:
  loadBalancer: {}
```

20.8 存储管理

20.8.1 CCE 支持的存储在持久化和多节点挂载方面的区别是怎样的?

容器存储是为容器工作负载提供存储的组件，支持多种类型的存储，同一个工作负载（pod)可以使用任意数量的存储。

当前云容器引擎CCE支持本地磁盘存储、云硬盘存储卷、文件存储卷、对象存储卷和极速文件存储卷。

各类存储的区别和对比如下：

表 20-15 各类存储的区别和对比

存储类型	持久化存储	伴随容器自动迁移	多节点挂载
本地磁盘存储	支持	不支持	不支持
云硬盘存储卷（EVS）	支持	支持	不支持
对象存储卷（OBS）	支持	支持	支持，可由多个节点或工作负载共享
文件存储卷（SFS）	支持	支持	支持，可由多个节点或工作负载共享
极速文件存储卷（SFS Turbo）	支持	支持	支持，可由多个节点或工作负载共享

CCE 存储类型选择

创建工作负载时，可以使用以下类型的存储。建议将工作负载pod数据存储在云存储上。若存储在本地磁盘上，节点异常无法恢复时，本地磁盘中的数据也将无法恢复。

- 本地硬盘：将容器所在宿主机的文件目录挂载到容器的指定路径中（对应Kubernetes的HostPath），也可以不填写源路径（对应Kubernetes的EmptyDir），不填写时将分配主机的临时目录挂载到容器的挂载点，指定源路径的本地硬盘数据卷适用于将数据持久化存储到容器所在宿主机，EmptyDir（不填写源路径）适用于容器的临时存储。配置项（ConfigMap）是一种用于存储工作负载所需配置信息的资源类型，内容由用户决定。密钥（Secret）是一种用于存储工作负载所需要认证信息、密钥的敏感信息等的资源类型，内容由用户决定。
- 云硬盘存储卷：CCE支持将EVS创建的云硬盘挂载到容器的某一路径下。当容器迁移时，挂载的云硬盘将一同迁移。这种存储方式适用于需要永久化保存的数据。
- 文件存储卷： CCE支持创建SFS存储卷并挂载到容器的某一路径下，也可以使用底层SFS服务创建的文件存储卷，SFS存储卷适用于多读多写的持久化存储，适用于多种工作负载场景，包括媒体处理、内容管理、大数据分析和分析工作负载程序等场景。
- 对象存储卷：CCE支持创建OBS对象存储卷并挂载到容器的某一路径下，对象存储适用于云工作负载、数据分析、内容分析和热点对象等场景。
- 极速文件存储卷：CCE支持创建SFS Turbo极速文件存储卷并挂载到容器的某一路径下，极速文件存储具有按需申请，快速供给，弹性扩展，方便灵活等特点，适用于DevOps、容器微服务、企业办公等应用场景。

20.8.2 添加节点时可以不要数据盘吗?

不可以，数据盘是必须要的。

新建节点会给节点绑定一个供kubelet及容器引擎使用的专用数据盘。CCE数据盘默认使用LVM（Logical Volume Manager）进行磁盘管理，开启后您可以通过空间分配调整数据盘中不同资源的空间占比。

若数据盘卸载或损坏，会导致容器引擎服务异常，最终导致节点不可用。

20.8.3 公网访问 CCE 部署的服务并上传 OBS，为何报错找不到 host?

线下机器访问CCE部署的服务并上传OBS，报错找不到host，报错截图如下：

Time	message
February 22nd 2020, 18:50:27.521	com.obs.services.exception.ObsException: OBS servcie Error Message. Request Error: java.net.UnknownHostException: obs.
February 22nd 2020, 18:50:27.521	18:50:27.520 [XNIO-1 task-16] ERROR c.h.f.c.provider.ExceptionProvider - OBS servcie Error Message. Request Error: java.net.UnknownHostException: obs.
February 22nd 2020, 18:50:27.298	18:50:27.298 [XNIO-1 task-9] ERROR c.h.f.c.provider.ExceptionProvider - OBS servcie Error Message. Request Error: java.net.UnknownHostException: obs.
February 22nd 2020, 18:50:27.298	com.obs.services.exception.ObsException: OBS servcie Error Message. Request Error: java.net.UnknownHostException: obs.
February 22nd 2020, 18:50:27.275	18:50:27.274 [XNIO-1 task-9] WARN c.o.s.internal.RestStorageService - com.obs.services.internal.ServiceException: Request Error: java.net.UnknownHostException: obs. HEAD 'https://obs./obs-it-problem-management-media-test?apiversion' on Host 'obs.'
February 22nd 2020, 18:50:27.275	com.obs.services.internal.ServiceException: Request Error: java.net.UnknownHostException: obs.
February 22nd 2020, 18:50:27.275	2020-02-22 18:50:27 274|com.obs.services.internal.RestStorageService|handleThrowable|205|com.obs.services.internal.ServiceException: Request Error: java.net.UnknownHostException:

问题定位

服务收到http请求之后，向OBS传输文件，这些报文都会经过Proxy。

传输文件总量很大的话，会消耗很多资源，目前proxy分配内存128M，在压测场景下，损耗非常大，最终导致请求失败。

目前压测所有流量都经过Proxy，业务量大就要加大分配资源。

解决方法

1. 传文件涉及大量报文拷贝，会占用内存，建议把Proxy内存根据实际场景调高后再进行访问和上传。
2. 可以考虑把该服务从网格内移除出去，因为这里的Proxy只是转发包，并没有做其他事情，如果是通过Ingress Gateway走进来的话，这个服务的灰度发布功能是不受影响的。

20.8.4 Pod 接口 ExtendPathMode: PodUID 如何与社区 client-go 兼容?

使用场景

社区Pod结构体中没有ExtendPathMode，用户使用client-go调用创建pod或deployment的API接口时，创建的pod中没有ExtendPathMode。为了与社区的client-go兼容，CCE提供了如下解决方案。

解决方案

须知

- 创建pod时，在pod的annotation中需增加**kubernetes.io/extend-path-mode**。
- 创建deployment时，需要在template中的annotation增加**kubernetes.io/extend-path-mode**。

如下为创建pod的yaml示例，在annotation中添加**kubernetes.io/extend-path-mode**关键字后，完全匹配到containername，name，mountpath三个字段，则会在volumeMount中增加对应的**extendpathmode**：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-8b59d5884-96vdz
  generateName: test-8b59d5884-
  namespace: default
  selfLink: /api/v1/namespaces/default/pods/test-8b59d5884-96vdz
  labels:
    app: test
    pod-template-hash: 8b59d5884
  annotations:
    kubernetes.io/extend-path-mode:
'[{"containername":"container-0","name":"vol-156738843032165499","mountpath":"/
tmp","extendpathmode":"PodUID"}]'
    metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
  ownerReferences:
    - apiVersion: apps/v1
      kind: ReplicaSet
      name: test-8b59d5884
      uid: 2633020b-cd23-11e9-8f83-fa163e592534
      controller: true
      blockOwnerDeletion: true
spec:
  volumes:
    - name: vol-156738843032165499
      hostPath:
        path: /tmp
        type: ''
    - name: default-token-4s959
      secret:
        secretName: default-token-4s959
        defaultMode: 420
  containers:
    - name: container-0
      image: 'nginx:latest'
      env:
        - name: PAAS_APP_NAME
          value: test
        - name: PAAS_NAMESPACE
          value: default
        - name: PAAS_PROJECT_ID
          value: b6315dd3d0ff4be5b31a963256794989
      resources:
        limits:
          cpu: 250m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 512Mi
      volumeMounts:
        - name: vol-156738843032165499
          mountPath: /tmp
```

```
        extendPathMode: PodUID
      - name: default-token-4s959
        readOnly: true
        mountPath: /var/run/secrets/kubernetes.io/serviceaccount
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    imagePullPolicy: Always
restartPolicy: Always
terminationGracePeriodSeconds: 30
dnsPolicy: ClusterFirst
serviceAccountName: default
serviceAccount: default
nodeName: 192.168.0.24
securityContext: {}
imagePullSecrets:
  - name: default-secret
  - name: default-secret
affinity: {}
schedulerName: default-scheduler
tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
  - key: node.kubernetes.io/unreachable
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
priority: 0
dnsConfig:
  options:
    - name: timeout
      value: ''
    - name: ndots
      value: '5'
    - name: single-request-reopen
enableServiceLinks: true
```

表 20-16 关键参数说明

参数	参数类型	描述
containername	String	容器名称。
name	String	volume的名称。
mountpath	String	挂载路径
extendpathmode	String	将在已创建的“卷目录/子目录”中增加一个三级目录，便于更方便获取单个Pod输出的文件。 支持如下五种类型。 ● None：不配置拓展路径。 ● PodUID：Pod的ID。 ● PodName：Pod的名称。 ● PodUID/ContainerName：Pod的ID/容器名称。 ● PodName/ContainerName：Pod名称/容器名称。

20.8.5 CCE 容器云存储 PVC 能否感知底层存储故障?

CCE PVC按照社区逻辑实现，PVC本身的定义是存储声明，与底层存储解耦，不负责感知底层存储细节，因此没有感知底层存储故障的能力。

云监控服务CES 具备查看云服务监控指标的能力：云监控服务基于云服务自身的服务属性，已经内置了详细全面的监控指标。当用户在云平台上开通云服务后，系统会根据服务类型自动关联该服务的监控指标，帮助用户实时掌握云服务的各项性能指标，精确掌握云服务的运行情况。

建议有存储故障感知诉求的用户配套云监控服务CES的云服务监控能力使用，实现对底层存储的监控和告警通知。

20.9 命名空间

20.9.1 命名空间因 APIService 对象访问失败无法删除

问题现象

删除命名空间时，命名空间一直处“删除中”状态，无法删除。查看命名空间yaml配置，status中有报错“DiscoveryFailed”，示例如下：

```
76 status:
77   phase: Terminating
78   conditions:
79     - type: NamespaceDeletionDiscoveryFailure
80       status: 'True'
81       lastTransitionTime: '2022-07-04T13:44:55Z'
82       reason: DiscoveryFailed
83       message: 'Discovery failed for some groups, 1 failing: unable to retrieve the complete list of server
   APIs: metrics.k8s.io/v1beta1: the server is currently unable to handle the request'
84     - type: NamespaceDeletionGroupVersionParsingFailure
85       status: 'False'
```

上图中报错信息为：Discovery failed for some groups, 1 failing: unable to retrieve the complete list of server APIs: metrics.k8s.io/v1beta1: the server is currently unable to handle the request

表示当前删除命名空间动作阻塞在kube-apiserver访问metrics.k8s.io/v1beta1接口的APIService资源对象。

问题根因

当集群中存在APIService对象时，删除命名空间会先访问APIService对象，若APIService资源无法正常访问，会阻塞命名空间删除。除用户创建的APIService对象资源外，CCE集群部分插件也会自动创建APIService资源，如metrics-server、prometheus插件。

说明

APIService使用介绍请参考：https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/

解决方法

可以采用如下两种方法解决：

- 修复报错信息中的APIService对象，使其能够正常访问，如果是插件中的APIService，请确保插件的Pod正常运行。
- 删除报错信息中的APIService对象，如果是插件中的APIService，可从页面卸载该插件。

20.10 模板插件

20.10.1 插件安装失败，提示 The release name is already exist 处理

问题现象

当安装插件失败，返回 **The release name is already exist** 错误。

问题原因

当安装插件返回**The release name is already exist**错误时，表示kuberneters集群中有残留该插件release记录，一般由于集群etcd做过备份恢复或者该插件之前安装删除异常导致。

解决方案

通过kubectl对接集群，手动清理该插件release对应的secret及configmap。以下以清理autoscaler插件release为示例。

步骤1 配置kubectl对接集群后，执行以下命令查看插件相关的release的secret列表。

kubectl get secret -nkube-system |grep cceaddon

```
[root@cce-123-vpc-node2 ~]# kubectl get secret -nkube-system |grep cceaddon
sh.helm.release.v1.cceaddon-autoscaler.v1       helm.sh/release.v1                    1      61s
sh.helm.release.v1.cceaddon-autoscaler.v2       helm.sh/release.v1                    1      47s
sh.helm.release.v1.cceaddon-coredns.v1          helm.sh/release.v1                    1      6h2m
sh.helm.release.v1.cceaddon-everest.v1          helm.sh/release.v1                    1      6h2m
[root@cce-123-vpc-node2 ~]#
```

插件release的secret名称为"**sh.helm.release.v1.cceaddon-{插件名称}.v***"格式，可能有多个版本，删除时多个版本同时删除。

步骤2 执行删除release secret命令。

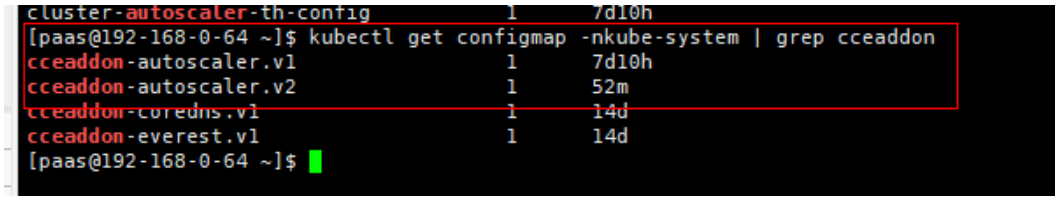
如删除上图中的autoscaler插件对应的release secret

kubectl delete secret sh.helm.release.v1.cceaddon-autoscaler.v1 sh.helm.release.v1.cceaddon-autoscaler.v2 -nkube-system

```
[root@cce-123-vpc-node2 ~]# kubectl delete secret sh.helm.release.v1.cceaddon-autoscaler.v1 sh.helm.release.v1.cceaddon-autoscaler.v2 -nkube-system
secret "sh.helm.release.v1.cceaddon-autoscaler.v1" deleted
secret "sh.helm.release.v1.cceaddon-autoscaler.v2" deleted
[root@cce-123-vpc-node2 ~]#
```

步骤3 若该插件为helm v2时创建，cce会在查看插件列表及插件详情等操作中自动将configmap中的v2 release转换至secret中的v3 release，原configmap中的v2 release不会删除。可执行以下命令查看插件相关的release的configmap列表。

kubectl get configmap -nkube-system | grep cceaddon

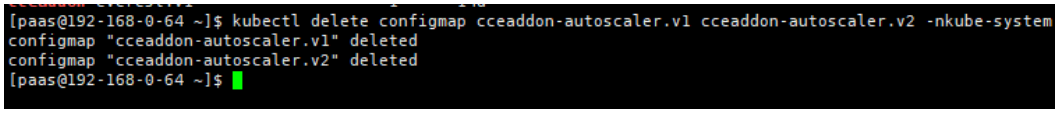


插件release的configmap名称为"**cceaddon-{插件名称}.v***"格式，可能有多个版本，删除时多个版本同时删除。

步骤4 执行删除release configmap命令。

如删除上图中的autoscaler插件对应的release configmap

kubectl delete configmap cceaddon-autoscaler.v1 cceaddon-autoscaler.v2 -nkube-system



注意

删除kube-system下资源属高风险操作，请确保命令正确后再执行，以免出现误删资源。

步骤5 在CCE控制台安装插件，然后再卸载保证之前的残留的插件资源清理干净，卸载完成后再进行第二次安装插件，安装成功即可。

说明

第一次安装插件是可能因之前的插件残留资源而导致安装后插件状态异常，属正常现象，这时在控制台卸载插件能保证这些残留资源清理干净，再次安装插件能正常运行。

----结束

20.11 API&kubectl

20.11.1 用户访问集群 API Server 的方式有哪些?

当前CCE提供两种访问集群API Server的方式：

- 集群API方式：（推荐）集群API需要使用证书认证访问。直接连接集群API Server，适合大规模调用。
- API网关方式：API网关采用token方式认证，需要使用账号信息获取token。适合小规模调用场景，大规模调用时可能会触发API网关流控。

20.11.2 通过 API 或 kubectl 操作 CCE 集群，创建的资源是否能在控制台展示?

在CCE控制台，暂时不支持显示的kubernetes资源有：**DaemonSet**、**ReplicationController**、**ReplicaSets**、**Endpoints**等。

若需要查询这些资源，请通过kubectl命令进行查询。

此外，Deployment、Statefulset、Service和Pod资源需满足以下条件，才能在控制台显示：

- **Deployment和Statefulset**：标签中必须至少有一个标签是以"app"为key的。
- **Pod**：只有创建了无状态工作负载（Deployment）和有状态工作负载（StatefulSet）后，对应Pod实例才会在工作负载详情页的“实例列表”页签中显示。
- **Service**：Service当前在无状态工作负载（Deployment）和有状态工作负载（StatefulSet）详情页的“访问方式”页签中显示。

  此处的显示需要Service与工作负载有一定的关联：

  a. 工作负载中的一个标签必须是以"app"为key。
  b. Service的标签和工作负载的标签保持一致。

20.11.3 通过 kubectl 连接集群时，其配置文件 config 如何下载?

步骤1 登录CCE控制台，单击需要连接的集群名称，进入“集群信息”页面。

步骤2 在“连接信息”版块中查看kubectl的连接方式。

步骤3 在弹出的窗口中可以下载kubectl配置文件kubeconfig.json。

----结束

20.11.4 kubectl top node 命令为何报错

故障现象

执行kubectl top node命令报错Error from server (ServiceUnavailable): the server is currently unable to handle the request (get nodes.metrics.k8s.io)

可能原因

执行kubectl时出现Error from server (ServiceUnavailable)时，表示未能连接到集群，需要检查kubectl到集群Master节点的网络是否能够连通。

解决方法

- 如果是在集群外部执行kubectl，请检查集群是否绑定公网IP，如已绑定，请重新下载kubeconfig文件配置，然后重新执行kubectl命令。
- 如果是在集群内节点上执行kubectl，请检查节点的安全组，是否放通Node节点与Master节点TCP/UDP互访，安全组的详细说明请参见**集群安全组规则配置**

20.11.5 kubectl 使用报错：Error from server (Forbidden)

故障现象

使用kubectl在创建或查询Kubernetes资源时，显示如下内容。

kubectl get deploy Error from server (Forbidden): deployments.apps is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "deployments" in API group "apps" in the namespace "default"

问题根因

用户没有操作该Kubernetes资源的权限。

解决方法

给该用户授权Kubernetes权限，具体方法如下。

步骤1 登录CCE控制台，在左侧导航栏中选择“权限管理”。

步骤2 在右边下拉列表中选择要添加权限的集群。

步骤3 在右上角单击“添加权限”，进入添加授权页面。

步骤4 在添加权限页面，确认集群名称，选择该集群下要授权使用的命名空间，例如选择“全部命名空间”，选择要授权的用户或用户组，再选择具体权限。

说明

对于没有IAM权限的用户，给其他用户和用户组配置权限时，无法选择用户和用户组，此时支持填写用户ID或用户组ID进行配置。

其中自定义权限可以根据需要自定义，选择自定义权限后，在自定义权限一行右侧单击新建自定义权限，在弹出的窗口中填写名称并选择规则。创建完成后，在添加权限的自定义权限下拉框中可以选择。

步骤5 单击“确定”。

----结束

20.12 域名 DNS

20.12.1 域名解析失败，如何定位处理?

排查项一：检查是否已安装 CoreDNS 插件

步骤1 登录CCE控制台，进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”中确认异常的集群是否已安装CoreDNS插件。

步骤3 如果未安装，请安装。详情请参见**为什么CCE集群的容器无法通过DNS解析?**

----结束

排查项二：检查 CoreDNS 实例是否已到达性能瓶颈

CoreDNS所能提供的域名解析QPS与CPU消耗成正相关，如遇QPS较高的场景，需要根据QPS的量级调整CoreDNS实例规格。

步骤1 登录CCE控制台，进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”中找到集群对应的CoreDNS插件，确认插件状态为“运行中”。

步骤3 单击CoreDNS插件名称，查看插件实例列表。

步骤4 单击CoreDNS实例的“监控”按钮，查看实例CPU、内存使用率。

如实例已达性能瓶颈，则需调整CoreDNS插件规格。

----结束

排查项三：解析外部域名很慢或超时

如果域名解析失败率低于1/10000，请参考**解析外部域名很慢或超时，如何优化配置?**进行参数优化，或在业务中增加重试。

排查项四：概率性出现 UnknownHostException

集群中的业务请求到外部域名服务器时发生域名解析错误，概率性出现UnknownHostException。UnknownHostException是一个常见的异常，发生该异常时优先检查域名是否存在问题或键入错误。

您可根据以下步骤进行排查：

步骤1 仔细检查主机名是否正确，检查域名的拼写并删除多余的空格。

步骤2 检查DNS设置。在运行应用程序之前，通过**ping hostname**命令确保DNS服务器已启动并正在运行。如果主机名是新的，则需要等待一段时间才能访问DNS服务器。

步骤3 检查CoreDNS实例的CPU、内存使用率监控，确认是否已到达性能瓶颈，具体操作步骤请参见**排查项二：检查CoreDNS实例是否已到达性能瓶颈**。

步骤4 检查CoreDNS是否有发生限流，如果触发限流可能出现部分请求处理时间延长，需要调整CoreDNS插件规格。

登录CoreDNS Pod所在节点，查看以下文件内容：

```
cat /sys/fs/cgroup/cpu/kubepods/pod<pod_uid>/<coredns容器id>/cpu.stat
```

- *<pod uid>*为CoreDNS的Pod UID，可通过以下命令获取：

  ```
  kubectl get po <pod name> -nkube-system -ojsonpath='{.metadata.uid}{"\n"}'
  ```

  以上命令中的*<pod name>*需要是在当前节点上运行的CoreDNS Pod名称。

- *<coredns容器id>*需要是完整的容器ID，可通过以下命令获取：

  ```
  docker ps --no-trunc | grep k8s_coredns | awk '{print $1}'
  ```

完整的命令示例如下：

```
cat /sys/fs/cgroup/cpu/kubepods/
pod27f58662-3979-448e-8f57-09b62bd24ea6/6aa98c323f43d689ac47190bc84cf4fadd23bd8dd25307f773df25003ef0eef0/cpu.stat
```

请关注以下指标：

- nr_throttled：被限流次数。
- throttled_time：被限流的总时间长度（纳秒）。

----结束

如果检查后无上述问题，可采用下方优化策略。

优化策略：

1. 修改CoreDNS的缓存时间
2. 配置存根域
3. 修改ndots

说明

- **增加coredns的缓存时间：**有利于同一个域名的第N次解析，较少级联DNS的请求数量。
- **配置存根域：**有利于减少DNS请求链路。

修改方式：

1. 修改CoreDNS缓存时间及配置存根域
   修改完成后重启CoreDNS。
2. 修改ndots
   修改方法请参见**解析外部域名很慢或超时，如何优化配置？**。
   示例：

```
dnsConfig:
  options:
    - name: timeout
      value: '2'
    - name: ndots
      value: '5'
    - name: single-request-reopen
```

   建议值修改成：2

20.12.2 为什么 CCE 集群的容器无法通过 DNS 解析?

问题描述

某客户在DNS服务中做内网解析，将自有的域名绑定到DNS服务中的内网域名中，并绑定到特定的VPC中，发现本VPC内的节点（ECS）可以正常解析内网域名的记录，而VPC内的容器则无法解析。

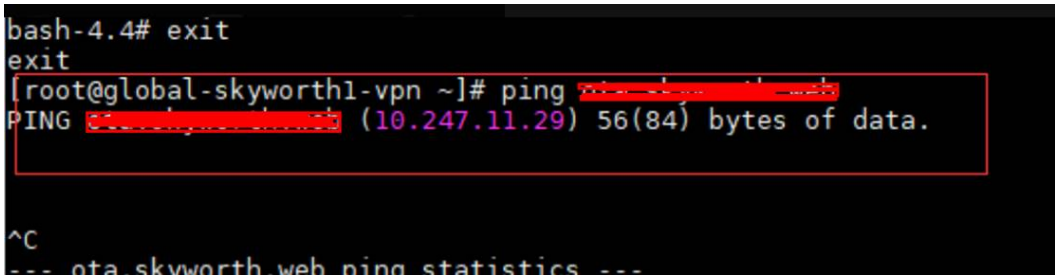
适用场景

VPC内的容器无法进行正常DNS解析的情况。

解决方案

由于本案例涉及的是内网域名解析，按照内网域名解析的规则，需要确保VPC内的子网DNS设置成的云上DNS，具体以内网DNS服务的控制台界面提示为准。

本案例的子网中已经完成该设置，其中用户可以在该VPC子网内的节点（ECS）进行域名解析，也说明已完成该设置，如下图：



但是在容器内进行解析却提示bad address无法解析域名返回地址，如下图：

登录CCE控制台查看该集群的插件安装情况。

如果已安装插件列表中没有coredns插件，可能是用户卸载了该插件等原因导致。

安装coredns插件，并添加相应的域名及对应的DNS服务地址，即可进行域名解析。

20.12.3 解析外部域名很慢或超时，如何优化配置?

工作负载的容器内的resolv.conf文件，示例如下：

```
root@test-5dffdddf95-vpt4m:/# cat /etc/resolv.conf
nameserver 10.247.3.10
search istio.svc.cluster.local svc.cluster.local cluster.local
options ndots:5 single-request-reopen timeout:2
```

其中：

- **nameserver**：DNS服务器的IP地址，此处为coredns的ClusterIP。
- **search**：域名的搜索列表，此处为Kubernetes的常用后缀。
- **ndots**：“.”的个数小于它的域名，会优先使用search进行解析。
- **timeout**：超时时间。
- **single-request-reopen**：发送A类型请求和AAAA类型请求使用不同的源端口。

在界面创建工作负载时，以上几项配置默认都会创建，具体参数如下：

```
dnsConfig:
  options:
    - name: timeout
      value: '2'
    - name: ndots
      value: '5'
    - name: single-request-reopen
```

以上参数可以根据业务需要进行优化或修改。

场景一：解析外部域名慢

优化方案：

1. 如果此工作负载不需要访问集群内的k8s服务，可以参考**如何设置容器内的DNS策略?**。
2. 如果此工作服务访问其他的k8s服务时，使用的域名中“.”的个数小于2，可以将ndots参数设置为2。

场景二：解析外部域名超时

优化方案：

1. 通常业务内的超时时间要大于timeout * attempts的时间。
2. 如果解析此域名通常要超过2s，可以将timeout改大。

20.12.4 如何设置容器内的 DNS 策略?

CCE支持通过dnsPolicy标记每个Pod配置不同的DNS策略：

- **None**：表示空的DNS设置，这种方式一般用于想要自定义DNS配置的场景，而且，往往需要和dnsConfig配合一起使用达到自定义DNS的目的。
- **Default**：从运行所在的节点继承名称解析配置。即容器的域名解析文件使用kubelet的“--resolv-conf”参数指向的域名解析文件（CCE集群在该配置下对接云上DNS）。
- **ClusterFirst**：这种方式表示Pod内的DNS使用集群中配置的DNS服务，简单来说，就是使用Kubernetes中kubedns或coredns服务进行域名解析。如果解析不成功，才会使用宿主机的DNS配置进行解析。

如果未明确指定dnsPolicy，则默认使用“ClusterFirst”：

- 如果将dnsPolicy设置为“Default”，则名称解析配置将从运行pod的工作节点继承。
- 如果将dnsPolicy设置为“ClusterFirst”，则DNS查询将发送到kube-dns服务。

  对于以配置的集群域后缀为根的域的查询将由kube-dns服务应答。所有其他查询（例如，www.kubernetes.io）将被转发到从节点继承的上游名称服务器。在此功能之前，通常通过使用自定义解析程序替换上游DNS来引入存根域。但是，这导致自定义解析程序本身成为DNS解析的关键路径，其中可伸缩性和可用性问题可能导致集群丢失DNS功能。此特性允许用户在不接管整个解析路径的情况下引入自定义解析。

如果某个工作负载不需要使用集群内的coredns，可以使用kubectl命令或API将此策略设置为dnsPolicy: Default。

20.13 镜像仓库

20.13.1 如何上传我的镜像到 CCE 中使用?

镜像的管理是由容器镜像服务（SoftWare Repository）提供的，当前容器镜像服务提供如下上传镜像的方法：

- **客户端上传镜像**
- **页面上传镜像**

20.14 权限

20.14.1 能否只配置命名空间权限，不配置集群管理权限?

命名空间权限和集群管理权限是相互独立又相互补充的两个权限体系：

- 命名空间权限：作用于集群内部，用于管理集群资源操作（如创建工作负载等）。
- 集群管理（IAM）权限：云服务层面的权限，用于管理CCE集群与周边资源（如VPC、ELB、ECS等）的操作。

对于IAM Admin用户组的管理员用户来说，可以为IAM子用户授予集群管理权限（如CCE Administrator、CCE FullAccess等），也可以在CCE控制台授予某个集群的命名空间权限。但由于CCE控制台界面权限是由IAM系统策略进行判断，如果IAM子用户未配置集群管理（IAM）权限，该子用户将无法进入CCE控制台。

如果您无需使用CCE控制台，只使用kubectl命令操作集群中的资源，则不受集群管理（IAM）权限的影响，您只需要获取具有命名空间权限的配置文件（kubeconfig），详情请参考**如果不配置集群管理权限，是否可以使用kubectl命令呢？**。集群配置文件在传递过程中可能存在泄露风险，应在实际使用中注意。

20.14.2 如果不配置集群管理权限的情况下，是否可以使用 API 呢?

CCE提供的API可以分为云服务接口和集群接口：

- 云服务接口：支持操作云服务层面的基础设施（如创建节点），也可以调用集群层面的资源（如创建工作负载）。

  使用云服务接口时，必须配置集群管理（IAM）权限。

- 集群接口：直接通过Kubernetes原生API Server来调用集群层面的资源（如创建工作负载），但不支持操作云服务层面的基础设施（如创建节点）。

  使用集群接口时，无需配置集群管理（IAM）权限，仅需在调用集群接口时带上集群证书。但是，集群证书需要有集群管理（IAM）权限的用户进行下载，在证书传递过程中可能存在泄露风险，应在实际使用中注意。

20.14.3 如果不配置集群管理权限，是否可以使用 kubectl 命令呢?

使用kubectl命令无需经过IAM认证，因此理论上不配置集群管理（IAM）权限是可以使用kubectl命令的。但前提是需要获取具有命名空间权限的kubectl配置文件（kubeconfig），以下场景认证文件传递过程中均存在安全泄露风险，应在实际使用中注意。

- 场景一

  如果某IAM子用户先配置了集群管理权限和命名空间权限，然后在界面下载kubeconfig认证文件。后面再删除集群管理权限（保留命名空间权限），依然可以使用kubectl来操作Kubernetes集群。因此如需彻底删除用户权限，必须同时删除该用户的集群管理权限和命名空间权限。

- 场景二

  如果某IAM用户拥有一定范围的集群管理权限和命名空间权限，然后在界面下载kubeconfig认证文件。此时CCE根据用户信息的权限判断kubectl有权限访问哪些Kubernetes资源，即哪个用户获取的kubeconfig文件，kubeconfig中就拥有哪个用户的认证信息，任何人都可以通过这个kubeconfig文件访问集群。

20.15 参考知识

20.15.1 如何扩容容器的存储空间?

使用场景

容器默认大小为10G，当容器中产生数据较多时，容易导致容器存储空间不足，可以通过此方法来扩容。

解决方案

步骤1 登录CCE控制台，单击集群列表中的集群名称。

步骤2 在左侧导航栏中选择“节点管理”。

步骤3 选择集群中的节点，单击操作列中的“更多 > 重置节点”。

须知

重置节点操作可能导致与节点有绑定关系的资源（本地存储，指定调度节点的负载等）无法正常使用。请谨慎操作，避免对运行中的业务造成影响。

步骤4 在确认页面中单击“是”。

步骤5 重新配置节点参数。

如需对容器存储空间进行调整，请重点关注以下配置。

存储配置：单击数据盘后方的“展开高级设置”可进行如下设置：

- 自定义容器引擎空间大小：容器引擎占用的存储空间，默认为数据盘空间的90%，用于存放容器引擎 (Docker/Containerd) 工作目录、容器镜像的数据和镜像元数据。
- 自定义容器Pod空间大小：CCE 支持对每个工作负载下的容器组 Pod 占用的磁盘空间设置上限（包含容器镜像占用的空间）。合理的配置可避免容器组无节制使用磁盘空间导致业务异常。建议此值不超过容器引擎空间的 80%。

  说明

  - 自定义容器Pod存储空间的能力与节点操作系统与容器存储Rootfs有关，设置规则如下：
    - 容器存储Rootfs使用DeviceMapper时，节点支持自定义容器Pod空间设置（basesize），单个容器存储空间大小默认为10GiB，可以配置为其他值。
    - 容器存储Rootfs使用OverlayFS时，大部分节点不支持自定义容器Pod空间设置（basesize），默认为不限制，即单个容器存储空间大小默认为容器引擎空间。

      仅1.19.16版本、1.21.3版本、1.23.3版本及之后版本集群中的EulerOS 2.9系统节点支持自定义容器Pod空间设置（basesize），可以配置为其他值。
  - 使用EulerOS 2.9 的docker basesize设置时，若容器配置CAP_SYS_RESOURCE权限或privileged的特权，basesize限制单容器数据空间不起作用。

步骤6 重置节点后登录该节点，执行如下命令进入容器，查看docker容器容量是否已扩容。

docker exec -it *container_id* **/bin/sh**或**kubectl exec -it** *container_id* **/bin/sh**

df -h

```
# df -h
Filesystem                                                                                                  Size  Used Avail Use% Mounted on
/dev/mapper/docker-253:1-787293-631c1bde2cbe82e39f32253b216ba914cb183b168b54700b3e5b9a54ee40a0d1   15G  229M   15G   2% /
tmpfs                                                                                                       32G     0   32G   0% /dev
tmpfs                                                                                                       32G     0   32G   0% /sys/fs/cgroup
/dev/mapper/vgpaas-kubernetes                                                                              9.8G   37M  9.2G   1% /etc/hosts
/dev/vda1                                                                                                   40G  5.2G   33G  14% /etc/hostname
shm                                                                                                         64M     0   64M   0% /dev/shm
tmpfs                                                                                                       32G   16K   32G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                                                                                                       32G     0   32G   0% /proc/acpi
tmpfs                                                                                                       32G     0   32G   0% /sys/firmware
tmpfs                                                                                                       32G     0   32G   0% /proc/scsi
tmpfs                                                                                                       32G     0   32G   0% /proc/kbox
tmpfs                                                                                                       32G     0   32G   0% /proc/oom_extend
```

----结束

20.15.2 如何使容器重启后所在容器 IP 仍保持不变?

单节点场景

如果集群下仅有1个节点时，要使容器重启后所在容器IP保持不变，需在工作负载中配置主机网络，在工作负载的yaml中的spec.spec.下加入hostNetwork: true字段。

多节点场景

如果集群下有多个节点时，除进行以上操作外，还需要设置节点的亲和策略，但工作负载创建后实例运行数不得超过亲和的节点数。

完成效果

进行如上设置并在工作负载运行后，工作负载实例ip与节点ip将保持一致，重启工作负载后ip也将保持不变。

21 最佳实践

21.1 容器应用部署上云 CheckList

简介

安全高效、稳定高可用是每一位涉云从业者的共同诉求。这一诉求实现的前提，离不开系统可用性、数据可靠性及运维稳定性三者的配合。本文将通过评估项目、影响说明及评估参考三个角度为您阐述容器应用部署上云的各个检查项，以便帮助您扫除上云障碍、顺利高效地完成业务迁移至云容器引擎（CCE），降低因为使用不当导致集群或应用异常的风险。

检查项

表 21-1 系统可用性

类别	评估项目	类型	影响说明
集群	创建集群前，根据业务场景提前规划节点网络和容器网络，避免后续业务扩容受限。	网络规划	集群所在子网或容器网段较小，将可能导致集群实际支持的可用节点数少于业务所需容量。
	创建集群前，提前梳理云专线、对等连接、容器网段、服务网段和子网网段等相关网段的规划，避免出现网段冲突影响业务。	网络规划	简单组网场景按照页面提示配置集群相关网段，避免冲突；业务复杂组网场景，例如对等连接、云专线、VPN等，网络规划不当将影响整体业务正常互访。
	创建集群时，会自动新建并绑定默认安全组，支持根据业务需求设置自定义安全组规则。	部署	安全组是重要的安全隔离手段，不当的安全策略配置可能会引起安全相关的隐患及服务连通性等问题。

类别	评估项目	类型	影响说明
	使用多控制节点模式，创建集群时将控制节点数设置为3。	可靠性	多控制节点模式开启后将创建三个控制节点，在单个控制节点发生故障后集群可以继续使用，不影响业务功能。商用场景建议选择多控制节点模式集群。
	创建集群时，根据业务场景选择合适的网络模型：容器隧道网络、VPC网络。	部署	集群创建成功后，网络模型不可更改，请谨慎选择。
工作负载	创建工作负载时需设置CPU和内存的限制范围，提高业务的健壮性。	部署	同一个节点上部署多个应用时，当未设置资源上下限的应用出现应用异常资源泄露问题时，将会导致其它应用分配不到资源而异常，且应用监控信息会出现误差。
	创建工作负载时可设置容器健康检查：“工作负载存活探针”和“工作负载业务探针”	可靠性	容器健康检查未配置，会导致用户业务出现异常时Pod无法感知，从而导致不会自动重启恢复业务，最终将会出现Pod状态正常，但Pod中的业务异常的现象。
	创建服务时需要根据实际访问需求选择合适的访问方式，目前支持以下几种：集群内访问（ClusterIP）、节点访问（NodePort）、负载均衡(LoadBalancer)。	部署	选择不当的访问方式，可能造成服务内外部访问逻辑混乱和资源浪费。
	工作负载创建时，避免单Pod副本数设置，请根据自身业务合理设置节点调度策略。	可靠性	如设置单Pod副本数，当节点异常或实例异常会导致服务异常。为确保您的Pod能够调度成功，请确保您在设置调度规则后，节点有空余的资源用于容器的调度。
	合理设置“亲和性”和“反亲和性”	可靠性	对外提供服务的应用，如果以“或”的关系同时配置“亲和性”和“反亲和性”，应用升级或者重启后，会概率出现服务无法访问的问题。
	设置应用生命周期中的“停止前处理”，确保升级或者实例删除时可以提前将实例中运行的业务处理完成	可靠性	如果没有配置，用户在应用升级时，Pod会被直接Kill，导致Pod中运行的业务中断。

表 21-2 数据可靠性

类别	评估项目	类型	影响说明
容器数据持久化	应用Pod数据存储，根据实际需求选择合适的数据卷类型。	可靠性	节点异常无法恢复时，存在本地磁盘中的数据无法恢复，而云存储此时可以提供极高的数据可靠性。
数据备份	对应用数据进行备份	可靠性	数据丢失后，无法恢复。

表 21-3 运维可靠性

类别	评估项目	类型	影响说明
工程	ECS、VPC、子网、EIP及EVS等资源配额是否满足客户需求。	部署	配额不足会导致创建资源失败，对于配置了自动扩容的用户尤其需要保障所使用的云服务配额充足。
	集群的节点上不建议用户随意修改内核参数、系统配置、集群核心组件版本、安全组及ELB相关参数，也不建议用户随意安装未经验证的软件。	部署	可能会导致CCE集群功能异常或安装在节点上的Kubernetes组件异常，节点状态变成不可用，无法部署应用到此节点。
	不要修改CCE创建的安全组、云硬盘等信息。CCE创建的资源标记有“cce”字样	部署	会导致CCE集群功能异常。
主动运维	云容器引擎提供多维度的监控和告警功能，配置监控告警，以便于异常时及时收到告警并进行故障定位。 ● 云监控服务AOM：CCE默认的基础资源监控，覆盖详细的容器相关指标，并提供告警配置能力。 ● 开源Prometheus：面向云原生应用程序的开源监控工具，并集成独立的告警系统，提供更高自由度的监控告警配置。	监控	未配置监控告警，将无法建立容器集群性能的正常标准，在出现异常时无法及时收到告警，需要人工巡检环境。

21.2 容器化改造

21.2.1 企业管理应用容器化改造（ERP）

21.2.1.1 方案概述

本手册基于云容器引擎实践所编写，用于指导您已有应用的容器化改造。

什么是容器

容器是操作系统内核自带能力，是基于Linux内核实现的轻量级高性能资源隔离机制。

云容器引擎CCE是基于开源Kubernetes的企业级容器服务，提供高可靠高性能的企业级容器应用管理服务，支持Kubernetes社区原生应用和工具，简化云上自动化容器运行环境搭建。

为什么需要使用容器

- 更高效的利用系统资源。

  容器不需要硬件虚拟化以及运行完整操作系统等额外开销，所以对系统资源利用率更高。相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

- 更快速的启动时间。

  容器直接运行于宿主机内核，无需启动完整的操作系统，可以做到秒级甚至毫秒级的启动时间。大大节约开发、测试、部署的时间。

- 一致的运行环境。

  容器镜像提供了完整的运行时环境，确保应用运行环境的一致性。从而不会再出现“这段代码在我机器上没问题”这类问题。

- 更轻松的迁移、维护和扩展。

  容器确保了执行环境的一致性，使得应用迁移更加容易。同时使用的存储及镜像技术，使应用重复部分的复用更为容易，基于基础镜像进一步扩展镜像也变得非常简单。

企业应用容器化改造方式

应用容器化改造，一般有以下三种方式：

- 方式一：单体应用整体容器化，应用代码和架构不做任何改动。
- 方式二：将应用中升级频繁，或对弹性伸缩要求高的组件拆分出来，将这部分组件容器化。
- 方式三：将应用做全面的微服务架构改造，再单独容器化。

这三种方式的优缺点如**表21-4**。

表 21-4 应用容器化改造方式

应用容器化改造方式	优点	缺点
方式一： 单体应用整体容器化	• 业务0修改：应用架构和代码不需要做任何改动。 • 提升部署和升级效率：应用可构建为容器镜像，确保应用环境一致性，提升部署效率。 • 降低资源成本：容器对系统资源利用率高。相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。	• 整体性架构扩展难度大，随着应用程序代码扩展，更新和维护工作非常复杂。 • 推出新功能、语言、框架和技术都比较困难。
方式二： 先将部分组件容器化（将对弹性扩展要求高，或更新频繁的组件拆分出来，先容器化改造）	• 渐进式变革：在原有架构推倒重建太伤筋动骨，通过较为缓和的改动，更容易接受。 • 弹性更灵活：将对弹性要求高的组件容器化，当需要扩展时，只针对该容器扩展，弹性更灵活，且能降低系统资源。 • 新特性上线更快：将更新频繁的组件容器化，只针对这个容器进行升级，上线更快。	需要对业务做部分解耦拆分。

应用容器化改造方式	优点	缺点
方式三： 整体微服务架构改造，再容器化	• 单独扩展：拆分为微服务后，可单独增加或缩减每个微服务的实例数量。 • 提升开发速度：各微服务之间解耦，某个微服务的代码开发不影响其他微服务。 • 通过隔离确保安全：整体应用中，若存在安全漏洞，会获得所有功能的权限。微服务架构中，若攻击了某个服务，只可获得该服务的访问权限，无法入侵其他服务。 • 隔离崩溃：如果其中一个微服务崩溃，其它微服务还可以持续正常运行。	业务需要微服务化改造，改动较大。

本教程以“方式一”为例，将单体的企业ERP系统做整体的容器化改造。

21.2.1.2 实施步骤

21.2.1.2.1 整体应用容器化改造

本教程以“整体应用容器化改造”为例，指导您将一个“部署在虚拟机上的ERP企业管理系统”进行容器化改造，部署到容器服务中。

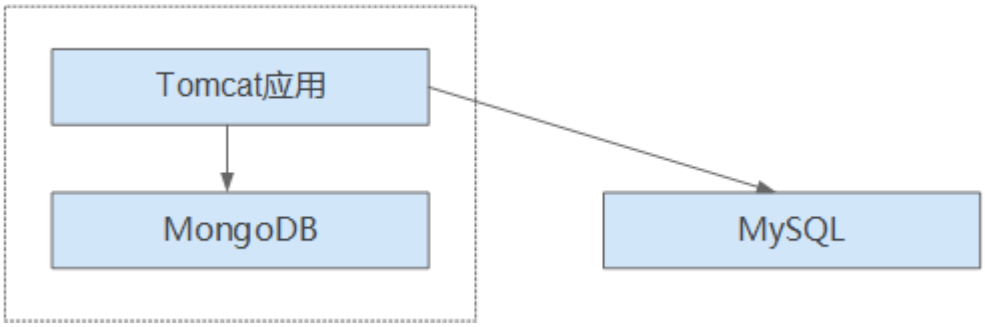
您不需要改动任何代码和架构，仅需将整体应用构建为容器镜像，部署到云容器引擎中。

本例应用简介

本例“企业管理应用”由某企业（简称A企业）开发，这款应用提供给不同的第三方企业客户，第三方客户仅需要使用应用，维护工作由A企业提供。

在第三方企业需要使用该应用时，需要在第三方企业内部部署一套“Tomcat应用和MongoDB数据库”，MySQL数据库由A企业提供，用于存储各第三方企业的数据。

图 21-1 应用架构



如图21-1，该应用是标准的tomcat应用，后端对接了MongoDB和MySQL。这种类型应用可以先不做架构的拆分，将整体应用构建为一个镜像，将tomcat应用和mongoDB共同部署在一个镜像中。这样，当其他企业需要部署或升级应用时，可直接通过镜像来部署或升级。

- 对接mongoDB：用于用户文件存储。
- 对接MySQL：用于存储第三方企业数据，MySQL使用外部云数据库。

本例应用容器化改造价值

本例应用原先使用虚机方式部署，在部署和升级时，遇到了一系列的问题，而容器化部署解决了这些问题。

通过使用容器，您可以轻松打包应用程序的代码、配置和依赖关系，将其变成易于使用的构建块，从而实现环境一致性、运营效率、开发人员工作效率和版本控制等诸多目标。容器可以帮助保证应用程序快速、可靠、一致地部署，不受部署环境的影响。

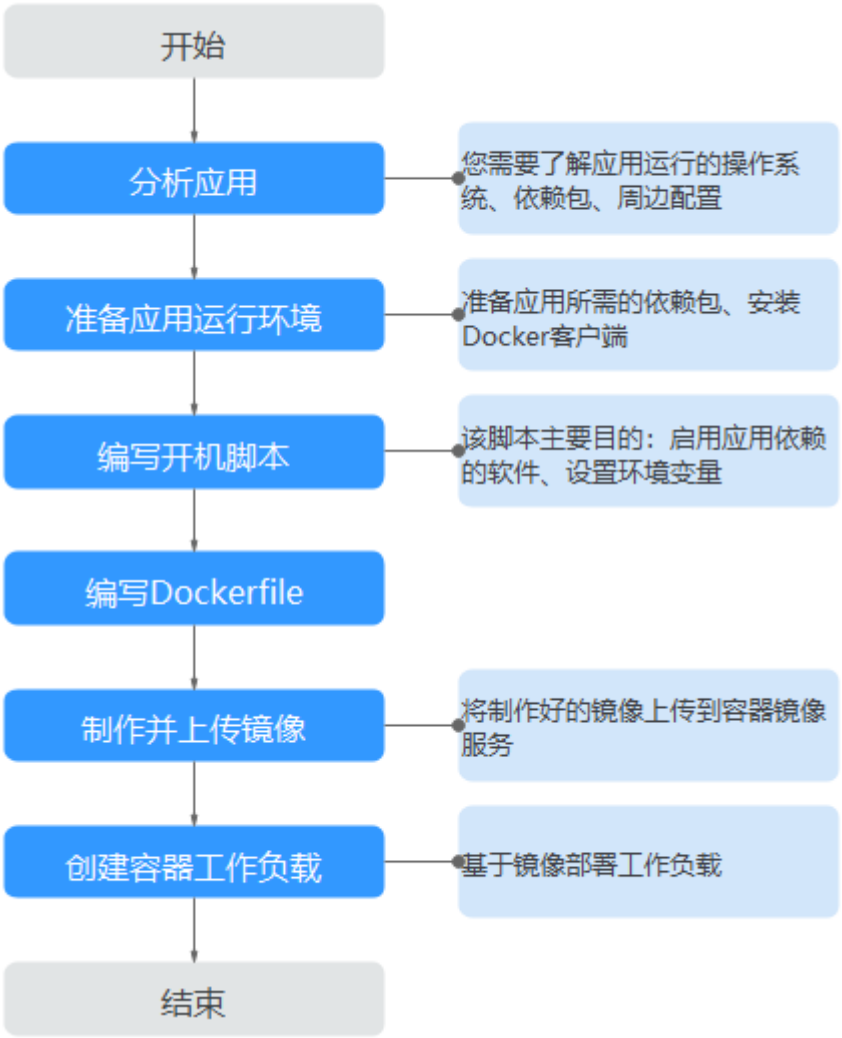
表 21-5 虚机和容器部署对比表

类别	before：虚机部署	after：容器部署
部署	部署成本高。 每给一家客户部署一套系统，就需要购置一台虚拟机。	成本降低50%以上。 通过容器服务实现了多租隔离，在同一台虚拟机上可以给多个企业部署系统。
升级	升级效率低。 版本升级时，需要逐台登录虚拟机手动配置升级，效率低且容易出错。	秒级升级。 通过更换镜像版本的方式，实现秒级升级。且CCE提供了滚动升级，使升级时业务不中断。
运维	运维成本高。 每给客户部署一套应用，就需要增加一台虚拟机的维护，随着客户量的增加，维护成本非常高。	自动化运维。 企业无需关注虚拟机的维护，只需要关注业务的开发。

21.2.1.2.2 改造流程

整体应用容器化改造时，一般需要执行如下流程。

图 21-2 容器化改造流程



21.2.1.2.3 分析应用

应用在容器化改造前，您需要了解自身应用的运行环境、依赖包等，并且熟悉应用的部署形态。需要了解的内容如表21-6。

表 21-6 了解应用环境

类别	子类	说明
运行环境	操作系统	应用需要运行在什么操作系统上，比如centos或者Ubuntu。 本例中，应用需要运行在centos:7.1操作系统上。
	运行环境	java应用需要jdk，go语言需要golang，web应用需要tomcat环境等，且需要确认对应版本号。 本例是tomcat类型的web应用，需要7.0版本的tomcat环境，且tomcat需要1.8版本的jdk。

类别	子类	说明
	依赖包	了解自己应用所需要的依赖包，类似openssl等系统软件，以及具体版本号。 本例不需要使用任何依赖包。
部署形态	周边配置	MongoDB：本例中MongoDB和Tomcat应用是在同一台机器中部署。因此对应配置可以固定，不需要将配置提取出来。

		应用需要对接哪些外部服务，例如数据库，文件存储等等。
		应用部署在虚拟机上时，该类配置需要每次部署时手动配置。容器化部署，可通过环境变量的方式注入到容器中，部署更为方便。
		本例需要对接MySQL数据库。您需要获取数据库的配置文件，如下“服务器地址”、“数据库名称”、“数据库登录用户名”和“数据库登录密码”将通过环境变量方式注入。
		url=jdbc:mysql://服务器地址/数据库名称 #数据库连接URL
		username=**** #数据库登录用户名
		password=**** #数据库登录密码

	自身配置	需要理出应用运行时的配置参数，哪些是需要经常变动的，哪些是不变的。 本例中，没有需要提取的自身配置项。 **说明** 为确保镜像无需经常更换，建议针对应用的各种配置进行分类。 • 经常变动的配置，例如周边对接信息、日志级别等，建议作为环境变量的方式来配置。 • 不变的配置，可以直接写到镜像中。

21.2.1.2.4 准备应用运行环境

在应用分析后，您已经了解到应用所需的操作系统、运行环境等。您需要准备好这些环境。

- **安装Docker**：应用容器化时，需要将应用构建为容器镜像。您需要准备一台机器，并安装Docker。
- **获取基础镜像版本名称**：根据应用运行的操作系统，确定基础镜像。本例应用运行在centos:7.1操作系统中，可以在“开源镜像中心”中获取到基础镜像。
- **获取运行环境**：获取运行应用的运行环境，以及对接的MongoDB数据库。

安装 Docker

Docker几乎支持在所有操作系统上安装，用户可以根据需要选择要安装的Docker版本。

说明

容器镜像服务支持使用Docker 1.11.2及以上版本上传镜像。

安装docker、构建镜像建议使用root用户进行操作，请提前获取待安装docker机器的root用户密码。

步骤1 以root用户登录待安装docker的机器。

步骤2 在Linux操作系统下，可以使用如下命令快速安装最新版本的Docker。如以下命令无法自动化安装，请根据操作系统进行手动安装，详细操作请参见**Docker Engine installation**。

curl -fsSL get.docker.com -o get-docker.sh

sh get-docker.sh

步骤3 执行以下命令，查看docker安装版本。

docker version

```
Client:
Version: 17.12.0-ce
API Version:1.35
……
```

Version字段表示版本号。

----结束

获取基础镜像版本名称

根据应用运行的操作系统，确定基础镜像。本例应用运行在centos:7.1操作系统中，可以在“开源镜像中心”中获取到基础镜像。

说明

此处请根据您应用实际使用的操作系统来进行搜索，主要目的是搜到镜像版本号。

步骤1 使用浏览器，登录docker官网。

步骤2 搜索centos，搜索到cenos7.1版本对应的镜像版本名为**centos7.1.1503**，后续编写dockerfile文件时需要用到该镜像名称。

图 21-3 获取 centos 版本名

----结束

获取运行环境

本例是tomcat类型的web应用，需要7.0版本的tomcat环境，tomcat需要1.8版本的jdk。并且应用对接MongoDB，均需要提前获取。

说明

此处请根据您应用的实际情况，下载应用所需的依赖环境。

步骤1 下载对应版本的Tomcat、JDK和MongoDB。

1. 下载JDK 1.8版本。
   下载地址：**https://www.oracle.com/java/technologies/jdk8-downloads.html**。
2. 下载Tomcat 7.0版本，链接为：**http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.82/bin/apache-tomcat-7.0.82.tar.gz**。
3. 下载MongoDB 3.2版本，链接为：**https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-3.2.9.tgz**。

步骤2 以root用户登录docker所在的机器。

步骤3 执行如下命令，新建用于存放该应用的目录。例如目录设为apptest。

mkdir apptest

cd apptest

步骤4 使用xShell工具，将已下载的依赖文件存放到apptest目录下。

步骤5 解压缩依赖文件。

tar -zxf apache-tomcat-7.0.82.tar.gz

tar -zxf jdk-8u151-linux-x64.tar.gz

tar -zxf mongodb-linux-x86_64-rhel70-3.2.9.tgz

步骤6 将企业应用（例如应用为apptest.war）放置到tomcat的webapps/apptest目录下。

说明

本例中的apptest.war为举例，请以贵公司应用进行实际操作。

mkdir -p apache-tomcat-7.0.82/webapps/apptest

cp apptest.war apache-tomcat-7.0.82/webapps/apptest

cd apache-tomcat-7.0.82/webapps/apptest

./../../../jdk1.8.0_151/bin/jar -xf apptest.war

rm -rf apptest.war

----结束

21.2.1.2.5 编写开机运行脚本

应用容器化时，一般需要准备开机运行的脚本，写作脚本的方式和写一般shell脚本相同。该脚本的主要目的包括：

- 启动应用所依赖的软件。
- 将需要修改的配置设置为环境变量。

说明

开机运行脚本与应用实际需求直接相关，每个应用所写的开机脚本会有所区别。请根据实际业务需求来写该脚本。

操作步骤

步骤1 以root用户登录docker所在的机器。

步骤2 执行如下命令，新建用于存放该应用的目录。

mkdir apptest

cd apptest

步骤3 编写脚本文件，脚本文件名称和内容会根据应用的不同而存在差别。此处仅为本例应用的指导，请根据实际应用来编写。

vi start_tomcat_and_mongo.sh

```
#!/bin/bash
# 加载系统环境变量
source  /etc/profile
# 启动mongodb，此处已写明数据存储路径为/usr/local/mongodb/data
./usr/local/mongodb/bin/mongod --dbpath=/usr/local/mongodb/data --logpath=/usr/local/mongodb/logs --port=27017 –fork
# 以下3条脚本，表示docker启动时将环境变量中MYSQL相关的内容写入配置文件中。
sed -i "s|mysql://.*/awcp_crmtile|mysql://$MYSQL_URL/$MYSQL_DB|g" /root/apache-tomcat-7.0.82/webapps/awcp/WEB-INF/classes/conf/jdbc.properties
sed -i "s|username=.*|username=$MYSQL_USER|g" /root/apache-tomcat-7.0.82/webapps/awcp/WEB-INF/classes/conf/jdbc.properties
sed -i "s|password=.*|password=$MYSQL_PASSWORD|g" /root/apache-tomcat-7.0.82/webapps/awcp/WEB-INF/classes/conf/jdbc.properties
# 启动tomcat
bash /root/apache-tomcat-7.0.82/bin/catalina.sh run
```

----结束

21.2.1.2.6 编写 Dockerfile 文件

镜像是容器的基础，容器基于镜像定义的内容来运行。镜像是多层存储，每一层是前一层基础上进行的修改。

定制镜像时，一般使用Dockerfile来完成。Dockerfile是一个文本文件，其内包含了一条条的指令，每一条指令构建镜像的其中一层，因此每一条指令的内容，就是描述该层应该如何构建。

本章节指导您如何编写dockerfile文件。

说明

Dockerfile文件编写与应用实际需求直接相关，每个应用所写的Dockerfile会有所区别，请根据业务实际需求来写Dockerfile文件。

操作步骤

步骤1 以root用户登录到安装有Docker的服务器上。

步骤2 编写Dockerfile文件。

vi Dockerfile

Dockerfile内容如下。

```
# 表示此镜像以centos:7.1.1503为基础镜像
FROM centos:7.1.1503
# 创建文件夹，存放数据和依赖文件，建议多个命令写成一条，可减少镜像大小
RUN mkdir -p /usr/local/mongodb/data \
 && mkdir -p /usr/local/mongodb/bin \
 && mkdir -p /root/apache-tomcat-7.0.82 \
 && mkdir -p /root/jdk1.8.0_151

# 将apache-tomcat-7.0.82目录下的文件拷贝到容器目录下
COPY ./apache-tomcat-7.0.82 /root/apache-tomcat-7.0.82
# 将jdk1.8.0_151目录下的文件拷贝到容器目录下
COPY ./jdk1.8.0_151 /root/jdk1.8.0_151
# 将mongodb-linux-x86_64-rhel70-3.2.9目录下的文件拷贝到容器目录下
COPY ./mongodb-linux-x86_64-rhel70-3.2.9/bin /usr/local/mongodb/bin
# 将start_tomcat_and_mongo.sh拷贝到容器/root/目录下
COPY ./start_tomcat_and_mongo.sh /root/

# 注入JAVA环境变量
RUN chown root:root -R /root \
 && echo "JAVA_HOME=/root/jdk1.8.0_151 " >> /etc/profile  \
 && echo "PATH=\$JAVA_HOME/bin:$PATH " >> /etc/profile  \
```

```
 && echo "CLASSPATH=.:\$JAVA_HOME/lib/dt.jar:\$JAVA_HOME/lib/tools.jar" >> /etc/profile  \
 && chmod +x /root \
 && chmod +x /root/start_tomcat_and_mongo.sh

# 容器启动的时候会自动运行start_tomcat_and_mongo.sh里面的命令，可以一条可以多条，也可以是一个脚本
ENTRYPOINT ["/root/start_tomcat_and_mongo.sh"]
```

其中：

- FROM语句：表示使用centos:7.1.1503镜像作为基础。
- RUN语句：表示在容器中执行某个shell命令。
- COPY语句：把本机中的文件拷贝到容器中。
- ENTRYPOINT语句：容器启动的命令。

----结束

21.2.1.2.7 制作并上传镜像

本章指导用户将整体应用制作成Docker镜像。制作完镜像后，每次应用的部署和升级即可通过镜像操作，减少了人工配置，提升效率。

说明

制作镜像时，要求制作镜像的文件在同个目录下。

使用云服务

容器镜像服务SWR：是一种支持容器镜像全生命周期管理的服务， 提供简单易用、安全可靠的镜像管理功能，帮助用户快速部署容器化服务。

基本概念

- 镜像：Docker镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。
- 容器：镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

操作步骤

步骤1 以root用户登录到安装有Docker的服务器上。

步骤2 进入apptest目录。

cd apptest

ll

此处必须确保制作镜像的文件均在同个目录下。

```
root@ecs-aos:~/apptest# ll
total 264456
drwxr-xr-x 5 root root      4096 Jan  2 19:59 ./
drwx------ 6 root root      4096 Jan  2 19:59 ../
drwxr-xr-x 9 root root      4096 Jan  2 19:55 apache-tomcat-7.0.82/
-rw-r--r-- 1 root root   8997403 Jan  2 19:52 apache-tomcat-7.0.82.tar.gz
-rw-r--r-- 1 root root       599 Jan  2 19:59 Dockerfile
drwxr-xr-x 8 uucp  143      4096 Sep  6 10:32 jdk1.8.0_151/
-rw-r--r-- 1 root root 189736377 Jan  2 19:54 jdk-8u151-linux-x64.tar.gz
drwxr-xr-x 3 root root      4096 Jan  2 19:55 mongodb-linux-x86_64-rhel70-3.2.9/
-rw-r--r-- 1 root root  72035914 Jan  2 19:53 mongodb-linux-x86_64-rhel70-3.2.9.tgz
-rw-r--r-- 1 root root       597 Jan  2 19:58 start_tomcat_and_mongo.sh
```

步骤3 构建镜像。

docker build -t apptest .

步骤4 上传镜像到容器镜像服务中。

----结束

21.2.1.2.8 创建容器工作负载

在本章节中，您将会把应用部署到CCE中。首次使用CCE时，您需要创建一个初始集群，并添加一个节点。

说明

应用镜像上传到容器镜像服务后，部署容器应用的方式都是基本类似的。不同点在于是否需要设置环境变量，是否需要使用云存储，这些也是和业务直接相关。

使用云服务

- 云容器引擎CCE：提供高可靠高性能的企业级容器应用管理服务，支持Kubernetes社区原生应用和工具，简化云上自动化容器运行环境搭建。
- 弹性云服务器ECS：一种可随时自助获取、可弹性伸缩的云服务器，帮助用户打造可靠、安全、灵活、高效的应用环境，确保服务持久稳定运行，提升运维效率。
- 虚拟私有云VPC：是用户在云上申请的隔离的、私密的虚拟网络环境。用户可以自由配置VPC内的IP地址段、子网、安全组等子服务，也可以申请弹性带宽和弹性IP搭建业务系统。

基本概念

- 集群：集群是计算资源的集合，包含一组节点资源，容器运行在节点上。在创建容器应用前，您需要存在一个可用集群。
- 节点：节点是指接入到平台的计算资源，包括虚拟机、物理机等。用户需确保节点资源充足，若节点资源不足，会导致创建应用等操作失败。
- 容器工作负载：容器工作负载指运行在CCE上的一组实例。CCE提供第三方应用托管功能，提供从部署到运维全生命周期管理。本节指导用户通过容器镜像创建您的第一个容器工作负载。

操作步骤

步骤1 创建集群前，您需要设置好如**表21-7**中的环境。

表 21-7 准备环境列表

序列	类别	操作步骤
1	创建虚拟私有云	您需要创建虚拟私有云，为CCE集群提供一个隔离的、用户自主配置和管理的虚拟网络环境。 若您已有虚拟私有云，可重复使用，无需多次创建。 1. 登录管理控制台。 2. 在服务列表中，选择“网络 > 虚拟私有云”。 3. 在“总览”界面，单击“创建虚拟私有云”。 4. 根据界面提示创建虚拟私有云。如无特殊需求，界面参数均可保持默认。
2	创建密钥对	您需要新建一个密钥对，用于远程登录节点时的身份认证。若您已有密钥对，可重复使用，无需多次创建。 1. 登录管理控制台。 2. 在服务列表中，选择“数据加密服务 DEW”。 3. 选择左侧导航中的“密钥对管理”，选择“私有密钥对”，单击“创建密钥对”。 4. 输入密钥对名称，勾选“我同意将密钥对私钥托管”和“我已经阅读并同意《密钥对管理服务免责声明》”，单击“确定”。 5. 在弹出的对话框中，单击“确定”。 请根据提示信息，查看并保存私钥。为保证安全，私钥只能下载一次，请妥善保管，否则将无法登录节点。

步骤2 创建集群和节点。

1. 登录CCE控制台。在“集群管理”页面选择需要创建的集群类型，单击“创建”。
   填写集群参数，选择**步骤1**中创建的VPC。
2. 购买节点，选择**步骤1**中创建的密钥对作为登录选项。

步骤3 部署工作负载到CCE。

1. 登录CCE控制台，进入集群，在左侧导航栏选择“工作负载”，单击右上角的“创建负载”。
2. 输入以下参数，其它保持默认。
   - 工作负载名称：apptest。
   - 实例数量：1。
3. 在“容器配置”中选择**制作并上传镜像**中上传的镜像。
4. 在“容器配置”中选择“环境变量”，添加环境变量，用于对接MySQL数据库。此处的环境变量由**开机运行脚本**中设置。

   说明

   本例对接了MySQL数据库，用环境变量的方式来对接。请根据您业务的实际情况，来决定是否需要使用环境变量。

表 21-8 配置环境变量

变量名称	变量/变量引用
MYSQL_DB	数据库名称。
MYSQL_URL	数据库部署的“IP:端口”。
MYSQL_USER	数据库用户名。
MYSQL_PASSWORD	数据库密码。

5. 在“容器配置”中选择“数据存储”，为实现数据的持久化存储，需要设置为云存储。

   说明

   本例使用了MongoDB数据库，并需要数据持久化存储，所以需要配置云存储。请根据您业务的实际情况，来决定是否需要使用云存储。

   此处挂载的路径，需要和docker开机运行脚本中的mongoDB存储路径相同，请参见**开机运行脚本**，本例中为**/usr/local/mongodb/data**。

6. 在“服务配置”中单击 + 添加服务，设置工作负载访问参数，设置完成后，单击“确定”。

   说明

   本例中，将应用设置为“通过弹性公网IP的方式”被外部互联网访问。

   - Service名称：输入应用发布的可被外部访问的名称，设置为：apptest。
   - 访问类型：选择“节点访问（NodePort）”。
   - 服务亲和：
     - 集群级别：集群下所有节点的IP+访问端口均可以访问到此服务关联的负载，服务访问会因路由跳转导致一定性能损失，且无法获取到客户端源IP。
     - 节点级别：只有通过负载所在节点的IP+访问端口才可以访问此服务关联的负载，服务访问没有因路由跳转导致的性能损失，且可以获取到客户端源IP。
   - 端口配置：
     - 协议：TCP。
     - 服务端口：访问Service的端口。
     - 容器端口：容器中应用启动监听的端口，该应用镜像请设置为：8080。
     - 节点端口：选择“自动生成”，系统会自动在当前集群下的所有节点上打开一个真实的端口号，映射到服务端口。

7. 单击“创建工作负载”。

   工作负载创建完成后，在工作负载列表中可查看到运行中的工作负载。

----结束

验证工作负载

工作负载创建完成后，可以通过访问工作负载验证部署是否成功。

在上面的部署中选择节点访问方式（NodePort），使用节点的“IP:端口”访问工作负载，如果能正常访问，则说明工作负载部署成功。

访问地址可以在工作负载详情页的访问方式页签下获取。

21.3 容灾

21.3.1 在 CCE 中实现应用高可用部署

基本原则

在CCE中，容器部署要实现高可用，可参考如下几点：

1. 集群选择3个控制节点的高可用模式。
2. 创建节点选择在不同的可用区，在多个可用区（AZ）多个节点的情况下，根据自身业务需求合理的配置自定义调度策略，可达到资源分配的最大化。
3. 创建多个节点池，不同节点池部署在不同可用区，通过节点池扩展节点。
4. 工作负载创建时设置实例数需大于2个。
5. 设置工作负载亲和性规则，尽量让Pod分布在不同可用区、不同节点上。

操作步骤

为了便于描述，假设集群中有4个节点，其可用区分布如下所示。

```
$ kubectl get node -L topology.kubernetes.io/zone,kubernetes.io/hostname
NAME           STATUS   ROLES    AGE   VERSION                 ZONE     HOSTNAME
192.168.5.112   Ready    <none>   42m   v1.21.7-r0-CCE21.11.1.B007   zone01   192.168.5.112
192.168.5.179   Ready    <none>   42m   v1.21.7-r0-CCE21.11.1.B007   zone01   192.168.5.179
192.168.5.252   Ready    <none>   37m   v1.21.7-r0-CCE21.11.1.B007   zone02   192.168.5.252
192.168.5.8     Ready    <none>   33h   v1.21.7-r0-CCE21.11.1.B007   zone03   192.168.5.8
```

按如下定义创建负载。这里定义了两条工作负载反亲和规则podAntiAffinity。

- 第一条在可用区下工作负载反亲和，参数设置如下。
  - 权重weight：权重值越高会被优先调度，本示例设置为50。
  - 拓扑域topologyKey：包含默认和自定义标签，用于指定调度时的作用域。本示例设置为topology.kubernetes.io/zone，此为节点上标识节点在哪个可用区的标签。
  - 标签选择labelSelector：选择Pod的标签，与工作负载本身反亲和。
- 第二条在节点名称作用域下工作负载反亲和，参数设置如下。
  - 权重weight：设置为50。
  - 拓扑域topologyKey：设置为kubernetes.io/hostname。
  - 标签选择labelSelector：选择Pod的标签，与工作负载本身反亲和。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 50
              podAffinityTerm:
                labelSelector:                    # 选择Pod的标签，与工作负载本身反亲和。
                  matchExpressions:
                    - key: app
                      operator: In
                      values:
                        - nginx
                namespaces:
                  - default
                topologyKey: topology.kubernetes.io/zone   # 在同一个可用区下起作用
            - weight: 50
              podAffinityTerm:
                labelSelector:                    # 选择Pod的标签，与工作负载本身反亲和
                  matchExpressions:
                    - key: app
                      operator: In
                      values:
                        - nginx
                namespaces:
                  - default
                topologyKey: kubernetes.io/hostname     # 在节点上起作用
      imagePullSecrets:
        - name: default-secret
```

创建工作负载，然后查看Pod所在的节点。

```
$ kubectl get pod -owide
NAME                     READY   STATUS    RESTARTS   AGE   IP           NODE
nginx-6fffd8d664-dpwbk   1/1     Running   0          17s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-qhclc   1/1     Running   0          17s   10.0.1.133   192.168.5.252
```

将Pod数量增加到3，可以看到Pod被调度到了另外一个节点，且这个当前这3个节点是在3个不同可用区。

```
$ kubectl scale --replicas=3 deploy/nginx
deployment.apps/nginx scaled
$ kubectl get pod -owide
NAME                     READY   STATUS    RESTARTS   AGE     IP           NODE
nginx-6fffd8d664-8t7rv   1/1     Running   0          3s      10.0.0.9     192.168.5.8
nginx-6fffd8d664-dpwbk   1/1     Running   0          2m45s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-qhclc   1/1     Running   0          2m45s   10.0.1.133   192.168.5.252
```

将Pod数量增加到4，可以看到Pod被调度到了最后一个节点。可见根据工作负载反亲和规则，可以将Pod按照可用区和节点较为均匀的分布，更为可靠。

```
$ kubectl scale --replicas=4 deploy/nginx
deployment.apps/nginx scaled
$ kubectl get pod -owide
NAME                     READY   STATUS    RESTARTS   AGE     IP            NODE
nginx-6fffd8d664-8t7rv   1/1     Running   0          2m30s   10.0.0.9      192.168.5.8
nginx-6fffd8d664-dpwbk   1/1     Running   0          5m12s   10.0.0.132    192.168.5.112
nginx-6fffd8d664-h796b   1/1     Running   0          78s     10.0.1.5      192.168.5.179
nginx-6fffd8d664-qhclc   1/1     Running   0          5m12s   10.0.1.133    192.168.5.252
```

21.4 安全

21.4.1 集群安全配置

从安全的角度，建议您对集群做如下配置。

使用最新版本的 CCE 集群

Kubernetes社区一般4个月左右发布一个大版本，CCE的版本发布频率跟随社区版本发布节奏，在社区发布Kubernetes版本后3个月左右同步发布新的CCE版本，例如Kubernetes v1.19于2020年9月发布后，CCE于2021年3月左右发布CCE v1.19版本。

最新版本的集群修复了已知的漏洞或者拥有更完善的安全防护机制，新建集群时推荐选择使用最新版本的集群。在集群版本停止提供服务前，请及时升级到新版本。

关闭 default 的 serviceaccount 的 token 自动挂载功能

kubernetes默认会给每个工作负载实例关联default服务账号，即在容器内挂载一个token，该token能够通过kube-apiserver和kubelet组件的认证。在没有开启RBAC的集群，得到该token相当于是得到了整个CCE集群的控制权。在开启RBAC的集群，该token所拥有的权限，取决于环境管理员给这个服务账号关联了什么角色。该服务账号的token一般是给需要访问kube-apiserver的容器使用，如CoreDNS、autoscaler、prometheus等。对于不需要访问kube-apiserver的工作负载，建议关闭服务账号的自动关联功能。

禁用方法:

- 方法一：将服务账号的automountServiceAccountToken字段设置为false。完成设置后，创建的工作负载将不会默认关联default服务账号。注意：每个命名空间都要按需设置。

  ```
  apiVersion: v1
  kind: ServiceAccount
  metadata:
    name: default
  automountServiceAccountToken: false
  ...
  ```

  当工作负载需要关联服务账号时，在工作负载的yaml描述文件中显式地指定。

  ```
  ...
  spec:
    template:
      spec:
        serviceAccountName: default
        automountServiceAccountToken: true
        ...
  ```

- 方法二：显式地关闭工作负载自动关联服务账号的功能。

  ```
  ...
  spec:
    template:
  ```

```
  spec:
    automountServiceAccountToken: false
    ...
```

合理配置用户的集群访问权限

CCE支持账号创建多个IAM用户。通过创建不同的用户组，并授予不同用户组不同的访问权限，然后在创建用户时将用户加入对应权限的用户组中，即可完成控制不同用户具备不同的区域（region）、是否只读的权限。同时也支持为用户或者用户组配置命名空间级别的权限。考虑到安全，建议最小化用户的访问权限。

如果主账号下需要配置多个IAM用户，应合理配置子用户和命名空间的权限。

配置集群命名空间资源配额限制

应限制每个命名空间能够分配的资源总量，控制的资源包括：CPU、内存、存储、pods、services、deployments、statefulsets等。合理配置命名空间的可分配资源总量，能够防止某个命名空间创建过多的资源影响整个集群的稳定性。

配置命名空间下容器的 Limit ranges

通过资源配额，集群管理员可以以命名空间为单位，限制其资源的使用与创建。 在命名空间中，一个 Pod 或 Container 最多能够使用命名空间的资源配额所定义的 CPU 和内存用量，这样一个 Pod 或 Container 可能会垄断该命名空间下所有可用的资源。建议配置LimitRange 在命名空间内限制资源分配。limitrange可以做到如下限制：

- 在一个命名空间中实施对每个 Pod 或 Container 最小和最大的资源使用量的限制。

  例如为一个命名空间的pod创建最大最小CPU使用限制：

  cpu-constraints.yaml

  ```
  apiVersion: v1
  kind: LimitRange
  metadata:
    name: cpu-min-max-demo-lr
  spec:
    limits:
    - max:
        cpu: "800m"
      min:
        cpu: "200m"
      type: Container
  ```

  然后使用**kubectl -n** *<namespace>* **create -f** *cpu-constraints.yaml*完成创建。注意，如果没有指定容器使用cpu的默认值，平台会自动配置CPU使用的默认值，即创建完成后自动添加default配置：

  ```
  ...
  spec:
    limits:
    - default:
        cpu: 800m
      defaultRequest:
        cpu: 800m
      max:
        cpu: 800m
      min:
        cpu: 200m
      type: Container
  ```

- 在一个命名空间中实施对每个 PersistentVolumeClaim 能申请的最小和最大的存储空间大小的限制。

storagelimit.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimit
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 2Gi
    min:
      storage: 1Gi
```

然后使用**kubectl -n** *<namespace>* **create -f** *storagelimit.yaml*完成创建。

配置集群内的网络隔离

- 容器隧道网络

  针对集群内命名空间之间以及同一命名空间下工作负载之间需要网络隔离的场景，可以通过配置NetworkPolicy来达到隔离的效果。

- VPC网络

  暂不支持网络隔离。

kubelet 开启 Webhook 鉴权模式

须知

v1.15.6-r1及之前版本的CCE集群涉及。v1.15.6-r1之后的版本不涉及。

将CCE集群版本升级至1.13或1.15版本，并开启集群RBAC能力，如果版本已经是1.13或以上版本，则无需升级。

创建节点时可通过postInstall文件注入的方式开启kubelet的鉴权模式（设置kubelet的启动参数：--authorization-node=Webhook），步骤如下：

步骤1 创建clusterrolebinding，执行命令：

kubectl create clusterrolebinding kube-apiserver-kubelet-admin --clusterrole=system:kubelet-api-admin --user=system:kube-apiserver

步骤2 已创建的节点，需要登录到节点更改kubelet的鉴权模式，更改节点上/var/paas/kubernetes/kubelet/kubelet_config.yaml里的authorization mode为Webhook，然后重启kubelet，执行如下命令：

sed -i s/AlwaysAllow/Webhook/g /var/paas/kubernetes/kubelet/kubelet_config.yaml; systemctl restart kubelet

步骤3 新创建的节点，在创建节点的安装后执行脚本里加入以下命令去后置修改kubelet的权限模式：

sed -i s/AlwaysAllow/Webhook/g /var/paas/kubernetes/kubelet/kubelet_config.yaml; systemctl restart kubelet

----结束

使用完成后及时卸载 webterminal 插件

web-terminal插件能够对CCE集群进行管理，请用户妥善保管好登录密码，避免密码泄漏造成损失。使用完成后及时卸载插件。

21.4.2 节点安全配置

节点不暴露到公网

- 如非必需，节点不建议绑定EIP，以减少攻击面。
- 在必须使用EIP的情况下，应通过合理配置防火墙或者安全组规则，限制非必须的端口和IP访问。

在使用cce集群过程中，由于业务场景需要，在节点上配置了kubeconfig.json文件，kubectl使用该文件中的证书和私钥信息可以控制整个集群。在不需要时，请清理节点上的/root/.kube目录下的目录文件，防止被恶意用户利用：

```
rm -rf /root/.kube
```

加固 VPC 安全组规则

CCE作为通用的容器平台，安全组规则的设置适用于通用场景。用户可根据安全需求，通过**网络控制台**的**安全组**找到CCE集群对应的安全组规则进行安全加固。

节点应按需进行加固

CCE服务的集群节点操作系统配置与开源操作系统默认配置保持一致，用户在节点创建完成后应根据自身安全诉求进行安全加固。

CCE提供以下建议的加固方法：

- 通过“创建节点”的“安装后执行脚本”功能，在节点创建完成后，执行命令加固节点。具体操作步骤参考创建节点的“**云服务器高级设置**”的“**安装后执行脚本**”。“安装后执行脚本”的内容需由用户提供。

禁止容器获取宿主机元数据

当用户将单个CCE集群作为共享集群，提供给多个用户来部署容器时，应限制容器访问openstack的管理地址（169.254.169.254），以防止容器获取宿主机的元数据。

修复方式参考ECS文档-元数据获取-使用须知。

⚠ 警告

该修复方案可能影响通过ECS Console修改密码，修复前须进行验证。

步骤1 获取集群的网络模式和容器网段信息。

在CCE的“集群信息”界面的“网络信息”部分查看集群的网络模式和容器网段。

步骤2 禁止容器获取宿主机元数据。

- VPC网络集群

a. 以root用户登录CCE集群的每一个node节点，执行以下命令：
```
iptables -I OUTPUT -s {container_cidr} -d 169.254.169.254 -j REJECT
```
其中，{container_cidr}是集群的容器网络，如10.0.0.0/16

为保证配置持久化，建议将该命令写入/etc/rc.local 启动脚本中

b. 在容器中执行如下命令访问openstack的userdata和metadata接口，验证请求是否被拦截
```
curl 169.254.169.254/openstack/latest/meta_data.json
curl 169.254.169.254/openstack/latest/user_data
```

- 容器隧道网络集群

a. 以root用户登录CCE集群的每一个node节点，执行以下命令：
```
iptables -I FORWARD -s {container_cidr} -d 169.254.169.254 -j REJECT
```
其中，{container_cidr}是集群的容器网络，如10.0.0.0/16

为保证配置持久化，建议将该命令写入/etc/rc.local 启动脚本中

b. 在容器中执行如下命令访问openstack的userdata和metadata接口，验证请求是否被拦截
```
curl 169.254.169.254/openstack/latest/meta_data.json
curl 169.254.169.254/openstack/latest/user_data
```

----结束

21.4.3 容器安全配置

控制 Pod 调度范围

通过nodeSelector或者nodeAffinity限定应用所能调度的节点范围，防止单个应用异常威胁到整个集群。

容器安全配置建议

- 通过设置容器的计算资源限制（request和limit），避免容器占用大量资源影响宿主机和同节点其他容器的稳定性
- 如非必须，不建议将宿主机的敏感目录挂载到容器中，如/、/boot、/dev、/etc、/lib、/proc、/sys、/usr等目录
- 如非必须，不建议在容器中运行sshd进程
- 如非必须，不建议容器与宿主机共享网络命名空间
- 如非必须，不建议容器与宿主机共享进程命名空间
- 如非必须，不建议容器与宿主机共享IPC命名空间
- 如非必须，不建议容器与宿主机共享UTS命名空间
- 如非必须，不建议将docker的sock文件挂载到任何容器中

容器的权限访问控制

使用容器应用时，遵循权限最小化原则，合理设置Deployment/Statefulset的securityContext：

- 通过配置runAsUser，指定容器使用非root用户运行。
- 通过配置privileged，在不需要特权的场景不建议使用特权容器。
- 通过配置capabilities，使用capability精确控制容器的特权访问权限。

- 通过配置allowPrivilegeEscalation, 在不需要容器进程提权的场景，建议关闭“允许特权逃逸”的配置。
- 通过配置安全计算模式seccomp，限制容器的系统调用权限，具体配置方法可参考社区官方资料**使用 Seccomp 限制容器的系统调用**。
- 通过配置ReadOnlyRootFilesystem的配置，保护容器根文件系统。

  如deployment配置如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: security-context-example
  namespace: security-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: security-context-example
      label: security-context-example
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      annotations:
        seccomp.security.alpha.kubernetes.io/pod: runtime/default
      labels:
        app: security-context-example
        label: security-context-example
    spec:
      containers:
        - image: ...
          imagePullPolicy: Always
          name: security-context-example
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            runAsUser: 1000
            capabilities:
              add:
              - NET_BIND_SERVICE
              drop:
              - all
          volumeMounts:
            - mountPath: /etc/localtime
              name: localtime
              readOnly: true
            - mountPath: /opt/write-file-dir
              name: tmpfs-example-001
      securityContext:
        seccompProfile:
          type: RuntimeDefault
      volumes:
        - hostPath:
            path: /etc/localtime
            type: ""
          name: localtime
        - emptyDir: {}
          name: tmpfs-example-001
```

限制业务容器访问管理面

在节点上的业务容器无需访问kubernetes时，可以通过以下方式禁止节点上的容器网络流量访问到kube-apiserver。

步骤1 查询容器网段和内网apiserver地址。

在CCE的“集群管理”界面查看集群的容器网段和内网apiserver地址。

步骤2 以root用户登录CCE集群的每一个Node节点，执行以下命令：

- VPC网络：
  ```
  iptables -I OUTPUT -s {container_cidr} -d {内网apiserver的IP} -j REJECT
  ```
- 容器隧道网络：
  ```
  iptables -I FORWARD -s {container_cidr} -d {内网apiserver的IP} -j REJECT
  ```

其中，{container_cidr}是集群的容器网络，如10.0.0.0/16。

为保证配置持久化，建议将该命令写入/etc/rc.local 启动脚本中。

步骤3 在容器中执行如下命令访问kube-apiserver接口，验证请求是否被拦截。

```
curl -k https://{内网apiserver的IP}:5443
```

----结束

21.4.4 密钥 Secret 安全配置

当前CCE已为secret资源配置了静态加密，用户创建的secret在CCE的集群的etcd里会被加密存储。当前secret主要有环境变量和文件挂载两种使用方式。不论使用哪种方式，CCE传递给用户的仍然是用户配置时的数据。因此建议：

1. 用户不应在日志中对相关敏感信息进行记录；
2. 通过文件挂载的方式secret时，默认在容器内映射的文件权限为0644，建议为其配置更严格的权限，例如：

   ```
   apiversion: v1
   kind: Pod
   metadata:
     name: mypod
   spec:
     containers:
     - name: mypod
       image: redis
       volumeMounts:
       - name: foo
         mountPath: "/etc/foo"
     volumes:
     - name: foo
       secret:
         secretName: mysecret
         defaultMode: 256
   ```

   其中“defaultMode： 256”，256为10进制，对应八进制的0400权限。
3. 使用文件挂载的方式时，通过配置secret的文件名实现文件在容器中“隐藏”的效果：

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: dotfile-secret
   data:
     .secret-file: dmFsdWUtMg0KDQo=
   ---
   apiVersion: v1
   kind: Pod
   metadata;
     name: secret-dotfiles-pod
   spec:
     volumes:
     - name: secret-volume
   ```

```
  secret:
    secretName: dotfile-secret
containers:
- name: dotfile-test-container
  image: k8s.gcr.io/busybox
  command:
  - ls
  - "-1"
  - "/etc/secret-volume"
  volumeMounts:
  - name: secret-volume
    readOnly: true
    mountPath: "/etc/secret-volume"
```

这样.secret-file目录在/etc/secret-volume/路径下通过“ls -l”查看不到，但可以通过“ls -al”查看到。

4. 用户应在创建secret前自行加密敏感信息，使用时解密。

使用 Bound ServiceAccount Token 访问集群

基于Secret的ServiceAccount Token由于token不支持设置过期时间、不支持自动刷新，并且由于存放在secret中，pod被删除后token仍然存在secret中，一旦泄露可能导致安全风险。1.23版本以及以上版本CCE集群推荐使用Bound Servcie Account Token，该方式支持设置过期时间，并且和pod生命周期一致，可减少凭据泄露风险。例如：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: security-token-example
  namespace: security-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: security-token-example
      label: security-token-example
  template:
    metadata:
      annotations:
        seccomp.security.alpha.kubernetes.io/pod: runtime/default
      labels:
        app: security-token-example
        label: security-token-example
    spec:
      serviceAccountName: test-sa
      containers:
        - image: ...
          imagePullPolicy: Always
          name: security-token-example
      volumes:
        - name: test-projected
          projected:
            defaultMode: 420
            sources:
              - serviceAccountToken:
                  expirationSeconds: 1800
                  path: token
              - configMap:
                  items:
                    - key: ca.crt
                      path: ca.crt
                  name: kube-root-ca.crt
              - downwardAPI:
                  items:
                    - fieldRef:
                        apiVersion: v1
```

```
          fieldPath: metadata.namespace
        path: namespace
```

具体可参考：https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/

21.5 弹性伸缩

21.5.1 使用 HPA+CA 实现工作负载和节点联动弹性伸缩

应用场景

企业应用的流量大小不是每时每刻都一样，有高峰，有低谷，如果每时每刻都要保持能够扛住高峰流量的机器数目，那么成本会很高。通常解决这个问题的办法就是根据流量大小或资源占用率自动调节机器的数量，也就是弹性伸缩。

当使用Pod/容器部署应用时，通常会设置容器的申请/限制值来确定可使用的资源上限，以避免在流量高峰期无限制地占用节点资源。然而，这种方法可能会存在资源瓶颈，达到资源使用上限后可能会导致应用出现异常。为了解决这个问题，可以通过伸缩Pod的数量来分摊每个应用实例的压力。如果增加Pod数量后，节点资源使用率上升到一定程度，继续扩容出来的Pod无法调度，则可以根据节点资源使用率继续伸缩节点数量。

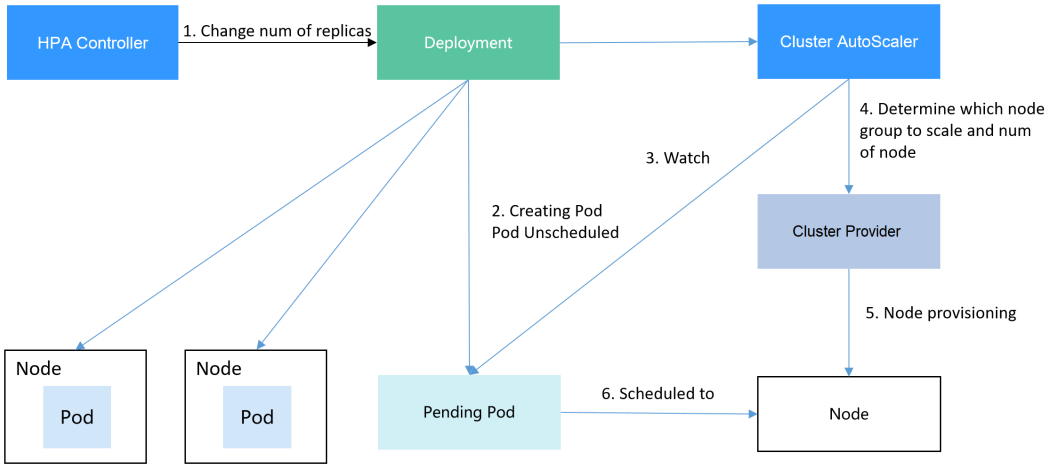
解决方案

CCE中弹性伸缩最主要的就是使用HPA（Horizontal Pod Autoscaling）和CA（Cluster AutoScaling）两种弹性伸缩策略，HPA负责工作负载弹性伸缩，也就是应用层面的弹性伸缩，CA负责节点弹性伸缩，也就是资源层面的弹性伸缩。

通常情况下，两者需要配合使用，因为HPA需要集群有足够的资源才能扩容成功，当集群资源不够时需要CA扩容节点，使得集群有足够资源；而当HPA缩容后集群会有大量空余资源，这时需要CA缩容节点释放资源，才不至于造成浪费。

如图21-4所示，HPA根据监控指标进行扩容，当集群资源不够时，新创建的Pod会处于Pending状态，CA会检查所有Pending状态的Pod，根据用户配置的扩缩容策略，选择出一个最合适的节点池，在这个节点池扩容。

图 21-4 HPA + CA 工作流程



使用HPA+CA可以很容易做到弹性伸缩，且节点和Pod的伸缩过程可以非常方便的观察到，使用HPA+CA做弹性伸缩能够满足大部分业务场景需求。

本文将通过一个示例介绍HPA+CA两种策略配合使用下弹性伸缩的过程，从而帮助您更好的理解和使用弹性伸缩。

准备工作

步骤1 创建一个有1个节点的集群，节点规格为2U4G及以上，并在创建节点时为节点添加弹性公网IP，以便从外部访问。如创建节点时未绑定弹性公网IP，您也可以前往ECS控制台为该节点进行手动绑定。

步骤2 给集群安装插件。

- autoscaler：节点伸缩插件。
- metrics-server：是Kubernetes集群范围资源使用数据的聚合器，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据。

步骤3 登录集群节点，准备一个算力密集型的应用。当用户请求时，需要先计算出结果后才返回给用户结果，如下所示。

1. 创建一个名为index.php的PHP文件，文件内容是在用户请求时先循环开方1000000次，然后再返回“OK!”。
   ```
   vi index.php
   ```
   文件内容如下：
   ```
   <?php
    $x = 0.0001;
    for ($i = 0; $i <= 1000000; $i++) {
      $x += sqrt($x);
    }
    echo "OK!";
   ?>
   ```
2. 编写Dockerfile制作镜像。
   ```
   vi Dockerfile
   ```
   Dockerfile内容如下：
   ```
   FROM php:5-apache
   COPY index.php /var/www/html/index.php
   RUN chmod a+rx index.php
   ```
3. 执行如下命令构建镜像，镜像名称为hpa-example，版本为latest。
   ```
   docker build -t hpa-example:latest .
   ```
4. （可选）登录SWR管理控制台，选择左侧导航栏的“组织管理”，单击页面右上角的“创建组织”，创建一个组织。

   如已有组织可跳过此步骤。
5. 在左侧导航栏选择“我的镜像”，单击右侧“客户端上传”，在弹出的页面中单击“生成临时登录指令”，单击 复制登录指令。
6. 在集群节点上执行上一步复制的登录指令，登录成功会显示“Login Succeeded”。
7. 为hpa-example镜像添加标签。

   docker tag [镜像名称1:版本名称1] [镜像仓库地址]/[组织名称]/[镜像名称2:版本名称2]

   - **[镜像名称1:版本名称1]**：请替换为您本地所要上传的实际镜像的名称和版本名称。
   - **[镜像仓库地址]**：可在SWR控制台上查询，**登录指令**中末尾的域名即为镜像仓库地址。

- **[组织名称]**：请替换为已创建的组织名称。
- **[镜像名称2:版本名称2]**：请替换为SWR镜像仓库中需要显示的镜像名称和镜像版本。

示例：

docker tag hpa-example:latest {*Image repository address*}/group/hpa-example:latest

8. 上传镜像至镜像仓库。

   docker push [镜像仓库地址]/[组织名称]/[镜像名称2:版本名称2]

   示例：

   docker push {*Image repository address*}/group/hpa-example:latest

   终端显示如下信息，表明上传镜像成功。

   ```
   6d6b9812c8ae: Pushed
   ...
   fe4c16cbf7a4: Pushed
   latest: digest: sha256:eb7e3bbd*** size: **
   ```

   返回容器镜像服务控制台，在“我的镜像”页面，执行刷新操作后可查看到对应的镜像信息。

----结束

创建节点池和节点伸缩策略

步骤1 登录CCE控制台，进入已创建的集群，在左侧单击“节点管理”，选择“节点池”页签并单击右上角“创建节点池”。

步骤2 填写节点池配置，添加2U4G的节点，并打开弹性扩缩容开关。

- 节点数量：设置为1，表示创建节点池时默认创建的节点数为1。
- 弹性伸缩：开启，表示节点池将根据集群负载情况自动创建或删除节点池内的节点。
- 节点数上限：设置为5，表示节点池中节点数的最大值。
- 节点规格：2核 | 4GiB

其余参数设置可使用默认值。

步骤3 在集群控制台左侧单击“插件管理”，单击autoscaler插件下的“编辑”按钮，修改autoscaler插件配置，将自动缩容开关打开，并配置缩容相关参数。例如节点资源使用率小于50%时进行缩容扫描，启动缩容。

上面配置的节点池弹性伸缩，会根据Pod的Pending状态进行扩容，根据节点的资源使用率进行缩容。

步骤4 在集群控制台左侧单击“节点伸缩”，单击页面右上角的“创建节点伸缩策略”。这里的节点伸缩策略可以根据CPU/内存分配率扩容、还可以按照时间定期扩容节点数量。

节点伸缩示例如下，这里配置当集群CPU分配率大于70%时，增加一个节点。CA策略需要关联节点池，可以关联多个节点池，当需要对节点扩缩容时，在节点池中根据最小浪费规则挑选合适规格的节点扩缩容。

----结束

创建工作负载

使用构建的hpa-example镜像创建无状态工作负载，副本数为1，镜像地址与上传到SWR仓库的组织有关，需要替换为实际取值。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hpa-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hpa-example
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
      - name: container-1
        image: 'hpa-example:latest' # 替换为您上传到SWR的镜像地址
        resources:
          limits:                 # limits与requests建议取值保持一致，避免扩缩容过程中出现震荡
            cpu: 500m
            memory: 200Mi
          requests:
            cpu: 500m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

然后再为这个负载创建一个Nodeport类型的Service，以便能从外部访问。

```
kind: Service
apiVersion: v1
metadata:
  name: hpa-example
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 31144
  selector:
    app: hpa-example
  type: NodePort
```

创建 HPA 策略

创建HPA策略，如下所示，该策略关联了名为hpa-example的负载，期望CPU使用率为50%。

另外有两条注解annotations，一条是CPU的阈值范围，最低30，最高70，表示CPU使用率在30%到70%之间时，不会扩缩容，防止小幅度波动造成影响。另一条是扩缩容时间窗，表示策略成功触发后，在缩容/扩容冷却时间内，不会再次触发缩容/扩容，以防止短期波动造成影响。

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-policy
  annotations:
    extendedhpa.metrics: '[{"type":"Resource","name":"cpu","targetType":"Utilization","targetRange":{"low":"30","high":"70"}}]'
    extendedhpa.option: '{"downscaleWindow":"5m","upscaleWindow":"3m"}'
```

```
spec:
  scaleTargetRef:
    kind: Deployment
    name: hpa-example
    apiVersion: apps/v1
  minReplicas: 1
  maxReplicas: 100
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

观察弹性伸缩过程

步骤1 首先查看集群节点情况，如下所示，有两个节点。

```
# kubectl get node
NAME            STATUS   ROLES    AGE     VERSION
192.168.0.183   Ready    <none>   2m20s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   55m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

查看HPA策略，可以看到目标负载的指标（CPU使用率）为0%

```
# kubectl get hpa hpa-policy
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%    1         100       1          4m
```

步骤2 通过如下命令访问负载，如下所示，其中{ip:port}为负载的访问地址，可以在负载的详情页中查询。

while true;do wget -q -O- http://*{ip:port}*; done

说明

如果此处不显示公网IP地址，则说明集群节点没有弹性公网IP，请创建弹性公网IP并绑定到节点，创建完后需要同步节点信息。

观察负载的伸缩过程。

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          4m
hpa-policy   Deployment/hpa-example   190%/50%   1         100       1          4m23s
hpa-policy   Deployment/hpa-example   190%/50%   1         100       4          4m31s
hpa-policy   Deployment/hpa-example   200%/50%   1         100       4          5m16s
hpa-policy   Deployment/hpa-example   200%/50%   1         100       4          6m16s
hpa-policy   Deployment/hpa-example   85%/50%    1         100       4          7m16s
hpa-policy   Deployment/hpa-example   81%/50%    1         100       4          8m16s
hpa-policy   Deployment/hpa-example   81%/50%    1         100       7          8m31s
hpa-policy   Deployment/hpa-example   57%/50%    1         100       7          9m16s
hpa-policy   Deployment/hpa-example   51%/50%    1         100       7          10m
hpa-policy   Deployment/hpa-example   58%/50%    1         100       7          11m
```

可以看到4m23s时负载的CPU使用率为190%，超过了目标值，此时触发了负载弹性伸缩，将负载扩容为4个副本/Pod，随后的几分钟内，CPU使用并未下降，直到到7m16s时CPU使用率才开始下降，这是因为新创建的Pod并不一定创建成功，可能是因为资源不足Pod处于Pending状态，这段时间内在扩容节点。

到7m16s时CPU使用率开始下降，说明Pod创建成功，开始分担请求流量，到8分钟时下降到81%，还是高于目标值，且高于70%，说明还会再次扩容，到9m16s时再次扩容到7个Pod，这时CPU使用率降为51%，在30%-70%的范围内，不会再次伸缩，可以观察到此后Pod数量一直稳定在7个。

观察负载和HPA策略的详情，从事件中可以看到负载的扩容的过程和策略生效的时间，如下所示。

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 7
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization (percentage of request) above target
```

此时查看节点数量，发现节点多了两个，也就是在刚才过程中节点扩容了两个。

```
# kubectl get node
NAME          STATUS   ROLES    AGE     VERSION
192.168.0.120   Ready    <none>   3m5s    v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.136   Ready    <none>   6m58s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.183   Ready    <none>   18m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   71m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

在控制台也可以看到伸缩历史，这里可以看到CA策略执行了一次，当集群中CPU分配率大于70%，将节点池中节点数量从2扩容到3。另一个节点是autoscaler默认的根据Pod的Pending状态扩容而来，在HPA初期。

这里节点扩容过程具体是这样：

1. Pod数量变为4后，由于没有资源，Pod处于Pending状态，触发了autoscaler默认的扩容策略，将节点数增加一个。
2. 第二次节点扩容是因为集群中CPU分配率大于70%，触发了CA策略，从而将节点数增加一个，从控制台上伸缩历史可以看出来。根据分配率扩容，可以保证集群一直处于资源充足的状态。

步骤3 停止访问负载，观察负载Pod数量。

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   50%/50%    1         100       7          12m
hpa-policy   Deployment/hpa-example   21%/50%    1         100       7          13m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          14m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          23m
```

可以看到从13m开始CPU使用率为21%，18m时Pod数量缩为3个，到23m时Pod数量缩为1个。

观察负载和HPA策略的详情，从事件中可以看到负载的扩容的过程和策略生效的时间，如下所示。

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 7
  Normal  ScalingReplicaSet  6m28s  deployment-controller  Scaled down replica set hpa-
example-79dd795485 to 3
  Normal  ScalingReplicaSet  72s    deployment-controller  Scaled down replica set hpa-
example-79dd795485 to 1
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  6m45s  horizontal-pod-autoscaler  New size: 3; reason: All metrics below target
  Normal  SuccessfulRescale  90s    horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

在控制台同样可以看到HPA策略生效历史，再继续等待，会看到节点也会被缩容一个。

这里为何没有被缩容掉两个节点，是因为节点池中这两个节点都存在kube-system namespace下的Pod（且不是DaemonSets创建的Pod）。

----结束

总结

通过上述内容可以看到，使用HPA+CA可以很容易做到弹性伸缩，且节点和Pod的伸缩过程可以非常方便的观察到，使用HPA+CA做弹性伸缩能够满足大部分业务场景需求。

21.6 监控

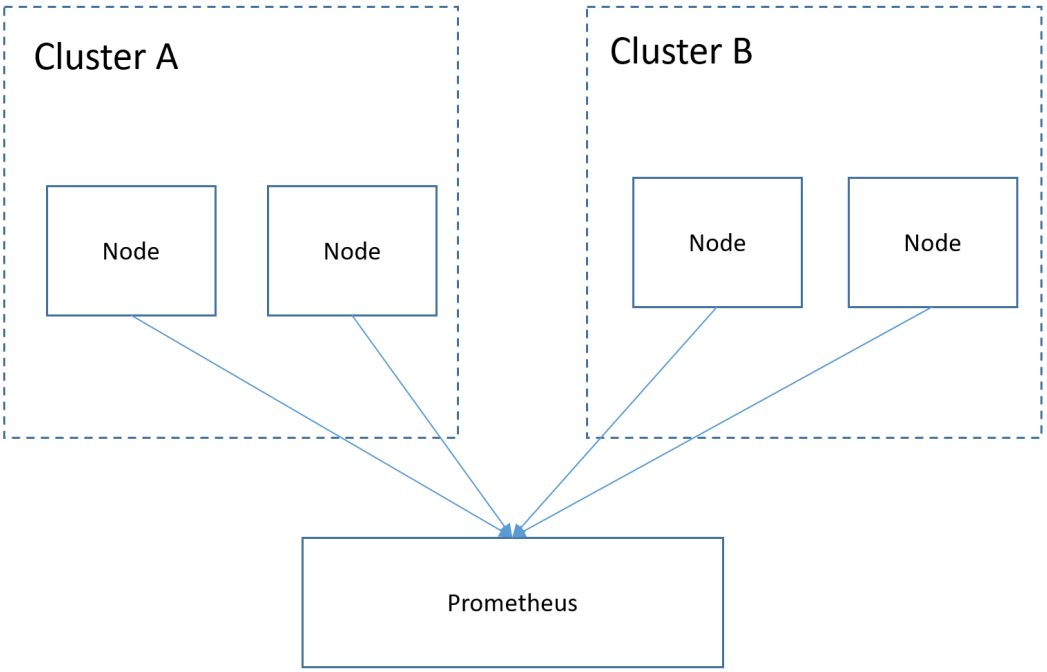
21.6.1 Prometheus 监控多个集群

应用场景

通常情况下，用户的集群数量不止一个，例如生产集群、测试集群、开发集群等。如果在每个集群安装Prometheus监控集群里的业务各项指标的话，很大程度上提高了维护成本和资源成本，同时数据也不方便汇聚到一块查看，这时候可以通过部署一套Prometheus，对接监控多个集群的指标信息。

方案架构

将多个集群对接到同一个Prometheus监控系统，如下所示，节约维护成本和资源成本，且方便汇聚监控信息。



前提条件

- 目标集群已创建。
- Prometheus与目标集群之间网络保持连通。
- 已在一台Linux主机中使用二进制文件安装Prometheus，详情请参见**Installation**。

操作步骤

步骤1 分别获取目标集群的bearer_token 信息。

1. 在目标集群创建rbac权限。

   登录到目标集群后台节点，创建prometheus_rbac.yaml文件。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus-test
  namespace: kube-system

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-test
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  - services
  - endpoints
  - pods
  - nodes/proxy
  verbs:
  - get
  - list
  - watch
```

```
- apiGroups:
  - "extensions"
  resources:
    - ingresses
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - configmaps
  - nodes/metrics
  verbs:
  - get
- nonResourceURLs:
  - /metrics
  verbs:
  - get
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus-test
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus-test
subjects:
- kind: ServiceAccount
  name: prometheus-test
  namespace: kube-system
```

执行以下命令创建rbac权限。

kubectl apply -f prometheus_rbac.yaml

2. 获取目标集群bearer_token信息。

说明

- 1.21以前版本的集群中，Pod中获取Token的形式是通过挂载ServiceAccount的Secret来获取Token，这种方式获得的Token是永久的。该方式在1.21及以上的版本中不再推荐使用，并且根据社区版本迭代策略，在1.25及以上版本的集群中，ServiceAccount将不会自动创建对应的Secret。

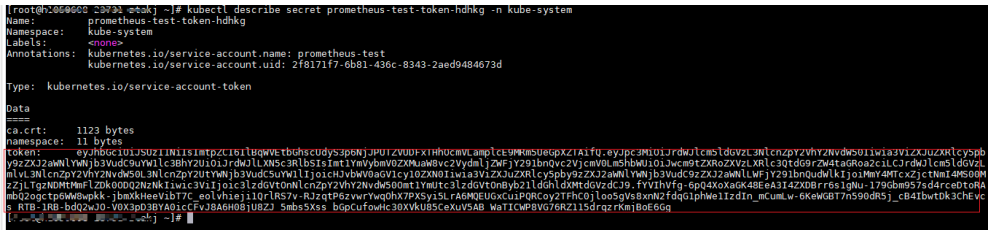
  1.21及以上版本的集群中，直接使用**TokenRequest** API**获得Token**，并使用投射卷（Projected Volume）挂载到Pod中。使用这种方法获得的Token具有固定的生命周期，并且当挂载的Pod被删除时这些Token将自动失效。
- 如果您在业务中需要一个永不过期的Token，您也可以选择**手动管理ServiceAccount的Secret**。尽管存在手动创建永久ServiceAccount Token的机制，但还是推荐使用**TokenRequest**的方式使用短期的Token，以提高安全性。

首先获取serviceaccount信息。

kubectl describe sa prometheus-test -n kube-system

```
[ [redacted]kj ~]# kubectl describe  sa  prometheus-test -n kube-system
Name:                prometheus-test
Namespace:           kube-system
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   prometheus-test-token-hdhkg
Tokens:              prometheus-test-token-hdhkg
Events:              <none>
[ [redacted]kj ~]#
```

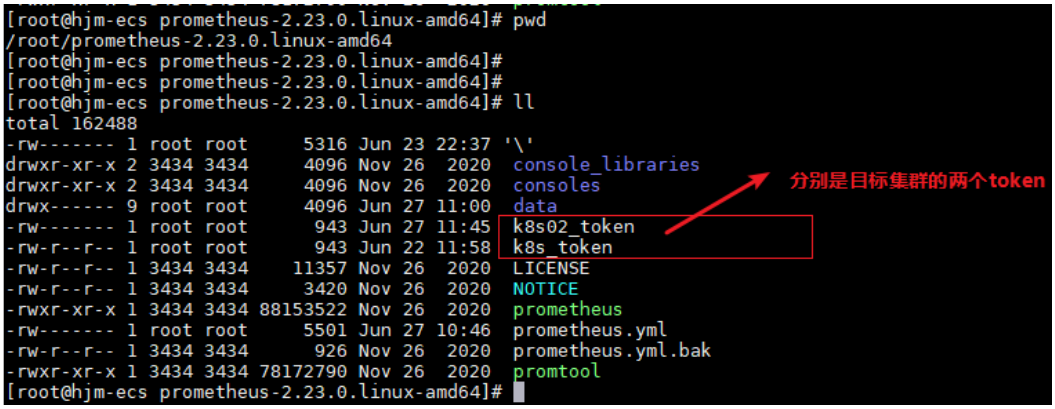
kubectl describe secret prometheus-test-token-hdhkg -n kube-system



记录下这个token值，就是要搜集的bearer_token信息。

步骤2 配置bearer_token 信息。

登录到Prometheus所在机器，进入Prometheus的安装目录，将目标集群的token信息保存在文件中。



步骤3 配置Prometheus监控job。

示例job监控的是容器指标。如果需要监控其他指标，可自行添加job编写抓取规则。

```
 - job_name: k8s_cAdvisor
   scheme: https
   bearer_token_file: k8s_token #上一步中的token文件
   tls_config:
     insecure_skip_verify: true
   kubernetes_sd_configs:  #kubernetes 自动发现配置
   - role: node    #node类型的自动发现
     bearer_token_file: k8s_token #上一步中的token文件
     api_server: https://192.168.0.153:5443  #K8s集群 apiserver地址
     tls_config:
       insecure_skip_verify: true    #跳过对服务端的认证
   relabel_configs:  ##用于在抓取metrics之前修改target的已有标签
   - target_label: __address__
     replacement: 192.168.0.153:5443
     action: replace
     ##将metrics_path地址转换为/api/v1/nodes/${1}/proxy/metrics/cadvisor
     #相当于通过APIServer代理到kubelet上获取数据
   - source_labels: [__meta_kubernetes_node_name]   #指定需要处理的源标签
     regex: (.+)    #匹配源标签的值,(.+)表示源标签什么值都可以匹配上
     target_label: __metrics_path__     #指定了需要replace后的标签
     replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor  # 表示替换后的标签即__metrics_path__ 对应的值。其中${1}表示正则匹配的值，即nodename
   - target_label: cluster
     replacement: xxxxx   ##根据实际情况填写 集群信息。也可不写

###下面这个job是监控另一个集群
 - job_name: k8s02_cAdvisor
   scheme: https
   bearer_token_file: k8s02_token #上一步中的token文件
   tls_config:
     insecure_skip_verify: true
```

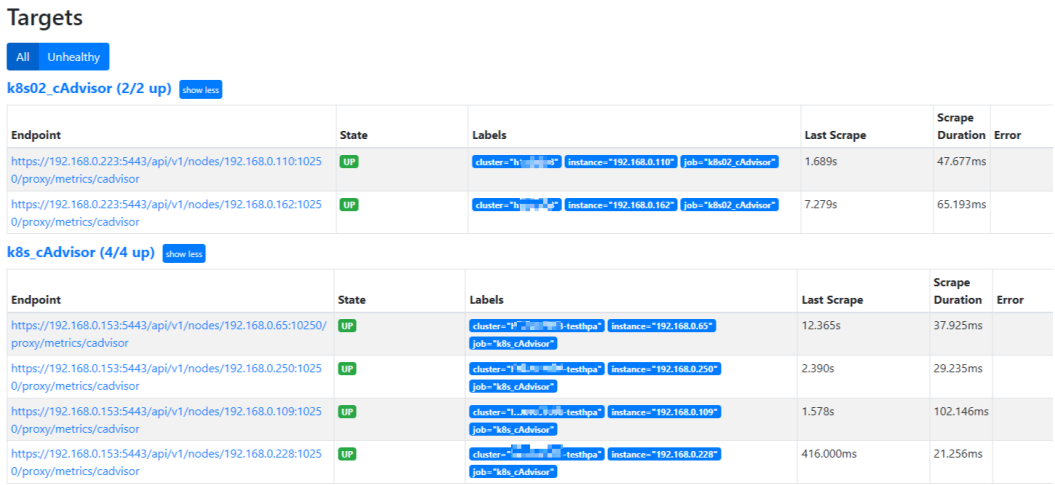
```
kubernetes_sd_configs:
- role: node
  bearer_token_file: k8s02_token #上一步中的token文件
  api_server: https://192.168.0.147:5443  #K8s集群 apiserver地址
  tls_config:
    insecure_skip_verify: true   #跳过对服务端的认证
relabel_configs:  ##用于在抓取metrics之前修改target的已有标签
- target_label: __address__
  replacement: 192.168.0.147:5443
  action: replace

- source_labels: [__meta_kubernetes_node_name]
  regex: (.+)
  target_label: __metrics_path__
  replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor

- target_label: cluster
  replacement: xxxx    ##根据实际情况填写 集群信息。也可不写
```

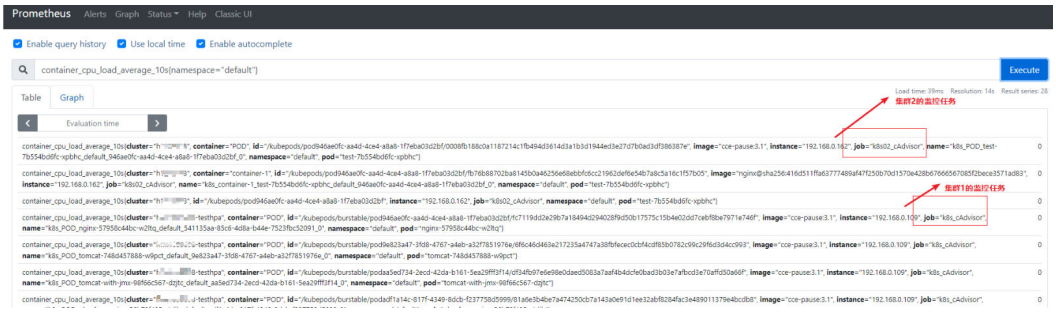
步骤4 启动prometheus服务。

配置完毕后，启动prometheus服务

./prometheus --config.file=prometheus.yml

步骤5 登录prometheus服务访问页面，查看监控信息。





----结束

21.7 集群

21.7.1 通过 kubectl 对接多个集群

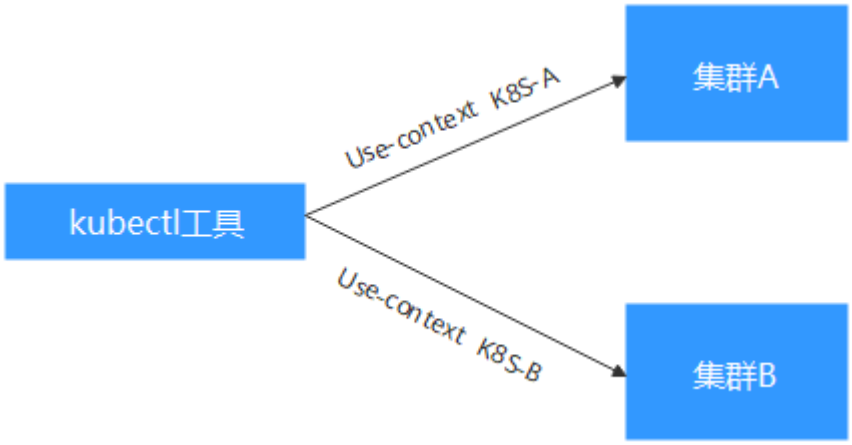
应用现状

在使用CCE时，通常会创建多个集群，如何方便快速的连接多个集群，成为一个问题。

解决方案

本文介绍一种通过修改kubeconfig.json配置文件配置多集群访问的方法，在kubeconfig.json中定义多个上下文、用户和集群。在使用时则只需要使用**kubectl config use-context**切换上下文连接到不同的集群。

图 21-5 kubectl 对接多集群示意



前提条件

kubectl需要能够访问多个集群。

kubeconfig.json 文件详解

kubeconfig.json是kubectl的配置文件，您可以在集群详情页面下载。

kubeconfig.json文件内容如下所示。

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [{
        "name": "internalCluster",
        "cluster": {
            "server": "https://192.168.0.85:5443",
            "certificate-authority-data": "LS0tLS1CRUULIE..."
        }
    }, {
        "name": "externalCluster",
        "cluster": {
            "server": "https://xxx.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
```

```
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTiBDRVJ...",
            "client-key-data": "LS0tLS1CRUdJTiBS..."
        }
    }],
    "contexts": [{
        "name": "internal",
        "context": {
            "cluster": "internalCluster",
            "user": "user"
        }
    }, {
        "name": "external",
        "context": {
            "cluster": "externalCluster",
            "user": "user"
        }
    }],
    "current-context": "external"
}
```

其中主要分为3部分内容。

- clusters：描述集群的信息，主要是集群的访问地址。
- users：描述访问集群访问用户的信息，主要是client-certificate-data和client-key-data这两个证书文件内容。
- contexts：描述配置的上下文，用于使用时切换。上下文会关联user和cluster，也就是定义使用哪个user去访问哪个集群。

从上面的kubeconfig.json文件可以看出，此处将集群的内网地址和公网访问地址分别定义成一个集群，且定义了两个上下文，从而能够通过切换上下文选择使用不同的地址访问集群。

配置多集群访问

下面以配置2个集群为例演示如何修改kubeconfig.json文件访问多个集群。

为简洁文档篇幅，如下示例选取公网访问的方式，删除内网访问方式。如果您需要在内网访问多集群，只需要保留内网访问的集群clusters字段，保证能够从内网访问到集群即可，其方法与下面内容并无本质区别。

步骤1 分别下载2个集群的kubeconfig.json文件，删除内网访问内容，如下所示。

- 集群A：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "externalCluster",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "external",
```

```
        "context": {
            "cluster": "externalCluster",
            "user": "user"
        }
    }],
    "current-context": "external"
}
```

- 集群B：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "externalCluster",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0rTUideUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "external",
        "context": {
            "cluster": "externalCluster",
            "user": "user"
        }
    }],
    "current-context": "external"
}
```

此时这两个文件除了集群访问地址clusters.cluster.server和user字段的client-certificate-data和client-key-data字段内容不同，文件结构完全一致。

步骤2 修改两个配置文件中的名称，如下所示。

- 集群A：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "Cluster-A",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "Cluster-A-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "Cluster-A-Context",
        "context": {
            "cluster": "Cluster-A",
            "user": "Cluster-A-user"
        }
    }],
    "current-context": "Cluster-A-Context"
}
```

- 集群B：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "Cluster-B",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "Cluster-B-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0rTUideUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "Cluster-B-Context",
        "context": {
            "cluster": "Cluster-B",
            "user": "Cluster-B-user"
        }
    }],
    "current-context": "Cluster-B-Context"
}
```

步骤3 将两个文件的内容合并。

文件结构不变，将clusters、users和contexts的内容合并即可，如下所示。

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "Cluster-A",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    },
     {
        "name": "Cluster-B",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "Cluster-A-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    },
    {
        "name": "Cluster-B-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0rTUideUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "Cluster-A-Context",
        "context": {
            "cluster": "Cluster-A",
```

```
            "user": "Cluster-A-user"
        }
    },
    {
        "name": "Cluster-B-Context",
        "context": {
            "cluster": "Cluster-B",
            "user": "Cluster-B-user"
        }
    }],
    "current-context": "Cluster-A-Context"
}
```

----结束

配置验证

将两个文件内容合并到一个kubeconfig.json后，执行如下命令将文件拷贝到kubectl配置路径下。

mkdir -p $HOME/.kube

mv -f kubeconfig.json $HOME/.kube/config

执行kubectl命令是否能够连接两个集群。

```
# kubectl config use-context Cluster-A-Context
Switched to context "Cluster-A-Context".
# kubectl cluster-info
Kubernetes control plane is running at https://119.xxx.xxx.xxx:5443
CoreDNS is running at https://119.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

# kubectl config use-context Cluster-B-Context
Switched to context "Cluster-B-Context".
# kubectl cluster-info
Kubernetes control plane is running at https://124.xxx.xxx.xxx:5443
CoreDNS is running at https://124.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

21.7.2 选择合适的节点数据盘大小

节点在创建时会默认创建一块数据盘，供容器运行时和Kubelet组件使用。由于容器运行时和Kubelet组件使用的数据盘不可被卸载，且默认大小为100G，出于使用成本考虑，节点上挂载的普通数据盘支持下调至10G。

须知

调整容器运行时和Kubelet组件使用的数据盘大小存在一些风险，根据本文提供的预估方法，建议综合评估后再做实际调整。

- 过小的数据盘容量可能会频繁出现磁盘空间不足，导致镜像拉取失败的问题。如果节点上需要频繁拉取不同的镜像，不建议将数据盘容量调小。
- 集群升级预检查会检查数据盘使用量是否超过95%，磁盘压力较大时可能会影响集群升级。
- Device Mapper类型比较容易出现空间不足的问题，建议使用OverlayFS类型操作系统，或者选择较大数据盘。
- 从日志转储的角度，应用的日志应单独挂盘存储，以免dockersys分区存储空间不足，影响业务运行。
- 调小数据盘容量后，建议您的集群安装npd插件，用于检测可能出现的节点磁盘压力问题，以便您及时感知。如出现节点磁盘压力问题，可根据**数据盘空间不足时如何解决**进行解决。

约束与限制

- 仅1.19及以上集群支持调小容器运行时和Kubelet组件使用的数据盘容量。
- 调整数据盘大小功能只支持云硬盘，不支持本地盘（本地盘仅在节点规格为“磁盘增强型”或“超高I/O型”时可选）。

如何选择合适的数据盘

在选择合适的数据盘大小时，需要结合以下考虑综合计算：

- 在拉取镜像过程中，会先从镜像仓库中下载镜像tar包然后解压，最后删除tar包保留镜像文件。在tar包的解压过程中，tar包和解压出来的镜像文件会同时存在，占用额外的存储空间，需要在计算所需的数据盘大小时额外注意。
- 在集群创建过程中，节点上可能会部署必装插件（如Everest插件、coredns插件等），这些插件会占用一定的空间，在计算数据盘大小时，需要为其预留大约2G的空间。
- 在应用运行过程中会产生日志，占用一定的空间，为保证业务正常运行，需要为每个Pod预留大约1G的空间。

根据不同的节点存储类型，详细的计算公式请参见**OverlayFS类型**及**Device Mapper类型**。

OverlayFS 类型

OverlayFS类型节点上的容器引擎和容器镜像空间默认占数据盘空间的90%（建议维持此值），这些容量全部用于dockersys分区，计算公式如下：

- 容器引擎和容器镜像空间：默认占数据盘空间的90%，其空间大小 = 数据盘空间 * 90%
  - dockersys分区（/var/lib/docker路径）：容器引擎和容器镜像空间（默认占90%）都在/var/lib/docker目录下，其空间大小 = 数据盘空间 * 90%
- Kubelet组件和EmptyDir临时存储：占数据盘空间的10%，其空间大小 = 数据盘空间 * 10%

在OverlayFS类型的节点上，由于拉取镜像时，下载tar包后会存在解压过程，该过程中tar包和解压出来的镜像文件会同时存在于dockersys空间，会占用约2倍的镜像实际容量大小，等待解压完成后tar包会被删除。因此，在实际镜像拉取过程中，除去系统插件镜像占用的空间后，需要保证dockersys分区的剩余空间大于2倍的镜像实际容量。为保证容器能够正常运行，还需要在dockersys分区预留出相应的Pod容器空间，用于存放容器日志等相关文件。

因此在选择合适的数据盘时，需满足以下公式：

dockersys分区容量 > 2*镜像实际总容量 + 系统插件镜像总容量（约2G） + 容器数量 * 单个容器空间（每个容器需预留约1G日志空间）

说明

当容器日志选择默认的json.log形式输出时，会占用dockersys分区，若容器日志单独设置持久化存储，则不会占用dockersys空间，请根据实际情况估算**单个容器空间**。

例如：

假设节点的存储类型是OverlayFS，节点数据盘大小为20G。根据**上述计算公式**，默认的容器引擎和容器镜像空间比例为90%，则dockersys分区盘占用：20G*90% = 18G，且在创建集群时集群必装插件可能会占用2G左右的空间。倘若此时您需要部署10G的镜像tar包，但是由于解压tar包时大约会占用20G的dockersys空间，再加上必装插件占用的空间，超出了dockersys剩余的空间大小，极有可能导致镜像拉取失败。

Device Mapper 类型

Device Mapper类型节点上的容器引擎和容器镜像空间默认占数据盘空间的90%（建议维持此值），这些容量又分为dockersys分区和thinpool空间，计算公式如下：

- 容器引擎和容器镜像空间：默认占数据盘空间的90%，其空间大小 = 数据盘空间 * 90%
  - dockersys分区（/var/lib/docker路径）：默认占比20%，其空间大小 = 数据盘空间 * 90% * 20%
  - thinpool空间：默认占比为80%，其空间大小 = 数据盘空间 * 90% * 80%
- Kubelet组件和EmptyDir临时存储：占数据盘空间的10%，其空间大小 = 数据盘空间 * 10%

在Device Mapper类型的节点上，拉取镜像时tar包会在dockersys分区临时存放，等tar包解压后会把实际镜像文件存放在thinpool空间，最后dockersys空间的tar包会被删除。因此，在实际镜像拉取过程中，需要保证dockersys分区的空间大小和thinpool空间大小均有剩余。由于dockersys空间比thinpool空间小，因此在计算数据盘空间大小时，需要额外注意。为保证容器能够正常运行，还需要在dockersys分区预留出相应的Pod容器空间，用于存放容器日志等相关文件。

因此在选择合适的数据盘时，需同时满足以下公式：

- **dockersys分区容量 > tar包临时存储（约等于镜像实际总容量） + 容器数量 * 单个容器空间（每个容器需预留约1G日志空间）**
- **thinpool空间 > 镜像实际总容量 + 系统插件镜像总容量（约2G）**

说明

当容器日志选择默认的json.log形式输出时，会占用dockersys分区，若容器日志单独设置持久化存储，则不会占用dockersys空间，请根据实际情况估算**单个容器空间**。

例如：

假设节点的存储类型是Device Mapper，节点数据盘大小为20G。根据**上述计算公式**，默认的容器引擎和容器镜像空间比例为90%，则dockersys分区盘占用：20G*90%*20% = 3.6G，且在创建集群时集群必装插件可能会占用2G左右的dockersys空间，所以剩余1.6G左右。倘若此时您需要部署大于1.6G的镜像tar包，虽然thinpool空间足够，但是由于解压tar包时dockersys分区空间不足，极有可能导致镜像拉取失败。

数据盘空间不足时如何解决

方案一：清理镜像

您可以执行以下步骤清理未使用的镜像：

- 使用containerd容器引擎的节点：
  a. 查看节点上的本地镜像。
  ```
  crictl images -v
  ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
  ```
  crictl rmi {镜像ID}
  ```
- 使用docker容器引擎的节点：
  a. 查看节点上的本地镜像。
  ```
  docker images
  ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
  ```
  docker rmi {镜像ID}
  ```

说明

请勿删除cce-pause等系统镜像，否则可能导致无法正常创建容器。

方案二：扩容磁盘

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。
  ```
  # lsblk
  NAME                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
  sda                 8:0    0   50G  0 disk
  └─sda1              8:1    0   50G  0 part /
  sdb                 8:16   0  200G  0 disk
  ├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker                # 容器引擎使用的空间
  └─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
  ```
  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。
  ```
  pvresize /dev/sdb
  lvextend -l+100%FREE -n vgpaas/dockersys
  resize2fs /dev/vgpaas/dockersys
  ```
- Devicemapper，单独划分了thinpool存储镜像相关数据。
  ```
  # lsblk
  NAME                      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
  sda                       8:0    0   50G  0 disk
  └─sda1                    8:1    0   50G  0 part /
  sdb                       8:16   0  200G  0 disk
  ```

```
├─vgpaas-dockersys               253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta          253:1    0    3G  0 lvm
│ └─vgpaas-thinpool              253:3    0   67G  0 lvm                # thinpool空间
│  ...
├─vgpaas-thinpool_tdata           253:2    0   67G  0 lvm
│ └─vgpaas-thinpool              253:3    0   67G  0 lvm
│  ...
└─vgpaas-kubernetes              253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

- 在节点上执行如下命令， 将新增的磁盘容量加到thinpool盘上。

  ```
  pvresize /dev/sdb
  lvextend -l+100%FREE -n vgpaas/thinpool
  ```

- 在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

  ```
  pvresize /dev/sdb
  lvextend -l+100%FREE -n vgpaas/dockersys
  resize2fs /dev/vgpaas/dockersys
  ```

----结束

21.8 网络

21.8.1 集群网络地址段规划实践

在CCE中创建集群时，您需要根据具体的业务需求规划VPC的数量、子网的数量、容器网段划分和服务网段连通方式。

本文将介绍VPC环境下CCE集群里各种地址的作用，以及地址段该如何规划。

约束与限制

通过搭建VPN方式访问CCE集群，需要注意VPN网络和集群所在的VPC网段、容器使用网段不能冲突。

集群各网段基本概念

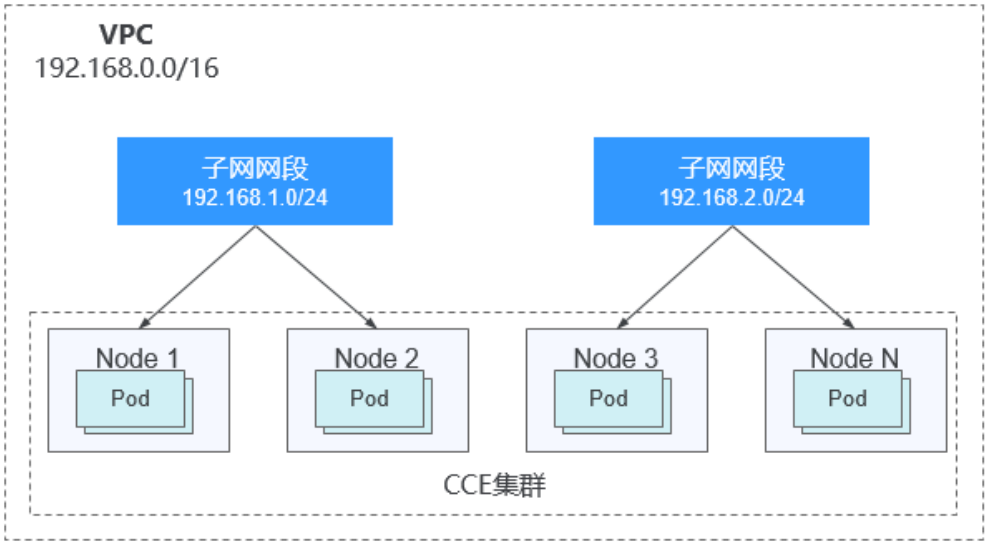
- **VPC网段**

  虚拟私有云（Virtual Private Cloud，简称VPC）可以为云服务器、云容器、云数据库等资源构建隔离的、用户自主配置和管理的虚拟网络环境。您可以自由配置VPC内的IP地址段、子网、安全组等子服务，也可以申请弹性带宽和弹性公网IP搭建业务系统。

- **子网网段**

  子网是用来管理弹性云服务器网络平面的一个网络，可以提供IP地址管理、DNS服务，子网内的弹性云服务器IP地址都属于该子网。

图 21-6 VPC 网段结构



默认情况下，同一个VPC的所有子网内的弹性云服务器均可以进行通信，不同VPC的弹性云服务器不能进行通信。

不同VPC的弹性云服务器可通过VPC创建对等连接通信。

- **容器网段**（Pod网段）

  Pod是Kubernetes内的概念，每个Pod具有一个IP地址。

  在CCE上创建集群时，可以指定Pod的地址段（即容器网段），容器网段不能和子网网段重叠。例如子网网段用的是 192.168.0.0/16，集群的容器网段就不能使用192.168.0.0/18，192.168.1.0/18等，因为这些地址都涵盖在 192.168.0.0/16 里了。

- **容器子网**（仅CCE Turbo集群）

  CCE Turbo集群中，容器直接从VPC网段中分配IP地址，容器子网可以和子网网段重叠，但需要注意该容器子网的大小决定了集群下容器的数量上限。在集群创建完成后，仅支持新增容器子网，不支持删除。

- **服务网段**

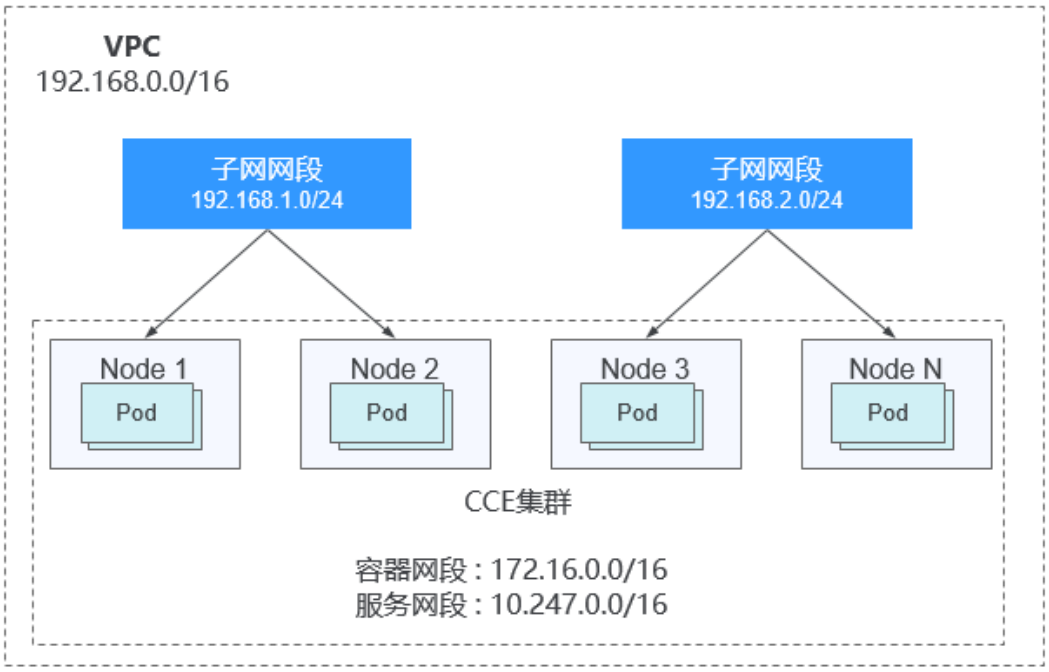
  Service也是Kubernetes内的概念，每个Service都有自己的地址，在CCE上创建集群时，可以指定Service的地址段（即服务网段）。同样，服务网段也不能和子网网段重合，而且服务网段也不能和容器网段重叠。服务网段只在集群内使用，不能在集群外使用。

单 VPC+单集群场景

CCE集群：包含VPC网络模式和容器隧道网络模式集群，集群网络地址段规划示意图如图21-7所示。

- VPC网段：集群所在的VPC网段，该网段的大小影响集群中可创建的节点数量上限。
- 子网网段：集群中节点所在的子网网段，子网网段包含在VPC网段中。同个集群中的不同节点可分配到不同的子网网段。
- 容器网段：容器网段不能和子网网段重叠。
- 服务网段：服务网段不能和子网网段重叠，而且也不能和容器网段重叠。

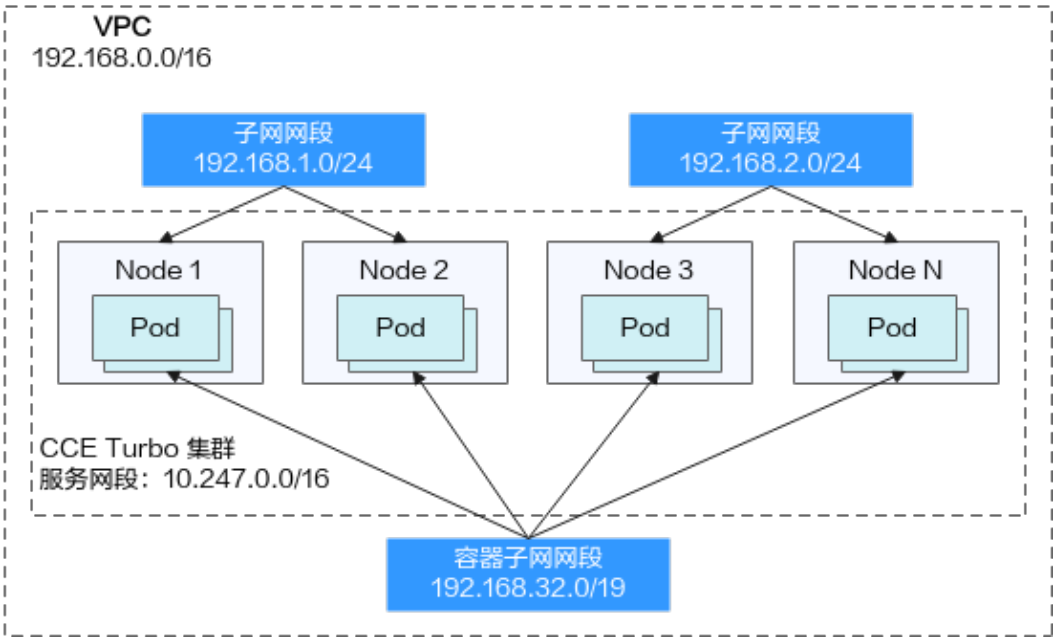
图 21-7 单 VPC 单集群场景网段规划-CCE 集群



CCE Turbo集群：即云原生网络2.0模式集群，集群网络地址段规划示意图如**图21-8**所示。

- VPC网段：集群所在的VPC网段，该网段的大小影响集群中可创建的节点数量上限。
- 子网网段：集群中节点所在的子网网段，子网网段包含在VPC网段中。同个集群中的不同节点可分配到不同的子网网段。
- 容器子网网段：容器子网包含在VPC网段中，且可以和子网网段重叠，甚至可以选择和子网网段相同。但需要注意的是，由于容器直接分配VPC中的IP，因此该容器子网的大小决定了集群下容器的数量上限。在集群创建完成后，仅支持新增容器子网，不支持删除。建议将容器子网的IP地址段设大一些，以免出现容器IP分配不足的情况。
- 服务网段：服务网段不能和子网网段重合，而且也不能和容器网段重叠。

图 21-8 单 VPC 单集群场景网段规划-CCE Turbo 集群



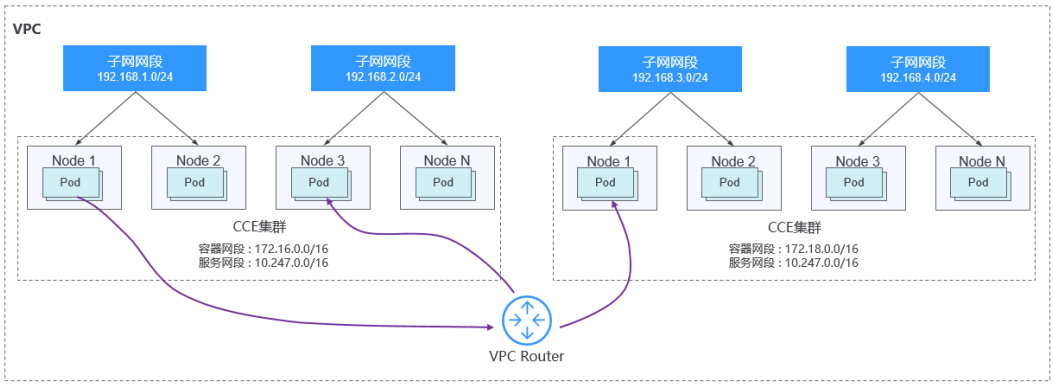
单 VPC+多集群场景

VPC网络模式

Pod的报文需要通过VPC路由转发，CCE会自动在VPC路由上配置到每个容器网段的路由表，集群组网规模受限于VPC路由表能力。集群网络地址段规划示意图如图21-9所示。

- VPC网段：集群所在的VPC网段，该网段的大小影响集群中可创建的节点数量上限。
- 子网网段：每个集群中的子网网段不能和容器网段重叠。
- 容器网段：单VPC中存在多个VPC网络模型集群的场景下，由于各个集群使用同一路由表，因此所有集群的容器网段不能相互重叠。在此情况下CCE集群部分互通，一个集群的Pod可以直接访问另外一个集群的Pod，但不能访问另外一个集群的Service。
- 服务网段：由于服务网段只能在集群中使用，因此集群之间服务网段可以重叠，但是不能和所属集群的子网网段和容器网段重叠。

图 21-9 VPC 网络-多集群场景示例

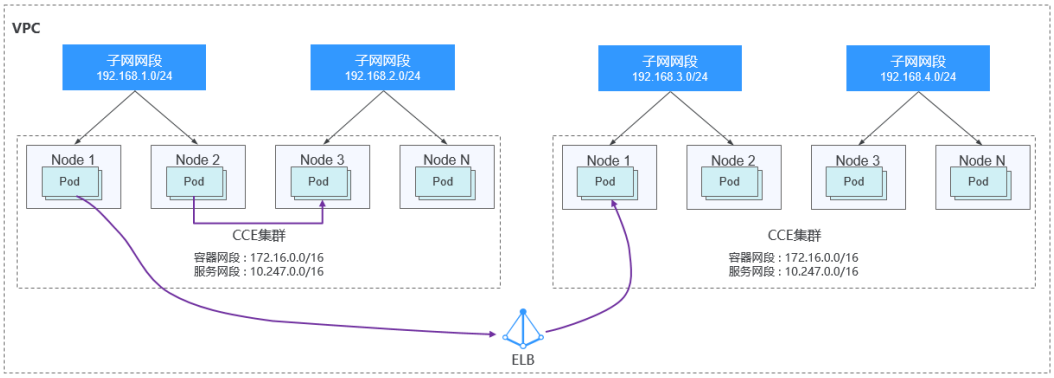


容器隧道网络

该模式下容器网络是承载于VPC网络之上的Overlay网络平面，具有少量隧道封装性能损耗，但获得了通用性强、互通性强、高级特性支持全面（例如Network Policy网络隔离）的优势，可以满足大多数应用需求。集群网络地址段规划示意图如图21-10所示。

- VPC网段：集群所在的VPC网段，该网段的大小影响集群中可创建的节点数量上限。
- 子网网段：每个集群中的子网网段不能和容器网段重叠。
- 容器网段：所有集群的容器网段可以重叠。在此情况下不同集群的Pod直接不能通过IP直接访问，跨集群容器之间的访问建议通过ELB实现。
- 服务网段：由于服务网段只能在集群中使用，因此集群之间服务网段可以重叠，但是不能和所属集群的子网网段和容器网段重叠。

图 21-10 容器隧道网络-多集群场景示例

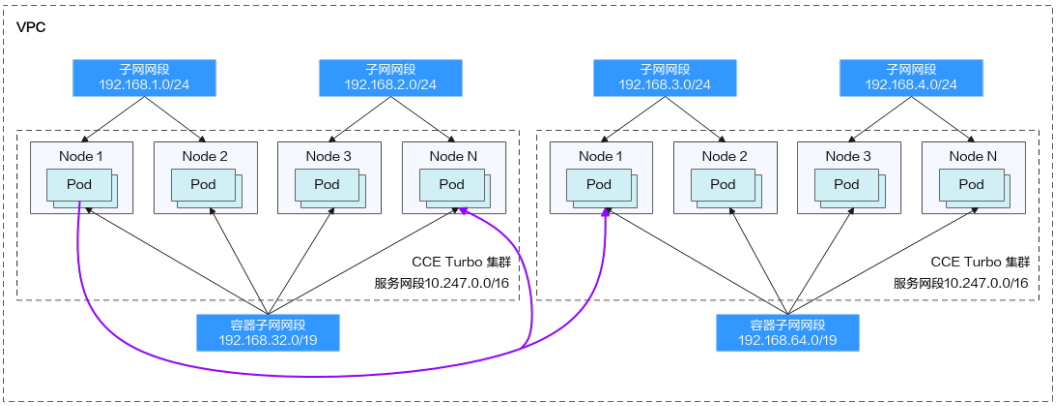


云原生网络2.0模式（即CCE Turbo集群）

该模式下集群直接从VPC网段内分配容器IP地址，支持ELB直通容器、支持容器直接绑定安全组等多种VPC网络的能力，极大提高了网络连通速度和转发效率。

- VPC网段：集群所在的VPC网段，在CCE Turbo集群中，该网段的大小影响集群中可创建的节点数量与容器数量之和。
- 子网网段：CCE Turbo集群中的子网网段没有特殊限制。
- 容器子网：容器子网的网段包含在VPC网段中，且不同集群的容器子网之间可以重叠，也可以和子网网段重叠。但仍建议您将不同集群的容器网段错开，且尽量保证容器子网网段的IP数充足。在此情况下，不同集群的Pod之间可以直接通过IP访问。
- 服务网段：由于服务网段只能在集群中使用，因此集群之间服务网段可以重叠，但是不能和所属集群的子网网段和容器子网网段重叠。

图 21-11 云原生网络 2.0-多集群场景示例



多网络模式集群并存场景

同一VPC中包含多个网络模式的集群时，应在创建新集群时遵循以下规律：

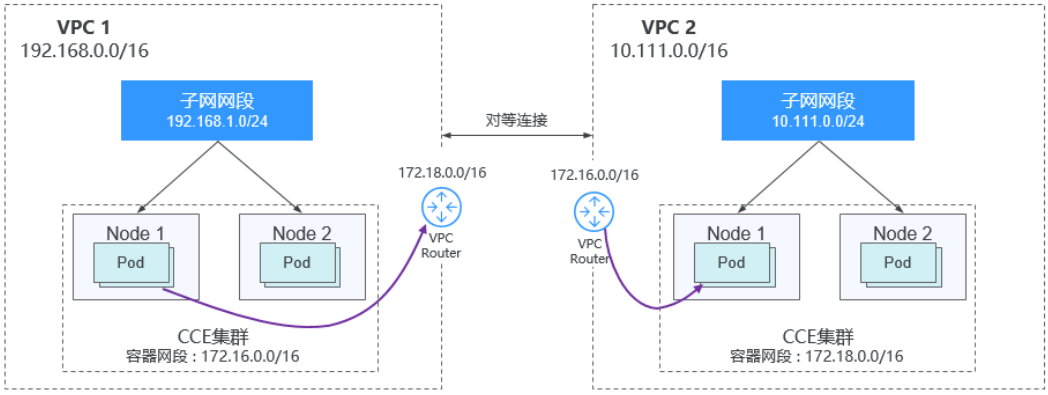
- VPC网段：该场景下各个集群所在的VPC网段相同，请保证VPC内可用的IP地址数充足。
- 子网网段：子网网段尽量避免和容器网段重叠。即使在某些场景下（例如和CCE Turbo集群共存时），子网网段可以和容器（子网）网段重叠，但从地址段规划角度出发，这是不推荐的。
- 容器网段：仅**VPC网络模式**的集群间的容器网段需要避免相互重叠。
- 服务网段：所有集群之间服务网段可以重叠，但是不能和所属集群的子网网段和容器网段重叠。

集群跨 VPC 互联场景

两个VPC网络互联的情况下，可以通过路由表配置哪些报文要发送到对端VPC里。

在“**VPC网络**”模式下，创建对等连接后，您需要在两端VPC内添加对等连接路由信息，才能使两个VPC互通。

图 21-12 VPC 网络-VPC 互联场景



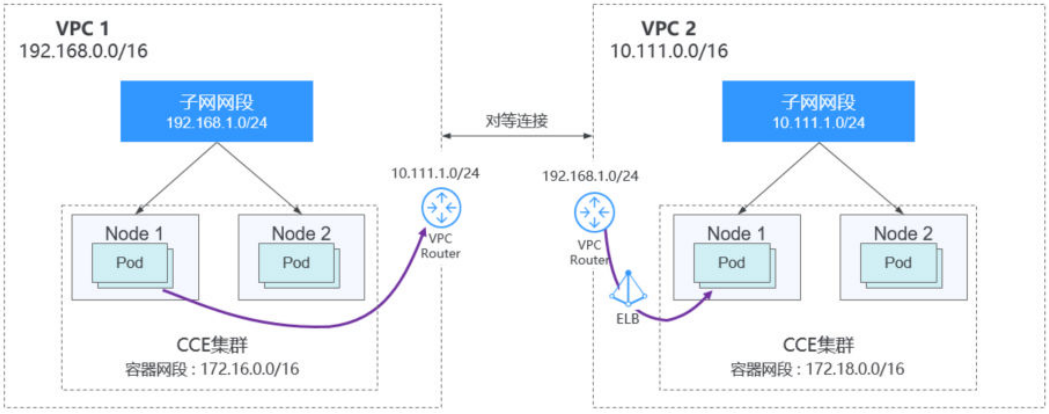
跨VPC的集群容器之间互联需要建立VPC对等连接时，需要注意如下几点：

- 两端集群所属的VPC地址段需要避免重叠，且在每个集群中，子网网段不能与容器网段重叠。

- 两端集群的容器网段不能相互重叠，但服务网段可以重叠。
- 两端的VPC路由表中不仅需要添加对端的VPC网段地址，还需要添加对端容器网段的地址。需要注意该操作在两侧的VPC路由表中均要进行。

同样，在**“容器隧道网络”**模式下，创建对等连接后，您需要在两端VPC内添加对等连接路由信息，才能使两个VPC互通。

图 21-13 容器隧道网络-VPC 互联场景



需要注意如下几点：

- 两端集群所属的VPC地址段需要避免重叠。
- 所有集群的容器网段可以重叠，服务网段也可以重叠。
- 对等连接的路由表中需要添加对端集群节点子网网段的地址。

在**“云原生2.0网络”**模式下，创建对等连接后，您仅需要在两端VPC内添加对等连接路由信息，使两个VPC互通，即可完成集群间的互通。仅需注意两端集群所属的VPC地址段避免重叠即可。

VPC 网络到 IDC 的场景

和VPC互联场景类似，同样存在VPC里部分地址段路由到IDC，CCE集群的Pod地址就不能和这部分地址重叠。IDC里如果需要访问集群里的Pod地址，同样需要在IDC端配置到专线VBR的路由表。

21.8.2 集群网络模型选择及各模型区别

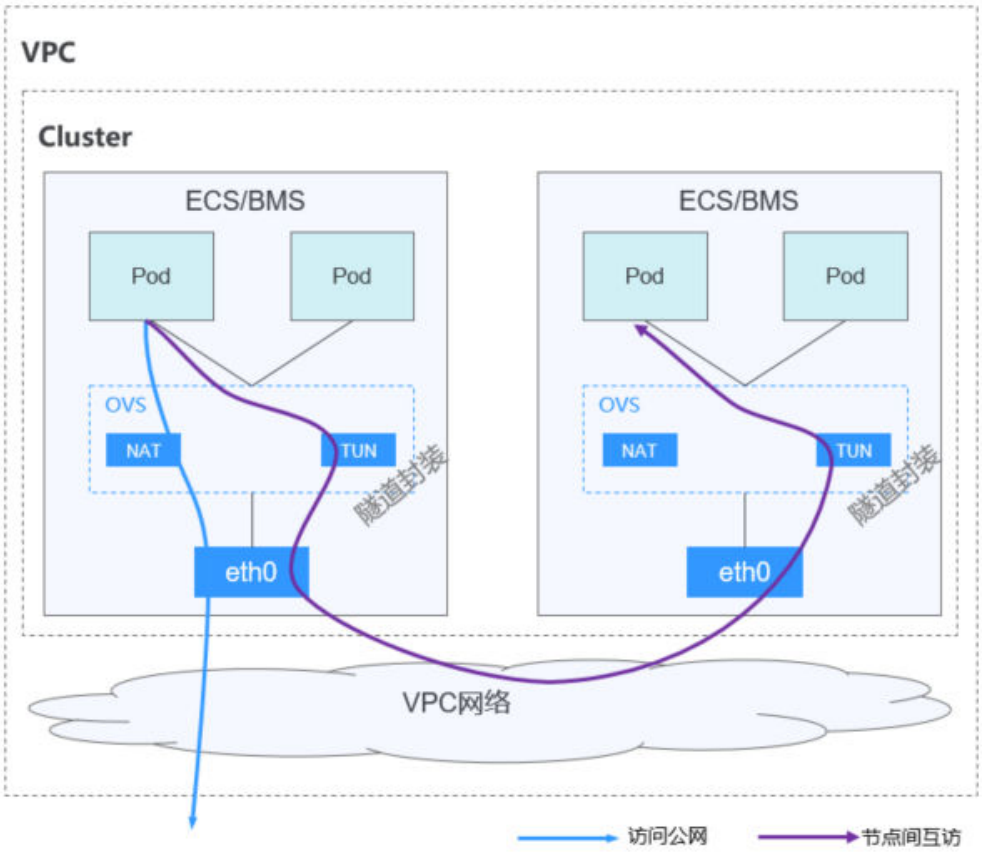
自研高性能商业版容器网络插件，支持容器隧道网络、VPC网络、云原生网络2.0网络模型：

⚠ 注意

集群创建成功后，网络模型不可更改，请谨慎选择。

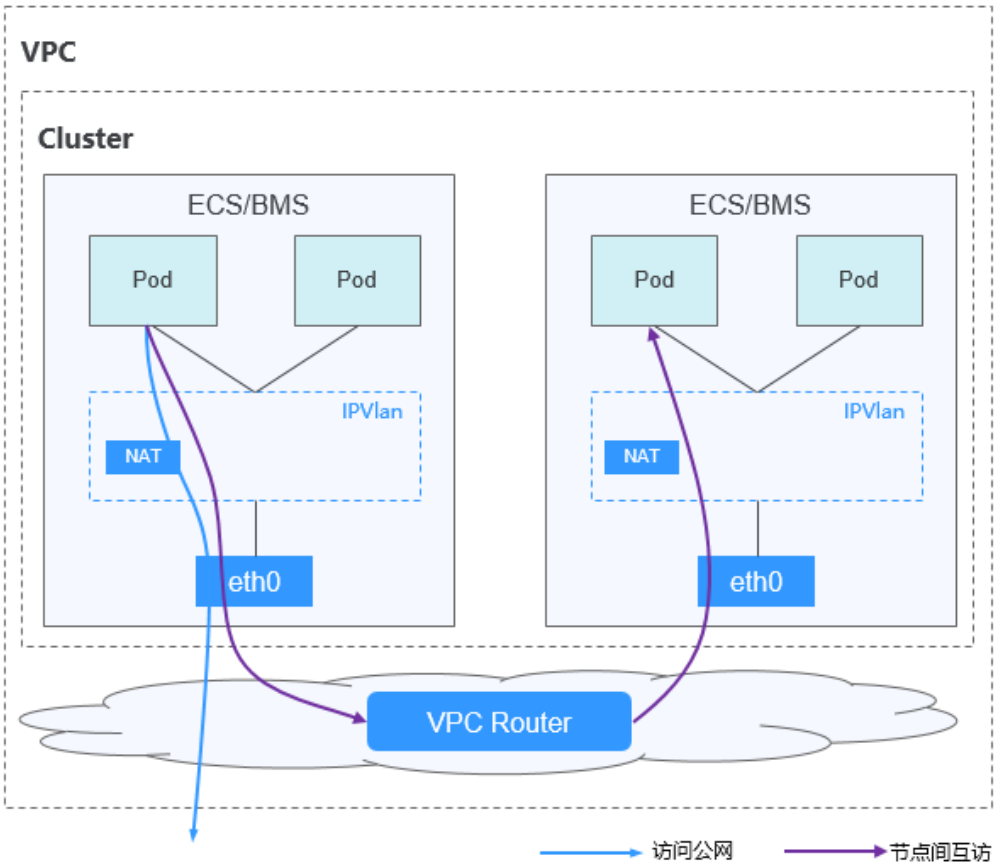
- **容器隧道网络（Overlay）：**基于底层VPC网络构建了独立的VXLAN隧道化容器网络，适用于一般场景。VXLAN是将以太网报文封装成UDP报文进行隧道传输。容器网络是承载于VPC网络之上的Overlay网络平面，具有付出少量隧道封装性能损耗，即可获得通用性强、互通性强、高级特性支持全面（例如Network Policy网络隔离）的优势，可以满足大多数应用需求。

图 21-14 容器隧道网络



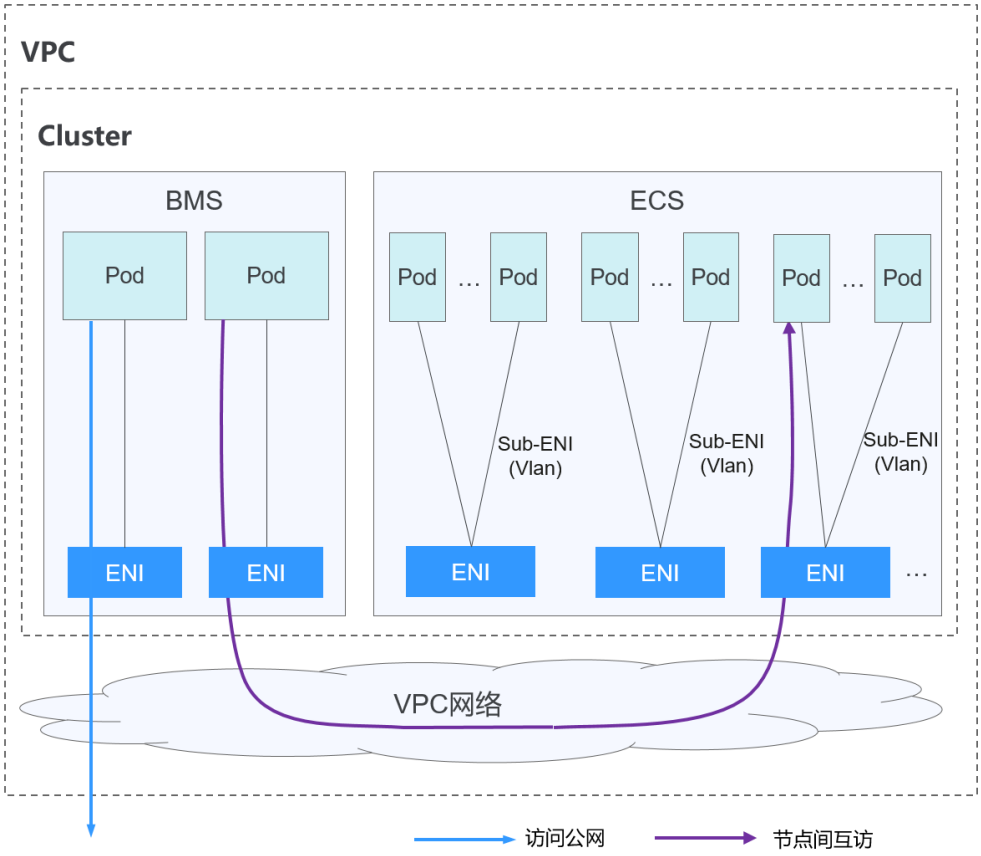
- **VPC网络：**采用VPC路由方式与底层网络深度整合，适用于高性能场景，节点数量受限于虚拟私有云VPC的路由配额。每个节点将会被分配固定大小的IP地址段。VPC网络由于没有隧道封装的消耗，容器网络性能相对于容器隧道网络有一定优势。VPC网络集群由于VPC路由中配置有容器网段与节点IP的路由，可以支持集群外直接访问容器实例等特殊场景。

图 21-15 VPC 网络



- **云原生网络2.0：**深度整合弹性网卡（Elastic Network Interface，简称ENI）能力，采用VPC网段分配容器地址，支持ELB直通容器，享有高性能。

图 21-16 云原生网络 2.0



网络模型对比如下：

表 21-9 网络模型对比

对比维度	容器隧道网络	VPC网络	云原生网络2.0
核心技术	OVS	IPVlan，VPC路由	VPC弹性网卡/弹性辅助网卡
适用集群	CCE集群	CCE集群	CCE Turbo集群
网络隔离	Pod支持Kubernetes原生NetworkPolicy	否	Pod支持使用安全组隔离
IP地址管理	• 容器网段单独分配 • 节点维度划分地址段，动态分配（地址段分配后可动态增加）	• 容器网段单独分配 • 节点维度划分地址段，静态分配（节点创建完成后，地址段分配即固定，不可更改）	容器网段从VPC子网划分，无需单独分配
网络性能	基于vxlan隧道封装，有一定性能损耗。	无隧道封装，跨节点通过VPC 路由器转发，性能好，媲美主机网络。	容器网络与VPC网络融合，性能无损耗

对比维度	容器隧道网络	VPC网络	云原生网络2.0
组网规模	最大可支持2000节点	默认支持200节点，受限于VPC路由表能力。 VPC网络模式下，集群每添加一个节点，会在VPC的路由表中添加一条路由，因此集群本身规模受VPC路由表上限限制。	最大可支持2000节点
适用场景	• 一般容器业务场景。 • 对网络时延、带宽要求不是特别高的场景。	• 对网络时延、带宽要求高。 • 容器与虚机IP互通，使用了微服务注册框架的，如Dubbo、CSE等。	• 对网络时延、带宽要求高，高性能场景。 • 容器与虚机IP互通，使用了微服务注册框架的，如Dubbo、CSE等。

须知

1. VPC路由网络集群实际支持规模受限于VPC的路由表路由条目配额，创建前请提前评估集群规模。
2. VPC路由网络默认支持容器与同一VPC的虚拟机直接互访，与其他VPC的主机在配置对等连接策略后可以支持直接互访。此外，云专线/VPN等混合组网场景在合理规划后可以支持对端直接与容器互访。

21.8.3 通过负载均衡配置实现会话保持

概念

会话保持是负载均衡最常见的问题之一，也是一个相对比较复杂的问题。

会话保持有时候又叫做粘滞会话（Sticky Sessions），开启会话保持后，负载均衡会把来自同一客户端的访问请求持续分发到同一台后端云服务器上进行处理。

在介绍会话保持技术之前，必须先花点时间弄清楚一些概念：什么是连接（Connection）、什么是会话（Session），以及这二者之间的区别。需要特别强调的是，如果仅仅是谈论负载均衡，会话和连接往往具有相同的含义。

从简单的角度来看，如果用户需要登录，那么就可以简单的理解为会话；如果不需要登录，那么就是连接。

实际上，会话保持机制与负载均衡的基本功能是完全矛盾的。负载均衡希望将来自客户端的连接、请求均衡的转发至后端的多台服务器，以避免单台服务器负载过高；而会话保持机制却要求将某些请求转发至同一台服务器进行处理。因此，在实际的部署环境中，需要根据应用环境的特点，选择适当的会话保持机制。

四层负载均衡（Service）

四层的模式下可以开启基于源IP的会话保持（基于客户端的IP进行hash路由），Service上开启基于源IP的会话保持需要满足以下条件：

CCE集群

1. Service的服务亲和级别选择“节点级别”（即Service的externalTrafficPolicy字段为Local）。
2. Service的负载均衡配置启用源IP地址会话保持。

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: 56dcc1b4-8810-480c-940a-a44f7736f0dc
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

3. Service后端的应用开启反亲和。

七层负载均衡（Ingress）

7层的模式下可以开启基于http cookie和app cookie的会话保持 ， 在ingress上开启基于cookie的会话保持需要满足以下条件：

1. Ingress对应的应用（工作负载）应该开启与自身反亲和。
2. Ingress对应的service需要开启节点亲和。

操作步骤：

步骤1 创建nginx工作负载。

实例数设置为3，通过工作负载反亲和设置Pod与自身反亲和。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
```

```
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
      imagePullSecrets:
        - name: default-secret
      affinity:
        podAntiAffinity:                    # Pod与自身反亲和
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              topologyKey: kubernetes.io/hostname
```

步骤2 创建NodePort类型Service。

会话保持的配置在Service这里设置，Ingress可以对接多个Service，每个Service可以有不同的会话保持配置。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN
    kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE      # HTTP Cookie类型
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout":"1440"}'   # 会话保持时间，单位为分钟，取值范围为1-1440
spec:
  selector:
    app: nginx
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 32633            # 节点端口
  type: NodePort
  externalTrafficPolicy: Local   # 节点级别转发
```

还可以选择应用程序Cookie，如下所示。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN
    kubernetes.io/elb.session-affinity-mode: APP_COOKIE     # 选择应用程序Cookie
    kubernetes.io/elb.session-affinity-option: '{"app_cookie_name":"test"}'  # 应用程序Cookie名称
...
```

步骤3 创建Ingress，关联Service。

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.autocreate:
```

```
    '{
        "type":"public",
        "bandwidth_name":"cce-bandwidth-test",
        "bandwidth_chargemode":"traffic",
        "bandwidth_size":1,
        "bandwidth_sharetype":"PER",
        "eip_type":"5_bgp"
      }'
spec:
  rules:
  - host: 'www.example.com'
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: nginx     # Service的名称
            port:
              number: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```

步骤4 登录ELB控制台，进入对应的ELB实例，查看会话保持是否开启。

----结束

21.8.4 不同场景下容器内获取客户端源 IP

背景

Kubernetes已经成为当今容器化的标准，人们在享受容器带来的高效与便利的同时，也遇到一些烦恼：客户端和容器服务器之间可能存在多种不同形式的代理服务器，那容器中如何获取到客户端真实的源IP呢？下面就几种场景类型进行讨论。

原理介绍

七层转发：

Ingress：应用在七层访问时，客户端源IP默认保存在HTTP头部的“X-Forwarded-For”字段，无需做其他操作。

- ELB型：自研ELB型Ingress基于弹性负载均衡服务ELB实现公网和内网（同一VPC内）的七层网络访问，当后端服务为NodePort类型时，需将“服务亲和”选项设置为“节点级别”。
- Nginx型：基于nginx-ingress插件实现七层网络访问，后端服务支持ClusterIP和NodePort类型。当后端服务为NodePort类型时，同样需将“服务亲和”选项设置为“节点级别”。

四层转发：

- 节点访问：Nodeport访问方式，是将容器端口映射到节点端口，如果“服务亲和”选择“集群级别”需要经过一次服务转发，无法实现获取客户端源IP，而“节点模式”不经过转发，可以获取客户端源ip。

七层转发（Ingress）

针对七层服务（HTTP/HTTPS协议），需要对应用服务器进行配置，然后使用X-Forwarded-For的方式获取来访者的真实IP地址。

真实的来访者IP会被负载均衡放在HTTP头部的X-Forwarded-For字段，格式如下：

```
X-Forwarded-For: 来访者真实IP, 代理服务器1-IP，代理服务器2-IP, ...
```

当使用此方式获取来访者真实IP时，获取的第一个地址就是来访者真实IP。

说明

- 添加Ingress时，若后端为NodePort类型的Service，需将服务亲和设置为**“节点级别”**，即**spec.externalTrafficPolicy**需设置为**“Local”**，参见**节点访问（NodePort）**。

步骤1 以Nginx工作负载为例，未配置源IP访问前，使用以下命令查看访问日志，其中nginx-c99fd67bb-ghv4q为Pod名称：

```
kubectl logs nginx-c99fd67bb-ghv4q
```

回显如下：

```
...
10.0.0.7 - - [17/Aug/2023:01:30:11 +0000] "GET / HTTP/1.1" 200 19 "http://114.114.114.114:9421/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 Edg/115.0.1901.203" "100.125.**.**"
```

其中**100.125.**.****为负载均衡的地址段，说明流量经过负载均衡转发。

步骤2 前往ELB控制台，开启ELB实例对应监听器的“获取客户端IP”功能。**独享型ELB默认开启源地址透传功能，无需手动开启。**

1. 登录弹性负载均衡ELB的管理控制台。
2. 在管理控制台左上角单击 图标，选择区域和项目。
3. 选择“服务列表 > 网络 > 弹性负载均衡”。
4. 在“负载均衡器”界面，单击需要操作的负载均衡名称。
5. 切换到“监听器”页签，单击需要修改的监听器名称右侧的“编辑”按钮。如果存在修改保护，请在监听器基本信息页面中关闭修改保护后重试。
6. 开启“获取客户端IP”开关。

步骤3 （该步骤仅Nginx Ingress需执行）编辑nginx-ingress插件，在nginx配置参数处，需要添加的配置字段和信息（配置参数范围请参考**社区文档**）。配置完成后，更新插件。

```
{
  "enable-real-ip": "true",
  "forwarded-for-header": "X-Forwarded-For",
  "proxy-real-ip-cidr": "100.125.0.0/16",
  "keep-alive-requests": "100"
}
```

说明

proxy-real-ip-cidr参数为代理服务器的网段。

- 共享型负载均衡的IP地址段 100.125.0.0/16（100.125.0.0/16是负载均衡服务保留地址，其他用户无法分配到该网段内，不会存在安全风险）和高防IP地址段。
- 独享型负载均衡需要添加ELB实例关联的VPC子网网段。

步骤4 重新访问工作负载，并查看新增的访问日志如下：

```
...
10.0.0.7 - - [17/Aug/2023:02:43:11 +0000] "GET / HTTP/1.1" 304 0 "http://114.114.114.114:9421/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 Edg/115.0.1901.203" "124.**.**.**"
```

显示成功获取到客户端源IP。

----结束

节点访问（NodePort）

节点访问（NodePort）类型的Service的服务亲和需选择“**节点级别**”而不是“集群级别”，即Service的**spec.externalTrafficPolicy**需要设置为**Local**。

21.9 存储

21.9.1 存储扩容

CCE节点可进行扩容的存储类型如下：

表 21-10 不同类型的扩容方法

类型	名称	用途	扩容方法
节点磁盘	系统盘	系统盘用于安装操作系统。	**系统盘扩容**
	数据盘	节点必须挂载一块数据盘，供容器引擎和Kubelet组件使用。	• **数据盘扩容——容器引擎空间** • **数据盘扩容——Kubelet空间**
容器存储	Pod容器空间	即容器的basesize设置，每个Pod 占用的磁盘空间设置上限（包含容器镜像占用的空间）。	**Pod容器空间（basesize）扩容**
	PVC	容器中挂载的存储资源。	**PVC扩容**

系统盘扩容

以“EulerOS 2.9”操作系统为例，系统盘“/dev/vda”原有容量50GB，只有一个分区“/dev/vda1”。将系统盘容量扩大至100GB，本示例将新增的50GB划分至已有的“/dev/vda1”分区内。

步骤1 在云硬盘EVS界面对系统盘进行扩容。

步骤2 登录节点，执行命令**growpart**，检查当前系统是否已安装growpart扩容工具。

若回显为工具使用介绍，则表示已安装，无需重复安装。若未安装growpart扩容工具，可执行以下命令安装。

```
yum install cloud-utils-growpart
```

步骤3 执行以下命令，查看系统盘“/dev/vda”的总容量。

```
fdisk -l
```

回显信息如下，系统盘“/dev/vda”的总容量为100GiB：

```
[root@test-48162 ~]# fdisk -l
Disk /dev/vda: 100 GiB, 107374182400 bytes, 209715200 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

```
Disklabel type: dos
Disk identifier: 0x78d88f0b

Device     Boot Start       End   Sectors Size Id Type
/dev/vda1  *     2048 104857566 104855519  50G 83 Linux

Disk /dev/vdb: 100 GiB, 107374182400 bytes, 209715200 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk /dev/mapper/vgpaas-dockersys: 90 GiB, 96632569856 bytes, 188735488 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk /dev/mapper/vgpaas-kubernetes: 10 GiB, 10733223936 bytes, 20963328 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

步骤4 执行以下命令，查看系统盘分区“/dev/vda1”的容量。

```
df -TH
```

回显信息如下：

```
[root@test-48162 ~]# df -TH
Filesystem                 Type      Size  Used Avail Use% Mounted on
devtmpfs                   devtmpfs  1.8G     0  1.8G   0% /dev
tmpfs                      tmpfs     1.8G     0  1.8G   0% /dev/shm
tmpfs                      tmpfs     1.8G   13M  1.8G   1% /run
tmpfs                      tmpfs     1.8G     0  1.8G   0% /sys/fs/cgroup
/dev/vda1                  ext4       53G  3.3G   47G   7% /
tmpfs                      tmpfs     1.8G   75M  1.8G   5% /tmp
/dev/mapper/vgpaas-dockersys  ext4       95G  1.3G   89G   2% /var/lib/docker
/dev/mapper/vgpaas-kubernetes ext4       11G   39M   10G   1% /mnt/paas/kubernetes/kubelet
...
```

步骤5 执行以下命令，指定系统盘待扩容的分区，通过growpart进行扩容。

```
growpart 系统盘 分区编号
```

命令示例（系统盘只有1个分区“/dev/vda1”，因此分区编号为1）：

```
growpart /dev/vda 1
```

回显信息如下：

```
CHANGED: partition=1 start=2048 old: size=104855519 end=104857567 new: size=209713119
end=209715167
```

步骤6 执行以下命令，扩展磁盘分区文件系统的大小。

```
resize2fs 磁盘分区
```

命令示例：

```
resize2fs /dev/vda1
```

回显信息如下：

```
resize2fs 1.45.6 (20-Mar-2020)
Filesystem at /dev/vda1 is mounted on /; on-line resizing required
old_desc_blocks = 7, new_desc_blocks = 13
The filesystem on /dev/vda1 is now 26214139 (4k) blocks long.
```

步骤7 执行以下命令，查看扩容后系统盘分区“/dev/vda1”的容量。

```
df -TH
```

回显类似如下信息：

```
[root@test-48162 ~]# df -TH
Filesystem                 Type      Size  Used Avail Use% Mounted on
devtmpfs                   devtmpfs  1.8G     0  1.8G   0% /dev
tmpfs                      tmpfs     1.8G     0  1.8G   0% /dev/shm
tmpfs                      tmpfs     1.8G   13M  1.8G   1% /run
tmpfs                      tmpfs     1.8G     0  1.8G   0% /sys/fs/cgroup
/dev/vda1                   ext4     106G  3.3G   98G   4% /
tmpfs                      tmpfs     1.8G   75M  1.8G   5% /tmp
/dev/mapper/vgpaas-dockersys  ext4      95G  1.3G   89G   2% /var/lib/docker
/dev/mapper/vgpaas-kubernetes ext4      11G   39M   10G   1% /mnt/paas/kubernetes/kubelet
...
```

步骤8 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

----结束

数据盘扩容——容器引擎空间

CCE 将数据盘空间默认划分为两块：一块用于存放容器引擎 (Docker/Containerd) 工作目录、容器镜像的数据和镜像元数据；另一块用于Kubelet组件和EmptyDir临时存储等。容器引擎空间的剩余容量将会影响镜像下载和容器的启动及运行。下面将以Docker为例，进行容器引擎空间扩容。

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。

```
# lsblk
NAME              MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                 8:0    0   50G  0 disk
└─sda1              8:1    0   50G  0 part /
sdb                 8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker              # 容器引擎使用的空间
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
```

  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

- Devicemapper，单独划分了thinpool存储镜像相关数据。

```
# lsblk
NAME                          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda                             8:0    0   50G  0 disk
└─sda1                          8:1    0   50G  0 part /
sdb                             8:16   0  200G  0 disk
├─vgpaas-dockersys              253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta         253:1    0    3G  0 lvm
│ └─vgpaas-thinpool             253:3    0   67G  0 lvm                  # thinpool空间
│   ...
├─vgpaas-thinpool_tdata         253:2    0   67G  0 lvm
│ └─vgpaas-thinpool             253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes             253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  – 在节点上执行如下命令， 将新增的磁盘容量加到thinpool盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/thinpool
```

- 在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----结束

数据盘扩容——Kubelet 空间

CCE 将数据盘空间默认划分为两块：一块用于存放容器引擎 (Docker/Containerd) 工作目录、容器镜像的数据和镜像元数据；另一块用于Kubelet组件和EmptyDir临时存储等。您可参考以下步骤进行Kubelet空间扩容。

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 然后在节点上执行如下命令， 将新增的磁盘容量加到Kubernetes盘上。

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/kubernetes
resize2fs /dev/vgpaas/kubernetes
```

----结束

Pod 容器空间（basesize）扩容

步骤1 登录CCE控制台，单击集群列表中的集群名称。

步骤2 在左侧导航栏中选择“节点管理”。

步骤3 选择集群中的节点，单击操作列中的“更多 > 重置节点”。

须知

重置节点操作可能导致与节点有绑定关系的资源（本地存储，指定调度节点的负载等）无法正常使用。请谨慎操作，避免对运行中的业务造成影响。

步骤4 重新配置节点参数。

如需对容器存储空间进行调整，请重点关注以下配置。

存储配置：单击数据盘后方的“展开高级设置”可进行如下设置：

- 自定义容器引擎空间大小：容器引擎占用的存储空间，默认为数据盘空间的90%，用于存放容器引擎 (Docker/Containerd) 工作目录、容器镜像的数据和镜像元数据。
- 自定义容器Pod空间大小：CCE 支持对每个工作负载下的容器组 Pod 占用的磁盘空间设置上限（包含容器镜像占用的空间）。合理的配置可避免容器组无节制使用磁盘空间导致业务异常。建议此值不超过容器引擎空间的 80%。

说明

- 自定义容器Pod存储空间的能力与节点操作系统与容器存储Rootfs有关，设置规则如下：
  - 容器存储Rootfs使用DeviceMapper时，节点支持自定义容器Pod空间设置（basesize），单个容器存储空间大小默认为10GiB，可以配置为其他值。
  - 容器存储Rootfs使用OverlayFS时，大部分节点不支持自定义容器Pod空间设置（basesize），默认为不限制，即单个容器存储空间大小默认为容器引擎空间。

    仅1.19.16版本、1.21.3版本、1.23.3版本及之后版本集群中的EulerOS 2.9系统节点支持自定义容器Pod空间设置（basesize），可以配置为其他值。
- 使用EulerOS 2.9 的docker basesize设置时，若容器配置CAP_SYS_RESOURCE权限或privileged的特权，basesize限制单容器数据空间不起作用。

步骤5 重置节点后登录该节点，执行如下命令进入容器，查看docker容器容量是否已扩容。

docker exec -it *container_id* **/bin/sh**或**kubectl exec -it** *container_id* **/bin/sh**

df -h

```
# df -h
Filesystem                                                                                      Size  Used Avail Use% Mounted on
/dev/mapper/docker-253:1-787293-631c1bde2cbe82e39f32253b216ba914cb183b168b54700b3e5b9a54ee40a0d1   15G  229M   15G   2% /
tmpfs                                                                                           32G     0   32G   0% /dev
tmpfs                                                                                           32G     0   32G   0% /sys/fs/cgroup
/dev/mapper/vgpaas-kubernetes                                                                  9.8G   37M  9.2G   1% /etc/hosts
/dev/vda1                                                                                       40G  5.2G   33G  14% /etc/hostname
shm                                                                                             64M     0   64M   0% /dev/shm
tmpfs                                                                                           32G   16K   32G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                                                                                           32G     0   32G   0% /proc/acpi
tmpfs                                                                                           32G     0   32G   0% /sys/firmware
tmpfs                                                                                           32G     0   32G   0% /proc/scsi
tmpfs                                                                                           32G     0   32G   0% /proc/kbox
tmpfs                                                                                           32G     0   32G   0% /proc/oom_extend
```

----结束

PVC 扩容

对于云存储：

- 对象存储及文件存储SFS：无存储限制，无需扩容。
- 云硬盘：
  - 对于自动创建的按需收费实例，可以通过直接提供控制台进行扩容。参考步骤如下：
    i. 在左侧导航栏选择“存储”，在右侧选择“存储卷声明”页签。单击PVC操作列的“更多 > 扩容”。
    ii. 输入新增容量，并单击“确定”。
- 极速文件存储SFS Turbo：需要先在SFS控制台扩容，然后再修改PVC中容量大小。

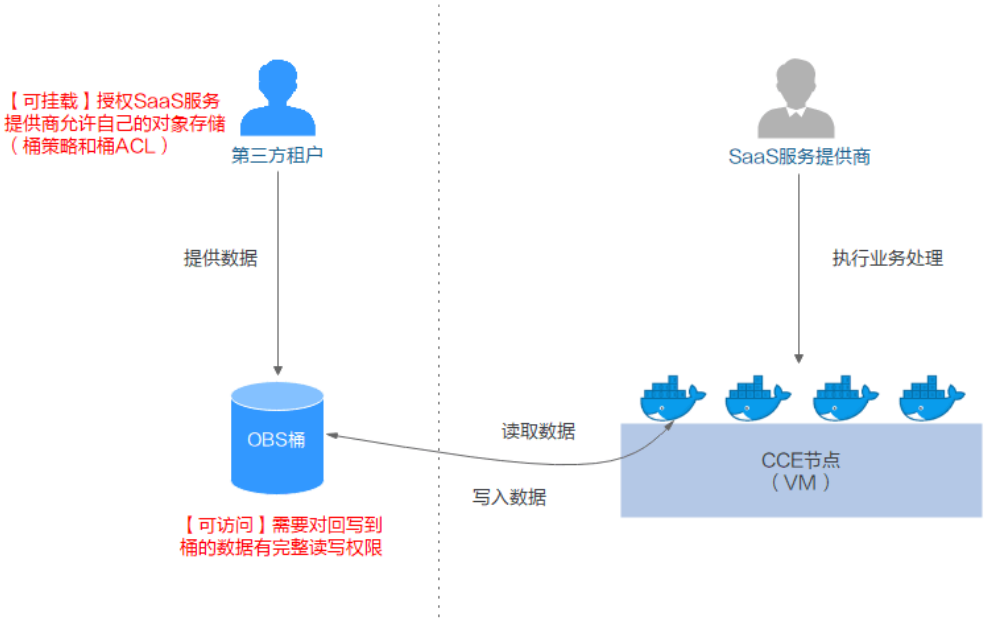
21.9.2 挂载第三方租户的对象存储

本章节介绍如何挂载第三方租户的OBS桶，包含OBS并行文件系统（优先）和OBS对象桶。

使用场景

SaaS服务提供商的CCE集群需要挂载使用第三方租户的OBS桶，使用场景如图21-17所示。

图 21-17 挂载第三方租户的对象存储使用场景



1. **第三方租户授权SaaS服务提供商访问所拥有的对象存储**，第三方租户对SaaS服务提供商需要访问的OBS对象桶或者并行文件系统设置桶策略和桶ACL。
2. **SaaS服务提供商静态导入第三方OBS对象桶和并行文件系统**。
3. SaaS服务提供商进行业务处理，最终需要将处理结果（结果文件或者结果数据）写回第三方租户的桶。

使用须知

- 仅支持挂载同一区域下的第三方租户的并行文件系统和对象桶。
- 仅已安装1.1.11及以上版本Everest存储插件的集群（集群版本需v1.15及以上），支持挂载第三方租户的OBS桶。
- CCE集群所在的SaaS服务提供商的业务平台需要管理第三方桶PV的生命周期，单独删除PVC时PV不会被关联删除而被保留（Kubernetes原生PV Retain回收策略的效果），需要调用Kubernetes原生接口进行静态PV的创建和删除。

第三方租户授权 SaaS 服务提供商访问所拥有的对象存储

以授权访问OBS对象桶为例介绍如何设置桶策略和桶ACL，OBS并行文件系统的设置方法和对象桶相同。

步骤1 登录OBS控制台。

步骤2 在桶列表单击待操作的桶，进入“概览”页面。

步骤3 在左侧导航栏，单击“访问权限控制”，在右侧选择桶策略，在自定义策略下单击“创建桶策略”。

- 策略类型：选择“自定义”。
- 效力：设置为“允许”。
- 被授权用户：选择包含 > 云服务用户，输入账号ID和用户ID，桶策略对指定的用户生效。

- 资源：选择允许操作的资源。
- 动作：勾选可以操作的动作。

步骤4 在左侧导航栏，单击“访问权限控制”，在右侧选择“桶ACL”，单击“增加”，输入授权用户的账号ID或账号名，桶访问权限勾选“读取权限”和“写入权限”，ACL访问权限勾选“读取权限”和“写入权限”，单击“保存”。

----结束

SaaS 服务提供商静态导入第三方 OBS 对象桶和并行文件系统

- **OBS对象桶静态PV：**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: objbucket      #名称替换为实际的对象桶PV名称
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  mountOptions:
  - default_acl=bucket-owner-full-control      #新增的OBS挂载参数
  csi:
    driver: obs.csi.everest.io
    fsType: s3fs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region:       #设置为当前区域id
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: objbucket            #名称替换为实际的第三方租户对象桶名称
  persistentVolumeReclaimPolicy: Retain    #必须设置为Retain，删除PV时保留桶实际也无法删除
  storageClassName: csi-obs-mountoption    #可以关联某个新增自定义OBS StorageClass, 也可关联集群自带的csi-obs
```

  - mountOptions：新增的OBS挂载参数，允许桶所有者对桶中数据有完整的访问权限，解决挂载第三方桶后写入数据、桶所有者无法读取的问题。挂载第三方租户的对象存储使用场景下，default_acl必须设置为**bucket-owner-full-control**。
  - persistentVolumeReclaimPolicy：挂载第三方租户的对象存储使用场景下，persistentVolumeReclaimPolicy必须设置为Retain。删除PV时保留桶实际也无法删除，因此CCE集群所在的SaaS服务提供商的业务平台需要管理第三方桶PV的生命周期，单独删除PVC时PV不会被关联删除而被保留（Kubernetes原生PV Retain回收策略的效果），需要调用Kubernetes原生接口进行静态PV的创建和删除。
  - storageClassName：可以关联某个新增自定义OBS StorageClass（**创建自定义OBS StorageClass**），也可关联集群自带的csi-obs。

绑定的OBS对象桶PVC：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/fstype: obsfs
    everest.io/obs-volume-type: STANDARD
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: objbucketpvc      #名称替换为实际的对象桶PVC名称
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
```

```
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs-mountoption     #与绑定的PV关联的StorageClass保持一致
  volumeName: objbucket       #名称替换为实际需要绑定的对象桶PV名称
```

- **OBS并行文件系统静态PV：**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: obsfscheck   #名称替换为实际的并行文件系统PV名称
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  mountOptions:
  - default_acl=bucket-owner-full-control     #新增的OBS挂载参数
  csi:
    driver: obs.csi.everest.io
    fsType: obsfs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region:
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: obsfscheck              #名称替换为实际的三方租户的并行文件系统名称
  persistentVolumeReclaimPolicy: Retain       #必须设置为Retain，删除PV时保留桶实际也无法删除
  storageClassName: csi-obs-mountoption       #可以关联某个新增自定义OBS StorageClass，也可关联集群自带的csi-obs
```

  - mountOptions：新增的OBS挂载参数，允许桶所有者对桶中数据有完整的访问权限，解决挂载第三方桶后写入数据、桶所有者无法读取的问题。挂载第三方租户的对象存储使用场景下，default_acl必须设置为**bucket-owner-full-control**。
  - persistentVolumeReclaimPolicy：挂载第三方租户的对象存储使用场景下，persistentVolumeReclaimPolicy必须设置为Retain。删除PV时保留桶实际也无法删除，因此CCE集群所在的SaaS服务提供商的业务平台需要管理第三方桶PV的生命周期，单独删除PVC时PV不会被关联删除而被保留（Kubernetes原生PV Retain回收策略的效果），需要调用Kubernetes原生接口进行静态PV的创建和删除。
  - storageClassName：可以关联某个新增自定义OBS StorageClass（**创建自定义OBS StorageClass**），也可关联集群自带的csi-obs。

  绑定的OBS并行文件系统PVC：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/fstype: obsfs
    everest.io/obs-volume-type: STANDARD
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: obsfscheckpvc     #名称替换为实际的并行文件系统PVC名称
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs-mountoption     #与绑定的PV关联的StorageClass保持一致
  volumeName: obsfscheck     #名称替换为实际的并行文件系统PV名称
```

- **创建自定义OBS StorageClass，用以关联静态PV（可选）：**

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

```
metadata:
  name: csi-obs-mountoption
mountOptions:
  - default_acl=bucket-owner-full-control
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: obsfs
  everest.io/obs-volume-type: STANDARD
provisioner: everest-csi-provisioner
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

- csi.storage.k8s.io/fstype：指定文件类型，支持“obsfs”与“s3fs”。取值为s3fs时说明使用的obs对象桶，配套使用s3fs挂载；取值为obsfs时说明使用的是obs并行文件系统，配套使用obsfs挂载。
- reclaimPolicy：指定创建的Persistent Volume的回收策略。该取值会设置到基于新的PVC关联该SC动态创建的PV.spec.persistentVolumeReclaimPolicy中。若设置为Delete，删除PVC时会触发关联删除外部OBS对象和该PV；若设置为Retain，删除PVC时会保留PV和外部存储，需要单独清理PV。对于使用导入关联三方桶对应的使用场景，该SC只做关联静态PV使用（**需要设置为Retain**），不涉及使用动态创建。

21.9.3 SFS Turbo 动态创建子目录并挂载

背景信息

SFS Turbo容量最小500G，且不是按使用量计费。SFS Turbo挂载时默认将根目录挂载到容器，而通常情况下负载不需要这么大容量，造成浪费。

everest插件支持一种在SFS Turbo下动态创建子目录的方法，能够在SFS Turbo下动态创建子目录并挂载到容器，这种方法能够共享使用SFS Turbo，从而更加经济合理的利用SFS Turbo存储容量。

约束与限制

- 仅支持1.15+集群。
- 集群必须使用everest插件，插件版本要求1.1.13+。
- 使用everest 1.2.69之前或2.1.11之前的版本时，使用子目录功能时不能同时并发创建超过10个PVC。推荐使用everest 1.2.69及以上或2.1.11及以上的版本。
- Ubuntu操作系统的节点使用Docker容器引擎时不支持该功能。

创建 subpath 类型 SFS Turbo 存储卷

⚠ 注意

subpath模式的卷请勿通过前端进行“扩容”、“解关联”、“删除”等操作。

步骤1 创建SFS Turbo资源，选择网络时，请选择与集群相同的VPC与子网。

步骤2 新建一个StorageClass的YAML文件，例如sfsturbo-subpath-sc.yaml。

配置示例：

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
```

```
kind: StorageClass
metadata:
  name: sfsturbo-subpath-sc
mountOptions:
- lock
parameters:
  csi.storage.k8s.io/csi-driver-name: sfsturbo.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/archive-on-delete: "true"
  everest.io/share-access-to: 7ca2dba2-1234-1234-1234-626371a8fb3a
  everest.io/share-expand-type: bandwidth
  everest.io/share-export-location: 192.168.1.1:/sfsturbo/
  everest.io/share-source: sfs-turbo
  everest.io/share-volume-type: STANDARD
  everest.io/volume-as: subpath
  everest.io/volume-id: 0d773f2e-1234-1234-1234-de6a35074696
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

其中：

- name：storageclass的名称。
- mountOptions：选填字段；mount挂载参数。
  - everest 1.2.8以下，1.1.13以上版本仅开放对nolock参数配置，mount操作默认使用nolock参数，无需配置。nolock=false时，使用lock参数。
  - everest 1.2.8及以上版本支持更多参数，默认使用如下所示配置。**此处不能配置为nolock=true，会导致挂载失败。**

    ```
    mountOptions:
    - vers=3
    - timeo=600
    - nolock
    - hard
    ```
- everest.io/volume-as：该参数需设置为“subpath”来使用subpath模式。
- everest.io/share-access-to：选填字段。subpath模式下，填写SFS Turbo资源的所在VPC的ID。
- everest.io/share-expand-type：选填字段。若SFS Turbo资源存储类型为增强版（标准型增强版、性能型增强版），设置为bandwidth。
- everest.io/share-export-location：挂载目录配置。由SFS Turbo共享路径和子目录组成，共享路径可至SFS Turbo服务页面查询，子路径由用户自定义，后续指定该StorageClass创建的PVC均位于该子目录下。
- everest.io/share-volume-type：选填字段。填写SFS Turbo的类型。标准型为STANDARD，性能型为PERFORMANCE。对于增强型需配合“everest.io/share-expand-type”字段使用，everest.io/share-expand-type设置为“bandwidth”。
- everest.io/zone：选填字段。指定SFS Turbo资源所在的可用区。
- everest.io/volume-id：SFS Turbo资源的卷ID，可至SFS Turbo界面查询。
- everest.io/archive-on-delete：若该参数设置为“true”，在回收策略为“Delete”时，删除PVC会将PV的原文档进行归档，归档目录的命名规则“archived-$pv名称.时间戳”。该参数设置为“false”时，会将PV对应的SFS Turbo子目录删除。默认设置为“true”，即删除PVC时进行归档。

步骤3 执行**kubectl create -f sfsturbo-subpath-sc.yaml**。

步骤4 新建一个PVC的YAML文件，sfs-turbo-test.yaml。

配置示例：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sfs-turbo-test
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
  storageClassName: sfsturbo-subpath-sc
  volumeMode: Filesystem
```

其中：

- name: PVC的名称。
- storageClassName：SC的名称。
- storage：subpath模式下，该参数无实际意义，容量受限于SFS Turbo资源的总容量，若SFS Turbo资源总容量不足，请及时到SFS Turbo界面扩容。

步骤5 执行**kubectl create -f sfs-turbo-test.yaml**。

----结束

说明

对subpath类型的SFS Turbo扩容时，没有实际的扩容意义。该操作不会对SFS Turbo资源进行实际的扩容，需要用户自行保证SFS Turbo的总容量不被耗尽。

创建 Deployment 挂载已有数据卷

步骤1 新建一个Deployment的YAML文件，例如deployment-test.yaml。

配置示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-turbo-subpath-example
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-turbo-subpath-example
  template:
    metadata:
      labels:
        app: test-turbo-subpath-example
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        volumeMounts:
        - mountPath: /tmp
          name: pvc-sfs-turbo-example
      restartPolicy: Always
      imagePullSecrets:
      - name: default-secret
      volumes:
      - name: pvc-sfs-turbo-example
```

```
      persistentVolumeClaim:
        claimName: sfs-turbo-test
```

其中：

- name：创建的工作负载名称。
- image：工作负载的镜像。
- mountPath：容器内挂载路径，示例中挂载到“/tmp”路径。
- claimName：已有的PVC名称。

步骤2 创建Deployment负载。

kubectl create -f deployment-test.yaml

----结束

StatefulSet 动态创建 subpath 模式的数据卷

步骤1 新建一个StatefulSet的YAML文件，例如statefulset-test.yaml。

配置示例：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: test-turbo-subpath
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test-turbo-subpath
  template:
    metadata:
      labels:
        app: test-turbo-subpath
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
        pod.alpha.kubernetes.io/initialized: 'true'
    spec:
      containers:
        - name: container-0
          image: 'nginx:latest'
          resources: {}
          volumeMounts:
            - name: sfs-turbo-160024548582479676
              mountPath: /tmp
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: default-secret
      affinity: {}
      schedulerName: default-scheduler
  volumeClaimTemplates:
    - metadata:
        name: sfs-turbo-160024548582479676
        namespace: default
```

```
      annotations: {}
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
      storageClassName: sfsturbo-subpath-sc
serviceName: wwww
podManagementPolicy: OrderedReady
updateStrategy:
  type: RollingUpdate
revisionHistoryLimit: 10
```

其中：

- name：创建的工作负载名称。
- image：工作负载的镜像。
- mountPath：容器内挂载路径，示例中挂载到“/tmp”路径。
- “spec.template.spec.containers.volumeMounts.name ”和“spec.volumeClaimTemplates.metadata.name”有映射关系，必须保持一致。
- storageClassName: 填写自建的SC名称。

步骤2 创建StatefulSet负载。

kubectl create -f statefulset-test.yaml

----结束

21.9.4 1.15 集群如何从 Flexvolume 存储类型迁移到 CSI Everest 存储类型

在v1.15.11-r1之后版本的集群中，CSI Everest插件已接管fuxi Flexvolume（即storage-driver插件）容器存储的所有功能，建议将对fuxi Flexvolume的使用切换CSI Everest上。

迁移的主要原理是通过创建静态PV的形式关联原有底层存储，并创建新的PVC关联该新建的静态PV，之后应用升级挂载这个新的PVC到原有挂载路径，实现存储卷迁移。

⚠ 警告

迁移时会造成服务断服，请合理规划迁移时间，并做好相关备份。

操作步骤

步骤1 数据备份（可选，主要防止异常情况下数据丢失）。

步骤2 根据FlexVolume格式的PV，准备CSI格式的PV的yaml文件关联已有存储。

执行如下命令，配置名为“pv-example.yaml”的创建PV的yaml文件。

touch pv-example.yaml

vi pv-example.yaml

云硬盘存储卷PV的配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    failure-domain.beta.kubernetes.io/region: eu-west-0
    failure-domain.beta.kubernetes.io/zone: <zone name>
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
  name: pv-evs-example
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 10Gi
  csi:
    driver: disk.csi.everest.io
    fsType: ext4
    volumeAttributes:
      everest.io/disk-mode: SCSI
      everest.io/disk-volume-type: SAS
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 0992dbda-6340-470e-a74e-4f0db288ed82
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-disk
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-11 云硬盘存储卷 PV 配置参数说明

参数	描述
failure-domain.beta.kubernetes.io/region	云硬盘所在region，可参考FlexVolume PV的相同字段。
failure-domain.beta.kubernetes.io/zone	云硬盘所在可用区，可参考FlexVolume PV的相同字段。
name	PV资源的名称，集群下唯一。
storage	云硬盘的容量，单位为Gi。可参考FlexVolume PV的spec.capacity.storage。
driver	挂载依赖的存储驱动，EVS云硬盘配置为“disk.csi.everest.io”。
volumeHandle	云硬盘的volumeID，可参考FlexVolume PV的spec.flexVolume.options.volumeID。
everest.io/disk-mode	云硬盘磁盘模式，可参考FlexVolume PV的spec.flexVolume.options.disk-mode。
everest.io/disk-volume-type	云硬盘类型。可参考FlexVolume PV的spec.storageClassName对应的sc中的parameters."kubernetes.io/volumetype"。
storageClassName	存储卷动态供应关联的K8s storage class名称；云硬盘需使用“csi-disk”。

文件存储卷PV配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-sfs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: nas.csi.everest.io
    fsType: nfs
    volumeAttributes:
      everest.io/share-export-location:  #文件存储的共享路径
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 682f00bb-ace0-41d8-9b3e-913c9aa6b695
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-nas
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-12 文件存储卷 PV 配置参数说明

参数	描述
name	PV资源的名称，集群下唯一。
storage	文件存储的大小，单位为Gi。可参考FlexVolume PV的spec.capacity.storage。
driver	挂载依赖的存储驱动，文件存储配置为“nas.csi.everest.io”。
everest.io/share-export-location	文件存储的共享路径。可参考FlexVolume PV的spec.flexVolume.options.deviceMountPath。
volumeHandle	文件存储的ID。可参考FlexVolume PV的spec.flexVolume.options.volumeID。
storageClassName	K8s storage class名称；需配置为"csi-nas"。

对象存储卷PV配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-obs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  csi:
    driver: obs.csi.everest.io
    fsType: s3fs
    volumeAttributes:
```

```
    everest.io/obs-volume-type: STANDARD
    everest.io/region: eu-west-0
    storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
  volumeHandle: obs-normal-static-pv
persistentVolumeReclaimPolicy: Delete
storageClassName: csi-obs
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-13 对象存储卷 PV 配置参数说明

参数	描述
name	PV资源的名称，集群下唯一。
storage	存储容量，单位为Gi。此处配置为固定值1Gi。
driver	挂载依赖的存储驱动，对象存储配置为“obs.csi.everest.io”。
fsType	文件类型，支持“obsfs”与“s3fs”，取值为s3fs时创建是obs对象桶，配套使用s3fs挂载；取值为obsfs时创建的是obs并行文件系统，配套使用obsfs挂载。可参考FlexVolume PV的spec.flexVolume.options.posix的对应关系：true（obsfs）、false/空值（s3fs）。
everest.io/obs-volume-type	存储类型，包括STANDARD（标准桶）、WARM（低频访问桶）。可参考FlexVolume PV的spec.flexVolume.options.storage_class的对应关系：standard（标准桶）、standard_ia（低频访问桶）。
everest.io/region	对象存储所在的region。可参考FlexVolume PV的spec.flexVolume.options.region。
volumeHandle	对象存储的桶名称。可参考FlexVolume PV的spec.flexVolume.options.volumeID。
storageClassName	K8s storage class名称；需配置为"csi-obs”。

极速文件存储卷PV配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-efs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: sfsturbo.csi.everest.io
    fsType: nfs
    volumeAttributes:
      everest.io/share-export-location: 192.168.0.169:/
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 8962a2a2-a583-4b7f-bb74-fe76712d8414
```

```
persistentVolumeReclaimPolicy: Delete
storageClassName: csi-sfsturbo
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-14 极速文件存储卷 PV 配置参数说明

参数	描述
name	PV资源的名称，集群下唯一。
storage	文件存储的大小。可参考FlexVolume PV的spec.capacity.storage。
driver	挂载依赖的存储驱动，极速文件存储配置为“sfsturbo.csi.everest.io”。
everest.io/share-export-location	极速文件存储的共享路径。可参考FlexVolume PV的spec.flexVolume.options.deviceMountPath。
volumeHandle	极速文件存储的ID。可参考FlexVolume PV的spec.flexVolume.options.volumeID。
storageClassName	指定K8s storage class名称；极速文件存储卷需配置为"csi-sfsturbo”。

步骤3 根据FlexVolume格式的PVC，准备CSI格式的PVC的yaml文件关联上述步骤准备的静态PV。

执行如下命令，配置名为“pvc-example.yaml”的创建PVC的yaml文件。

touch pvc-example.yaml

vi pvc-example.yaml

云硬盘存储卷PVC的配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    failure-domain.beta.kubernetes.io/region: eu-west-0
    failure-domain.beta.kubernetes.io/zone: <zone name>
  annotations:
    everest.io/disk-volume-type: SAS
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-evs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeName:  pv-evs-example
  storageClassName: csi-disk
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-15 云硬盘存储卷 PVC 配置参数说明

参数	描述
failure-domain.beta.kubernetes.io/region	集群所在region。可参考FlexVolume PVC的相同字段。
failure-domain.beta.kubernetes.io/zone	EVS云硬盘所在可用区。可参考FlexVolume PVC的相同字段。
everest.io/disk-volume-type	云硬盘存储类型，支持SAS、SSD。和上述步骤的PV保持一致。
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storage	PVC申请容量，必须和已有PV的storage大小保持一致。
volumeName	PV的名称。使用上述步骤的静态PV的名称。
storageClassName	指定K8s storage class名称；云硬盘需使用“csi-disk”。

文件存储卷PVC配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-sfs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas
  volumeName: pv-sfs-example
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-16 文件存储卷 PVC 配置参数说明

参数	描述
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storage	存储容量，单位Gi，必须和已有pv的storage大小保持一致。

参数	描述
storageClassName	需配置为"csi-nas"。
volumeName	PV的名称。参考上述步骤的静态PV的名称。

对象存储卷PVC配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/obs-volume-type: STANDARD
    csi.storage.k8s.io/fstype: s3fs
  name: pvc-obs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs
  volumeName: pv-obs-example
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-17 对象存储卷 PVC 配置参数说明

参数	描述
everest.io/obs-volume-type	obs存储类型；当前支持标准（STANDARD）和低频（WARM）两种存储类型。和上述步骤的PV保持一致。
csi.storage.k8s.io/fstype	指定文件类型，支持“obsfs”与“s3fs”。与上述步骤中静态obs存储的PV的fsType保持一致。
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storage	存储容量，单位为Gi。此处配置为固定值1Gi。
storageClassName	K8s storage class名称；需配置为"csi-obs”。
volumeName	PV的名称。参考上述步骤创建的静态PV的名称。

极速文件存储卷PVC配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
```

```
  name: pvc-efs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-sfsturbo
  volumeName: pv-efs-example
```

加粗标红字段需要重点关注，其中参数说明如下：

表 21-18 极速文件存储卷 PVC 配置参数说明

参数	描述
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storageClassName	指定K8s storage class名称；需配置为"csi-sfsturbo”。
storage	存储容量，单位Gi，必须和已有pv的storage大小保持一致。
volumeName	PV的名称。参考上述步骤创建的静态PV的名称。

步骤4 应用升级替换成新的PVC。

无状态应用

1. 通过kubectl create -f的形式创建pv和pvc。

   kubectl create -f pv-example.yaml

   kubectl create -f pvc-example.yaml

   说明

   命令中的yaml名称是示例，请以实际步骤**步骤2**和步骤**步骤3**创建的pv和pvc的yaml名字为准。

2. 进入应用更新升级界面：更新升级 - 高级设置 - 数据存储 - 云存储。
3. 卸载老存储，同时添加CSI格式的PVC的云存储，容器内挂载路径和以前保持一致，实现存储迁移。
4. 单击提交，确认后升级生效。
5. 等待pod running。

升级使用已有存储的有状态应用

1. 通过kubectl create -f的形式创建pv和pvc

   kubectl create -f pv-example.yaml

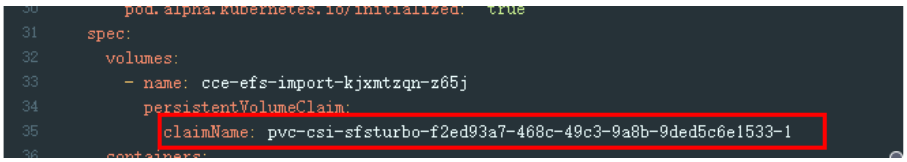
   kubectl create -f pvc-example.yaml

   说明

   命令中的yaml名称是示例，请以实际步骤**步骤2**和步骤**步骤3**创建的pv和pvc的yaml名字为准。

2. 通过kubectl edit的方式修改有状态应使用新建的PVC。

   kubectl edit sts sts-example -n xxx



   说明

   命令中的sts-example为待升级的有状态应用的名称，请以实际为准。xxx指代有状态应用所在的命名空间。

3. 等待pod running。

说明

当前界面暂未提供有状态应用添加新的云存储，因此升级替换成新PVC需要通过后台kubectl命里实现。

升级使用动态分配存储的有状态应用

1. 备份当前有状态应用使用的flexVolume格式的PV和PVC。

   kubectl get pvc xxx -n {namespaces} -oyaml > pvc-backup.yaml

   kubectl get pv xxx -n {namespaces} -oyaml > pv-backup.yaml

2. 将应用的实例数修改成0。
3. 在存储界面解关联有状态应用使用的flexVolume格式的PVC。
4. 通过kubectl create -f的形式创建pv和pvc。

   kubectl create -f pv-example.yaml

   kubectl create -f pvc-example.yaml

   说明

   命令中的yaml名称是示例，请以实际步骤**步骤2**和步骤**步骤3**创建的pv和pvc的yaml名字为准。

5. 将应用的实例数恢复，等待pod running。

说明

有状态应用动态创建存储是通过volumeClaimTemplates机制实现，而该字段K8s无法修改，因此无法通过更换新PVC的方式实现数据迁移。

volumeClaimTemplates的PVC命名格式是固定的，当符合命名格式的PVC已经存在的时候则直接使用该PVC。

因此需要些解关联原有PVC之后，创建同名的CSI格式的PVC来实现存储迁移。

6. 迁移完成，但是如果不重建有状态应用，扩容时仍是FlexVolume格式的PVC（按需操作）。

- 获取当前有状态应用yaml：

kubectl get sts xxx -n {namespaces} -oyaml > sts.yaml

- 备份当前有状态应用yaml：

cp sts.yaml sts-backup.yaml

- 修改有状态应用yaml中volumeClaimTemplates的定义：

vi sts.yaml

云硬盘存储卷volumeClaimTemplates的配置示例如下：

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161070049798261342
      namespace: default
      creationTimestamp: null
      annotations:
        everest.io/disk-volume-type: SAS
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-disk
```

其中参数和上述步骤创建的云硬盘存储卷的PVC保持一致。

文件存储卷volumeClaimTemplates配置示例如下：

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161063441560279697
      namespace: default
      creationTimestamp: null
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-nas
```

其中参数和上述步骤创建的文件存储卷PVC保持一致。

对象存储卷PVC配置示例如下：

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161070100417416148
      namespace: default
      creationTimestamp: null
      annotations:
        csi.storage.k8s.io/fstype: s3fs
        everest.io/obs-volume-type: STANDARD
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-obs
```

其中参数和上述步骤创建的对象存储卷PVC保持一致。

- 删除原有状态应用：

kubectl delete sts xxx -n {namespaces}

- 创建新的有状态应用

kubectl create -f sts.yaml

步骤5 检查业务功能。

1. 检查业务功能是否正常。
2. 检查数据是否丢失。

说明

若功能或数据检查异常需要回退，请执行步骤**步骤4**，选择FlexVolume格式的PVC并单击提交升级。

步骤6 卸载FlexVolume格式的PVC。

检查正常，存储管理界面执行解关联操作。

也可以后台通过kubectl指令删除Flexvolume格式的PVC和PV。

注意

在删除之前需要修改PV的回收策略persistentVolumeReclaimPolicy为Retain，否则底层存储会被回收。

在存储迁移执行前已完成集群升级可能会导致无法删除PV，可以去除PV的保护字段finalizers来实现PV删除

kubectl patch pv {pv_name} -p '{"metadata":{"finalizers":null}}'

----结束

21.9.5 自定义 StorageClass

应用现状

CCE中使用存储时，最常见的方法是创建PVC时通过指定StorageClassName定义要创建存储的类型，如下所示，使用PVC申请一个SAS（高I/O）类型云硬盘/块存储。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
```

可以看到在CCE中如果需要指定云硬盘的类型，是通过everest.io/disk-volume-type: SAS字段指定，这里SAS是云硬盘的类型，代表高I/O，还有SATA（普通I/O）、SSD（超高I/O）可以指定。

这种写法在如下几种场景下存在问题：

- 部分用户觉得使用everest.io/disk-volume-type指定云硬盘类型比较繁琐，希望只通过StorageClassName指定。
- 部分用户是从自建Kubernetes或其他Kubernetes服务切换到CCE，已经写了很多应用的YAML文件，这些YAML文件中通过不同StorageClassName指定不同类型存

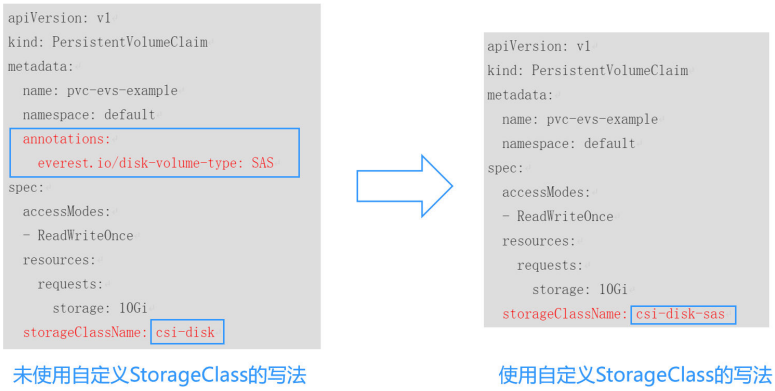
储，迁移到CCE上时，使用存储就需要修改大量YAML文件或Helm Chart包，这非常繁琐且容易出错。

- 部分用户希望能够设置默认的StorageClassName，所有应用都使用默认存储类型，在YAML中不用指定StorageClassName也能按创建默认类型存储。

解决方案

本文介绍在CCE中自定义StorageClass的方法，并介绍设置默认StorageClass的方法，通过不同StorageClassName指定不同类型存储。

- 对于第一个问题：可以将SAS、SSD类型云硬盘分别定义一个StorageClass，比如定义一个名为csi-disk-sas的StorageClass，这个StorageClass创建SAS类型的存储，则前后使用的差异如下图所示，编写YAML时只需要指定StorageClassName，符合特定用户的使用习惯。



未使用自定义StorageClass的写法　　使用自定义StorageClass的写法

- 对于第二个问题：可以定义与用户现有YAML中相同名称的StorageClass，这样可以省去修改YAML中StorageClassName的工作。
- 对于第三个问题：可以设置默认的StorageClass，则YAML中无需指定StorageClassName也能创建存储，按如下写法即可。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

CCE 中默认的 StorageClass

执行如下命令即可查询默认StorageClass。

```
# kubectl get sc
NAME                PROVISIONER                     AGE
csi-disk            everest-csi-provisioner         17d         # 云硬盘 StorageClass
csi-disk-topology   everest-csi-provisioner         17d         # 延迟绑定的云硬盘 StorageClass
csi-nas             everest-csi-provisioner         17d         # 文件存储 StorageClass
csi-obs             everest-csi-provisioner         17d         # 对象存储 StorageClass
csi-sfsturbo        everest-csi-provisioner         17d         # 极速文件存储 StorageClass
```

查看下csi-disk的详情，可以发现csi-disk创建云硬盘的默认类似是SAS。

```
# kubectl get sc csi-disk -oyaml
allowVolumeExpansion: true
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  creationTimestamp: "2021-03-17T02:10:32Z"
  name: csi-disk
  resourceVersion: "760"
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-disk
  uid: 4db97b6c-853b-443d-b0dc-41cdcb8140f2
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

自定义 StorageClass

自定义高I/O类型StorageClass，使用YAML描述如下，这里取名为csi-disk-sas，指定云硬盘类型为SAS，即高I/O。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-sas                      # 高IO StorageClass名字，用户可自定义
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS        # 云硬盘高I/O类型，用户不可自定义
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true                # true表示允许扩容
```

超高I/O类型StorageClass，这里取名为csi-disk-ssd，指定云硬盘类型为SSD，即超高I/O。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd                      # 超高I/O StorageClass名字，用户可自定义
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD        # 云硬盘超高I/O类型，用户不可自定义
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

reclaimPolicy：底层云存储的回收策略，支持Delete、Retain回收策略。

- **Delete**：删除PVC，PV资源与云硬盘均被删除。
- **Retain**：删除PVC，PV资源与底层存储资源均不会被删除，需要手动删除回收。PVC删除后PV资源状态为“已释放（Released）”，不能直接再次被PVC绑定使用。

说明

此处设置的回收策略对SFS Turbo类型的存储无影响。

如果数据安全性要求较高，建议使用**Retain**以免误删数据。

定义完之后，使用kubectl create命令创建。

```
# kubectl create -f sas.yaml
storageclass.storage.k8s.io/csi-disk-sas created
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
```

再次查询StorageClass，回显如下，可以看到多了两个类型的StorageClass。

```
# kubectl get sc
NAME                PROVISIONER                  AGE
csi-disk            everest-csi-provisioner      17d
csi-disk-sas        everest-csi-provisioner      2m28s
csi-disk-ssd        everest-csi-provisioner      16s
csi-disk-topology   everest-csi-provisioner      17d
csi-nas             everest-csi-provisioner      17d
csi-obs             everest-csi-provisioner      17d
csi-sfsturbo        everest-csi-provisioner      17d
```

其他类型存储自定义方法类似，可以使用 kubectl 获取YAML，在YAML基础上根据需要修改。

- 文件存储

  ```
  # kubectl get sc csi-nas -oyaml
  kind: StorageClass
  apiVersion: storage.k8s.io/v1
  metadata:
    name: csi-nas
  provisioner: everest-csi-provisioner
  parameters:
    csi.storage.k8s.io/csi-driver-name: nas.csi.everest.io
    csi.storage.k8s.io/fstype: nfs
    everest.io/share-access-level: rw
    everest.io/share-access-to: 5e3864c6-e78d-4d00-b6fd-de09d432c632   # 集群所在VPC ID
    everest.io/share-is-public: 'false'
    everest.io/zone: xxxxx          # 可用区
  reclaimPolicy: Delete
  allowVolumeExpansion: true
  volumeBindingMode: Immediate
  ```

- 对象存储

  ```
  # kubectl get sc csi-obs -oyaml
  kind: StorageClass
  apiVersion: storage.k8s.io/v1
  metadata:
    name: csi-obs
  provisioner: everest-csi-provisioner
  parameters:
    csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
    csi.storage.k8s.io/fstype: s3fs          # 对象存储文件类型，s3fs是对象桶，obsfs是并行文件系统
    everest.io/obs-volume-type: STANDARD      # OBS桶的存储类别
  reclaimPolicy: Delete
  volumeBindingMode: Immediate
  ```

指定 StorageClass 的企业项目

CCE支持使用存储类创建云硬盘和对象存储类型PVC时指定企业项目，将创建的存储资源（云硬盘和对象存储）归属于指定的企业项目下，**企业项目可选为集群所属的企业项目或default企业项目**。

若不指定企业项目，则创建的存储资源默认使用存储类StorageClass中指定的企业项目，CCE提供的 csi-disk 和 csi-obs 存储类，所创建的存储资源属于default企业项目。

如果您希望通过StorageClass创建的存储资源能与集群在同一个企业项目，则可以自定义StorageClass，并指定企业项目ID，如下所示。

说明

该功能需要everest插件升级到1.2.33及以上版本。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk-epid      # 自定义名称
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/enterprise-project-id: 86bfc701-9d9e-4871-a318-6385aa368183  # 指定企业项目id
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

指定默认 StorageClass

您还可以指定某个StorageClass作为默认StorageClass，这样在创建PVC时不指定StorageClassName就会使用默认StorageClass创建。

例如将csi-disk-ssd指定为默认StorageClass，则可以按如下方式设置。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"   # 指定集群中默认的StorageClass，一个集群中只能有一个默认的StorageClass
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

先删除之前创建的csi-disk-ssd，再使用kubectl create命令重新创建，然后再查询StorageClass，显示如下。

```
# kubectl delete sc csi-disk-ssd
storageclass.storage.k8s.io "csi-disk-ssd" deleted
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
# kubectl get sc
NAME                    PROVISIONER                  AGE
csi-disk                everest-csi-provisioner      17d
csi-disk-sas            everest-csi-provisioner      114m
csi-disk-ssd (default)  everest-csi-provisioner      9s
csi-disk-topology       everest-csi-provisioner      17d
csi-nas                 everest-csi-provisioner      17d
csi-obs                 everest-csi-provisioner      17d
csi-sfsturbo            everest-csi-provisioner      17d
```

配置验证

- 使用csi-disk-sas创建PVC。

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  sas-disk
  ```

```
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk-sas
```

创建并查看详情，如下所示，可以发现能够创建，且StorageClass显示为csi-disk-sas

```
# kubectl create -f sas-disk.yaml
persistentvolumeclaim/sas-disk created
# kubectl get pvc
NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            csi-disk-sas
24s
# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                  STORAGECLASS   REASON   AGE
pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            Delete           Bound       default/
sas-disk         csi-disk-sas            30s
```

在CCE控制台界面上查看PVC详情，在“PV详情”页签下可以看到磁盘类型是高I/O。

- 不指定StorageClassName，使用默认配置，如下所示，并未指定storageClassName。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name:  ssd-disk
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

创建并查看，可以看到PVC ssd-disk的StorageClass为csi-disk-ssd，说明默认使用了csi-disk-ssd。

```
# kubectl create -f ssd-disk.yaml
persistentvolumeclaim/ssd-disk created
# kubectl get pvc
NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            csi-disk-sas
16m
ssd-disk   Bound    pvc-4d2b059c-0d6c-44af-9994-f74d01c78731   10Gi       RWO            csi-disk-ssd
10s
# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                  STORAGECLASS   REASON   AGE
pvc-4d2b059c-0d6c-44af-9994-f74d01c78731   10Gi       RWO            Delete           Bound
default/ssd-disk         csi-disk-ssd            15s
pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            Delete           Bound       default/
sas-disk         csi-disk-sas            17m
```

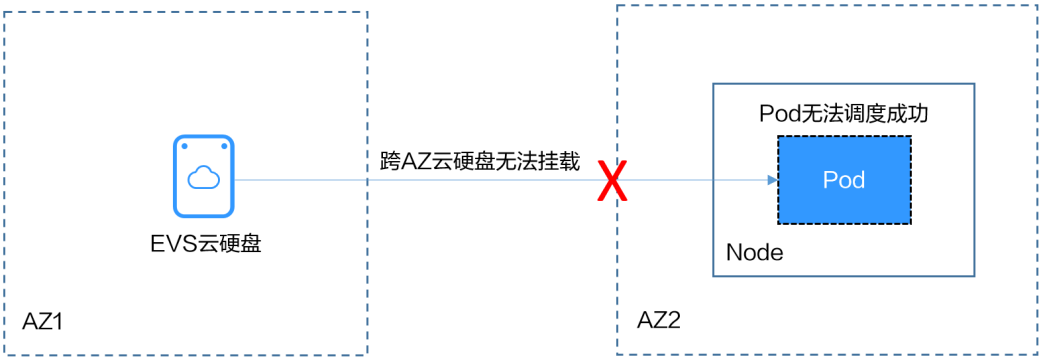
在CCE控制台界面上查看PVC详情，在“PV详情”页签下可以看到磁盘类型是超高I/O。

21.9.6 节点跨 AZ 时云硬盘自动拓扑（csi-disk-topology）

应用现状

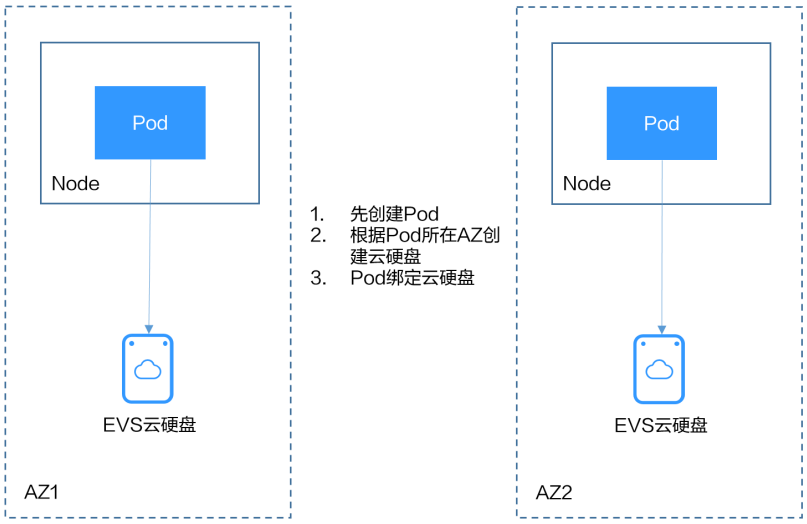
云硬盘使用在使用时无法实现跨AZ挂载，即AZ1的云硬盘无法挂载到AZ2的节点上。有状态工作负载调度时，如果使用csi-disk存储类，会立即创建PVC和PV（创建PV会同

时创建云硬盘），然后PVC绑定PV。但是当集群节点位于多AZ下时，PVC创建的云硬盘可能会与Pod调度到的节点不在同一个AZ，导致Pod无法调度成功。



解决方案

CCE提供了名为csi-disk-topology的StorageClass，也叫延迟绑定的云硬盘存储类型。使用csi-disk-topology创建PVC时，不会立即创建PV，而是等Pod先调度，然后根据Pod调度到节点的AZ信息再创建PV，在Pod所在节点同一个AZ创建云硬盘，这样确保云硬盘能够挂载，从而确保Pod调度成功。



节点多 AZ 情况下使用 csi-disk 导致 Pod 调度失败

创建一个3节点的集群，3个节点在不同AZ下。

使用csi-disk创建一个有状态应用，观察该应用的创建情况。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                    # headless service的名称
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
```

```
    labels:
      app: nginx
  spec:
    containers:
      - name: container-0
        image: nginx:alpine
        resources:
          limits:
            cpu: 600m
            memory: 200Mi
          requests:
            cpu: 600m
            memory: 200Mi
        volumeMounts:                    # Pod挂载的存储
        - name:  data
          mountPath:  /usr/share/nginx/html     # 存储挂载到/usr/share/nginx/html
    imagePullSecrets:
      - name: default-secret
  volumeClaimTemplates:
  - metadata:
      name: data
      annotations:
        everest.io/disk-volume-type: SAS
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-disk
```

有状态应用使用如下Headless Service。

```
apiVersion: v1
kind: Service       # 对象类型为Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Pod间通信的端口名称
      port: 80        # Pod间通信的端口号
  selector:
    app: nginx        # 选择标签为app:nginx的Pod
  clusterIP: None     # 必须设置为None，表示Headless Service
```

创建后查看PVC和Pod状态，如下所示，可以看到PVC都已经创建并绑定成功，而有一个Pod处于Pending状态。

```
# kubectl get pvc -owide
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
AGE   VOLUMEMODE
data-nginx-0   Bound    pvc-04e25985-fc93-4254-92a1-1085ce19d31e   1Gi        RWO            csi-disk
64s   Filesystem
data-nginx-1   Bound    pvc-0ae6336b-a2ea-4ddc-8f63-cfc5f9efe189   1Gi        RWO            csi-disk
47s   Filesystem
data-nginx-2   Bound    pvc-aa46f452-cc5b-4dbd-825a-da68c858720d   1Gi        RWO            csi-disk
30s   Filesystem
data-nginx-3   Bound    pvc-3d60e532-ff31-42df-9e78-015cacb18a0b   1Gi        RWO            csi-disk
14s   Filesystem

# kubectl get pod -owide
NAME      READY   STATUS    RESTARTS   AGE     IP            NODE           NOMINATED NODE   READINESS
GATES
nginx-0   1/1     Running   0          2m25s   172.16.0.12   192.168.0.121   <none>           <none>
nginx-1   1/1     Running   0          2m8s    172.16.0.136   192.168.0.211   <none>           <none>
nginx-2   1/1     Running   0          111s    172.16.1.7     192.168.0.240   <none>           <none>
nginx-3   0/1     Pending   0          95s     <none>         <none>          <none>           <none>
```

查看这个Pod的事件信息，可以发现调度失败，没有一个可用的节点，其中两个节点是因为没有足够的CPU，一个是因为创建的云硬盘不是节点所在的可用区，Pod无法使用该云硬盘。

```
# kubectl describe pod nginx-3
Name:           nginx-3
...
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  111s  default-scheduler  0/3 nodes are available: 3 pod has unbound immediate PersistentVolumeClaims.
  Warning  FailedScheduling  111s  default-scheduler  0/3 nodes are available: 3 pod has unbound immediate PersistentVolumeClaims.
  Warning  FailedScheduling  28s   default-scheduler  0/3 nodes are available: 1 node(s) had volume node affinity conflict, 2 Insufficient cpu.
```

查看PVC创建的云硬盘所在的可用区，发现data-nginx-3是在可用区1，而此时可用区1的节点没有资源，只有可用区3的节点有CPU资源，导致无法调度。由此可见PVC先绑定PV创建云硬盘会导致问题。

延迟绑定的云硬盘 StorageClass

在集群中查看StorageClass，可以看到csi-disk-topology的绑定模式为WaitForFirstConsumer，表示等有Pod使用这个PVC时再创建PV并绑定，也就是根据Pod的信息创建PV以及底层存储资源。

```
# kubectl get storageclass
NAME                PROVISIONER                     RECLAIMPOLICY    VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION    AGE
csi-disk            everest-csi-provisioner         Delete           Immediate              true                    156m
csi-disk-topology   everest-csi-provisioner         Delete           WaitForFirstConsumer   true                    156m
csi-nas             everest-csi-provisioner         Delete           Immediate              true                    156m
csi-obs             everest-csi-provisioner         Delete           Immediate              false                   156m
```

如上内容中VOLUMEBINDINGMODE列是在1.19版本集群中查看到的，1.17和1.15版本不显示这一列。

从csi-disk-topology的详情中也能看到绑定模式。

```
# kubectl describe sc csi-disk-topology
Name:                  csi-disk-topology
IsDefaultClass:        No
Annotations:           <none>
Provisioner:           everest-csi-provisioner
Parameters:            csi.storage.k8s.io/csi-driver-name=disk.csi.everest.io,csi.storage.k8s.io/fstype=ext4,everest.io/disk-volume-type=SAS,everest.io/passthrough=true
AllowVolumeExpansion:  True
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

下面创建csi-disk和csi-disk-topology两种类型的PVC，观察两者之间的区别。

- csi-disk

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  disk
    annotations:
      everest.io/disk-volume-type: SAS
  spec:
    accessModes:
    - ReadWriteOnce
  ```

```
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk        # StorageClass
```

- csi-disk-topology

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name:  topology
  annotations:
    everest.io/disk-volume-type: SAS
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk-topology        # StorageClass
```

创建并查看，如下所示，可以发现csi-disk已经是Bound也就是绑定状态，而csi-disk-topology是Pending状态。

```
# kubectl create -f pvc1.yaml
persistentvolumeclaim/disk created
# kubectl create -f pvc2.yaml
persistentvolumeclaim/topology created
# kubectl get pvc
NAME           STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS       AGE
disk           Bound     pvc-88d96508-d246-422e-91f0-8caf414001fc   10Gi       RWO            csi-disk           18s
topology       Pending                                                                         csi-disk-topology  2s
```

查看topology PVC的详情，可以在事件中看到“waiting for first consumer to be created before binding”，意思是等使用PVC的消费者也就是Pod创建后再绑定。

```
# kubectl describe pvc topology
Name:          topology
Namespace:     default
StorageClass:  csi-disk-topology
Status:        Pending
Volume:
Labels:        <none>
Annotations:   everest.io/disk-volume-type: SAS
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
VolumeMode:    Filesystem
Used By:       <none>
Events:
  Type    Reason              Age             From                         Message
  ----    ------              ----            ----                         -------
  Normal  WaitForFirstConsumer  5s (x3 over 30s)  persistentvolume-controller  waiting for first consumer to be created before binding
```

创建工作负载使用该PVC，其中申明PVC名称的地方填写topology，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
```

```
      app: nginx
  spec:
    containers:
    - image: nginx:alpine
      name: container-0
      volumeMounts:
      - mountPath: /tmp                          # 挂载路径
        name: topology-example
    restartPolicy: Always
    volumes:
    - name: topology-example
      persistentVolumeClaim:
        claimName: topology                      # PVC的名称
```

创建完成后查看PVC的详情，可以看到此时已经绑定成功。

```
# kubectl describe pvc topology
Name:          topology
Namespace:     default
StorageClass:  csi-disk-topology
Status:        Bound
....
Used By:       nginx-deployment-fcd9fd98b-x6tbs
Events:
  Type    Reason                Age
From                                                                                    Message
  ----    ------                ----
----                                                                                    -------
  Normal  WaitForFirstConsumer  84s (x26 over 7m34s)  persistentvolume-
controller                                                                              waiting for first consumer to be created before
binding
  Normal  Provisioning          54s                   everest-csi-provisioner_everest-csi-
controller-7965dc48c4-5k799_2a6b513e-f01f-4e77-af21-6d7f8d4dbc98  External provisioner is provisioning
volume for claim "default/topology"
  Normal  ProvisioningSucceeded  52s                  everest-csi-provisioner_everest-csi-
controller-7965dc48c4-5k799_2a6b513e-f01f-4e77-af21-6d7f8d4dbc98  Successfully provisioned volume
pvc-9a89ea12-4708-4c71-8ec5-97981da032c9
```

节点多 AZ 情况下使用 csi-disk-topology

下面使用csi-disk-topology创建有状态应用，将上面应用改为使用csi-disk-topology。

```
volumeClaimTemplates:
- metadata:
    name: data
    annotations:
      everest.io/disk-volume-type: SAS
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 1Gi
    storageClassName: csi-disk-topology
```

创建后查看PVC和Pod状态，如下所示，可以看到PVC和Pod都能创建成功，nginx-3这个Pod是创建在可用区3这个节点上。

```
# kubectl get pvc -owide
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS       AGE   VOLUMEMODE
data-nginx-0   Bound    pvc-43802cec-cf78-4876-bcca-e041618f2470   1Gi        RWO           csi-disk-
topology   55s   Filesystem
data-nginx-1   Bound    pvc-fc942a73-45d3-476b-95d4-1eb94bf19f1f   1Gi        RWO           csi-disk-
topology   39s   Filesystem
data-nginx-2   Bound    pvc-d219f4b7-e7cb-4832-a3ae-01ad689e364e   1Gi        RWO           csi-disk-
topology   22s   Filesystem
data-nginx-3   Bound    pvc-b54a61e1-1c0f-42b1-9951-410ebd326a4d   1Gi        RWO           csi-disk-
topology   9s    Filesystem
```

```
# kubectl get pod -owide
NAME      READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS
GATES
nginx-0   1/1     Running   0          65s   172.16.1.8     192.168.0.240   <none>           <none>
nginx-1   1/1     Running   0          49s   172.16.0.13    192.168.0.121   <none>           <none>
nginx-2   1/1     Running   0          32s   172.16.0.137   192.168.0.211   <none>           <none>
nginx-3   1/1     Running   0          19s   172.16.1.9     192.168.0.240   <none>           <none>
```

21.10 容器

21.10.1 合理分配容器计算资源

只要节点有足够的内存资源，那容器就可以使用超过其申请的内存，但是不允许容器使用超过其限制的资源。如果容器分配了超过限制的内存，这个容器将会被优先结束。如果容器持续使用超过限制的内存，这个容器就会被终结。如果一个结束的容器允许重启，kubelet就会重启它，但是会出现其他类型的运行错误。

场景一

节点的内存超过了节点内存预留的上限，导致触发OOMkill。

解决方法：

可扩容节点或迁移节点中的pod至其他节点。

场景二

pod的内存的limit设置较小，实际使用率超过limit，导致容器触发了OOMkill。

解决方法：

扩大工作负载内存的limit设置。

示例

本例将创建一个Pod尝试分配超过其限制的内存，如下这个Pod的配置文档，它申请50M的内存， 内存限制设置为100M。

memory-request-limit-2.yaml，此处仅为示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
spec:
  containers:
  - name: memory-demo-2-ctr
    image: vish/stress
    resources:
      requests:
        memory: 50Mi
      limits:
        memory: "100Mi"
    args:
    - -mem-total
    - 250Mi
    - -mem-alloc-size
    - 10Mi
```

```
    - -mem-alloc-sleep
    - 1s
```

在配置文件里的args段里，可以看到容器尝试分配250M的内存，超过了限制的100M。

创建Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit-2.yaml --namespace=mem-example
```

查看Pod的详细信息:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

这时候，容器可能会运行，也可能会被关闭。如果容器还没被关闭，重复之前的命令直至您看到这个容器被关闭：

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          24s
```

查看容器更详细的信息:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-example
```

这个输出显示了容器被关闭因为超出了内存限制。

```
lastState:
  terminated:
    containerID: docker://7aae52677a4542917c23b10fb56fcb2434c2e8427bc956065183c1879cc0dbd2
    exitCode: 137
    finishedAt: 2020-02-20T17:35:12Z
    reason: OOMKilled
    startedAt: null
```

本例中的容器可以自动重启，因此kubelet会再去启动它。输入多几次这个命令看看它是怎么被关闭又被启动的：

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

这个输出显示了容器被关闭，被启动，又被关闭，又被启动的过程：

```
$ kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          37s
$ kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-2   1/1       Running   2          40s
```

查看Pod的历史详细信息：

```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

这个输出显示了Pod一直重复着被关闭又被启动的过程：

```
... Normal  Created   Created container with id 66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511
... Warning BackOff   Back-off restarting failed container
```

21.10.2 通过特权容器功能优化内核参数

前提条件

从客户端机器访问Kubernetes集群，需要使用Kubernetes命令行工具kubectl，请先连接kubectl。

操作步骤

步骤1 通过后台创建daemonSet，选择nginx镜像、开启特权容器、配置生命周期、添加hostNetwork: true字段。

1. 新建daemonSet文件。

   vi daemonSet.yaml

   Yaml示例如下：

   须知

   spec.spec.containers.lifecycle字段是指容器启动后执行设置的命令。

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: daemonset-test
  labels:
    name: daemonset-test
spec:
  selector:
    matchLabels:
      name: daemonset-test
  template:
    metadata:
      labels:
        name: daemonset-test
    spec:
      hostNetwork: true
      containers:
      - name: daemonset-test
        image: nginx:alpine-perl
        command:
        - "/bin/sh"
        args:
        - "-c"
        - while :; do  time=$(date);done
        imagePullPolicy: IfNotPresent
        lifecycle:
          postStart:
            exec:
              command:
              - sysctl
              - "-w"
              - net.ipv4.tcp_tw_reuse=1
        securityContext:
          privileged: true
      imagePullSecrets:
      - name: default-secret
```

2. 创建daemonSet。

   kubectl create –f daemonSet.yaml

步骤2 查询daemonset是否创建成功。

kubectl get daemonset *daemonset名称*

本示例执行命令为：

kubectl get daemonset daemonset-test

命令行终端显示如下类似信息：

```
NAME              DESIRED   CURRENT   READY   UP-T0-DATE   AVAILABLE   NODE SELECTOR   AGE
daemonset-test    2         2         2       2            2           <node>          2h
```

步骤3 在节点上查询daemonSet的容器id。

docker ps -a|grep *daemonSet名称*

本示例执行命令为：

docker ps -a|grep daemonset-test

命令行终端显示如下类似信息：

```
897b99faa9ce      3e094d5696c1                “/bin/sh  -c while...”     31 minutes ago     Up  30
minutes  ault_fa7cc313-4ac1-11e9-a716-fa163e0aalba_0
```

步骤4 进入容器。

docker exec -it *containerid* **/bin/sh**

本示例执行命令如下：

docker exec -it *897b99faa9ce* **/bin/sh**

步骤5 查看容器中设置的启动后命令是否执行。

sysctl -a |grep net.ipv4.tcp_tw_reuse

命令行终端显示如下信息，表明修改系统参数成功。

```
net.ipv4.tcp_tw_reuse=1
```

----结束

21.10.3 使用 Init 容器初始化应用

概念

init-Containers，即初始化容器，顾名思义容器启动的时候，会先启动可一个或多个容器，如果有多个，那么这几个Init Container按照定义的顺序依次执行，只有所有的Init Container执行完后，主容器才会启动。由于一个Pod里的存储卷是共享的，所以Init Container里产生的数据可以被主容器使用到。

Init Container可以在多种K8s资源里被使用到如Deployment、DaemonSet、Job等，但归根结底都是在Pod启动时，在主容器启动前执行，做初始化工作。

使用场景

部署服务时需要做一些准备工作，在运行服务的pod中使用一个init container，可以执行准备工作，完成后Init Container结束退出，再启动要部署的容器。

- **等待其它模块Ready：**比如有一个应用里面有两个容器化的服务，一个是Web Server，另一个是数据库。其中Web Server需要访问数据库。但是当启动这个应用的时候，并不能保证数据库服务先启动起来，所以可能出现在一段时间内Web Server有数据库连接错误。为了解决这个问题，可以在运行Web Server服务的Pod里使用一个Init Container，去检查数据库是否准备好，直到数据库可以连接，Init Container才结束退出，然后Web Server容器被启动，发起正式的数据库连接请求。
- **初始化配置：**比如集群里检测所有已经存在的成员节点，为主容器准备好集群的配置信息，这样主容器起来后就能用这个配置信息加入集群。

- **其它使用场景：**如将pod注册到一个中央数据库、下载应用依赖等。

更多内容请参见**初始容器文档参考**。

操作步骤

步骤1 编辑initcontainer工作负载yaml文件。

vi deployment.yaml

Yaml示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      name: mysql
  template:
    metadata:
      labels:
        name: mysql
    spec:
      initContainers:
      - name: getresource
        image: busybox
        command: ['sleep 20']
      containers:
      - name: mysql
        image: percona:5.7.22
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
        resources:
          limits:
            memory: "500Mi"
            cpu: "500m"
          requests:
            memory: "500Mi"
            cpu: "250m"
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "mysql"
```

步骤2 创建initcontainer工作负载。

kubectl create -f deployment.yaml

命令行终端显示如下类似信息：

```
deployment.apps/mysql created
```

步骤3 在工作负载运行的节点上查询创建的docker容器。

docker ps -a|grep mysql

init容器运行后会直接退出，查询到的是exited(0)的退出状态。

```
0dc822969e3f        percona             "docker-entrypoint..."   34 seconds ago       Up 33 seconds
ql_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
a745881214e7        busybox             "sh -c 'sleep 20'"       About a minute ago   Exited (0) 50 seconds ago
resource_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
615db9e60a80        cfe-pause:11.23.1   "/pause"                 About a minute ago   Up About a minute
_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
```

----结束

21.10.4 使用 hostAliases 配置 Pod /etc/hosts

使用场景

DNS配置或其他选项不合理时，可以向pod的“/etc/hosts”文件中添加条目，使用**hostAliases**在pod级别覆盖对主机名的解析。

操作步骤

步骤1 使用kubectl连接集群。

步骤2 创建hostaliases-pod.yaml文件。

vi hostaliases-pod.yaml

Yaml中加粗字段为镜像及镜像版本，可根据实际需求进行修改：

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  hostAliases:
  - ip: 127.0.0.1
    hostnames:
    - foo.local
    - bar.local
  - ip: 10.1.2.3
    hostnames:
    - foo.remote
    - bar.remote
  containers:
    - name: cat-hosts
      image: tomcat:9-jre11-slim
      lifecycle:
        postStart:
          exec:
            command:
              - cat
              - /etc/hosts
  imagePullSecrets:
    - name: default-secret
```

表 21-19 pod 字段说明

参数名	是否必选	参数解释
apiVersion	是	api版本号。
kind	是	创建的对象类别。
metadata	是	资源对象的元数据定义。
name	是	Pod的名称。
spec	是	spec是集合类的元素类型，pod的主体部分都在spec中给出。具体请参见表21-20。

表 21-20 spec 数据结构说明

参数名	是否必选	参数解释
hostAliases	是	主机别名。
containers	是	具体请参见表21-21。

表 21-21 containers 数据结构说明

参数名	是否必选	参数解释
name	是	容器名称。
image	是	容器镜像名称。
lifecycle	否	生命周期。

步骤3 创建pod。

kubectl create -f hostaliases-pod.yaml

命令行终端显示如下信息表明pod已创建。

```
pod/hostaliases-pod created
```

步骤4 查看pod状态。

kubectl get pod hostaliases-pod

pod状态显示为Running，表示pod已创建成功。

```
NAME              READY     STATUS    RESTARTS   AGE
hostaliases-pod   1/1       Running   0          16m
```

步骤5 查看配置的**hostAliases**是否正常，执行如下命令：

docker ps |grep hostaliases-pod

docker exec -ti 容器ID /bin/sh

```
root@hostaliases-pod:/# cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.0.0.25       hostaliases-pod

# Entries added by HostAliases.
127.0.0.1       foo.local       bar.local
10.1.2.3        foo.remote      bar.remote
```

----结束

21.10.5 容器 Core Dump

应用场景

Core Dump是Linux操作系统在程序突然异常终止或者崩溃时将当时的内存状态记录下来，保存在一个文件中。通过Core Dump文件可以分析查找问题原因。

容器一般将业务应用程序作为容器主程序，程序崩溃后容器直接退出，且被回收销毁，因此容器Core Dump需要将Core文件持久化存储在主机或云存储上。本文将介绍容器Core Dump的方法。

约束与限制

容器Core Dump持久化存储至OBS（并行文件系统或对象桶）时，由于CCE挂载OBS时默认挂载参数中带有umask=0的设置，这导致Core Dump文件虽然生成但由于umask原因Core Dump信息无法写入到Core文件中。

开启节点 Core Dump

登录节点，执行如下命令开启Core Dump，设置core文件的存放路径及格式。

echo "/tmp/cores/core.%h.%e.%p.%t" > /proc/sys/kernel/core_pattern

其中%h、%e、%p、%t均表示占位符，说明如下：

- %h：主机名（在 Pod 内即为 Pod 的名称），建议配置。
- %e：程序文件名，建议配置。
- %p：进程 ID，可选。
- %t：coredump 的时间，可选。

即通过以上命令开启Core Dump后，生成的core文件的命名格式为“core.{主机名}.{程序文件名}.{进程ID}.{时间}”。

您也可以在创建节点时候通过设置安装前或安装后脚本自动执行该命令。

容器 Core Dump 持久化

core文件可以考虑使用HostPath或PVC存放在本机或云存储，如下为使用HostPath方式示例pod.yaml。

```
apiVersion: v1
kind: Pod
metadata:
  name: coredump
spec:
  volumes:
  - name: coredump-path
    hostPath:
      path: /home/coredump
  containers:
  - name: ubuntu
    image: ubuntu:12.04
    command: ["/bin/sleep","3600"]
    volumeMounts:
    - mountPath: /tmp/cores
      name: coredump-path
```

使用kubectl创建Pod。

kubectl create -f pod.yaml

配置验证

Pod创建后，进入到容器内，触发当前shell终端的段错误。

```
$ kubectl get pod
NAME                       READY   STATUS    RESTARTS   AGE
coredump                   1/1     Running   0          56s
$ kubectl exec -it coredump -- /bin/bash
root@coredump:/# kill -s SIGSEGV $$
command terminated with exit code 139
```

登录节点，在/home/coredump路径下查看core文件是否生成，如下示例表示已经生成了core文件。

```
# ls /home/coredump
core.coredump.bash.18.1650438992
```

21.11 权限

21.11.1 通过配置 kubeconfig 文件实现集群权限精细化管理

问题场景

CCE默认的给用户的kubeconfig文件为cluster-admin角色的用户，相当于root权限，对于一些用户来说权限太大，不方便精细化管理。

目标

对集群资源进行精细化管理，让特定用户只能拥有部分权限（如：增、查、改）。

注意事项

确保您的机器上有kubectl工具，若没有请到**Kubernetes版本发布页面**下载与集群版本对应的或者最新的kubectl。

配置方法

> **说明**
>
> 下述示例配置只能查看和添加**test**空间下面的Pod和Deployment，不能删除。

步骤1 配置sa，名称为**my-sa**，命名空间为**test**。

```
kubectl create sa my-sa -n test
```

```
[root@test-arm-54016 ~]#
[root@test-arm-54016 ~]# kubectl create sa my-sa -n test
serviceaccount/my-sa created
[root@test-arm-54016 ~]#
```

步骤2 配置role规则表，赋予不同资源相应的操作权限。

```
vi role-test.yaml
```

内容如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: myrole
  namespace: test
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - pods
  - deployments
  verbs:
  - get
  - list
  - watch
  - create
```

创建Role：

```
kubectl create -f role-test.yaml
```

```
[root@test-arm-54016 ~]# kubectl create -f role-test.yaml
role.rbac.authorization.k8s.io/myrole created
[root@test-arm-54016 ~]#
```

步骤3 配置rolebinding，将sa绑定到role上，让sa获取相应的权限。

```
vi myrolebinding.yaml
```

内容如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: myrolebinding
  namespace: test
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: Role
  name: myrole
subjects:
- kind: ServiceAccount
  name: my-sa
  namespace: test
```

创建RoleBinding：

```
kubectl create -f myrolebinding.yaml
```

```
[root@test-arm-54016 ~]# kubectl create -f myrolebinding.yaml
rolebinding.rbac.authorization.k8s.io/myrolebinding created
[root@test-arm-54016 ~]#
```

此时，用户信息配置完成，继续执行步骤步骤4~步骤步骤6将用户信息写入到配置文件中。

步骤4 配置集群访问信息。

1. 通过sa的名称**my-sa**获取sa对应的密钥，第一列的**my-sa-token-z4967**即为密钥名：

```
kubectl get secret -n test |grep my-sa
```

```
[root@test-arm-54016 ~]# kubectl get secret -n test |grep my-sa
my-sa-token-5gpl4     kubernetes.io/service-account-token   3      21m
[root@test-arm-54016 ~]#
```

2. 将密钥中的ca.crt解码后导出备用：

```
kubectl get secret my-sa-token-5gpl4 -n test -oyaml |grep ca.crt: | awk '{print $2}' |base64 -d > /home/ca.crt
```

3. 设置集群访问方式，其中**test-arm**为需要访问的集群，**10.0.1.100**为集群apiserver地址，**/home/test.config**为配置文件的存放路径。
   - 如果通过内部apiserver地址，执行如下命令：
     ```
     kubectl config set-cluster test-arm --server=https://10.0.1.100:5443  --certificate-authority=/home/ca.crt  --embed-certs=true --kubeconfig=/home/test.config
     ```
   - 如果通过公网apiserver地址，执行如下命令：
     ```
     kubectl config set-cluster test-arm --server=https://10.0.1.100:5443 --kubeconfig=/home/test.config --insecure-skip-tls-verify=true
     ```

```
[root@test-arm-54016 home]# kubectl config set-cluster test-arm --server=https://10.0.1.100:5443  --certificate-authority=/home/ca.crt  --embed-certs=true --kubeconfig=/home/test.config
Cluster "test-arm" set.
[root@test-arm-54016 home]# _
```

说明

若**在集群内节点上执行操作**或者**最后使用该配置的节点为集群节点**，不要将kubeconfig的路径设为/root/.kube/config。

集群apiserver地址为内网apiserver地址，绑定弹性IP后也可为公网apiserver地址。

步骤5 配置集群认证信息。

1. 获取集群的token信息（这里如果是get获取需要based64 -d解码）。

```
token=$(kubectl describe secret my-sa-token-5gpl4 -n test | awk '/token:/{print $2}')
```

2. 设置使用集群的用户**ui-admin**。

```
kubectl config set-credentials ui-admin --token=$token --kubeconfig=/home/test.config
```

```
[root@test-arm-54016 home]# kubectl config set-credentials ui-admin --token=$token --kubeconfig=/home/test.config
User "ui-admin" set.
[root@test-arm-54016 home]#
```

步骤6 配置集群认证访问的上下文信息，**ui-admin@test**为上下文的名称。

kubectl config set-context **ui-admin@test** --cluster=**test-arm** --user=**ui-admin** --kubeconfig=**/home/test.config**

```
[root@test-arm-54016 home]# kubectl config set-context ui-admin@test --cluster=test-arm --user=ui-admin --kubeconfig=/home/test.
config
Context "ui-admin@test" created.
[root@test-arm-54016 home]#
```

步骤7 设置上下文，设置完成后使用方式见**验证权限**。

kubectl config use-context **ui-admin@test** --kubeconfig=**/home/test.config**

```
[paas@test-arm-54016 home]$ kubectl config use-context ui-admin@test --kubeconfig=/home/test.config
Switched to context "ui-admin@test".
[paas@test-arm-54016 home]$
```

说明

若需授予其他用户操作该集群并限制为上述权限，在步骤**步骤6**结束后将生成的配置文件/**home/test.config**提供给该用户，由该用户置于自己机器上（**用户机器须保证能访问集群apiserver地址**），在该机器上执行步骤**步骤7**使用kubectl时kubeconfig参数须指定为配置文件所在路径。

----结束

验证权限

1. 可以查询**test**命名空间下的pod资源，被拒绝访问其他命名空间的Pod资源。

   kubectl get pod -n **test** --kubeconfig=**/home/test.config**

   ```
   [paas@test-arm-54016 home]$ kubectl get pod -n test --kubeconfig=/home/test.config
   NAME                        READY   STATUS             RESTARTS   AGE
   test-pod-56cfcbf45b-l2q92   0/1     CrashLoopBackOff   27         91m
   [paas@test-arm-54016 home]$
   [paas@test-arm-54016 home]$ kubectl get pod --kubeconfig=/home/test.config
   Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:test:my-sa" cannot list resource "pods" in API gro
   up "" in the namespace "default"
   [paas@test-arm-54016 home]$
   ```

2. 不可删除**test**命名空间下的Pod资源。

   ```
   [paas@test-arm-54016 home]$ kubectl delete pod -n test test-pod-56cfcbf45b-l2q92 --kubeconfig=/home/test.config
   Error from server (Forbidden): pods "test-pod-56cfcbf45b-l2q92" is forbidden: User "system:serviceaccount:test:my-sa" cannot del
   ete resource "pods" in API group "" in the namespace "test"
   [paas@test-arm-54016 home]$
   ```

延伸阅读

更多Kubernetes中的用户与身份认证授权内容，请参见**Authenticating**。

21.12 发布

21.12.1 发布概述

应用现状

应用程序升级面临最大挑战是新旧业务切换，将软件从测试的最后阶段带到生产环境，同时要保证系统不间断提供服务。如果直接将某版本上线发布给全部用户，一旦遇到线上事故（或BUG），对用户的影响极大，解决问题周期较长，甚至有时不得不回滚到前一版本，严重影响了用户体验。

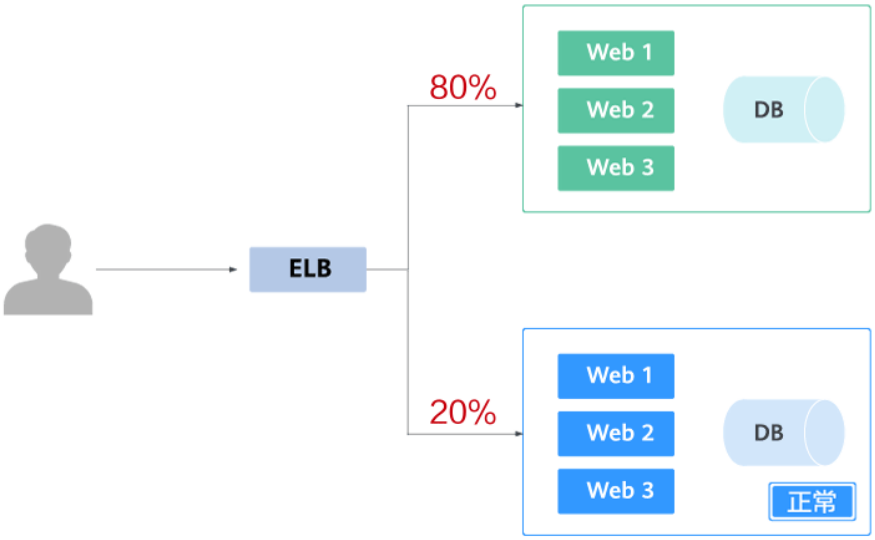
解决方案

长期以来，业务升级逐渐形成了几个发布策略：灰度发布、蓝绿发布、A/B测试、滚动升级以及分批暂停发布，尽可能避免因发布导致的流量丢失或服务不可用问题。

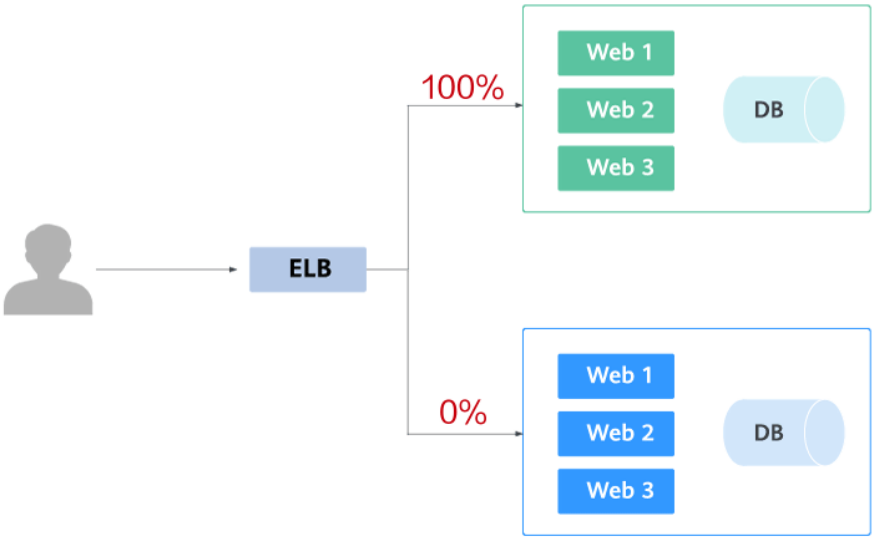
本文着重介绍灰度发布和蓝绿发布的原理及实践案例。

- 灰度发布，又称金丝雀发布，是版本升级平滑过渡的一种方式，当版本升级时，使部分用户使用新版本，其他用户继续使用老版本，待新版本稳定后，逐步扩大范围把所有用户流量都迁移到新版本上面来。这样可以最大限度地控制新版本发布带来的业务风险，降低故障带来的影响面，同时支持快速回滚。

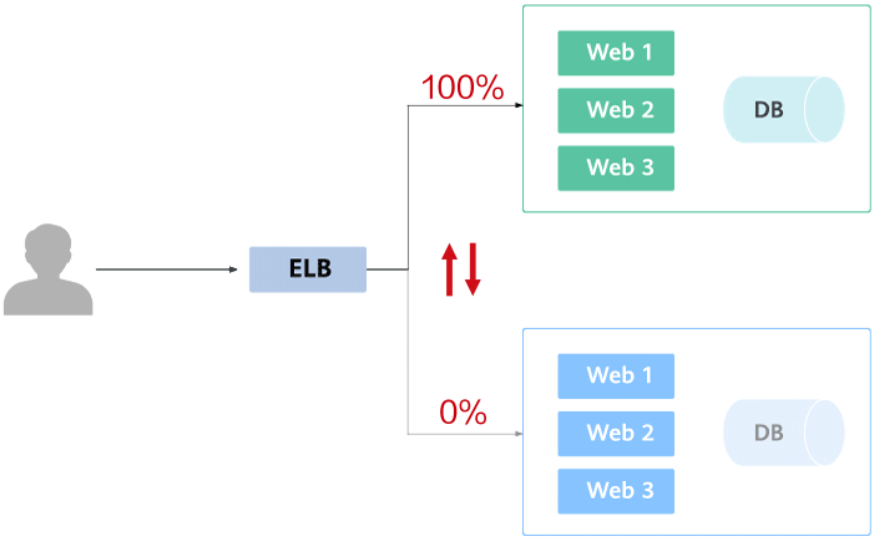
  以下示意图可描述灰度发布的大致流程：先切分20%的流量到新版本，若表现正常，逐步增加流量占比，继续测试新版本表现。若新版本一直很稳定，那么将所有流量都切分到新版本，并下线老版本。



  切分20%的流量到新版本后，新版本出现异常，则快速将流量切回老版本。



- 蓝绿发布提供了一种零宕机的部署方式，是一种以可预测的方式发布应用的技术，目的是减少发布过程中服务停止的时间。在保留老版本的同时部署新版本，将两个版本同时在线，新版本和老版本相互热备，通过切换路由权重的方式（非0即100）实现应用的不同版本上线或者下线，如果有问题可以快速地回滚到老版本。

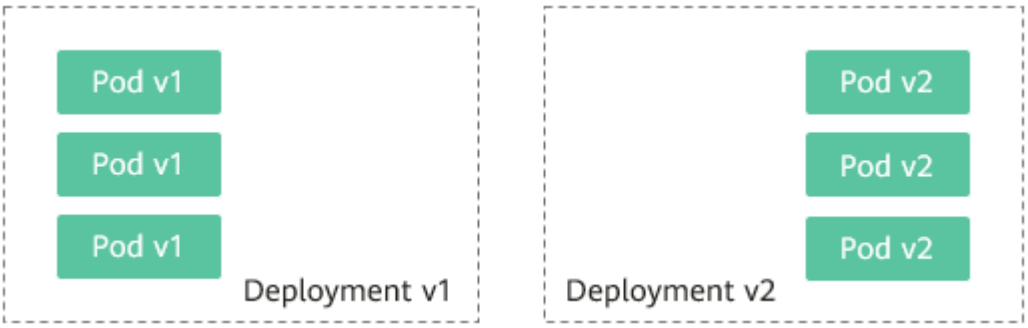


21.12.2 使用 Service 实现简单的灰度发布和蓝绿发布

CCE实现灰度发布通常需要向集群额外部署其他开源工具，例如Nginx Ingress，或将业务部署至服务网格，利用服务网格的能力实现。这些方案均有一些难度，如果您的灰度发布需求比较简单，且不希望引入过多的插件或复杂的用法，则可以参考本文利用Kubernetes原生的特性实现简单的灰度发布和蓝绿发布。

原理介绍

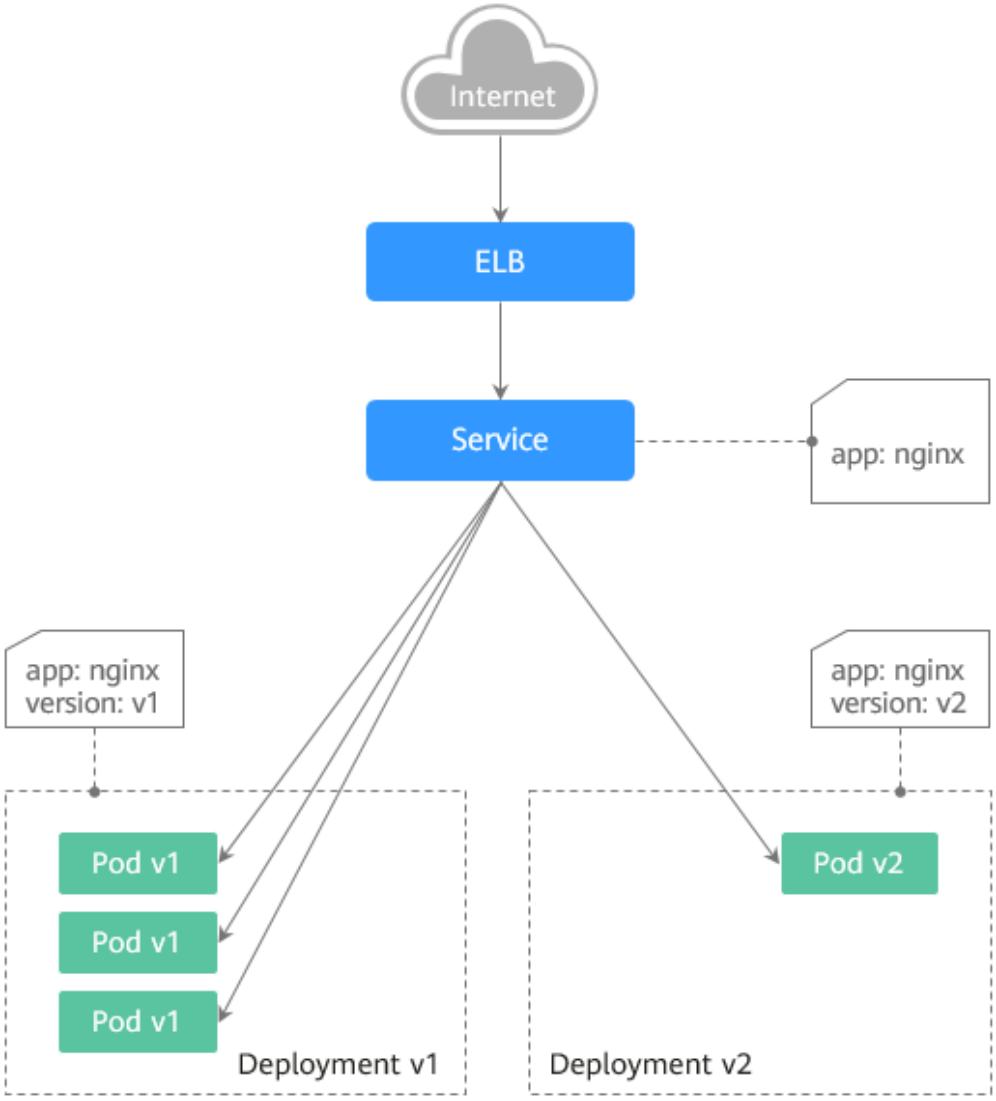
用户通常使用无状态负载 Deployment、有状态负载 StatefulSet等Kubernetes对象来部署业务，每个工作负载管理一组Pod。以Deployment为例，示意图如下：



通常还会为每个工作负载创建对应的Service，Service使用selector来匹配后端Pod，其他服务或者集群外部通过访问Service即可访问到后端Pod提供的服务。如需对外暴露可直接设置Service类型为LoadBalancer，弹性负载均衡ELB将作为流量入口。

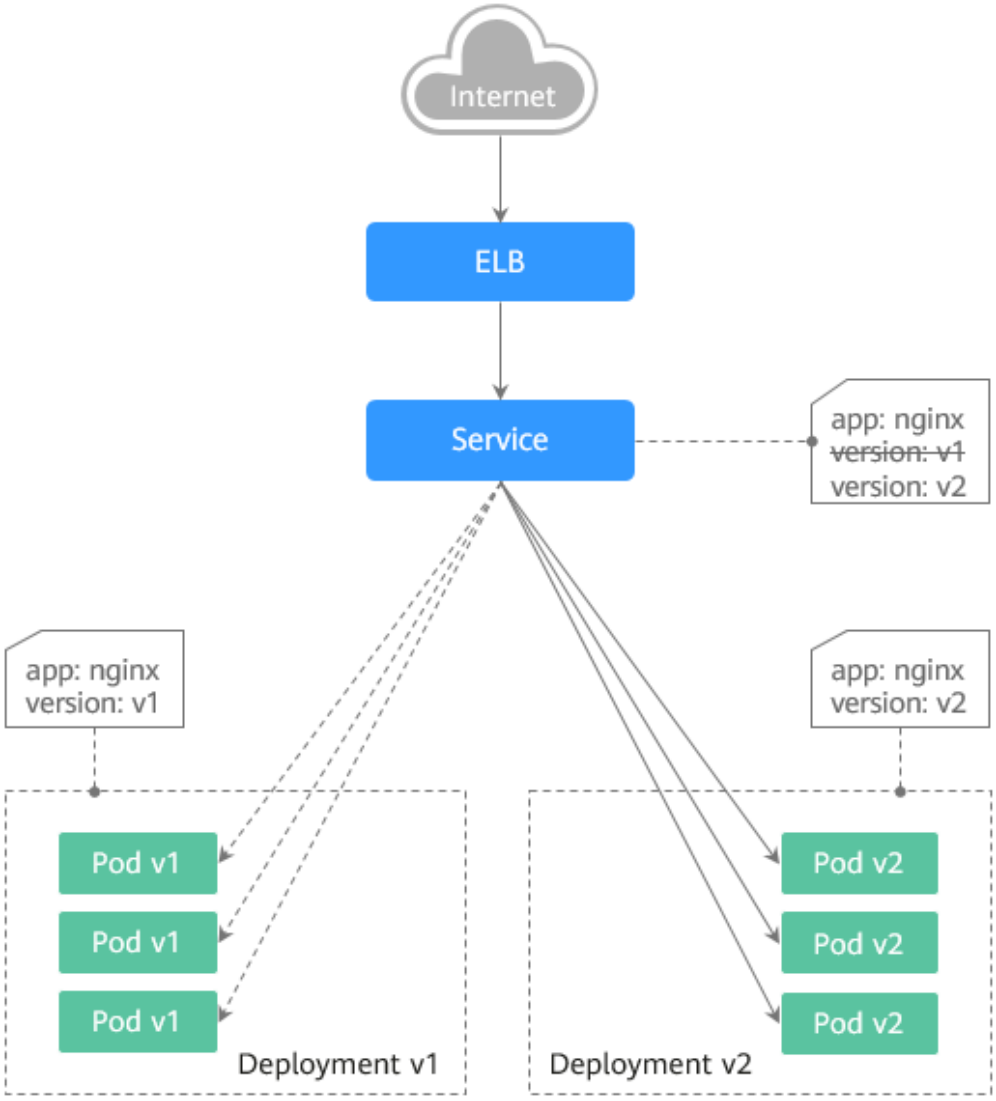
- **灰度发布原理**

  以Deployment为例，用户通常会为每个Deployment创建一个Service，但Kubernetes并未限制Service需与Deployment一一对应。Service通过selector匹配后端Pod，若不同Deployment的Pod被同一selector选中，即可实现一个Service对应多个版本Deployment。调整不同版本Deployment的副本数，即可调整不同版本服务的权重，实现灰度发布。示意图如下：



- **蓝绿发布原理**

  以Deployment为例，集群中已部署两个不同版本的Deployment，其Pod拥有共同的label。但有一个label值不同，用于区分不同的版本。Service使用selector选中了其中一个版本的Deployment的Pod，此时通过修改Service的selector中决定服务版本的label的值来改变Service后端对应的Pod，即可实现让服务从一个版本直接切换到另一个版本。示意图如下：



前提条件

已上传Nginx镜像至容器镜像服务。为方便观测流量切分效果，Nginx镜像包含v1和v2两个版本，欢迎页分别为“Nginx-v1”和“Nginx-v2”。

资源创建方式

本文提供以下两种方式使用YAML部署Deployment和Service：

- 方式1：在创建无状态工作负载向导页面，单击右侧“YAML创建”，再将本文示例的YAML文件内容输入编辑窗中。
- 方式2：将本文的示例YAML保存为文件，再使用kubectl指定YAML文件进行创建。例如：**kubectl create -f xxx.yaml**。

步骤 1：部署两个版本的服务

在集群中部署两个版本的Nginx服务，通过ELB对外提供访问。

步骤1 创建第一个版本的Deployment，本文以nginx-v1为例。YAML示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v1
spec:
  replicas: 2              # Deployment的副本数，即Pod的数量
  selector:                # Label Selector（标签选择器）
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:              # Pod的标签
        app: nginx
        version: v1
    spec:
      containers:
      - image: {your_repository}/nginx:v1 # 容器使用的镜像为：nginx:v1
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

步骤2 创建第二个版本的Deployment，本文以nginx-v2为例。YAML示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v2
spec:
  replicas: 2              # Deployment的副本数，即Pod的数量
  selector:                # Label Selector（标签选择器）
    matchLabels:
      app: nginx
      version: v2
  template:
    metadata:
      labels:              # Pod的标签
        app: nginx
        version: v2
    spec:
      containers:
      - image: {your_repository}/nginx:v2   # 容器使用的镜像为：nginx:v2
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

您可以登录云容器引擎控制台查看部署情况。

----结束

步骤 2：实现灰度发布

步骤1 为部署的Deployment创建LoadBalancer类型的Service对外暴露服务，selector中不指定版本，让Service同时选中两个版本的Deployment的Pod。YAML示例如下：

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: 586c97da-a47c-467c-a615-bd25a20de39c    # ELB实例的ID，请替换为实际取值
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:             # selector中不包含version信息
    app: nginx
  type: LoadBalancer    # 类型为LoadBalancer
```

步骤2 执行以下命令，测试访问。

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

其中，<EXTERNAL_IP>为ELB实例的IP地址。

返回结果如下，一半为v1版本的响应，一半为v2版本的响应。

```
Nginx-v2
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v2
Nginx-v2
```

步骤3 通过控制台或kubectl方式调整Deployment的副本数，将v1版本调至4个副本，v2版本调至1个副本。

kubectl scale deployment/nginx-v1 --replicas=4

kubectl scale deployment/nginx-v2 --replicas=1

步骤4 执行以下命令，再次测试访问。

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

其中，<EXTERNAL_IP>为ELB实例的IP地址。

返回结果如下，可以看到10次访问中仅2次为v2版本的响应，v1与v2版本的响应比例与其副本数比例一致，为4:1。通过控制不同版本服务的副本数就实现了灰度发布。

```
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v1
Nginx-v1
```

说明

如果10次访问中v1和v2版本比例并非4:1，可以将访问次数调整至更大，比如20。理论上来说，次数越多，v1与v2版本的响应比例将越接近于4:1。

----结束

步骤 3：实现蓝绿发布

步骤1 为部署的Deployment创建LoadBalancer类型的Service对外暴露服务，指定使用v1版本的服务。YAML示例如下：

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: 586c97da-a47c-467c-a615-bd25a20de39c    # ELB实例的ID，请替换为实际取值
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:             # selector中指定version为v1
    app: nginx
    version: v1
  type: LoadBalancer    # 类型为LoadBalancer
```

步骤2 执行以下命令，测试访问。

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

其中，<EXTERNAL_IP>为ELB实例的IP地址。

返回结果如下，均为v1版本的响应。

```
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
```

步骤3 通过控制台或kubectl方式修改Service的selector，使其选中v2版本的服务。

kubectl patch service nginx -p '{"spec":{"selector":{"version":"v2"}}}'

步骤4 执行以下命令，再次测试访问。

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

其中，<EXTERNAL_IP>为ELB实例的IP地址。

返回结果如下，均为v2版本的响应，成功实现了蓝绿发布。

```
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
```

----结束

22 将老版本的数据迁移到最新版本

通过如下操作将旧版本的应用迁移到最新版本。

1. **镜像迁移**：将旧版本镜像仓库中的镜像下载到最新版本的镜像仓库中。
2. **迁移集群**：创建新版本的虚拟机集群，用于部署应用。
3. **迁移应用**：将老版CCE中的应用重新部署到新版CCE的集群上。

22.1 版本间差异

CCE新版本除了修改继承了之前老版本的功能，还新增了一些新功能。

- 修改功能：
  - 老版本的集群相当于新版本的虚拟机集群。
  - 新版本没有组件模板的概念。前一版本中，创建应用时，可以在组件模板中设置参数。
  - 前一版本中的应用相当于新版本中的无状态工作负载。同时新版本中，还增加了有状态工作负载。
- 新增功能：
  - 在CCE控制台创建工作负载时，支持创建和直接挂载云硬盘和文件存储。
  - 集群节点弹性伸缩。
  - Docker升级到1706版本。
  - 自定义的工作负载程序编排的helm模板。
- 在新版本中，上传镜像前，需要先创建镜像仓库。

新旧版本之间比较如**表22-1**。

表 22-1 两个版本之间的比较

老版本	最新版本
容器注册表	镜像仓库
集群	资源管理 > 虚拟机集群
组件模板	-

老版本	最新版本
应用设计器	-
应用管理器	无状态工作负载

22.2 镜像迁移

将上一个版本中上传的镜像迁移到新版本CCE。

本章节以将apache镜像迁移到最新的CCE版本为例进行说明。

操作步骤

上传容器镜像前，请确保本地Docker客户端能够访问私有容器注册表。

步骤1 以**root**用户登录Docker客户端。

步骤2 创建组织。组织用于隔离镜像，并为不同的用户分配访问权限（读、编辑、管理）。

1. 登录CCE控制台，在左侧导航栏中选择“镜像仓库”。
2. 在容器镜像服务控制台，单击总览页面的“创建组织”。
3. 输入组织名称，即账号的域名。配置完成后，单击“确定”。

步骤3 以**root**用户登录Docker客户端。

步骤4 按照老版本镜像的下载步骤，将旧版本的镜像文件拉至本地目录。

说明

登录老版本控制台，在左侧导航栏选择“镜像仓库”，单击需要下载的镜像名称。进入镜像详情页面，可看到对应镜像的Pull命令。

示例：

docker pull *10.154.57.150:443/test/apache-php:latest*

步骤5 连接新版本CCE。

1. 登录CCE新版本控制台，单击左侧导航栏的“镜像仓库”。
2. 在容器镜像服务控制台，单击我的镜像中的“客户端上传”。单击“生成临时docker login指令”。
3. 在弹出的Docker login命令中，单击，复制命令。

步骤6 登录安装Docker的服务器，执行步骤5中拷贝的Docker login命令。

成功登录Docker客户端后，系统提示“login succeeded”。

步骤7 执行以下命令，标记nginx:1.10镜像。

docker tag *{imageid} {image address}:tag*

其中，*{image address}*为镜像仓库地址。若您是从外部网络上传的镜像，请输入外部镜像地址。如果是从内部网络上传的镜像，请输入内部镜像地址。

示例：

docker tag *2e233ad9329b 10.125.1.15:20202/test/apache-php:latest*

步骤8 执行以下命令，将镜像推送到镜像仓库。

docker push *{image address}:tag*

示例：

docker push *10.125.1.15:20202/test/apache-php:latest*

显示如下信息，表明推送成功。

```
6d6b9812c8ae: Pushed
695da0025de6: Pushed
fe4c16cbf7a4: Pushed
1.10: digest: sha256:eb7e3bbd8e3040efa71d9c2cacfa12a8e39c6b2ccd15eac12bdc49e0b66cee63 size: 948
```

在容器镜像服务控制台，进入“我的镜像 > 镜像名称”页面。在镜像列表中可看到已上传的镜像。

----结束

22.3 迁移集群

在新版本CCE控制台创建虚拟机集群，规格请与之前版本相同。

如果需要使用API创建集群，请参见《 云容器引擎接口参考2.0 》。

操作步骤

步骤1 登录CCE控制台，在左侧导航栏中选择“资源管理 > 集群管理”。单击“创建虚拟机集群”。

步骤2 设置集群参数，其中带“*”的参数为必填项。

表 22-2 创建集群参数说明

新增参数	旧参数	配置说明
* 集群名称	名称	待创建集群的名称。
* 版本	该参数在之前的CCE版本中不存在。保持默认值即可。	集群版本，对应Kubernetes基础版本。
* 集群管理规模	该参数在之前的CCE版本中不存在。根据业务需要配置。	当前集群支持管理的最大节点规模。
* 高可用性	集群类型	• 是：集群将创建三个管理节点，当其中某两个管理节点故障时，集群依然可用。 • 否：集群仅创建一个管理节点，当这个管理节点故障时，集群将不可用，但已运行的应用不受影响。

新增参数	旧参数	配置说明
* 虚拟私有云	新版本可以使用旧版本创建的VPC。	新建集群所在的虚拟私有云。 若没有可选虚拟私有云，单击“创建虚拟私有云”进行创建。
* 所在子网	可以使用上一个CCE版本创建的子网。	节点虚拟机运行的子网环境。
* 网络模型	该参数在之前的CCE版本中不存在。根据业务需要配置。	● 容器隧道网络：基于VPC网络之上虚拟出的一层网络，一般场景均可满足。 ● VPC网络：直接基于VPC网络，性能更高，适用于高性能多交互的场景。但同个VPC下只能建一个VPC网络模型的集群。
容器网段	该参数在之前的CCE版本中不存在。根据业务需要配置。	请根据业务需求选择容器网段，确定容器网段后，容器实例将在规划的网段内分配IP。 ● 未勾选“自动选择”：请手动选择网段。若与子网网段有冲突时将有红色文字提示，请重新选择。建议使用网段：10.0.0.0/12~19，172.16.0.0/16~19，192.168.0.0/16~19。**不同集群使用相同的容器网段，会导致容器IP冲突，应用访问异常。** ● 勾选“自动选择”：系统将自动分配与子网网段无冲突的网段。 容器网段要设置合理的掩码，掩码决定集群内可用节点数量。集群中容器网段掩码设置不合适，会导致集群实际可用的节点较少。设置掩码后，选项下方会有当前网段最多支持的实例估算值，请作参考。
服务网段	该参数在之前的CCE版本中不存在。根据业务需要配置。	默认不设置。此参数仅支持1.11.7及以上版本的集群。 服务网段为kubernetes service ip网段，请根据业务需求选择该网段。服务网段要设置合理的掩码，掩码决定集群内可用service ip数量。

新增参数	旧参数	配置说明
集群开放EIP	该参数在之前的CCE版本中不存在。根据业务需要配置。	独立申请的公网IP地址。选择EIP必须是空闲，EIP会预制在集群证书中，创建完集群后不能手动删除该IP，否则会导致双向认证不可用。 ● 暂不配置：集群不开放EIP。 ● 现在配置：给集群添加EIP，若没有可用的EIP，请先创建。
鉴权方式	该参数在之前的CCE版本中不存在。根据业务需要配置。	“RBAC”默认勾选。**请务必阅读CCE权限管理说明并勾选“我已知晓上述限制”。** 开启RBAC能力后，设置了细粒度权限的子用户使用集群下资源将受到权限控制。
认证方式	该参数在之前的CCE版本中不存在。根据业务需要配置。	认证机制主要用于对集群下的资源做权限控制。例如A用户只能对某个命名空间下的应用有读写权限，B用户对集群下的资源只有读权限等等。 ● “认证能力增强”默认状态下不选定，此时默认开启X509认证模式，X.509是一种非常通用的证书格式。 ● 若需要对集群进行权限控制，请勾选“认证能力增强”，选择“认证代理”。 单击“CA根证书”后的“上传文件”，上传符合规范且合法的证书，并**勾选“我已确认上传的证书合法”**。 证书若不合法，集群将无法创建成功。请上传小于1MB的文件，上传格式支持.crt或.cer格式。
集群描述	描述	新建容器集群的描述信息。

步骤3 配置完成后，单击“下一步”，添加节点。

步骤4 是否创建节点，选择“是”。

步骤5 参照表22-3配置新增节点参数。

表 22-3 添加节点参数

新参数	旧参数	配置说明
所属区域		

新参数	旧参数	配置说明
当前区域	所属区域	节点实例所在的物理位置。
可用区		指在同一地域下，电力、网络隔离的物理区域，可用分区之间内网互通，不同可用分区之间物理隔离。
产品规格		
节点名称	产品规格	自定义节点名称。
节点规格		• 通用型：为大多数应用场景提供通用的计算、存储、网络配置。通用实例可以在Web服务器、开发测试环境和小型数据库应用程序中使用。 • 内存优化型：该类型实例提供内存比例更高的实例，可以用于对内存要求较高、数据量大的工作负载，例如关系数据库、NoSQL等场景。 • 通用计算增强型：该类型实例具有性能稳定且资源独享的特点，满足计算性能高且稳定的企业级工作负载诉求。 • GPU加速型：提供优秀的浮点计算能力，从容应对高实时、高并发的海量计算场景。P系列适合于深度学习，科学计算，CAE等；G系列适合于3D动画渲染，CAD等。目前仅支持1.11版本的集群添加GPU加速型节点；1.13及以上版本集群暂不支持，界面中不显示该选项。 • 超高I/O型：该类型实例提供超低SSD盘访问延迟和超高IOPS性能，适用于高性能关系型数据库、NoSQL数据库(如Cassandra、MongoDB)、ElasticSearch搜索等场景。
操作系统		请选择节点对应的操作系统。 重装操作系统或修改操作系统配置将导致节点不可用，请务必谨慎操作。详情请参见**集群节点高危操作**。
虚拟私有云	该参数在之前的CCE版本中不存在。根据业务需要配置。	跟随集群，不可变更。该参数仅在v1.13.10-r0及以上版本的集群中支持，否则不显示。

新参数	旧参数	配置说明
所在子网	该参数在之前的CCE版本中不存在。根据业务需要配置。	通过子网提供与其他网络隔离的、可以独享的网络资源，以提高网络安全。 可选择该集群虚拟私有云下的任意子网，集群节点支持跨子网。该参数仅在v1.13.10-r0及以上版本的集群中支持，否则不显示。
节点创建数量	数量	新增节点数。
网络 **说明** 若新增节点有互联网访问的需求，则弹性IP请选择“现在创建或使用已有”。若新增节点未绑定弹性IP，则在该节点上运行的应用将不能被外网访问。		
弹性IP	弹性IP	独立的公网IP地址。 ● 暂不使用：不使用弹性IP的节点不能与互联网互通，仅可作为私有网络中部署业务或者集群所需云服务器进行使用。 ● 现在申请：自动为每台云服务器分配独享带宽的弹性IP。创建弹性云服务器过程中，请确保弹性IP配额充足。请根据界面要求，选择规格、创建量、计费模式、带宽等。 ● 使用已有：为当前节点分配已有弹性IP，请选择已有的弹性IP。
磁盘	存储	分为系统盘和数据盘。 ● 系统盘的规格为[40,1024]GB，用户可以配置，缺省值为40GB。 ● 数据盘的规格为[100,32678]GB，用户可以配置，缺省值为100GB。 提供以下性能规格的云硬盘。 ● 普通IO：是指由SATA存储提供资源的磁盘类型。提供可靠的块存储，单个云硬盘的最大IOPS可达到1000，可运行关键应用程序。 ● 高IO：是指由SAS存储提供资源的磁盘类型。提供可达到3000的高IO和低至1 ms的读写延时，支持NoSQL/关系型数据库，数据仓库，文件系统等应用。 ● 超高IO：是指由SSD存储提供资源的磁盘类型。提供可达到20000的超高IO和低至1 ms超低读写时延，支持NoSQL/关系型数据库，数据仓库等应用。
登录信息		
密钥对	密钥对	密钥对用于远程登录节点时的身份认证，若没有密钥对，可单击“创建密钥对”来新建。
ECS高级设置		

新参数	旧参数	配置说明
云服务器组	该参数在之前的CCE版本中不存在。根据业务需要配置。	选择已创建的云服务器组，或单击右侧的“新建云服务器组”创建，创建完成后单击刷新按钮。 通过云服务器组功能，弹性云服务器在创建时，将尽量分散地创建在不同的主机上，提高业务的可靠性。
资源标签		通过为资源添加标签，可以对资源进行自定义标记，实现资源的分类。 您可以在TMS中创建“预定义标签”，预定义标签对所有支持标签功能的服务资源可见，通过使用预定义标签可以提升标签创建和迁移效率。 CCE服务会自动帮您创建CCE-Dynamic-Provisioning-Node=节点id的标签，允许增加5个标签。
委托		委托是由租户管理员在统一身份认证服务（IAM）上创建的。通过委托，可以将云主机资源共享给其他账号，或委托更专业的人或团队来代为管理。创建委托时委托类型选择“云服务”，单击“选择”按钮并在弹出的窗口中选择“ECS BMS”，即允许ECS或BMS调用云服务。
安装前执行脚本		请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装前执行，可能导致Kubernetes软件无法正常安装，需谨慎使用。常用于格式化数据盘等场景。
安装后执行脚本		请输入脚本命令，大小限制为0~1000字符。 脚本将在Kubernetes软件安装后执行，不影响Kubernetes软件安装。常用于修改Docker配置参数等场景。
新增数据盘		单击“新增数据盘”增加一个数据盘并设置数据盘容量，该数据盘需要在**安装前执行脚本**中输入脚本命令进行格式化。
子网IP		可选择“自动分配IP地址”和“手动分配IP地址”，推荐使用“自动分配IP地址”。
Kubernetes高级设置		
最大实例数	该参数在之前的CCE版本中不存在。根据业务需要配置。	节点最大允许创建的实例数(Pod)，该数量包含系统默认实例，取值范围为16~250。 该设置的目的为防止节点因管理过多实例而负载过重，请根据您的业务需要进行设置。

新参数	旧参数	配置说明
自定义镜像仓库		单击“新增自定义镜像仓库地址”输入镜像仓库地址。 添加自定义镜像仓库地址（非SSL镜像源地址）到docker启动参数中，避免拉取个人镜像仓库的镜像失败，格式可为“IP地址:端口或者域名”。安装后执行脚本与自定义镜像仓库不能同时使用。
单容器可用数据空间	该参数在之前的CCE版本中不存在。根据业务需要配置。	该参数用于设置一个容器可用的数据空间大小，设置范围为 10G 到 80G。如果设置的参数超过数据盘中Docker可占用的实际数据空间（由数据盘设置项中的数据盘空间分配参数指定，默认为数据盘大小的90%），将以Docker的实际空间大小为主。该参数仅在v1.13.10-r0及以上版本的集群中支持，否则不显示。

步骤6 单击“下一步”，安装插件。

系统资源插件为必装插件，高级功能插件可根据实际需求进行选择性安装。

高级功能插件也可以在集群创建完成后，在左侧导航栏中单击“插件管理”进行安装，详情请参见**插件管理**。

步骤7 单击“立即创建”查看配置信息无误后，单击“提交”。

集群创建预计需要6-10分钟。请根据界面提示查看集群创建过程。

----结束

22.4 迁移应用

本节介绍如何在CCE新版本控制台创建一个与之前版本相同规格的无状态工作负载。

建议在新版本CCE中创建无状态工作负载成功后，再删除旧版本的应用。

22.4.1 通过 API 或 kubectl 创建的应用

若之前版本的应用是通过API或kubectl创建的，请使用相同的方法在新版本的CCE中创建相同规格的无状态工作负载，详细操作请参见《云容器引擎接口参考 2.0》或**通过kubectl连接集群**。

- 通过kubectl连接集群，建议在CCE2.0中操作。在CCE1.0中，需要下载三个证书文件，在客户端手动执行多个步骤。而在CCE2.0中，只需执行以下步骤连接集群。
  a. 下载kubectl命令行配置文件（所有认证信息和集群地址在同一个文件中）。
  b. 安装和配置kubectl命令行工具（执行Linux命令完成对接）。
  具体操作请参见**通过kubectl连接集群**。
- 通过API访问CCE控制台时，需要更新集群管理API。原生Kubernetes接口没有变化，按照之前的方法使用Kubernetes接口即可。具体操作请参见《云容器引擎接口参考 2.0》。

22.4.2 通过组件模板创建的应用

如果老版本应用是使用组件模板创建的，请按照下面的步骤将应用迁移到新CCE版本。

迁移方法

在CCE新版本控制台创建无状态工作负载。应用在新版本运行正常后，再删除旧版本中的应用。

操作步骤

步骤1 在CCE左侧导航栏中选择“工作负载 > 无状态(Deployment)”，单击“创建无状态工作负载”。

步骤2 参照表22-4设置基本信息，其中带“*”标志的参数为必填参数。

表 22-4 基本参数

参数名称	配置说明
* 工作负载名称	新建工作负载的名称，命名必须唯一。
* 集群名称	新建工作负载所在的集群。
* 命名空间	新建工作负载所在的命名空间，默认为default。
* 实例数量	工作负载可以有一个或多个实例，用户可以设置具体实例个数。 每个工作负载实例都由相同的容器部署而成。设置多个实例主要用于实现高可靠性，当某个实例故障时，工作负载还能正常运行。
时区同步	勾选“开启”，容器将和节点使用相同时区。 **须知** 时区同步功能开启后，在“数据存储 > 本地磁盘”中，将会自动添加HostPath类型的磁盘，请勿修改删除该磁盘。
工作负载描述	工作负载描述信息。

步骤3 单击“下一步”，添加容器。

1. 单击“添加容器”，选择需要部署的镜像，单击“确定”。
2. 参照表22-5设置镜像参数。

表 22-5 配置镜像参数

新参数	旧参数	配置说明
镜像名称	容器镜像	导入的镜像，您可单击“更换镜像”进行更换。
镜像版本		镜像版本信息。
容器名称		容器的名称，可修改。

新参数	旧参数	配置说明
特权容器		特权容器是指容器里面的程序具有一定的特权。 若选中，容器将获得超级权限，例如可以操作宿主机上面的网络设备、修改内核参数等。
容器规格	内存、CPU	CPU配额： – 申请：容器需要使用的最小CPU值，默认0.25Core。 – 限制：允许容器使用的CPU最大值。建议设容器配额的最高限额，避免容器资源超额导致系统故障。 内存配额： – 申请：容器需要使用的内存最小值，默认512MiB。 – 限制：允许容器使用的内存最大值。如果超过，容器会被终止。 申请和限制的详情请参见**设置容器规格**。 GPU配额：当集群中包含GPU节点时，才能设置GPU，无GPU节点不显示此选项。 – GPU配额：容器需要使用的GPU百分比。勾选“使用”并设置百分比，例如设置为10%，表示该容器需使用GPU资源的10%。若不勾选“使用”，或设置为0，则无法使用GPU资源。 – GPU显卡：工作负载实例将被调度到GPU显卡类型为指定显卡的节点上。若勾选“不限制”，容器将会随机使用节点中的任一显卡。您也可以勾选某个显卡，容器将使用特定显卡。

3. 参照**表22-6**配置环境变量、数据存储和容器日志。

表 22-6 配置高级参数

新参数	旧参数	配置说明
生命周期	该参数在之前的CCE版本中不存在。对于迁移的应用，不需要配置该参数。	用于管理容器启动和运行时需要执行的命令。 – 启动命令：设置容器启动时执行的命令，详情请参见**设置容器启动命令**。 – 启动后处理：设置容器成功运行后执行的命令，详细配置方法请参见**设置容器生命周期**。 – 停止前处理：设置容器结束前执行的命令，通常用于删除日志/临时文件等，详细配置方法请参见**设置容器生命周期**。

新参数	旧参数	配置说明
健康检查	该参数在之前的CCE版本中不存在。对于迁移的应用，不需要配置该参数。	用于判断容器和用户业务是否正常运行。设置了存活与业务两种探针，详细配置方法请参见**设置容器健康检查**。 – 工作负载存活探针：检查容器是否正常，不正常则重启实例。 – 工作负载业务探针：检查用户业务是否就绪，不就绪则不转发流量到当前实例。
环境变量	环境变量	在“环境变量”页签，单击“添加环境变量”。当前支持三种类型。 – 手动添加：输入变量名称、变量/变量引用。 – 密钥导入：输入变量名称，选择导入的密钥名称和数据。您需要提前创建密钥，详情请参见**创建密钥**。 – 配置项导入：输入变量名称，选择导入的配置项名称和数据。您需要提前创建配置项，详情请参见**创建配置项**。
数据存储	卷	对于老版本组件模板版本的应用，请执行如下操作： 1. 选择“数据存储 > 本地磁盘”，单击“添加本地磁盘”。 2. 选择“主机路径挂载”。 3. 设置如下参数： ■ 主机路径：本地卷挂载的主机路径，对应卷的/tmp目录。 ■ 单击“添加容器挂载”，输入数据卷挂载的容器路径。对应卷的/ test。 ■ 权限：设置为“读写”。 4. 配置完成后，单击“确定”。
安全设置	该参数在之前的CCE版本中不存在。对于迁移的应用，不需要配置该参数。	对容器权限进行设置，保护系统和其他容器不受其影响。请输入用户ID，容器将以当前用户权限运行。
容器日志	该参数在之前的CCE版本中不存在。对于迁移的应用，不需要配置该参数。	设置工作负载日志收集策略和日志目录，防止日志过大，具体操作请参见**采集容器内路径日志**。

步骤4 单击“下一步”。单击“添加服务”，设置工作负载访问方式。

若工作负载需要和其它服务互访，或需要被公网访问，您需要添加服务，设置工作负载访问方式。

工作负载访问的方式决定了这个工作负载的网络属性，不同访问方式的工作负载可以提供不同网络能力。详情请参见**网络管理**。

- 访问类型：本例选择“负载均衡（LoadBalancer）”。
- 服务名称：指定服务名称，可以使用工作负载的名称作为服务名称。
- **服务亲和：**
  - 集群级别：将外部流量路由到集群下所有的节点，并且隐藏客户端源IP。
  - 节点级别：将外部流量路由到服务关联的负载所在的节点，并且保留客户端源IP。
- **负载均衡：**可以将互联网访问流量自动分发到工作负载所在的多个节点上。负载均衡实例需与当前集群处于相同VPC且为相同公私网类型。
  - 公网：支持自动创建和使用已有负载均衡实例两种方式。

    规格配置：选择“公网 > 自动创建”时，单击规格配置下的“更改配置”，可修改待创建的负载均衡实例的名称、规格、计费模式和带宽。
  - 私网：支持自动创建和使用已有负载均衡实例两种方式。
- **端口配置：**
  - 协议：请根据业务的协议类型选择。
  - 容器端口：容器镜像中工作负载实际监听端口，需用户确定。nginx程序实际监听的端口为80。
  - 访问端口：容器端口最终映射到负载均衡服务地址的端口，用负载均衡服务地址访问工作负载时使用，端口范围为1-65535，可任意指定。

步骤5 单击“确定”，单击“下一步”。跳过高级设置。

步骤6 配置完成后，单击“创建”。单击“返回工作负载列表”。

在工作负载列表中，当工作负载状态为“运行中”时，表示工作负载创建成功。

工作负载状态不会实时更新，请单击右上角的 图标或按F5刷新页面查看。

步骤7 在工作负载列表中，复制“外部访问地址”，可在浏览器中访问工作负载。

说明

当工作负载访问方式设为“节点访问（NodePort）”并绑定弹性IP或设为“负载均衡（LoadBalancer）”时，才可以获取外部访问地址，可以访问外网。

----结束

22.4.3 通过设计器创建的应用

如果之前版本的应用是通过图形化编排创建的，请按照本节的步骤将应用迁移到新版本CCE中。在新版本CCE中，使用自定义的HelmTemplate创建无状态工作负载，更多信息，请参见**模板市场**。

操作步骤

步骤1 登录前一个CCE版本的控制台。在左侧导航栏中，单击“应用管理”。单击需要迁移的应用名称，进入该应用的详情页面。获取组件名称和服务名称。

步骤2 连接集群，详细信息请参见**通过kubectl连接集群**。

步骤3 在kubectl客户端执行以下命令，获取应用相关的YAML文件。

kubectl get rc *new-appcomponent-1-11d3* **-o yaml >** *new-appcomponent-1-11d3*.**yaml**

kubectl get svc *new-port-3* **-o yaml >** *new-port-3*.**yaml**

请将命令中*new-appcomponent-1-11d3*和*new-appcomponent-1-11d3*替换为**步骤1**中获取的值。

步骤4 编辑“new appcomponent-1-11d3.yaml”文件，删除加粗部分，修改斜体部分。

```
apiVersion: v1
kind: ReplicationController
metadata:
  annotations:
    cce/app-createTimestamp: 2018-04-19-11-39-27
    cce/app-description: ""
    cce/app-updateTimestamp: 2018-04-19-11-39-27
  creationTimestamp: 2018-04-19T11:37:17Z
  generation: 1
  labels:
    cce/appgroup: app-design
    name: new-appcomponent-1-11d3
    rollingupdate: "false"
  name: new-appcomponent-1-11d3
  namespace: default
  resourceVersion: "8325781"
  selfLink: /api/v1/namespaces/default/replicationcontrollers/new-appcomponent-1-11d3
  uid: 039ada96-43c6-11e8-8f34-fa163e738aa3
spec:
  replicas: 1
  selector:
    cce/appgroup: app-design
    name: new-appcomponent-1-11d3
    rollingupdate: "false"
  template:
    metadata:
      annotations:
        scheduler.alpha.kubernetes.io/affinity: '{"nodeAffinity":
{"requiredDuringSchedulingIgnoredDuringExecution":{"nodeSelectorTerms":[{"matchExpressions":
[{"key":"failure-domain.beta.kubernetes.io/zone","operator":"NotIn","values":[""]}]}]}}}'
      creationTimestamp: null
      labels:
        cce/appgroup: app-design
        name: new-appcomponent-1-11d3
        rollingupdate: "false"
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: NotIn
                values:
                - ""
      containers:
      - image: 10.125.1.108:6443/test/apache-php:latest -->Change it to the image address of the later CCE version
        imagePullPolicy: Always
```

```
      name: new-container-2-11d3f16
      ports:
      - containerPort: 3333
        protocol: TCP
      resources: {}
      securityContext:
        privileged: true
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    imagePullSecrets:
    - name: myregistry    -----> Change it to default-secret
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
status:
  availableReplicas: 1
  fullyLabeledReplicas: 1
  observedGeneration: 1
  readyReplicas: 1
  replicas: 1
```

步骤5 打开文件，删除加粗部分的内容。

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2018-04-19T11:37:17Z
  labels:
    cce/appgroup: app-design
    name: new-appcomponent-1-11d3
  name: new-port-3
  namespace: default
  resourceVersion: "8325766"
  selfLink: /api/v1/namespaces/default/services/new-port-3
  uid: 039c918f-43c6-11e8-8f34-fa163e738aa3
spec:
  clusterIP: 10.247.145.136
  externalTrafficPolicy: Cluster
  ports:
  - name: new-port-30
    nodePort: 31471
    port: 3333
    protocol: TCP
    targetPort: 3333
  selector:
    cce/appgroup: app-design
    name: new-appcomponent-1-11d3
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

步骤6 将kubectl客户端连接到CCE新版本的集群，具体操作请参见**通过kubectl连接集群**。

步骤7 执行以下命令，重新创建应用。

kubectl create -f *new-appcomponent-1-11d3*.**yaml**

kubectl create -f *new-port-3*.**yaml**

----结束