天翼云

# 云容器引擎

## 最佳实践

天翼云科技有限公司

# 目 录

# 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1-1 系统可用性

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| 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。 | • 集群网络地址段规划实践<br>• 云容器引擎 > 常见问题 > 网络管理 > 网络规划 |
| | 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。 | • 虚拟私有云 > 常见问题 > 连接类<br>• 集群网络地址段规划实践 |
| | 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。 | • 虚拟私有云 > 用户指南 > 安全组<br>• 云容器引擎 > 常见问题 > 网络管理 > 安全加固 |
| | 使用多控制节点模 | 可靠 | 多控制节点模式开启 | • 云容器引擎 > 常 |

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| | 式，创建集群时将控制节点数设置为3。 | 性 | 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。 | 见问题 > 集群管理 > 集群运行<br>集群一旦创建，便无法更改控制节点数，需要重新创建集群才能调整，请在创建时谨慎选择。 |
| | 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。 | 云容器引擎 > 用户指南 > 网络管理 > 容器网络模型 |
| 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。 | • 设置容器健康检查<br>• 健康检查 UDP 协议安全组规则说明 |
| | 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）、DNAT 网关（DNAT）。 | 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。 | • 网络管理 |
| | 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 | 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 | - |

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| | 调度策略。 | | 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。 | 亲和性/反亲和性调度策略概述<br>**反例：**<br>应用 A 设置对节点 1、节点 2 亲和性，节点 3、节点 4 反亲和性，应用 A 通过 ELB 发布 Service，ELB 监听节点 1 和节点 2 上面。对应用 A 进行升级，会概率出现应用 A 被调度到节点 1-4 之外的节点，无法通过 Service 访问应用 A。<br>**原因：**<br>Kubernetes 的亲和反亲和调度策略是满足一个就可以调度成功，此时是满足了节点 3、节点 4 反亲和性调度策略。 |
| | 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。 | • 设置容器生命周期<br>• 云容器引擎 > 常见问题 > 工作负载 > 容器设置 |

表1-2 数据可靠性

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| 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 | • 存储管理 |

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| | | | 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。 | 云容器引擎 > 常见问题 > 存储管理 |

表1-3 运维可靠性

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| 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。 | • 云容器引擎 > 常见问题 > 集群 > 集群限制<br>• 使用限制 |
| | 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。 | 详情参见高危操作及解决方案。<br>**反例：**<br>1. 用户升级了节点内核，可能会导致容器网络异常；<br>2. 用户在节点上安装了开源的 Kubernetes 网络插件，导致容器网络异常；<br>3. 用户在节点上将 /var/paas，/mnt/paas/kubernetes 删除，导致该节点异常。 |
| | 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。 | **反例：**<br>1. 在弹性负载均衡页面修改 CCE 创建的监听器名称；<br>2. 在虚拟私有云页面修改 CCE 创建的安全组；<br>3. 在云硬盘页面删除或者卸载 CCE 集群节 |

| 类别 | 评估项目 | 类型 | 影响说明 | FAQ&样例 |
| --- | --- | --- | --- | --- |
| | | | | 点挂载的数据盘；<br>4. 在统一身份认证页面删除 cce 的委托“cce_admin_trust”。<br>以上修改都会导致 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。<br>• 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。<br>• 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。 | • 监控概述<br>• Prometheus 监控多个集群<br>• 使用 dcgm-exporter 监控 GPU 指标 |

# 2 容器化改造

## 2.1 企业管理应用容器化改造（ERP）

### 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。

**表2-1 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：<br>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。<br>• 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。<br>• 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。<br>• 推出新功能、语言、框架和技术都比较困难。 |
| 方式二：<br>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。<br>• 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。<br>• 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。 |
| 方式三：<br>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。<br>• 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 | 业务需要微服务化改造，改动较大。 |

| 应用容器化改造方式 | 优点 | 缺点 |
| --- | --- | --- |
| | 他微服务。<br>• 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。<br>• 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。 | |

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。

## 2.1.2 资源与成本规划

**须知**

**本文提供的成本预估费用仅供参考，资源的实际费用以天翼云管理控制台显示为准。**

完成本实践所需的资源如下：

**表2-2 资源和成本规划**

| 资源 | 资源说明 | 数量 | 参考费用（元） |
| --- | --- | --- | --- |
| 弹性云服务器 ECS | • ECS 虚拟机规格：4 核 8G 或以上规格，Ubuntu 18.04 操作系统。<br>• 建议选择按需计费。 | 1 | 0.652 元/小时 |
| 云容器引擎 CCE | • CCE 集群版本：v1.21。<br>• 虚拟机节点规格：4 核 8G 或以上规格、CentOS 7.6 操作系统。<br>• 建议选择按需计费。 | 1 | 3.69 元/小时 |
| 云硬盘 EVS | • 云硬盘规格：100G。<br>• 建议选择按需计费。 | 1 | 0.09 元/小时 |

## 2.1.3 实施步骤

### 2.1.3.1 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。

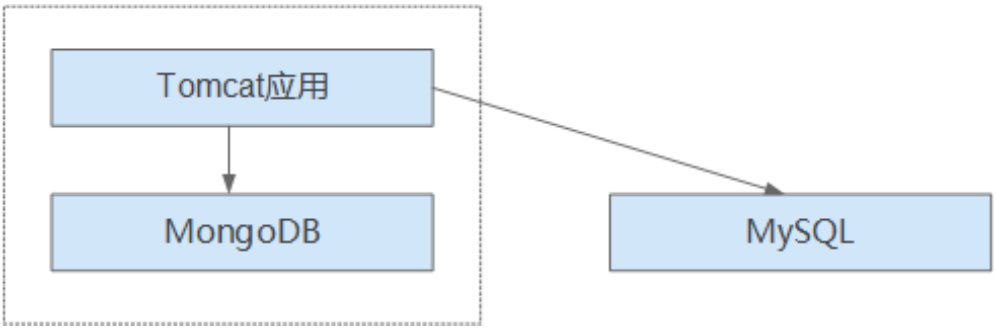
您不需要改动任何代码和架构，仅需将整体应用构建为容器镜像，部署到云容器引擎中。

#### 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2-1 应用架构




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。

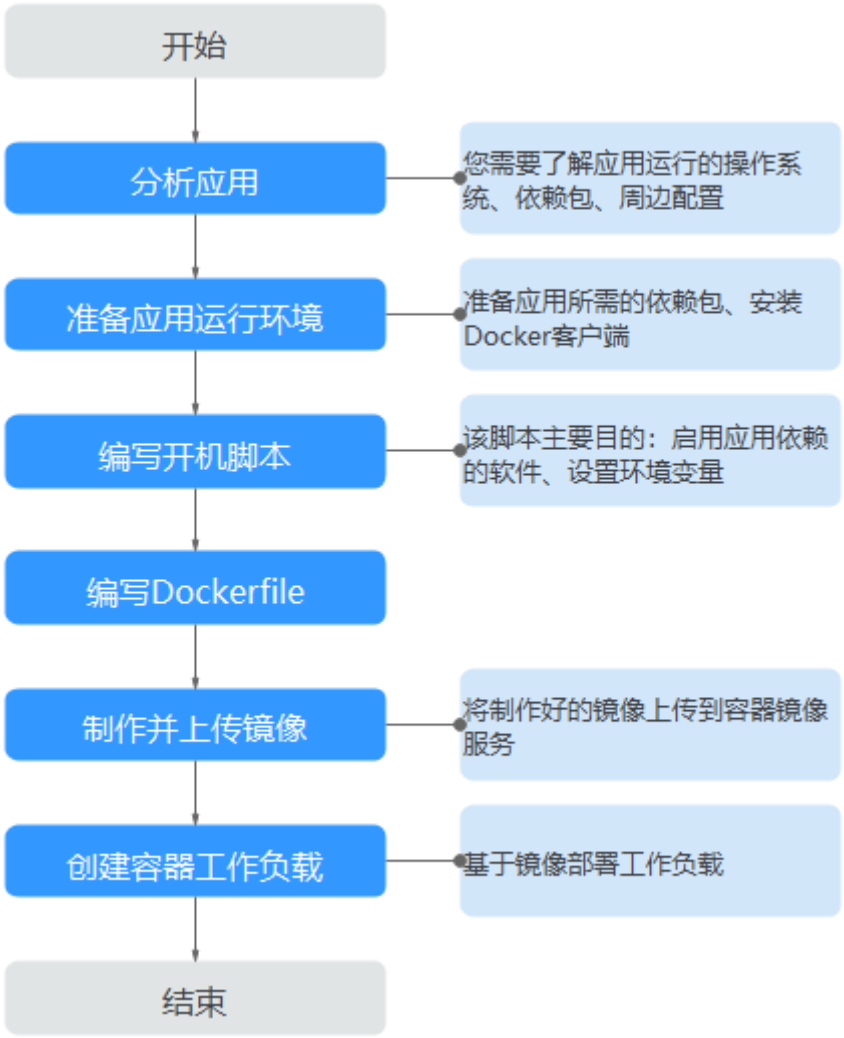
表2-3 虚机和容器部署对比表

| 类别 | before：虚机部署 | after：容器部署 |
| --- | --- | --- |
| 部署 | 部署成本高。<br>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。<br>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。<br>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。<br>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。<br>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。<br>企业无需关注虚拟机的维护，只需要关注业务的开发。 |

## 2.1.3.2 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2-2 容器化改造流程




## 2.1.3.3 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。

表2-4 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。<br>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。<br>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。<br>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。<br>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。<br>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。<br>`url=jdbc:mysql://服务器地址/数据库名称 #数据库连接 URL`<br>`username=**** #数据库登录用户名`<br>`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。<br>本例中，没有需要提取的自身配置项。<br>说明<br>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。<br>• 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。<br>• 不变的配置，可以直接写到镜像中。 |

## 2.1.3.4 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。

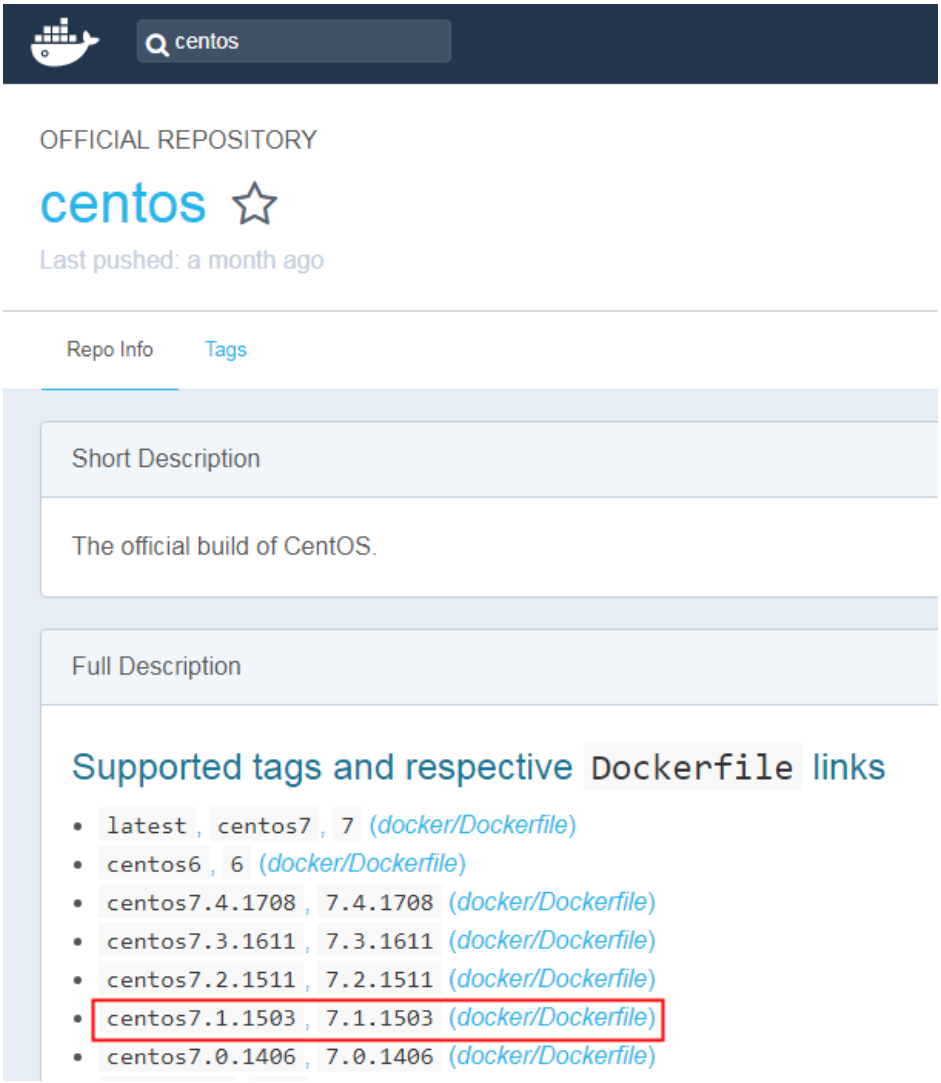
**说明**

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2-3 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，请以应用实际情况进行操作。

**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**

## 2.1.3.5 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**

## 2.1.3.6 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**

## 2.1.3.7 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，上传镜像具体步骤请参见通过客户端上传镜像。

**----结束**

# 2.1.3.8 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，您需要设置如下的运行环境。

**表2-5 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。<br>若您已有虚拟私有云，可重复使用，无需多次创建。<br>1. 登录管理控制台。<br>2. 在服务列表中，选择“网络 > 虚拟私有云”。<br>3. 在“总览”界面，单击“创建虚拟私有云”。<br>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。<br>1. 登录管理控制台。<br>2. 在服务列表中，选择“数据加密服务 DEW”。<br>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”。<br>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”。<br>5. 在弹出的对话框中，单击“确定”。<br>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“购买”。

填写集群参数，选择步骤 1 中创建的 VPC。具体请参见云容器引擎 > 用户指南 > 集群管理 > 购买集群。

2. 购买节点，选择步骤 1 中创建的密钥对作为登录选项。具体请参见云容器引擎 > 用户指南 > 节点管理 > 购买集群。

**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 2.1.3.7 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2-6 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**。

图2-4 设置云存储


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。

# 3 迁移

## 3.1 将自建 K8s 集群迁移到 CCE

### 3.1.1 方案概述

#### 操作场景

随着容器化技术的发展，越来越多的企业使用容器代替了虚拟机完成应用的运行部署，而 Kubernetes 的发展让容器化的部署变得简单并且高效。目前许多企业选择自建 Kubernetes 集群，但是自建集群往往有着沉重的运维负担，需要运维人员自己配置管理系统和监控解决方案，伴随而来的就是企业人力成本的上升和效率的降低。

在性能方面，自建集群的规模固定，可扩展性又比较弱，在业务流量高峰期无法实现自适应的弹性扩缩容，很容易出现集群资源不足或浪费等现象。而且自建集群往往没有考虑容灾风险，导致可靠性较差，一旦出现故障将会使整个集群无法使用，可能会形成十分严重的生产事件。

面对以上的种种不足，CCE 提供了简单的集群管理能力和灵活的弹性放缩能力，能够有效帮助企业简化集群运维管理方式，降低运营成本，以简单易用、高性能、安全可靠、开放兼容等诸多优点，获取了大量企业用户的青睐。因此很多企业选择将自建集群全量搬迁至 CCE 进行管理，本文主要介绍集群迁移上云的方案和步骤。

#### 上云须知

与自建 K8s 集群相比，CCE 集群具有多种优势，同时在 CCE 集群的使用过程中也存在着部分限制，请参见约束与限制，务必在使用前做好评估。

#### 迁移方案

本文介绍一种集群迁移方案，适合如下几类集群：

- 本地 IDC 自建的 K8s 集群
- 通过多台 ECS 自建的集群
- 其他云服务商提供的集群服务

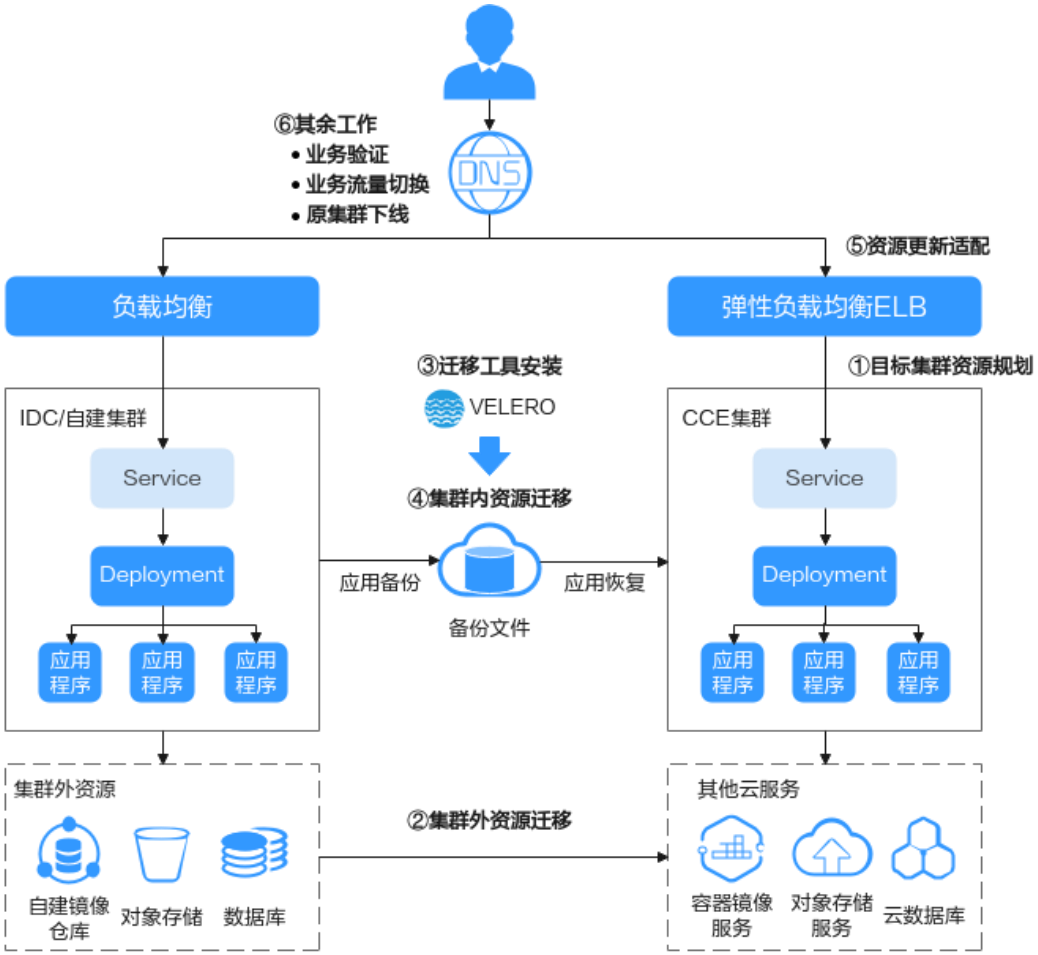
在迁移前，需对原集群的所有资源进行分析再决定迁移方案，可迁移的资源包括集群内资源和集群外资源，如下表所示。

表3-1 可迁移资源列表

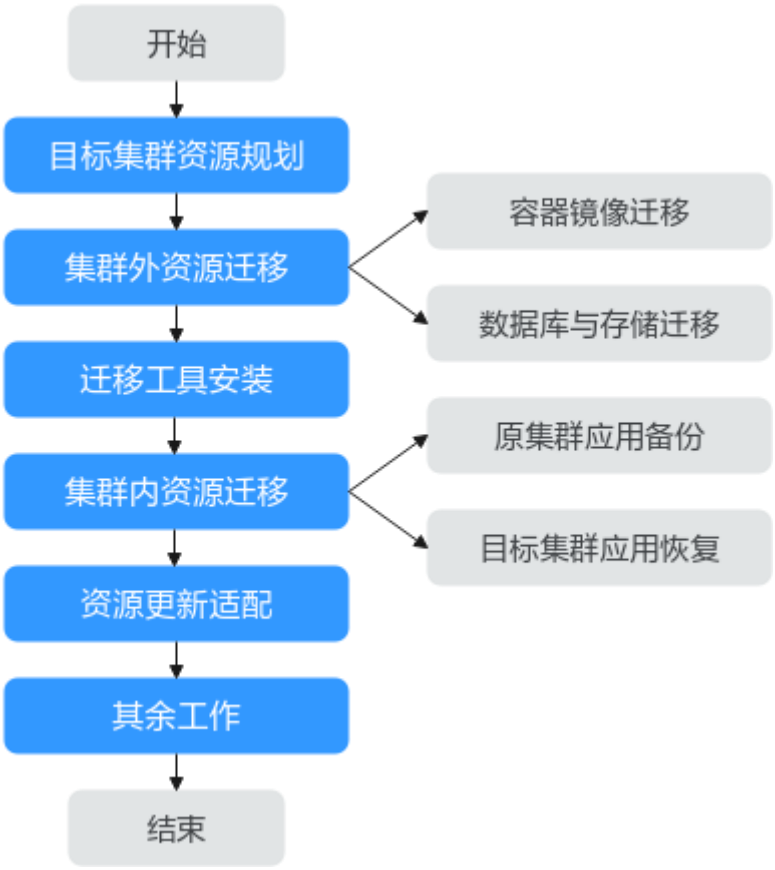
| 资源类别 | 可迁移对象 | 备注 |
| --- | --- | --- |
| 集群内资源 | 集群中的所有对象，Pod、Job、Service、Deployment、ConfigMap 等。 | **不建议迁移的资源：**命名空间 velero 和 kube-system 下的资源。<br>• velero：该命名空间下的资源为迁移工具创建，无需迁移。<br>• kube-system：该命名空间下的资源为系统资源。如原集群该命名空间下包含用户自行创建的资源，建议按需迁移。<br>注意<br>如果您是迁移或备份 CCE 中集群的资源，比如从一个 Namespace 到另外一个 Namespace，请不要备份名称为 paas.elb 的 Secret。因为 paas.elb 的内容是会定期更新，备份后再恢复时可能已经失效，会影响网络存储相关功能。 |
| | 挂载到容器的持久化存储。 | 由于 Restic 工具限制，不支持进行 HostPath 类型存储迁移，解决方法请参考无法备份 HostPath 类型存储卷。 |
| 集群外资源 | 自建镜像仓库。 | 可迁移至容器镜像服务 SWR。 |
| | 非容器化的数据库。 | 可迁移至云数据库服务 RDS。 |
| | 对象存储等非本地存储。 | 可迁移至对象存储服务 OBS 等云存储服务。 |

迁移流程如下图所示，对于集群外资源您可根据实际需求进行选择性迁移。

图3-1 迁移方案示意图


⑥其余工作
• 业务验证
• 业务流量切换
• 原集群下线
DNS
⑤资源更新适配
负载均衡
弹性负载均衡ELB
③迁移工具安装
VELERO
①目标集群资源规划
IDC/自建集群
CCE集群
Service
Service
④集群内资源迁移
Deployment
应用备份
应用恢复
Deployment
备份文件
应用程序
应用程序
应用程序
应用程序
应用程序
应用程序
集群外资源
②集群外资源迁移
其他云服务
自建镜像仓库
对象存储
数据库
容器镜像服务
对象存储服务
云数据库

## 迁移步骤




集群迁移大致包含如下 6 个步骤：

**步骤 1 目标集群资源规划。**

请详细了解 CCE 集群与自建集群间的差异，参考目标集群资源规划中的关键性能参数，按需进行资源规划，建议尽量保持迁移后集群与原集群中性能配置相对一致。

**步骤 2 集群外资源迁移。**

若您需对集群外的相关资源进行迁移，可使用对应的迁移解决方案，具体请参见集群外资源迁移。

**步骤 3 迁移工具安装。**

完成集群外资源迁移后，可通过迁移工具在原集群和目标集群内分别进行应用配置的备份和还原，工具的安装步骤请参考迁移工具安装。

**步骤 4 集群内资源迁移。**

使用 Velero 将原集群内资源备份至对象存储中，并在目标集群中进行恢复，详细步骤可参考集群内资源迁移（Velero）。

- 原集群应用备份

  当用户执行备份时，首先通过 Velero 工具在原集群中创建 Backup 对象，并查询集群相关的数据和资源进行备份，并将数据打包上传至 S3 协议兼容的对象存储中，各类集群资源将以 JSON 格式文件进行存储。

- 目标集群应用恢复

  在目标集群中进行还原时，Velero 将指定之前存储备份数据的临时对象桶，并把备份的数据下载至新集群，再根据 JSON 文件对资源进行重新部署。

步骤 5 **资源更新适配。**

迁移后的集群资源可能存在无法部署的问题，需要对出现错误的资源进行更新适配，可能发生的适配问题主要包括如下几类：

- 镜像更新适配
- 访问服务更新适配
- StorageClass 更新适配
- 数据库更新适配

步骤 6 **其余工作。**

集群资源正常部署后，需对迁移后应用内的功能进行验证，并将业务流量切换至新集群。最后确定所有服务正常运行后，可将原集群下线。

**----结束**

## 3.1.2 目标集群资源规划

CCE 支持对集群资源进行自定义选择，以满足您的多种业务需求。下表列举了集群的主要性能参数，并给出了本示例的规划值，您可根据业务的实际需求大小进行设置，建议与原集群性能配置保持相对一致。

**须知**

**集群创建成功后，下表中带“*”号的资源参数将不可更改，请谨慎选择。**

表3-2 CCE 集群规划

| 资源 | 主要性能参数 | 参数说明 | 本示例规划 |
| --- | --- | --- | --- |
| 集群 | *集群类型 | • **CCE 集群：**支持虚拟机节点。基于高性能网络模型提供全方位、多场景、安全稳定的容器运行环境。<br>• **CCE Turbo 集群：**基于云原生基础设施构建的云原生 2.0 容器引擎服务，具备软硬协同、网络无损、安全可靠、调度智能的优势，为用户提供一站式、高性价比的全新容器服务体验。支持裸金属节点。 | CCE 集群 |
| | *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。<br>• **容器隧道网络（Overlay）：**基于底层 VPC 网络，另构 | VPC 网络 |

| 资源 | 主要性能参数 | 参数说明 | 本示例规划 |
| --- | --- | --- | --- |
| | | 建了独立的 VXLAN 隧道化容器网络，适用于一般场景。<br>• **云原生 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。 | |
| | *****控制节点数 | • **3**：三个控制节点，容灾性能好，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。<br>• **1**：单个控制节点，不建议在商用场景使用。 | 3 |
| 节点 | OS 类型 | • **CentOS**<br>• **Ubuntu** | CentOS |
| | 节点规格（根据实际区域可能存在差异） | • **通用型**：该类型实例提供均衡的计算、存储以及网络配置，适用于大多数的使用场景。通用型实例可用于 Web 服务器、开发测试环境以及小型数据库工作负载等场景。<br>• **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。<br>• **通用入门型**：通用入门型实例提供均衡的计算、存储以及网络配置，利用 CPU 积分机制保证基准性能，适合平时不会持续高压力使用 CPU，但偶尔需要提高计算性能完成工作负载的场景，可用于轻量级 Web 服务器、开发、测试环境以及中低性能数据库等场景。<br>• **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。仅支持 1.11 及以上版本集群添加 GPU 加速型节点。<br>• **高性能计算型**：实例提供具有更稳定、超高性能计算性能的实例，可以用于超高性能计算能力、高吞吐量的工作负载场景，例如科学计算。<br>• **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。<br>• **磁盘增强型**：该类型实例能提供可使用本地磁盘存储以及更高网络性能的实例，可以用于处理需要高吞吐以及高数据交换处理的工作负载，例如大数据工作负载等场景。<br>• **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。 | 通用型（节点规格为 4U8G） |

| 资源 | 主要性能参数 | 参数说明 | 本示例规划 |
| --- | --- | --- | --- |
| | 系统盘类型 | • **高 IO**：后端存储介质为 SAS 类型。<br>• **超高 IO**：后端存储介质为 SSD 类型。 | 高 IO |
| | 存储类型 | • **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。<br>• **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。<br>• **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。<br>• **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。 | 云硬盘存储卷 |

## 3.1.3 集群外资源迁移

若您的集群不涉及集群外资源，或迁移后无需使用其他云服务进行资源替换，可忽略本章节内容。

### 容器镜像迁移

为保证集群迁移后容器镜像可正常拉取，提升容器部署效率，十分建议您将私有镜像迁移至容器镜像服务 SWR。CCE 配合 SWR 为您提供容器自动化交付流水线，采用并行传输的镜像拉取方式，能够大幅提升容器的交付效率。

**步骤 1** 远程登录原集群中任意一个节点，使用 docker pull 命令拉取所有镜像到本地。

**步骤 2** 登录 SWR 控制台，单击页面右上角的“登录指令”并复制。

**步骤 3** 在节点上执行上一步复制的登录指令。

登录成功会显示“Login Succeeded”。

**步骤 4** 为所有本地镜像打上标签。

```
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]：等待上传的本地镜像名称和版本名称。
- [镜像仓库地址]：可在 SWR 控制台上查询。
- [组织名称]：您在 SWR 控制台创建的组织名称。

- [镜像名称 2:版本名称 2]：SWR 中显示的镜像名称和镜像版本。

**示例：**

```
docker tag nginx:v1 registry.cn-jssz1.ctyun.cn/cloud-develop/mynginx:v1
```

步骤 5 使用 docker push 命令将所有本地容器镜像文件上传到 SWR。

```
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 registry.cn-jssz1.ctyun.cn/cloud-develop/mynginx:v1
```

**----结束**

## 数据库与存储迁移（按需）

您可根据实际生产需求，选择是否使用**云数据库服务 RDS** 和**对象存储服务 OBS**。完成迁移后，新建 CCE 集群中的应用需要重新配置数据库与存储。

**数据库迁移**

若您的数据库采用集群外的非容器化部署方案，且需将数据库同步搬迁上云，可以使用**数据复制服务 DRS** 帮助完成数据库迁移。DRS 服务具有实时迁移、备份迁移、实时同步、数据订阅和实时灾备等多种功能。请由运维或者开发人员进行数据库的迁移，详情请参见 DRS 用户指南。完成迁移后，可参考数据库更新适配进行对接。

**存储迁移**

若您的集群对接了对象存储服务，且需同步搬迁至上云，可以使用**云迁移工具 RDA**，帮助您将对象存储中的数据在线迁移至对象存储服务。其他存储类型暂未提供官方工具支持。

请由运维或者开发人员进行对象存储数据的迁移，详情请参见云迁移工具 RDA 帮助中心。完成迁移后，可参考 CCE 用户指南将 OBS 挂载到应用实例。

**说明**

目前云迁移工具 RDA 支持阿里云、微软云、百度云、华为云、金山云、青云、七牛云、腾讯云平台的对象存储数据迁移到天翼云对象存储服务。

# 3.1.4 迁移工具安装

Velero 是开源的 Kubernetes 集群备份、迁移工具，集成了 Restic 工具对 PV 数据的备份能力，可以通过 Velero 工具将原集群中的 K8s 资源对象（如 Deployment、Job、Service、ConfigMap 等）和 Pod 挂载的持久卷数据保存备份上传至对象存储。在发生灾难或需要迁移时，目标集群可使用 Velero 从对象存储中拉取对应的备份，按需进行集群资源的还原。

根据迁移方案所述，在迁移开始前需准备临时的对象存储用于存放资源的备份文件，Velero 支持使用 OBS 或者 **MinIO** 对象存储。对象存储需要准备足够的存储空间用于存放备份文件，请根据您的集群规模和数据量自行估算存储空间。建议您使用 OBS 进行备份存储，可直接参考安装 Velero 进行 Velero 的部署。

源自建集群和 CCE 中目标集群都需要安装 Velero。

## 前提条件

- 原始自建集群 Kubernetes 版本需 1.10 及以上，且集群可正常使用 DNS 与互联网服务。
- 若您使用 OBS 存放备份文件，需已有 OBS 操作权限用户的 AK/SK。
- 若您使用 MinIO 存放备份文件，则安装 MinIO 的服务器需要绑定 EIP 并在安全组中开放 MinIO 的 API 端口和 Console 端口。
- 已创建迁移的目标 CCE 集群。
- 原集群和目标集群中需要至少各拥有一个空闲节点，节点规格建议为 4U8G 及以上。

## 安装 MinIO

MinIO 是一个兼容 S3 接口协议的高性能对象存储开源工具。若使用 MinIO 进行存放集群迁移的备份文件，您需要一台临时服务器用于部署 MinIO 并对外提供服务。若您使用 OBS 存放备份文件，请忽略此步骤，前往安装 Velero。

MinIO 的安装位置选择有如下几种：

- 集群外临时 ECS

  将 MinIO 服务端安装在集群外，能够保障集群发生灾难性故障时，备份文件不会受到影响。
- 集群内的空闲节点

  您可以远程登录节点安装 MinIO 服务端，也可以选择容器化安装 MinIO，请参考 Velero 官方文档 https://velero.io/docs/v1.7/contributions/minio/#set-up-server。

**须知**

**如使用容器化安装 MinIO：**

- **Velero 官方文档提供的 YAML 文件中存储类型为 empty dir，建议将其修改为 HostPath 或 Local 类型，否则容器重启后将永久丢失备份文件。**
- **您需将 MinIO 服务对外提供访问，否则将无法在集群外下载备份文件，可选择将 Service 修改为 NodePort 类型或使用其他类型的公网访问服务。**

无论使用何种方法进行部署，安装 MinIO 的服务器需要有**足够的存储空间**，且均需要**绑定 EIP** 并**在安全组中开放 MinIO 的服务端口**，否则将无法上传（下载）备份文件。

本示例选择在一台集群外的临时 ECS 上安装 MinIO，步骤如下。

**步骤 1** 下载 MinIO 对象存储。

```
mkdir /opt/minio
mkdir /opt/miniodata
cd /opt/minio
```

```
wget https://dl.minio.io/server/minio/release/linux-amd64/minio
chmod +x minio
```

步骤 2 设置 MinIO 的用户名及密码。

此方法设置的用户名及密码为临时环境变量，在服务重启后需要重新设定，否则会使用默认 root 凭据 minioadmin:minioadmin 来创建服务。

```
export MINIO_ROOT_USER=minio
export MINIO_ROOT_PASSWORD=minio123
```

步骤 3 创建服务，其中/opt/miniodata/为 MinIO 存储数据的本地磁盘路径。

MinIO 的 API 端口默认为 9000，console 端口默认为随机生成，您可使用--console-address 参数指定 console 访问端口。

```
./minio server /opt/miniodata/ --console-address ":30840" &
```

**说明**

安装 MinIO 工具的服务器需开放防火墙、安全组中对应的 API 和 console 端口，否则将无法访问对象桶。

步骤 4 浏览器访问 http://{minio 所在节点的 eip}:30840，可进入 MinIO console 界面。

**----结束**

## 安装 Velero

首先前往 OBS 控制台或 MinIO console 界面，创建存放备份文件的桶并命名为 **velero**。此处桶名称可自定义，但安装 Velero 时必须指定此桶名称，否则将无法访问导致备份失败，参见步骤 4。

**须知**

- **原集群和目标集群**中均需要安装部署 Velero 实例，安装步骤一致，分别用于备份和恢复。
- CCE 集群的 Master 节点不对外提供远程登录端口，您可通过 kubectl 操作集群完成 Velero 安装。
- 如果备份资源量较大，请调整 Velero 及 Restic 工具的 cpu 和内存资源（建议调整至 1U1G 及以上），请参考备份工具资源分配不足。
- 用于存放备份文件的对象存储桶**需要是空桶**。

从 Velero 官方发布路径 https://github.com/vmware-tanzu/velero/releases 下载最新的稳定版二进制文件，本文以 Velero 1.7.0 版本为例。原集群和目标集群中的安装过程一致，请参考如下步骤。

步骤 1 下载 Velero 1.7.0 版本的二进制文件。

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.7.0/velero-v1.7.0-linux-amd64.tar.gz
```

步骤 2 安装 Velero 客户端。

```
tar -xvf velero-v1.7.0-linux-amd64.tar.gz
cp ./velero-v1.7.0-linux-amd64/velero /usr/local/bin
```

步骤 3 创建备份对象存储访问密钥文件 credentials-velero。

```
vim credentials-velero
```

文件内容如下，其中的 AK/SK 请根据实际情况进行替换。如使用 MinIO，此处 AK/SK 则为步骤 2 中所创建的用户名及密码。

```
[default]
aws_access_key_id = {AK}
aws_secret_access_key = {SK}
```

步骤 4 部署 Velero 服务端。注意其中--bucket 参数需要修改为已创建的对象存储桶名称，本例中为 **velero**。关于更多自定义安装参数，请参考自定义安装 Velero。

```
velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:v1.2.1 \
  --bucket velero \
  --secret-file ./credentials-velero \
  --use-restic \
  --use-volume-snapshots=false \
  --backup-location-config region=cn-
jssz1,s3ForcePathStyle="true",s3Url=http://obs.cn-jssz1.ctyun.cn
```

表3-3 Velero 安装参数说明

| 安装参数 | 参数说明 |
| --- | --- |
| --provider | 声明使用“aws”提供的插件类型。 |
| --plugins | 使用 AWS S3 兼容的 API 组件，本文使用的 OBS 和 MinIO 对象存储均支持该 S3 协议。 |
| --bucket | 用于存放备份文件的对象存储桶名称，需提前创建。 |
| --secret-file | 访问对象存储的密钥文件，即步骤 3 中创建的“credentials-velero”文件。 |
| --use-restic | 使用 Restic 工具支持 PV 数据备份，建议开启，否则将无法备份存储卷资源。 |
| --use-volume-snapshots | 是否创建 VolumeSnapshotLocation 对象进行 PV 快照，需要提供快照程序支持。该值设为 false。 |
| --backup-location-config | 对象存储桶相关配置，包括 region、s3ForcePathStyle、s3Url 等。 |
| region | 对象存储桶所在区域。<br>• OBS：请根据实际区域填写，如“cn-jssz1”。 |

| 安装参数 | 参数说明 |
| --- | --- |
| | • MinIO：参数值为 minio。 |
| s3ForcePathStyle | 参数值为“true”，表示使用 S3 文件路径格式。 |
| s3Url | 对象存储桶的 API 访问地址。<br>• OBS：该参数值需根据对象存储桶地域决定，参数值为“http://obs.{region}.ctyun.cn”。例如区域为苏州（cn-jssz1），则参数值为“http://obs.cn-jssz1.ctyun.cn”。<br>• MinIO：该参数值需根据 MinIO 安装节点的 IP 及暴露端口确定，参数值为“http://{minio 所在节点的 eip}:9000”。<br>说明<br>• s3Url 中的访问端口需填写 MinIO 的 API 端口，而非 console 端口。MinIO API 端口默认为 9000。<br>• 访问集群外安装的 MinIO 时，需填写其公网 IP 地址。 |

步骤 5 Velero 实例将默认创建一个名为 velero 的 namespace，执行以下命令可查看 pod 状态。

```
$ kubectl get pod -n velero
NAME                READY   STATUS    RESTARTS   AGE
restic-rn29c          1/1     Running   0          16s
velero-c9ddd56-tkzpk   1/1     Running   0          16s
```

**说明**

为防止在实际生产环境中备份时出现内存不足的情况，建议您参照备份工具资源分配不足，修改 Restic 和 Velero 分配的 CPU 和内存大小。

步骤 6 查看 Velero 工具与对象存储的对接情况，状态需要为 available。

```
$ velero backup-location get
NAME      PROVIDER   BUCKET/PREFIX   PHASE       LAST VALIDATED                ACCESS
MODE   DEFAULT
default   aws        velero          Available   2021-10-22 15:21:12 +0800 CST
ReadWrite    true
```

----结束

## 3.1.5 集群内资源迁移（Velero）

### 操作场景

本文使用 Wordpress 应用为例，将自建 Kubernetes 集群中应用整体迁移到 CCE 集群。Wordpress 应用包含 Wordpress 和 MySQL 两个组件，均为容器化实例，分别绑定了两个 Local 类型的本地存储卷，并通过 NodePort 服务对外提供访问。

迁移前通过浏览器访问 Wordpress 站点，创建站点名称为“Migrate to CCE”，并发布一篇文章用于验证迁移后 PV 数据的完整性。Wordpress 中发布的文章会被存储在

MySQL 数据库的“wp_posts”表中，若迁移成功，数据库中的内容也将会被全量搬迁至新集群，可依此进行 PV 数据迁移校验。

## 前提条件

- 请在迁移前提前清理原集群中异常的 Pod 资源。当 Pod 状态异常但是又挂载了 PVC 的资源时，在集群迁移后，PVC 状态会处于 pending 状态。
- 请确保 CCE 侧集群中没有与被迁移集群侧相同的资源，因为 Velero 工具在检测到相同资源时，默认不进行恢复。
- 为确保集群迁移后容器镜像资源可以正常拉取，请将镜像资源迁移至容器镜像服务（SWR）。
- CCE **不支持 ReadWriteMany** 的云硬盘存储，在原集群中存在该类型资源时，需要先修改为 ReadWriteOnce。
- Velero 集成 Restic 工具对存储卷进行备份还原，当前**不支持 HostPath 类型**的存储卷，详情请参见 Restic 限制。若您需备份该类型的存储卷，请参考无法备份 HostPath 类型存储卷将 HostPath 类型替换为 Local 类型。当备份任务中存在 HostPath 类型的存储，该类型存储卷将会被自动跳过并产生 Warning 信息，并不会导致备份失败。

## 原集群应用备份

**步骤 1** （可选）如果需要对 Pod 中指定的存储卷数据进行备份，需对 Pod 添加 annotation，标记模板如下：

```
kubectl -n <namespace> annotate <pod/pod_name> backup.velero.io/backup-volumes=<volume_name_1>,<volume_name_2>,...
```

- <namespace>：Pod 所在的 namespace。
- <pod_name>：Pod 名称。
- <volume_name>：Pod 挂载的持久卷名称。可通过 describe 语句查询 Pod 信息，Volume 字段下即为该 Pod 挂载的所有持久卷名称。

对 Wordpress 和 MySQL 的 Pod 添加 annotation，pod 名称分别为 wordpress-758fbf6fc7-s7fsr 和 mysql-5ffdfbc498-c45lh。由于 Pod 在默认命名空间 default 下，-n <NAMESPACE>参数可省略：

```
kubectl annotate pod/wordpress-758fbf6fc7-s7fsr backup.velero.io/backup-volumes=wp-storage
kubectl annotate pod/mysql-5ffdfbc498-c45lh backup.velero.io/backup-volumes=mysql-storage
```

**步骤 2** 对应用进行备份。备份时可以根据参数指定资源，若不添加任何参数，则默认对整个集群资源进行备份，详细参数请参考 Resource filtering。

- **--default-volumes-to-restic：**表示使用 Restic 工具对 Pod 挂载的所有存储卷进行备份，**不支持 HostPath 类型**的存储卷。如不指定该参数，将默认对步骤 1 中 annotation 指定的存储卷进行备份。此参数仅在安装 Velero 时指定“--use-restic”后可用。

  ```
  velero backup create <backup-name> --default-volumes-to-restic
  ```

- **--include-namespaces：**用于指定 namespace 下的资源进行备份。

```
velero backup create <backup-name> --include-namespaces <namespace>
```

- **--include-resources：**用于指定资源进行备份。

  ```
  velero backup create <backup-name> --include-resources deployments
  ```

- **--selector：**用于指定与 selector 相匹配的资源备份。

  ```
  velero backup create <backup-name> --selector <key>=<value>
  ```

本文指定 default 命名空间下的资源进行备份，wordpress-backup 为备份名称，进行应用恢复时也需指定相同的备份名称。示例如下：

```
velero backup create wordpress-backup --include-namespaces default --default-volumes-to-restic
```

回显如下，表示成功创建备份任务：

```
Backup request "wordpress-backup" submitted successfully. Run `velero backup describe wordpress-backup` or `velero backup logs wordpress-backup` for more details.
```

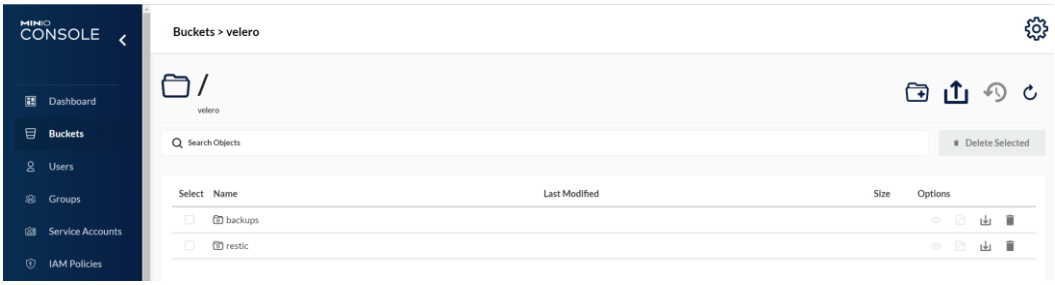
**步骤 3** 查看备份情况。

```
velero backup get
```

回显如下：

```
NAME              STATUS      ERRORS   WARNINGS   CREATED                         EXPIRES
STORAGE LOCATION   SELECTOR
wordpress-backup   Completed   0        0          2021-10-14 15:32:07 +0800 CST   29d
default            <none>
```

同时可前往对象桶中查看备份的文件，其中 backups 路径为应用资源备份，restic 路径为 PV 数据备份。




**----结束**

## 目标集群应用恢复

由于自建集群与后端的存储基础设施不同，集群迁移后会遇到 Pod 无法挂载 PV 的问题。因此在进行迁移时需要对新集群中的 StorageClass 进行适配，从而在创建工作负载时可以屏蔽两个集群之间底层存储接口的差异，申请相应类型的存储资源，相关操作请参考 StorageClass 更新适配。

**步骤 1** 通过 kubectl 连接 CCE 集群，这里选择创建一个原集群中相同名称 StorageClass 来完成适配。

本例中原集群的 StorageClass 名为 local，存储类型为本地磁盘。本地磁盘存储完全依赖节点可用性，数据容灾性能差，节点不可用时将直接影响已有存储数据。因此 CCE 集群中的存储资源选用云硬盘存储卷，后端存储介质使用 SAS 存储。

**说明**

- 包含 PV 数据的应用在 CCE 集群中进行恢复时，定义的 StorageClass 将会根据 PVC 动态创建相应的存储资源（如云硬盘）并挂载，请您知悉。
- 此处集群的存储资源可以根据需求进行更改，并非仅限于云硬盘存储卷。若需要挂载其他类型的存储，如文件存储、对象存储，请参考 StorageClass 更新适配进行适配。

被迁移侧集群 YAML 文件：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

迁移侧集群 YAML 文件示例如下：

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-disk
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
```

步骤 2 使用 Velero 工具创建 restore，指定名称为 wordpress-backup 的备份，将 Wordpress 应用恢复至 CCE 集群。

```
velero restore create --from-backup wordpress-backup
```

可通过 **velero restore get** 语句查看应用恢复情况。

步骤 3 恢复完成后查看应用实例是否正常运行，可能存在其他的更新适配问题，请参考 3.1.6 资源更新适配中的步骤排查解决。

**----结束**

# 3.1.6 资源更新适配

## 镜像更新适配

由于本例使用的 Wordpress 和 MySQL 镜像均可从 SWR 正常拉取，因此不会出现镜像拉取失败（ErrImagePull）问题。如迁移应用为私有镜像，请执行以下步骤完成镜像更新适配。

**步骤 1** 将镜像资源迁移至容器镜像服务（SWR），具体步骤请参考 SWR 用户指南 > 镜像管理 > 客户端上传镜像。

**步骤 2** 登录 SWR 控制台查看获取迁移后的镜像地址。

镜像地址格式如下：

```
'registry.{区域}.ctyun.cn/{所属组织名称}/{镜像名称}:{版本名称}'
```

**步骤 3** 使用如下命令对工作负载进行修改，并将 YAML 文件中的 image 字段替换成迁移后的镜像地址。

```
kubectl edit deploy wordpress
```

**步骤 4** 查看应用实例运行情况。

**----结束**

## 访问服务更新适配

集群迁移后，原有集群的访问服务可能无法生效，可执行如下步骤更新服务。如原集群中设置了 Ingress 资源，迁移后需重新对接 ELB，您可参考云容器引擎用户指南 > 网络管理 > Ingress，添加 Ingress-对接已有 ELB。

**步骤 1** 通过 kubectl 连接集群。

**步骤 2** 编辑对应 Service 的 YAML 文件，修改服务类型及端口。

```
kubectl edit svc wordpress
```

LoadBanlancer 资源进行更新时，需要重新对接 ELB。请参考如下配置，通过 kubectl 命令行创建-使用已有 ELB，添加如下 Annotation：

```
annotations:
  kubernetes.io/elb.class: union   #共享型 ELB
  kubernetes.io/elb.id: 9d06a39d-xxxx-xxxx-xxxx-c204397498a3    #ELB 的 ID，可前往 ELB 控制台查询
  kubernetes.io/elb.subnet-id: f86ba71c-xxxx-xxxx-xxxx-39c8a7d4bb36    #集群所在子网的 ID
  kubernetes.io/session-affinity-mode: SOURCE_IP    #开启会话保持，基于源 IP 地址
```

**步骤 3** 浏览器访问查看服务是否可用。

**----结束**

## StorageClass 更新适配

由于集群的存储基础设施不同，迁移后的集群将无法正常挂载存储卷，您可执行以下方法的任意一种来完成存储卷的更新适配。

**须知**

两种 StorageClass 的适配方法均需在**目标集群**中于**恢复应用前**完成，否则可能出现 PV 数据资源无法恢复的情况，此时在完成 StorageClass 适配后使用 Velero 重新恢复应用即可，请参考目标集群应用恢复。

**方式一：创建 ConfigMap 映射**

步骤 1 在 CCE 集群中创建如下所示的 ConfigMap，将原集群使用的 StorageClass 映射到 CCE 集群默认的 StorageClass。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: change-storageclass-plugin-config
  namespace: velero
  labels:
    app.kubernetes.io/name: velero
    velero.io/plugin-config: "true"
    velero.io/change-storage-class: RestoreItemAction
data:
  {原集群 StorageClass name01}: {目标集群 StorageClass name01}
  {原集群 StorageClass name02}: {目标集群 StorageClass name02}
```

步骤 2 执行以下命令，应用上述的 ConfigMap 配置。

```
$ kubectl create -f change-storage-class.yaml
configmap/change-storageclass-plugin-config created
```

**----结束**

**方式二：创建同名 StorageClass**

步骤 1 查询 CCE 支持的默认 StorageClass。

```
kubectl get sc
```

回显如下：

```
NAME              PROVISIONER                  RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
csi-disk          everest-csi-provisioner        Delete          Immediate
true                   3d23h
csi-disk-topology   everest-csi-provisioner        Delete
WaitForFirstConsumer   true                   3d23h
csi-nas           everest-csi-provisioner        Delete          Immediate
true                   3d23h
csi-obs           everest-csi-provisioner        Delete          Immediate
```

```
false                3d23h
csi-sfsturbo       everest-csi-provisioner       Delete          Immediate
true                 3d23h
```

表3-4 StorageClass

| StorageClass 名称 | 对应的存储资源 |
| --- | --- |
| csi-disk | 云硬盘 |
| csi-disk-topology | 延迟绑定的云硬盘 |
| csi-nas | 文件存储 |
| csi-obs | 对象存储 |
| csi-sfsturbo | 极速文件存储 |

步骤 2 通过如下命令将所需的 StorageClass 详细信息输出为 YAML 格式。

```
kubectl get sc <storageclass-name> -o=yaml
```

步骤 3 复制 YAML 文件并创建一个新的 StorageClass。

编辑 StorageClass 名称，将其命名为原有集群中使用的名称，用于调用云上基础存储资源。

以 csi-obs 的 YAML 文件为例。请删除 metadata 字段下如斜体部分所示的不必要信息，并修改加粗部分，其余参数不建议修改。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  creationTimestamp: "2021-10-18T06:41:36Z"
  name: <your_storageclass_name>     #命名为原有集群中使用的 StorageClass 名称
  resourceVersion: "747"
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-obs
  uid: 4dbbe557-ddd1-4ce8-bb7b-7fa15459aac7
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: Delete
volumeBindingMode: Immediate
```

**说明**

- 极速文件存储无法通过 StorageClass 直接创建，您需要前往 SFS Turbo 控制台创建相同 VPC 子网下且安全组开放入方向端口(111、445、2049、2051、2052、20048)的极速文件存储。
- CCE 不支持 ReadWriteMany 的云硬盘存储，在原集群中存在该类型资源时，需要先修改为 ReadWriteOnce。

步骤 4 参考目标集群应用恢复进行集群应用恢复，检查 PVC 是否创建成功。

```
kubectl get pvc
```

回显如下，其中 VOLUME 列为通过 StorageClass 自动创建的 PV 名称。

```
NAME   STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
pvc    Bound    pvc-4c8e655a-1dbc-4897-ae6c-446b502f5e77   5Gi        RWX
local          13s
```

**----结束**

## 数据库更新适配

本例中数据库为本地 MySQL 数据库，迁移后无需重新配置。若您通过**数据复制服务 DRS** 将本地数据库迁移至云数据库 RDS，则在迁移后需重新配置数据库的访问，请您根据实际情况进行配置。

**说明**

- 若云数据库 RDS 实例与 CCE 集群处于同一 VPC 下，则可通过内网地址访问，否则只能通过绑定 EIP 的方式进行公网访问。建议使用内网访问方式，安全性高，并且可实现 RDS 的较好性能。
- 请确认 RDS 所在安全组入方向规则已对集群放通，否则将连接失败。

**步骤 1** 登录 RDS 控制台，在该实例的“基本信息”页面获取其“内网地址”及端口。

**步骤 2** 使用如下命令对 Wordpress 工作负载进行修改。

```
kubectl edit deploy wordpress
```

设置 env 字段下的环境变量：

- WORDPRESS_DB_HOST：数据库的访问地址和端口，即上一步中获取的内网地址及端口。
- WORDPRESS_DB_USER：访问数据库的用户名。
- WORDPRESS_DB_PASSWORD：访问数据库的密码。
- WORDPRESS_DB_NAME：需要连接的数据库名。

**步骤 3** 检查 RDS 数据库是否正常连接。

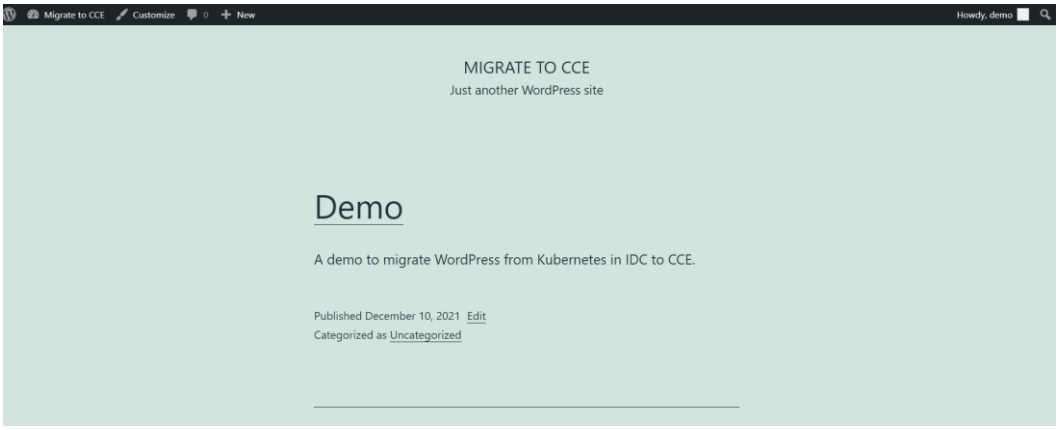
**----结束**

# 3.1.7 其余工作

## 应用功能验证

由于集群迁移是对应用数据的全量迁移，可能存在应用内适配问题。例如本示例中，集群迁移后，Wordpress 中发布的文章跳转链接仍是原域名，单击文章标题将会重定向至原集群中的应用实例，因此需要通过搜索将 Wordpress 中原有的旧域名并替换为新域名，并修改数据库中的 site_url 和主 url 值，具体操作可参考更改站点 URL。

最后在浏览器上访问迁移后的 Wordpress 应用新地址，可以看到迁移前发布的文章，说明持久卷的数据还原成功。




### 业务流量切换

由运维人员做 DNS 切换，将流量引到新集群。

- DNS 流量切换：调整 DNS 配置实现流量切换。
- 客户端流量切换：升级客户端代码或更新配置实现流量切换。

### 原集群下线

由运维人员确认新集群业务正常后，下线原集群并清理备份文件。

- 确认新集群业务正常。
- 下线原集群。
- 清理备份文件。

## 3.1.8 异常排查及解决

### 无法备份 HostPath 类型存储卷

HostPath 与 Local 均为本地存储卷，但由于 Velero 集成的 Restic 工具无法对 HostPath 类型的 PV 进行备份，只支持 Local 类型，因此需要在原集群中将 HostPath 类型存储卷替换为 Local 类型。

**说明**

Local volume 类型的存储建议在 Kubernetes v1.10 及以上的版本中使用，且只能静态创建，详情请参考 local。

步骤 1 创建 Local volume 的 StorageClass。

YAML 文件如下：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
```

```
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

步骤 2 需要修改 hostPath 字段为 local 字段，指定宿主机原有的本地磁盘路径，并添加 nodeAffinity 字段。

YAML 文件示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    app: mysql
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 5Gi
  storageClassName: local     #指定上一步创建的 StorageClass
  persistentVolumeReclaimPolicy: Delete
  local:
    path: "/mnt/data"     #指定挂载的本地磁盘路径
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: Exists
```

步骤 3 执行以下命令验证创建结果。

```
kubectl get pv
```

回显如下：

```
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   REASON   AGE
mysql-pv   5Gi       RWO          Delete         Available                   local
3s
```

**----结束**

## 备份工具资源分配不足

在生产环境中，当备份资源较多时，如备份工具资源大小使用默认值，可能会出现资源分配不足的情况，请参考以下步骤调整分配给 Velero 和 Restic 的 CPU 和内存大小。

**安装 Velero 前：**

您可在安装 Velero 时指定 Velero 和 Restic 使用的资源大小。

安装参数示例如下：

```
velero install \
  --velero-pod-cpu-request 500m \
  --velero-pod-mem-request 1Gi \
```

```
  --velero-pod-cpu-limit 1000m \
  --velero-pod-mem-limit 1Gi \
  --use-restic \
  --restic-pod-cpu-request 500m \
  --restic-pod-mem-request 1Gi \
  --restic-pod-cpu-limit 1000m \
  --restic-pod-mem-limit 1Gi
```

**安装 Velero 后：**

步骤 1 编辑命名空间 velero 下 Velero 和 Restic 工作负载的 YAML 文件。

```
kubectl edit deploy velero -n velero
kubectl edit deploy restic -n velero
```

步骤 2 修改 resources 字段下分配的资源大小，Velero 和 Restic 工作负载的修改内容一致，如下所示。

```
resources:
  limits:
    cpu: "1"
    memory: 1Gi
  requests:
    cpu: 500m
    memory: 1Gi
```

**----结束**

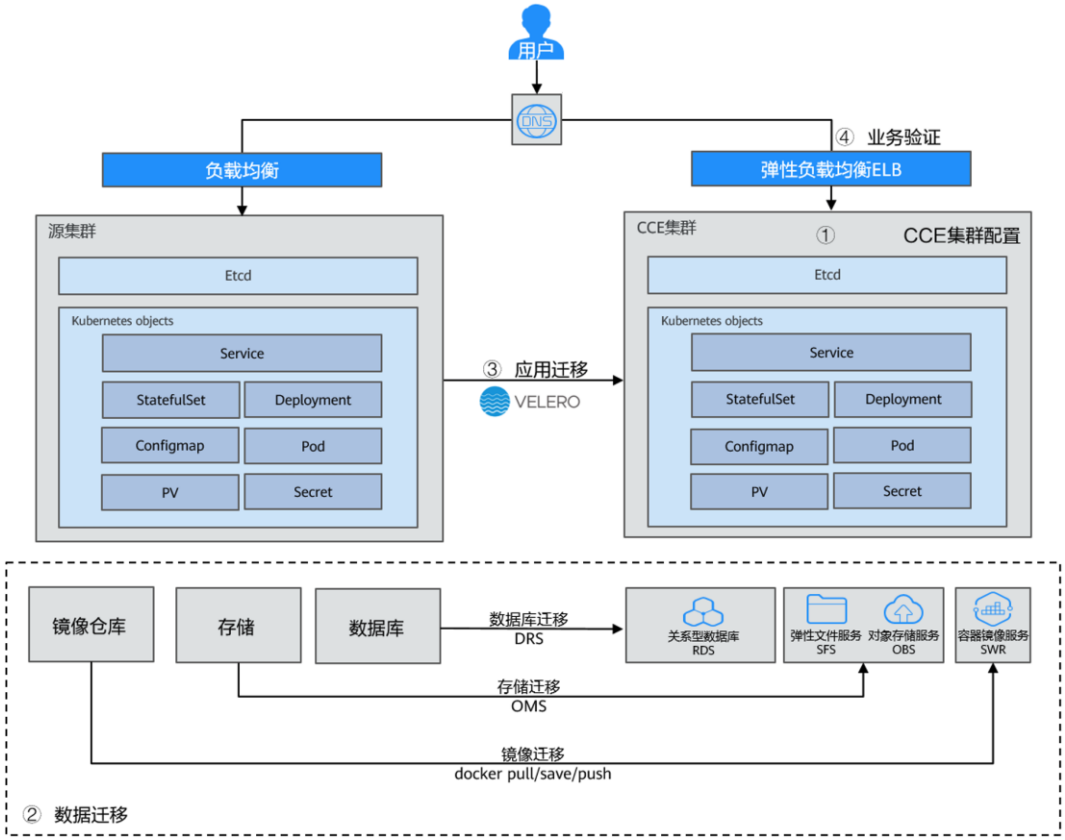
# 3.2 将第三方云集群迁移到 CCE

## 3.2.1 方案概述

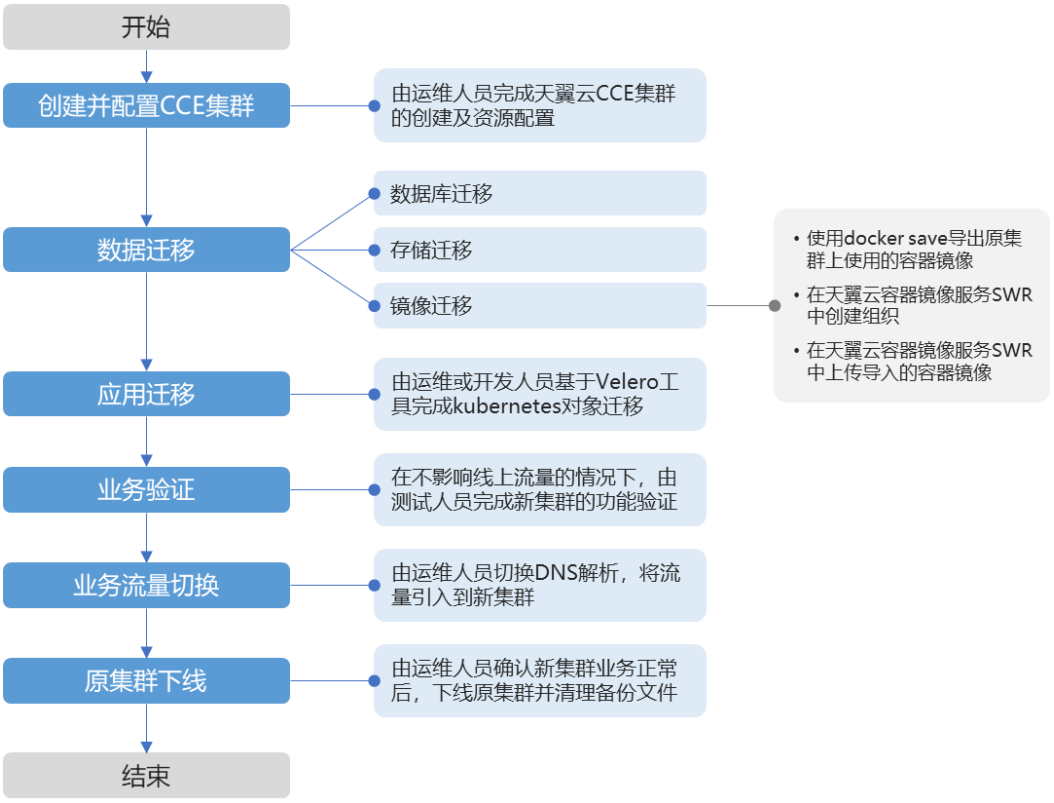
本文将以 WordPress 应用从阿里云 ACK 迁移到天翼云 CCE 为例来讲解，并假设您已在阿里云上部署了 WordPress 应用，并创建了自己的博客。

本文简单介绍如何通过如下 6 个步骤，将应用从阿里云 ACK 集群平滑迁移到 CCE 集群，并尽量确保迁移期间对业务无影响。

# 迁移方案


用户
DNS
负载均衡
④ 业务验证
弹性负载均衡ELB
源集群
Etcd
Kubernetes objects
Service
StatefulSet
Deployment
Configmap
Pod
PV
Secret
③ 应用迁移
VELERO
CCE集群
①
CCE集群配置
镜像仓库
存储
数据库
数据库迁移
DRS
关系型数据库
RDS
弹性文件服务
SFS
对象存储服务
OBS
容器镜像服务
SWR
存储迁移
OMS
镜像迁移
docker pull/save/push
② 数据迁移

## 迁移步骤




# 3.2.2 资源与成本规划

**须知**

本文提供的成本预估费用仅供参考，资源的实际费用以天翼云管理控制台显示为准。

完成本实践所需的资源如下：

**表3-5 资源和成本规划**

| 资源 | 资源说明 | 数量 | 每月费用（元） |
| --- | --- | --- | --- |
| 云容器引擎 CCE | • CCE 集群版本：v1.21。<br>• 虚拟机节点规格：4 核 8G 或以上规格、CentOS 7.6 操作系统。<br>• 建议选择按需计费。 | 1 | 3.69 元/小时 |

## 3.2.3 实施步骤

### 3.2.3.1 数据迁移

#### 数据库与存储迁移

**数据库迁移**

由运维或者开发人员基于**数据复制服务 DRS** 完成数据库迁移，详情请参见数据库复制服务用户指南。

**存储迁移**

由运维或者开发人员基于**云迁移工具 RDA** 完成对象存储中的数据迁移，详情请参见云迁移工具 RDA 帮助中心。

**说明**

目前云迁移工具 RDA 支持将阿里云、微软云、百度云、华为云、金山云、青云、七牛云、腾讯云平台平台的对象存储数据迁移到对象存储服务 OBS。

- 在对象存储服务 OBS 上创建桶，详情请参见 OBS 帮助中心。
- 在云迁移工具 RDA 上创建迁移任务，详情请参见云迁移工具帮助中心。

#### 容器镜像迁移

步骤 1 导出 ACK 集群上使用的容器镜像。

按照阿里云容器镜像服务上的操作指南拉取镜像到客户端机器。

步骤 2 将导出的容器镜像文件上传到天翼云 SWR。

使用 docker pull 命令将镜像上传到 SWR，具体操作方法请参见客户端上传镜像。

**----结束**

### 3.2.3.2 迁移工具安装

Velero 是开源的 Kubernetes 集群备份、迁移工具，集成了 Restic 工具对 PV 数据的备份能力，可以通过 Velero 工具将原集群中的 K8s 资源对象（如 Deployment、Job、Service、ConfigMap 等）和 Pod 挂载的持久卷数据保存备份上传至对象存储。在发生灾难或需要迁移时，目标集群可使用 Velero 从对象存储中拉取对应的备份，按需进行集群资源的还原。

根据迁移方案所述，在迁移开始前需准备临时的对象存储用于存放资源的备份文件，Velero 支持使用 OBS 或者 **MinIO** 对象存储。对象存储需要准备足够的存储空间用于存放备份文件，请根据您的集群规模和数据量自行估算存储空间。建议您使用 OBS 进行备份存储，可直接参考安装 Velero 进行 Velero 的部署。

源自建集群和 CCE 中目标集群都需要安装 Velero，请按照本章节内容安装。

## 前提条件

- 原始自建集群 Kubernetes 版本需 1.10 及以上，且集群可正常使用 DNS 与互联网服务。
- 若您使用 OBS 存放备份文件，需已有 OBS 操作权限用户的 AK/SK。
- 若您使用 MinIO 存放备份文件，则安装 MinIO 的服务器需要绑定 EIP 并在安全组中开放 MinIO 的 API 端口和 Console 端口。
- 已创建迁移的目标 CCE 集群。
- 原集群和目标集群中需要至少各拥有一个空闲节点，节点规格建议为 4U8G 及以上。

## 安装 MinIO

MinIO 是一个兼容 S3 接口协议的高性能对象存储开源工具。若使用 MinIO 进行存放集群迁移的备份文件，您需要一台临时服务器用于部署 MinIO 并对外提供服务。若您使用 OBS 存放备份文件，请忽略此步骤，前往安装 Velero。

MinIO 的安装位置选择有如下几种：

- 集群外临时 ECS

  将 MinIO 服务端安装在集群外，能够保障集群发生灾难性故障时，备份文件不会受到影响。
- 集群内的空闲节点

  您可以远程登录节点安装 MinIO 服务端，也可以选择容器化安装 MinIO，请参考 Velero 官方文档 https://velero.io/docs/v1.7/contributions/minio/#set-up-server。

**须知**

**如使用容器化安装 MinIO：**

- **Velero 官方文档提供的 YAML 文件中存储类型为 empty dir，建议将其修改为 HostPath 或 Local 类型，否则容器重启后将永久丢失备份文件。**
- **您需将 MinIO 服务对外提供访问，否则将无法在集群外下载备份文件，可选择将 Service 修改为 NodePort 类型或使用其他类型的公网访问服务。**

无论使用何种方法进行部署，安装 MinIO 的服务器需要有**足够的存储空间**，且均需要**绑定 EIP** 并**在安全组中开放 MinIO 的服务端口**，否则将无法上传（下载）备份文件。

本示例选择在一台集群外的临时 ECS 上安装 MinIO，步骤如下。

步骤 1 下载 MinIO 对象存储。

```
mkdir /opt/minio
mkdir /opt/miniodata
cd /opt/minio
wget https://dl.minio.io/server/minio/release/linux-amd64/minio
chmod +x minio
```

步骤 2 设置 MinIO 的用户名及密码。

此方法设置的用户名及密码为临时环境变量，在服务重启后需要重新设定，否则会使用默认 root 凭据 minioadmin:minioadmin 来创建服务。

```
export MINIO_ROOT_USER=minio
export MINIO_ROOT_PASSWORD=minio123
```

步骤 3 创建服务，其中/opt/miniodata/为 MinIO 存储数据的本地磁盘路径。

MinIO 的 API 端口默认为 9000，console 端口默认为随机生成，您可使用--console-address 参数指定 console 访问端口。

```
./minio server /opt/miniodata/ --console-address ":30840" &
```

**说明**

安装 MinIO 工具的服务器需开放防火墙、安全组中对应的 API 和 console 端口，否则将无法访问对象桶。

步骤 4 浏览器访问 http://{minio 所在节点的 eip}:30840，可进入 MinIO console 界面。

**----结束**

## 安装 Velero

首先前往 OBS 控制台或 MinIO console 界面，创建存放备份文件的桶并命名为 **velero**。此处桶名称可自定义，但安装 Velero 时必须指定此桶名称，否则将无法访问导致备份失败，参见步骤 4。

**须知**

- **原集群和目标集群**中均需要安装部署 Velero 实例，安装步骤一致，分别用于备份和恢复。
- CCE 集群的 Master 节点不对外提供远程登录端口，您可通过 kubectl 操作集群完成 Velero 安装。
- 如果备份资源量较大，请调整 Velero 及 Restic 工具的 cpu 和内存资源（建议调整至 1U1G 及以上），请参考备份工具资源分配不足。
- 用于存放备份文件的对象存储桶**需要是空桶**。

从 Velero 官方发布路径 https://github.com/vmware-tanzu/velero/releases 下载最新的稳定版二进制文件，本文以 Velero 1.7.0 版本为例。原集群和目标集群中的安装过程一致，请参考如下步骤。

步骤 1 下载 Velero 1.7.0 版本的二进制文件。

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.7.0/velero-v1.7.0-linux-amd64.tar.gz
```

步骤 2 安装 Velero 客户端。

```
tar -xvf velero-v1.7.0-linux-amd64.tar.gz
cp ./velero-v1.7.0-linux-amd64/velero /usr/local/bin
```

**步骤 3** 创建备份对象存储访问密钥文件 credentials-velero。

```
vim credentials-velero
```

文件内容如下，其中的 AK/SK 请根据实际情况进行替换。如使用 MinIO，此处 AK/SK 则为步骤 2 中所创建的用户名及密码。

```
[default]
aws_access_key_id = {AK}
aws_secret_access_key = {SK}
```

**步骤 4** 部署 Velero 服务端。注意其中--bucket 参数需要修改为已创建的对象存储桶名称，本例中为 **velero**。关于更多自定义安装参数，请参考自定义安装 Velero。

```
velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:v1.2.1 \
  --bucket velero \
  --secret-file ./credentials-velero \
  --use-restic \
  --use-volume-snapshots=false \
  --backup-location-config region=cn-
jssz1,s3ForcePathStyle="true",s3Url=http://obs.cn-jssz1.ctyun.cn
```

表3-6 Velero 安装参数说明

| 安装参数 | 参数说明 |
| --- | --- |
| --provider | 声明使用“aws”提供的插件类型。 |
| --plugins | 使用 AWS S3 兼容的 API 组件，本文使用的 OBS 和 MinIO 对象存储均支持该 S3 协议。 |
| --bucket | 用于存放备份文件的对象存储桶名称，需提前创建。 |
| --secret-file | 访问对象存储的密钥文件，即步骤 3 中创建的“credentials-velero”文件。 |
| --use-restic | 使用 Restic 工具支持 PV 数据备份，建议开启，否则将无法备份存储卷资源。 |
| --use-volume-snapshots | 是否创建 VolumeSnapshotLocation 对象进行 PV 快照，需要提供快照程序支持。该值设为 false。 |
| --backup-location-config | 对象存储桶相关配置，包括 region、s3ForcePathStyle、s3Url 等。 |
| region | 对象存储桶所在区域。<br>• OBS：请根据实际区域填写，如“cn-jssz1”。<br>• MinIO：参数值为 minio。 |
| s3ForcePathStyle | 参数值为“true”，表示使用 S3 文件路径格式。 |

| 安装参数 | 参数说明 |
| --- | --- |
| s3Url | 对象存储桶的 API 访问地址。<br>• OBS：该参数值需根据对象存储桶地域决定，参数值为“http://obs.{region}.ctyun.cn”。例如区域为苏州（cn-jssz1），则参数值为“http://obs.cn-jssz1.ctyun.cn”。<br>• MinIO：该参数值需根据 MinIO 安装节点的 IP 及暴露端口确定，参数值为“http://{minio 所在节点的 eip}:9000”。<br>说明<br>• s3Url 中的访问端口需填写 MinIO 的 API 端口，而非 console 端口。MinIO API 端口默认为 9000。<br>• 访问集群外安装的 MinIO 时，需填写其公网 IP 地址。 |

步骤 5 Velero 实例将默认创建一个名为 velero 的 namespace，执行以下命令可查看 pod 状态。

```
$ kubectl get pod -n velero
NAME                READY   STATUS    RESTARTS   AGE
restic-rn29c         1/1    Running   0          16s
velero-c9ddd56-tkzpk   1/1    Running   0          16s
```

**说明**

为防止在实际生产环境中备份时出现内存不足的情况，建议您参照备份工具资源分配不足，修改 Restic 和 Velero 分配的 CPU 和内存大小。

步骤 6 查看 Velero 工具与对象存储的对接情况，状态需要为 available。

```
$ velero backup-location get
NAME      PROVIDER   BUCKET/PREFIX   PHASE       LAST VALIDATED                ACCESS
MODE   DEFAULT
default   aws       velero         Available   2021-10-22 15:21:12 +0800 CST
ReadWrite    true
```

**----结束**

## 3.2.3.3 集群内资源迁移（Velero）

### 操作场景

本文使用 Wordpress 应用为例，将自建 Kubernetes 集群中应用整体迁移到 CCE 集群。Wordpress 应用包含 Wordpress 和 MySQL 两个组件，均为容器化实例，分别绑定了两个 Local 类型的本地存储卷，并通过 NodePort 服务对外提供访问。

迁移前通过浏览器访问 Wordpress 站点，创建站点名称为“Migrate to CCE”，并发布一篇文章用于验证迁移后 PV 数据的完整性。Wordpress 中发布的文章会被存储在 MySQL 数据库的“wp_posts”表中，若迁移成功，数据库中的内容也将会被全量搬迁至新集群，可依此进行 PV 数据迁移校验。

## 前提条件

- 请在迁移前提前清理原集群中异常的 Pod 资源。当 Pod 状态异常但是又挂载了 PVC 的资源时，在集群迁移后，PVC 状态会处于 pending 状态。
- 请确保 CCE 侧集群中没有与被迁移集群侧相同的资源，因为 Velero 工具在检测到相同资源时，默认不进行恢复。
- 为确保集群迁移后容器镜像资源可以正常拉取，请将镜像资源迁移至容器镜像服务（SWR）。
- CCE **不支持 ReadWriteMany** 的云硬盘存储，在原集群中存在该类型资源时，需要先修改为 ReadWriteOnce。
- Velero 集成 Restic 工具对存储卷进行备份还原，当前**不支持 HostPath 类型**的存储卷，详情请参见 Restic 限制。若您需备份该类型的存储卷，请参考无法备份 HostPath 类型存储卷将 HostPath 类型替换为 Local 类型。当备份任务中存在 HostPath 类型的存储，该类型存储卷将会被自动跳过并产生 Warning 信息，并不会导致备份失败。

## 原集群应用备份

**步骤 1** （可选）如果需要对 Pod 中指定的存储卷数据进行备份，需对 Pod 添加 annotation，标记模板如下：

```
kubectl -n <namespace> annotate <pod/pod_name> backup.velero.io/backup-volumes=<volume_name_1>,<volume_name_2>,...
```

- <namespace>：Pod 所在的 namespace。
- <pod_name>：Pod 名称。
- <volume_name>：Pod 挂载的持久卷名称。可通过 describe 语句查询 Pod 信息，Volume 字段下即为该 Pod 挂载的所有持久卷名称。

对 Wordpress 和 MySQL 的 Pod 添加 annotation，pod 名称分别为 wordpress-758fbf6fc7-s7fsr 和 mysql-5ffdfbc498-c45lh。由于 Pod 在默认命名空间 default 下，-n <NAMESPACE>参数可省略：

```
kubectl annotate pod/wordpress-758fbf6fc7-s7fsr backup.velero.io/backup-volumes=wp-storage
kubectl annotate pod/mysql-5ffdfbc498-c45lh backup.velero.io/backup-volumes=mysql-storage
```

**步骤 2** 对应用进行备份。备份时可以根据参数指定资源，若不添加任何参数，则默认对整个集群资源进行备份，详细参数请参考 Resource filtering。

- **--default-volumes-to-restic：**表示使用 Restic 工具对 Pod 挂载的所有存储卷进行备份，**不支持 HostPath 类型**的存储卷。如不指定该参数，将默认对步骤 1 中 annotation 指定的存储卷进行备份。此参数仅在安装 Velero 时指定“--use-restic”后可用。

  ```
  velero backup create <backup-name> --default-volumes-to-restic
  ```

- **--include-namespaces：**用于指定 namespace 下的资源进行备份。

  ```
  velero backup create <backup-name> --include-namespaces <namespace>
  ```

- **--include-resources：**用于指定资源进行备份。

  ```
  velero backup create <backup-name> --include-resources deployments
  ```

- **--selector**：用于指定与 selector 相匹配的资源备份。

  ```
  velero backup create <backup-name> --selector <key>=<value>
  ```

本文指定 default 命名空间下的资源进行备份，wordpress-backup 为备份名称，进行应用恢复时也需指定相同的备份名称。示例如下：

```
velero backup create wordpress-backup --include-namespaces default --default-volumes-to-restic
```

回显如下，表示成功创建备份任务：

```
Backup request "wordpress-backup" submitted successfully. Run `velero backup describe wordpress-backup` or `velero backup logs wordpress-backup` for more details.
```

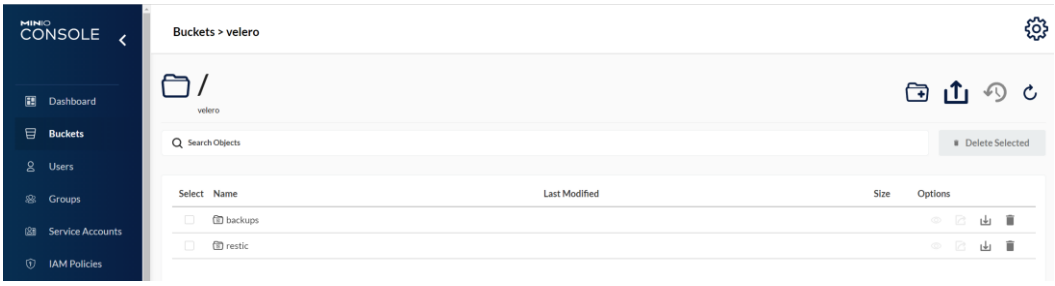
步骤 3 查看备份情况。

```
velero backup get
```

回显如下：

```
NAME              STATUS      ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
wordpress-backup   Completed   0        0          2021-10-14 15:32:07 +0800 CST   29d       default            <none>
```

同时可前往对象桶中查看备份的文件，其中 backups 路径为应用资源备份，restic 路径为 PV 数据备份。




**----结束**

## 目标集群应用恢复

由于自建集群与后端的存储基础设施不同，集群迁移后会遇到 Pod 无法挂载 PV 的问题。因此在进行迁移时需要对新集群中的 StorageClass 进行适配，从而在创建工作负载时可以屏蔽两个集群之间底层存储接口的差异，申请相应类型的存储资源，相关操作请参考 StorageClass 更新适配。

步骤 1 通过 kubectl 连接 CCE 集群，这里选择创建一个原集群中相同名称 StorageClass 来完成适配。

本例中原集群的 StorageClass 名为 local，存储类型为本地磁盘。本地磁盘存储完全依赖节点可用性，数据容灾性能差，节点不可用时将直接影响已有存储数据。因此 CCE 集群中的存储资源选用云硬盘存储卷，后端存储介质使用 SAS 存储。

**说明**

- 包含 PV 数据的应用在 CCE 集群中进行恢复时，定义的 StorageClass 将会根据 PVC 动态创建相应的存储资源（如云硬盘）并挂载，请您知悉。
- 此处集群的存储资源可以根据需求进行更改，并非仅限于云硬盘存储卷。若需要挂载其他类型的存储，如文件存储、对象存储，请参考 StorageClass 更新适配进行适配。

被迁移侧集群 YAML 文件：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

迁移侧集群 YAML 文件示例如下：

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-disk
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
```

步骤 2 使用 Velero 工具创建 restore，指定名称为 wordpress-backup 的备份，将 Wordpress 应用恢复至 CCE 集群。

```
velero restore create --from-backup wordpress-backup
```

可通过 **velero restore get** 语句查看应用恢复情况。

步骤 3 恢复完成后查看应用实例是否正常运行，可能存在其他的更新适配问题，请参考资源更新适配中的步骤排查解决。

**----结束**

## 3.2.3.4 准备对象存储及 Velero

由运维或者开发人员基于 Velero 工具完成 Kubernetes 对象迁移。

### 准备对象存储 MinIO

**MinIO 官网地址：**https://docs.min.io/

准备对象存储，保存其 AK/SK。

**步骤 1** 安装 MinIO。

MinIO is a high performance,distributed,Kubernetes Native Object Storage.

```
# 二进制安装
mkdir /opt/minio
mkdir /opt/miniodata
cd /opt/minio
wget https://dl.minio.io/server/minio/release/linux-amd64/minio
chmod +x minio
export MINIO_ACCESS_KEY=minio
export MINIO_SECRET_KEY=minio123
./minio server /opt/miniodata/ &
浏览器输入： http://{minio 所在节点的 eip}:9000 （注意防火墙、安全组需要放开对应端口）

# kubectl 容器化安装
# 如果需要将 minio 发布为集群外可访问的服务，请修改 00-minio-deployment.yaml 中的服务类型为
NodePort 或 LoadBalancer
kubectl apply -f ./velero-v1.4.0-linux-amd64/examples/minio/00-minio-
deployment.yaml
```

**步骤 2** 创建后面迁移需要使用的桶。

```
打开 minio 的 web 页面
使用 MINIO_ACCESS_KEY/MINIO_SECRET_KEY 登录 minio，本文中为 minio/minio123
单击‘+’上方的“Create bucket”，创建桶，本文中桶名为 velero
```

**----结束**

## 准备 Velero

**Velero 官网地址：**https://velero.io/docs/v1.4/contributions/minio/

Velero is an open source tool to safely backup and restore, perform disaster recovery, and migrate Kubernetes cluster resources and persistent volumes.

在 ACK 和 CCE 的可执行 kubectl 命令的节点上执行如下操作：

**步骤 1** 下载迁移工具 Velero

```
从 https://github.com/heptio/velero/releases 下载最新的稳定版
本文下载的是 velero-v1.4.0-linux-amd64.tar.gz
```

**步骤 2** 安装 Velero 客户端

```
mkdir /opt/ack2cce
cd /opt/ack2cce
tar -xvf velero-v1.4.0-linux-amd64.tar.gz -C /opt/ack2cce
cp /opt/ack2cce/velero-v1.4.0-linux-amd64/velero /usr/local/bin
```

**步骤 3** 安装 Velero 服务端

```
cd /opt/ack2cce
# 准备 minio 认证文件，ak/sk 要正确
vi credentials-velero

[default]
aws_access_key_id = minio
```

```
aws_secret_access_key = minio123

# 安装 velero 服务端，注意 s3Url 要修改为正确的 minio 地址
velero install \
 --provider aws \
 --plugins velero/velero-plugin-for-aws:v1.0.0 \
 --bucket velero \
 --secret-file ./credentials-velero \
 --use-restic \
 --use-volume-snapshots=false \
 --backup-location-config region=minio,s3ForcePathStyle="true",s3Url=http://{minio 所在节点的 eip}:9000
```

**----结束**

## 3.2.3.5 备份原 ACK 集群的 Kubernetes 对象

**步骤 1** 如果需要备份带 PV 数据的 wordpress 应用，请先给对应 pod 加 annotation，不备份 PV 可跳过此步。

```
# kubectl -n YOUR_POD_NAMESPACE annotate pod/YOUR_POD_NAME backup.velero.io/backup-volumes=YOUR_VOLUME_NAME_1,YOUR_VOLUME_NAME_2,...

[root@iZbp1cqobeh1iyyf7qgvvzZ ack2cce]# kubectl get pod -n wordpress
NAME   READY   STATUSRESTARTS   AGE
wordpress-67796d86b5-f9bfm 1/1 Running   1  39m
wordpress-mysql-645b796d8d-6k8wh   1/1 Running   0  38m

[root@iZbp1cqobeh1iyyf7qgvvzZ ack2cce]# kubectl -n wordpress annotate pod/wordpress-67796d86b5-f9bfm backup.velero.io/backup-volumes=wordpress-pvc
pod/wordpress-67796d86b5-f9bfm annotated
[root@iZbp1cqobeh1iyyf7qgvvzZ ack2cce]# kubectl -n wordpress annotate pod/wordpress-mysql-645b796d8d-6k8wh backup.velero.io/backup-volumes=wordpress-mysql-pvc
pod/wordpress-mysql-645b796d8d-6k8wh annotated
```

**步骤 2** 执行备份任务

```
[root@iZbp1cqobeh1iyyf7qgvvzZ ack2cce]# velero backup create  wordpress-ack-backup --include-namespaces wordpress
Backup request "wordpress-ack-backup" submitted successfully.
Run `velero backup describe wordpress-ack-backup` or `velero backup logs wordpress-ack-backup` for more details.
```

**步骤 3** 查看备份任务是否成功

```
[root@iZbp1cqobeh1iyyf7qgvvzZ ack2cce]# velero backup get
NAME   STATUS   CREATED EXPIRES   STORAGE LOCATION   SELECTOR
wordpress-ack-backup   InProgress   2020-07-07 20:31:19 +0800 CST   29d
default<none>
[root@iZbp1cqobeh1iyyf7qgvvzZ ack2cce]# velero backup get
NAME   STATUS  CREATED EXPIRES   STORAGE LOCATION   SELECTOR
wordpress-ack-backup   Completed   2020-07-07 20:31:19 +0800 CST   29d
default<none>
```

**----结束**

## 3.2.3.6 在 CCE 集群恢复 Kubernetes 对象

### 创建 StorageClass

本示例 WordPress 应用使用阿里云 SSD 类型持久化数据卷，相应的在 CCE 中需要适配成天翼云 SSD。

本示例中使用的 StorageClass 是 alicloud-disk-ssd，需要创建一个同名的 SC，但后端存储介质使用天翼云 SSD 存储。此处请根据自己的应用来适配。

```
[root@ccenode-roprr hujun]# cat cce-sc-csidisk-ack.yaml
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: alicloud-disk-ssd
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-disk
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

[root@ccenode-roprr hujun]# kubectl create -f cce-sc-csidisk-ack.yaml
```

### 恢复应用

```
[root@ccenode-roprr hujun]# velero restore create --from-backup  wordpress-ack-
backup
Restore request "wordpress-ack-backup-20200707212519" submitted successfully.
Run `velero restore describe wordpress-ack-backup-20200707212519` or `velero
restore logs wordpress-ack-backup-20200707212519` for more details

[root@ccenode-roprr hujun]# velero restore get
NAME  BACKUP STATUS  WARNINGS  ERRORS  CREATED SELECTOR
wordpress-ack-backup-20200708112940  wordpress-ack-backup  Completed  0  02020-07-
08 11:29:42 +0800 CST  <none>
```

此时查看 wordpress 应用运行情况，可能会有镜像拉取失败，服务访问不通等问题，需要进行适配处理。

## 3.2.3.7 资源更新适配

### 镜像更新适配

由于本例使用的 Wordpress 和 MySQL 镜像均可从 SWR 正常拉取，因此不会出现镜像拉取失败（ErrImagePull）问题。如迁移应用为私有镜像，请执行以下步骤完成镜像更新适配。

**步骤 1** 将镜像资源迁移至容器镜像服务（SWR），具体步骤请参考客户端上传镜像。

步骤 2 登录 SWR 控制台查看获取迁移后的镜像地址。

镜像地址格式如下：

```
'registry.{区域}.ctyun.cn/{所属组织名称}/{镜像名称}:{版本名称}'
```

步骤 3 使用如下命令对工作负载进行修改，并将 YAML 文件中的 image 字段替换成迁移后的镜像地址。

```
kubectl edit deploy wordpress
```

步骤 4 查看应用实例运行情况。

**----结束**

## 访问服务更新适配

集群迁移后，原有集群的访问服务可能无法生效，可执行如下步骤更新服务。如原集群中设置了 Ingress 资源，迁移后需重新对接 ELB。

步骤 1 通过 kubectl 连接集群。

步骤 2 编辑对应 Service 的 YAML 文件，修改服务类型及端口。

```
kubectl edit svc wordpress
```

LoadBanlancer 资源进行更新时，需要重新对接 ELB，添加如下 Annotation：

```
annotations:
  kubernetes.io/elb.class: union   #共享型 ELB
  kubernetes.io/elb.id: 9d06a39d-xxxx-xxxx-xxxx-c204397498a3    #ELB 的 ID，可前往 ELB 控制台查询
  kubernetes.io/elb.subnet-id: f86ba71c-xxxx-xxxx-xxxx-39c8a7d4bb36    #集群所在子网的 ID
  kubernetes.io/session-affinity-mode: SOURCE_IP    #开启会话保持，基于源 IP 地址
```

步骤 3 浏览器访问查看服务是否可用。

**----结束**

## StorageClass 更新适配

由于集群的存储基础设施不同，迁移后的集群将无法正常挂载存储卷，您可执行以下方法的任意一种来完成存储卷的更新适配。

**须知**

两种 StorageClass 的适配方法均需在**目标集群**中于**恢复应用前**完成，否则可能出现 PV 数据资源无法恢复的情况，此时在完成 StorageClass 适配后使用 Velero 重新恢复应用即可，请参考目标集群应用恢复。

**方式一：创建 ConfigMap 映射**

**步骤 1** 在 CCE 集群中创建如下所示的 ConfigMap，将原集群使用的 StorageClass 映射到 CCE 集群默认的 StorageClass。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: change-storageclass-plugin-config
  namespace: velero
  labels:
    app.kubernetes.io/name: velero
    velero.io/plugin-config: "true"
    velero.io/change-storage-class: RestoreItemAction
data:
  {原集群 StorageClass name01}: {目标集群 StorageClass name01}
  {原集群 StorageClass name02}: {目标集群 StorageClass name02}
```

**步骤 2** 执行以下命令，应用上述的 ConfigMap 配置。

```
$ kubectl create -f change-storage-class.yaml
configmap/change-storageclass-plugin-config created
```

**----结束**

**方式二：创建同名 StorageClass**

**步骤 1** 查询 CCE 支持的默认 StorageClass。

```
kubectl get sc
```

回显如下：

```
NAME              PROVISIONER                  RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
csi-disk          everest-csi-provisioner        Delete         Immediate
true                3d23h
csi-disk-topology   everest-csi-provisioner        Delete
WaitForFirstConsumer   true                3d23h
csi-nas           everest-csi-provisioner        Delete         Immediate
true                3d23h
csi-obs           everest-csi-provisioner        Delete         Immediate
false               3d23h
csi-sfsturbo       everest-csi-provisioner        Delete         Immediate
true                3d23h
```

表3-7 StorageClass

| StorageClass 名称 | 对应的存储资源 |
| --- | --- |
| csi-disk | 云硬盘 |
| csi-disk-topology | 延迟绑定的云硬盘 |
| csi-nas | 文件存储 |
| csi-obs | 对象存储 |
| csi-sfsturbo | 极速文件存储 |

**步骤 2** 通过如下命令将所需的 StorageClass 详细信息输出为 YAML 格式。

```
kubectl get sc <storageclass-name> -o=yaml
```

**步骤 3** 复制 YAML 文件并创建一个新的 StorageClass。

编辑 StorageClass 名称，将其命名为原有集群中使用的名称，用于调用云上基础存储资源。

以 csi-obs 的 YAML 文件为例。请删除 metadata 字段下如斜体部分所示的不必要信息，并修改加粗部分，其余参数不建议修改。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  creationTimestamp: "2021-10-18T06:41:36Z"
  name: <your_storageclass_name>     #命名为原有集群中使用的 StorageClass 名称
  resourceVersion: "747"
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-obs
  uid: 4dbbe557-ddd1-4ce8-bb7b-7fa15459aac7
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: Delete
volumeBindingMode: Immediate
```

**说明**

- 极速文件存储无法通过 StorageClass 直接创建，您需要前往 SFS Turbo 控制台创建相同 VPC 子网下且安全组开放入方向端口(111、445、2049、2051、2052、20048)的极速文件存储。
- CCE 不支持 ReadWriteMany 的云硬盘存储，在原集群中存在该类型资源时，需要先修改为 ReadWriteOnce。

**步骤 4** 参考目标集群应用恢复进行集群应用恢复，检查 PVC 是否创建成功。

```
kubectl get pvc
```

回显如下，其中 VOLUME 列为通过 StorageClass 自动创建的 PV 名称。

```
NAME   STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
pvc    Bound    pvc-4c8e655a-1dbc-4897-ae6c-446b502f5e77   5Gi        RWX
local          13s
```

**----结束**

## 数据库更新适配

本例中数据库为本地 MySQL 数据库，迁移后无需重新配置。若您通过**数据复制服务 DRS** 将本地数据库迁移至云数据库 RDS，则在迁移后需重新配置数据库的访问，请您根据实际情况进行配置。

**说明**

- 若云数据库 RDS 实例与 CCE 集群处于同一 VPC 下，则可通过内网地址访问，否则只能通过绑定 EIP 的方式进行公网访问。建议使用内网访问方式，安全性高，并且可实现 RDS 的较好性能。
- 请确认 RDS 所在安全组入方向规则已对集群放通，否则将连接失败。

**步骤 1** 登录 RDS 控制台，在该实例的“基本信息”页面获取其“内网地址”及端口。

**步骤 2** 使用如下命令对 Wordpress 工作负载进行修改。

```
kubectl edit deploy wordpress
```

设置 env 字段下的环境变量：

- WORDPRESS_DB_HOST：数据库的访问地址和端口，即上一步中获取的内网地址及端口。
- WORDPRESS_DB_USER：访问数据库的用户名。
- WORDPRESS_DB_PASSWORD：访问数据库的密码。
- WORDPRESS_DB_NAME：需要连接的数据库名。

**步骤 3** 检查 RDS 数据库是否正常连接。

**----结束**

## 3.2.3.8 调试启动应用

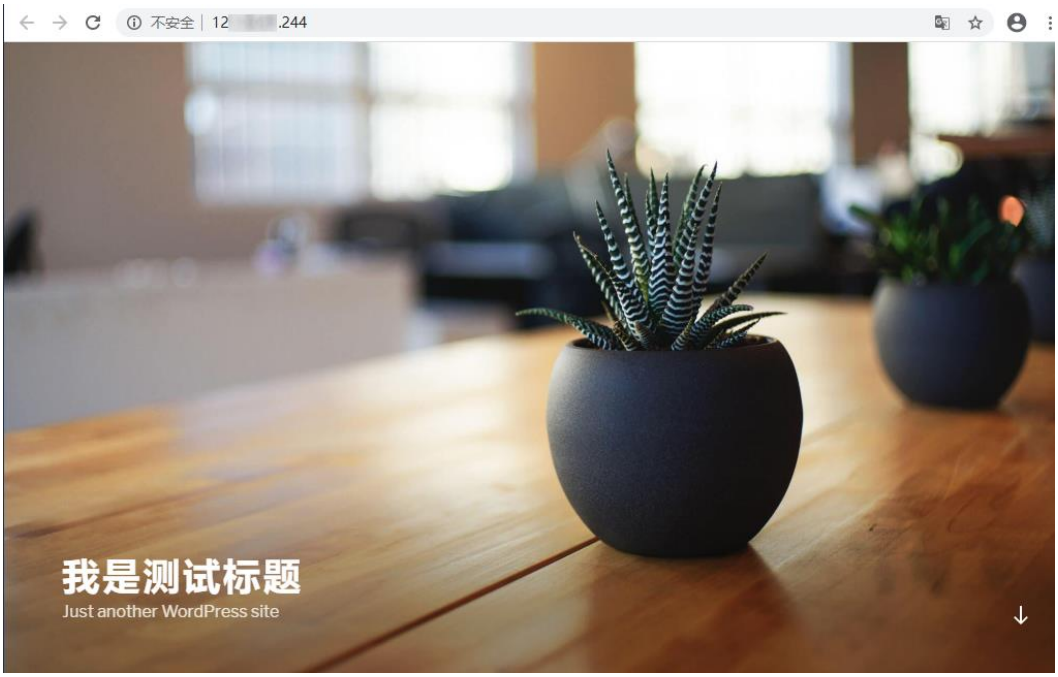
调试并访问应用，检查数据。

**步骤 1** 登录 CCE 控制台，在左侧导航栏中选择“资源管理 > 网络管理”，单击 wordpress 服务后方的弹性公网访问地址。

**图3-2 获取访问地址**


**步骤 2** 访问正常，迁移成功。

图3-3 wordpress 欢迎页




----结束

## 3.2.3.9 其它

### 业务验证

在不影响线上流量的情况下，由测试人员完成新集群的功能验证。

- 配置测试域名
- 测试业务功能
- 测试监控日志告警等运维功能

### 业务流量切换

由运维人员做 DNS 切换，将流量引到新集群。

- DNS 流量切换：调整 DNS 配置实现流量切换
- 客户端流量切换：升级客户端代码或更新配置实现流量切换

### 原 ACK 集群下线

由运维人员确认新集群业务正常后，下线原集群并清理备份文件。

- 确认新集群业务正常
- 下线原集群
- 清理备份文件

# 4 DevOps

## 4.1 Jenkins 安装部署及对接 SWR 和 CCE 集群

### 4.1.1 方案概述

#### Jenkins 是什么

Jenkins 是一个开源的、提供友好操作界面的持续集成（CI）工具，起源于 Hudson，主要用于持续、自动的构建/测试软件项目、监控外部任务的运行。

Jenkins 用 Java 语言编写，可在 Tomcat 等流行的 servlet 容器中运行，也可独立运行。通常与版本管理工具（SCM）、构建工具结合使用。Jenkins 可以很好的支持各种语言的项目构建，也完全兼容 Maven、Ant、Gradle 等多种第三方构建工具，同时跟 SVN、GIT 等常用的版本控制工具无缝集成，也支持直接对接 GitHub 等源代码托管网站。

#### 约束与限制

- 该实践方案仅支持在 CCE 集群下部署，不适用专属云场景。
- Jenkins 系统的维护由开发者自行负责，使用过程中 CCE 服务不对 Jenkins 系统提供额外维护与支持。

#### 方案架构

Jenkins 部署分为以下两种模式：

- 一种是直接使用单 Master 安装 Jenkins，直接进行任务管理和业务构建发布，但可能存在一定的生产安全风险。
- 一种是 Master 加 Agent 模式。Master 节点主要是处理调度构建作业，把构建分发到 Agent 实际执行，监视 Agent 的状态。业务构建发布的工作交给 Agent 进行，即执行 Master 分配的任务，并返回任务的进度和结果。

Jenkins 的 Master 和 Agent 均可安装在虚拟机或容器中，且组合形式可多样。

表4-1 Jenkins 部署模式

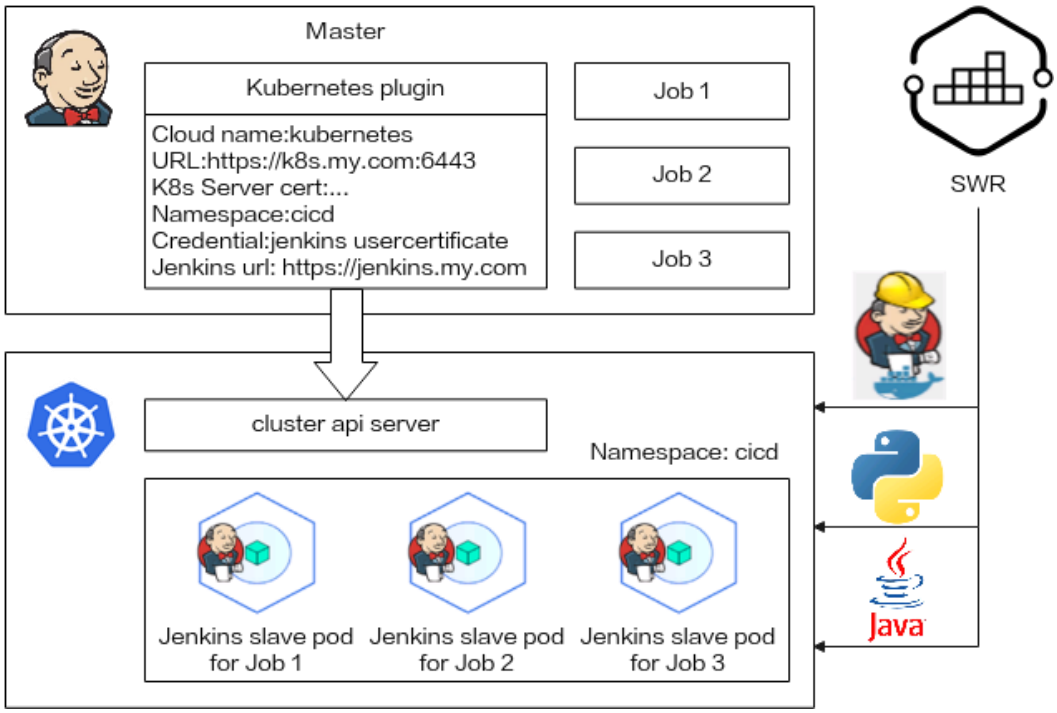
| 部署模式 | Master | Agent | 优缺点分析 |
| --- | --- | --- | --- |

| 部署模式 | Master | Agent | 优缺点分析 |
| --- | --- | --- | --- |
| 单 Master | 虚拟机 | - | • 优点：本地化构建，操作简单。<br>• 缺点：任务管理和执行都在同一台虚拟机上，安全风险较高。 |
| 单 Master | 容器 | - | • 优点：利用 K8s 容器调度机制，拥有一定的自愈能力。<br>• 缺点：任务管理和执行没有分离，安全风险问题仍未解决。 |
| Master 加 Agent | 虚拟机 | 虚拟机 | • 优点：任务管理和执行分离，降低了一定的安全风险。<br>• 缺点：只能固定 Agent，无法进行资源调度，资源利用率低，且环境维护成本高。 |
| | | 容器（K8s 集群） | • 优点：容器化的 Agent 可以选择固定 Agent，也可以通过 K8s 实现动态 Agent，动态 Agent 的方式资源利用率高。并且可以根据调度策略均匀分配任务，后期也比较容易维护。<br>• 缺点：Jenkins 的 Master 存在小概率的宕机风险，恢复成本较高。 |
| Master 加 Agent | 容器（K8s 集群） | 容器（K8s 集群） | • 优点：容器化的 Agent 可以选择固定 Agent，也可以通过 K8s 实现动态 Agent，资源利用率高。且 Master 具有自愈能力，维护成本低。Agent 可以选择和 Master 共集群，也可以分集群。<br>• 缺点：系统复杂程度高，环境搭建较困难。 |

本文采用 Master 加 Agent 模式，Master 和 Agent 均为容器化安装的方案，并使用在 K8s 集群实现动态 Agent。

- Jenkins Master 负责管理任务（Job），为了能够利用 Kubernetes 平台上的资源，需要在 Master 上安装 Kubernetes 的插件。
- Kubernetes 平台负责产生 Pod，用作 Jenkins Agent 执行 Job 任务。当 Jenkins Master 上有 Job 被调度时，Jenkins Master 通过 Kubernetes 插件向 Kubernetes 平台发起请求，请 Kubernetes 根据 Pod 模板产生对应的 Pod 对象，Pod 对象会向 Jenkins Master 发起请求，Master 连接成功后，就可以在 Pod 上面执行 Job 了。

图4-1 K8s 安装 Jenkins 架构




## 操作流程

步骤 1 Jenkins Master 安装部署。

Jenkins Master 使用容器化镜像部署在 CCE 集群中。

步骤 2 Jenkins Agent 配置。

Jenkins 可以在集群中创建固定 Agent，也可以使用 pipeline 与 CCE 的对接，动态提供 Agent Pod。其中动态 Agent 还需要使用 Kubernetes 相关插件配置集群认证信息及用户权限。

步骤 3 使用 Jenkins 构建流水线。

Jenkins 流水线与 SWR 对接，在 Agent 中调用 docker build/login/push 相关的命令，实现自动化的镜像打包、推送。

您也可以通过流水线实现 Kubernetes 资源（deployment/service/ingress/job 等）的部署、升级等能力。

**----结束**

# 4.1.2 资源和成本规划

**须知**

本文提供的资源规划方案仅供参考，资源的实际配置以天翼云管理控制台显示为准。

完成本实践所需的资源如下：

表4-2 资源规划

| 资源 | 资源说明 | 数量 |
| --- | --- | --- |
| 云容器引擎 CCE | 建议选择按需计费。<br>• 集群类型：CCE 集群<br>• 集群版本：v1.23<br>• 集群规模：50 节点<br>• 高可用：是 | 1 |
| 虚拟机节点 | 建议选择按需计费。<br>• 虚拟机节点类型：通用计算增强型<br>• 节点规格：4 核 \| 8GiB<br>• 操作系统：EulerOS 2.9<br>• 系统盘：50GiB \| 通用型 SSD<br>• 数据盘：100GiB \| 通用型 SSD | 1 |
| 云硬盘 EVS | 建议选择按需计费。<br>• 云硬盘规格：100G<br>• 云硬盘类型：通用型 SSD | 1 |
| 负载均衡器 ELB | 建议选择按需计费。<br>• 实例规格：共享型<br>• 公网带宽：按流量计费<br>• 带宽：5 Mbit/s | 1 |

## 4.1.3 实施步骤

### 4.1.3.1 Jenkins Master 安装部署

**说明**

Jenkins 界面中的词条可能因版本不同而存在一些差异，例如中英文不同等，本文中的截图仅供您参考。

## 镜像选择

在 DockerHub 上选择 1 个相对较新的稳定镜像，本次搭建测试用的 Jenkins 使用的镜像为 **jenkinsci/blueocean:2.346.3**，该镜像捆绑了所有 Blue Ocean 插件和功能，不需要再单独安装 Blue Ocean 插件，详情请参见在 Docker 中下载并运行 Jenkins。

## 准备工作

- 在创建容器工作负载前，您需要购买一个可用集群（集群至少包含 1 个 4 核 8G 的节点，避免资源不足）。
- 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，或已购买负载均衡实例。

## 通过 CCE 安装部署 Jenkins

**步骤 1** 在 CCE 控制台，单击左侧栏目树中的“工作负载 > 无状态负载”，单击右侧“创建负载”按钮进入无状态工作负载创建页面。

**步骤 2** 填写工作负载基本参数。

- 负载名称：jenkins（可自定义）。
- 命名空间：选择 Jenkins 部署的命名空间，可自行创建。
- 实例数量：1 个。


**步骤 3** 填写容器基本信息参数。

- 镜像名称：**jenkinsci/blueocean**。请根据实际情况进行选择镜像版本，若不设置版本，则默认拉取 latest 版本。
- CPU 配额：本例中 CPU 配额限制为 2 Core
- 内存配额：本例中内存配额限制为 2048 MiB
- 特权容器：如果选择使用单 Master 部署的 Jenkins，必需要开启“特权容器”，使容器获得操作宿主机的权限，否则 Jenkins Master 容器中无法执行 docker 命令。

其他参数默认。

图4-2 容器基本信息参数


步骤 4 在“数据存储”中的“存储卷声明 PVC”页签下，添加持久化存储。

在弹出的窗口中选择 1 个云存储卷，并在挂载路径下输入**/var/jenkins_home**，将云存储挂载到 Jenkins 容器的**/var/jenkins_home** 目录，供 Jenkins 保留持久化数据。

**说明**

云存储类型可选择“云硬盘 EVS”或“文件存储 SFS”，若没有云存储可单击“创建存储卷声明”创建。

如选择“云硬盘 EVS”类型，要求 EVS 的可用区与节点可用区一致。

图4-3 添加云存储


步骤 5 给 Jenkins 容器添加权限，让 Jenkins 容器中可以执行相关命令。

1. 确认 3 中已开启“特权容器”开关。
2. 在“数据存储”中的“本地存储”页签下添加本地存储，将主机路径挂载到容器对应路径。

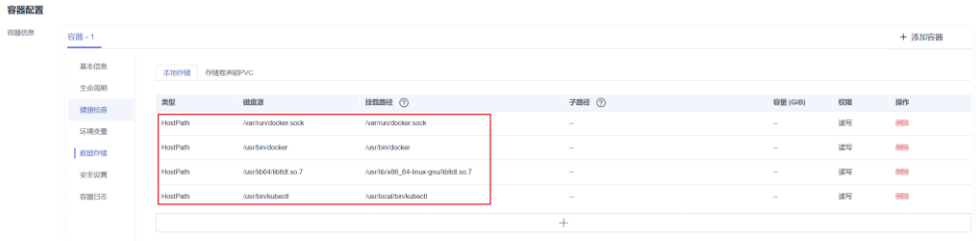
表4-3 挂载路径

| 存储类型 | 主机路径（HostPath） | 挂载路径 |
| --- | --- | --- |
| 主机路径（HostPath） | • 节点使用 docker 容器引擎：**/var/run/docker.sock**<br>• 节点使用 containerd 容器引擎：**/var/run/containerd** | • 点使用 docker 容器引擎：**/var/run/docker.sock**<br>• 节点使用 containerd 容器引擎：**/var/run/containerd** |
| 主机路径（HostPath） | • 节点使用 docker 容器引擎：**/usr/bin/docker** | • 节点使用 docker 容器引擎：**/usr/bin/docker** |

| 存储类型 | 主机路径（HostPath） | 挂载路径 |
| --- | --- | --- |
| | • 节点使用 containerd 容器引擎：**/usr/local/bin/ctr** | • 节点使用 containerd 容器引擎：**/usr/local/bin/ctr** |
| 主机路径（HostPath） | **/usr/lib64/libltdl.so.7** | **/usr/lib/x86_64-linux-gnu/libltdl.so.7** |
| 主机路径（HostPath） | **/usr/bin/kubectl** | **/usr/local/bin/kubectl** |

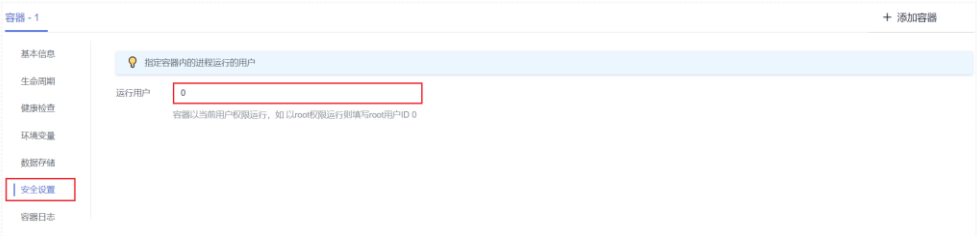
挂载完成后，如下图所示。

图4-4 挂载主机到容器对应路径



3. 在“安全设置”中配置“运行用户”为：0（即 root 用户）。

图4-5 配置运行用户



步骤 6 在“服务配置”中，设置访问方式。

Jenkins 容器镜像有两个端口：8080 和 50000，需要分别配置。其中 8080 端口供 Web 登录使用，50000 端口供 Master 和 Agent 连接使用。

本例中创建了两个 Service：

- 负载均衡（LoadBalancer）：仅用于提供 Web 的外部访问，使用 8080 端口。您也可以选择使用“节点访问（NodePort）”类型的 Service 提供外部访问。

  Service 名称：jenkins（可自定义），容器端口：8080，访问端口：8080，其他默认。

- 集群内访问（ClusterIP）：用于 Agent 连接 Master。Jenkins 要求 jenkins-web 的地址要和 jenkins-agent 的地址一致，因此包含 Web 访问的 8080 端口和 Agent 访问的 50000 端口。

  Service 名称：agent（可自定义），容器端口 1：8080，访问端口 1：8080，容器端口 2：50000，访问端口 2：50000，其他默认。

**说明**

本例中，后续步骤创建的 Agent 均与 Master 处于同一集群，因此 Agent 连接使用 ClusterIP 类型的 Service。

如果 Agent 需要跨集群或使用公网连接 Jenkins Master，请自行选择合适的 Service 类型。但需要注意的是，Jenkins 要求 jenkins-web 的地址要和 jenkins-agent 的地址一致，因此 **Agent 连接的地址必须同时开放 8080 和 50000 端口**，而仅用于 Web 访问的地址可以只开放 8080 端口、不开放 50000 端口。

图4-6 添加服务


步骤 7 “高级配置”步骤可以保持默认，直接单击“创建工作负载”，完成工作负载创建。

步骤 8 在创建成功页面单击“返回工作负载列表”，查看工作负载状态，若显示为“运行中”则 jenkins 应用已可以正常访问。

图4-7 查看工作负载状态


----结束

## 登录并初始化 Jenkins

步骤 1 在 CCE 控制台，单击左侧栏目树中的“服务发现”，在“服务”页签下查看 jenkins 的访问方式。

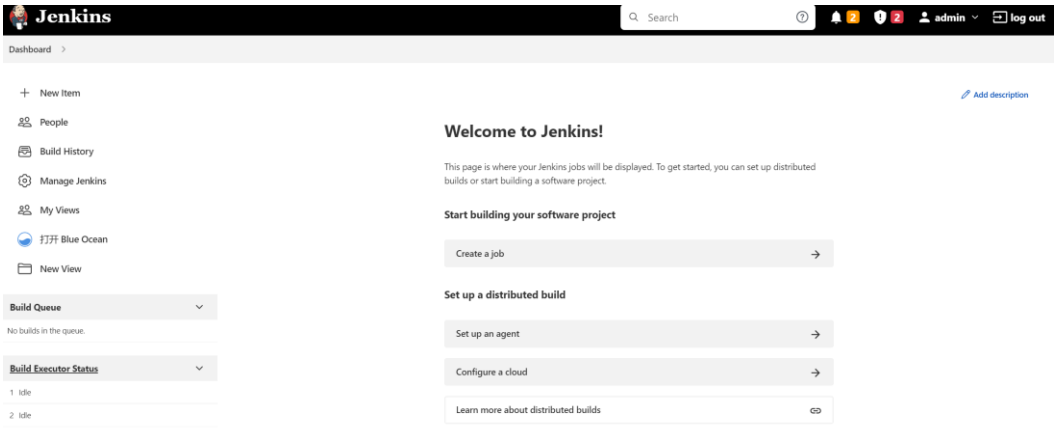
图4-8 访问 8080 端口对应的访问方式


步骤 2 在浏览器中输入负载均衡的“EIP:8080”，即可打开 jenkins 配置页面。

初次访问时界面会提示获取初始管理员密码，该密码可在 jenkins 的 Pod 中获取。在执行下述命令之前您需要先通过 kubectl 连接集群，具体操作请参见通过 kubectl 连接集群。

```
# kubectl get pod -n cicd
NAME                                READY   STATUS    RESTARTS   AGE
jenkins-7c69b6947c-5gvlm            1/1     Running   0          17m
# kubectl exec -it jenkins-7c69b6947c-5gvlm -n cicd -- /bin/sh
# cat /var/jenkins_home/secrets/initialAdminPassword
b10eabe29a9f427c9b54c01a9c3383ae
```

步骤 3 首次登录时选择默认推荐的插件即可，并根据页面提示创建一个管理员。完成初始配置后，即可进入 Jenkins 页面。




**----结束**

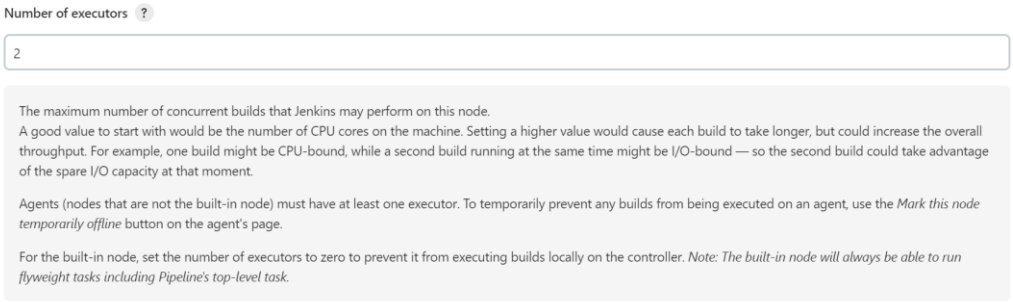
## 修改并发构建数量

步骤 1 在 Jenkins Dashboard 页面，单击左侧“Manage Jenkins”，选择“System Configuration > Manage nodes and clouds”，选择目标节点下拉框里的“Configure”，如下图所示：


**说明**

- Master 和 Agent 节点均可修改并发构建数量，此处以 Master 为例。
- 如果使用 Master+Agent 模式，建议将 Master 的并发构建数设置为 0，即全部使用 Agent 进行构建。如果使用单 Master 模式，则无需修改为 0。

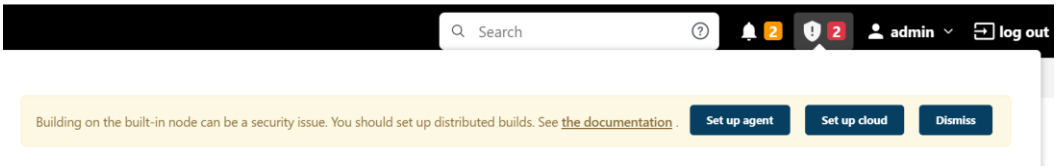
步骤 2 修改执行并发构建的最大数目，示例中修改为 2，您可根据实际需求并结合节点性能进行修改该值。




**----结束**

## 4.1.3.2 Jenkins Agent 配置

安装完 Jenkins 后，可能会出现以下提示，说明 Jenkins 使用 Master 进行本地构建，未配置 Agent。




如果您选择单 Master 安装 Jenkins，执行完毕 4.1.3.1 Jenkins Master 安装部署中的操作后已完成，可直接进行流水线构建，请参见 4.1.3.3 使用 Jenkins 构建流水线。

如果您选择 Master+Agent 模式的 Jenkins，请继续完成 Agent 的配置，您可根据自身需求选择其中一种方案执行：

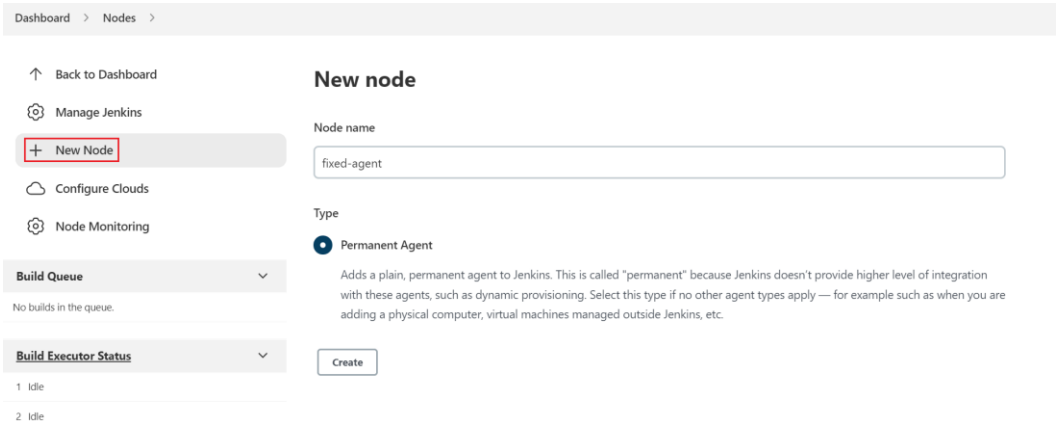
- 固定 Agent：Agent 容器一直运行，任务构建完成后不会销毁，创建完成后将一直占用集群资源，配置过程较简单。
- 动态 Agent：构建任务时动态创建 Agent 容器，并在任务构建完成后销毁容器，可实现资源动态分配，资源利用率高，但是配置过程较为复杂。

本文使用容器化安装 Agent，示例的 Agent 镜像为 **jenkins/inbound-agent:4.13.3-1**。

# Jenkins 添加固定 Agent

**步骤 1** 登录 Jenkins Dashboard，单击左侧“Manage Jenkins”，选择“System Configuration > Manage nodes and clouds”。

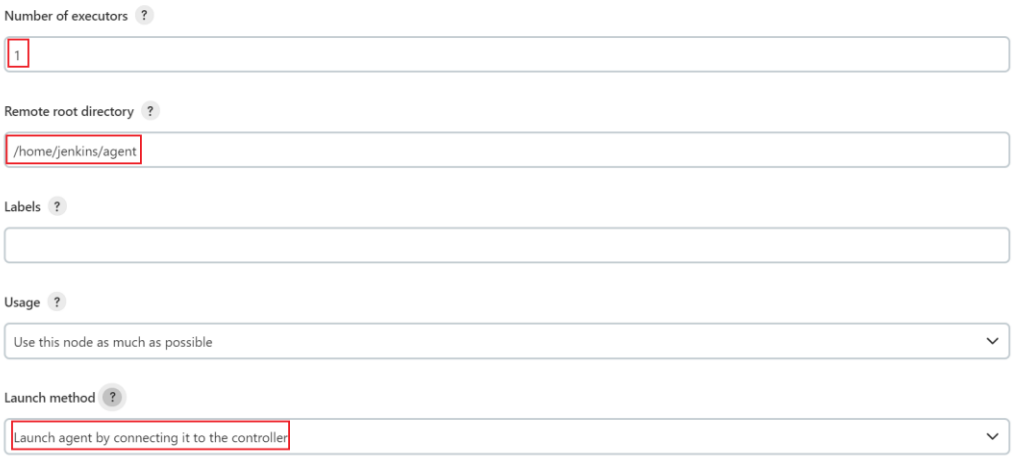
**步骤 2** 单击新页面左侧的“New Node”，输入节点名称为 fixed-agent（该名称可自定义），类型选择固定节点。



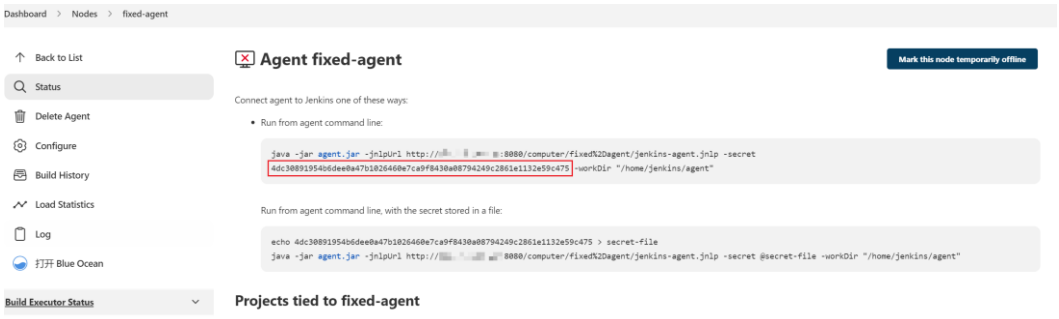

**步骤 3** 配置以下节点信息：

- Number of executors（并发构建的最大数目）：默认为 1，可根据实际需求填写。
- Remote root directory（远程工作目录）：/home/jenkins/agent
- Launch method（启动方式）：选择“Launch agent by connecting it to the controller（通过 Java Web 启动代理）”。

其余参数可保持默认，无需填写，并单击“保存”。




**步骤 4** 在“节点列表”中单击新增的节点名称，可看到 Agent 状态未连接，并提供了节点连接 Jenkins 的方式。该命令适用于虚拟机安装，而本示例为容器化安装，因此仅需复制其中的 secret，如下图所示。




步骤 5 前往 CCE 控制台，单击左侧栏目树中的“工作负载 > 无状态负载”，单击右侧“创建负载”按钮进入无状态工作负载创建页面。

步骤 6 填写工作负载基本参数。

- 负载名称：agent（可自定义）。
- 命名空间：选择 Jenkins 部署的命名空间，可自行创建。
- 实例数量：1 个。


步骤 7 填写容器基本信息参数。

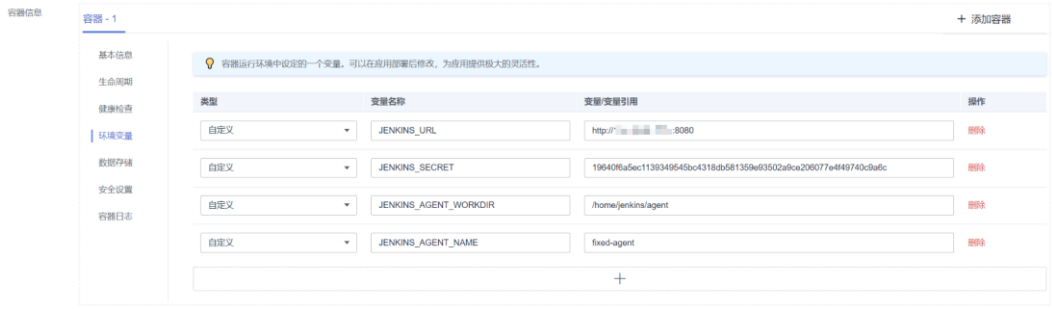
- 镜像名称：**jenkins/inbound-agent:4.13.3-1**。此处镜像版本可能随时间变化发生变动，请根据实际情况进行选择，或拉取 latest 版本。
- CPU 配额：本例中 CPU 配额限制为 2 Core
- 内存配额：本例中内存配额限制为 2048 MiB
- 特权容器：必需要开启“特权容器”，使容器获得操作宿主机的权限，否则容器中无法执行 docker 命令。

其他参数默认。


步骤 8 配置环境变量：

- JENKINS_URL：Jenkins 的访问路径，需要填写步骤 6 中设置的 8080 端口地址（**此处填写的地址必须同时开放 8080 和 50000 端口**），例如“http://10.247.222.254:8080”。

- JENKINS_AGENT_NAME：步骤 2 中设置的 Agent 的名称，本例中为 fixed-agent。
- JENKINS_SECRET：步骤 4 中复制的 secret。
- JENKINS_AGENT_WORKDIR：步骤 3 中配置的远程工作目录，即 /home/jenkins/agent。




**步骤 9** 给 Jenkins 容器添加权限，让 Jenkins 容器中可以执行 docker 命令。

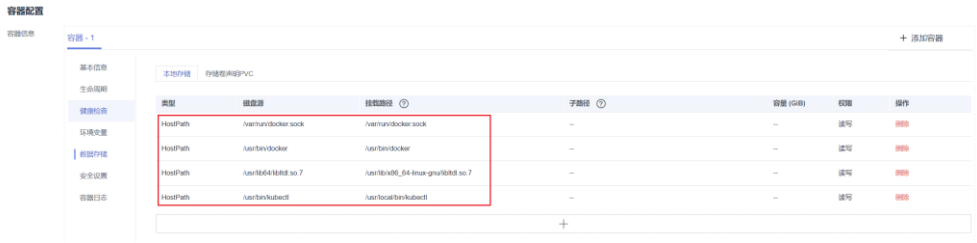
1. 确认已开启“特权容器”开关。
2. 在“数据存储”中的“本地存储”页签下添加本地存储，将主机路径挂载到容器对应路径。

表4-4 挂载路径

| 存储类型 | 主机路径（HostPath） | 挂载路径 |
| --- | --- | --- |
| 主机路径（HostPath） | • 节点使用 docker 容器引擎：**/var/run/docker.sock**<br>• 节点使用 containerd 容器引擎：**/var/run/containerd** | • 点使用 docker 容器引擎：**/var/run/docker.sock**<br>• 节点使用 containerd 容器引擎：**/var/run/containerd** |
| 主机路径（HostPath） | • 节点使用 docker 容器引擎：**/usr/bin/docker**<br>• 节点使用 containerd 容器引擎：**/usr/local/bin/ctr** | • 节点使用 docker 容器引擎：**/usr/bin/docker**<br>• 节点使用 containerd 容器引擎：**/usr/local/bin/ctr** |
| 主机路径（HostPath） | **/usr/lib64/libltdl.so.7** | **/usr/lib/x86_64-linux-gnu/libltdl.so.7** |
| 主机路径（HostPath） | **/usr/bin/kubectl** | **/usr/local/bin/kubectl** |

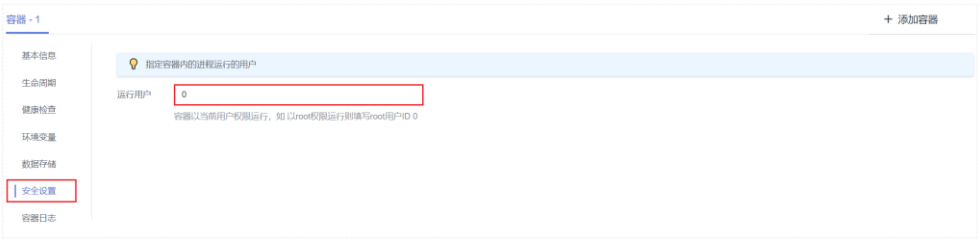
挂载完成后，如下图所示。

图4-9 挂载主机到容器对应路径



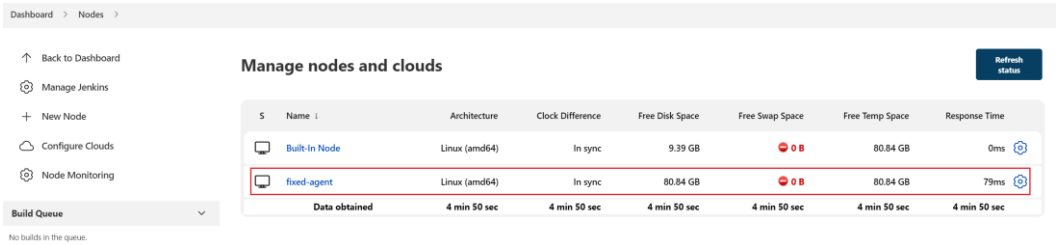
3. 在“安全设置”中配置“运行用户”为：0（即 root 用户）。

图4-10 配置运行用户



步骤 10 “高级配置”步骤可以保持默认，直接单击“创建工作负载”，完成工作负载创建。

步骤 11 前往 Jenkins 页面，刷新节点状态为“已同步”。




**说明**

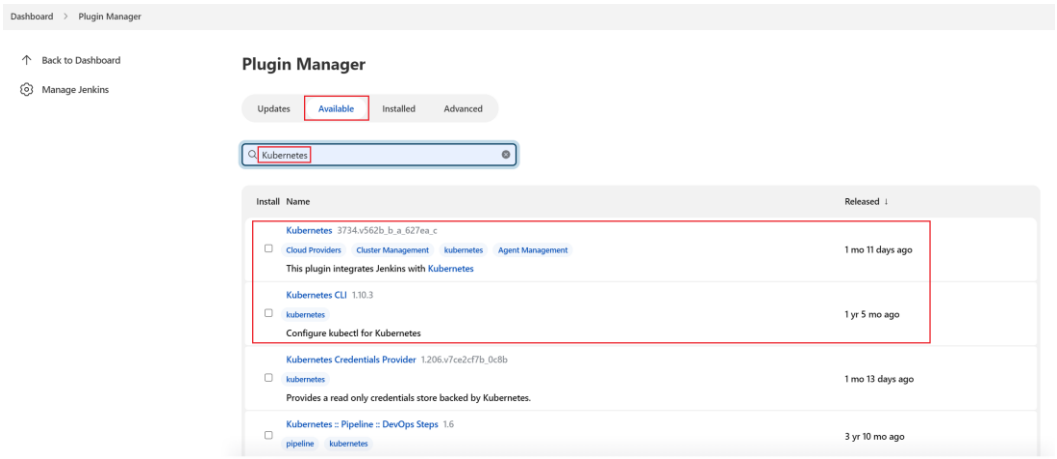
Agent 设置完成后，建议将 Master 的并发构建数设置为 0，即不使用 Master 进行本地构建，全部使用 Agent 进行构建，具体操作步骤请参见修改并发构建数量。

----结束

## Jenkins 设置动态 Agent

步骤 1 **安装插件。**

在 Jenkins Dashboard 页面单击左侧“Manage Jenkins”，选择“System Configuration > Manage Plugins”，在“Available”页签中筛选安装“Kubernetes CLI”和“Kubernetes”插件。




本文安装的插件版本为，插件版本可能随时间变化发生变动，请您自行选择：

- Kubernetes Plugin：3734.v562b_b_a_627ea_c

  用于在 Kubernetes 集群中运行动态 Agent，为每个启动的 Agent 创建一个 Kubernetes Pod，并在每次构建完成后停止 Pod。

- Kubernetes CLI Plugin：1.10.3

  允许为 Job 配置 kubectl，从而与 Kubernetes 集群进行交互。

**说明**

Jenkins 插件由插件维护者提供，可能因为存在安全风险进行版本迭代。

步骤 2 **添加集群凭据到 Jenkins。**

将集群的访问凭据提前添加至 Jenkins，具体操作步骤请参见设置集群访问凭证。

步骤 3 **配置集群基本信息。**

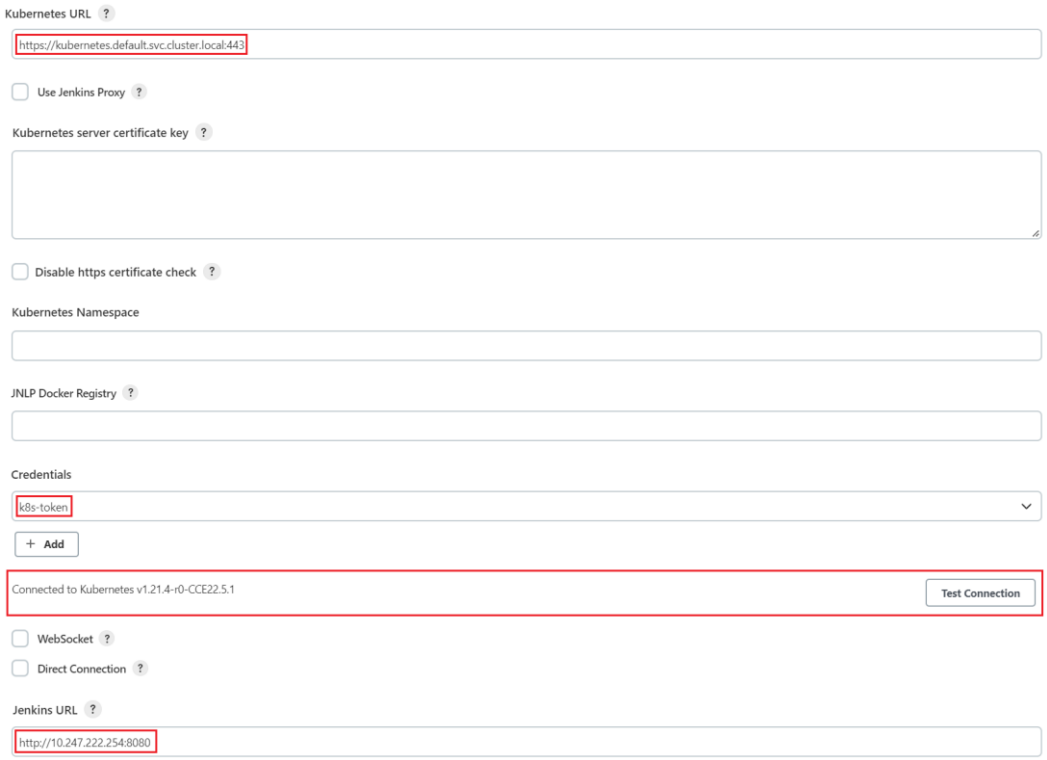
在 Jenkins Dashboard 页面单击左侧“Manage Jenkins”，选择“System Configuration > Manage nodes and clouds”，单击左侧的“Configure Clouds”配置集群，单击“Add a new cloud”，并选择 **Kubernetes**，集群名称可自定义。

步骤 4 **填写 Kubernetes Cloud details**。

填写以下集群配置，其余参数可保持默认，如图 4-11 所示。

- Kubernetes URL：集群 APIserver 地址，可填写“https://kubernetes.default.svc.cluster.local:443”。
- Credentials：选择步骤 2 中添加的集群凭据，可单击“连接测试”，查看是否正常连接集群。
- Jenkins URL：Jenkins 的访问路径，需要填写步骤 6 中设置的 8080 端口地址（**此处填写的地址必须同时开放 8080 和 50000 端口，即集群内访问的 IP 地址**），例如“http://10.247.222.254:8080”。

图4-11 Kubernetes Cloud details 填写示例



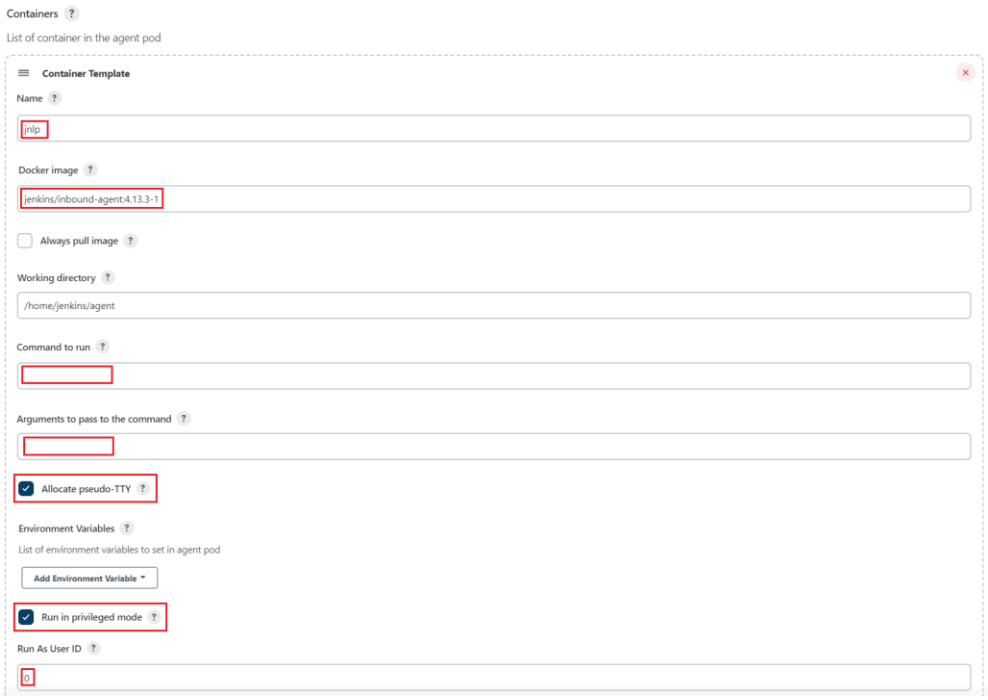

步骤 5 **Pod Templates**：单击“Add Pod Template > Pod Template details”，填写 Pod 模板参数。

- 配置 Pod 模板基本参数：参数配置如图 4-12 所示。
  - Name：jenkins-agent
  - Namespace：cicd
  - Labels：jenkins-agent
  - Usage：选择“Use this node as much as possible”

图4-12 Pod Template 基本参数


- 添加容器：单击“Add Container > Container Template”，参数配置如下图所示。
  - Name（名称）：必须为 **jnlp**。
  - Docker image（镜像）：**jenkins/inbound-agent:4.13.3-1**。此处镜像版本可能随时间变化发生变动，请根据实际情况进行选择，或使用 latest 版本。
  - Working directory（工作目录）：默认为/home/jenkins/agent
  - Command to run（运行的命令）/Arguments to pass to the command（命令参数）：需要删除已有的默认值，保持空值。
  - Allocate pseudo-TTY（分配伪终端）：勾选该参数。
  - Advanced（高级）：勾选“Run in privileged mode”，并填写“Run As User ID”为 0（即 root 用户）。

图4-13 Container Template 参数



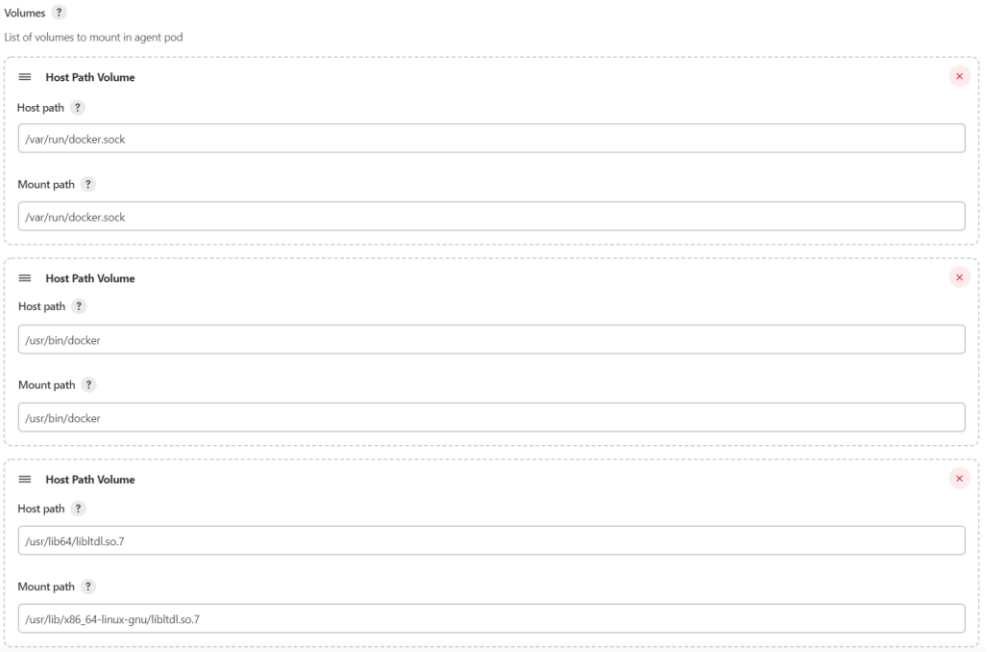

- 添加卷：单击“Add Volume > Host Path Volume”，将下表中的主机路径挂载到容器对应路径。

表4-5 挂载路径

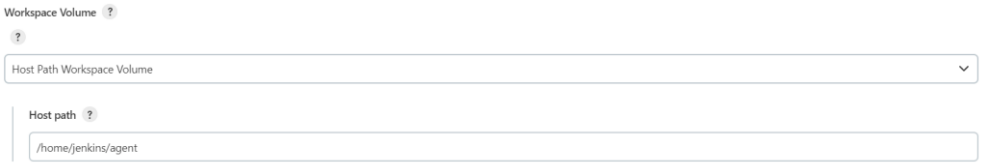
| 存储类型 | 主机路径（HostPath） | 挂载路径 |
| --- | --- | --- |
| 主机路径（HostPath） | • 节点使用 docker 容器引擎：**/var/run/docker.sock**<br>• 节点使用 containerd 容器引擎：**/var/run/containerd** | • 点使用 docker 容器引擎：**/var/run/docker.sock**<br>• 节点使用 containerd 容器引擎：**/var/run/containerd** |
| 主机路径（HostPath） | • 节点使用 docker 容器引擎：**/usr/bin/docker**<br>• 节点使用 containerd 容器引擎：**/usr/local/bin/ctr** | • 节点使用 docker 容器引擎：**/usr/bin/docker**<br>• 节点使用 containerd 容器引擎：**/usr/local/bin/ctr** |
| 主机路径（HostPath） | **/usr/lib64/libltdl.so.7** | **/usr/lib/x86_64-linux-gnu/libltdl.so.7** |
| 主机路径（HostPath） | **/usr/bin/kubectl** | **/usr/local/bin/kubectl** |

挂载完成后，如下图所示。

图4-14 挂载主机到容器对应路径




- Run As User ID：0（即 root 用户）。
- Workspace Volume（工作空间卷）：agent 的工作目录，建议做持久化。选择“Host Path Workspace Volume”，主机路径填写/home/jenkins/agent。




步骤 6 填写完成后，单击“Save”保存。

**说明**

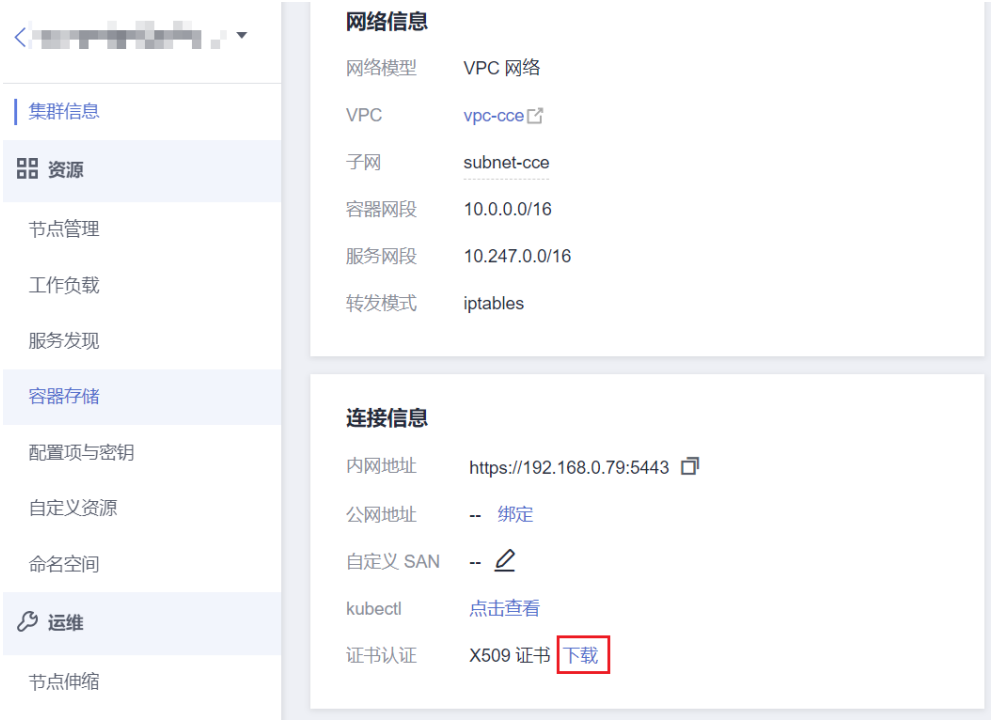
Agent 设置完成后，建议将 Master 的并发构建数设置为 0，即不使用 Master 进行本地构建，全部使用 Agent 进行构建，具体操作步骤请参见修改并发构建数量。

**----结束**

## 设置集群访问凭证

在 Jenkins 中能够识别的证书文件为 PKCS#12 certificate，因此需要先将集群证书转换生成 PKCS#12 格式的 pfx 证书文件。

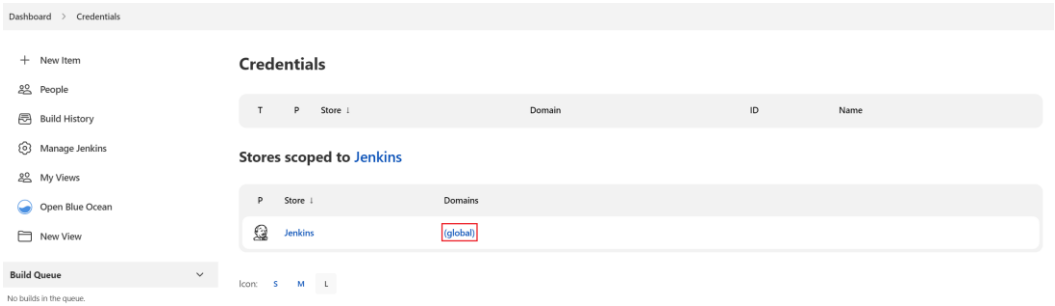
步骤 1 前往 CCE 控制台的“集群信息 > 连接信息”页面中下载集群证书，证书包含 ca.crt、client.crt、client.key 三个文件。




步骤 2 登录一台 Linux 主机，将三个证书文件放在同一目录，并通过 openssl 进行证书格式的转换，生成一个 Client P12 认证文件 cert.pfx。生成证书时，会提示输入密码 ，这里密码可自定义。

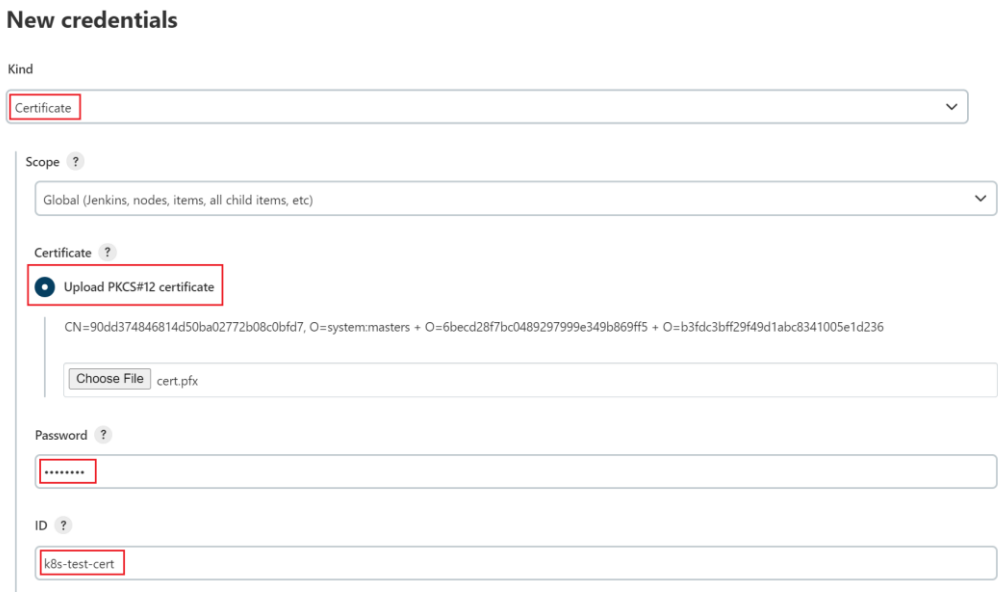
```
openssl pkcs12 -export -out cert.pfx -inkey client.key -in client.crt -certfile
ca.crt
```

步骤 3 在 Jenkins 的“系统管理 > Manage Credentials”中，单击 Jenkins 默认的“全局”凭据存储域，您也可以自行新建域。




步骤 4 单击“添加凭据”，创建新的凭据。

- 类型：选择 Certificate。
- 范围：全局。
- 证书：选择 Upload PKCS#12 certificate，并上传步骤 2 时生成的 cert.pfx 文件。
- 密码：转换 cert.pfx 时自定义设置的密码。
- ID：可以自定义，此处设置为 k8s-test-cert。




**----结束**

## 4.1.3.3 使用 Jenkins 构建流水线

### 获取长期的 docker login 命令

在 Jenkins 安装部署过程中，若构建容器运行在使用 docker 容器引擎的节点上，会在容器中挂载 docker 命令（参见 9），故 Jenkins 对接 SWR 无需额外配置，可直接执行 docker 命令。仅需获取长期有效的 SWR 登录指令，具体步骤请参见获取长期有效 docker login 指令。

若构建容器运行在 containerd 容器引擎的节点上，需要在节点上安装 docker（参见 Docker Engine installation）并在容器中挂载（参见 9）。

例如本帐号的命令为：

```
docker login -u cn-jssz1@xxxxx -p xxxxx registry.cn-jssz1.ctyun.cn
```

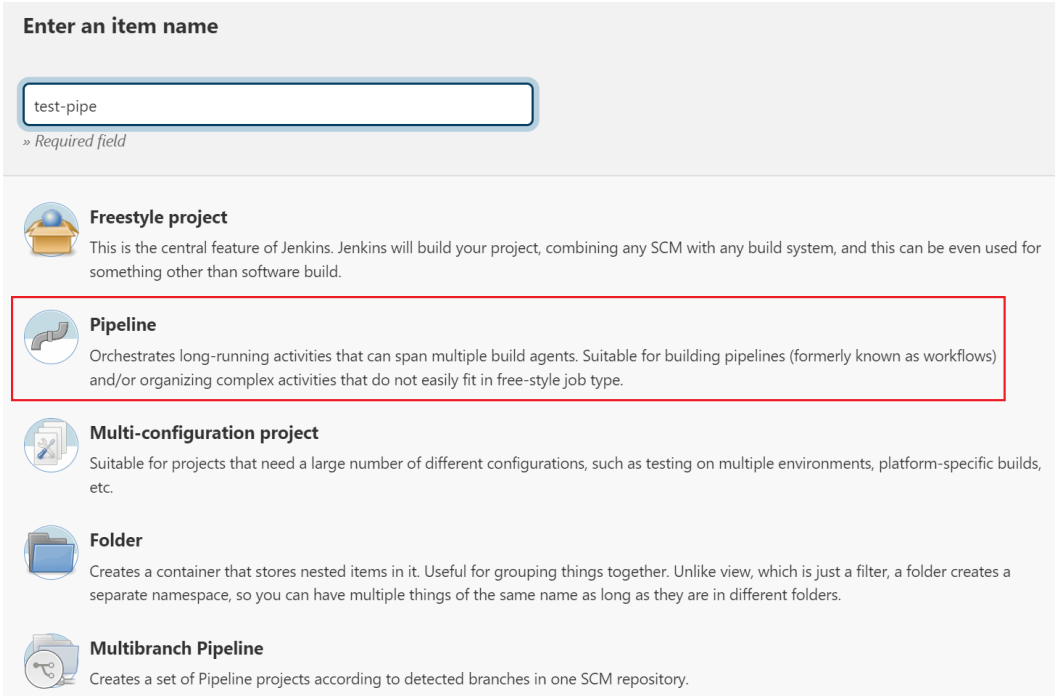
### 创建 pipeline 完成镜像构建及 push

本示例将使用 Jenkins 构建一条流水线，该流水线的作用是从代码仓中拉取代码并打包成镜像推送到 SWR 镜像仓库中。

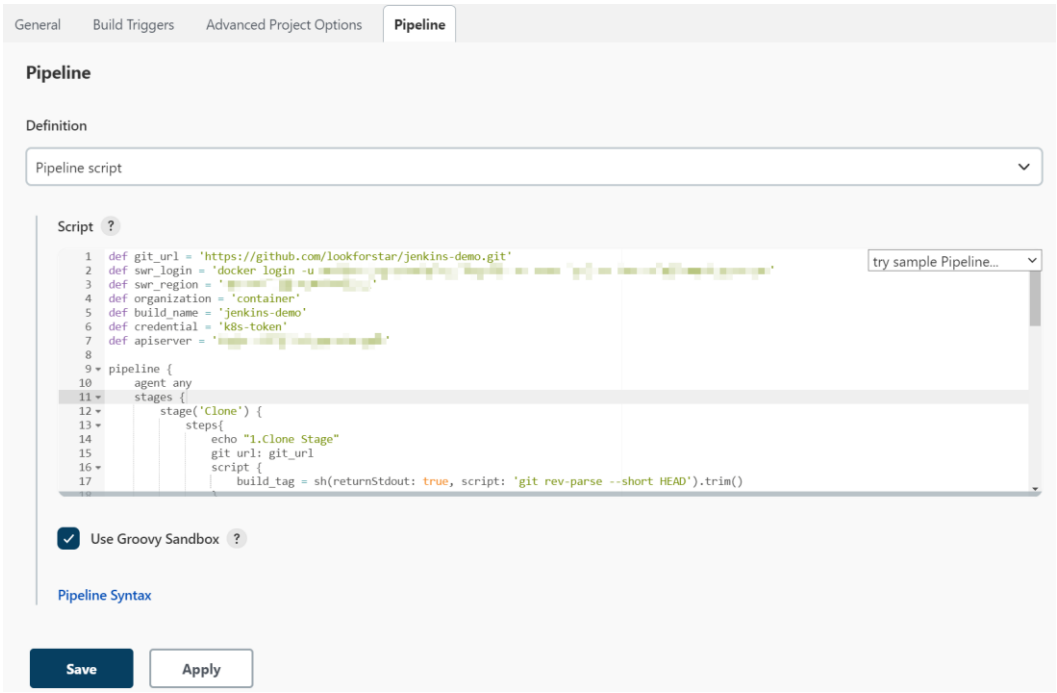
创建 pipeline 步骤如下：

**步骤 1** 在 Jenkins 界面单击“New Item”。

**步骤 2** 输入任务名称，并选择创建流水线。




步骤 3　配置 pipeline 脚本，其他步骤不配置。




以下 pipeline 脚本仅供您参考，您可根据自身业务自定义脚本内容，关于更多关于流水线脚本的语法请参考 Pipeline。

示例脚本中的部分参数需要修改：

- git_url：您代码仓库的地址，需要替换为实际取值。
- swr_login：登录命令为获取长期的 docker login 命令获取的命令。
- swr_region：SWR 的区域。

- organization：SWR 中的实际组织名称。
- build_name：制作的镜像名称。
- credential：添加到 Jenkins 的集群凭证，请填写凭证 ID。如果需要部署在另一个集群，需要重新将这个集群的访问凭证添加到 Jenkins，具体操作请参考设置集群访问凭证。
- apiserver：部署应用集群的 APIserver 地址，需保证从 Jenkins 集群可以正常访问该地址。

```
//定义代码仓地址
def git_url = 'https://github.com/lookforstar/jenkins-demo.git'
//定义 SWR 登录指令
def swr_login = 'docker login -u cn-jssz1@xxxxx -p xxxxx registry.cn-
jssz1.ctyun.cn'
//定义 SWR 区域
def swr_region = 'cn-jssz1'
//定义需要上传的 SWR 组织名称
def organization = 'container'
//定义镜像名称
def build_name = 'jenkins-demo'
//部署集群的证书 ID
def credential = 'k8s-token'
//集群的 APIserver 地址，需保证从 Jenkins 集群可以正常访问该地址
def apiserver = 'https://192.168.0.100:6443'

pipeline {
    agent any
    stages {
        stage('Clone') {
            steps{
                echo "1.Clone Stage"
                git url: git_url
                script {
                    build_tag = sh(returnStdout: true, script: 'git rev-parse --short
HEAD').trim()
                }
            }
        }
        stage('Test') {
            steps{
                echo "2.Test Stage"
            }
        }
        stage('Build') {
            steps{
                echo "3.Build Docker Image Stage"
                sh "docker build -t
registry.${swr_region}.ctyun.cn/${organization}/${build_name}:${build_tag} ."
                //${build_tag}表示获取上文中的 build_tag 变量作为镜像标签，为 git rev-parse --
short HEAD 命令的返回值，即 commit ID。
            }
        }
        stage('Push') {
            steps{
                echo "4.Push Docker Image Stage"
```

```
            sh swr_login
            sh "docker push
registry.${swr_region}.ctyun.cn/${organization}/${build_name}:${build_tag}"
         }
      }
      stage('Deploy') {
         steps{
            echo "5. Deploy Stage"
            echo "This is a deploy step to test"
            script {
            sh "cat k8s.yaml"
            echo "begin to config kubenetes"
            try {
               withKubeConfig([credentialsId: credential, serverUrl: apiserver]) {
                  sh 'kubectl apply -f k8s.yaml'
                  //该 YAML 文件位于代码仓中，此处仅做示例，请您自行替换
               }
               println "hooray, success"
            } catch (e) {
               println "oh no! Deployment failed! "
               println e
            }
            }
         }
      }
   }
}
```

步骤 4 保存后执行 Jenkins job。

**----结束**

## 4.1.3.4 参考：Jenkins 对接 Kubernetes 集群的 RBAC

### 前提条件

集群需要开启 RBAC。

### 场景一：基于 namespace 的权限控制

**新建 ServiceAccount 和 role，然后做 rolebinding**

```
$ kubectl create ns dev
$ kubectl -n dev create sa dev

$ cat <<EOF > dev-user-role.yml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: dev
  name: dev-user-pod
rules:
- apiGroups: ["*"]
  resources: ["deployments", "pods", "pods/log"]
```

```
  verbs: ["get", "watch", "list", "update", "create", "delete"]
EOF
kubectl create -f dev-user-role.yml

$ kubectl create rolebinding dev-view-pod \
   --role=dev-user-pod \
   --serviceaccount=dev:dev \
   --namespace=dev
```

### 生成指定 ServiceAccount 的 kubeconfig 文件

**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。

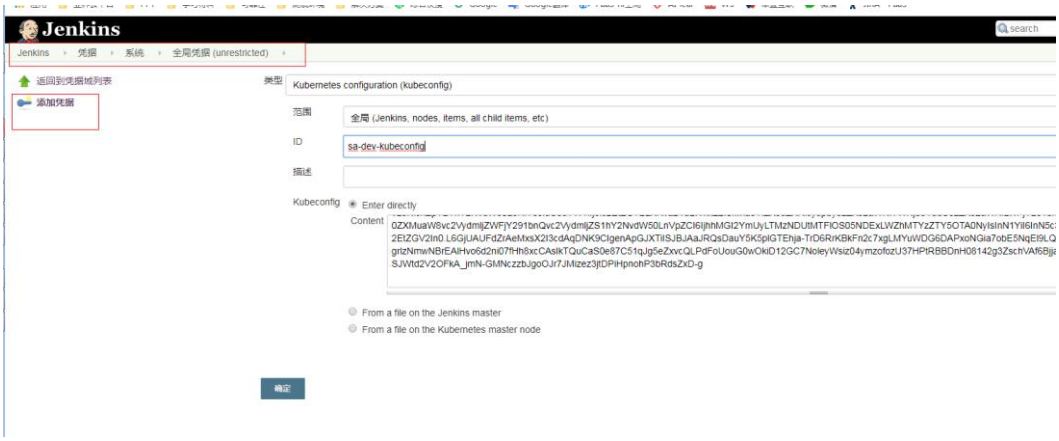
```
$ SECRET=$(kubectl -n dev get sa dev -o go-
template='{{range .secrets}}{{.name}}{{end}}')
$ API_SERVER="https://172.22.132.51:6443"
$ CA_CERT=$(kubectl -n dev get secret ${SECRET} -o yaml | awk '/ca.crt:/{print $2}')
$ cat <<EOF > dev.conf
apiVersion: v1
kind: Config
clusters:
- cluster:
   certificate-authority-data: $CA_CERT
   server: $API_SERVER
  name: cluster
EOF

$ TOKEN=$(kubectl -n dev get secret ${SECRET} -o go-template='{{.data.token}}')
$ kubectl config set-credentials dev-user \
   --token=`echo ${TOKEN} | base64 -d` \
   --kubeconfig=dev.conf

$ kubectl config set-context default \
   --cluster=cluster \
   --user=dev-user \
   --kubeconfig=dev.conf

$ kubectl config use-context default \
   --kubeconfig=dev.conf
```

命令行中验证

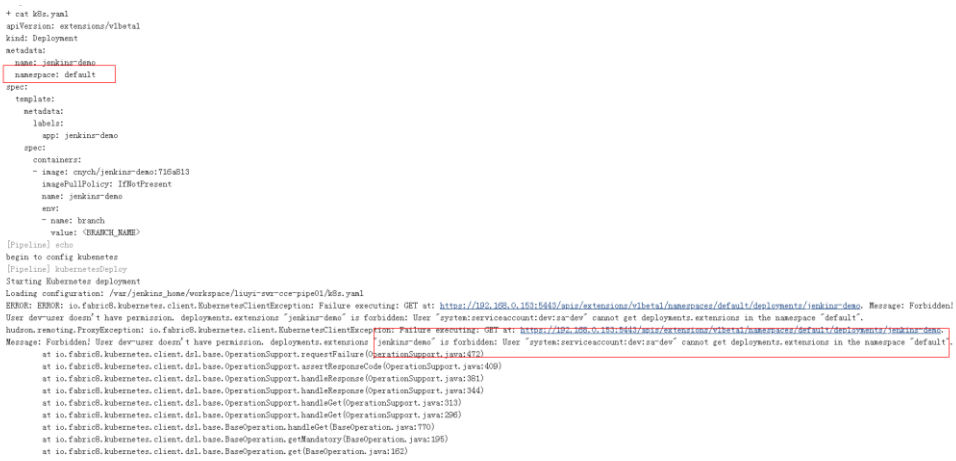
```
$ kubectl --kubeconfig=dev.conf get po
Error from server (Forbidden): pods is forbidden: User
"system:serviceaccount:dev:dev" cannot list pods in the namespace "default"

$ kubectl -n dev --kubeconfig=dev.conf run nginx --image nginx --port 80 --
restart=Never
$ kubectl -n dev --kubeconfig=dev.conf get po
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          39s
```

**Jenkins 中验证权限是否符合预期**

步骤 1 添加有权限控制的 kubeconfig 到 Jenkins 系统中




步骤 2 启动 Jenkins 任务，部署到 namespace default 失败，部署到 namespace dev 成功。




**----结束**

## 场景二：基于具体资源的权限控制

步骤 1　生成 SA 和 role 及绑定：

```
 kubectl -n dev create sa sa-test0304

 cat <<EOF > test0304-role.yml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: dev
  name: role-test0304
rules:
- apiGroups: ["*"]
  resources: ["deployments"]
  resourceNames: ["tomcat03", "tomcat04"]
  verbs: ["get", "update", "patch"]
EOF
kubectl create -f test0304-role.yml

 kubectl create rolebinding test0304-bind \
    --role=role-test0304 \
    --serviceaccount=dev:sa-test0304\
    --namespace=dev
```

步骤 2　生成 kubeconfig 文件：

**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。

```
 SECRET=$(kubectl -n dev get sa sa-test0304 -o go-
template='{{range .secrets}}{{.name}}{{end}}')
 API_SERVER=" https://192.168.0.153:5443"
 CA_CERT=$(kubectl -n dev get secret ${SECRET} -o yaml | awk '/ca.crt:/{print $2}')
 cat <<EOF > test0304.conf
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_CERT
    server: $API_SERVER
  name: cluster
EOF

TOKEN=$(kubectl -n dev get secret ${SECRET} -o go-template='{{.data.token}}')
 kubectl config set-credentials test0304-user \
    --token=`echo ${TOKEN} | base64 -d` \
    --kubeconfig=test0304.conf

 kubectl config set-context default \
    --cluster=cluster \
    --user=test0304-user \
    --kubeconfig=test0304.conf

 kubectl config use-context default \
    --kubeconfig=test0304.conf
```

步骤 3 Jenkins 中的运行效果符合预期。

Pipeline 脚本，依次更新 tomcat03/04/05 的 deployment。

```
      try {
        kubernetesDeploy(
            kubeconfigId: "test0304",
            configs: "test03.yaml")
        println "hooray, success"
       } catch (e) {
          println "oh no! Deployment failed! "
          println e
       }
       echo "test04"
       try {
        kubernetesDeploy(
            kubeconfigId: "test0304",
            configs: "test04.yaml")
        println "hooray, success"
       } catch (e) {
          println "oh no! Deployment failed! "
```

```
        println e
    }
    echo "test05"
    try {
     kubernetesDeploy(
         kubeconfigId: "test0304",
         configs: "test05.yaml")
     println "hooray, success"
    } catch (e) {
        println "oh no! Deployment failed! "
        println e
    }
```

查看运行效果：

图4-15 test03

图4-16 test04

```
test04
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/liuyi-swr-cce-pipe01/test04.yaml
Applied Deployment: Deployment(apiVersion=extensions/v1beta1, kind=Deployment, metadata=Obje
deletionTimestamp=null, finalizers=[], generateName=null, generation=3, initializers=null, l
selfLink=/apis/extensions/v1beta1/namespaces/dev/deployments/tomcat04, uid=06af3b14-3356-11e
progressDeadlineSeconds=null, replicas=1, revisionHistoryLimit=null, rollbackTo=null, select
strategy=DeploymentStrategy(rollingUpdate=RollingUpdateDeployment(maxSurge=IntOrString(IntVa
additionalProperties={}), additionalProperties={}), type=RollingUpdate, additionalProperties
deletionGracePeriodSeconds=null, deletionTimestamp=null, finalizers=[], generateName=null, g
resourceVersion=null, selfLink=null, uid=null, additionalProperties={}), spec=PodSpec(active
[EnvVar(name=branch, value=<BRANCH_NAME>, valueFrom=null, additionalProperties={})], envFrom
livenessProbe=null, name=tomcat04, ports=[], readinessProbe=null, resources=ResourceRequirem
terminationMessagePath=/dev/termination-log, terminationMessagePolicy=File, tty=null, volume
hostNetwork=null, hostPID=null, hostname=null, imagePullSecrets=[], initContainers=[], nodeN
securityContext=PodSecurityContext(fsGroup=null, runAsNonRoot=null, runAsUser=null, seLinuxO
terminationGracePeriodSeconds=30, tolerations=[], volumes=[], additionalProperties={}), addi
conditions=[DeploymentCondition(lastTransitionTime=2019-02-18T08:56:55Z, lastUpdateTime=2019
type=Available, additionalProperties={}), DeploymentCondition(lastTransitionTime=2019-02-18T
reason=ReplicaSetUpdated, status=True, type=Progressing, additionalProperties={})], observed
additionalProperties={})
Finished Kubernetes deployment
[Pipeline] echo
hooray, success
[Pipeline] echo
test05
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/liuyi-swr-cce-pipe01/test05.yaml
ERROR: ERROR: io.fabric8.kubernetes.client.KubernetesClientException: Failure executing: GET
test0304-user doesn't have permission. deployments.extensions "tomcat05" is forbidden: User
hudson.remoting.ProxyException: io.fabric8.kubernetes.client.KubernetesClientException: Fail
Forbidden! User test0304-user doesn't have permission. deployments.extensions "tomcat05" is
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.requestFailure(OperationSu
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.assertResponseCode(Operati
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleResponse(OperationSu
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleResponse(OperationSu
```

**----结束**

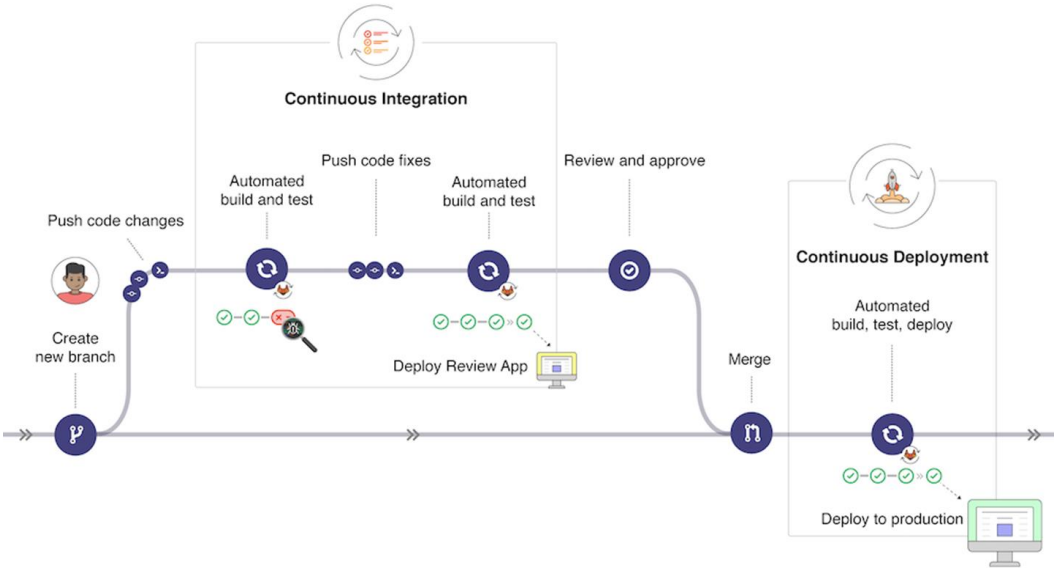
# 4.2 Gitlab 对接 SWR 和 CCE 执行 CI/CD

## 应用现状

GitLab 是利用 Ruby on Rails 一个开源的版本管理系统，实现一个自托管的 Git 项目仓库，可通过 Web 界面进行访问公开的或者私人项目。与 Github 类似，GitLab 能够浏览源代码，管理缺陷和注释。可以管理团队对仓库的访问，它非常易于浏览提交过的版本并提供一个文件历史库。团队成员可以利用内置的简单聊天程序(Wall)进行交流。

GitLab 的 CI/CD 功能强大，在软件开发业界有着广泛的应用。

图4-17 GitLab CI/CD 流程




本文介绍在 Gitlab 中对接 SWR 和 CCE 执行 CI/CD，并通过一个具体的过程演示该过程。

## 准备工作

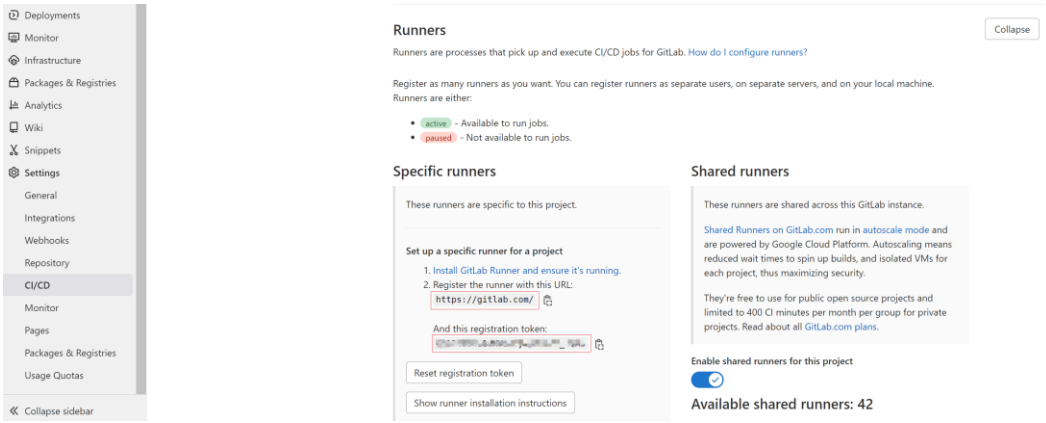
1. 创建一台 CCE 集群，且需要给节点绑定一个 EIP，用于安装 Gitlab Runner 时下载镜像。
2. 下载并配置 kubectl 连接集群。

   登录 CCE 控制台，在集群信息页面单击 kubectl 对应的“配置”按钮，按照指导配置 kubectl。


3. 安装 helm 3，具体请参见 https://helm.sh/zh/docs/intro/install/。

## 安装 Gitlab Runner

登录 Gitlab，进入项目视图的 Settings->CI/CD，单击 Runners 旁边的 Expand，查找 Gitlab Runner 注册 URL 和 Token，如下图所示。




创建 values.yaml 文件，填写如下信息。

```
# 注册 URL
gitlabUrl: https://gitlab.com/
# 注册 token
runnerRegistrationToken: "GR13489411dKVzmTyaywEDTF_1QXb"
rbac:
    create: true
runners:
    privileged: true
```

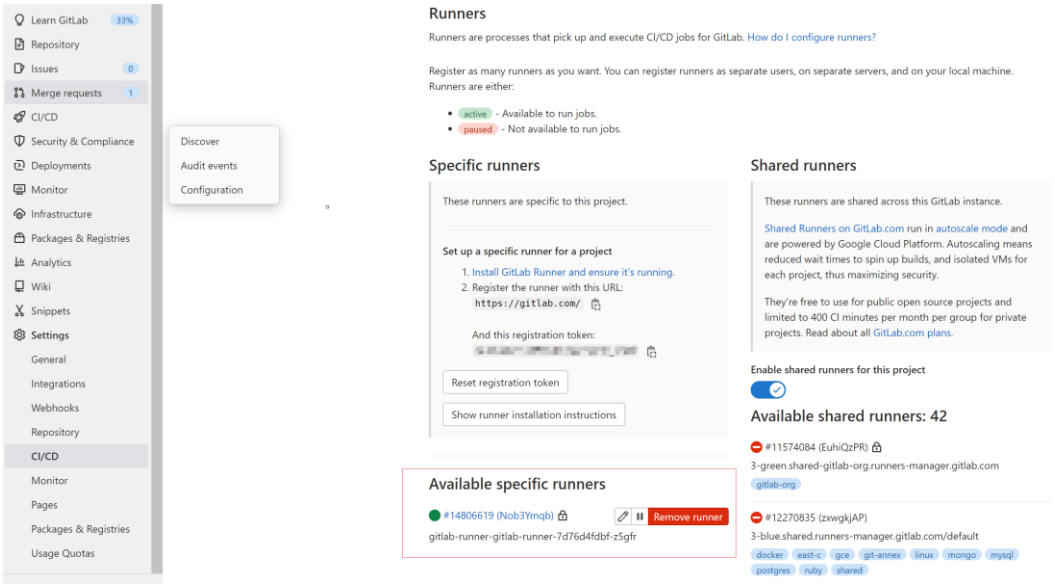
创建 gitlab 命名空间。

```
kubectl create namespace gitlab
```

通过 helm 安装 Gitlab Runner。

```
helm repo add gitlab https://charts.gitlab.io
helm install --namespace gitlab gitlab-runner -f values.yaml gitlab/gitlab-runner
```

安装完成后，可以在 CCE 控制台查询到 gitlab-runner 的工作负载，等待一段时间后在 Gitlab 中可以看到连接信息，如下图所示。




## 创建应用

将您要创建的应用放到 Gitlab 项目仓库中，本文使用一个修改 nginx 的示例，具体请参见 https://gitlab.com/c8147/cidemo/-/tree/main。

其中包括如下文件。

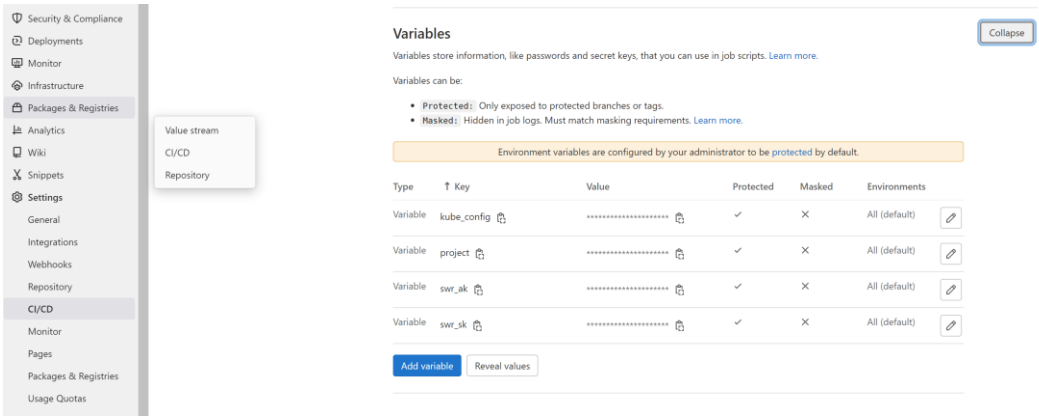
- **.gitlab-ci.yml**：Gitlab CI 文件，将在创建流水线中详细讲解。
- **Dockerfile**：用于制作 Docker 镜像。
- **index.html**：用于替换 nginx 的 index 页面。
- **k8s.yaml**：用于部署 nginx 应用，会创建一个名为 nginx-test 的 Deployment，和一个名为 nginx-test 的 Service。

以上文件仅为示例，您可以根据您的业务需求进行替换或修改。

## 设置全局变量

流水线运行过程中，会先 Build 镜像上传到 SWR，然后执行 kubectl 命令在集群中部署，这就需要能够登录 SWR 镜像仓库，并且要有集群的连接凭证。实际执行中可以将这些信息在 Gitlab 中定义成变量。

登录 Gitlab，进入项目视图的 Settings->CI/CD，单击 Variables 旁边的 Expand，添加变量。




- kube_config：

  kubeconfig.json 文件，用于执行 kubectl 命令鉴权使用，需要转换成 base64 格式，在配置好 kubectl 的机器上，执行如下命令。

  **echo $(cat ~/.kube/config | base64) | tr -d " "**

  回显的字符串即为 kubeconfig.json 的内容。

- project：项目名称。

  登录管理控制台，单击右上角您的用户名处，单击“我的凭证”。在“API 凭证”的项目列表中查找当前区域对应的项目。

- swr_ak：密钥的 AK。

  登录管理控制台，单击右上角您的用户名处，单击“我的凭证”。在左侧导航栏中选择“访问密钥”，单击“新增访问密钥”，输入描述信息，单击“确定”。在弹出的提示页面单击“立即下载”。下载成功后，在“credentials”文件中即可获取 AK 和 SK 信息。

- swr_sk：登录 SWR 镜像仓库的密钥。

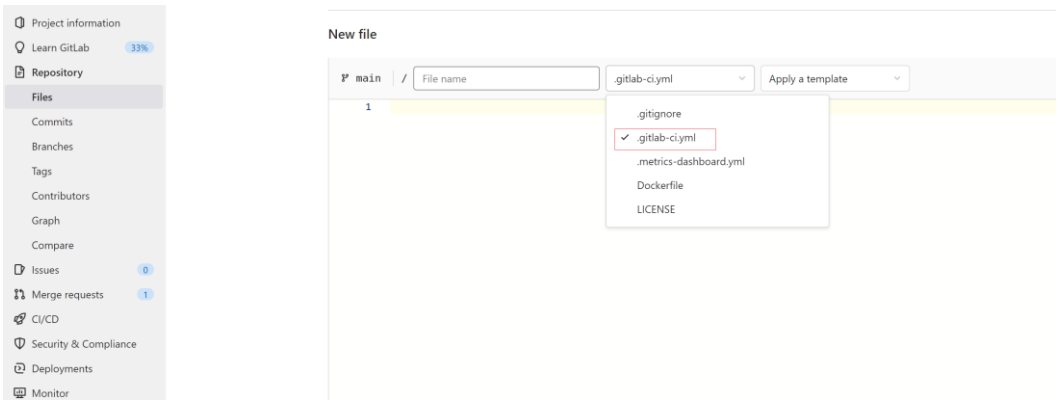
  执行如下命令获取密钥，其中$AK 和$SK 为上面获取的 AK/SK。

  **printf "*$AK*" | openssl dgst -binary -sha256 -hmac "*$SK*" | od -An -vtx1 | sed 's/[ \n]//g' | sed 'N;s/\n//'**

  回显的字符串即为登录密钥。

## 创建流水线

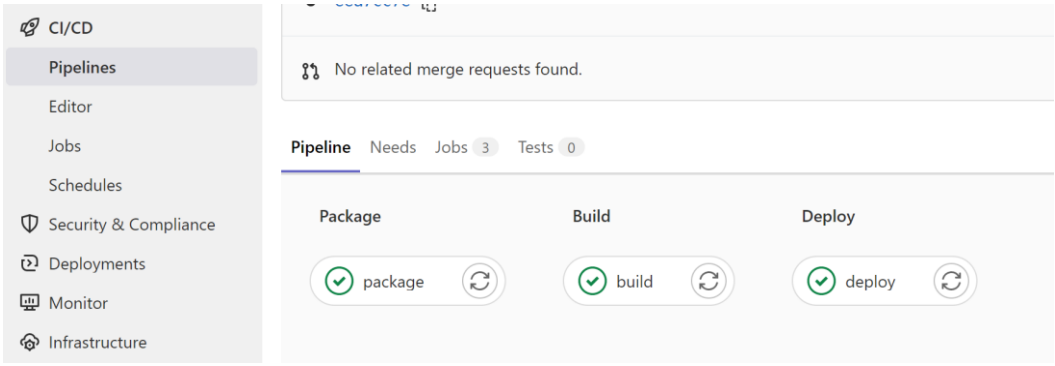
登录 Gitlab，在 Repository 中添加**.gitlab-ci.yml** 文件。



其内容如下所示。

```
#定义 pipeline 中的阶段，包含打包、构建和部署三个阶段
stages:
  - package
  - build
  - deploy
#各个构建阶段不指定镜像时，使用默认镜像 docker:latest
image: docker:latest
#package 阶段只打印，不做任何操作
package:
  stage: package
  script:
    - echo "package"
# build 阶段使用 docker in docker 方式
build:
  stage: build
  # 定义 build 阶段的环境变量
  variables:
    DOCKER_HOST: tcp://docker:2375
  # 定义 docker in docker 运行的镜像
  services:
    - docker:18.09-dind
  script:
    - echo "build"
    # 登录 SWR
    - docker login -u $project@$swr_ak -p $swr_sk registry.cn-jssz1.ctyun.cn
    # 构建镜像，其中 k8s-dev 为 SWR 中的组织名称，请根据实际情况替换
    - docker build -t registry.cn-jssz1.ctyun.cn/k8s-dev/nginx:$CI PIPELINE ID .
    # 推送镜像到 SWR
    - docker push registry.cn-jssz1.ctyun.cn/k8s-dev/nginx:$CI PIPELINE ID
deploy:
  # 使用 kubectl 镜像
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  stage: deploy
  script:
    # 配置 kubeconfig 文件
    - mkdir -p $HOME/.kube
    - export KUBECONFIG=$HOME/.kube/config
    - echo $kube_config |base64 -d > $KUBECONFIG
    # 替换 k8s.yaml 文件中的镜像
    - sed -i "s/<IMAGE_NAME>/registry.cn-jssz1.ctyun.cn\/k8s-
dev\/nginx:$CI_PIPELINE_ID/g" k8s.yaml
    - cat k8s.yaml
    # 部署应用
    - kubectl apply -f k8s.yaml
```

**.gitlab-ci.yml** 文件保存后，会立即启动执行流水线，在 Gitlab 中查看流水线执行情况，如下所示。




## 验证结果

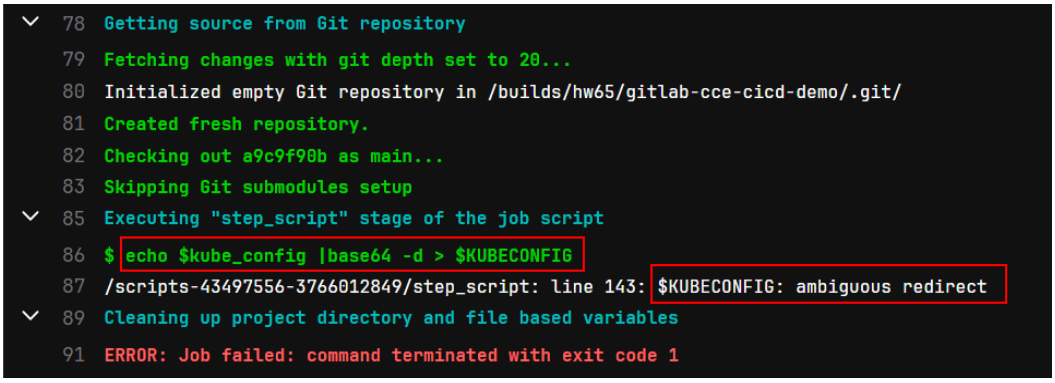
流水线部署成功后，在 CCE 控制台找到名为 nginx-test 的 Service，查询到 nginx-test 的访问地址，使用 curl 命令访问。

```
# curl xxx.xxx.xxx.xxx:31111
Hello Gitlab!
```

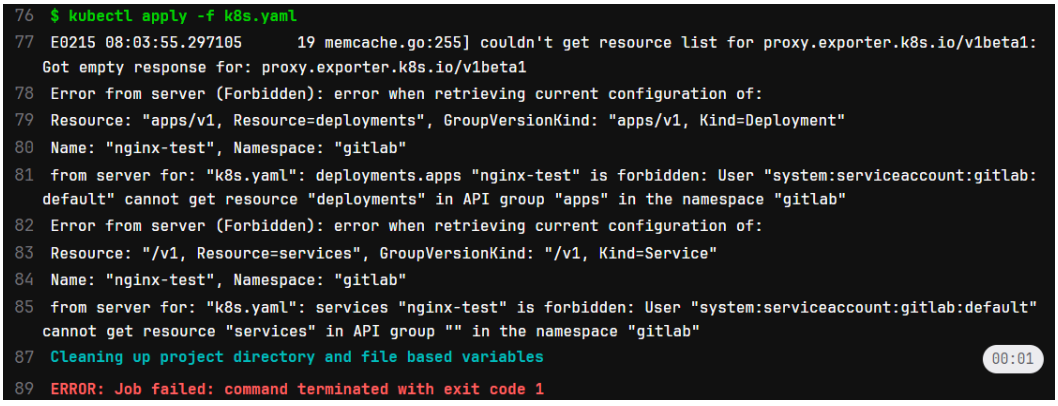
如果能得到如上回显，则说明部署正确。

## 常见问题

如果在部署阶段出现如下问题：




或




请检查**.gitlab-ci.yml** 文件中是否缺少如下两行命令，如果缺少请在**.gitlab-ci.yml** 中补充命令。

```
...
deploy:
  # 使用 kubectl 镜像
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  stage: deploy
  script:
    # 配置 kubeconfig 文件
    - mkdir -p $HOME/.kube
    - export KUBECONFIG=$HOME/.kube/config
    - echo $kube_config |base64 -d > $KUBECONFIG
    # 替换 k8s.yaml 文件中的镜像
...
```

# 5 容灾

## 5.1 在 CCE 中实现高可用部署

### 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
```

# 6 安全

## 6.1 CCE 集群选用建议

基于安全责任共担模式，CCE 服务确保集群内 master 节点和 CCE 自身组件的安全，并在集群、容器级别提供一系列的层次化的安全能力，而用户则负责集群 Node 节点的安全并遵循 CCE 服务提供的安全最佳实践，做好安全配置和运维。

### CCE 服务的应用场景

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。

### 集群不建议在要求强资源隔离的场景下使用

CCE 给租户提供的是一个专属的独享集群，由于节点、网络等资源当前没有严格的隔离，在集群同时被多个外部不可控用户使用时，如果安全防护措施不严，就会存在较大的安全隐患。比如开发流水线场景，当允许多用户使用时，不同用户的业务代码逻辑不可控，存在集群以及集群下的其它服务被攻击的风险。

### 启用企业主机安全服务（HSS）

企业主机安全服务（HSS）拥有主机管理、风险预防、入侵检测、高级防御、安全运营、网页防篡改功能，能够全面识别并管理主机中的信息资产，实时监测主机中的风险并阻止非法入侵行为。推荐启用 HSS 服务保护用户 CCE 集群下的主机。

### 启用容器安全服务（CGS）

用户使用 CCE 时，配合使用容器安全服务（CGS）服务，该服务能够扫描镜像中的漏洞与配置信息，帮助用户解决传统安全软件无法感知容器环境的问题；同时提供容器进程白名单、文件只读保护和容器逃逸检测功能，有效防止容器运行时安全风险事件的发生。

# 6.2 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 3~6 个月左右同步发布新的 CCE 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 服务帐号，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帐号关联了什么角色。该服务帐号的 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，建议关闭服务帐号的自动关联功能。

禁用方法:

- 方法一：将服务帐号的 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 服务帐号。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帐号时，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帐号的功能。

  ```
  ...
   spec:
     template:
       spec:
         automountServiceAccountToken: false
         ...
  ```

## 合理配置用户的集群访问权限

CCE 支持帐号创建多个 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。

如果主帐号下需要配置多个 IAM 用户，应合理配置子用户和命名空间的权限。

- 配置集群权限请参考集群权限（IAM 授权）。
- 设置命名空间权限请参考命名空间权限（Kubernetes RBAC 授权）。

## 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。

- 云原生网络 2.0

  云原生网络 2.0 模型下，可以通过配置安全组达到 Pod 间网络隔离。

- 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。

## 6.3 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。

详情请参见常见问题 > 网络管理 > 网络规划。

## 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。

**⚠ 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“集群管理”界面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
  1. 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
  2. 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
  1. 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
  2. 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**

# 6.4 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**

# 6.5 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
    - "-l"
    - "/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/

# 7 弹性伸缩

## 7.1 使用 HPA+CA 实现工作负载和节点联动弹性伸缩

### 应用现状

企业应用的流量大小不是每时每刻都一样，有高峰和低谷，如果每时每刻都要保持可承载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 CCE 集群中工作负载及节点的数量，也就是弹性伸缩。

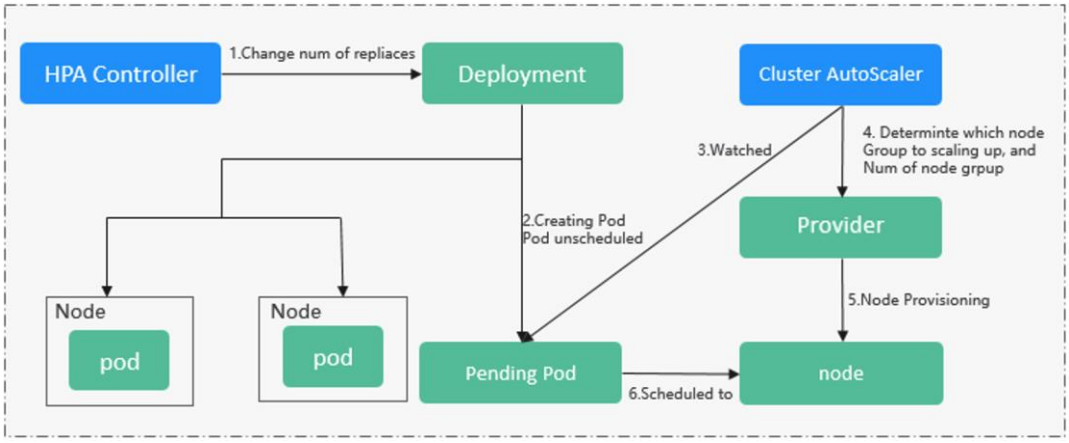
在 CCE 中，由于使用 Pod/容器部署应用，容器可使用的资源是在部署时即配置好，不会无限制使用 CCE 节点中的资源，所以在 CCE 中弹性伸缩需要先对 Pod 数量进行伸缩，Pod 数量增加后节点资源使用率才会增加，进而根据节点资源使用率再去伸缩集群中节点的数量。

### 解决方案

CCE 中的弹性伸缩主要使用 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 vCPU 和内存等资源才能扩容成功，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，这时就需要 CA 对集群节点进行缩容以释放资源，才不至于造成浪费。

如下图所示，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。




CCE 同时支持创建 CA 策略，根据 CPU/内存分配率扩容、还可以按照时间定期扩容节点，CA 策略可以与 autoscaler 默认的根据 Pod 的 Pending 状态进行扩容配合使用。

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。

本实践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 CCE 中的弹性伸缩。

## 准备工作

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，如下示例，这是一个 PHP 页面，代码将保存在 index.php 文件中，用户请求时先循环开方 1000000 次，然后再返回“OK!”。

```
<?php
  $x = 0.0001;
  for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
  }
  echo "OK!";
?>
```

编写 Dockerfile 制作容器镜像。

```
FROM php:5-apache
COPY index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

执行如下命令构建镜像，镜像名称为 busy-php。

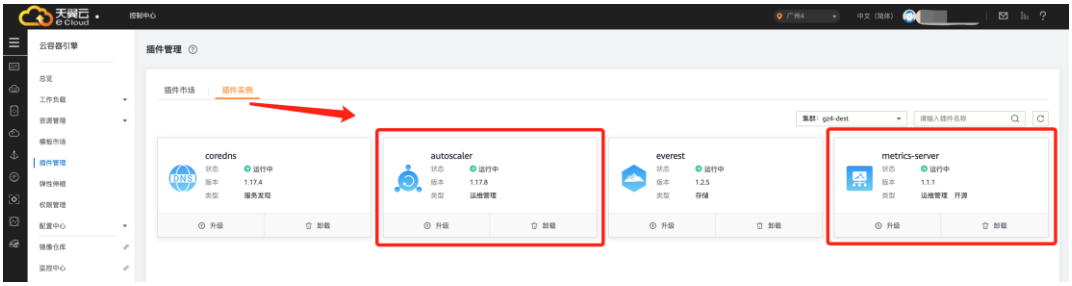
```
docker build -t busy-php:latest .
```

构建完成后上传到目标资源池的 SWR 镜像仓库，上传容器镜像的步骤请参见“容器镜像服务 > 客户端上传镜像”。


创建有 1 个工作节点的 CCE 集群，工作节点规格 2U4G，节点需要带弹性公网 IP，以便访问公网。

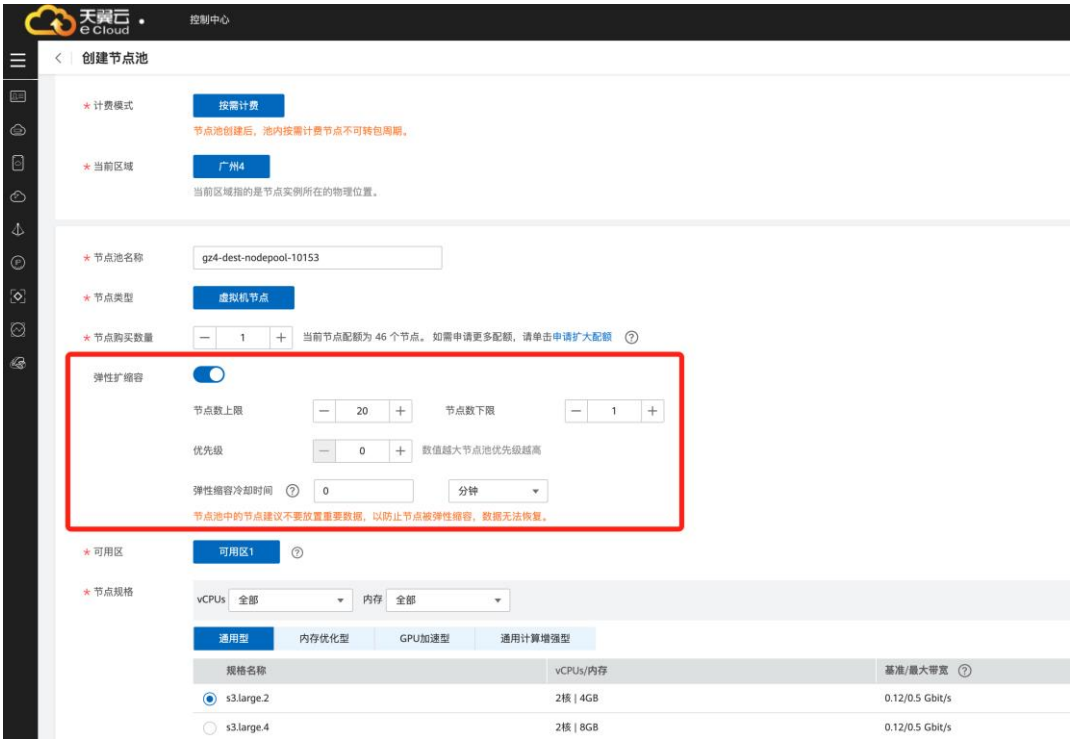
在 CCE 控制台“插件管理”中，如未安装请首先给集群安装好以下插件：

- **autoscaler**：CA 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。




## 创建节点池和 CA 策略

在 CCE 控制台中，创建一个节点池，添加一个 2U4G 的节点，并打开节点池的弹性扩缩容开关，如下图所示。




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。插件规格建议至少选择“高可用 50”，即保证运行不少于 2 个 autoscaler 实例。

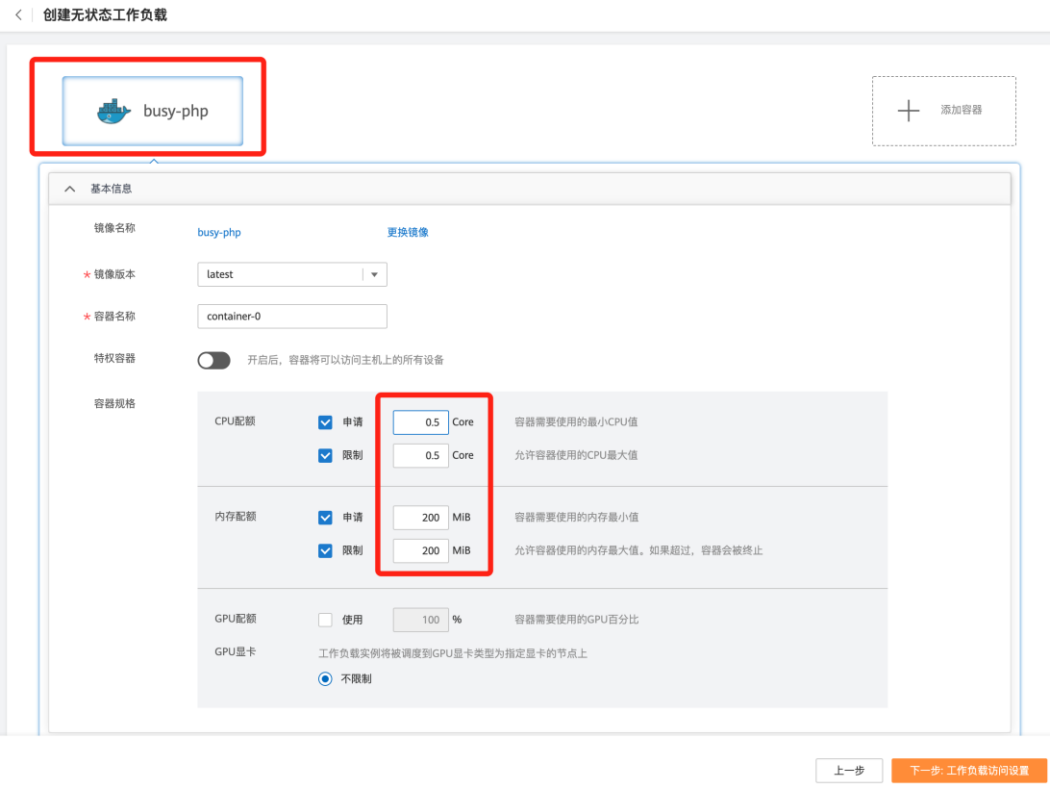
上面配置的节点池弹性伸缩，会根据 Pod 的 Pending 状态进行扩容，根据节点的资源使用率进行缩容。

CCE 同时支持创建 CA 策略，这里的 CA 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。CA 策略可以与 autoscaler 默认的根据 Pod 的 Pending 状态进行扩容共同作用。

如下图所示，在 CCE 控制台 > 弹性伸缩 > 节点伸缩中创建一个“节点伸缩策略”，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。


## 创建工作负载

使用刚构建的 busy-php 容器镜像创建无状态工作负载，副本数为 1。vCPU 设置为 0.5 core、内存设置为 200MiB，limits 与 requests 建议取值保持一致，避免扩缩容过程中出现震荡。



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。


## 创建 HPA 策略

创建 HPA 策略，如下图所示，该策略关联了名为 busy-php 的工作负载，期望 CPU 使用率为 50%。


另外有两个配置参数，一个是 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。另一个是扩缩容冷却时间窗，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。

## 准备压测环境

在本例中使用 linux 开源压测工具 wrk 模拟外部压力负载，您也可以使用其它压测工具进行模拟，确保对集群中的工作负载可形成持续的压力即可。

为确保压测效果，建议在节点池外的同一集群工作节点上安装并运行压测工具，本例以在 linux 工作节点上安装 wrk 为例：

如未安装 git、gcc，首先安装：

```
yum install git -y
yum install gcc -y
```

下载 wrk 工具：

```
git clone https://github.com/wg/wrk.git
```

进入 wrk 目录，编译：

```
cd wrk && make
```

完成编译后，可将 wrk 可执行文件软连接至/usr/local/bin 等目录下，方便后续使用。

首先通过如下命令测试工作负载是否正常，正常结果应为返回“OK!”。

```
curl http://192.168.0.149:31504
```

其中的{ip:port}为 busy-php 工作负载的访问地址和端口，可在负载详情页中获取：


验证 wrk 工具并查看结果是否正常：

```
wrk -t2 -c10 -d3s http://192.168.0.149:31504/
```

wrk 的详细使用方法和参数说明，请参考官方介绍：https://github.com/wg/wrk。

## 观察弹性伸缩过程

首先查看 CCE 集群中刚才新建的节点池情况，初始状态节点池中有 1 个节点。

**说明：本实践中的压测、工作负载和节点等相关指标值仅为参考示例。**

查看 HPA 策略，因为之前已进行过连通性测试，可以看到目标负载 busy-php 的指标（CPU 使用率）为 16%

```
[root@gz4-dest-node-d4e7j .kube]# kubectl get hpa gz-hpa --watch
NAME     REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS
gz-hpa   Deployment/busy-php   16%/50%   1         100       1
```

通过如下命令开始打压，其中{ip:port}为负载的访问地址，可以在 busy-php 负载的详情页中查询。

```
wrk -t10 -c1000 -d1200s http://192.168.0.149:31504/
```

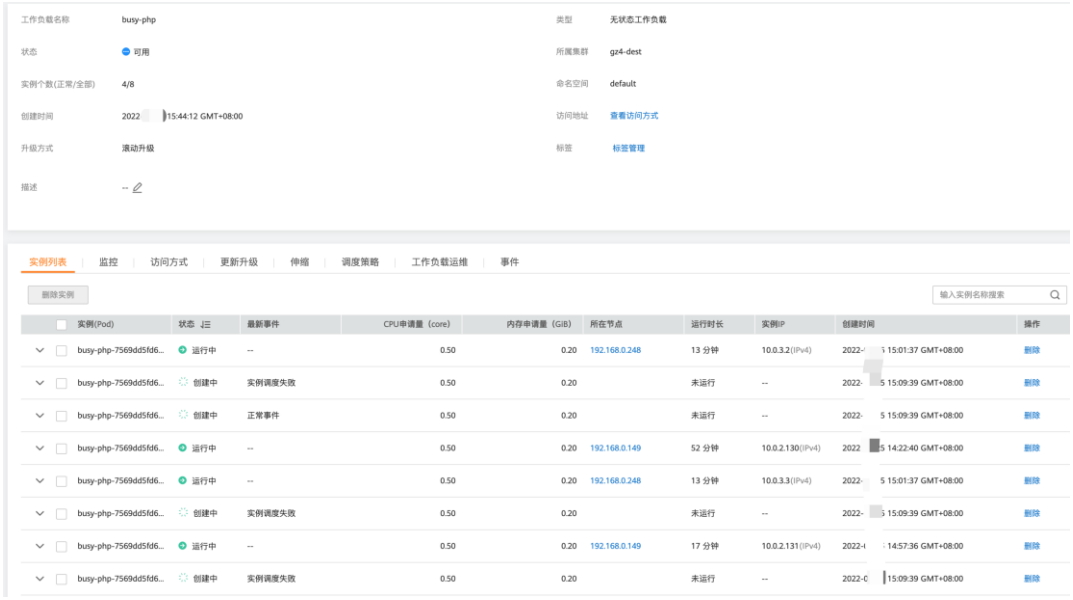
**说明：上述压测命令中的并发数、连接数、持续时间仅为示例，请根据节点规格等参数进行合理的设置。**

观察工作负载的伸缩过程。

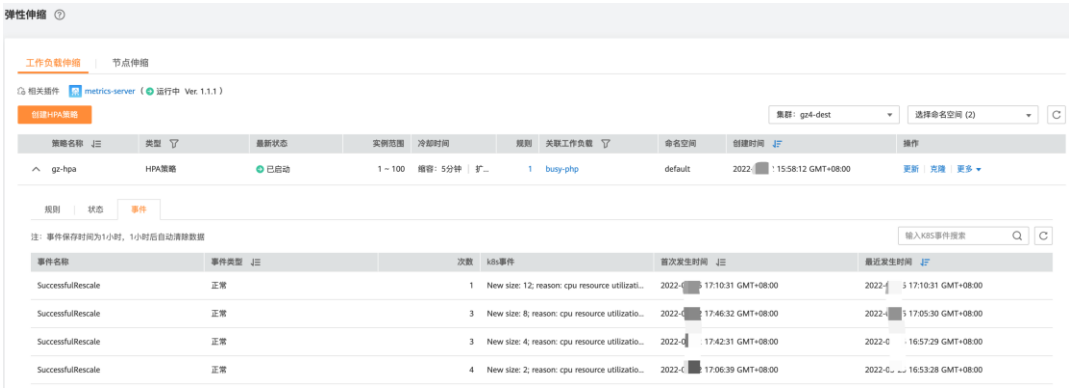
```
[root@gz4-dest-node-d4e7j .kube]# kubectl get hpa gz-hpa --watch
NAME     REFERENCE             TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
gz-hpa   Deployment/busy-php   16%/50%   1         100       1          3d
gz-hpa   Deployment/busy-php   99%/50%   1         100       1          3d
gz-hpa   Deployment/busy-php   99%/50%   1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   99%/50%   1         100       2          3d
gz-hpa   Deployment/busy-php   99%/50%   1         100       2          3d
gz-hpa   Deployment/busy-php   99%/50%   1         100       2          3d
gz-hpa   Deployment/busy-php   99%/50%   1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       2          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       4          3d
gz-hpa   Deployment/busy-php   100%/50%  1         100       4          3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       4          3d1h
gz-hpa   Deployment/busy-php   99%/50%   1         100       4          3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       4          3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       4          3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       4          3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       8          3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       8          3d1h
gz-hpa   Deployment/busy-php   99%/50%   1         100       8          3d1h
gz-hpa   Deployment/busy-php   98%/50%   1         100       8          3d1h
gz-hpa   Deployment/busy-php   98%/50%   1         100       12         3d1h
gz-hpa   Deployment/busy-php   100%/50%  1         100       12         3d1h
gz-hpa   Deployment/busy-php   8%/50%    1         100       12         3d1h
gz-hpa   Deployment/busy-php   8%/50%    1         100       12         3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       12         3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       12         3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       12         3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       2          3d1h
gz-hpa   Deployment/busy-php   0%/50%    1         100       1          3d1h
```

可以看到第二行开始负载的 CPU 使用率达到 99%，超过了目标值，此时触发了工作负载的弹性伸缩，将负载扩容为 2 个副本/Pod，随后的几分钟内，CPU 使用并未下降，这是因为虽然工作负载进行了扩容，但新创建的 Pod 并不一定创建成功，一般是因为资源不足 Pod 处于 Pending 状态，此时需同步进行节点扩容。

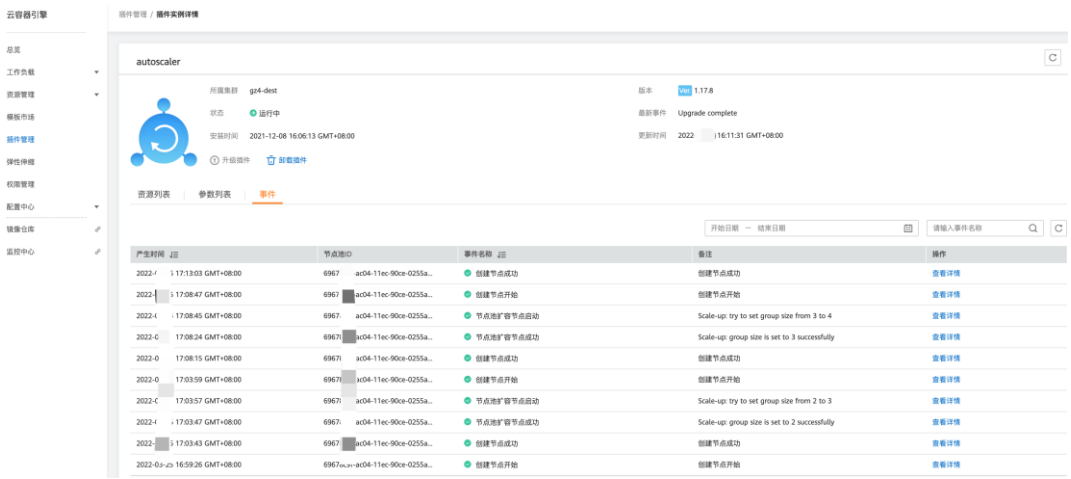
如下图所示，工作负载的副本数已通过动态扩容达到 8，但因为没有充足的 vCPU 和内存资源，会被 k8s 集群标记为“实例调度失败”。




之后工作负载 CPU 使用率一直保持在 99%以上，工作负载持续进行扩容，副本数从 2 个扩容到 4 个，再扩容到 8 个最后扩容至 12 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线，如下所示。




与此同时，查看节点池中的节点数量，发现在刚才工作负载扩容的同时，节点数量也扩容了。在 CCE 控制台中可以看到伸缩历史，节点数量会根据 CA 及 autoscaler 策略，通过判断 Pod 的 Pending 状态进行扩容。



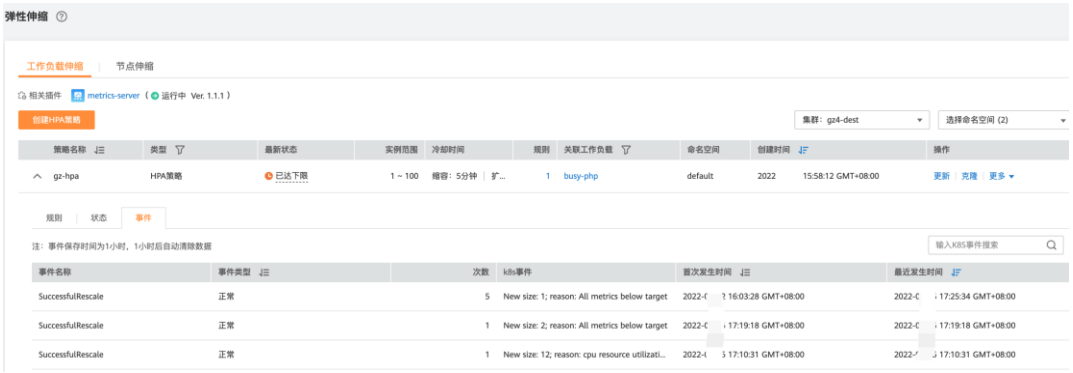
另外还可以看到 CA 策略也执行了一次，当集群中 CPU 分配率大于 70%，将节点池中节点数量从 2 扩容到了 3。


本例中节点扩容机制具体是这样：

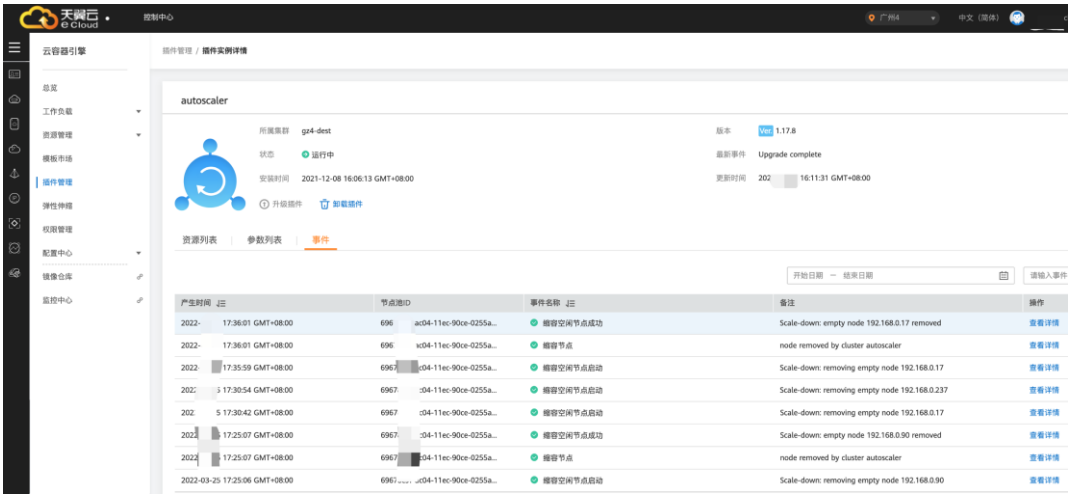
- Pod 数量变为 4 后，由于没有资源，Pod 处于 Pending 状态，触发了 autoscaler 默认的扩容策略，将节点数量进行增加。
- 同时因为集群中 CPU 分配率大于 70%，触发了 CA 策略，从而将节点数增加一个，从控制台上伸缩历史可以看出来。根据分配率扩容，可以保证集群一直处于资源充足的状态。

本例中启动压测时设置了压力持续时间，因此当压测工具停止打压后，观察负载 Pod 数量。CPU 负载快速下降，工作负载开始缩容，工作负载副本数也快速由 12 缩容至 2 个，最后恢复到 1 个副本。

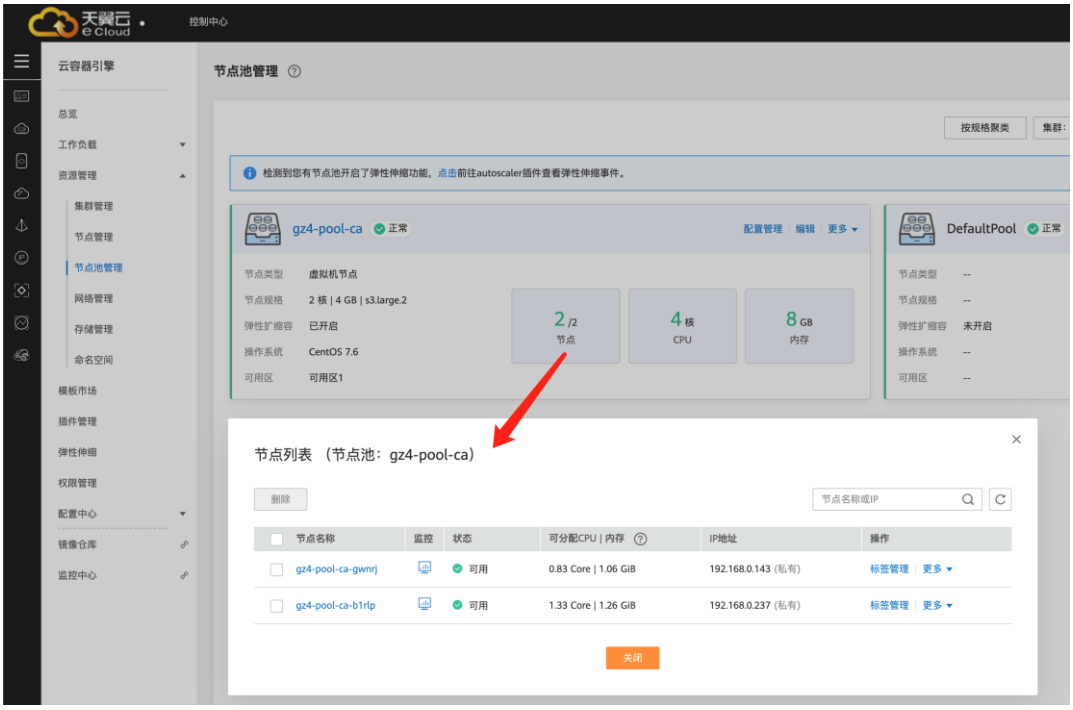
观察负载和 HPA 策略的详情，从事件中可以看到负载的缩容过程和策略生效的时间线，在控制台中同样可以看到 HPA 策略生效历史。




再继续观察，会看到节点池中的节点数量会被不断缩容。




最终，节点池节点数量将稳定在 2。




这里节点没有继续被缩容，是因为节点池中这两个节点都存在 namespace 为 kube-system 的 Pod（且不是 DaemonSets 创建的 Pod）。关于节点在什么情况下不会被缩容请参考 CCE 帮助中心 > 弹性伸缩 > 集群/节点弹性伸缩 > 节点伸缩原理。

如需继续缩容，可编辑节点池，手动减少其节点数量。

## 总结

通过上述的实践可以看到，使用 CCE 的 HPA+CA 机制，可以很容易做到工作负载及节点的弹性伸缩，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。

# 8 监控

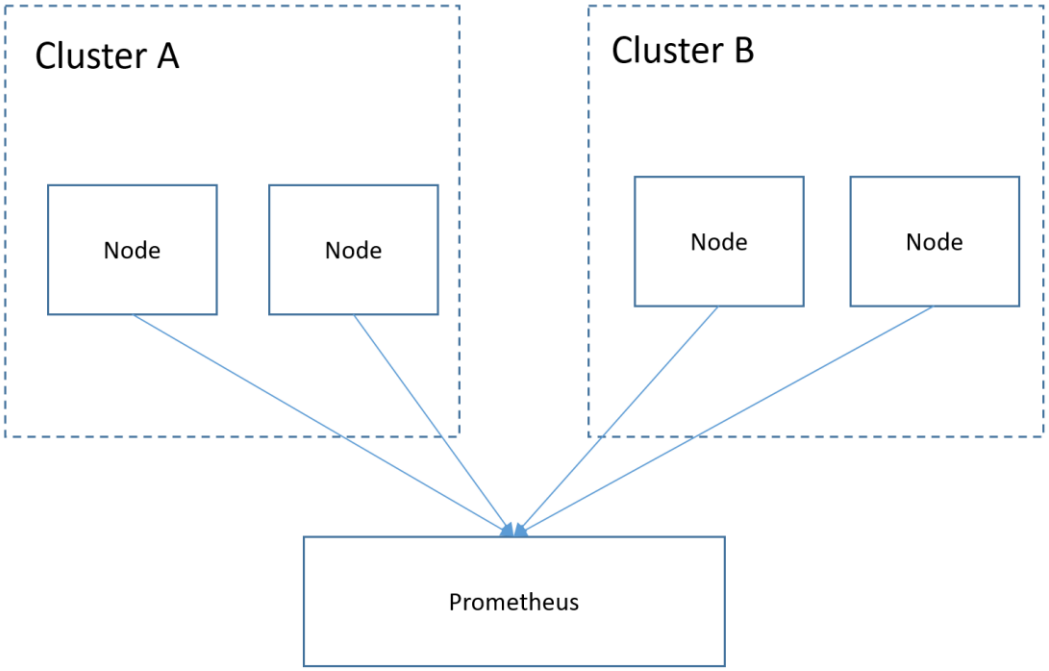
## 8.1 Prometheus 监控多个集群

### 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**

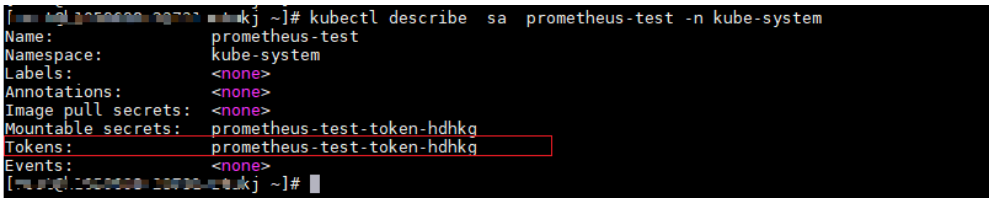
- 1.21 以前版本的集群中，Pod 中获取 Token 的形式是通过挂载 ServiceAccount 的 Secret 来获取 Token，这种方式获得的 Token 是永久的。该方式在 1.21 及以上的版本中不再推荐使用，并且根据社区版本迭代策略，在 1.25 及以上版本的集群中，ServiceAccount 将不会自动创建对应的 Secret。

  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。详情请参见用户指南 > 权限 > ServiceAccount Token 安全性提升说明。

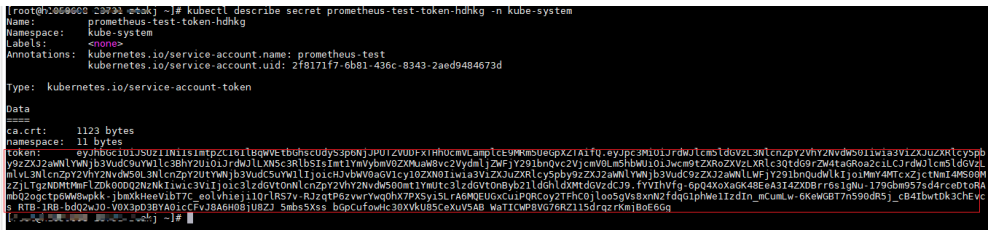
- 如果您在业务中需要一个永不过期的 Token，您也可以选择手动管理 ServiceAccount 的 Secret。尽管存在手动创建永久 ServiceAccount Token 的机制，但还是推荐使用 TokenRequest 的方式使用短期的 Token，以提高安全性。

首先获取 serviceaccount 信息。

**kubectl describe sa prometheus-test -n kube-system**



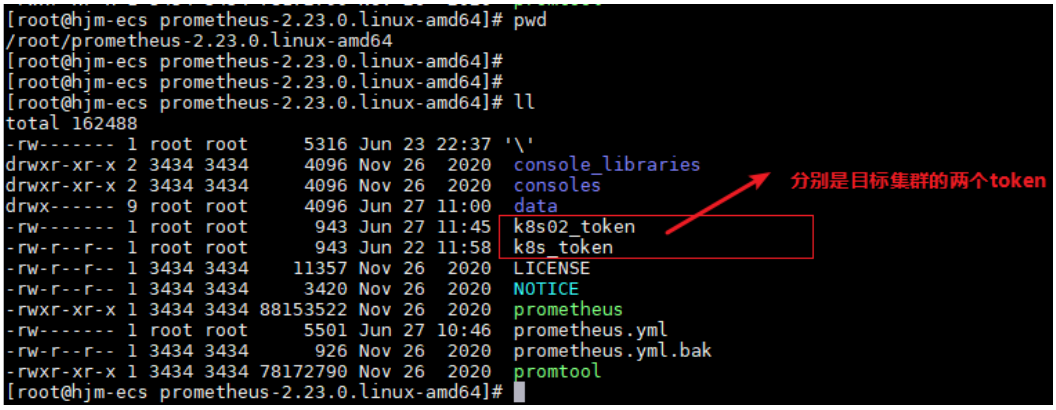

**kubectl describe secret prometheus-test-token-hdhkg -n kube-system**




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
```

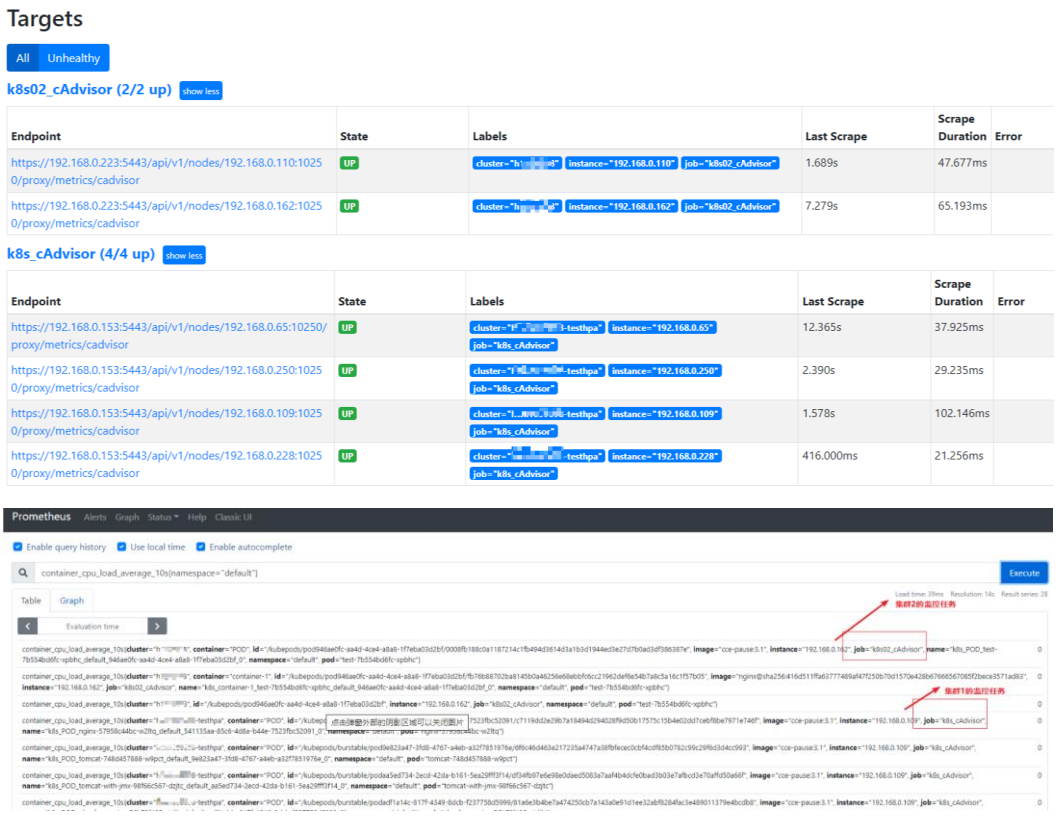
```
    api_server: https://192.168.0.147:5443  #K8s 集群 apiserver 地址
    tls_config:
      insecure_skip_verify: true   #跳过对服务端的认证
  relabel_configs:  ##用于在抓取 metrics 之前修改 target 的已有标签
  - target_label: __address__
    replacement: 192.168.0.147:5443
    action: replace

  - 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**

# 8.2 使用 dcgm-exporter 监控 GPU 指标

## 应用场景

集群中包含 GPU 节点时，需要了解 GPU 应用使用节点 GPU 资源的情况，例如 GPU 利用率、显存使用量、GPU 运行的温度、GPU 的功率等。在获取 GPU 监控指标后，用户可根据应用的 GPU 指标配置弹性伸缩策略，或者根据 GPU 指标设置告警规则。本文基于开源 Prometheus 和 DCGM Exporter 实现丰富的 GPU 观测场景，关于 DCGM Exporter 的更多信息，请参见 DCGM Exporter。

## 前提条件

- 目标集群已创建，且集群中包含 GPU 节点，并已运行 GPU 相关业务。
- 在集群中安装 gpu-device-plugin（原 gpu-beta）和 kube-prometheus-stack 插件。
  - gpu-device-plugin（gpu-beta）插件是在容器中使用 GPU 显卡的设备管理插件，集群中使用 GPU 节点时必须安装该插件。安装 GPU 驱动时，需要匹配 GPU 类型和 CUDA 版本选择对应的驱动进行安装。
  - kube-prometheus-stack 插件负责监控集群相关指标信息，安装时可选择对接 grafana，以便获得更好的观测性体验。


## 采集 GPU 监控指标

本文在集群部署 dcgm-exporter 组件进行 GPU 指标的采集，同时以 9400 端口对外暴露 GPU 指标。

**步骤 1** 登录一台已绑定 EIP 的集群节点。

**步骤 2** 将 dcgm-exporter 镜像拉取到本地。该镜像地址来自 DCGM 官方示例，详情请参见 https://github.com/NVIDIA/dcgm-exporter/blob/main/dcgm-exporter.yaml。

```
docker pull nvcr.io/nvidia/k8s/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04
```

**步骤 3** 上传 dcgm-exporter 镜像到 SWR。

1. （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。

2. 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。
3. 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”。
4. 为 dcgm-exporter 镜像打标签。

**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，2 中登录指令末尾的域名即为镜像仓库地址。
- **[组织名称]**：请替换为 1 中创建的组织。
- **[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 nvcr.io/nvidia/k8s/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04
registry.cn-jssz1.ctyun.cn/container/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04
```

5. 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 registry.cn-jssz1.ctyun.cn/container/dcgm-exporter:3.0.4-3.0.0-
ubuntu20.04
```

终端显示如下信息，表明上传镜像成功。

```
489a396b91d1: Pushed
...
c3f11d77a5de: Pushed
3.0.4-3.0.0-ubuntu20.04: digest: sha256:bd2b1a73025*** size: 2414
```

6. 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。

### 步骤 4 部署核心组件 dcgm-exporter

在 CCE 中部署 dcgm-exporter，需要添加一些特定配置，才可以正常监控 GPU 信息。详细 yaml 如下，其中 yaml 中标红的部分为较为重要的配置项。

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: "dcgm-exporter"
  namespace: "monitoring"      #请根据实际情况选择命名空间安装
  labels:
    app.kubernetes.io/name: "dcgm-exporter"
    app.kubernetes.io/version: "3.0.0"
```

```
spec:
  updateStrategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app.kubernetes.io/name: "dcgm-exporter"
      app.kubernetes.io/version: "3.0.0"
  template:
    metadata:
      labels:
        app.kubernetes.io/name: "dcgm-exporter"
        app.kubernetes.io/version: "3.0.0"
      name: "dcgm-exporter"
    spec:
      containers:
      - image: "registry.cn-jssz1.ctyun.cn/container/dcgm-exporter:3.0.4-3.0.0-
ubuntu20.04"   #dcgm-exporter 的 SWR 镜像地址，该地址为 5 中的镜像地址。
        env:
        - name: "DCGM_EXPORTER_LISTEN"                # 服务端口号
          value: ":9400"
        - name: "DCGM_EXPORTER_KUBERNETES"             # 支持 Kubernetes 指标映射到 Pod
          value: "true"
        - name: "DCGM_EXPORTER_KUBERNETES_GPU_ID_TYPE"   # GPU ID 类型，可选值为 uid 或
device-name
          value: "device-name"
        name: "dcgm-exporter"
        ports:
        - name: "metrics"
          containerPort: 9400
        resources:      #建议根据实际情况配置资源使用申请值和限制值
          limits:
            cpu: '200m'
            memory: '256Mi'
          requests:
            cpu: 100m
            memory: 128Mi
        securityContext:      #需要给 dcgm-exporter 容器开启特权模式
          privileged: true
          runAsNonRoot: false
          runAsUser: 0
        volumeMounts:
        - name: "pod-gpu-resources"
          readOnly: true
          mountPath: "/var/lib/kubelet/pod-resources"
        - name: "nvidia-install-dir-host"      #dcgm-exporter 镜像中配置的环境变量依赖容器
中的/usr/local/nvidia 目录下的文件
          readOnly: true
          mountPath: "/usr/local/nvidia"
      volumes:
      - name: "pod-gpu-resources"
        hostPath:
          path: "/var/lib/kubelet/pod-resources"
      - name: "nvidia-install-dir-host"       #GPU 驱动的安装目录
        hostPath:
          path: "/opt/cloud/cce/nvidia"      #gpu-device-plugin 插件版本为 2.0.0 及以上
```

```
时，该驱动的安装目录需替换为"/usr/local/nvidia"
     affinity:      #CCE 在创建 GPU 节点时生成的标签，部署该监控组件可根据这个标签设置节点亲和。
       nodeAffinity:
         requiredDuringSchedulingIgnoredDuringExecution:
           nodeSelectorTerms:
           - matchExpressions:
             - key: accelerator
               operator: Exists
---
kind: Service
apiVersion: v1
metadata:
  annotations:     #以下注解可以让 prometheus 自动发现并拉取指标
    prometheus.io/port: "9400"
    prometheus.io/scrape: "true"
  name: "dcgm-exporter"
  namespace: "monitoring"      #请根据实际情况选择命名空间安装
  labels:
    app.kubernetes.io/name: "dcgm-exporter"
    app.kubernetes.io/version: "3.0.0"
spec:
  selector:
    app.kubernetes.io/name: "dcgm-exporter"
    app.kubernetes.io/version: "3.0.0"
  ports:
  - name: "metrics"
    port: 9400
```

**步骤 5 监控应用 GPU 指标**

1. 确认 dcgm-exporter 组件正常运行：

```
kubectl get po -n monitoring -owide
```

回显如下：

```
# kubectl get po -n monitoring -owide
NAME                                  READY   STATUS    RESTARTS   AGE   IP
NODE          NOMINATED NODE   READINESS GATES
alertmanager-alertmanager-0              0/2     Pending   0          19m   <none>
<none>        <none>           <none>
custom-metrics-apiserver-5bb67f4b99-grxhq   1/1     Running   0          19m
172.16.0.6    192.168.0.73   <none>           <none>
dcgm-exporter-hkr77                      1/1     Running   0          17m
172.16.0.11   192.168.0.73   <none>           <none>
grafana-785cdcd47-9jlgr                  1/1     Running   0          19m
172.16.0.9    192.168.0.73   <none>           <none>
kube-state-metrics-647b6585b8-6l2zm      1/1     Running   0          19m
172.16.0.8    192.168.0.73   <none>           <none>
node-exporter-xvk82                      1/1     Running   0          19m
192.168.0.73  192.168.0.73   <none>           <none>
prometheus-operator-5ff8744d5f-mhbqv     1/1     Running   0          19m
172.16.0.7    192.168.0.73   <none>           <none>
prometheus-server-0                      2/2     Running   0          19m
172.16.0.10   192.168.0.73   <none>           <none>
```

2. 调用 dcgm-exporter 接口，验证采集的应用 GPU 信息。

其中 172.16.0.11 为 dcgm-exporter 组件的 Pod IP。

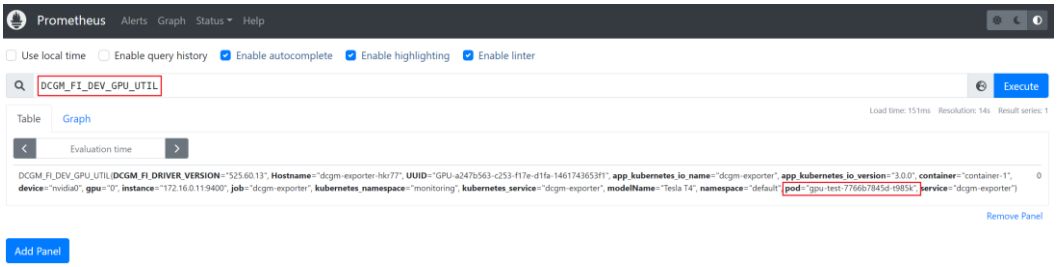
```
curl 172.16.0.11:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
```

```
[root@ ~]# curl 172.16.0.11:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  5514    0  5514    0     0  5384k      0 --:--:-- --:--:-- --:--:-- 5384k
# HELP DCGM_FI_DEV_GPU_UTIL GPU utilization (in %).
# TYPE DCGM_FI_DEV_GPU_UTIL gauge
DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-a247b563-c253-f17e-d1fa-1461743653f1",device="nvidia0",modelName="Tesla T4",Hostname="dcgm-exporter-hkr77",DCGM_FI_DRIVER_VERSION="525.60.13",container="container-1",namespace="default",pod="gpu-test-7766b7845d-t985k"} 0
```

**步骤 6 Prometheus 页面查看指标监控信息。**

安装完 Prometheus 相关插件后，Prometheus 默认会创建 ClusterIP 类型的服务，如果需要对外暴露，需要将 Prometheus 发布为外部访问（NodePort 类型或 LoadBalancer 类型）。

如下图，可以看到 GPU 节点上的 GPU 利用率以及其他相关指标，更多 GPU 指标请参见可观测指标。




**步骤 7 登录 Grafana 页面查看 GPU 信息**

如您在安装 kube-prometheus-stack 插件时选择安装了 Grafana，您可通过导入 NVIDIA DCGM Exporter Dashboard 来展示 gpu 的相关指标信息。

关于在 Grafana 导入 Dashboard 的方法，请参见 Manage dashboards。


**----结束**

## 可观测指标

以下是一些常用的 GPU 观测指标，更多指标详情请参见 Field Identifiers。

表8-1 利用率

| 指标名称 | 指标类型 | 单位 | 说明 |
| --- | --- | --- | --- |
| DCGM_FI_DEV_GPU_UTIL | Gauge | % | GPU 利用率 |
| DCGM_FI_DEV_MEM_COPY_UTIL | Gauge | % | 内存利用率 |
| DCGM_FI_DEV_ENC_UTIL | Gauge | % | 编码器利用率 |
| DCGM_FI_DEV_DEC_UTIL | Gauge | % | 解码器利用率 |

表8-2 内存指标

| 指标名称 | 指标类型 | 单位 | 说明 |
| --- | --- | --- | --- |
| DCGM_FI_DEV_FB_FREE | Gauge | MB | 表示帧缓存剩余数，帧缓存一般被称为显存 |
| DCGM_FI_DEV_FB_USED | Gauge | MB | 表示帧缓存已使用数，该值与 nvidia-smi 命令中 memory-usage 的已使用值对应 |

表8-3 温度及功率指标

| 指标名称 | 指标类型 | 单位 | 说明 |
| --- | --- | --- | --- |
| DCGM_FI_DEV_GPU_TEMP | Gauge | 摄氏度 | 设备的当前 GPU 温度读数 |
| DCGM_FI_DEV_POWER_USAGE | Gauge | W | 设备的电源使用情况 |

# 9 集群

## 9.1 集群选型

当您使用云容器引擎 CCE 创建 Kubernetes 集群时，常常会面对多种配置选项以及不同的名词，难以进行选择。本文将从不同的关键配置进行对比并给出选型建议，帮助您创建一个满足业务需求的集群。

### 集群类型

云容器引擎支持 CCE Turbo 集群与普通 CCE 集群两种类型，以满足您各种业务需求，如下为 CCE Turbo 集群与 CCE 集群区别：

表9-1 集群类型对比

| 维度 | 子维度 | CCE 集群 | CCE Turbo 集群 |
| --- | --- | --- | --- |
| 集群 | 定位 | 标准版本集群，提供商用级的容器集群服务 | 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 |
| | 节点形态 | 支持虚拟机和裸金属服务器混合 | 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 1.0**：面向性能和规模要求不高的场景。<br>• 容器隧道网络模式<br>• VPC 网络模式 | **云原生网络 2.0**：面向大规模和高性能的场景。<br>组网规模最大支持 2000 节点。 |
| | 网络性能 | VPC 网络叠加容器网络，性能有一定损耗 | VPC 网络和容器网络融合，性能无损耗 |
| | 容器网络隔离 | • 隧道网络模式：集群内部网络隔离策略，支持 Networkpolicy。<br>• VPC 网络模式：不支持 | 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。 |

| 维度 | 子维度 | CCE 集群 | CCE Turbo 集群 |
| --- | --- | --- | --- |
| 安全 | 隔离性 | 普通容器，Cgroups 隔离 | • 物理机：安全容器，支持虚机级别的隔离<br>• 虚拟机：普通容器，Cgroups 隔离 |

## 集群版本

由于 Kubernetes 社区版本迭代较快，新版本中通常包含许多 Bug 修复和新功能，而旧版本会根据时间推移逐渐淘汰。建议您在创建集群时，选择当前 CCE 支持的最新稳定版本。

关于 CCE 集群版本的更新策略，请参考 集群 kubernetes 版本发布说明。

## 集群网络模型

云容器引擎支持以下几种网络模型，您可根据实际业务需求进行选择。

**须知**

**集群创建成功后，网络模型不可更改，请谨慎选择。**

表9-2 网络类型对比

| 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。<br>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。<br>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。<br>容器与虚机 IP 互通，使用了微服务注册框架的，如 Dubbo、CSE 等。 |
| 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<br>节点维度划分地址 | 容器网段单独分配<br>节点维度划分地址段， | 容器网段从 VPC 子网划分，无需单独分配 |

| 对比维度 | 容器隧道网络 | VPC 网络 | 云原生网络 2.0 |
| --- | --- | --- | --- |
| | 段，动态分配（地址段分配后可动态增加） | 静态分配（节点创建完成后，地址段分配即固定，不可更改） | |
| 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。<br><br>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（包括默认路由表和自定义路由表），因此集群本身规模受 VPC 路由表上限限制。路由表配额请参见使用限制。 | 最大可支持 2000 节点 |

## 集群网段

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。

如业务需求复杂，如多个集群使用同一 VPC、集群跨 VPC 互联等场景，需要同步规划 VPC 的数量、子网的数量、容器网段划分和服务网段连通方式，详情请参见集群网络地址段规划实践。

## 服务转发模式

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链接的场景。

对稳定性要求极高且 Service 数量小于 2000 时，建议选择 iptables，其余场景建议首选 IPVS。

## 节点规格

使用云容器引擎时，集群节点最小规格要求为 CPU ≥ 2 核且内存 ≥ 4GB，但使用很多小规格 ECS 并非是最优选择，需要根据业务需求合理评估。使用过多的小规格节点会存在以下弊端：

- 小规格节点的网络资源的上限较小，可能存在单点瓶颈。
- 当容器申请的资源较大时，一个小规格节点上无法运行多个容器，节点剩余资源就无法利用，存在资源浪费的情况。

使用大规格节点的优势：

- 网络带宽上限较大，对于大带宽类的应用，资源利用率高。
- 多个容器可以运行在同一节点，容器间通信延迟低，减少网络传输。
- 拉取镜像的效率更高。因为镜像只需要拉取一次就可以被节点上的多个容器使用。而对于小规格的 ECS 拉取镜像的次数就会增多，在节点弹性伸缩时则需要花费更多的时间，反而达不到立即响应的目的。

另外，还需要根据业务需求选择合适的 CPU/内存配比。例如，使用内存较大但 CPU 较少的容器业务，建议选择 CPU/内存配比为 1:4 的节点，减少资源浪费。

## 节点容器引擎

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。并且 Kubernetes 在 v1.24 版本中移除了 Dockershim，并从此不再默认支持 Docker 容器引擎，详情请参见 Kubernetes 即将移除 Dockershim，CCE v1.27 版本中也将不再支持 Docker 容器引擎。

因此，在一般场景使用时建议选择 Containerd 容器引擎。但在以下场景中，仅支持使用 Docker 容器引擎：

- Docker in Docker（通常在 CI 场景）。
- 节点上使用 Docker 命令。
- 调用 Docker API。

## 节点操作系统

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。

# 9.2 通过 CCE 搭建 IPv4/IPv6 双栈集群

本教程将指引您搭建一个 IPv6 网段的 VPC，并在 VPC 中创建一个带有 IPv6 地址的集群和节点，使节点可以访问 Internet 上的 IPv6 服务。

## 简介

IPv6 的使用，可以有效弥补 IPv4 网络地址资源有限的问题。如果当前集群中的工作节点（如 ECS）使用 IPv4，那么启用 IPv6 后，工作节点可在双栈模式下运行，即工作节点可以拥有两个不同版本的 IP 地址：IPv4 地址和 IPv6 地址，这两个 IP 地址都可以进行内网/公网访问。

## 使用场景

- 如果您的应用需要为使用 IPv6 终端的用户提供访问服务，则您可使用：IPv6 弹性公网 IP 或 IPv6 双栈。
- 如果您的应用既需要为使用 IPv6 终端的用户提供访问服务，又需要对这些访问来源进行数据分析处理，则您必须使用 IPv6 双栈。
- 如果您的应用系统与其他系统（例如：数据库系统）、应用系统之间需要使用 IPv6 进行内网访问，则您必须使用 IPv6 双栈。

## 约束与限制

- 支持双栈的集群：

| 集群类型 | 集群网络模型 | 支持的集群版本 | 其他说明 |
| --- | --- | --- | --- |
| CCE 集群 | 容器隧道网络 | v1.15 及以上 | v1.23 版本 GA（Generally Available）<br>暂不支持 ELB 使用双栈能力 |
| CCE Turbo 集群 | 云原生网络 2.0 | v1.23.8-r0 及以上<br>v1.25.3-r0 及以上 | 暂不支持创建 kata 安全容器<br>仅支持弹性云服务器-虚拟机或弹性云服务器-物理机（机型为 c6.22xlarge.4.physical 或 c7.32xlarge.4.physical） |

- Kubernetes 内部 Node 和 Master 之间通信使用 IPv4 地址。
- Service 类型选择“负载均衡（LoadBalancer）”或“DNAT 网关（DNAT）”时，仅支持对接 IPv4。

- 同一个网卡上，只能绑定一个 IPv6 地址。
- 集群开启 IPv4/IPv6 双栈时，所选节点子网不允许开启 DHCP 无限租约。
- 使用双栈集群时，请勿在 ELB 控制台修改 ELB 的协议版本。
- ELB 使用双栈仅支持 CCE Turbo 集群，且存在以下约束：

| 使用场景 | 独享型 ELB | 共享型 ELB |
| --- | --- | --- |
| ELB 型 Ingress | 支持 ELB 使用双栈。<br>后端服务器不支持使用 IPv6 协议，仅支持 IPv4 协议。如您使用 IPv6 协议，将产生相关告警事件，请前往对应 Ingress 的“事件”查看。 | 仅支持 IPv4 协议。 |
| Nginx 型 Ingress | 不支持使用双栈。 | 不支持使用双栈。 |
| LoadBalancer 类型的 Service | • 七层：不支持使用双栈。<br>• 四层：支持使用双栈。 | 仅支持 IPv4 协议。 |

## 步骤 1：创建虚拟私有云和子网

在创建 VPC 之前，您需要根据具体的业务需求规划 VPC 的数量、子网的数量和 IP 网段划分等。

**说明**

- IPv4/IPv6 双栈网络的基本操作与之前的 IPv4 网络相同。只有部分页面的配置参数会略有差异，具体请以管理控制台显示为准。
- 如需了解 IPv6 收费策略、支持的 ECS 类型及支持的区域等信息，请参见相关产品帮助中心。

请按如下操作，创建一个 VPC“vpc-ipv6”和一个 IPv6 默认子网“subnet-ipv6”。

3. 登录管理控制台。
4. 在管理控制台左上角单击 ，选择区域和项目。
5. 选择“网络>虚拟私有云 VPC”。
6. 单击“创建虚拟私有云”。
7. 根据界面提示配置虚拟私有云和子网参数。

   子网配置时，请务必勾选“开启 IPv6”，将自动为子网分配 IPv6 网段。该功能一旦开启，将不能关闭。暂不支持自定义设置 IPv6 网段。

表9-3 虚拟私有云参数说明

| 参数 | 说明 | 取值样例 |
| --- | --- | --- |
| 区域 | 不同区域的资源之间内网不互通。 | 苏州 |

| 参数 | 说明 | 取值样例 |
| --- | --- | --- |
| | 请选择靠近您客户的区域，可以降低网络时延、提高访问速度。 | |
| 名称 | VPC 名称。 | vpc-ipv6 |
| IPv4 网段 | VPC 的地址范围，VPC 内的子网地址必须在 VPC 的地址范围内。<br>目前支持网段范围：<br>10.0.0.0/8~24<br>172.16.0.0/12~24<br>192.168.0.0/16~24 | 192.168.0.0/16 |
| 企业项目 | 创建 VPC 时，可以将 VPC 加入已启用的企业项目。<br>企业项目管理提供了一种按企业项目管理云资源的方式，帮助您实现以企业项目为基本单元的资源及人员的统一管理，默认项目为default。 | default |
| 标签（高级配置） | 虚拟私有云的标示，包括键和值。可以为虚拟私有云创建 10 个标签。 | • 键：vpc_key1<br>• 值：vpc-01 |

表9-4 子网参数说明

| 参数 | 说明 | 取值样例 |
| --- | --- | --- |
| 可用区 | 可用区是指在同一地域内，电力和网络互相独立的物理区域。在同一 VPC 网络内可用区与可用区之间内网互通，可用区之间能做到物理隔离。 | 可用区 2 |
| 名称 | 子网的名称。 | subnet-ipv6 |
| 子网 IPv4 网段 | 子网的 IPv4 地址范围，需要在 VPC 的地址范围内。 | 192.168.0.0/24 |
| 子网 IPv6 网段 | 勾选“开启 IPv6”，将自动为子网分配 IPv6 网段。该功能一旦开启，将不能关闭。暂不支持自定义设置 IPv6 网段。 | - |
| 关联路由表 | 子网创建完成后默认关联默认路由表，您也可以通过子网的更换路由表操作，切换至自定义路由表。 | 默认 |

| 参数 | 说明 | 取值样例 |
| --- | --- | --- |
| 高级配置 | | |
| 网关 | 子网的网关。<br>通向其他子网的 IP 地址，用于实现与其他子网的通信。 | 192.168.0.1 |
| DNS 服务器地址 | 默认配置了 2 个 DNS 服务器地址，您可以根据需要修改。多个 IP 地址以英文逗号隔开。 | 100.125.x.x |
| 标签 | 子网的标示，包括键和值。可以为子网创建 10 个标签。<br>标签的命名规则请参见表 9-6。 | • 键：subnet_key1<br>• 值：subnet-01 |

表9-5 虚拟私有云标签命名规则

| 参数 | 规则 | 样例 |
| --- | --- | --- |
| 键 | • 不能为空。<br>• 对于同一虚拟私有云键值唯一。<br>• 长度不超过 36 个字符。<br>• 由英文字母、数字、下划线、中划线、中文字符组成。 | vpc_key1 |
| 值 | • 长度不超过 43 个字符。<br>• 由英文字母、数字、下划线、点、中划线、中文字符组成。 | vpc-01 |

表9-6 子网标签命名规则

| 参数 | 规则 | 样例 |
| --- | --- | --- |
| 键 | • 不能为空。<br>• 对于同一子网键值唯一。<br>• 长度不超过 36 个字符。<br>• 由英文字母、数字、下划线、中划线、中文字符组成。 | subnet_key1 |
| 值 | • 长度不超过 43 个字符。<br>• 由英文字母、数字、下划线、点、中划线、中文字符组成。 | subnet-01 |

8. 单击“立即创建”。

## 步骤 2：创建集群

### 创建 CCE 集群场景

1. 登录 CCE 控制台，创建一个 CCE 集群。

网络配置请按如下设置，其余配置可参考购买 CCE 集群：

- 网络模型：选择“容器隧道网络”。
- 虚拟私有云：选择已创建的“vpc-ipv6”。
- 控制节点子网：请务必选择已开启了 IPv6 的子网。
- IPv6 双栈：选择开启，开启后将支持通过 IPv6 地址段访问集群资源，包括节点，工作负载等。
- 容器网段：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。

图9-1 网络配置


2. 创建节点。

CCE 控制台会过滤出支持 IPv6 的机型，可直接选择。创建节点时的配置详情可参考创建节点。

创建完成后，您可以进入集群，单击节点名称进入 ECS 详情页查看自动分配的 IPv6 地址。

### 创建 CCE Turbo 集群场景

3. 登录 CCE 控制台，创建一个 CCE Turbo 集群。

网络配置请按如下设置，其余配置可参考购买 CCE 集群：

- 网络模型：选择“云原生网络 2.0”。
- IPv6 双栈：选择开启，开启后将支持通过 IPv6 地址段访问集群资源，包括节点，工作负载等。
- 虚拟私有云：选择已创建的“vpc-ipv6”。
- 控制节点子网：仅支持选择已开启了 IPv6 的子网。

- 默认容器子网：仅支持选择已开启了 IPv6 的子网。
- IPv4 服务网段：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。
- IPv6 服务网段：该网段决定了支持 IPv6 地址的 Service 资源的上限，创建后不可修改，默认为 fc00::/112。如需自定义该网段，需要满足以下要求：
  - IPv6 服务网段需属于 fc00::/8 网段内。
  - IPv6 地址前缀长度范围为 112-120，您可以通过调整前缀数值，调整地址个数，地址数最多可支持 65536 个。

图9-2 网络配置


4. 创建节点。

CCE 控制台会过滤出支持 IPv6 的机型，可直接选择。创建节点时的配置详情可参考创建节点。

创建完成后，您可以进入集群，单击节点名称进入 ECS 详情页查看自动分配的 IPv6 地址。

## 步骤 3：购买和加入共享带宽

默认 IPv6 地址只具备私网通信能力，如果您需要通过该 IPv6 地址访问 Internet 或被 Internet 上的 IPv6 客户端访问，您需要购买和绑定共享带宽。

如您已有共享带宽，可以不用重新购买，直接将 IPv6 地址加入共享带宽即可。

**购买共享带宽**

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. 在系统首页，选择“网络 > 虚拟私有云 VPC”。
4. 在左侧导航栏，选择“弹性公网 IP 和带宽 > 共享带宽”。

5. 在页面右上角，单击“购买共享带宽”，按照提示配置参数。

表9-7 参数说明

| 参数 | 说明 | 取值样例 |
| --- | --- | --- |
| 计费模式 | 购买共享带宽时使用的计费模式，分为以下两种：<br>• 包年/包月：在使用前一次性支付一定期限（如 1 个月、1 年等）的费用，后续使用期限内不再针对此共享带宽资源扣费。<br>• 按需计费：按照共享带宽的使用时长进行计费。 | 包年/包月 |
| 区域 | 不同区域的资源之间内网不互通。请选择靠近您客户的区域，可以降低网络时延、提高访问速度。 | 苏州 |
| 计费方式 | 共享带宽的计费方式。 | 按带宽计费 |
| 带宽大小 | 共享带宽的大小，单位 Mbit/s，5M 起售。 | 10 |
| 带宽名称 | 共享带宽的名称。 | Bandwidth-001 |
| 企业项目 | 申请共享带宽时，可以将共享带宽加入已启用的企业项目。<br>企业项目管理提供了一种按企业项目管理云资源的方式，帮助您实现以企业项目为基本单元的资源及人员的统一管理，默认项目为 default。 | default |
| 购买时长 | 包年包月场景需要选择，购买共享带宽的时长。 | 2 个月 |

6. 单击“立即购买”。

**加入共享带宽**

7. 在共享带宽列表页，单击操作列的“添加公网 IP”。

图9-3 加入共享带宽入口


8. 将 IPv6 地址加入共享带宽。
9. 单击“确定”。

**结果验证**

登录到 ECS 实例，ping 一个公网上的 IPv6 服务，验证连通性。例如：ping6 ipv6.baidu.com，执行结果如图 9-4 所示。

图9-4 结果验证

```
root@ecs-tang:~# ping6 ipv6.baidu.com
PING ipv6.baidu.com(2400:da00:2::29) 56 data bytes
64 bytes from 2400:da00:2::29: icmp_seq=1 ttl=42 time=45.6 ms
64 bytes from 2400:da00:2::29: icmp_seq=2 ttl=42 time=45.1 ms
64 bytes from 2400:da00:2::29: icmp_seq=3 ttl=42 time=44.8 ms
64 bytes from 2400:da00:2::29: icmp_seq=4 ttl=42 time=45.1 ms
```

# 9.3 创建节点注入脚本最佳实践

## 应用现状

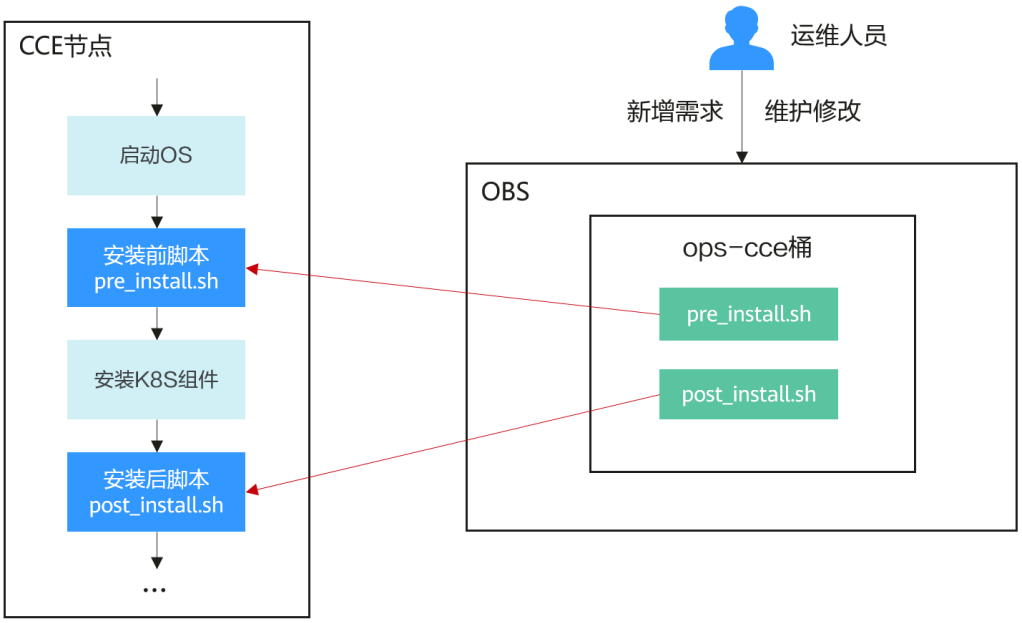
对于需要在节点上提前安装一些工具或者做用户自定义的安全加固等操作时，需要在创建节点的时候注入一些脚本。CCE 创建节点提供了 Kubernetes 安装前和安装后两处注入脚本的功能。但是使用通常碰到如下限制：

- 注入脚本的字符有限。
- 各种需求、场景的多样性可能会经常修改注入的脚本内容，而对于 CCE 节点池的注入脚本是固定的，不适合经常修改。

## 解决方案

本文提供 CCE+OBS 结合的方式，提供了一种简化、可扩展、易维护的最佳实践方式，方便用户在 CCE 节点上做一些自定义操作。

将安装前和安装后脚本存放在 OBS 中，在创建节点池的时候，安装前和安装后注入脚本直接拉取 OBS 对应脚本的地址并执行既可。对于 CCE 节点池这处的配置基本来就可以不用变化了，后期如果有新的需求只需要更新 OBS 的脚本内容即可。




## OBS 桶维护建议

- 若无单独用于运维的 OBS 的桶，建议单独创建一个专用于运维的桶，方便后续整体运维组使用。
- 建议在桶中新建多级目录 tools/cce，表示工具集合中的 cce 部分下，方便维护，后续还可以放其他的工具脚本。

## 注意事项

- 脚本实现的自定义操作如果失败，会影响正常业务运行，建议在脚本最后添加检查程序。若检查失败，可以在安装后脚本中将 kubelet 进程停止掉，避免业务调度到该节点上。

  systemctl stop kubelet-monitor

  systemctl stop kubelet
- 脚本中尽量不要放敏感信息，避免泄露。

## 操作步骤

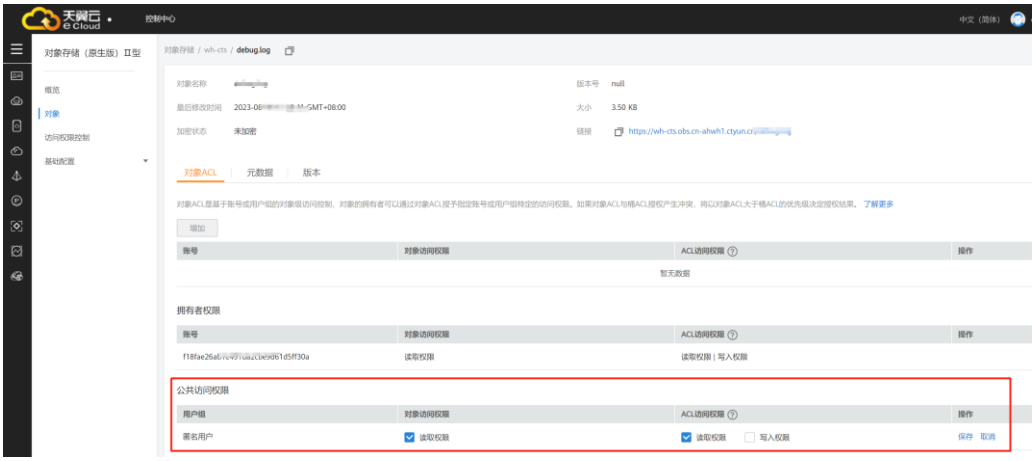
**步骤 1** 创建 OBS 桶。

**步骤 2** 上传安装前和安装后脚本。以 pre_install.sh 和 post_install.sh 为例。

图9-5 上传脚本


步骤 3 为脚本配置只读安全策略，保证 CCE 节点可以免密下载的的同时，外网不可以下载。

图9-6 配置对象策略




步骤 4 在 CCE 创建节点池时配置安装前执行脚本和安装后执行脚本。

在创建节点池的云服务器高级配置中填写如下命令。

安装前执行脚本

```
curl -H "User-Agent: ccePrePostInstall" https://ops-cce.obs.cn-jssz1.ctyun.cn/tools/cce/pre_install.sh -o  /tmp/pre_install.sh && bash -x /tmp/pre_install.sh > /tmp/pre_install.log 2>&1
```

安装后执行脚本

```
curl -H "User-Agent: ccePrePostInstall" https://ops-cce.obs.cn-jssz1.ctyun.cn/tools/cce/post_install.sh -o  /tmp/post_install.sh && bash -x /tmp/post_install.sh > /tmp/post_install.log 2>&1
```

如下命令是先使用 curl 命令从 OBS 中下载 pre_install.sh 和 post_install.sh 到/tmp 目录，然后执行 pre_install.sh 和 post_install.sh。

安装前执行脚本：

```
curl -H "User-Agent: ccePrePostInstall" https://ops-cce.obs.cn-jssz1.ctyun.cn/tools/cce/pre_install.sh -o  /tmp/pre_install.sh && bash -x /tmp/pre_install.sh > /tmp/pre_install.log 2>&1
```

安装后执行脚本：

```
curl -H "User-Agent: ccePrePostInstall" https://ops-cce.obs.cn-
jssz1.ctyun.cn/tools/cce/post_install.sh -o  /tmp/post_install.sh && bash -x
/tmp/post_install.sh > /tmp/post_install.log 2>&1
```

**说明**

- User-Agent 的实际取值根据 OBS 桶策略中配置
- 链接中的桶和地址均需要按实际链接地址配置

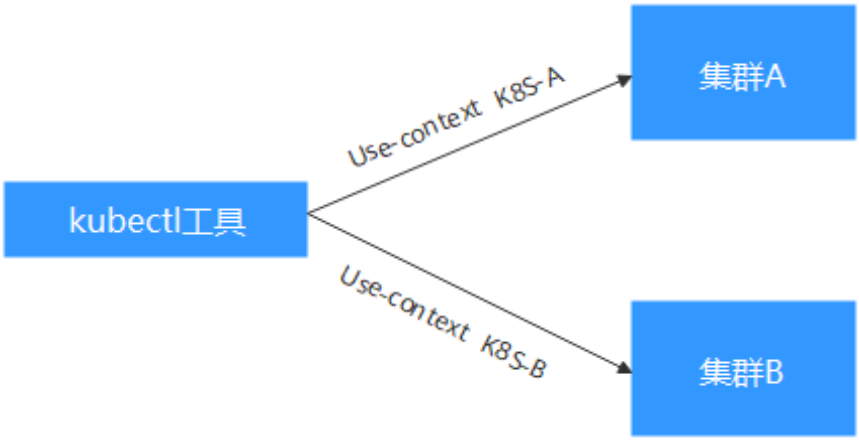
**----结束**

# 9.4 通过 kubectl 对接多个集群

## 使用场景

客户想使用同一个 kubectl 客户端工具，通过切换当前用户的方式达到切换集群的效果。

图9-7 kubectl 对接多集群示意




## 配置前提

- kubectl 命令客户端工具所在的 ECS 云主机，能够 curl 通集群 A 和集群 B 的 vip 地址+5443 端口。
- 以下配置仅供参考，为了方便，直接在 A 集群的一个节点作为客户端（节点绑定了 EIP）。
- 为了方便访问，给 B 集群的 VIP 地址绑定了公网 IP（假设为：1.2.3.4），理论上如果 A、B 在同一 VPC，可以不这样操作。

## 客户端配置对 A 的访问

直接参考《CCE 用户指南》中的“通过 kubectl 连接 CCE 集群”。

## 客户端上配置对 B 的访问

我们要用访问 B 的集群，要知道 B 的地址，还要带上认证去访问，那么必须要把相应的信息补齐。

步骤如下：

**步骤 1** 录入 B 集群的集群信息

```
kubectl config set-cluster cluster-k8s --server=https://1.2.3.4:5443  --insecure-skip-tls-verify=true
```

--insecure-skip-tls-verify=true 这个参数一定要带上，这是忽略校验客户端证书

**步骤 2** 录入去 B 集群的认证信息

首先在 B 上操作：

方式一：将 B 集群的证书传入到客服端使用，在客户端上操作，用户详细信息添加到配置文件中：

```
kubectl config set-credentials ui-admin --client-certificate=client.crt --embed-certs=true
```

方式二：在集群 B 上获取认证 token

1. 创建一个 sa 用户

```
kubectl create  sa  my-sa
```

2. 给 sa 用户授权

```
kubectl create clusterrolebinding  myrolebinding --serviceaccount=default:my-sa --clusterrole=cluster-admin
```

3. 获取用户的 token

```
kubectl describe secret my-sa-token-xxx | awk '/token:/{print $2}' > token
```

将获取的 token 传入到客户端。

4. 客户端上操作，用户详细信息添加到配置文件中：

```
kubectl config set-credentials ui-admin --token=$token
```

**步骤 3** 将集群 B 的上下文详细信息添加到配置文件中

```
kubectl config set-context ui-admin@cyd --cluster=cluster-k8s --user=ui-admin
```

此时在客户端上：

Kubectl config use-context  internal  切换到集群 A

Kubectl config use-context  ui-admin@cyd  切换到集群 B

**步骤 4** 验证配置


**其他情况：**

如果配置完报 X509 错误，应该是忘记传入参数 --insecure-skip-tls-verify=true

可在使用时带上该参数即可，例如：kubectl get pod --insecure-skip-tls-verify=true

**----结束**

# 9.5 给 CCE 集群的节点添加第二块数据盘

您可以使用“安装前执行脚本”功能来配置 CCE 集群节点（弹性云服务器 ECS）。

**说明**

- 1.13.10 及更高版本的集群创建节点时，若未开启 LVM 管理的数据盘，请参考本章节的介绍填写安装前执行脚本进行格式化，否则该数据盘仍会被 LVM 管理。
- 1.13.10 之前版本的集群创建节点时，若未开启 LVM 管理的数据盘请务必格式化，否则会与第一块数据盘进行二选一被 LVM 管理，进而导致与预期不符的情况。

在使用“安装前执行脚本”功能前，请预先编写一个可以格式化数据盘的脚本（**该脚本需以 root 用户执行**）。

**输入参数：**

1. 设定该脚本名字为 formatdisk.sh，将 formatdisk.sh 保存到您的 OBS 中，获取该文件在 OBS 中的地址。
2. 需要指定 docker 数据盘（使用 LVM 管理的数据盘称之为 docker 数据盘）的大小，且 docker 盘不能与第二块盘大小一致。例如：100G 的 docker 盘，110G 的第二块数据盘。
3. 第二块数据盘的挂载路径。例如：/data/code。

在“安装前执行脚本”中执行以下命令来实现格式化能力：

```
cd /tmp;curl -k -X GET OBS 的地址/formatdisk.sh -1 -O;fdisk -l;sleep 30;bash -x formatdisk.sh 100 /data/code;fdisk -l
```

**formatdisk.sh 脚本示例如下：**

```
dockerdisksize=$1
mountdir=$2
systemdisksize=40
i=0
while [ 20 -gt $i ]; do
```

```
   echo $i;
   if [ $(lsblk -o KNAME,TYPE | grep disk | grep -v nvme | awk '{print $1}' | awk
'{ print "/dev/"$1}' |wc -l) -ge 3 ]; then
      break
   else
      sleep 5
   fi;
   i=$[i+1]
done
all_devices=$(lsblk -o KNAME,TYPE | grep disk | grep -v nvme | awk '{print $1}' |
awk '{ print "/dev/"$1}')
for device in ${all_devices[@]}; do
   isRawDisk=$(lsblk -n $device 2>/dev/null | grep disk | wc -l)
   if [[ ${isRawDisk} > 0 ]]; then
      # is it partitioned ?
      match=$(lsblk -n $device 2>/dev/null | grep -v disk | wc -l)
      if [[ ${match} > 0 ]]; then
         # already partited
         [[ -n "${DOCKER_BLOCK_DEVICES}" ]] && echo "Raw disk ${device} has been
partition, will skip this device"
         continue
      fi
   else
      isPart=$(lsblk -n $device 2>/dev/null | grep part | wc -l)
      if [[ ${isPart} -ne 1 ]]; then
         # not parted
         [[ -n "${DOCKER_BLOCK_DEVICES}" ]] && echo "Disk ${device} has not been
partition, will skip this device"
         continue
      fi
      # is used ?
      match=$(lsblk -n $device 2>/dev/null | grep -v part | wc -l)
      if [[ ${match} > 0 ]]; then
         # already used
         [[ -n "${DOCKER_BLOCK_DEVICES}" ]] && echo "Disk ${device} has been used,
will skip this device"
         continue
      fi
      isMount=$(lsblk -n -o MOUNTPOINT $device 2>/dev/null)
      if [[ -n ${isMount} ]]; then
         # already used
         [[ -n "${DOCKER_BLOCK_DEVICES}" ]] && echo "Disk ${device} has been used,
will skip this device"
         continue
      fi
      isLvm=$(sfdisk -lqL 2>>/dev/null | grep $device | grep "8e.*Linux LVM")
      if [[ ! -n ${isLvm} ]]; then
         # part system type is not Linux LVM
         [[ -n "${DOCKER_BLOCK_DEVICES}" ]] && echo "Disk ${device} system type is
not Linux LVM, will skip this device"
         continue
      fi
   fi
   block_devices_size=$(lsblk -n -o SIZE $device 2>/dev/null | awk '{ print $1}')
   if [[ ${block_devices_size}"x" != "${dockerdisksize}Gx" ]] &&
```

```
[[ ${block_devices_size}"x" != "${systemdisksize}Gx" ]]; then
echo "n
p
1



w
" | fdisk $device
       mkfs -t ext4 ${device}1
       mkdir -p $mountdir
    uuid=$(blkid ${device}1 |awk '{print $2}')
    echo "${uuid}  $mountdir ext4  noatime  0 0" | tee -a /etc/fstab >/dev/null
       mount $mountdir
   fi
done
```

**说明**

如果直接拷贝上方示例不能正常执行，请使用 dos2unix 工具进行格式转换。

# 9.6 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，您可手动调整该数据盘容量，最小支持下调至 20G，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**

您可以执行以下命令清理未使用的 Docker 镜像：

```
docker system prune -a
```

**说明**

该命令会把暂时没有用到的 Docker 镜像都删除，执行命令前请确认。

**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            # docker 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**

# 9.7 快速清理已删除节点上的 CCE 组件

## 使用场景

若集群中包含包周期节点或纳管节点，删除对应集群和节点不会删除对应的 ECS，此时请按照界面提示清理节点上 CCE 组件。若未按照提示清理节点组件，后续需要清理 ECS 时，可按照如下操作进行清理。

**须知**

**卸载将会删除弹性云主机上的 CCE 系统用户 paas 以及 docker 相关资源，执行清理操作前，若有关键数据需要保留，请提前备份。**

## 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 > 节点管理”，在右侧的节点列表中找到要执行操作的纳管节点，在节点的“操作”区域下单击“更多 > 移除”。

**步骤 2** 在弹出的“移除纳管节点”对话框中输入"REMOVE"确认移除节点，单击“确认”。

**步骤 3** 认真阅读弹出的“提示”页面内容，按照步骤清理 CCE 相关资源。

**----结束**

# 10 网络

## 10.1 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

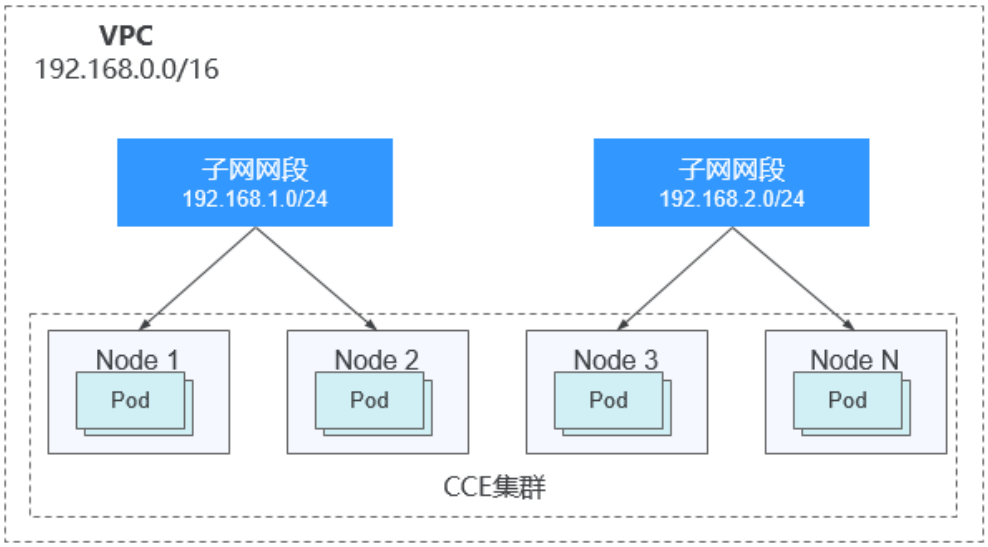
- **VPC 网段**

  虚拟私有云（Virtual Private Cloud，简称 VPC）是您申请的为云服务器、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10-1 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。

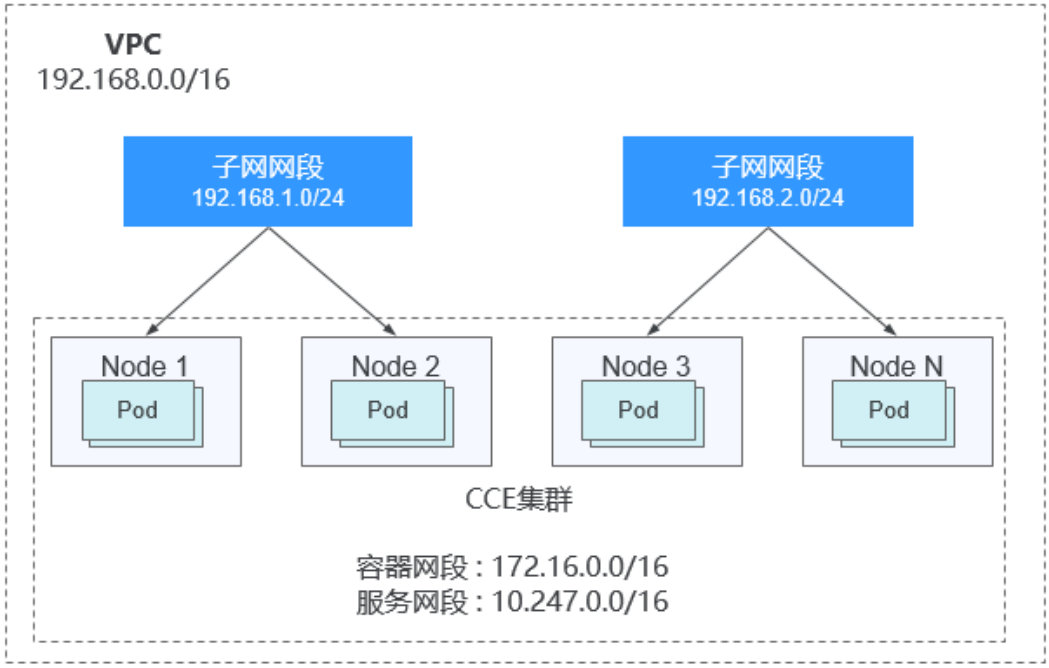
## 单 VPC+单集群场景

**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 10-2 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。

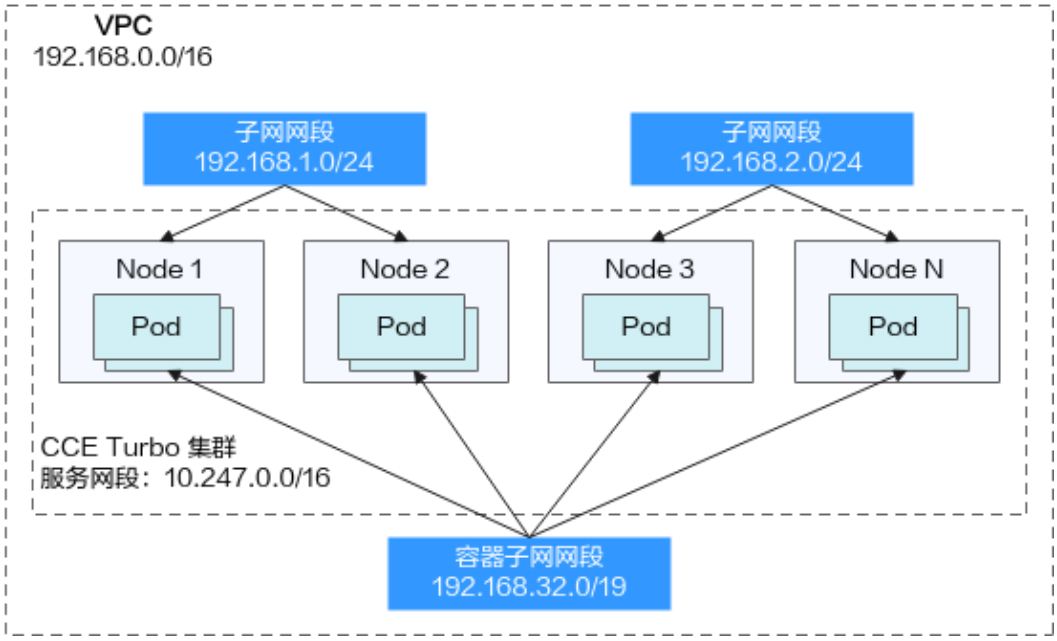
图10-2 单 VPC 单集群场景网段规划-CCE 集群




**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下图所示。

- 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10-3 单 VPC 单集群场景网段规划-CCE Turbo 集群



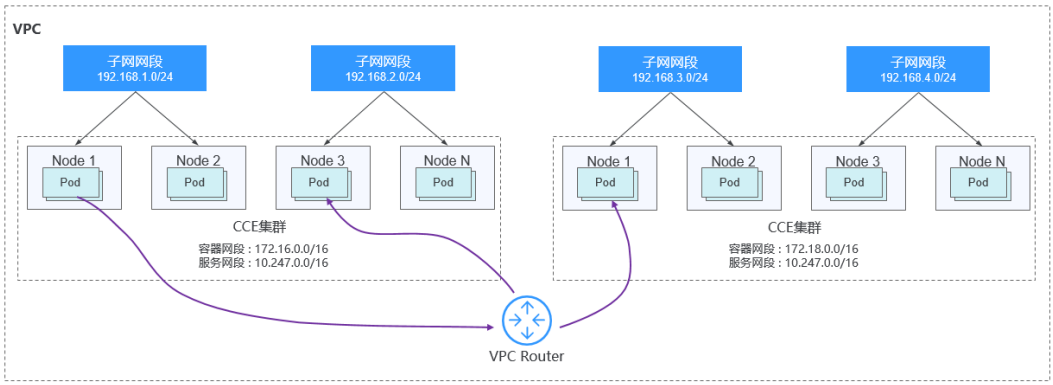

## 单 VPC+多集群场景

**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下图所示。

- 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10-4 VPC 网络-多集群场景示例

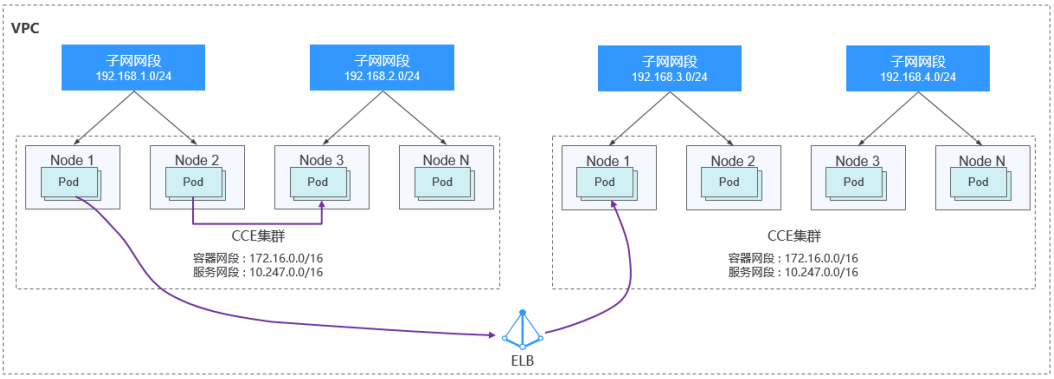



### 容器隧道网络

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下图所示。

- 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10-5 容器隧道网络-多集群场景示例

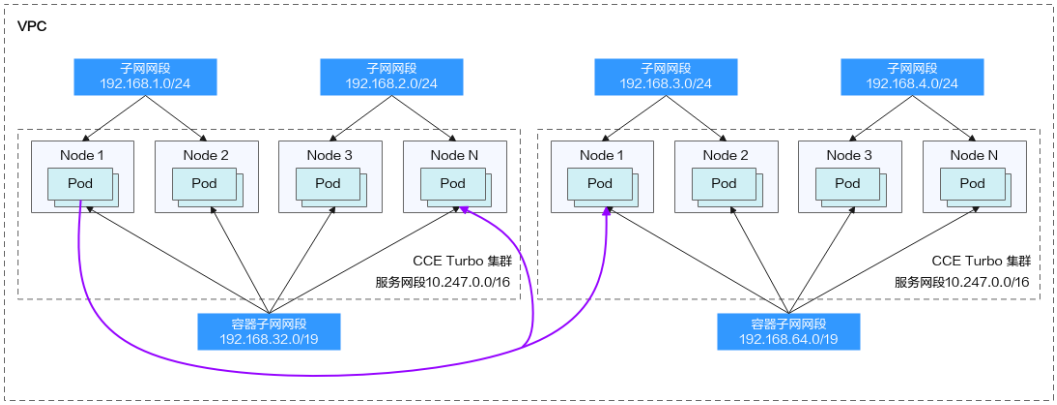



### 云原生网络 2.0 模式（即 CCE Turbo 集群）

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10-6 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：

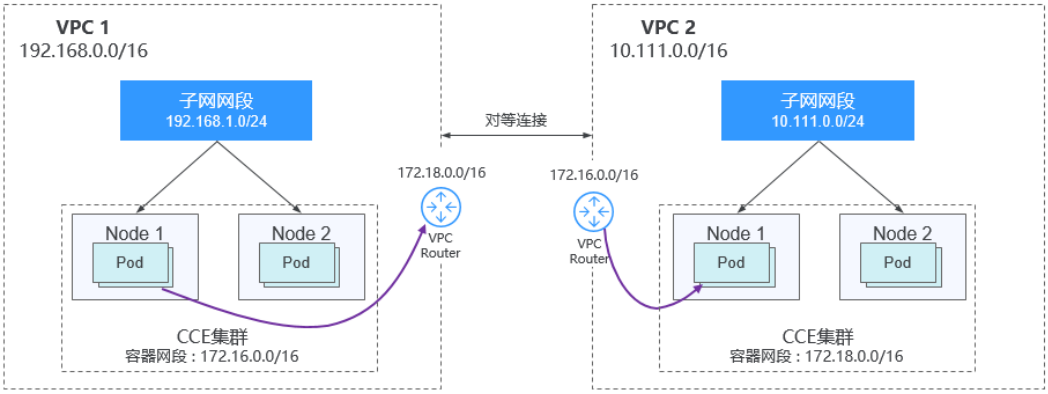
- VPC 网段：该场景下各个集群所在的 VPC 网段相同，请保证 VPC 内可用的 IP 地址数充足。
- 子网网段：子网网段尽量避免和容器网段重叠。即使在某些场景下（例如和 CCE Turbo 集群共存时），子网网段可以和容器（子网）网段重叠，但从地址段规划角度出发，这是不推荐的。
- 容器网段：仅 **VPC 网络模式**的集群间的容器网段需要避免相互重叠。
- 服务网段：所有集群之间服务网段可以重叠，但是不能和所属集群的子网网段和容器网段重叠。

## 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10-7 VPC 网络-VPC 互联场景



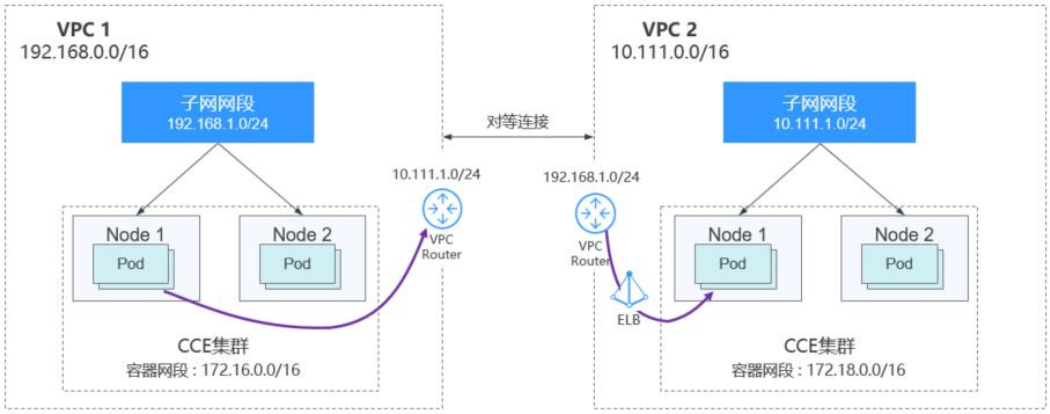

跨 VPC 的集群容器之间互联需要建立 VPC 对等连接时，需要注意如下几点：

- 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。

图10-8 在本端路由中添加对端容器网段的地址


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。

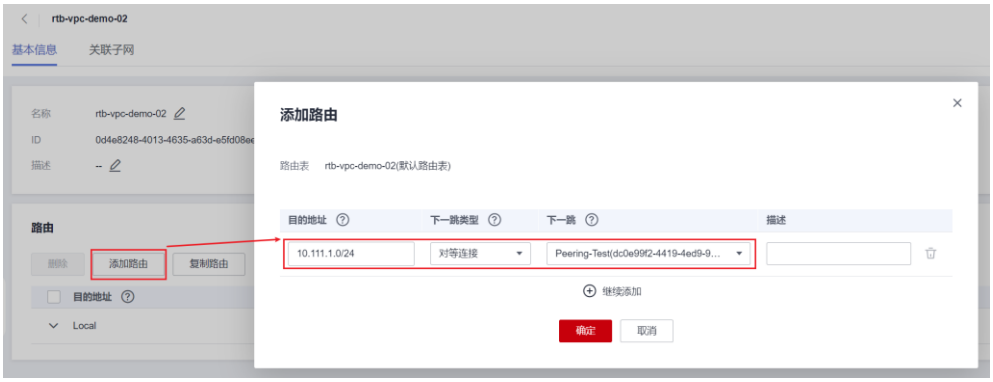
图10-9 容器隧道网络-VPC 互联场景




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。

图10-10 在本端路由中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。

# 10.2 集群网络模型选择及各模型区别

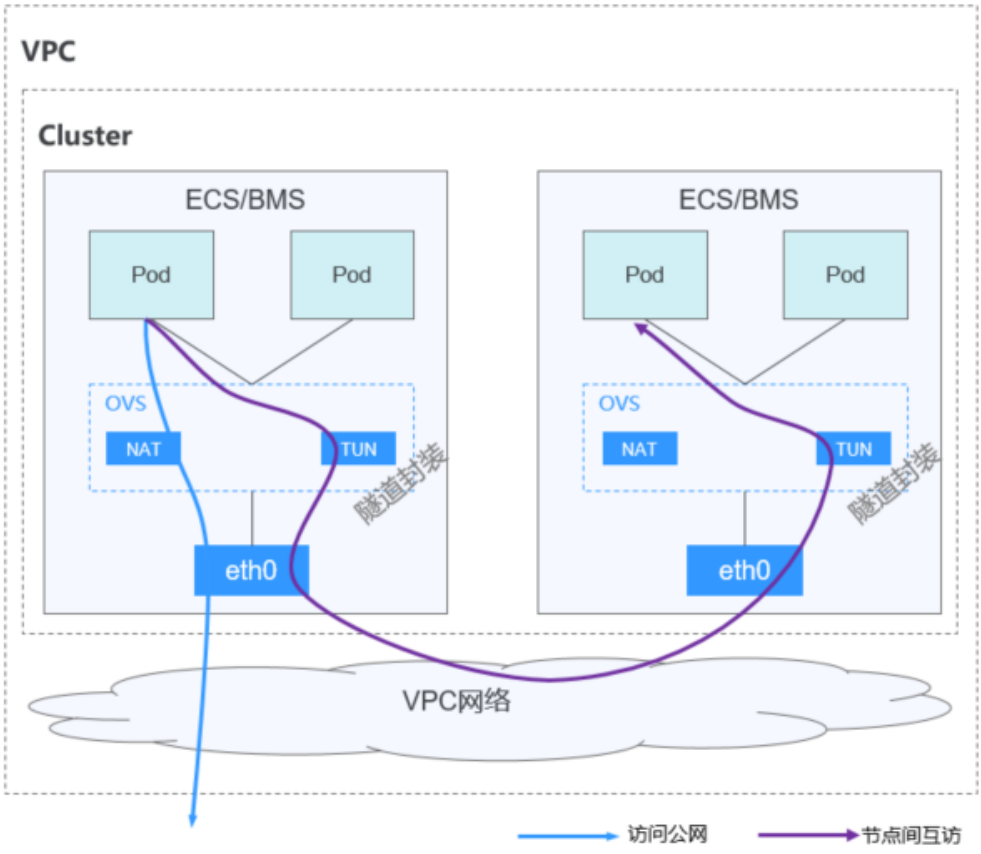
CCE 的容器网络插件，支持容器隧道网络、VPC 网络、云原生网络 2.0 网络模型：

**⚠ 注意**

集群创建成功后，网络模型不可更改，请谨慎选择。

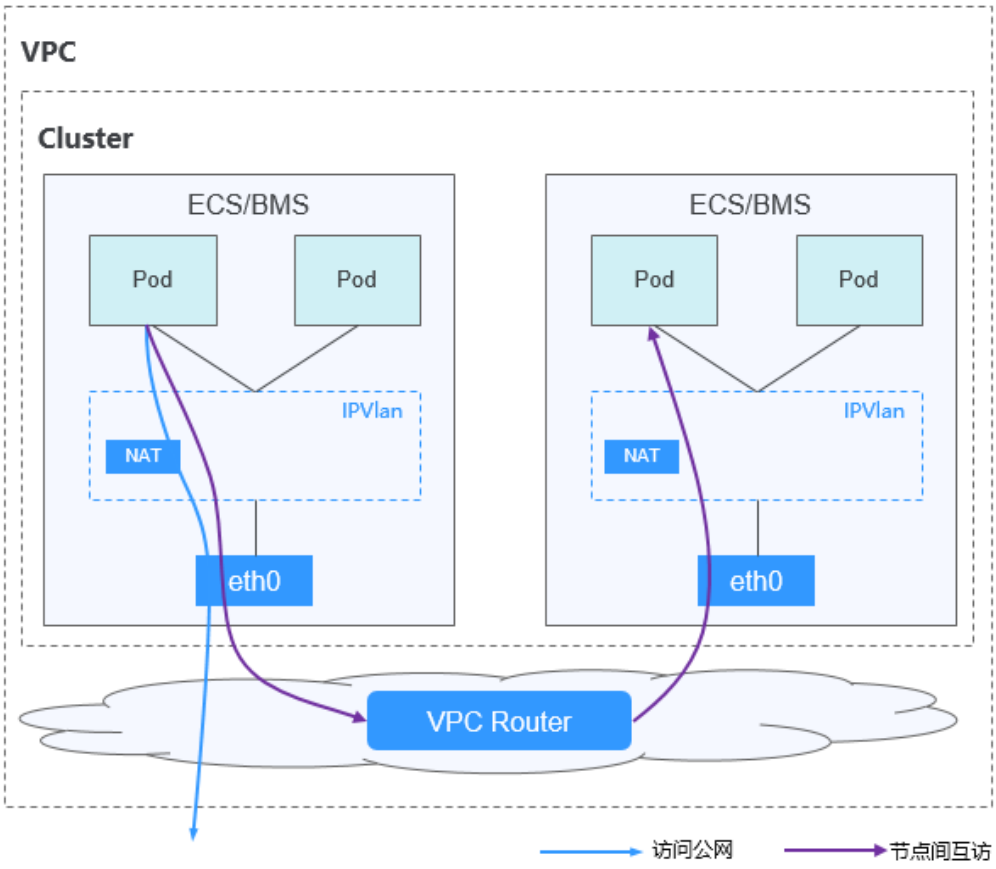
- **容器隧道网络（Overlay）：**基于底层 VPC 网络构建了独立的 VXLAN 隧道化容器网络，适用于一般场景。VXLAN 是将以太网报文封装成 UDP 报文进行隧道传输。容器网络是承载于 VPC 网络之上的 Overlay 网络平面，具有付出少量隧道封装性能损耗，即可获得通用性强、互通性强、高级特性支持全面（例如 Network Policy 网络隔离）的优势，可以满足大多数应用需求。

  图10-11 容器隧道网络



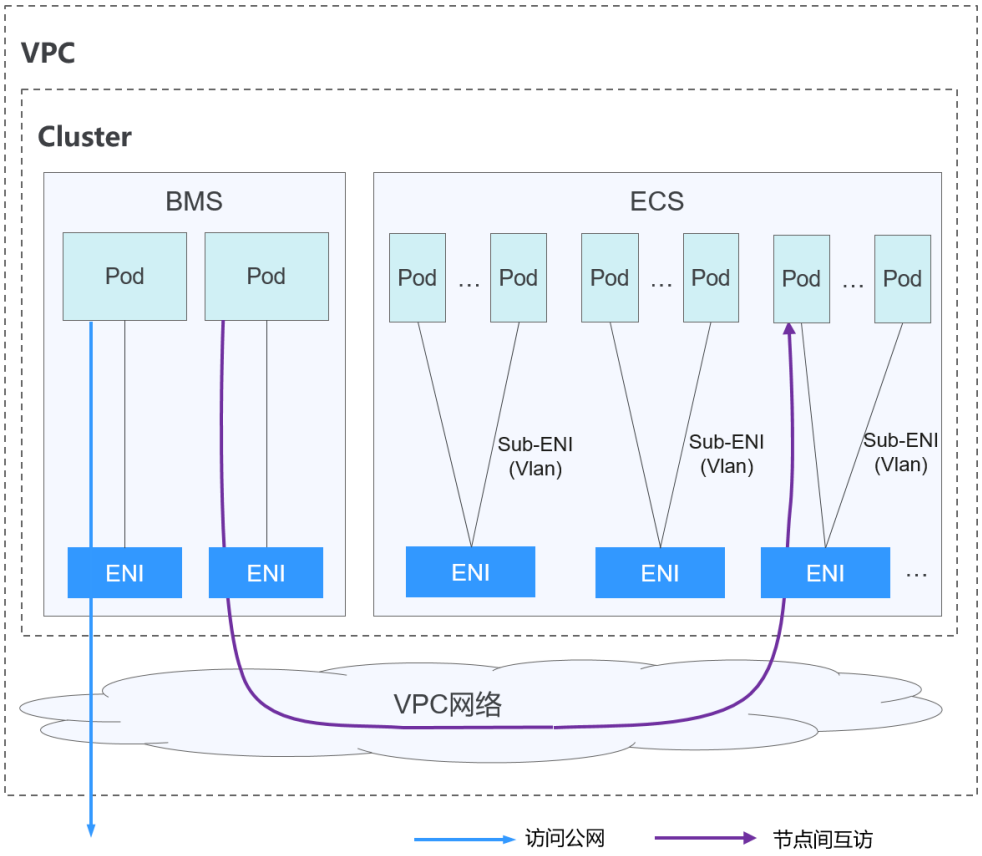

- **VPC 网络：**采用 VPC 路由方式与底层网络深度整合，适用于高性能场景，节点数量受限于虚拟私有云 VPC 的路由配额。每个节点将会被分配固定大小的 IP 地址段。VPC 网络由于没有隧道封装的消耗，容器网络性能相对于容器隧道网络有一定优势。VPC 网络集群由于 VPC 路由中配置有容器网段与节点 IP 的路由，可以支持集群外直接访问容器实例等特殊场景。

图10-12 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10-13 云原生网络 2.0




网络模型对比如下：

表10-1 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 |
| --- | --- | --- | --- |
| 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<br>• 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<br>• 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 |

| 对比维度 | 容器隧道网络 | VPC 网络 | 云原生网络 2.0 |
| --- | --- | --- | --- |
| 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 | 默认支持 200 节点，受限于 VPC 路由表能力。<br>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。<br>• 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。<br>• 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。<br>• 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。

# 10.3 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）。

**说明**

CCE Turbo 集群无需设置该参数。

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   # CCE Turbo 集群无需设置该参数
  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
...
```

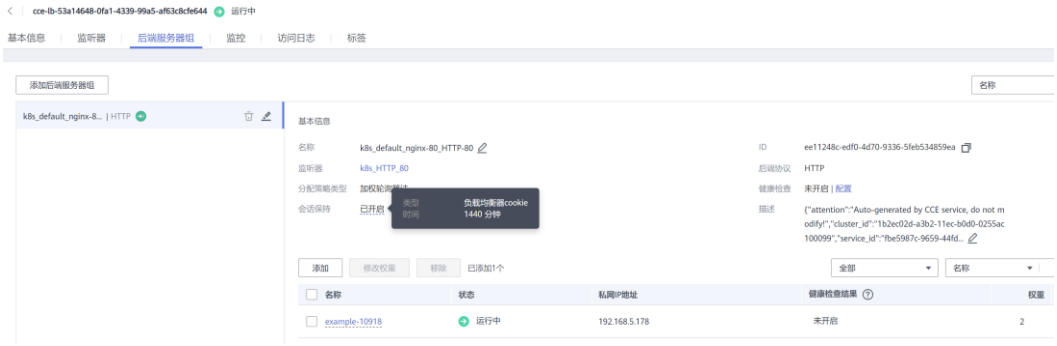
**步骤 3** 创建 Ingress，关联 Service。以下示例以自动创建共享型 ELB 为例，如需指定其他类型的 ELB，可通过 Kubectl 命令行添加 ELB Ingress。

```
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。

**图10-14 开启会话保持**



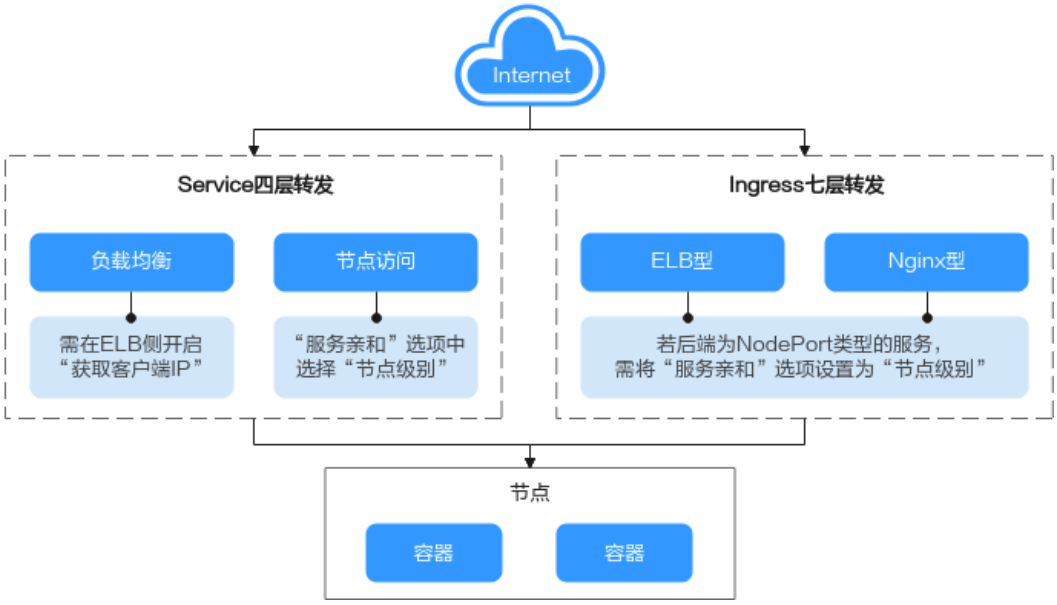

**----结束**

# 10.4 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：**

- 负载均衡：ELB 访问方式，是通过弹性负载均衡 ELB 产品来实现负载均衡。共享型负载均衡四层监听器（TCP/UDP）支持开启“获取客户端 IP”功能。而独享型负载均衡的四层监听器（TCP/UDP）默认开启源地址透传功能，无需手动开启。
- 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。

## 支持获取源 IP 地址的场景

由于网络模型差异，CCE 在部分使用场景下暂不支持获取源 IP，若在集群使用过程中需要获取源 IP，请尽量避免此类场景。表中“-”代表无此场景。

表10-2 支持获取源 IP 地址的场景分类

| 一级分类 | 二级分类 | 负载均衡类型 | VPC、容器隧道网络模型 | 云原生网络 2.0 模型 | 操作指导 |
| --- | --- | --- | --- | --- | --- |

| 一级分类 | 二级分类 | 负载均衡类型 | VPC、容器隧道网络模型 | 云原生网络 2.0 模型 | 操作指导 |
| --- | --- | --- | --- | --- | --- |
| 七层转发（Ingress） | ELB 型 | 共享型 | 支持 | 支持 | 七层转发（Ingress） |
| | | 独享型 | 支持 | 支持 | |
| | Nginx 型（对接 nginx-ingress 插件） | 共享型 | 支持 | 暂不支持 | |
| | | 独享型 | 支持 | 支持 | |
| 四层转发（Service） | 负载均衡（LoadBalancer） | 共享型 | 支持 | 暂不支持（使用 hostNetwork 的工作负载支持） | 负载均衡( LoadBalancer ) |
| | | 独享型 | 支持 | 支持 | |
| | 节点访问（NodePort） | - | 支持 | 暂不支持（使用 hostNetwork 的工作负载支持） | 节点访问( NodePort ) |

## 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**

- 云原生网络 2.0 模型下，**Ingress 开启对接 nginx-ingress 插件时使用共享型 ELB 暂不支持获取源 IP**，参见支持获取源 IP 地址的场景。如需获取源 IP，请卸载 nginx-ingress 插件并在重新安装时使用独享型 ELB。
- 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）。

## 负载均衡( LoadBalancer )

负载均衡( LoadBalancer )的 Service 模式下，不同类型的集群获取源 IP 的场景不一，部分场景下暂不支持获取源 IP，参见支持获取源 IP 地址的场景。

- CCE 集群（VPC、容器隧道网络模型）：使用共享型和独享型 ELB 均支持获取源 IP。
- CCE Turbo 集群（云原生网络 2.0 模型）：使用独享型 ELB 时支持获取源 IP；使用共享型 ELB 时，仅开启 hostNetwork 的工作负载支持获取源 IP。

**VPC、容器隧道网络模型**

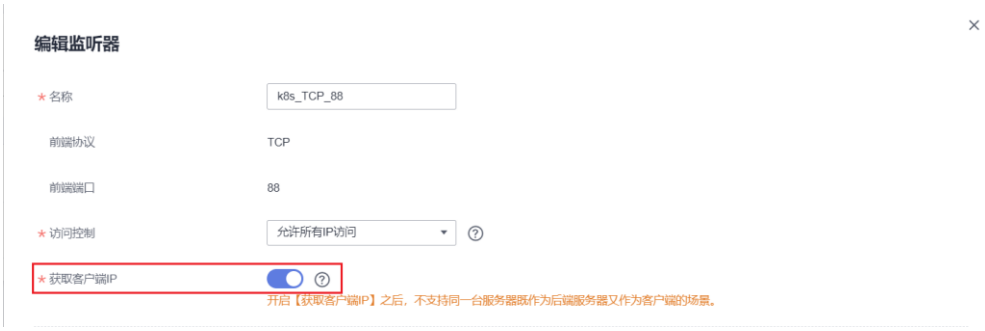
开启获取源 IP 的步骤如下：

**步骤 1** 在 CCE 控制台创建负载均衡类型的 Service，服务亲和选择“**节点级别**”而不是“集群级别”。


**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。
   - 新增场景：单击“添加监听器”。
   - 修改场景：单击需要修改的监听器名称右侧的“编辑”按钮。
6. 开启“获取客户端 IP”开关。

图10-15 开启开关

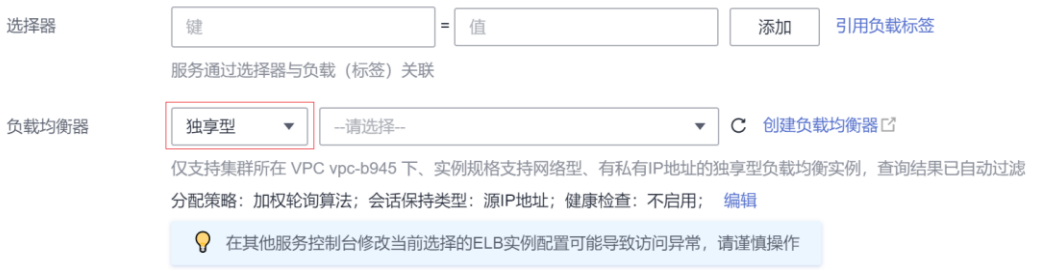


----结束

### 云原生网络 2.0 模型

云原生网络 2.0 模型下，使用共享型 ELB 创建负载均衡时服务亲和无法设置服务亲和选项为“节点级别”，因此无法获取源 IP。如需获取源 IP，必须使用**独享型 ELB**，外部访问可不经过转发直通容器。

独享型 ELB 默认开启源地址透传，无需前往 ELB 控制台手动开启“获取客户端 IP”开关，只需要在 CCE 控制台创建 ENI 负载均衡时选择独享型负载均衡，即可获取客户端源 IP。




## 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**。

**说明**

云原生网络 2.0 模型集群中使用节点访问（NodePort）类型的 Service 时，仅开启 hostNetwork 的工作负载支持获取源 IP。

图10-16 服务亲和选择节点级别

# 10.5 用户在 CCE 集群的节点上使用多网卡的配置指导

## 使用场景

用户在 CCE 集群的节点上使用多个网卡时，可以通过如下方法配置弹性网卡。

## 配置方法

集群版本为 **v1.15.11 和 v1.17.9** 时，CCE 网络管理组件使用 **Network**。

下面以 CentOS 7 系列虚拟机为例，介绍如何配置虚拟机网络。具体步骤如下：

**步骤 1** 登录虚拟机操作系统。

**步骤 2** 执行 ifconfig 命令查看当前虚拟机绑定的弹性网卡。

假设查询的网卡名称为：eth1，编辑“/etc/sysconfig/network-scripts/ifcfg-eth1”。

```
vim /etc/sysconfig/network-scripts/ifcfg-eth1
```

按以下格式编辑：

```
DEVICE="eth1"
BOOTPROTO="dhcp"
ONBOOT="yes"
TYPE="Ethernet"
PERSISTENT_DHCLIENT="yes"
```

**步骤 3** 执行如下命令重启虚拟机网络，使网络配置生效。

```
systemctl restart network
```

**----结束**

当集群版本为 **v1.17.11 及以上新版本**时，CCE 网络管理组件使用 **NetworkManager**，客户在 CCE 的节点配置弹性网卡服务时，因网络模式不同会有如下三种不同的配置场景：

- **overlay_l2 网络模式：**可以自动配置业务弹性网卡，如：自动获取 IP、续租 IP 等，不需要单独配置弹性网卡。
- **vpc 网络模式：**除主网卡、容器绑定的弹性网卡外，客户需要单独配置弹性网卡。
- **云原生网络 2.0 网络模式：**除主网卡、容器绑定的弹性网卡外，客户需要单独配置弹性网卡。

**说明**

云原生网络 2.0 网络模式的集群（CCE Turbo 集群），容器网络承载于弹性网卡/辅助弹性网卡，若进行此类配置请提工单修改集群预留网卡配置。

**具体配置步骤如下：**

下面以 CentOS 7 系列虚拟机为例，介绍如何配置虚拟机网络。具体步骤如下：

**步骤 1** 登录虚拟机操作系统。

**步骤 2** 执行 ifconfig 命令查看当前虚拟机绑定的弹性网卡。

假设查询的网卡名称为：eth1，编辑“/etc/sysconfig/network-scripts/ifcfg-eth1”。

```
vim /etc/sysconfig/network-scripts/ifcfg-eth1
```

按以下格式编辑：

```
DEVICE="eth1"
BOOTPROTO="dhcp"
ONBOOT="yes"
TYPE="Ethernet"
PERSISTENT_DHCLIENT="yes"
```

**步骤 3** 执行如下命令重启虚拟机网络，使网络配置生效。

```
systemctl restart NetworkManager
```

**步骤 4** 查看节点 eth1 网卡的 dhclient 进程已经启动，说明配置生效。

```
[root@jpjp-179-xingneng-56869 ~]# ps -ef | grep dhc
root      8783  8760  0 19:58 ?        00:00:00 /sbin/dhclient -d -q -sf /usr/libexec/nm-dhcp-helper -pf /var/run/dhclient-eth0.pid -lf /var/lib/NetworkMa
nager/dhclient-5fb06bd0-0bb0-7ffb-45f1-d6edd65f3e03-eth0.lease -cf /var/lib/NetworkManager/dhclient-eth0.conf eth0
root      8785  8760  0 19:58 ?        00:00:00 /sbin/dhclient -d -q -sf /usr/libexec/nm-dhcp-helper -pf /var/run/dhclient-eth1.pid -lf /var/lib/NetworkMa
nager/dhclient-9c92fad9-6ecb-3e6c-eb4d-8a47c6f50c04-eth1.lease -cf /var/lib/NetworkManager/dhclient-eth1.conf eth1
root      8809  2447  0 19:58 pts/1    00:00:00 grep --color=auto dhc
```

**----结束**

# 10.6 CCE Turbo 配置容器网卡动态预热

在云原生网络 2.0 下，每个 Pod 都会分配（申请并绑定）一张弹性网卡或辅助弹性网卡（统一称为：容器网卡）。由于容器场景下 Pod 的极速弹性与慢速的容器网卡创建绑定的差异，严重影响了大规模批创场景下的容器启动速度。因此，云原生 2.0 网络提供了容器网卡动态预热的能力，在尽可能提高 IP 的资源利用率的前提下，尽可能加快 Pod 的启动速度。

## 约束与限制

- CCE Turbo 的 1.19.16-r4、1.21.7-r0、1.23.5-r0、1.25.1-r0 及以上版本支持用户配置容器网卡动态预热；支持集群级别的全局配置以及节点池级别的差异化配置，暂不支持非节点池下的节点差异化配置。
- CCE Turbo 的 1.19.16-r2、1.21.5-r0、1.23.3-r0 到 1.19.16-r4、1.21.7-r0、1.23.5-r0 之间的集群版本只支持节点最少绑定容器网卡数(nic-minimum-target)和节点动态预热容器网卡数(nic-warm-target)两个参数配置，且不支持节点池级别的差异化配置。
- 请通过 console 页面或 API 修改容器网卡动态预热参数配置，请勿直接后台修改节点 annotations 上对应的容器网卡动态预热参数，集群升级后，后台直接修改的 annotations 会被覆盖为原始的值。
- CCE Turbo 的 1.19.16-r4、1.21.7-r0、1.23.5-r0、1.25.1-r0 之前的集群版本支持用户配置容器网卡高低水位预热，如果用户配置了全局的容器网卡高低水位预热。集群升级后，原始的高低水位预热参数配置会自动转换为容器网卡动态预热参数配置；但如果用户要通过 console 页面进一步修改容器网卡动态预热参数，需要先通过**集群的配置管理 console 页面**把原始的高低水位预热配置修改为（0:0）。

- CCE Turbo 的节点池 BMS 裸机场景下，1.19.16-r4、1.21.7-r0、1.23.5-r0、1.25.1-r0 之前的集群版本默认采用的是容器网卡高低水位预热（默认值 0.3:0.6）。集群升级后，原始的高低水位预热依然生效，建议客户通过**节点池的配置管理 console 页面**把高低水位预热参数配置转换为容器网卡动态预热参数配置并一并删除高低水位预热配置，以启用最新的容器网卡动态预热的能力。
- CCE Turbo 的非节点池下 BMS 裸机场景下，1.19.16-r4、1.21.7-r0、1.23.5-r0、1.25.1-r0 之前的集群版本默认采用的是容器网卡高低水位预热（默认值 0.3:0.6）。集群升级后，原始的高低水位预热依然生效，如果用户想启用集群级别的全局配置，客户需要后台删除该节点的 annotation（node.yangtse.io/eni-warm-policy），以启用集群级别配置的容器网卡动态预热的能力。

## 原理说明

CCE Turbo 的容器网卡动态预热提供了 4 个相关的容器网卡动态预热参数，您可以根据业务规划，合理设置集群的配置管理或节点池的配置管理中的容器网卡动态预热参数（其中节点池的容器网卡动态预热配置优先级高于集群的容器网卡动态预热配置）。

表10-3 容器网卡动态预热参数

| 容器网卡动态预热参数 | 默认值 | 参数说明 | 配置建议 |
| --- | --- | --- | --- |
| 节点最少绑定容器网卡数(nic-minimum-target) | 10 | 保障节点最少有多少张容器网卡绑定在节点上，支持数值跟百分比两种配置方式。<br>• 数值配置：参数值需为正整数。例如 10，表示节点最少有 10 张容器网卡绑定在节点上。当超过节点的容器网卡配额时，后台取值为节点的容器网卡配额。<br>• 百分比配置：参数值范围为 1%-100%。例如 10%，如果节点容器网卡配额 128，表示节点最少有 12 张（向下取整）容器网卡绑定在节点上。<br>建议 nic-minimum-target 与 nic-maximum-target 为同类型的配置方式（同采用数值配置或同采用百分比配置）。 | 建议配置为大部分节点平时日常运行的 Pod 数。 |
| 节点预热容器网卡上限检查值(nic-maximum-target) | 0 | 当节点绑定的容器网卡数超过节点预热容器网卡上限检查值(nic-maximum-target)，不再主动预热容器网卡。<br>当该参数大于等于节点最少绑定容器网卡数(nic-minimum-target)时，则开启预热容器网卡上限值检查；反之，则关闭预热容器网卡上限值检查。支持数值跟百分比两种配置方式。 | 建议配置为大部分节点平时最多运行的 Pod 数。 |

| 容器网卡动态预热参数 | 默认值 | 参数说明 | 配置建议 |
| --- | --- | --- | --- |
| | | • 数值配置：参数值需为正整数。例如0，表示关闭预热容器网卡上限值检查。当超过节点的容器网卡配额时，后台取值为节点的容器网卡配额。<br>• 百分比配置：参数值范围为 1%-100%。例如 50%，如果节点容器网卡配额 128，表示节点预热容器网卡上限检查值 64（向下取整）。<br>建议 nic-minimum-target 与 nic-maximum-target 为同类型的配置方式（同采用数值配置或同采用百分比配置）。 | |
| 节点动态预热容器网卡数(nic-warm-target) | 2 | 当 Pod 使用完节点最少绑定容器网卡数(nic-minimum-target)后，会始终额外预热多少张容器网卡，只支持数值配置。<br>当 节点动态预热容器网卡数(nic-warm-target) + 节点当前绑定的容器网卡数 大于 节点预热容器网卡上限检查值(nic-maximum-target) 时，只会预热 nic-maximum-target 与节点当前绑定的容器网卡数的差值。 | 建议配置为大部分节点日常 10s 内会瞬时弹性扩容的 Pod 数。 |
| 节点预热容器网卡回收阈值(nic-max-above-warm-target) | 2 | 只有当 节点上空闲的容器网卡数 - 节点动态预热容器网卡数(nic-warm-target) 大于此阈值 时，才会触发预热容器网卡的解绑回收。只支持数值配置。<br>• 调大此值会减慢空闲容器网卡的回收，加快 Pod 的启动速度，但会降低 IP 地址的利用率，特别是在 IP 地址紧张的场景，**请谨慎调大**。<br>• 调小此值会加快空闲容器网卡的回收，提高 IP 地址的利用率，但在瞬时大量 Pod 激增的场景，部分 Pod 启动会稍微变慢。 | 建议配置为大部分节点日常在分钟级时间范围内会频繁弹性扩容缩容的 Pod 数 - 大部分节点日常 10s 内会瞬时弹性扩容的 Pod 数。 |

## 配置示例

| 级别 | 用户业务场景 | 配置示例 |
| --- | --- | --- |
| 集群级别 | 集群中所有节点采用 c7n.4xlarge.2 机型（辅助弹性网卡配额 128）<br>集群下大部分节点平时日常运行 20 个 Pod 左 | 集群级别的全局配置：<br>• nic-minimum-target: 20 或 16% |

| 级别 | 用户业务场景 | 配置示例 |
| --- | --- | --- |
| | 右<br>集群下大部分节点最多运行 60 个 Pod<br>集群下大部分节点日常 10s 内会瞬时弹性扩容 10 个 Pod<br>集群下大部分节点日常在分钟级时间范围内会频繁弹性扩容缩容 15 个 Pod | • nic-maximum-target: 60 或 47%<br>• nic-warm-target: 10<br>• nic-max-above-warm-target: 5 |
| 节点池级别 | 集群中用户新创建了一个使用大规格机型 c7.8xlarge.2 的节点池（辅助弹性网卡配额 256）<br>节点池下大部分节点平时日常运行 100 个 Pod 左右<br>节点池下大部分节点最多运行 128 个 Pod<br>节点池下大部分节点日常在 10s 内会瞬时弹性扩容 10 个 Pod<br>节点池下大部分节点日常在分钟级时间范围内会频繁弹性扩容缩容 12 个 Pod | 节点池级别的差异化配置：<br>• nic-minimum-target: 100 或 40%<br>• nic-maximum-target: 120 或 50%<br>• nic-warm-target: 10<br>• nic-max-above-warm-target: 2 |

**说明**

使用 HostNetwork 的 Pod 不计入 Pod 数中。

## 集群级别的全局配置

**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 。

图10-17 配置管理


**步骤 3** 在侧边栏滑出的“配置管理”窗口中，选择网络组件配置，参数值请参见配置示例。


**步骤 4** 配置完后单击“确定”，等待 10s 左右即可生效。

**----结束**

## 节点池级别的差异化配置

**步骤 1** 登录 CCE 控制台。

**步骤 2** 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，选择网络组件配置，参数值请参见配置示例。


**步骤 5** 配置完后单击“确定”，等待 10s 左右即可生效。

**----结束**

# 11 存储

## 11.1 存储扩容

CCE 节点可进行扩容的存储类型如下：

表11-1 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<br>• 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            # docker
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 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。

  关于节点操作系统与容器存储 Rootfs 的关系，请参见节点操作系统与容器引擎对应关系。

- 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。

更多关于容器存储空间分配的内容，请参考数据盘空间分配说明。

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**

## 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“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”。
    ii. 输入新增容量，并单击“确定”。
  - 对于包周期收费的实例，需要先在 EVS 控制台扩容，然后再修改 PVC 中容量大小。
- 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。

# 11.2 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+。
- 不支持安全容器。
- 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 sc 的 yaml 文件，例如 sfsturbo-sc-test.yaml。

配置示例：

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  name: sfsturbo-sc-test
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：挂载根配置。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 sc 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”，为“false”时，会将 pv 对应的 SFS Turbo 子目录删除。默认进行归档。

**步骤 3** 执行 **kubectl create -f sfsturbo-sc-test.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-sc-test
  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 Turbo 资源的总容量，若 Turbo 资源总容量不足，请及时到 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 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，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 **kubectl create -f deployment-test.yaml** 创建 deployment 负载。

**----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，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'
          env:
            - name: PAAS APP NAME
              value: deploy-sfs-nfs-rw-in
            - name: PAAS NAMESPACE
              value: default
            - name: PAAS PROJECT ID
              value: 8190a2a1692c46f284585c56fc0e2fb9
          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-sc-test
  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 **kubectl create -f statefulset-test.yaml** 创建 statefulset 负载。

**----结束**

# 11.3 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 SSD（超高 I/O）可以指定。

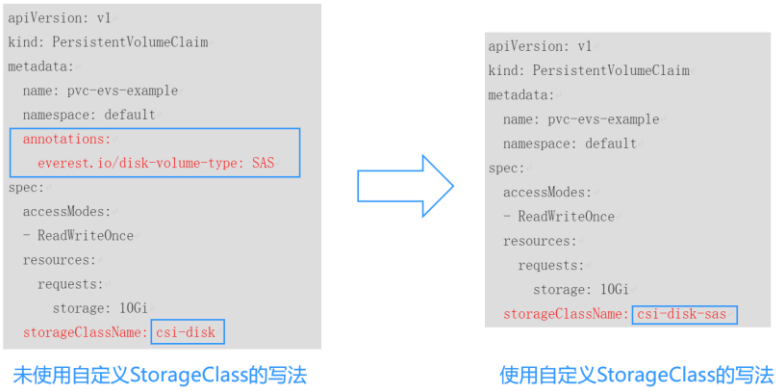
这种写法在如下几种场景下存在问题：

- 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。




- 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，因此删除集群或删除 PVC 时不会回收包周期的 SFS Turbo 资源。

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

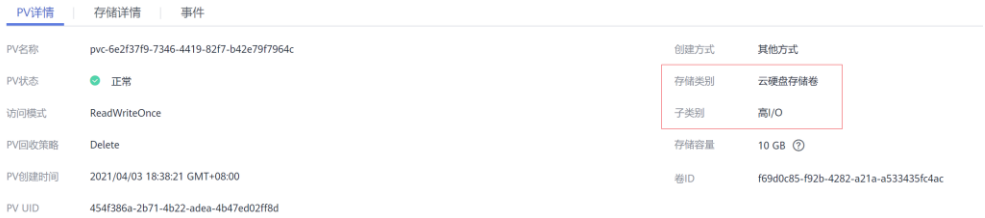
- 使用 csi-disk-sas 创建 PVC。

```
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。



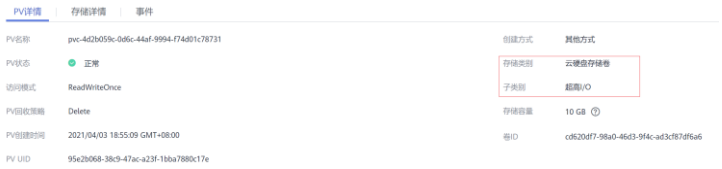

- 不指定 StorageClassName，使用默认配置，如下所示，并未指定 storageClassName。

```
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
```

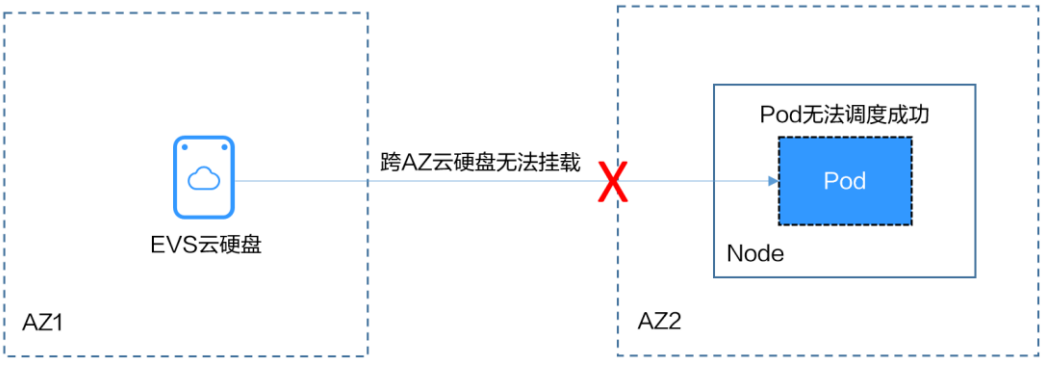
在 CCE 控制台界面上查看 PVC 详情，在“PV 详情”页签下可以看到磁盘类型是超高 I/O。




# 11.4 节点跨 AZ 时云硬盘自动拓扑（csi-disk-topology）

## 应用现状

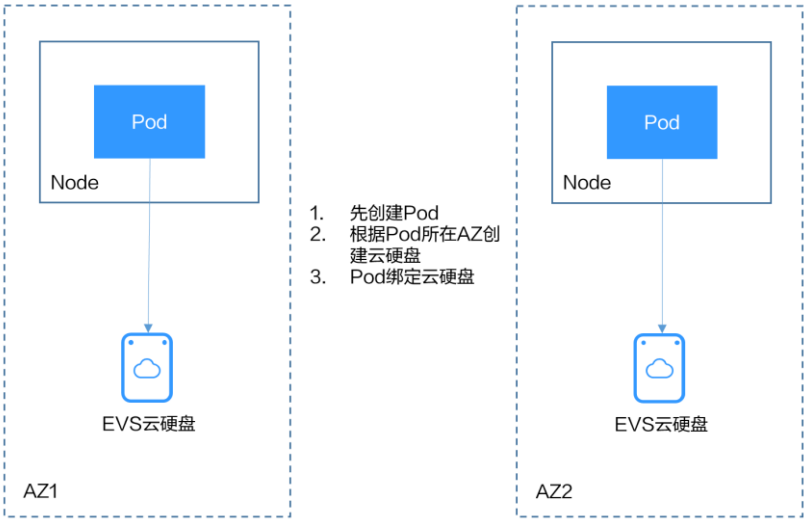
云硬盘使用在使用时无法实现跨 AZ 挂载，即 AZ1 的云硬盘无法挂载到 AZ2 的节点上。有状态工作负载调度时，如果使用 csi-disk 存储类，会立即创建 PVC 和 PV（创建 PV 会同时创建云硬盘），然后 PVC 绑定 PV。但是当集群节点位于多 AZ 下时，PVC 创建的云硬盘可能会与 Pod 调度到的节点不在同一个 AZ，导致 Pod 无法调度成功。




## 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>
```

# 12 容器

## 12.1 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
```

# 12.2 实现升级实例过程中的业务不中断

## 应用场景

在 Kubernetes 集群中，应用通常采用 Deployment + LoadBalancer 类型 Service 的方式对外提供访问。应用更新或升级时，Deployment 会创建新的 Pod 并逐步替换旧的 Pod，这个过程中可能会导致服务中断。

## 解决方案

避免服务中断可以从 Deployment 和 Service 两类资源入手：

- Deployment 可以采用**滚动升级**的升级方式，为对各个实例逐个进行更新，而不是同时对所有实例进行全部更新，可以控制 Pod 的更新速度和并发数，从而确保了升级过程中业务不中断。例如，可以设置 maxSurge 和 maxUnavailable 参数，控制同时创建的新 Pod 数量和同时删除的旧 Pod 数量。尽量确保待升级的工作负载至少有 2 个实例。如果只有 1 个实例，建议在手动伸缩到 2 个实例后，再进行升级操作。
- LoadBalancer 类型的 Service 存在两种服务亲和模式：
  - **集群级别**的服务亲和（externalTrafficPolicy: Cluster）：Cluster 模式下，如果当前节点没有业务 Pod，会将请求转发给其他节点上的 Pod，在跨节点转发会丢失源 IP。
  - **节点级别**的服务亲和（externalTrafficPolicy: Local）：Local 模式下，请求会直接转发给 Pod 所在的节点，不存在跨节点转发，因此可以保留源 IP。但是在 Local 模式下，如果实例滚动升级时 Pod 所在节点发生变化，导致 ELB 侧后端服务器会同步变化，可以通过实例原地升级的方式避免服务中断。

综上，实现升级实例过程中的业务不中断的方案可参考下表：

| 场景 | Service | Deployment |
| --- | --- | --- |
| 不需要保留源 IP | 选用**集群级别**的服务亲和模式 | 滚动升级 + 优雅终止 + 存活/就绪探针 |
| 需要保留源 IP | 选用**节点级别**的服务亲和模式 | 滚动升级 + 优雅终止 + 存活/就绪探针 + 节点亲和（保证更新过程中每个节点上至少有一个 Running Pod） |

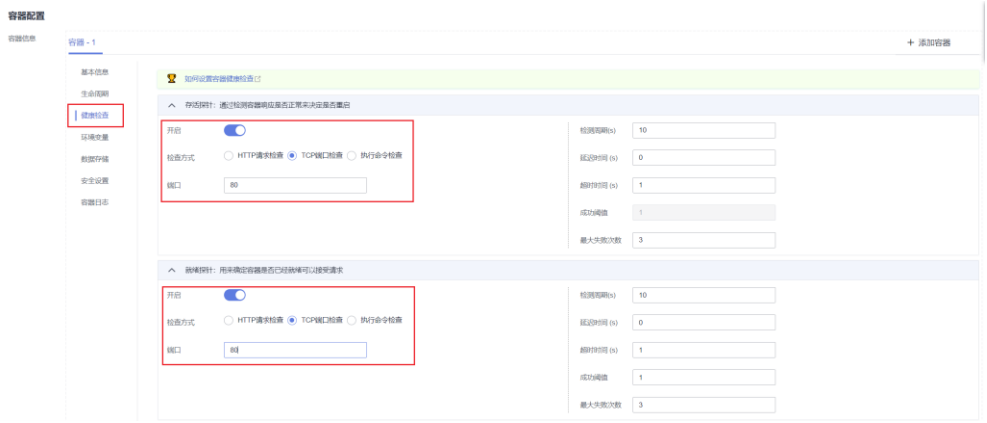
## 操作步骤

**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”。

**步骤 2** 在工作负载列表中，单击待升级的工作负载操作列的“升级”，进入升级工作负载页面。

1. 设置存活/就绪探针：在容器配置中选择“健康检查”，开启存活探针和就绪探针。示例中均为 TCP 端口检查，请根据应用实际情况进行设置。检测周期、延时时间、超时时间等数据需要合理设置，部分应用启动时间较长，如果设置的时间过短，会导致 Pod 反复重启。

图12-1 存活/就绪探针

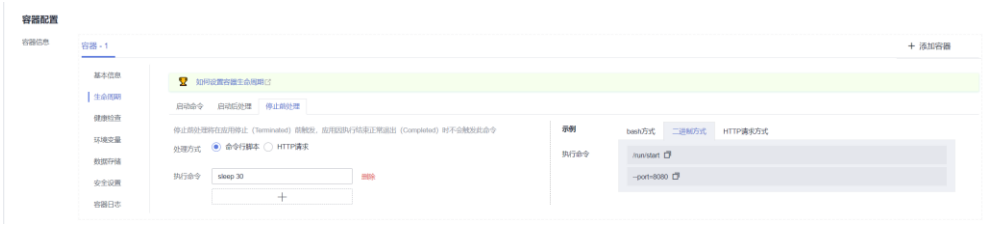



2. 设置滚动升级：在高级配置中选择“升级策略”，升级方式设置为“滚动升级”，逐步用新版本实例替换旧版本实例。

图12-2 滚动升级


3. 设置优雅终止：
   1. 在容器配置中选择“生命周期”，设置停止前处理，建议设置为业务处理完所有剩余请求所需的时间。
   2. 在高级配置中选择“升级策略”，设置缩容时间窗，即 terminationGracePeriodSeconds 参数，指定容器停止前命令执行的等待时间。建议在容器停止前命令执行时间的基础上加 30s。

图12-3 停止前命令



4. 设置节点亲和：Service 为**节点级别**的服务亲和模式时建议设置。在高级配置中选择“调度策略”，设置节点亲和性，在添加调度策略时，指定工作负载需要亲和的节点。

图12-4 节点亲和性


步骤 3 设置完成后单击“升级工作负载”。

在“实例列表”页签下，可查看到会先创建实例，然后再停止实例，始终保证有实例正在运行。

**----结束**

# 12.3 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**

# 12.4 对容器进行初始化操作

## 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**

# 12.5 容器与节点时区同步

## 案例场景

- 场景一：容器与节点时区同步
- 场景二：容器、容器日志与节点时区同步
- 场景三：工作负载与节点时区同步

## 场景一：容器与节点时区同步

步骤 1 登录 CCE 控制台。

步骤 2 在创建工作负载基本信息页面，开启“时区同步”，即容器与节点使用相同时区。

图12-5 开启时区同步


步骤 3 登录节点进入容器查询容器时区是否与节点保持一致。

**date -R**

命令行终端显示如下信息：

```
Tue, 04 Jun 2019 15::08:47 +0800
```

**docker ps -a|grep** *test*

命令行终端显示如下信息：

```
0edd74c66bdb        b2b9b536b744        "nginx -g 'daemon ..."   6 hours ago         Up 6 hours
                    k8s_container-0_test-7d7d7f4965-xwqkx_default_abf6df2e-85f7-11e9-93df-fa163ee0f9
1a_1
```

**docker exec -it** *oedd74c66bdb* **/bin/sh**

**date -R**

命令行终端显示如下信息：

```
Tue, 04 Jun 2019 15:09:20 +0800
```

**----结束**

## 场景二：容器、容器日志与节点时区同步

Java 应用打印的日志时间和通过 date -R 方式获取的容器标准时间相差 8 小时。

**步骤 1** 登录 CCE 控制台。

**步骤 2** 在创建工作负载基本信息页面，开启“时区同步”，即容器与节点使用相同时区。

图12-6 开启时区同步


**步骤 3** 登录节点进入容器，修改 catalina.sh 脚本。

**cd /usr/local/tomcat/bin**

**vi catalina.sh**

若无法在容器中执行 vi 命令，可以直接执行步骤 4，也可以执行 vi 命令，在脚本中添加 -Duser.timezone=GMT+08，如下图所示：

```
# Do this here so custom URL handles (specifically 'war:...') can be used in the security policy
JAVA_OPTS="$JAVA_OPTS -Djava.protocol.handler.pkgs=org.apache.catalina.webresources -Duser.timezone=GMT+08"
```

步骤 4 将脚本先从容器内拷贝至节点，在脚本中添加-Duser.timezone=GMT+08 后，从节点拷贝到容器中。

容器内的文件拷贝至宿主机：

**docker cp mycontainer: /usr/local/tomcat/bin/catalina.sh /home/catalina.sh**

宿主机中的文件拷贝至容器内：

**docker cp /home/catalina.sh mycontainer:/ usr/local/tomcat/bin/catalina.sh**

步骤 5 重启容器。

**docker restart** *container_id*

步骤 6 重启后查看日志中的时区是否与节点同一时区。

查看方法：单击工作负载名称进入工作负载详情页，单击右上角的“日志”按钮可查看日志详情。日志约需要等待 5 分钟查看。

**----结束**

## 场景三：工作负载与节点时区同步

- 方法一：制作容器镜像时，将时区设置为 CST。
- 方法二：若不希望修改容器，可在 CCE 控制台创建工作负载时，将本机的“/etc/localtime”目录挂载到容器的“/etc/localtime”目录下。

  **示例如下：**

```
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
```

# 12.6 容器网络带宽限制

## 应用场景

同一个节点上的容器会共用主机网络带宽，对容器的网络带宽进行限制，可以有效避免容器之间相互干扰，提升容器间的网络稳定性。

## 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）<br>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）<br>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）<br>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<br>• 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 | 支持 K 级别以上的限速 | 目前仅支持兆（M）级别以上的限速 | |

| 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 |
| --- | --- | --- | --- |
| 限 | | | |

## 操作步骤

步骤 1 编辑工作负载 yaml 文件。

**vi deployment.yaml**

根据需要在 spec.template.metadata.**annotations** 中设置工作负载实例的网络带宽，限制容器的网络流量。

未设置默认不限制。

示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
      annotations:
       # 入方向网络带宽
        kubernetes.io/ingress-bandwidth: 100M
       # 出方向网络带宽
        kubernetes.io/egress-bandwidth: 1G
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

表12-1 工作负载实例网络带宽限制字段详解

| 字段名称 | 字段说明 | 必选/可选 |
| --- | --- | --- |
| kubernetes.io/ingress-bandwidth | 工作负载实例入方向网络带宽<br>取值范围：1k-1P，带宽如果设置大于32G，实际带宽将等于 32G | 可选 |

| 字段名称 | 字段说明 | 必选/可选 |
| --- | --- | --- |
| kubernetes.io/egress-bandwidth | 工作负载实例出方向网络带宽<br>取值范围：1k-1P，带宽如果设置大于32G，实际带宽将等于 32G | 可选 |

步骤 2 创建工作负载。

**kubectl create -f deployment.yaml**

命令行终端显示如下类似信息：

```
deployment.apps/nginx created
```

**----结束**

# 12.7 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12-2 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 12-3。 |

表12-3 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 12-4。 |

表12-4 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**

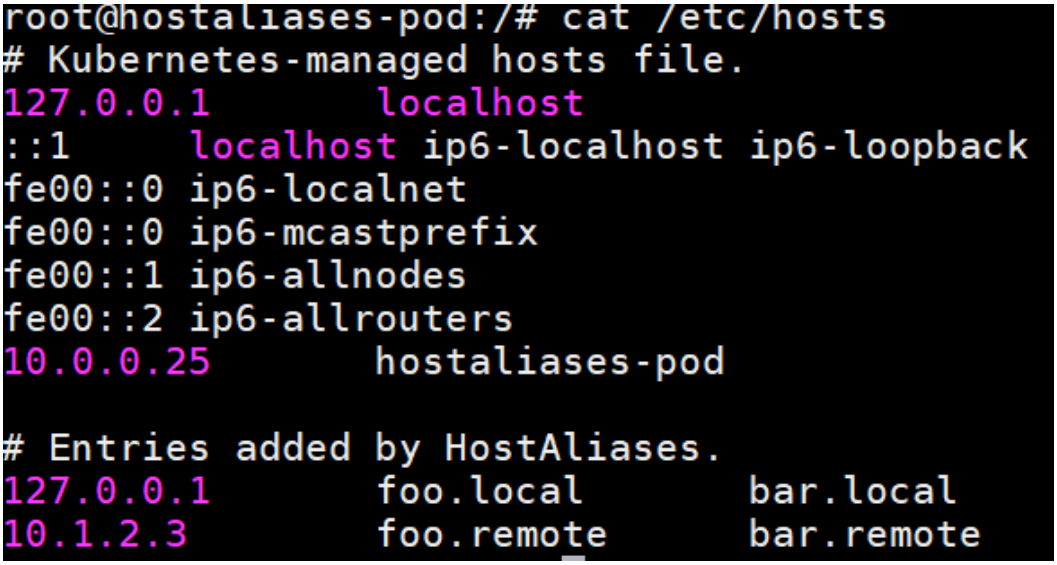
pod 状态显示为 Running，表示 pod 已创建成功。

```
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**




**----结束**

# 12.8 CCE 容器中域名解析的最佳实践

本文档重点介绍在 CCE 容器中如何配置域名解析。

## 服务

- 在创建工作负载（Deployment 或 ReplicaSet）之前，需要先创建与之相关联的服务。 因为 Kubernetes 在启动容器时，会为容器提供所有正在运行的服务作为环境变量。 例如，如果存在名为 foo 的服务，则所有容器将在其初始环境中获得以下变量。

  ```
  FOO_SERVICE_HOST=<the host the Service is running on>
  FOO_SERVICE_PORT=<the port the Service is running on>
  ```

  因此必须在 Pod 被创建之前创建它想要访问的任何 Service， 否则环境变量将不会生效，而使用 DNS 则没有此限制。
- CCE 集群提供了 CoreDNS 插件作为集群中的 DNS 服务器。DNS 服务器为新的 Services 监视 Kubernetes API，并为每个 Services 创建一组 DNS 记录。 如果在整个集群中启用了 DNS，则所有 Pods 应该能够自动对 Services 进行名称解析。
- 除非绝对必要，否则不要为 Pod 指定 hostPort。 将 Pod 绑定到 hostPort 时，它会限制 Pod 可以调度的位置数，因为每个<hostIP, hostPort, protocol>组合必须是唯一

的。 如果您没有明确指定 hostIP 和 protocol，Kubernetes 将使用 0.0.0.0 作为默认 hostIP 和 TCP 作为默认 protocol。

如果您只需要访问端口以进行调试，则可以使用 apiserver proxy 或 kubectl port-forward。

如果您明确需要在节点上公开 Pod 的端口，请在使用 hostPort 之前考虑使用 NodePort 服务。

- 避免使用 hostNetwork，原因与 hostPort 相同。
- 当您不需要 kube-proxy 负载均衡时，使用无头服务 headless-services(ClusterIP 被设置为 None)以便于服务发现。

## DNS

CCE 的 Kubernetes 集群默认提供了一个 DNS 插件 Service，即使用 CoreDNS 自动为其它 Service 指派 DNS 域名。 如果它在集群中处于运行状态，可以通过如下命令来检查：

```
kubectl get services coredns --namespace=kube-system
NAME      TYPE       CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
kube-dns   ClusterIP   10.0.0.10    <none>       53/UDP,53/TCP   8m
```

如果没有在运行，可以 describe 这个 pod 查看没有启用的原因。 假设已经有一个 Service，它具有一个长久存在的 IP， 一个为该 IP 指派名称的 DNS 服务器（coredns 集群插件），可以通过标准做法，使在集群中的任何 Pod 都能与该 Service 通信。可以运行另一个 curl 应用来进行测试，启用新的 pod 并通过进入容器内部 curl 当前这个 service 的域名，查看是否能正确解析域名。当然，有的场景下是无法 curl 通的，这与接下来的 Dns 的查找原理与配置有关。

使用 CCE 提供的托管式 Kubernetes 创建 Pod，Pod 的域名解析参数采用了一些默认值，没有开放全部的 dnsConfig 配置。在使用时候，您需要了解清楚提供的默认配置。典型的一个配置是 ndots ，如果您在 Pod 内访问的域名字符串，点数量在 ndots 阈值范围内，则被认为是 Kubernetes 集群内部域名，会被追加 **..svc.cluster.local** 后缀。

## DNS 查找原理与规则

DNS 域名解析配置文件 /etc/resolv.conf

```
nameserver 10.247.x.x
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:3
```

**参数说明：**

- nameserver：域名解析服务器。
- search：域名的查找后缀规则，查找配置越多，说明域名解析查找匹配次数越多，这里匹配有 3 个后缀，则查找规则至少 6 次，因为 IPv4，IPv6 都要匹配一次。
- options：域名解析选项，多个 KV 值；其中典型的有 ndots ，访问的域名字符串内的点字符数量超过 ndots 值，则认为是完整域名，直接解析，如不足，则追加 **..svc.cluster.local** 后缀。

## Kubernetes 的 dnsConfig 配置说明

- nameservers：将用作 Pod 的 DNS 服务器的 IP 地址列表。最多可以指定 3 个 IP 地址。当 Pod 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。列出的服务器将合并到从指定的 DNS 策略生成的基本名称服务器，并删除重复的地址。
- searches：Pod 中主机名查找的 DNS 搜索域列表。此属性是可选的。指定后，提供的列表将合并到从所选 DNS 策略生成的基本搜索域名中，并删除重复的域名。Kubernetes 最多允许 6 个搜索域。
- options：可选的对象列表，其中每个对象可以具有 name 属性（必需）和 value 属性（可选）。此属性中的内容将合并到从指定的 DNS 策略生成的选项中，并删除重复的条目。

详情请参考：Kubernetes 官网的 dns 配置说明。

## DnsPolicy 域名解析的几种场景应用

POD 里的 DNS 策略可以对每个 pod 进行设置，他支持三种策略：Default、ClusterFirst、None。

- Default：表示 Pod 里面的 DNS 配置继承了宿主机上的 DNS 配置。简单来说，就是该 Pod 的 DNS 配置会跟宿主机完全一致，也就是和 node 上的 dns 配置是一样的。
- ClusterFirst：相对于上述的 Default，ClusterFirst 是完全相反的操作，它会预先把 kube-dns（或 CoreDNS）的信息当作预设参数写入到该 Pod 内的 DNS 配置。ClusterFirst 是默认的 pod 设置，若没有在 Pod 内特别描述 PodPolicy，则会将 dnsPolicy 预设为 ClusterFirst。不过 ClusterFirst 还有一个冲突，如果您的 Pod 设置了 HostNetwork=true，则 ClusterFirst 就会被强制转换成 Default。
- None：它表示会清除 Pod 预设的 DNS 配置，当 dnsPolicy 设置成这个值之后，Kubernetes 不会为 Pod 预先载入任何自身逻辑判断得到的 DNS 配置。因此若要将 dnsPolicy 的值设为 None，为了避免 Pod 里面没有配置任何 DNS，建议再添加 dnsConfig 来描述自定义的 DNS 参数。

请参阅下面的 DNS 配置场景：

**场景一：采用自定义 DNS**

采用自己建的 DNS 来解析 Pods 中的应用域名配置，可以参考以下代码配置，此配置在 Pod 中的 DNS 可以完全自定义，适用于已经有自己建的 DNS，迁移后的应用也不需要去修改相关的配置。

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
```

```
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

**场景 2：采用 kubernets 的 DNS 插件 CoreDNS**

优先使用 Kubernetes 的 DNS 服务解析，失败后再使用外部级联的 DNS 服务解析。

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: ClusterFirst
```

**场景 3：采用公网域名解析**

适用于 Pods 中的域名配置都在公网访问，这样的话 Pods 中的应用都从外部的 DNS 中解析对应的域名。

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: Default
```

**场景 4：采用 HostNet 的 DNS 解析**

如果在 Pod 中使用 hostNetwork:true 来配置网络，pod 中运行的应用程序可以直接看到宿主机的网络接口，宿主机所在的局域网上所有网络接口都可以访问到该应用程序，配置如下所示：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
    hostNetwork: true
    dnsPolicy: ClusterFirstWithHostNet
    containers:
    - name: nginx
      image: nginx:1.7.9
      ports:
      - containerPort: 80
```

如果不加上 dnsPolicy: ClusterFirstWithHostNet，即便 pod 默认使用宿主机的 DNS，也会导致容器内不能通过 service name 访问 K8s 集群中其他 Pod。

## CoreDNS 配置

**1、CoreDNS ConfigMap 选项**

先来看看默认的 CoreDns 的配置文件：

```
Corefile: |
  .:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
       pods insecure
       upstream
       fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    forward . /etc/resolv.conf
    cache 30
    loop
    reload
    loadbalance
```

**参数说明：**

- error：错误记录到 stdout。
- health：CoreDNS 的运行状况报告为 http://localhost:8080/health。
- kubernetes：CoreDNS 将根据 Kubernetes 服务和 pod 的 IP 回复 DNS 查询。
- prometheus：CoreDNS 的度量标准可以在 http://localhost:9153/Prometheus 格式的指标中找到；可以通过 http://localhost:9153/metrics 获取 prometheus 格式的监控数据。
- proxy、forward：任何不在 Kubernetes 集群域内的查询都将转发到预定义的解析器（/etc/resolv.conf）；本地无法解析后，向上级地址进行查询，默认使用宿主机的 /etc/resolv.conf 配置。
- cache：启用前端缓存。
- loop：检测简单的转发循环，如果找到循环则停止 CoreDNS 进程。
- 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。
- loadbalance：这是一个循环 DNS 负载均衡器，可以在答案中随机化 A，AAAA 和 MX 记录的顺序。

**2、配置外部 dns**

有些服务不在 Kubernetes 内部，在内部环境内需要通过 dns 去访问，名称后缀为 carey.com。

```
carey.com:53 {
      errors
      cache 30
      proxy . 10.150.0.1
   }
```

完整的配置文件：

```
Corefile: |
  .:53 {
     errors
     health
     kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
     }
     prometheus :9153
     forward . /etc/resolv.conf
     cache 30
     loop
     reload
     loadbalance
  }
  carey.com:53 {
     errors
     cache 30
     proxy . 10.150.0.1
  }
```

当前 CCE 的插件管理支持配置存根域，相比较直接编辑 comfigmap 更加灵活方便，无需关注 pod 的域名解析配置场景。

# 12.9 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。您可通过设置 OBS 的挂载参数 umask=0077，将 Core Dump 文件正常存储到 OBS 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。

**说明**

EulerOS 2.3 Systemd 有一个社区 bug 影响容器 Core Dump，如需使用 Core Dump 需执行如下操作。

1. 在节点的/usr/lib/systemd/system/docker.service 文件中，将 **LimitCORE** 的值修改为 **infinity**。
2. 重启 Docker。
3. 业务容器重新部署。

## 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
```

# 13 权限

## 13.1 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（获取方法参见图 13-1），**/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。

图13-1 获取内网或外网 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。

# 13.2 集群命名空间 RBAC 授权

## 应用现状

CCE 的权限控制分为集群权限和命名空间权限两种权限范围，其中命名空间权限是基于 Kubernetes RBAC 能力的授权，可以对集群和命名空间内的资源进行授权。

当前，在 CCE 控制台，命名空间权限默认提供 cluster-admin、admin、edit、view 四种 ClusterRole 角色的权限，这四种权限是针对命名空间中所有资源进行配置，无法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。

## 解决方案

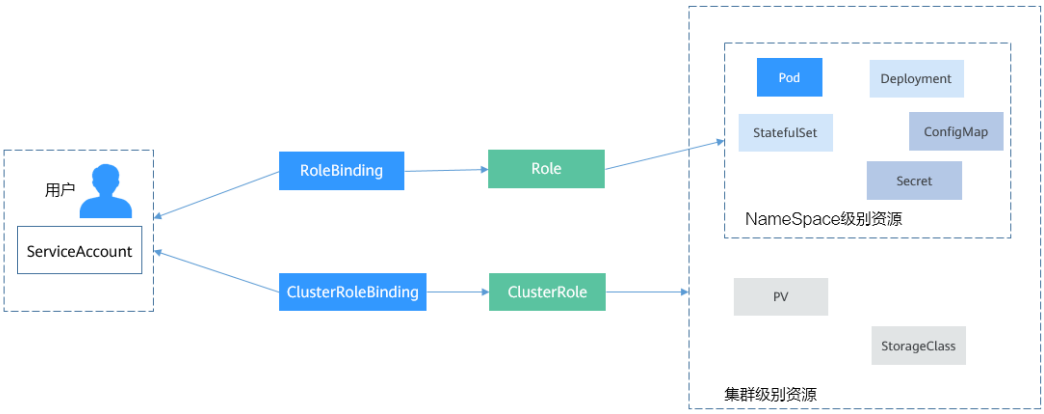
Kubernetes 提供一套 RBAC 授权机制，可以非常方便的实现命名空间内容资源的权限控制。

- 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13-2 角色绑定




上图中的用户在 CCE 中可以是 IAM 用户或用户组，通过这样的绑定设置，就可以非常方便的实现命名空间内容资源的权限控制。

下面将通过给一个 IAM 用户 user-example 配置查看 Pod 的权限（该用户只有查看 Pod 的权限，没有其他权限），演示 Kubernetes RBAC 授权方法。

## 前提条件

本文所述方法仅在 v1.11.7-r2 及以上版本集群上生效，因为只有 v1.11.7-r2 及以上版本集群开启了 RBAC 功能。

## 创建 IAM 用户和用户组

使用帐号登录 IAM，在 IAM 中创建一个名为 user-example 的 IAM 用户和名为 cce-role-group 的用户组。


创建后给 cce-role-group 用户组授予 CCE FullAccess 权限，如下所示。


CCE FullAccess 拥有集群操作相关权限（包括创建集群等），但是没有操作 Kubernetes 资源的权限（如查看 Pod）。

## 创建集群

使用帐号登录 CCE，并创建一个集群。

**须知**

**注意不要使用 IAM 用户 user-example 创建集群，因为 CCE 会自动为创建集群的用户添加该集群所有命名空间 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。**

使用 IAM 用户 user-example 登录 CCE 控制台，在集群中下载 kubectl 配置文件并连接集群，执行命令获取 Pod 信息，可以看到没有相关权限，同样也无查看其它资源的权限。这说明 user-example 这个 IAM 用户没有操作 Kubernetes 资源的权限。

```
# kubectl get pod
Error from server (Forbidden): pods is forbidden: User
"0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "pods" in API group "" in
the namespace "default"
# 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"
```

## 创建 Role 和 RoleBinding

使用帐号登录 CCE 控制台，在上一步创建的集群中下载 kubectl 配置文件并连接集群，然后创建 Role 和 RoleBinding。

**说明**

此处使用帐号是因为集群是使用帐号创建，CCE 在创建集群时会自动给该帐号添加 cluster-admin 权限，也就是有权限创建 Role 和 RoleBinding。您也可以使用其他拥有创建 Role 和 RoleBinding 权限的 IAM 用户来操作。

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。

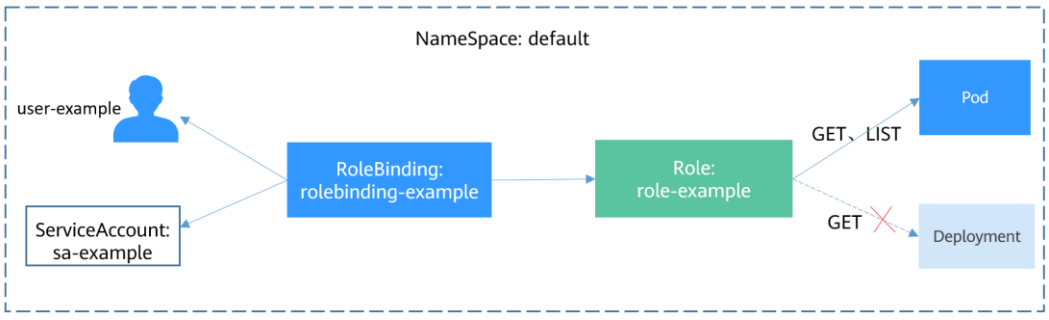
详细的类型和操作请参见使用 RBAC 鉴权。

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    # IAM 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13-3 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
```

## 配置验证

使用 IAM 用户 user-example 连接集群，查看 Pod，发现可以查看。

```
# kubectl get pod
NAME                              READY   STATUS    RESTARTS   AGE
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，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"
```

# 14 发布

## 14.1 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。

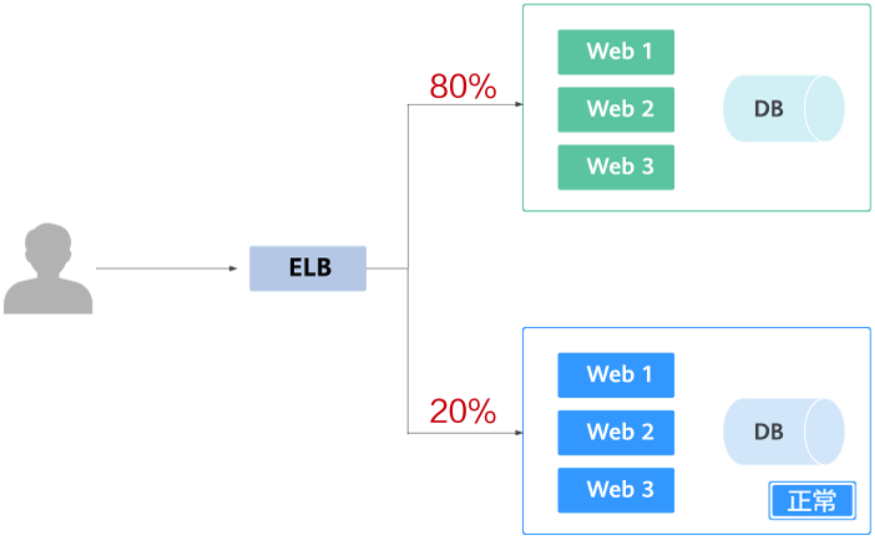
### 解决方案

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。

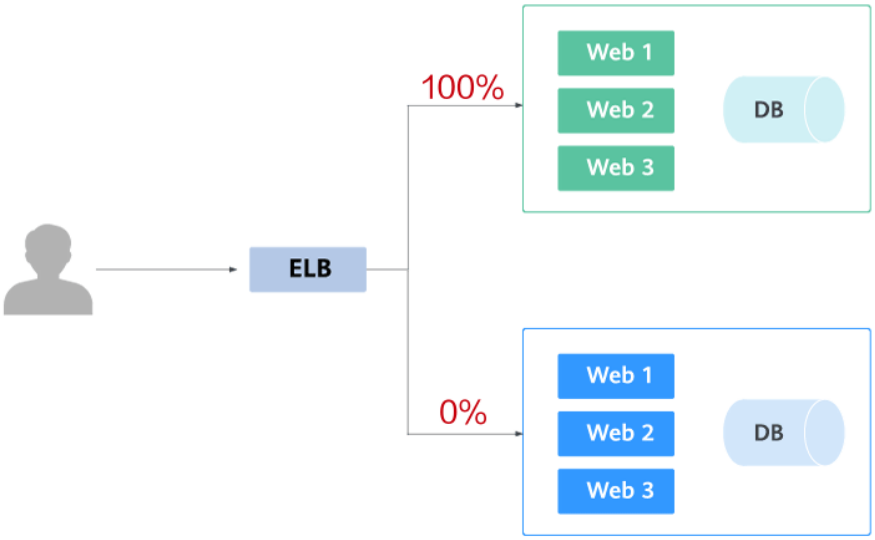
本文着重介绍灰度发布和蓝绿发布的原理及实践案例。

- 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。

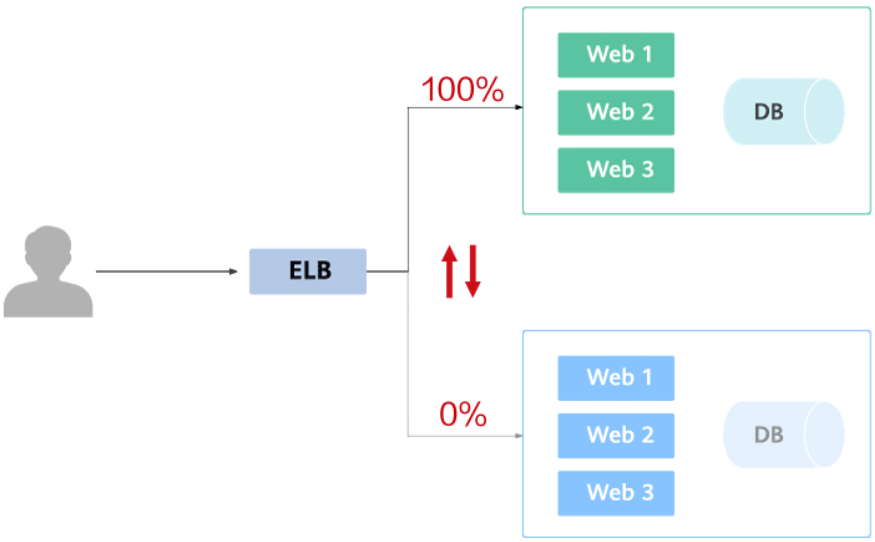
  以下示意图可描述灰度发布的大致流程：先切分 20%的流量到新版本，若表现正常，逐步增加流量占比，继续测试新版本表现。若新版本一直很稳定，那么将所有流量都切分到新版本，并下线老版本。




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。




## 灰度发布或蓝绿发布实现方式

利用 Kubernetes 原生的特性可以实现简单的灰度发布或蓝绿发布，比如：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，实现让服务从一个版本直接切换到另一个版本，从而实现蓝绿发布。如果您的灰度或蓝绿发布需求较复杂，可以向集群额外部署其他开源工具，例如 Nginx Ingress、Traefik，或将业务部署至服务网格，利用开源工具和服务网格的能力实现。几种方式分别对应本文如下内容：

- 使用 Service 实现简单的灰度发布和蓝绿发布
- 使用 Nginx Ingress 实现灰度发布和蓝绿发布

表14-1 实现方式对比

| 实现方式 | 适用场景 | 特点 | 缺点 |
| --- | --- | --- | --- |
| Service | 发布需求简单，小规模测试场景 | 无需引入过多插件或复杂的用法 | 纯手工操作，自动化程度差 |
| Nginx Ingress | 无特殊要求 | • 配置 Nginx Ingress 所支持的 Annotation 即可实现灰度发布或蓝绿发布，无需关注内部原理<br>• 支持基于 Header、Cookie 和服务权重三种流量切分的策略 | 集群需要安装 nginx-ingress 插件，存在资源消耗 |

Service 和 Nginx Ingress 方式均利用 Kubernetes 开源能力实现灰度发布和蓝绿发布，在这个过程中，CCE 也提供了很多便捷性，例如：

- 所有资源的创建、查看、修改均可以在管理控制台实现，相比 kubectl 命令行工具更为直观。
- LoadBalancer 类型的 Service 由 ELB 服务实现，在创建 Service 时，可以使用已有 ELB 实例，也可以新建一个 ELB 实例。
- 支持一键式安装 nginx-ingress 插件，并且在安装过程中实现 ELB 的创建与对接。

# 14.2 使用 Service 实现简单的灰度发布和蓝绿发布

CCE 集群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。

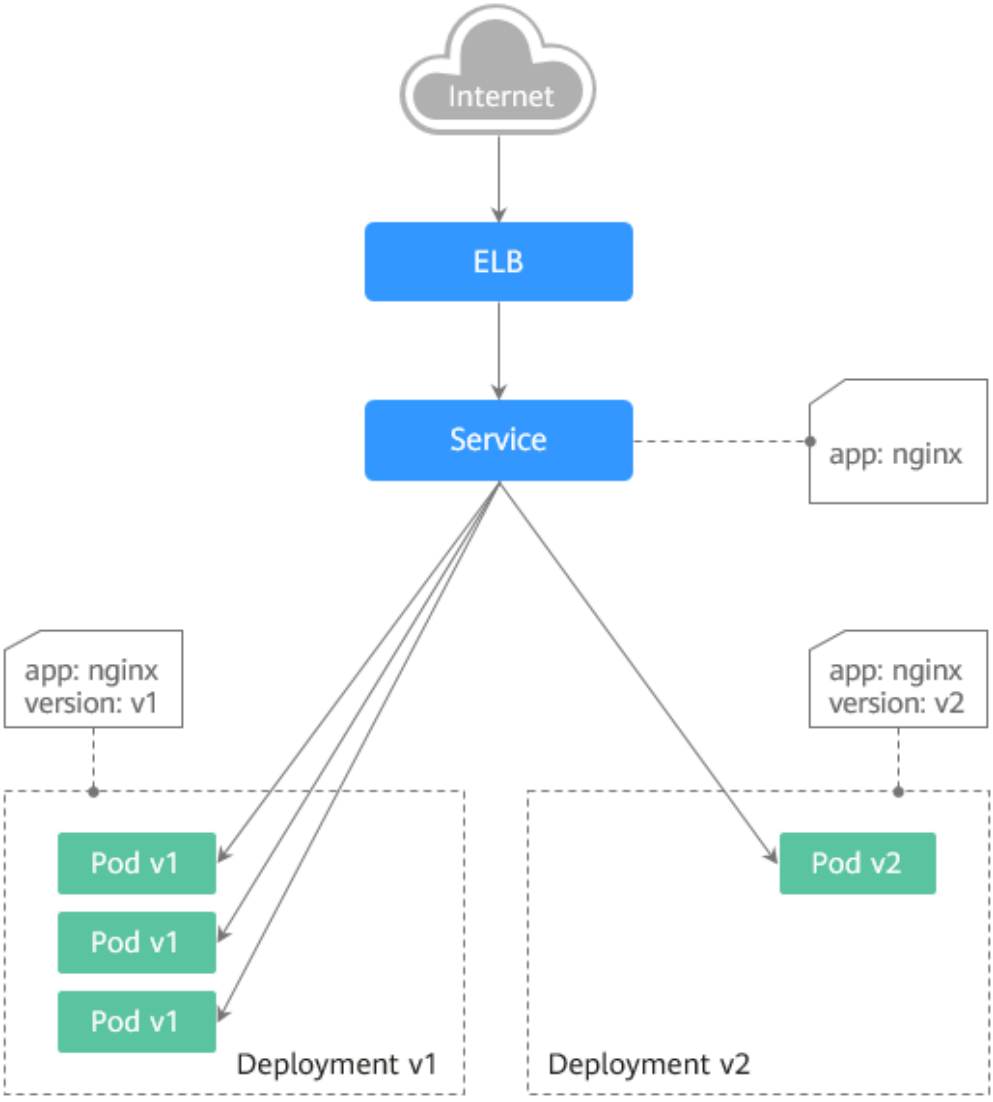
## 原理介绍

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。

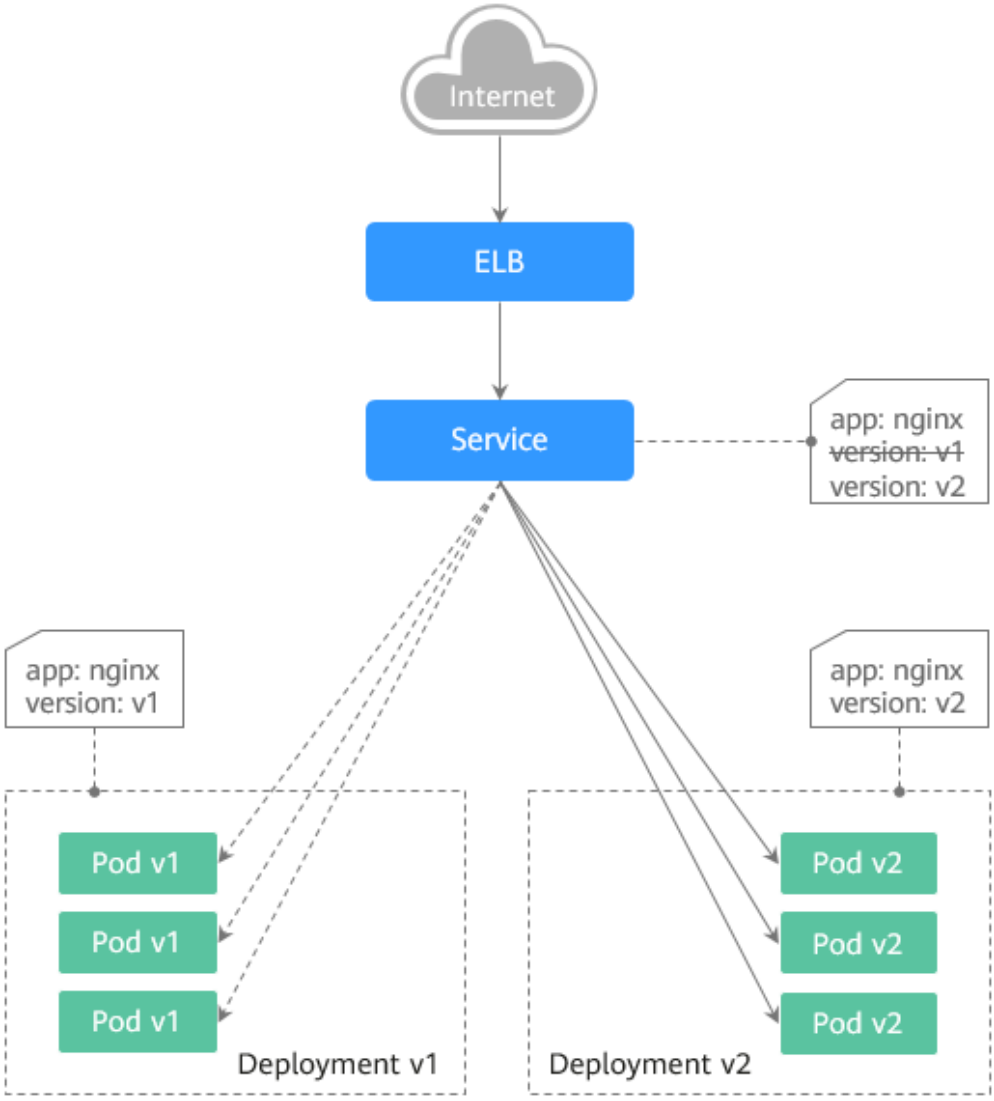
- **灰度发布原理**

  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**

# 14.3 使用 Nginx Ingress 实现灰度发布和蓝绿发布

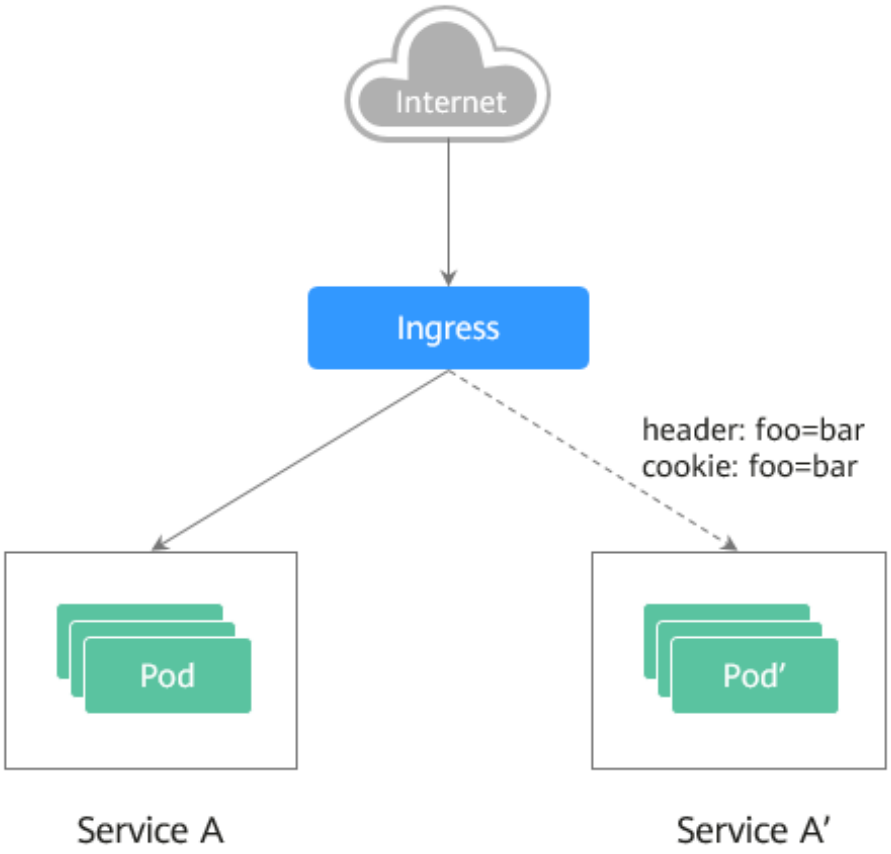
本文将介绍使用 Nginx Ingress 实现灰度发布和蓝绿发布的应用场景、用法详解及实践步骤。

## 应用场景

使用 Nginx Ingress 实现灰度发布适用场景主要取决于业务流量切分的策略，目前 Nginx Ingress 支持基于 Header、Cookie 和服务权重三种流量切分的策略，基于这三种策略可实现以下两种发布场景：

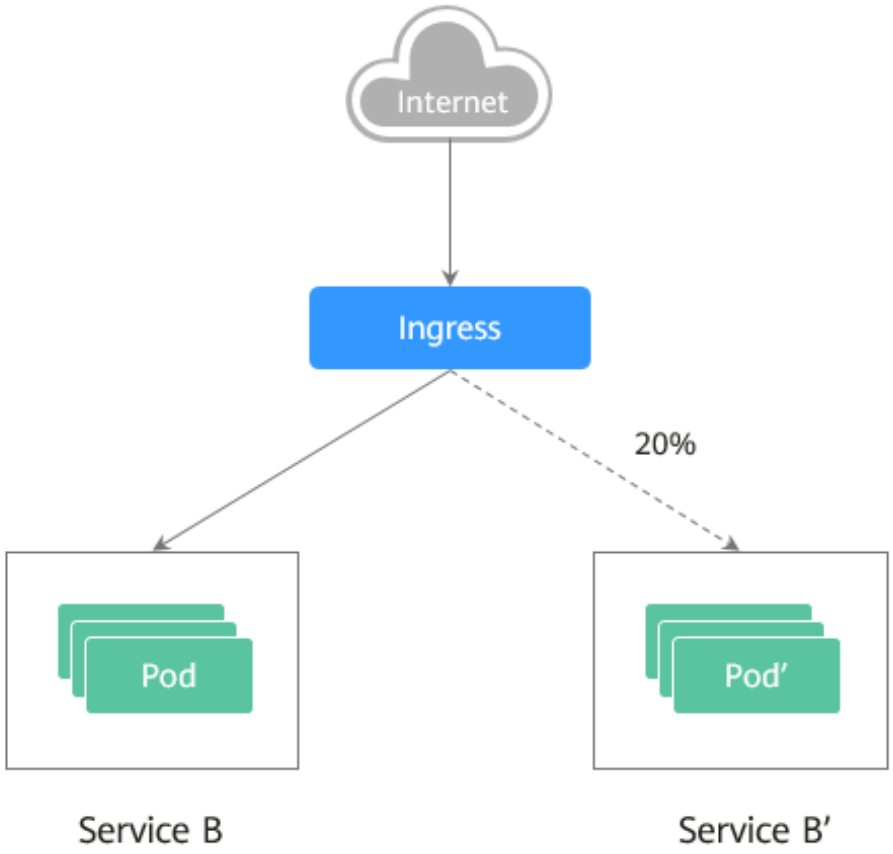
- **场景一：切分部分用户流量到新版本**

  假设线上已运行了一套对外提供七层服务的 Service A，此时开发了一些新的特性，需要发布上线一个新的版本 Service A'，但又不想直接替换原有的 Service A，而是期望将 Header 中包含 foo=bar 或者 Cookie 中包含 foo=bar 的用户请求转发到新版本 Service A'中。待运行一段时间稳定后，再逐步全量上线新版本，平滑下线旧版本。示意图如下：




- **场景二：切分一定比例的流量到新版本**

  假设线上已运行了一套对外提供七层服务的 Service B，此时修复了一些问题，需要发布上线一个新的版本 Service B'，但又不想直接替换原有的 Service B，而是期望将 20%的流量切换到新版本 Service B'中。待运行一段时间稳定后，再将所有的流量从旧版本切换到新版本中，平滑下线旧版本。




## 注解说明

Nginx Ingress 支持通过配置注解（Annotations）来实现不同场景下的发布和测试，可以满足灰度发布、蓝绿发布、A/B 测试等业务场景。具体实现过程如下：为服务创建两个 Ingress，一个为常规 Ingress，另一个为带 nginx.ingress.kubernetes.io/canary: "true"注解的 Ingress，称为 Canary Ingress；为 Canary Ingress 配置流量切分策略 Annotation，两个 Ingress 相互配合，即可实现多种场景的发布和测试。Nginx Ingress 的 Annotation 支持以下几种规则：

- **nginx.ingress.kubernetes.io/canary-by-header**

  基于 Header 的流量切分，适用于灰度发布。如果请求头中包含指定的 header 名称，并且值为“always”，就将该请求转发给 Canary Ingress 定义的对应后端服务。如果值为“never”则不转发，可用于回滚到旧版本。如果为其他值则忽略该 annotation，并通过优先级将请求流量分配到其他规则。

- **nginx.ingress.kubernetes.io/canary-by-header-value**

  必须与 canary-by-header 一起使用，可自定义请求头的取值，包含但不限于“always”或“never”。当请求头的值命中指定的自定义值时，请求将会转发给 Canary Ingress 定义的对应后端服务，如果是其他值则忽略该 annotation，并通过优先级将请求流量分配到其他规则。

- **nginx.ingress.kubernetes.io/canary-by-header-pattern**

  与 canary-by-header-value 类似，唯一区别是该 annotation 用正则表达式匹配请求头的值，而不是某一个固定值。如果该 annotation 与 canary-by-header-value 同时存在，该 annotation 将被忽略。

- **nginx.ingress.kubernetes.io/canary-by-cookie**

  基于 Cookie 的流量切分，适用于灰度发布。与 canary-by-header 类似，该 annotation 用于 cookie，仅支持“always”和“never”，无法自定义取值。

- **nginx.ingress.kubernetes.io/canary-weight**

  基于服务权重的流量切分，适用于蓝绿部署。表示 Canary Ingress 所分配流量的百分比，取值范围[0-100]。例如，设置为 100，表示所有流量都将转发给 Canary Ingress 对应的后端服务。

**说明**

- 以上注解规则会按优先级进行评估，优先级为：canary-by-header -> canary-by-cookie -> canary-weight。
- 当 Ingress 被标记为 Canary Ingress 时，除了 nginx.ingress.kubernetes.io/load-balance 和 nginx.ingress.kubernetes.io/upstream-hash-by 外，所有其他非 Canary 的注解都将被忽略。
- 更多内容请参阅官方文档 Annotations。

## 前提条件

- 使用 Nginx Ingress 实现灰度发布的集群，需安装 nginx-ingress 插件作为 Ingress Controller，并且对外暴露统一的流量入口。
- 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 镜像包含新旧两个版本，欢迎页分别为“Old Nginx”和“New Nginx”。

## 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，并通过 Nginx Ingress 对外提供七层域名访问。

步骤 1 创建第一个版本的 Deployment 和 Service，本文以 old-nginx 为例。YAML 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: old-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: old-nginx
  template:
    metadata:
      labels:
        app: old-nginx
```

```
    spec:
      containers:
      - image: {your_repository}/nginx:old  # 容器使用的镜像为：nginx:old
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret

---

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

**步骤 2** 创建第二个版本的 Deployment 和 Service，本文以 new-nginx 为例。YAML 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: new-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: new-nginx
  template:
    metadata:
      labels:
        app: new-nginx
    spec:
      containers:
      - image: {your_repository}/nginx:new  # 容器使用的镜像为：nginx:new
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
```

```
    imagePullSecrets:
    - name: default-secret

---

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

您可以登录云容器引擎控制台看部署情况。

**步骤 3** 创建 Ingress，对外暴露服务，指向 old 版本的服务。YAML 示例如下：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: gray-release
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx    # 使用 Nginx 型 Ingress
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: old-nginx     # 指定后端服务为 old-nginx
              servicePort: 80
```

**步骤 4** 执行以下命令，进行访问验证。

```
curl -H "Host: www.example.com"  http://<EXTERNAL_IP>
```

其中，<EXTERNAL_IP>为 Nginx Ingress 对外暴露的 IP。

预期输出：

```
Old Nginx
```

**----结束**

## 步骤 2：灰度发布新版本服务

设置访问新版本服务的流量切分策略。云容器引擎 CCE 支持设置以下三种策略，实现灰度发布和蓝绿发布，您可以根据实际情况进行选择。

基于 Header 的流量切分、基于 Cookie 的流量切分、基于服务权重的流量切分

基于 Header、Cookie 和服务权重三种流量切分策略均可实现灰度发布；基于服务权重的流量切分策略，调整新服务权重为 100%，即可实现蓝绿发布。您可以在下述示例中了解具体使用方法。

**⚠ 注意**

**示例中，有以下两点需要注意：**

- **相同服务的 Canary Ingress 仅能够定义一个，从而使后端服务最多支持两个版本。**
- **即使流量完全切到了 Canary Ingress 上，旧版服务仍需存在，否则会出现报错。**

- **基于 Header 的流量切分**

  以下示例仅 Header 中包含 Region 且值为 bj 或 gz 的请求才能转发到新版本服务。

  1. 创建 Canary Ingress，指向新版本的后端服务，并增加 annotation。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"                    # 启用
Canary
    nginx.ingress.kubernetes.io/canary-by-header: "Region"
    nginx.ingress.kubernetes.io/canary-by-header-pattern: "bj|gz"    #
Header 中包含 Region 且值为 bj 或 gz 的请求转发到 Canary Ingress
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx      # 指定后端服务为 new-nginx
              servicePort: 80
```

  2. 执行以下命令，进行访问测试。

```
$ curl -H "Host: www.example.com" -H "Region: bj" http://<EXTERNAL_IP>
New Nginx
$ curl -H "Host: www.example.com" -H "Region: sh" http://<EXTERNAL_IP>
Old Nginx
$ curl -H "Host: www.example.com" -H "Region: gz" http://<EXTERNAL_IP>
New Nginx
$ curl -H "Host: www.example.com" http://<EXTERNAL_IP>
Old Nginx
```

  其中，<EXTERNAL_IP>为 Nginx Ingress 对外暴露的 IP。

可以看出，仅当 Header 中包含 Region 且值为 bj 或 gz 的请求才由新版本服务响应。

- **基于 Cookie 的流量切分**

  以下示例仅 Cookie 中包含 user_from_bj 的请求才能转发到新版本服务。

  1. 创建 Canary Ingress，指向新版本的后端服务，并增加 annotation。

  **说明**

  若您已在上述步骤创建 Canary Ingress，则请删除后再参考本步骤创建。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"                    # 启用
Canary
    nginx.ingress.kubernetes.io/canary-by-cookie: "user_from_bj"    # Cookie
中包含 user_from_bj 的请求转发到 Canary Ingress
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx     # 指定后端服务为 new-nginx
              servicePort: 80
```

  2. 执行以下命令，进行访问测试。

```
$ curl -s -H "Host: www.example.com" --cookie "user_from_bj=always"
http://<EXTERNAL_IP>
New Nginx
$ curl -s -H "Host: www.example.com" --cookie "user_from_gz=always"
http://<EXTERNAL_IP>
Old Nginx
$ curl -s -H "Host: www.example.com" http://<EXTERNAL_IP>
Old Nginx
```

  其中，<EXTERNAL_IP>为 Nginx Ingress 对外暴露的 IP。

  可以看出，仅当 Cookie 中包含 user_from_bj 且值为 always 的请求才由新版本服务响应。

- **基于服务权重的流量切分**

  **示例 1**：仅允许 20%的流量被转发到新版本服务中，实现灰度发布。

  1. 创建 Canary Ingress，并增加 annotation，将 20%的流量导入新版本后端服务。

  **说明**

  若您已在上述步骤创建 Canary Ingress，则请删除后再参考本步骤创建。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
```

```
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"        # 启用 Canary
    nginx.ingress.kubernetes.io/canary-weight: "20"    # 将 20%的流量转发到
Canary Ingress
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx      # 指定后端服务为 new-nginx
              servicePort: 80
```

2. 执行以下命令，进行访问测试。

```
$ for i in {1..20}; do curl -H "Host: www.example.com"
http://<EXTERNAL_IP>; done;
Old Nginx
Old Nginx
Old Nginx
New Nginx
Old Nginx
New Nginx
Old Nginx
New Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
New Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
```

其中，<EXTERNAL_IP>为 Nginx Ingress 对外暴露的 IP。

可以看出，有 4/20 的几率由新版本服务响应，符合 20%服务权重的设置。

**说明**

基于权重（20%）进行流量切分后，访问到新版本的概率接近 20%，流量比例可能会有小范围的浮动，这属于正常现象。

**示例 2**：允许所有的流量被转发到新版本服务中，实现蓝绿发布。

1. 创建 Canary Ingress，并增加 annotation，将 100%的流量导入新版本的后端服务。

**说明**

若您已在上述步骤创建 Canary Ingress，则请删除后再参考本步骤创建。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"         # 启用 Canary
    nginx.ingress.kubernetes.io/canary-weight: "100"    # 所有流量均转发到
Canary Ingress
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx     # 指定后端服务为 new-nginx
              servicePort: 80
```

2. 执行以下命令，进行访问测试。

```
$ for i in {1..10}; do curl -H "Host: www.example.com"
http://<EXTERNAL_IP>; done;
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
```

其中，<EXTERNAL_IP>为 Nginx Ingress 对外暴露的 IP。

可以看出，所有的访问均由新版本服务响应，成功实现了蓝绿发布。