云容器引擎

最佳实践

文档版本 01

发布日期 2024-05-31

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

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。	• 网络规划 • 10.1 集群网络地址段规划实践 • 如何设置CCE集群中的VPC网段和子网网段?
	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。	• VPC连接 • 10.1 集群网络地址段规划实践

类别	评估项目	类型	影响说明	FAQ&样例
	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。	• 如何设置安全组? • 如何加固CCE集群的节点VPC安全组规则?
	使用多控制节点模式，创建集群时将控制节点数设置为3。	可靠性	多控制节点模式开启后将创建三个控制节点，在单个控制节点发生故障后集群可以继续使用，不影响业务功能。商用场景建议选择多控制节点模式集群。	如何确认已创建的集群是否为多控制节点模式? 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： • CCE Standard集群支持选择“VPC网络”和“容器隧道网络”。 • CCE Turbo集群支持选择“云原生网络2.0”。	部署	集群创建成功后，网络模型不可更改，请谨慎选择。	容器网络模型对比
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 • 健康检查UDP协议安全组规则说明

类别	评估项目	类型	影响说明	FAQ&样例
	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调度策略。	可靠性	如设置单Pod副本数，当节点异常或实例异常会导致服务异常。为确保您的Pod能够调度成功，请确保您在设置调度规则后，节点有空余的资源用于容器的调度。	-
	合理设置“亲和性”和“反亲和性”	可靠性	对外提供服务的应用，如果以“或”的关系同时配置“亲和性”和“反亲和性”，应用升级或者重启后，会概率出现服务无法访问的问题。	亲和性/反亲和性调度策略概述 **反例：** 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。 **原因：** 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 • 在什么场景下设置工作负载生命周期中的“停止前处理”？

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据可靠性。	• 存储管理
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。	CCE支持的存储在持久化和多节点挂载方面的区别是怎样的?

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。	• 使用CCE需要关注哪些配额限制? • 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。 **反例:** 1. 用户升级了节点内核，可能会导致容器网络异常； 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； 3. 用户在节点上将/var/paas，/mnt/paas/kubernetes删除，导致该节点异常。

类别	评估项目	类型	影响说明	FAQ&样例
	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：** 1. 在弹性负载均衡页面修改CCE创建的监听器名称； 2. 在虚拟私有云页面修改CCE创建的安全组； 3. 在云硬盘页面删除或者卸载CCE集群节点挂载的数据盘； 4. 在统一身份认证页面删除cce的委托“cce_admin_trust”。 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。 ● 云监控服务AOM：CCE默认的基础资源监控，覆盖详细的容器相关指标，并提供告警配置能力。 ● 开源Prometheus：面向云原生应用程序的开源监控工具，并集成独立的告警系统，提供更高自由度的监控告警配置。	监控	未配置监控告警，将无法建立容器集群性能的正常标准，在出现异常时无法及时收到告警，需要人工巡检环境。	● **监控概述** ● **使用Prometheus插件监控**

2 容器化改造

2.1 企业管理应用容器化改造（ERP）

2.1.1 应用容器化改造方案概述

本手册基于云容器引擎实践所编写，用于指导您已有应用的容器化改造。

什么是容器

容器是操作系统内核自带能力，是基于Linux内核实现的轻量级高性能资源隔离机制。

云容器引擎CCE是基于开源Kubernetes的企业级容器服务，提供高可靠高性能的企业级容器应用管理服务，支持Kubernetes社区原生应用和工具，简化云上自动化容器运行环境搭建。

为什么需要使用容器

- 更高效的利用系统资源。

  容器不需要硬件虚拟化以及运行完整操作系统等额外开销，所以对系统资源利用率更高。相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

- 更快速的启动时间。

  容器直接运行于宿主机内核，无需启动完整的操作系统，可以做到秒级甚至毫秒级的启动时间。大大节约开发、测试、部署的时间。

- 一致的运行环境。

  容器镜像提供了完整的运行时环境，确保应用运行环境的一致性。从而不会再出现“这段代码在我机器上没问题”这类问题。

- 更轻松的迁移、维护和扩展。

  容器确保了执行环境的一致性，使得应用迁移更加容易。同时使用的存储及镜像技术，使应用重复部分的复用更为容易，基于基础镜像进一步扩展镜像也变得非常简单。

企业应用容器化改造方式

应用容器化改造，一般有以下三种方式：

- 方式一：单体应用整体容器化，应用代码和架构不做任何改动。
- 方式二：将应用中升级频繁，或对弹性伸缩要求高的组件拆分出来，将这部分组件容器化。
- 方式三：将应用做全面的微服务架构改造，再单独容器化。

这三种方式的优缺点如表2-1。

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： 单体应用整体容器化	• 业务0修改：应用架构和代码不需要做任何改动。 • 提升部署和升级效率：应用可构建为容器镜像，确保应用环境一致性，提升部署效率。 • 降低资源成本：容器对系统资源利用率高。相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。	• 整体性架构扩展难度大，随着应用程序代码扩展，更新和维护工作非常复杂。 • 推出新功能、语言、框架和技术都比较困难。
方式二： 先将部分组件容器化（将对弹性扩展要求高，或更新频繁的组件拆分出来，先容器化改造）	• 渐进式变革：在原有架构推倒重建太伤筋动骨，通过较为缓和的改动，更容易接受。 • 弹性更灵活：将对弹性要求高的组件容器化，当需要扩展时，只针对该容器扩展，弹性更灵活，且能降低系统资源。 • 新特性上线更快：将更新频繁的组件容器化，只针对这个容器进行升级，上线更快。	需要对业务做部分解耦拆分。

应用容器化改造方式	优点	缺点
方式三： 整体微服务架构改造，再容器化	• 单独扩展：拆分为微服务后，可单独增加或缩减每个微服务的实例数量。 • 提升开发速度：各微服务之间解耦，某个微服务的代码开发不影响其他微服务。 • 通过隔离确保安全：整体应用中，若存在安全漏洞，会获得所有功能的权限。微服务架构中，若攻击了某个服务，只可获得该服务的访问权限，无法入侵其他服务。 • 隔离崩溃：如果其中一个微服务崩溃，其它微服务还可以持续正常运行。	业务需要微服务化改造，改动较大。

本教程以“方式一”为例，将单体的企业ERP系统做整体的容器化改造。

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，资源的实际费用以华为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	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 16.04操作系统。 • 绑定弹性IP规格：按带宽计费，5Mbit/s。 • 建议选择按需计费。	1	0.8538元/小时

资源	资源说明	数量	费用（元）
云容器引擎CCE	• CCE集群版本：v1.21。 • 虚拟机节点规格：4核8G或以上规格、EulerOS2.9操作系统。 • 建议选择按需计费。	1	3.72元/小时
云硬盘EVS	• 云硬盘规格：100G。 • 建议选择按需计费。	1	0.1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。

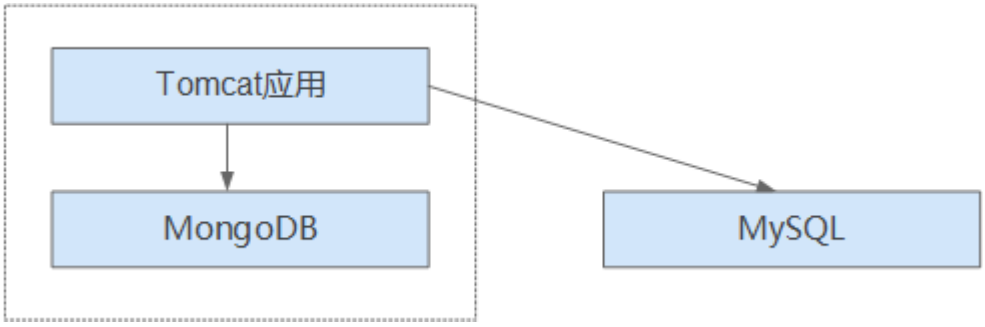
您不需要改动任何代码和架构，仅需将整体应用构建为容器镜像，部署到云容器引擎中。

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



如图2-1，该应用是标准的tomcat应用，后端对接了MongoDB和MySQL。这种类型应用可以先不做架构的拆分，将整体应用构建为一个镜像，将tomcat应用和mongoDB共同部署在一个镜像中。这样，当其他企业需要部署或升级应用时，可直接通过镜像来部署或升级。

- 对接mongoDB：用于用户文件存储。
- 对接MySQL：用于存储第三方企业数据，MySQL使用外部云数据库。

本例应用容器化改造价值

本例应用原先使用虚机方式部署，在部署和升级时，遇到了一系列的问题，而容器化部署解决了这些问题。

通过使用容器，您可以轻松打包应用程序的代码、配置和依赖关系，将其变成易于使用的构建块，从而实现环境一致性、运营效率、开发人员工作效率和版本控制等诸多目标。容器可以帮助保证应用程序快速、可靠、一致地部署，不受部署环境的影响。

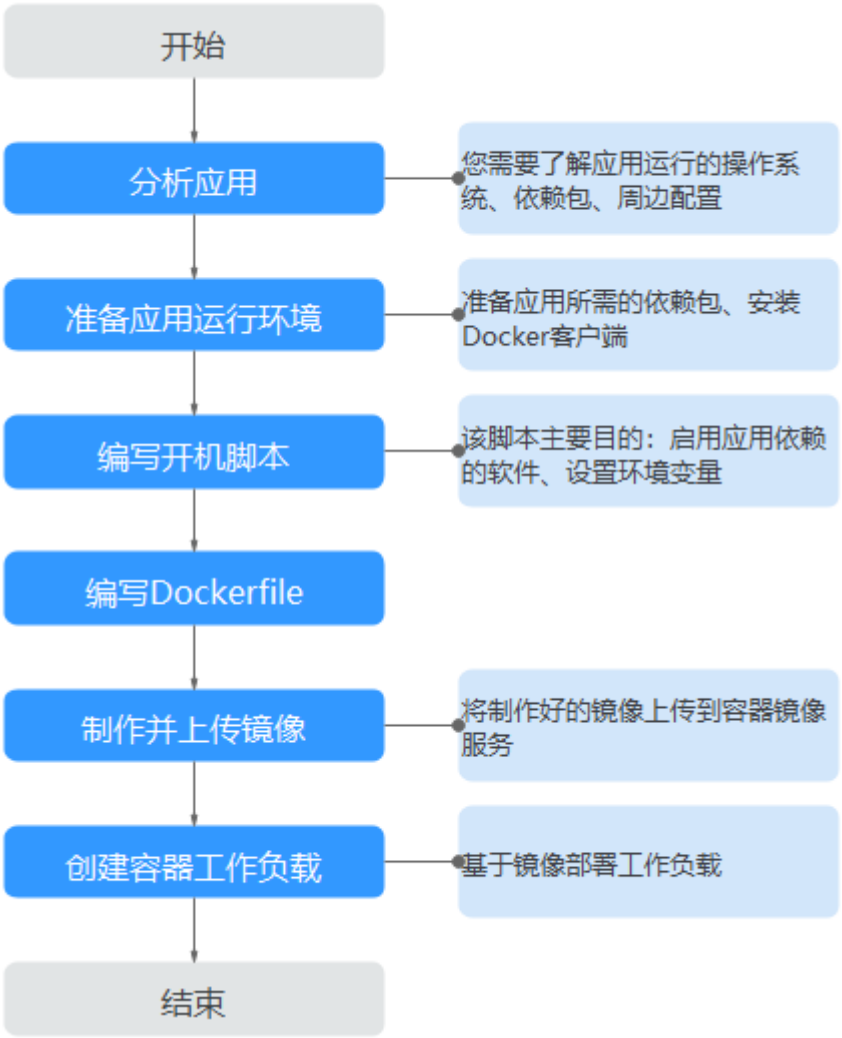
表 2-3 虚机和容器部署对比表

类别	before：虚机部署	after：容器部署
部署	部署成本高。 每给一家客户部署一套系统，就需要购置一台虚拟机。	成本降低50%以上。 通过容器服务实现了多租隔离，在同一台虚拟机上可以给多个企业部署系统。
升级	升级效率低。 版本升级时，需要逐台登录虚拟机手动配置升级，效率低且容易出错。	秒级升级。 通过更换镜像版本的方式，实现秒级升级。且CCE提供了滚动升级，使升级时业务不中断。
运维	运维成本高。 每给客户部署一套应用，就需要增加一台虚拟机的维护，随着客户量的增加，维护成本非常高。	自动化运维。 企业无需关注虚拟机的维护，只需要关注业务的开发。

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。需要了解的内容如表2-4。

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。 本例中，应用需要运行在centos:7.1操作系统上。
	运行环境	java应用需要jdk，go语言需要golang，web应用需要tomcat环境等，且需要确认对应版本号。 本例是tomcat类型的web应用，需要7.0版本的tomcat环境，且tomcat需要1.8版本的jdk。

类别	子类	说明
	依赖包	了解自己应用所需要的依赖包，类似openssl等系统软件，以及具体版本号。 本例不需要使用任何依赖包。
部署形态	周边配置	MongoDB：本例中MongoDB和Tomcat应用是在同一台机器中部署。因此对应配置可以固定，不需要将配置提取出来。

		应用需要对接哪些外部服务，例如数据库，文件存储等等。
		应用部署在虚拟机上时，该类配置需要每次部署时手动配置。容器化部署，可通过环境变量的方式注入到容器中，部署更为方便。
		本例需要对接MySQL数据库。您需要获取数据库的配置文件，如下“服务器地址”、“数据库名称”、“数据库登录用户名”和“数据库登录密码”将通过环境变量方式注入。
		url=jdbc:mysql://服务器地址/数据库名称 #数据库连接URL
		username=**** #数据库登录用户名
		password=**** #数据库登录密码

	自身配置	需要理出应用运行时的配置参数，哪些是需要经常变动的，哪些是不变的。 本例中，没有需要提取的自身配置项。 **说明** 为确保镜像无需经常更换，建议针对应用的各种配置进行分类。 • 经常变动的配置，例如周边对接信息、日志级别等，建议作为环境变量的方式来配置。 • 不变的配置，可以直接写到镜像中。

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，请以贵公司应用进行实际操作。

mkdir -p apache-tomcat-7.0.82/webapps/apptest

cp apptest.war apache-tomcat-7.0.82/webapps/apptest

cd apache-tomcat-7.0.82/webapps/apptest

./../../../jdk1.8.0_151/bin/jar -xf apptest.war

rm -rf apptest.war

----结束

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。

如何写出可读性更好的Dockerfile，请参见**编写高效的Dockerfile**。

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目录下的文件复制到容器目录下
COPY ./apache-tomcat-7.0.82 /root/apache-tomcat-7.0.82
# 将jdk1.8.0_151目录下的文件复制到容器目录下
COPY ./jdk1.8.0_151 /root/jdk1.8.0_151
# 将mongodb-linux-x86_64-rhel70-3.2.9目录下的文件复制到容器目录下
COPY ./mongodb-linux-x86_64-rhel70-3.2.9/bin /usr/local/mongodb/bin
# 将start_tomcat_and_mongo.sh复制到容器/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：把本机中的文件复制到容器中。
- 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，您需要设置好如**表2-5**中的环境。

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。 若您已有虚拟私有云，可重复使用，无需多次创建。 1. 登录管理控制台。 2. 在服务列表中，选择“网络 > 虚拟私有云”。 3. 在“总览”界面，单击“创建虚拟私有云”。 4. 根据界面提示创建虚拟私有云。如无特殊需求，界面参数均可保持默认。
2	创建密钥对	您需要新建一个密钥对，用于远程登录节点时的身份认证。若您已有密钥对，可重复使用，无需多次创建。 1. 登录管理控制台。 2. 在服务列表中，选择“数据加密服务 DEW”。 3. 选择左侧导航中的“密钥对管理”，选择“私有密钥对”，单击“创建密钥对”。 4. 输入密钥对名称，勾选“我同意将密钥对私钥托管”和“我已经阅读并同意《密钥对管理服务免责声明》”，单击“确定”。 5. 在弹出的对话框中，单击“确定”。 请根据提示信息，查看并保存私钥。为保证安全，私钥只能下载一次，请妥善保管，否则将无法登录节点。

步骤2 创建集群和节点。

1. 登录CCE控制台。在“集群管理”页面单击“购买集群”，选择需要创建的集群类型。

   填写集群参数，选择**步骤1**中创建的VPC。具体请参见**购买CCE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**创建节点**。

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“创建工作负载”。
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 容器镜像迁移

3.1.1 容器镜像迁移方案概述

应用现状

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。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。企业自运维大批镜像资源，意味着要付出高昂的运维、人力、管理成本，且效率不高。

容器镜像服务支持Linux、ARM等多架构容器镜像托管。企业可以将镜像仓库迁移到容器镜像服务，节省运维成本。

如何把已有的镜像仓库平滑地迁移到容器镜像服务？这里将介绍3种常见的方案，您可以根据自己的实际使用场景来选择。

迁移方案

表 3-1 迁移方案及适用场景对比

方案类型	适用场景	注意事项
3.1.2 使用docker命令将镜像迁移至SWR	待迁移的镜像数量较少	• 依赖磁盘存储，需要及时进行本地镜像的清理，而且落盘形成多余的时间开销，难以胜任生产场景中大量镜像的迁移。 • 依赖docker程序，docker daemon对pull/push的并发数进行了严格的限制，没法进行高并发同步。 • 一些功能只能经过HTTP api进行操作，单纯使用docker cli 没法做到，使脚本变得复杂。
3.1.3 使用image-migrator将镜像迁移至SWR	待迁移的镜像数量庞大	• 支持多对多镜像仓库同步。 • 支持基于Docker Registry V2搭建的docker镜像仓库服务 (如Docker Hub、Quay、Harbor等)。 • 同步只通过内存和网络，不依赖磁盘存储，同步速度快。 • 增量同步, 经过对同步过的镜像blob信息落盘，不重复同步已同步的镜像。 • 并发同步，能够经过配置文件调整并发数。 • 自动重试失败的同步任务，能够解决大部分镜像同步中的网络抖动问题。 • 不依赖 docker 以及其余程序。
3.1.4 跨云Harbor同步镜像至华为云SWR	部分客户存在多云场景，并且使用某一家云上的自建Harbor作为镜像仓库	仅支持 Harbor V1.10.5 及以上版本

3.1.2 使用 docker 命令将镜像迁移至 SWR

场景描述

容器镜像服务提供了简便、易用的镜像托管和高效分发业务。当要迁移的镜像数量较少时，企业可以通过简单的docker pull、docker push命令行，将之前维护的镜像迁移到SWR上。

操作步骤

步骤1 从源仓库下载镜像。

使用docker pull命令下载镜像。

示例：**docker pull nginx:latest**

使用**docker images**命令查看是否下载成功。

```
# docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
nginx                         latest    22f2bf2e2b4f   5 hours ago     22.8MB
```

步骤2 将**步骤1**中下载的镜像上传到SWR。

1. 登录到目标端容器所在虚拟机，并登录SWR。详细步骤请参考**客户端上传镜像**。
2. 给镜像打标签。

   docker tag [镜像名称:版本名称] [镜像仓库地址]/[组织名称]/[镜像名称:版本名称]

   示例：

   docker tag nginx:v1 swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/nginx:v1
3. 上传镜像至目标镜像仓库。

   docker push [镜像仓库地址]/[组织名称]/[镜像名称:版本名称]

   示例：

   docker push swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/nginx:v1
4. 终端显示如下信息，表明上传镜像成功。

   ```
   fbce26647e70: Pushed
   fb04ab8effa8: Pushed
   8f736d52032f: Pushed
   009f1d338b57: Pushed
   678bbd796838: Pushed
   d1279c519351: Pushed
   f68ef921efae: Pushed
   v1: digest: sha256:0cdfc7910db531bfa7726de4c19ec556bc9190aad9bd3de93787e8bce3385f8d size: 1780
   ```

   返回容器镜像服务控制台，在“我的镜像”页面，执行刷新操作后可查看到对应的镜像信息。

----结束

3.1.3 使用 image-migrator 将镜像迁移至 SWR

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，建议您将自建镜像仓库迁移至华为云容器镜像服务（SWR）。

image-migrator是一个镜像迁移工具，能够自动将基于Docker Registry v2搭建的Docker镜像仓库中的镜像迁移到SWR中。

准备工作

在开始迁移之前，请确保您已准备了一台安装了kubectl的服务器，用于连接源集群和目标集群。该服务器需要至少拥有5GB左右的本地磁盘空间和≥8G的内存，以确保迁移工具可以正常运行，并存储相关数据，如源集群的采集数据和目标集群的推荐数据等。

迁移工具支持在Linux（x86、arm）、Windows环境中运行，因此您可以在这些操作系统中任选一种作为服务器的操作系统。

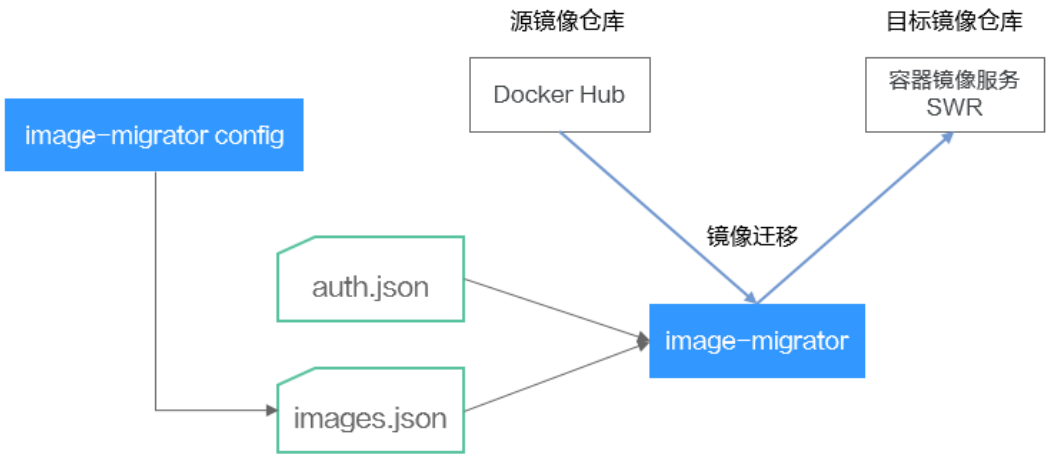
对于Linux操作系统来说，使用image-migrator前，需要运行chmod u+x 工具名 命令（例如chmod u+x image-migrator-linux-amd64），授予可执行权限。

表 3-2 image-migrator 工具包获取

image-migrator	image-migrator是一个镜像迁移工具，能够自动将基于Docker Registry v2搭建的Docker镜像仓库或第三方云镜像仓库中的镜像迁移到SWR中。	Linux x86：**https://ucs-migration.obs.cn-north-4.myhuaweicloud.com/toolkits/image-migrator-linux-amd64** Linux arm：**https://ucs-migration.obs.cn-north-4.myhuaweicloud.com/toolkits/image-migrator-linux-arm64** Windows：**https://ucs-migration.obs.cn-north-4.myhuaweicloud.com/toolkits/image-migrator-windows-amd64.exe**

image-migrator 工作原理

图 3-1 image-migrator 工作原理



使用image-migrator工具将镜像迁移到SWR时，需要准备两个文件，一个为镜像仓库访问权限文件“auth.json”，该文件中的两个对象分别为源镜像仓库和目标镜像仓库

（即Registry）的账号和密码；另一个为镜像列表文件“images.json”，文件内容由多条镜像同步规则组成，每一条规则包括一个源镜像仓库（键）和一个目标镜像仓库（值）。将这两个文件准备好以后，放在image-migrator工具所在目录下，执行一条简单的命令，就可以完成镜像的迁移。关于两个文件的详细介绍如下：

- **“auth.json”文件**

  “auth.json”为镜像仓库访问权限文件，其中每个对象为一个registry的用户名和密码。通常源镜像仓库需要具有pull以及访问tags权限，目标镜像仓库需要拥有push以及创建仓库权限。如果是匿名访问镜像仓库，则不需要填写用户名、密码等信息。“auth.json”文件的结构如下：

  ```
  {
      "源镜像仓库地址": { },
      "目标镜像仓库地址": {
          "username": "xxxxxx",
          "password": "xxxxxx",
          "insecure": true
      }
  }
  ```

  其中，

  - “源镜像仓库地址”和“目标镜像仓库地址”支持“registry”和“registry/namespace”两种形式，需要跟下述“images.json”中的registry或registry/namespace对应。images中被匹配到的url会使用对应用户名密码进行镜像同步，优先匹配“registry/namespace”的形式。

    目标镜像仓库地址如果为“registry”形式，可以从SWR控制台页面获取，具体方法如下：在“总览”页面单击右上角“登录指令”，登录指令末尾的域名即为SWR镜像仓库地址，例如swr.cn-north-4.myhuaweicloud.com。注意每个Region的地址不同，请切换到对应Region获取。如果为“registry/namespace”形式，还要将namespace替换为SWR的组织名称。
  - username：（可选）用户名，values可以填写具体取值，也可以使用“${env}”或者“$env”类型的字符串引用环境变量。
  - password：（可选）密码，values可以填写具体取值，也可以使用“${env}”或者“$env”类型的字符串引用环境变量。
  - insecure：（可选）registry是否为http服务，如果是，insecure为true；默认是false。

  说明

  目标镜像仓库SWR的用户名形式为：区域项目名称@AK；密码为AK和SK经过加密处理后的登录密钥，详细指导请参考**获取长期有效登录指令**。

  示例：

  ```
  {
      "quay.io/coreos": { },
      "swr.cn-north-4.myhuaweicloud.com": {
          "username": "cn-north-4@RVHVMX******",
          "password": "cab4ceab4a1545***************",
          "insecure": true
      }
  }
  ```

- **“images.json”文件**

  该文件本质上是一个待迁移的镜像清单，由多条镜像同步规则组成，每一条规则包括一个源镜像仓库（键）和一个目标镜像仓库（值）。具体的要求如下：

  a. 同步的最大单位是仓库（repository），不支持通过一条规则同步整个namespace以及registry。

b. 源仓库和目标仓库的格式与docker pull/push命令使用的镜像url类似（registry/namespace/repository:tag）。
c. 源仓库和目标仓库（如果目标仓库不为空字符串）都至少包含registry/namespace/repository。
d. 源仓库字段不能为空，如果需要将一个源仓库同步到多个目标仓库需要配置多条规则。
e. 目标仓库名可以和源仓库名不同，此时同步功能类似于：docker pull + docker tag + docker push。
f. 当源仓库字段中不包含tag时，表示将该仓库所有tag同步到目标仓库，此时目标仓库不能包含tag。
g. 当源仓库字段中包含tag时，表示只同步源仓库中的一个tag到目标仓库，如果目标仓库中不包含tag，则默认使用源tag。
h. 当目标仓库为空字符串时，会将源镜像同步到默认registry的默认namespace下，并且repository以及tag与源仓库相同，默认registry和默认namespace可以通过命令行参数以及环境变量配置。

示例如下：

```
{
    "quay.io/coreos/etcd:1.0.0": "swr.cn-north-4.myhuaweicloud.com/test/etcd:1.0.0",
    "quay.io/coreos/etcd": "swr.cn-north-4.myhuaweicloud.com/test/etcd",
    "quay.io/coreos/etcd:2.7.3": "swr.cn-north-4.myhuaweicloud.com/test/etcd"
}
```

我们提供了一个自动获取集群中工作负载正在使用的镜像的方法，即image-migrator工具的config子命令，具体用法请参见**image-migrator config使用方法**。得到images.json文件后，您还可以根据需要进行修改、添加或删除。

image-migrator 使用方法

说明

image-migrator工具支持在Linux（x86、arm）和Windows环境中运行，使用方法相似。本文将以Linux（x86）环境为例进行介绍。

若使用Linux（arm）或Windows环境，请将下述命令中的**image-migrator-linux-amd64**分别替换为**image-migrator-linux-arm64**或**image-migrator-windows-amd64.exe**。

在image-migrator工具所在目录下执行**./image-migrator-linux-amd64 -h**，可以查看image-migrator工具的使用方法。

- --auth：指定auth.json的路径，默认在image-migrator所在目录下。
- --images：指定images.json的路径，默认在image-migrator所在目录下。
- --log：指定image-migrator生成日志的路径，默认是image-migrator当前目录下的image-migrator.log。
- --namespace：默认的目标仓库的namespace，也就是说，如果images.json中没有指定目标仓库中的namespace，可以在执行迁移命令时指定。
- --registry：默认的目标仓库的registry，也就是说，如果images.json中没有指定目标仓库中的registry，可以在执行迁移命令时指定。
- --retries：迁移失败时的重试次数，默认为3。
- --workers：镜像搬迁的worker数量（并发数），默认是7。

```
$ ./image-migrator-linux-amd64 -h
A Fast and Flexible docker registry image images tool implement by Go.

Usage:
```



```
  image-migrator [flags]

Aliases:
  image-migrator, image-migrator

Flags:
      --auth string        auth file path. This flag need to be pair used with --images. (default "./auth.json")
  -h, --help               help for image-migrator
      --images string      images file path. This flag need to be pair used with --auth (default "./images.json")
      --log string         log file path (default "./image-migrator.log")
      --namespace string   default target namespace when target namespace is not given in the images config file, can also be set with DEFAULT_NAMESPACE environment value
      --registry string    default target registry url when target registry is not given in the images config file, can also be set with DEFAULT_REGISTRY environment value
  -r, --retries int        times to retry failed tasks (default 3)
  -w, --workers int        numbers of working goroutines (default 7)

$./image-migrator --workers=5 --auth=./auth.json --images=./images.json --namespace=test \
--registry=swr.cn-north-4.myhuaweicloud.com --retries=2
$ ./image-migrator
Start to generate images tasks, please wait ...
Start to handle images tasks, please wait ...
Images(38) migration finished, 0 images tasks failed, 0 tasks generate failed
```

示例如下：

./image-migrator --workers=5 --auth=./auth.json --images=./images.json --namespace=test --registry=swr.cn-north-4.myhuaweicloud.com --retries=2

该命令表示将“images.json”文件中的镜像迁移至“swr.cn-north-4.myhuaweicloud.com/test”镜像仓库下，迁移失败时可以重试2次，一次可以同时搬迁5个镜像。

image-migrator config 使用方法

image-migrator工具的config子命令可用于获取集群应用中使用的镜像，在工具所在目录下生成images.json。执行**./image-migrator-linux-amd64 config -h**命令可以查看config子命令的使用方法。

- -k, --kubeconfig：指定kubectl的kubeConfig位置，默认是$HOME/.kube/config。kubeConfig文件：用于配置对Kubernetes集群的访问，KubeConfig文件中包含访问注册kubernetes集群所需要的认证凭据以及Endpoint（访问地址），详细介绍可参见**Kubernetes文档**。
- -n, --namespaces：指定获取镜像的命名空间，多个命名空间用逗号分隔（如：ns1,ns2,ns3），默认是""，表示获取所有命名空间的镜像。
- -t, --repo：指定目标仓库的地址（registry/namespace）。

```
$ ./image-migrator-linux-amd64 config -h
generate images.json

Usage:
   image-migrator config [flags]

Flags:
   -h, --help               help for config
   -k, --kubeconfig string   The kubeconfig of k8s cluster's. Default is the $HOME/.kube/config. (default "/root/.kube/config")
   -n, --namespaces string   Specify a namespace for information collection. If multiple namespaces are specified, separate them with commas (,), such as ns1,ns2. default("") is all namespaces
   -t, --repo string         target repo,such as swr.cn-north-4.myhuaweicloud.com/test
```

示例如下：

- 指定一个命名空间

  ./image-migrator-linux-amd64 config -n default -t swr.cn-north-4.myhuaweicloud.com/test

- 指定多个命名空间

  ./image-migrator-linux-amd64 config -n default,kube-system -t swr.cn-north-4.myhuaweicloud.com/test

- 不指定命名空间（表示获取所有命名空间的镜像）

  ./image-migrator-linux-amd64 config -t swr.cn-north-4.myhuaweicloud.com/test

镜像迁移操作步骤

步骤1 准备镜像仓库访问权限文件：auth.json。

新建一个auth.json文件，并按照格式修改，如果是匿名访问仓库，则不需要填写用户名、密码等信息。将文件放置在image-migrator所在目录下。

示例：

```
{
    "quay.io/coreos": { },
    "swr.cn-north-4.myhuaweicloud.com": {
        "username": "cn-north-4@RVHVMX******",
        "password": "cab4ceab4a1545***************",
        "insecure": true
    }
}
```

详细的参数说明请参见“**auth.json**”**文件**。

步骤2 准备镜像列表文件：images.json。

1. 通过kubectl连接源集群。具体方法可参考**使用kubectl连接集群**。
2. 执行镜像迁移config子命令，生成images.json文件。

   您可以参考**image-migrator config使用方法**中的方法和示例，不指定命名空间，或者指定一个、多个命名空间来获取源集群应用中使用的镜像。
3. 根据需求调整images.json文件内容，但要遵循“**images.json**”**文件**中所讲的八项要求。

步骤3 镜像迁移。

您可以执行默认的**./image-migrator-linux-amd64**命令进行镜像迁移，也可以根据需要设置image-migrator的参数。

例如以下命令：

./image-migrator-linux-amd64 --workers=5 --auth=./auth.json --images=./images.json --namespace=test --registry=swr.cn-north-4.myhuaweicloud.com --retries=2

示例：

```
$ ./image-migrator-linux-amd64
Start to generate images tasks, please wait ...
Start to handle images tasks, please wait ...
Images(38) migration finished, 0 images tasks failed, 0 tasks generate failed
```

步骤4 结果查看。

上述命令执行完毕后，回显如下类似信息：

```
Images(38) migration finished, 0 images tasks failed, 0 tasks generate failed
```

表示按照配置，成功将38个镜像迁移到SWR仓库中。

----结束

3.1.4 跨云 Harbor 同步镜像至华为云 SWR

场景描述

部分客户存在多云场景，并且使用某一家云上的自建Harbor作为镜像仓库。跨云Harbor同步镜像至SWR存在两种场景：

1. Harbor可以通过公网访问SWR，配置方法参见**公网访问场景**。
2. 通过专线打通Harbor到VPC间的网络，使用VPC终端节点访问SWR，配置方法参见**专线打通场景**。

背景知识

Harbor是VMware公司开源的企业级Docker Registry管理项目，在开源Docker Distribution能力基础上扩展了例如权限管理（RBAC）、镜像安全扫描、镜像复制等功能。Harbor目前已成为自建容器镜像托管及分发服务的首选。

公网访问场景

步骤1 Harbor上配置镜像仓库。

说明

Harbor在1.10.5以上版本，集成了华为云的SWR对接，只需要在目标（ENDPOINT）上选择就可以。本文以Harbor 2.4.1为例。

1. 新建目标。

2. 填写如下参数。

- 提供者：必须选“Huawei SWR”。
- 目标名：自定义。
- 目标URL：使用SWR的公网域名，格式为https://{*SWR镜像仓库地址*}。镜像仓库地址获取方法：登录容器镜像服务控制台，进入“我的镜像”，单击“客户端上传”，在弹出的页面即可查看SWR当前Region的镜像仓库地址。
- 访问ID：遵循SWR的长期有效的认证凭证规则，以“区域项目名称@[AK]”形式填写。
- 访问密码：遵循SWR的长期有效的认证凭证规则，需要用AK和SK来生成，详细说明请参考**获取长期有效登录指令**。
- 验证远程证书：建议**取消勾选**。

步骤2 配置同步规则。

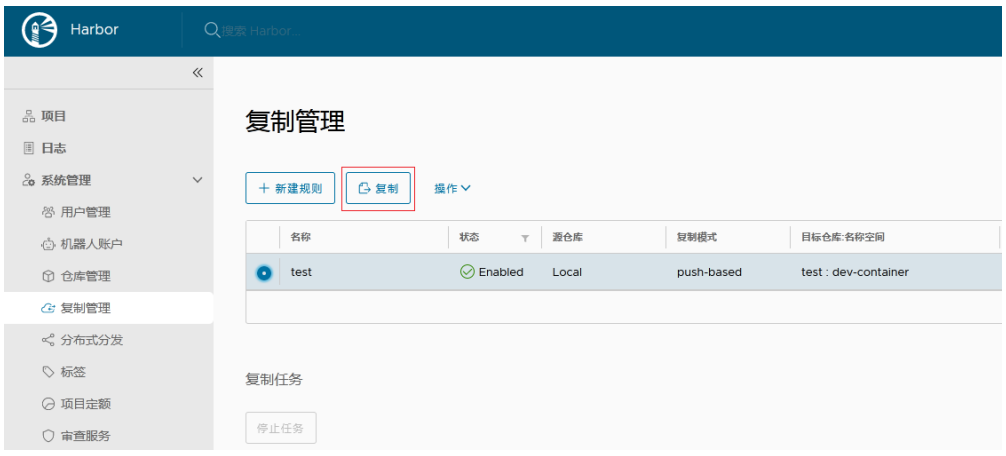
1. 新建规则

2. 填写如下参数。
   - 名称：自定义。
   - 复制模式：选择“Push-based”，表示把镜像由本地Harbor推送到远端仓库。
   - 源资源过滤器：根据填写的规则过滤Harbor上的镜像。
   - 目标仓库：选择**步骤1**中创建的目标。
   - 目标

     名称空间：填写SWR上的组织名称。

     仓库扁平化：用以在复制镜像时减少仓库的层级结构，建议选择“替换所有级”。假设Harbor仓库的层级结构为“library/nginx”，目标名称空间为dev-container，“替换所有级”对应的结果为：library/nginx -> dev-container/nginx。
   - 触发模式：选择“手动”。
   - 带宽：设置执行该条同步规则时的最大网络带宽，“-1”表示无限制。

步骤3 创建完成后，选中后单击“复制”即可完成同步。



----结束

专线打通场景

步骤1 配置VPC终端节点。

步骤2 获取VPC内网访问IP及域名（华为云VPC内默认会自动加域名解析规则，不需要配置hosts；非华为云节点需要配置hosts），可以在终端节点详情页的“内网域名”中查询。

步骤3 Harbor上配置镜像仓库。

说明

Harbor在1.10.5以上版本，集成了华为云的SWR对接，只需要在目标（ENDPOINT）上选择就可以。本文以Harbor 2.4.1为例。

1. 新建目标。
2. 填写如下参数。
   - 提供者：必须选“Huawei SWR”。
   - 目标名：自定义。
   - 目标URL：**使用VPC终端节点中的内网域名**，必须以https开头，同时Harbor所在的容器内要配置域名映射。
   - 访问ID：遵循SWR的长期有效的认证凭证规则，以“区域项目名称@[AK]”形式填写。
   - 访问密码：遵循SWR的长期有效的认证凭证规则，需要用AK和SK来生成，详细说明请参考**获取长期有效登录指令**。
   - 验证远程证书：必须**取消勾选**。

步骤4 配置同步规则。

1. 新建规则

2. 填写如下参数。

- 名称：自定义。
- 复制模式：选择“Push-based”，表示把镜像由本地Harbor推送到远端仓库。
- 源资源过滤器：根据填写的规则过滤Harbor上的镜像。
- 目标仓库：选择**步骤3**中创建的目标。
- 目标

  名称空间：填写SWR上的组织名称。

  仓库扁平化：用以在复制镜像时减少仓库的层级结构，建议选择“替换所有级”。假设Harbor仓库的层级结构为“library/nginx”，目标名称空间为dev-container，“替换所有级”对应的结果为：library/nginx -> dev-container/nginx。

- 触发模式：选择“手动”。
- 带宽：设置执行该条同步规则时的最大网络带宽，“-1”表示无限制。

步骤5 创建完成后，选中后单击“复制”即可完成同步。

----结束

3.2 将 K8s 集群迁移到 CCE

3.2.1 自建 K8s 集群迁移方案概述

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，深度集成应用服务网格和Helm标准模板，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，您可参考**云容器引擎与自建Kubernetes集群对比**进行额外的了解。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
- 停止维护，无法原地升级的需要迁移的CCE集群

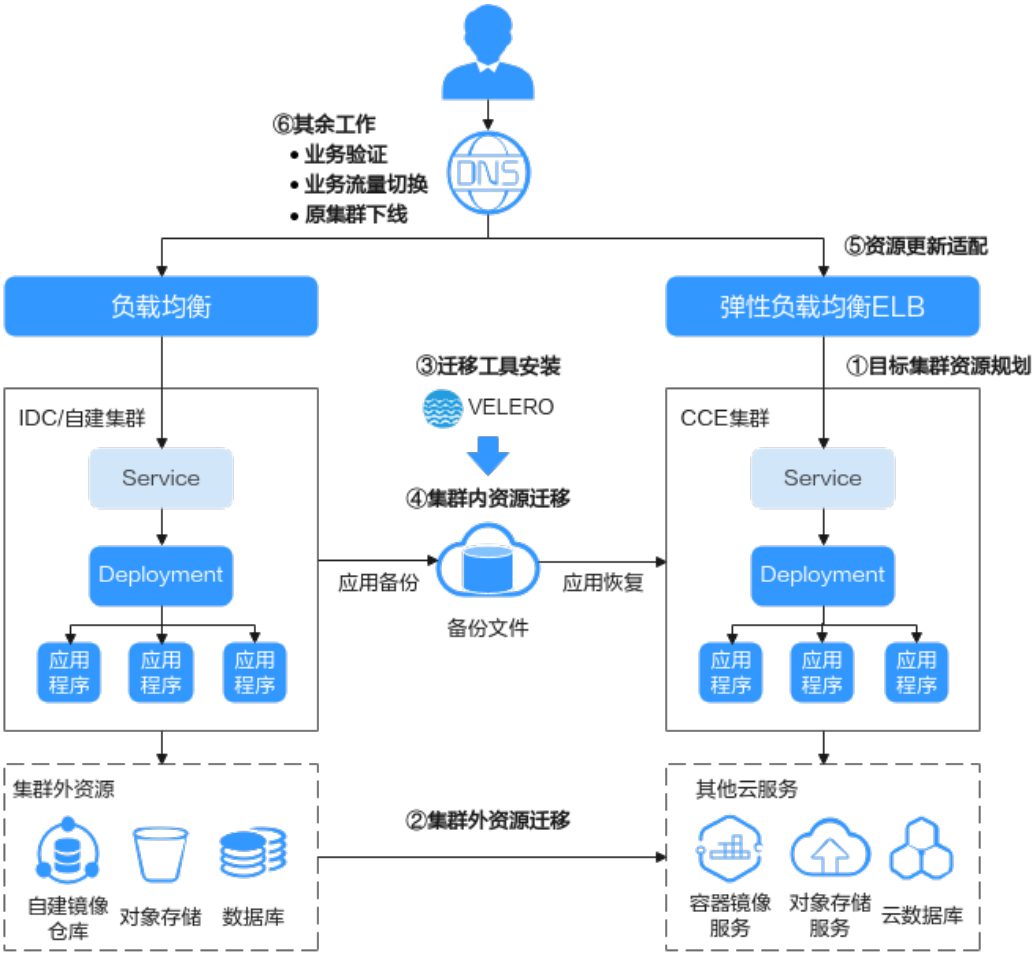
在迁移前，需对原集群的所有资源进行分析再决定迁移方案，可迁移的资源包括集群内资源和集群外资源，如下表所示。

表 3-3 可迁移资源列表

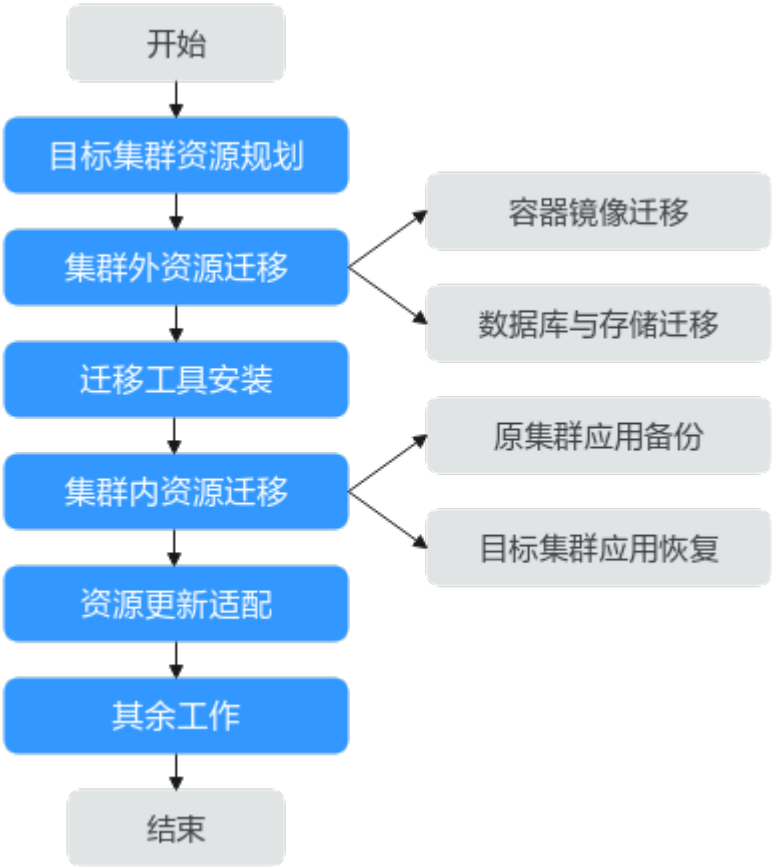
资源类别	可迁移对象	备注
集群内资源	集群中的所有对象，Pod、Job、Service、Deployment、ConfigMap等。	**不建议迁移的资源：**命名空间velero和kube-system下的资源。 ● velero：该命名空间下的资源为迁移工具创建，无需迁移。 ● kube-system：该命名空间下的资源为系统资源。如原集群该命名空间下包含用户自行创建的资源，建议按需迁移。 **注意** 如果您是迁移或备份CCE中集群的资源，比如从一个Namespace到另外一个Namespace，请不要备份名称为paas.elb的Secret。因为paas.elb的内容是会定期更新，备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**图3-2**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 3-2 迁移方案示意图



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**3.2.2 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**3.2.3.1 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**3.2.3.2 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**3.2.3.3 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

3.2.2 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。**表3-4**中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，**表3-4**中带“*”号的资源参数将不可更改，请谨慎选择。

表 3-4 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。 • **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。 • **容器隧道网络（Overlay）：**基于底层VPC网络，另构建了独立的VXLAN隧道化容器网络，适用于一般场景。 • **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。	VPC网络

资源	主要性能参数	参数说明	本示例规划
	*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。 • **1**：单个控制节点，不建议在商用场景使用。	3
节点	OS类型	• **EulerOS** • **CentOS** • **Ubuntu**	EulerOS
	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。 • **内存优化型**：该类型实例提供内存比例更高的实例，可以用于对内存要求较高、数据量大的工作负载，例如关系数据库、NoSQL等场景。 • **通用入门型**：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。 • **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。 • **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。 • **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。 • **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。 • **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。 • **AI加速型**：AI加速型节点实例，搭载高性能、低功耗的海思Ascend 310 AI处理器，实现快速高效地处理推理和图像识别等工作，适用于图像识别、视频处理、推理计算以及机器学习等场景。	通用型（节点规格为4U8G）
	系统盘类型	• **高IO**：后端存储介质为SAS类型。 • **超高IO**：后端存储介质为SSD类型。	高IO

资源	主要性能参数	参数说明	本示例规划
	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。 • **文件存储卷：** CCE支持创建SFS存储卷并挂载到容器的某一路径下，也可以使用底层SFS服务创建的文件存储卷，SFS存储卷适用于多读多写的持久化存储，适用于多种工作负载场景，包括媒体处理、内容管理、大数据分析和分析工作负载程序等场景。 • **对象存储卷：**CCE支持创建OBS对象存储卷并挂载到容器的某一路径下，对象存储适用于云工作负载、数据分析、内容分析和热点对象等场景。 • **极速文件存储卷：**CCE支持创建SFS Turbo极速文件存储卷并挂载到容器的某一路径下，极速文件存储具有按需申请，快速供给，弹性扩展，方便灵活等特点，适用于DevOps、容器微服务、企业办公等应用场景。	云硬盘存储卷

3.2.3 实施步骤

3.2.3.1 集群外资源迁移

若您的集群不涉及表3-3中的集群外资源，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。

您可以按以下操作手动完成容器镜像的迁移。

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 swr.ap-southeast-1.myhuaweicloud.com/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 swr.ap-southeast-1.myhuaweicloud.com/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**跨云数据库在线迁移**。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**对象存储迁移服务OMS**，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**创建单个迁移任务**。完成迁移后，可参考**对接已有对象存储**挂载到应用实例。

说明

目前对象存储迁移服务OMS支持亚马逊云（中国）、阿里云、微软云、百度云、华为云、金山云、优刻得、青云、七牛云、腾讯云平台的对象存储数据迁移到华为云**对象存储服务OBS**。

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。

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，请参考**获取访问密钥（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（可选）

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**。

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5**。

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node-agent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 1.13.1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 登录一台可以访问公网的虚拟机，并使用kubectl连接需要安装Velero的集群。

步骤2 下载Velero 1.13.1版本的二进制文件。

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.13.1/velero-v1.13.1-linux-amd64.tar.gz
```

步骤3 安装Velero客户端。

```
tar -xvf velero-v1.13.1-linux-amd64.tar.gz
cp ./velero-v1.13.1-linux-amd64/velero /usr/local/bin
```

步骤4 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。使用OBS时，可参考**获取访问密钥（AK/SK）**获取AK/SK。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5 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:v1.9.1 \
  --bucket velero \
  --secret-file ./credentials-velero \
  --use-node-agent \
```



```
 --use-volume-snapshots=false \
 --backup-location-config region=ap-southeast-1,s3ForcePathStyle="true",s3Url=http://obs.ap-southeast-1.myhuaweicloud.com
```

表 3-5 Velero 安装参数说明

安装参数	参数说明
--provider	声明使用的插件类型为AWS S3组件。
--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4中创建的“credentials-velero”文件。
--use-node-agent	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。 • OBS：请根据实际区域填写，如“ap-southeast-1”。 • 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。 • OBS：该参数值需根据对象存储桶地域决定，参数值为“http://obs.{region}.myhuaweicloud.com”。例如区域为香港（ap-southeast-1），则参数值为“http://obs.ap-southeast-1.myhuaweicloud.com”。 • 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”。 **说明** – 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。 – 访问集群外安装的MinIO时，需填写其公网IP地址。

步骤6 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
node-agent-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node-agent和Velero分配的CPU和内存大小。

步骤7 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对存储卷进行备份还原时**不支持HostPath类型**的存储卷，详情请参见**文件系统备份限制**。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-fs-backup**：表示使用PV备份工具对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-node-agent”后可用。
  ```
  velero backup create <backup-name> --default-volumes-to-fs-backup
  ```
- **--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-fs-backup
```

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.
```

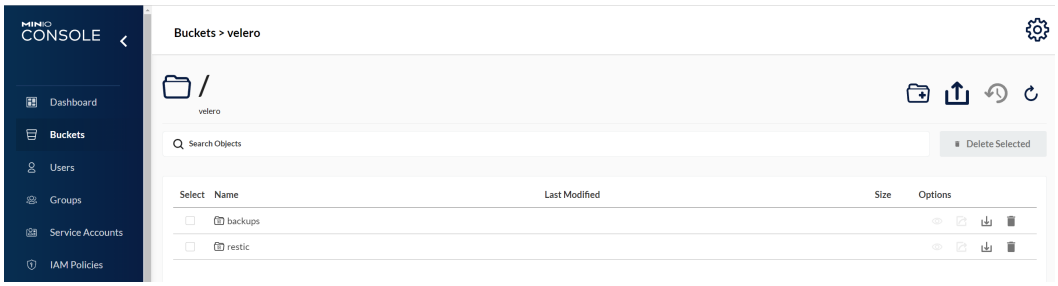
步骤3 查看备份情况。

```
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，另一路径为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**。若您使用对象存储迁移服务OMS完成存储迁移，可参考**对接已有对象存储**将对象存储桶挂载到应用实例。

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**3.2.3.4 资源更新适配**中的步骤排查解决。

----结束

3.2.3.4 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：

```
'swr.{区域}.myhuaweicloud.com/{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**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**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 3-6 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

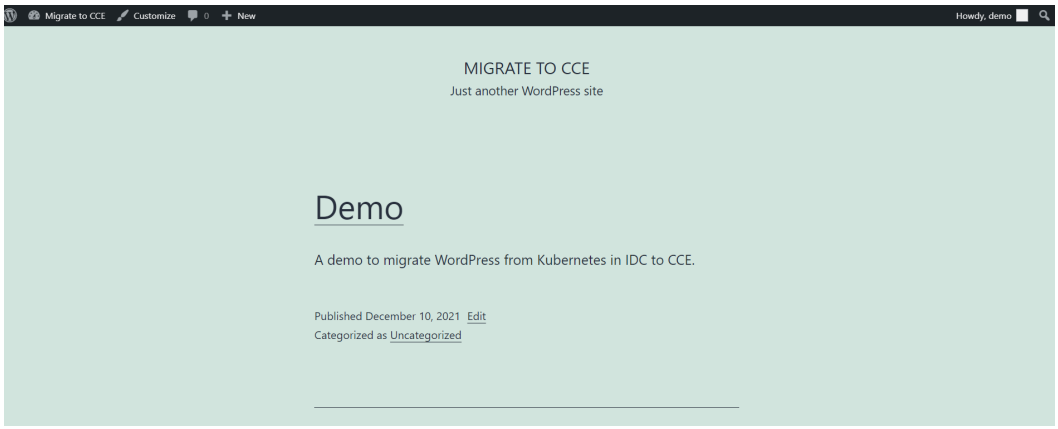
3.2.3.5 其余工作

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。

3.2.3.6 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-node-agent \
  --node-agent-pod-cpu-request 500m \
  --node-agent-pod-mem-request 1Gi \
  --node-agent-pod-cpu-limit 1000m \
  --node-agent-pod-mem-limit 1Gi
```

安装Velero后：

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

```
kubectl edit deploy velero -n velero
kubectl edit ds node-agent -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

4 DevOps

4.1 在 CCE 中安装部署 Jenkins

4.1.1 在 CCE 中安装部署 Jenkins 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，参见**表4-1**。

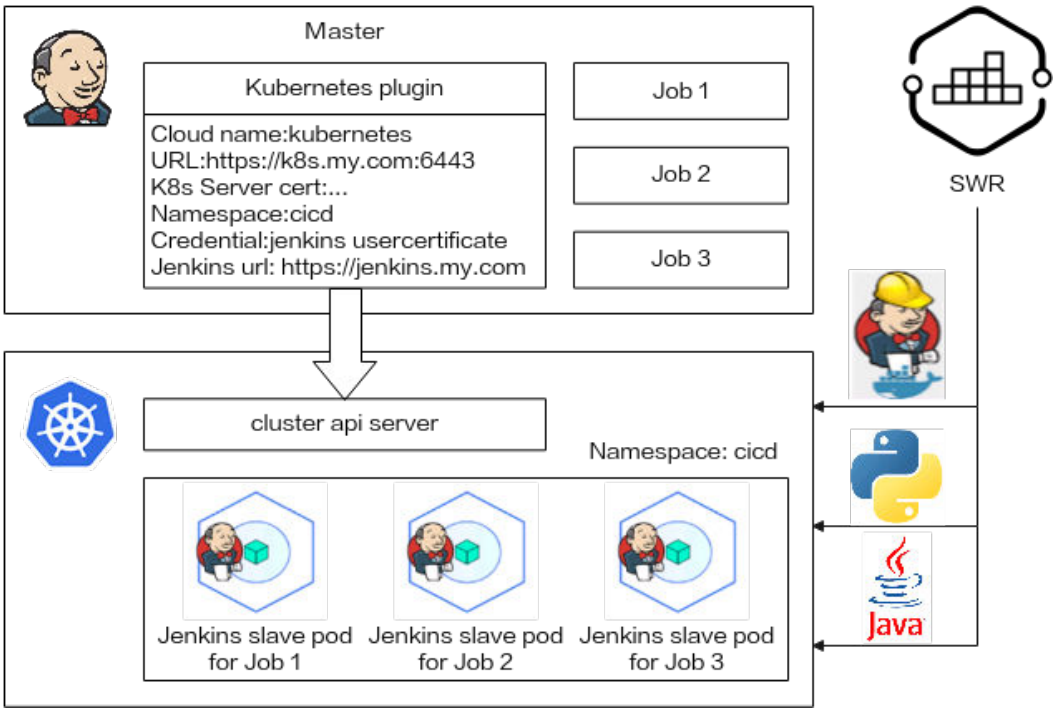
表 4-1 Jenkins 部署模式

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。 • 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。 • 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。 • 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。 • 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。 • 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，具体架构如**图4-1**所示。

- 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 **4.1.3.1 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 **4.1.3.2 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 **4.1.3.3 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

本文提供的成本预估费用仅供参考，资源的实际费用以华为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 资源和成本规划

资源	资源说明	数量	费用（元）
云容器引擎 CCE	建议选择按需计费。 • 集群类型：CCE集群 • 集群版本：v1.25 • 集群规模：50节点 • 高可用：是	1	2.91元/小时
虚拟机节点	建议选择按需计费。 • 虚拟机节点类型：通用计算增强型 • 节点规格：4核 | 8GiB • 操作系统：EulerOS 2.9 • 系统盘：50GiB | 通用型SSD • 数据盘：100GiB | 通用型SSD	1	1.00元/小时
云硬盘EVS	建议选择按需计费。 • 云硬盘规格：100G • 云硬盘类型：通用型SSD	1	0.1元/小时
负载均衡器 ELB	建议选择按需计费。 • 实例规格：共享型 • 公网带宽：按流量计费 • 带宽：5 Mbit/s	1	0.32元/小时+公网流量费用0.80元/GB（按照您实际使用的出云流量收取流量费）

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**，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），详情请参照**购买CCE集群**创建。

本实践需要使用Docker in Docker场景，即在容器中运行Docker命令，节点需要选择Docker容器引擎。

- 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。

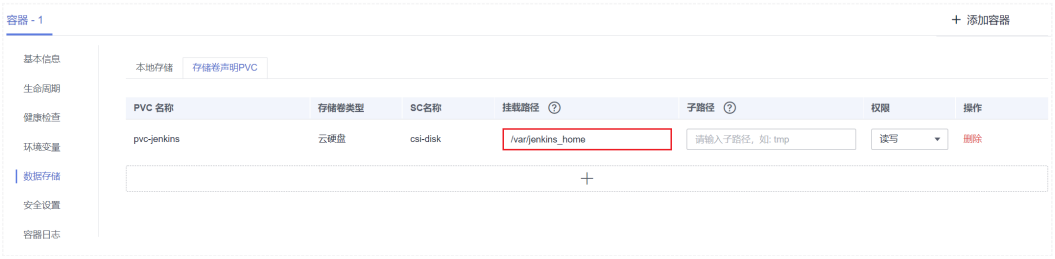
在弹出的窗口中选择1个云存储卷，并在挂载路径下输入**/var/jenkins_home**，将云存储挂载到Jenkins容器的**/var/jenkins_home**目录，供Jenkins保留持久化数据。

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。

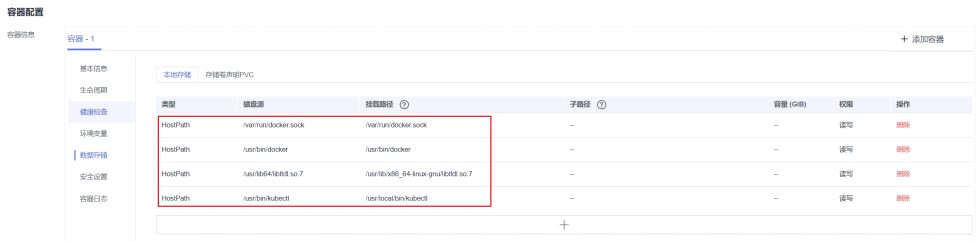
1. 确认3中已开启“特权容器”开关。
2. 在“数据存储”中的“本地存储”页签下添加本地存储，将主机路径挂载到容器对应路径。

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）	/var/run/docker.sock	/var/run/docker.sock
主机路径（HostPath）	/usr/bin/docker	/usr/bin/docker
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

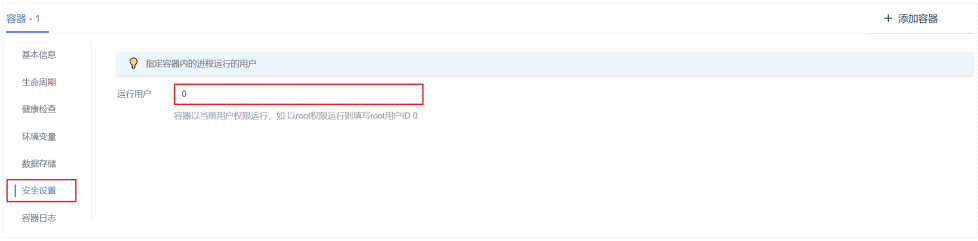
挂载完成后，如图4-4所示。

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。

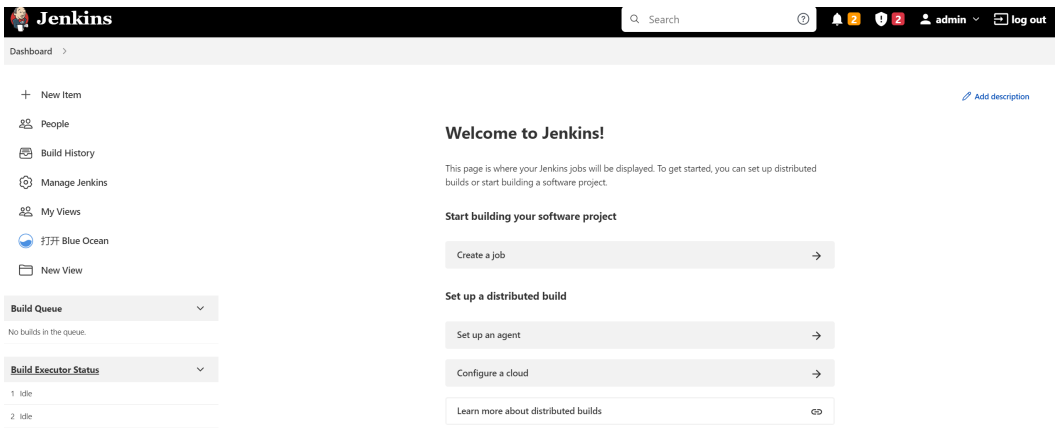
图 4-8 访问 8080 端口对应的访问方式

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
```

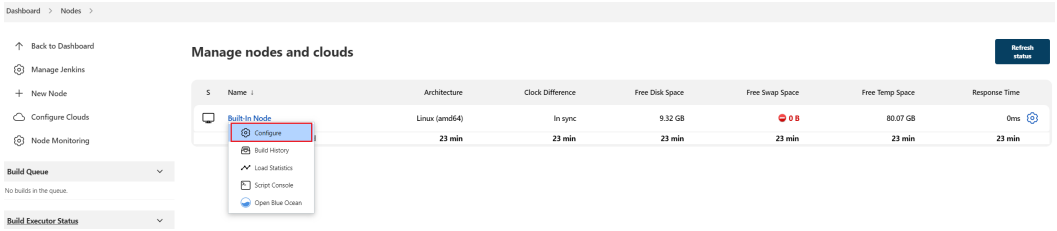
步骤3 首次登录时选择默认推荐的插件即可，并根据页面提示创建一个管理员。完成初始配置后，即可进入Jenkins页面。



----结束

修改并发构建数量

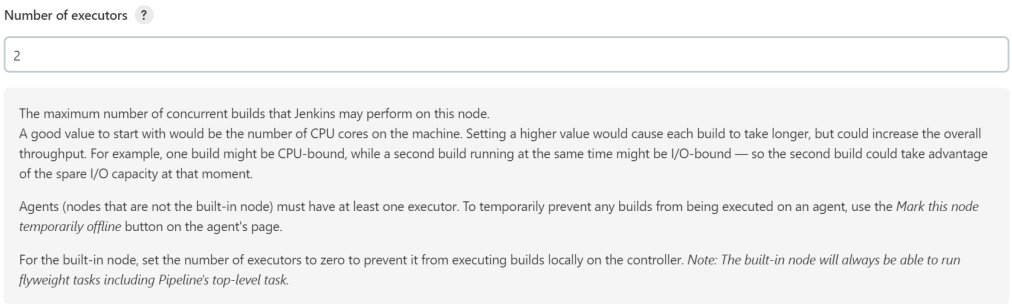
步骤1 在Jenkins Dashboard页面，单击左侧“Manage Jenkins”，选择“System Configuration > Manage nodes and clouds”，选择目标节点下拉框里的“Configure”，如下图所示：



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。

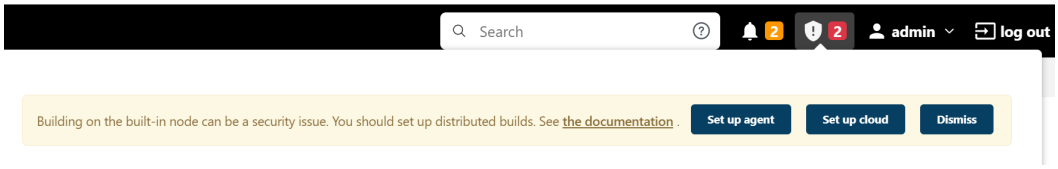
步骤2 修改执行并发构建的最大数目，示例中修改为2，您可根据实际需求并结合节点性能进行修改该值。



----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。

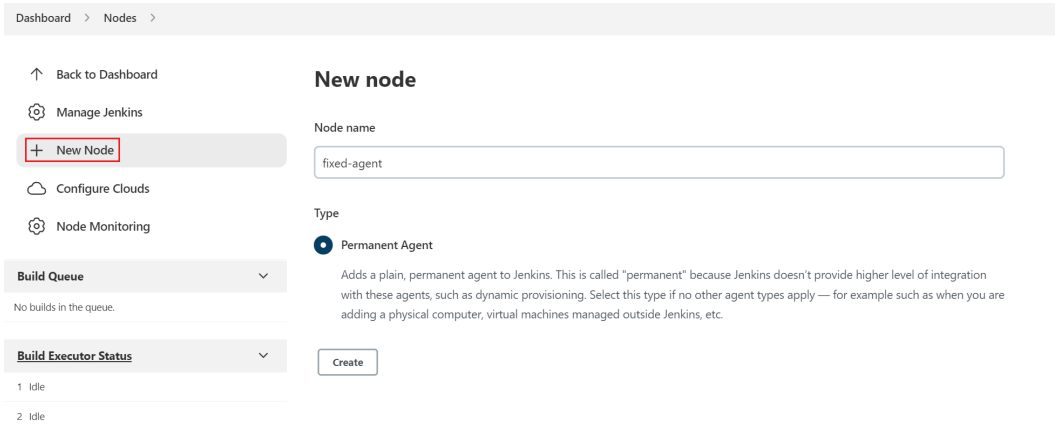
- **动态Agent**：构建任务时动态创建Agent容器，并在任务构建完成后销毁容器，可实现资源动态分配，资源利用率高，但是配置过程较为复杂。

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”。

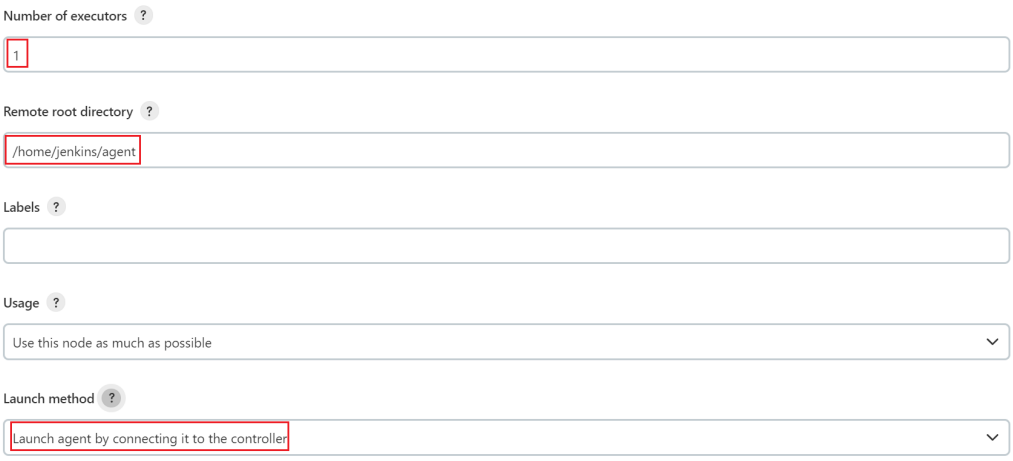
步骤2 单击新页面左侧的“New Node”，输入节点名称为fixed-agent（该名称可自定义），类型选择固定节点。



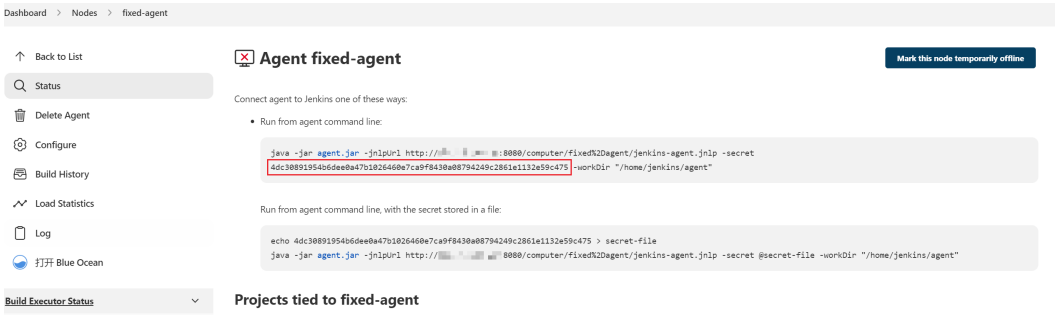
步骤3 配置以下节点信息：

- 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。

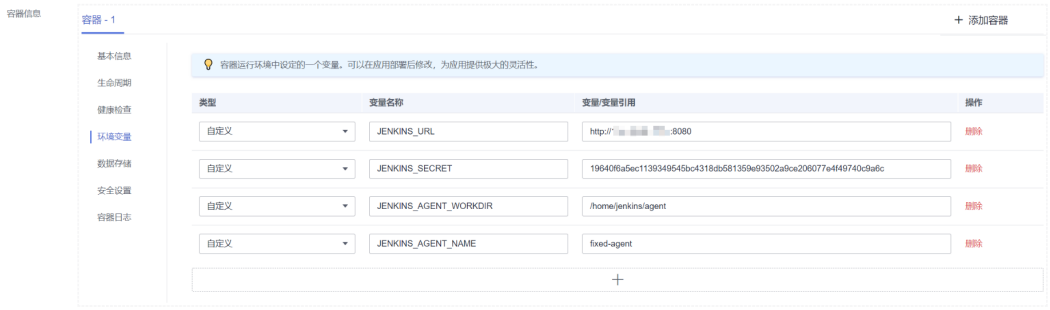
- 镜像名称：**jenkins/inbound-agent:4.13.3-1**。此处镜像版本可能随时间变化发生变动，请根据实际情况进行选择，或拉取latest版本。
- CPU配额：本例中CPU配额限制为2 Core
- 内存配额：本例中内存配额限制为2048 MiB
- 特权容器：必需要开启“特权容器”，使容器获得操作宿主机的权限，否则容器中无法执行docker命令。

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。

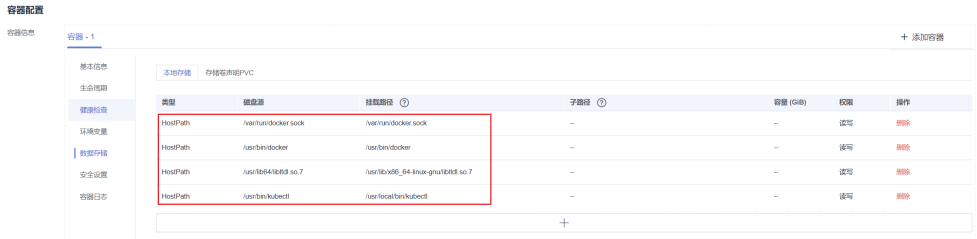
1. 确认3中已开启“特权容器”开关。
2. 在“数据存储”中的“本地存储”页签下添加本地存储，将主机路径挂载到容器对应路径。

表 4-4 挂载路径

存储类型	主机路径（HostPath）	挂载路径
主机路径（HostPath）	**/var/run/docker.sock**	**/var/run/docker.sock**
主机路径（HostPath）	**/usr/bin/docker**	**/usr/bin/docker**
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**

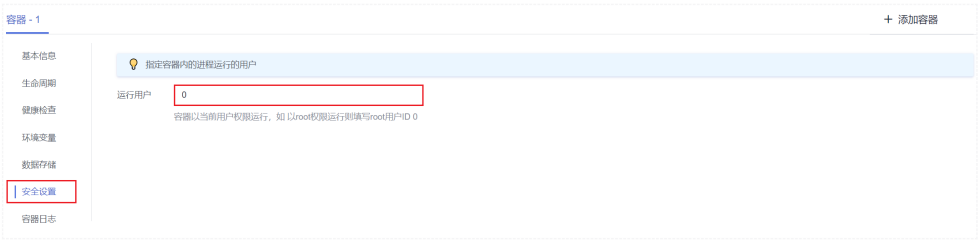
挂载完成后，如图4-9所示。

图 4-9 挂载主机到容器对应路径



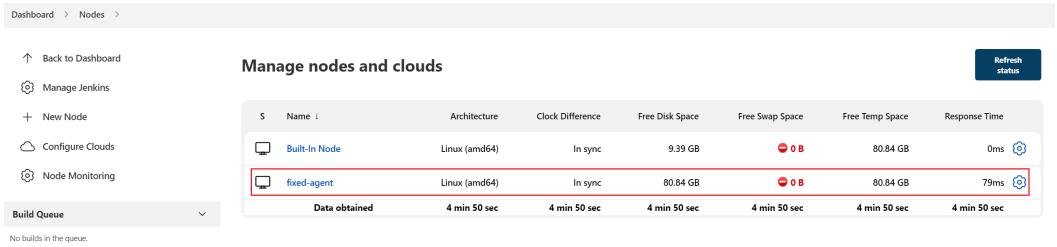
3. 在“安全设置”中配置“运行用户”为：0（即root用户）。

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

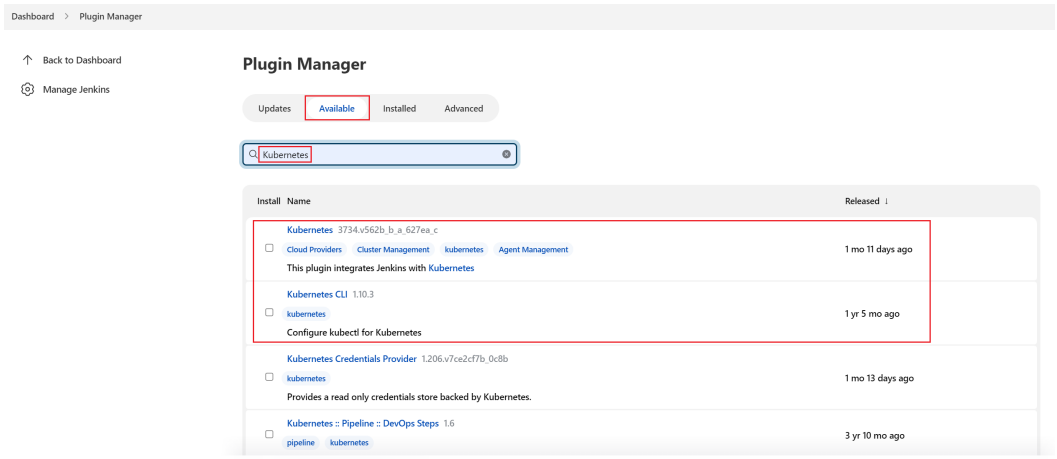
Agent设置完成后，建议将Master的并发构建数设置为0，即不使用Master进行本地构建，全部使用Agent进行构建，具体操作步骤请参见**修改并发构建数量**。

----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。

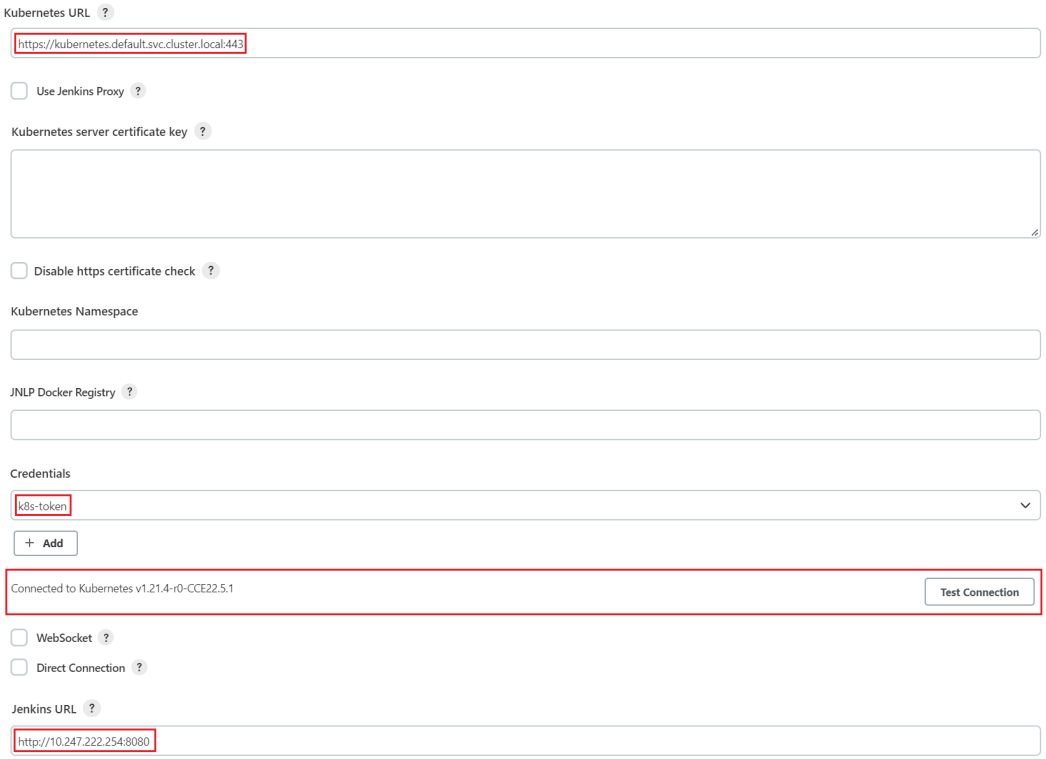
在Jenkins Dashboard页面单击左侧“Manage Jenkins”，选择“System Configuration > Manage nodes and clouds”，单击左侧的“Configure Clouds”配置集群，单击“Add a new cloud”，并选择**Kubernetes**，集群名称可自定义。

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”。

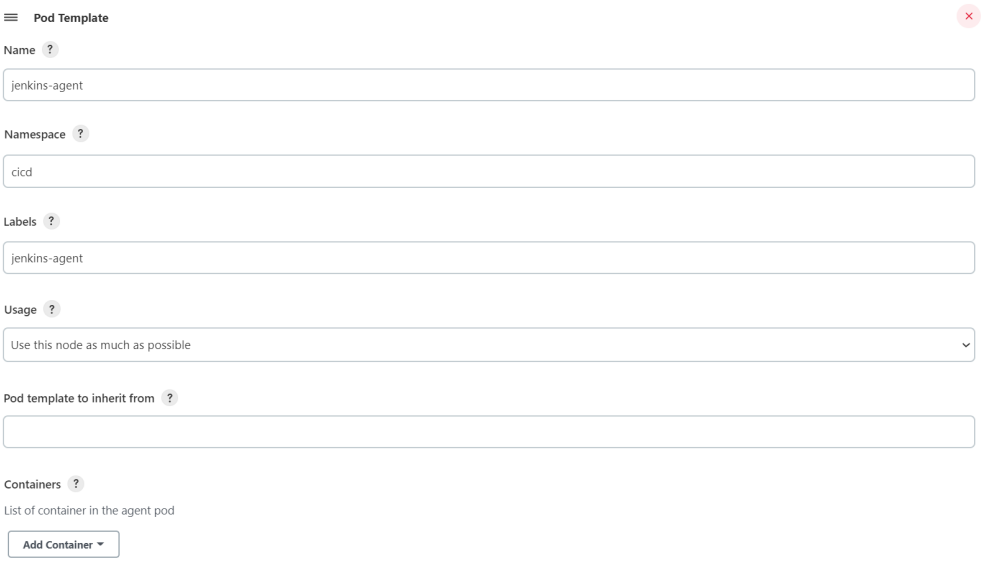
图 4-11 Kubernetes Cloud details 填写示例



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

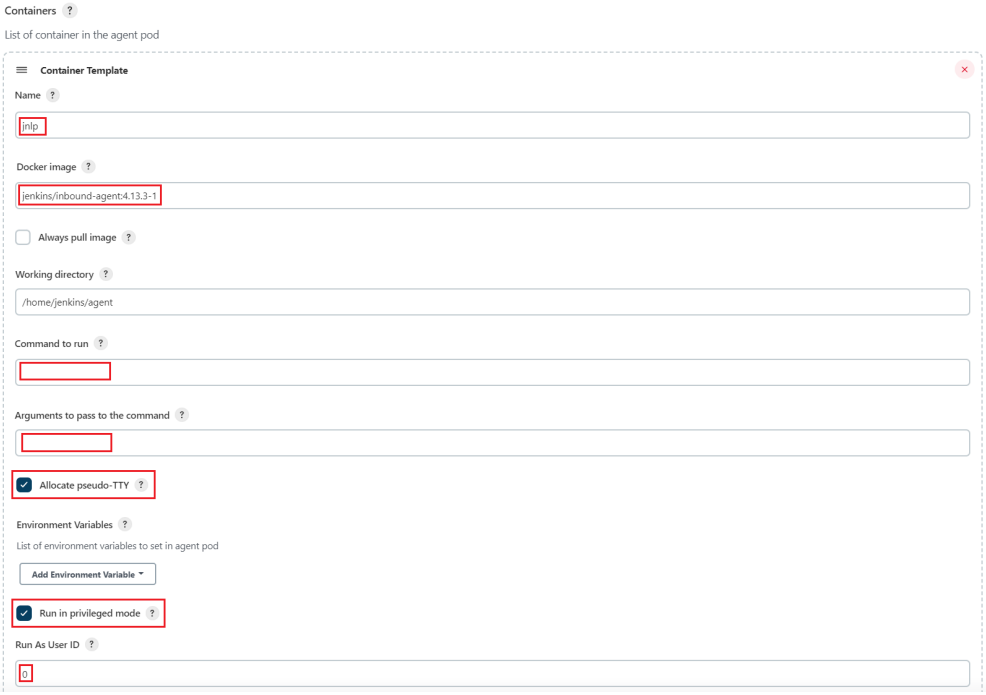
- 配置Pod模板基本参数：参数配置如图4-12所示。
  - Name：jenkins-agent
  - Namespace：cicd
  - Labels：jenkins-agent
  - Usage：选择“Use this node as much as possible”

  图 4-12 Pod Template 基本参数

- 添加容器：单击“Add Container > Container Template”，参数配置如图4-13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



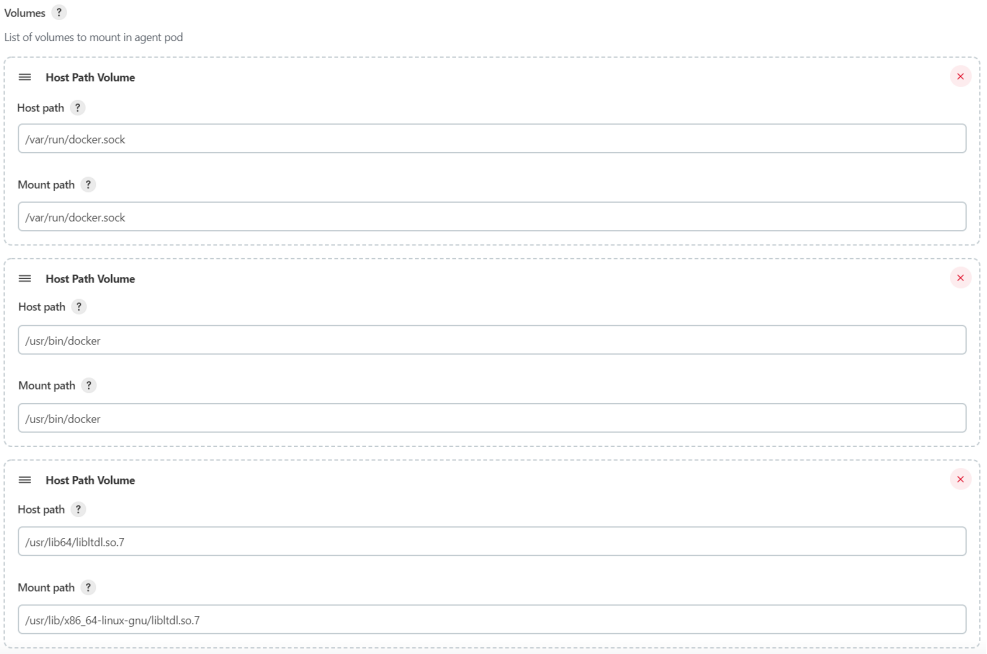
- 添加卷：单击“Add Volume > Host Path Volume”，将表4-5中的主机路径挂载到容器对应路径。

表 4-5 挂载路径

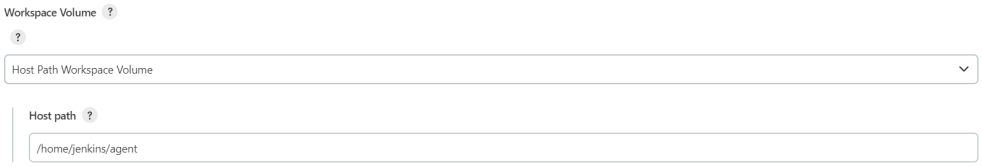
存储类型	主机路径（HostPath）	挂载路径
主机路径（HostPath）	/var/run/docker.sock	/var/run/docker.sock
主机路径（HostPath）	/usr/bin/docker	/usr/bin/docker
主机路径（HostPath）	/usr/lib64/libltdl.so.7	/usr/lib/x86_64-linux-gnu/libltdl.so.7
主机路径（HostPath）	/usr/bin/kubectl	/usr/local/bin/kubectl

挂载完成后，如图4-14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

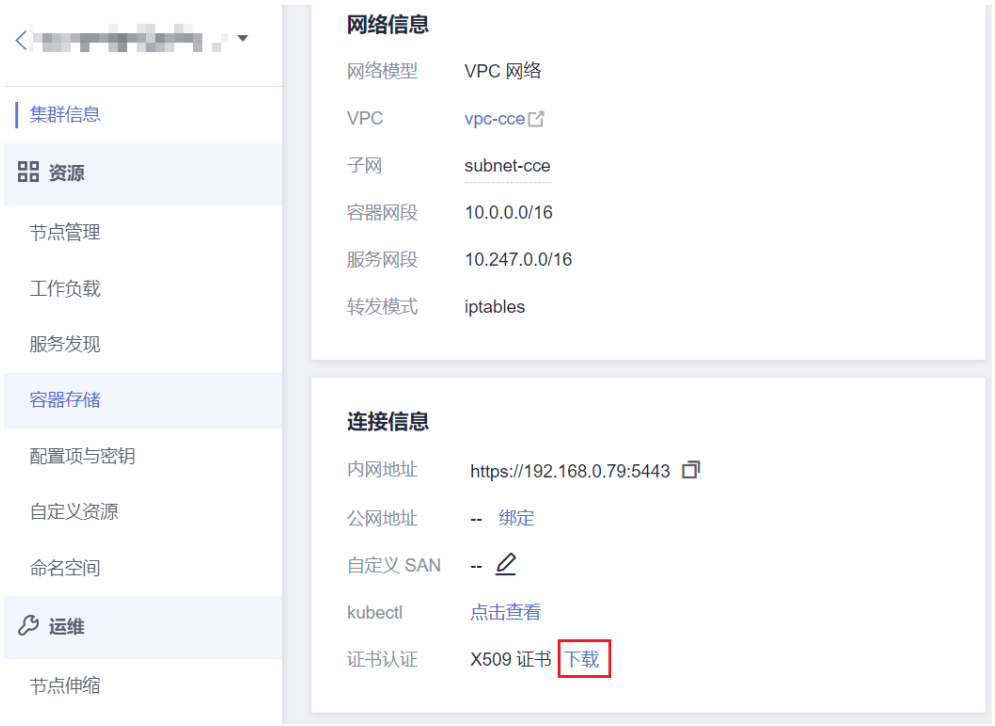
Agent设置完成后，建议将Master的并发构建数设置为0，即不使用Master进行本地构建，全部使用Agent进行构建，具体操作步骤请参见**修改并发构建数量**。

----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。

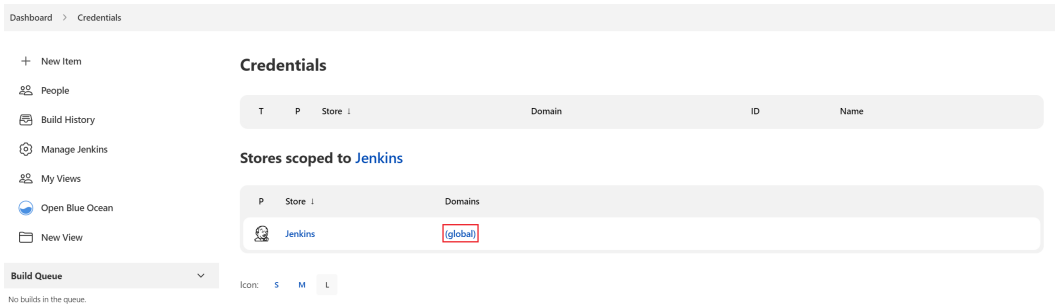
步骤1 前往CCE控制台的“集群信息 > 连接信息”页面中下载集群证书，证书包含ca.crt、client.crt、client.key三个文件。



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。

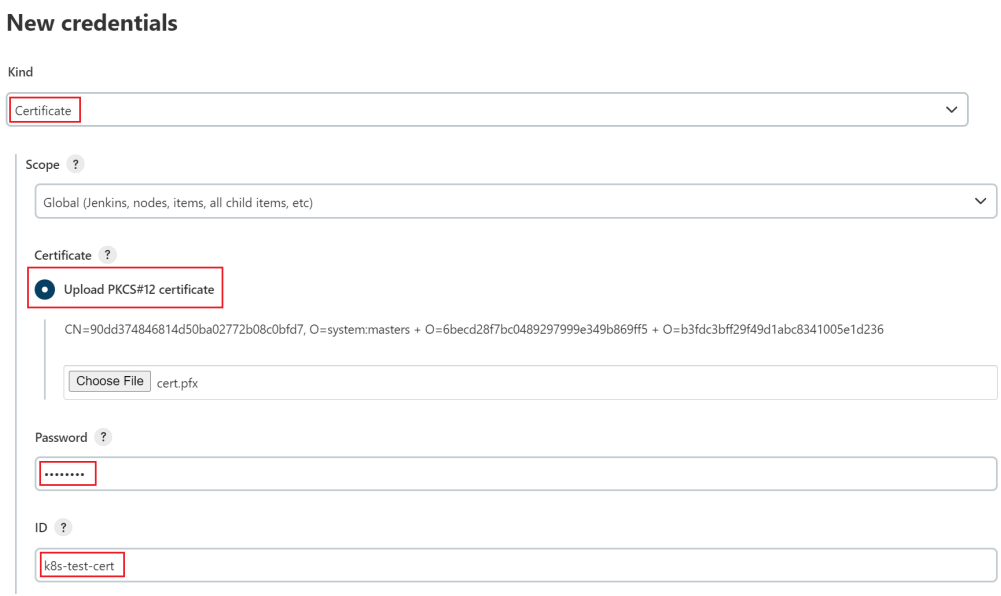
```
openssl pkcs12 -export -out cert.pfx -inkey client.key -in client.crt -certfile ca.crt
```

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，已经完成了容器中执行docker命令的配置（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**。

例如本账号的命令为：

```
docker login -u ap-southeast-1@xxxxx -p xxxxx swr.ap-southeast-1.myhuaweicloud.com
```

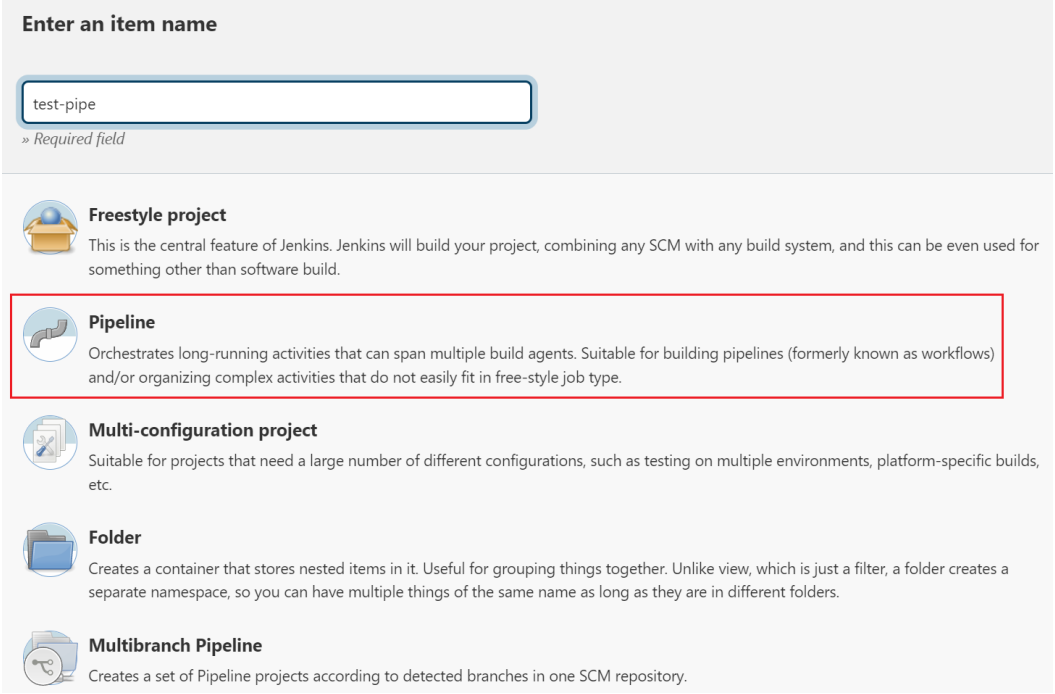
创建 pipeline 完成镜像构建及 push

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。

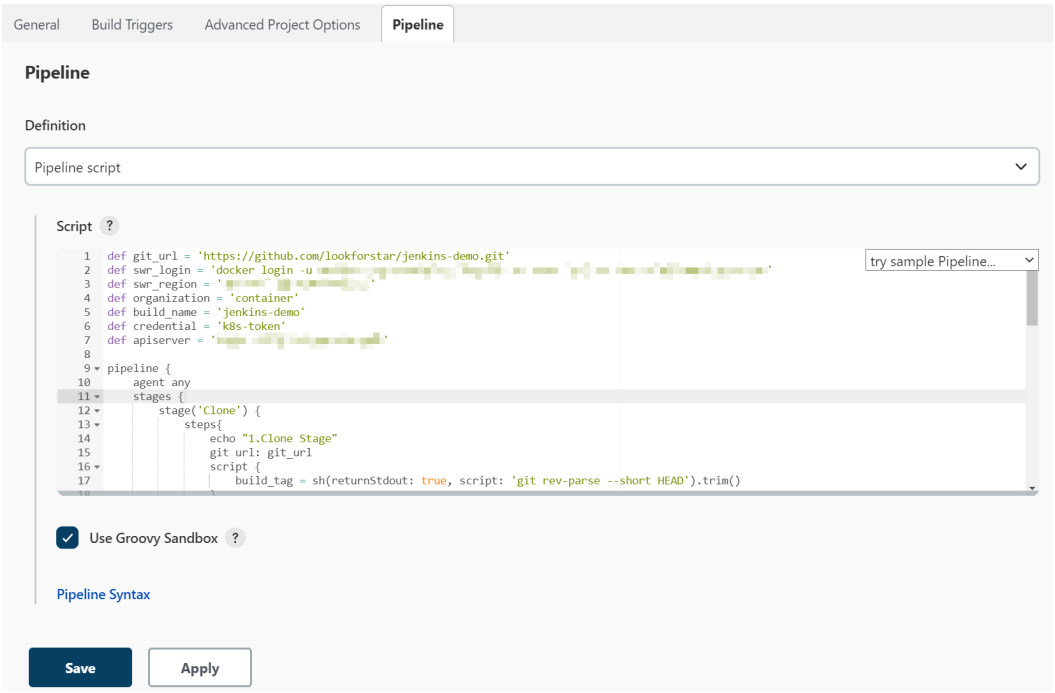
创建pipeline步骤如下：

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//定义SWR区域
def swr_region = 'ap-southeast-1'
//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 swr.${swr_region}.myhuaweicloud.com/${organization}/${build_name}:${build_tag} ."
                //${build_tag}表示获取上文中的build_tag变量作为镜像标签，为git rev-parse --short HEAD命令的返回值，即commit ID。
            }
        }
        stage('Push') {
            steps{
                echo "4.Push Docker Image Stage"
                sh swr_login
                sh "docker push swr.${swr_region}.myhuaweicloud.com/${organization}/${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。详情请参见**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。

```
$ 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}')
```

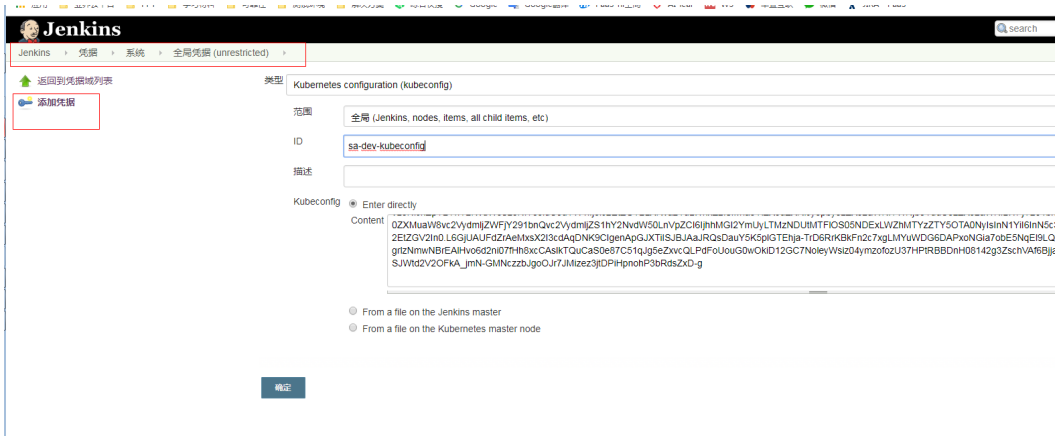
```
$ cat <<EOF > dev.conf
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_CERT
    server: $API_SERVER
  name: cluster
EOF

$ 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

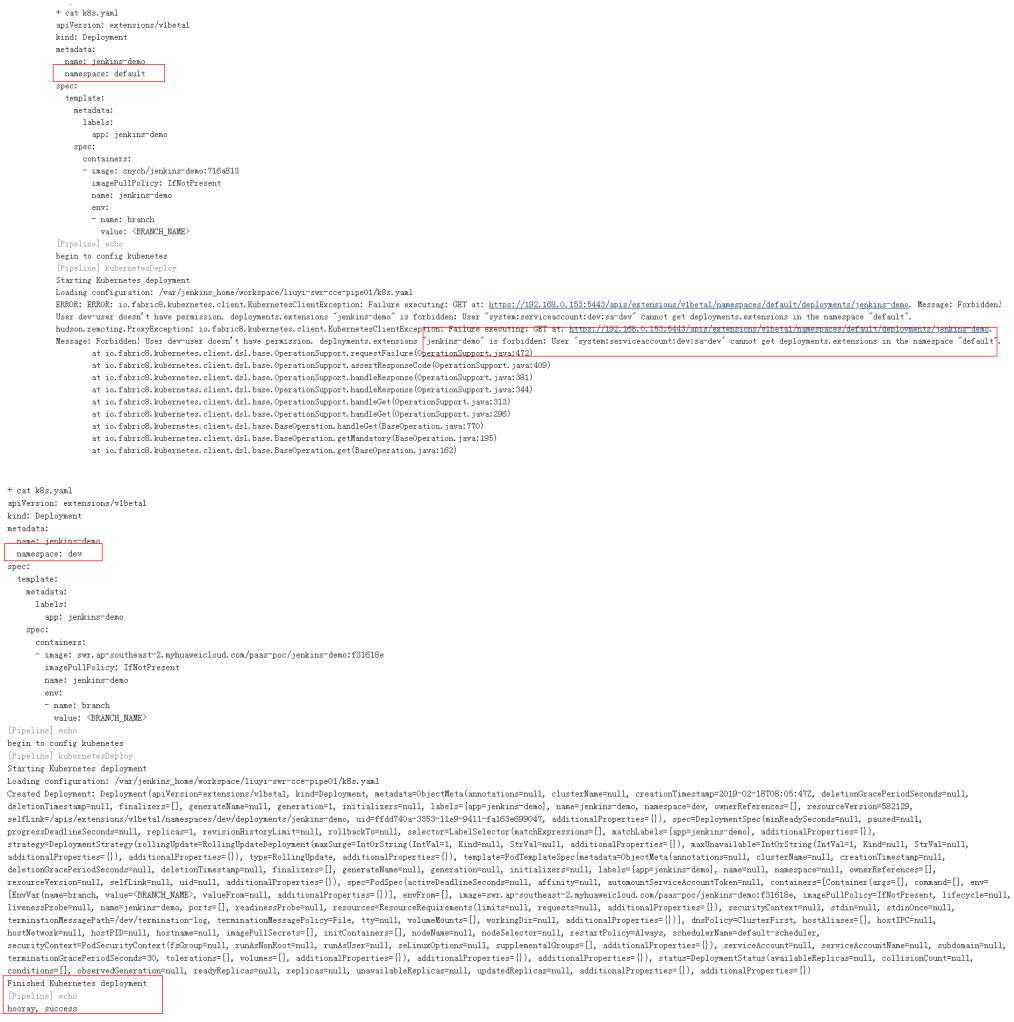
```
$ kubectl --kubeconfig=dev.conf get po
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:dev:dev" cannot list pods in the namespace "default"

$ 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。详情请参见**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。

```
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

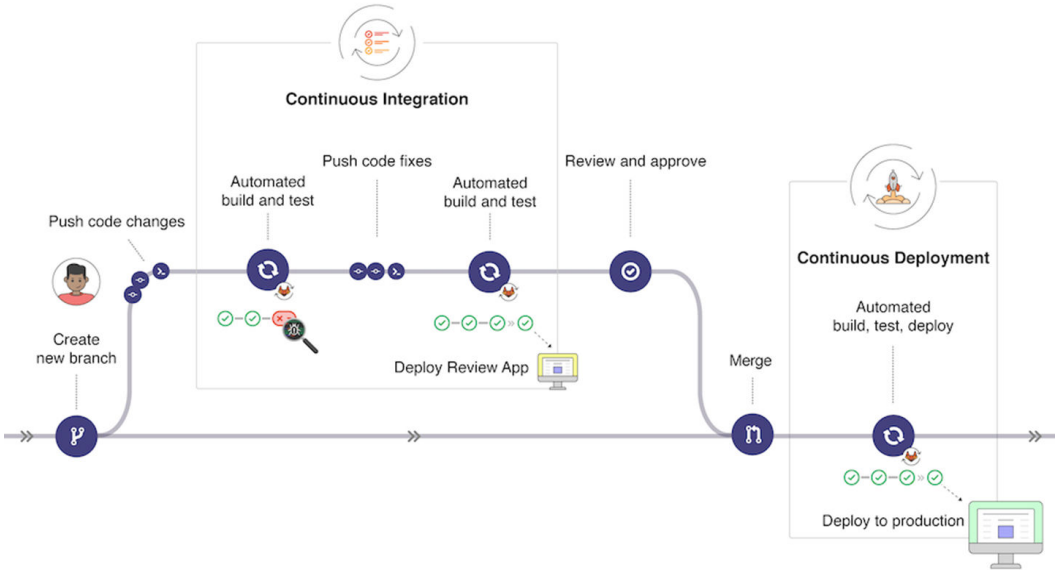
4.2 Gitlab 对接 SWR 和 CCE 执行 CI/CD

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

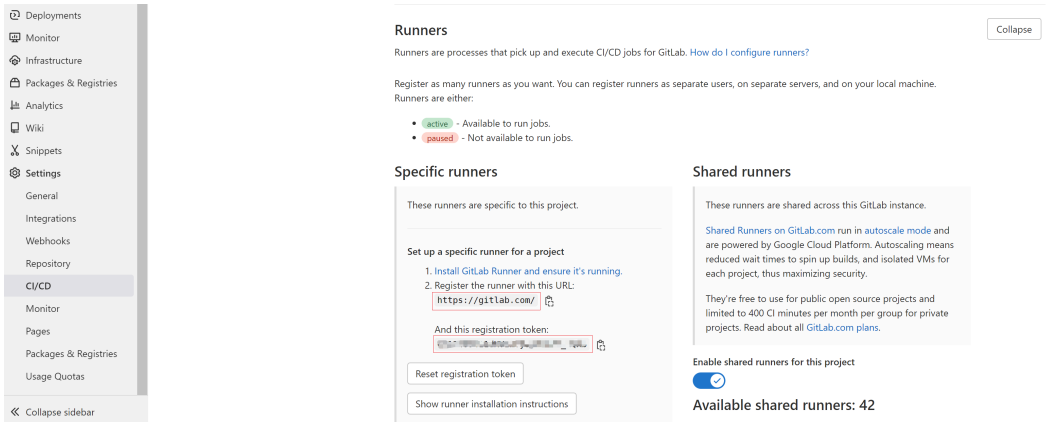
1. 创建一台CCE集群，且需要给节点绑定一个EIP，用于安装Gitlab Runner时下载镜像。
2. 下载并配置kubectl连接集群。

   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: "******"
rbac:
    create: true
runners:
    privileged: true
```

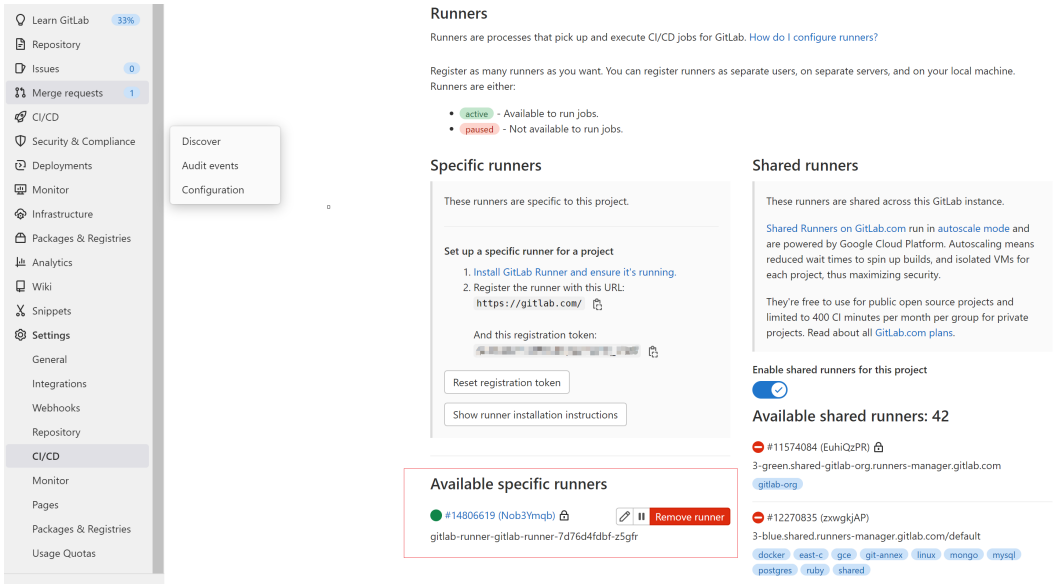
创建gitlab命名空间。

```
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 --version=0.43.1
```

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。

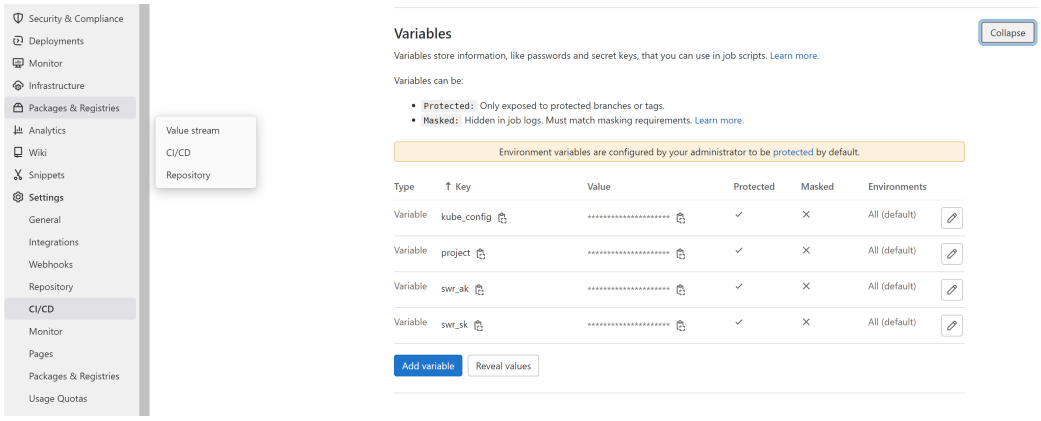
- **.gitlab-ci.yml**：Gitlab CI文件，将在**创建流水线**中详细讲解。
- **Dockerfile**：用于制作Docker镜像。
- **index.html**：用于替换nginx的index页面。
- **k8s.yaml**：用于部署nginx应用，会创建一个名为nginx-test的Deployment，和一个名为nginx-test的Service。

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。

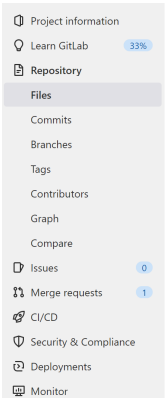
执行如下命令获取密钥，其中$AK和$SK为上面获取的AK/SK。

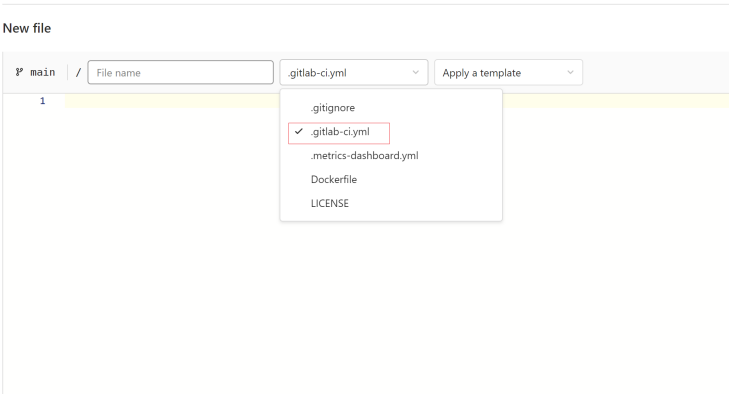
printf "*$AK*" | openssl dgst -binary -sha256 -hmac "*$SK*" | od -An -vtx1 | sed 's/[\n]//g' | sed 'N;s/\n//'

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。



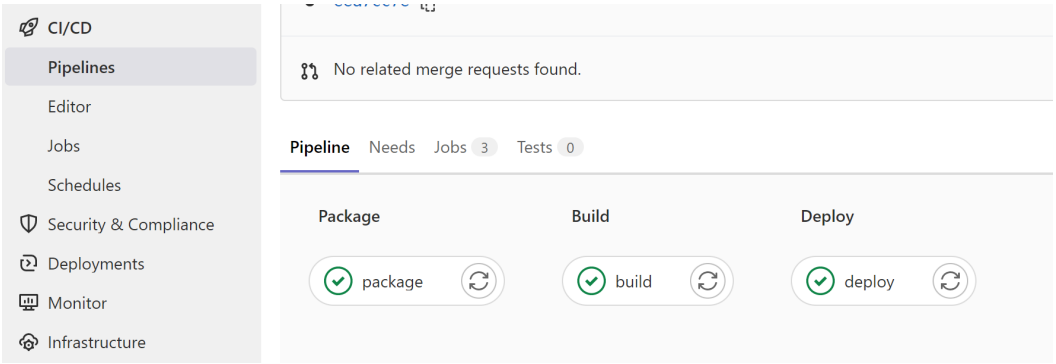


其内容如下所示。

```
#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 $swr_sk swr.ap-southeast-1.myhuaweicloud.com
    # 构建镜像，其中k8s-dev为SWR中的组织名称，请根据实际情况替换
    - docker build -t swr.ap-southeast-1.myhuaweicloud.com/k8s-dev/nginx:$CI_PIPELINE_ID .
    # 推送镜像到SWR
    - docker push swr.ap-southeast-1.myhuaweicloud.com/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 > $KUBECONFIG
  # 替换k8s.yaml文件中的镜像
  - sed -i "s/<IMAGE_NAME>/swr.ap-southeast-1.myhuaweicloud.com\/k8s-dev\/nginx:$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

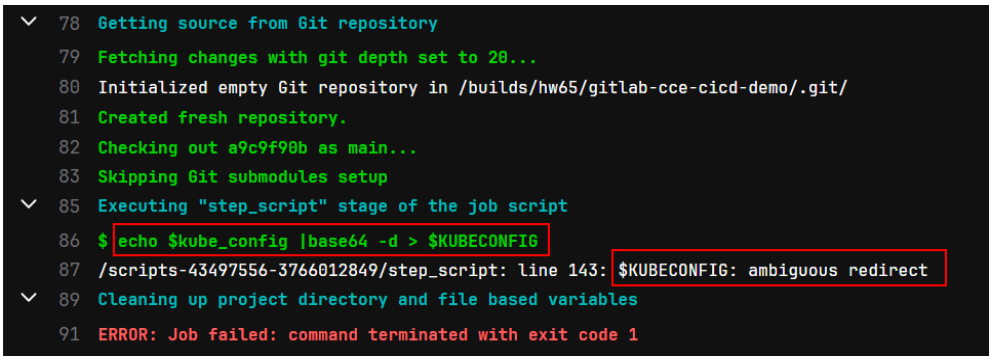
流水线部署成功后，在CCE控制台找到名为nginx-test的Service，查询到nginx-test的访问地址，使用curl命令访问。

```
# 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


```
$ kubectl apply -f k8s.yaml
E0215 08:03:55.297105      19 memcache.go:255] couldn't get resource list for proxy.exporter.k8s.io/v1beta1: Got empty response for: proxy.exporter.k8s.io/v1beta1
Error from server (Forbidden): error when retrieving current configuration of:
Resource: "apps/v1, Resource=deployments", GroupVersionKind: "apps/v1, Kind=Deployment"
Name: "nginx-test", Namespace: "gitlab"
from server for: "k8s.yaml": deployments.apps "nginx-test" is forbidden: User "system:serviceaccount:gitlab:default" cannot get resource "deployments" in API group "apps" in the namespace "gitlab"
Error from server (Forbidden): error when retrieving current configuration of:
Resource: "/v1, Resource=services", GroupVersionKind: "/v1, Kind=Service"
Name: "nginx-test", Namespace: "gitlab"
from server for: "k8s.yaml": services "nginx-test" is forbidden: User "system:serviceaccount:gitlab:default" cannot get resource "services" in API group "" in the namespace "gitlab"
Cleaning up project directory and file based variables
00:01
ERROR: Job failed: command terminated with exit code 1
```

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
...
```

- 如果流水线执行过程出现无法执行docker的情况，如下所示。

```
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
Login Succeeded
$ docker build -t swr.[redacted] com/wpf-test/nginx:$CI_PIPELINE_ID .
Cannot connect to the Docker daemon at tcp://docker:2375. Is the docker daemon running?
Cleaning up project directory and file based variables
00:00
ERROR: Job failed: command terminated with exit code 1
```

安装gitlab runner过程中传递privileged: true参数未成功，导致没有权限运行docker命令，请在CCE控制台工作负载列表中找到gitlab runner，添加环境变量**KUBERNETES_PRIVILEGED**，取值为**true**，如下所示。

5 容灾

5.1 CCE 集群高可用推荐配置

为了保证应用可以稳定可靠的运行在Kubernetes里，本文介绍构建Kubernetes集群时的推荐配置。

类型	说明	高可靠配置建议
集群控制面	CCE是一项托管式的Kubernetes服务，集群控制面（即控制节点）无需由用户进行运维，您可以通过一些集群配置来提高集群整体的稳定性和可靠性。	• 集群Master节点多可用区 • 集群网络选择 • 服务转发模式 • 关注配额限制 • 监控Master指标
集群数据面	在Kubernetes集群中，数据面由工作节点组成，这些节点可以运行应用程序容器并处理网络流量。在使用CCE过程中，数据面的节点需要您自行运维。为实现高可靠目标，您需要保证数据面的可扩展性及可修复性，并及时关注关键组件的运行状态。	• 节点数据盘分区及大小 • 运行npd • 配置DNS缓存 • 合理部署CoreDNS
应用层面	如果您希望应用程序始终可用，尤其是在流量高峰期间确保您的应用程序和服务不间断地运行，您需要通过可扩展且有弹性的方式运行应用，并及时关注应用的运行状态。	• 运行多个实例 • 设置资源配额 • 应用多可用区部署 • 系统插件多可用区部署 • 自动弹性伸缩 • 日志监控告警

集群 Master 节点多可用区

华为云支持多区域（Region），每个区域下又有不同的可用区（AZ，Availability Zone）。可用区是一个或多个物理数据中心的集合，有独立的风火水电。一个Region中的多个AZ间通过高速光纤相连，以满足用户跨AZ构建高可用性系统的需求。

创建集群时，您可以设置集群的高可用模式，并选择控制节点的分布方式。控制节点默认尽可能随机分布在不同可用区以提高容灾能力 。

您还可以展开高级配置自定义控制节点分布方式，支持如下2种方式。

- 随机分配：通过把控制节点随机创建在不同的可用区中实现容灾。
- 自定义：自定义选择每台控制节点的位置。
  - 主机：通过把控制节点创建在相同可用区下的不同主机中实现容灾。
  - 自定义：用户自行决定每台控制节点所在的位置。

图 5-1 集群高可用

集群网络选择

- 集群网络模型选择：CCE支持云原生网络2.0、VPC网络、容器隧道网络模型。不同的网络模型存在性能和功能各方面的差异，请合理选择，详情请参见**集群网络模型**。
- VPC选择：如果您的应用需要连接其他云服务如RDS数据库等，则需要考虑将相关服务创建在同一个VPC中，因为VPC间网络是相互隔离的。如果您已经创建好实例，也可以将VPC之间通过**对等连接**进行互通。
- 容器网段选择：容器网络的网段不能设置太小，如果太小会导致可创建的节点数量受限。
  - 对于VPC网络模型的集群来说，如果容器网络的网段掩码是/16，那么就有256*256个地址，如果每个节点预留的Pod数量上限是128，则最多可以支持512个节点。

  - 对于容器隧道网络模型的集群来说，如果容器网络的网段掩码是/16，那么就有256*256个地址，节点从容器网段中一次分配的IP网段默认为16，则最多可创建节点数量为65536/16=4096。
  - 对于云原生网络2.0模型的集群来说，容器网段为VPC的子网，容器数量则取决于选择的子网大小。
- 服务网段选择：服务网段决定集群中Service资源的上限，调整该网段需要根据实际需求进行评估，创建后不可修改，请勿设置过小的服务网段。

关于集群网络地址段的选择详情，可参见**10.1 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。在使用集群时，您需要考虑服务转发模式潜在的性能问题。

CCE当前支持iptables和IPVS两种服务转发模式，各有优缺点。

- IPVS：吞吐更高，速度更快的转发模式。适用于集群规模较大或Service数量较多的场景。
- iptables：社区传统的kube-proxy模式。适用于Service数量较少或客户端会出现大量并发短连接的场景。当集群中超过1000个Service时，可能会出现网络延迟的情况。

关注配额限制

CCE支持设置配额限制，您设置云服务级别和集群级别的资源数量上限，以防止您过度意外使用资源。在构建创建应用时，应考虑这些限制值并定期审视，防止在应用运行过程中出现配额不足的瓶颈导致扩缩容失败。

- 云服务配额：使用CCE时也会使用其他云服务，包括弹性云服务器、云硬盘、虚拟私有云、弹性负载均衡、容器镜像服务等。如果当前资源配额限制无法满足使用需要，您可以提交工单申请扩大配额。
- 集群配额：集群中支持设置命名空间的配额，可限制命名空间下创建某一类型对象的数量以及对象消耗计算资源（CPU、内存）的总量，详情请参见**设置资源配额及限制**。

监控 Master 指标

监控控制节点（Master节点）的指标可以帮助您深入了解控制节点性能并识别问题，运行状况不佳的控制节点可能会损害应用的可靠性。

CCE支持对Master节点的kube-apiserver、kube-controller、kube-scheduler、etcd-server组件进行监控，您需要在集群中安装**kube-prometheus-stack**插件。通过插件自带的grafana组件，您可以使用**Kubernetes监控概述仪表盘**来可视化和监控Kubernetes API服务器请求以及延迟和etcd延迟指标。

在集群中自建Prometheus的场景，您可以手动添加指标，详情请参见**Master节点组件指标监控**。

节点数据盘分区及大小

节点第一块数据盘默认供容器运行时及kubelet组件使用，其剩余的容量大小会影响镜像下载和容器启动及运行，数据盘的分配详情请参见**数据盘空间分配说明**。

该数据盘默认大小为100G，您也可以根据需求调整该数据盘大小。由于镜像、系统日志、应用日志都保存在数据盘上，您需要考虑每个节点上要部署的Pod数量，每个Pod的日志大小、镜像大小、临时数据，再加上一些系统预留的值，详情请参考**9.7 选择合适的节点数据盘大小**。

运行 npd

工作节点中的故障可能会影响应用程序的可用性。**npd**插件是一款监控集群节点异常事件的插件，帮助您及时感知节点上可能存在的异常并及时处理。您也可以对npd插件的故障检查项进行自定义配置，包括检查的目标节点、检查周期、触发阈值等，详情请参见**节点故障检测策略**。

配置 DNS 缓存

当集群中的DNS请求量增加时，CoreDNS将会承受更大的压力，可能会导致如下影响：

- 延迟增加：CoreDNS需要处理更多的请求，可能会导致DNS查询变慢，从而影响业务性能。
- 资源占用率增加：为保证DNS性能，CoreDNS往往需要更高规格的配置。

为了避免DNS延迟的影响，可以在集群中部署NodeLocal DNSCache来提升服务发现的稳定性和性能。NodeLocal DNSCache会在集群节点上运行DNS缓存代理，所有注入DNS配置的Pod都会使用节点上运行的DNS缓存代理进行域名解析，而不是使用CoreDNS服务，以此来减少CoreDNS服务的压力，提高集群DNS性能。

您可以安装**node-local-dns**插件部署NodeLocal DNSCache，详情请参见**使用NodeLocal DNSCache提升DNS性能**。

合理部署 CoreDNS

建议您在部署CoreDNS时，将CoreDNS实例分布在不同可用区、不同节点上，尽可能避免单节点、单可用区故障。

且CoreDNS所运行的节点应避免CPU、内存打满，否则会影响域名解析的QPS和响应延迟。

更多关于合理配置CoreDNS的详情，请参见**10.10.3.3 合理配置CoreDNS**。

运行多个实例

如果你的整个应用程序在独立的Pod中运行，那么如果该Pod出现异常，应用程序将不可用。请使用Deployment或其他类型的副本集来部署应用，每当Pod失败或被终止，控制器会自动重新启动一个与之相同的新Pod，以确保指定数量的Pod始终运行。

同时，在创建工作负载时，您可以指定实例数量大于2。如果一个实例发生故障，剩余的实例仍将运行，直到Kubernetes自动创建另一个Pod来弥补损失。此外，您还可以使用**7.1 使用HPA+CA实现工作负载和节点联动弹性伸缩**根据工作负载需求自动进行伸缩。

使用容器隔离进程

容器可以提供进程级别的隔离，每个容器都有自己的文件系统、网络和资源分配，可以避免不同进程之间相互干扰，也可以避免恶意进程的攻击和数据泄露。使用容器隔离进程可以提高应用程序的可靠性、安全性和可移植性。

如果有几个进程需要协同工作，可以在一个Pod创建多个容器，以便它们可以共享相同的网络、存储卷和其他资源。例如init容器，init容器会在主容器启动之前运行，可以用于完成一些初始化任务，比如配置环境变量、加载数据库或数据存储以及拉取Git库等操作。

但需要注意的是，一个Pod中存在多个容器时会共享同一个Pod的生命周期。因此如果其中一个容器异常，整个Pod将被重新启动。

设置资源配额

为所有工作负载配置和调整资源请求/限制

当在一个节点上调度了太多的Pod时，会导致节点负载太高，无法正常对外提供服务。

为避免上述问题，在Kubernetes中部署Pod时，您可以指定这个Pod需要Request及Limit的资源，Kubernetes在部署这个Pod的时候，就会根据Pod的需求找一个具有充足空闲资源的节点部署这个Pod。下面的例子中，声明Nginx这个Pod需要1核CPU，1024M的内存，运行中实际使用不能超过2核CPU和4096M内存。

Kubernetes采用静态资源调度方式，对于每个节点上的剩余资源，它是这样计算的：节点剩余资源=节点总资源-已经分配出去的资源，并不是实际使用的资源。如果您自己手动运行一个很耗资源的程序，Kubernetes并不能感知到。

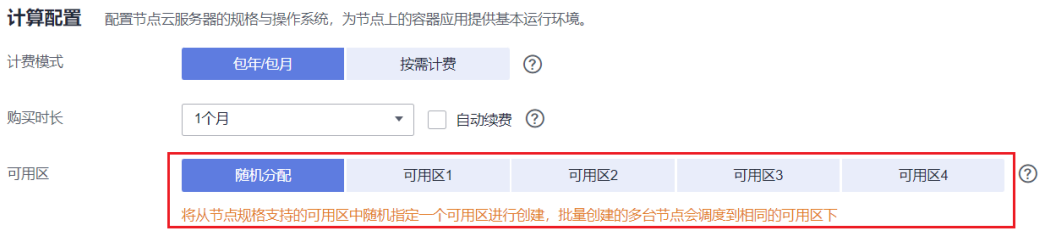
另外所有Pod上都要声明resources。对于没有声明resources的Pod，它被调度到某个节点后，Kubernetes也不会在对应节点上扣掉这个Pod使用的资源。可能会导致节点上调度过去太多的Pod。

应用多可用区部署

您可以通过在多个可用区的节点上运行Pod，以避免应用受单个可用区故障的影响。

在创建节点时，您可以手动指定节点的可用区。

图 5-2 指定可用区



在部署应用时，您可以为Pod设置反亲和性规则，实现跨多个可用区调度Pod，详情请参见**5.2 在CCE中实现应用高可用部署**。示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
```

```
  metadata:
    labels:
      app: web-server
  spec:
    containers:
    - name: web-app
      image: nginx
    imagePullSecrets:
    - name: default-secret
    affinity:
      podAntiAffinity:  # 工作负载反亲和
        preferredDuringSchedulingIgnoredDuringExecution:  # 表示尽量满足规则，否则Pod数超过可用区数量时会无法调度
        - podAffinityTerm:
            labelSelector:  # Pod标签匹配规则，设置Pod与自身标签反亲和
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-server
            topologyKey: topology.kubernetes.io/zone  # 节点可用区拓扑域
          weight: 100
```

您也可以使用**Pod的拓扑分布约束**实现多可用区部署。

系统插件多可用区部署

与应用多可用区部署类似，CCE系统核心插件（如CoreDNS、Everest等）的无状态应用（Deployment）实例支持多种多可用区部署模式，满足不同场景下的用户诉求。

表 5-1 插件多可用区部署说明

部署模式	配置说明	使用说明	推荐配置场景
优先模式	插件实例配置Pod间弱反亲和策略，拓扑域为可用区级别（topology.kubernetes.io/zone），反亲和类型为preferredDuringSchedulingIgnoredDuringExecution。	尽量将插件实例调度到不同可用区，但当部分可用区资源不足时，插件实例有可能调度到其他资源充足的可用区，不能完全保证实例分布在多个不同可用区。	对多可用区容灾没有强制要求，可使用默认的优先模式。
强制模式	插件实例配置Pod间强反亲和策略，拓扑域为可用区级别（topology.kubernetes.io/zone），反亲和类型为requiredDuringSchedulingIgnoredDuringExecution。	限制每个可用区最多部署一个同一组件实例，实际能够运行的实例数无法超过当前集群下节点的可用区数量，同时由于限制单个可用区最多一个实例，实例所在节点的故障后，故障实例无法自动迁移到同可用区下的其他节点。	强制模式一般用于可用区数量后续有变动场景，避免所有实例都提前调度到当前的可用区节点上。

部署模式	配置说明	使用说明	推荐配置场景
均分模式	插件实例配置Pod拓扑分布约束，拓扑域为可用区级别（topology.kubernetes.io/zone），强限制不同拓扑域间的实例差不超过1，以达到插件实例在不同可用区间实现均衡分布效果。	该模式的效果介于优先模式和强制模式之间，既能达到实例在不同可用区部署要求，同时也支持实例数大于可用区场景，实现单可用区部署多实例。使用均分模式时需提前规划好各可用区节点资源，保证各可用区有足够的节点资源供实例部署（当单可用区的插件实例大于1时，建议各可用区可供插件实例可调度的节点数超过该可用区下实际插件实例数量1个以上），避免部分可用区节点资源不足阻塞插件实例的部署及更新过程中的整体调度。	均分模式在容灾要求较高场景推荐使用。

设置容器健康检查

Kubernetes对处于异常运行状态的Pod存在自动重启机制，可以避免一些Pod异常导致的服务中断问题，但是有的时候，即使Pod处于正常Running状态也不代表这个Pod能正常提供服务。例如，Pod里面的进程可能发生了死锁，但Pod的状态依然是Running，所以Kubernetes也不会自动重启这个Pod。因此，可以Pod上配置存活探针（Liveness Probe），探测Pod是否真的存活。如果存活探针发现了问题，Kubernetes会重启Pod。

同时，您也可以配置就绪探针（Readiness Probe），用于探测Pod是不是可以正常对外提供服务。应用在启动过程中可能会需要一些时间完成初始化，在这个过程中是没法对外提供服务的，为Pod添加过就绪探针后，当检测到Pod就绪时才会允许Service将请求转给Pod。当Pod出现问题的时候，就绪探针可以避免新流量继续转发到这个Pod。

启动探针（Startup Probe）用于探测应用程序容器启动是否成功。配置了启动探针后可以控制容器在启动成功后再进行存活性和就绪检查， 确保这些存活、就绪探针不会影响应用程序的启动。 这可以用于对启动慢的容器进行存活性检测，避免它们在启动运行之前就被终止。

您可以在创建应用时配置上述探针，YAML示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    args:
    - /server
    livenessProbe:
      httpGet:
```

```
      path: /healthz
      port: 80
      httpHeaders:
      - name: Custom-Header
        value: Awesome
    initialDelaySeconds: 3
    periodSeconds: 3
  readinessProbe:
    exec:
      command:
        - cat
        - /tmp/healthy
    initialDelaySeconds: 5
    periodSeconds: 5
  startupProbe:
    httpGet:
      path: /healthz
      port: 80
    failureThreshold: 30
    periodSeconds: 10
```

更多详情请参见**设置容器健康检查**。

自动弹性伸缩

弹性伸缩功能可以根据需求自动调整应用程序的实例数和节点数，可以在流量高峰期间快速扩容，并在业务低谷时进行缩容以节约资源与成本。

一般情况下，在流量高峰期间可能会出现两种类别的弹性伸缩：

- 工作负载伸缩：当使用Pod/容器部署应用时，通常会设置容器的申请/限制值来确定可使用的资源上限，以避免在流量高峰期无限制地占用节点资源。然而，这种方法可能会存在资源瓶颈，达到资源使用上限后可能会导致应用出现异常。为了解决这个问题，可以通过伸缩Pod的数量来分摊每个应用实例的压力。
- 节点伸缩：在增加Pod数量后，节点资源使用率可能会上升到一定程度，导致继续扩容出来的Pod无法调度。为解决这个问题，可以根据节点资源使用率伸缩节点数量，扩容Pod可以使用的资源。

关于实现自动弹性伸缩的详情请参见**7.1 使用HPA+CA实现工作负载和节点联动弹性伸缩**。

日志监控告警

- 日志
  - 控制面日志：控制面日志记录直接从Master节点上报，支持kube-controller-manager、kube-apiserver、kube-scheduler、audit四种日志类型，详情请参见**查看集群控制面日志**。
  - 应用日志：应用日志是由集群内运行的Pod生成的日志，包括运行业务应用和Kubernetes系统组件（如CoreDNS）的Pod生成的日志。CCE支持配置应用日志策略，便于日志的统一收集、管理和分析，以及按周期防爆处理，详情请参见**日志概述**。
- 监控
  - 控制面指标：控制面指标监控有助于识别控制节点的问题风险，详情请参见**监控Master指标**。
  - 应用指标：CCE支持对集群中的应用程序进行全方位的监控。除了监控Kubernetes标准指标外，您还可以在应用程序中上报符合规范的自定义指标，以提高应用程序的可观测性，详情请参见**监控概述**。

- 告警

  配合监控功能，针对指标添加告警规则可以在集群发生故障时能够及时发现问题并预警，协助您维护业务稳定性，详情请参见**自定义告警配置**。

5.2 在 CCE 中实现应用高可用部署

基本原则

在CCE中，容器部署要实现高可用，可参考如下几点：

1. 集群选择3个控制节点的高可用模式。
2. 创建节点选择在不同的可用区，在多个可用区（AZ）多个节点的情况下，根据自身业务需求合理的配置自定义调度策略，可达到资源分配的最大化。
3. 创建多个节点池，不同节点池部署在不同可用区，通过节点池扩展节点。
4. 工作负载创建时设置实例数需大于2个。
5. 设置工作负载亲和性规则，尽量让Pod分布在不同可用区、不同节点上。

操作步骤

为了便于描述，假设集群中有4个节点，其可用区分布如下所示。

```
$ kubectl get node -L topology.kubernetes.io/zone,kubernetes.io/hostname
NAME          STATUS   ROLES    AGE   VERSION                 ZONE     HOSTNAME
192.168.5.112   Ready    <none>   42m   v1.21.7-r0-CCE21.11.1.B007   zone01   192.168.5.112
192.168.5.179   Ready    <none>   42m   v1.21.7-r0-CCE21.11.1.B007   zone01   192.168.5.179
192.168.5.252   Ready    <none>   37m   v1.21.7-r0-CCE21.11.1.B007   zone02   192.168.5.252
192.168.5.8     Ready    <none>   33h   v1.21.7-r0-CCE21.11.1.B007   zone03   192.168.5.8
```

按如下定义创建负载。这里定义了两条工作负载反亲和规则podAntiAffinity。

- 第一条在可用区下工作负载反亲和，参数设置如下。
  - 权重weight：权重值越高会被优先调度，本示例设置为50。
  - 拓扑域topologyKey：包含默认和自定义标签，用于指定调度时的作用域。本示例设置为topology.kubernetes.io/zone，此为节点上标识节点在哪个可用区的标签。
  - 标签选择labelSelector：选择Pod的标签，与工作负载本身反亲和。
- 第二条在节点名称作用域下工作负载反亲和，参数设置如下。
  - 权重weight：设置为50。
  - 拓扑域topologyKey：设置为kubernetes.io/hostname。
  - 标签选择labelSelector：选择Pod的标签，与工作负载本身反亲和。

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

```
  spec:
    containers:
      - name: container-0
        image: nginx:alpine
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 50
            podAffinityTerm:
              labelSelector:                    # 选择Pod的标签，与工作负载本身反亲和。
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              namespaces:
                - default
              topologyKey: topology.kubernetes.io/zone   # 在同一个可用区下起作用
          - weight: 50
            podAffinityTerm:
              labelSelector:                    # 选择Pod的标签，与工作负载本身反亲和
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              namespaces:
                - default
              topologyKey: kubernetes.io/hostname     # 在节点上起作用
    imagePullSecrets:
      - name: default-secret
```

创建工作负载，然后查看Pod所在的节点。

```
$ kubectl get pod -owide
NAME                 READY   STATUS    RESTARTS   AGE   IP          NODE
nginx-6fffd8d664-dpwbk   1/1     Running   0          17s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-qhclc   1/1     Running   0          17s   10.0.1.133   192.168.5.252
```

将Pod数量增加到3，可以看到Pod被调度到了另外一个节点，且这个当前这3个节点是在3个不同可用区。

```
$ kubectl scale --replicas=3 deploy/nginx
deployment.apps/nginx scaled
$ kubectl get pod -owide
NAME                 READY   STATUS    RESTARTS   AGE     IP          NODE
nginx-6fffd8d664-8t7rv   1/1     Running   0          3s      10.0.0.9    192.168.5.8
nginx-6fffd8d664-dpwbk   1/1     Running   0          2m45s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-qhclc   1/1     Running   0          2m45s   10.0.1.133   192.168.5.252
```

将Pod数量增加到4，可以看到Pod被调度到了最后一个节点。可见根据工作负载反亲和规则，可以将Pod按照可用区和节点较为均匀的分布，更为可靠。

```
$ kubectl scale --replicas=4 deploy/nginx
deployment.apps/nginx scaled
$ kubectl get pod -owide
NAME                 READY   STATUS    RESTARTS   AGE     IP          NODE
nginx-6fffd8d664-8t7rv   1/1     Running   0          2m30s   10.0.0.9    192.168.5.8
nginx-6fffd8d664-dpwbk   1/1     Running   0          5m12s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-h796b   1/1     Running   0          78s     10.0.1.5    192.168.5.179
nginx-6fffd8d664-qhclc   1/1     Running   0          5m12s   10.0.1.133   192.168.5.252
```

5.3 插件高可用部署

应用场景

CCE提供了多种插件扩展集群云原生能力，涵盖了容器调度与弹性、云原生可观测、容器网络、容器存储、容器安全等方向，插件通过Helm模板方式部署，将插件中的工作负载部署至集群的工作节点。

随着插件使用的普及化，业务对插件的稳定性、可靠性保证已成为基本诉求。目前CCE服务默认的插件部署策略是工作节点之间配置了强反亲和，AZ之间配置了弱反亲和的调度策略。本文提供了CCE插件调度策略的优化实践，业务可以根据自身可靠性的要求优化插件的部署策略。

高可靠部署方案

插件一般由无状态工作负载、守护进程等组成，守护进程默认会在所有节点上部署，而无状态工作负载在高可用的情况下会设置多实例、设置AZ亲和策略以及指定节点调度来保证插件应用的高可靠性。

实例级别的高可用方案：

- **增加实例数量**：采用多实例部署方式可以有效避免单点故障造成的整个服务的不可用。

节点级别的高可用方案：

- **独占节点部署**：建议将核心插件独占Node节点部署，进行节点级别的资源限制和隔离，以避免业务应用与核心插件资源抢占。
- **多可用区部署**：采用多可用区部署可以有效避免单可用区故障造成的整个服务的不可用。

以域名解析CoreDNS插件为例，默认部署2个实例，多可用区部署为优先模式，其调度策略为节点强反亲和、AZ弱反亲和，因此集群需要2个节点才能保证所有实例正常运行，且优先将插件的Deployment实例调度到不同可用区的节点上。

以下介绍进一步提升插件SLA的一种实践方案。

增加实例数量

通过调整CoreDNS的Pod副本数量，保证高性能和高可靠性。

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“插件中心”，在右侧找到**CoreDNS域名解析**插件，单击“编辑”。

步骤2 增加实例数。

图 5-3 修改实例数量

步骤3 单击“安装”。

----结束

独占节点部署

调整CoreDNS的节点亲和策略，建议将CoreDNS独占Node节点，以避免业务应用与CoreDNS发生资源抢占。

以自定义亲和策略为例：

步骤1 登录CCE控制台，进入集群，单击左侧导航栏的“节点管理”。

步骤2 切换至“节点”页签，选择CoreDNS需要独占的节点，单击“标签与污点管理”。

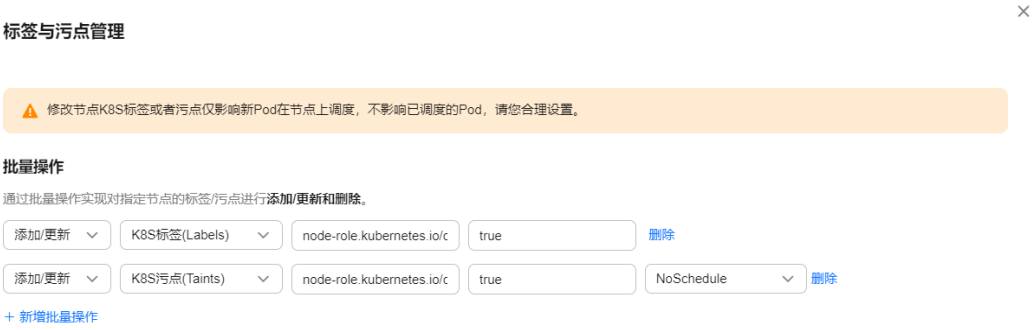
添加以下标签：

- 标签键：node-role.kubernetes.io/coredns
- 标签值：true

添加以下污点：

- 污点键：node-role.kubernetes.io/coredns
- 污点值：true
- 污点效果：NoSchedule

图 5-4 添加标签与污点



步骤3 单击左侧导航栏的“插件中心”，选择“CoreDNS域名解析”插件，单击编辑。

步骤4 在“节点亲和”中，选择“自定义亲和策略”，并添加上述节点标签。

在“容忍策略”中添加对上述污点的容忍。

图 5-5 添加容忍策略

步骤5 单击“确定”。

----结束

多可用区部署

默认的插件调度策略可以容忍单节点的故障，当业务对SLA有更高诉求，您可以在节点池界面创建不同的可用区规格节点，并且设置插件调度策略的多可用区部署为强制模式。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 创建不同可用区的节点。

重复以下步骤，选择创建不同可用区的节点。您也可以通过创建多个节点池，节点池中关联不同的可用区规格，扩容不同节点池的实例数来为集群创建不同AZ的节点。

1. 在集群控制台左侧导航栏中选择“节点管理”，切换至“节点”页签并单击右上角的“创建节点”。
2. 在节点配置步骤中，选择节点可用区。

图 5-6 创建节点

3. 根据提示填写其他必要参数后，单击“创建”。

步骤3 在左侧导航栏中选择“插件中心”，在右侧找到**CoreDNS域名解析**插件，单击“编辑”。

步骤4 设置插件的多可用区部署策略为“强制模式”，单击“安装”。

图 5-7 更新多可用区部署模式为强制模式

步骤5 在“工作负载”页签查看CoreDNS容器，切换至“kube-system”命名空间查看coredns实例的分布。

图 5-8 查看 CoreDNS 容器的部署分布

步骤6 插件的无状态工作负载已经分配到了两个可用区的节点上。

图 5-9 CoreDNS 容器分布

----结束

6 安全

6.1 安全配置概述

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。

CCE梳理了产品的功能列表和典型的应用场景，功能列表参见**功能总览**，应用场景参见**应用场景**。

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。HSS服务以及使用方法请参考**企业主机安全 HSS**。

6.2 CCE 集群安全配置建议

从安全的角度，建议您对集群做如下配置。

使用最新版本的 CCE 集群

Kubernetes社区一般4个月左右发布一个大版本，CCE的版本发布频率跟随社区版本发布节奏，在社区发布Kubernetes版本后3个月左右同步发布新的CCE版本，例如Kubernetes v1.19于2020年9月发布后，CCE于2021年3月左右发布CCE v1.19版本。

最新版本的集群修复了已知的漏洞或者拥有更完善的安全防护机制，新建集群时推荐选择使用最新版本的集群。在集群版本停止提供服务前，请及时升级到新版本。

及时跟踪处理官网发布的漏洞

CCE服务会不定期发布涉及的漏洞，用户需及时关注和处理，参见**漏洞公告**。

关闭 default 的 serviceaccount 的 token 自动挂载功能

kubernetes默认会给每个工作负载实例关联default服务账号，即在容器内挂载一个token，该token能够通过kube-apiserver和kubelet组件的认证。在没有开启RBAC的集群，得到该token相当于是得到了整个CCE集群的控制权。在开启RBAC的集群，该token所拥有的权限，取决于环境管理员给这个服务账号关联了什么角色。该服务账号的token一般是给需要访问kube-apiserver的容器使用，如CoreDNS、autoscaler、prometheus等。对于不需要访问kube-apiserver的工作负载，建议关闭服务账号的自动关联功能。

禁用方法:

- 方法一：将服务账号的automountServiceAccountToken字段设置为false。完成设置后，创建的工作负载将不会默认关联default服务账号。注意：每个命名空间都要按需设置。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
automountServiceAccountToken: false
...
```

当工作负载需要关联服务账号时，在工作负载的yaml描述文件中显式地指定。

```
...
 spec:
   template:
     spec:
       serviceAccountName: default
       automountServiceAccountToken: true
       ...
```

- 方法二：显式地关闭工作负载自动关联服务账号的功能。

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

合理配置用户的集群访问权限

CCE支持账号创建多个IAM用户。通过创建不同的用户组，并授予不同用户组不同的访问权限，然后在创建用户时将用户加入对应权限的用户组中，即可完成控制不同用户具备不同的区域（region）、是否只读的权限。同时也支持为用户或者用户组配置命名空间级别的权限。考虑到安全，建议最小化用户的访问权限。

如果主账号下需要配置多个IAM用户，应合理配置子用户和命名空间的权限。

- 配置集群权限请参考**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。

详情请参见：**设置资源配额及限制**。

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。具体使用请参考**网络策略(NetworkPolicy)**。
- 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。具体使用请参见**SecurityGroup**。
- 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 CCE 节点安全配置建议

及时跟踪处理官网发布的漏洞

节点在新的镜像发布前请参考**漏洞公告**，完成节点漏洞修复。

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**如何加固CCE集群的自动创建的安全组规则?**

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。
- 通过CCE提供的“私有镜像制作”功能，制作私有镜像作为集群的工作节点镜像。用户按照指导，基于自己的安全加固镜像制作可用于集群工作节点创建的私有镜像，制作流程参考**自定义镜像**。

禁止容器获取宿主机元数据

当用户将单个CCE集群作为共享集群，提供给多个用户来部署容器时，应限制容器访问openstack的管理地址（169.254.169.254），以防止容器获取宿主机的元数据。

修复方式参考ECS文档-**元数据获取**-使用须知。

警告

该修复方案可能影响通过ECS Console修改密码，修复前须进行验证。

步骤1 获取集群的网络模式和容器网段信息。

在CCE的“集群管理”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
  a. 以root用户登录集群的每一个node节点，执行以下命令：
  ```
  iptables -I OUTPUT -s {container_cidr} -d 169.254.169.254 -j REJECT
  ```
  其中，{container_cidr}是集群的容器网络，如10.0.0.0/16。
  为保证配置持久化，建议将该命令写入/etc/rc.local 启动脚本中。
  b. 在容器中执行如下命令访问openstack的userdata和metadata接口，验证请求是否被拦截。
  ```
  curl 169.254.169.254/openstack/latest/meta_data.json
  curl 169.254.169.254/openstack/latest/user_data
  ```

- 容器隧道网络集群
  a. 以root用户登录集群的每一个node节点，执行以下命令：
  ```
  iptables -I FORWARD -s {container_cidr} -d 169.254.169.254 -j REJECT
  ```
  其中，{container_cidr}是集群的容器网络，如10.0.0.0/16。
  为保证配置持久化，建议将该命令写入/etc/rc.local 启动脚本中。
  b. 在容器中执行如下命令访问openstack的userdata和metadata接口，验证请求是否被拦截。
  ```
  curl 169.254.169.254/openstack/latest/meta_data.json
  curl 169.254.169.254/openstack/latest/user_data
  ```
- CCE Turbo集群
  无需进行额外配置。

----结束

6.4 在 CCE 集群中使用容器的安全配置建议

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。参考**节点亲和性**。

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 设置容器网络流量访问规则。

- CCE集群：以root用户登录集群的每一个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。
- CCE Turbo集群：在集群的ENI安全组中添加出方向规则。
  a. 登录VPC控制台。
  b. 在左侧导航栏中选择“访问控制>安全组”。
  c. 找到集群对应的ENI安全组，命名格式为**{集群名}-cce-eni-{随机ID}**，单击该安全组名称配置规则。
  d. 切换至“出方向规则”页签，并单击“添加规则”，为安全组添加出方向规则。
     - 优先级：设置为1。
     - 策略：选择“拒绝”，表示禁止访问目标地址。
     - 类型：选择“IPv4”。
     - 协议端口：根据内网apiserver地址中的端口，填写“5443”。
     - 目的地址：选择“IP地址”，并填写内网apiserver地址的IP。
  e. 填写完成后，单击“确定”。

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 在 CCE 集群中使用密钥 Secret 的安全配置建议

当前CCE已为secret资源配置了静态加密，用户创建的secret在CCE的集群的etcd里会被加密存储。当前secret主要有环境变量和文件挂载两种使用方式。不论使用哪种方式，CCE传递给用户的仍然是用户配置时的数据。因此建议：

1. 用户不应在日志中对相关敏感信息进行记录；
2. 通过文件挂载的方式secret时，默认在容器内映射的文件权限为0644，建议为其配置更严格的权限，例如：

```
apiversion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
```

```
  secret:
    secretName: mysecret
    defaultMode: 256
```

其中“defaultMode： 256”，256为10进制，对应八进制的0400权限。

3. 使用文件挂载的方式时，通过配置secret的文件名实现文件在容器中“隐藏”的效果：

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: dotfile-secret
  containers:
  - name: dotfile-test-container
    image: k8s.gcr.io/busybox
    command:
    - ls
    - "-1"
    - "/etc/secret-volume"
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

这样.secret-file目录在/etc/secret-volume/路径下通过“ls -l”查看不到，但可以通过“ls -al”查看到。

4. 用户应在创建secret前自行加密敏感信息，使用时解密。

使用 Bound ServiceAccount Token 访问集群

基于Secret的ServiceAccount Token由于token不支持设置过期时间、不支持自动刷新，并且由于存放在secret中，pod被删除后token仍然存在secret中，一旦泄露可能导致安全风险。1.23版本以及以上版本CCE集群推荐使用Bound Servcie Account Token，该方式支持设置过期时间，并且和pod生命周期一致，可减少凭据泄露风险。例如：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: security-token-example
  namespace: security-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: security-token-example
      label: security-token-example
  template:
    metadata:
      annotations:
        seccomp.security.alpha.kubernetes.io/pod: runtime/default
      labels:
        app: security-token-example
        label: security-token-example
    spec:
```



```
  serviceAccountName: test-sa
containers:
  - image: ...
    imagePullPolicy: Always
    name: security-token-example
volumes:
  - name: test-projected
    projected:
      defaultMode: 420
      sources:
        - serviceAccountToken:
            expirationSeconds: 1800
            path: token
        - configMap:
            items:
              - key: ca.crt
                path: ca.crt
            name: kube-root-ca.crt
        - downwardAPI:
            items:
              - fieldRef:
                  apiVersion: v1
                  fieldPath: metadata.namespace
                path: namespace
```

具体可参考：https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/

6.6 在 CCE 集群中使用工作负载 Identity 的安全配置建议

工作负载Identity允许集群中的工作负载模拟IAM用户来访问云服务，从而无需直接使用IAM账号的AK/SK等信息，降低安全风险。

本文档介绍如何在CCE中使用工作负载Identity。

约束与限制

支持1.19.16及以上版本集群。

使用流程

1. 在CCE侧获取集群的jwks（即集群的service account token的签名公钥）。
2. 在IAM创建身份提供商。
3. 部署应用并将挂载身份提供商。
   a. 使用oidc token访问IAM获取IAM Token（用户实现）
   b. 使用IAM Token访问云服务（用户实现）

获取 CCE 集群的 jwks

步骤1 使用kubectl连接集群。

步骤2 执行如下命令获取公钥。

kubectl get --raw /openid/v1/jwks

```
# kubectl get --raw /openid/v1/jwks
{"keys":[{"use":"sig","kty":"RSA","kid":"*****","alg":"RS256","n":"*****","e":"AQAB"}]}
```


返回的字段即集群的公钥。

----结束

配置身份提供商

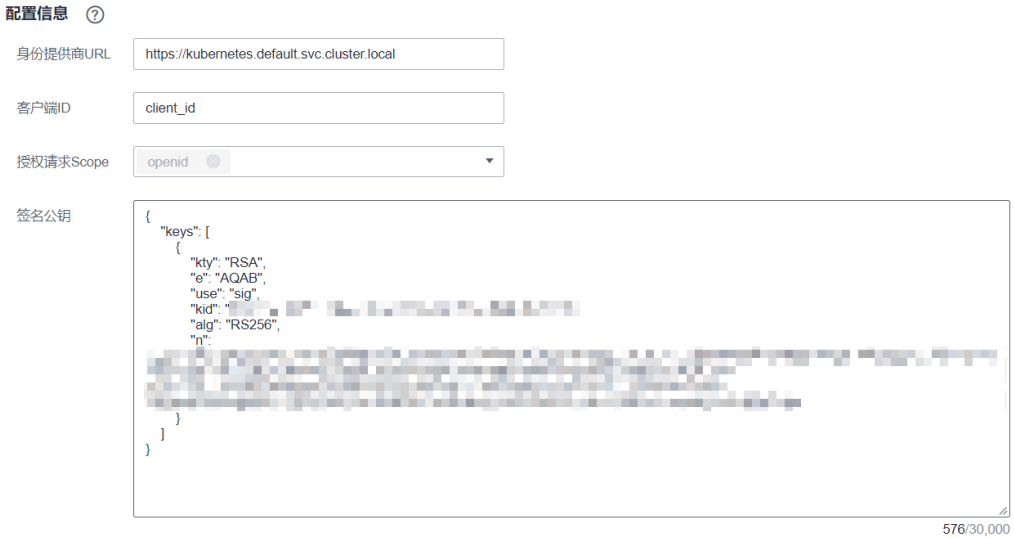
步骤1 登录IAM控制台，创建身份提供商，协议选择OpenID Connect。

步骤2 单击“确定”，然后修改身份提供商信息。

访问方式：选择**编程访问**。

配置信息

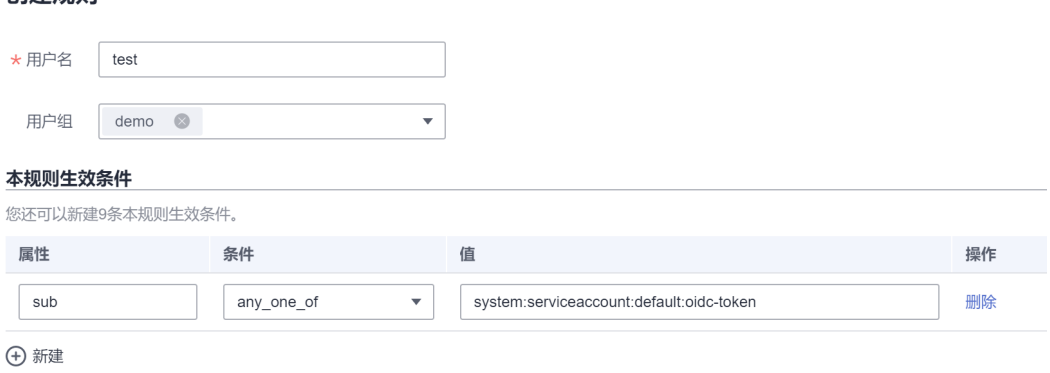
- 身份提供商URL：填写为 **https://kubernetes.default.svc.cluster.local**
- 客户端ID：填写一个ID，后续创建容器时使用。
- 签名公钥：CCE集群的jwks，获取方法请参见**获取CCE集群的jwks**。



身份转换规则

身份映射规则是将工作负载的ServiceAccount和IAM用户组做映射。

例如在集群default命名空间下创建一个名为oidc-token的ServiceAccount，映射到demo用户组（后续使用身份提供商ID访问云服务就具有demo用户组的权限）。此处属性必须是**sub**，值的格式为：**system:serviceaccount:*Namespace*:*ServiceAccountName***



规则的json格式如下。

```
[
  {
    "local": [
      {
        "user": {
          "name": "test"
        }
      },
      {
        "group": {
          "name": "demo"
        }
      }
    ],
    "remote": [
      {
        "type": "sub",
        "any_one_of": [
```



```
                "system:serviceaccount:default:oidc-token"
            ]
        }
    ]
  }
]
```

步骤3 单击“确定”。

----结束

使用工作负载 Identity

创建ServiceAccount，此处ServiceAccount的名称需要与**配置身份提供商**时填写的ServiceAccountName保持一致。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: oidc-token
```

在工作负载中使用示例如下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - mountPath: "/var/run/secrets/tokens"     # 将Kubernetes生成的serviceAccountToken挂载到/var/run/secrets/tokens/oidc-token文件内
          name: oidc-token
      imagePullSecrets:
      - name: default-secret
      serviceAccountName: oidc-token      # 前面创建的ServiceAccount的名称
      volumes:
      - name: oidc-token
        projected:
          defaultMode: 420
          sources:
          - serviceAccountToken:
              audience: client_id   # 此处取值必须为身份提供商的客户端ID
              expirationSeconds: 7200       # 过期时间
              path: oidc-token              # 路径名称，可自定义
```

创建完成后登录到容器中，/var/run/secrets/tokens/oidc-token文件内容就是Kubernetes生成的serviceAccountToken，调用**获取联邦认证token(OpenID Connect ID token方式)**接口即可获取IAM Token，响应消息头中X-Subject-Token字段即为IAM Token，从而使用IAM Token访问云服务。

说明

当serviceAccountToken超过24小时或超过80%的过期时间时，kubelet会主动轮转serviceAccountToken。

访问示例如下：

```
curl -i --location --request POST 'https://{{iam endpoint}}/v3.0/OS-AUTH/id-token/tokens' \
 --header 'X-Idp-Id: workload_identity' \
 --header 'Content-Type: application/json' \
 --data @token_body.json
```

其中：

- **{{iam endpoint}}**为IAM服务的Endpoint，具体请参见**地区和终端节点**。
- **workload_identity**即为身份提供商名称，与**配置身份提供商**中配置的名称相同。
- **token_body.json**为本地文件，内容如下所示：

  ```
  {
    "auth" : {
      "id_token" : {
        "id" : "eyJhbGciOiJSU..."
      },
      "scope": {
        "project" : {
          "id" : "46419baef4324...",
          "name" : "cn-north-4"
        }
      }
    }
  }
  ```

  - $.auth.id_token.id：此处取值为容器中/var/run/secrets/tokens/oidc-token文件的内容。
  - $.auth.scope.project.id：项目ID。获取方式请参见**获取项目ID**。
  - $.auth.scope.project.name：项目名称，例如cn-north-4。

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

应用场景

企业应用的流量大小不是每时每刻都一样，有高峰，有低谷，如果每时每刻都要保持能够扛住高峰流量的机器数目，那么成本会很高。通常解决这个问题的办法就是根据流量大小或资源占用率自动调节机器的数量，也就是弹性伸缩。

当使用Pod/容器部署应用时，通常会设置容器的申请/限制值来确定可使用的资源上限，以避免在流量高峰期无限制地占用节点资源。然而，这种方法可能会存在资源瓶颈，达到资源使用上限后可能会导致应用出现异常。为了解决这个问题，可以通过伸缩Pod的数量来分摊每个应用实例的压力。如果增加Pod数量后，节点资源使用率上升到一定程度，继续扩容出来的Pod无法调度，则可以根据节点资源使用率继续伸缩节点数量。

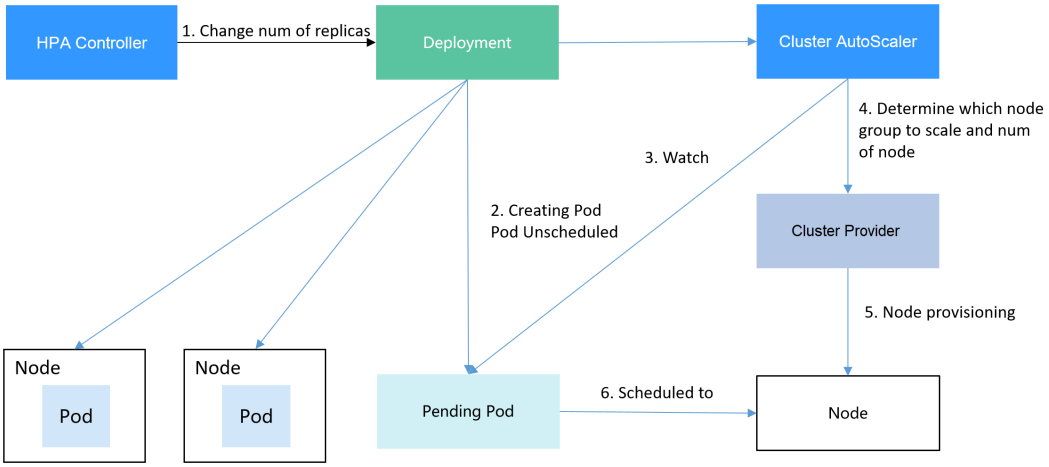
解决方案

CCE中弹性伸缩最主要的就是使用HPA（Horizontal Pod Autoscaling）和CA（Cluster AutoScaling）两种弹性伸缩策略，HPA负责工作负载弹性伸缩，也就是应用层面的弹性伸缩，CA负责节点弹性伸缩，也就是资源层面的弹性伸缩。

通常情况下，两者需要配合使用，因为HPA需要集群有足够的资源才能扩容成功，当集群资源不够时需要CA扩容节点，使得集群有足够资源；而当HPA缩容后集群会有大量空余资源，这时需要CA缩容节点释放资源，才不至于造成浪费。

如**图7-1**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。HPA和CA的工作原理详情请参见**工作负载伸缩原理**和**节点伸缩原理**。

图 7-1 HPA + CA 工作流程



使用HPA+CA可以很容易做到弹性伸缩，且节点和Pod的伸缩过程可以非常方便的观察到，使用HPA+CA做弹性伸缩能够满足大部分业务场景需求。

本文将通过一个示例介绍HPA+CA两种策略配合使用下弹性伸缩的过程，从而帮助您更好的理解和使用弹性伸缩。

准备工作

步骤1 创建一个有1个节点的集群，节点规格为2U4G及以上，并在创建节点时为节点添加弹性公网IP，以便从外部访问。如创建节点时未绑定弹性公网IP，您也可以前往ECS控制台为该节点进行手动绑定。

步骤2 给集群安装插件。

- autoscaler：节点伸缩插件。
- metrics-server：是Kubernetes集群范围资源使用数据的聚合器，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据。

步骤3 登录集群节点，准备一个算力密集型的应用。当用户请求时，需要先计算出结果后才返回给用户结果，如下所示。

1. 创建一个名为index.php的PHP文件，文件内容是在用户请求时先循环开方1000000次，然后再返回“OK!”。

```
vi index.php
```


文件内容如下：

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

2. 编写Dockerfile制作镜像。

```
vi Dockerfile
```

Dockerfile内容如下：

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

3. 执行如下命令构建镜像，镜像名称为hpa-example，版本为latest。

```
docker build -t hpa-example:latest .
```

4. （可选）登录SWR管理控制台，选择左侧导航栏的“组织管理”，单击页面右上角的“创建组织”，创建一个组织。

   如已有组织可跳过此步骤。

5. 在左侧导航栏选择“我的镜像”，单击右侧“客户端上传”，在弹出的页面中单击“生成临时登录指令”，单击 复制登录指令。

6. 在集群节点上执行上一步复制的登录指令，登录成功会显示“Login Succeeded”。

7. 为hpa-example镜像添加标签。

   docker tag [镜像名称1:版本名称1] [镜像仓库地址]/[组织名称]/[镜像名称2:版本名称2]

   - **[镜像名称1:版本名称1]**：请替换为您本地所要上传的实际镜像的名称和版本名称。
   - **[镜像仓库地址]**：可在SWR控制台上查询，**登录指令**中末尾的域名即为镜像仓库地址。
   - **[组织名称]**：请替换为**已创建的组织名称**。
   - **[镜像名称2:版本名称2]**：请替换为SWR镜像仓库中需要显示的镜像名称和镜像版本。

   示例：

   docker tag hpa-example:latest swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/hpa-example:latest

8. 上传镜像至镜像仓库。

   docker push [镜像仓库地址]/[组织名称]/[镜像名称2:版本名称2]

   示例：

   docker push swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/hpa-example:latest

   终端显示如下信息，表明上传镜像成功。

```
6d6b9812c8ae: Pushed
...
fe4c16cbf7a4: Pushed
latest: digest: sha256:eb7e3bbd*** size: **
```

   返回容器镜像服务控制台，在“我的镜像”页面，执行刷新操作后可查看到对应的镜像信息。

----结束

创建节点池和节点伸缩策略

步骤1 登录CCE控制台，进入已创建的集群，在左侧单击“节点管理”，选择“节点池”页签并单击右上角“创建节点池”。

步骤2 填写节点池配置。

- 节点数量：设置为1，表示创建节点池时默认创建的节点数为1。
- 节点规格：2核 | 4GiB

其余参数设置可使用默认值，详情请参见**创建节点池**。

步骤3 节点池创建完成后，在目标节点池所在行右上角单击“弹性伸缩”，设置弹性伸缩配置。关于节点伸缩策略设置的详细说明，请参见**创建节点伸缩策略**。

若集群中未安装CCE集群弹性引擎插件，请先安装该插件。详情请参见**CCE集群弹性引擎**。

- 弹性扩容：开启，表示节点池将根据集群负载情况自动创建节点池内的节点。
- 自定义弹性策略：单击“添加策略”，在弹出的添加规则窗口中设置参数。例如CPU分配率大于70%时，关联的节点池都增加一个节点。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。
- 弹性缩容：开启，表示节点池将根据集群负载情况自动删除节点池内的节点。例如节点资源使用率小于50%时进行缩容扫描，启动缩容。
- 伸缩配置：修改节点数范围，弹性伸缩时节点池下的节点数量会始终介于节点数范围内。
- 伸缩对象：对节点池中的节点规格单独设置开启弹性伸缩。

步骤4 设置完成后，单击“确定”。

----结束

创建工作负载

使用构建的hpa-example镜像创建无状态工作负载，副本数为1，镜像地址与上传到SWR仓库的组织有关，需要替换为实际取值。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hpa-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hpa-example
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
      - name: container-1
        image: 'hpa-example:latest' # 替换为您上传到SWR的镜像地址
        resources:
          limits:                   # limits与requests建议取值保持一致，避免扩缩容过程中出现震荡
            cpu: 500m
            memory: 200Mi
          requests:
```



```
        cpu: 500m
        memory: 200Mi
    imagePullSecrets:
    - name: default-secret
```

然后再为这个负载创建一个Nodeport类型的Service，以便能从外部访问。

说明

Nodeport类型的Service从外网访问需要为集群某个节点创建EIP，创建完后需要同步节点信息，具体请参见**同步节点信息**。如果节点已有EIP则无需再次创建。

或者您也可以创建带ELB的Service从外部访问，具体请参见**通过kubectl命令行创建-自动创建ELB**。

```
kind: Service
apiVersion: v1
metadata:
  name: hpa-example
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 31144
  selector:
    app: hpa-example
  type: NodePort
```

创建 HPA 策略

创建HPA策略，如下所示，该策略关联了名为hpa-example的负载，期望CPU使用率为50%。

另外有两条注解annotations，一条是CPU的阈值范围，最低30，最高70，表示CPU使用率在30%到70%之间时，不会扩缩容，防止小幅度波动造成影响。另一条是扩缩容时间窗，表示策略成功触发后，在缩容/扩容冷却时间内，不会再次触发缩容/扩容，以防止短期波动造成影响。

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-policy
  annotations:
    extendedhpa.metrics: '[{"type":"Resource","name":"cpu","targetType":"Utilization","targetRange":{"low":"30","high":"70"}}]'
    extendedhpa.option: '{"downscaleWindow":"5m","upscaleWindow":"3m"}'
spec:
  scaleTargetRef:
    kind: Deployment
    name: hpa-example
    apiVersion: apps/v1
  minReplicas: 1
  maxReplicas: 100
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

在控制台创建则参数填写如下所示。

观察弹性伸缩过程

步骤1 首先查看集群节点情况，如下所示，有两个节点。

```
# kubectl get node
NAME            STATUS   ROLES    AGE     VERSION
192.168.0.183   Ready    <none>   2m20s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   55m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

查看HPA策略，可以看到目标负载的指标（CPU使用率）为0%

```
# kubectl get hpa hpa-policy
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%    1         100       1          4m
```

步骤2 通过如下命令访问负载，如下所示，其中{ip:port}为负载的访问地址，可以在负载的详情页中查询。

while true;do wget -q -O- http://*{ip:port}*; done

说明

如果此处不显示公网IP地址，则说明集群节点没有弹性公网IP，请创建弹性公网IP并绑定到节点，创建完后需要同步节点信息，具体请参见**同步节点信息**。

观察负载的伸缩过程。

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          4m
hpa-policy   Deployment/hpa-example   190%/50%   1         100       1          4m23s
hpa-policy   Deployment/hpa-example   190%/50%   1         100       4          4m31s
hpa-policy   Deployment/hpa-example   200%/50%   1         100       4          5m16s
hpa-policy   Deployment/hpa-example   200%/50%   1         100       4          6m16s
hpa-policy   Deployment/hpa-example   85%/50%    1         100       4          7m16s
hpa-policy   Deployment/hpa-example   81%/50%    1         100       4          8m16s
hpa-policy   Deployment/hpa-example   81%/50%    1         100       7          8m31s
hpa-policy   Deployment/hpa-example   57%/50%    1         100       7          9m16s
hpa-policy   Deployment/hpa-example   51%/50%    1         100       7          10m
hpa-policy   Deployment/hpa-example   58%/50%    1         100       7          11m
```

可以看到4m23s时负载的CPU使用率为190%，超过了目标值，此时触发了负载弹性伸缩，将负载扩容为4个副本/Pod，随后的几分钟内，CPU使用并未下降，直到到7m16s时CPU使用率才开始下降，这是因为新创建的Pod并不一定创建成功，可能是因为资源不足Pod处于Pending状态，这段时间内在扩容节点。

到7m16s时CPU使用率开始下降，说明Pod创建成功，开始分担请求流量，到8分钟时下降到81%，还是高于目标值，且高于70%，说明还会再次扩容，到9m16s时再次扩容到7个Pod，这时CPU使用率降为51%，在30%-70%的范围内，不会再次伸缩，可以观察到此后Pod数量一直稳定在7个。

观察负载和HPA策略的详情，从事件中可以看到负载的扩容的过程和策略生效的时间，如下所示。

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 7
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization (percentage of request) above target
```

此时查看节点数量，发现节点多了两个，也就是在刚才过程中节点扩容了两个。

```
# kubectl get node
NAME          STATUS   ROLES    AGE     VERSION
192.168.0.120   Ready    <none>   3m5s    v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.136   Ready    <none>   6m58s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.183   Ready    <none>   18m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   71m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

在控制台也可以看到伸缩历史，这里可以看到CA策略执行了一次，当集群中CPU分配率大于70%，将节点池中节点数量从2扩容到3。另一个节点是autoscaler默认的根据Pod的Pending状态扩容而来，在HPA初期。

这里节点扩容过程具体是这样：

1. Pod数量变为4后，由于没有资源，Pod处于Pending状态，触发了autoscaler默认的扩容策略，将节点数增加一个。
2. 第二次节点扩容是因为集群中CPU分配率大于70%，触发了CA策略，从而将节点数增加一个，从控制台上伸缩历史可以看出来。根据分配率扩容，可以保证集群一直处于资源充足的状态。

步骤3 停止访问负载，观察负载Pod数量。

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   50%/50%    1         100       7          12m
hpa-policy   Deployment/hpa-example   21%/50%    1         100       7          13m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          14m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          23m
```


可以看到从13m开始CPU使用率为21%，18m时Pod数量缩为3个，到23m时Pod数量缩为1个。

观察负载和HPA策略的详情，从事件中可以看到负载的扩容的过程和策略生效的时间，如下所示。

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485 to 7
  Normal  ScalingReplicaSet  6m28s  deployment-controller  Scaled down replica set hpa-example-79dd795485 to 3
  Normal  ScalingReplicaSet  72s    deployment-controller  Scaled down replica set hpa-example-79dd795485 to 1
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  6m45s  horizontal-pod-autoscaler  New size: 3; reason: All metrics below target
  Normal  SuccessfulRescale  90s    horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

在控制台同样可以看到HPA策略生效历史，再继续等待，会看到节点也会被缩容一个。

这里为何没有被缩容掉两个节点，是因为节点池中这两个节点都存在kube-system namespace下的Pod（且不是DaemonSets创建的Pod），节点在什么情况下不会缩容请参见**Cluster Autoscaler工作原理**。

----结束

总结

通过上述内容可以看到，使用HPA+CA可以很容易做到弹性伸缩，且节点和Pod的伸缩过程可以非常方便的观察到，使用HPA+CA做弹性伸缩能够满足大部分业务场景需求。

7.2 CCE 容器实例弹性伸缩到 CCI 服务

CCE突发弹性引擎（对接 CCI）作为一种虚拟的kubelet用来连接Kubernetes集群和其他平台的API。Bursting的主要场景是将Kubernetes API扩展到无服务器的容器平台（如CCI）。

基于该插件，支持用户在短时高负载场景下，将部署在云容器引擎CCE上的无状态负载（Deployment）、有状态负载（StatefulSet）、普通任务（Job）、定时任务（CronJob）四种资源类型的容器实例（Pod），弹性创建到**云容器实例CCI**服务上，以减少集群扩容带来的消耗。

安装插件

1. 登录CCE控制台。
2. 选择CCE集群，单击进入CCE集群总览页面。
3. 在导航栏左侧单击“插件中心”，进入插件中心首页。
4. 选择“CCE 突发弹性引擎 (对接 CCI)”插件，单击“安装”。
5. 配置插件参数。

表 7-1 插件参数说明

插件参数	说明
选择版本	插件的版本。插件版本和CCE集群存在配套关系，更条信息可以参考**CCE突发弹性引擎（对接CCI）插件版本记录**。
规格配置	用于配置插件负载的实例数。
网络互通	勾选后将开启CCE集群和CCI两侧pod互访的功能，用户可以根据自身业务选择是否打开。详细功能介绍请参考**网络**。

工作负载下发

1. 登录CCE控制台。
2. 选择CCE集群，单击进入CCE集群总览页面。
3. 在导航栏左侧单击“工作负载”，进入工作负载首页。
4. 单击“创建工作负载”，具体操作步骤详情请参见**创建工作负载**。
5. 填写基本信息。“CCI弹性承载”选择“强制调度策略”。关于调度策略更多信息，请参考**调度负载到CCI**。

6. 进行容器配置。
7. 配置完成后，单击“创建工作负载”。
8. 在工作负载页面，选择工作负载名称，单击进入工作负载管理界面。
9. 工作负载所在节点为CCI集群，说明负载成功已调度到CCI。

插件卸载

1. 登录CCE控制台。
2. 选择CCE集群，单击进入CCE集群总览页面。
3. 在导航栏左侧单击“插件中心”，进入插件中心首页。
4. 选择“CCE 突发弹性引擎 (对接 CCI)”插件，单击“卸载”。

表 7-2 特殊场景说明

特殊场景描述	场景现象	场景说明
CCE集群无节点，卸载插件。	插件卸载失败。	bursting插件卸载时会在集群中启动Job用于清理资源，卸载插件时请保证集群中至少有一个可以调度的节点。

特殊场景描述	场景现象	场景说明
用户直接删除集群，未卸载插件。	用户在CCI侧的命名空间中有资源残留，如果命名空间有计费资源，会造成额外计费。	由于直接删除集群，没有执行插件的资源清理Job，造成资源残留。用户可以手动清除残留命名空间及其下的计费资源来避免额外计费。

关于CCE突发弹性引擎（对接CCI）更多内容详情请参见：**CCE突发弹性引擎（对接CCI）**。

7.3 基于 ELB 监控指标的弹性伸缩实践

应用现状

在使用工作负载弹性伸缩时，Kubernetes默认提供基于CPU/内存等资源使用率指标进行伸缩。但是在流量突发的场景下，基于CPU/内存使用率资源使用率数据会滞后于ELB流量指标，无法及时反映应用实际需求。因此，对于某些需要快速弹性扩缩容的业务（例如抢购和社交媒体），仅依靠资源使用率进行扩缩容可能存在伸缩不及时的问题，无法及时满足业务的实际需求。在这种情况下，通过基于ELB的QPS数据进行弹性伸缩可以更加及时地响应业务需求。

解决方案

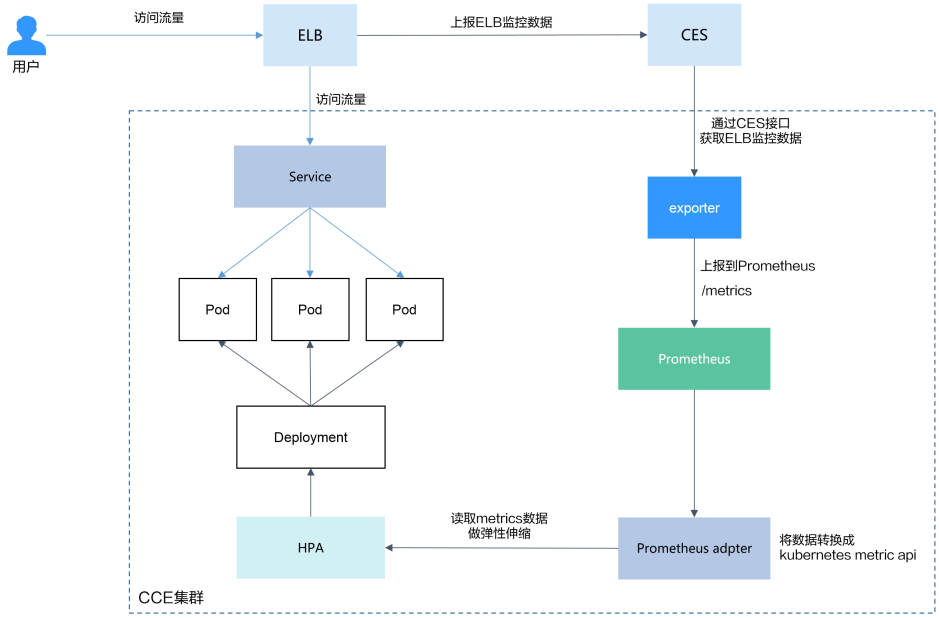
本文介绍一种基于ELB监控指标的弹性伸缩方法，相比CPU/内存使用率进行弹性伸缩，基于ELB的QPS数据弹性伸缩更有针对性，更加及时。

本方案的关键点是获取ELB的指标数据并上报到Prometheus，再将Prometheus中的数据转换成HPA能够识别的metric数据，然后HPA根据metric数据进行弹性伸缩。

基于ELB监控指标的弹性伸缩具体实施方案如下所示：

1. 开发一个Prometheus exporter，获取ELB的指标，并转化成Prometheus需要的格式，上报到Prometheus。本文使用**cloudeye-exporter**作为示例。
2. 将Prometheus的数据转换成Kubernetes metric api提供给HPA controller使用。
3. 设置HPA规则，使用ELB的监控数据作为弹性伸缩指标。

图 7-2 ELB 流量与监控数据示意图



说明

本文介绍的方法不限于ELB指标，其他指标可按照类似方法操作。

前提条件

- 本实践需要您熟悉Prometheus。
- 在集群中安装基于开源Prometheus的云原生监控插件（kube-prometheus-stack）。该插件支持v1.17及以后的集群版本。

说明

插件部署模式需选择“Server模式”。

构建 exporter 镜像

本文使用**cloudeye-exporter**实现ELB指标监控，如您需要自行开发exporter，请参见**附录：自行开发一个exporter**。

步骤1 登录一台可访问公网且安装Docker的虚拟机，编写Dockerfile。

```
vi Dockerfile
```

Dockerfile内容如下：

```
FROM ubuntu:18.04
RUN apt-get update \
  && apt-get install -y git ca-certificates curl \
    && update-ca-certificates \
      && curl -O  https://dl.google.com/go/go1.14.14.linux-amd64.tar.gz \
        && tar -zxf go1.14.14.linux-amd64.tar.gz -C /usr/local \
          && git clone https://github.com/huaweicloud/cloudeye-exporter \
            && export PATH=$PATH:/usr/local/go/bin \
              && export GO111MODULE=on \
                && export GOPROXY=https://goproxy.cn,direct \
                  && export GONOSUMDB=* \
                    && cd cloudeye-exporter \
                      && go build
CMD ["/cloudeye-exporter/cloudeye-exporter -config=/tmp/clouds.yml"]
```

步骤2 构建镜像，镜像名称为cloudeye-exporter，版本为1.0。

```
docker build --network host . -t cloudeye-exporter:1.0
```

步骤3 上传镜像至SWR镜像仓库。

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cloudeye-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，**b**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**a**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 cloudeye-exporter:1.0 swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/cloudeye-exporter:1.0

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 swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/cloudeye-exporter:1.0

   终端显示如下信息，表明上传镜像成功。

   ```
   ...
   030***: Pushed
   1.0: digest: sha256:eb7e3bbd*** size: **
   ```

   返回容器镜像服务控制台，在“我的镜像”页面，执行刷新操作后可查看到对应的镜像信息。

----结束

部署 exporter

Prometheus可以动态监测，一般来说给资源打上Prometheus对应的annotations，Prometheus会自动采集监控信息（默认为“/metrics”路径）。本文使用**cloudeye-exporter**作为示例。

Prometheus中常用的annotations如下：

- prometheus.io/scrape：true表示该资源会作为监控目标。
- prometheus.io/path：采集的url，默认为/metrics。

- prometheus.io/port：采集endpoint的端口号。
- prometheus.io/scheme：默认为http，如果为了安全设置了https，此处需要改为https。

步骤1 使用kubectl连接集群。

步骤2 创建Secret，cloudeye-exporter将使用该Secret进行认证。

1. 创建clouds.yml文件，文件内容如下：

```
global:
  prefix: "huaweicloud"
  scrape_batch_size: 10
auth:
  auth_url: "https://iam.ap-southeast-1.myhuaweicloud.com/v3"
  project_name: "ap-southeast-1"
  access_key: "********"
  secret_key: "**********"
  region: "ap-southeast-1"
```

其中：

- auth_url：IAM终端节点，可通过**地区和终端节点**获取。
- project_name：项目名称。您可在“我的凭证”页面，前往“项目列表”区域查看项目名称和项目ID。
- access_key和secret_key：可通过**访问密钥**获取。
- region：区域名称，需要与project_name中的项目对应。

2. 获取上述文件的base64加密内容字符串。

```
cat clouds.yml | base64 -w0 ;echo
```

回显如下：

```
ICAga*****
```

3. 创建clouds-secret.yaml文件，其内容如下：

```
apiVersion: v1
kind: Secret
data:
  clouds.yml: ICAga*****  #替换为base64加密字符串
metadata:
  annotations:
    description: ''
  name: 'clouds.yml'
  namespace: default  #密钥所在的命名空间，需和deployment命名空间保持一致
  labels: {}
type: Opaque
```

4. 创建密钥。

```
kubectl apply -f clouds-secret.yaml
```

步骤3 创建cloudeye-exporter-deployment.yaml文件，内容如下：

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: cloudeye-exporter
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cloudeye-exporter
      version: v1
  template:
    metadata:
      labels:
        app: cloudeye-exporter
        version: v1
```



```
    spec:
      volumes:
        - name: vol-166055064743016314
          secret:
            secretName: clouds.yml
            defaultMode: 420
      containers:
        - name: container-1
          image: swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/cloudeye-exporter:1.0  # 上文构建的
exporter镜像地址
          command:
            - /cloudeye-exporter/cloudeye-exporter
            - '-config=/tmp/clouds.yml'
          resources: {}
          volumeMounts:
            - name: vol-166055064743016314
              readOnly: true
              mountPath: /tmp
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: default-secret
      schedulerName: default-scheduler
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

创建上述工作负载。

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

步骤4 创建cloudeye-exporter-service.yaml文件。

```
apiVersion: v1
kind: Service
metadata:
  name: cloudeye-exporter
  namespace: default
  labels:
    app: cloudeye-exporter
    version: v1
  annotations:
    prometheus.io/port: '8087'      #采集endpoint的端口号
    prometheus.io/scrape: 'true'    #设置为true表示该资源会作为监控目标
    prometheus.io/path: "/metrics"  #采集的url，默认为/metrics
    prometheus.io/scheme: "http"    #默认为http，如果为了安全设置了https，此处需要改为https
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 8087
      targetPort: 8087
  selector:
    app: cloudeye-exporter
    version: v1
  type: ClusterIP
```

创建上述Service。

```
kubectl apply -f cloudeye-exporter-service.yaml
```

----结束

对接 Prometheus

Prometheus收集到监控数据后，需要将Prometheus的数据转换成Kubernetes metric api提供给HPA controller使用，这样HPA controller就能根据监控数据进行弹性伸缩。

本示例中需要监控工作负载相关联的ELB指标，因此目标工作负载需要使用负载均衡类型的Service或Ingress。

步骤1 查看需要监控的工作负载访问方式，获取ELB监听器ID。

1. 在CCE集群控制台中，选择左侧“服务发现”，在“服务”或“路由”页签下查看负载均衡类型的Service或Ingress，单击对应的负载均衡器名称跳转至ELB页面。

2. 在“监听器”页面，查看工作负载所对应的监听器，并复制该监听器ID。

步骤2 使用kubectl连接集群，添加Prometheus的配置。本示例中的采集配置为ELB指标，更多高级用法详情请参见**Configuration**。

1. 新建prometheus-additional.yaml文件，添加以下内容并保存：

```
- job_name: elb_metric
  params:
    services: ['SYS.ELB']
  kubernetes_sd_configs:
    - role: endpoints
  relabel_configs:
    - action: keep
      regex: '8087'
      source_labels:
        - __meta_kubernetes_service_annotation_prometheus_io_port
    - action: replace
      regex: ([^:]+)(?::\d+)?;(\d+)
      replacement: $1:$2
      source_labels:
        - __address__
        - __meta_kubernetes_service_annotation_prometheus_io_port
      target_label: __address__
    - action: labelmap
      regex: __meta_kubernetes_service_label_(.+)
    - action: replace
      source_labels:
        - __meta_kubernetes_namespace
      target_label: kubernetes_namespace
```

```
  - action: replace
    source_labels:
      - __meta_kubernetes_service_name
    target_label: kubernetes_service
```

2. 使用上述配置文件创建一个名为additional-scrape-configs的Secret。

```
kubectl create secret generic additional-scrape-configs --from-file prometheus-additional.yaml -n monitoring --dry-run=client -o yaml | kubectl apply -f -
```

3. 修改Prometheus对象。

```
kubectl edit prometheus server -n monitoring
```

在spec字段下添加以下内容并保存：

```
spec:
  additionalScrapeConfigs:
    key: prometheus-additional.yaml
    name: additional-scrape-configs
```

4. 执行以下命令，检验修改是否生效。

```
kubectl get secret prometheus-server -n monitoring -o jsonpath="{.data['prometheus\.yaml\.gz']}" | base64 --decode | gzip -d | grep -A3 elb
```

如有回显，则说明修改已生效。

步骤3 修改名为user-adapter-config的配置项（历史版本插件中该配置项的名称为adapter-config）。

```
kubectl edit configmap user-adapter-config -nmonitoring
```

在rules字段下添加以下内容并保存，其中seriesQuery参数中需要替换**步骤1**中获取的监听器ID。

```
apiVersion: v1
data:
  config.yaml: |-
    rules:
    - metricsQuery: sum(<<.Series>>{<<.LabelMatchers>>}) by (<<.GroupBy>>)
      resources:
        overrides:
          kubernetes_namespace:
            resource: namespace
          kubernetes_service:
            resource: service
      name:
        matches: huaweicloud_sys_elb_(.*)
        as: "elb01_${1}"
      seriesQuery: '{lbaas_listener_id="*****"}' #ELB监听器ID
    ...
```

步骤4 重新部署monitoring命名空间下的custom-metrics-apiserver工作负载。

----结束

创建 HPA 弹性伸缩规则

exporter上报到Prometheus的数据，经过Prometheus adapter监控数据转换成Kubernetes metric api后，就可以创建HPA规则实现弹性伸缩。

步骤1 创建HPA规则示例如下，使用ELB的入流量来作为扩容的标准，当m7_in_Bps（网络流入速率）的值超过1k时，会触发名为nginx的Deployment弹性伸缩。

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Object
      object:
        metric:
          name: elb01_listener_m7_in_Bps  #监控指标名称
        describedObject:
          apiVersion: v1
          kind: Service
          name: cloudeye-exporter
        target:
          type: Value
          value: 1000
```

图 7-3 已创建的 HPA 策略

步骤2 创建完后，可以对负载进行压测（也就是通过ELB访问Pod），然后HPA controller会根据设置的值计算是否需要扩容。

单击HPA策略操作栏中的“事件”，可从K8s事件中查询扩缩容记录。

图 7-4 扩缩容事件

----结束

ELB 监听器指标

通过本文方法可采集的ELB监听器指标如下：

表 7-3 ELB 监听器指标

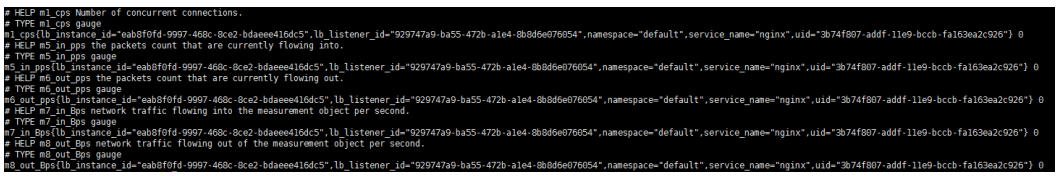
指标	指标名称	单位	说明
m1_cps	并发连接数	个	统计负载均衡器当前处理的并发连接数量。
m1e_server_rps	后端服务器重置数量	个/秒	该指标用于统计后端服务器发送至客户端的重置（RST）数据包的计数。这些重置由后端服务器生成，然后由负载均衡器转发。
m1f_lvs_rps	负载均衡器重置数量	个/秒	该指标用于统计负载均衡器生成的重置（RST）数据包的计数。
m21_client_rps	客户端重置数量	个/秒	该指标用于统计客户端发送至后端服务器的重置（RST）数据包的计数。这些重置由客户端生成，然后由负载均衡器转发。
m22_in_bandwidth	入网带宽	bit/s	该指标用于统计负载均衡器当前入网带宽。
m23_out_bandwidth	出网带宽	bit/s	该指标用于统计负载均衡器当前出网带宽。
m2_act_conn	活跃连接数	个	该指标用于统计当前处理的活跃连接数量。
m3_inact_conn	非活跃连接数	个	该指标用于统计当前处理的非活跃连接数量。

指标	指标名称	单位	说明
m4_ncps	新建连接数	个	该指标用于统计当前处理的新建连接数量。
m5_in_pps	流入数据包数	个	该指标用于统计流入负载均衡器的数据包。
m6_out_pps	流出数据包数	个	该指标用于统计流出负载均衡器的数据包。
m7_in_Bps	网络流入速率	byte/s	该指标用于统计每秒流入负载均衡器的网络流量。
m8_out_Bps	网络流出速率	byte/s	该指标用于统计每秒流出负载均衡器的网络流量。

附录：自行开发一个 exporter

Prometheus通过周期性的调用exporter的“/metrics”接口获取指标信息，应用只需要通过“/metrics”上报监控数据即可。Prometheus提供了各种语言的客户端，在应用中集成Prometheus客户端可以方便的实现“/metrics”接口，客户端具体请参见**Prometheus CLIENT LIBRARIES**，开发Exporter具体方法请参见**WRITING EXPORTERS**。

监控数据需要Prometheus的格式提供，每条数据提供ELB ID、监听器ID、Service所在的命名空间、Service名称以及Service的uid作为标签，如下所示。



获取上述数据的方法如下所示。

步骤1 查询当前所有Service。

Service的返回信息中annotations字段可以查出Service关联的ELB。

- kubernetes.io/elb.id
- kubernetes.io/elb.class

步骤2 根据上一步查询到ELB实例ID，使用**查询监听器**接口查询监听器ID。

步骤3 获取ELB监控数据。

ELB的调用CES**批量查询监控数据**接口，查询ELB的监控数据，ELB详细的监控数据指标请参见**ELB监控指标说明**。例如如下几种参数

- m1_cps：并发连接数

- m5_in_pps：流入数据包数
- m6_out_pps：流出数据包数
- m7_in_Bps：网络流入速率
- m8_out_Bps：网络流出速率

步骤4 按Prometheus的格式汇聚数据，并通过“/metrics”接口开放出去。

Prometheus客户端可以方便的实现“/metrics”接口，具体请参见**Prometheus CLIENT LIBRARIES**，开发Exporter具体方法请参见**WRITING EXPORTERS**。

----结束

8 监控

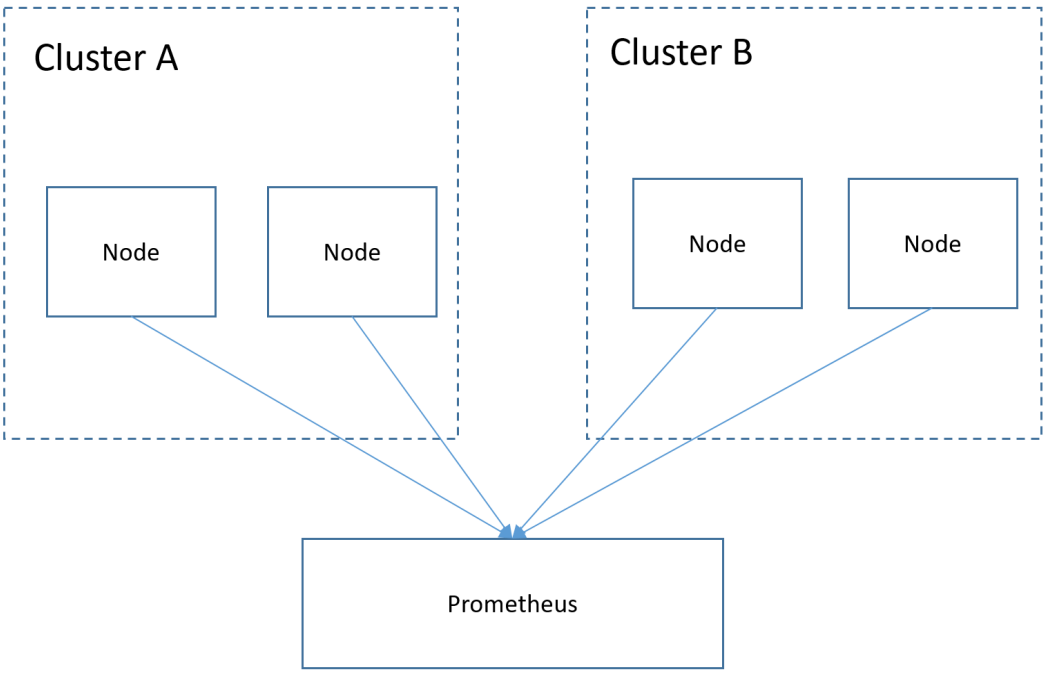
8.1 使用 Prometheus 监控多个集群

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**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

```
[ kj ~]# kubectl describe  sa  prometheus-test -n kube-system
Name:                prometheus-test
Namespace:           kube-system
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   prometheus-test-token-hdhkg
Tokens:              prometheus-test-token-hdhkg
Events:              <none>
[ kj ~]#
```

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

```
[root@h 33734 etak ~]# kubectl describe secret prometheus-test-token-hdhkg -n kube-system
Name:         prometheus-test-token-hdhkg
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: prometheus-test
              kubernetes.io/service-account.uid: 2f8171f7-6b81-436c-8343-2aed9484673d

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1123 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6Il8qWVetbGhscUdysp6NjJPU1ZVUUFxIHhOcmVLamplcE9MKm5UaGpXZ1AifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhYZNvdWs0IiwiaXVizXJuZXKlcyspby9zZXJ2aWNlYWNjb3VudC9uYWllIjoicHJvbWV0aGV1cyI6ZXN0IiwiaXVizXJuZXRlcySpby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQudWlkIjoiMmY4MTcxZjctNmI4MS00M
zZjLTgzNDMtMmFlZDk0ODQ2NzNkIiwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOnByb21ldGhldXMtdGVzdCJ9.fYVIhVfg-6pQ4XoXaGK48EeA3I4ZXDBrr6s1gNu-179Gbm957sd4rceDtoRA
mbQ2ogctp6W8wpkk-jbmXkHeeVibT7C_eolvhiejii1QrlRS7v-RJzqtP6zvwrYwqOhXPXSyi5Lr A6MQEUGxCuiPQRCoy2TFhCojloo5gVs8xnN2fdqG1phWe1IzdIn_mCumLw-6KeWGBT7n590dR5j_cB4IbwtDk3ChEvc
s RT8-1RB-bdQ2wJO-V0X3pD3BYAOicCFvJ8A6H08jU8ZJ 5mbs5Xss bGpCufowHc30XVkU85CeXuV5AB WaTICWP8VG76RZ115drqzrKmjBoE6Gq
[ kj ~]#
```

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。

```
[root@hjm-ecs prometheus-2.23.0.linux-amd64]# pwd
/root/prometheus-2.23.0.linux-amd64
[root@hjm-ecs prometheus-2.23.0.linux-amd64]#
[root@hjm-ecs prometheus-2.23.0.linux-amd64]#
[root@hjm-ecs prometheus-2.23.0.linux-amd64]# ll
total 162488
-rw------- 1 root root     5316 Jun 23 22:37 '\'
drwxr-xr-x 2 3434 3434     4096 Nov 26  2020  console_libraries
drwxr-xr-x 2 3434 3434     4096 Nov 26  2020  consoles
drwx------ 9 root root     4096 Jun 27 11:00  data
-rw------- 1 root root      943 Jun 27 11:45  k8s02_token
-rw-r--r-- 1 root root      943 Jun 22 11:58  k8s_token
-rw-r--r-- 1 3434 3434    11357 Nov 26  2020  LICENSE
-rw-r--r-- 1 3434 3434     3420 Nov 26  2020  NOTICE
-rwxr-xr-x 1 3434 3434 88153522 Nov 26  2020  prometheus
-rw------- 1 root root     5501 Jun 27 10:46  prometheus.yml
-rw-r--r-- 1 3434 3434      926 Nov 26  2020  prometheus.yml.bak
-rwxr-xr-x 1 3434 3434 78172790 Nov 26  2020  promtool
[root@hjm-ecs prometheus-2.23.0.linux-amd64]#
```

分别是目标集群的两个token

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。

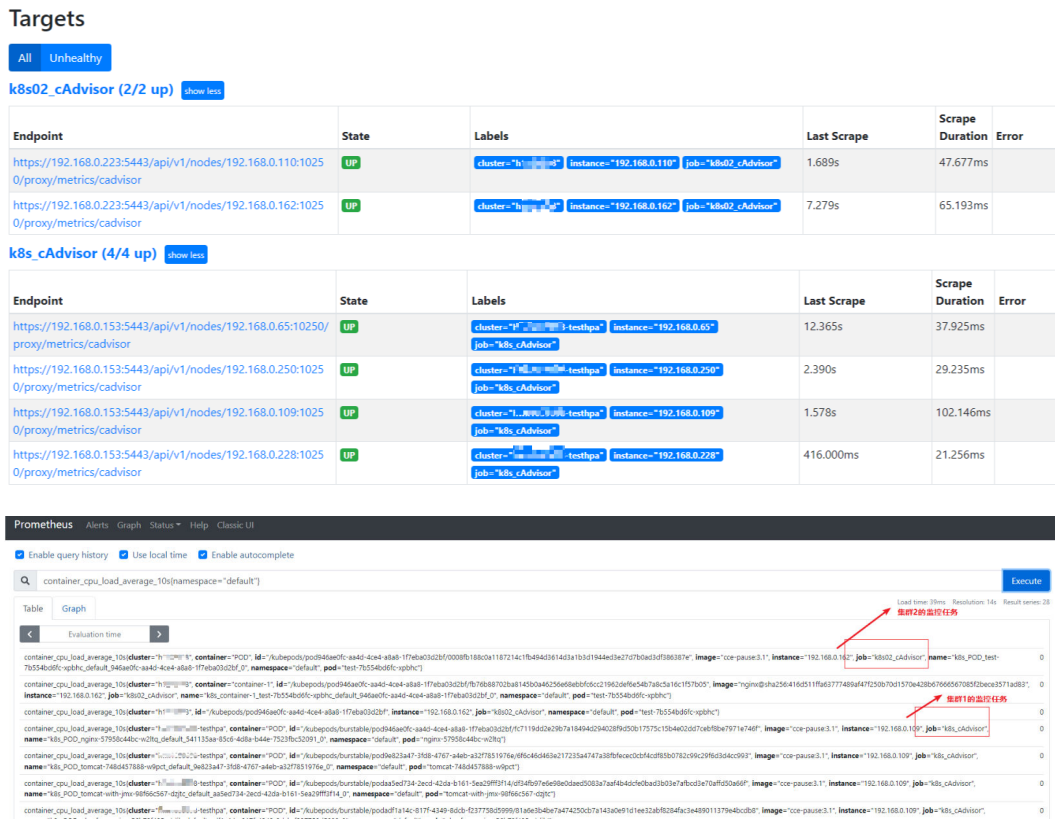
```
 - job_name: k8s_cAdvisor
   scheme: https
   bearer_token_file: k8s_token #上一步中的token文件
   tls_config:
     insecure_skip_verify: true
   kubernetes_sd_configs:  #kubernetes 自动发现配置
   - role: node    #node类型的自动发现
     bearer_token_file: k8s_token #上一步中的token文件
     api_server: https://192.168.0.153:5443  #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CCE AI 套件 (NVIDIA GPU)和云原生监控插件。
  - CCE AI 套件 (NVIDIA GPU)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。
  - 云原生监控插件（kube-prometheus-stack）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。

    说明

    - 插件部署模式需选择“Server模式”。
    - 对接Grafana的配置在3.9.0以下版本的云原生监控插件中支持。对于3.9.0及以上版本的插件，如果存在使用Grafana的需求，请单独Grafana插件。

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 swr.cn-east-3.myhuaweicloud.com/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 swr.cn-east-3.myhuaweicloud.com/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: "swr.cn-east-3.myhuaweicloud.com/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
   ```

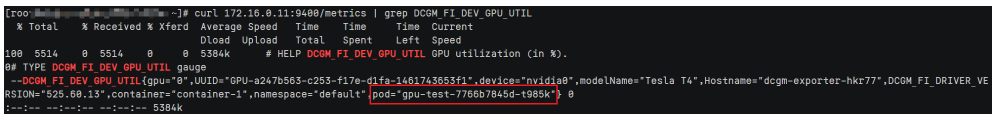
   回显如下：

   ```
   # 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。

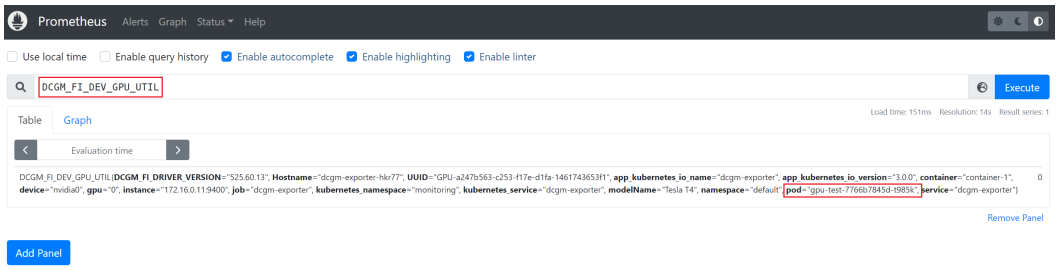
   ```
   curl 172.16.0.11:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
   ```

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），详情请参见**使用Prometheus插件监控**。

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 CCE 集群选型建议

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

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

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支持的最新商用版本。

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 网络模型对比

对比维度	容器隧道网络	VPC网络	云原生网络2.0
适用场景	• 一般容器业务场景。 • 对网络时延、带宽要求不是特别高的场景。	• 对网络时延、带宽要求高。 • 容器与虚机IP互通，使用了微服务注册框架，如Dubbo、CSE等。	• 对网络时延、带宽要求高，高性能场景。 • 容器与虚机IP互通，使用了微服务注册框架的，如Dubbo、CSE等。
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 Standard集群	CCE Standard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

对比维度	容器隧道网络	VPC网络	云原生网络2.0
IP地址管理	• 容器网段单独分配 • 节点维度划分地址段，动态分配（地址段分配后可动态增加）	• 容器网段单独分配 • 节点维度划分地址段，静态分配（节点创建完成后，地址段分配即固定，不可更改）	容器网段从VPC子网划分，无需单独分配
网络性能	基于vxlan隧道封装，有一定性能损耗。	无隧道封装，跨节点通过VPC 路由器转发，性能好，媲美主机网络。	容器网络与VPC网络融合，性能无损耗
组网规模	最大可支持2000节点	受限于VPC路由表能力，适合中小规模组网，建议规模为1000节点及以下。 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

如需了解更多信息，请参见**容器网络模型对比**。

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**10.1 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。

如需了解更多信息，请参见**iptables与IPVS如何选择**。

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。

使用IPv6双栈请参考**IPv4/IPv6双栈网络**。

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	于v1.23版本GA（Generally Available） 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 v1.25.3-r0及以上	暂不支持创建kata安全容器 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“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。

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。具体请参见“网络规划”。

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**IPv4/IPv6双栈网络**。

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

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. 选择“网络>虚拟私有云 VPC”。
4. 单击“创建虚拟私有云”。
5. 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。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。	亚太-新加坡
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。 目前支持网段范围： 10.0.0.0/8~24 172.16.0.0/12~24 192.168.0.0/16~24	192.168.0.0/16

参数	说明	取值样例
企业项目	创建VPC时，可以将VPC加入已启用的企业项目。 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。 关于创建和管理企业项目的详情，请参见《企业管理用户指南》。	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。 标签的命名规则请参见表9-5。	• 键：vpc_key1 • 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
高级配置		
网关	子网的网关。 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

参数	说明	取值样例
DHCP租约时间	DHCP租约时间是指DHCP服务器自动分配给客户端的IP地址的使用期限。超过租约时间，IP地址将被收回，需要重新分配。DHCP租约时间改后，会在一段时间后自动生效（与您的DHCP租约时长有关），如果需要立即生效，请重启ECS或者在实例中主动触发DHCP更新。 **注意** 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。	365天或300小时
标签	子网的标示，包括键和值。可以为子网创建10个标签。 标签的命名规则请参见表9-6。	• 键：subnet_key1 • 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。 • 对于同一虚拟私有云键值唯一。 • 长度不超过36个字符。 • 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。 • 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。 • 对于同一子网键值唯一。 • 长度不超过36个字符。 • 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。 • 由英文字母、数字、下划线、点、中划线、中文字符组成。	subnet-01

6. 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。

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： • 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。 • 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。	亚太-新加坡 -
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
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。 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。 关于创建和管理企业项目的详情，请参见**《企业管理用户指南》**。	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

1. 在共享带宽列表页，单击操作列的“添加公网IP”。

图 9-2 加入共享带宽入口

2. 将IPv6地址加入共享带宽。

图 9-3 添加 IPv6 双栈网卡

3. 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 制作 CCE 节点自定义镜像

CCE自定义镜像制作基于开源工具HashiCorp Packer(>=1.7.2)以及开源插件实现，并提供了cce-image-builder配置模板帮助您快速制作符合要求的自定义镜像。

Packer是一款可以创建自定义镜像的开源工具。Packer包含构建器（Builder）、配置器（Provisioner）、后处理器（Post-Processor）三个组件，支持通过json或者hcl格式的模板文件，可以灵活组合这三种组件并行地、自动化地创建镜像文件。

Packer作为镜像制作的工具有如下优势：

1. 构建过程自动化：创建镜像的过程变成可以通过Packer配置文件的形式固化，支持自动化构建。
2. 云平台兼容性强：原生支持对接绝大多数的云平台，也包括各类第三方插件。
3. 配置文件易用性高：Packer配置文件中模块职责清晰，参数定义简单直观，学习成本低。

4. 镜像构建功能全面：原生支持较多常用功能模块，例如Provisioner中支持使用远程执行脚本的shell模块、支持远程传输文件的file模块、支持暂停流程的breakpoint模块。

约束与限制

- CCE节点镜像使用建议：
  - 节点镜像推荐优先使用CCE服务维护的默认节点镜像。相关镜像经过严格的测试，且能获得最新的更新推送，具有更好的兼容性、稳定性和安全性。
  - 如果您有特殊场景需要使用自定义镜像功能，请使用CCE提供的基础镜像来制作自定义镜像。
  - 基础镜像中预置了节点运行依赖的组件包，该包的版本会跟着集群版本变更。对于自定义镜像，CCE不再提供组件包的更新推送。
- 自定义镜像时，请**谨慎修改内核参数**，部分内核参数过大或者过小时都会影响系统运行效率，参考值请参见**修改节点内核参数**。

  已知存在影响的内核参数有：tcp_keepalive_time、tcp_max_tw_buckets、somaxconn、max_user_instances、max_user_watches、netdev_max_backlog、net.core.wmem_max、net.core.rmem_max。

  如修改节点内核参数，请您在测试环境中自行完成充分的测试验证，再应用于生产环境。

使用须知

- 制作镜像前需要准备：
  - ECS执行机：一台ECS x86服务器作为Linux执行机，建议选择CentOS7操作系统并绑定EIP支持访问公网，便于后续安装Packer工具。
  - 认证凭据：制作镜像需要获取具有相关权限用户或者租户的AK/SK认证凭据，请提前获取，详情请参考**IAM 如何获取访问密钥AK/SK**。
  - 安全组：由于Packer通过创建临时ECS服务器并使用密钥对SSH登录，需要提前确保临时ECS服务器指定的安全组放开**TCP:22**端口通道，允许执行机SSH登录。详情请参考**安全组配置示例**。
- 制作过程中：
  - 制作镜像的过程中必须按照操作指南进行制作，防止一些不可预期问题出现。
  - 根据基础镜像创建出来的虚机，默认SSH登录用户需要支持sudo root权限，或者拥有root权限。
- 制作结束时：
  - 制作过程会消耗一定计费资源，主要涉及ECS服务器、EVS云硬盘、弹性IP及带宽和IMS镜像。正常场景制作成功或者失败后会自动释放。但制作完成后，建议再次确认资源已彻底释放，以避免非预期的消耗。

获取镜像 ID

步骤1 登录CCE控制台，单击集群名称进入集群，在左侧导航栏中选择“节点管理”。

步骤2 单击右上角“创建节点”，在“操作系统”选项中选择“私有镜像”。

步骤3 单击“查看CCE基础镜像信息”，即可复制操作系统的镜像ID。

须知

不同区域的镜像ID存在区别，切换区域后请重新获取。

图 9-5 获取镜像 ID

----结束

制作镜像

步骤1 下载cce-image-builder

登录ECS执行机，下载cce-image-builder并解压。

```
wget https://cce-north-4.obs.cn-north-4.myhuaweicloud.com/cce-image-builder/cce-image-builder.tgz
tar zvxf cce-image-builder.tgz
cd cce-image-builder/
```

说明

cce-image-builder主要包含：

- turbo-node.pkr.hcl 镜像制作使用的Packer配置模板，修改请参见**步骤3**。
- scripts/* 模板预置的CCE镜像制作。建议不要修改，避免影响最终镜像可用性。
- user-scripts/* **模板预置的自定义包脚本目录**。以example.sh为例，制作自定义镜像时将自动上传至临时服务器并执行。
- user-packages/* **模板预置的自定义包目录**。制作自定义镜像时，以example.package为例，制作自定义镜像时将自动上传至临时服务器的/tmp/example.package路径。

步骤2 安装Packer

手动下载并安装**HashiCorp Packer**，建议参考**官方指导**执行。

说明

Packer版本要求：packer = 1.10.0

以CentOS 7执行机为例，执行如下命令自动安装packer（**以官方指导为准**）：

```
# 配置Packer的yum源后安装Packer
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
sudo yum -y install packer-1.10.0

# 配置alias用于避免OS中同名packer二进制影响,并检查packer版本
rpm -q packer
alias packer=$(rpm -ql packer)
packer -v
```

步骤3 定义Packer模板参数

cce-image-builder/turbo-node.pkr.hcl文件中定义了packer构建镜像的过程，涉及的参数字段如下。关于pkr.hcl文件更详细的说明请参考**Packer官网文档**。

说明

- ***variables/variable***

  *变量定义。turbo-node.pkr.hcl中主要定义了构建所需填写的参数，支持用户按需修改或填写，参数定义请参见 **表1**。*

- ***packer***

  packer模块定义。其中required_plugins定义了Packer的插件依赖，包括插件来源与插件版本范围，packer init时会自动下载插件并初始化。修改如下：

```
packer {
  required_plugins {
    huaweicloud = {
      version = ">= 1.0.4"
      source  = "github.com/huaweicloud/huaweicloud"
    }
  }
}
```

- ***source***

  source定义。通过引用上述定义的变量，自动配置插件创建ECS所需的参数，无需用户手动填写。

- ***build***

  build定义。按顺序从上到下依次执行，支持文件上传file模块、脚本执行shell模块等常用模块，对应的脚本及文件分别存放在cce-image-builder下user-scripts和user-packages目录。

  样例参考：

```
build {
  sources = ["source.huaweicloud-ecs.builder"]

# Example:
  provisioner "file" {
    source      = "<source file path>"
    destination = "<destination file path>"
  }

  provisioner "shell" {
    scripts = [
      "<source script file: step1.sh>",
      "<source script file: step2.sh>"
    ]
  }

  provisioner "shell" {
    inline = ["echo foo"]
  }
}
```

步骤4 设置环境变量

在执行机上设置以下环境变量：

```
export REGION_NAME=xxx
export IAM_ACCESS_KEY=xxx
export IAM_SECRET_KEY=xxx
export ECS_VPC_ID=xxx
export ECS_NETWORK_ID=xxx
export ECS_SECGRP_ID=xxx
export CCE_SOURCE_IMAGE_ID=xxx
export PKR_VAR_ecs_flavor=xxx
```


表 9-8 variables 配置参数说明

参数	参数说明	备注
REGION_NAME	所属项目区域。	请在**我的凭证**下查看所属区域。
IAM_ACCESS_KEY	用户认证凭据Access Key。	建议临时申请，制作完成后删除。
IAM_SECRET_KEY	用户认证凭据Secret Key。	建议临时申请，制作完成后删除。
ECS_VPC_ID	虚拟私有云ID。	临时ECS服务器使用，需要与执行机一致。
ECS_NETWORK_ID	子网的网络ID。	临时ECS服务器使用，建议与执行机一致，非子网的子网ID。
ECS_SECGRP_ID	安全组ID。	临时ECS服务器使用，该安全组入方向的22端口需要放通执行机的公网IP，保证执行机可通过SSH登录临时服务器。
CCE_SOURCE_IMAGE_ID	CCE服务的最新节点镜像ID。	请参考**获取镜像ID**。
PKR_VAR_ecs_flavor	临时ECS服务器的规格。	请填写CCE支持的节点规格，建议规格为2U4G及以上。关于规格名称详情请参见**规格清单（X86）**。

备注：其他参数通常无需配置，默认即可。如需修改，可以参考*turbo-node.pkr.hcl*中variable定义中的description描述通过环境变量配置。

以ECS规格变量ecs_az为例，未指定时选择随机可用区，如需指定可以按照如下格式通过环境变量配置。其他参数类似。

```
# export PKR_VAR_<variable name>=<variable value>
export PKR_VAR_ecs_az=xxx
```

步骤5 设置自定义脚本及文件

参考pkr.hcl文件中**build**字段定义的file和shell模块，编写对应的脚本及文件，并分别存放在cce-image-builder下的user-scripts和user-packages目录。

须知

自定义镜像时，请**谨慎修改内核参数**，部分内核参数过大或者过小时都会影响系统运行效率，参考值请参见**修改节点内核参数**。

已知存在影响的内核参数有：tcp_keepalive_time、tcp_max_tw_buckets、somaxconn、max_user_instances、max_user_watches、netdev_max_backlog、net.core.wmem_max、net.core.rmem_max。

如修改节点内核参数，请您在测试环境中自行完成充分的测试验证，再应用于生产环境。

步骤6 制作自定义镜像

定制修改结束后，执行以下命令制作镜像并等待镜像制作完成，通常需要耗时3~5min。

```
make image
```

说明

构建封装脚本packer.sh中，

- 隐私和安全考虑，默认关闭了packer自动访问hashicorp.com检查版本的功能。
  ```
  export CHECKPOINT_DISABLE=false
  ```
- 构建过程可见性与可追溯考虑，默认开启了debugging详细日志，并指定了本地packer构建日志packer_{timestamp}.log以在构建时打包至镜像/var/log/目录。如涉及敏感信息，移除相关逻辑即可。
  ```
  export PACKER_LOG=1
  export PACKER_BUILD_TIMESTAMP=$(date +%Y%m%d%H%M%S)
  export PACKER_LOG_PATH="packer_$PACKER_BUILD_TIMESTAMP.log"
  ```

更多packer配置详细参见**官网说明**。

执行结束日志：

```
==> huaweicloud-ecs.builder: Setting a 15m0s timeout for the next provisioner...
==> huaweicloud-ecs.builder: Provisioning with shell script: /tmp/packer-shell1759174699
==> huaweicloud-ecs.builder: Setting a 15m0s timeout for the next provisioner...
==> huaweicloud-ecs.builder: Uploading packer_20210530185050.log => /var/log/packer_20210530185050.log
    huaweicloud-ecs.builder: packer_20210530185050.log 43.63 KiB / 43.50 KiB [==============================] 100.29% 0s
==> huaweicloud-ecs.builder: Stopping server: 9c901ac9-37b5-40af-934e-7190e6fa088e ...
    huaweicloud-ecs.builder: Waiting for server to stop: 9c901ac9-37b5-40af-934e-7190e6fa088e ...
==> huaweicloud-ecs.builder: Creating the image: image-by-packer-20210530185050
==> huaweicloud-ecs.builder: Waiting for image image-by-packer-20210530185050 to become available ...
    huaweicloud-ecs.builder: Image: 64e940f4-d674-4ae1-89cc-299501581c59
==> huaweicloud-ecs.builder: Deleted temporary floating IP '494617cc-a7c9-442a-b3e8-3b90c2c3f804' (94.74.101.22)
==> huaweicloud-ecs.builder: Terminating the source server: 9c901ac9-37b5-40af-934e-7190e6fa088e ...
==> huaweicloud-ecs.builder: Deleting volume: bf769e29-e1fd-407b-bbec-79f353a3e671 ...
==> huaweicloud-ecs.builder: Deleting temporary keypair: packer_60b36e0b-1f16-acc5-df04-d045aba70856 ...
Build 'huaweicloud-ecs.builder' finished after 3 minutes 53 seconds.

==> Wait completed after 3 minutes 53 seconds

==> Builds finished. The artifacts of successful builds are:
--> huaweicloud-ecs.builder: An image was created: 64e940f4-d674-4ae1-89cc-299501581c59
[Sun May 30 18:54:45 CST 2021] packer.sh finish.
```

步骤7 构建文件清理

清理执行机上构建文件，主要涉及turbo-node.pkr.hcl中认证凭据清理。

- 建议直接释放执行机，如认证凭据为临时申请建议直接删除；
- 如需自动构建，建议自行在配置文件中添加post-processor实现相关功能。

----结束

常见问题

- 执行packer制作镜像过程，会自动从github获取最新Huawei Cloud ECS开源插件。此过程会由于网络环境原因导致获取失败，如下所示。

```
#   Start to packer(/usr/bin/packer) with [turbo-node.pkr.hcl] ...
...
[Fri Dec 31 17:06:15 CST 2021] packer version checked: 1.7.8
[Fri Dec 31 17:06:15 CST 2021] start to init/build with [turbo-node.pkr.hcl] ...
Failed getting the "github.com/huaweicloud/huaweicloud" plugin:
1 error occurred:
        * could not get sha256 checksum file for github.com/huaweicloud/huaweicloud version 0.4.0. Is the file present on the release and correctly named ? Get "https
://github.com/huaweicloud/packer-plugin-huaweicloud/releases/download/v0.4.0/packer-plugin-huaweicloud_v0.4.0_SHA256SUMS": unexpected EOF


Error: no plugin installed for github.com/huaweicloud/huaweicloud >= 0.4.0

Did you run packer init for this project ?


[Fri Dec 31 17:08:16 CST 2021] packer.sh finish.
```

该问题可以通过如下两种方案解决：

- 在网络环境相对较优的香港等区域创建执行机，对接到原区域创建自定义镜像，例如北京4：

  export REGION_NAME=cn-north-4

- 手动下载对应插件并初始化至对应本地插件路径。

  插件对应github发布地址为：**https://github.com/huaweicloud/packer-plugin-huaweicloud/releases**

  以CentOS 7.6 x86执行机为例，下载的插件为packer-plugin-huaweicloud_v1.0.4_x5.0_linux_amd64.zip，参考如下命令进行初始化：

  ```
  PLUGIN_PATH="$HOME/.packer.d/plugins/github.com/huaweicloud/huaweicloud"
  mkdir -p $PLUGIN_PATH
  unzip packer-plugin-huaweicloud_v1.0.4_x5.0_linux_amd64.zip -d /tmp/
  cp /tmp/packer-plugin-huaweicloud_v1.0.4_x5.0_linux_amd64 $PLUGIN_PATH/
  sha256sum /tmp/packer-plugin-huaweicloud_v1.0.4_x5.0_linux_amd64 | awk '{print $1}' > $PLUGIN_PATH/packer-plugin-huaweicloud_v1.0.4_x5.0_linux_amd64_SHA256SUM
  ll $PLUGIN_PATH/*
  ```

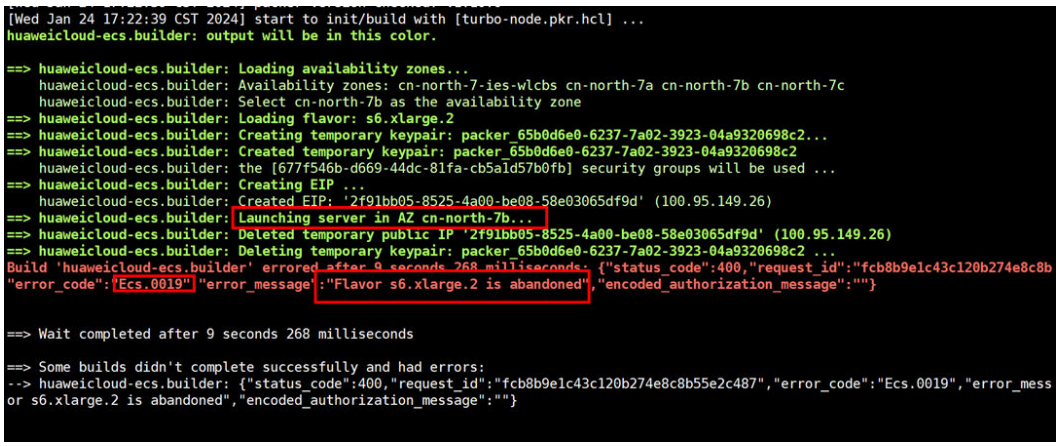
  再执行make image命令制作镜像。

- 如果出现报错PublicIp type is invalid，请查看不同环境EIP的type类型（参见**申请EIP**），然后执行**export PKR_VAR_eip_type='xxx'**后再进行镜像制作。

  例如：

  ```
  export PKR_VAR_eip_type='5_bgp'
  ```

- 如果出现错误码Ecs.0019，错误信息为Flavor xxxx is abandoned，可能是该规格在对应可用区不可用，请重试或者在turbo-node.pkr.hcl文件中更换规格即可。

- 如果出现错误信息为no such host，可能是由于当前IAM域名无法解析。

  请在执行机上设置以下环境变量，其中*{IAM endpoint}*为当前区域的IAM域名。

  ```
  export PKR_VAR_auth_url='{IAM endpoint}'
  ```

9.4 创建节点时执行安装前/后脚本

应用现状

在创建节点时，对于需要在节点上安装一些工具或者进行安全加固等操作时，可以使用安装前/后脚本实现。本文为您提供正确使用安装前/后脚本的指导，帮助您了解和使用安装前/后脚本。如果有进阶的安装脚本使用需求，可以将脚本存放在OBS中，避免脚本字符数超限等问题，详情请参见**9.5 创建节点时使用OBS桶实现自定义脚本注入**。

注意事项

- 请避免使用执行耗时过长的安装前/后脚本。

  安装前脚本的时间限制为15min、安装后脚本的时间限制为30min，如果指定时间内节点没能到达可用状态，则会触发节点的回收操作。因此需要避免执行耗时过长的安装前/后脚本。

- 请避免在安装后脚本中直接使用reboot指令。

  当前CCE会在执行完节点必备组件的安装之后，再执行安装后脚本。当安装后脚本执行完之后才会将节点状态置为可用状态。如果直接使用reboot命令，可能会导致节点在上报状态之前就被重启，从而造成节点无法在30min内到达运行中状态，触发超时回滚。因此请尽量避免使用reboot指令。

  如果确实需要重启节点，可以选择：

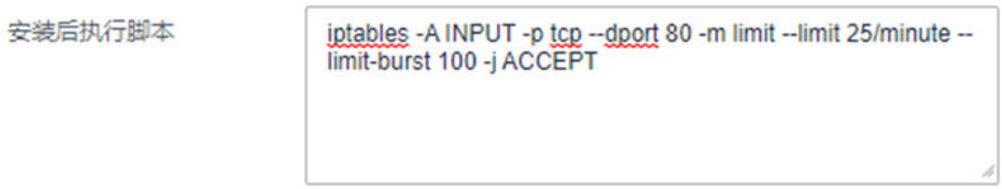
  - 在安装后脚本中使用**shutdown -r <时间>**命令，延迟重启。例如，使用**shutdown -r 1**命令可以延迟1分钟重启。
  - 在节点状态为可用状态之后，手动进行节点重启。

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”，单击要创建节点的集群进入集群控制台。

步骤2 在集群控制台左侧导航栏中选择“节点管理”，切换至“节点”页签，单击右侧“创建节点”，并设置节点参数。

步骤3 在“高级配置”中，填写安装前/后执行脚本。



例如，您可以通过安装后执行脚本创建iptables规则，限制每分钟最多只能有25个TCP协议的数据包通过端口80进入，并且在超过这个限制时，允许最多100个数据包通过，以防止DDoS攻击。

```
iptables -A INPUT -p tcp --dport 80 -m limit --limit 25/minute --limit-burst 100 -j ACCEPT
```

说明

此处的脚本示例仅供参考。

步骤4 完成以上配置后，您可以设置需要购买的节点数量，并单击“下一步：规格确认”。

步骤5 单击“提交”，开始创建节点。

----结束

9.5 创建节点时使用 OBS 桶实现自定义脚本注入

应用现状

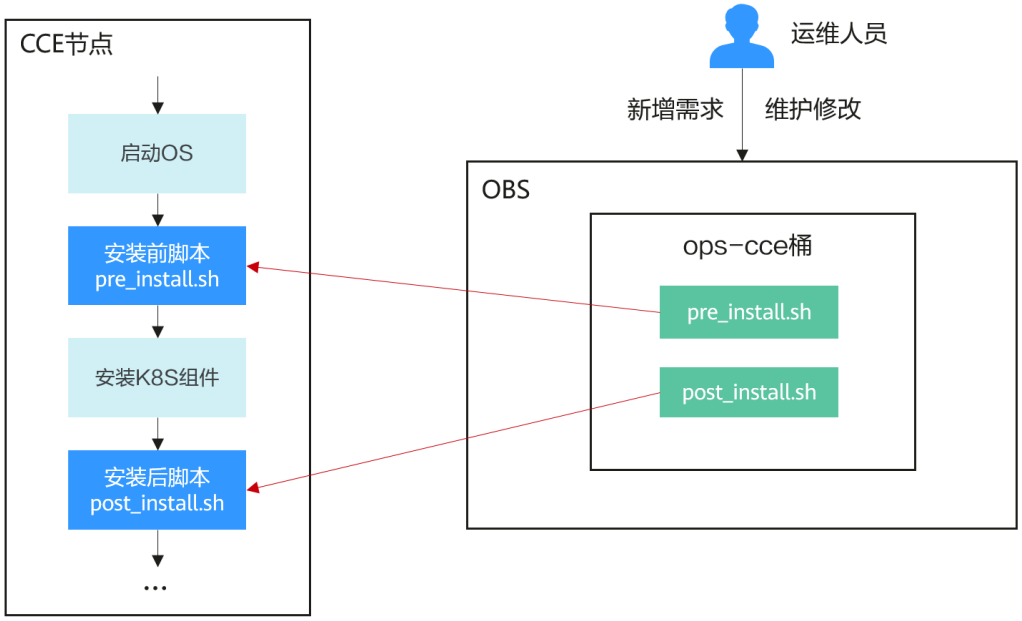
对于需要在节点上提前安装一些工具或者做用户自定义的安全加固等操作时，需要在创建节点的时候注入一些脚本。CCE创建节点提供了Kubernetes安装前和安装后两处注入脚本的功能。但是使用通常碰到如下限制：

- 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-monit
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 9-6 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。

1. 配置tools/cce目录的策略，选择“只读模板”。

图 9-7 配置对象策略

2. 修改匿名用户、指定对象、条件的配置信息。

图 9-8 匿名用户

图 9-9 指定对象

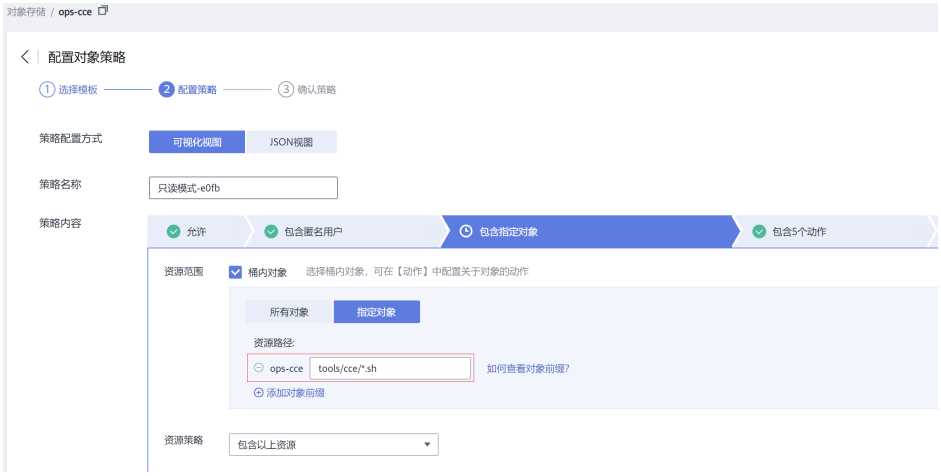


图 9-10 设置条件

设置的条件中主要包含两个：UserAgent和源IP。

- UserAgent类似密钥的作用，访问时必须带上指定的User-Agent请求头以及对应的设置的key值。
- 源IP主要防止外网访问，配置了100.0.0.0/8,10.0.0.0/8,172.16.0.0/12,192.168.0.0/16,214.0.0.0/8。其中，100.0.0.0/8和214.0.0.0/8是内网地址；10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16是通常的私网地址。

OBS的策略的详细配置方法请参见**配置对象策略**和**桶策略参数说明**。

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。

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.ap-southeast-1.myhuaweicloud.com/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.ap-southeast-1.myhuaweicloud.com/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

9.6 通过 kubectl 对接多个集群

应用现状

kubectl命令行工具使用kubeconfig配置文件来查找选择集群所需的认证信息，并与集群的API服务器进行通信。默认情况下，kubectl会使用“$HOME/.kube/config”文件作为访问集群的凭证。

在CCE集群的日常使用过程中，我们通常需要同时管理多个集群，因此在使用kubectl命令行工具连接集群时需要经常切换kubeconfig配置文件，为日常运维带来许多不便。本文将为您介绍如何便捷地使用同一个kubectl客户端连接多个集群。

说明

用于配置集群访问的文件称为kubeconfig配置文件，并不意味着文件名称为kubeconfig。

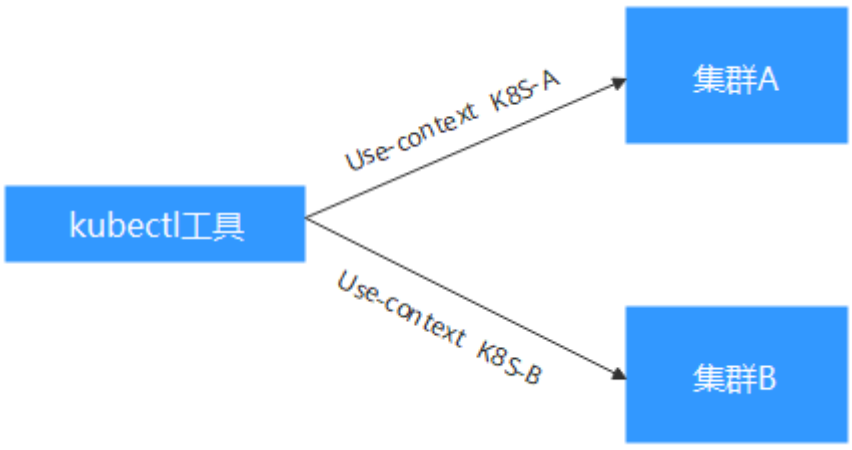
解决方案

在K8s集群的运维中，多集群之间的切换是无法避免的问题，常见的集群切换方案如下：

- **方案一**：您可以通过指定kubectl的“--kubeconfig”参数来选择每个集群所使用的kubeconfig配置文件，并可使用alias别名的方式来简化命令。
- **方案二**：将多个kubeconfig文件中的集群、用户和凭证合并成一个配置文件，并使用“kubectl config use-context”命令进行集群切换。

  该方案与方案一相比，需要手动配置kubeconfig文件，相对来说较为复杂。

图 9-11 kubectl 对接多集群示意

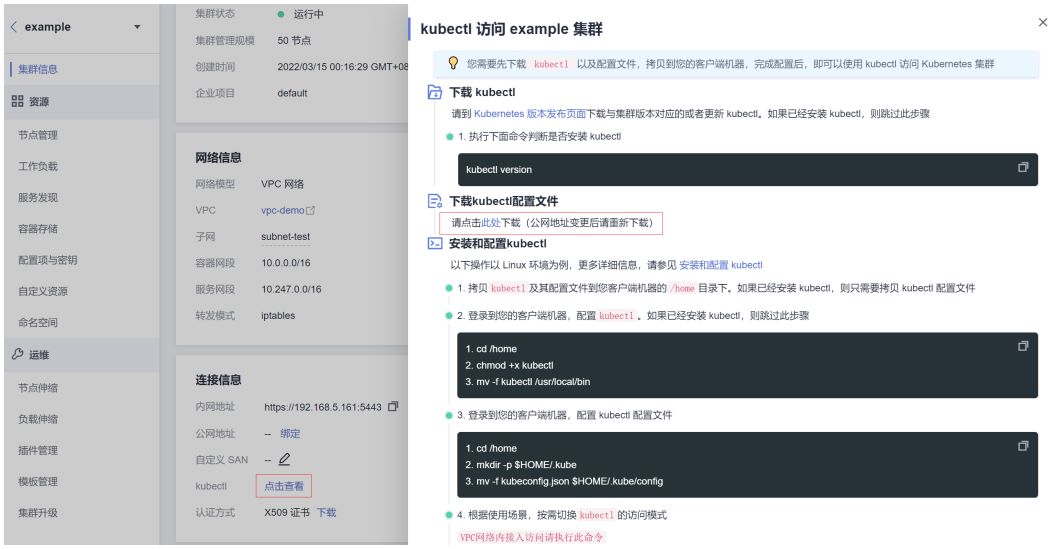


前提条件

- 您需要一台Linux虚拟机上安装kubectl命令行工具，kubectl的版本应该与集群版本相匹配，详情请参见**安装kubectl**。
- 安装kubectl的虚拟机需要可以访问每个集群的网络环境。

kubeconfig 文件结构解析

kubeconfig是kubectl的配置文件，您可以在集群详情页面下载。



kubeconfig文件内容如下所示。

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [{
        "name": "internalCluster",
        "cluster": {
            "server": "https://192.168.0.85:5443",
            "certificate-authority-data": "LS0tLS1CRUULIE..."
        }
    }, {
        "name": "externalCluster",
        "cluster": {
            "server": "https://xxx.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTiBDRVJ...",
            "client-key-data": "LS0tLS1CRUdJTiBS..."
        }
    }],
    "contexts": [{
        "name": "internal",
        "context": {
            "cluster": "internalCluster",
            "user": "user"
        }
    }, {
        "name": "external",
```

```
      "context": {
         "cluster": "externalCluster",
         "user": "user"
      }
   }],
   "current-context": "external"
}
```

其中主要分为3部分内容。

- clusters：描述集群的信息，主要是集群的访问地址。
- users：描述访问集群访问用户的信息，主要是client-certificate-data和client-key-data这两个证书文件内容。
- contexts：描述配置的上下文，用于使用时切换。上下文会关联user和cluster，也就是定义使用哪个user去访问哪个集群。

从上面的kubeconfig文件可以看出，此处将集群的内网地址和公网访问地址分别定义成一个集群，且定义了两个上下文，从而能够通过切换上下文选择使用不同的地址访问集群。

方案一：在命令中指定不同的 kubeconfig 配置文件

步骤1 登录安装kubectl的虚拟机。

步骤2 分别下载2个集群的kubeconfig文件到kubectl客户端机器的“/home”目录下，本文中使用以下名称作为示例。

集群名称	kubeconfig配置文件名称
集群A	kubeconfig-a.json
集群B	kubeconfig-b.json

步骤3 假设以集群A作为kubectl的默认连接集群，将kubeconfig-a.json文件移动至“$HOME/.kube/config”。

```
cd /home
mkdir -p $HOME/.kube
mv -f kubeconfig-a.json $HOME/.kube/config
```

步骤4 将集群B对应的kubeconfig-b.json文件移动至“$HOME/.kube/config-test”。

```
mv -f kubeconfig-b.json $HOME/.kube/config-test
```

其中config-test文件名称可以自定义。

步骤5 由于集群A为kubectl的默认连接集群，使用kubectl命令时无需添加“--kubeconfig”参数。而在使用kubectl连接集群B时，需要添加“--kubeconfig”参数用于指定kubectl命令所使用的凭证。例如查看集群B的节点命令如下：

```
kubectl --kubeconfig=$HOME/.kube/config-test get node
```

上述使用方式命令较长，频繁使用的情况下带来诸多不便，您可使用alias别名的方式简化命令，例如：

```
alias ka='kubectl --kubeconfig=$HOME/.kube/config'
alias kb='kubectl --kubeconfig=$HOME/.kube/config-test'
```

其中ka、kb可根据喜好自定义。使用kubectl命令时，可直接输入ka或kb来代替kubectl，并自动添加“--kubeconfig”参数。例如上述查看集群B的节点命令可简化为：


```
kb get node
```

----结束

方案二：将两个集群的 kubeconfig 文件合并

下面以配置2个集群为例演示如何修改kubeconfig文件访问多个集群。

为简洁文档篇幅，如下示例选取公网访问的方式，删除内网访问方式。如果您需要在内网访问多集群，只需要保留内网访问的集群clusters字段，保证能够从内网访问到集群即可，其方法与下面内容并无本质区别。

步骤1 分别下载2个集群的kubeconfig文件，删除内网访问内容，如下所示。

- 集群A：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "externalCluster",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "external",
        "context": {
            "cluster": "externalCluster",
            "user": "user"
        }
    }],
    "current-context": "external"
}
```

- 集群B：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "externalCluster",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0rTUideUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "external",
        "context": {
            "cluster": "externalCluster",
            "user": "user"
        }
```



```
    }],
    "current-context": "external"
}
```

此时这两个文件除了集群访问地址clusters.cluster.server和user字段的client-certificate-data和client-key-data字段内容不同，文件结构完全一致。

步骤2 修改两个配置文件中的名称，如下所示。

- 集群A：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "Cluster-A",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "Cluster-A-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "Cluster-A-Context",
        "context": {
            "cluster": "Cluster-A",
            "user": "Cluster-A-user"
        }
    }],
    "current-context": "Cluster-A-Context"
}
```

- 集群B：

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "Cluster-B",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "Cluster-B-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0rTUideUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "Cluster-B-Context",
        "context": {
            "cluster": "Cluster-B",
            "user": "Cluster-B-user"
        }
    }],
    "current-context": "Cluster-B-Context"
}
```

步骤3 将两个文件的内容合并。

文件结构不变，将clusters、users和contexts的内容合并即可，如下所示。


```
{
   "kind": "Config",
   "apiVersion": "v1",
   "preferences": {},
   "clusters": [ {
      "name": "Cluster-A",
      "cluster": {
         "server": "https://119.xxx.xxx.xxx:5443",
         "insecure-skip-tls-verify": true
      }
   },
    {
      "name": "Cluster-B",
      "cluster": {
         "server": "https://124.xxx.xxx.xxx:5443",
         "insecure-skip-tls-verify": true
      }
   }],
   "users": [{
      "name": "Cluster-A-user",
      "user": {
         "client-certificate-data": "LS0tLS1CRUdJTxM...",
         "client-key-data": "LS0tLS1CRUdJTiB...."
      }
   },
   {
      "name": "Cluster-B-user",
      "user": {
         "client-certificate-data": "LS0tLS1CRUdJTxM...",
         "client-key-data": "LS0rTUideUdJTiB...."
      }
   }],
   "contexts": [{
      "name": "Cluster-A-Context",
      "context": {
         "cluster": "Cluster-A",
         "user": "Cluster-A-user"
      }
   },
   {
      "name": "Cluster-B-Context",
      "context": {
         "cluster": "Cluster-B",
         "user": "Cluster-B-user"
      }
   }],
   "current-context": "Cluster-A-Context"
}
```

步骤4 将两个文件内容合并到一个kubeconfig文件后，执行如下命令将文件复制到kubectl配置路径下。

mkdir -p $HOME/.kube

mv -f kubeconfig.json $HOME/.kube/config

步骤5 执行kubectl命令验证是否能够连接两个集群。

```
# kubectl config use-context Cluster-A-Context
Switched to context "Cluster-A-Context".
# kubectl cluster-info
Kubernetes control plane is running at https://119.xxx.xxx.xxx:5443
CoreDNS is running at https://119.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

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

# kubectl config use-context Cluster-B-Context
Switched to context "Cluster-B-Context".
# kubectl cluster-info
```



```
Kubernetes control plane is running at https://124.xxx.xxx.xxx:5443
CoreDNS is running at https://124.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

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

上述使用方式命令较长，频繁切换的情况下带来诸多不便，您也可以使用alias别名的方式简化命令，如下：

```
alias ka='kubectl config use-context Cluster-A-Context;kubectl'
alias kb='kubectl config use-context Cluster-B-Context;kubectl'
```

其中ka、kb可根据喜好自定义。使用kubectl命令时，可直接输入ka或kb来代替kubectl，在使用时会先切换context，再执行kubectl命令。例如：

```
# ka cluster-info
Switched to context "Cluster-A-Context".
Kubernetes control plane is running at https://119.xxx.xxx.xxx:5443
CoreDNS is running at https://119.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

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

----结束

9.7 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，详情请参见**数据盘空间分配说明**。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

您可以执行以下步骤清理未使用的镜像：

- 使用containerd容器引擎的节点：
  a. 查看节点上的本地镜像。
  ```
  crictl images -v
  ```
  b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。
  ```
  crictl rmi {镜像ID}
  ```
- 使用docker容器引擎的节点：
  a. 查看节点上的本地镜像。
  ```
  docker images
  ```

b. 确认镜像无需使用，并通过镜像ID删除无需使用的镜像。

```
docker rmi {镜像ID}
```

说明

请勿删除cce-pause等系统镜像，否则可能导致无法正常创建容器。

方案二：扩容磁盘

步骤1 在EVS界面扩容数据盘。

步骤2 登录CCE控制台，进入集群，在左侧选择“节点管理”，单击节点后的“同步云服务器”。

步骤3 登录目标节点。

步骤4 使用**lsblk**命令查看节点块设备信息。

这里存在两种情况，根据容器存储Rootfs而不同。

- Overlayfs，没有单独划分thinpool，在dockersys空间下统一存储镜像相关数据。

```
# lsblk
NAME              MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda               8:0    0   50G  0 disk
└─vda1            8:1    0   50G  0 part /
vdb               8:16   0  200G  0 disk
├─vgpaas-dockersys 253:0    0   90G  0 lvm  /var/lib/docker              # 容器引擎使用的空间
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
```

  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

```
pvresize /dev/vdb
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
vda                       8:0    0   50G  0 disk
└─vda1                    8:1    0   50G  0 part /
vdb                       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/vdb
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/vdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----结束

9.8 集群视角的成本可视化最佳实践

应用现状

当前使用CCE时，默认是以CCE整个云服务的粒度体现计费信息，没有划分不同集群使用的成本。

解决方案

通过给集群使用的资源打上**CCE-Cluster-ID**标签，在成本中心通过标签过滤汇聚整个集群所使用资源的成本，以集群为单位进行成本分析，降本增效。

约束与限制

应用在资源上的标签，一般在创建并产生费用24小时后才会在“成本标签”页面展示。

成本标签激活后，才能在分析成本数据时通过标签进行过滤或汇总。成本标签不会应用于激活之前产生的成本中。

操作步骤

步骤1 激活**CCE-Cluster-ID**成本标签。

登录**成本中心**，查找**CCE-Cluster-ID**，在操作一列单击“激活”，如下图所示。

图 9-12 激活成本标签

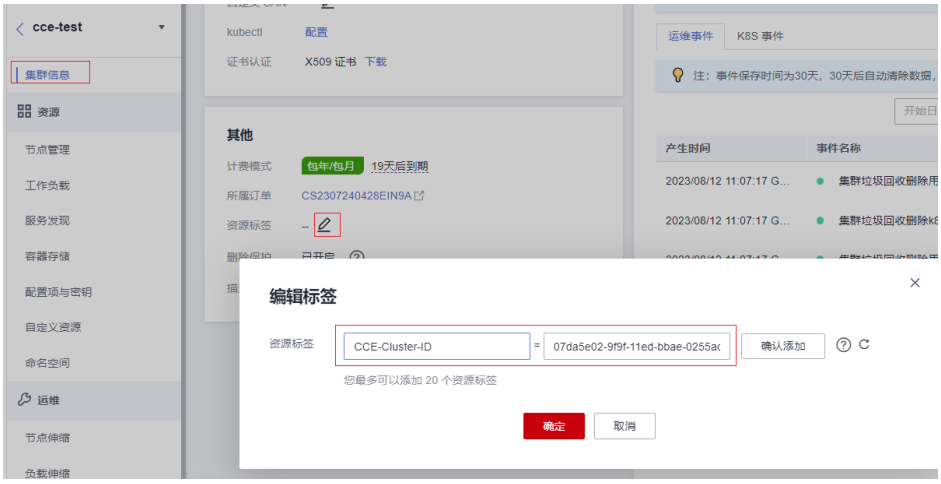
激活成功后显示如下。

图 9-13 激活成功

步骤2 给集群所使用资源打标签。

集群所使用资源包括集群Master节点、节点，存储资源（EVS、SFS、OBS等），网络资源（ELB、EIP等）。其中节点会默认添加**CCE-Cluster-ID**标签。

- 给集群添加标签。
  在CCE控制台进入集群信息页面，在资源标签处给集群打标签。



- EVS
  在EVS控制台，单击具体云硬盘，进入详情页，在“标签”Tab页添加标签。

- OBS
  在OBS控制台，单击具体OBS桶，进入详情页，在“基础配置 > 标签”页添加标签。

- SFS-turbo
  在SFS控制台，单击具体SFS-turbo实例，进入详情页，在“标签”Tab页添加标签。

- ELB
  在ELB控制台，单击具体ELB实例，进入详情页，在“标签”Tab页添加标签。

- EIP
  在EIP控制台，单击具体IP地址，进入详情页，在“标签”Tab页添加标签。

步骤3 分析集群成本。

在成本分析页面过滤器中选择**CCE-Cluster-ID**，如下图所示，要分析哪个集群的成本就选择哪个集群的ID。

图 9-14 成本分析

单击“确定”，即可查看集群的各项成本。

图 9-15 集群成本

产品类型	应计总计	预测总计 **	2023/08
总成本 (¥)	53.08	--	53.08
弹性云服务器 ECS (¥)	30.44	--	30.44
弹性文件服务 SFS (¥)	17.50	--	17.50
云硬盘 EVS (¥)	5.14	--	5.14
虚拟私有云 VPC (¥)	0.00	--	0.00
对象存储服务 OBS (¥)	0.00	--	0.00
弹性负载均衡 ELB (¥)	0.00	--	0.00

----结束

9.9 使用共享 VPC 创建 CCE Turbo 集群

共享 VPC 简介

共享VPC是通过资源访问管理服务（RAM）将本账号的VPC资源共享给其他账号使用。例如，租户A可以将自己账号下创建的VPC和子网共享给租户B。在租户B接受共享以后，租户B账号下可以查看到该共享子网及其所属的共享VPC，并可以使用该共享子网和共享VPC创建资源，如CCE Turbo集群。详情请参见**共享VPC概述**。

使用场景

企业按企业的组织结构或业务形态，将账号有序组织集中管理。统一资源管理并与其他成员共享，节省资源重复配置。统一安全运维管理，便于企业集中配置安全策略，利于审计跟踪。

例如，资源所有者为企业IT账号，创建VPC及子网，并将多个子网分别共享给其他账号：

- 账号A为企业业务账号，使用子网1创建资源。
- 账号B为企业业务账号，使用子网2创建资源。

约束与限制

- 当前仅CCE Turbo集群支持共享VPC特性。
- 使用共享VPC创建的集群不支持使用共享ELB及NAT网关功能。
- 使用共享VPC创建的集群暂不支持文件存储、对象存储和极速文件存储。
- 如果当前共享VPC下已创建CCE Turbo集群，则共享VPC的所有者不应删除该共享，否则将会导致CCE Turbo集群功能异常。

操作步骤

账号A将VPC共享给账号B后，账号B即可在创建CCE Turbo集群时选择共享VPC及其共享子网。

步骤1 （账号A操作）使用资源访问管理服务（RAM）创建共享VPC，并指定资源使用者为账号B，详情请参见**创建共享**。

共享创建完成后，RAM会向指定的使用者发送共享邀请，账号B需接受共享邀请后，才可以访问和使用被共享的资源。

步骤2 （账号B操作）登录CCE控制台，创建一个CCE Turbo集群。

在网络配置中，请选择由账号A共享的VPC。其余配置可参考**购买CCE集群**。

图 9-16 选择共享 VPC

----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

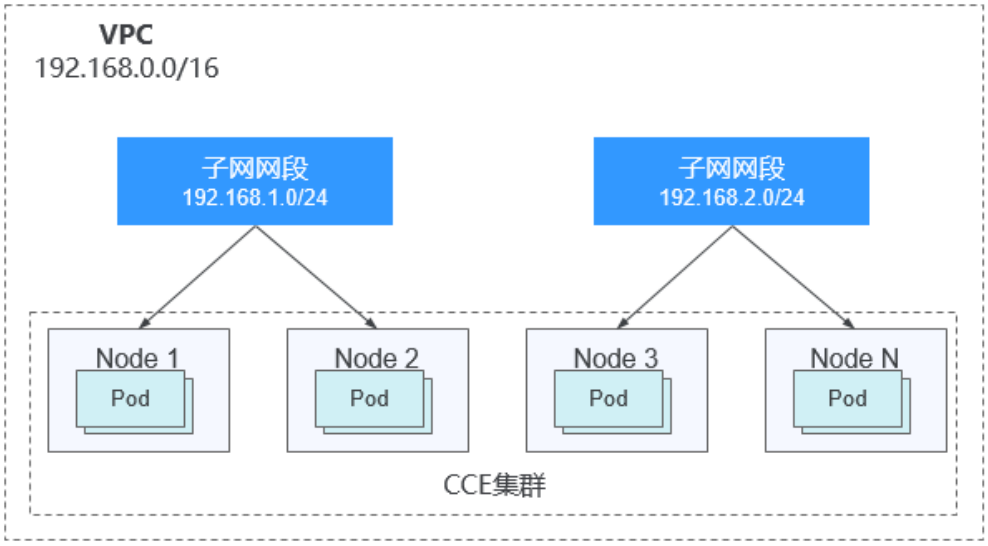
- **VPC网段**

  虚拟私有云（Virtual Private Cloud，简称VPC）可以为云服务器、云容器、云数据库等资源构建隔离的、用户自主配置和管理的虚拟网络环境。您可以自由配置VPC内的IP地址段、子网、安全组等子服务，也可以申请弹性带宽和弹性公网IP搭建业务系统。

- **子网网段**

  子网是用来管理弹性云服务器网络平面的一个网络，可以提供IP地址管理、DNS服务，子网内的弹性云服务器IP地址都属于该子网。

图 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**

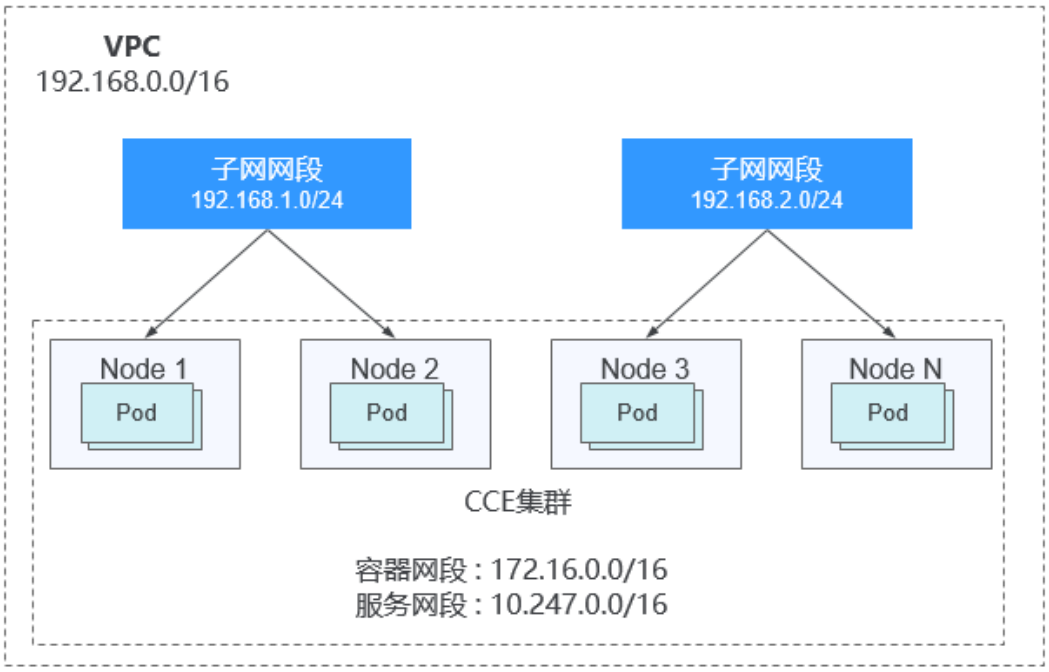
  Service也是Kubernetes内的概念，每个Service都有自己的地址，在CCE上创建集群时，可以指定Service的地址段（即服务网段）。同样，服务网段也不能和子网网段重合，而且服务网段也不能和容器网段重叠。服务网段只在集群内使用，不能在集群外使用。

单 VPC 下单集群场景

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。

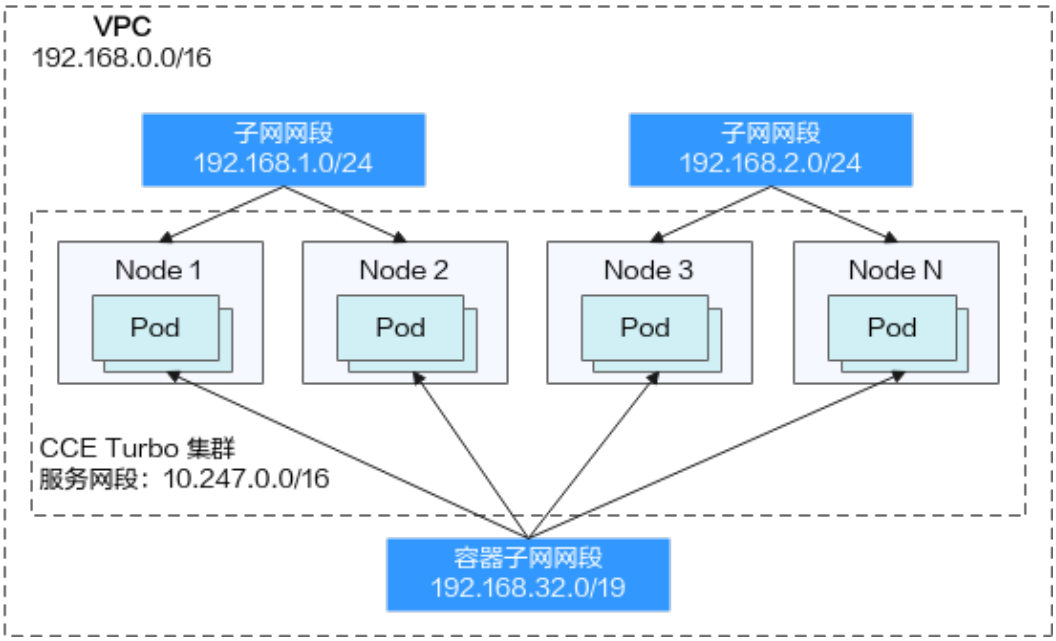
图 10-2 单 VPC 单集群场景网段规划-CCE 集群



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**图10-3**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



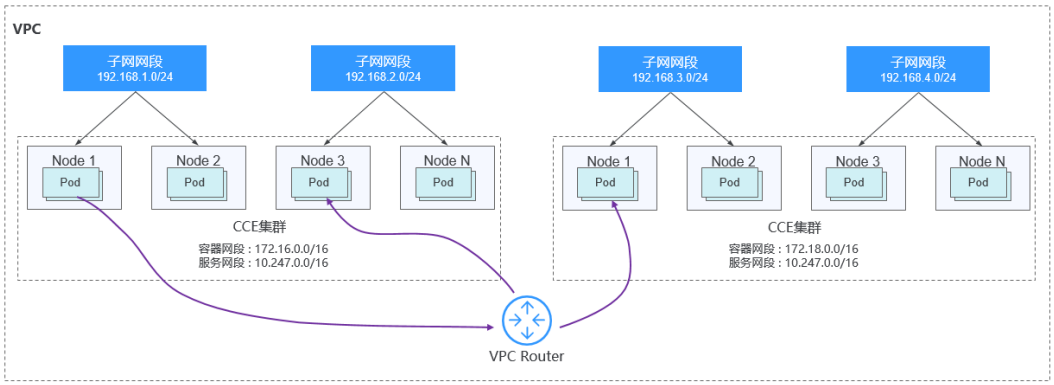
单 VPC 下多集群场景

VPC网络模式

Pod的报文需要通过VPC路由转发，CCE会自动在VPC路由上配置到每个容器网段的路由表，集群组网规模受限于VPC路由表能力。集群网络地址段规划示意图如图10-4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，如果节点安全组在入方向上放通对端集群的容器网段，一个集群的Pod可以通过容器IP直接访问另外一个集群的Pod。
- 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

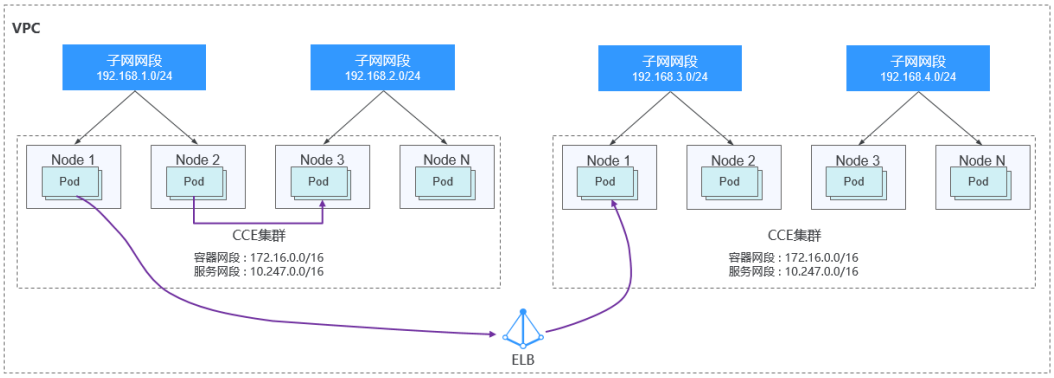


容器隧道网络

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图10-5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不能通过容器IP直接访问，跨集群容器之间的访问需要通过Service，建议使用负载均衡类型的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 10-5 容器隧道网络-多集群场景示例

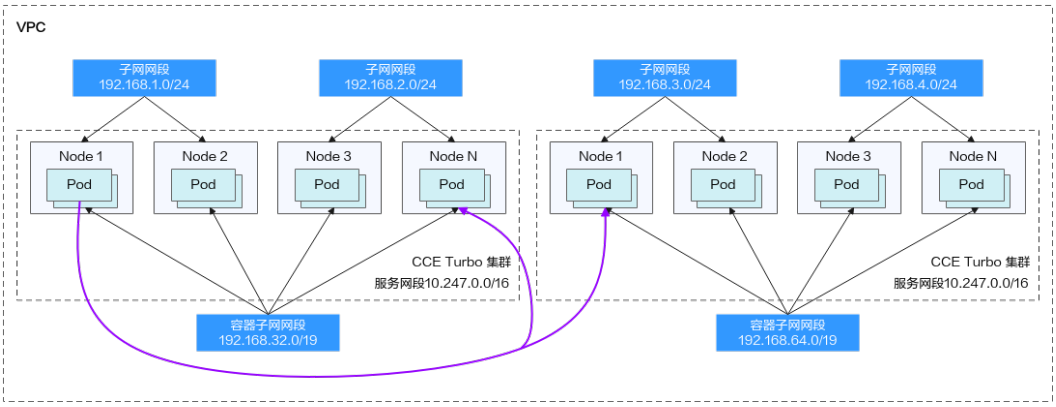


云原生网络2.0模式（即CCE Turbo集群）

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，如果集群ENI安全组在入方向上放通对端集群的容器子网网段，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。

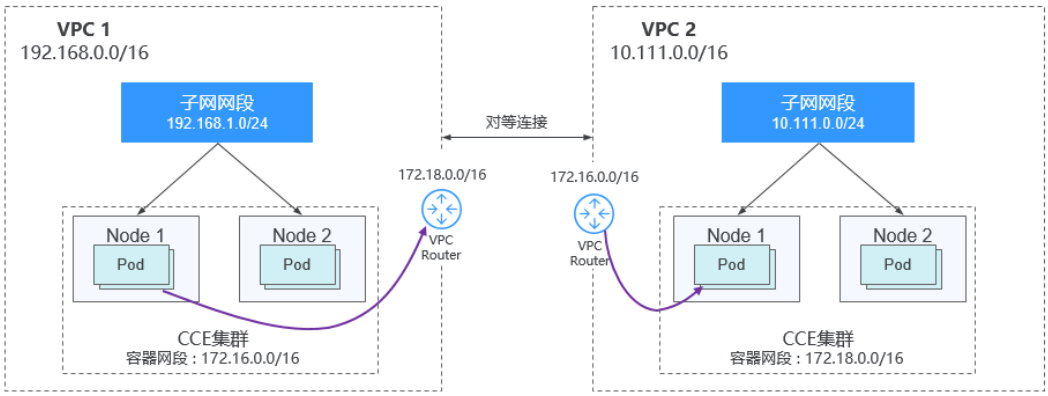
集群跨 VPC 互联场景

不同VPC之间网络不通，对等连接用于连通同一个区域内的VPC，实现不同VPC的网络互联。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。详情请参见**对等连接简介**。

VPC网络模式集群

VPC网络模式的集群跨VPC互联时，在创建对等连接后，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：

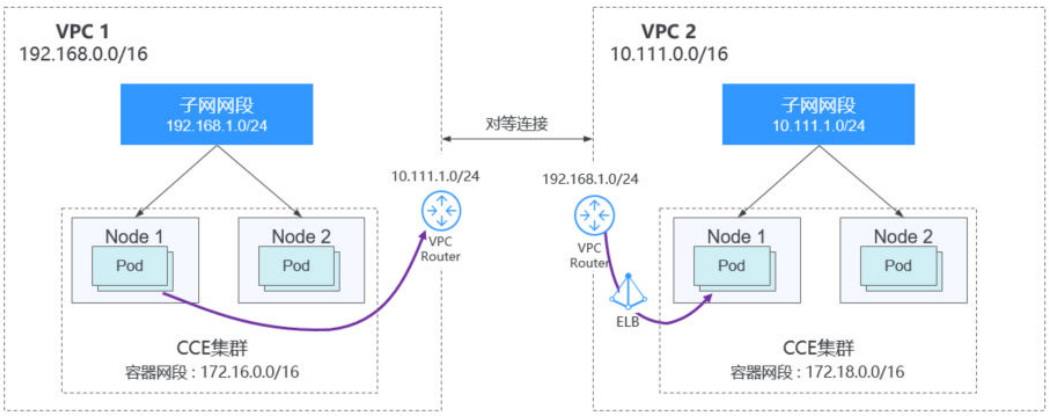
- 两端集群所属的VPC地址段需要避免重叠，且在每个集群中，子网网段不能与容器网段重叠。
- 两端集群的容器网段不能相互重叠，但服务网段可以重叠。
- 当请求端集群为VPC网络模型时，需要**关注目的端集群的节点安全组在入方向上是否放通了请求端集群的容器网段**。此时，一个集群的Pod可以通过容器IP直接访问另外一个集群的Pod。同理，如果两端集群的节点需要相互访问，节点安全组需要放通对端集群的VPC网段。
- 两端的VPC路由表中均需要添加访问对端网段的路由。例如，VPC 1的路由表需添加访问VPC 2网段的路由，同时，VPC 2的路由表也需要添加访问VPC 1的路由。
  - **添加对端集群VPC网段**：添加VPC网段的路由后，Pod可以访问另外一个集群节点，例如访问NodePort类型的Service端口。
  - **添加对端容器网段**：添加容器网段路由后，Pod可以通过容器IP直接访问另外一个集群的Pod。

图 10-8 在本端路由中添加对端容器网段的地址

容器隧道网络模式集群

两个容器隧道网络模式的集群跨VPC互联时，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。

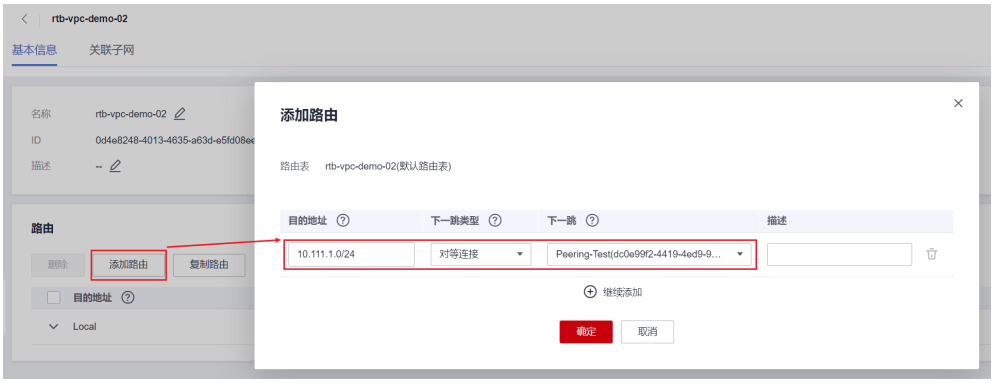
图 10-9 容器隧道网络-VPC 互联场景



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。
- 当请求端集群为容器隧道网络模型时，需要**关注目的端集群的节点安全组在入方向上是否放通了请求端集群的VPC网段（包含节点子网）**。在此情况下，一个集群的节点可以访问另一个集群的节点。但不同集群的Pod不能通过容器IP直接访问，跨集群容器之间的访问需要通过Service，建议使用负载均衡类型的Service。
- 两端的VPC路由表中均需要**添加对端集群VPC网段**路由。例如，VPC 1的路由表需添加访问VPC 2网段的路由，同时，VPC 2的路由表也需要添加访问VPC 1的路由。添加VPC网段的路由后，Pod可以访问另外一个集群节点，例如访问NodePort类型的Service端口。

图 10-10 在本端路由中添加对端集群节点子网网段的地址



云原生网络2.0集群（即CCE Turbo集群）

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，使两个VPC互通，即可完成集群间的互通。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。
- 当请求端集群为云原生网络2.0模型时，需要**关注目的端集群的ENI安全组（名为{集群名}-cce-eni-{随机ID}）在入方向上是否放通了请求端集群的VPC网段（包含节点子网和容器子网）**。此时，一个集群的Pod可以通过容器IP直接访问另外一个集群的Pod。同理，如果两端集群的节点需要相互访问，则需要在集群的**节点安全组（名为{集群名}-cce-node-{随机ID}）**放通对端集群的VPC网段。
- 两端的VPC路由表中均需要**添加对端集群VPC网段**路由。例如，VPC 1的路由表需添加访问VPC 2网段的路由，同时，VPC 2的路由表也需要添加访问VPC 1的路由。添加VPC网段的路由后，Pod可以访问另外一个集群的Pod IP或节点。

多网络模式集群并存场景

在不同网络模式集群间需要跨VPC互访的情况下，每种类型的集群均可能作为请求端和目的端。一般情况下需遵循以下规律：

- 集群所属的VPC地址段需要避免和对端集群的VPC地址段重叠。
- 集群子网网段尽量避免和自身的容器网段重叠。
- 集群间的容器网段需要避免相互重叠。
- 如集群间容器或节点存在相互访问，则两侧集群的安全组在入方向上均需按以下规则放通对应网段：
  - 请求端集群为VPC网络模型时，目的端集群的节点安全组需放通**请求端集群的VPC网段（包含节点子网）和容器网段**。

  - 请求端集群为容器隧道网络模型时，目的端集群的节点安全组需放通**请求端集群的VPC网段（包含节点子网）**。
  - 请求端集群为云原生网络2.0模型时，目的端集群的ENI安全组和节点安全组需放通**请求端集群的VPC网段（包含节点子网和容器子网）**。
- 两端的VPC路由表中均需要**添加对端集群VPC网段**路由。例如，VPC 1的路由表需添加访问VPC 2网段的路由，同时，VPC 2的路由表也需要添加访问VPC 1的路由。添加VPC网段的路由后，Pod可以访问另外一个集群节点，例如访问NodePort类型的Service端口。

  若某个集群为VPC网络模型，两端的VPC路由表中还需要**添加容器网段**的地址。添加容器网段路由后，Pod可以通过容器IP直接访问另外一个集群的Pod。

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

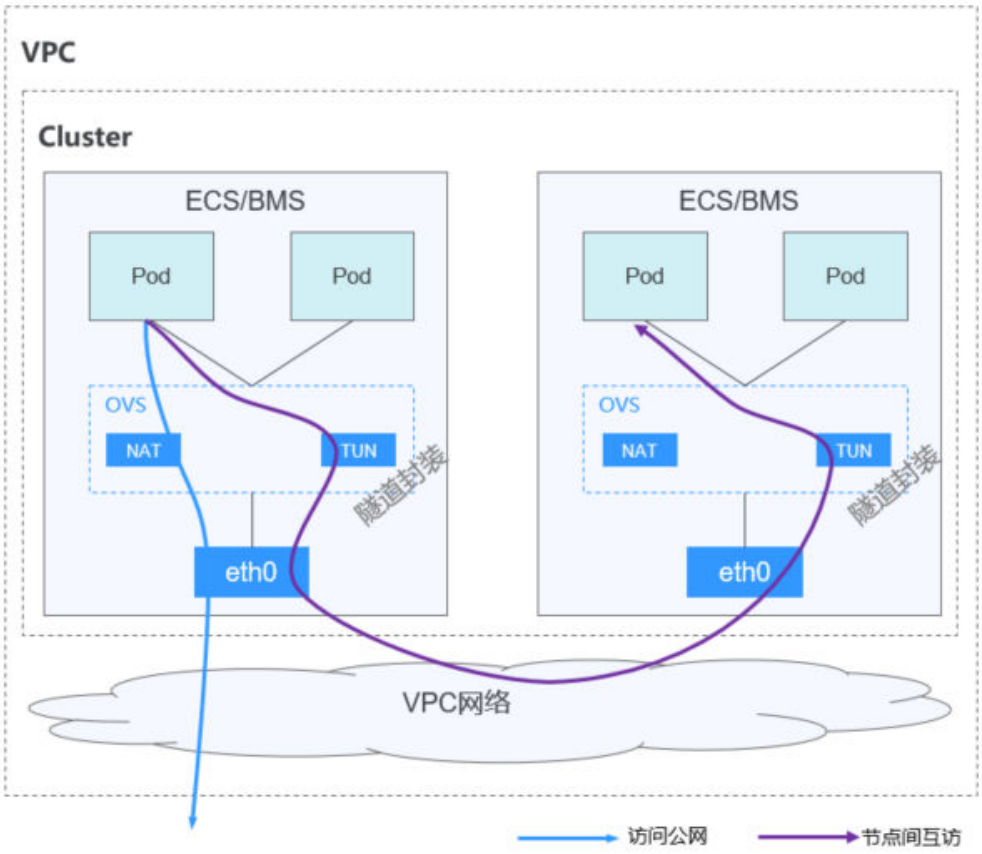
自研高性能商业版容器网络插件，支持容器隧道网络、VPC网络、云原生网络2.0网络模型：

⚠ 注意

集群创建成功后，网络模型不可更改，请谨慎选择。

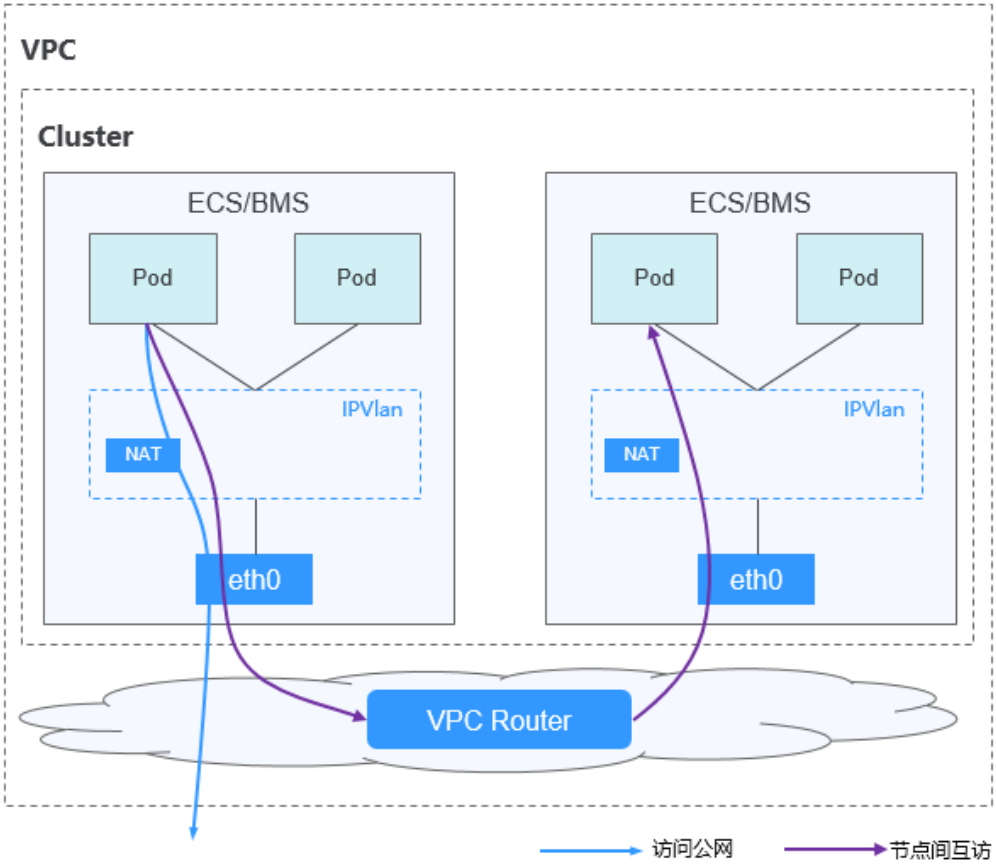
- **容器隧道网络（Overlay）：** 基于底层VPC网络构建了独立的VXLAN隧道化容器网络，适用于一般场景。VXLAN是将以太网报文封装成UDP报文进行隧道传输。容器网络是承载于VPC网络之上的Overlay网络平面，具有付出少量隧道封装性能损耗，即可获得通用性强、互通性强、高级特性支持全面（例如Network Policy网络隔离）的优势，可以满足大多数应用需求。

图 10-11 容器隧道网络



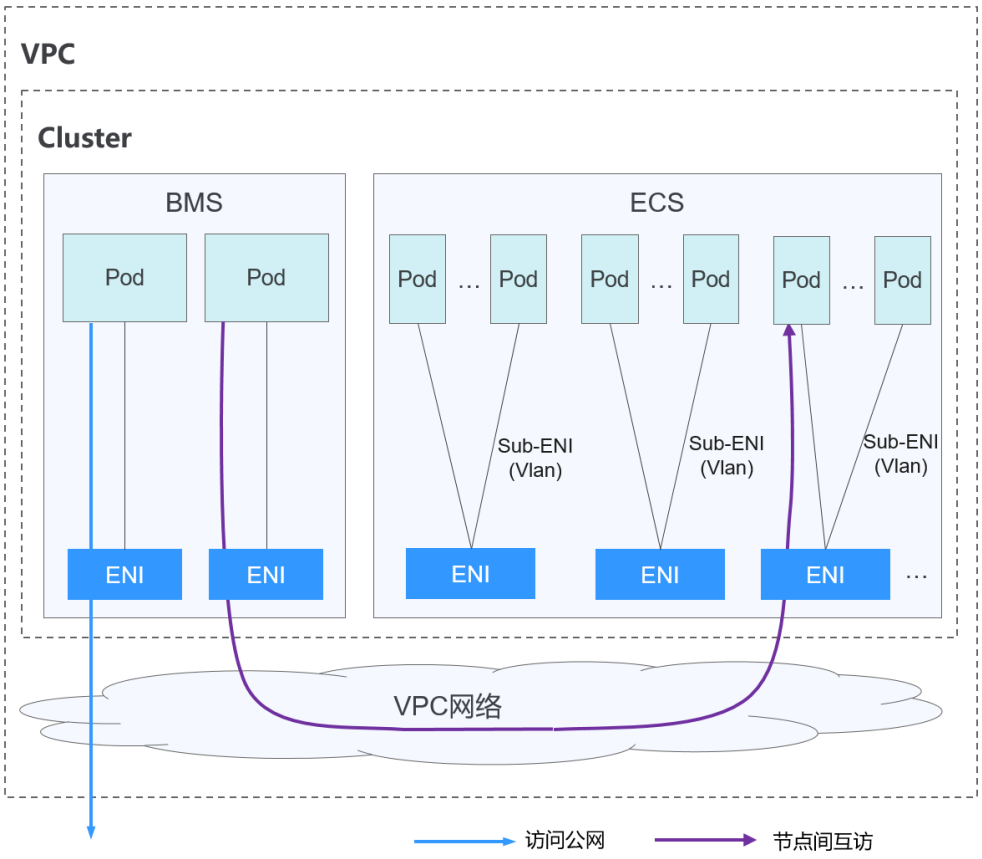
- **VPC网络：**采用VPC路由方式与底层网络深度整合，适用于高性能场景，节点数量受限于虚拟私有云VPC的路由配额。每个节点将会被分配固定大小的IP地址段。VPC网络由于没有隧道封装的消耗，容器网络性能相对于容器隧道网络有一定优势。VPC网络集群由于VPC路由中配置有容器网段与节点IP的路由，可以支持集群外直接访问容器实例等特殊场景。

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
适用场景	• 一般容器业务场景。 • 对网络时延、带宽要求不是特别高的场景。	• 对网络时延、带宽要求高。 • 容器与虚机IP互通，使用了微服务注册框架，如Dubbo、CSE等。	• 对网络时延、带宽要求高，高性能场景。 • 容器与虚机IP互通，使用了微服务注册框架的，如Dubbo、CSE等。
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 Standard集群	CCE Standard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

对比维度	容器隧道网络	VPC网络	云原生网络2.0
IP地址管理	• 容器网段单独分配 • 节点维度划分地址段，动态分配（地址段分配后可动态增加）	• 容器网段单独分配 • 节点维度划分地址段，静态分配（节点创建完成后，地址段分配即固定，不可更改）	容器网段从VPC子网划分，无需单独分配
网络性能	基于vxlan隧道封装，有一定性能损耗。	无隧道封装，跨节点通过VPC 路由器转发，性能好，媲美主机网络。	容器网络与VPC网络融合，性能无损耗
组网规模	最大可支持2000节点	受限于VPC路由表能力，适合中小规模组网，建议规模为1000节点及以下。 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

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 使用 VPC 和云专线实现容器与 IDC 之间的网络通信

使用场景

借助VPC和云专线，在VPC网络模型的集群中实现集群容器网段（172.56.0.0/16）与IDC网段（10.1.123.0/24）的相互通信。

图 10-14 网络示意

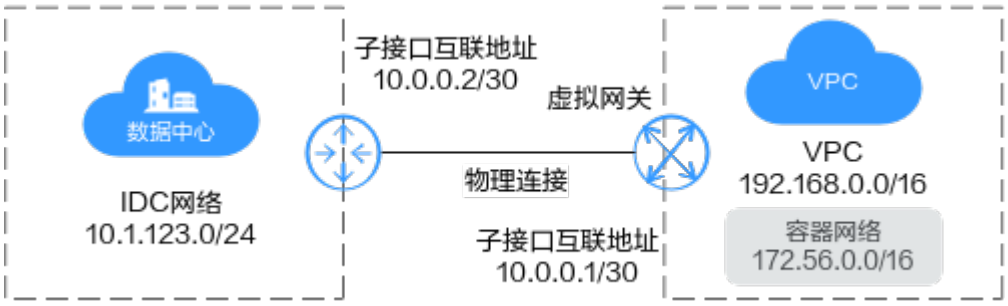


表 10-2 地址信息

网络	网段
用户侧IDC网络	10.1.123.0/24
专线互联地址	华为云侧：10.0.0.1/30 用户侧：10.0.0.2/30
VPC地址段	192.168.0.0/16
容器网段	172.56.0.0/16

前提条件

需要具备IDC并提前申请云专线服务。

操作步骤

步骤1 创建物理连接

1. 登录管理控制台，在管理控制台左上角单击 ，选择区域和项目。在控制台首页，单击左上角的 ，在展开的列表中单击“网络 > 云专线 DC”。
2. 在“网络控制台”的左侧栏目树中，单击“云专线 > 物理连接”，单击右侧的“创建物理连接”。
3. 在创建物理连接页面，单击“自建专线接入”页签。

   根据界面提示，在物理连接购买页面配置机房地址、华为云接入点、物理连接端口等信息，可参照表10-3输入相关参数。

表 10-3 表 2 购买物理连接参数

参数	说明
计费模式	专线服务付费方式，目前仅支持包年/包月方式付费。
区域	物理连接开通的区域。用户可以在管理控制台左上角或购买页面切换区域。

参数	说明
物理连接名称	用户将要创建的物理连接的名称（可自定义）。
华为云接入点	物理连接接入点的位置。
运营商	提供物理连接的运营商。
端口类型	物理连接接入端口的类型：1GE，10GE、40GE、100GE。
专线带宽	物理连接的带宽大小，请在下拉框中选择对应的带宽。仅作为运营商接入带宽描述。
您的机房地址	用户填写机房地址，可精确到楼层。例如上海市浦东新区华京路xx号xx楼xx机房。
标签	云专线服务的标识，包括键和值。可以为云专线服务创建10个标签。 标签的命名规则请参考**表10-4**。 **说明** 如果已经通过TMS的预定义标签功能预先创建了标签，则可以直接选择对应的标签键和值。 预定义标签的详细内容，请参见**预定义标签简介**。
描述	用户可以对物理连接添加备注信息。
联系人姓名/手机/Email	用户可以在此提供用户侧专线负责人信息。 注意：如不提供负责人信息，将只能通过账号信息查询，会增加需求确认时长。
购买时长	购买专线服务的时长。
自动续费	自动续费时长与购买时长相同。 例如：用户购买时长为三个月，当勾选该项后，将自动续费三个月，以此类推。
企业项目	企业项目是一种云资源管理方式，企业项目管理服务提供统一的云资源按项目管理，以及项目内的资源管理、成员管理。

表 10-4 物理连接标签命名规则

参数	规则
键	- 不能为空。 - 对于同一资源键值唯一。 - 长度不超过36个字符。 - 取值只能包含大写字母、小写字母、数字、中划线、下划线、以及从\u4e00到\u9fff的Unicode字符。

参数	规则
值	– 可以为空。 – 长度不超过43个字符。 – 取值只能包含大写字母、小写字母、数字、点、中划线、下划线、以及从\u4e00到\u9fff的Unicode字符。

4. 单击“立即购买”。
5. 确认订单信息，单击“去支付”。
6. 单击“确认付款”。

步骤2 创建虚拟网关

1. 单击“云专线 > 虚拟网关”，单击右侧的“创建虚拟网关”。在虚拟网关处添加VPC网段和容器网络网段。

图 10-15 创建虚拟网关

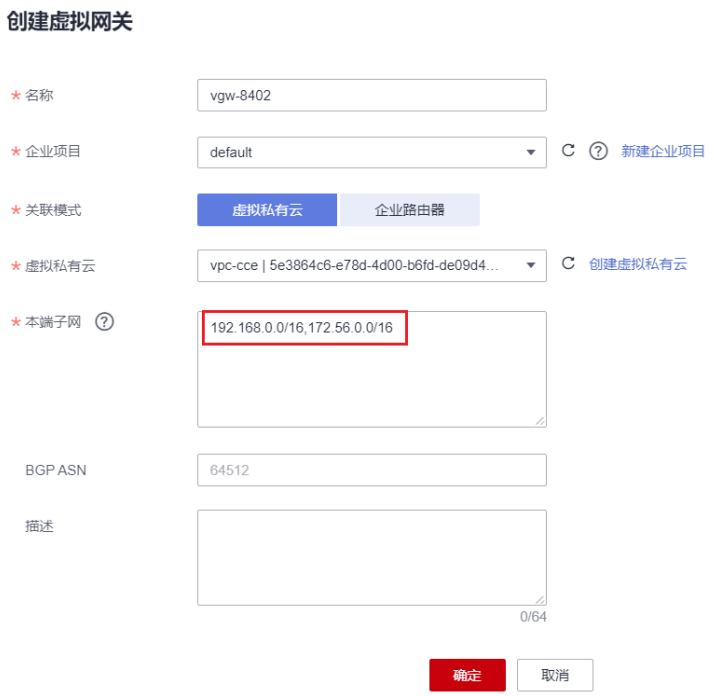


表 10-5 表 3 虚拟网关参数

参数	说明
名称	虚拟网关名称。 字符长度为1~64。
企业项目	企业项目是一种云资源管理方式，企业项目管理服务提供统一的云资源按项目管理，以及项目内的资源管理、成员管理。

参数	说明
关联模式	选择“虚拟私有云”。
虚拟私有云	在下拉框中选择虚拟网关所关联的虚拟私有云。
本端子网	需要与本地网络实现互通的VPC网段。 本例中为VPC网络模型的集群，需要填写VPC网段（192.168.0.0/16）和容器网络的网段（172.56.0.0/16）。容器隧道网络模型和云原生2.0网络模型的集群仅需填写VPC网段即可。
描述	虚拟网关描述。 字符长度中文为0~64，英文为0~128。

2. 单击“确定”。
   当所创建的虚拟网关状态列为“正常”时，完成虚拟网关的创建。

步骤3 创建虚拟接口

1. 单击“云专线 > 虚拟接口”，单击右侧的“创建虚拟接口”。
2. 根据界面提示输入相关参数，详细请参考**表10-6**。

图 10-16 创建虚拟接口

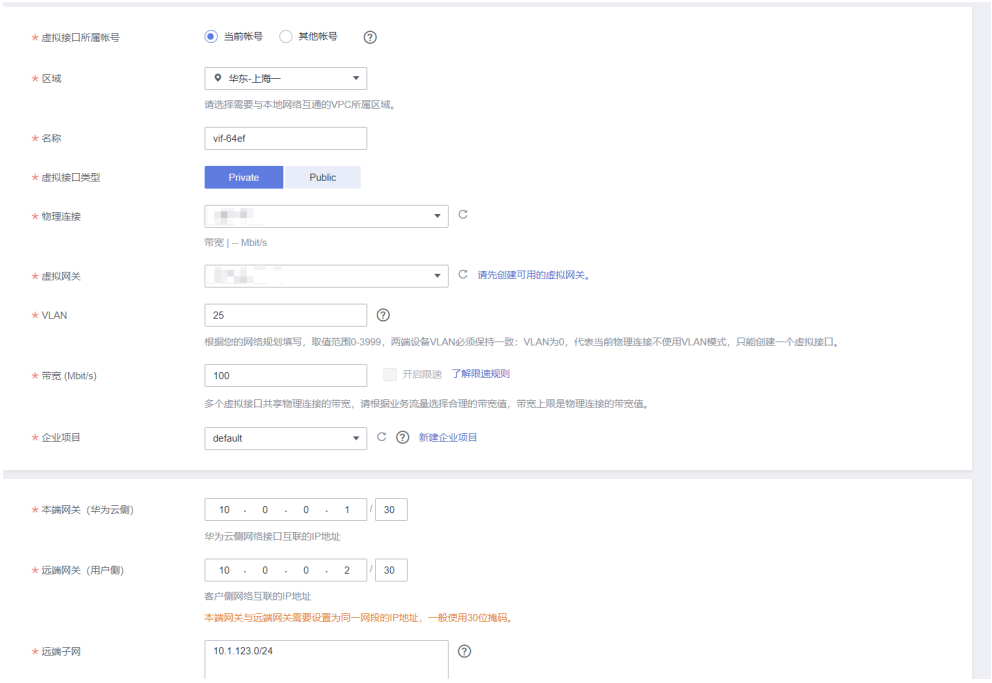


表 10-6 虚拟接口参数

参数	说明
区域	物理连接开通的区域。用户可以在管理控制台左上角或购买页面切换区域。

参数	说明
名称	虚拟接口名称。 字符长度为1~64。
物理连接	选择可用的物理连接。
虚拟网关	虚拟接口关联的虚拟网关。
VLAN	虚拟接口的VLAN。 标准专线的虚拟接口的VLAN由用户配置 。 托管专线的虚拟接口的VLAN会使用运营商或合作伙伴为托管专线分配的VLAN，用户无需配置。
带宽	虚拟接口带宽，单位为Mbit/s。虚拟接口带宽不可以超过物理连接带宽。
企业项目	企业项目是一种云资源管理方式，企业项目管理服务提供统一的云资源按项目管理，以及项目内的资源管理、成员管理。
本端网关（华为云侧）	云专线华为云侧接口互联IP地址。 本例中为10.0.0.1/30。
远端网关（用户侧）	用户本地数据中心侧网络的互联IP地址。 远端网关与本端网关需要设置为同一网段的IP地址，一般使用30位掩码。 本例中为10.0.0.2/30。
远端子网	用户数据中心的子网和子网掩码。多个远端子网时，请以逗号隔开。 本例中为10.1.123.0/24。
路由模式	路由模式：静态路由/BGP 双线或者后期有冗余专线接入请选择BGP模式。
BGP邻居AS号	BGP邻居自治系统的标识。 当路由模式为BGP时，需要设置此参数。
BGP MD5认证密码	BGP邻居的MD5值即BGP密码。 当路由模式为BGP时，可设置此参数，两侧网关参数需保持一致。 字符长度为8~255，至少包含以下字符的两种： – 大写字母 – 小写字母 – 数字 – 特殊字符（ ~!,.:;-_"(){}[]/@#$%^&*+|= ）
描述	可自定义虚拟接口的相关描述。

3. 单击“提交”，当所创建的虚拟接口状态列为“正常”时，完成虚拟接口的创建。
4. 虚拟接口创建完成，即可打通客户IDC与云上VPC之间的网络。

   客户可通过VPC内的主机设备向云下数据中心网络主机IP地址进行ping操作，以确认网络连通。

   说明

   创建虚拟接口后您还需要配置用户侧设备，云上放通安全组规则，允许云上云下访问。

步骤4 连通性测试

1. 使用traceroute命令测试IDC机器和云上容器间是否可以互通：
   a. 如果路由正常表明专线有回程路由。
   b. 如果出现IDC机器路由没有到达专线的云上网关等情况，请排查专线两端的路由设置是否正常。
2. 如果traceroute不通请尝试ping、telnet等方式，使用ping工具前如果ping的对象是云服务器，需确保安全组已放开ICMP策略。

----结束

10.4 自建 IDC 与 CCE 集群共享域名解析

10.4.1 方案概述

应用现状

当前，越来越多的软件采用微服务架构，构建一个产品时会大量使用微服务，不同微服务之间访问时涉及到域名访问。

拥有自建IDC的企业，在使用CCE时通常需要在CCE集群与自建IDC之间通信，而且当IDC有内部域名时，需要CCE集群内的节点和容器既能够解析IDC的域名，也能够解析云域名。

例如，某企业APP微服务改造后，其管理后台部署在CCE集群上，内容审核服务部署在企业原有的IDC，该企业同时购买了华为云图像识别服务。CCE所在的VPC和原有的IDC之间通过专线进行连接。部署架构如**图10-17**所示。

当用户访问该企业的APP时，不同微服务之间涉及到如下交互：

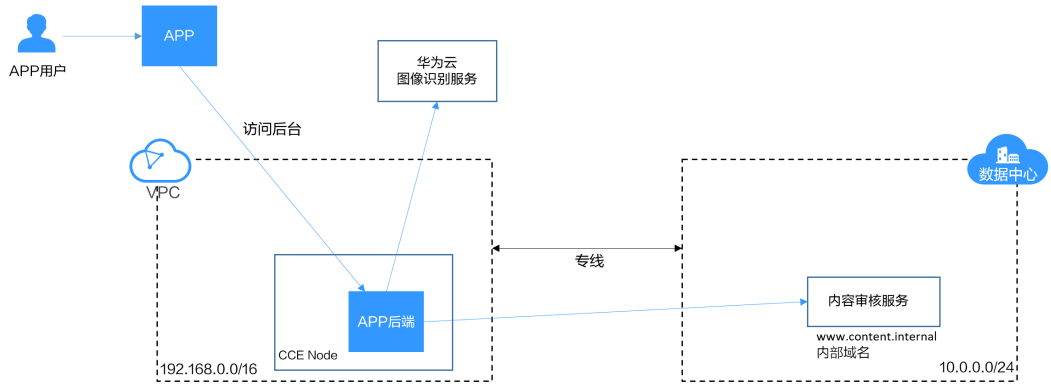
- CCE集群访问华为云图像识别服务时，默认使用华为云域名解析服务器。
- CCE集群访问IDC上部署的内容审核服务时，需要使用IDC内部域名服务器。

这就需要在CCE集群上既能使用华为云域名解析服务器，也能够使用IDC内部域名服务器。如果将CCE节点上域名解析服务器指向IDC的域名解析服务器，那会导致无法解析华为云的域名；如果修改hosts文件配置增加IDC内部域名IP，在IDC内部服务IP变化时需要实时刷新CCE节点的配置，这很难做到且会导致不可用。

说明

内容审核服务与图像识别服务仅为举例，也可以是其他服务。

图 10-17 某企业 APP 微服务示意图



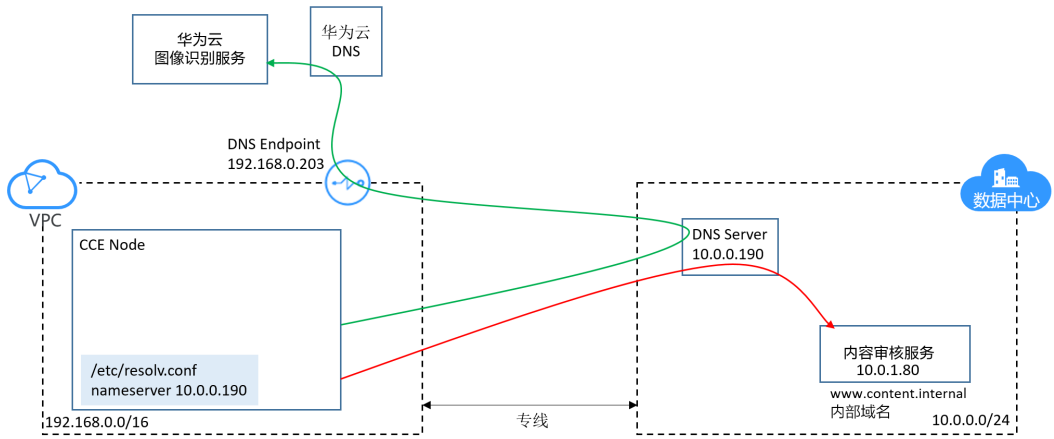
本文介绍一种自建IDC与CCE集群共享域名解析的方案，可做到同时解析华为云域名和外部域名。

解决方案一：通过 DNS Endpoint 做级联解析

利用VPCEP服务创建DNS Endpoint，使得IDC能访问华为云域名解析服务器，将DNS Endpoint与IDC的域名解析服务器做级联，从而使得CCE集群中节点及容器在域名解析时，通过IDC的域名解析服务器进行解析。

- 如果是需要解析华为云内域名，则转发给DNS Endpoint，使用华为云DNS解析出地址并返回。
- 如果是需要解析IDC域名，则直接通过IDC的域名解析服务器直接解析出地址并返回。

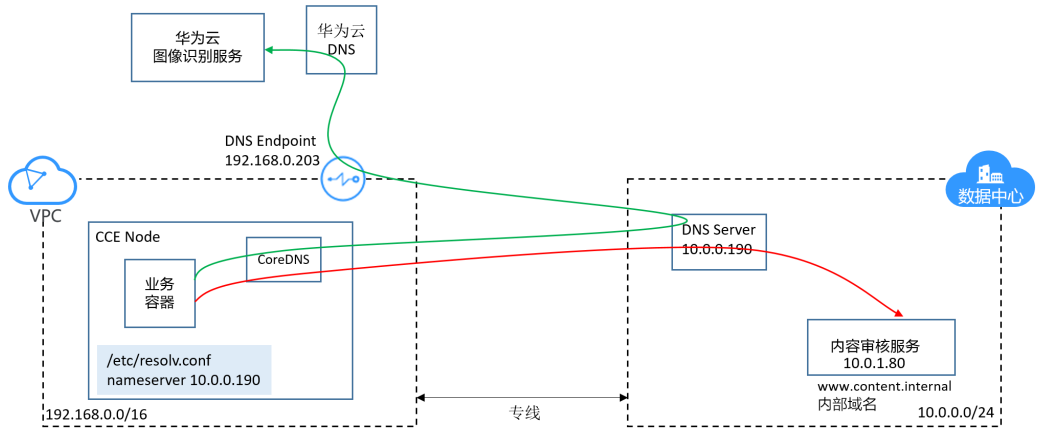
图 10-18 节点同时访问华为云域名和外部域名



对于容器中的域名解析，可在创建Pod时将DNS Policy设置为ClusterFirst，这样容器中的域名解析直接走CoreDNS。

- 如果是需要解析集群内域名，则CoreDNS直接解析出结果返回。
- 如果是解析外部域名，则通过CoreDNS，转到IDC的域名解析服务器解析。

图 10-19 容器同时访问华为云域名和外部域名

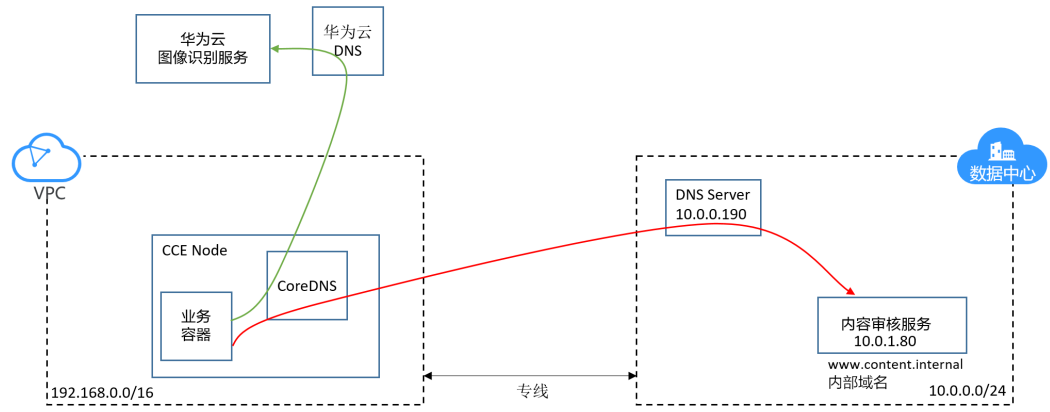


解决方案二：修改 CoreDNS 配置直接解析

直接修改CoreDNS配置，可在创建Pod时将DNS Policy设置为ClusterFirst，这样容器中的域名解析直接走CoreDNS。

- 如果是需要解析集群内域名，则CoreDNS直接解析出结果返回。
- 如果是解析外部域名，则通过CoreDNS，转到IDC的域名解析服务器解析。
- 如果是访问华为云内部域名，则直接使用华为云内部域名解析服务器解析

图 10-20 方案二：修改 CoreDNS 配置



方案比较

方案	优点	缺点
方案一：通过DNS Endpoint做级联解析	支持在CCE集群节点容器与节点同时解析外部域名	解析华为云内部域名时也需要通过外部域名解析服务器转发，存在性能损耗

方案	优点	缺点
方案二：修改CoreDNS配置	解析华为云内部域名时无需要通过外部域名解析服务器转发，不存在性能损耗	• 不支持在CCE集群节点解析外部域名。 • Coredns升级、回退会导致配置会丢失，需要重新配置。

10.4.2 通过 DNS Endpoint 做级联解析

前提条件

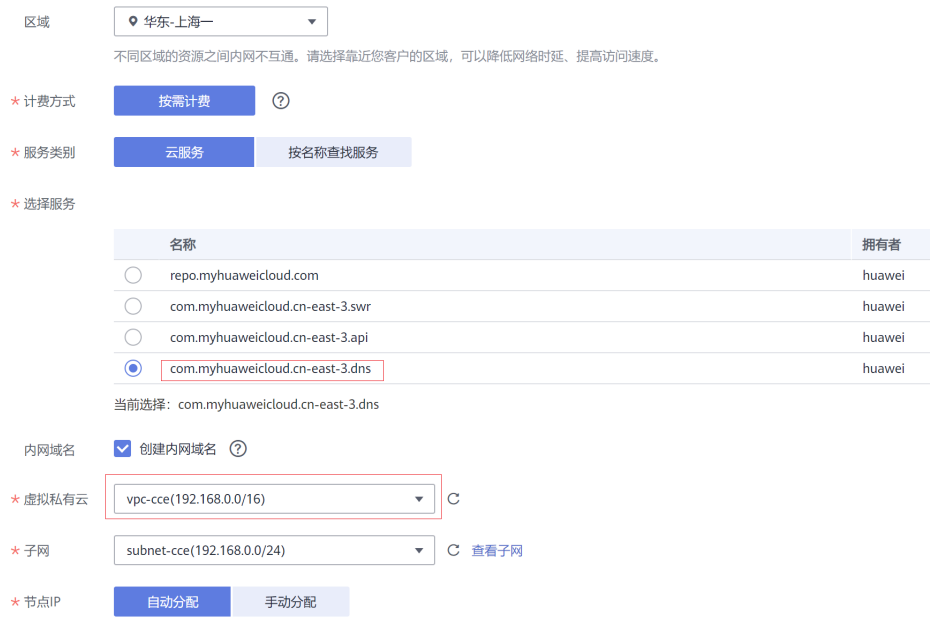
CCE集群所在VPC与线下IDC已经使用专线或其他方式正确连接，IDC与VPC网段和CCE集群容器网段能够互访。专线的创建方法请参见**云专线快速入门**。

操作步骤

步骤1 在CCE集群所在VPC创建 DNS Endpoint。

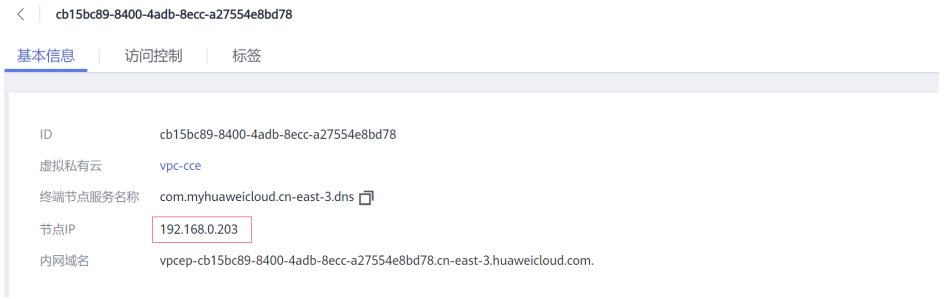
1. 登录**VPCEP控制台**。
2. 单击右上角“购买终端节点”。
3. 选择DNS服务和VPC，注意此处VPC需要选择CCE集群所在VPC。

图 10-21 创建 DNS Endpoint



4. 单击“立即购买”，完成创建。
   创建完成后，可在详情页中查看到DNS Endpoint的IP地址，如下图所示。

图 10-22 DNS Endpoint 的 IP 地址



步骤2 在IDC的域名解析服务器上做级联配置。

说明

此处配置跟具体域名解析服务器相关，不同域名解析服务器的配置方法不同，请根据实际情况配置。

这里使用**BIND**软件（一个常用的域名解析服务器软件）为例进行说明。

域名解析服务器上配置的关键是将需要解析华为云内部域名的任务转发给上一步创建的DNS Endpoint。

例如BIND中可以修改/etc/named.conf文件，将域名解析任务转发给DNS Endpoint，如下所示，添加如下两行，将转发解析转发到DNS Endpoint的地址，其中192.168.0.203为**步骤1**创建的DNS Endpoint的地址。

```
options {
        listen-on port 53 { any; };
        listen-on-v6 port 53 { ::1; };
        directory       "/var/named";
        dump-file       "/var/named/data/cache_dump.db";
        statistics-file "/var/named/data/named_stats.txt";
        memstatistics-file "/var/named/data/named_mem_stats.txt";
        recursing-file  "/var/named/data/named.recursing";
        secroots-file   "/var/named/data/named.secroots";
        allow-query     { any; };

        forward first;
        forwarders { 192.168.0.203; };

        .....
};
```

步骤3 修改CCE集群Node节点的DNS配置。

修改Node节点的DNS有如下两种方法。

方法一：

在Node节点创建完成后，修改节点的DNS配置。

1. 登录CCE集群的Node节点。
2. 修改/etc/resolv.conf文件，修改nameserver的地址为IDC的域名解析服务器地址。

   ```
   # vi /etc/resolv.conf
   nameserver 10.0.0.190
   ```

3. 执行如下命令锁定resolv.conf文件，防止被华为云自动更新。

   chattr +i /etc/resolv.conf

   更详细的配置DNS步骤可以参考**配置DNS**。

方法二：

修改CCE集群所在的VPC子网的DNS配置，这样新创建的Node节点的/etc/resolv.conf文件中会直接刷新成指定的域名解析服务器地址。

此方法需要确保节点能够正常使用IDC的域名解析服务器解析华为云内网域名，否则会导致节点无法创建。建议在调试无问题后再修改VPC子网的DNS配置。

图 10-23 子网的 DNS 配置

步骤4 配置工作负载的DNS Policy。

创建工作负载时，容器中的域名解析可以在YAML中配置dnsPolicy为ClusterFirst，如下所示。Kubernetes默认也是将dnsPolicy设置为ClusterFirst，如此处不做配置，Kubernetes也是默认设置为ClusterFirst。工作负载的DNS详细配置可以参考**DNS配置说明**。

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

----结束

配置验证

配置完成后，可以在集群节点中使用dig命令查看到服务器是10.0.0.190，也就是使用IDC的域名解析服务器做了域名解析，这说明在节点访问IDC的域名没有问题。

```
# dig cce.ap-southeast-1.myhuaweicloud.com

; <<>> DiG 9.9.4-61.1.h14.eulerosv2r7 <<>> cce.ap-southeast-1.myhuaweicloud.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24272
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;cce.ap-southeast-1.myhuaweicloud.com. IN    A

;; ANSWER SECTION:
cce.ap-southeast-1.myhuaweicloud.com. 274 IN A       100.125.4.16

;; Query time: 4 msec
;; SERVER: 10.0.0.190#53(10.0.0.190)
;; WHEN: Tue Feb 23 19:16:08 CST 2021
;; MSG SIZE  rcvd: 76
```

在集群节点访问华为云内域名，如下所示，能够看到已经解析出对应的IP地址，说明能够解析。

```
# ping cce.ap-southeast-1.myhuaweicloud.com
PING cce.ap-southeast-1.myhuaweicloud.com (100.125.4.16) 56(84) bytes of data.
```

创建一个Pod访问华为云域名，如下所示，同样能够ping通，说明能够解析。

```
# kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
# ping cce.ap-southeast-1.myhuaweicloud.com
PING cce.ap-southeast-1.myhuaweicloud.com (100.125.4.16) 56(84) bytes of data.
```

10.4.3 修改 CoreDNS 配置直接解析

前提条件

CCE集群所在VPC与线下IDC已经使用专线或其他方式正确连接，IDC与VPC网段和CCE集群容器网段能够互访。专线的创建方法请参见**云专线快速入门**。

操作步骤

CoreDNS的配置都存储在名为coredns的ConfigMap下，您可以在kube-system命名空间下找到该配置项。使用如下命令可以查看到默认配置的内容。

kubectl get configmap coredns -n kube-system -oyaml

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: coredns
  namespace: kube-system
  selfLink: /api/v1/namespaces/kube-system/configmaps/coredns
  uid: d54ed5df-f4a0-48ec-9bc0-3efc1ac76af0
  resourceVersion: '21789515'
  creationTimestamp: '2021-03-02T09:21:55Z'
  labels:
    app: coredns
    k8s-app: coredns
    kubernetes.io/cluster-service: 'true'
    kubernetes.io/name: CoreDNS
    release: cceaddon-coredns
data:
  Corefile: |-
    .:5353 {
        bind {$POD_IP}
        cache 30
        errors
        health {$POD_IP}:8080
        kubernetes cluster.local in-addr.arpa ip6.arpa {
          pods insecure
          upstream /etc/resolv.conf
          fallthrough in-addr.arpa ip6.arpa
        }
        loadbalance round_robin
        prometheus {$POD_IP}:9153
        forward . /etc/resolv.conf
        reload
    }
```

如上所示，CoreDNS的所有配置都在Corefile这个配置项下。默认情况下任何不属于Kubernetes集群内部的域名，其DNS请求都将指向forward指定的 DNS 服务器地址，这里“forward . /etc/resolv.conf”里面第一个“.”代表所有域名，后面“/etc/resolv.conf”表示使用节点的域名解析服务器。

通常要解析特定外部域名时，可以单独添加配置项，执行如下步骤修改coredns的Corefile配置，对于content.internal这个域，将域名解析都转发到10.0.0.190这个域名解析服务器上。

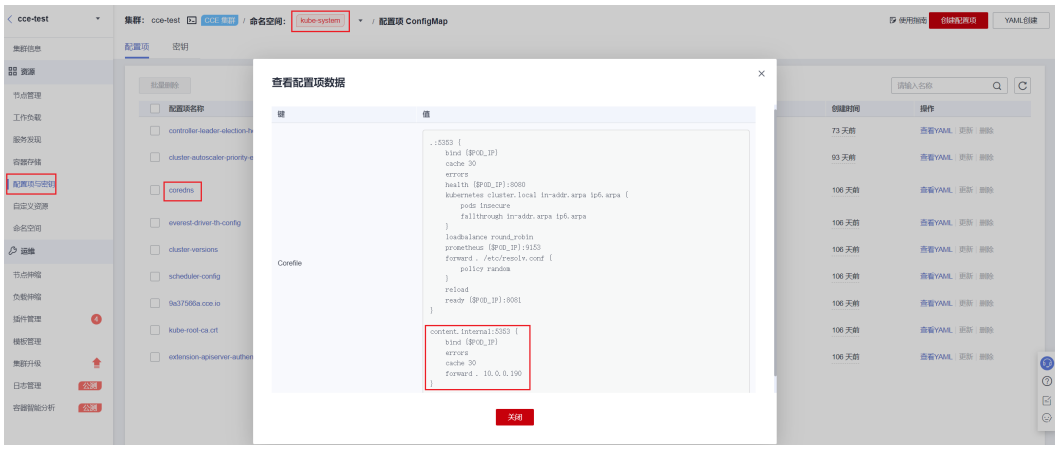
步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下添加存根域。格式为一个键值对，键为DNS后缀域名，值为一个或一组DNS IP地址。本例中，配置为'content.internal --10.0.0.190'。

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置项与密钥”，在“kube-system”命名空间下，查看coredns配置项数据，确认是否更新成功。



对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
content.internal:5353 {
    bind {$POD_IP}
    errors
    cache 30
    forward . 10.0.0.190
}
```

CoreDNS其他配置详解，请参见**自定义 DNS 服务**。

----结束

配置验证

创建一个Pod访问IDC的域名，如下所示，同样能够ping通，说明能够解析。

```
web-terminal-568c6566df-c9jhl:~# kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
# ping www.content.internal
PING www.content.internal (10.0.1.80) 56(84) bytes of data.
64 bytes from 10.0.1.80: icmp_seq=1 ttl=64 time=1.08 ms
64 bytes from 10.0.1.80: icmp_seq=2 ttl=64 time=0.337 ms
```

继续访问华为云域名，能够看到已经解析出对应的IP地址，说明能够解析。

```
# ping cce.ap-southeast-1.myhuaweicloud.com
PING cce.ap-southeast-1.myhuaweicloud.com (100.125.4.16) 56(84) bytes of data.
```

在集群节点上访问IDC的域名，无法ping通，说明CoreDNS的配置无法影响节点域名解析。

10.5 通过负载均衡配置实现会话保持

概念

会话保持可以确保用户在访问应用时的连续性和一致性。如果在客户端和服务器之间部署了负载均衡设备，很有可能这多个连接会被转发至不同的服务器进行处理。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。

例如在大多数需要用户身份认证的在线系统中，一个用户需要与服务器实现多次交互才能完成一次会话。由于多次交互过程中存在连续性，如果不配置会话保持，负载均衡可能会将部分请求分配至另一个后端服务器，但由于其他后端服务器并未经过用户身份认证，则会出现用户登录失效等交互异常。

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。

表 10-7 会话保持类型

类型	说明	支持的会话保持类型	会话保持时间	会话保持失效的场景
四层会话保持	当创建Service时，使用的协议为TCP或UDP，默认为四层会话保持。	**源IP地址**：基于源IP地址的简单会话保持，将请求的源IP地址作为散列键（HashKey），从静态分配的散列表中找出对应的服务器。即来自同一IP地址的访问请求会被转发到同一台后端服务器上进行处理。	• 默认时间：20分钟 • 最长时间：60分钟 • 取值范围：1-60分钟	• 客户端的源IP地址发生变化。 • 客户端访问请求超过会话保持时间。

类型	说明	支持的会话保持类型	会话保持时间	会话保持失效的场景
七层会话保持	当创建Ingress时，使用的协议为HTTP或HTTPS，默认为七层会话保持。	• **负载均衡器cookie**：负载均衡器会根据客户端第一个请求生成一个cookie，后续所有包含这个cookie值的请求都会由同一个后端服务器处理。 • **应用程序cookie**：该选项依赖于后端应用。后端应用生成一个cookie值，后续所有包含这个cookie值的请求都会由同一个后端服务器处理。	• 默认时间：20分钟 • 最长时间：1440分钟 • 取值范围：1-1440分钟	• 如果客户端发送请求未附带cookie，则会话保持无法生效。 • 客户端访问请求超过会话保持时间。

说明

在创建负载均衡时，分配策略选择“加权轮询算法”（即kubernetes.io/elb.lb-algorithm参数为ROUND_ROBIN）或“加权最少连接”（即kubernetes.io/elb.lb-algorithm参数为LEAST_CONNECTIONS）可配置会话保持；选择“源IP算法”（即kubernetes.io/elb.lb-algorithm参数为SOURCE_IP）时已支持基于源IP地址的会话保持，无需重复配置会话保持。

四层会话保持（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）。

在 CCE Standard 集群中开启四层会话保持

在CCE Standard集群中，Service开启基于源IP的会话保持需要满足以下条件：

1. Service的服务亲和级别选择“节点级别”（即Service的externalTrafficPolicy字段为Local）。
2. Service后端的应用开启反亲和，避免所有Pod均部署在同一节点。

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Service的负载均衡，以使用已有的ELB为例，配置源IP地址会话保持的YAML示例如下。

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # 加权轮询分配策略
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP    # 开启源IP会话保持
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local    # 服务亲和级别为“节点级别”
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

步骤3 登录ELB控制台，进入对应的ELB实例，查看监听器下的后端服务器组中，会话保持配置是否开启。

图 10-24 开启会话保持

----结束

在 CCE Turbo 集群中开启四层会话保持

在CCE Turbo集群中，Service开启基于源IP的会话保持与ELB类型相关。

- 使用独享型ELB时，ELB可以直通Pod（即Pod直接作为ELB的后端服务器组），因此Service开启基于源IP的会话保持无需配置服务亲和及应用反亲和。
- 使用共享型ELB时，Service开启基于源IP的会话保持需要满足以下条件：
  a. Service的服务亲和级别选择“节点级别”（即Service的externalTrafficPolicy字段为Local）。
  b. Service后端的应用开启反亲和，避免所有Pod均部署在同一节点。

操作步骤

- **独享型ELB场景**

  以使用已有的ELB为例，Service的负载均衡配置源IP地址会话保持的YAML示例如下。

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # 加权轮询分配策略
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP    # 开启源IP会话保持
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local     # CCE Turbo集群中，使用独享型ELB对服务亲和策略无要求
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

- **共享型ELB场景**

  a. 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
```

b. 创建Service的负载均衡。以使用已有的ELB为例，配置源IP地址会话保持的YAML示例如下。

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # 加权轮询分配策略
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP    # 开启源IP会话保持
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local    # 服务亲和级别为“节点级别”
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

c. 登录ELB控制台，进入对应的ELB实例，查看监听器下的后端服务器组中，会话保持配置是否开启。

图 10-25 开启会话保持

七层会话保持（Ingress）

七层的模式下可以开启基于http cookie和app cookie的会话保持 。

在 CCE Standard 集群中开启七层会话保持

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Service服务亲和级别选择“节点级别”（即Service的externalTrafficPolicy字段为Local）。
2. Ingress对应的应用（工作负载）应该开启与自身反亲和，避免所有Pod均部署在同一节点。

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 为工作负载创建Service。本文以NodePort类型Service为例。

会话保持的配置需在Service的配置中进行设置，由于Ingress可以对接多个Service，因此每个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      # 加权轮询分配策略
    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            # 自定义节点端口
  type: NodePort
  externalTrafficPolicy: Local   # 服务亲和级别为“节点级别”
```

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      # 加权轮询分配策略
    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。以下示例以使用已有的ELB为例，如需使用自动创建ELB的方式，请参考**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.id: *****
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，查看监听器下的后端服务器组中，会话保持配置是否开启。

图 10-26 开启会话保持

----结束

在 CCE Turbo 集群中开启七层会话保持

在Ingress上开启基于cookie的会话保持：

- 使用独享型ELB时，ELB可以直通Pod（即Pod直接作为ELB的后端服务器组），因此Ingress开启基于cookie的会话保持无需配置服务亲和及应用反亲和。
- 使用共享型ELB时，Ingress开启基于cookie的会话保持需要满足以下条件：
  a. Ingress对应的Service服务亲和级别选择“节点级别”（即Service的externalTrafficPolicy字段为Local）。
  b. Ingress对应的应用（工作负载）应该开启与自身反亲和，避免所有Pod均部署在同一节点。

操作步骤

- **独享型ELB场景**
  a. 为工作负载创建Service。CCE Turbo集群中，使用独享型ELB的Ingress需对接CLusterIP类型的Service。
  会话保持的配置需在Service的配置中进行设置，由于Ingress可以对接多个Service，因此每个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      # 加权轮询分配策略
    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: 0
  type: ClusterIP
```

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      # 加权轮询分配策略
    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
...
```

b. 创建Ingress，关联Service。以下示例以使用已有的ELB为例，如需使用自动创建ELB的方式，详情请参考**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: performance
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.id: *****
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
```

c. 登录ELB控制台，进入对应的ELB实例，查看监听器下的后端服务器组中，会话保持配置是否开启。

图 10-27 开启会话保持

- **共享型ELB场景**

a. 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
```

b. 为工作负载创建Service。CCE Turbo集群中，使用共享型ELB的Ingress需对接NodePort类型的Service。

会话保持的配置需在Service的配置中进行设置，由于Ingress可以对接多个Service，因此每个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      # 加权轮询分配策略
    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            # 自定义节点端口
  type: NodePort
  externalTrafficPolicy: Local   # 服务亲和级别为“节点级别”
```

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      # 加权轮询分配策略
    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
...
```


c. 创建Ingress，关联Service。以下示例以使用已有的ELB为例，如需使用自动创建ELB的方式，详情请参考**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.id: *****
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
```

d. 登录ELB控制台，进入对应的ELB实例，查看监听器下的后端服务器组中，会话保持配置是否开启。

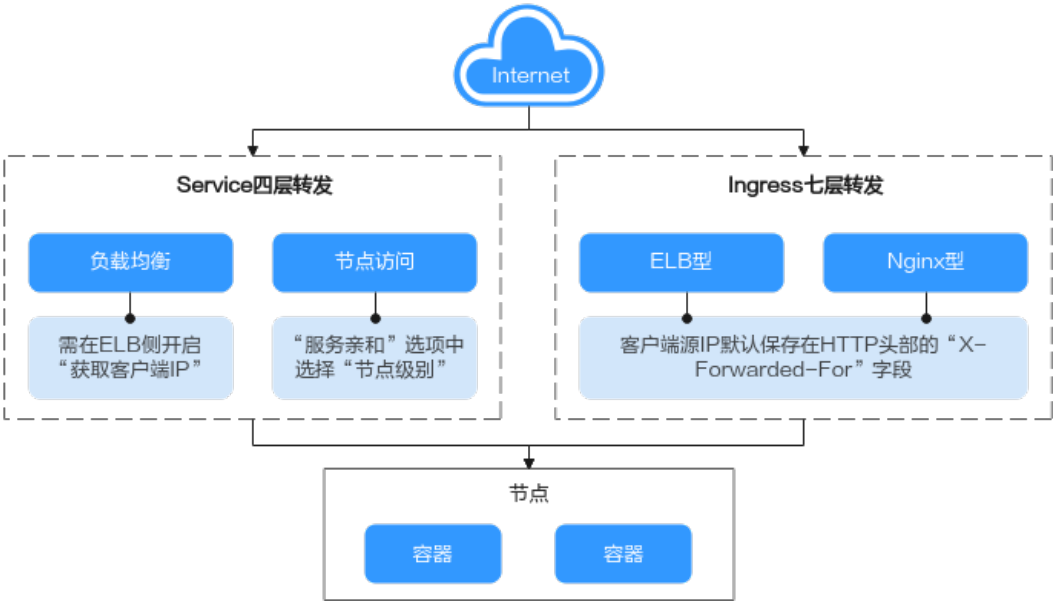
图 10-28 开启会话保持

10.6 不同场景下容器内获取客户端源 IP

在容器化场景下，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，外部请求在经过多次转发后，客户端源IP无法透传至容器，导致容器中的业务无法获取到客户端真实的源IP。

场景介绍

图 10-29 容器中获取源 IP



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。
- 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。

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。

说明

若已使用Istio服务网格，获取源IP地址的方法请参见**服务加入Istio后，如何获取客户端真实源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，如**表10-8**，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 10-8 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（CCE Turbo集群）	操作指导
七层转发（Ingress）	ELB型	共享型	支持	支持	ELB 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	Nginx Ingress
		独享型	支持	支持	默认开启，无需进行其他配置
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）

ELB Ingress

对于ELB Ingress（即使用HTTP/HTTPS协议），ELB默认会开启获取客户端源IP，无需其他操作。

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。

Nginx Ingress

说明

云原生网络2.0模型下（即CCE Turbo集群），**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。

- Nginx Ingress使用独享型ELB时，默认开启源地址透传功能，无需其他配置即可获取客户端源IP。
- Nginx Ingress使用共享型ELB时，获取客户端源IP的操作步骤如下：

步骤1 以Nginx工作负载为例，未配置源IP访问前，使用以下命令查看访问日志，其中nginx-c99fd67bb-ghv4q为Pod名称：

```
kubectl logs nginx-c99fd67bb-ghv4q
```

回显如下：

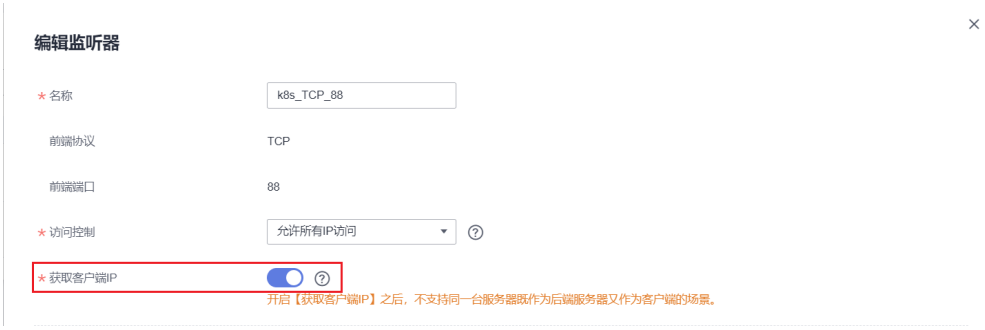
```
...
10.0.0.7 - - [17/Aug/2023:01:30:11 +0000] "GET / HTTP/1.1" 200 19 "http://114.114.114.114:9421/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 Edg/115.0.1901.203" "100.125.**.**"
```

其中**100.125.**.****为负载均衡的地址段，说明流量经过负载均衡转发。

步骤2 前往ELB控制台，开启ELB实例对应监听器的“获取客户端IP”功能。**独享型ELB默认开启源地址透传功能，无需手动开启。**

1. 登录弹性负载均衡ELB的管理控制台。
2. 在管理控制台左上角单击 图标，选择区域和项目。
3. 选择“服务列表 > 网络 > 弹性负载均衡”。
4. 在“负载均衡器”界面，单击需要操作的负载均衡名称。
5. 切换到“监听器”页签，单击需要修改的监听器名称右侧的“编辑”按钮。如果存在修改保护，请在监听器基本信息页面中关闭修改保护后重试。
6. 开启“获取客户端IP”开关。

图 10-30 开启开关



步骤3 重新访问工作负载，并查看新增的访问日志如下：

```
...
10.0.0.7 - - [17/Aug/2023:02:43:11 +0000] "GET / HTTP/1.1" 304 0 "http://114.114.114.114:9421/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 Edg/115.0.1901.203" "124.**.**.**"
```

显示成功获取到客户端源IP。

----结束

说明

如果您需要为集群Nginx Ingress Controller所使用的ELB开启WAF功能，不同的WAF模式会影响Nginx Ingress Controller获取真实的客户端IP：

- **使用WAF云模式的CNAME接入**

  采用CNAME模式接入，会导致请求先通过WAF，经过WAF进行防护检查之后再转发给ELB。因此即使ELB已开启源地址透传，实际上客户端得到为WAF的回源IP，造成Nginx Ingress Controller将默认无法获得真实的客户端IP。此时您可以编辑nginx-ingress插件，在nginx配置参数处添加以下配置。

  ```
  {
      "enable-real-ip": "true",
      "forwarded-for-header": "true",
      "proxy-real-ip-cidr": <您从WAF获取到的回源IP段>
  }
  ```

- **使用WAF云模式ELB接入**

  该模式为透明接入（旁路部署），仅支持独享型ELB，Nginx Ingress Controller默认可以获得真实的客户端IP。

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：

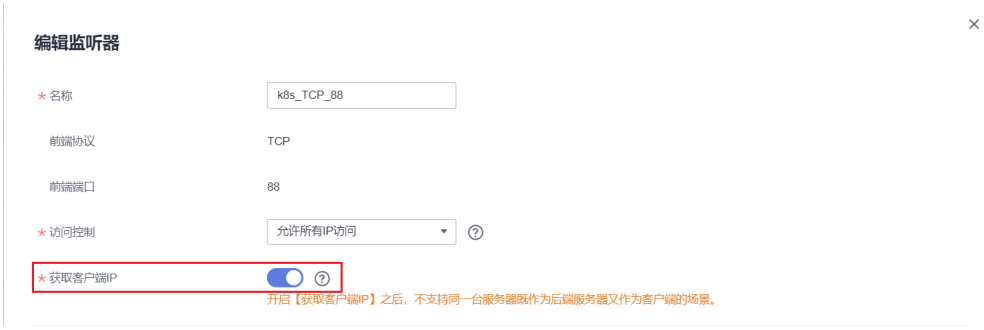
步骤1 在CCE控制台创建负载均衡类型的Service，服务亲和选择“**节点级别**”而不是“集群级别”。

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，单击需要修改的监听器名称右侧的“编辑”按钮。如果存在修改保护，请在监听器基本信息页面中关闭修改保护后重试。
6. 开启“获取客户端IP”开关。

图 10-31 开启开关

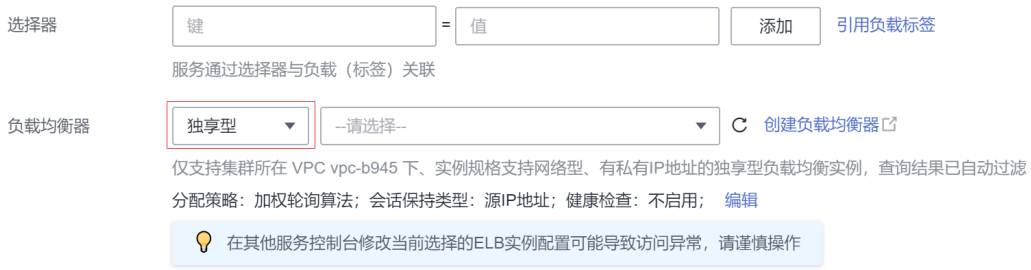


----结束

云原生网络2.0模型（CCE Turbo集群）

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控制台创建负载均衡服务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 10-32 服务亲和选择节点级别

10.7 通过配置容器内核参数增大监听队列长度

使用场景

net.core.somaxconn默认监听队列（backlog）长度为128，当服务繁忙不能满足要求时，需要增大监听队列长度。

操作步骤

步骤1 修改kubelet配置。

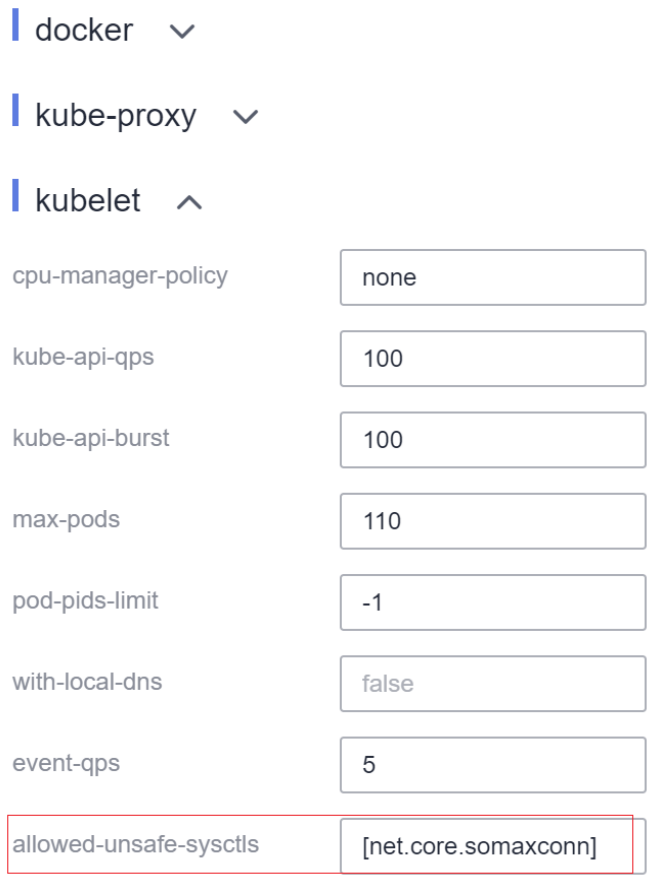
有两种方法可以修改kubelet配置。

- 修改节点池kubelet配置（仅支持1.15及以上集群）。

  登录CCE控制台，进入集群，单击节点池后“更多 > 配置管理”，修改kubelet配置参数。

图 10-33 节点池配置管理

图 10-34 修改 kubelet 参数



- **修改节点kubelet参数**

  a. 登录节点。

  b. 编辑**/opt/cloud/cce/kubernetes/kubelet/kubelet** 文件。1.15之前版本为 **/var/paas/kubernetes/kubelet/kubelet** 文件。

  开启net.core.somaxconn开关。

  ```
  --allowed-unsafe-sysctls=net.core.somaxconn
  ```

  ```
  [root@test-565556-38106 ~]# cat /var/paas/kubernetes/kubelet/kubelet
  DAEMON_ARGS=" --bootstrap-kubeconfig=/var/paas/kubernetes/kubelet/boot.conf --cert-dir=/var/paas/kubernetes/kubelet/pki --rotate
  -certificates=true --network-plugin=cni --cni-conf-dir=/etc/cni/net.d/ --node-ip=192.168.116.149 --provider-id=acb56430-9a20-11e
  9-bf1e-0255ac101f9c --kubeconfig=/var/paas/kubernetes/kubelet/kubeconfig --config=/var/paas/kubernetes/kubelet/kubelet_config.ya
  ml --authentication-token-webhook=true --pod-infra-container-image=cce-pause:2.0 --root-dir=/mnt/paas/kubernetes/kubelet --hostn
  ame-override=192.168.116.149 --allow-privileged=True --v=2 --node-labels="os.name=EulerOS_2.0_SP5,os.version=3.10.0-862.14.0.1.h
  147.eulerosv2r7.x86_64,os.architecture=amd64,failure-domain.beta.kubernetes.io/zone=cn-north-1a,failure-domain.beta.kubernetes.i
  o/region=cn-north-1,kubernetes.io/availablezone=cn-north-1a,failure-domain.beta.kubernetes.io/is-baremetal='false'" --machine_id
  _file=/var/paas/conf/server.conf --cadvisor-port=4194 --allowed-unsafe-sysctls=net.core.somaxconn"
  [root@test-565556-38106 ~]#
  ```

  c. 重启kubelet

  systemctl restart kubelet

  查看kubelet状态：

  systemctl status kubelet

  说明

  1.13及以下版本的集群在修改kubelet后，如果升级集群到更高版本则该配置会还原。

步骤2 创建pod安全策略。

CCE从**1.17.17**集群版本开始，kube-apiserver开启了pod安全策略，需要在pod安全策略的allowedUnsafeSysctls中增加net.core.somaxconn配置才能生效（1.17.17以下版本的集群可跳过此步骤）。

- CCE的安全策略介绍，详情请参见**Pod安全策略配置**。
- K8s社区安全策略介绍，详情请参见**PodSecurityPolicy**。
- 对于开启net.core.somaxconn开关的集群，需要在对应的Pod安全策略的allowedUnsafeSysctls中增加此项配置，如创建新的Pod安全策略，示例如下：

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
  - net.core.somaxconn
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  fsGroup:
    rule: RunAsAny
  hostIPC: true
  hostNetwork: true
  hostPID: true
  hostPorts:
  - max: 65535
    min: 0
  privileged: true
  runAsGroup:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - '*'
```

创建Pod安全策略sysctl-psp后，还需要为它绑定RBAC权限控制。

示例如下：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: sysctl-psp
rules:
  - apiGroups:
      - "*"
    resources:
      - podsecuritypolicies
    resourceNames:
      - sysctl-psp
    verbs:
      - use

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: sysctl-psp
roleRef:
  kind: ClusterRole
  name: sysctl-psp
  apiGroup: rbac.authorization.k8s.io
subjects:
```



```
  - kind: Group
    name: system:authenticated
    apiGroup: rbac.authorization.k8s.io
```

步骤3 创建工作负载，配置内核参数值，且配置与**步骤1**中节点亲和。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    description: ''
  labels:
    appgroup: ''
  name: test1
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test1
  template:
    metadata:
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
      labels:
        app: test1
    spec:
      containers:
        - image: 'nginx:1.14-alpine-perl'
          name: container-0
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
            limits:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
      securityContext:
        sysctls:
          - name: net.core.somaxconn
            value: '3000'
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: kubernetes.io/hostname
                    operator: In
                    values:
                      - 192.168.x.x       #节点名称
```

步骤4 登录部署工作负载的节点，进入容器查看参数配置是否生效。

在容器中执行如下命令查询配置参数是否生效。

sysctl -a |grep somax

图 10-35 查看参数配置

```
user@uw8i8he1ka75nyk-machine:~$ kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
test1-7794f95b55-fmsll   1/1     Running   0          17s
user@uw8i8he1ka75nyk-machine:~$ kubectl exec -it test1-7794f95b55-fmsll -- /bin/sh
/ # sysctl -a |grep somax
net.core.somaxconn = 3000
sysctl: error reading key 'net.ipv6.conf.all.stable_secret': I/O error
sysctl: error reading key 'net.ipv6.conf.default.stable_secret': I/O error
sysctl: error reading key 'net.ipv6.conf.eth0.stable_secret': I/O error
sysctl: error reading key 'net.ipv6.conf.lo.stable_secret': I/O error
```

----结束

10.8 LoadBalancer 类型 Service 使用 pass-through 能力

应用现状

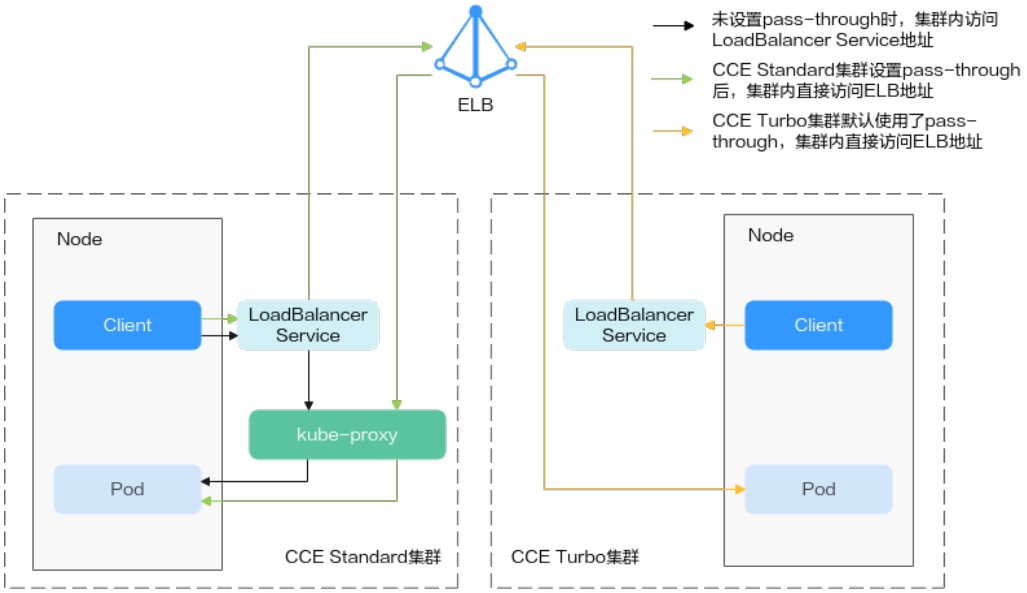
Kubernetes集群可以将运行在一组Pod上的应用程序发布为服务，提供统一的四层访问入口。对于Loadbalancer类型的service，kube-proxy默认会将Service的status中LoadbalanceIP地址配置到节点本地的转发规则中，集群内部访问ELB的地址，流量就会在集群内部转发，而不会经过ELB转发。

集群内部转发功能是kube-proxy组件负责，kube-proxy有iptables和IPVS两种转发模式，iptables是一种简单的轮询转发，IPVS虽有多种转发模式，但也需要修改kube-proxy的启动参数，不能像ELB那样灵活配置转发策略，且无法利用ELB的健康检查能力。

解决方案

CCE服务支持pass-through能力，通过Loadbalance类型Service配置kubernetes.io/elb.pass-through的annotation实现集群内部访问Service的ELB地址时绕出集群，并通过ELB的转发最终转发到后端的Pod。

图 10-36 pass-through 访问示例



- 对于CCE集群：

  集群内部客户端访问LB类型Service时，访问请求默认是通过集群服务转发规则（iptables或IPVS）转发到后端的容器实例。

  当LB类型Service配置elb.pass-through后，集群内部客户端访问Service地址时会先访问到ELB，再通过ELB的负载均衡能力先访问到节点，然后通过集群服务转发规则（iptables或IPVS）转发到后端的容器实例。

- 对于CCE Turbo集群：

  集群内部客户端访问LB类型Service时，默认使用pass-through方式，此时客户端会直接访问ELB私网地址，然后通过ELB直接连接容器实例。

约束限制

- 独享型负载均衡配置pass-through后，CCE Standard集群在工作负载同节点和同节点容器内无法通过Service访问。
- 1.15及以下老版本集群暂不支持该能力。
- IPVS网络模式下，对接同一个ELB的Service需保持pass-through设置情况一致。
- 使用节点级别（Local）的服务亲和的场景下，会自动设置kubernetes.io/elb.pass-through为onlyLocal，开启pass-through能力。

操作步骤

下文以nginx镜像创建无状态工作负载，并创建一个具有pass-through的Service。

步骤1 使用nginx镜像创建无状态负载。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

步骤2 Loadbalance类型的Service，并设置kubernetes.io/elb.pass-through为true。本示例中自动创建了一个名为james的共享型ELB实例。

Loadbalance类型的Service具体创建方法请参见**自动创建负载均衡类型Service**。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
```

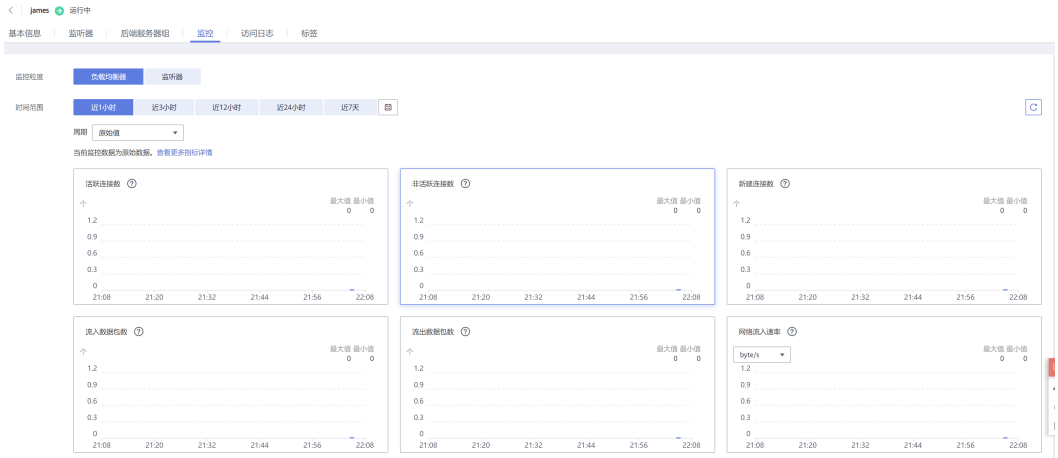


```
    kubernetes.io/elb.pass-through: "true"
    kubernetes.io/elb.class: union
    kubernetes.io/elb.autocreate: '{"type":"public","bandwidth_name":"cce-
bandwidth","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_ty
pe":"5_bgp","name":"james"}'
  labels:
    app: nginx
  name: nginx
spec:
  externalTrafficPolicy: Local
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

----结束

配置验证

查看上面创建的Service对应的ELB，名称为james，可以看到ELB的连接数为0，如下图所示。



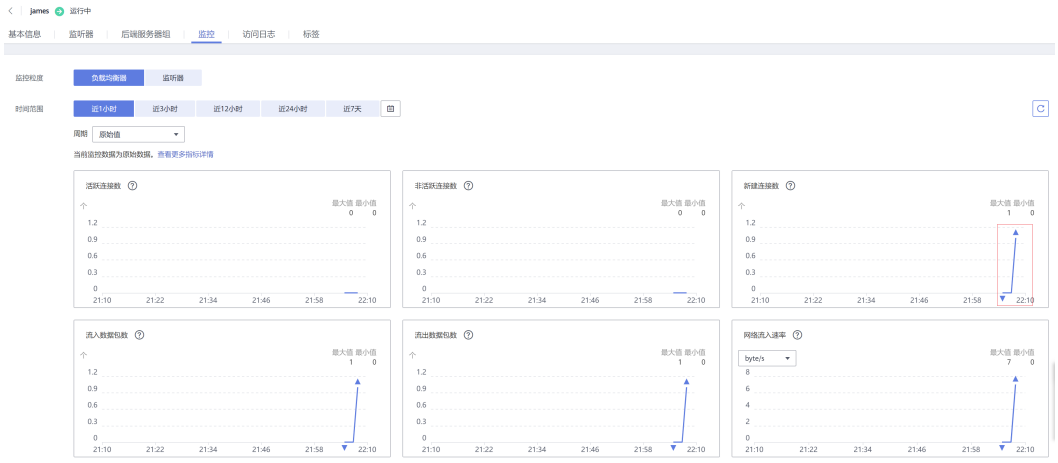
使用kubectl连接集群，进入到某一个nginx容器中，然后访问ELB的地址。可以看到能够正常访问。

```
# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
nginx-7c4c5cc6b5-vpncx   1/1     Running   0          9m47s
nginx-7c4c5cc6b5-xj5wl   1/1     Running   0          9m47s
# kubectl exec -it nginx-7c4c5cc6b5-vpncx -- /bin/sh
# curl 120.46.141.192
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```



```
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

稍微等待一段时间看ELB的监控数据，可以看到ELB有一个新建访问连接，这就证明了这次访问经过ELB，与预期一致。



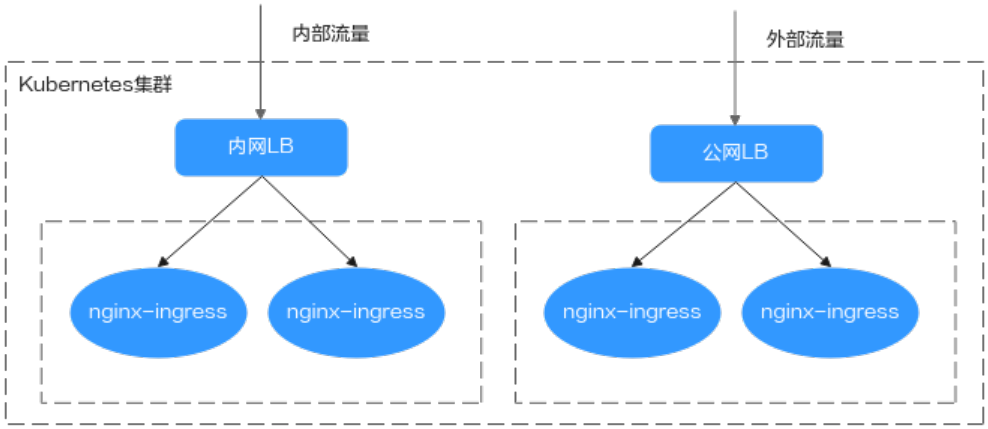
10.9 通过模板包部署 Nginx Ingress Controller

10.9.1 自定义部署 Nginx Ingress Controller

应用现状

Nginx Ingress Controller是一款业界流行的开源Ingress控制器，有着广泛的应用。在大规模集群场景下，用户有在集群中部署多套Nginx Ingress Controller的诉求，不同流量使用不同的控制器，将流量区分开。例如，集群中部分服务需要通过公网Ingress方式对外提供访问，但是又有部分对内开放的服务不允许使用公网访问，只支持对同VPC内的其他服务访问，您可以通过部署两套独立的Nginx Ingress Controller，绑定两个不同的ELB实例来满足这类需求场景。

图 10-37 多个 Nginx Ingress 应用场景



解决方案

您可以通过以下方案，实现在同一个集群中部署多个Nginx Ingress Controller。

- （推荐）安装NGINX Ingress控制器插件，在同一个集群中一键部署多个实例。详情请参见**安装多个NGINX Ingress控制器**。

  v1.23集群需安装2.2.52及以上版本的插件，对于v1.23以上的集群需安装2.5.4及以上版本的插件。
- 通过安装开源Helm包，在集群中自定义部署多套Nginx Ingress Controller。该方案需要配置的参数较复杂，您需要通过配置ingress-class**参数**（缺省值为nginx）来声明Nginx Ingress Controller监听的范围。这样在创建Ingress时可以通过选择不同的Nginx Ingress Controller进行流量区分。

前提条件

- 安装模板过程中，可能需要拉取公网镜像，因此节点需要绑定EIP。

约束与限制

- 部署多个Nginx Ingress Controller时，每个Controller需要对接一个ELB，并保证对接的ELB需要拥有至少两个监听器配额，且端口80和443没有被监听器占用。如果使用独享型ELB，需要支持网络型规格。
- 使用社区提供的nginx-ingress模板与镜像时，使用过程中对于因社区软件本身缺陷导致的业务受损CCE服务不提供额外维护。**商用场景请谨慎使用**。

部署 Nginx Ingress Controller

您可以通过以下步骤在集群中部署多套完全独立的Nginx Ingress Controller服务。

步骤1 获取模板包。

前往**社区模板发布页面**，选择合适的版本并下载tgz格式的Helm Chart包。本文以社区**4.4.2**版本的模板包为例，该模板包适用于v1.21及以上的CCE集群。由于不同版本的模板包配置项可能存在差异，本文中的配置仅对**4.4.2**版本生效。

步骤2 上传模板

1. 登录CCE控制台，进入集群，在左侧导航栏中选择“应用模板”，在右上角单击“上传模板”。
2. 单击“添加文件”，选中待上传的模板包后，单击“上传”。

步骤3 自定义value.yaml

您可在本地创建一个**value.yaml**配置文件用于设置安装工作负载参数，在安装时只需导入此配置文件进行自定义安装，其他未指定的参数将会使用默认配置。

配置内容如下：

```
controller:
  image:
    repository: registry.k8s.io/ingress-nginx/controller
    registry: ""
    image: ""
    tag: "v1.5.1"  #controller版本
    digest: ""
  ingressClassResource:
    name: ccedemo          #同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为nginx和cce
    controllerValue: "k8s.io/ingress-nginx-demo"  #同一个集群中不同套Ingress Controller的监听标识必须唯一，且不能设置为k8s.io/ingress-nginx
  ingressClass: ccedemo   #同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为nginx和cce
  service:
    annotations:
      kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0     #ELB ID
      kubernetes.io/elb.class: performance  #仅独享型ELB需要添加此注解
  config:
    keep-alive-requests: 100
  extraVolumeMounts: # 挂载节点上的/etc/localtime文件，进行时区同步
    - name: localtime
      mountPath: /etc/localtime
      readOnly: true
  extraVolumes:
    - name: localtime
      type: Hostpath
      hostPath:
        path: /etc/localtime
  admissionWebhooks: # 关闭webhook验证开关
    enabled: false
    patch:
      enabled: false
  resources: # 设定controller的资源限制，可根据需求自定义
    requests:
      cpu: 200m
      memory: 200Mi
defaultBackend: # 设置defaultBackend
  enabled: true
  image:
    repository: registry.k8s.io/defaultbackend-amd64
    registry: ""
```

```
image: ""
tag: "1.5"
digest: ""
```

上述参数详情请参见**表10-9**。

步骤4 创建模板实例。

1. 登录CCE控制台，进入集群，在左侧导航栏中选择“应用模板”。
2. 在已上传的模板中，单击“安装”。
3. 填写“实例名称”，选择“命名空间”和“选择版本”。
4. 单击“配置文件”后的“添加文件”按钮，选择本地创建的YAML配置文件，单击“安装”。
5. 在“模板实例”页签下可以查看模板实例的安装情况。

----结束

测试验证

创建一个工作负载，配置新部署的Nginx Ingress Controller为其提供网络访问。

步骤1 创建nginx工作负载。

1. 登录CCE控制台，进入集群，在左侧导航栏中选择“工作负载”，单击右上角“YAML创建”。
2. 填写以下内容，并单击“确定”。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx    #若使用“开源镜像中心”的镜像，可直接填写镜像名称；若使用“我的镜像”中的镜像，请在SWR中获取具体镜像地址。
        imagePullPolicy: Always
        name: nginx
      imagePullSecrets:
      - name: default-secret
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  ports:
```



```
- name: service0
  port: 80              # 访问Service的端口
  protocol: TCP         # 访问Service的协议，支持TCP和UDP
  targetPort: 80        # Service访问目标容器的端口，本例中nginx镜像默认使用80端口
selector:               # 标签选择器，Service通过标签选择Pod，将访问Service的流量转发给Pod
  app: nginx
type: ClusterIP         # Service的类型，ClusterIP表示在集群内访问
```

步骤2 创建Ingress，通过新部署的Nginx Ingress Controller提供网络访问。

1. 在左侧导航栏中选择“服务”，切换至“路由”页签，单击右上角“YAML创建”。

   说明

   对接非插件部署的Nginx Ingress Controller时，只支持使用YAML的方式创建Ingress。

2. 填写以下内容，并单击“确定”。

   v1.23及以上版本集群：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
spec:
  ingressClassName: ccedemo  # 填写新创建的nginx ingress controller的ingressClass
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /
        pathType: ImplementationSpecific   #匹配取决于 IngressClass
        backend:
          service:
            name: nginx    #替换为您的目标服务名称
            port:
              number: 80   #替换为您的目标服务端口
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

   v1.23以下版本集群：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tomcat-t1
  namespace: test
  annotations:
    kubernetes.io/ingress.class: ccedemo  # 填写新创建的nginx ingress controller的ingressClass
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - path: /
            pathType: ImplementationSpecific
            backend:
              serviceName: nginx  #替换为您的目标服务名称
              servicePort: 80     #替换为您的目标服务端口
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

步骤3 登录集群节点，分别通过集群Nginx Ingress插件中的Controller和新部署的Nginx Ingress Controller服务来访问该应用。

- 通过新建的Nginx Ingress Controller服务来访问该应用（预期返回Nginx页面），其中192.168.114.60为新建的Nginx Ingress Controller服务对应的ELB地址。

```
curl -H "Host: foo.bar.com" http://192.168.114.60
```



```
[root@192-168-9-72 paas]# curl -H "Host: foo.bar.com" http://192.168.114.60
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

- 通过Nginx Ingress插件中的Controller服务（预期返回404），其中192.168.9.226为Nginx Ingress插件对应的ELB地址。

  ```
  curl -H "Host: foo.bar.com" http://192.168.9.226
  ```

  ```
  [root@192-168-9-72 paas]# curl -H "Host: foo.bar.com" http://192.168.9.226
  default backend - 404[root@192-168-9-72 paas]#
  ```

----结束

配置参数说明

表 10-9 nginx-ingress 主要参数配置

参数	描述
controller.image.repository	ingress-nginx镜像地址，建议与CCE提供的Nginx Ingress插件镜像一致，也可自定义。 ● Nginx Ingress插件镜像：Nginx Ingress插件的镜像地址可通过已安装插件实例的YAML文件查看。 ● 自定义：自定义地址需要保证镜像可拉取。
controller.image.registry	镜像仓库域名，该参数需要与controller.image.image同时填写。 如已填写controller.image.repository，则无需再填写该参数，建议将controller.image.registry和controller.image.image设为空值。

参数	描述
controller.image.image	镜像名称。该参数需要与controller.image.registry同时填写。 如已填写controller.image.repository，则无需再填写该参数，建议将controller.image.registry和controller.image.image设为空值。
controller.image.tag	ingress-nginx镜像版本，建议与CCE提供的Nginx Ingress插件镜像一致，也可自定义。 Nginx Ingress插件的镜像版本可通过已安装插件实例的YAML文件查看，需要根据插件版本进行替换。
controller.ingressClass	设置Ingress Controller所对应的IngressClass的名称。 **说明** 同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为**nginx**和**cce**（**nginx**是集群默认Nginx Ingress Controller的监听标识，**cce**则是使用ELB Ingress Controller的配置）。 示例：**ccedemo**
controller.image.digest	建议为空值，该参数非空时可能无法拉取CCE提供的Nginx Ingress插件镜像。
controller.ingressClassResource.name	需要与ingressClass值相同。 示例：**ccedemo**
controller.ingressClassResource.controllerValue	同一个集群中不同套Ingress Controller的监听标识必须唯一，且不能设置为**k8s.io/ingress-nginx**（**k8s.io/ingress-nginx**是默认Nginx Ingress Controller的监听标识）。 示例：**k8s.io/ingress-nginx-demo**
controller.config	nginx配置参数，配置参数范围请参考**社区文档**。不在范围内的参数配置不会生效。 建议增加如下配置： "keep-alive-requests": "100"
controller.extraInitContainers	init容器，在主容器启动前执行，可用于Pod参数的初始化配置。 配置参数示例请参见**高并发业务场景参数优化**。
controller.admissionWebhooks.enabled	是否开启admissionWebhooks，可以对Ingress对象进行有效性校验，避免因配置错误导致ingress-controller不断重新加载资源，导致业务中断。 此处设置为false，表示不开启。如需开启，请参见**admissionWebhook配置**示例。
controller.admissionWebhooks.patch.enabled	同上，表示是否开启admissionWebhooks。此处设置为false。

参数	描述
controller.service.annotations	Key: value类型，此处需加上ELB ID，如下所示： kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0 独享型负载均衡还需要加上elb.class，如下所示： kubernetes.io/elb.class: performance
controller.resources.requests.cpu	Nginx controller的CPU资源申请值，可根据需求自定义。
controller.resources.requests.memory	Nginx controller的内存资源申请值，可根据需求自定义。
defaultBackend.image.repository	default-backend镜像地址，建议与CCE提供的Nginx Ingress插件镜像一致，也可自定义。 • Nginx Ingress插件镜像：Nginx Ingress插件的镜像地址可通过已安装插件实例的YAML文件查看。 • 自定义：自定义地址需要保证镜像可拉取。
defaultBackend.image.tag	default-backend镜像版本，建议与CCE提供的Nginx Ingress插件镜像一致，也可自定义。

更多参数配置说明请参见**ingress-nginx**。

10.9.2 Nginx Ingress Controller 高级配置

高并发业务场景参数优化

针对高并发业务场景，可通过参数配置进行优化：

1. 通过ConfigMap对Nginx Ingress Controller整体参数进行优化。
2. 通过InitContainers对Nginx Ingress Controller内核参数进行优化。

优化后的**value.yaml**配置文件如下：

```
controller:
  image:
    repository: registry.k8s.io/ingress-nginx/controller
    registry: ""
    image: ""
    tag: "v1.5.1"  #controller版本
    digest: ""
  ingressClassResource:
    name: ccedemo         #同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为nginx和cce
    controllerValue: "k8s.io/ingress-nginx-demo"  #同一个集群中不同套Ingress Controller的监听标识必须唯一，且不能设置为k8s.io/ingress-nginx
  ingressClass: ccedemo   #同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为nginx和cce
  service:
    annotations:
      kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0     #ELB ID
      kubernetes.io/elb.class: performance  #仅独享型ELB需要添加此注解
# Nginx参数优化
  config:
    keep-alive-requests: 10000
    upstream-keepalive-connections: 200
```

```
    max-worker-connections: 65536
# 内核参数优化
  extraInitContainers:
   - name: init-myservice
     image: busybox
     securityContext:
       privileged: true
     command: ['sh', '-c', 'sysctl -w net.core.somaxconn=65535;sysctl -w
net.ipv4.ip_local_port_range="1024 65535"']
  extraVolumeMounts: # 挂载节点上的/etc/localtime文件，进行时区同步
    - name: localtime
      mountPath: /etc/localtime
      readOnly: true
  extraVolumes:
    - name: localtime
      type: Hostpath
      hostPath:
        path: /etc/localtime
  admissionWebhooks: # 关闭webhook验证开关
    enabled: false
    patch:
      enabled: false
  resources: # 设定controller的资源限制，可根据需求自定义
    requests:
      cpu: 200m
      memory: 200Mi
defaultBackend: # 设置defaultBackend
  enabled: true
  image:
    repository: registry.k8s.io/defaultbackend-amd64
    registry: ""
    image: ""
    tag: "1.5"
    digest: ""
```

admissionWebhook 配置

Nginx Ingress Controller支持admissionWebhook配置，通过设置controller.admissionWebhook参数，可以对Ingress对象进行有效性校验，避免因配置错误导致ingress-controller不断重新加载资源，导致业务中断。

说明

- 使用admissionWebhook特性时，APIServer需要开启webhook相关配置，必须包含MutatingAdmissionWebhook与ValidatingAdmissionWebhook

  特性开关为 --admission-control=MutatingAdmissionWebhook,ValidatingAdmissionWebhook

  如未开启，需要提交工单申请开启。
- 开启admissionWebhook特性后，如需卸载重装Nginx Ingress Controller，会有Secret残留，需要手动清理。

开启admissionWebhook的**value.yaml**配置文件如下：

```
controller:
  image:
    repository: registry.k8s.io/ingress-nginx/controller
    registry: ""
    image: ""
    tag: "v1.5.1"  #controller版本
    digest: ""
  ingressClassResource:
    name: ccedemo         #同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为nginx和cce
    controllerValue: "k8s.io/ingress-nginx-demo"  #同一个集群中不同套Ingress Controller的监听标识必须唯一，且不能设置为k8s.io/ingress-nginx
  ingressClass: ccedemo    #同一个集群中不同套Ingress Controller名称必须唯一，且不能设置为nginx和cce
```

```
  service:
    annotations:
      kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0     #ELB ID
      kubernetes.io/elb.class: performance  #仅独享型ELB需要添加此注解
  config:
    keep-alive-requests: 100
  extraVolumeMounts: # 挂载节点上的/etc/localtime文件，进行时区同步
    - name: localtime
      mountPath: /etc/localtime
      readOnly: true
  extraVolumes:
    - name: localtime
      type: Hostpath
      hostPath:
        path: /etc/localtime
  admissionWebhooks:
    annotations: {}
    enabled: true
    extraEnvs: []
    failurePolicy: Fail
    port: 8443
    certificate: "/usr/local/certificates/cert"
    key: "/usr/local/certificates/key"
    namespaceSelector: {}
    objectSelector: {}
    labels: {}
    existingPsp: ""
    networkPolicyEnabled: false
    service:
      annotations: {}
      externalIPs: []
      loadBalancerSourceRanges: []
      servicePort: 443
      type: ClusterIP
    createSecretJob:
      resources:            #注释{}
        limits:
          cpu: 20m
          memory: 40Mi
        requests:
          cpu: 10m
          memory: 20Mi
    patchWebhookJob:
      resources: {}
    patch:
      enabled: true
      image:
        registry: registry.k8s.io       #registry.k8s.io为webhook官网镜像仓库，需要替换成自己镜像所在仓库地址
        image: ingress-nginx/kube-webhook-certgen           #webhook镜像
        tag: v1.1.1
        digest: ""
        pullPolicy: IfNotPresent
      priorityClassName: ""
      podAnnotations: {}
      nodeSelector:
        kubernetes.io/os: linux
      tolerations: []
      labels: {}
      securityContext:
        runAsNonRoot: true
        runAsUser: 2000
        fsGroup: 2000
  resources: # 设定controller的资源限制，可根据需求自定义
    requests:
      cpu: 200m
      memory: 200Mi
defaultBackend: # 设置defaultBackend
  enabled: true
  image:
```

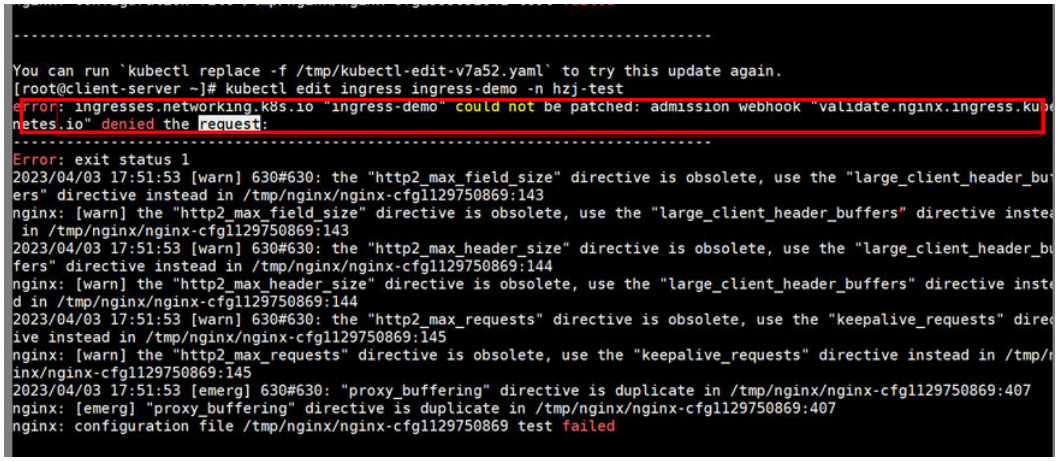
```
  repository: registry.k8s.io/defaultbackend-amd64
  registry: ""
  image: ""
  tag: "1.5"
  digest: ""
```

验证Ingress配置错误annotation场景下，admissionWebhook是否会进行校验。

例如，为Ingress配置以下错误的annotation：

```
...
 annotations:
   nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream: "false"
   nginx.ingress.kubernetes.io/auth-tls-verify-client: optional
   nginx.ingress.kubernetes.io/auth-tls-verify-depth: "1"
...
```

创建此Ingress服务，将会出现以下拦截信息：



10.10 CoreDNS 配置优化实践

10.10.1 概述

应用场景

DNS是K8s中至关重要的基础服务之一，当容器DNS策略配置不合理，集群规模较大时，DNS容易出现解析超时、解析失败等现象，极端场景下甚至会引起集群内业务大面积解析失败。本文介绍Kubernetes集群中CoreDNS配置优化的最佳实践，帮助您避免此类问题。

解决方案

CoreDNS配置优化包含客户端优化及服务端优化。

在客户端，您可以通过优化域名解析请求来降低解析延迟，通过使用合适的容器镜像、节点DNS缓存NodeLocal DNSCache等方式来减少解析异常。

- **10.10.2.1 优化域名解析请求**
- **10.10.2.2 选择合适的镜像**
- **10.10.2.3 避免IPVS缺陷导致的DNS概率性解析超时**

- **10.10.2.4 使用节点DNS缓存NodeLocal DNSCache**
- **10.10.2.5 及时升级集群中的CoreDNS版本**
- **10.10.2.6 谨慎调整VPC和虚拟机的DNS配置**

在服务端，您可以合理的调整CoreDNS部署状态或者调整CoreDNS配置来提升集群CoreDNS的可用性和吞吐量。

- **10.10.3.1 监控CoreDNS运行状态**
- **10.10.3.2 调整CoreDNS部署状态**
- **10.10.3.3 合理配置CoreDNS**

更多CoreDNS配置，详见CoreDNS官网：**https://coredns.io/**

CoreDNS开源社区地址：**https://github.com/coredns/coredns**

前提条件

- 已创建一个CCE集群，具体操作步骤请参见**快速创建Kubernetes集群**。
- 已通过kubectl连接集群，详情请参见**通过kubectl连接集群**。
- 安装CoreDNS插件（推荐安装CoreDNS最新版本），详情请参见**CoreDNS**。

10.10.2 客户端

10.10.2.1 优化域名解析请求

DNS解析是Kubernetes集群中最高频的网络行为之一，针对Kubernetes中的DNS解析的特点，您可以通过以下的方式优化域名解析请求。

客户端使用连接池

当一个容器应用需要频繁请求另一服务时，推荐使用连接池配置，连接池可以缓存上游服务的链接信息，避免每次访问都经过DNS解析和TCP重新建链的开销。

优化容器内的 resolve.conf 文件

由于resolve.conf文件中的ndots和search两个参数的作用，容器内resolve.conf文件的不同写法决定了域名解析的效率，关于ndots和search两个参数机制的详情请参考**工作负载DNS配置说明**。

优化域名的配置

当容器需要访问某域名时，请按照以下原则进行配置，可最大程度减少域名解析次数，减少域名解析耗时。

1. Pod访问同命名空间的Service，优先使用<service-name>访问，其中service-name代指Service名称。
2. Pod跨命名空间访问Service，优先使用<service-name>.<namespace-name>访问，其中namespace-name代指Service所处的命名空间。
3. Pod访问集群外部域名时，优先使用FQDN类型域名访问，这类域名通过常见域名最后加半角句号（.）的方式来指定地址，可以避免search搜索域拼接带来的多次无效搜索，例如需要访问www.huaweicoud.com，则优先使用FQDN类型域名www.huaweicoud.com.来访问。

使用本地域名缓存

集群规格较大，DNS解析请求量大的情况下可以考虑在节点上缓存DNS解析的结果，推荐使用节点DNS缓存NodeLocal DNSCache，具体使用请参考**使用NodeLocal DNSCache提升DNS性能**。

10.10.2.2 选择合适的镜像

Alpine容器镜像内置的musl libc库与标准的glibc存在以下差异：

- 3.3版本及更早版本的Alpine不支持search参数，不支持搜索域，无法完成服务发现。
- 并发请求/etc/resolve.conf中配置的多个DNS服务器，导致NodeLocal DNSCache的优化失效。
- 并发使用同一Socket请求A和AAAA记录，在旧版本内核上触发Conntrack源端口冲突导致丢包问题。
- 当使用Alpine作为容器基础镜像出现域名无法正常解析的情况下，建议更新容器基础镜像进行测试。

更多与 glibc 的功能差异问题，请参考**Functional differences from glibc**。

10.10.2.3 避免 IPVS 缺陷导致的 DNS 概率性解析超时

当集群使用IPVS作为kube-proxy负载均衡模式时，您可能会在CoreDNS缩容或重启时遇到DNS概率性解析超时的问题。该问题由社区Linux内核缺陷导致，具体信息请参见**https://github.com/torvalds/linux/commit/35dfb013149f74c2be1ff9c78f14e6a3cd1539d1**

您可以通过使用节点DNS缓存NodeLocal DNSCache降低IPVS缺陷的影响，具体操作请参见**使用NodeLocal DNSCache提升DNS性能**。

10.10.2.4 使用节点 DNS 缓存 NodeLocal DNSCache

NodeLocal DNSCache可以提升服务发现的稳定性和性能，关于NodeLocal DNSCache的介绍及如何在CCE集群中部署NodeLocal DNSCache的具体步骤，请参见**使用NodeLocal DNSCache提升DNS性能**。

10.10.2.5 及时升级集群中的 CoreDNS 版本

CoreDNS功能较为单一，对不同的Kubernetes版本也实现了较好的兼容性，CCE会定期同步社区bug，升级CoreDNS插件的版本，建议客户定期升级集群的CoreDNS版本。CCE的插件管理中心提供了CoreDNS的安装及升级功能。您可以定义关注集群中的CoreDNS版本，如果版本可以升级请尽快安排业务无缝升级集群中的CoreDNS组件。

您可以通过以下流程升级集群中的CoreDNS：

步骤1 登录CCE控制台，选择一个集群，在左侧导航栏中单击“插件管理”。

步骤2 在“已安装插件下”找到CoreDNS插件，单击“升级”按钮。

步骤3 根据页面提示填写插件安装参数，详细说明请参见**CoreDNS（系统资源插件，必装）**。

----结束

10.10.2.6 谨慎调整 VPC 和虚拟机的 DNS 配置

CoreDNS启动时会默认从部署的实例上获取resolve.conf中的DNS配置，作为上游的解析服务器地址，并且在CoreDNS重启之前不会再重新加载节点上的resolve.conf配置。建议：

- 保持集群中各个节点的resolve.conf配置一致，这样CoreDNS可以调度到集群中的任意一个节点。
- 修改集群中节点的resolve.conf文件时，如果节点有CoreDNS实例，请及时重启节点上的CoreDNS，保持状态一致。

10.10.3 服务端

10.10.3.1 监控 CoreDNS 运行状态

- CoreDNS通过标准的Promethues接口暴露出解析结果等健康指标，发现CoreDNS服务端甚至上游DNS服务器的异常。
- CoreDNS自身metrics数据接口，默认zone侦听{$POD_IP}:9153，请保持此默认值，否则普罗无法采集coredns metrics数据。
- 若您是自建Prometheus监控Kubernetes集群，可以在Prometheus观测相关指标并对以下重点指标设置告警，具体操作请参见**enables Prometheus metrics**。

10.10.3.2 调整 CoreDNS 部署状态

CCE集群默认安装CoreDNS插件，CoreDNS应用默认情况下与您的业务容器运行在同样的集群节点上，部署时的注意事项如下：

- **合理调整CoreDNS副本数**
- **合理分配CoreDNS所在位置**

- **使用自定义参数完成CoreDNS隔离部署**
- **基于HPA自动扩容CoreDNS**

合理调整 CoreDNS 副本数

建议您在任何情况下设置CoreDNS副本数应至少为2，且副本数维持在一个合适的水位以承载整个集群的解析。CCE集群安装CoreDNS插件的默认实例数为2。

- CoreDNS使用资源的规格与解析能力相关，修改CoreDNS副本数量、CPU/内存的大小会对影响解析性能，请经过评估后再做修改。
- CoreDNS插件默认配置了podAntiAffinity（Pod反亲和），当一个节点已有一个CoreDNS Pod时无法再添加新的Pod，即一个节点上只能运行一个CoreDNS Pod。如果您配置的CoreDNS副本数量大于集群节点数量，会导致多出的Pod无法调度。建议Pod的副本数量不大于节点的数量。

合理分配 CoreDNS 所在位置

- CoreDNS插件默认配置了podAntiAffinity（Pod反亲和），各个CoreDNS副本会强制部署在不同节点上。建议您将CoreDNS副本打散在不同可用区的节点上，避免单可用区故障。
- CoreDNS所运行的集群节点应避免CPU、内存用满的情况，否则会影响域名解析的QPS和响应延迟。建议您使用插件自定义参数完成**CoreDNS隔离部署**。

使用自定义参数完成 CoreDNS 隔离部署

建议CoreDNS插件与资源使用率高的负载隔离部署，防止因业务波动导致CoreDNS性能下降或不可用。您可以通过自定义参数完成CoreDNS独占节点部署。

说明

节点数应大于CoreDNS副本数，避免单个节点上运行多个CoreDNS副本。

步骤1 登录CCE控制台，进入集群，单击左侧导航栏的“节点管理”。

步骤2 切换至“节点”页签，选择CoreDNS需要独占的节点，单击“标签与污点管理”。

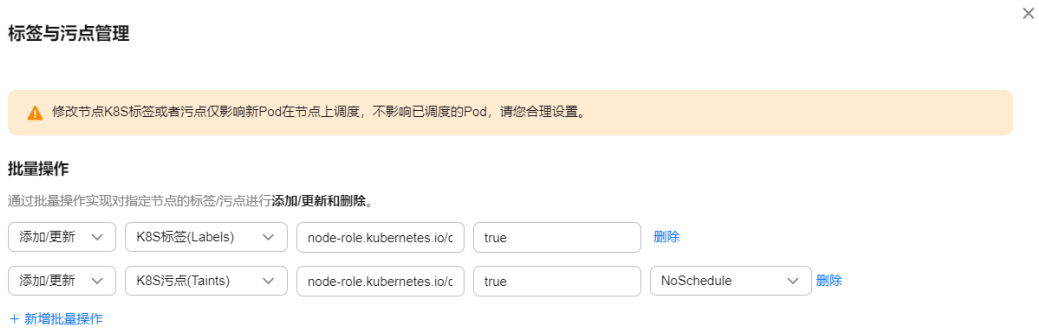
添加以下标签：

- 标签键：node-role.kubernetes.io/coredns
- 标签值：true

添加以下污点：

- 污点键：node-role.kubernetes.io/coredns
- 污点值：true
- 污点效果：NoSchedule

图 10-38 添加标签与污点

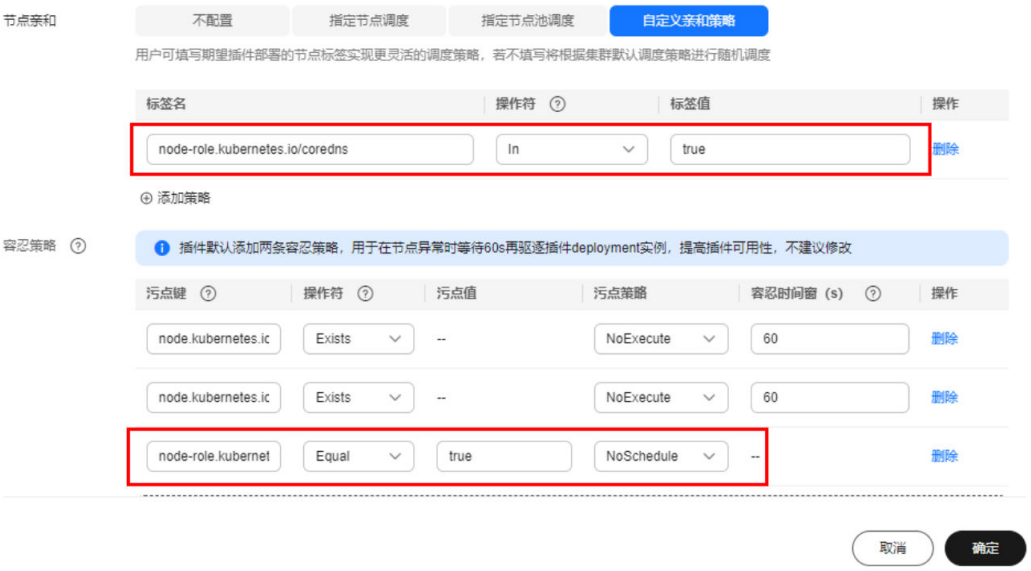


步骤3 单击左侧导航栏的“插件中心”，选择“CoreDNS域名解析”插件，单击编辑。

步骤4 在“节点亲和”中，选择“自定义亲和策略”，并添加上述节点标签。

在“容忍策略”中添加对上述污点的容忍。

图 10-39 添加容忍策略



步骤5 单击“确定”。

----结束

基于 HPA 自动扩容 CoreDNS

由于HPA会频繁触发CoreDNS副本缩容，建议不要使用容器水平扩缩容（HPA）。如果您的场景下必须依赖于HPA，建议您通过“CCE容器弹性引擎”插件配置HPA自动扩容策略，流程如下：

步骤1 登录CCE控制台，进入集群，单击左侧导航栏的“插件中心”，在右侧找到CCE容器弹性引擎，单击“安装”。

步骤2 配置插件规格后单击“安装”，插件详细说明请参考**CCE容器弹性引擎**。

步骤3 在CCE控制台单击左侧导航栏的“工作负载”，命名空间选择“kube-system”，找到CoreDNS实例，单击操作栏中的“弹性伸缩”。

在“HPA策略”中，您可以根据业务需求，通过CPU利用率、内存利用率等指标自定义HPA策略以自动扩容CoreDNS。

图 10-40 创建弹性伸缩策略

步骤4 单击“创建”，当最新状态为“已启动”时，代表HPA自动扩容CoreDNS策略生效。

----结束

10.10.3.3 合理配置 CoreDNS

CoreDNS在插件界面仅支持按预设规格配置，通常情况下，这满足绝大多数使用场景。但在一些少数对CoreDNS资源用量有要求的场景下，不能根据需要灵活配置。

CoreDNS参数自定义配置方法的具体操作请参见**自定义CoreDNS规格**。

CoreDNS官方文档：**https://coredns.io/plugins/**

合理配置 CoreDNS 规格

步骤1 登录CCE控制台，进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，单击CoreDNS插件的“编辑”按钮，进入插件详情页。

步骤3 在“规格配置”下配置CoreDNS参数规格。

步骤4 您可以根据业务需求调整不同的实例数、CPU配额和内存配额，来调整CoreDNS所能提供的域名解析QPS。

步骤5 单击“确定”，完成配置下发。

----结束

合理配置 DNS 存根域

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件中心”，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下添加存根域。格式为一个键值对，键为DNS后缀域名，值为一个或一组DNS IP地址，如 'consul.local --10.150.0.1'。

图 10-41 添加存根域

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置与密钥”，在“kube-system”命名空间下，查看名为coredns的配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    cache 30 {
        servfail 5s
    }
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
```



```
consul.local:5353 {
    bind {$POD_IP}
    errors
    cache 30
    forward . 10.150.0.1
}
```

----结束

合理配置 Host

如果您需要为特定域名指定hosts，可以使用Hosts插件来配置。示例配置如下：

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件中心”，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下编辑高级配置，在plugins字段添加以下内容。

```
{
  "configBlock": "192.168.1.1 www.example.com\nfallthrough",
  "name": "hosts"
}
```

须知

此处配置不能遗漏fallthrough字段，fallthrough表示当在hosts找不到要解析的域名时，会将解析任务传递给CoreDNS的下一个插件。如果不写fallthrough的话，任务就此结束，不会继续解析，会导致集群内部域名解析失败的情况。

hosts的详细配置请参见https://coredns.io/plugins/hosts/。

图 10-42 修改 CoreDNS Hosts 配置

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置与密钥”，在“kube-system”命名空间下，查看名为coredns的配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
```



```
    hosts {
      192.168.1.1 www.example.com
      fallthrough
    }
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
```

----结束

配置 Forward 插件与上游 DNS 服务的默认协议

步骤1 NodeLocal DNSCache采用TCP协议与CoreDNS进行通信，CoreDNS会根据请求来源使用的协议与上游DNS服务器进行通信。因此默认情况下，来自业务容器的集群外部域名解析请求会依次经过NodeLocal DNSCache、CoreDNS，最终以TCP协议请求VPC内DNS服务器。

步骤2 VPC内DNS服务器对TCP协议支持有限，如果您使用了NodeLocal DNSCache，您需要修改CoreDNS配置，让其总是优先采用UDP协议与上游DNS服务器进行通信，避免解析异常。建议您使用以下方式修改CoreDNS配置文件。

在forward插件用于设置 upstream Nameservers 上游 DNS 服务器。包含以下参数：

prefer_udp：即使请求通过TCP传入，也要首先尝试使用UDP。

若您希望CoreDNS会优先使用UDP协议与上游通信，则forward中指定请求上游的协议为prefer_udp，forward插件的详细信息请参考**https://coredns.io/plugins/forward/**。

1. 登录CCE控制台，单击集群名称进入集群。
2. 在左侧导航栏中选择“插件中心”，在CoreDNS下单击“编辑”，进入插件详情页。
3. 在“参数配置”下编辑高级配置，在plugins字段修改以下内容。

```
{
    "configBlock": "prefer_udp",
    "name": "forward",
    "parameters": ". /etc/resolv.conf"
}
```

----结束

合理配置 IPv6 的解析

如果K8s集群宿主机没有关闭IPv6内核模块的话，容器请求CoreDNS时的默认行为是同时发起IPv4和IPv6解析。由于通常只使用IPv4地址，所以此时如果仅仅在CoreDNS中配置“DOMAIN -> IPV4地址”的解析的话，当CoreDNS收到IPv6解析请求的时候就会因为本地找不到配置而forward到上游DNS服务器解析，从而导致容器的DNS解析请求变慢。

CoreDNS提供了一种plugin叫做template，经过配置后可以给所有的IPv6请求立即返回一个空结果的应答，避免请求forward到上游DNS。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下编辑高级配置，在plugins字段添加以下内容。

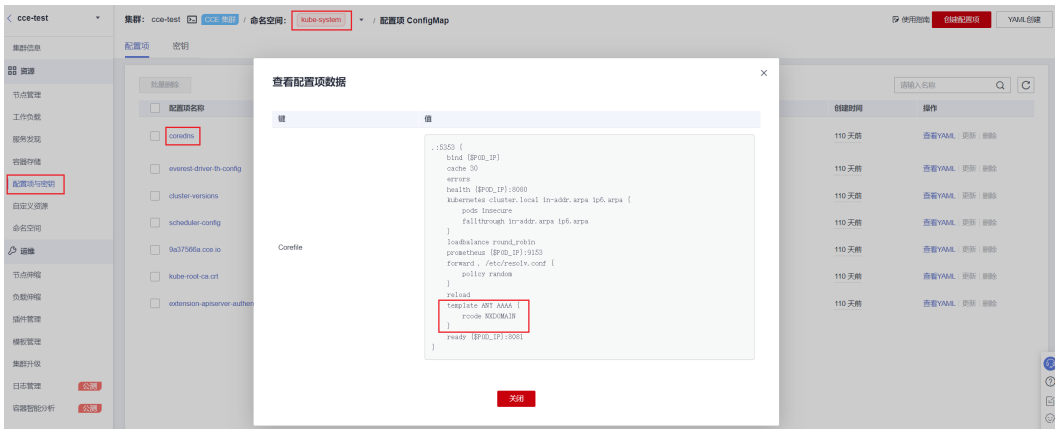
- AAAA表示IPv6解析请求，rcode控制应答返回NXDOMAIN，即表示没有解析结果。

template插件的官方文档地址请参考：**https://github.com/coredns/coredns/tree/master/plugin/template**

```
{
  "configBlock": "rcode NXDOMAIN",
  "name": "template",
  "parameters": "ANY AAAA"
}
```

步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置项与密钥”，在“kube-system”命名空间下，查看coredns配置项数据，确认是否更新成功。



对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    template ANY AAAA {
        rcode NXDOMAIN
    }
```

```
    ready {$POD_IP}:8081
}
```

----结束

合理配置缓存策略

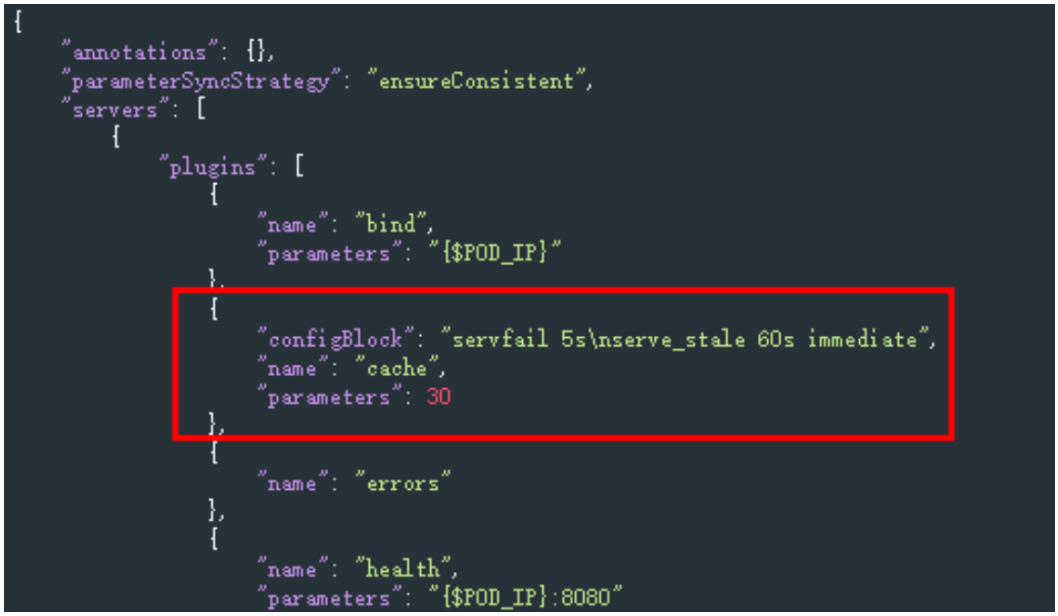
如果CoreDNS配置了上游DNS服务器时，可以通过合理的缓存策略允许CoreDNS在无法连接上游DNS服务器时使用已过期的本地缓存。

步骤1 登录CCE控制台，单击集群名称进入集群。

步骤2 在左侧导航栏中选择“插件管理”，在“已安装插件”下，在CoreDNS下单击“编辑”，进入插件详情页。

步骤3 在“参数配置”下编辑高级配置，在plugins字段修改cache内容。cache的详细配置请参见**https://coredns.io/plugins/cache/**。

```
{
    "configBlock": "servfail 5s\nserve_stale 60s immediate",
    "name": "cache",
    "parameters": 30
}
```



步骤4 单击“确定”完成配置更新。

步骤5 在左侧导航栏中选择“配置与密钥”，在“kube-system”命名空间下，查看名为coredns的配置项数据，确认是否更新成功。

对应Corefile内容如下：

```
.:5353 {
    bind {$POD_IP}
    cache 30 {
        servfail 5s
        serve_stale 60s immediate
    }
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
```

```
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
```

----结束

10.11 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 10-10 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。 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。	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。 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。 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。	建议配置为大部分节点平时最多运行的Pod数。
节点动态预热容器网卡数(nic-warm-target)	2	保障节点至少预热的容器网卡数，只支持数值配置。 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。 ● 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**。 ● 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c7.4xlarge.2机型（辅助弹性网卡配额128） 集群下大部分节点平时日常运行20个Pod左右 集群下大部分节点最多运行60个Pod 集群下大部分节点日常10s内会瞬时弹性扩容10个Pod 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	集群级别的全局配置： • nic-minimum-target: 20 • nic-maximum-target: 60 • nic-warm-target: 10 • 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） 节点池下大部分节点平时日常运行100个Pod左右 节点池下大部分节点最多运行128个Pod 节点池下大部分节点日常在10s内会瞬时弹性扩容10个Pod 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： • nic-minimum-target: 100 • nic-maximum-target: 120 • nic-warm-target: 10 • 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 ··· ，选择“配置管理”。

图 10-43 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“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”，并开启节点池网卡预热参数配置开关，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

10.12 集群通过企业路由器连接对端 VPC

应用场景

企业路由器（Enterprise Router, ER）可以连接虚拟私有云（Virtual Private Cloud, VPC）或本地网络来构建中心辐射型组网，实现同区域的VPC互通，是云上大规格、高带宽、高性能的集中路由器。借助企业路由器的能力，可以实现不同VPC下CCE集群互通。

通过ER连接对端VPC，可以解决不同VPC下的集群创建容器之后短期内无法和对端VPC虚拟机互通的问题。在CCE Turbo集群中，您还可以使用延迟启动Pod的方案解决该问题，详情请参见**12.11 在CCE Turbo集群中配置Pod延时启动参数**。

规划组网

在VPC通过ER连接之前，需要规划VPC的子网网段及ER路由表信息。需要满足如下的要求：

资源	说明
VPC	• VPC网段（CIDR）不能重叠。 • ER路由表使用的是“虚拟私有云（VPC）”连接的传播路由，由ER自动学习VPC网段作为目的地址，不支持修改，因此重叠的VPC网段会导致路由冲突。同时容器网段也不可与对端VPC的节点网段冲突，否则也会造成网络不通。 • 如果已有VPC网段重叠，则需要在ER路由表中手动添加静态路由，目的地址可以为VPC子网网段或者范围更小的网段。
ER	开启“默认路由表关联”和“默认路由表传播”功能，添加完“虚拟私有云（VPC）”连接，系统会自动执行以下配置： • 将虚拟私有云连接关联至ER默认路由器。 • 在默认路由表中创建“虚拟私有云（VPC）”连接的传播，路由自动学习VPC网段。

具体规划组网和资源指南请参考**组网规划和资源指南**。

创建企业路由器

步骤1 登录管理控制台。

步骤2 单击管理控制台左上角的 ，选择区域和项目。

步骤3 单击“服务列表”，选择“网络 > 企业路由器”。

步骤4 进入企业路由器主页面。

步骤5 单击页面右上角的“创建企业路由器”。

步骤6 进入“创建企业路由器”页面。

步骤7 根据界面提示，配置企业路由器的基本信息，配置参数说明请参见**表10-11**。

图 10-44 创建企业路由器

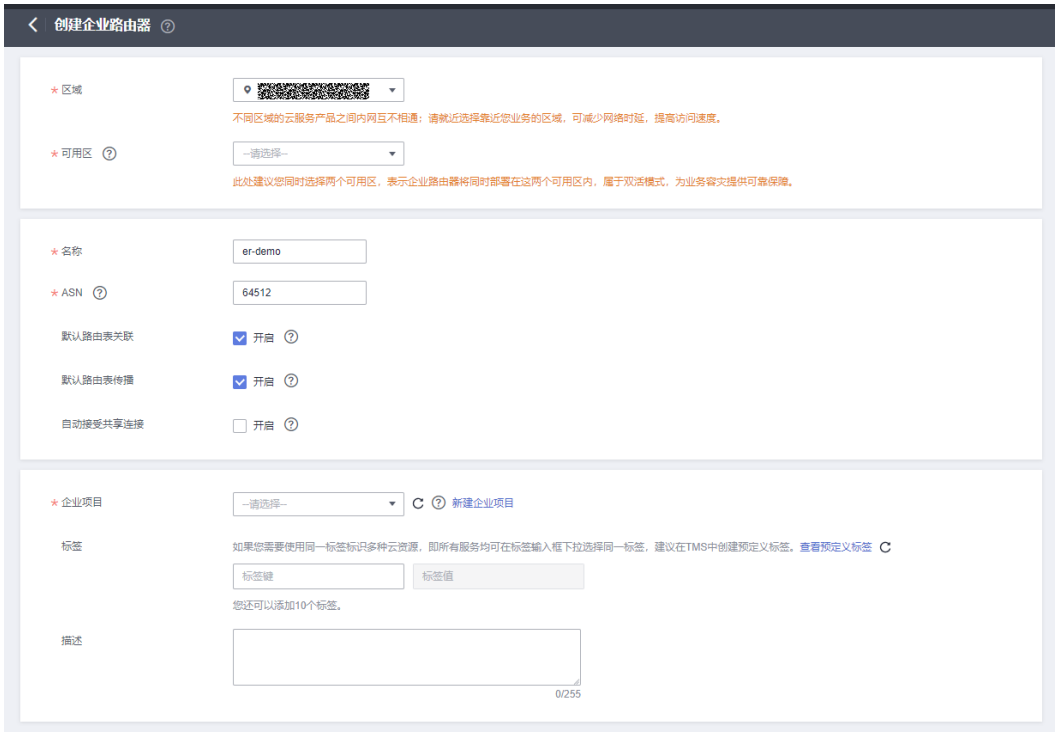


表 10-11 创建企业路由器-参数说明

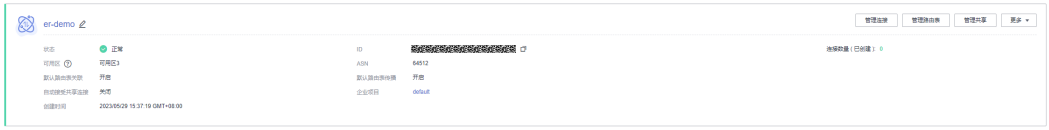
参数名称	参数说明	取值样例
区域	选择靠近业务的区域，不可修改。	中国-香港
可用区	选择两个可用区部署企业路由器。	可用区1 可用区2
名称	企业路由器名称，支持修改。	er-test-01
ASN	根据网络规划指定自治系统编号，不支持修改。	64512
默认路由表关联	开启“默认路由表关联”功能，可以免去创建路由表、创建关联等操作，支持修改。	开启
默认路由表传播	开启“默认路由表传播”功能，可免去创建路由表、创建传播、添加路由等操作，支持修改。	开启
自动接受共享连接	关闭“自动接受共享连接”功能，接受者创建的连接需要所有者审批，所有者接受后才会创建。	关闭
企业项目	将企业路由器加入已有的企业项目内，支持修改。	default
标签	为企业路由器绑定标签，用来标识资源，支持修改。	“标签键”：test “标签值”：01
描述	该企业路由器的描述信息，支持修改。	-

步骤8 等待基本信息设置完成后，单击“立即创建”。

步骤9 在产品配置信息确认页面，再次核对企业路由器信息，确认无误后，单击“提交”。返回企业路由器列表页面。

步骤10 在企业路由器列表页面，查看企业路由器状态。待状态由“创建中”变为“正常”，表示企业路由器创建完成。

图 10-45 ER 创建完成



----结束

在企业路由器中添加 VPC 连接

步骤1 登录管理控制台。

步骤2 单击管理控制台左上角的，选择区域和项目。

步骤3 单击“服务列表”，选择“网络 > 企业路由器”。 进入企业路由器主页面。

步骤4 通过名称过滤，快速找到待添加连接的企业路由器。

步骤5 您可以通过以下两种操作入口，进入企业路由器的“连接”页签。

- 在企业路由器右上角区域，单击“管理连接”。
- 单击企业路由器名称，并选择“连接”页签。详情可参见**在企业路由器中添加VPC连接**。

步骤6 在“连接”页签下，单击“添加连接”。弹出“添加连接”对话框。

步骤7 根据界面提示，配置连接的基本信息，如**表10-12**所示。

图 10-46 连接创建页面

表 10-12 添加连接-参数说明

参数名称	参数说明	取值样例
名称	“虚拟私有云（VPC）”连接的名称，支持修改。	er-attach-01
连接类型	选择“虚拟私有云（VPC）”，不支持修改。	虚拟私有云（VPC）
连接资源	1. 连接类型选择完成后，在下拉列表中选择待接入企业路由器的虚拟私有云，不支持修改。 2. 虚拟私有云选择完成后，在下拉列表中选择待接入企业路由器的子网，不支持修改。	• 虚拟私有云：vpc-demo-01 • 子网：subnet-demo-01
配置连接侧路由	• 如果您在创建连接时开启“配置连接侧路由”选项，则不用手动在VPC路由表中配置静态路由，系统会在VPC的所有路由表中自动添加指向ER的路由，目的地址固定为10.0.0.0/8，172.16.0.0/12，192.168.0.0/16。 • 如果VPC路由表中的路由与这三个固定网段冲突，则会添加失败。此时建议您不要开启“配置连接侧路由”选项，并在连接创建完成后，手动添加路由。 • 不建议在VPC路由表中将ER的路由配置为默认路由网段0.0.0.0/0，如果VPC内的ECS绑定了EIP，会在ECS内增加默认网段的策略路由，并且优先级高于ER路由，此时会导致流量转发至EIP，无法抵达ER。	开启
描述	该连接的描述信息，支持修改。	-
标签	为连接绑定标签，用来标识资源，支持修改。	“标签键”：test “标签值”：01

步骤8 设置完成之后，等待连接列表从“创建中”变为“正常”，表示连接创建成功。重复执行上述的**步骤6**-**步骤7**步，完成其它VPC的连接。

图 10-47 ER 连接示意图

----结束

验证网络连通情况

步骤1 登录CCE控制台，查找已经添加在VPC下CCE集群。

步骤2 单击CCE集群名称进入集群，单击左侧导航栏的“节点管理”，查看节点的IP。

图 10-48 CCE 侧查看节点 IP

步骤3 登录节点，详情请参见**登录节点**。本示例通过管理控制台远程登录（VNC方式）。

步骤4 在弹性云服务器的远程登录窗口，执行以下命令，验证网络互通情况。

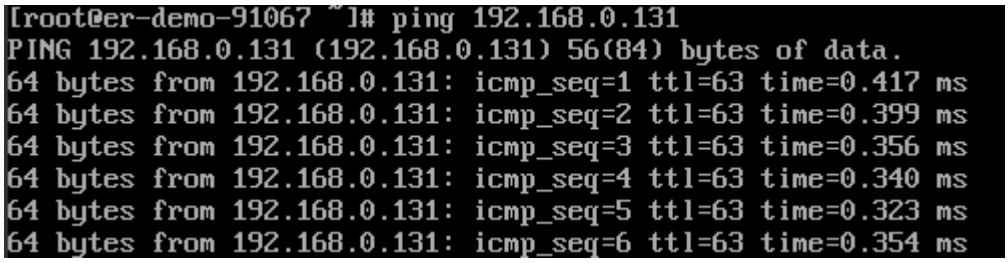
```
ping {弹性服务器地址}
```

以vpc-ER-demo2这个VPC下的集群为例，登录节点er-demo2-04260，访问vpc-ER-demo1这个VPC下集群的节点er-demo1-61379，该节点IP为192.168.0.131。

```
ping 192.168.0.131
```

回显如下，表示网络已通。

图 10-49 网络回显结果

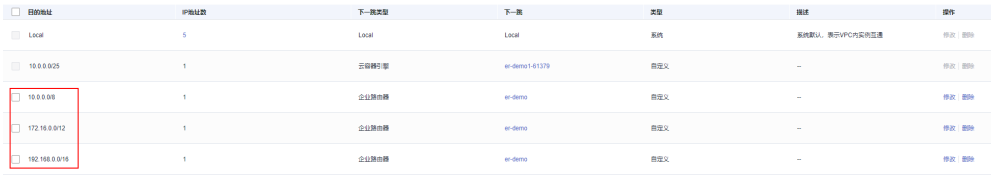


步骤5 重复上述过程，依次确认节点的连通性。

如果存在ping对端VPC的节点不通，可能存在如下情况，请依次确认：

1. 集群节点的安全组规则是否放通ICMP协议。
2. VPC的路由表是否存在网段冲突。请额外注意，容器网段也不可与默认的企业路由器网段冲突，详情请参见**规划组网**。

图 10-50 VPC 路由表



----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 • 数据盘扩容——Kubelet空间 • 数据盘扩容——容器引擎和Kubelet共享磁盘空间
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
  vda               8:0    0   50G  0 disk
  └─vda1              8:1    0   50G  0 part /
  vdb               8:16   0  200G  0 disk
  ├─vgpaas-dockersys 253:0    0   90G  0 lvm  /var/lib/docker              # 容器引擎使用的空间
  └─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
  ```

  在节点上执行如下命令， 将新增的磁盘容量加到dockersys盘上。

  ```
  pvresize /dev/vdb
  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
  vda                       8:0    0   50G  0 disk
  ```

```
└─vda1                          8:1    0   50G  0 part /
vdb                           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/vdb
  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/vdb
  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 使用**lsblk**命令查看节点块设备信息。

```
# lsblk
NAME                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                 8:0    0   50G  0 disk
└─vda1                8:1    0   50G  0 part /
vdb                 8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker             # 容器引擎使用的空间
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # kubernetes使用的空间
```

步骤5 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/vdb
lvextend -l+100%FREE -n vgpaas/kubernetes
resize2fs /dev/vgpaas/kubernetes
```

----结束

数据盘扩容——容器引擎和 Kubelet 共享磁盘空间

v1.21.10-r0、v1.23.8-r0、v1.25.3-r0及之后版本的集群中，CCE使用的数据盘支持采用共享数据盘的方式，即不再划分容器引擎 (Docker/Containerd) 和Kubelet组件的空间。您可参考以下步骤进行扩容。

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。

```
# lsblk
NAME          MAJ:MIN RM SIZE RO TYPE MOUNTPOINTS
vda           253:0    0  50G  0 disk
└─vda1        253:1    0  50G  0 part /
vdb           253:16   0  60G  0 disk
└─vgpaas-share 252:0    0  60G  0 lvm  /mnt/paas # 容器引擎和Kubelet组件共用的空间
```

步骤5 然后在节点上执行如下命令， 将新增的磁盘容量加到共享盘上。

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/share
resize2fs /dev/vgpaas/share
```

----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 切换至“节点”页签，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：

Pod容器空间分配：即容器的basesize设置，每个工作负载下的容器组 Pod 占用的磁盘空间设置上限（包含容器镜像占用的空间）。合理的配置可避免容器组无节制使用磁盘空间导致业务异常。建议此值不超过容器引擎空间的 80%。该参数与节点操作系统和容器存储Rootfs相关，部分场景下不支持设置。

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“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。

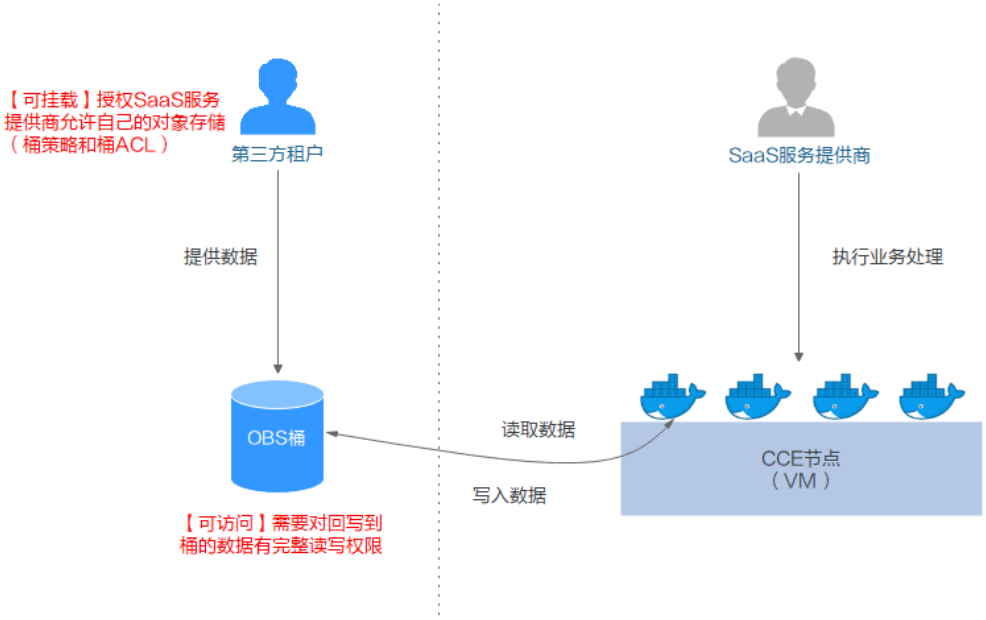
11.2 挂载第三方租户的对象存储

本章节介绍如何挂载第三方租户的OBS桶，包含OBS并行文件系统（优先）和OBS对象桶。

使用场景

SaaS服务提供商的CCE集群需要挂载使用第三方租户的OBS桶，使用场景如图11-1所示。

图 11-1 挂载第三方租户的对象存储使用场景



1. **第三方租户授权SaaS服务提供商访问所拥有的对象存储**，第三方租户对SaaS服务提供商需要访问的OBS对象桶或者并行文件系统设置桶策略和桶ACL。

2. **SaaS服务提供商静态导入第三方OBS对象桶和并行文件系统**。
3. SaaS服务提供商进行业务处理，最终需要将处理结果（结果文件或者结果数据）写回第三方租户的桶。

使用须知

- 仅支持挂载同一区域下的第三方租户的并行文件系统和对象桶。
- 仅已安装1.1.11及以上版本Everest存储插件的集群（集群版本需v1.15及以上），支持挂载第三方租户的OBS桶。
- CCE集群所在的SaaS服务提供商的业务平台需要管理第三方桶PV的生命周期，单独删除PVC时PV不会被关联删除而被保留（Kubernetes原生PV Retain回收策略的效果），需要调用Kubernetes原生接口进行静态PV的创建和删除。

第三方租户授权 SaaS 服务提供商访问所拥有的对象存储

以授权访问OBS对象桶为例介绍如何设置桶策略和桶ACL，OBS并行文件系统的设置方法和对象桶相同。

步骤1 登录OBS控制台。

步骤2 在桶列表单击待操作的桶，进入“概览”页面。

步骤3 在左侧导航栏，单击“访问权限控制 > 桶策略”，在右侧单击“创建”。

图 11-2 创建桶策略

- 效力：设置为“允许”。
- 被授权用户：选择其他账号，输入授权用户的账号ID和用户ID，桶策略对指定的用户生效。
- 授权资源：选择允许操作的资源。
- 授权操作：勾选可以操作的动作。

步骤4 在左侧导航栏，单击“访问权限控制 > 桶ACLs”，在右侧单击“增加”，输入授权用户的账号ID，桶访问权限勾选“读取权限”、“对象读权限”和“写入权限”，ACL访问权限勾选“读取权限”和“写入权限”，单击“保存”。

----结束

SaaS 服务提供商静态导入第三方 OBS 对象桶和并行文件系统

- **OBS对象桶静态PV：**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: objbucket      #名称替换为实际的对象桶PV名称
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  mountOptions:
  - default_acl=bucket-owner-full-control      #新增的OBS挂载参数
  csi:
    driver: obs.csi.everest.io
    fsType: s3fs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region: ap-southeast-1       #设置为当前区域id
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: objbucket            #名称替换为实际的第三方租户对象桶名称
  persistentVolumeReclaimPolicy: Retain     #必须设置为Retain，删除PV时保留桶实际也无法删除
  storageClassName: csi-obs-mountoption     #可以关联某个新增自定义OBS StorageClass, 也可关联集群自带的csi-obs
```

  - mountOptions：新增的OBS挂载参数，允许桶所有者对桶中数据有完整的访问权限，解决挂载第三方桶后写入数据、桶所有者无法读取的问题。挂载第三方租户的对象存储使用场景下，default_acl必须设置为**bucket-owner-full-control**，default_acl的其他取值请参见**OBS预定义的权限控制策略**。
  - persistentVolumeReclaimPolicy：挂载第三方租户的对象存储使用场景下，persistentVolumeReclaimPolicy必须设置为Retain。删除PV时保留桶实际也无法删除，因此CCE集群所在的SaaS服务提供商的业务平台需要管理第三方桶PV的生命周期，单独删除PVC时PV不会被关联删除而被保留（Kubernetes原生PV Retain回收策略的效果），需要调用Kubernetes原生接口进行静态PV的创建和删除。
  - storageClassName：可以关联某个新增自定义OBS StorageClass（**创建自定义OBS StorageClass**），也可关联集群自带的csi-obs。

  绑定的OBS对象桶PVC：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/fstype: obsfs
    everest.io/obs-volume-type: STANDARD
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: objbucketpvc       #名称替换为实际的对象桶PVC名称
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs-mountoption     #与绑定的PV关联的StorageClass保持一致
  volumeName: objbucket       #名称替换为实际需要绑定的对象桶PV名称
```

- **OBS并行文件系统静态PV：**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: obsfscheck   #名称替换为实际的并行文件系统PV名称
  annotations:
```

```
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  mountOptions:
  - default_acl=bucket-owner-full-control     #新增的OBS挂载参数
  csi:
    driver: obs.csi.everest.io
    fsType: obsfs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region:  ap-southeast-1
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: obsfscheck                #名称替换为实际的三方租户的并行文件系统名称
  persistentVolumeReclaimPolicy: Retain       #必须设置为Retain，删除PV时保留桶实际也无法删除
  storageClassName: csi-obs-mountoption       #可以关联某个新增自定义OBS StorageClass，也可关联集群自带的csi-obs
```

- mountOptions：新增的OBS挂载参数，允许桶所有者对桶中数据有完整的访问权限，解决挂载第三方桶后写入数据、桶所有者无法读取的问题。挂载第三方租户的对象存储使用场景下，default_acl必须设置为**bucket-owner-full-control**，default_acl的其他取值请参见**OBS预定义的权限控制策略**。
- persistentVolumeReclaimPolicy：挂载第三方租户的对象存储使用场景下，persistentVolumeReclaimPolicy必须设置为Retain。删除PV时保留桶实际也无法删除，因此CCE集群所在的SaaS服务提供商的业务平台需要管理第三方桶PV的生命周期，单独删除PVC时PV不会被关联删除而被保留（Kubernetes原生PV Retain回收策略的效果），需要调用Kubernetes原生接口进行静态PV的创建和删除。
- storageClassName：可以关联某个新增自定义OBS StorageClass（**创建自定义OBS StorageClass**），也可关联集群自带的csi-obs。

绑定的OBS并行文件系统PVC：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/fstype: obsfs
    everest.io/obs-volume-type: STANDARD
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: obsfscheckpvc     #名称替换为实际的并行文件系统PVC名称
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs-mountoption    #与绑定的PV关联的StorageClass保持一致
  volumeName: obsfscheck     #名称替换为实际的并行文件系统PV名称
```

- **创建自定义OBS StorageClass，用以关联静态PV（可选）：**

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-obs-mountoption
mountOptions:
  - default_acl=bucket-owner-full-control
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: obsfs
  everest.io/obs-volume-type: STANDARD
provisioner: everest-csi-provisioner
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

- csi.storage.k8s.io/fstype：指定文件类型，支持“obsfs”与“s3fs”。取值为s3fs时说明使用的obs对象桶，配套使用s3fs挂载；取值为obsfs时说明使用的是obs并行文件系统，配套使用obsfs挂载。
- reclaimPolicy：指定创建的Persistent Volume的回收策略。该取值会设置到基于新的PVC关联该SC动态创建的PV.spec.persistentVolumeReclaimPolicy中。若设置为Delete，删除PVC时会触发关联删除外部OBS对象和该PV；若设置为Retain，删除PVC时会保留PV和外部存储，需要单独清理PV。对于使用导入关联三方桶对应的使用场景，该SC只做关联静态PV使用（**需要设置为Retain**），不涉及使用动态创建。

11.3 通过 StorageClass 动态创建 SFS Turbo 子目录

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。
- subpath类型的卷实际为SFS Turbo的子目录，对该类型的PVC进行扩容仅会调整PVC声明的资源范围，并不会调整SFS Turbo资源的总容量。若SFS Turbo资源总容量不足，subpath类型卷的实际可使用的容量大小也会受限，您需要前往SFS Turbo界面进行扩容。

  同理，删除subpath类型的卷也不会实际删除后端的SFS Turbo资源。

创建 subpath 类型 SFS Turbo 存储卷

步骤1 创建SFS Turbo资源，选择网络时，请选择与集群相同的VPC与子网。

步骤2 新建一个StorageClass的YAML文件，例如sfsturbo-subpath-sc.yaml。

配置示例：

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  name: sfsturbo-subpath-sc
mountOptions:
- lock
parameters:
  csi.storage.k8s.io/csi-driver-name: sfsturbo.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/archive-on-delete: "true"
  everest.io/share-access-to: 7ca2dba2-1234-1234-1234-626371a8fb3a
```



```
  everest.io/share-expand-type: bandwidth
  everest.io/share-export-location: 192.168.1.1:/sfsturbo/
  everest.io/share-source: sfs-turbo
  everest.io/share-volume-type: STANDARD
  everest.io/volume-as: subpath
  everest.io/volume-id: 0d773f2e-1234-1234-1234-de6a35074696
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

其中：

- name：storageclass的名称。
- mountOptions：选填字段；mount挂载参数。
  - everest 1.2.8以下，1.1.13以上版本仅开放对nolock参数配置，mount操作默认使用nolock参数，无需配置。nolock=false时，使用lock参数。
  - everest 1.2.8及以上版本支持更多参数，默认使用如下所示配置，具体请参见**设置挂载参数**。**此处不能配置为nolock=true，会导致挂载失败。**

    ```
    mountOptions:
    - vers=3
    - timeo=600
    - nolock
    - hard
    ```
- everest.io/volume-as：该参数需设置为“subpath”来使用subpath模式。
- everest.io/share-access-to：选填字段。subpath模式下，填写SFS Turbo资源的所在VPC的ID。
- everest.io/share-expand-type：选填字段。若SFS Turbo资源存储类型为增强版（标准型增强版、性能型增强版），设置为bandwidth。
- everest.io/share-export-location：挂载目录配置。由SFS Turbo共享路径和子目录组成，共享路径可至SFS Turbo服务页面查询，子路径由用户自定义，后续指定该StorageClass创建的PVC均位于该子目录下。
- everest.io/share-volume-type：选填字段。填写SFS Turbo的类型。标准型为STANDARD，性能型为PERFORMANCE。对于增强型需配合“everest.io/share-expand-type”字段使用，everest.io/share-expand-type设置为“bandwidth”。
- everest.io/zone：选填字段。指定SFS Turbo资源所在的可用区。
- everest.io/volume-id：SFS Turbo资源的卷ID，可至SFS Turbo界面查询。
- everest.io/archive-on-delete：若该参数设置为“true”，在回收策略为“Delete”时，删除PVC会将PV的原文档进行归档，归档目录的命名规则“archived-$pv名称.时间戳”。该参数设置为“false”时，会将PV对应的SFS Turbo子目录删除。默认设置为“true”，即删除PVC时进行归档。

步骤3 执行**kubectl create -f sfsturbo-subpath-sc.yaml**。

步骤4 新建一个PVC的YAML文件，sfs-turbo-test.yaml。

配置示例：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sfs-turbo-test
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
```



```
    storage: 50Gi
  storageClassName: sfsturbo-subpath-sc
  volumeMode: Filesystem
```

其中：

- name: PVC的名称。
- storageClassName：SC的名称。
- storage：subpath模式下，调整该参数的大小不会对SFS Turbo容量进行调整。实际上，subpath类型的卷是SFS Turbo中的一个文件路径，因此在PVC中对subpath类型的卷扩容时，不会同时扩容SFS Turbo资源。

  说明

  subpath子目录的容量受限于SFS Turbo资源的总容量，若SFS Turbo资源总容量不足，请您及时到SFS Turbo界面调整。

步骤5 执行**kubectl create -f sfs-turbo-test.yaml**。

----结束

创建 Deployment 挂载已有数据卷

步骤1 新建一个Deployment的YAML文件，例如deployment-test.yaml。

配置示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-turbo-subpath-example
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-turbo-subpath-example
  template:
    metadata:
      labels:
        app: test-turbo-subpath-example
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        volumeMounts:
        - mountPath: /tmp
          name: pvc-sfs-turbo-example
      restartPolicy: Always
      imagePullSecrets:
      - name: default-secret
      volumes:
      - name: pvc-sfs-turbo-example
        persistentVolumeClaim:
          claimName: sfs-turbo-test
```

其中：

- name：创建的工作负载名称。
- image：工作负载的镜像。

- mountPath：容器内挂载路径，示例中挂载到“/tmp”路径。
- claimName：已有的PVC名称。

步骤2 创建Deployment负载。

kubectl create -f deployment-test.yaml

----结束

StatefulSet 动态创建 subpath 模式的数据卷

步骤1 新建一个StatefulSet的YAML文件，例如statefulset-test.yaml。

配置示例：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: test-turbo-subpath
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test-turbo-subpath
  template:
    metadata:
      labels:
        app: test-turbo-subpath
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
        pod.alpha.kubernetes.io/initialized: 'true'
    spec:
      containers:
        - name: container-0
          image: 'nginx:latest'
          resources: {}
          volumeMounts:
            - name: sfs-turbo-160024548582479676
              mountPath: /tmp
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: default-secret
      affinity: {}
      schedulerName: default-scheduler
  volumeClaimTemplates:
    - metadata:
        name: sfs-turbo-160024548582479676
        namespace: default
        annotations: {}
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        storageClassName: sfsturbo-subpath-sc
  serviceName: wwww
```



```
podManagementPolicy: OrderedReady
updateStrategy:
  type: RollingUpdate
revisionHistoryLimit: 10
```

其中：

- name：创建的工作负载名称。
- image：工作负载的镜像。
- mountPath：容器内挂载路径，示例中挂载到“/tmp”路径。
- “spec.template.spec.containers.volumeMounts.name ”和“spec.volumeClaimTemplates.metadata.name”有映射关系，必须保持一致。
- storageClassName: 填写自建的SC名称。

步骤2 创建StatefulSet负载。

kubectl create -f statefulset-test.yaml

----结束

11.4 1.15 集群如何从 Flexvolume 存储类型迁移到 CSI Everest 存储类型

在v1.15.11-r1之后版本的集群中，CSI Everest插件已接管fuxi Flexvolume（即storage-driver插件）容器存储的所有功能，建议将对fuxi Flexvolume的使用切换CSI Everest上。

迁移的主要原理是通过创建静态PV的形式关联原有底层存储，并创建新的PVC关联该新建的静态PV，之后应用升级挂载这个新的PVC到原有挂载路径，实现存储卷迁移。

⚠ 警告

迁移时会造成服务断服，请合理规划迁移时间，并做好相关备份。

操作步骤

步骤1 数据备份（可选，主要防止异常情况下数据丢失）。

步骤2 根据FlexVolume格式的PV，准备CSI格式的PV的yaml文件关联已有存储。

执行如下命令，配置名为“pv-example.yaml”的创建PV的yaml文件。

touch pv-example.yaml

vi pv-example.yaml

云硬盘存储卷PV的配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    failure-domain.beta.kubernetes.io/region: ap-southeast-1
    failure-domain.beta.kubernetes.io/zone: <zone name>
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
```

```
  name: pv-evs-example
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 10Gi
  csi:
    driver: disk.csi.everest.io
    fsType: ext4
    volumeAttributes:
      everest.io/disk-mode: SCSI
      everest.io/disk-volume-type: SAS
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 0992dbda-6340-470e-a74e-4f0db288ed82
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-disk
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-2 云硬盘存储卷 PV 配置参数说明

参数	描述
failure-domain.beta.kubernetes.io/region	云硬盘所在region，可参考FlexVolume PV的相同字段。
failure-domain.beta.kubernetes.io/zone	云硬盘所在可用区，可参考FlexVolume PV的相同字段。
name	PV资源的名称，集群下唯一。
storage	云硬盘的容量，单位为Gi。可参考FlexVolume PV的spec.capacity.storage。
driver	挂载依赖的存储驱动，EVS云硬盘配置为“disk.csi.everest.io”。
volumeHandle	云硬盘的volumeID，可参考FlexVolume PV的spec.flexVolume.options.volumeID。
everest.io/disk-mode	云硬盘磁盘模式，可参考FlexVolume PV的spec.flexVolume.options.disk-mode。
everest.io/disk-volume-type	云硬盘类型，当前支持高I/O（SAS）、超高I/O（SSD）。可参考FlexVolume PV的spec.storageClassName对应的sc中的parameters."kubernetes.io/volumetype"。
storageClassName	存储卷动态供应关联的K8s storage class名称；云硬盘需使用“csi-disk”。

文件存储卷PV配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-sfs-example
  annotations:
```

```
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: nas.csi.everest.io
    fsType: nfs
    volumeAttributes:
      everest.io/share-export-location: sfs-nas01.ap-southeast-1.myhuaweicloud.com:/share-436304e8
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 682f00bb-ace0-41d8-9b3e-913c9aa6b695
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-nas
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-3 文件存储卷 PV 配置参数说明

参数	描述
name	PV资源的名称，集群下唯一。
storage	文件存储的大小，单位为Gi。可参考FlexVolume PV的spec.capacity.storage。
driver	挂载依赖的存储驱动，文件存储配置为“nas.csi.everest.io”。
everest.io/share-export-location	文件存储的共享路径。可参考FlexVolume PV的spec.flexVolume.options.deviceMountPath。
volumeHandle	文件存储的ID。可参考FlexVolume PV的spec.flexVolume.options.volumeID。
storageClassName	K8s storage class名称；需配置为"csi-nas"。

对象存储卷PV配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-obs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  csi:
    driver: obs.csi.everest.io
    fsType: s3fs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region: ap-southeast-1
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: obs-normal-static-pv
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-obs
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-4 对象存储卷 PV 配置参数说明

参数	描述
name	PV资源的名称，集群下唯一。
storage	存储容量，单位为Gi。此处配置为固定值1Gi。
driver	挂载依赖的存储驱动，对象存储配置为“obs.csi.everest.io”。
fsType	文件类型，支持“obsfs”与“s3fs”，取值为s3fs时创建是obs对象桶，配套使用s3fs挂载；取值为obsfs时创建的是obs并行文件系统，配套使用obsfs挂载。可参考FlexVolume PV的spec.flexVolume.options.posix的对应关系：true（obsfs）、false/空值（s3fs）。
everest.io/obs-volume-type	存储类型，包括STANDARD（标准桶）、WARM（低频访问桶）。可参考FlexVolume PV的spec.flexVolume.options.storage_class的对应关系：standard（标准桶）、standard_ia（低频访问桶）。
everest.io/region	对象存储所在的region。可参考FlexVolume PV的spec.flexVolume.options.region。
volumeHandle	对象存储的桶名称。可参考FlexVolume PV的spec.flexVolume.options.volumeID。
storageClassName	K8s storage class名称；需配置为"csi-obs”。

极速文件存储卷PV配置示例如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-efs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: sfsturbo.csi.everest.io
    fsType: nfs
    volumeAttributes:
      everest.io/share-export-location: 192.168.0.169:/
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 8962a2a2-a583-4b7f-bb74-fe76712d8414
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-sfsturbo
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-5 极速文件存储卷 PV 配置参数说明

参数	描述
name	PV资源的名称，集群下唯一。
storage	文件存储的大小。可参考FlexVolume PV的spec.capacity.storage。
driver	挂载依赖的存储驱动，极速文件存储配置为“sfsturbo.csi.everest.io”。
everest.io/share-export-location	极速文件存储的共享路径。可参考FlexVolume PV的spec.flexVolume.options.deviceMountPath。
volumeHandle	极速文件存储的ID。可参考FlexVolume PV的spec.flexVolume.options.volumeID。
storageClassName	指定K8s storage class名称；极速文件存储卷需配置为"csi-sfsturbo”。

步骤3 根据FlexVolume格式的PVC，准备CSI格式的PVC的yaml文件关联上述步骤准备的静态PV。

执行如下命令，配置名为“pvc-example.yaml”的创建PVC的yaml文件。

touch pvc-example.yaml

vi pvc-example.yaml

云硬盘存储卷PVC的配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    failure-domain.beta.kubernetes.io/region: ap-southeast-1
    failure-domain.beta.kubernetes.io/zone: <zone name>
  annotations:
    everest.io/disk-volume-type: SAS
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-evs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeName:  pv-evs-example
  storageClassName: csi-disk
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-6 云硬盘存储卷 PVC 配置参数说明

参数	描述
failure-domain.beta.kubernetes.io/region	集群所在region。可参考FlexVolume PVC的相同字段。
failure-domain.beta.kubernetes.io/zone	EVS云硬盘所在可用区。可参考FlexVolume PVC的相同字段。
everest.io/disk-volume-type	云硬盘存储类型，支持SAS、SSD。和上述步骤的PV保持一致。
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storage	PVC申请容量，必须和已有PV的storage大小保持一致。
volumeName	PV的名称。使用上述步骤的静态PV的名称。
storageClassName	指定K8s storage class名称；云硬盘需使用“csi-disk”。

文件存储卷PVC配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-sfs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas
  volumeName: pv-sfs-example
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-7 文件存储卷 PVC 配置参数说明

参数	描述
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storage	存储容量，单位Gi，必须和已有pv的storage大小保持一致。

参数	描述
storageClassName	需配置为"csi-nas"。
volumeName	PV的名称。参考上述步骤的静态PV的名称。

对象存储卷PVC配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/obs-volume-type: STANDARD
    csi.storage.k8s.io/fstype: s3fs
  name: pvc-obs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs
  volumeName: pv-obs-example
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-8 对象存储卷 PVC 配置参数说明

参数	描述
everest.io/obs-volume-type	obs存储类型；当前支持标准（STANDARD）和低频（WARM）两种存储类型。和上述步骤的PV保持一致。
csi.storage.k8s.io/fstype	指定文件类型，支持“obsfs”与“s3fs”。与上述步骤中静态obs存储的PV的fsType保持一致。
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storage	存储容量，单位为Gi。此处配置为固定值1Gi。
storageClassName	K8s storage class名称；需配置为"csi-obs”。
volumeName	PV的名称。参考上述步骤创建的静态PV的名称。

极速文件存储卷PVC配置示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
```



```
  name: pvc-efs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-sfsturbo
  volumeName: pv-efs-example
```

加粗标红字段需要重点关注，其中参数说明如下：

表 11-9 极速文件存储卷 PVC 配置参数说明

参数	描述
name	PVC资源名称，同namespace下唯一。保证在namespace下唯一即可。（若PVC是由有状态应用动态创建，则保持和FlexVolume PVC的name一致）。
namespace	PVC资源命名空间。可参考FlexVolume PVC的相同字段。
storageClassName	指定K8s storage class名称；需配置为"csi-sfsturbo”。
storage	存储容量，单位Gi，必须和已有pv的storage大小保持一致。
volumeName	PV的名称。参考上述步骤创建的静态PV的名称。

步骤4 应用升级替换成新的PVC。

无状态应用

1. 通过kubectl create -f的形式创建pv和pvc。

   kubectl create -f pv-example.yaml

   kubectl create -f pvc-example.yaml

   说明

   命令中的yaml名称是示例，请以实际步骤**步骤2**和步骤**步骤3**创建的pv和pvc的yaml名字为准。

2. 进入应用更新升级界面：更新升级 - 高级设置 - 数据存储 - 云存储。

3. 卸载老存储，同时添加CSI格式的PVC的云存储，容器内挂载路径和以前保持一致，实现存储迁移。
4. 单击提交，确认后升级生效。
5. 等待pod running。

升级使用已有存储的有状态应用

1. 通过kubectl create -f的形式创建pv和pvc

kubectl create -f pv-example.yaml

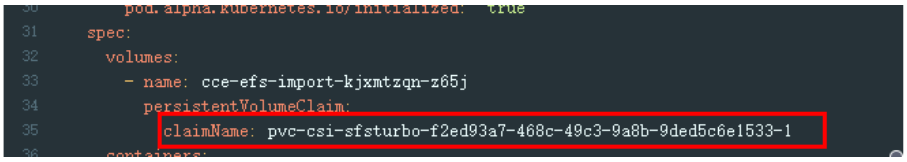
kubectl create -f pvc-example.yaml

说明

命令中的yaml名称是示例，请以实际步骤**步骤2**和步骤**步骤3**创建的pv和pvc的yaml名字为准。

2. 通过kubectl edit的方式修改有状态应使用新建的PVC。

kubectl edit sts sts-example -n xxx



说明

命令中的sts-example为待升级的有状态应用的名称，请以实际为准。xxx指代有状态应用所在的命名空间。

3. 等待pod running。

说明

当前界面暂未提供有状态应用添加新的云存储，因此升级替换成新PVC需要通过后台kubectl命里实现。

升级使用动态分配存储的有状态应用

1. 备份当前有状态应用使用的flexVolume格式的PV和PVC。

kubectl get pvc xxx -n {namespaces} -oyaml > pvc-backup.yaml

kubectl get pv xxx -n {namespaces} -oyaml > pv-backup.yaml

2. 将应用的实例数修改成0。
3. 在存储界面解关联有状态应用使用的flexVolume格式的PVC。
4. 通过kubectl create -f的形式创建pv和pvc。

kubectl create -f pv-example.yaml

kubectl create -f pvc-example.yaml

说明

命令中的yaml名称是示例，请以实际步骤**步骤2**和步骤**步骤3**创建的pv和pvc的yaml名字为准。

5. 将应用的实例数恢复，等待pod running。

说明

有状态应用动态创建存储是通过volumeClaimTemplates机制实现，而该字段K8s无法修改，因此无法通过更换新PVC的方式实现数据迁移。

volumeClaimTemplates的PVC命名格式是固定的，当符合命名格式的PVC已经存在的时候则直接使用该PVC。

因此需要些解关联原有PVC之后，创建同名的CSI格式的PVC来实现存储迁移。

6. 迁移完成，但是如果不重建有状态应用，扩容时仍是FlexVolume格式的PVC（按需操作）。

- 获取当前有状态应用yaml：

kubectl get sts xxx -n {namespaces} -oyaml > sts.yaml

- 备份当前有状态应用yaml：

cp sts.yaml sts-backup.yaml

- 修改有状态应用yaml中volumeClaimTemplates的定义：

vi sts.yaml

云硬盘存储卷volumeClaimTemplates的配置示例如下：

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161070049798261342
      namespace: default
      creationTimestamp: null
      annotations:
        everest.io/disk-volume-type: SAS
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-disk
```

其中参数和上述步骤创建的云硬盘存储卷的PVC保持一致。

文件存储卷volumeClaimTemplates配置示例如下：

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161063441560279697
      namespace: default
      creationTimestamp: null
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-nas
```

其中参数和上述步骤创建的文件存储卷PVC保持一致。

对象存储卷PVC配置示例如下：

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161070100417416148
      namespace: default
      creationTimestamp: null
      annotations:
        csi.storage.k8s.io/fstype: s3fs
        everest.io/obs-volume-type: STANDARD
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-obs
```

其中参数和上述步骤创建的对象存储卷PVC保持一致。

- 删除原有状态应用：

kubectl delete sts xxx -n {namespaces}

- 创建新的有状态应用

kubectl create -f sts.yaml

步骤5 检查业务功能。

1. 检查业务功能是否正常。
2. 检查数据是否丢失。

说明

若功能或数据检查异常需要回退，请执行步骤**步骤4**，选择FlexVolume格式的PVC并单击提交升级。

步骤6 卸载FlexVolume格式的PVC。

检查正常，存储管理界面执行解关联操作。

也可以后台通过kubectl指令删除Flexvolume格式的PVC和PV。

注意

在删除之前需要修改PV的回收策略persistentVolumeReclaimPolicy为Retain，否则底层存储会被回收。

在存储迁移执行前已完成集群升级可能会导致无法删除PV，可以去除PV的保护字段finalizers来实现PV删除

kubectl patch pv {pv_name} -p '{"metadata":{"finalizers":null}}'

----结束

11.5 自定义 StorageClass

应用现状

CCE中使用存储时，最常见的方法是创建PVC时通过指定StorageClassName定义要创建存储的类型，如下所示，使用PVC申请一个SAS（高I/O）类型云硬盘/块存储。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
```

可以看到在CCE中如果需要指定云硬盘的类型，是通过everest.io/disk-volume-type: SAS字段指定，这里SAS是云硬盘的类型，代表高I/O，还有SSD（超高I/O）可以指定。

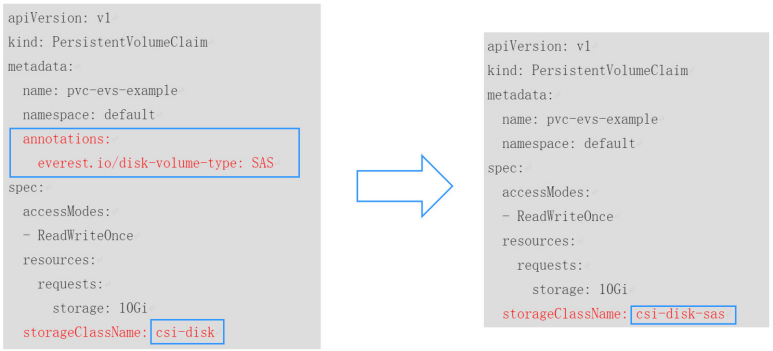
这种写法在如下几种场景下存在问题：

- 部分用户觉得使用everest.io/disk-volume-type指定云硬盘类型比较繁琐，希望只通过StorageClassName指定。
- 部分用户是从自建Kubernetes或其他Kubernetes服务切换到CCE，已经写了很多应用的YAML文件，这些YAML文件中通过不同StorageClassName指定不同类型存储，迁移到CCE上时，使用存储就需要修改大量YAML文件或Helm Chart包，这非常繁琐且容易出错。
- 部分用户希望能够设置默认的StorageClassName，所有应用都使用默认存储类型，在YAML中不用指定StorageClassName也能按创建默认类型存储。

解决方案

本文介绍在CCE中自定义StorageClass的方法，并介绍设置默认StorageClass的方法，通过不同StorageClassName指定不同类型存储。

- 对于第一个问题：可以将SAS、SSD类型云硬盘分别定义一个StorageClass，比如定义一个名为csi-disk-sas的StorageClass，这个StorageClass创建SAS类型的存储，则前后使用的差异如下图所示，编写YAML时只需要指定StorageClassName，符合特定用户的使用习惯。



未使用自定义StorageClass的写法　　使用自定义StorageClass的写法

- 对于第二个问题：可以定义与用户现有YAML中相同名称的StorageClass，这样可以省去修改YAML中StorageClassName的工作。
- 对于第三个问题：可以设置默认的StorageClass，则YAML中无需指定StorageClassName也能创建存储，按如下写法即可。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

CCE 默认存储类

目前CCE默认提供csi-disk、csi-nas、csi-obs等StorageClass，在声明PVC时使用对应StorageClassName，就可以自动创建对应类型PV，并自动创建底层的存储资源。

执行如下kubectl命令即可查询CCE提供的默认StorageClass。您可以使用CCE提供的CSI插件自定义创建StorageClass。

```
# kubectl get sc
NAME                PROVISIONER                     AGE
csi-disk            everest-csi-provisioner         17d         # 云硬盘
csi-disk-topology   everest-csi-provisioner         17d         # 延迟创建的云硬盘
csi-nas             everest-csi-provisioner         17d         # 文件存储 1.0
csi-obs             everest-csi-provisioner         17d         # 对象存储
csi-sfsturbo        everest-csi-provisioner         17d         # 极速文件存储
csi-local           everest-csi-provisioner         17d         # 本地持久卷
csi-local-topology  everest-csi-provisioner         17d         # 延迟创建的本地持久卷
```

每个StorageClass都包含了动态制备PersistentVolume时会使用到的默认参数。如以下云硬盘存储类的示例：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

表 11-10 关键参数说明

参数	描述
provisioner	存储资源提供商，CCE均由everest插件提供，此处只能填写everest-csi-provisioner。
parameters	存储参数，不同类型的存储支持的参数不同。详情请参见**表11-11**。
reclaimPolicy	用来指定创建PV的persistentVolumeReclaimPolicy字段值，支持Delete和Retain。如果StorageClass 对象被创建时没有指定reclaimPolicy，它将默认为Delete。 • Delete：表示动态创建的PV，在PVC销毁的时候PV也会自动销毁。 • Retain：表示动态创建的PV，在PVC销毁的时候PV不会自动销毁。
allowVolumeExpansion	定义由此存储类创建的PV是否支持动态扩容，默认为false。是否能动态扩容是由底层存储插件来实现的，这里只是一个开关。

参数	描述
volumeBindingMode	表示卷绑定模式，即动态创建PV的时间，分为立即创建和延迟创建。 ● Immediate：创建PVC时完成PV绑定和动态创建。 ● WaitForFirstConsumer：延迟PV的绑定和创建，当在工作负载中使用该PVC时才执行PV创建和绑定流程。
mountOptions	该字段需要底层存储支持，如果不支持挂载选项，却指定了挂载选项，会导致创建PV操作失败。

表 11-11 parameters 参数说明

存储类型	参数	是否必选	描述
云硬盘	csi.storage.k8s.io/csi-driver-name	是	驱动类型，使用云硬盘类型时，参数取值固定为“disk.csi.everest.io”。
	csi.storage.k8s.io/fstype	是	使用云硬盘时，支持的参数值为“ext4”。
	everest.io/disk-volume-type	是	云硬盘类型，全大写。 ● SAS：高I/O ● SSD：超高I/O ● GPSSD：通用型SSD ● ESSD：极速型SSD ● GPSSD2：通用性SSD v2，Everest版本为2.4.4及以上支持使用，使用时需同时指定everest.io/disk-iops和everest.io/disk-throughput注解。
	everest.io/passthrough	是	参数取值固定为“true”，表示云硬盘的设备类型为SCSI类型。不允许设置为其他值。
文件存储	csi.storage.k8s.io/csi-driver-name	是	驱动类型，使用文件存储类型时，参数取值固定为“nas.csi.everest.io”。
	csi.storage.k8s.io/fstype	是	使用文件存储时，支持的参数值为“nfs”。
	everest.io/share-access-level	是	参数取值固定为“rw”，表示文件存储可读写。
	everest.io/share-access-to	是	集群所在VPC ID。
	everest.io/share-is-public	否	参数取值固定为“false”，表示文件共享为私人可见。 使用SFS 3.0时无需填写。

存储类型	参数	是否必选	描述
	everest.io/sfs-version	否	仅使用SFS 3.0时需要填写，固定值为“sfs3.0”。
极速文件存储	csi.storage.k8s.io/csi-driver-name	是	驱动类型，使用极速文件存储类型时，参数取值固定为“sfsturbo.csi.everest.io”。
	csi.storage.k8s.io/fstype	是	使用极速文件存储时，支持的参数值为“nfs”。
	everest.io/share-access-to	是	集群所在VPC ID。
	everest.io/share-expand-type	否	扩展类型，默认值为“bandwidth”，表示增强型的文件系统。该字段不起作用。
	everest.io/share-source	是	参数取值固定为“sfs-turbo”。
	everest.io/share-volume-type	否	极速文件存储类型，默认值为“STANDARD”，表示标准型和标准型增强版。该字段不起作用。
对象存储	csi.storage.k8s.io/csi-driver-name	是	驱动类型，使用对象存储类型时，参数取值固定为“obs.csi.everest.io”。
	csi.storage.k8s.io/fstype	是	实例类型，支持的参数值为“s3fs”和“obsfs”。 ● obsfs：并行文件系统，配套使用obsfs挂载，推荐使用。 ● s3fs：对象桶，配套使用s3fs挂载。
	everest.io/obs-volume-type	是	对象存储类型。 ● fsType设置为s3fs时，支持STANDARD（标准桶）、WARM（低频访问桶）。 ● fsType设置为obsfs时，该字段不起作用。

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

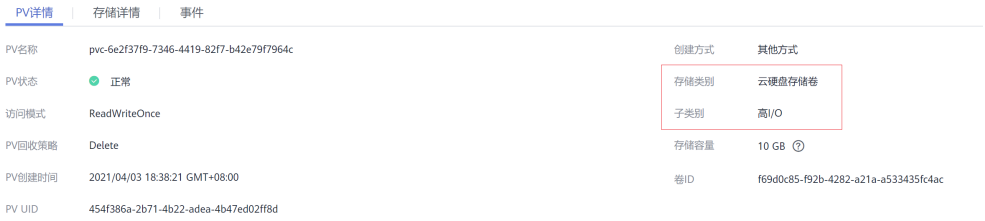
- 使用csi-disk-sas创建PVC。

  ```
  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。



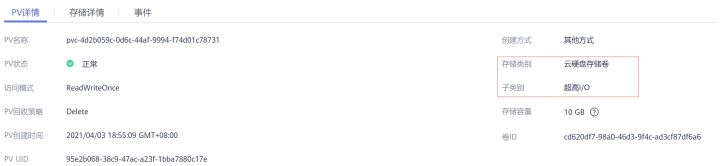
- 不指定StorageClassName，使用默认配置，如下所示，并未指定storageClassName。

```
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
```

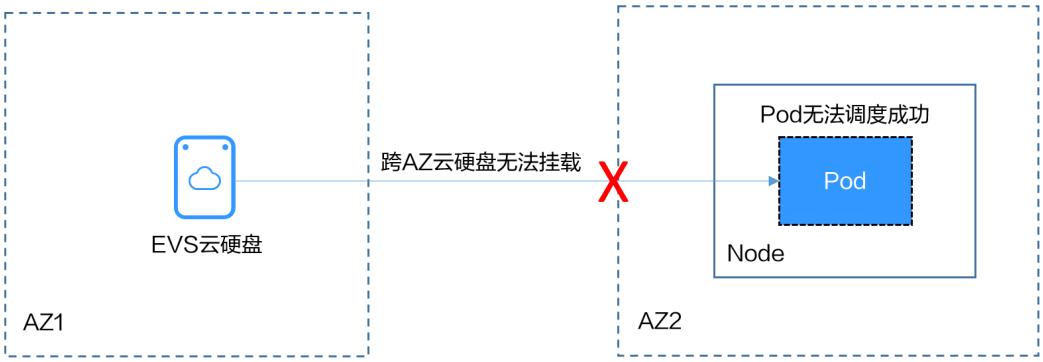
在CCE控制台界面上查看PVC详情，在“PV详情”页签下可以看到磁盘类型是超高I/O。



11.6 使用延迟绑定的云硬盘（csi-disk-topology）实现跨AZ调度

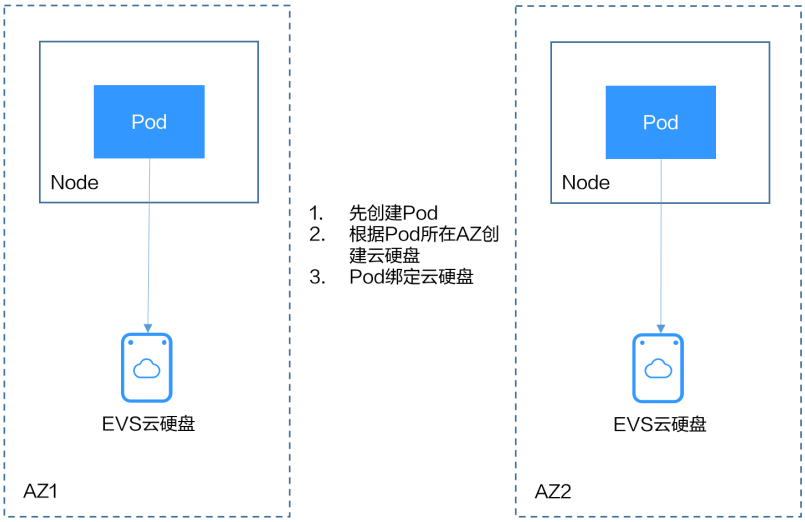
应用现状

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
csi-sfsturbo        everest-csi-provisioner         Delete           Immediate              true
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 升级实例过程中实现业务不中断

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。确保升级过程中始终有工作负载能够提供服务。
- 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，可能会出现服务中断。这种情况下可以通过实例原地升级的方式避免服务中断，即保证ELB后端的节点上存在一个正常状态的Pod。

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

本示例中，工作负载副本个数为200，并通过LoadBalance类型Service对外暴露服务。关联有Loadbalance类型Service/Ingress的工作负载滚动升级由于涉及跨服务调用，因此需要格外注意滚动升级参数的配置。

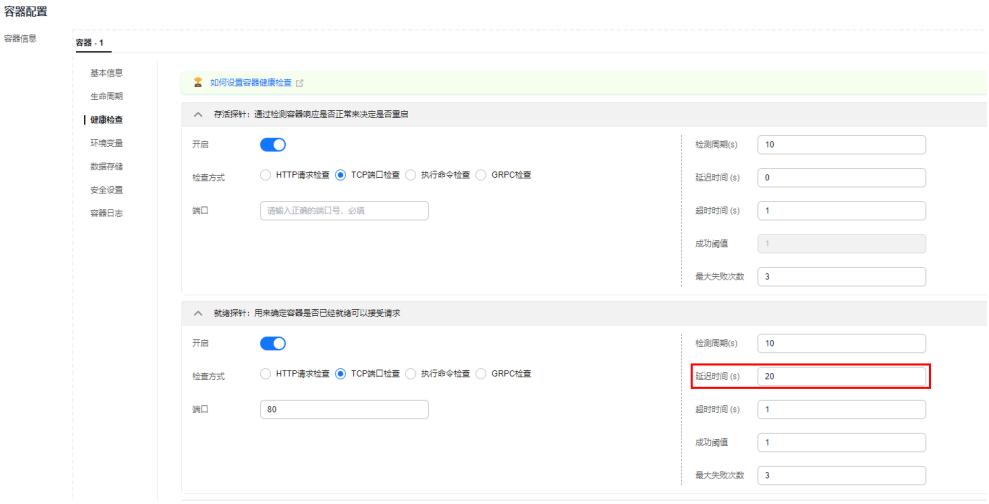
步骤1 登录CCE控制台，单击集群名称进入集群，在左侧选择“工作负载”。

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。

   本示例中配置就绪探针延迟探测时间为20s，用于控制工作负载批量滚动的时间间隔。

   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。

   本示例中配置最大无效实例数（maxUnavailable）为2%，最大浪涌（maxSurge）为2%，用于控制工作负载的滚动步长。配合就绪探针的延迟探测，控制每20s时间内，升级8个工作负载。

   图 12-2 滚动升级

3. 设置优雅终止：

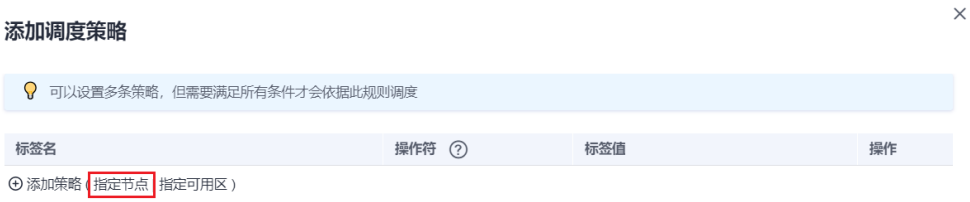
   a. 在容器配置中选择“生命周期”，设置停止前处理，建议设置为业务处理完所有剩余请求所需的时间，其中多为长连接请求。例如，您可以设置工作负载收到删除请求后休眠30s，能够有充足的时间来处理剩余的请求，保证服务的正常运行。

   b. 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。缩容时间窗时间设置需大于“生命周期”的停止前处理时间，建议在容器停止前命令执行时间的基础上加30s。例如，停止前处理时间设置为30s，因此缩容时间窗设置为60s。

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，可查看到会先创建新实例，然后再停止旧实例，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。详情请参见**通过kubectl连接集群**。

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 使用 Init 容器初始化应用

概念

init-Containers，即初始化容器，顾名思义容器启动的时候，会先启动可一个或多个容器，如果有多个，那么这几个Init Container按照定义的顺序依次执行，只有所有的Init Container执行完后，主容器才会启动。由于一个Pod里的存储卷是共享的，所以Init Container里产生的数据可以被主容器使用到。

Init Container可以在多种K8s资源里被使用到如Deployment、DaemonSet、Job等，但归根结底都是在Pod启动时，在主容器启动前执行，做初始化工作。

使用场景

部署服务时需要做一些准备工作，在运行服务的pod中使用一个init container，可以执行准备工作，完成后Init Container结束退出，再启动要部署的容器。

- **等待其它模块Ready：**比如有一个应用里面有两个容器化的服务，一个是Web Server，另一个是数据库。其中Web Server需要访问数据库。但是当启动这个应用的时候，并不能保证数据库服务先启动起来，所以可能出现在一段时间内Web Server有数据库连接错误。为了解决这个问题，可以在运行Web Server服务的Pod里使用一个Init Container，去检查数据库是否准备好，直到数据库可以连接，Init Container才结束退出，然后Web Server容器被启动，发起正式的数据库连接请求。
- **初始化配置：**比如集群里检测所有已经存在的成员节点，为主容器准备好集群的配置信息，这样主容器起来后就能用这个配置信息加入集群。
- **其它使用场景：**如将pod注册到一个中央数据库、下载应用依赖等。

更多内容请参见**初始容器文档参考**。

操作步骤

步骤1 编辑initcontainer工作负载yaml文件。

vi deployment.yaml

Yaml示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
```

```
  matchLabels:
    name: mysql
template:
  metadata:
    labels:
      name: mysql
  spec:
    initContainers:
    - name: getresource
      image: busybox
      command: ['sleep 20']
    containers:
    - name: mysql
      image: percona:5.7.22
      imagePullPolicy: Always
      ports:
      - containerPort: 3306
      resources:
        limits:
          memory: "500Mi"
          cpu: "500m"
        requests:
          memory: "500Mi"
          cpu: "250m"
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "mysql"
```

步骤2 创建initcontainer工作负载。

kubectl create -f deployment.yaml

命令行终端显示如下类似信息：

```
deployment.apps/mysql created
```

步骤3 在工作负载运行的节点上查询创建的docker容器。

docker ps -a|grep mysql

init容器运行后会直接退出，查询到的是exited(0)的退出状态。

```
0dc822969e3f        percona             "docker-entrypoint..."   34 seconds ago       Up 33 seconds
ql_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
a745881214e7        busybox             "sh -c 'sleep 20'"       About a minute ago   Exited (0) 50 seconds ago
resource_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
615db9e60a80        cfe-pause:11.23.1   "/pause"                 About a minute ago   Up About a minute
 mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
```

----结束

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 将脚本先从容器内复制至节点，在脚本中添加-Duser.timezone=GMT+08后，从节点复制到容器中。

容器内的文件复制至宿主机：

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

宿主机中的文件复制至容器内：

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 容器网络带宽限制的配置建议

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		
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		

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
限制的场景	无	无	• 不支持Pod访问100.64.0.0/10和214.0.0.0/8外部云服务网段的限速场景 • 不支持健康检查的流量限速场景
限速值取值范围	只支持单位M或G的限速配置，如100M，1G；最小取值1M，最大取值4.29G。		

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，网络带宽限制字段详解请参见**表12-1**。

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 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
kubernetes.io/egress-bandwidth	工作负载实例出方向网络带宽 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 参数配置 Pod 的/etc/hosts 文件

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

```
root@hostaliases-pod:/# cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.0.0.25       hostaliases-pod

# Entries added by HostAliases.
127.0.0.1       foo.local       bar.local
10.1.2.3        foo.remote      bar.remote
```

----结束

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內特别描述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 CCE 中使用 x86 和 ARM 双架构镜像

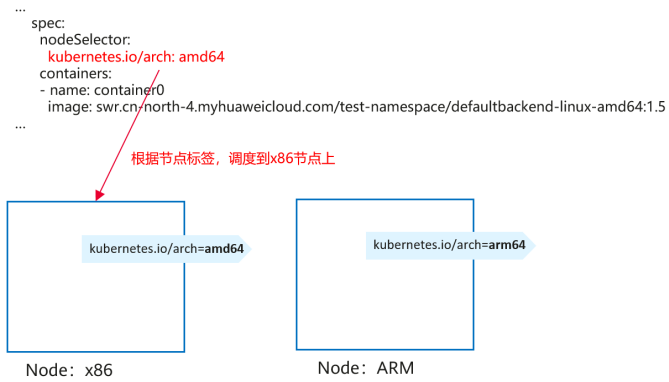
应用现状

CCE支持在同一集群下创建x86节点及ARM架构节点。由于ARM和x86底层架构不同，通常ARM架构的镜像（也就是应用程序）无法在x86架构节点上运行，反之亦然。这就容易造成工作负载在拥有x86与ARM节点的集群上部署失败。

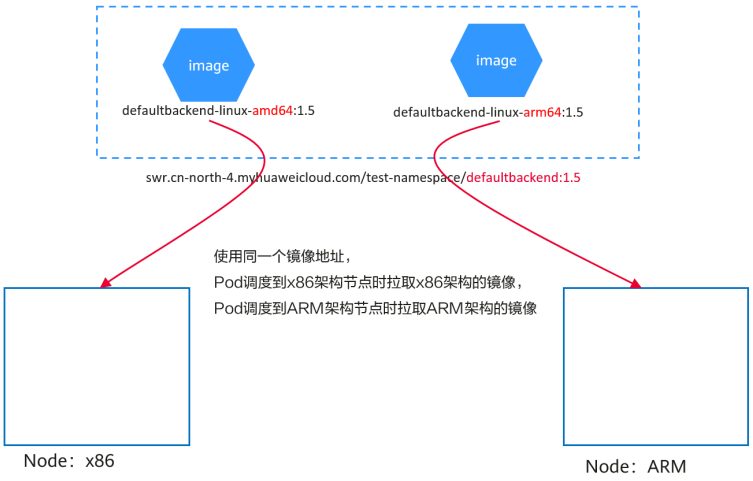
解决方案

解决在不同架构的节点使用镜像创建工作负载通常有两种方法：

- 创建工作负载的时候通过亲和性设置，使用ARM架构镜像时让Pod调度到ARM架构的节点上，使用x86架构镜像时让Pod调度到x86架构的节点上。



- 构建双架构镜像，同时支持两种架构，当Pod调度到ARM架构节点时拉取ARM架构的镜像，当Pod调度到x86架构节点时拉取x86架构的镜像。双架构镜像的一个特征是镜像可以只使用一个地址，但背后有两个镜像，这样在描述工作负载时，可以使用同一个镜像地址，且不用配置亲和性，工作负载描述文件更简洁更容易维护。



亲和性配置说明

CCE在创建节点时，会自动给节点打上kubenetes.io/arch的标签，表示节点架构，如下所示。

```
kubernetes.io/arch=amd64
```

取值**amd64**表示是x86架构，**arm64**表示是ARM架构。

在创建工作负载时，可以通过配置节点亲和性，将Pod调度到对应架构的节点上。

使用YAML可以通过nodeSelector进行配置，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test
spec:
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      nodeSelector:
        kubernetes.io/arch: amd64
      containers:
      - name: container0
        image: swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
      imagePullSecrets:
      - name: default-secret
```

双架构镜像构建方法

说明

制作双架构镜像，Docker客户端版本需要大于18.03。

构建双架构镜像的本质是先分别构建x86和ARM架构的镜像，然后通过构建双架构的镜像manifest。

例如已经构建好了defaultbackend-linux-amd64:1.5和defaultbackend-linux-arm64:1.5两个镜像，分别是x86架构和ARM架构。

将这两个镜像上传到SWR镜像仓库，如下所示。上传镜像的具体方法请参见**客户端上传镜像**。

```
# 给原始amd64镜像defaultbackend-linux-amd64:1.5加tag
docker tag defaultbackend-linux-amd64:1.5  swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
# 给原始arm64镜像defaultbackend-linux-arm64:1.5加tag
docker tag defaultbackend-linux-arm64:1.5  swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5
# 上传amd64镜像至swr镜像仓库
docker push swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
# 上传arm64镜像至swr镜像仓库
docker push swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5
```

创建双架构manifest文件并上传。

```
#  开启DOCKER_CLI_EXPERIMENTAL
export DOCKER_CLI_EXPERIMENTAL=enabled
# 创建镜像manifest文件
```

```
docker manifest create --amend --insecure swr.ap-southeast-1.myhuaweicloud.com/test-namespace/
defaultbackend:1.5 swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-
arm64:1.5 swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
# 给镜像manifest文件添加arch信息
docker manifest annotate swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5
swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5 --arch amd64
docker manifest annotate swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5
swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5 --arch arm64
# 向swr镜像仓库推送镜像manifest
docker manifest push -p --insecure swr.ap-southeast-1.myhuaweicloud.com/test-namespace/
defaultbackend:1.5
```

这样在创建负载时就只需要使用swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5这个镜像地址。

- 当Pod调度到x86架构的节点时，会拉取swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5这个镜像。
- 当Pod调度到ARM架构的节点时，会拉取swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5这个镜像。

12.10 通过 Core Dump 文件定位容器问题

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。设置umask的方法请参见**设置挂载参数**。

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
```

12.11 在 CCE Turbo 集群中配置 Pod 延时启动参数

应用场景

CCE Turbo集群在某些特定场景下（例如跨VPC、专线互联），会出现对端Pod的路由规则生效慢的情况。在这种情况下，可以利用Pod延时启动的能力进行规避。

您也可以使用企业路由器连接对端VPC来解决该问题，详情请参见**10.12 集群通过企业路由器连接对端VPC**。

约束与限制

仅以下指定版本的CCE Turbo集群支持配置Pod延时启动参数：：

- v1.19集群：v1.19.16-r40及以上版本
- v1.21集群：v1.21.11-r0及以上版本
- v1.23集群：v1.23.9-r0及以上版本
- v1.25集群：v1.25.4-r0及以上版本

通过 kubectl 命令行设置

您可以通过对工作负载添加annotations来设置是否开启Pod延时启动功能，如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        cni.yangtse.io/readiness-delay-seconds: "20"
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
      imagePullSecrets:
        - name: default-secret
```

该示例为一个需要配置访问跨VPC虚机IP的Deployment，该Deployment的副本数最大为10且滚动升级最大浪涌为25%，即升级过程中可能的最大Pod数为13。在该工作负载注解中，指定了Pod延时启动时间为20s，在该时间内保证Pod正常启动后，跨VPC网络可正常访问。

表 12-5 Pod 延迟启动 annotation 配置

注解	默认值	参数说明	取值范围
cni.yangtse.io/readiness-delay-seconds	无	Pod健康检查延时等待的时间。	0-60

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：

说明

本示例中权限规则包含test命名空间下Pod资源的只读权限（get/list/watch）以及deployment的读取（get/list/watch）和创建（create）权限。

```
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5**~步骤**步骤7**将用户信息写入到配置文件中。

步骤4 手动为ServiceAccount创建长期有效的Token。

```
vi my-sa-token.yaml
```

内容如下：

```
apiVersion: v1
kind: Secret
metadata:
  name: my-sa-token-secret
  namespace: test
  annotations:
    kubernetes.io/service-account.name: my-sa
type: kubernetes.io/service-account-token
```

创建Token：

```
kubectl create -f my-sa-token.yaml
```

步骤5 配置集群访问信息。

1. 将密钥中的ca.crt解码后导出备用：

```
kubectl get secret my-sa-token-secret -n test -oyaml |grep ca.crt: | awk '{print $2}' |base64 -d > /home/ca.crt
```

2. 设置集群访问方式，其中**test-arm**为需要访问的集群，https://192.168.0.110:5443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://192.168.0.110:5443  --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://192.168.0.110:5443 --kubeconfig=/home/test.config --insecure-skip-tls-verify=true
     ```

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地址默认为内网地址，绑定弹性IP后可使用公网地址访问。

图 13-1 获取内网或外网 apiserver 地址

步骤6 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-secret -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7 配置集群认证访问的context信息，**ui-admin@test**为context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8 设置context，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7**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8**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 group "" 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 delete 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。

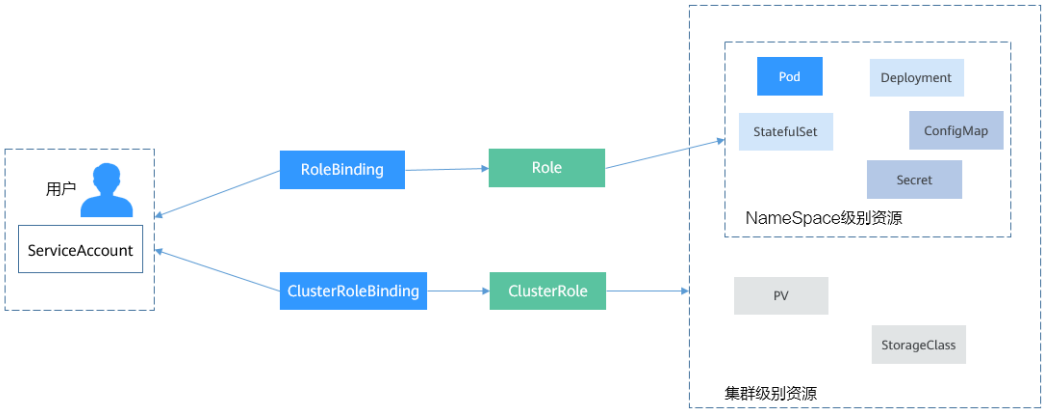
解决方案

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

使用账号登录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，如下所示。创建IAM用户和用户组的具体步骤请参见**创建IAM用户**和**创建用户组**。

创建后给cce-role-group用户组授予CCE FullAccess权限，如下所示。给用户组授权的方法具体请参见**给用户组授权**。

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

使用账号登录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

使用账号登录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

此处使用账号是因为集群是使用账号创建，CCE在创建集群时会自动给该账号添加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。

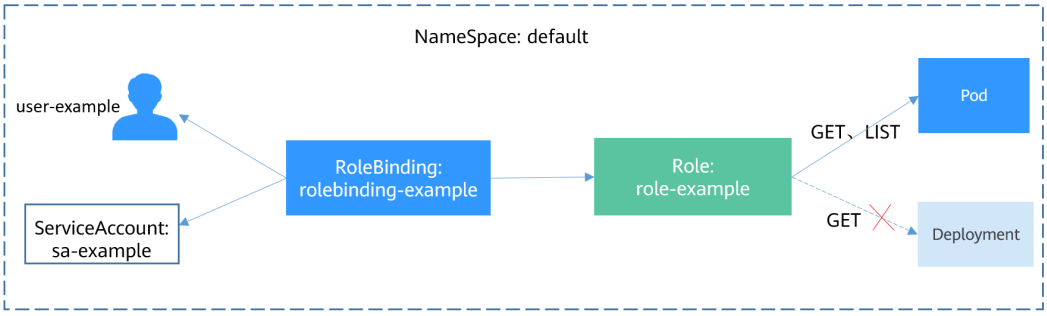
详细的类型和操作请参见**使用 RBAC 鉴权**。

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。

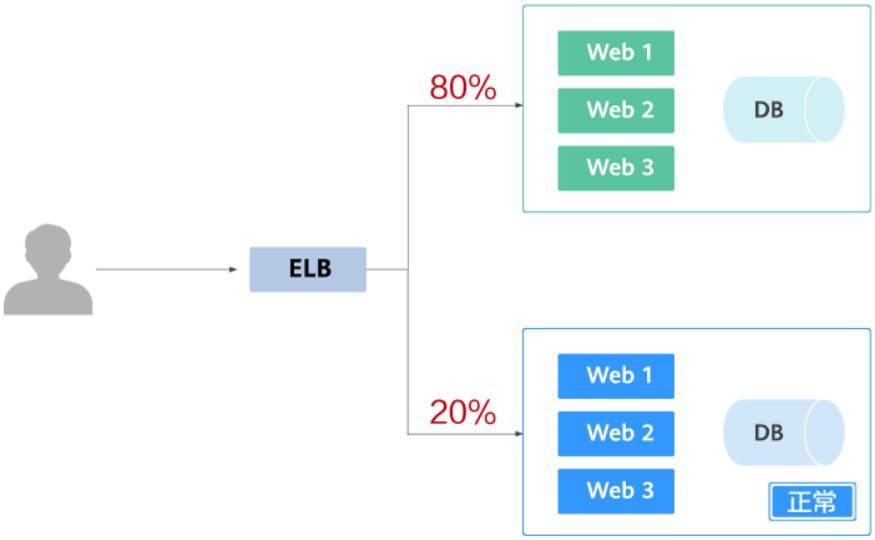
解决方案

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。

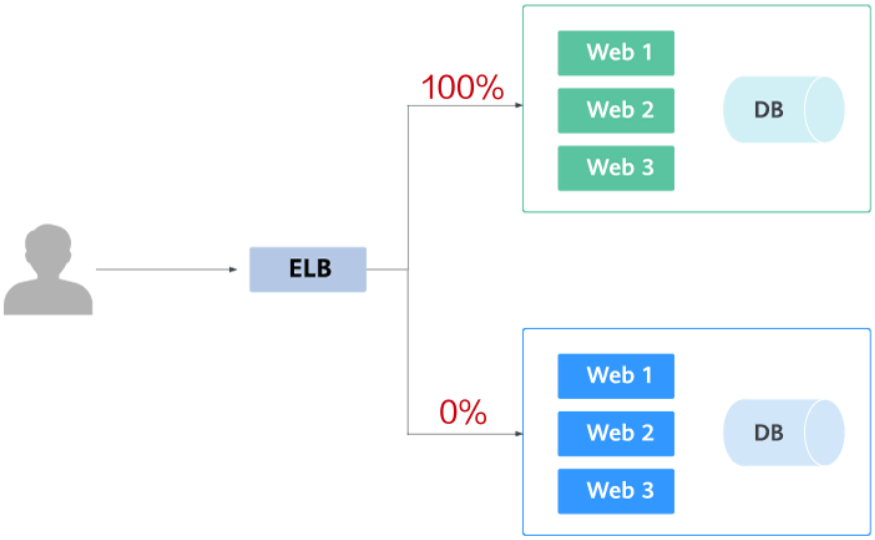
本文着重介绍灰度发布和蓝绿发布的原理及实践案例。

- 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。

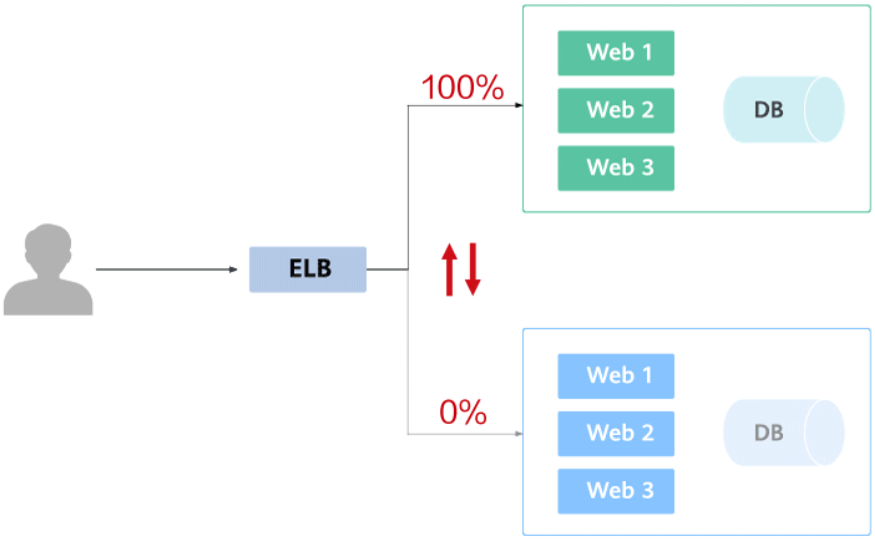
  以下示意图可描述灰度发布的大致流程：先切分20%的流量到新版本，若表现正常，逐步增加流量占比，继续测试新版本表现。若新版本一直很稳定，那么将所有流量都切分到新版本，并下线老版本。



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，利用开源工具和服务网格的能力实现。这两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 • 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。

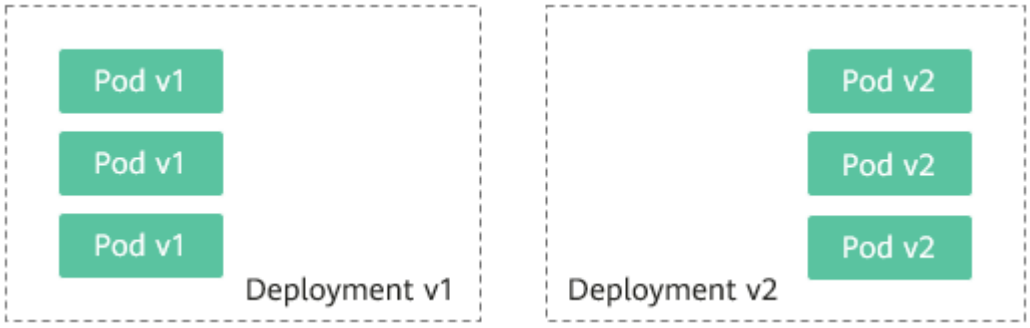
- LoadBalancer类型的Service由ELB服务实现，在创建Service时，可以使用已有ELB实例，也可以新建一个ELB实例。
- 支持一键式安装nginx-ingress插件，并且在安装过程中实现ELB的创建与对接。

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

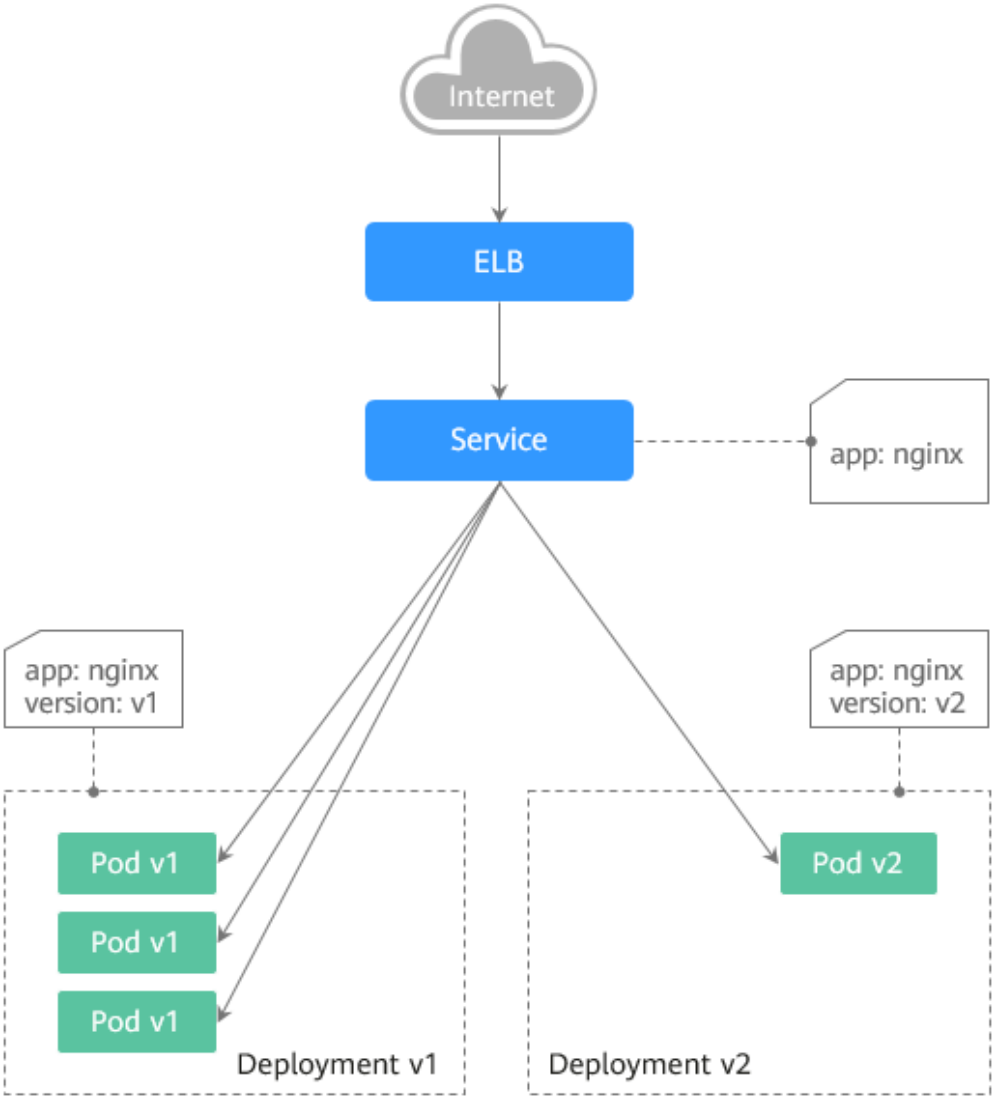
用户通常使用无状态负载 Deployment、有状态负载 StatefulSet等Kubernetes对象来部署业务，每个工作负载管理一组Pod。以Deployment为例，示意图如下：



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。

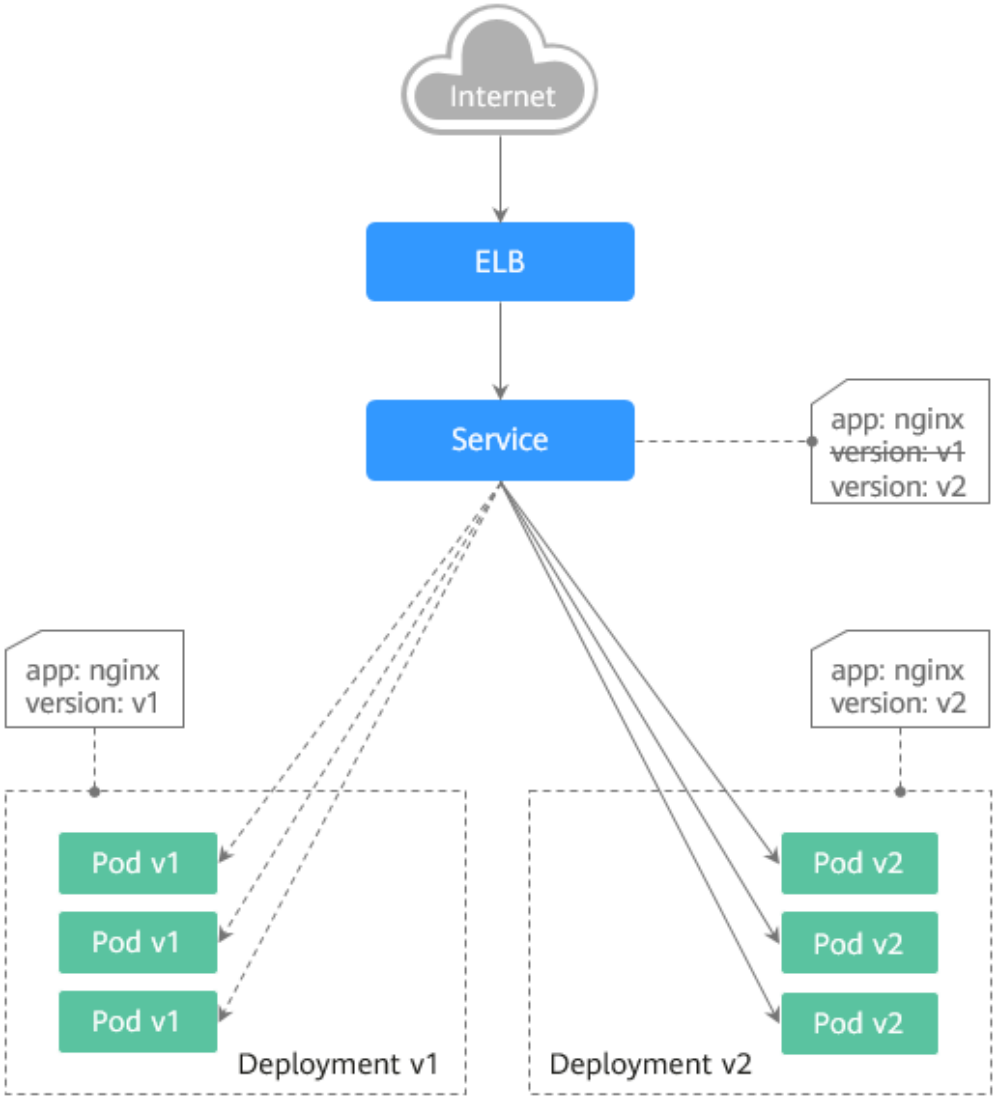
- **灰度发布原理**

  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

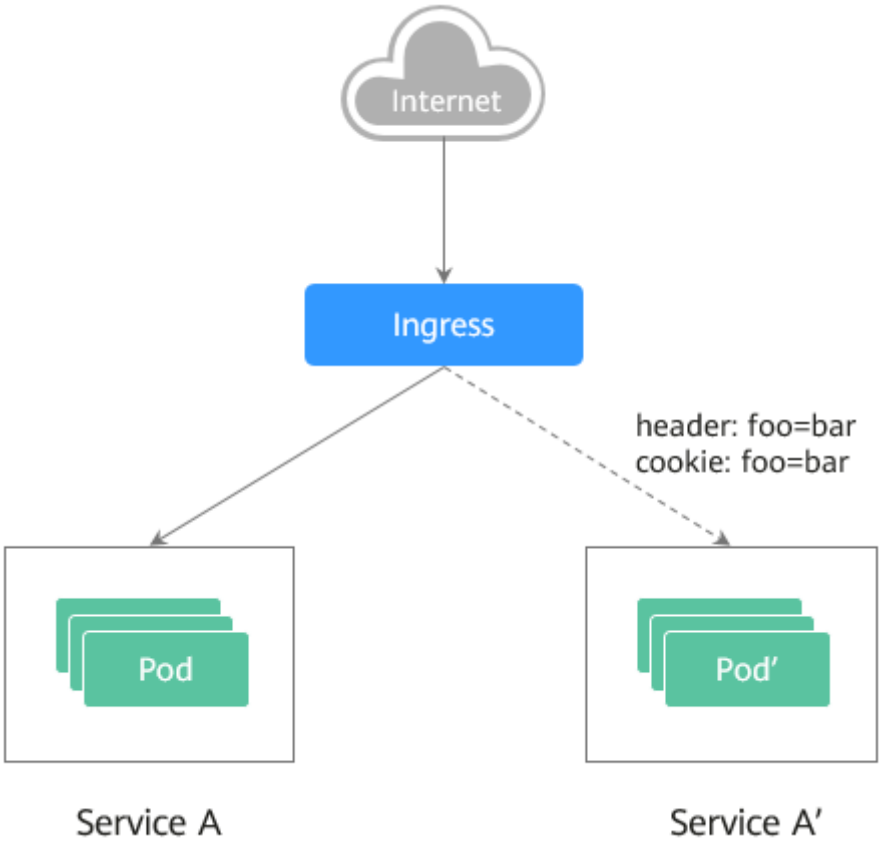
本文将介绍使用Nginx Ingress实现灰度发布和蓝绿发布的应用场景、用法详解及实践步骤。

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：

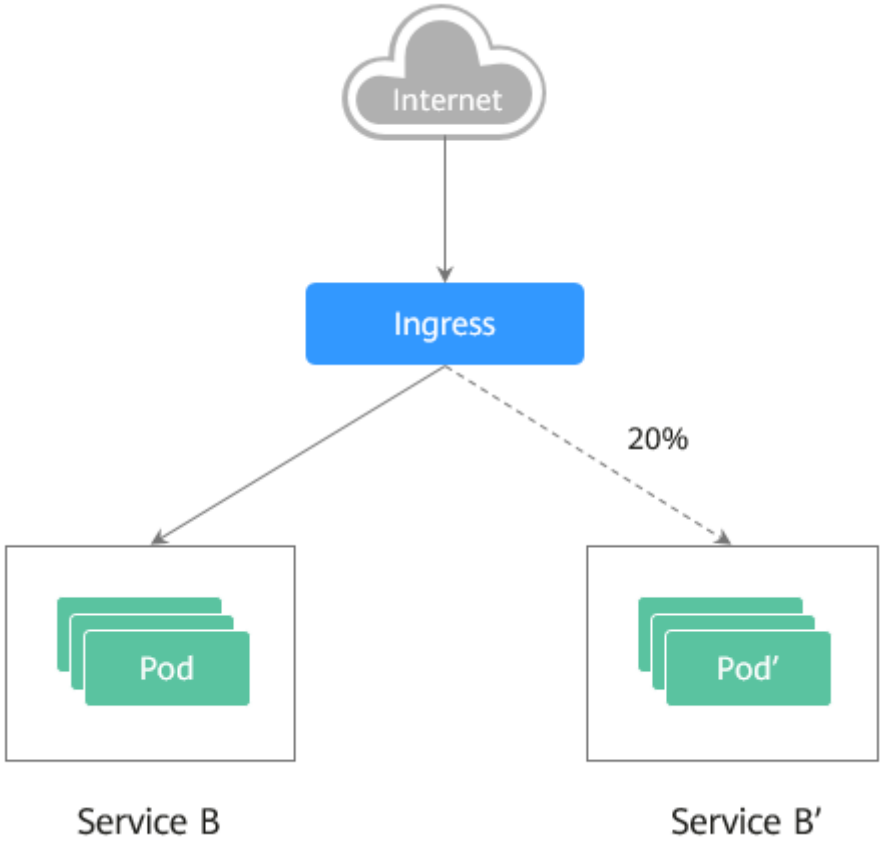
- **场景一：切分部分用户流量到新版本**

  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。详细操作可参考**安装插件**。
- 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。

  a. 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
```

b. 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。

a. 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
```

b. 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。

  a. 创建Canary Ingress，并增加annotation，将20%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: "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
```

  b. 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。

  a. 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
```

b. 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。

15 批量计算

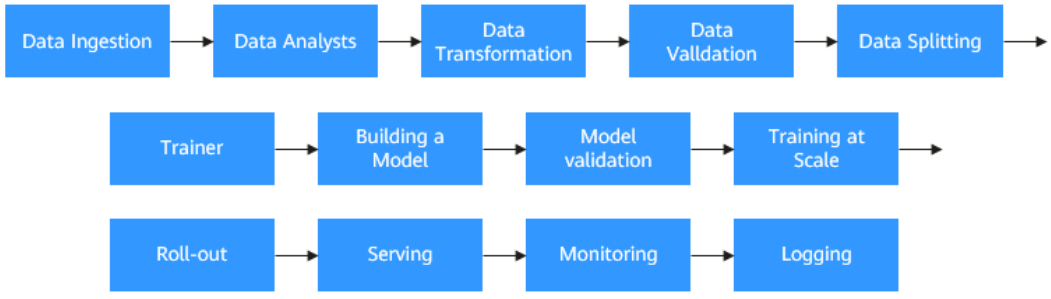
15.1 CCE 部署使用 Kubeflow

15.1.1 Kubeflow 部署

Kubeflow 的诞生背景

基于Kubernetes构建一个端到端的AI计算平台是非常复杂和繁琐的过程，它需要处理很多个环节。如图15-1所示，除了熟知的模型训练环节之外还包括数据收集、预处理、资源管理、特性提取、数据验证、模型的管理、模型发布、监控等环节。对于一个AI算法工程师来讲，如果要做模型训练，就不得不搭建一套AI计算平台，这个过程耗时费力，而且需要很多的知识积累。

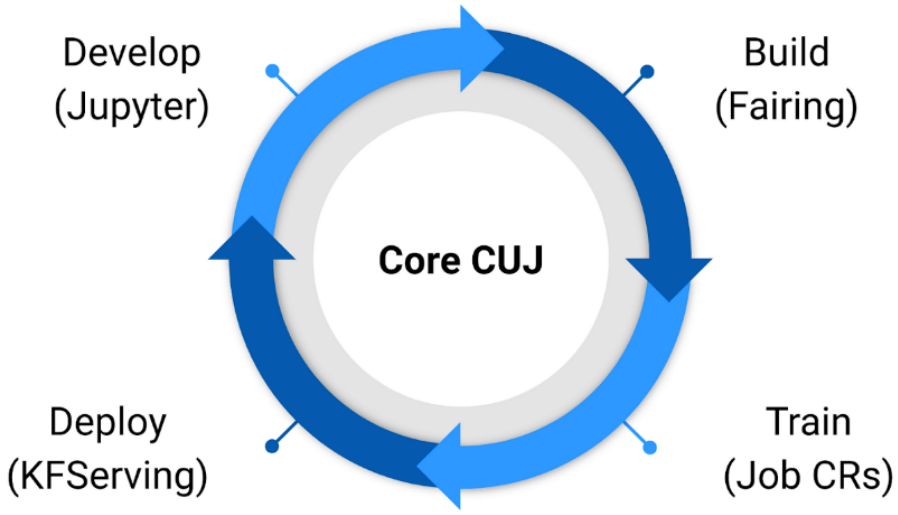
图 15-1 模型训练环节



Kubeflow诞生于2017年，Kubeflow项目是基于容器和Kubernetes构建，旨在为数据科学家、机器学习工程师、系统运维人员提供面向机器学习业务的敏捷部署、开发、训练、发布和管理平台。它利用了云原生技术的优势，让用户更快速、方便的部署、使用和管理当前最流行的机器学习软件。

目前Kubeflow 1.0版本已经发布，包含开发、构建、训练、部署四个环节，可全面支持企业用户的机器学习、深度学习完整使用过程。

如下图所示：



通过Kubeflow 1.0，用户可以使用Jupyter开发模型，然后使用fairing（SDK）等工具构建容器，并创建Kubernetes资源训练其模型。模型训练完成后，用户还可以使用KFServing创建和部署用于推理的服务器。再结合pipeline（流水线）功能可实现端到端机器学习系统的自动化敏捷构建，实现AI领域的DevOps。

前提条件

- 已在CCE创建一个集群clusterA，集群下有一个可用GPU节点，节点上的GPU卡数量大于等于2。
- 节点上绑定了EIP，并配置了kubectl命令行工具，详情请参见**通过kubectl连接集群**。

安装 Kustomize

Kustomize是一个开源工具，用于管理Kubernetes应用程序的配置。它允许您将应用程序的配置从应用程序本身中分离出来，并根据需要进行修改。从Kubeflow 1.3开始，所有组件都应仅使用Kustomize进行部署。

步骤1 安装Kustomize。由于Kubeflow与Kustomize的早期版本不兼容，仅支持Kustomize 5及更高版本，本文中使用5.1.0版本。

```
curl -o install_kustomize.sh "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh"
sh install_kustomize.sh 5.1.0 .
```

安装过程可能需要等待3-5分钟，回显如下：

```
v5.1.0
kustomize installed to /root/kubeflow/./kustomize
```

步骤2 将kustomize移到/bin目录，以便在全局使用kustomize命令。

```
cp kustomize /bin/
```

----结束

安装 Kubeflow

您可以参考以下步骤安装所有Kubeflow官方组件。成功安装所有内容后，您可以访问Kubeflow中央仪表板，详情请参见**连接Kubeflow**。

步骤1 安装Kubeflow 1.7.0版本。

```
wget https://github.com/kubeflow/manifests/archive/refs/tags/v1.7.0.zip
unzip v1.7.0.zip
```

步骤2 使用Kustomize创建用于部署Kubeflow的YAML文件。

```
cd ./manifests-1.7.0/
kustomize build example -o example.yaml
```

步骤3 配置Kubeflow所需存储资源。

- katib-mysql
- mysql-pv-claim
- minio-pv-claim
- authservice-pvc

由于Kubeflow在创建时需要配置一些存储资源，官方示例中的存储配置无法在CCE中生效，导致上述PVC无法创建。因此需要在集群中提前创建同名的PVC，本文中均以云硬盘类型为例，您可以按需替换云存储类型。

创建pvc.yaml文件，YAML示例如下。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: katib-mysql
  namespace: kubeflow
  annotations:
    everest.io/disk-volume-type: SAS    # 云硬盘的类型
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  namespace: kubeflow
  annotations:
    everest.io/disk-volume-type: SAS    # 云硬盘的类型
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: csi-disk
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```



```
  name: minio-pvc
  namespace: kubeflow
  annotations:
    everest.io/disk-volume-type: SAS    # 云硬盘的类型
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: csi-disk
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: authservice-pvc
  namespace: istio-system
  annotations:
    everest.io/disk-volume-type: SAS    # 云硬盘的类型
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # 替换为您待部署应用的节点所在的区域
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # 替换为您待部署应用的节点所在的可用区
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
```

创建PVC。

```
kubectl apply -f pvc.yaml
```

步骤4 创建Kubeflow相关资源。

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

说明

由于网络原因，官方镜像可能无法拉取，导致工作负载出现ImagePullBackOff或FailedPullImage错误，请您自行添加合适的镜像代理。

步骤5 查看所有命名空间下的Pod是否都处于运行状态。

```
kubectl get pod -A
```

如果创建资源时出现非预期问题，请参见**常见问题**进行处理。

----结束

常见问题

- 遇到一些CRD资源不存在的场景，报错如下：

  ```
  error: resource mapping not found for name: "<RESOURCE_NAME>" namespace:
  "<SOME_NAMESPACE>" from "STDIN": no matches for kind "<CRD_NAME>" in version
  "<CRD_FULL_NAME>"
  ensure CRDs are installed first
  ```

  解决方案：

  这是因为kustomization创建CRD和CR速度较快，可能会出现CRD尚未创建就创建CR的情况。如果您遇到此错误，建议您重新创建资源。

- 工作负载创建时，遇到节点Pod过多的错误，报错如下：

  ```
  0/x nodes are available: x Too many pods.
  ```

解决方案：

该错误说明节点上调度的Pod超过节点最大实例数，建议扩容节点数。

- training-operator负载不能正常运行，日志中报的错误如下：

```
Waited for 1.039518449s due to client-side throttling, not priority and fairness, request: GET:https://10.247.0.1:443/apis/xxx/xx?timeout=32s
```

解决方案：

需要排除集群中不可用的APIService，执行以下命令查看集群中的APIService状态：

```
kubectl get apiservice
```

如果没有FALSE状态的APIService，等一到两分钟training-operator负载会正常运行。

15.1.2 Tensorflow 训练

Kubeflow部署成功后，使用ps-worker的模式来进行Tensorflow训练就变得非常容易。本节介绍一个Kubeflow官方的Tensorflow训练范例，您可参考**TensorFlow Training (TFJob)**获取更详细的信息。

创建 MNIST 示例

步骤1 部署TFJob资源以开始训练。

创建tf-mnist.yaml文件，示例如下：

```
apiVersion: "kubeflow.org/v1"
kind: TFJob
metadata:
  name: tfjob-simple
  namespace: kubeflow
spec:
  tfReplicaSpecs:
    Worker:
      replicas: 2
      restartPolicy: OnFailure
      template:
        spec:
          containers:
            - name: tensorflow
              image: kubeflow/tf-mnist-with-summaries:latest
              command:
                - "python"
                - "/var/tf_mnist/mnist_with_summaries.py"
```

步骤2 创建TFJob。

```
kubectl apply -f tf-mnist.yaml
```

步骤3 等待worker运行完毕后，查看运行日志。

```
kubectl -n kubeflow logs tfjob-simple-worker-0
```

回显如下：

```
...
Accuracy at step 900: 0.964
Accuracy at step 910: 0.9653
Accuracy at step 920: 0.9665
Accuracy at step 930: 0.9681
Accuracy at step 940: 0.9664
Accuracy at step 950: 0.9667
Accuracy at step 960: 0.9694
Accuracy at step 970: 0.9683
Accuracy at step 980: 0.9687
```

```
Accuracy at step 990: 0.966
Adding run metadata for 999
```

步骤4 删除TFJob。

```
kubectl delete -f tf-mnist.yaml
```

----结束

使用 GPU 训练

TFJob可在GPU场景下进行，该场景需要集群中包含GPU节点，并安装合适的驱动。

步骤1 在TFJob中指定GPU资源。

创建tf-gpu.yaml文件，示例如下：

该示例的主要功能是基于Tensorflow的分布式架构，利用卷积神经网络（CNN）中的ResNet50模型对随机生成的图像进行训练，每次训练32张图像（batch_size），共训练100次（step），记录每次训练过程中的性能（image/sec）。

```
apiVersion: "kubeflow.org/v1"
kind: "TFJob"
metadata:
  name: "tf-smoke-gpu"
spec:
  tfReplicaSpecs:
    PS:
      replicas: 1
      template:
        metadata:
          creationTimestamp: null
        spec:
          containers:
            - args:
                - python
                - tf_cnn_benchmarks.py
                - --batch_size=32
                - --model=resnet50
                - --variable_update=parameter_server
                - --flush_stdout=true
                - --num_gpus=1
                - --local_parameter_device=cpu
                - --device=cpu
                - --data_format=NHWC
              image: docker.io/kubeflow/tf-benchmarks-cpu:v20171202-bdab599-dirty-284af3
              name: tensorflow
              ports:
                - containerPort: 2222
                  name: tfjob-port
              resources:
                limits:
                  cpu: "1"
              workingDir: /opt/tf-benchmarks/scripts/tf_cnn_benchmarks
          restartPolicy: OnFailure
    Worker:
      replicas: 1
      template:
        metadata:
          creationTimestamp: null
        spec:
          containers:
            - args:
                - python
                - tf_cnn_benchmarks.py
                - --batch_size=32
                - --model=resnet50
                - --variable_update=parameter_server
```



```
      - --flush_stdout=true
      - --num_gpus=1
      - --local_parameter_device=cpu
      - --device=gpu
      - --data_format=NHWC
    image: docker.io/kubeflow/tf-benchmarks-gpu:v20171202-bdab599-dirty-284af3
    name: tensorflow
    ports:
      - containerPort: 2222
        name: tfjob-port
    resources:
      limits:
        nvidia.com/gpu: 1       # GPU数量
    workingDir: /opt/tf-benchmarks/scripts/tf_cnn_benchmarks
  restartPolicy: OnFailure
```

步骤2 创建TFJob。

```
kubectl apply -f tf-gpu.yaml
```

步骤3 等待worker运行完毕后（一般GPU训练大约需要5分钟），执行如下命令查看运行结果：

```
kubectl logs tf-smoke-gpu-worker-0
```

回显如下：

```
...
INFO|2023-09-02T12:04:25|/opt/launcher.py|27| Running warm up
INFO|2023-09-02T12:08:55|/opt/launcher.py|27| Done warm up
INFO|2023-09-02T12:08:55|/opt/launcher.py|27| Step      Img/sec loss
INFO|2023-09-02T12:08:56|/opt/launcher.py|27| 1 images/sec: 68.8 +/- 0.0 (jitter = 0.0) 8.777
INFO|2023-09-02T12:09:00|/opt/launcher.py|27| 10        images/sec: 70.4 +/- 0.4 (jitter = 1.8) 8.557
INFO|2023-09-02T12:09:04|/opt/launcher.py|27| 20        images/sec: 70.5 +/- 0.3 (jitter = 1.5) 8.090
INFO|2023-09-02T12:09:09|/opt/launcher.py|27| 30        images/sec: 70.3 +/- 0.3 (jitter = 1.6) 8.041
INFO|2023-09-02T12:09:13|/opt/launcher.py|27| 40        images/sec: 70.1 +/- 0.2 (jitter = 1.7) 9.464
INFO|2023-09-02T12:09:18|/opt/launcher.py|27| 50        images/sec: 70.1 +/- 0.2 (jitter = 1.6) 7.797
INFO|2023-09-02T12:09:23|/opt/launcher.py|27| 60        images/sec: 70.1 +/- 0.2 (jitter = 1.6) 8.595
INFO|2023-09-02T12:09:27|/opt/launcher.py|27| 70        images/sec: 70.0 +/- 0.2 (jitter = 1.7) 7.853
INFO|2023-09-02T12:09:32|/opt/launcher.py|27| 80        images/sec: 69.9 +/- 0.2 (jitter = 1.7) 7.849
INFO|2023-09-02T12:09:36|/opt/launcher.py|27| 90        images/sec: 69.8 +/- 0.2 (jitter = 1.7) 7.911
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| 100       images/sec: 69.7 +/- 0.1 (jitter = 1.7) 7.853
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| ----------------------------------------------------------------
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| total images/sec: 69.68
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| ----------------------------------------------------------------
INFO|2023-09-02T12:09:42|/opt/launcher.py|80| Finished: python tf_cnn_benchmarks.py --batch_size=32 --
model=resnet50 --variable_update=parameter_server --flush_stdout=true --num_gpus=1 --
local_parameter_device=cpu --device=gpu --data_format=NHWC --job_name=worker --ps_hosts=tf-smoke-
gpu-ps-0.default.svc:2222 --worker_hosts=tf-smoke-gpu-worker-0.default.svc:2222 --task_index=0
INFO|2023-09-02T12:09:42|/opt/launcher.py|84| Command ran successfully sleep for ever.
```

可以看到单个GPU的训练性能为69.68 images/sec。

----结束

15.1.3 使用 Kubeflow 和 Volcano 实现典型 AI 训练任务

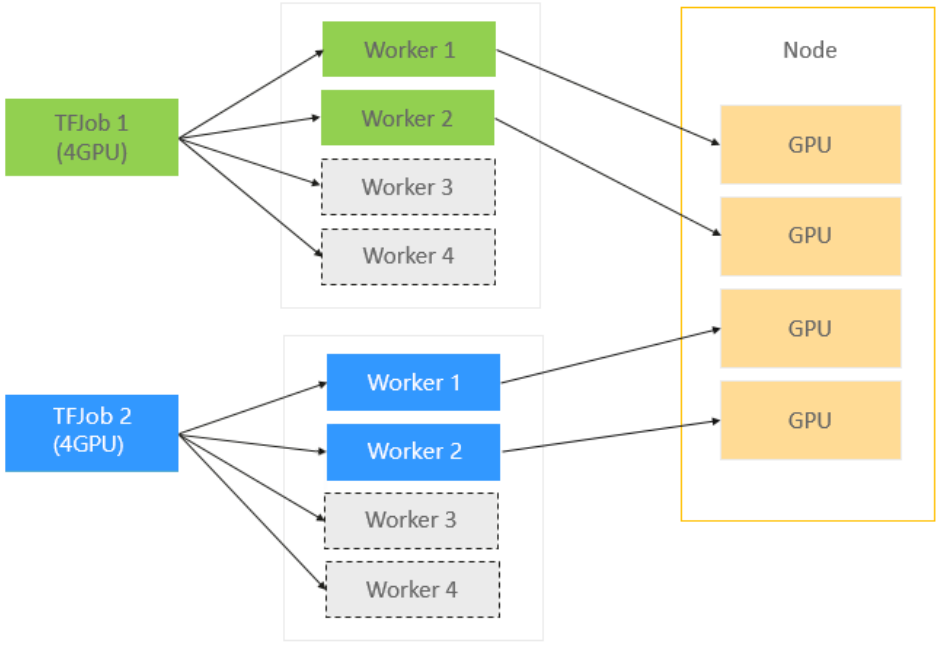
Kubernetes已经成为云原生应用编排、管理的事实标准， 越来越多的应用选择向Kubernetes迁移。人工智能和机器学习领域天然的包含大量的计算密集型任务，开发者非常愿意基于Kubernetes构建AI平台，充分利用Kubernetes提供的资源管理、应用编排、运维监控能力。

Kubernetes 存在的问题

Kubeflow在调度环境使用的是Kubernetes的默认调度器。而Kubernetes默认调度器最初主要是为长期运行的服务设计的，对于AI、大数据等批量和弹性调度方面还有很多的不足。主要存在以下问题：

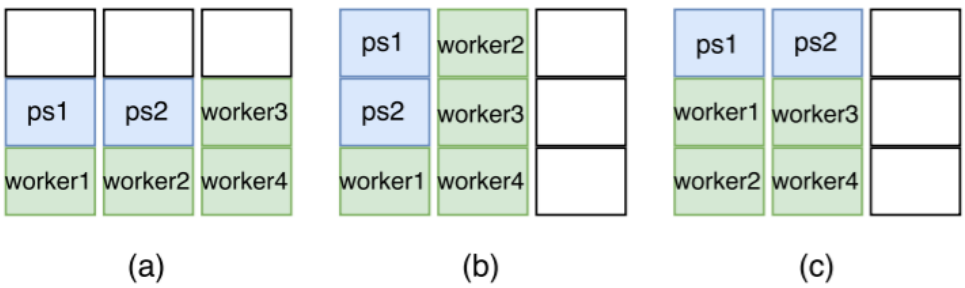
资源争抢问题

TensorFlow的作业包含Ps和Worker两种不同的角色，这两种角色的Pod要配合起来完成整个作业，如果只是运行一种角色Pod，整个作业是无法正常执行的，而默认调度器对于Pod调度是逐个进行的，对于Kubeflow作业TFJob的Ps和Worker是不感知的。在集群高负载（资源不足）的情况下，会出现多个作业各自分配到部分资源运行一部分Pod，而又无法正执行完成的状况，从而造成资源浪费。以下图为例，集群有4块GPU卡，TFJob1和TFJob2作业各自有4个Worker，TFJob1和TFJob2各自分配到2个GPU。但是TFJob1和TFJob2均需要4块GPU卡才能运行起来。这样TFJob1和TFJob2处于互相等待对方释放资源，这种死锁情况造成了GPU资源的浪费。



亲和调度问题

分布式训练中，Ps和Worker存在很频繁的数据交互，所以Ps和Worker之间的带宽直接影响了训练的效率。 Kubernetes默认调度器并不考虑Ps和Worker的这种逻辑关系，Ps和Worker是被随机调度的。如下图所示，2个TFJob（1个Ps + 2 Worker），使用默认调度器，有可能会出现（a）、（b）、（c）三种情况的任意一种情况，（c）才是最想要的调度结果。因为在(c)中，Ps和Worker可以利用本机网络提供传输效率，缩短训练时间。



Volcano 批量调度系统：加速 AI 计算的利器

Volcano是一款构建于Kubernetes之上的增强型高性能计算任务批量处理系统。作为一个面向高性能计算场景的平台，它弥补了Kubernetes在机器学习、深度学习、HPC、大数据计算等场景下的基本能力缺失，其中包括gang-schedule的调度能力、计算任务队列管理、task-topology和GPU亲和性调度。另外，Volcano在原生Kubernetes能力基础上对计算任务的批量创建及生命周期管理、fair-share、binpack调度等方面做了增强。Volcano充分解决了上文提到的Kubeflow分布式训练面临的问题。

Volcano更多信息请参见：**https://github.com/volcano-sh/volcano**。

Volcano 在华为云的应用

Kubeflow和Volcano两个开源项目的结合充分简化和加速了Kubernetes上AI计算进程。当前已经成为越来越多用户的最佳选择，应用于生产环境。Volcano目前已经应用于华为云CCE、CCI产品以及容器批量计算解决方案。未来Volcano会持续迭代演进，优化算法、增强调度能力如智能调度的支持，在推理场景增加GPU Share等特性的支持，进一步提升kubeflow批量训练和推理的效率。

实现典型分布式 AI 训练任务

下面将展示如何基于Kubeflow和Volcano，并使用MNIST数据集轻松的完成数字图像分类模型的分布式训练。

步骤1 登录CCE控制台，单击集群名称进入一个集群。

步骤2 在CCE集群上部署Volcano环境。

单击左侧栏目树中的“插件管理”，单击**Volcano**插件下方的“安装”，在安装插件页面中选择插件的规格配置，并单击“安装”。

步骤3 部署Mnist示例。

1. 下载kubeflow/examples到本地并根据环境选择指南，命令如下：
```
yum install git
git clone https://github.com/kubeflow/examples.git
```
2. 安装python3。
```
wget https://www.python.org/ftp/python/3.6.8/Python-3.6.8.tgz
tar -zxvf Python-3.6.8.tgz
cd Python-3.6.8 ./configure
make make install
```

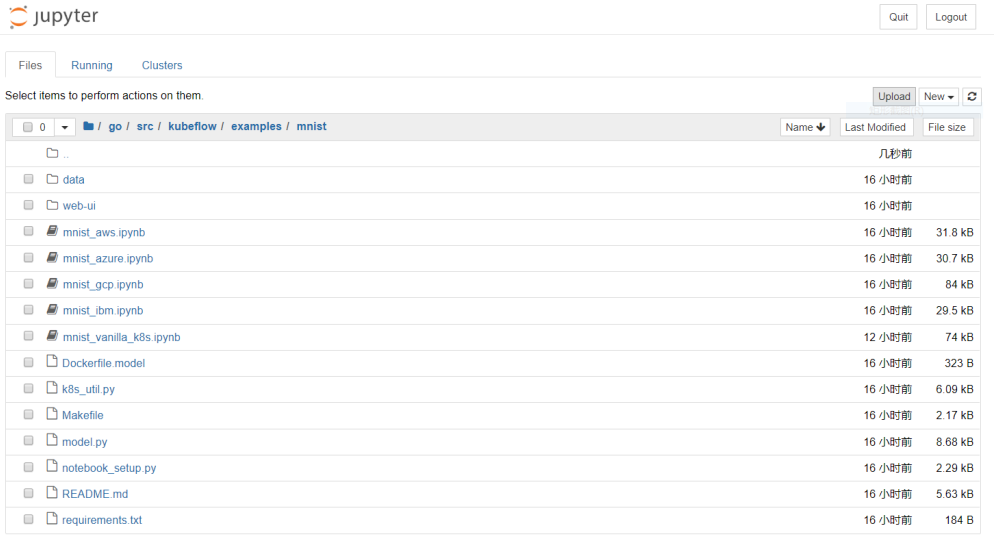
安装完成后执行如下命令检查是否安装成功

```
python3 -V
pip3 -V
```

3. 安装jupyter notebook并启动，命令如下：

```
pip3 install jupyter notebook
jupyter notebook --allow-root
```

4. Putty设置tunnel，远程连接notebook。
5. 连接成功后浏览器输入localhost:8000，登录notebook。



6. 根据jupyter的指引，创建分布式训练作业。通过简单的设置schedulerName字段的值为“volcano”，启用Volcano调度器（以下加粗字体部分）：

```
kind: TFJob
metadata:
  name: {train_name}
spec:
  schedulerName: volcano
  tfReplicaSpecs:
    Ps:
      replicas: {num_ps}
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          serviceAccount: default-editor
          containers:
          - name: tensorflow
            command:
              ...
            env:
              ...
            image: {image}
            workingDir: /opt
          restartPolicy: OnFailure
    Worker:
      replicas: 1
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          serviceAccount: default-editor
          containers:
          - name: tensorflow
            command:
```

```
    ...
    env:
    ...
    image: {image}
    workingDir: /opt
  restartPolicy: OnFailure
```

步骤4 提交作业，开始训练。

```
kubectl apply -f mnist.yaml
```

```
root@ecs-1953:~/tmp# kubectl get po
NAME                          READY   STATUS    RESTARTS   AGE
tensorflow-mnist-ps-0         1/1     Running   0          5s
tensorflow-mnist-worker-0     1/1     Running   0          5s
```

等待训练作业完成，通过Kubeflow的UI可以查询训练结果信息。至此就完成了一次简单的分布式训练任务。Kubeflow的借助TFJob简化了作业的配置。Volcano通过简单的增加一行配置就可以让用户启动组调度、Task-topology等功能来解决死锁、亲和性等问题，在大规模分布式训练情况下，可以有效的缩短整体训练时间。

----结束

15.2 CCE 部署使用 Caffe

15.2.1 预置条件

本实践提供在CCE上运行caffe的基础分类例子**https://github.com/BVLC/caffe/blob/master/examples/00-classification.ipynb**的过程。

OBS 存储数据预置

创建OBS桶，并确认以下文件夹已创建，文件已上传至指定位置（需要使用OBS Browser工具）。

例如：桶内文件路径/文件名，文件下载地址可至github中指定项目的指定路径下查找，示例如**1**、**2**所示。

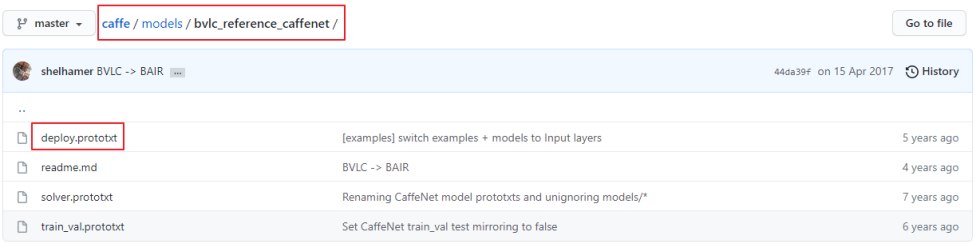
1. models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel

   https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet

name	caffemodel	caffemodel_url	license
BAIR/BVLC CaffeNet Model	bvlc_reference_caffenet.caffemodel	http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel	unrestricted

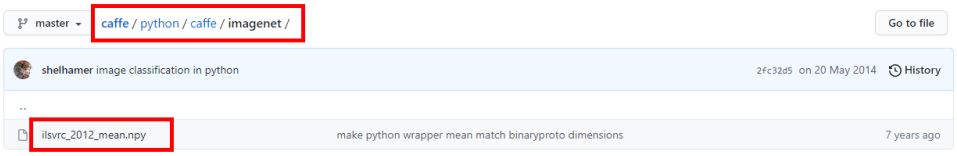
2. models/bvlc_reference_caffenet/deploy.prototxt

   https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet



3. python/caffe/imagenet/ilsvrc_2012_mean.npy

   https://github.com/BVLC/caffe/tree/master/python/caffe/imagenet



4. outputimg/

   创建一个空文件夹outputimg，以供存放输出文件。

5. examples/images/cat.jpg

   https://github.com/BVLC/caffe/blob/master/examples/00-classification.ipynb

   另存链接中里面小猫图片。

6. data/ilsvrc12/*

   https://github.com/BVLC/caffe/tree/master/data/ilsvrc12

   获取get_ilsvrc_aux.sh这个脚本并执行，这个脚本会下载一个压缩包并解压，执行完毕后将解压出来的所有文件上传至目录下。

7. caffeEx00.py

```
# set up Python environment: numpy for numerical routines, and matplotlib for plotting
import numpy as np
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
# display plots in this notebook
#%matplotlib inline

# set display defaults
plt.rcParams['figure.figsize'] = (10, 10)        # large images
plt.rcParams['image.interpolation'] = 'nearest'  # don't interpolate: show square pixels
plt.rcParams['image.cmap'] = 'gray'  # use grayscale output rather than a (potentially misleading)
color heatmap

# The caffe module needs to be on the Python path;
#  we'll add it here explicitly.
import sys
caffe_root = '/home/'  # this file should be run from {caffe_root}/examples (otherwise change this
line)
sys.path.insert(0, caffe_root + 'python')

import caffe
# If you get "No module named _caffe", either you have not built pycaffe or you have the wrong path.

import os
#if os.path.isfile(caffe_root + 'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel'):
#    print 'CaffeNet found.'
#else:
#    print 'Downloading pre-trained CaffeNet model...'
#    !../scripts/download_model_binary.py ../models/bvlc_reference_caffenet
```



```
caffe.set_mode_cpu()

model_def = caffe_root + 'models/bvlc_reference_caffenet/deploy.prototxt'
model_weights = caffe_root + 'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel'

net = caffe.Net(model_def,      # defines the structure of the model
                model_weights,  # contains the trained weights
                caffe.TEST)     # use test mode (e.g., don't perform dropout)

# load the mean ImageNet image (as distributed with Caffe) for subtraction
mu = np.load(caffe_root + 'python/caffe/imagenet/ilsvrc_2012_mean.npy')
mu = mu.mean(1).mean(1)  # average over pixels to obtain the mean (BGR) pixel values
print 'mean-subtracted values:', zip('BGR', mu)

# create transformer for the input called 'data'
transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})

transformer.set_transpose('data', (2,0,1))  # move image channels to outermost dimension
transformer.set_mean('data', mu)            # subtract the dataset-mean value in each channel
transformer.set_raw_scale('data', 255)      # rescale from [0, 1] to [0, 255]
transformer.set_channel_swap('data', (2,1,0))  # swap channels from RGB to BGR

# set the size of the input (we can skip this if we're happy
#  with the default; we can also change it later, e.g., for different batch sizes)
net.blobs['data'].reshape(50,        # batch size
                          3,         # 3-channel (BGR) images
                          227, 227)  # image size is 227x227

image = caffe.io.load_image(caffe_root + 'examples/images/cat.jpg')
transformed_image = transformer.preprocess('data', image)
plt.imshow(image)
plt.savefig(caffe_root + 'outputimg/img1.png')

# copy the image data into the memory allocated for the net
net.blobs['data'].data[...] = transformed_image

### perform classification
output = net.forward()

output_prob = output['prob'][0]  # the output probability vector for the first image in the batch

print 'predicted class is:', output_prob.argmax()

# load ImageNet labels
labels_file = caffe_root + 'data/ilsvrc12/synset_words.txt'
#if not os.path.exists(labels_file):
#    !../data/ilsvrc12/get_ilsvrc_aux.sh

labels = np.loadtxt(labels_file, str, delimiter='\t')

print 'output label:', labels[output_prob.argmax()]

# sort top five predictions from softmax output
top_inds = output_prob.argsort()[::-1][:5]  # reverse sort and take five largest items

print 'probabilities and labels:'
zip(output_prob[top_inds], labels[top_inds])
```

15.2.2 资源准备

在集群中添加 GPU 节点

步骤1 登录CCE控制台，单击已创建的集群，进入集群控制台。

步骤2 安装GPU插件。

1. 在左侧导航栏中选择“插件管理”，在右侧找到**gpu-beta**（或**gpu-device-plugin**），单击“安装”。

2. 在安装插件页面，设置插件关键参数。
   - Nvidia驱动：填写Nvidia驱动的下载链接，请根据GPU节点的显卡型号选择驱动。

   其余参数可保持默认，详情请参见**gpu-beta**。
3. 单击“安装”，安装插件的任务即可提交成功。

步骤3 创建GPU节点。

1. 在左侧菜单栏选择“节点管理”，单击右上角“创建节点”，在弹出的页面中配置节点的参数。
2. 选择一个“GPU加速型”的节点规格，其余参数请根据实际需求填写，详情请参见**创建节点**。
3. 完成配置后，单击“下一步：规格确认”，确认所设置的服务选型参数、规格和费用等信息，并单击“提交”，开始创建节点。

步骤4 待GPU节点创建完成后，可前往“节点列表”查看节点状态。

----结束

导入 OBS 存储卷

进入存储管理页面，导入**OBS存储数据预置**中创建的OBS存储卷。

15.2.3 Caffe 分类范例

本实践采用caffe官方的分类例子，地址为**https://github.com/BVLC/caffe/blob/master/examples/00-classification.ipynb**。

使用 CPU

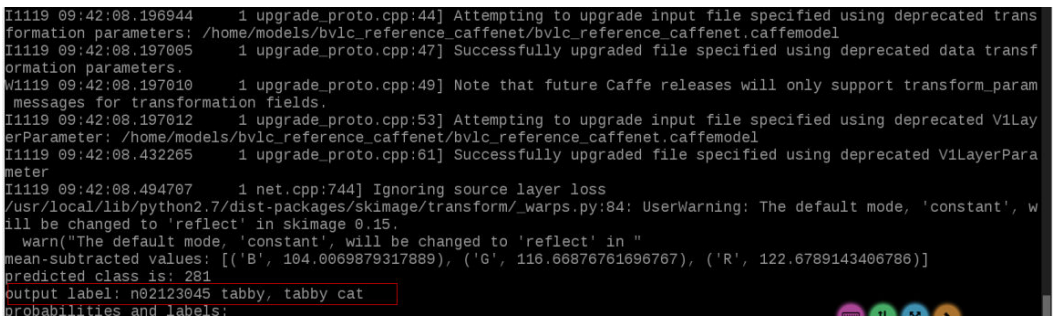
创建一个普通job，镜像输入第三方镜像bvlc/caffe:cpu，设置对应的容器规格。

启动命令添加python /home/caffeEx00.py。

挂载刚刚导入的OBS存储盘：

单击“创建”。等待job执行完成，进入OBS存储盘的outputimg下，可以看到推理使用的图片。

登录**在集群中添加GPU节点**添加的节点，执行**docker logs {*容器id*}**查看归类结果，可以看到结果：tabby cat。

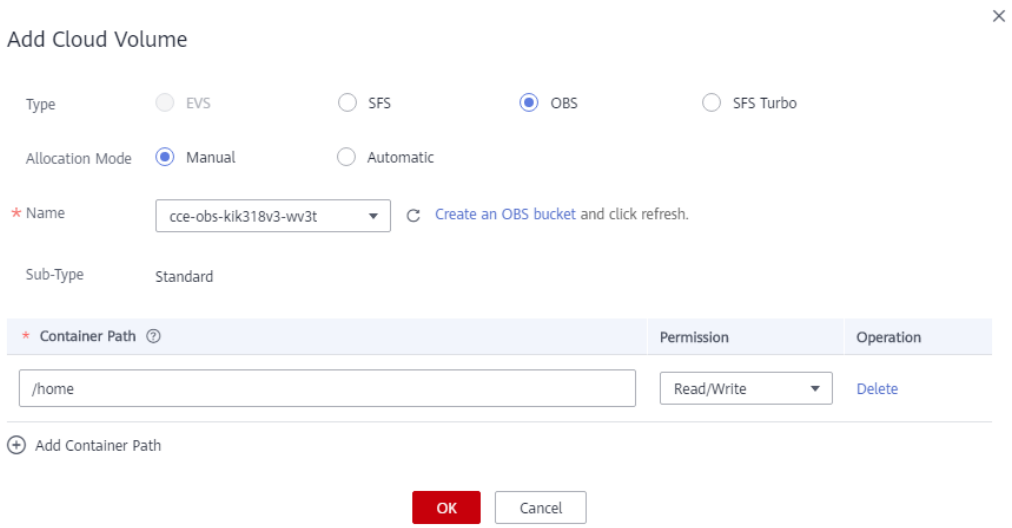


使用 GPU

创建一个普通job，镜像输入第三方镜像bvlc/caffe:gpu，设置对应的容器规格。

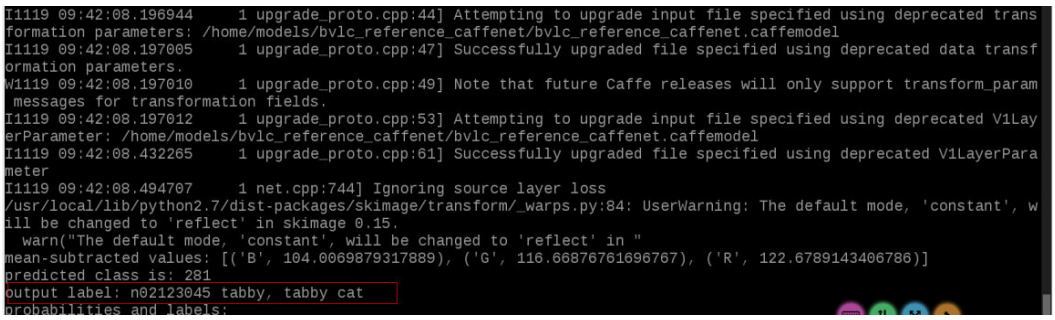
启动命令添加python /home/caffeEx00_GPU.py。

挂载刚刚导入的OBS存储盘：



单击“创建”。等待job执行完成，进入OBS存储盘的outputimg下，可以看到推理使用的图片。

登录**在集群中添加GPU节点**添加的节点，执行docker logs {容器id}查看归类结果，可以看到结果：tabby cat。



15.3 CCE 部署使用 Tensorflow

资源准备

- 购买CCE集群，购买GPU节点并使用gpu-beta插件安装显卡驱动。
- 在集群下添加一个对象存储卷。

数据预置

从**https://github.com/zalandoresearch/fashion-mnist**下载数据。

Get the Data

Many ML libraries already include Fashion-MNIST data/API, give it a try!

You can use direct links to download the dataset. The data is stored in the **same** format as the original MNIST data.

Name	Content	Examples	Size	Link	MD5 Checksum
train-images-idx3-ubyte.gz	training set images	60,000	26 MBytes	Download	8d4fb7e6c68d591d4c3dfef9ec88bf0d
train-labels-idx1-ubyte.gz	training set labels	60,000	29 KBytes	Download	25c81989df183df01b3e8a0aad5dffbe
t10k-images-idx3-ubyte.gz	test set images	10,000	4.3 MBytes	Download	bef4ecab320f06d8554ea6380940ec79
t10k-labels-idx1-ubyte.gz	test set labels	10,000	5.1 KBytes	Download	bb300cfdad3c16e7a12a480ee83cd310

获取tensorflow的ML范例，加以简单的修改。

basicClass.py

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import gzip
from tensorflow.python.keras.utils import get_file
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt

print(tf.__version__)

#fashion_mnist = keras.datasets.fashion_mnist
#(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

def load_data():
    base = "file:////home/data/"
    files = [
        'train-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz',
        't10k-labels-idx1-ubyte.gz', 't10k-images-idx3-ubyte.gz'
    ]

    paths = []
    for fname in files:
        paths.append(get_file(fname, origin=base + fname))

    with gzip.open(paths[0], 'rb') as lbpath:
        y_train = np.frombuffer(lbpath.read(), np.uint8, offset=8)

    with gzip.open(paths[1], 'rb') as imgpath:
        x_train = np.frombuffer(
            imgpath.read(), np.uint8, offset=16).reshape(len(y_train), 28, 28)

    with gzip.open(paths[2], 'rb') as lbpath:
        y_test = np.frombuffer(lbpath.read(), np.uint8, offset=8)

    with gzip.open(paths[3], 'rb') as imgpath:
        x_test = np.frombuffer(
            imgpath.read(), np.uint8, offset=16).reshape(len(y_test), 28, 28)

    return (x_train, y_train), (x_test, y_test)

(train_images, train_labels), (test_images, test_labels) = load_data()
```

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.savefig('/home/img/basicimg1.png')

train_images = train_images / 255.0

test_images = test_images / 255.0

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.savefig('/home/img/basicimg2.png')

model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)

predictions = model.predict(test_images)

def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array[i], true_label[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)
```



```
  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')

i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.savefig('/home/img/basicimg3.png')

i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.savefig('/home/img/basicimg4.png')

# Plot the first X test images, their predicted label, and the true label
# Color correct predictions in blue, incorrect predictions in red
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
plt.savefig('/home/img/basicimg5.png')
```

进入刚刚创建的OBS桶页面，创建文件夹data和img，并将basicClass.py上传。

进入data文件夹，将刚刚下载的四个gz文件上传。

机器学习范例

本篇范例采用tensorflow官网的ml example，可参考**https://www.tensorflow.org/tutorials/keras/classification?hl=zh-cn**。

创建一个普通job，镜像输入第三方镜像tensorflow/tensorflow:1.15.5-gpu，设置对应的容器规格。

启动命令添加 pip install matplotlib;python /home/basicClass.py 。

挂载刚刚创建的OBS存储盘：

单击“创建”。等待job执行完成，进入OBS页面，可以查看到以图片形式展示的执行结果。

通过kubectl创建可以按如下YAML执行。

```
kind: Job
apiVersion: batch/v1
metadata:
  name: testjob
  namespace: default
spec:
  parallelism: 1
  completions: 1
  backoffLimit: 6
  template:
    metadata:
      name: testjob
    spec:
      volumes:
        - name: cce-obs-tensorflow
          persistentVolumeClaim:
            claimName: cce-obs-tensorflow
      containers:
        - name: container-0
          image: 'tensorflow/tensorflow:1.15.5-gpu'
          restartPolicy: OnFailure
          command:
            - /bin/bash
          args:
            - '-c'
            - pip install matplotlib;python /home/basicClass.py
          resources:
            limits:
              cpu: '2'
              memory: 4Gi
              nvidia.com/gpu: '1'
            requests:
              cpu: '2'
              memory: 4Gi
              nvidia.com/gpu: '1'
          volumeMounts:
            - name: cce-obs-tensorflow
              mountPath: /home
          imagePullPolicy: IfNotPresent
```

```
    imagePullSecrets:
      - name: default-secret
```

15.4 CCE 部署使用 Flink

本实践提供在华为云CCE集群中部署flink集群，并执行WordCount任务的流程说明。

预置条件

已创建CCE集群，且集群下有可用节点，集群内节点已绑定弹性IP，且配置了kubectl命令行工具。

部署流程

主要参照**https://ci.apache.org/projects/flink/flink-docs-stable/ops/deployment/kubernetes.html**。

创建 flink session cluster

根据上述网页中的指引，创建两个deploy、一个service和一个configmap即可。

flink-configuration-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: flink-config
  labels:
    app: flink
data:
  flink-conf.yaml: |+
    jobmanager.rpc.address: flink-jobmanager
    taskmanager.numberOfTaskSlots: 2
    blob.server.port: 6124
    jobmanager.rpc.port: 6123
    taskmanager.rpc.port: 6122
    queryable-state.proxy.ports: 6125
    jobmanager.memory.process.size: 1600m
    taskmanager.memory.process.size: 1728m
    parallelism.default: 2
  log4j-console.properties: |+
    # This affects logging for both user code and Flink
    rootLogger.level = INFO
    rootLogger.appenderRef.console.ref = ConsoleAppender
    rootLogger.appenderRef.rolling.ref = RollingFileAppender

    # Uncomment this if you want to _only_ change Flink's logging
    #logger.flink.name = org.apache.flink
    #logger.flink.level = INFO

    # The following lines keep the log level of common libraries/connectors on
    # log level INFO. The root logger does not override this. You have to manually
    # change the log levels here.
    logger.akka.name = akka
    logger.akka.level = INFO
    logger.kafka.name= org.apache.kafka
    logger.kafka.level = INFO
    logger.hadoop.name = org.apache.hadoop
    logger.hadoop.level = INFO
    logger.zookeeper.name = org.apache.zookeeper
    logger.zookeeper.level = INFO

    # Log all infos to the console
```



```
  appender.console.name = ConsoleAppender
  appender.console.type = CONSOLE
  appender.console.layout.type = PatternLayout
  appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x - %m%n

  # Log all infos in the given rolling file
  appender.rolling.name = RollingFileAppender
  appender.rolling.type = RollingFile
  appender.rolling.append = false
  appender.rolling.fileName = ${sys:log.file}
  appender.rolling.filePattern = ${sys:log.file}.%i
  appender.rolling.layout.type = PatternLayout
  appender.rolling.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x - %m%n
  appender.rolling.policies.type = Policies
  appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
  appender.rolling.policies.size.size=100MB
  appender.rolling.strategy.type = DefaultRolloverStrategy
  appender.rolling.strategy.max = 10

  # Suppress the irrelevant (wrong) warnings from the Netty channel handler
  logger.netty.name = org.apache.flink.shaded.akka.org.jboss.netty.channel.DefaultChannelPipeline
  logger.netty.level = OFF
```

jobmanager-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: flink-jobmanager
spec:
  type: ClusterIP
  ports:
  - name: rpc
    port: 6123
  - name: blob-server
    port: 6124
  - name: webui
    port: 8081
  selector:
    app: flink
    component: jobmanager
```

jobmanager-session-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flink-jobmanager
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flink
      component: jobmanager
  template:
    metadata:
      labels:
        app: flink
        component: jobmanager
    spec:
      containers:
      - name: jobmanager
        image: flink:1.11.0-scala_2.11
        args: ["jobmanager"]
        ports:
        - containerPort: 6123
          name: rpc
        - containerPort: 6124
          name: blob-server
        - containerPort: 8081
```



```
        name: webui
      livenessProbe:
        tcpSocket:
          port: 6123
        initialDelaySeconds: 30
        periodSeconds: 60
      volumeMounts:
      - name: flink-config-volume
        mountPath: /opt/flink/conf
      securityContext:
        runAsUser: 9999  # refers to user _flink_ from official flink image, change if necessary
    volumes:
    - name: flink-config-volume
      configMap:
        name: flink-config
        items:
        - key: flink-conf.yaml
          path: flink-conf.yaml
        - key: log4j-console.properties
          path: log4j-console.properties
```

taskmanager-session-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flink-taskmanager
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flink
      component: taskmanager
  template:
    metadata:
      labels:
        app: flink
        component: taskmanager
    spec:
      containers:
      - name: taskmanager
        image: flink:1.11.0-scala_2.11
        args: ["taskmanager"]
        ports:
        - containerPort: 6122
          name: rpc
        - containerPort: 6125
          name: query-state
        livenessProbe:
          tcpSocket:
            port: 6122
          initialDelaySeconds: 30
          periodSeconds: 60
        volumeMounts:
        - name: flink-config-volume
          mountPath: /opt/flink/conf/
        securityContext:
          runAsUser: 9999  # refers to user _flink_ from official flink image, change if necessary
      volumes:
      - name: flink-config-volume
        configMap:
          name: flink-config
          items:
          - key: flink-conf.yaml
            path: flink-conf.yaml
          - key: log4j-console.properties
            path: log4j-console.properties
```

kubectl create -f flink-configuration-configmap.yaml

kubectl create -f jobmanager-service.yaml

kubectl create -f jobmanager-session-deployment.yaml

kubectl create -f taskmanager-session-deployment.yaml

```
[root@ ~]# kubectl get cm |grep flink
flink-config                    2      17m
```

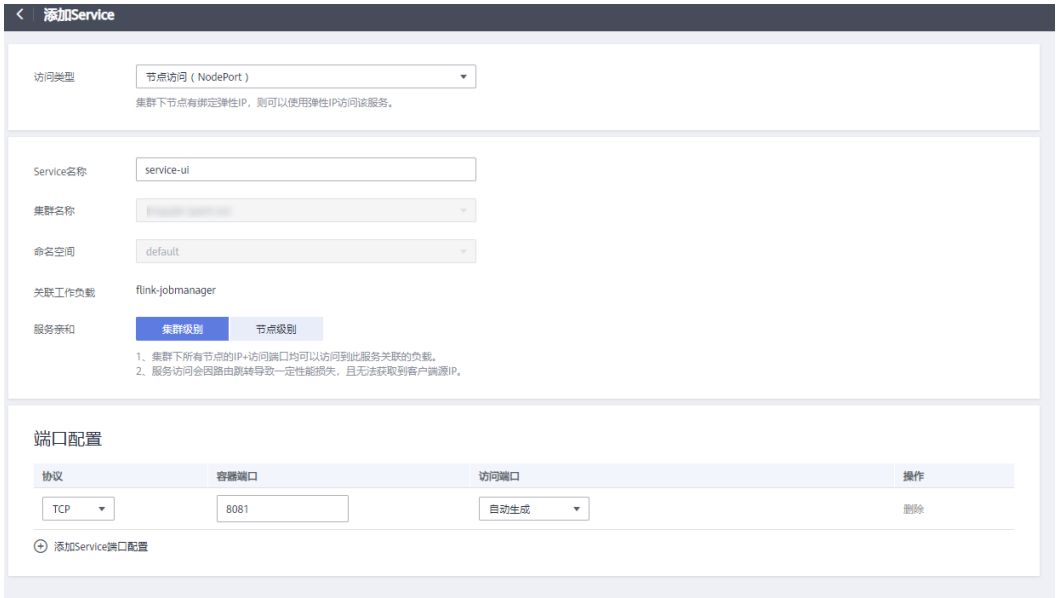
```
[root@ ~]# kubectl get svc |grep flink
flink-jobmanager                ClusterIP   10.247.4.151    <none>        6123/TCP,6124/TCP,8081/TCP   20m
```

```
[root@ ~]# kubectl get pod -owide|grep flink
flink-jobmanager-b8545cf98-rnf5m      1/1     Running   0          15m   172.16.1.143   192.168.15.231   <none>           <none>
flink-taskmanager-7674f748bd-h6ztw    1/1     Running   0          15m   172.16.1.144   192.168.15.231   <none>           <none>
flink-taskmanager-7674f748bd-v6tss    1/1     Running   0          15m   172.16.1.145   192.168.15.231   <none>           <none>
```

对外发布服务

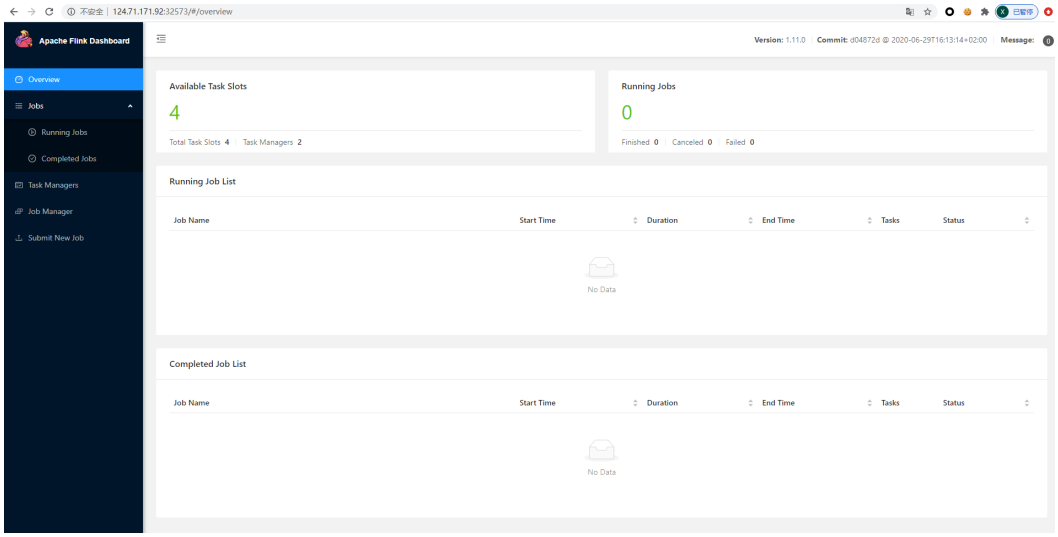
登录华为云CCE页面，进入“工作负载 > 无状态负载”页面，选择flink-jobmanager，单击“访问方式”页签。

单击“添加service”，选择节点访问，输入容器端口为8081。



访问对外发布的链接：

可以看到flink的dashboard页面：

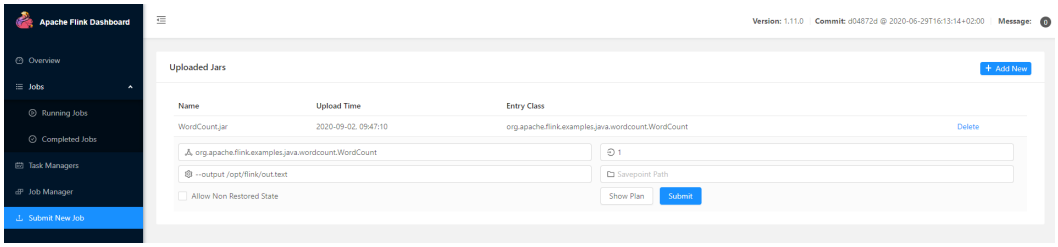


执行 flink 任务

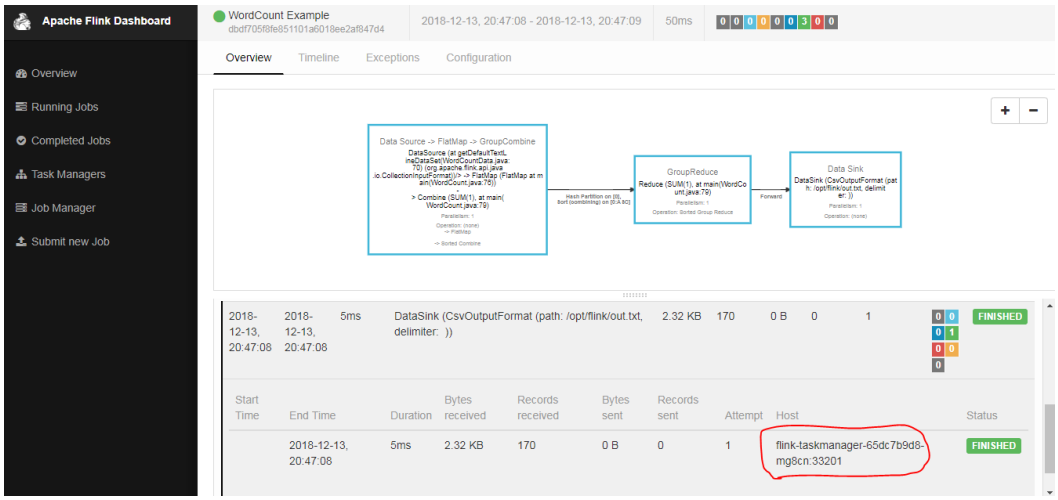
使用官方范例的WordCount.jar文件来执行flink任务。

下载 **https://archive.apache.org/dist/flink/flink-1.11.0/flink-1.11.0-bin-scala_2.11.tgz**，解压后examples\streaming下有WordCount.jar包。

添加Jar包，将wordCount.jar上传到页面，并填入如下参数：



执行计算任务：



等待任务执行完毕后，查看任务状态，进入指定的taskmanager查看/opt/flink/out文件中是否正确输出了每个单词出现的次数。

```
[root@ ~]# kubectl exec -ti flink-taskmanager-65dc7b9d8-mg8cn bash
root@flink-taskmanager-65dc7b9d8-mg8cn:/opt/flink#
root@flink-taskmanager-65dc7b9d8-mg8cn:/opt/flink# ls
LICENSE  NOTICE  README.txt  bin  conf  examples  lib  log  opt  out  out.txt
root@flink-taskmanager-65dc7b9d8-mg8cn:/opt/flink#
root@flink-taskmanager-65dc7b9d8-mg8cn:/opt/flink#
root@flink-taskmanager-65dc7b9d8-mg8cn:/opt/flink# cat out.txt
a 5
action 1
after 1
against 1
all 2
and 12
arms 1
```

15.5 ClickHouse on CCE 部署指南

15.5.1 资源规划

clickhouse-operator用于K8s中创建、配置、管理ClickHouse集群 。

本文指导在CCE集群中安装部署clickhouse-operator并举例创建clickhouse集群资源 ，参考文档：**https://github.com/Altinity/clickhouse-operator**。

clickhouse-operator安装要求K8s版本为1.15.11+，本文使用1.19。

集群类型	CCE集群
集群版本	1.19
Region	上海一
Docker版本	18.09.0.91
网络模型	VPC网络
服务转发模式	iptables

15.5.2 配置 kubectl 工具

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

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

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

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

15.5.3 部署 clickhouse operator

kubectl apply -f https://github.com/Altinity/clickhouse-operator/blob/master/deploy/operator/clickhouse-operator-install-bundle.yaml

```
[root@click-house-49966 ~]# kubectl apply -f https://raw.githubusercontent.com/Altinity/clickhouse-operator/master/deploy/operator/clickhouse-operator-install.yaml
customresourcedefinition.apiextensions.k8s.io/clickhouseinstallations.clickhouse.altinity.com created
customresourcedefinition.apiextensions.k8s.io/clickhouseinstallationtemplates.clickhouse.altinity.com created
customresourcedefinition.apiextensions.k8s.io/clickhouseoperatorconfigurations.clickhouse.altinity.com created
serviceaccount/clickhouse-operator created
clusterrolebinding.rbac.authorization.k8s.io/clickhouse-operator-kube-system created
configmap/etc-clickhouse-operator-files created
configmap/etc-clickhouse-operator-confd-files created
configmap/etc-clickhouse-operator-configd-files created
configmap/etc-clickhouse-operator-templatesd-files created
configmap/etc-clickhouse-operator-usersd-files created
deployment.apps/clickhouse-operator created
service/clickhouse-operator-metrics created
```

一段时间后，查看资源运行情况：

kubectl get pod -n kube-system|grep clickhouse

```
[root@click-house-49966 ~]# kubectl get pod -n kube-system|grep clickhouse
clickhouse-operator-5d56f7b4b7-zmp98      2/2     Running   0          16m
```

15.5.4 示例

创建 namespace

为方便验证基本功能创建test-clickhouse-operator的namespace。

kubectl create namespace test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl create namespace test-clickhouse-operator
namespace/test-clickhouse-operator created
```

Simple example

本示例来源于**https://github.com/Altinity/clickhouse-operator/blob/master/docs/chi-examples/01-simple-layout-01-1shard-1repl.yaml**。

yaml文件如下：

```
apiVersion: "clickhouse.altinity.com/v1"
kind: "ClickHouseInstallation"
metadata:
  name: "simple-01"
```

使用如下命令创建：

kubectl apply -n test-clickhouse-operator -f https://raw.githubusercontent.com/Altinity/clickhouse-operator/master/docs/chi-examples/01-simple-layout-01-1shard-1repl.yaml

```
[root@click-house-49966 ~]# kubectl apply -n test-clickhouse-operator -f https://raw.githubusercontent.com/Altinity/clickhouse-operator/master/docs/chi-examples/01-simple-layout-01-1shard-1repl.yaml
clickhouseinstallation.clickhouse.altinity.com/simple-01 created
```

一段时间后，查看资源运行情况：

kubectl get pod -n test-clickhouse-operator

kubectl get service -n test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl get pod -n test-clickhouse-operator
NAME                          READY   STATUS    RESTARTS   AGE
chi-simple-01-cluster-0-0-0   1/1     Running   0          2m27s
[root@click-house-49966 ~]# kubectl get service -n test-clickhouse-operator
NAME                          TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                       AGE
chi-simple-01-cluster-0-0     ClusterIP      None           <none>        8123/TCP,9000/TCP,9009/TCP    2m3s
clickhouse-simple-01          LoadBalancer   10.247.1.133   <pending>     8123:30485/TCP,9000:30301/TCP   2m55s
```

连接ClickHouse Database：

kubectl -n test-clickhouse-operator exec -ti chi-simple-01-cluster-0-0-0 -- clickhouse-client

```
[root@click-house-49966 ~]# kubectl -n test-clickhouse-operator exec -ti chi-simple-01-cluster-0-0-0 -- clickhouse-client
ClickHouse client version 20.8.2.3 (official build).
Connecting to localhost:9000 as user default.
Connected to ClickHouse server version 20.8.2 revision 54438.

chi-simple-01-cluster-0-0-0.chi-simple-01-cluster-0-0.test-clickhouse-operator.svc.cluster.local :)
```

Simple Persistent Volume Example

本示例来源于**https://github.com/Altinity/clickhouse-operator/blob/master/docs/chi-examples/03-persistent-volume-01-default-volume.yaml**。

在CCE使用PVC时，yaml文件需做一些适配：

- 使用云硬盘存储卷EVS
  a. 创建storageClass

     CCE默认支持的csi-disk为SAS类型的云硬盘。 若想使用超高I/O类型的云硬盘，需要创建对应的storageClass。

     vim csi-disk-ssd.yaml

     复制以下内容：

     ```
     allowVolumeExpansion: true
     apiVersion: storage.k8s.io/v1
     kind: StorageClass
     metadata:
       name: csi-disk-ssd
     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
     ```

     保存后退出：

     kubectl create -f csi-disk-ssd.yaml

  b. accessModes需填写为ReadWriteOnce。

  c. 添加storageClassName: csi-disk-ssd。

- 使用文件存储卷SFS

  a. accessModes需填写为ReadWriteMany。

  b. 添加storageClassName: csi-nas。

  下面以文件存储卷SFS为例，yaml文件如下：

```
apiVersion: "clickhouse.altinity.com/v1"
kind: "ClickHouseInstallation"
metadata:
  name: "pv-simple"
spec:
  defaults:
    templates:
      dataVolumeClaimTemplate: data-volume-template
      logVolumeClaimTemplate: log-volume-template
  configuration:
    clusters:
      - name: "simple"
        layout:
          shardsCount: 1
          replicasCount: 1
  templates:
    volumeClaimTemplates:
      - name: data-volume-template
        spec:
          accessModes:
            - ReadWriteMany
          resources:
            requests:
              storage: 10Gi
          storageClassName: csi-nas
      - name: log-volume-template
        spec:
          accessModes:
            - ReadWriteMany
          resources:
            requests:
              storage: 10Gi
          storageClassName: csi-nas
```

使用如下命令创建：

kubectl -n test-clickhouse-operator create -f 03-persistent-volume-01-default-volume.yaml

```
[root@click-house-49966 ~]# kubectl -n test-clickhouse-operator create -f 03-persistent-volume-01-default-volume.yaml
clickhouseinstallation.clickhouse.altinity.com/pv-simple created
```

一段时间后，查看资源运行情况：

kubectl get pvc -n test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl get pvc -n test-clickhouse-operator
NAME                                          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-volume-template-chi-pv-simple-simple-0-0-0   Bound    pvc-b3f93d71-ee6a-4d19-96f4-cc93ed69a023   10Gi       RWX            csi-nas        28s
log-volume-template-chi-pv-simple-simple-0-0-0    Bound    pvc-a227b285-e955-414c-bb7f-89b7400bb0fa   10Gi       RWX            csi-nas        28s
```

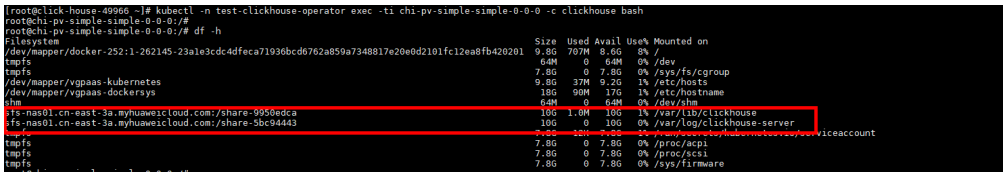
kubectl get pod -n test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl get pod -n test-clickhouse-operator
NAME                              READY   STATUS    RESTARTS   AGE
chi-pv-simple-simple-0-0-0        2/2     Running   0          4m10s
chi-simple-01-cluster-0-0-0       1/1     Running   0          4h11m
```

查看存储卷挂载情况：

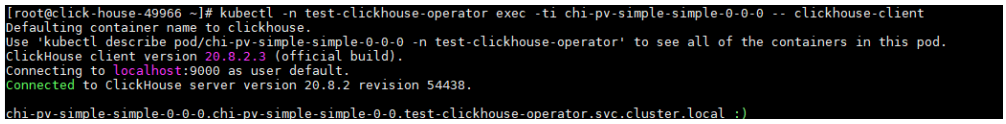
kubectl -n test-clickhouse-operator exec -ti chi-pv-simple-simple-0-0-0 -c clickhouse bash

df -h



连接ClickHouse Database：

kubectl -n test-clickhouse-operator exec -ti chi-pv-simple-simple-0-0-0 -- clickhouse-client



Simple Load Balancer Example

```
apiVersion: "clickhouse.altinity.com/v1"
kind: "ClickHouseInstallation"
metadata:
  name: "ck-elb"
spec:
  defaults:
    templates:
      dataVolumeClaimTemplate: data-volume-nas
      serviceTemplate: chi-service-elb
  configuration:
    clusters:
      - name: "ck-elb"
        templates:
          podTemplate: pod-template-with-nas
        layout:
          shardsCount: 1
          replicasCount: 1
  templates:
    podTemplates:
      - name: pod-template-with-nas
        spec:
          containers:
            - name: clickhouse
              image: yandex/clickhouse-server:21.6.3.14
              volumeMounts:
                - name: data-volume-nas
                  mountPath: /var/lib/clickhouse
    volumeClaimTemplates:
      - name: data-volume-nas
        spec:
          accessModes:
            - ReadWriteMany
          resources:
            requests:
              storage: 20Gi
          storageClassName: csi-nas
    serviceTemplates:
      - name: chi-service-elb
        metadata:
          annotations:
            kubernetes.io/elb.class: union
            kubernetes.io/elb.autocreate: >-
              {"type":"public","bandwidth_name":"cce-bandwidth-
ck","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_type":"5_bgp"}
        spec:
          ports:
            - name: http
              port: 8123
```

```
    - name: client
      port: 9000
  type: LoadBalancer
```

添加加粗部分内容。其中annotations kubernetes.io/elb.autocreate支持以下参数：

参数	参数类型	描述
name	String	自动创建的负载均衡的名称。 取值范围：1-64个字符，小写字母，数字，下划线，小写字母开头，小写字母或者数字结尾。
type	String	负载均衡实例网络类型，公网或者私网。 ● public ： 公网型负载均衡 ● inner ： 私网型负载均衡
bandwidth_name	String	带宽的名称，默认值为：cce-bandwidth-******。 取值范围：1-64个字符，小写字母，数字，下划线，小写字母开头，小写字母或者数字结尾。
bandwidth_chargemode	String	带宽付费模式。 ● bandwidth ： 按带宽计费 ● traffic ： 按流量计费
bandwidth_size	Integer	带宽大小
bandwidth_sharetype	String	带宽共享方式。 ● PER ： 独享带宽
eip_type	String	弹性公网IP类型

15.6 Spark on CCE with OBS 安装使用指南

15.6.1 安装 Spark

前提条件

您需要准备一台可访问公网的Linux机器，节点规格建议为4U8G及以上。

配置 JDK

以CentOS系统为例，安装JDK 1.8。

步骤1 查询可用的JDK版本。

```
yum -y list java*
```

步骤2 选择安装JDK 1.8。

```
yum install -y java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

步骤3 安装完成后，查看JDK版本。

```
# java -version
openjdk version "1.8.0_382"
```



```
OpenJDK Runtime Environment (build 1.8.0_382-b05)
OpenJDK 64-Bit Server VM (build 25.382-b05, mixed mode)
```

步骤4 添加环境变量。

1. Linux环境变量配置在/etc/profile文件中。
   ```
   vim /etc/profile
   ```
2. 在编辑模式下，在文件最后添加如下内容：
   ```
   JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.382.b05-1.el7_9.x86_64
   PATH=$PATH:$JAVA_HOME/bin
   CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
   export JAVA_HOME  PATH  CLASSPATH
   ```
3. 保存并关闭profile文件，执行如下命令使其生效。
   ```
   source /etc/profile
   ```
4. 查看JDK环境变量。
   ```
   echo $JAVA_HOME
   echo $PATH
   echo $CLASSPATH
   ```

----结束

获取 Spark 包

由于OBS适配hadoop2.8.3和3.1.1版本，本文使用3.1.1。

步骤1 下载v3.1.1版本的Spark。如环境中未安装git，您需要先执行**yum install git**安装git。

```
git clone -b v3.1.1 https://github.com/apache/spark.git
```

步骤2 修改/dev/make-distribution.sh文件，指定Spark版本，目的是为了让编译的时候跳过检测。

1. 使用搜索找到 VERSION 所在行，查看版本号所在行数。
   ```
   cat ./spark/dev/make-distribution.sh |grep -n '^VERSION=' -A18
   ```
2. 显示129行到147行，将这些内容注释，并指定版本。
   ```
   sed -i '129,147s/^/#/g' ./spark/dev/make-distribution.sh
   sed -i '148a
   VERSION=3.1.3\nSCALA_VERSION=2.12\nSPARK_HADOOP_VERSION=3.1.1\nSPARK_HIVE=1' ./
   spark/dev/make-distribution.sh
   ```

步骤3 下载依赖包。

```
wget https://archive.apache.org/dist//maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
tar -zxvf apache-maven-3.6.3-bin.tar.gz && mv apache-maven-3.6.3 ./spark/build
```

步骤4 执行如下命令进行编译。

```
./spark/dev/make-distribution.sh --name hadoop3.1 --tgz -Pkubernetes -Pyarn -Dhadoop.version=3.1.1
```

步骤5 编译可能需要一定时间，请耐心等待。编译完成后，软件包名称为spark-3.1.3-bin-hadoop3.1.tgz。

----结束

配置 Spark 运行环境

为了操作简便，使用root用户，并将编译出的软件包spark-3.1.3-bin-hadoop3.1.tgz放置于操作节点/root目录下。

步骤1 将软件包移动至/root目录。

```
mv ./spark/spark-3.1.3-bin-hadoop3.1.tgz /root
```

步骤2 执行命令安装Spark。


```
tar -zxvf spark-3.1.3-bin-hadoop3.1.tgz
mv spark-3.1.3-bin-hadoop3.1 spark-obs
cat >> ~/.bashrc <<EOF
PATH=/root/spark-obs/bin:\$PATH
PATH=/root/spark-obs/sbin:\$PATH
export SPARK_HOME=/root/spark-obs
EOF

source ~/.bashrc
```

步骤3 此时已经可以使用spark-submit等二进制，执行以下命令查看所用的Spark版本。

```
spark-submit --version
```

----结束

配置 Spark 对接 OBS

步骤1 获取华为云OBS jar包。本文使用hadoop-huaweicloud-3.1.1-hw-45.jar，获取地址：**https://github.com/huaweicloud/obsa-hdfs/tree/master/release**。

```
wget https://github.com/huaweicloud/obsa-hdfs/releases/download/v45/hadoop-huaweicloud-3.1.1-hw-45.jar
```

步骤2 复制华为云OBS jar包到相应目录。

```
cp hadoop-huaweicloud-3.1.1-hw-45.jar /root/spark-obs/jars/
```

步骤3 修改Spark配置项。为了对接OBS，需要为Spark添加对应的配置项。

1. 获取AK/SK，详情请参见**访问密钥**。
2. 修改AK_OF_YOUR_ACCOUNT / SK_OF_YOUR_ACCOUNT / OBS_ENDPOINT为实际值。
   - AK_OF_YOUR_ACCOUNT：上一步中获取的AK。
   - SK_OF_YOUR_ACCOUNT：上一步中获取的SK。
   - OBS_ENDPOINT：OBS的Endpoint，可前往**地区和终端节点**查询。

```
cp ~/spark-obs/conf/spark-defaults.conf.template ~/spark-obs/conf/spark-defaults.conf

cat >> ~/spark-obs/conf/spark-defaults.conf <<EOF
spark.hadoop.fs.obs.readahead.inputstream.enabled=true
spark.hadoop.fs.obs.buffer.max.range=6291456
spark.hadoop.fs.obs.buffer.part.size=2097152
spark.hadoop.fs.obs.threads.read.core=500
spark.hadoop.fs.obs.threads.read.max=1000
spark.hadoop.fs.obs.write.buffer.size=8192
spark.hadoop.fs.obs.read.buffer.size=8192
spark.hadoop.fs.obs.connection.maximum=1000
spark.hadoop.fs.obs.access.key=AK_OF_YOUR_ACCOUNT
spark.hadoop.fs.obs.secret.key=SK_OF_YOUR_ACCOUNT
spark.hadoop.fs.obs.endpoint=OBS_ENDPOINT
spark.hadoop.fs.obs.buffer.dir=/root/hadoop-obs/obs-cache
spark.hadoop.fs.obs.impl=org.apache.hadoop.fs.obs.OBSFileSystem
spark.hadoop.fs.obs.connection.ssl.enabled=false
spark.hadoop.fs.obs.fast.upload=true
spark.hadoop.fs.obs.socket.send.buffer=65536
spark.hadoop.fs.obs.socket.recv.buffer=65536
spark.hadoop.fs.obs.max.total.tasks=20
spark.hadoop.fs.obs.threads.max=20
spark.kubernetes.container.image.pullSecrets=default-secret
EOF
```

----结束

预置镜像到 SWR

在K8s内运行Spark任务，需要构建相同版本的Spark容器镜像，并将其上传到SWR。在编译Spark时，会自动生成配套的Dockerfile文件，您可通过此文件制作镜像并上传至SWR。

步骤1 制作镜像。

```
cd ~/spark-obs
docker build -t spark:3.1.3-obs --build-arg spark_uid=0 -f kubernetes/dockerfiles/spark/Dockerfile .
```

步骤2 上传镜像。

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. 登录制作镜像的节点，复制登录指令。

   ```
   docker tag [{镜像名称}:{版本名称}] swr.ap-southeast-1.myhuaweicloud.com/{组织名称}/{镜像名称}:{版本名称}
   docker push swr.ap-southeast-1.myhuaweicloud.com/{组织名称}/{镜像名称}:{版本名称}
   ```

   记录下镜像的访问地址以供后文填写。

   例如记录下地址为：swr.ap-southeast-1.myhuaweicloud.com/dev-container/spark:3.1.3-obs

----结束

配置 Spark History Server

步骤1 修改~/spark-obs/conf/spark-defaults.conf文件，开启Spark事件日志记录，并配置OBS桶名称及目录。

```
cat >> ~/spark-obs/conf/spark-defaults.conf <<EOF
spark.eventLog.enabled=true
spark.eventLog.dir=obs://{bucket-name}/{log-dir}/
EOF
```

- spark.eventLog.enabled：设置为true，表示开启Spark事件日志记录。
- spark.eventLog.dir：OBS桶名称及路径，格式为*obs://{bucket-name}/{log-dir}/*，例如obs://spark-sh1/history-obs/。请务必修改OBS桶名称及目录为正确值。

步骤2 修改~/spark-obs/conf/spark-env.sh文件，如果该文件不存在，使用命令复制模板为文件。

```
cp ~/spark-obs/conf/spark-env.sh.template ~/spark-obs/conf/spark-env.sh

cat >> ~/spark-obs/conf/spark-env.sh <<EOF
SPARK_HISTORY_OPTS="-Dspark.history.fs.logDirectory=obs://{bucket-name}/{log-dir}/"
EOF
```

此处的OBS地址需要与上一步spark-default.conf中的一致。

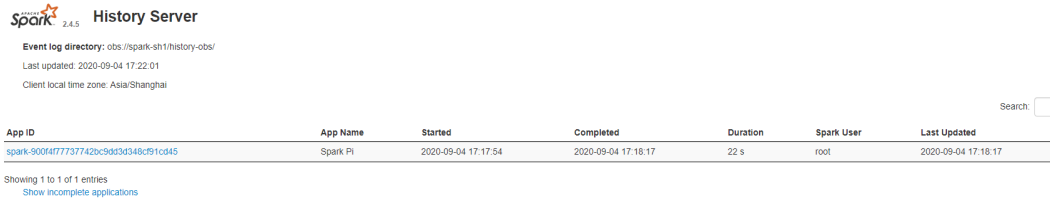
步骤3 直接启动history server。

```
start-history-server.sh
```

回显如下：


```
starting org.apache.spark.deploy.history.HistoryServer, logging to /root/spark-obs/logs/spark-root-org.apache.spark.deploy.history.HistoryServer-1-spark-sh1.out
```

步骤4 启动后可以通过节点端口18080访问。



如需关闭history server，可执行以下脚本：

```
stop-history-server.sh
```

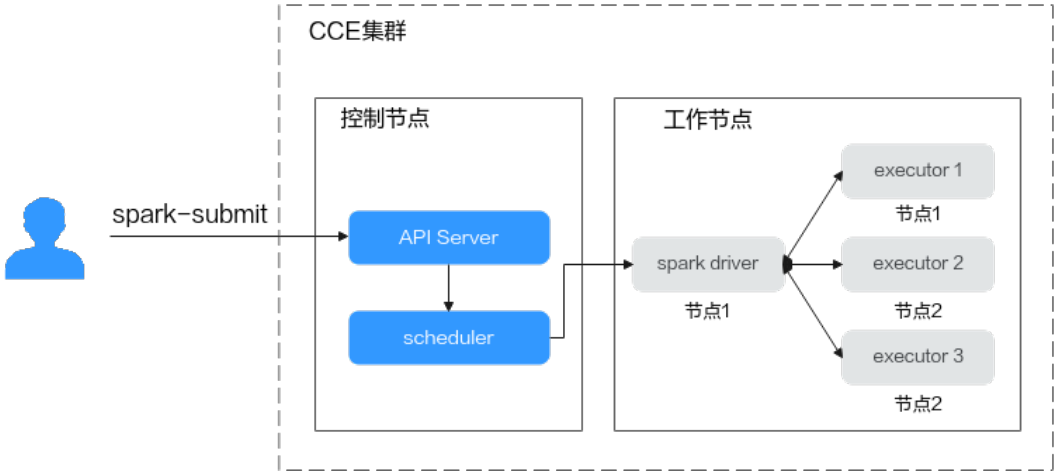
----结束

15.6.2 使用 Spark on CCE

使用Spark的Kubernetes调度程序spark-submit，可以将Spark应用程序提交到Kubernetes集群中运行，详情请参见**在Kubernetes上运行Spark**。使用spark-submit提交Spark应用程序的工作原理如下：

- 创建一个Pod，用于运行Spark的驱动程序。
- 驱动程序在集群中创建执行程序的Pod并与其建立连接，用于执行应用程序代码。
- 应用程序完成后，执行程序的Pod将终止并清理，但驱动程序Pod仍然存在并保持在“已停止”状态，直到最终进行垃圾回收或手动清理。在“已停止”状态下，驱动程序Pod不会使用任何计算或内存资源。

图 15-2 提交机制的工作原理



在 CCE 上运行 SparkPi 例子

步骤1 在执行Spark的机器上安装kubectl，详情请参见**通过kubectl连接集群**。

步骤2 kubectl安装成功后，执行如下命令授予集群权限。

```
# 创建服务账号
kubectl create serviceaccount spark
# 将集群角色spark-role和上一步创建服务账号绑定，并指定default命名空间授予edit的clusterrole权限
kubectl create clusterrolebinding spark-role --clusterrole=edit --serviceaccount=default:spark --namespace=default
```

步骤3 以提交Spark-Pi的作业到CCE为例：

```
spark-submit \
  --master k8s://https://**.**.**.**:5443 \
  --deploy-mode cluster \
  --name spark-pi \
  --class org.apache.spark.examples.SparkPi \
  --conf spark.executor.instances=2 \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
  --conf spark.kubernetes.container.image=swr.ap-southeast-1.myhuaweicloud.com/dev-container/
spark:3.1.3-obs \
  local:///root/spark-obs/examples/jars/spark-examples_2.12-3.1.1.jar
```

配置说明：

- --master：集群的API Server，其中https://**.**.**.**:5443为 ~/.kube/config中使用的master地址，可通过**kubectl cluster-info**获取。
- --deploy-mode：
  - cluster：在集群的工作节点上部署驱动程序。
  - client：（默认值）作为外部客户端在本地部署驱动程序。
- --name：作业名称，集群中的Pod将以此开头。
- --class：应用程序，例如org.apache.spark.examples.SparkPi。
- --conf：Spark配置参数，使用键值格式。值得一提的是，所有能使用--conf指定的参数均会默认从文件~/spark-obs/conf/spark-defaults.conf中读取，所以通用配置可以如**配置Spark对接OBS**一样，直接写入作为默认值。
  - spark.executor.instances：执行程序的Pod数量。
  - spark.kubernetes.authenticate.driver.serviceAccountName：驱动程序的集群权限，选择**步骤2**中创建的serviceaccount。
  - spark.kubernetes.container.image：**预置镜像到SWR**步骤中上传至SWR的镜像地址。
- local：使用本地的jar包路径。本例中使用本地文件存放jar包，因此使用local类型。根据实际情况，该参数可采用多种类型（file/http/local等），详情请参见**官方文档**。

----结束

访问对象存储服务 OBS

使用spark-submit下发hdfs任务。请修改命令最后的参数为租户内实际的文件obs://bucket-name/filename。

```
spark-submit \
  --master k8s://https://**.**.**.**:5443 \
  --deploy-mode cluster \
  --name spark-hdfs-test \
  --class org.apache.spark.examples.HdfsTest \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
  --conf spark.kubernetes.container.image=swr.ap-southeast-1.myhuaweicloud.com/dev-container/
spark:3.1.3-obs \
  local:///root/spark-obs/examples/jars/spark-examples_2.12-3.1.1.jar obs://bucket-name/filename
```

Spark-shell 交互式 scala 命令支持

```
spark-shell \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
  --conf spark.kubernetes.container.image=swr.ap-southeast-1.myhuaweicloud.com/dev-container/
spark:3.1.3-obs \
  --master k8s://https://**.**.**.**:5443
```


下述命令定义linecount及wordcount两个spark计算任务的算法。

```
def linecount(input:org.apache.spark.sql.Dataset[String]):Long=input.filter(line => line.length()>0).count()
def wordcount(input:org.apache.spark.sql.Dataset[String]):Long=input.flatMap(value => value.split("\\s
+")).groupByKey(value => value).count().count()
```

下述命令定义了各种数据来源：

```
var alluxio = spark.read.textFile("alluxio://alluxio-master:19998/sample-1g")
var obs = spark.read.textFile("obs://gene-container-gtest/sample-1g")
var hdfs = spark.read.textFile("hdfs://192.168.1.184:9000/user/hadoop/books/sample-1g")
```

下述命令开始正式计算：

```
spark.time(wordcount(obs))
spark.time(linecount(obs))
```