

云容器实例

最佳实践

文档版本 01
发布日期 2024-01-16



版权所有 © 华为云计算技术有限公司 2024。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HUAWEI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 弹性伸缩	1
1.1 CCE 容器实例弹性伸缩到 CCI 服务	1
2 负载创建	8
2.1 概述	8
2.2 使用 Docker run 运行容器	8
2.3 使用控制台创建负载	10
2.4 调用 API 创建负载	15
2.5 Dockerfile 参数在云容器实例中如何使用	21
3 负载管理	23
3.1 CCI 应用进行优雅滚动升级	23
3.2 在容器中通过环境变量获取 Pod 基础信息	27
3.3 内核参数配置	29
3.4 修改/dev/shm 容量大小	29

1 弹性伸缩

1.1 CCE 容器实例弹性伸缩到 CCI 服务

Virtual Kubelet是基于社区Virtual Kubelet开源项目开发的插件，该插件支持用户在短时高负载场景下，将部署在CCE上的容器实例（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。弹性Pod实例可以在创建无状态负载（Deployment）、有状态负载（StatefulSet）、普通任务（Job）三种资源类型时设置，也可以在单独创建Pod时在YAML中指定Label，详情请参见[通过标签labels设置弹性策略](#)。

📖 说明

Virtual Kubelet插件目前不支持在“拉美-圣保罗一”区域使用

具体功能如下：

- 支持容器实例实现秒级弹性伸缩：Virtual Kubelet插件将自动为您在云容器实例CCI侧创建容器实例，减少运维成本。
- 无缝对接容器镜像服务SWR，支持使用公用镜像和私有镜像。
- 支持CCI容器实例的事件同步、监控、日志、exec、查看状态等操作。
- 支持查看虚拟弹性节点的节点容量信息。
- 支持CCE和CCI两侧实例的service网络互通。

约束及限制

- 仅支持VPC网络模式的CCE Standard集群和CCE Turbo集群（virtual-kubelet 1.2.5版本及以上支持），暂不支持ARM集群。如果集群中包含ARM节点，插件实例将不会部署至ARM节点。
- 调度到CCI的实例的存储类型支持ConfigMap、Secret、EmptyDir、DownwardAPI、Projected、PersistentVolumeClaims几种Volume类型，其中Projected和DownwardAPI类型仅virtual-kubelet 1.3.25版本及以上支持。
 - EmptyDir：不支持子路径。
 - PersistentVolumeClaims：只支持SFS、SFS Turbo云存储类型，且只支持使用CSI类型的StorageClass。
 - Projected：如配置了serviceAccountToken类型的source，那么弹性到CCI后挂载的会是对应service-account-token secret中的token，该token为长期有

效的token且没有预期受众，即expirationSeconds和audience两项配置不会生效。

- 暂不支持守护进程集（DaemonSet）以及HostNetwork网络模式的容器实例（Pod）弹性到CCI。
- 跨CCE和CCI实例Service网络互通只支持集群内访问（ClusterIP）类型。不支持在init-container中访问CCE侧ClusterIP service。
- 跨CCE和CCI实例，在对接LoadBalancer类型的Service或Ingress时：
 - a. 禁止指定健康检查端口，在CCE集群下，由于CCI的容器与CCE的容器在ELB注册的后端使用端口不一致，指定健康检查端口会导致部分后端健康检查异常。
 - b. 跨集群使用Service对接同一个ELB的监听器时，需确认健康检查方式，避免服务访问异常。
 - c. 跨CCE和CCI实例，在对接共享型LoadBalancer类型的Service时，需要放通node安全组下100.125.0.0/16网段的容器端口。
 - d. 当前对接LoadBalancer类型的Service或Ingress仅支持auto和enforce调度策略。
- 集群所在子网不能与10.247.0.0/16重叠，否则会与CCI命名空间下的Service网段冲突，导致无法使用。
- 使用插件前需要用户在CCI界面对CCI服务进行授权。
- 安装virtual-kubelet插件后会在CCI服务新建一个名为"cce-burst-"+集群ID的命名空间，该命名空间完全由virtual-kubelet管理，不建议直接在CCI服务使用该命名空间，如需使用CCI服务，请另外新建命名空间。
- 弹性到CCI的业务量不同时，插件的资源占用也不相同。业务申请的POD、Secret、ConfigMap、PV、PVC会占用虚拟机资源。建议用户评估自己的业务使用量，按以下规格申请对应的虚拟机大小：1000pod+1000CM（300KB）推荐2U4G规格节点，2000pod+2000CM推荐4U8G规格节点，4000pod+4000CM推荐8U16G规格节点。
- 当弹性到CCI的资源调度失败时，virtual-kubelet节点会被锁定半小时，期间无法调度至CCI。用户可通过CCE集群控制台，使用kubectl工具查看virtual-kubelet节点状态，若节点被锁定，可手动解锁virtual-kubelet。

```
user@imkrz4xzkz30alq-machine:~$ kubectl get node
NAME                STATUS              ROLES    AGE    VERSION
192.168.182.101    Ready              <none>   33d    v1.23.0-CCE23.0.1
virtual-kubelet     Ready,SchedulingDisabled virtual-kubelet 4d5h   v1.19.16-v1.3.4-145-g8114c8a-dev
user@imkrz4xzkz30alq-machine:~$
user@imkrz4xzkz30alq-machine:~$
user@imkrz4xzkz30alq-machine:~$ kubectl uncordon virtual-kubelet
node/virtual-kubelet uncordoned
user@imkrz4xzkz30alq-machine:~$ kubectl get node
NAME                STATUS              ROLES    AGE    VERSION
192.168.182.101    Ready              <none>   33d    v1.23.0-CCE23.0.1
virtual-kubelet     Ready              virtual-kubelet 4d5h   v1.19.16-v1.3.4-145-g8114c8a-dev
user@imkrz4xzkz30alq-machine:~$
```

- 使用日志管理功能采集弹性到CCI的Pod中的日志具有如下约束：
 - 当前仅支持容器文件路径采集。
 - 依赖virtual-kubelet的Service互通能力，请在安装插件时勾选“网络互通”开关。
 - 弹性到CCI的Pod不支持日志策略热更新，即更新日志策略后需要重新部署弹性到CCI的Pod才可生效。
 - 日志采集会增加Pod内存消耗，Pod被单个日志策略关联时，建议您预留50MB的内存。Pod被多个日志策略关联时，每多一个关联的日志策略，建议您额外再多预留5MB的内存。

- 单条日志长度限制为250KB，如果超过则会被丢弃。
- 不支持指定系统、设备、cgroup、tmpfs等挂载目录的日志采集，这些目录下的日志文件将不会被采集。
- 同一个容器中关联到同一个日志策略的待采集的日志文件不能重名，如果有重复文件则只会采集到采集器首次感知到的日志文件。
- 日志文件的文件名，最大长度为190，超过长度限制的日志文件将不会被采集。

pod 规格的计算与约束

Pod规格的计算方式遵循如下规则：

- 所有应用容器和初始化容器对某个资源的Requests和Limits均会被设置为相等的值，若配置了Limits，则取Limits值，没有配Limits，则取Requests值。
- Pod对某个资源的Requests和Limits，是取如下两项的较大者：
 - 所有应用容器对某个资源的Limits之和。
 - 所有初始化容器对某个资源Limits的最大值。

Pod规格约束：

- Pod的CPU需大于0。
- 经过[资源自动规整](#)后，Pod的CPU在0.25核~32核范围内，或者等于48核或64核；Memory在1Gi~512Gi范围内，且必须为1Gi的整数倍，且满足CPU/Memory配比在1:2~1:8之间。

资源自动规整

对弹性到CCI的Pod，若其配置的资源规格不满足CCI容器规范，且规格不高于32U 256Gi，virtual-kubelet会自动尝试将Pod资源向上规整到满足CCI容器规范的范围，以规整后的资源规格创建Pod到CCI。

自动规整规则如下：

1. 将Pod中除了BestEffort的每个应用容器和初始化容器的CPU向上调整至0.25核的整数倍，Memory向上调整至大于等于0.2Gi。
2. 若此时Pod的CPU大于32U或者Memory大于256Gi，则不再继续进行自动规整，否则继续自动规整。
3. 将整个Pod的Memory向上调整至1Gi的整数倍。
4. 若Pod Memory/CPU的比值小于2，则将Pod Memory向上调整至大于等于CPU的2倍，且满足是1Gi的整数倍；若Pod Memory/CPU的比值大于8，则将Pod CPU向上调整至大于等于Memory的1/8，且满足是0.25核的整数倍。
5. 以上对Pod级别资源向上调整造成的增量CPU和Memory，全部添加到Pod中第一个不为BestEffort的应用容器上。

安装插件

步骤1 在CCE控制台，单击集群名称进入集群，单击左侧导航栏的“插件中心”，在右侧找到**CCE突发弹性引擎**插件，单击“安装”。

步骤2 在“规格配置”步骤中，勾选“网络互通”后的选择框，可实现CCE集群中的Pod与CCI集群中的Pod通过Kubernetes Service互通。

图 1-1 勾选“网络互通”



步骤3 单击“安装”。

说明

勾选了网络互通后, 会在CCI运行的Pod中注入一个sidecar用于支持service访问的能力, 查询到的运行容器数量会比定义的多一个, 属于正常情况。

----结束

通过标签 labels 设置弹性策略

您成功安装virtual-kubelet插件后, 在工作负载中添加virtual-kubelet.io/burst-to-cci这个标签即可设置弹性到CCI。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test
  namespace: default
  labels:
    virtual-kubelet.io/burst-to-cci: 'auto' # 弹性到CCI
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
        - image: 'nginx:perl'
          name: container-0
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
            limits:
              cpu: 250m
              memory: 512Mi
          volumeMounts: []
```

```
imagePullSecrets:  
- name: default-secret
```

📖 说明

创建弹性至CCI的负载时需要在工作负载或Pod的labels中添加如下字段：

```
virtual-kubelet.io/burst-to-cci: "auto"
```

其中，value值支持以下选项：

- auto：根据用户集群内调度器实际打分结果自动决定是否弹性至CCI，其中在 [TaintToleration算法](#) 上会优先选择调度到CCE节点。
- localPrefer：集群资源不足时，将Pod部署到CCI。
- enforce：强制调度至CCI。
- off：不调度至CCI。

使用 profile 管理线下 IDC 和云上分配数量

使用profile配置管理集群内pod，通过labelSelector类方式关联profile和pod，并配置关联pod的分配策略，实现pod在线下IDC和云上的分配或数量限制。

约束与限制

- 客户可通过Profile配置管理集群内pod策略，仍兼容原有pod的label内burst-to-cci的配置方式，优先级比profile高。
- localPrefer不可同时配置local、cci。
- auto和localPrefer策略允许关联未被profile关联过的pod，enforce策略不允许关联未被profile关联过的pod。
- 目前profile在配置localPrefer策略下，为避免全局性问题，在极限场景下限制local数量的配置可能会失效。
- 在deployment滚动升级场景下，推荐配置尽可能小的maxSurge值（如直接配置为0），避免出现升级时限制maxNum的区域调度量少于预期的现象。
- pod只能关联一个profile，即关联度最大的profile。若pod创建后，对其label进行修改导致与原profile不匹配，pod会重新选择关联度最大的profile进行关联。关联度最大的profile确定方法：
 - 根据profile中objectLabels计算labelSelector内matchLabels的数量及matchExpression的数量之和，和最大的profile即为pod关联度最大的profile；
 - 若出现和相同的profile，选择profile的name字母序最小的profile为pod关联度最大的profile。
- 不支持使用log-agent插件采集profile管理的负载日志。

使用方式

配置local maxNum和scaleDownPriority

```
apiVersion: scheduling.cci.io/v1  
kind: ScheduleProfile  
metadata:  
  name: test-cci-profile  
  namespace: default  
spec:  
  objectLabels:  
    matchLabels:  
      app: nginx  
  strategy: localPrefer  
  location:
```



```
local:
  maxNum: 20 # 当前暂不支持local/ccl同时配置maxNum
  scaleDownPriority: 10
  cci: {}
status:
  phase: initialized
  restrict:
    local: 20 # restrict内随着location内配置进行填写，即不会同时出现 local/ccl
  used:
    local: 20
    cci: 0
```

配置cci maxNum和scaleDownPriority

```
apiVersion: scheduling.cci.io/v1
kind: ScheduleProfile
metadata:
  name: test-cci-profile
  namespace: default
spec:
  objectLabels:
    matchLabels:
      app: nginx
  strategy: localPrefer
  location:
    local: {}
    cci:
      maxNum: 20 # 当前暂不支持local/ccl同时配置maxNum
      scaleDownPriority: 10
status:
  phase: initialized
  restrict:
    cci: 20 # restrict内随着location内配置进行填写，即不会同时出现 local/ccl
  used:
    local: 0
    cci: 20
```

参数说明：

- strategy：调度策略选择。可配置策略值：auto、enforce、localPrefer；
- location内可配置线下IDC和云上pod限制数量maxNum和pod缩容优先级scaleDownPriority，maxNum取值范围[0~int32]，scaleDownPriority取值范围[-100, 100]；

创建无状态负载，使用selector方式选择含有app: nginx的pod，关联上文创建的profile。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-1
          image: nginx:latest
          imagePullPolicy: IfNotPresent
          resources:
            requests:
              cpu: 250m
```

```
memory: 512Mi
limits:
  cpu: 250m
  memory: 512Mi
imagePullSecrets:
- name: default-secret
```

卸载插件

1. 登录CCE控制台，进入集群，在左侧导航栏选择“插件管理”，在右侧“已安装插件”页签下，单击virtual kubelet下的“卸载”。
2. 在弹出的窗口中，单击“是”，可卸载该插件。（卸载插件会自动删除CCI侧的所有资源，以确保不会有资源残留造成额外计费）

注意

- 由于virtual-kubelet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，否则卸载插件会失败。若已经因无可调度节点造成插件卸载失败，需要在有可调度节点后重新单击插件卸载。
- 如果在未卸载virtual-kubelet插件的情况下直接删除集群，CCI侧的资源不会被自动清理，会导致CCI侧资源残留，可能会造成额外计费。因此请确保完成以下任一操作。
 - 在删除集群前先卸载virtual-kubelet插件。
 - 在直接删除集群后登录CCI控制台删除名为cce-burst- $\{\text{CLUSTER_ID}\}$ 的命名空间。
 - 集群休眠时CCI侧正在运行的实例不会自动停止，会持续运行并计费。因此如不需要实例继续运行，请确保在集群休眠前将弹性到CCI的负载实例数缩至0。

2 负载创建

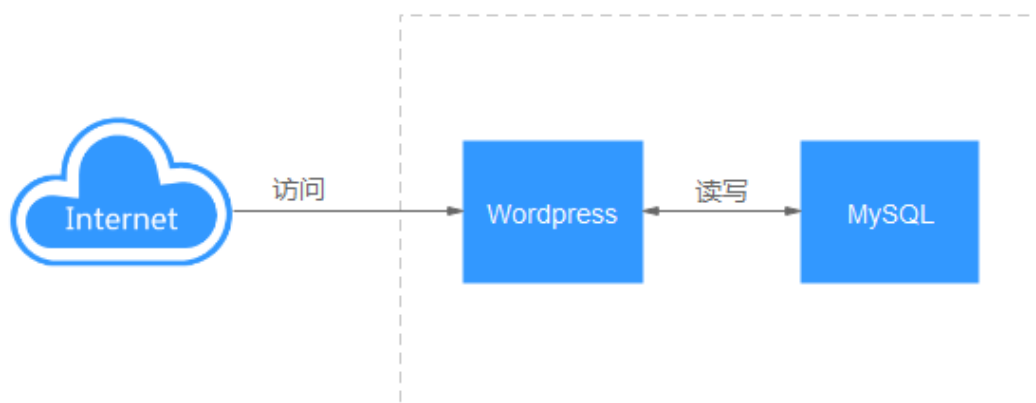
2.1 概述

在云容器实例中，您可以使用多种方法创建负载，包括使用云容器实例的Console控制台界面、调用API部署应用，那这些方式的使用有什么不同的地方呢？这些方法又与直接运行Docker run命令运行容器有什么区别呢？

本文将通过运行一个Wordpress + MySQL的博客为例，比较这几种方法之间的异同，以利于您挑选合适的使用方法。

WordPress是使用PHP语言开发的博客平台。用户可以在支持PHP和MySQL数据库的服务上架设属于自己的网站，也可以把WordPress当作一个内容管理系统来使用。更多WordPress信息可以通过官方网站了解：<https://wordpress.org/>。

WordPress需配合MySQL一起使用，WordPress运行内容管理程序，MySQL作为数据库存储数据。在容器中运行通常会将WordPress和MySQL分别运行两个容器中，如下图所示。



2.2 使用 Docker run 运行容器

Docker是一个开源的应用容器引擎。容器引擎是Kubernetes (k8s) 最重要的组件之一，负责管理镜像和容器的生命周期。使用Docker，无需配置运行环境，镜像中会包含一整套环境，同时进程间是隔离的，不会相互影响。

Docker容器都是由docker镜像创建，Docker利用容器来运行应用，Docker容器包含了应用运行所需要的所有环境。

镜像准备

WordPress和MySQL的镜像都是通用镜像，可以直接从镜像中心获取。

您可以在安装了容器引擎的机器上使用**docker pull**命令即可下载镜像，如下所示。

```
docker pull mysql:5.7
docker pull wordpress
```

下载完成后，执行**docker images**命令可以看到本地已经存在两个镜像，如下图所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
wordpress	latest	6a837ea4bd22	6 days ago	408MB
mysql	5.7	0d16d0a97dd1	5 weeks ago	372MB

运行容器

使用容器引擎可以直接运行Wordpress和MySQL，且可以使用**--link**参数将两个容器连接，在不改动代码的情况下让Wordpress的容器访问MySQL的容器。

执行下面的命令运行MySQL。

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=***** -e MYSQL_DATABASE=wordpress -d mysql:5.7
```

参数解释如下：

- **--name**指定容器的名称为some-mysql。
- **-e**指定容器的环境变量。如这里指定环境变量MYSQL_ROOT_PASSWORD的值为*****，请替换为您设置的密码。指定环境变量MYSQL_DATABASE，镜像启动时要创建的数据库名称为wordpress。
- **-d**表示在后台运行。

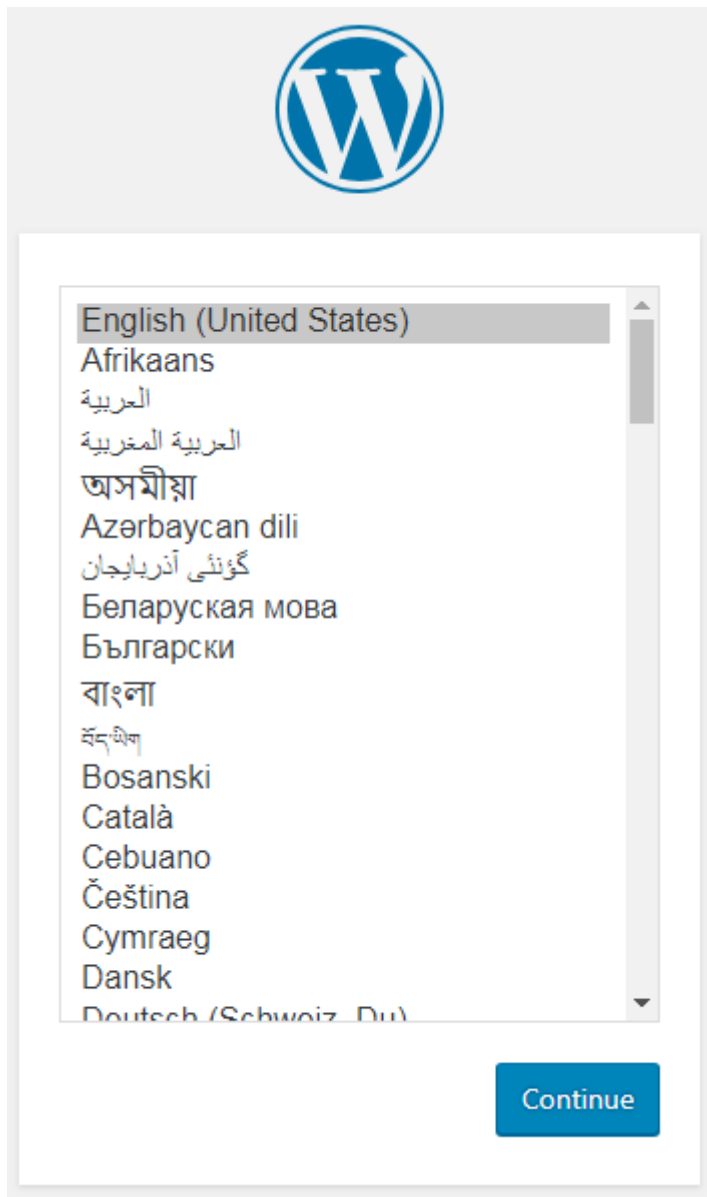
执行下面的命令运行Wordpress。

```
docker run --name some-wordpress --link some-mysql:mysql -p 8080:80 -e WORDPRESS_DB_PASSWORD=***** -e WORDPRESS_DB_USER=root -d wordpress
```

参数解释如下：

- **--name**指定容器的名称为some-wordpress。
- **--link**指定some-wordpress容器链接some-mysql容器，并将some-mysql命名为mysql。这里**--link**只是提供了一种方便，不使用**--link**的话，可以指定some-wordpress的环境变量WORDPRESS_DB_HOST访问mysql的IP与端口。
- **-p**指定端口映射，如这里将容器的80端口映射到主机的8080端口。
- **-e**指定容器的环境变量，如这里指定环境变量WORDPRESS_DB_PASSWORD的值为*****，请替换为您设置的密码。Wordpress的环境变量WORDPRESS_DB_PASSWORD必须与MySQL的环境变量MYSQL_ROOT_PASSWORD值相同，这是因为Wordpress需要密码访问MySQL数据库。WORDPRESS_DB_USER为访问数据的用户名，使用用户root去连接MySQL。
- **-d**表示在后台运行。

Wordpress运行之后，就可以在本机通过http://127.0.0.1:8080访问Wordpress博客了，如下所示。



2.3 使用控制台创建负载

[使用Docker run运行容器](#)章节使用docker run命令运行了Wordpress博客，但是在很多场景下使用容器引擎并不方便，如应用弹性伸缩、滚动升级等。

云容器实例提供无服务器容器引擎，让您不需要管理集群和服务器，只需要三步简单配置，即可畅享容器的敏捷和高性能。云容器实例支持创建无状态负载（Deployment）和有状态负载（StatefulSet），并基于Kubernetes的负载模型增强了容器安全隔离、负载快速部署、弹性负载均衡、弹性扩缩容、蓝绿发布等重要能力。

创建命名空间

步骤1 登录云容器实例管理控制台，左侧导航栏中选择“命名空间”。

步骤2 在对应类型的命名空间下单击“创建”。

步骤3 填写命名空间名称。

步骤4 设置VPC。

选择使用已有VPC或新建VPC，新建VPC需要填写VPC网段，建议使用网段：10.0.0.0/8~24，172.16.0.0/12~24，192.168.0.0/16~24。

步骤5 设置子网网段。

您需要关注子网的可用IP数，确保有足够数量的可用IP，如果没有可用IP，则会导致负载创建失败。

步骤6 单击“创建”。

----结束

创建 MySQL 负载

步骤1 登录云容器实例管理控制台，左侧导航栏中选择“工作负载 > 无状态（Deployment）”，在右侧页面单击“镜像创建”。

步骤2 添加基本信息。

- **负载名称**：mysql。
- **命名空间**：选择[创建命名空间](#)创建的命名空间。
- **Pod数量**：本例中修改Pod数量为1。
- **Pod规格**：选择通用计算型，CPU 0.5核，内存 1GiB。

* 负载名称: mysql

请输入以小写字母或数字开头，小写字母、数字、中划线 (-)、点 (.) 组成 (其中两点不能相连，点不能与中划线相连)，小写字母或数字结尾的1到63字符的字符串

* 命名空间: gene-test1 [创建命名空间](#)

正常 通用计算型 | CCI-VPC-1928286404 192.168.0.0/16

负载描述: 请输入描述信息

* Pod数量: 1

* Pod规格: 通用计算型

1X	2X	4X	8X	自定义
CPU 0.5核 内存 1GB	CPU 1核 内存 2GB	CPU 2核 内存 4GB	CPU 4核 内存 8GB	

- **容器配置**
 - a. 在开源镜像中心搜索并选择mysql镜像。



b. 配置镜像参数，选择镜像版本为5.7，CPU和内存配置为0.5核和1G。



c. 在高级配置中，添加容器的环境变量MYSQL_ROOT_PASSWORD，并填入变量，变量值为MySQL数据库的密码（需自行设置）。



步骤3 单击“下一步”，配置负载信息，负载访问选择内网访问（可以被云容器实例中其他负载通过“服务名称:端口”方法），将“服务名称”定义为mysql，并指定负载访问端口3306映射到容器的3306端口（mysql镜像的默认访问端口）。

这样在云容器实例内部，通过mysql:3306就可以访问MySQL负载。



步骤4 配置完成后，单击“下一步”，确认规格后单击“提交”。

在负载列表中，待负载状态为“运行中”，负载创建成功。

----结束

创建 Wordpress 负载

步骤1 登录云容器实例管理控制台，左侧导航栏中选择“工作负载 > 无状态（Deployment）”，在右侧页面单击“镜像创建”。

步骤2 添加基本信息。

- **负载名称**：wordpress。
- **命名空间**：选择[创建命名空间](#)创建的命名空间。
- **Pod数量**：本例中修改Pod数量为2。
- **Pod规格**：选择通用计算型，CPU 0.5核，内存 1GiB。

* 负载名称 X
请输入以小写字母或数字开头，小写字母、数字、中划线 (-)、点 (.) 组成（其中两点不能相连，点不能与中划线相连），小写字母或数字结尾的1到63字符的字符串

* 命名空间 C 创建命名空间
● 正常 通用计算型 | CCI.VPC-1928286404 192.168.0.0/16

负载描述 0/250

* Pod数量 +

* Pod规格

1X	2X	4X	8X	自定义
CPU 0.5核 内存 1GB	CPU 1核 内存 2GB	CPU 2核 内存 4GB	CPU 4核 内存 8GB	<input type="button" value="自定义"/>

- **容器配置**：
 - a. 在开源镜像中心搜索并选择wordpress镜像。

* 容器配置

温馨提示：镜像数据来自于容器镜像服务（SWR）

我的镜像 | 开源镜像中心 | 共享镜像

X | Q | C

wordpress
下载次数：373118115

- b. 配置镜像参数，选择镜像版本为php7.1，CPU和内存配置为0.5核和1G。



- c. 在高级配置中，设置环境变量，使WordPress可以访问MySQL数据库。



表 2-1 环境变量说明

变量名	变量/变量引用
WORDPRESS_DB_HOST	MySQL的访问地址。 示例: 10.***.***.***:3306
WORDPRESS_DB_PASSWORD	MySQL数据库的密码，此处密码必须与 创建MySQL负载 设置MySQL的密码相同。

步骤3 单击“下一步”，配置负载信息。

负载访问选择公网访问，服务名称为“wordpress”，选择ELB实例（如果没有实例可以单击“新建增强型ELB实例”创建），选择ELB协议为“HTTP”，ELB端口号为9012，指定负载访问端口“8080”映射到容器的“80”端口（wordpress镜像的默认访问端口），HTTP路由设置为“/”（即通过“http://elb ip:外部端口”就可以访问wordpress）并映射到8080负载端口。

负载访问

访问类型 内网访问 公网访问 不启用
将为工作负载提供一个可以从Internet访问的入口，支持HTTP协议并根据URL转发请求，如WordPress等前台类服务可以选择公网访问。 [如何配置负载公网访问](#)

* 服务名称

* ELB实例 [创建增强型ELB实例，完成后点击刷新按钮生效](#)

ELB协议 HTTP/HTTPS TCP/UDP

* Ingress名称

公网域名
将通过该公网域名访问您的负载。不配置时通过ELB EIP访问负载；需要您购买公网域名，并将域名解析指向所选ELB实例的EIP

* ELB端口
如果需要公网提供HTTPS访问，请选择HTTPS协议；将通过ELB实例上该端口访问负载

* 负载端口协议 TCP

* 负载端口配置 (设置负载访问端口与容器端口映射关系；负载请求由负载域名、负载访问端口 转发至 容器实例 容器端口)

负载访问端口	容器端口	操作
<input type="text" value="8080"/>	<input type="text" value="80"/>	删除

[添加端口](#)

* HTTP路由配置 (设置映射路径到后端负载访问端口的路由关系；公网请求由http (或https) //公网域名(或ELB EIP)/外部端口/映射路径 转发至 负载域名.负载访问端口)

映射路径	负载访问端口	操作
<input type="text" value="/"/>	<input type="text" value="8080"/>	删除

步骤4 配置完成后，单击“下一步”，确认规格后单击“提交”。

在负载列表中，待负载状态为“运行中”，负载创建成功。您可以单击负载名进入负载详情界面。

在“访问配置”处选择“公网访问”，查看访问地址，即ELB实例的“IP地址:端口”。

访问配置

[公网访问](#) | [内网访问](#) | [事件](#)

公网访问地址	公网IP	内网访问地址	内网负载域名地址	协议
<input type="text" value="http://192.168.24.162:9012/"/>	<input type="text" value="192.168.24.162"/>	<input type="text" value="http://192.168.24.162:9012/"/>	<input type="text" value="wordpress:8080"/>	HTTP

----结束

2.4 调用 API 创建负载

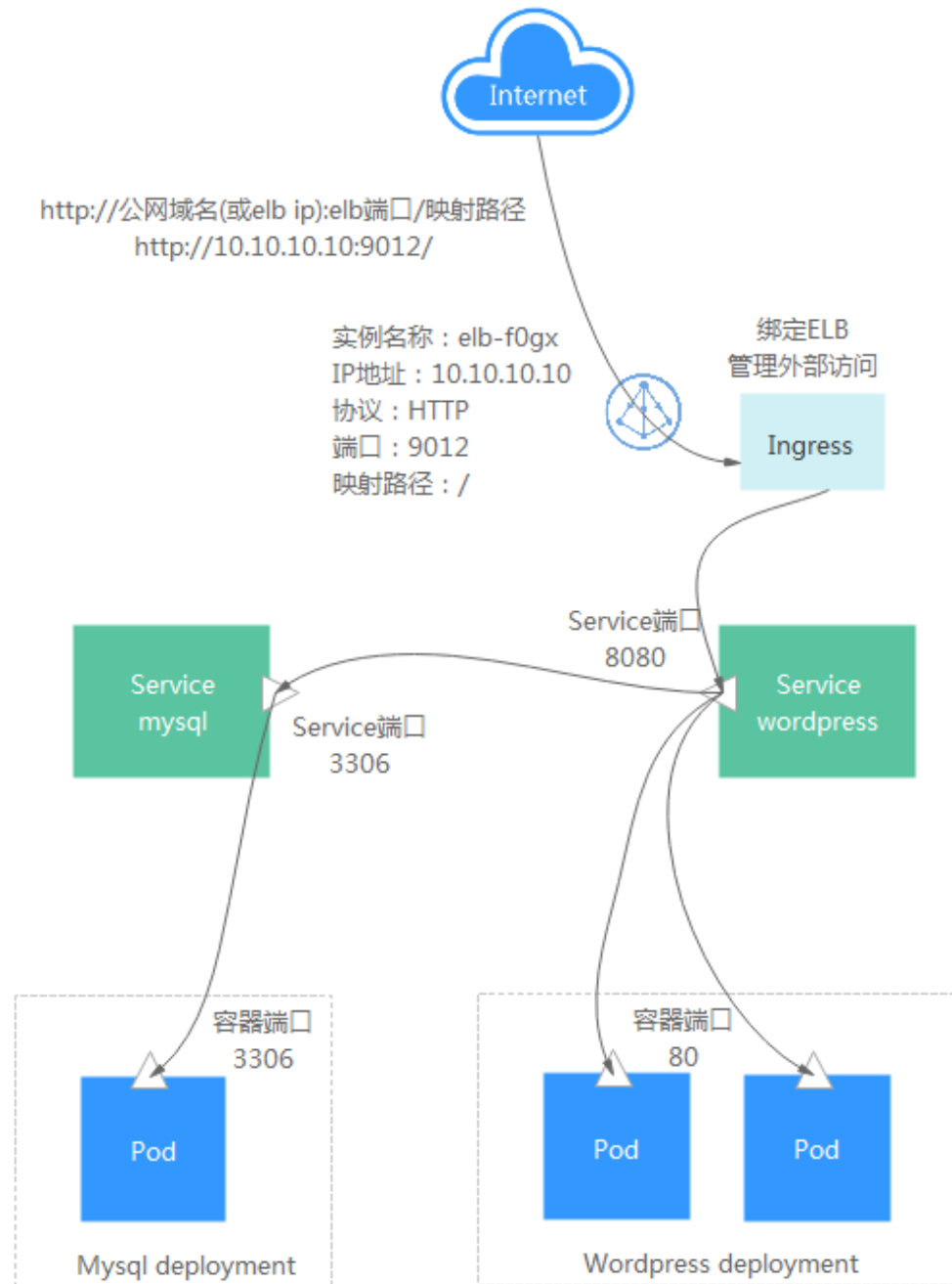
云容器实例原生支持Kubernetes API，相比从控制台创建负载，使用API的粒度更细一些。

Kubernetes中，运行容器的最小资源单位是Pod，一个Pod封装一个或多个容器、存储资源、一个独立的网络IP等。实际使用中很少直接创建Pod，而是使用Kubernetes中称为Controller的抽象层来管理Pod实例，例如Deployment和StatefulSet。另外在Kubernetes中使用Service定义一系列Pod以及访问这些Pod的策略的资源对象，使用Ingress管理外部访问的资源对象。如果您对Kubernetes的资源不熟悉，请参见《[云容器实例开发指南](#)》了解各资源的关系。

对于Wordpress应用，可以按照下图调用API创建一系列资源。

- MySQL：创建一个Deployment部署mysql，创建一个Service定义mysql的访问策略。

- Wordpress: 创建一个Deployment部署wordpress, 创建Service和Ingress定义wordpress的访问策略。



Namespace

步骤1 调用**创建Namespace**接口创建命名空间, 并指定使用命名空间的类型。

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "namespace-test",
    "annotations": {
      "namespace.kubernetes.io/flavor": "gpu-accelerated"
    }
  }
}
```

```
    },  
    "spec": {  
      "finalizers": [  
        "kubernetes"  
      ]  
    }  
  }  
}
```

步骤2 调用**创建Network**接口创建网络，与VPC与子网关联。

```
{  
  "apiVersion": "networking.cci.io/v1beta1",  
  "kind": "Network",  
  "metadata": {  
    "annotations": {  
      "network.alpha.kubernetes.io/default-security-group": "{{security-group-id}}",  
      "network.alpha.kubernetes.io/domain-id": "{{domain-id}}",  
      "network.alpha.kubernetes.io/project-id": "{{project-id}}"  
    },  
    "name": "test-network"  
  },  
  "spec": {  
    "availableZone": "{{zone}}",  
    "cidr": "192.168.0.0/24",  
    "attachedVPC": "{{vpc-id}}",  
    "networkID": "{{network-id}}",  
    "networkType": "underlay_neutron",  
    "subnetID": "{{subnet-id}}"  
  }  
}
```

----结束

MySQL

步骤1 调用**创建Deployment**接口部署MySQL。

- Deployment名称为mysql。
- 设置Pod的标签为app:mysql。
- 使用mysql:5.7镜像。
- 设置容器环境变量MYSQL_ROOT_PASSWORD为“*****”，请替换为您设置的密码。

```
{  
  "apiVersion": "apps/v1",  
  "kind": "Deployment",  
  "metadata": {  
    "name": "mysql"  
  },  
  "spec": {  
    "replicas": 1,  
    "selector": {  
      "matchLabels": {  
        "app": "mysql"  
      }  
    },  
    "template": {  
      "metadata": {  
        "labels": {  
          "app": "mysql"  
        }  
      },  
      "spec": {  
        "containers": [  
          {  
            "image": "mysql:5.7",  
            "name": "container-0",  
          }  
        ]  
      }  
    }  
  }  
}
```

```
    "resources": {
      "limits": {
        "cpu": "500m",
        "memory": "1024Mi"
      },
      "requests": {
        "cpu": "500m",
        "memory": "1024Mi"
      }
    },
    "env": [
      {
        "name": "MYSQL_ROOT_PASSWORD",
        "value": "*****"
      }
    ]
  },
  "imagePullSecrets": [
    {
      "name": "imagepull-secret"
    }
  ]
}
}
```

步骤2 调用[创建Service接口](#)创建一个Service，定义[步骤1](#)中创建的Pod的访问策略。

- Service名称为mysql。
- 选择标签为app:mysql的Pod，即关联[步骤1](#)中创建的Pod。
- 负载访问端口3306映射到容器的3306端口。
- Service的访问类型为ClusterIP，即使用ClusterIP在内部访问Service。

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "mysql",
    "labels": {
      "app": "mysql"
    }
  },
  "spec": {
    "selector": {
      "app": "mysql"
    },
    "ports": [
      {
        "name": "service0",
        "targetPort": 3306,
        "port": 3306,
        "protocol": "TCP"
      }
    ],
    "type": "ClusterIP"
  }
}
```

----结束

Wordpress

步骤1 调用[创建Deployment接口](#)部署Wordpress。

- Deployment名称为wordpress。
- replicas值为2，表示创建2个pod。
- 设置Pod的标签为app:wordpress。
- 使用wordpress:latest镜像。
- 设置容器环境变量WORDPRESS_DB_PASSWORD为“*****”，请替换为您设置的密码。此处的密码必须与MySQL的MYSQL_ROOT_PASSWORD一致。

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "wordpress"
  },
  "spec": {
    "replicas": 2,
    "selector": {
      "matchLabels": {
        "app": "wordpress"
      }
    },
    "template": {
      "metadata": {
        "labels": {
          "app": "wordpress"
        }
      },
      "spec": {
        "containers": [
          {
            "image": "wordpress:latest",
            "name": "container-0",
            "resources": {
              "limits": {
                "cpu": "500m",
                "memory": "1024Mi"
              },
              "requests": {
                "cpu": "500m",
                "memory": "1024Mi"
              }
            },
            "env": [
              {
                "name": "WORDPRESS_DB_PASSWORD",
                "value": "*****"
              }
            ]
          }
        ]
      }
    },
    "imagePullSecrets": [
      {
        "name": "imagepull-secret"
      }
    ]
  }
}
```

步骤2 调用[创建Service接口](#)创建一个Service，定义[步骤1](#)中创建的Pod的访问策略。

- Service名称为wordpress。
- 选择标签为app:wordpress的Pod，即关联[步骤1](#)中创建的Pod。
- 负载访问端口8080映射到容器的80端口，80端口为wordpress镜像的默认对外暴露的端口。

- Service的访问类型为ClusterIP，即使用ClusterIP在内部访问Service。

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "wordpress",
    "labels": {
      "app": "wordpress"
    }
  },
  "spec": {
    "selector": {
      "app": "wordpress"
    },
    "ports": [
      {
        "name": "service0",
        "targetPort": 80,
        "port": 8080,
        "protocol": "TCP"
      }
    ],
    "type": "ClusterIP"
  }
}
```

步骤3 调用[创建Ingress](#)接口创建一个Ingress，定义wordpress的外部访问策略，即关联ELB实例（ELB实例需要与Wordpress负载在同一个VPC内）。

- metadata.annotations.kubernetes.io/elb.id: ELB实例的ID。
- metadata.annotations.kubernetes.io/elb.ip: ELB实例的IP地址。
- metadata.annotations.kubernetes.io/elb.port: ELB实例的端口。
- spec.rules: 访问服务的规则集合。path列表，每个path（比如：/）都关联一个backend（比如“wordpress:8080”）。backend是一个service:port的组合。Ingress的流量被转发到它所匹配的backend。

这里配置完后，访问ELB的IP:端口的流量就会流向wordpress:8080这个Service，由于Service是关联了wordpress的Pod，所以最终访问的就是[步骤1](#)中部署的wordpress容器。

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "wordpress",
    "labels": {
      "app": "wordpress",
      "isExternal": "true",
      "zone": "data"
    },
    "annotations": {
      "kubernetes.io/elb.id": "2d48d034-6046-48db-8bb2-53c67e8148b5",
      "kubernetes.io/elb.ip": "10.10.10.10",
      "kubernetes.io/elb.port": "9012"
    }
  },
  "spec": {
    "rules": [
      {
        "http": {
          "paths": [
            {
              "path": "/",
              "backend": {
                "serviceName": "wordpress",

```

```
        "servicePort": 8080
      }
    ]
  }
}
```

----结束

2.5 Dockerfile 参数在云容器实例中如何使用

应用场景

如果您了解容器引擎的使用，明白定制镜像时，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。

本章节将介绍Dockerfile文件的一些配置如何对应到云容器实例中去使用。

Dockerfile 参数在 CCI 中的使用

下面通过一个例子来说明他们之间的关系，这样您就可以更好的了解和熟悉云容器实例。

```
FROM ubuntu:16.04
ENV VERSION 1.0
VOLUME /var/lib/app
EXPOSE 80
ENTRYPOINT ["/entrypoint.sh"]
CMD ["start"]
```

上面是一个Dockerfile文件，包含一些常见的参数ENV、VOLUME、EXPOSE、ENTRYPOINT、CMD，这些参数在云容器实例中可以按如下方法配置。

- ENV为环境变量，在云容器实例中创建负载的时候，可以在高级配置中设置，“ENV VERSION 1.0”指令在CCI中的使用，如下所示。



- VOLUME为定义容器卷，通常配合docker run -v 宿主机路径:容器卷路径一起使用。

云容器实例中支持将云硬盘挂载到容器中，只需在创建负载时添加云硬盘卷，并配置大小、挂载路径（也就是容器卷的路径）即可。



- ENTRYPOINT与CMD对应云容器实例中高级配置的启动命令，详细内容请参见[容器启动命令](#)。



- EXPOSE即暴露某个端口，通常在启动容器时配合**docker run -p <宿主端口>:<容器端口>**一起使用，云容器实例中容器如果要对外暴露端口，只需在创建负载的时候配置**负载访问端口:容器端口**的映射，这样就可以通过**负载请求域名:负载访问端口**访问到容器。



3 负载管理

3.1 CCI 应用进行优雅滚动升级

应用场景

用户在CCI中部署工作负载时，应用发布成了LoadBalance类型的Service或Ingress且对接的独享型ELB，经过ELB的访问流量支持直通到容器中；当应用进行滚动升级或者弹性扩缩容，通过配置容器探针，最短就绪时间等可以做到优雅升级，从而实现优雅弹性扩缩容（在升级或者扩缩容过程中业务不会出现5xx的错误响应）。

操作步骤

在此以nginx的无状态工作负载为例，提供了CCI中应用进行优雅滚动升级或者弹性扩缩容最佳实践。

- 步骤1** 在CCI控制台，单击左侧栏目树中的“工作负载 > 无状态 Deployment”，单击右上角“镜像创建”。

图 3-1 创建无状态负载



- 步骤2** 在“容器配置”，单击“使用该镜像”，选择镜像完成。

- 步骤3** 在“容器设置”，单击展开“高级设置 > 健康检查 > 应用业务探针”，如下图设置工作负载业务探针。

图 3-2 配置应用业务探针



说明

该配置是检查用户业务是否就绪，不就绪则不转发流量到当前实例。

步骤4 单击展开“生命周期”，配置容器的“停止前处理”，保证容器在退出过程中能够对外提供服务。

图 3-3 配置生命周期



说明

该配置是保证业务容器在退出过程中能够对外提供服务。

步骤5 单击“下一步：访问设置”，如**图3-4**。

图 3-4 配置访问类型及端口

负载访问

访问类型 内网访问 公网访问 不启用
将工作负载提供一个可以从Internet访问的入口，支持HTTP协议并根据URL转发请求，如WordPress等前台类服务可以选择公网访问。 [如何配置负载公网访问](#)

* 服务名称

* ELB实例 [创建共享型ELB实例](#)

ELB协议 HTTP/HTTPS TCP/UDP

* 负载端口协议 TCP UDP

* 负载端口配置 (设置ELB端口与容器端口映射关系；负载请求由负载域名:ELB端口 转发至 容器实例:容器端口)

ELB端口(未占用)	容器端口	操作
<input type="text" value="6044"/>	<input type="text" value="80"/>	删除

[添加端口](#)

步骤6 单击“下一步”完成工作负载的创建。

步骤7 配置最短就绪时间。

最短就绪时间，用于指定新创建的Pod在没有任意容器崩溃情况下的最小就绪时间，只有超出这个时间Pod才被视为可用。

“最短就绪时间”需在右上角的“YAML编辑”进行配置。如图3-5:

图 3-5 配置最短就绪时间

```
54 /bin/bash
55 - '-c'
56 - sleep 30
57   terminationMessagePath: /dev/termination-log
58   terminationMessagePolicy: File
59   imagePullPolicy: IfNotPresent
60   restartPolicy: Always
61   terminationGracePeriodSeconds: 30
62   dnsPolicy: ClusterFirst
63   securityContext: {}
64   imagePullSecrets:
65     - name: imagepull-secret
66   schedulerName: default-scheduler
67   dnsConfig: {}
68   strategy:
69     type: RollingUpdate
70     rollingUpdate:
71       maxUnavailable: 1
72       maxSurge: 0
73   minReadySeconds: 10
74   revisionHistoryLimit: 10
75   progressDeadlineSeconds: 600
76   status:
77     observedGeneration: 2
78     replicas: 2
79     updatedReplicas: 2
80     readyReplicas: 2
81     availableReplicas: 2
82     conditions:
83       - type: Available
84         status: 'True'
85         lastUpdateTime: '2021-08-24T09:16:17Z'
86         lastTransitionTime: '2021-08-24T09:16:17Z'
87         reason: MinimumReplicasAvailable
88         message: Deployment has minimum availability.
89       - type: Progressing
90         status: 'True'
91         lastUpdateTime: '2021-08-24T09:16:23Z'
```

说明

- 推荐的配置minReadySeconds时长，为业务容器的启动预期时间加上ELB服务下发member到生效的时间。
- minReadySeconds的时长需要小于sleep时长，保证旧的容器停止并退出之前，新的容器已经准备就绪。

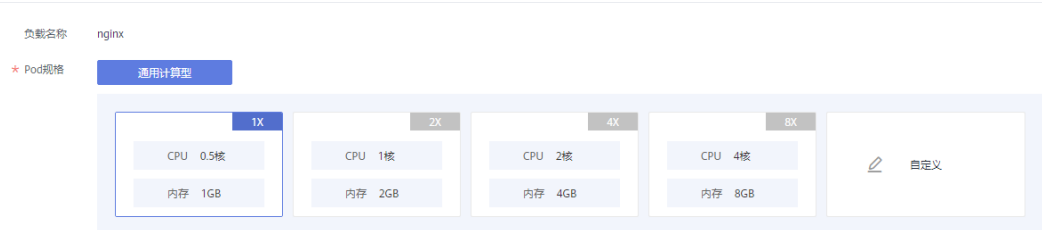
步骤8 配置完成后，对应用进行升级和弹性扩缩容的打流测试。

准备一台集群外的客户端节点，预置检测脚本detection_script.sh，内容如下，其中100.85.125.90:7552为service的公网访问地址：

```
#!/bin/bash
for ((;;))
do
    curl -I 100.85.125.90:7552 | grep "200 OK"
    if [ $? -ne 0 ]; then
        echo "response error!"
        exit 1
    fi
done
```

步骤9 运行检测脚本：`bash detection_script.sh`，并在CCI界面触发应用的滚动升级，如图3-6修改了容器规格，触发了应用的滚动升级。

图 3-6 修改容器规格



滚动升级的过程中，应用的访问并未中断，并且返回的请求都是“200OK”，说明升级过程是优雅升级，没有中断的。

----结束

3.2 在容器中通过环境变量获取 Pod 基础信息

客户如果需要在容器内获取POD的基础信息，可以通过kubernetes中的 [Downward API](#)注入环境变量的方式实现。本操作实践展示如何在Deployment和POD的定义中增加环境变量配置，获取Pod的namespace、name、uid、IP、Region和AZ。

CCI创建Pod并分配节点的同时，Pod Annotations中新增所在节点的region和az信息。

此时Pod中Annotations格式为：

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    topology.kubernetes.io/region: "{{region}}"
    topology.kubernetes.io/zone: "{{available-zone}}"
```

topology.kubernetes.io/region为所在节点的region信息。

topology.kubernetes.io/zone为所在节点的az信息。

Deployment 配置示例

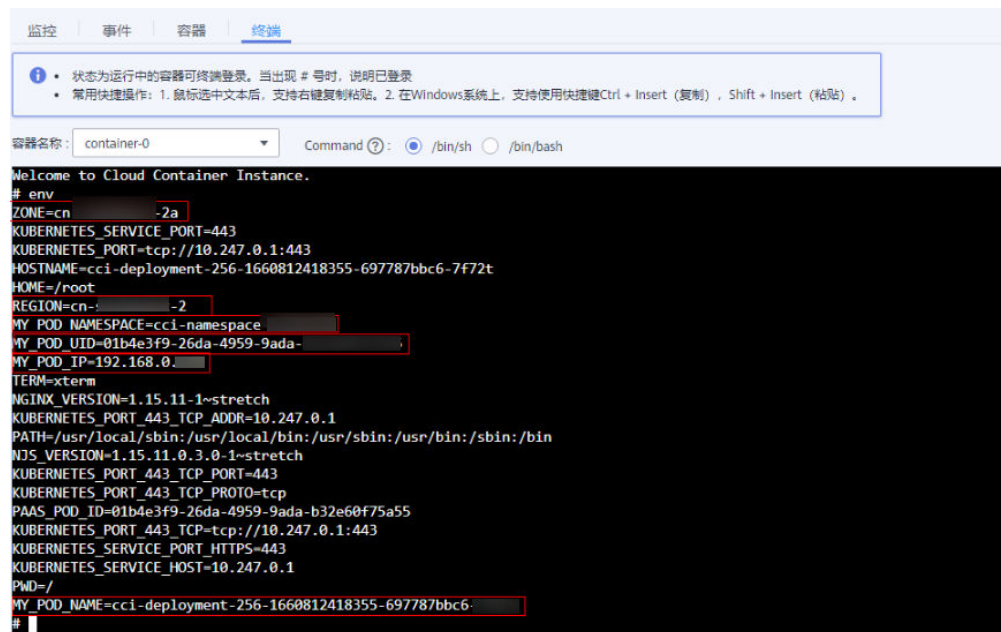
通过环境变量获取Pod基础信息，示例如下：

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: cci-downwardapi-test
  namespace: cci-test # 填写具体的命名空间
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cci-downwardapi-test
  template:
    metadata:
      labels:
        app: cci-downwardapi-test
    spec:
      containers:
        - name: container-0
```

```
image: 'library/euleros:latest'
command:
- /bin/bash
- '-c'
- while true; do echo hello; sleep 10; done
env:
- name: MY_POD_UID
  valueFrom:
    fieldRef:
      fieldPath: metadata.uid
- name: MY_POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
- name: MY_POD_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: MY_POD_IP
  valueFrom:
    fieldRef:
      fieldPath: status.podIP
- name: REGION
  valueFrom:
    fieldRef:
      fieldPath: metadata.annotations['topology.kubernetes.io/region']
- name: ZONE
  valueFrom:
    fieldRef:
      fieldPath: metadata.annotations['topology.kubernetes.io/zone']
resources:
  limits:
    cpu: 500m
    memory: 1Gi
  requests:
    cpu: 500m
    memory: 1Gi
```

负载运行起来后就可以通过环境变量在容器内查看到具体的Pod信息：

图 3-7 Pod 基础信息



```
Welcome to Cloud Container Instance.
# env
ZONE=cn-2a
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.247.0.1:443
HOSTNAME=cci-deployment-256-1660812418355-697787bbc6-7f72t
HOME=/root
REGION=cn-2
MY_POD_NAMESPACE=cci-namespace
MY_POD_UID=01b4e3f9-26da-4959-9ada-
MY_POD_IP=192.168.0.
TERM=xterm
NGINX_VERSION=1.15.11-1~stretch
KUBERNETES_PORT_443_TCP_ADDR=10.247.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NDS_VERSION=1.15.11.0.3.0-1~stretch
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
PAAS_POD_ID=01b4e3f9-26da-4959-9ada-b32e60f75a55
KUBERNETES_PORT_443_TCP=tcp://10.247.0.1:443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_HOST=10.247.0.1
PWD=/
MY_POD_NAME=cci-deployment-256-1660812418355-697787bbc6-
#
```

3.3 内核参数配置

CCI服务底座使用安全容器构建了业内领先的Serverless容器平台，同物理机系统内核隔离且互不影响。对于资深业务部署场景，内核参数调优是比较通用的方式。在安全范围内，CCI服务允许客户根据Kubernetes社区推荐的方案，通过Pod的安全上下文（Security Context）对内核参数进行配置，极大提升用户业务部署的灵活性。如果你对securityContext概念不够熟悉，更多信息可阅读[Security Context](#)。

在Linux中，最通用的内核参数修改方式是通过sysctl接口进行配置。在Kubernetes中，也是通过Pod的sysctl安全上下文（Security Context）对内核参数进行配置，如果你对sysctl概念不够熟悉，可阅读[在Kubernetes集群中使用sysctl](#)。安全上下文（Security Context）作用于同一个Pod内的所有容器。

CCI服务支持修改的内核参数范围如下：

```
kernel.shm*,
kernel.msg*,
kernel.sem,
fs.mqueue.*,
net.* ( net.netfilter.*和net.ipv4.vs.*除外)
```

以下示例中，使用Pod SecurityContext来对两个sysctl参数net.core.somaxconn和net.ipv4.tcp_tw_reuse进行设置。

```
apiVersion:v1
kind:Pod
metadata:
  name: xxxxx
  namespace: auto-test-namespace
spec:
  securityContext:
    sysctls:
      - name: net.core.somaxconn
        value: "65536"
      - name: net.ipv4.tcp_tw_reuse
        value: "1"
  ...
```

进入容器确认配置生效：

```
[root@master-2 ~]# kubectl get pod -n auto-test-namespace
NAME                                READY   STATUS    RESTARTS   AGE
cci-deployment-20225241-76dff9f854-6fwlm  1/1     Running   0           15m
cci-deployment-20225241-76dff9f854-nwst7  1/1     Running   0           29m
[root@master-2 ~]# kubectl exec -it cci-deployment-20225241-76dff9f854-nwst7 /bin/bash -n auto-test-namespace
root@cci-deployment-20225241-76dff9f854-nwst7:/#
root@cci-deployment-20225241-76dff9f854-nwst7:/# cat /proc/sys/net/core/somaxconn
65536
root@cci-deployment-20225241-76dff9f854-nwst7:/# cat /proc/sys/net/ipv4/tcp_tw_reuse
1
root@cci-deployment-20225241-76dff9f854-nwst7:/#
```

3.4 修改/dev/shm 容量大小

应用场景

/dev/shm由tmpfs文件系统构成，tmpfs是Linux/Unix系统上的一种基于内存的文件系统，故读写效率非常高。

目前有用户希望通过/dev/shm实现进程间数据交互或通过/dev/shm实现临时数据存储，此时CCI场景/dev/shm默认大小64M无法满足客户诉求，故提供修改/dev/shm size大小的能力。

本操作实践展示通过“memory类型EmptyDir”和“配置securityContext与mount命令”两种方式修改/dev/shm容量。

限制与约束

- /dev/shm使用基于内存的tmpfs文件系统，不具备持久性，容器重启后数据不保留。
- 用户可通过两种方式修改/dev/shm容量，但不建议在一个Pod中同时使用两种方式进行配置。
- EmptyDir所使用的memory从Pod申请的memory中进行分配，不会额外占用资源。
- 在/dev/shm中写数据相当于申请内存，此种场景下需评估进程内存使用量，当容器内的进程申请内存与EmptyDir中数据量之和超过容器请求的限制内存时，会出现内存溢出异常。
- 当需要修改/dev/shm容量时，容量大小通常设定为Pod内存申请量的50%。

通过 memory 类型 EmptyDir 修改/dev/shm 容量

临时路径(EmptyDir)：适用于临时存储、灾难恢复、共享运行时数据等场景，任务实例的删除或迁移会导致临时路径被删除。

CCI支持挂载Memory类型的EmptyDir，用户可通过指定EmptyDir分配内存的大小并挂载到容器内/dev/shm目录来实现/dev/shm的容量修改。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-emptydir-name
spec:
  containers:
  - image: 'library/ubuntu:latest'
    volumeMounts:
    - name: volume-emptydir1
      mountPath: /dev/shm
    name: container-0
  resources:
    limits:
      cpu: '4'
      memory: 8Gi
    requests:
      cpu: '4'
      memory: 8Gi
  volumes:
  - emptyDir:
      medium: Memory
      sizeLimit: 4Gi
    name: volume-emptydir1
```

待Pod启动后，执行“df -h”指令，进入/dev/shm目录，如下图所示，/dev/shm容量修改成功。

图 3-8 /dev/shm 目录详情

```
root@pod-emptydir-name:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/vdc        20G  182M   19G   1% /
tmpfs           64M    0    64M   0% /dev
tmpfs           4.0G    0   4.0G   0% /sys/fs/cgroup
tmpfs           4.0G   52K   4.0G   1% /etc/hosts
kataShared      20G   45M   19G   1% /dev/termination-log
shm             4.0G    0   4.0G   0% /dev/shm
tmpfs           4.0G    0   4.0G   0% /proc/acpi
tmpfs           4.0G    0   4.0G   0% /proc/scsi
tmpfs           4.0G    0   4.0G   0% /sys/firmware
```

通过配置 securityContext 和 mount 命令修改/dev/shm 容量

- 容器赋予SYS_ADMIN权限

linux原生提供了SYS_ADMIN权限，将该权限应用于容器中，首先需要kubernetes在pod级别带入这个信息，在pod的描述文件中添加securityContext字段的描述。例如：

```
"securityContext": {
  "capabilities": {
    "add": [
      "SYS_ADMIN"
    ]
  }
}
```

同时容器的描述信息中也需要加入另外一个描述字段CapAdd。

```
"CapAdd": [
  "SYS_ADMIN"
],
```

这样的话容器在自动被kubelet拉起的时候就会带入一个参数。

```
docker run --cap-add=SYS_ADMIN
```

- 在给容器赋予SYS_ADMIN权限后，可直接在启动命令中通过mount命令实现/dev/shm的size修改。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-emptydir-name
spec:
  containers:
    - command:
      - /bin/sh
      - '-c'
      - mount -o size=4096M -o remount /dev/shm;bash
      securityContext:
        capabilities:
          add: ["SYS_ADMIN"]
      image: 'library/ubuntu:latest'
      name: container-0
      resources:
        limits:
          cpu: '4'
          memory: 8Gi
        requests:
```

```
cpu: '4'  
memory: 8Gi
```

待Pod启动后，执行“df -h”指令，进入/dev/shm目录，如下图所示，/dev/shm容量修改成功。

图 3-9 /dev/shm 目录详情

```
root@pod-emptydir-name:/# df -h  
Filesystem      Size  Used Avail Use% Mounted on  
/dev/vdc        20G  182M   19G   1% /  
tmpfs           64M    0    64M   0% /dev  
tmpfs           4.0G    0   4.0G   0% /sys/fs/cgroup  
tmpfs           4.0G   52K   4.0G   1% /etc/hosts  
kataShared      20G   45M   19G   1% /dev/termination-log  
shm             4.0G    0   4.0G   0% /dev/shm  
tmpfs           4.0G    0   4.0G   0% /proc/acpi  
tmpfs           4.0G    0   4.0G   0% /proc/scsi  
tmpfs           4.0G    0   4.0G   0% /sys/firmware
```