云容器实例（CCI 1.0）

最佳实践

文档版本 01

发布日期 2025-08-14

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 弹性伸缩

1.1 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。

约束与限制

- 仅支持VPC网络模式的CCE Standard集群和CCE Turbo集群。
- CCE突发弹性引擎（对接 CCI）插件1.5.37及以下版本不支持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。
- 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。
- Volcano调度器1.17.10及以下版本暂不支持使用Volcano调度器将挂载云存储卷的容器实例（Pod）弹性到CCI。
- 使用CCE集群中的Bursting插件对接CCI 2.0服务，支持配置独享型ELB的Ingress和Service。Bursting插件1.5.5以下版本不支持配置ELB类型的Service。

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 1-1 插件参数说明

插件参数	说明
选择版本	插件的版本。

规格配置	用于配置插件负载的实例数及资源配额。
	● 选择“系统预置规格”时，您可选择“单实例”或“高可用”规格。
	● 选择“自定义规格”时，您可根据需求修改插件各个组件的副本数以及CPU/内存配置。
	说明
	● CCE 突发弹性引擎 (对接 CCI) 插件在1.5.2及以上版本，将占用更多节点资源，请在升级CCE突发弹性引擎（对接 CCI）插件前预留空间配额。
	  ● 单实例：需要预留一个节点，节点下至少需要有7个Pod空间配额。若开启网络互通，则需要有8个Pod空间配额。
	  ● 高可用：需要预留两个节点，节点下至少需要有7个Pod空间配额，共计14个Pod空间配额。若开启网络互通，则需要有8个Pod空间配额，共计16个Pod空间配额。
	● 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。

网络互通	开启后，支持CCE集群中的Pod与CCI集群中的Pod通过Kubernetes Service互通，并在插件安装时部署组件proxy。

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”。

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 1-2 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。
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。

1.2 VK 支持的 Annotation 列表以及用法

1.2.1 HostPath 转 localDir、emptyDir 或 flexVolume

使用场景

在使用CCE或者其他K8s集群时，可以使用HostPath。由于CCI为共享集群，未开放HostPath能力，所以当使用HostPath的Pod想通过Virtual Kubelet弹到CCI时，会被vk-webhook拦截。如无法改变Pod spec.volumes中配置的HostPath，当前可通过配置Annotation的形式，允许让使用HostPath的Pod弹性到CCI上，但是VK在校验时需要去掉Pod中的HostPath或者将HostPath替换为localDir、emptyDir或者flexVolume。

通过在Pod.Annotations中加入注解可以做到HostPath转localDir、emptyDir或者flexVolume。

- 单个HostPath替换为localDir配置方式：
  ```
  "coordinator.cci.io/hostpath-replacement": '[{"name":"source-hostpath-volume","policyType":"replaceByLocalDir","localDir":{"sizeLimit":"1Gi"}}]'
  ```
- 单个HostPath替换为flexVolume配置方式：
  ```
  "coordinator.cci.io/hostpath-replacement": '[{"name":"source-hostpath-volume","policyType":"replaceByFlexVolume","flexVolume":{"driver": "huawei.com/fuxidisk", "fsType": "ext4", "options": {"volumeType": "sata", "volumeSize": "10Gi"}}}]'
  ```
  EVS目前支持普通、高I/O、超高I/O三种规格，volumeType根据实际情况填写：

  表 1-3 EVS 规格

EVS规格	类型	适用场景
普通I/O	sata	后端存储由SATA存储介质提供，适用于大容量，读写速率要求不高，事务性处理较少的应用场景，如：开发测试、企业办公应用。

EVS规格	类型	适用场景
高I/O	sas	后端存储由SAS存储介质提供，适用于性能相对较高，读写速率要求高，有实时数据存储需求应用场景，如：创建文件系统、分布式文件共享。
超高I/O	ssd	后端存储SSD存储介质提供，适用于高性能、高读写速率要求、数据密集型应用场景，如：NoSQL、关系型数据库、数据仓库（如Oracle RAC、SAP HANA)。

- 全部hostPath都忽略：
  ```
  "coordinator.cci.io/hostpath-replacement": '[{"name":"*","policyType":"remove"}]'
  ```
- 多个hostPath差异化替换策略：
  ```
  "coordinator.cci.io/hostpath-replacement": '[{"name":"source-hostpath-volume-1","policyType":"remove"},{"name":"source-hostpath-volume-2","policyType":"replaceByLocalDir","localDir":{"sizeLimit":"1Gi"}},{"name":"source-hostpath-volume-3","policyType":"replaceByEmptyDir","emptyDir":{"sizeLimit":"10Gi"}}]'
  ```

示例deployment：

```
apiVersion: apps/v1
 kind: Deployment
 metadata:
   annotations:
     description:''
   labels:
     virtual-kubelet.io/burst-to-cci: enforce
     appgroup:''
     version: v1
   name: test
   namespace: default
 spec:
   replicas:2
   selector:
     matchLabels:
       app: test
       version: v1
   template:
     metadata:
       labels:
         app: test
         version: v1
       annotations:
         coordinator.cci.io/hostpath-replacement:'[{"name": "test-log2", "policyType": "remove"}, {"name": "test-log", "policyType": "replaceByEmptyDir", "emptyDir":{"sizeLimit":"10Gi"}}, {"name": "test-log1", "policyType": "replaceByLocalDir", "localDir":{"sizeLimit":"1Gi"}}]'
     spec:
       containers:
         -name: container-1
           image: nginx
           imagePullPolicy: IfNotPresent
           env:
             -name: PAAS_APP_NAME
               value: test
             -name: PAAS_NAMESPACE
               value: default
             -name: PAAS_PROJECT_ID
               value: 0b52a6e40b00d3682f36c0005163a82c
           resources:
             requests:
               cpu: 250m
               memory: 512Mi
             limits:
```

```
            cpu: 250m
            memory: 512Mi
        volumeMounts:
          -name: test-log
            mountPath: /tmp/log
          -name: test-log1
            mountPath: /tmp/log1
          -name: test-log2
            mountPath: /tmp/log2
    volumes:
    -hostPath:
        path: /var/paas/sys/log/virtual-kubelet
        type:""
      name: test-log
    -hostPath:
        path: /var/paas/sys/log
        type:""
      name: test-log1
    -hostPath:
        path: /var/paas/sys/log2
        type:""
      name: test-log2
```

1.2.2 镜像地址替换

使用场景

用户在线下IDC有自建的harbor镜像仓库，同时也会将镜像同步到华为云SWR镜像仓库。创建负载后，希望调度到自建Kubernetes集群节点上运行的Pod使用自建镜像仓库的镜像，弹性到CCI的Pod使用SWR的镜像，以提高镜像拉取效率。可通过在Pod.Annotations中加入注解可以实现弹性到CCI的Pod中容器使用的镜像与自建Kubernetes集群使用的镜像为不同地址。

通过Annotation "coordinator.cci.io/image-replacement"配置镜像替换策略。其值为如下所示json字符串：

```
[
  {
    "repositoryPrefix":"harbor.domain/a/b/c/d",
    "replaceWith":"swr.cn-north-4.myhuaweicloud.com/org"
  }
]
```

相关字段说明

表 1-4 相关字段说明

字段	说明
repository Prefix	不为空时表示需要被替换的镜像名前缀，即镜像名中最后一个"/"字符之前的部分。此时该字段字符校验规则与容器镜像名的规则一致，且不能以"/"字符结尾。该字段也可为空。
replaceWith	待替换字符串。该字段字符校验规则与容器镜像名的规则一致，且不能以"/"字符结尾。该值不能与repositoryPrefix相同。

匹配与替换规则

- 替换策略可配置多条。各条替换策略的原字符串（repositoryPrefix）不允许重复。每个容器至多只会执行一条替换策略。配置替换策略无需考虑先后顺序，不同的配置顺序结果一致。
- 原Pod.spec中配置的container和initContainer镜像均支持替换。
- 若repositoryPrefix不为空，则匹配Pod中所有镜像名中最后一个"/"字符之前的字符串与该值相同的容器，对其最后一个"/"字符之前的部分替换为replaceWith的内容。
- 若repositoryPrefix为空，则对Pod中镜像名不包含前缀（即没有"/"字符）的容器，镜像名前加上replaceWith的内容和一个"/"字符。

配置场景一

原Pod中所有容器镜像均使用同一镜像仓库与组织，配置一条替换策略即可。

示例：将所有容器镜像"harbor.domain/a/b/c/d"前缀都替换为SWR的镜像前缀"swr.cn-north-4.myhuaweicloud.com/org"。

替换策略：

```
"coordinator.cci.io/image-replacement": '[{"repositoryPrefix":"harbor.domain/a/b/c/d","replaceWith":"swr.cn-north-4.myhuaweicloud.com/org"}]'
```

替换前：

```
    containers:
      - name: container-0
        image: 'harbor.domain/a/b/c/d/ubuntu:latest'
      - name: container-1
        image: 'harbor.domain/a/b/c/d/nginx:latest'
```

替换后：

```
    containers:
      - name: container-0
        image: 'swr.cn-north-4.myhuaweicloud.com/org/ubuntu:latest'
      - name: container-1
        image: 'swr.cn-north-4.myhuaweicloud.com/org/nginx:latest'
```

示例Deployment：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-vk
  labels:
    virtual-kubelet.io/burst-to-cci: 'auto'
spec:
  replicas: 20
  selector:
    matchLabels:
      app: test-vk
  template:
    metadata:
      labels:
        app: test-vk
      annotations:
        coordinator.cci.io/image-replacement: '[{"repositoryPrefix":"harbor.domain/a/b/c/
d","replaceWith":"swr.cn-north-4.myhuaweicloud.com/org"}]'
    spec:
      containers:
      - name: container-0
        image: harbor.domain/a/b/c/d/ubuntu:latest
```

```
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
         cpu: 500m
         memory: 1024Mi
    command:
      - /bin/bash
      - '-c'
      - sleep 10000
  - name: container-1
    image: harbor.domain/a/b/c/d/nginx:latest
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
         cpu: 500m
         memory: 1024Mi
    command:
      - /bin/bash
      - '-c'
      - sleep 10000
```

配置场景二

原Pod中容器使用不同的镜像仓库，可配置多条替换策略。

示例：

1. 第一个容器匹配到第一条策略，将"harbor.domain"替换为"swr.cn-north-4.myhuaweicloud.com/org1"。
2. 第二个容器匹配到第二条策略，将镜像名前加上"swr.cn-north-4.myhuaweicloud.com/org1"和"/"。
3. 第三个容器匹配到第三条策略，将前缀"harbor.domain/a/b/c/d"替换为"swr.cn-north-4.myhuaweicloud.com/org2"。

   因为需要完整匹配最后一个"/"之前的部分，第一条策略即便有与该镜像相同的前缀"harbor.domain"，也不会生效。

替换策略：

```
"coordinator.cci.io/image-replacement": '[{"repositoryPrefix":"harbor.domain","replaceWith":"swr.cn-north-4.myhuaweicloud.com/org1"},{"repositoryPrefix":"","replaceWith":"swr.cn-north-4.myhuaweicloud.com/org1"},{"repositoryPrefix":"harbor.domain/a/b/c/d","replaceWith":"swr.cn-north-4.myhuaweicloud.com/org2"}]'
```

替换前：

```
  containers:
    - name: container-0
      image: 'harbor.domain/ubuntu:latest'
    - name: container-1
      image: 'nginx:latest'
    - name: container-2
      image: 'harbor.domain/a/b/c/d/redis:latest'
```

替换后：

```
  containers:
    - name: container-0
      image: 'swr.cn-north-4.myhuaweicloud.com/org1/ubuntu:latest'
    - name: container-1
      image: 'swr.cn-north-4.myhuaweicloud.com/org1/nginx:latest'
```

```
    - name: container-2
      image: 'swr.cn-north-4.myhuaweicloud.com/org2/redis:latest'
```

2 负载创建

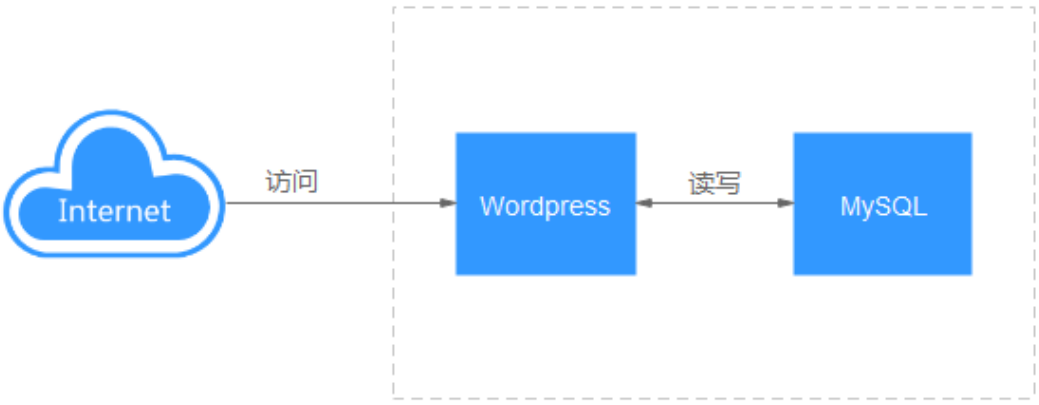
2.1 概述

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。

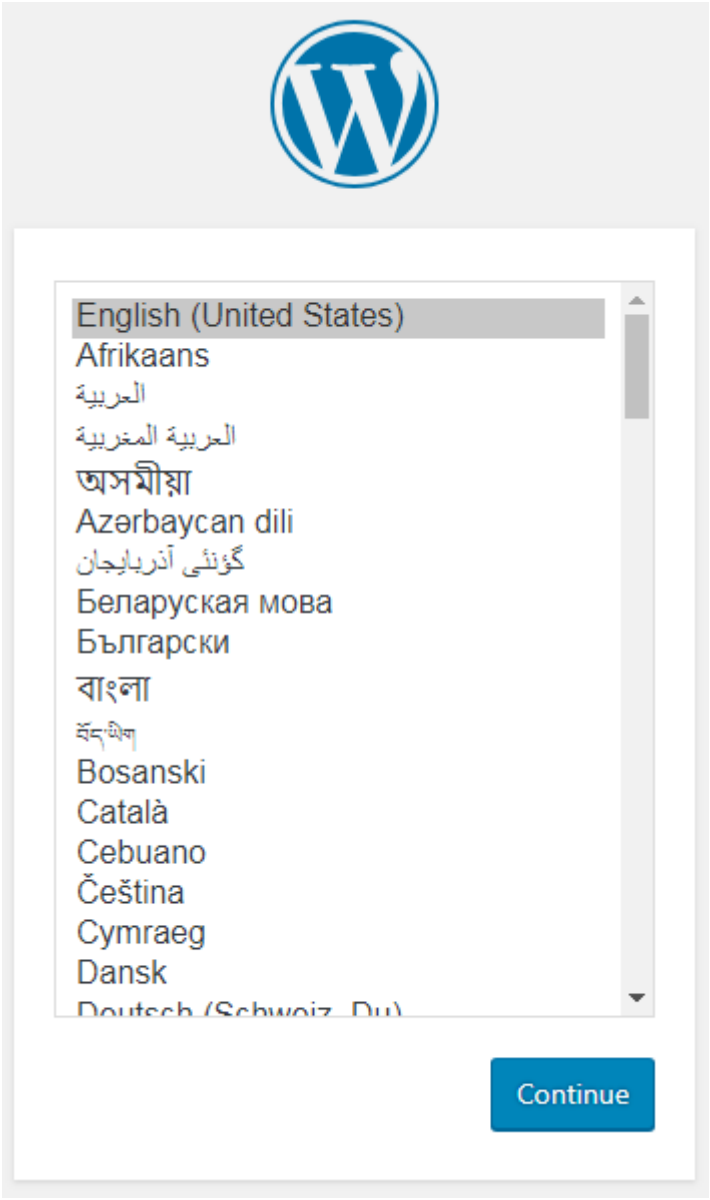
执行下面的命令运行Wordpress。

```
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。

- **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。

- 容器配置：

  a. 在开源镜像中心搜索并选择wordpress镜像。

  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。

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”。

----结束

2.4 调用 API 创建负载

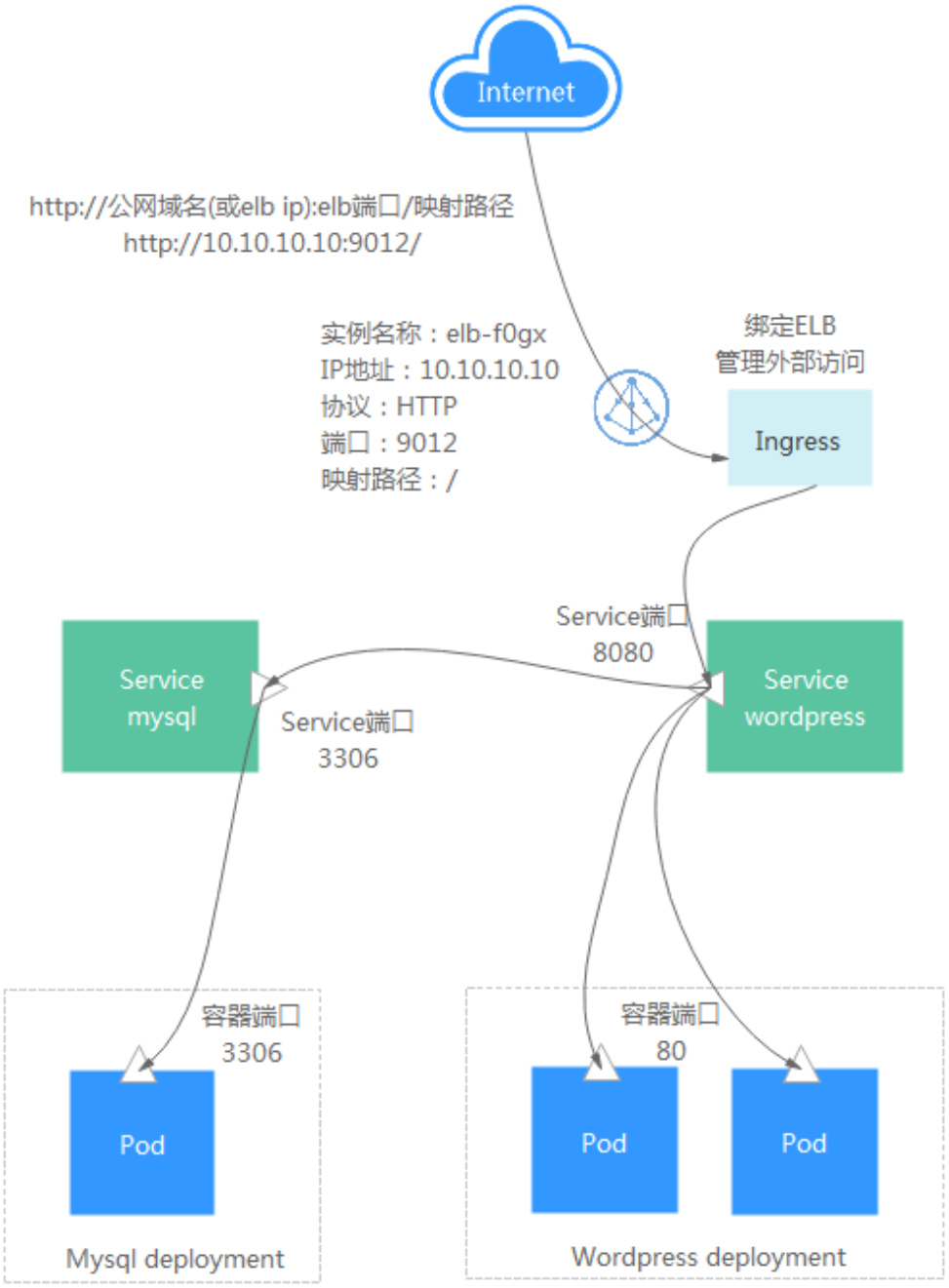
云容器实例原生支持Kubernetes API，相比从控制台创建负载，使用API的粒度更细一些。

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**，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它们之间的关系，这样您就可以更好地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

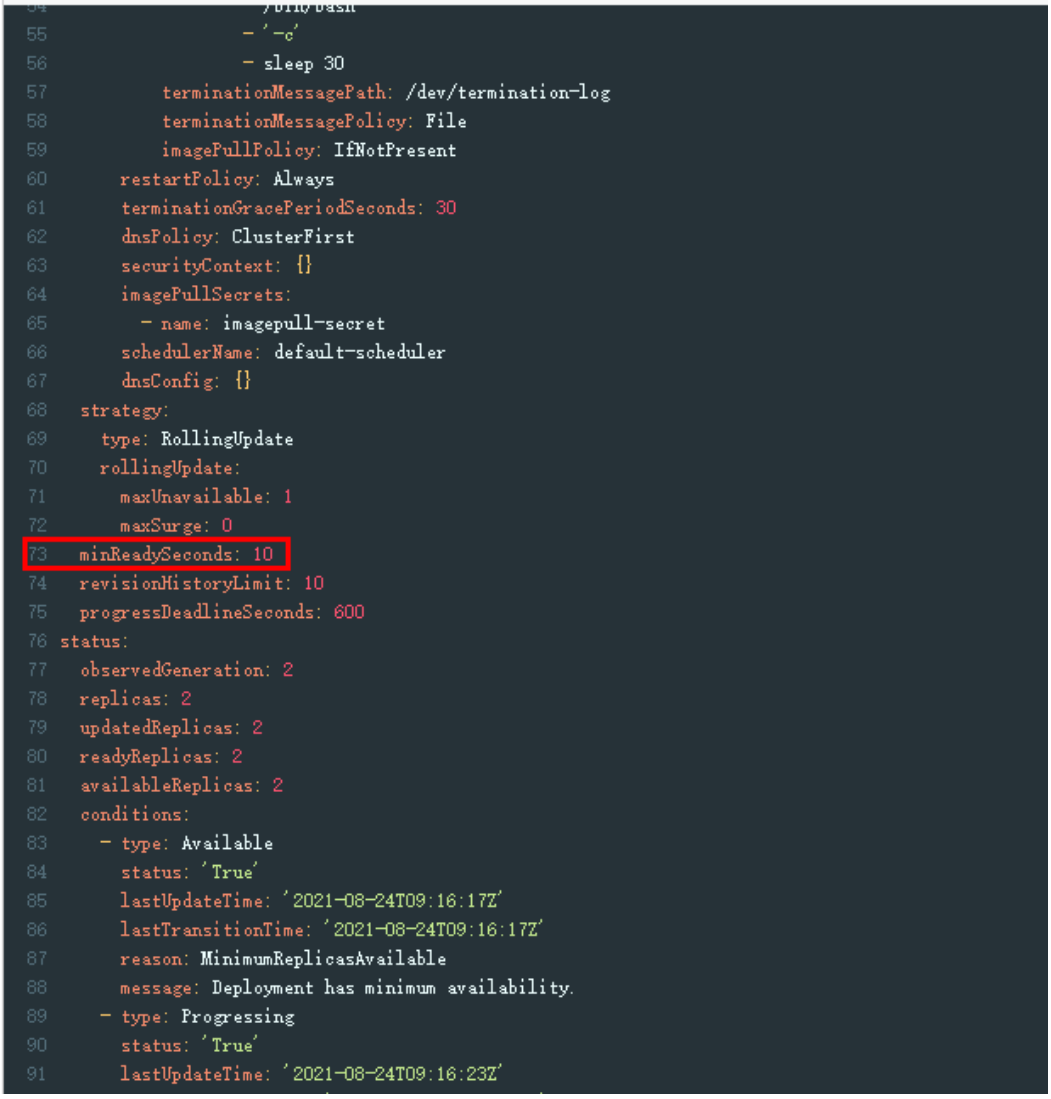
步骤6 单击“下一步”完成工作负载的创建。

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



```
                  /bin/bash
                - '-c'
                - sleep 30
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: IfNotPresent
    restartPolicy: Always
    terminationGracePeriodSeconds: 30
    dnsPolicy: ClusterFirst
    securityContext: {}
    imagePullSecrets:
      - name: imagepull-secret
    schedulerName: default-scheduler
    dnsConfig: {}
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 0
minReadySeconds: 10
revisionHistoryLimit: 10
progressDeadlineSeconds: 600
status:
  observedGeneration: 2
  replicas: 2
  updatedReplicas: 2
  readyReplicas: 2
  availableReplicas: 2
  conditions:
    - type: Available
      status: 'True'
      lastUpdateTime: '2021-08-24T09:16:17Z'
      lastTransitionTime: '2021-08-24T09:16:17Z'
      reason: MinimumReplicasAvailable
      message: Deployment has minimum availability.
    - type: Progressing
      status: 'True'
      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

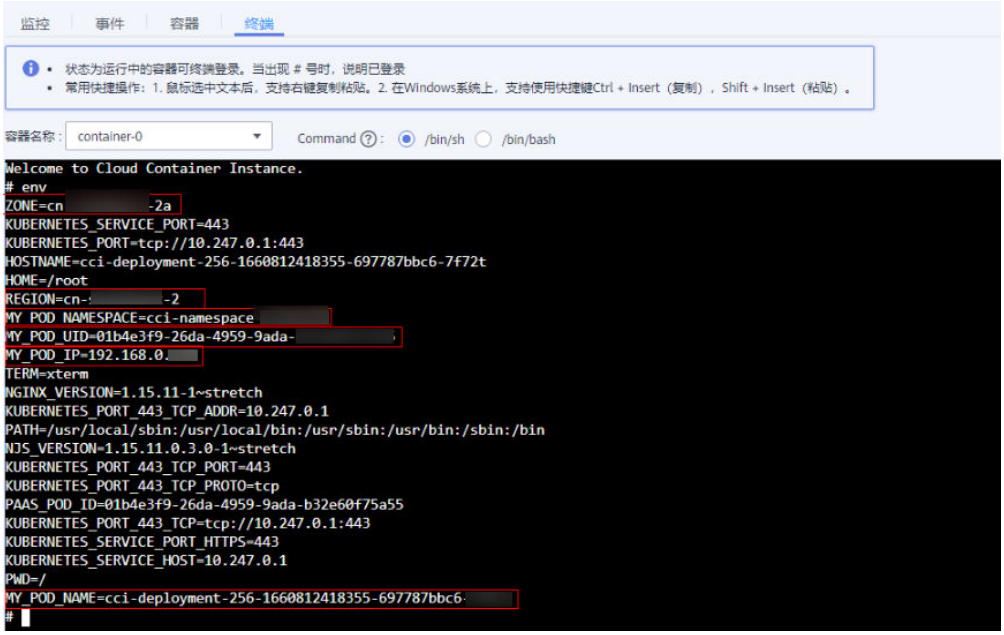
通过环境变量获取Pod基础信息，示例如下：

```
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



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。如果您对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，如果您对**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_rmid_forced",
"kernel.shmall",
"kernel.shmmax",
"kernel.shmmni",
"kernel.msgmax",
"kernel.msgmnb",
"kernel.msgmni",
"kernel.sem",
"fs.mqueue.msg_default",
"fs.mqueue.msg_max",
"fs.mqueue.msgsize_default",
"fs.mqueue.msgsize_max",
"fs.mqueue.queues_max",
"net.core.busy_poll",
"net.core.busy_read",
"net.core.default_qdisc",
"net.core.dev_weight",
"net.core.dev_weight_rx_bias",
"net.core.dev_weight_tx_bias",
"net.core.fb_tunnels_only_for_init_net",
"net.core.flow_limit_cpu_bitmap",
"net.core.flow_limit_table_len",
"net.core.max_skb_frags",
"net.core.message_burst",
"net.core.message_cost",
"net.core.netdev_budget",
"net.core.netdev_budget_usecs",
"net.core.netdev_max_backlog",
"net.core.netdev_rss_key",
"net.core.netdev_tstamp_prequeue",
"net.core.optmem_max",
"net.core.rmem_default",
"net.core.rmem_max",
"net.core.rps_sock_flow_entries",
"net.core.somaxconn",
"net.core.tstamp_allow_data",
"net.core.warnings",
"net.core.wmem_default",
"net.core.wmem_max",
"net.core.xfrm_acq_expires",
"net.core.xfrm_aevent_etime",
"net.core.xfrm_aevent_rseqth",
"net.core.xfrm_larval_drop",
"net.ipv4.conf.all.accept_local",
"net.ipv4.conf.all.accept_redirects",
"net.ipv4.conf.all.accept_source_route",
"net.ipv4.conf.all.arp_accept",
"net.ipv4.conf.all.arp_announce",
"net.ipv4.conf.all.arp_filter",
```

```
            "net.ipv4.conf.all.arp_ignore",
            "net.ipv4.conf.all.arp_notify",
            "net.ipv4.conf.all.bc_forwarding",
            "net.ipv4.conf.all.bootp_relay",
            "net.ipv4.conf.all.disable_policy",
            "net.ipv4.conf.all.disable_xfrm",
            "net.ipv4.conf.all.drop_gratuitous_arp",
            "net.ipv4.conf.all.drop_unicast_in_l2_multicast",
            "net.ipv4.conf.all.force_igmp_version",
            "net.ipv4.conf.all.forwarding",
            "net.ipv4.conf.all.igmpv2_unsolicited_report_interval",
            "net.ipv4.conf.all.igmpv3_unsolicited_report_interval",
            "net.ipv4.conf.all.ignore_routes_with_linkdown",
            "net.ipv4.conf.all.log_martians",
            "net.ipv4.conf.all.mc_forwarding",
            "net.ipv4.conf.all.medium_id",
            "net.ipv4.conf.all.promote_secondaries",
            "net.ipv4.conf.all.proxy_arp",
            "net.ipv4.conf.all.proxy_arp_pvlan",
            "net.ipv4.conf.all.route_localnet",
            "net.ipv4.conf.all.rp_filter",
            "net.ipv4.conf.all.secure_redirects",
            "net.ipv4.conf.all.send_redirects",
            "net.ipv4.conf.all.shared_media",
            "net.ipv4.conf.all.src_valid_mark",
            "net.ipv4.conf.all.tag",
            "net.ipv4.conf.default.accept_local",
            "net.ipv4.conf.default.accept_redirects",
            "net.ipv4.conf.default.accept_source_route",
            "net.ipv4.conf.default.arp_accept",
            "net.ipv4.conf.default.arp_announce",
            "net.ipv4.conf.default.arp_filter",
            "net.ipv4.conf.default.arp_ignore",
            "net.ipv4.conf.default.arp_notify",
            "net.ipv4.conf.default.bc_forwarding",
            "net.ipv4.conf.default.bootp_relay",
            "net.ipv4.conf.default.disable_policy",
            "net.ipv4.conf.default.disable_xfrm",
            "net.ipv4.conf.default.drop_gratuitous_arp",
            "net.ipv4.conf.default.drop_unicast_in_l2_multicast",
            "net.ipv4.conf.default.force_igmp_version",
            "net.ipv4.conf.default.forwarding",
            "net.ipv4.conf.default.igmpv2_unsolicited_report_interval",
            "net.ipv4.conf.default.igmpv3_unsolicited_report_interval",
            "net.ipv4.conf.default.ignore_routes_with_linkdown",
            "net.ipv4.conf.default.log_martians",
            "net.ipv4.conf.default.mc_forwarding",
            "net.ipv4.conf.default.medium_id",
            "net.ipv4.conf.default.promote_secondaries",
            "net.ipv4.conf.default.proxy_arp",
            "net.ipv4.conf.default.proxy_arp_pvlan",
            "net.ipv4.conf.default.route_localnet",
            "net.ipv4.conf.default.rp_filter",
            "net.ipv4.conf.default.secure_redirects",
            "net.ipv4.conf.default.send_redirects",
            "net.ipv4.conf.default.shared_media",
            "net.ipv4.conf.default.src_valid_mark",
            "net.ipv4.conf.default.tag",
            "net.ipv4.conf.eth0.accept_local",
            "net.ipv4.conf.eth0.accept_redirects",
            "net.ipv4.conf.eth0.accept_source_route",
            "net.ipv4.conf.eth0.arp_accept",
            "net.ipv4.conf.eth0.arp_announce",
            "net.ipv4.conf.eth0.arp_filter",
            "net.ipv4.conf.eth0.arp_ignore",
            "net.ipv4.conf.eth0.arp_notify",
            "net.ipv4.conf.eth0.bc_forwarding",
            "net.ipv4.conf.eth0.bootp_relay",
            "net.ipv4.conf.eth0.disable_policy",
```

```
            "net.ipv4.conf.eth0.disable_xfrm",
            "net.ipv4.conf.eth0.drop_gratuitous_arp",
            "net.ipv4.conf.eth0.drop_unicast_in_l2_multicast",
            "net.ipv4.conf.eth0.force_igmp_version",
            "net.ipv4.conf.eth0.forwarding",
            "net.ipv4.conf.eth0.igmpv2_unsolicited_report_interval",
            "net.ipv4.conf.eth0.igmpv3_unsolicited_report_interval",
            "net.ipv4.conf.eth0.ignore_routes_with_linkdown",
            "net.ipv4.conf.eth0.log_martians",
            "net.ipv4.conf.eth0.mc_forwarding",
            "net.ipv4.conf.eth0.medium_id",
            "net.ipv4.conf.eth0.promote_secondaries",
            "net.ipv4.conf.eth0.proxy_arp",
            "net.ipv4.conf.eth0.proxy_arp_pvlan",
            "net.ipv4.conf.eth0.route_localnet",
            "net.ipv4.conf.eth0.rp_filter",
            "net.ipv4.conf.eth0.secure_redirects",
            "net.ipv4.conf.eth0.send_redirects",
            "net.ipv4.conf.eth0.shared_media",
            "net.ipv4.conf.eth0.src_valid_mark",
            "net.ipv4.conf.eth0.tag",
            "net.ipv4.conf.lo.accept_local",
            "net.ipv4.conf.lo.accept_redirects",
            "net.ipv4.conf.lo.accept_source_route",
            "net.ipv4.conf.lo.arp_accept",
            "net.ipv4.conf.lo.arp_announce",
            "net.ipv4.conf.lo.arp_filter",
            "net.ipv4.conf.lo.arp_ignore",
            "net.ipv4.conf.lo.arp_notify",
            "net.ipv4.conf.lo.bc_forwarding",
            "net.ipv4.conf.lo.bootp_relay",
            "net.ipv4.conf.lo.disable_policy",
            "net.ipv4.conf.lo.disable_xfrm",
            "net.ipv4.conf.lo.drop_gratuitous_arp",
            "net.ipv4.conf.lo.drop_unicast_in_l2_multicast",
            "net.ipv4.conf.lo.force_igmp_version",
            "net.ipv4.conf.lo.forwarding",
            "net.ipv4.conf.lo.igmpv2_unsolicited_report_interval",
            "net.ipv4.conf.lo.igmpv3_unsolicited_report_interval",
            "net.ipv4.conf.lo.ignore_routes_with_linkdown",
            "net.ipv4.conf.lo.log_martians",
            "net.ipv4.conf.lo.mc_forwarding",
            "net.ipv4.conf.lo.medium_id",
            "net.ipv4.conf.lo.promote_secondaries",
            "net.ipv4.conf.lo.proxy_arp",
            "net.ipv4.conf.lo.proxy_arp_pvlan",
            "net.ipv4.conf.lo.route_localnet",
            "net.ipv4.conf.lo.rp_filter",
            "net.ipv4.conf.lo.secure_redirects",
            "net.ipv4.conf.lo.send_redirects",
            "net.ipv4.conf.lo.shared_media",
            "net.ipv4.conf.lo.src_valid_mark",
            "net.ipv4.conf.lo.tag",
            "net.ipv4.fwmark_reflect",
            "net.ipv4.icmp_echo_ignore_all",
            "net.ipv4.icmp_echo_ignore_broadcasts",
            "net.ipv4.icmp_errors_use_inbound_ifaddr",
            "net.ipv4.icmp_ignore_bogus_error_responses",
            "net.ipv4.icmp_msgs_burst",
            "net.ipv4.icmp_msgs_per_sec",
            "net.ipv4.icmp_ratelimit",
            "net.ipv4.icmp_ratemask",
            "net.ipv4.igmp_link_local_mcast_reports",
            "net.ipv4.igmp_max_memberships",
            "net.ipv4.igmp_max_msf",
            "net.ipv4.igmp_qrv",
            "net.ipv4.inet_peer_maxttl",
            "net.ipv4.inet_peer_minttl",
            "net.ipv4.inet_peer_threshold",
```



```
            "net.ipv4.ip_default_ttl",
            "net.ipv4.ip_dynaddr",
            "net.ipv4.ip_early_demux",
            "net.ipv4.ip_forward",
            "net.ipv4.ip_forward_update_priority",
            "net.ipv4.ip_forward_use_pmtu",
            "net.ipv4.ip_local_port_range",
            "net.ipv4.ip_local_reserved_ports",
            "net.ipv4.ip_no_pmtu_disc",
            "net.ipv4.ip_nonlocal_bind",
            "net.ipv4.ip_unprivileged_port_start",
            "net.ipv4.ipfrag_high_thresh",
            "net.ipv4.ipfrag_low_thresh",
            "net.ipv4.ipfrag_max_dist",
            "net.ipv4.ipfrag_secret_interval",
            "net.ipv4.ipfrag_time",
            "net.ipv4.neigh.default.anycast_delay",
            "net.ipv4.neigh.default.app_solicit",
            "net.ipv4.neigh.default.base_reachable_time",
            "net.ipv4.neigh.default.base_reachable_time_ms",
            "net.ipv4.neigh.default.delay_first_probe_time",
            "net.ipv4.neigh.default.gc_interval",
            "net.ipv4.neigh.default.gc_stale_time",
            "net.ipv4.neigh.default.gc_thresh1",
            "net.ipv4.neigh.default.gc_thresh2",
            "net.ipv4.neigh.default.gc_thresh3",
            "net.ipv4.neigh.default.locktime",
            "net.ipv4.neigh.default.mcast_resolicit",
            "net.ipv4.neigh.default.mcast_solicit",
            "net.ipv4.neigh.default.proxy_delay",
            "net.ipv4.neigh.default.proxy_qlen",
            "net.ipv4.neigh.default.retrans_time",
            "net.ipv4.neigh.default.retrans_time_ms",
            "net.ipv4.neigh.default.ucast_solicit",
            "net.ipv4.neigh.default.unres_qlen",
            "net.ipv4.neigh.default.unres_qlen_bytes",
            "net.ipv4.neigh.eth0.anycast_delay",
            "net.ipv4.neigh.eth0.app_solicit",
            "net.ipv4.neigh.eth0.base_reachable_time",
            "net.ipv4.neigh.eth0.base_reachable_time_ms",
            "net.ipv4.neigh.eth0.delay_first_probe_time",
            "net.ipv4.neigh.eth0.gc_stale_time",
            "net.ipv4.neigh.eth0.locktime",
            "net.ipv4.neigh.eth0.mcast_resolicit",
            "net.ipv4.neigh.eth0.mcast_solicit",
            "net.ipv4.neigh.eth0.proxy_delay",
            "net.ipv4.neigh.eth0.proxy_qlen",
            "net.ipv4.neigh.eth0.retrans_time",
            "net.ipv4.neigh.eth0.retrans_time_ms",
            "net.ipv4.neigh.eth0.ucast_solicit",
            "net.ipv4.neigh.eth0.unres_qlen",
            "net.ipv4.neigh.eth0.unres_qlen_bytes",
            "net.ipv4.neigh.lo.anycast_delay",
            "net.ipv4.neigh.lo.app_solicit",
            "net.ipv4.neigh.lo.base_reachable_time",
            "net.ipv4.neigh.lo.base_reachable_time_ms",
            "net.ipv4.neigh.lo.delay_first_probe_time",
            "net.ipv4.neigh.lo.gc_stale_time",
            "net.ipv4.neigh.lo.locktime",
            "net.ipv4.neigh.lo.mcast_resolicit",
            "net.ipv4.neigh.lo.mcast_solicit",
            "net.ipv4.neigh.lo.proxy_delay",
            "net.ipv4.neigh.lo.proxy_qlen",
            "net.ipv4.neigh.lo.retrans_time",
            "net.ipv4.neigh.lo.retrans_time_ms",
            "net.ipv4.neigh.lo.ucast_solicit",
            "net.ipv4.neigh.lo.unres_qlen",
            "net.ipv4.neigh.lo.unres_qlen_bytes",
            "net.ipv4.ping_group_range",
```

```
            "net.ipv4.route.error_burst",
            "net.ipv4.route.error_cost",
            "net.ipv4.route.gc_elasticity",
            "net.ipv4.route.gc_interval",
            "net.ipv4.route.gc_min_interval",
            "net.ipv4.route.gc_min_interval_ms",
            "net.ipv4.route.gc_thresh",
            "net.ipv4.route.gc_timeout",
            "net.ipv4.route.max_size",
            "net.ipv4.route.min_adv_mss",
            "net.ipv4.route.min_pmtu",
            "net.ipv4.route.mtu_expires",
            "net.ipv4.route.redirect_load",
            "net.ipv4.route.redirect_number",
            "net.ipv4.route.redirect_silence",
            "net.ipv4.tcp_abort_on_overflow",
            "net.ipv4.tcp_adv_win_scale",
            "net.ipv4.tcp_allowed_congestion_control",
            "net.ipv4.tcp_app_win",
            "net.ipv4.tcp_autocorking",
            "net.ipv4.tcp_available_congestion_control",
            "net.ipv4.tcp_available_ulp",
            "net.ipv4.tcp_base_mss",
            "net.ipv4.tcp_challenge_ack_limit",
            "net.ipv4.tcp_comp_sack_delay_ns",
            "net.ipv4.tcp_comp_sack_nr",
            "net.ipv4.tcp_congestion_control",
            "net.ipv4.tcp_dsack",
            "net.ipv4.tcp_early_demux",
            "net.ipv4.tcp_early_retrans",
            "net.ipv4.tcp_ecn",
            "net.ipv4.tcp_ecn_fallback",
            "net.ipv4.tcp_fack",
            "net.ipv4.tcp_fastopen",
            "net.ipv4.tcp_fastopen_blackhole_timeout_sec",
            "net.ipv4.tcp_fastopen_key",
            "net.ipv4.tcp_fin_timeout",
            "net.ipv4.tcp_frto",
            "net.ipv4.tcp_fwmark_accept",
            "net.ipv4.tcp_invalid_ratelimit",
            "net.ipv4.tcp_keepalive_intvl",
            "net.ipv4.tcp_keepalive_probes",
            "net.ipv4.tcp_keepalive_time",
            "net.ipv4.tcp_limit_output_bytes",
            "net.ipv4.tcp_low_latency",
            "net.ipv4.tcp_max_orphans",
            "net.ipv4.tcp_max_reordering",
            "net.ipv4.tcp_max_syn_backlog",
            "net.ipv4.tcp_max_tw_buckets",
            "net.ipv4.tcp_mem",
            "net.ipv4.tcp_min_rtt_wlen",
            "net.ipv4.tcp_min_snd_mss",
            "net.ipv4.tcp_min_tso_segs",
            "net.ipv4.tcp_moderate_rcvbuf",
            "net.ipv4.tcp_mtu_probing",
            "net.ipv4.tcp_no_metrics_save",
            "net.ipv4.tcp_notsent_lowat",
            "net.ipv4.tcp_orphan_retries",
            "net.ipv4.tcp_pacing_ca_ratio",
            "net.ipv4.tcp_pacing_ss_ratio",
            "net.ipv4.tcp_probe_interval",
            "net.ipv4.tcp_probe_threshold",
            "net.ipv4.tcp_recovery",
            "net.ipv4.tcp_reordering",
            "net.ipv4.tcp_retrans_collapse",
            "net.ipv4.tcp_retries1",
            "net.ipv4.tcp_retries2",
            "net.ipv4.tcp_rfc1337",
            "net.ipv4.tcp_rmem",
```



```
            "net.ipv4.tcp_sack",
            "net.ipv4.tcp_slow_start_after_idle",
            "net.ipv4.tcp_stdurg",
            "net.ipv4.tcp_syn_retries",
            "net.ipv4.tcp_synack_retries",
            "net.ipv4.tcp_syncookies",
            "net.ipv4.tcp_thin_linear_timeouts",
            "net.ipv4.tcp_timestamps",
            "net.ipv4.tcp_tso_win_divisor",
            "net.ipv4.tcp_tw_reuse",
            "net.ipv4.tcp_window_scaling",
            "net.ipv4.tcp_wmem",
            "net.ipv4.tcp_workaround_signed_windows",
            "net.ipv4.udp_early_demux",
            "net.ipv4.udp_mem",
            "net.ipv4.udp_rmem_min",
            "net.ipv4.udp_wmem_min",
            "net.ipv4.xfrm4_gc_thresh",
            "net.nf_conntrack_max",
            "net.unix.max_dgram_qlen"
```

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”两种方式来修改/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
```

3.5 使用 Prometheus 监控 CCI 实例

为实现CCI用户对部署的应用负载进行基础资源监控和自定义监控，我们提供了Prometheus对接CCI实例监控的能力，您可以手动部署Prometheus插件，直接使用Prometheus作为监控平台，对命名空间中部署的应用负载进行实时监控。

Prometheus概念及详细配置请参阅**Prometheus 官方文档**。

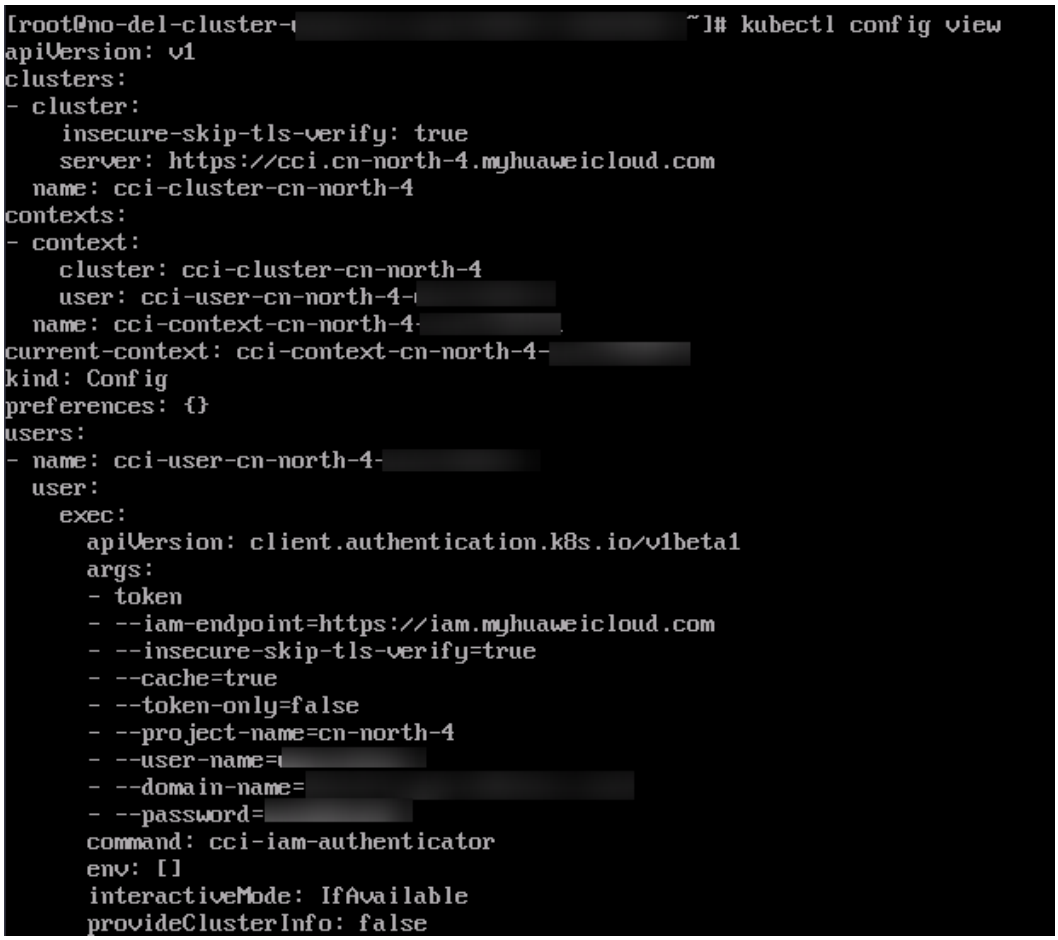
本章节将介绍如何在CCI命名空间中，部署Prometheus，实现对Pod实例的监控。

首先需要完成两步配置，一是访问CCI并使用“用户名/密码”或“AK/SK”的方式配置IAM认证信息，二是使用ConfigMap管理Prometheus配置。两个ConfigMap资源创建成功之后。接下来使用Deployment部署Prometheus，通过Volume挂载的方式，将Prometheus的配置文件挂载到容器中。

使用 ConfigMap 配置 Prometheus 访问 CCI

首先使用cci-iam-authenticator作为k8s client端的认证插件，通过用户名/密码的方式配置IAM认证信息。您可参照**安装并设置kubectl**进行配置。配置完成后，执行“kubectl config view”命令，获取$HOME/.kube/config路径下的kubeconfig文件。

图 3-10 kubeconfig 文件



以下示例是为Prometheus访问CCI所做的配置。使用此配置文件构造Prometheus连接API Server的信息。您只需将获取到的kubeconfig配置文件内容写入ConfigMap。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kubeconfig
data:
  kubeconfig: |-
    apiVersion: v1    # cci-iam-authenticator生成的kubeconfig配置文件
    ...
```

使用 ConfigMap 管理 Prometheus 配置

为了能够方便地管理配置文件，我们这里将 prometheus.yml 文件用 ConfigMap 的形式进行管理。通过ConfigMap可以方便地做到配置解耦，使得不同环境有不同的配置。相比环境变量，Pod中引用的ConfigMap可以做到实时更新，当您更新ConfigMap的数据后，Pod中引用的ConfigMap会同步刷新。

创建prometheus-config.yml文件，并写入以下内容：

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: prometheus-config
  labels:
    name: prometheus-config
```

```
data:
  prometheus.yml: |-
    global:
      scrape_interval: 15s
      evaluation_interval: 15s
    scrape_configs:
      - job_name: kuberenete-pods   # 对pod中应用的监控，自定义监控
        kubernetes_sd_configs:
        - role: pod
          kubeconfig_file: /etc/kube/kubeconfig    # 指定deployment挂载kubeconfig的路径
          namespaces:
            names:
            - test     # 要监控的命名空间列表
          relabel_configs:
          - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
            action: keep
            regex: true
          - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
            action: replace
            target_label: __metrics_path__
            regex: (.+)
          - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
            action: replace
            regex: ([^:]+)(?::\d+)?;(\d+)
            replacement: $1:$2
            target_label: __address__
          - action: labelmap
            regex: __meta_kubernetes_pod_label_(.+)
          - source_labels: [__meta_kubernetes_namespace]
          - action: replace
            target_label: kubernetes_namespace
          - source_labels: [__meta_kubernetes_pod_name]
          - action: replace
            target_label: kubernetes_pod_name
      - job_name: cci-monitor     # 监控pod指标
        kubernetes_sd_configs:
        - role: pod
          kubeconfig_file: /etc/kube/kubeconfig     #指定deployment挂载kubeconfig的路径
          namespaces:
            names:
            - test     # 要监控的命名空间列表
          relabel_configs:
          - source_labels: [__meta_kubernetes_pod_annotation_monitoring_cci_io_enable_pod_metrics]
          - action: drop
            regex: false
          - action: replace
            regex: ([^:]+)(?::\d+)?
            replacement: $1:19100
            source_labels: [__meta_kubernetes_pod_ip]
            target_label: __address__
          - action: replace
            regex: ([^:]+)(?::\d+)?;(\d+)
            replacement: $1:$2
            source_labels: [__meta_kubernetes_pod_ip,
__meta_kubernetes_pod_annotation_monitoring_cci_io_metrics_port]
            target_label: __address__
```

global模块：控制Prometheus Server的全局配置。

- scrape_interval：表示拉取targets的默认时间间隔。
- evaluation_interval：表示执行rules的时间间隔。

scrape_configs模块：用于控制prometheus监控哪些资源。

利用Pod服务自动发现机制，在Prometheus主配置文件（prometheus.yml）中，配置监控任务（job），对工作负载进行自定义监控和基础指标监控。

- **自定义监控：**对接开源Prometheus监控指标，为pod中的应用提供prometheus的监控功能，利用Pod服务自动发现机制，在Prometheus主配置文件（prometheus.yml）中，配置port和path指定要采集的端口和路径。
  - prometheus.io/scrape为true，则会将pod作为监控目标。
  - prometheus.io/path，默认为/metrics
  - prometheus.io/port , 端口
- **基础指标监控**是CCI为了让Prometheus能获取Kubernetes集群的pod基础监控数据，利用Pod服务自动发现机制，在Prometheus主配置文件（prometheus.yml）中，配置一个单独的监控任务（job），在kubernetes_sd_configs项下，指定模式为Pod，列出需要被监控的命名空间列表。

使用 Deployment 部署 Prometheus

创建prometheus的工作负载，将配置项挂载到工作负载中。使用Deployment部署Prometheus所用的镜像，相比于官方镜像额外打包了cci-iam-authenticator二进制。

示例中创建一个名为prometheus-config的Volume，Volume引用名为“prometheus-config”、“kubeconfig”、“prometheus-storage”的ConfigMap，再将Volume挂载到容器的“/tmp”路径下。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: prometheus
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
spec:
  replicas: 1
  selector:
     matchLabels:
       app.kubernetes.io/component: prometheus       # 架构中的组件
       app.kubernetes.io/instance: k8s               # 标识应用程序实例的唯一名称
       app.kubernetes.io/name: prometheus            # 应用程序的名称
       app.kubernetes.io/part-of: kube-prometheus    # 这是一个更高级别应用程序的名称
  template:
    metadata:
      labels:
        app.kubernetes.io/component: prometheus
        app.kubernetes.io/instance: k8s
        app.kubernetes.io/name: prometheus
        app.kubernetes.io/part-of: kube-prometheus
    spec:
      volumes:      # 在Volume中引用ConfigMap
        - name: prometheus-config
          configMap:
            name: prometheus-config
            defaultMode: 420        # ConfigMap卷中的所有文件默认设置为420
        - name: kubeconfig
          configMap:
            name: kubeconfig
            defaultMode: 420
        - name: prometheus-storage
          emptyDir:
            medium: LocalAuto
            sizeLimit: 10Gi
      containers:
        - name: prometheus
          image: 'swr.cn-north-7.myhuaweicloud.com/cci_k8s_gcr_io/...'
          args:         # 传给可执行文件的参数（启动参数）
            - '--storage.tsdb.retention.time=12h'              # 监控数据保留的时间
```



```
        - '--config.file=/etc/prometheus/prometheus.yml'   # 配置文件
        - '--storage.tsdb.path=/prometheus/'               # Prometheus写入数据库的地方
      ports:
        - containerPort: 9090
          protocol: TCP
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 1Gi
      volumeMounts:
        - name: prometheus-config
          mountPath: /etc/prometheus/
        - name: kubeconfig
          mountPath: /etc/kube/
        - name: prometheus-storage
          mountPath: /prometheus/
      terminationMessagePath: /dev/termination-log       # 表示容器的异常终止消息的路径
      terminationMessagePolicy: File                     # 仅从终止消息文件中检索终止消息。
      imagePullPolicy: Always
    restartPolicy: Always
    terminationGracePeriodSeconds: 30      # 优雅关闭的宽限期，即在收到停止请求后，有多少时间来进行资源释放或者做其它操作，如果到了最大时间还没有停止，会被强制结束。
    dnsPolicy: ClusterFirst
    securityContext: {}
    schedulerName: default-scheduler
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 25%
    maxSurge: 25%
revisionHistoryLimit: 10
progressDeadlineSeconds: 600
```

通过公网访问 Prometheus 平台

当工作负载创建完成后，您可以给负载绑定ELB实例，通过公网访问Prometheus平台，查看监控数据。

配置公网访问-工作负载创建完成后设置

在工作负载创建完成后对Service进行配置，此配置对工作负载状态无影响，且实时生效。具体操作如下：

步骤1 登录云容器实例管理控制台，左侧导航栏中选择“网络管理 > 服务（Service）”，在右侧页面单击“添加服务”。

步骤2 在“添加服务”页面，访问类型选择“负载均衡 LoadBalancer”。

步骤3 设置弹性负载均衡访问参数。

- 服务名称：服务名称即Service的名称，Service是用于管理Pod访问的对象。
- 命名空间：工作负载所在命名空间。
- 关联工作负载：要添加Service的工作负载。
- 负载均衡：选择公网ELB实例，如没有ELB实例可以单击“创建ELB实例”去创建。

须知

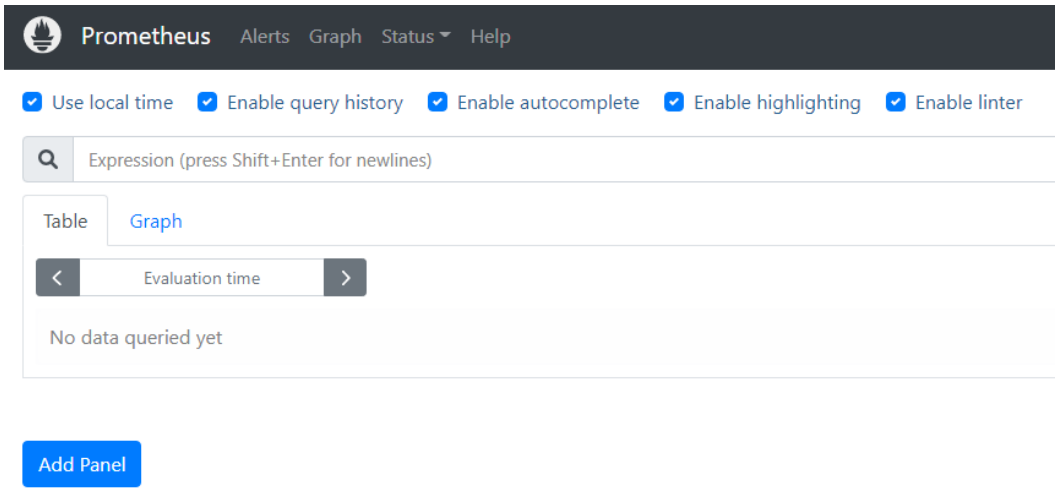
此处创建的ELB需要与负载所在命名空间在同一个VPC内。
CCI暂时不支持独享型负载均衡，建议您创建共享型ELB实例。

- 负载端口配置
  - 协议：访问负载的通信协议，可选择TCP或UDP。
  - 访问端口：负载提供的访问端口，即为服务端口，可自定义。
  - 容器端口：容器监听的端口，负载访问端口映射到容器端口，Prometheus的默认端口号为9090。

图 3-11 配置公网访问-工作负载创建完成后

步骤4 单击“提交”，工作负载已添加“负载均衡 LoadBalancer”的服务。

步骤5 创建完成后在浏览器访问“负载均衡公网IP地址:服务端口”，访问Prometheus。



----结束

4 GPU 负载

4.1 使用 Tensorflow 训练神经网络

应用场景

当前主流的大数据、AI训练和推理等应用（如Tensorflow、Caffe）均采用容器化方式运行，并需要大量GPU、高性能网络和存储等硬件加速能力，并且都是任务型计算，需要快速申请大量资源，计算任务完成后快速释放。本文将演示在云容器实例中创建GPU类型的负载，以tensorflow的图像分类为示例，演示在容器中直接使用GPU训练一个简单的神经网络。

优势

使用容器化的方式做此类人工智能训练与推理有如下优势：

- 容器化消除环境差异，不需要自己安装各种软件和配套版本，如python，tensorflow，cuda toolkit等软件。
- GPU驱动免安装。
- 低成本，按秒计费。
- serverless带来的免VM运维。

镜像制作

tensorflow社区有tensoflow的基础镜像，已经装好了基础的tensorflow库，它分支持GPU和支持CPU两个版本，在镜像中心即可下载。

- GPU版本地址为 tensorflow/tensorflow:1.15.0-gpu
- CPU版本地址为 tensorflow/tensorflow:1.13.0

本文采用tensorflow官网中一个已经训练好的模型，对图片进行分类，模型名称Inception-v3。Inception-v3是在2012年ImageNet视觉识别挑战赛上训练出的模型，它将一个非常大的图片集进行了1000个种类的图片分类。Github有使用Inception-v3进行图片分类的代码。

训练模型的代码，均在工程**https://gpu-demo.obs.cn-north-1.myhuaweicloud.com/gpu-demo.zip**中，您需要将代码下载解压，并将代码工程打入镜像中。下面附上制作镜像的Dockerfile文件内容：

```
FROM tensorflow/tensorflow:1.15.0-gpu
ADD gpu-demo /home/project/gpu-demo
```

其中ADD将gpu-demo工程拷贝到镜像的/home/project目录下，可以根据自己需要修改。

执行**docker build -t tensorflow/tensorflow:v1 .** 命令制作镜像（.表示当前目录，即Dockerfile文件所在目录）。

镜像制作好后需要上传到容器镜像服务，具体步骤请参见**https://support.huaweicloud.com/usermanual-swr/swr_01_0009.html**。

创建 Tensorflow 负载

步骤1 登录云容器实例管理控制台。

步骤2 创建GPU型命名空间，填写命名空间名称，设置好VPC和子网网段后，单击“创建”。

图 4-1 GPU 型命名空间

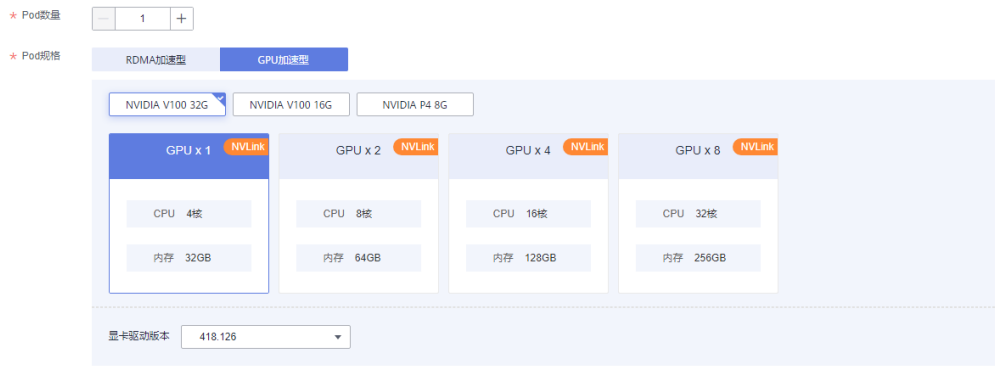
步骤3 左侧导航栏中选择“工作负载 > 无状态（Deployment）”，在右侧页面中单击“镜像创建”。

步骤4 配置负载信息。

1. 填写基本信息，选择**步骤2**创建的命名空间，Pod数量选择为“1”，选择Pod规格为“GPU加速型”，显卡的驱动版本选择“418.126”，如下所示。

   GPU Pod的详细规格和显卡驱动的说明请参见**Pod规格**。

   图 4-2 选择 GPU 容器规格



2. 选择需要的容器镜像，这里选择的上传到镜像容器仓库的tensorflow镜像。
3. 在容器设置下面的高级设置中，挂载一个NFS类型的文件存储卷，用于保存训练后的数据。

图 4-3 挂载 NFS 存储

4. 在启动命令中输出执行命令和参数。
   - 可执行命令：/bin/bash
   - 参数1：-c
   - 参数2：python /home/project/gpu-demo/cifar10/cifar10_multi_gpu_train.py --num_gpus=1 --data_dir=/home/project/gpu-demo/cifar10/data --max_steps=10000 --train_dir=/tmp/sfs0/train_data; while true; do sleep 10; done

     此处 **--train_dir** 表示训练结果存储路径，其前缀 **/tmp/sfs0** 需要与**步骤4.3**中设置的NFS“容器内挂载路径”路径保持一致，否则训练结果无法写入NFS中。

     --max_steps表示训练迭代的次数，这里指定了10000次迭代，完成模型训练大概耗时3分钟，如果不指定，默认是1000000次迭代，耗时会比较长。max_steps数值越大，训练时间越久，结果越精确。

   该命令是训练图片分类模型，然后单击“下一步”。

图 4-4 设置容器启动命令

5. 配置负载访问信息。

   本例中选择“不启用”，单击“下一步”。

6. 单击“提交”，然后单击“返回工作负载列表”。

   在负载列表中，待负载状态为“运行中”时，负载创建成功。

----结束

使用已有模型分类图片

步骤1 单击负载名称，进入负载详情界面，选择“Pod列表>终端”Tab页。当黑色区域文本框中出现#号时，说明已登录。

图 4-5 Pod 终端访问

步骤2 进入到工程所在目录，执行python classify_image.py --model_dir=model命令，可以看到分类结果。

```
# cd /home/project/gpu-demo
# ls -l
total 96
-rw-r--r-- 1 root root  6874 Aug 30 08:09 airplane.jpg
drwxr-xr-x 3 root root  4096 Sep  4 07:54 cifar10
drwxr-xr-x 3 root root  4096 Aug 30 08:09 cifar10_estimator
-rw-r--r-- 1 root root 30836 Aug 30 08:09 dog.jpg
-rw-r--r-- 1 root root 43675 Aug 30 08:09 flower.jpg
drwxr-xr-x 4 root root  4096 Sep  4 02:14 inception
# cd inception
# python classify_image.py --model_dir=model --image_file=/home/project/gpu-demo/
airplane.jpg
...
2019-01-02 08:05:24.891201: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1084] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15131 MB memory) -> physical GPU
(device: 0, name: Tesla P100-PCIE-16GiB, pci bus id: 0000:00:0a.0, compute capability: 6.0)
airliner (score = 0.84250)
wing (score = 0.03228)
space shuttle (score = 0.02524)
warplane, military plane (score = 0.00691)
airship, dirigible (score = 0.00664)
```

这里通过**--image_file**指定了要分类的图片，图片如下。执行结果最后几行是分类的label和对应的打分，其中有一行显示airliner(score = 0.84250)，分数越高越准确，可见模型认为这个图片是一架客机。

图 4-6 airliner

也可以不指定要分类的图片，默认将使用下面这张图片分类。

图 4-7 熊猫

执行命令**python classify_image.py ––model_dir=mode**

```
# python classify_image.py --model_dir=model
...
2019-01-02 08:02:33.271527: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1084] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15131 MB memory) -> physical GPU
(device: 0, name: Tesla P100-PCIE-16GiB, pci bus id: 0000:00:0a.0, compute capability:
6.0)
giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (score =
0.89107)
indri, indris, Indri indri, Indri brevicaudatus (score = 0.00779)
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score =
0.00296)
custard apple (score = 0.00147)
earthstar (score = 0.00117)
```

执行结果显示模型认为这是一只大熊猫。

----结束

使用训练的图片分类模型

tensorflow官网中给了一个深度卷积网络的模型代码和训练数据：CIFAR-10。这是个简化的图片分类模型，将图片分成以下10类：airplane, automobile, bird, cat, deer, dog, frog, horse, ship和truck。当然给模型的图，也就是训练数据，也是这10种类型的图片。

步骤1 单击负载名称，进入负载详情界面，选择“Pod列表>终端”Tab页，使用代码中提供的cifar10_eval.py校验模型的准确性，这里的checkpoint_dir指定使用刚刚训练出来的模型进行准确性校验。

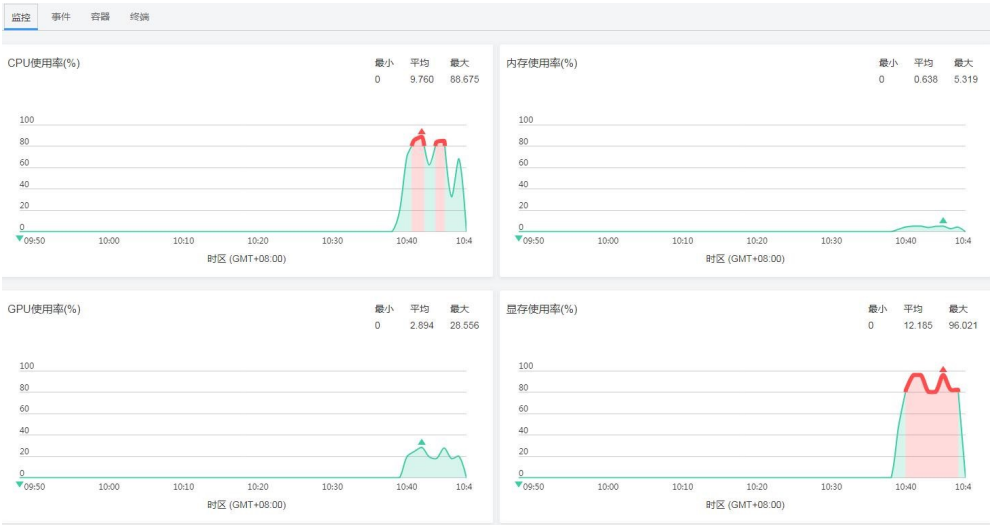
```
# cd /home/project/gpu-demo/cifar10
# python cifar10_eval.py --data_dir=data --checkpoint_dir=/tmp/sfs0/train_data --run_once
...
2019-01-02 08:25:43.914186: precision @1 = 0.817
```

步骤2 继续使用上面的飞机图片进行测试，这里的checkpoint_dir指定使用刚刚训练出来的模型进行图片分类，test_file指定用哪张图片测试。

```
# python label_image.py --checkpoint_dir=/tmp/sfs0/train_data --test_file=/home/project/gpu-demo/
airplane.jpg
    ...
2019-01-02 08:36:42.149700: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1084] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15131 MB memory) -> physical GPU
(device: 0, name: Tesla P100-PCIE-16GiB, pci bus id: 0000:00:0a.0, compute capability:
6.0)
airplane (score = 4.28143)
ship (score = 1.92319)
cat (score = 0.03095)
```

可见它准确识别出图中是架飞机。label_image.py是使用刚刚训练的模型来进行图片分类的代码。

同时，在“Pod列表>监控”Tab页中，可以看到各种资源的使用率。



----结束

5 存储管理

5.1 通过创建子用户方式，缩小 OBS 场景下上传 ak/sk 的权限

使用场景

CCI使用OBS时需要上传密钥，用户可通过创建子用户方式，将上传密钥替换为仅有OBS访问权限的子用户密钥。

操作步骤

步骤1 进入云容器实例控制台，上传密钥并导入已有的对象存储卷。

1. 登录云容器实例控制台，单击左侧导航栏的“存储管理 > 对象存储卷”，在右侧页面中选择命名空间，单击“上传密钥”。
2. 选择本地的密钥文件，单击“确认”。

   请添加csv格式的文件，且文件大小不能超过2M。若您在本地没有访问密钥，请前往“我的凭证”的**访问密钥**新增并下载访问密钥。

   说明

   需要下载主账号的访问密钥。
3. 在“对象存储卷”页面，单击“导入”。
4. 从列表里选择要导入的对象存储，单击“导入”。

   若无可用的对象存储，请单击“创建并行文件系统”按钮，填写相关参数，然后单击“立即创建”。

   创建完成后，进入“导入对象存储”页面，选择新创建的对象存储，然后单击“导入”。

步骤2 创建obs_only用户组，并授权OBS Adminstrator权限。

1. 登录“统一身份认证控制台”。
2. 在统一身份认证服务，左侧导航窗格中，选择“用户组”页签，单击右上方的“创建用户组”。

图 5-1 创建用户组

3. 在“创建用户组”界面，输入用户组名称“obs_only”。
4. 单击“确定”，用户组创建完成，用户组列表中显示新创建的用户组。
5. 在用户组列表中，单击新建用户组“obs_only”右侧的“授权”。

图 5-2 进入用户组权限设置页面

6. 在用户组选择策略页面中，勾选需要授予用户组的“OBS Adminstrator”权限。单击“下一步”。

图 5-3 选择权限

7. 选择权限的作用范围。单击“确定”，完成用户组授权。

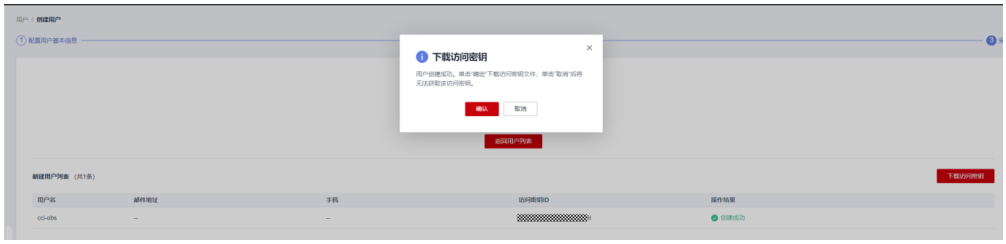
步骤3 创建子用户“cci-obs”，将子用户加入obs_only用户组并下载访问密钥。

1. 在统一身份认证服务，左侧导航窗格中，选择“用户”，单击右上方的“创建用户”。
2. 在“创建用户”页面，配置“用户信息”。填写用户名为“cci-obs”。
3. 选择“访问方式”为“编程访问”，选择“凭证类型”为“访问密钥”。

图 5-4 创建用户

4. 单击“下一步”，勾选要加入的用户组，将用户加入到“用户组obs_only”。加入用户组后，用户将具备用户组的权限。
5. 单击“创建用户”，IAM用户创建完成，并在此页面下载访问密钥。

图 5-5 下载访问密钥



步骤4 通过kubectl连接CCI。

使用kubectl连接CCI，请参见**kubectl配置指南**。

步骤5 替换上传的密钥。

将上传的密钥替换为仅有OBS访问权限的子用户密钥，即**步骤3**中下载的访问密钥，访问密钥中的ak/sk需进行手动base64编码。

使用如下命令，将secret中data的ak/sk字段替换为base64编码后的子用户ak/sk。

```
kubectl edit secret -n $上传密钥的命名空间 longaksk -oyaml
```

步骤6 验证

创建Job类型任务负载，挂载OBS，请参见**使用对象存储卷**。

负载启动正常，OBS存储卷读写正常，则成功缩小OBS场景下上传ak/sk的权限。

----结束