云容器实例(CCI) 2.0

最佳实践

文档版本 01

发布日期 2025-08-12

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。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 负载创建

1.1 使用控制台创建 Wordpress

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），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、弹性扩缩容等重要能力。

创建命名空间

步骤1 登录**云容器实例 CCI2.0**控制台。

步骤2 左侧导航栏中选择“命名空间”。

步骤3 进入命名空间页面，单击右侧“创建命名空间”。

步骤4 填写命名空间名称。

说明

- 命名空间名称在云容器实例中需全局唯一。
- 请输入1到63个字符的字符串，可以包含小写英文字母、数字和中划线（-），并以小写英文字母或数字开头，小写英文字母或数字结尾。

步骤5 监控配置(可选)。

参数	说明
对接AOM(可选)	开启后可选择对接AOM实例。

步骤6 网络平面配置。

表 1-1 网络平面配置

参数	说明
启用IPv6	开启后支持ipv4/ipv6双栈协议网络。

参数	说明
虚拟私有云	选择云容器实例所在的虚拟私有云VPC，如果没有可选项可以单击右侧“新建虚拟私有云”创建。创建后不可修改。 建议使用网段：10.0.0.0/8~22，172.16.0.0/12~22，192.168.0.0/16~22。 **须知** • 此处VPC和子网的网段不能为10.247.0.0/16，10.247.0.0/16是云容器实例预留给负载访问的网段。如果您使用此网段，后续可能会造成IP冲突，导致负载无法创建或服务不可用；如果您不需要通过负载访问，而是直接访问Pod，则可以使用此网段。 • 命名空间创建完成后，在“命名空间 > 子网”中可查看到VPC和子网信息。
子网	选择子网，如果没有可选项可以单击右侧“新建子网”创建。创建后子网不可修改。 • 创建的namespace会在设置的子网中预热部分IP，默认个数为10个。 • 在**高级设置**中可以设置预热的个数。 • 创建namespace后，由于预热了部分IP，会影响设置的subnet和VPC的删除，需要删除namespace之后才能正常删除对应的subnet和VPC。 **说明** 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。
安全组	选择已有安全组作为该命名空间下安全组，如果没有可选项可以单击右侧“创建安全组”创建。创建后不可修改。

步骤7 高级设置（可选）。

每个命名空间下都提供了一个IP池，申请IP需要一段时间，如果需要快速创建负载，减少IP的申请时间，可通过自定义资源池大小来实现。

例如，某业务线日常的负载数为200，当达到流量高峰时，IP资源池会自动扩容，瞬间将IP资源池扩容到65535（IP资源池大小），同时会在回收间隔23h（IP资源池回收间隔）之后，进行回收超过资源池大小的部分即（65535-200=65335）个。

表 1-2 高级设置（**可选**）

参数	说明
预热 IP 资源池大小（个）	• 为每个命名空间预热一个 IP 池，用来加速容器组创建。 • 预热IP资源池的大小不能超过65535个。 • 使用通用型pod规格时，建议根据业务配置合理的预热IP资源池大小，以加快负载启动速度。 • 请合理配置预热IP数量，如配置IP数量过大，可能出现预热IP资源池大于子网可用IP数，导致子网IP被耗尽，进而影响用户使用其他服务。

参数	说明
预热 IP 资源池回收间隔（h）	IP 资源池弹性扩容出来的空闲 IP 资源，在一定时间内可进行回收。 **说明** 网卡回收机制： • 回收时间要求：根据network上配置的 yangtse.io/warm-pool-recycle-interval（单位小时）时间，控制该网卡是否能够完成回收。如 yangtse.io/warm-pool-recycle-interval 配置为 24，则回收的网卡必须已经申请了超过24小时后才会被回收。 • 回收速率：内部维持一个合理的速率，按照每次最多50个网卡的速率进行回收，避免回收过快或频繁回收导致网卡的重复创建和删除动作。

步骤8 单击“确定”。

创建完成后，可以在命名空间详情中看到VPC、子网等信息。

----结束

使用 YAML 创建无状态负载

步骤1 登录云容器实例 CCI2.0控制台，左侧导航栏中选择“负载管理”，选择无状态负载，单击“YAML创建”。

步骤2 添加基本信息，yaml示例如下：

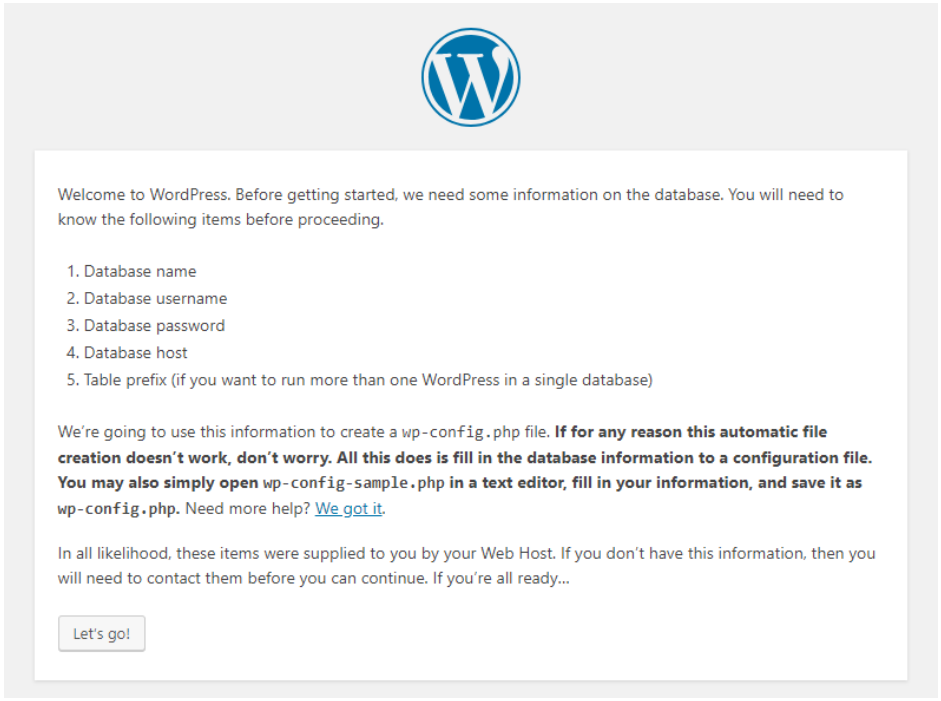
```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
        - name: wordpress
          image: wordpress:latest
          ports:
          - containerPort: 80
```

```
        resources:
          limits:
            cpu: 500m
            memory: 1Gi
          requests:
            cpu: 500m
            memory: 1Gi
      dnsPolicy: Default
```

步骤3 创建Load Balancer类型service，并且使用的ELB包含公网IP，详情见**公网访问**，yaml示例如下：

```
kind: Service
apiVersion: cci/v2
metadata:
  name: service-wordpress
  annotations:
    kubernetes.io/elb.class: elb
    kubernetes.io/elb.id: '${elb_id}'
spec:
  ports:
    - name: service-wordpress-port
      protocol: TCP
      port: 80
      targetPort: 80
  selector:
    app: wordpress
  type: LoadBalancer
```

步骤4 service创建完成后，您可通过浏览器访问地址中的公网IP。



----结束

1.2 使用 ccictl 创建 Wordpress

云容器实例 CCI 2.0支持使用ccictl工具，相比从控制台和API创建负载，使用ccictl的体验更接近Kubernetes。

约束与限制

已配置ccictl连接到CCI 2.0。

使用步骤

对于Wordpress应用，可以按照以下步骤创建一系列资源。

步骤1 使用命令“ccictl create namespace cci-test”创建命名空间，示例如下：

```
ccictl create namespace cci-test
```

图 1-1 创建命名空间回显示例

```
                          :~/yamls$ ccictl create namespace cci-test
namespace.cci/a8:                                  24/cci-test created
```

步骤2 使用命令“ccictl create -f network.yaml”创建network，yaml示例如下：

```
apiVersion: yangtse/v2
kind: Network
metadata:
  annotations:
    yangtse.io/domain-id: ${domain-id}
    yangtse.io/project-id: ${project-id}
  name: cci-network
  namespace: cci-test
spec:
  networkType: underlay_neutron
  securityGroups:
    - ${security-group-id}
  subnets:
    - subnetID: ${subnet-id}
```

图 1-2 创建 network 回显示例

```
                          :~/yamls$ ccictl create -f network.yaml
network.yangtse/cci-network created
```

步骤3 使用命令“ccictl apply -f deployment.yaml”创建wordpress负载，yaml示例如下：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: wordpress
  namespace: cci-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
        - name: wordpress
```



```
      image: wordpress:latest
      ports:
      - containerPort: 80
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 1Gi
  dnsPolicy: Default
```

图 1-3 创建 wordpress 负载回显示例

```
                          :~/yamls$ ccictl apply -f deployment.yaml
deployment.cci/wordpress created
```

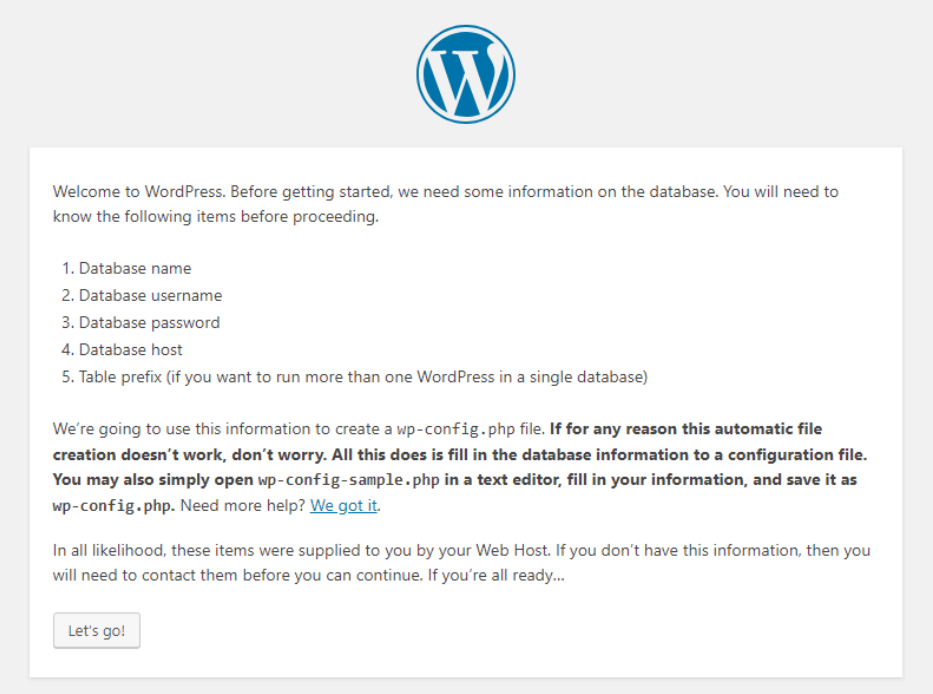
步骤4 使用命令“ccictl apply -f service.yaml”创建wordpress service，yaml示例如下：

```
kind: Service
apiVersion: cci/v2
metadata:
  name: service-wordpress
  namespace: cci-test
  annotations:
    kubernetes.io/elb.class: elb
    kubernetes.io/elb.id: '${elb_id}' # 关联的独享型ELB的id，并且ELB包含公网IP
spec:
  ports:
    - name: service-wordpress-port
      protocol: TCP
      port: 80
      targetPort: 80
  selector:
    app: wordpress
  type: LoadBalancer
```

图 1-4 创建 wordpress service 回显示例

```
                          :~/yamls$ ccictl apply -f service.yaml
service.cci/service-wordpress created
```

步骤5 使用ELB的公网IP和端口，访问wordpress前端。



----结束

1.3 Dockerfile 参数在云容器实例中如何使用

应用场景

定制镜像时，一般使用Dockerfile来完成。Dockerfile是一个文本文件，其内包含了多条的指令，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文件的一些配置在云容器实例中的应用。

Dockerfile 参数在 CCI 2.0 中的使用

下面通过一个例子来说明他们之间的关系，帮助您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yaml配置。

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: wordpress
spec:
```



```
replicas: 1
selector:
  matchLabels:
    app: wordpress
template:
  metadata:
    labels:
      app: wordpress
  spec:
    containers:
    - name: wordpress
      image: wordpress:latest
      command: ["./entrypoint.sh"] #对应ENTRYPOINT
      args: ["start"] #对应CMD
      ports: #对应EXPOSE
      - containerPort: 80
      env: #对应ENV
      - name: VERSION
        value: "1.0"
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 1Gi
      volumeMounts: #对应VOLUME
      - name: cache-volume
        mountPath: /cache
    volumes:
    - name: cache-volume
      emptyDir:
        sizeLimit: 1Gi
        medium: Memory
    dnsPolicy: Default
```

2 负载管理

2.1 CCI 2.0 应用进行优雅滚动升级

应用场景

用户在云容器实例 CCI 2.0中部署工作负载时，应用发布为LoadBalancer类型的Service且对接的独享型ELB，经过ELB的访问流量支持直通到容器中，应用进行滚动升级或者弹性扩缩容时，可以配置容器探针、最短就绪时间等实现优雅升级和优雅弹性扩缩容，避免升级或扩缩容过程中出现5xx的错误响应。

表 2-1 应用场景

场景	措施
Deployment升级	滚动升级+存活/就绪探针+优雅终止
Deployment缩容	存活/就绪探针+优雅终止

- 滚动升级

  Deployment可以采用滚动升级的升级方式，会对各个实例逐步进行更新，并非同时对所有实例进行全部更新，可以控制Pod的更新速度和并发数，从而确保了升级过程中业务不中断。例如，可以设置maxSurge和maxUnavailable参数，控制同时创建的新Pod数量和同时删除的旧Pod数量。确保升级过程中始终有工作负载能够提供服务。

  示例：配置最大浪涌（maxSurge）为25%，最大无效实例数（maxUnavailable）为25%，假设Deployment的副本数为2，那实际升级过程中，maxSurge允许最多3个Pod存在（向上取整，2*1.25=2.5，取整为3），而maxUnavailable则不允许有Pod Unavailable（向下取整，2*0.25=0.5，取整为0），也就是说在升级过程中，不允许有Pod处于非运行状态，每次新建一个Pod，待新的Pod创建成功后再删掉一个旧的Pod，直至Pod全部为新Pod。

- 设置存活/就绪探针

  存活探针可以用来确定什么时候需要重启容器，如果容器的存活检查的结果为失败，云容器实例会对该容器执行重启操作，有利于提高应用的可用性。

就绪探针可以知道容器何时准备好接口请求流量，若Pod尚未就绪，将会被从负载均衡器后端服务器组中剔除。

- 优雅终止

  负载升级或缩容过程中，负载均衡器会将正在终止的pod从后端服务器组中移除，但会保持还在处理的请求的连接。如果后端服务Pod在收到结束信号后立即退出，可能会导致正在处理的请求失败或部分流量仍被转发到已经退出的Pod中，导致流量损失和服务中断。为了避免这种情况，建议在后端服务的Pod中配置优雅删除时间（terminationGracePeriodSeconds）和preStop Hook。

  优雅删除时间（terminationGracePeriodSeconds）：默认为30秒，删除Pod时发送SIGTERM终止信号，然后等待容器中的应用程序终止执行。如果在terminationGracePeriodSeconds时间内未能终止，则发送SIGKILL的系统信号强行终止。

  如果用户的业务代码中没有处理SIGTERM信号，也可以尝试为Pod配置preStop Hook，在Pod被移除后继续工作一段时间，以解决流量中断的问题。

  建议用户在业务代码中对SIGTERM信号进行处理，实现优雅终止，同时也可以利用preStop先sleep一会，等待 kube-proxy 完成规则同步再开始停止容器内进程。用户可根据实际情况自定义优雅删除时间（preStop + 业务进程停止可能超过30s），避免过早被SIGKILL信号强行终止。

操作步骤

以部署nginx无状态工作负载为例，在云容器实例 CCI 2.0中进行优雅滚动升级或者弹性扩缩容。

步骤1 在云容器实例 CCI2.0控制台，单击左侧导航栏中的“负载管理 > 无状态负载”，单击左上角“YAML创建”。

步骤2 参考以下yaml示例创建负载：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
            memory: 1Gi
          requests:
            cpu: 500m
            memory: 1Gi
        livenessProbe:            # 存活探针
          httpGet:                # 以HTTP请求检查为例
            path: /               # HTTP检查路径为/
            port: 80              # 检查端口为80
          initailDelySeconds: 5
          periodSeconds: 5
        readinessProbe:           # 就绪探针，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。
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
        lifecycle:
          preStop:                # 停止前处理，该配置是保证业务容器在退出过程中能够对外提供服务。
            exec:
              command:
              - /bin/bash
              - '-c'
              - sleep 30
    dnsPolicy: Default
    imagePullSecrets:
      - name: imagepull-secret
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0           # 滚动升级期间最多容忍的不可用的pod数量
      maxSurge: 100%              # 滚动升级期间可冗余的pod比例
  minReadySeconds: 10             # 最小就绪时间，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。
```

说明

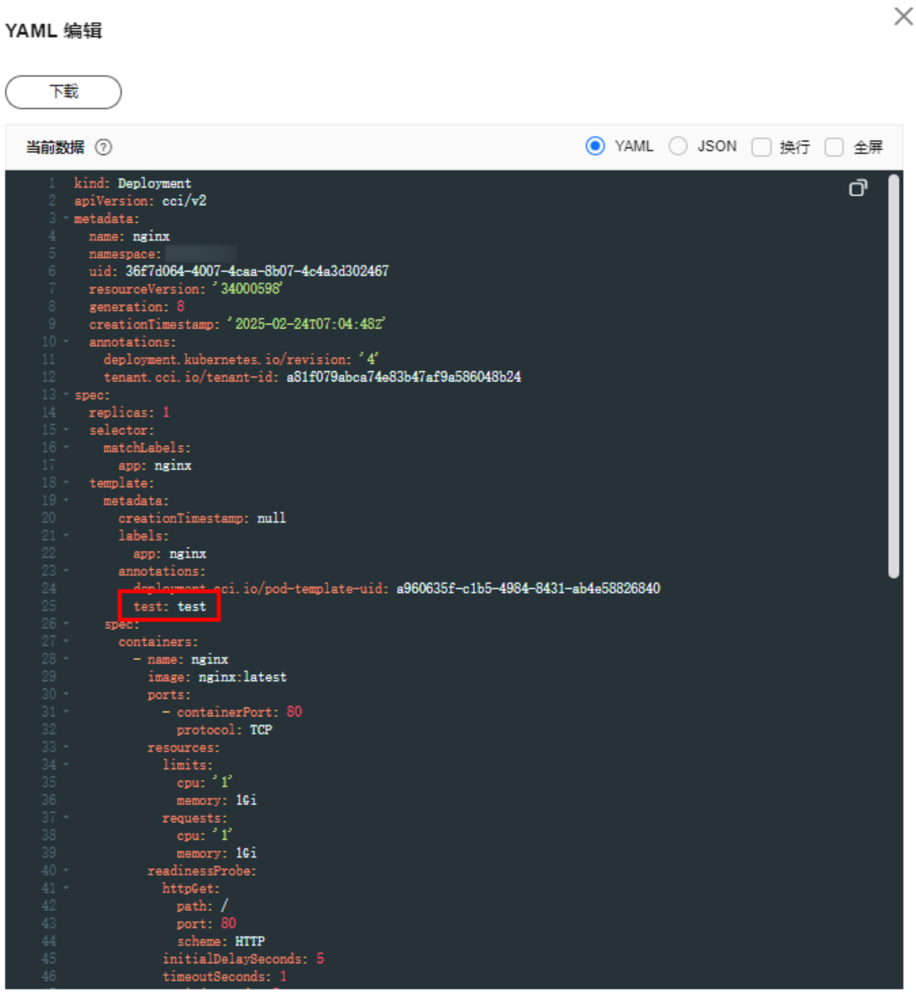
- 推荐的配置minReadySeconds时长，为业务容器的启动预期时间加上ELB服务下发member到生效的时间。
- minReadySeconds的时长需要小于sleep时长，保证旧的容器停止并退出之前，新的容器已经准备就绪。

步骤3 配置完成后，对应用进行升级和弹性扩缩容的测试。

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100.85.125.90:80为service的公网访问地址：

```
#! /bin/bash
for (( ; ; ))
do
        curl -I 100.85.125.90:80 | grep "200 OK"
        if [ $? -ne 0 ]; then
                echo "response error!"
                exit 1
        fi
done
```

步骤4 运行检测脚本：**bash detection_script.sh**，并在云容器实例 CCI2.0控制台，单击左侧导航栏中的“负载管理 > 无状态负载”，选择相关工作负载，单击“YAML编辑”，来触发应用的滚动升级。



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“200 OK”，说明升级过程是优雅升级，无中断。

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0   612    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200 OK
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0   612    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200 OK
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0   612    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200 OK
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0   612    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200 OK
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0   612    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200 OK
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0   612    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200 OK
```

----结束

2.2 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 2.0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: cci/v2
 kind: Pod
 metadata:
   annotations:
     topology.kubernetes.io/region: "{{region}}"  #所在节点的region信息
     topology.kubernetes.io/zone: "{{available-zone}}" #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：

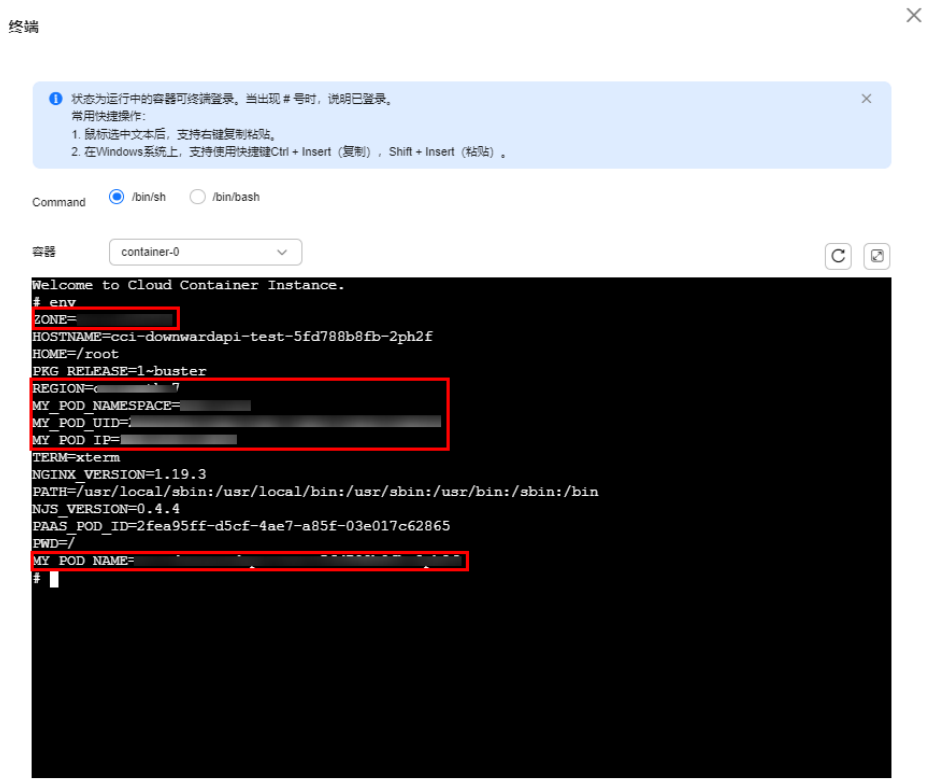
```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: cci-downwardapi-test
  namespace: cci-test  # 填写具体的命名空间
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cci-downwardapi-test
  template:
    metadata:
      labels:
        app: cci-downwardapi-test
    spec:
      containers:
        - 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
```

负载运行期间可以通过环境变量在容器中查看具体的Pod信息。

图 2-1 Pod 基础信息



2.3 内核参数配置

CCI 2.0服务底座构建了业内领先的Serverless容器平台。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 2.0服务允许用户根据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，Security Context更多信息可参考***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，通过Pod的sysctl安全上下文（Security Context）对内核参数进行配置，**sysctl**更多信息可参考***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 2.0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: cci/v2
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:/#
```

2.4 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。

由于CCI 2.0场景/dev/shm默认大小64M无法满足客户诉求，您可通过修改/dev/shm size大小，使用/dev/shm实现进程间数据交互或通过/dev/shm实现临时数据存储。

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。
- 您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 2.0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: cci/v2
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 2-2 /dev/shm 目录详情

```
root@pod-emptydir-name:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay          27G  161M   25G   1% /
tmpfs            64M     0   64M   0% /dev
tmpfs           4.0G     0  4.0G   0% /sys/fs/cgroup
tmpfs           4.0G     0  4.0G   0% /dev/shm
overlay          27G  161M   25G   1% /etc/hosts
/dev/vda4        27G  161M   25G   1% /etc/hostname
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
```

2.5 配置内存透明大页

应用场景

透明大页 Transparent Huge Pages（THP）是Linux内核中的一种内存管理技术，旨在自动将多个小页面（通常是4KB）合并为一个大页面（通常是2MB）。其核心原理是通过动态调整页面大小来优化内存管理，减少页表查找次数和TLB未命中次数，从而提高内存访问效率。然而，在内存碎片化严重或内存整理开销较大的情况下，它可能会导致内存占用增加。

透明大页优势

- 减少TLB未命中次数

  透明大页通过将多个小页面（通常是4KB）合并为一个大页面（通常是2MB），显著减少了TLB（Translation Lookaside Buffer）未命中次数。TLB未命中会导致处理器需要访问页表进行地址转换，从而增加内存访问延迟。对于内存密集型应用程序，透明大页的这种优化可以有效降低内存访问延迟，显著提升应用性能。

- 降低内核管理内存的资源消耗

  当内存页的大小增加时，相同物理内存下需要维护的页数量会大幅减少。这不仅降低了内核管理内存的资源消耗，还减少了页表查询的时间。页表查询时间与页表的层次结构和页数成正比，页数的减少直接提高了查询效率，从而提升系统整体性能。

透明大页缺点

- 内存占用上升

  在内存分配和释放频繁的环境中，透明大页可能导致内存碎片化问题，进而引发内存占用上升。当系统中存在大量无法合并为大页面的小页面时，透明大页的动态管理机制可能无法有效工作，导致内存分配失败或性能下降。碎片化问题尤其在内存密集型工作负载中更为显著。

- 可能引发OOM（Out Of Memory）

  透明大页的分配机制可能导致实际内存使用量与应用程序需求不匹配。例如，一个应用程序仅需8 KB内存（2个小页面），但内核可能分配一个完整的透明大页（2MB），当系统内存紧张时，这种额外的内存占用可能触发OOM（Out Of Memory）错误，影响系统稳定性。

透明大页配置策略建议

- 当应用程序频繁申请和释放小块内存（如4KB页面），透明大页（THP）的自动管理机制可能会频繁尝试合并和拆分页面。这种动态管理过程需要额外的计算资源，可能导致内存管理开销显著增加。此外，频繁的合并和拆分操作可能引发内存碎片化问题，进一步降低内存利用率。因此，在这种场景下，不建议开启透明大页功能，以避免性能下降。
- 如果应用程序对性能没有过高要求，但系统内存较为紧张，关闭透明大页可以有效降低内存占用。透明大页的分配机制可能导致实际内存使用量与应用程序需求不匹配，从而增加RSS（Resident Set Size）。关闭透明大页后，系统会使用传统的4KB小页面管理方式，虽然可能增加TLB未命中次数，但可以显著减少内存碎片化问题，降低内存占用，从而避免因内存不足引发的OOM（Out Of Memory）错误。

约束与限制

- CCI 2.0支持在创建负载时通过annotations配置透明大页策略，通过Deployment的YAML修改透明大页策略时，会造成POD的重建。
- 透明大页enabled策略为开启时：
  - transparent_hugepage/defrag策略默认为madvise，仅针对madvise(MADV_HUGEPAGE)区域进行直接回收。
  - transparent_hugepage/khugepaged/defrag策略默认为1，表示默认启用khugepaged内存碎片整理。

操作步骤

1. 登录**云容器实例 CCI2.0**控制台。
2. 左侧导航栏中选择“负载管理 > 无状态负载”，单击“YAML创建”。
3. 配置deployment代码，示例如下所示：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: deploy-example
  namespace: test-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deploy-example
  template:
    metadata:
      labels:
        app: deploy-example
      annotations:
        system.cci.io/transparent_hugepage.enabled: never
    spec:
      containers:
        - name: deploy-example
          image: nginx
          resources:
            limits:
              cpu: 500m
              memory: 1Gi
            requests:
              cpu: 500m
              memory: 1Gi
      dnsPolicy: Default
      imagePullSecrets:
        - name: imagepull-secret
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 100%
```

表 2-2 重要字段说明

字段名称	类型	说明
system.cci.io/transparent_hugepage.enabled	String	• always（默认值）：系统全局开启THP功能。 • madvise：仅在通过madvise(MADV_HUGEPAGE)区域开启THP功能进行系统调用。 • never：系统全局关闭THP功能。

4. 单击“确定”。

2.6 配置 Pod 使用多网卡多 EIP 功能

应用场景

当Pod需要创建多张网卡管理网络流量时，可使用CCI服务多网卡多EIP功能。多网卡多EIP功能允许客户在同一命名空间下创建多个network，并通过Pod中annotation字段配置各个网卡使用的network和EIP。

功能说明

- 创建多个network时，必须指定命名空间下某一个network为默认网络（spec.defaultNetwork为true），如果命名空间下仅有一个network，该network被识别为默认网络。
- 命名空间存在多个network时，默认网络不能更改成非默认网络。
- 命名空间存在多个network时，可以删除默认网络配置，但是需要删除后重新配置新的默认网络，否则网络功能异常。
- 创建多个network时，network之间的子网不能重复，并且所有的network配置的子网必须都在同一个VPC下。
- 多网卡多EIP功能禁止在Deployment上使用，仅支持在Pod上使用。
- Pod上可以通过yangtse.io/multi-eip-ids和k8s.v1.cni.cncf.io/networks两种annotation进行配置，但不可同时配置。
- 多网卡多EIP功能仅允许计费类型为通用型（轻享）（即general-computing-lite类型）的Pod使用。
- 单个Pod最多可以配置5张网卡。
- 单个Pod配置多张网卡时，Pod IP只会展示主网卡的IP（status.podIP和status.podIPs展示的是第一张网卡的IP信息）。

约束与限制

- 使用多网卡的Pod在启动速度上会略微变慢，请合理规划使用。
- 多网卡多EIP配置，仅支持IPv4协议。
- Pod配置多网卡多EIP时，网络将默认使用主网卡，需要把其他网卡的特殊配置（例如EIP访问公网、安全组网段放通等）同步到主网卡上，否则可能出现镜像拉取失败、挂卷失败等问题。
- Pod配置多网卡多EIP时，不支持配置自动创建镜像快照cci.io/image-snapshot-create-if-not-present annotation功能。
- Pod配置多网卡多EIP时，不支持配置单个EIP annotation功能（包括yangtse.io/eip-bandwidth-id、yangtse.io/eip-bandwidth-size、yangtse.io/eip-network-type、yangtse.io/eip-charge-mode、yangtse.io/eip-bandwidth-name、yangtse.io/pod-with-eip、yangtse.io/eip-id）。
- 当主network下subnet IP耗尽时，不会选择非主network。

操作步骤

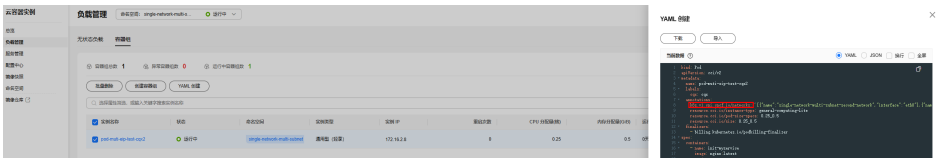
1. 访问CCI API设置默认网络，并创建多个Network。

```
apiVersion: yangtse/v2
kind: Network
```



```
metadata:
  name: test-network
  namespace: test
spec:
  defaultNetwork: true          // defaultNetwork配置为true表示该network为默认网络
  networkType: underlay_neutron
  subnets:
  - subnetID: ${subnet1}
  - subnetID: ${subnet2}
```

2. 登录**云容器实例 CCI2.0**控制台。
3. 左侧导航栏中选择“负载管理 > 容器组”，单击“YAML创建”。



4. 配置Pod代码，示例如下所示：

说明

本示例以**k8s.v1.cni.cncf.io/networks**字段配置为例，如需使用**yangtse.io/multi-eip-ids**字段，使用方法请参见**表2-3**。

```
kind: Pod
apiVersion: cci/v2
metadata:
  name: pod-muti-eip-test
  annotations:
    k8s.v1.cni.cncf.io/networks: '[{"name":"second-network","interface":"eth0"},{"name":"default-network","interface":"eth1"},{"name":"default-network","interface":"eth2"},{"name":"second-network","interface":"eth3"}]'
    resource.cci.io/instance-type: general-computing-lite
    resource.cci.io/pod-size-specs: 0.25_0.5
spec:
  containers:
    - name: init-myservice
      image: nginx:latest
      ports:
        - containerPort: 80
          protocol: TCP
      resources:
        limits:
          cpu: 250m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 512Mi
  restartPolicy: Always
  terminationGracePeriodSeconds: 30
  dnsPolicy: Default
  securityContext: {}
  imagePullSecrets:
    - name: imagepull-secret
```

表 2-3 yangtse.io/multi-eip-ids 和 k8s.v1.cni.cncf.io/networks 字段说明

字段名称	类型	格式	说明	示例
yangtse.io/multi-eip-ids	String	eip-id1,eip-id2,eip-id3,eip-id4	• EIP的ID值，以","分割，单个ID使用EIP服务未绑定EIP资源的ID且EIP的ID不重复。	yangtse.io/multi-eip-ids: eip-id1,eip-id2,eip-id3,eip-id4

字段名称	类型	格式	说明	示例

k8s.v1.cni.cncf.io/networks	String	[
		{
		"name": "second-network",
		"interface": "eth1",
		"eip": {
		"id": "eip-id",
		"bandwidth-size": "5",
		"network-type": "5_g-vm",
		"charge-mode": "bandwidth",
		"bandwidth-name": "eip-myself",
		"bandwidth-id": "bandwidth-id"
		}
		}
		]
			• name：非必填，默认值为""，不填使用默认network。network必须提前创建。
			• interface：非必填，默认值为""，不填按照数组顺序构造网卡名。如数组的第一个元素，网卡名为 eth0,Pod内网卡的名称，请务必按照网卡顺序配置为eth0、eth1。
			• eip：非必填，不填表示该网卡不需要配置EIP，Pod该网卡关联的EIP信息，一张网卡只能绑定一个EIP的ID，多张网卡不能绑定同一个EIP的ID。
			• id：非必填，填写要求为已创建且未绑定的EIP的ID，不填表示不使用已有的EIP的ID。填写该字段后，bandwidth-id、bandwidth-size、bandwidth-name、network-type、charge-mode均无效。
			• bandwidth-id：非必填，填写共享带宽ID表示使用该共享带宽自动创建EIP；不填写，表示使用独享带宽创建EIP。填写该字段后，
				k8s.v1.cni.cncf.io/networks:[{"name":"default-network","interface":"eth0","eip":{"id":"eip-id"}},{"name":"second-network","interface":"eth1","eip":{"bandwidth-size":"5","network-type":"5_g-vm","charge-mode":"bandwidth","bandwidth-name":"eip-myself"}},{"name":"third-network","interface":"eth2","eip":{"bandwidth-id":"bandwidth-id"}},{"name":"network-no-eip","interface":"eth3"}]

字段名称	类型	格式	说明	示例
			bandwidth-size、bandwidth-name、charge-mode均无效。 ● bandwidth-size：非必填，默认值为5，独享带宽大小，单位为Mbit/s。填写后将按照带宽大小创建独享带宽的EIP，和network-type、charge-mode、bandwidth-name一起使用。具体配置大小以各区域配置为准，详情请参见弹性公网IP控制台的购买页面。 ● network-type：非必填，默认值为5_bgp，公网IP类型。填写后将按照公网IP类型创建EIP，和bandwidth-size、charge-mode、bandwidth-name一起使用。具体类型以各区域配置为准，详情请参见弹性公网IP控制台的购买页面。例如，“华东-上海一”区域支持以下类型： – 5_bgp：全动态BGP – 5_sbgp：静态BGP ● charge-mode：非必填，计费类	

字段名称	类型	格式	说明	示例
			型。填写后将按照计费类型创建EIP，和bandwidth-size、network-type、bandwidth-name一起使用。默认值以各区域配置为准，配置参考： – bandwidth：按带宽计费 – traffic：按流量计费 ● bandwidth-name：非必填，默认值随机生成，独享带宽名称。填写后将按照名称创建独享带宽，和bandwidth-size、network-type、charge-mode一起使用。配置参考： – 1-64个字符，开头结尾仅支持使用数字、字母，支持数字、字母、中文、_(下划线)、-（中划线）、.（点）。 – 最小长度：1 – 最大长度：64	

5. 单击“确定”。

3 网络管理

3.1 使用 PoolBinding 发布应用

应用场景

您在云容器实例CCI服务部署工作负载后，可以通过弹性负载均衡（ELB）服务实现使用公网访问负载实例。CCI提供了PoolBinding对象帮助您自动将部署的容器实例更新至对应的后端服务器组（Pool）后端，您可以在弹性负载均衡（ELB）侧自定义负载均衡器和监听器配置，以及配置高级转发策略等，实现七层负载均衡的能力。

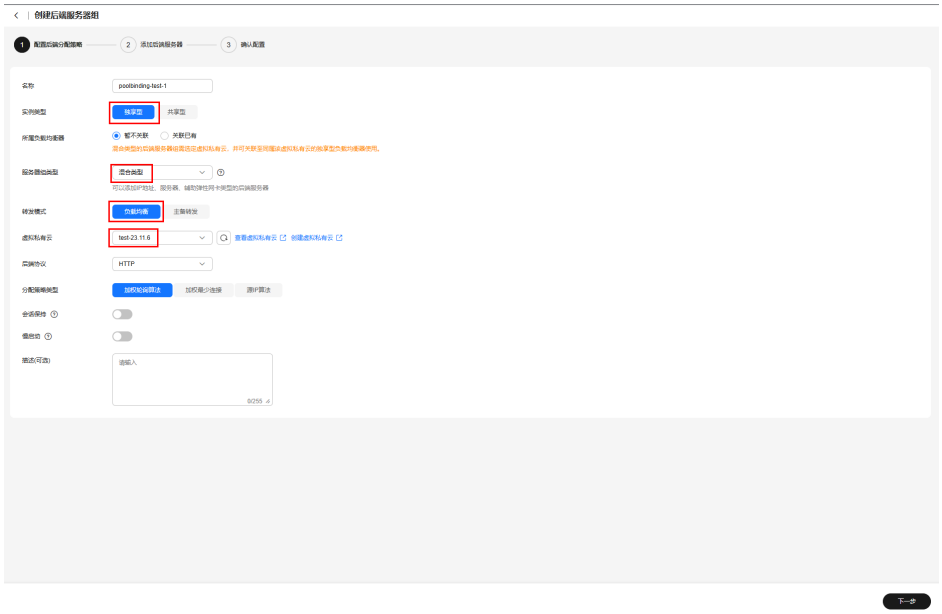
示例说明

本文以Nginx镜像为例创建无状态工作负载，部署两个版本的服务。创建后端服务器组和PoolBinding对象，将部署的实例自动更新至后端服务器组后端。创建弹性负载均衡器和监听器，并配置高级转发策略，将流量导向不同的后端应用。

操作步骤

步骤1 登录华为云弹性负载均衡服务控制台，左侧菜单栏选择“后端服务器组”，单击“创建后端服务器组”。由于示例将部署两个工作负载，请参考以下步骤创建两个后端服务器组。

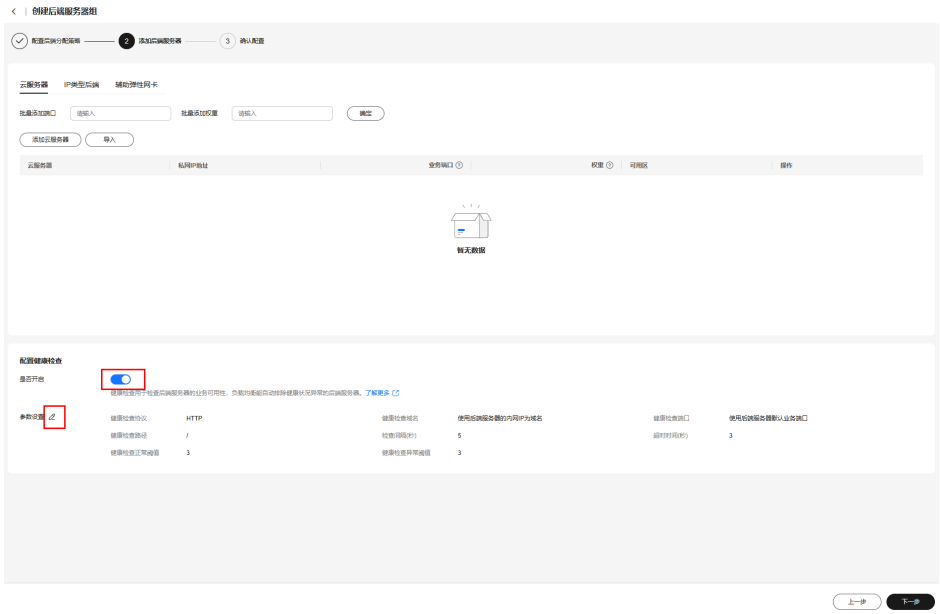
1. 配置后端分配策略。
   - 实例类型：独享型
   - 服务器组类型：混合类型
   - 转发模式：负载均衡
   - 虚拟私有云：需与后续创建的Deployment和Service所在Namespace下的虚拟私有云保持一致，因此请合理选择虚拟私有云
   - 其它参数：可自定义配置



2. 添加后端服务器。

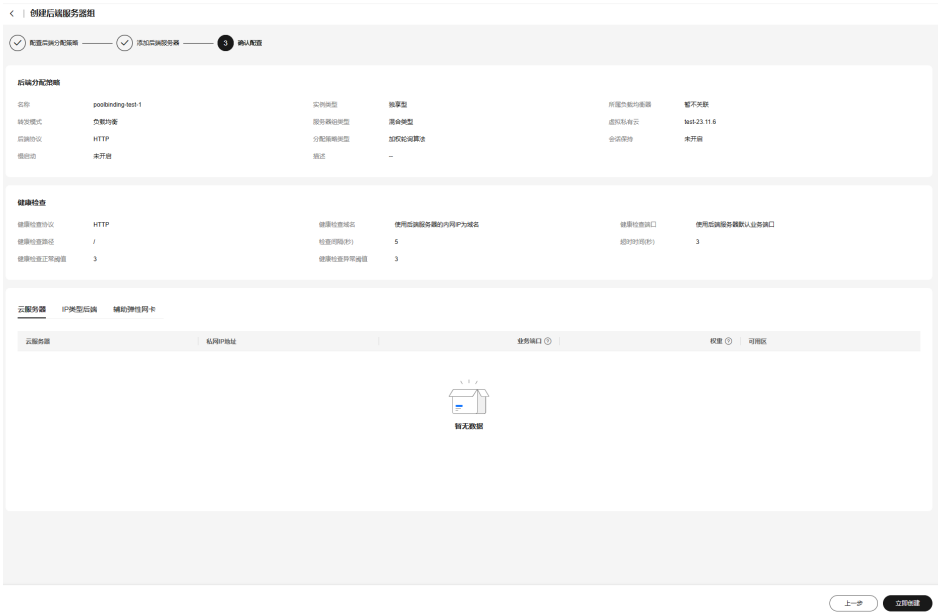
   通过PoolBinding关联的对象无需用户手动添加，用户只需配置对应的健康检查信息，选择是否开启健康检查，若开启，编辑“参数设置”，修改健康检查参数，并确认配置的参数信息可成功访问后端业务。

   示例以Nginx镜像为例，开启健康检查并使用默认配置。



3. 确认配置

   确认配置无误后，单击“立即创建”。



步骤2 在华为云云容器实例CCI2.0控制台，选择“负载管理”，选择对应的命名空间，单击“YAML创建”，创建无状态负载。

参考如下yaml示例，分别创建两个工作负载。

Deployment的YAML示例：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: deploy-1
  namespace: test-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-test-1
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx-test-1
    spec:
      containers:
        - name: nginx
          image: 'nginx:latest'
          ports:
            - containerPort: 80 //容器开放端口
              protocol: TCP
          resources:
            limits:
              cpu: 500m
              memory: 1Gi
            requests:
              cpu: 500m
              memory: 1Gi
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: Default
      securityContext: {}
      schedulerName: default-scheduler
  strategy:
```



```
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  progressDeadlineSeconds: 600
```

- 如果您希望创建的PoolBinding关联的后端对象为Service，请执行**步骤3**。
- 如果您希望创建的PoolBinding关联的后端对象为Deployment，请执行**步骤4**。

步骤3 返回华为云云容器实例CCI2.0控制台，选择“服务管理”，选择对应的命名空间，单击“YAML创建”，创建Service。参考如下yaml示例，分别创建两个Service。

Service YAML示例：

```
apiVersion: cci/v2
kind: Service
metadata:
  name: svc-1
  namespace: test-ns
  annotations:
    kubernetes.io/elb.class: elb
    kubernetes.io/elb.id: 123*******              //关联的elb id，只支持独享型elb，不支持共享型elb。
    kubernetes.io/elb.protocol-port: "http:2222"    // HTTP协议及端口号，需要与spec.ports中的端口号对应
spec:
  selector:
    app: nginx-test-1   //关联的负载label。
  ports:
    - name: service-0
      targetPort: 80    //容器开放的端口。
      port: 2222        //访问端口。
      protocol: TCP     //访问负载的协议。
      type: LoadBalancer
```

步骤4 创建PoolBinding对象。

目前console暂不支持PoolBinding对象创建，您可以使用curl命令或者ccictl工具创建。

以ccictl工具为示例创建PoolBinding对象。ccictl配置步骤请参考**ccictl配置指南**。

参考以下ccictl命令和示例，创建2个PoolBinding，分别与**步骤1**创建的两个后端服务器组，以及**步骤2**创建的两个Deployment（或**步骤3**创建的两个Service）关联。

```
ccictl create -f poolbinding.yaml
```

- 若PoolBinding关联的后端对象是Deployment，则YAML示例如下：

```
apiVersion: loadbalancer.networking.openvessel.io/v1
kind: PoolBinding
metadata:
  name: plb-test-1    //创建的PoolBinding对象名称。
  namespace: test-ns  //创建的PoolBinding对象所属的命名空间，需与关联的Deployment在同一命名空间下。
spec:
  poolRef:
    id: e08*****-****-****-****-********29c1  //关联的后端服务器组ID，即步骤1创建的后端服务器组ID。
  targetRef:
    kind: Deployment  //关联的后端对象类型，该示例为Deployment。
    group: cci/v2     //关联的后端对象group。
    name: deploy-1    //关联的后端对象名称，即步骤2创建的Deployment名称。
    port: 80          //关联的后端对象目标端口号，若后端对象为Deployment，即开放的容器端口。
```

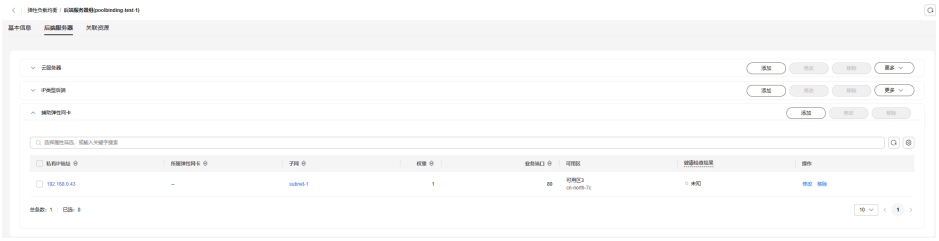
- 若PoolBinding关联的后端对象是Service，则YAML示例如下：

```
apiVersion: loadbalancer.networking.openvessel.io/v1
kind: PoolBinding
metadata:
  name: plb-test-c1   //创建的PoolBinding对象名称。
  namespace: test-ns  //创建的PoolBinding对象所属的命名空间，需与关联的Deployment在同一命名空间下。
spec:
  poolRef:
```



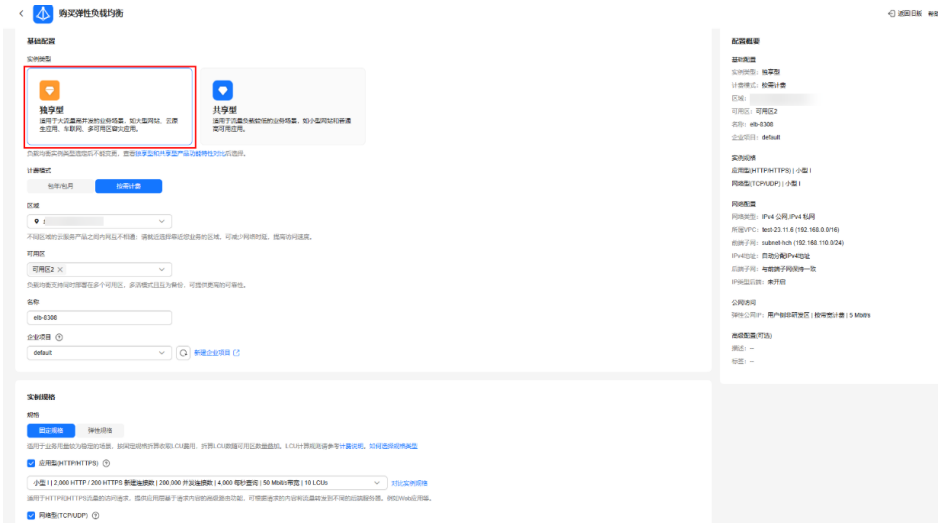
```
  id: e08*****-****-****-****-********29c1  //关联的后端服务器组ID，即步骤1创建的后端服务器组ID。
targetRef:
  kind: Service    //关联的后端对象类型，该示例为Service。
  group: cci/v2    //关联的后端对象group。
  name: svc-1      //关联的后端对象名称，即步骤3创建的Service名称。
  port: 2222       //关联的后端对象目标端口号，若后端对象为Service，即Service中配置的访问端口。
```

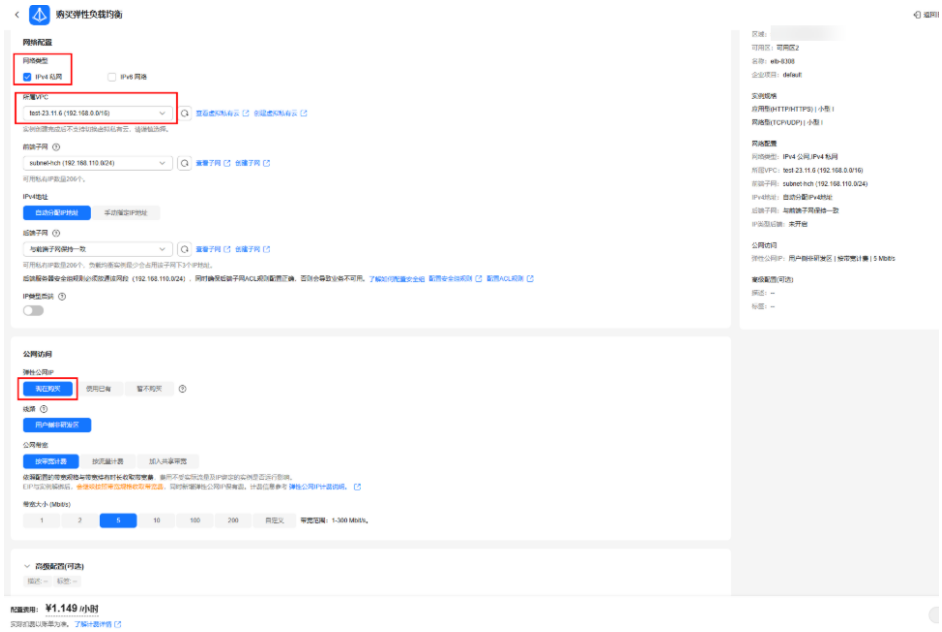
步骤5 在华为云弹性负载均衡服务控制台，单击查看**步骤1**创建的后端服务器组详情，选择“后端服务器”，检查对应的负载实例是否完成刷新。



步骤6 返回弹性负载均衡服务控制台，单击“购买弹性负载均衡”。

- 基础配置中，实例类型选择“独享型”。
- 实例规格中，规格选择“固定规格”，同时勾选“网络型"和“应用型”。
- 网络配置中，网络类型选择“IPv4私网”，所属VPC需与**步骤2**中工作负载所在VPC相同。
- 公网访问中，选择“现在购买”。
- 其余参数用户可自定义。

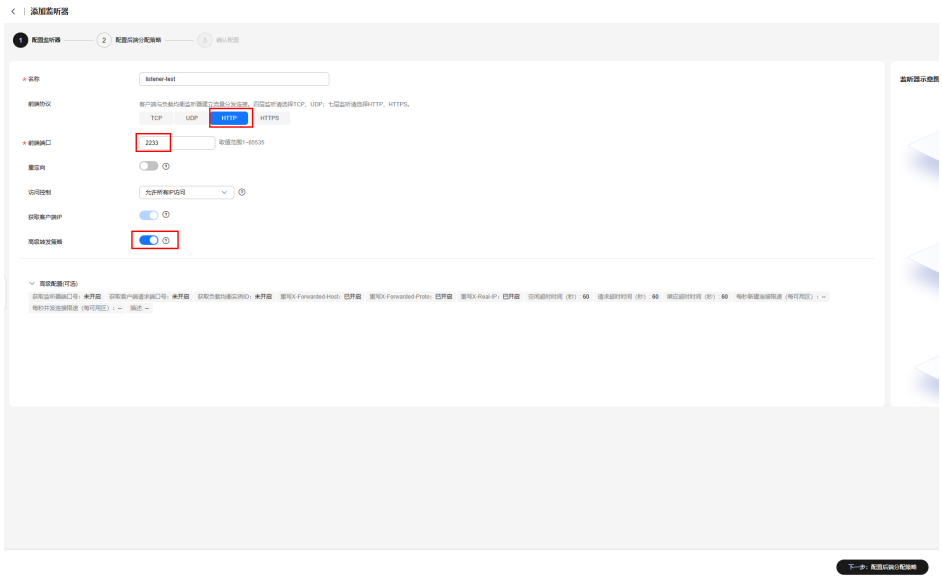




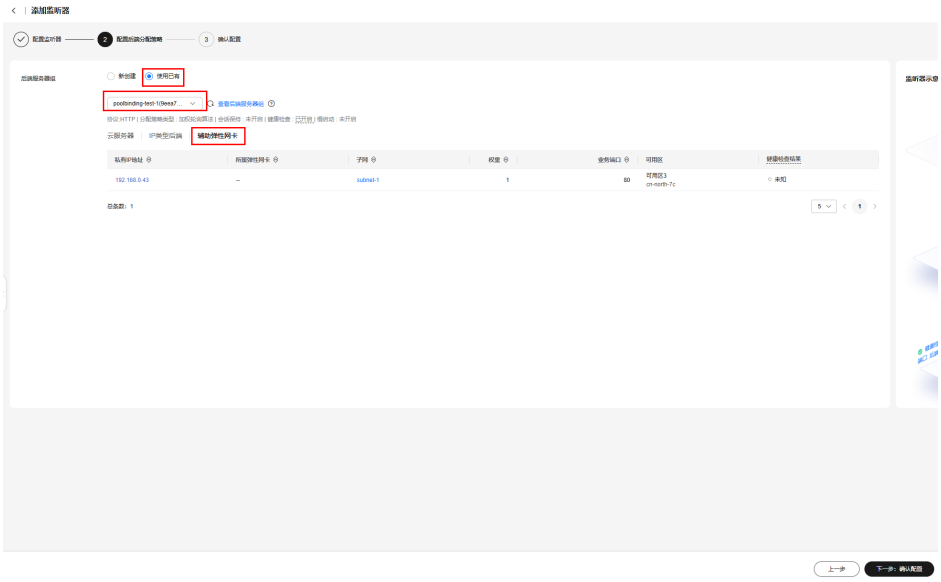
步骤7 确认配置概要信息无误后，单击“立即购买”。

步骤8 返回弹性负载均衡服务控制台，选择**步骤6**创建的弹性负载均衡，查看“监听器”，单击“添加监听器”。

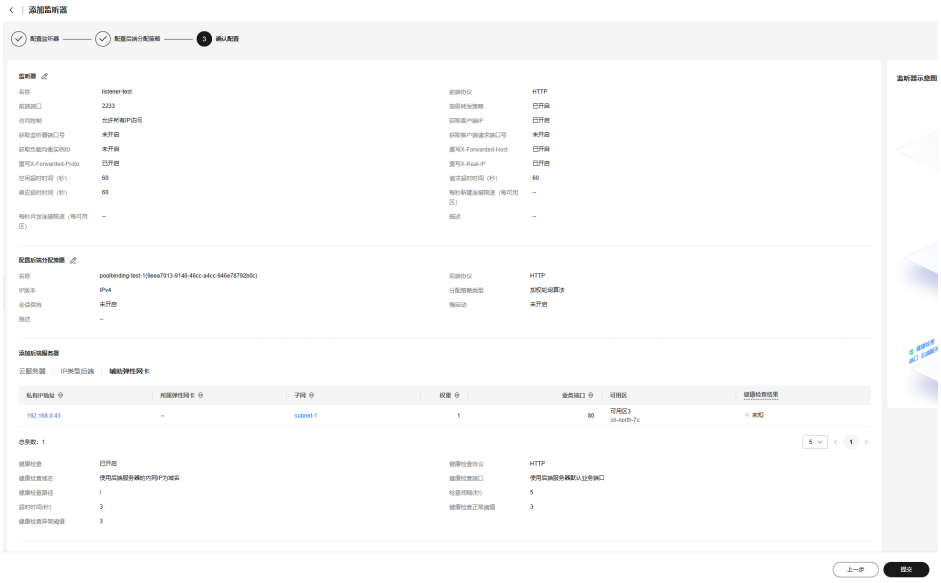
1. 配置监听器。您可根据实际情况自定义配置其它参数。完成后单击“下一步：配置后端分配策略”。
   - 名称：自定义名称
   - 前端协议：HTTP
   - 前端端口：填写前端端口
   - 高级转发策略：开启



2. 配置后端分配策略。选择后端服务器，勾选“使用已有”，搜索并选择**步骤1**创建的其中一个后端服务器组作为默认后端服务器组，单击“下一步：确认配置”。

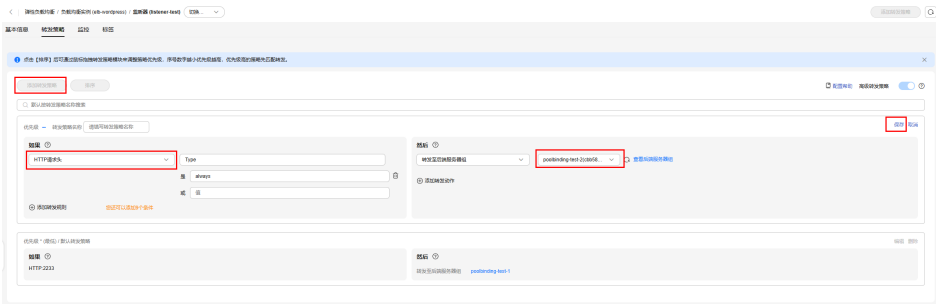


3. 确认配置。确认配置无误后，单击“提交”。



步骤9 返回弹性负载均衡服务控制台，选择**步骤6**创建的弹性负载均衡，查看“监听器”，选择**步骤8**创建的监听器，单击“添加/编辑转发策略”。

步骤10 添加转发策略。单击“添加转发策略”，编辑策略，选择转发规则，以“HTTP请求头”为例，填写HTTP请求头键值信息。选择“转发的后端服务器组”，即**步骤1**创建的另一个未绑定至监听器的后端服务器组。确认信息无误后，单击“保存”。



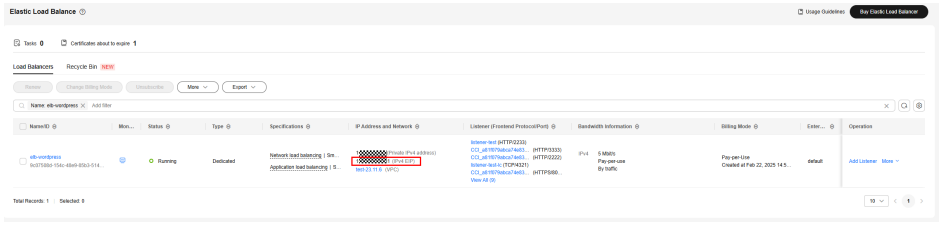
步骤11 测试验证。

1. 构造HTTP请求。

   查看弹性负载均衡IPv4公网地址，构造curl请求命令。命令格式如下：

   ```
   //未携带请求头
   while sleep 0.2;do curl http://$IP:$port;done
   //携带请求头
   while sleep 0.2;do curl http://$IP:$port -H "Type: always";done
   ```

   - $IP：弹性负载均衡IPv4公网地址。
   - $port：配置的监听器前端全口。

2. 发送请求，查看后端流量。

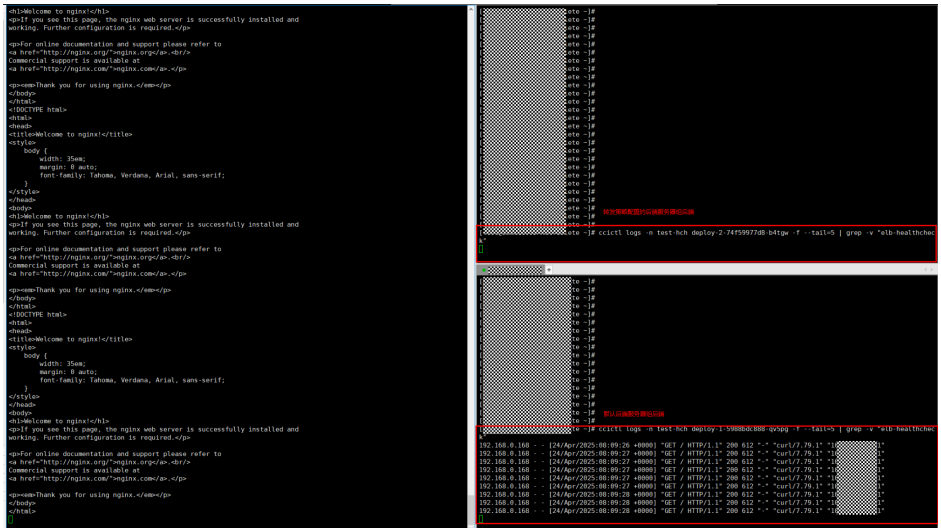
   执行HTTP请求命令，以Nginx镜像为例部署的工作负载，可以通过ccictl logs命令查看后端接收的请求情况。

   ```
   ccictl logs -n $namespace $pod-name -f --tail=5 | grep -v "elb-healthcheck"
   ```

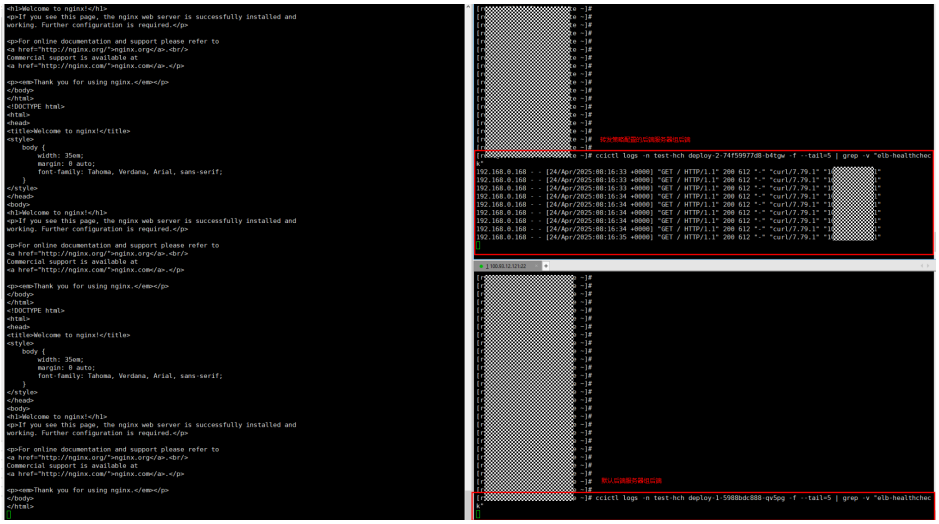
   - $namespace：创建的工作负载所在的命令空间。
   - $pod-name：创建的工作负载Pod实例名称。

3. 查看结果示例。

   - 如果HTTP请求头不包含对应的请求头“Type: always”，则流量导向默认后端服务器组后端。

- 如果HTTP请求头包含对应的请求头“Type: always”，则流量导向转发策略中配置的后端服务器组后端。



----结束

4 存储管理

4.1 如何扩展临时存储空间

本章节介绍如何使用临时存储空间扩容功能解决大镜像大数据量写入的问题。

使用场景

pod需要写入大量数据至rootfs或emptyDir以及镜像大于30GiB时，需要扩容临时存储空间。

使用须知

临时存储空间扩容大小限制为[0,994]。

创建临时存储空间扩容负载

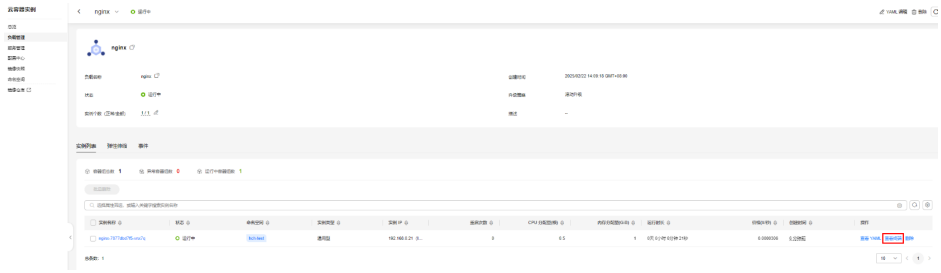
步骤1 登录云容器实例 CCI2.0控制台，左侧导航栏中选择“负载管理 > 无状态负载”，再单击“YAML创建”。

步骤2 填写yaml（以nginx为例），示例如下：

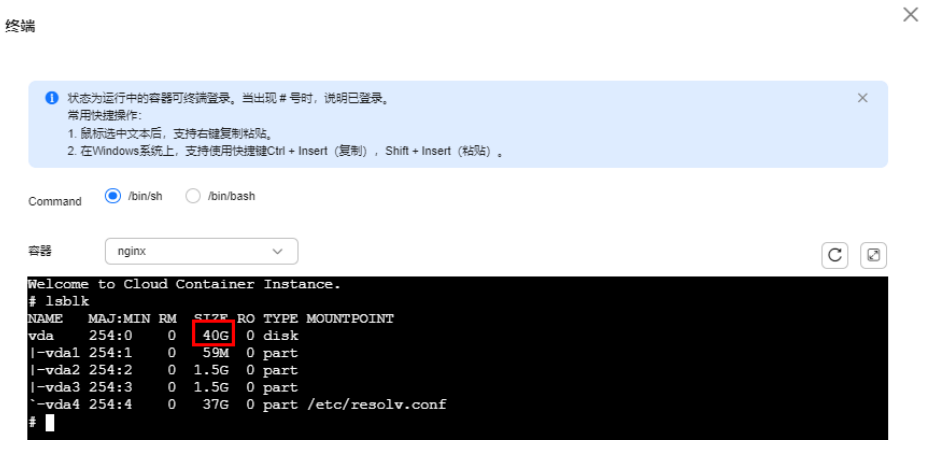
```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: nginx
spec:
```

```
replicas: 1
selector:
  matchLabels:
    app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:latest
        resources:
          limits:
            cpu: 500m
            memory: 1Gi
          requests:
            cpu: 500m
            memory: 1Gi
    dnsPolicy: Default
    extraEphemeralStorage:
      sizeInGiB: 10 #临时存储空间扩容大小，单位为GiB
```

步骤3 进入负载详情，单击“查看终端”。



步骤4 输入lsblk并回车，查看扩容后的系统盘大小（默认为30G）。



----结束

5 弹性伸缩

5.1 HPA 弹性扩缩容

简介

本章节主要指导用户在CCI 2.0配置使用HorizontalPodAutoscaler（HPA，Pod水平自动扩缩容），自动更新无状态负载资源， 目的是自动扩缩无状态负载以满足需求。

HPA会定期调整无状态负载的所需规模，以匹配期望的指标， 例如，平均CPU利用率、平均内存利用率或您指定的任何其他自定义指标。

如何配置 HPA

步骤1 登录云容器实例CCI 2.0控制台，单击左侧导航栏“负载管理”，单击“无状态负载”中的某个负载进入负载详情界面。

步骤2 单击“弹性伸缩”，并单击“YAML创建”创建HPA策略。

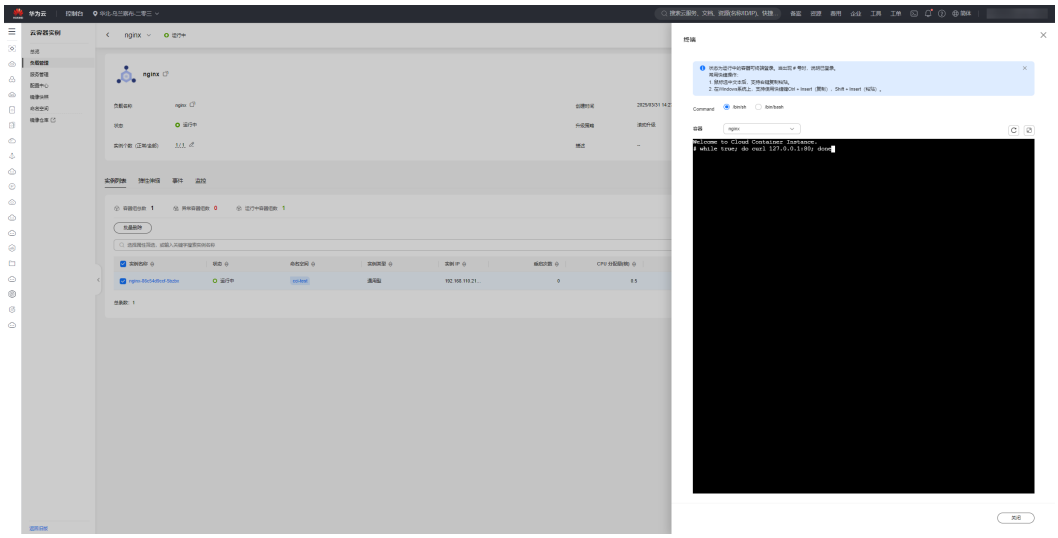
创建HPA的YAML示例如下：

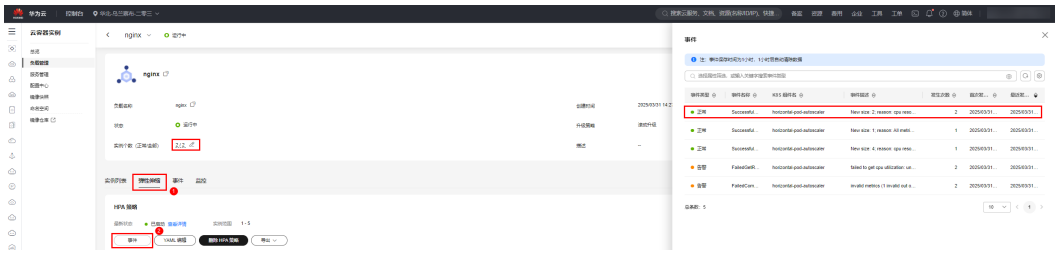
```
kind: HorizontalPodAutoscaler
apiVersion: cci/v2
metadata:
  name: alpha-test-hpa
  namespace: cci-test              # 命名空间
spec:
  scaleTargetRef:
    kind: Deployment
    name: nginx
    apiVersion: cci/v2
  minReplicas: 1                   # 最小副本数
  maxReplicas: 5                   # 最大副本数
  metrics:
    - type: Resource
      resource:
        name: memory               # 支持CPU、Memory资源指标
        target:
          type: Utilization        # 扩缩类型
          averageUtilization: 50   # 触发扩缩的平均使用率
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

步骤3 进入负载详情，选择负载内的某个Pod单击“查看终端”按钮进入Pod终端，输入命令：

```
while true; do curl 127.0.0.1:80; done
```

等待HPA触发，负载扩容并上报事件。





----结束

5.2 CCE 容器实例弹性伸缩到 CCI 2.0 服务

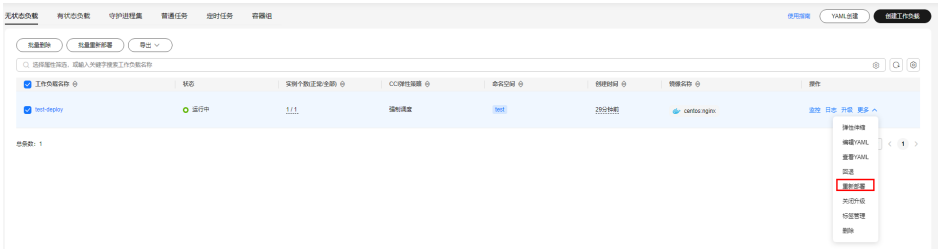
对于使用CCE集群和CCI的使用场景，用户可以按需将工作负载调度到CCE集群节点或者对接CCI的虚拟节点，本文详细介绍如何将CCE集群的工作负载调度到CCI 2.0上。

bursting插件当前提供了以下方式管理CCE集群的pod，使其能够调度到CCI 2.0：

- **方式一：通过配置labels控制pod调度到CCI**
- **方法二：通过配置profile控制pod调度到CCI**

约束与限制

- 当前只有负载的原生标签能够被ScheduleProfile匹配，使负载能够被ScheduleProfile管理，将CCE集群的Pod调度到CCI 2.0。例如通过**ExtensionProfile**加到负载上的标签不能够被ScheduleProfile匹配，因此不会被ScheduleProfile管理调度功能。
- 通过配置labels控制pod调度到CCI 2.0的方式，需要在创建工作负载前添加label，若对已创建的工作负载进行追加label操作，存量工作负载对应的pod不会更新，可选择该工作负载，单击“更多 -> 重新部署”实现更新。



前提条件

- 插件版本：已安装CCE突发弹性引擎（对接 CCI）且插件，如果使用配置profile控制pod调度到CCI的方式请安装CCE突发弹性引擎（对接 CCI）插件版本为1.3.19及以上版本。
- 集群类型：如果使用配置profile控制pod调度到CCI方式仅支持VPC网络模式的CCE Standard集群和CCE Turbo集群。

调度策略

CCE集群工作负载弹性调度到CCI 2.0策略有以下几种：

调度策略	策略图解	适用场景
强制调度策略（enforce）		CCE工作负载强制弹性到CCI 2.0。
自动调度策略（auto）		根据用户集群调度器实际打分决定是否弹性到CCI 2.0。
本地优先调度策略（localPrefer）		工作负载优先调度到CCE集群，集群资源不足时工作负载弹性到CCI 2.0。
不开启（off）		工作负载不会弹性到CCI 2.0。

方法一：通过配置 labels 控制 pod 调度到 CCI

您可以通过控制台或YAML配置labels控制pod调度到CCI。

通过控制台配置

步骤1 登录CCE控制台，单击集群名称，进入概览页面。

步骤2 选择“工作负载”，单击“创建工作负载”。

步骤3 在基本信息中“弹性至CCI”选择“不开启”以外的任意策略。

- 本地优先调度：Pod优先调度至CCE节点，当CCE节点资源不足时将Pod弹性至CCI。
- 强制调度：所有Pod均会弹性至CCI。

步骤4 勾选相关的CCI资源池。

- CCI 2.0资源池(bursting-node)：新一代Serverless容器资源池。
- CCI 1.0资源池(virtual-kubelet)：存量Serverless容器资源池，即将日落。

说明

在CCE集群控制台（console）创建工作负载时，CCI弹性承载支持勾选“CCI 2.0资源池(bursting-node)”或者“CCI 1.0资源池(virtual-kubelet)”，如果使用CCI 1.0请勾选“virtual-kubelet”，如果使用CCI 2.0请勾选“bursting-node”。目前CCI 2.0仅对白名单用户开放，如需使用CCI 2.0服务，请提交工单申请开启CCI 2.0服务。

步骤5 CCE工作负载具体操作步骤详情请参见**创建工作负载**，完成后单击“确定”即可。

----结束

通过 YAML 文件配置

步骤1 登录CCE控制台，单击集群名称，进入概览页面。

步骤2 选择“工作负载”，单击“YAML创建”。

步骤3 在工作负载的yaml文件中添加virtual-kubelet.io/burst-to-cci标签。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test
  namespace: default
  labels:
    bursting.cci.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
```

表 5-1 关键参数说明

参数	类型	描述
bursting.cci.io/burst-to-cci	String	表示CCE集群工作负载弹性调度到CCI策略，取值如下： • enforce，CCE工作负载强制弹性到CCI 2.0。 • auto，根据用户集群调度器实际打分决定是否弹性到CCI 2.0。 • localPrefer，工作负载优先调度到CCE集群，集群资源不足时工作负载弹性到CCI 2.0。 • off，工作负载不会弹性到CCI 2.0。

步骤4 完成后单击“确定”。

----结束

方法二：通过配置 profile 控制 pod 调度到 CCI

您可以通过控制台或YAML配置profile控制pod调度到CCI。

通过控制台配置

步骤1 登录CCE控制台，单击集群名称，进入概览页面。

步骤2 选择“策略 > CCI弹性承载策略”。

步骤3 单击“创建CCI弹性承载策略”，并填写相关信息。

表 5-2 创建 CCI 弹性承载策略

参数	参数说明
策略名称	输入策略名称。
命名空间	选择弹性策略生效的命名空间。您可选择命名空间或创建命名空间，创建命名空间操作步骤详情请参见**创建命名空间**。
关联负载	填写键值或引用负载标签。
调度策略	选择调度策略。 • 本地优先调度：Pod优先调度至CCE节点，当CCE节点资源不足时将Pod弹性至CCI。 • 强制调度：所有Pod均会弹性至CCI。

参数	参数说明
分配策略	• 本地：设置本地CCE集群。 • CCI：设置CCI上运行的“最大实例数”。
最大实例数	设置“本地”或“CCI”运行的最大实例数。
本地缩容优先级	取值范围[-100, 100]，数值越大越先缩容。
CCI缩容优先级	取值范围[-100, 100]，数值越大越先缩容。
CCI资源池	• CCI 2.0资源池(bursting-node)：新一代Serverless容器资源池。 • CCI 1.0资源池(virtual-kubelet)：存量Serverless容器资源池，即将日落。

步骤4 单击“确定”。

步骤5 选择“工作负载”，单击并创建工作负载，其中“高级配置 > 标签与注解”中添加Pod标签，其中键值与**步骤3**创建关联负载中的键值需保持一致，其他操作步骤请参考**创建工作负载**。

步骤6 完成后单击“创建工作负载”即可。

----结束

通过 YAML 配置

步骤1 登录CCE控制台，单击集群名称，进入概览页面。

步骤2 选择“策略 > CCI弹性承载策略”。

步骤3 单击“YAML创建”，创建profile对象。

示例一：限制调度到CCE集群最大负载数，配置local maxNum和scaleDownPriority的profile模板如下：

```
apiVersion: scheduling.cci.io/v2
kind: ScheduleProfile
metadata:
  name: test-local-profile
  namespace: default
spec:
  objectLabels:
    matchLabels:
      app: nginx
  strategy: localPrefer
  virtualNodes:
    - type: bursting-node
  location:
    local:
      maxNum: 20 # 当前暂不支持local/cci同时配置maxNum
      scaleDownPriority: 2
    cci:
      scaleDownPriority: 10
```

示例二：限制CCI集群最大负载数，配置cci maxNum和scaleDownPriority的profile的模板如下：

```
apiVersion: scheduling.cci.io/v2
kind: ScheduleProfile
metadata:
  name: test-cci-profile
  namespace: default
spec:
  objectLabels:
    matchLabels:
      app: nginx
  strategy: localPrefer
  virtualNodes:
    - type: bursting-node
  location:
    local: {}
    cci:
      maxNum: 20 # 当前暂不支持local/cci同时配置maxNum
      scaleDownPriority: 10
```

表 5-3 关键参数说明

参数	类型	描述
strategy	String	表示CCE集群工作负载弹性调度到CCI 2.0策略，取值如下： ● enforce，CCE工作负载强制弹性到CCI 2.0。 ● auto，根据用户集群调度器实际打分决定是否弹性到CCI 2.0。 ● localPrefer，工作负载优先调度到CCE集群，集群资源不足时工作负载弹性到CCI 2.0。
maxNum	int	表示pod最大数量配置。 取值范围[0~int32]。
scaleDownPriority	int	表示缩容优先级配置，数值越大所关联的pod越先缩容。 取值范围[-100, 100]。

说明

- location配置local字段（CCE侧）和cci字段（CCI侧）控制pod数量和缩容优先级。
- local字段和cci字段不可以同时设置maxNum。
- 缩容优先级为非必填项参数，如果不配置缩容优先级，默认值将为空。

步骤4 单击“确定”。

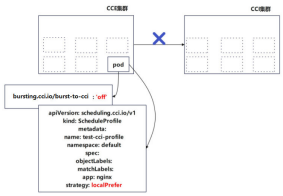
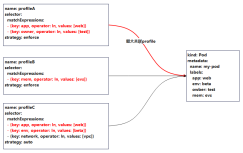
步骤5 创建无状态负载，使用selector方式选择含有“app：nginx”的pod，关联上文创建的profile。

```
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
```

表 5-4 特殊场景使用说明

特殊使用场景	使用说明
pod同时使用label和profile控制调度CCI 2.0	使用label控制调度CCI 2.0的方式优先级高于使用profile控制调度CCI 2.0的方式。 例如label中策略为off，profile的策略为enforce，最终pod不会调度到CCI 2.0。 
pod同时被多个profile关联	pod有且仅有一个关联profile。当pod被多个profile关联，根据profile匹配pod的label的数量，选出最大关联profile，存在多个时，选择字典序小的profile关联。 pod最终被profileA关联 

步骤6 单击“确定”。

----结束

6 镜像管理

6.1 使用镜像快照加速镜像拉取

场景描述

镜像快照服务提供更加高效的容器启动方式，使用镜像快照创建CCI 2.0实例可以加速拉取镜像，减少CCI实例的启动耗时，使用镜像快照可参考**使用镜像快照**。

操作步骤

步骤1 登录云容器实例 CCI2.0控制台，在左侧导航栏中选择“镜像快照”，单击右上角“YAML创建”，按以下yaml示例填写：

```
apiVersion: cci/v2
kind: ImageSnapshot
metadata:
  name: imagesnapshot-a
spec:
  buildingConfig:
    namespace: test #命名空间
  imageSnapshotSize: 20 #快照大小，单位为GiB
  ttlDaysAfterCreated: 100 #镜像有效期
  images:
    - image: nginx:stable-alpine-perl #镜像
  registries: []
```

步骤2 在左侧导航栏中选择“负载管理 -> 无状态负载”，单击左上角“YAML创建”创建负载，annotation配置cci.io/image-snapshot-auto-match: 'true'，示例如下：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        cci.io/image-snapshot-auto-match: 'true' #镜像快照自动匹配
    spec:
```



```
  containers:
    - name: nginx
      image: nginx:stable-alpine-perl
      resources:
        requests:
          cpu: 500m
          memory: 1Gi
```

步骤3 检查启动后的 pod annotations中有字段：cci.io/image-snapshot-detail、cci.io/imagesnapshot-volume，显示匹配的快照，且挂载数据盘大小和快照大小一致。

----结束

6.2 如何下载企业仓镜像

本章节介绍如何拉取企业仓库中的镜像。

使用场景

需要使用企业仓库中的镜像。

约束与限制购买

- 已有容器镜像服务企业仓库，操作步骤可参见**购买仓库**。
- 仓库中已上传镜像，操作步骤可参见**镜像仓库**。
- 使用SWR企业版镜像仓库镜像时，请确保工作负载运行的节点可访问对应的企业版镜像仓库实例。

创建 SWR 企业版镜像仓库密钥

步骤1 登录容器镜像服务（企业版），进入希望拉取的企业仓，左侧导航栏选择“访问管理->访问控制”，单击右上角“创建内网访问”并配置在CCI 2.0中命名空间使用的VPC和子网。创建成功后记录可供访问的IP地址。

步骤2 左侧导航栏选择“访问管理->域名管理”，记录默认域名（默认域名不可编辑）。

步骤3 左侧导航栏选择“访问管理->访问凭证”，单击“新建长期凭证”创建凭证并下载csv文件。

步骤4 登录云解析服务DNS控制台，单击左侧导航栏“内网域名”，再单击右上角“创建内网域名”，配置在CCI 2.0中命名空间使用的VPC和步骤**步骤2**中的默认域名。

步骤5 内网域名创建完成后，进入详情添加记录集，配置步骤**步骤1**中获得的IP地址。

步骤6 登录云容器实例CCI 2.0控制台，在左侧导航栏选择“配置中心”，选择“密钥（Secret）”，单击“YAML创建”。创建secret，secret示例如下：

```
kind: Secret
apiVersion: cci/v2
metadata:
  name: ${secret_name}                    # secret名称
data:
  .dockerconfigjson: ${access_credential}   # 访问swr企业仓的凭证
type: kubernetes.io/dockerconfigjson
```

${access_credential}，为json格式化的凭证base64编码后的结果，按如下方法获取：

将步骤**步骤3**获取的csv文件内容填入以下命令中：

```
echo -n '{"auths":{"${url}":{"username":"${user_name}","password":"${password}"}}}' | base64 | sed ':a;N;$!ba;s/\n//g'
```

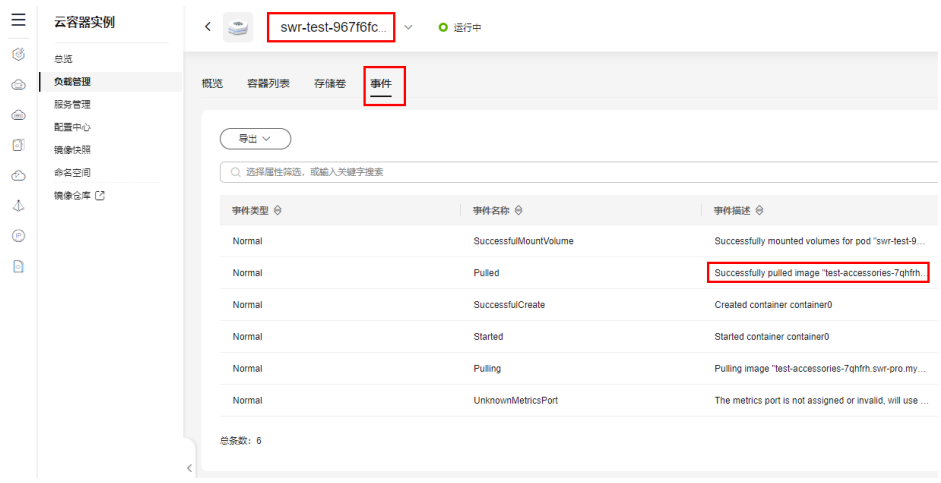
${url}为步骤**步骤2**中的默认域名，${user_name}为csv文件中的用户名，${password}为csv文件中的密码。
执行该命令，获得base64编码后的访问凭证，填入secret yaml中的${access_credential}中，最后创建secret。

步骤7 单击左侧导航栏“负载管理”，选择无状态负载，单击“YAML创建”。

步骤8 使用企业仓镜像创建无状态负载，yaml示例如下：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: swr-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: swr-test
  template:
    metadata:
      labels:
        app: swr-test
    spec:
      containers:
        - name: container0
          image: ${image:tag}  # 企业仓镜像地址，可从SWR镜像管理处查询，模板为${url}/${namespace}/${image-name}@sha256:${sha256}
          resources:
            limits:
              cpu: 500m
              memory: 1Gi
            requests:
              cpu: 500m
              memory: 1Gi
      imagePullSecrets:
        - name: ${secret-name}  # 步骤步骤6中创建的secret名称
```

步骤9 查看创建出的pod事件，观察拉取镜像是否成功。



----结束

6.3 如何下载第三方镜像

本章节介绍如何拉取第三方仓库中的镜像。

使用场景

需要使用第三方仓库中的镜像。

约束与限制

harbor需先配置EIP。

使用第三方仓库镜像创建容器组

步骤1 登录云容器实例 CCI2.0控制台，在左侧导航栏中选择“负载管理 > 无状态负载”，再单击“YAML创建”。

步骤2 按如下示例填写yaml：

```
kind: Deployment
apiVersion: cci/v2
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        cci.io/insecure-registries: 100.85.XXX.XXX #harbor EIP地址，拉取自建镜像仓库的镜像
        yangtse.io/pod-with-eip: 'true' #pod访问EIP开关
    spec:
      containers:
        - name: nginx
          image: 100.85.XXX.XXX/library/nginx:stable-alpine-perl #harbor镜像地址
          resources:
            requests:
              cpu: 500m
              memory: 1Gi
```

步骤3 负载运行成功。

----结束

7 日志监控

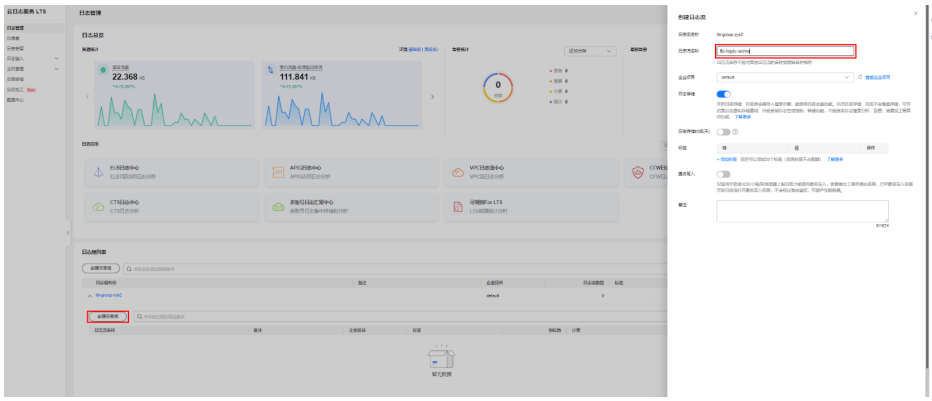
7.1 日志上报 LTS

应用场景

云日志服务（Log Tank Service，简称LTS）通过海量日志数据的分析与处理，可以将云服务和应用程序的可用性和性能最大化，CCI 2.0支持将pod内的日志上报到云日志服务LTS。

步骤1 登录华为云云日志服务LTS控制台，创建日志组。

步骤2 在创建出的日志组下创建日志流。



步骤3 登录云容器实例 CCI2.0控制台，在左侧导航栏中选择“负载管理”，选择“无状态负载”并单击“YAML创建”。

步骤4 YAML示例如下：

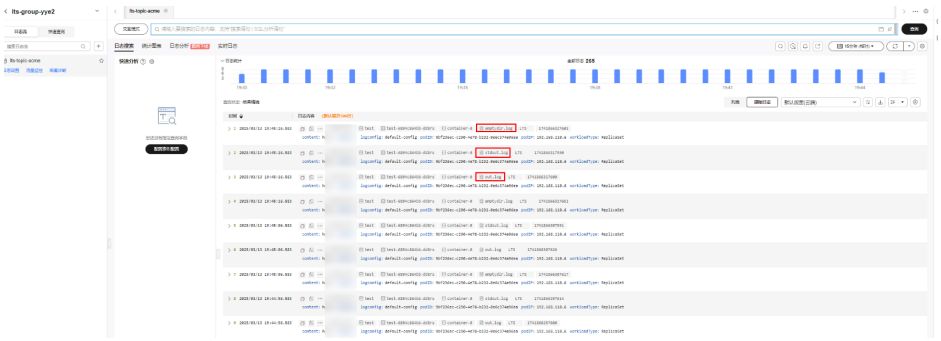
```
apiVersion: cci/v2
kind: Deployment
metadata:
  name: test
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      annotations:
        logconf.k8s.io/fluent-bit-log-type: lts      # 日志上报类型
        logconfigs.logging.openvessel.io: "{\"default-config\":{\"container_files\":{\"container-0\":\"stdout.log;/
root/out.log;/data/emptydir-volume/*.log\"},\"regulation\":\"/(?<log>\\\\d+-\\\\d+-\\\\d+ \\\\d+:\\\\d+:\\\\d
+.*)/\",\"lts-log-info\":{\"349b50f7-f6ba-4768-86f9-e7f84ab677c0\":\"8a9e0589-c3b5-4ed1-
b8cd-1e8a54612d0e\"}}}"  # 日志上报配置
      labels:
        app: test
    spec:
      containers:
      - image: centos:latest
        command: ['sh', '-c', "while true; do echo hello; touch /root/out.log; echo hello >> /root/out.log; touch /
data/emptydir-volume/emptydir.log; echo hello >> /data/emptydir-volume/emptydir.log; sleep 10; done"]
# 模拟日志写入
        volumeMounts:
        - name: emptydir-volume
          mountPath: /data/emptydir-volume
        - name: emptydir-memory-volume
          mountPath: /data/emptydir-memory-volume
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 100Mi
          requests:
            cpu: 100m
            memory: 100Mi
      volumes:
      - name: emptydir-volume
        emptyDir: {}
      - name: emptydir-memory-volume
        emptyDir:
          sizeLimit: 1Gi
          medium: Memory
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
```

annotation中"logconfigs.logging.openvessel.io"为日志上报相关的各类配置项，为json形式，各字段介绍如下：

```
{
    "default-config": { // 配置名称，可自定义名称
        "container_files": { // 支持设置多个容器的日志收集路径（stdout.log表示标准输出，/root/out.log表示
rootfs（包含挂卷）下文本日志，/data/emptydir-xxx/*.log表示rootfs（包含挂卷）下文本目录）
            "container-0": "stdout.log;/root/out.log;/data/emptydir-volume/*.log",
            ...
        },
        "regulation": "/(?<log>\\d+-\\d+-\\d+ \\d+:\\d+:\\d+.*)/", // 多行日志收集正则匹配规则
        "lts-log-info": { // 仅支持设置单个日志组和日志流
            "f938a11a-678e-4631-80a0-3667be1280f7": "a248c18b-61ed-4de2-8f7f-c042b78bb0fd" // 日志组
id：日志流id
        }
    },
    "other-config": { // 支持多配置
        ...
    }
}
```

步骤5 等待负载变为运行中状态，单击“日志”查看Pod上报的日志。



----结束

7.2 AOM 聚合指标与监控

7.2.1 基础指标范围

Prometheus 语句，也称为 PromQL（Prometheus Query Language）查询语句，是用于在 Prometheus 中查询和操作时间序列数据的语言。在CCI 2.0中，提供了以下基础指标，通过这些基础指标可以查询资源对象的聚合指标，用户能够快速地获取资源对象的状态信息。

CCI 2.0提供了以下基础指标，分别涵盖了CPU、磁盘、内存、容器状态、网络、对象状态指标6个维度，具体指标范围如下表所示。

指标类别	指标名称	指标含义
CPU	container_cpu_system_seconds_total	容器系统CPU总时长
	container_cpu_usage_seconds_total	容器在所有CPU内核上的累积占用时间
	container_cpu_user_seconds_total	容器用户CPU总时长
	container_cpu_cfs_periods_total	容器已经执行的CPU时间周期数
	container_cpu_cfs_throttled_periods_total	容器被限流的CPU时间周期数
	container_cpu_cfs_throttled_seconds_total	容器被限流的CPU时间
文件系统/磁盘	container_fs_inodes_free	文件系统的可用inode数量
	container_fs_usage_bytes	文件系统的使用量

指标类别	指标名称	指标含义
	container_fs_inodes_total	文件系统的总计inode数量
	container_fs_io_current	磁盘/文件系统当前正在进行的 I/O 数量
	container_fs_io_time_seconds_total	磁盘/文件系统花费在 I/O 上的累计秒数
	container_fs_io_time_weighted_seconds_total	磁盘/文件系统累积加权 I/O 时间
	container_fs_limit_bytes	容器可以使用的磁盘/文件系统总量
	container_fs_reads_bytes_total	容器累积读取磁盘/文件系统数据的总量
	container_fs_read_seconds_total	容器累积读取磁盘/文件系统数据的秒数
	container_fs_reads_merged_total	容器合并读取磁盘/文件系统的累积计数
	container_fs_reads_total	容器已完成读取磁盘/文件系统的累积计数
	container_fs_sector_reads_total	容器已完成扇区读取磁盘/文件系统的累积计数
	container_fs_sector_writes_total	容器已完成扇区写入磁盘/文件系统的累积计数
	container_fs_writes_bytes_total	容器累积写入磁盘/文件系统数据的总量
	container_fs_write_seconds_total	容器累计写入磁盘/文件系统的秒数
	container_fs_writes_merged_total	容器合并写入磁盘/文件系统的累积计数
	container_fs_writes_total	容器已完成写入磁盘/文件系统的累积计数
	container_blkio_device_usage_total	容器区分IO操作对磁盘的使用总量
内存	container_memory_failures_total	容器内存分配失败的累积计数
	container_memory_failcnt	容器内存使用达到限制的次数
	container_memory_cache	容器总页缓存内存
	container_memory_mapped_file	容器内存映射文件的大小

指标类别	指标名称	指标含义
	container_memory_max_usage_bytes	容器历史最大内存使用量
	container_memory_rss	容器常驻内存集的大小
	container_memory_swap	容器虚拟内存使用量
	container_memory_usage_bytes	容器当前的内存使用量
	container_memory_working_set_bytes	容器工作集内存使用量
网络	container_network_receive_bytes_total	容器网络累积接收数据总量
	container_network_receive_errors_total	接收时遇到的错误累积计数
	container_network_receive_packets_dropped_total	接收时丢弃的数据包的累积计数
	container_network_receive_packets_total	接收数据包的累积计数
	container_network_transmit_bytes_total	容器网络累积传输数据总量
	container_network_transmit_errors_total	传输时遇到的错误累积计数
	container_network_transmit_packets_dropped_total	传输时丢弃的数据包的累积计数
	container_network_transmit_packets_total	传输数据包的累积计数
容器spec/状态	container_processes	容器当前运行的进程数
	container_sockets	容器当前打开套接字的个数
	container_file_descriptors	容器打开的文件描述符数量
	container_threads	容器内当前运行的线程数
	container_threads_max	容器内允许运行的最大线程数
	container_ulimits_soft	容器内1号进程的软 ulimit 值。如果为-1，则无限制，优先级和nice除外
	container_spec_cpu_period	容器分配的CPU周期
	container_spec_cpu_shares	容器分配的CPU份额
	container_spec_cpu_quota	容器分配的CPU配额

指标类别	指标名称	指标含义
	container_spec_memory_limit_bytes	容器可以使用的总内存量限制
	container_spec_memory_reservation_limit_bytes	容器可以使用的预留内存限制
	container_spec_memory_swap_limit_bytes	容器可以使用的虚拟内存限制
	container_start_time_seconds	容器已经运行的时间
	container_last_seen	最近一次监控采集器感知到容器的时间
对象状态指标	kube_pod_info	Pod信息
	kube_pod_owner	Pod的Owner信息
	kube_pod_container_resource_limits	容器的资源limits
	kube_replicaset_owner	RS的所有者信息

7.2.2 创建 Prometheus 实例

使用云服务Prometheus监控，可以监控CCI 2.0云服务的多种指标，本章节将指导用户如何创建普罗实例实现监控。

约束与限制

一个企业项目下仅可以创建一个云服务类型的Prometheus实例。

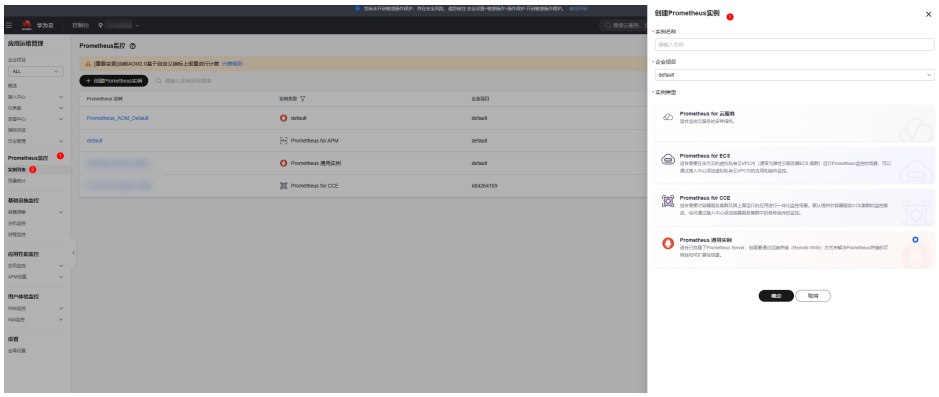
CCI 2.0当前仅支持在华南-广州和华东-上海一局点使用监控管理。

步骤一 创建 Prometheus 实例

1. 登录华为云AOM 2.0控制台。
2. 在左侧导航栏选择“Prometheus监控 -> 实例列表”，单击“创建Prometheus实例”。
3. 设置实例名称、企业项目和实例类型信息。

   说明

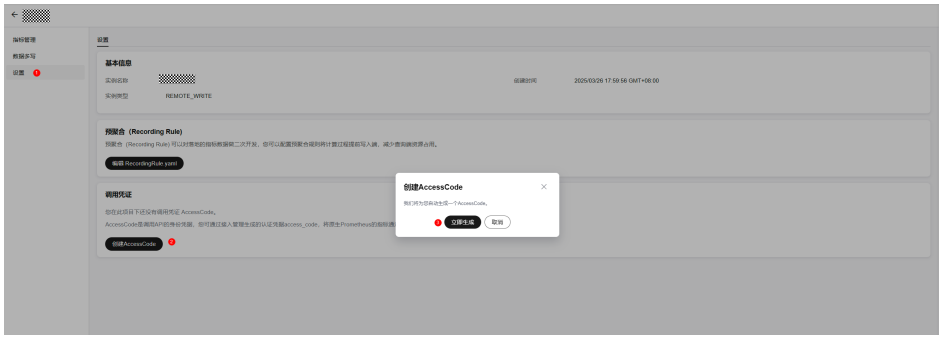
   如果已使用“CCE 突发弹性引擎 (对接 CCI)”插件或需要使用“CCE 突发弹性引擎 (对接 CCI)”插件，Prometheus实例同时需要给CCE集群使用，实例类型需选Prometheus for CCE。



4. 完成后单击“确定”。

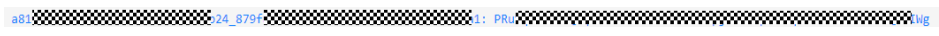
步骤二 获取密钥

1. 单击所创建的Prometheus实例名称，进入详情页。
2. 单击“设置”，在调用凭证栏单击“创建AccessCode”。
3. 单击“立即生成”。



4. 在服务地址栏获取<project_id>、<aom_id>、<aom_secret>。

5. 将以上3个信息拼接为字符串，格式为：<project_id>_<aom_id>:<aom_secret>。



⚠ 注意

冒号后需包含一个空格。

6. 将其生成base64编码。

   YTgxZ[redacted]XU3SVdn

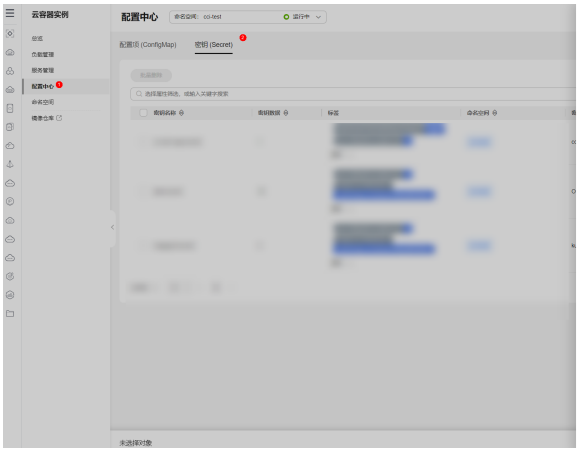
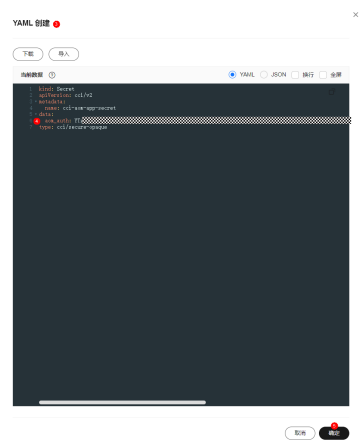
   您可参考以下shell命令生成base64编码：

   ```
   project_id=<project_id>
   aom_id=<aom_id>
   aom_secret=<aom_secret>
   echo -n "${project_id}_${aom_id}: ${aom_secret}" |base64 -w 0
   ```

7. 将生成的base64编码字符串填充到以下模板中，替换{AOMAuthBase64}，并复制此代码。

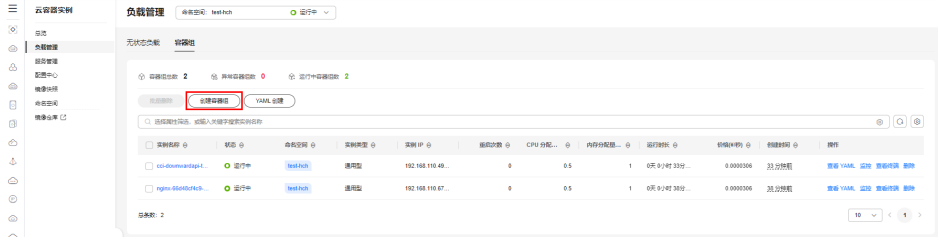
   ```
   kind: Secret
   apiVersion: cci/v2
   metadata:
     name: cci-aom-app-secret
   data:
     aom_auth: {AOMAuthBase64}
   type: cci/secure-opaque
   ```

8. 登录**云容器实例 CCI2.0**控制台。
9. 在左侧导航栏单击“配置中心”。
10. 在配置中心页面选择“密钥（Secret）”。
11. 单击“YAML创建”，将代码替换到CCI界面创建AOM APP Secret。

步骤三 创建容器组

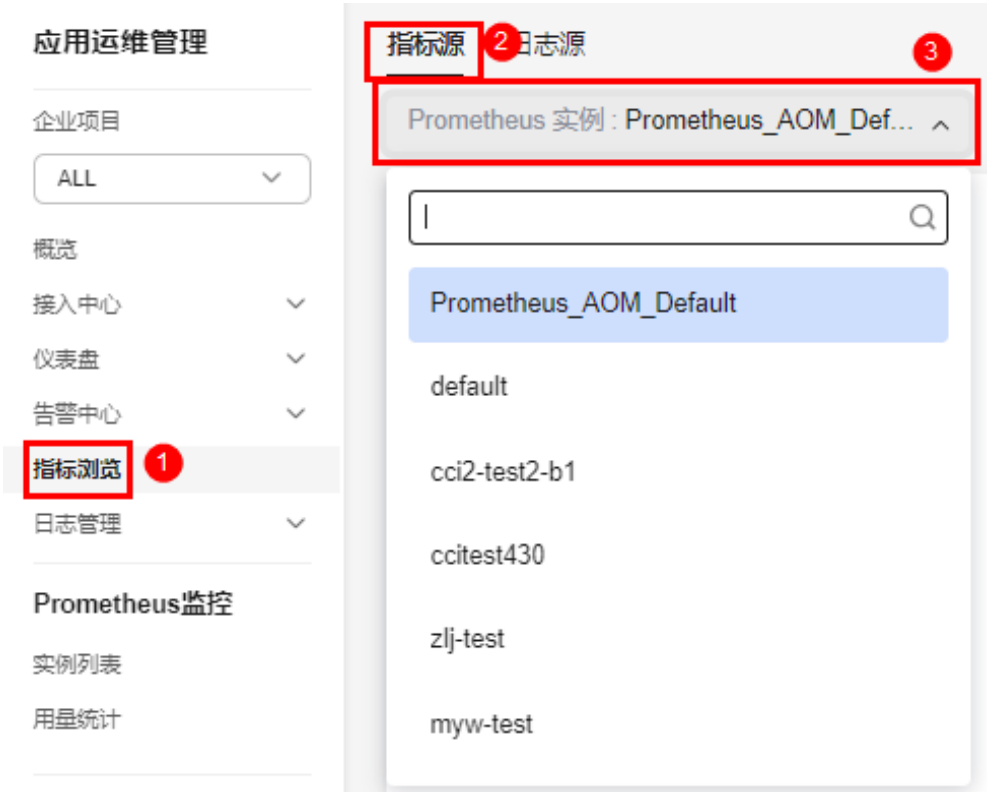
1. 登录华为云云容器实例 CCI2.0控制台。
2. 在左侧导航栏单击“负载管理”，选择“容器组”页签。
3. 单击“创建容器组”，并填写相关信息，可参考**创建容器组**。



4. 完成后单击“立即创建”。

步骤四 监控容器组

1. 登录华为云AOM 2.0控制台。
2. 在左侧导航栏选择“指标浏览”。
3. 在指标源页面选择**步骤一 创建Prometheus实例**。



4. 在“按普罗语句添加”页签，在搜索框输入普罗语句后单击 即可。

说明

使用普罗语句可以查询和监控CCI 2.0云服务的多种指标，使用普罗语句之前请确保容器组已经与Prometheus实例连接。

7.2.3 查询聚合指标样例

通过普罗语句可以将基础指标结合为聚合指标，方便快速查询并查看对象资源信息。本文档列出了以下聚合指标，涵盖CPU利用率、内存利用率、磁盘利用率、磁盘读写速度、网络传输错误率、网络丢包率、网络传输速率指标，支持Deployment、Pod、Container级别查询。本文档列出的聚合指标不代表仅支持以下聚合指标的查询，普罗语句语法详情可登录**prometheus官网**查看。

指标类别	指标名称	指标描述	指标实现

CPU	deployment_cpu_usage	查询Deployment CPU利用率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace) (rate(container_cpu_usage_seconds_total{image!=""}[5m])) / sum by(pod,namespace)(container_spec_cpu_quota{image!=""} / 100000)) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
		查询单个Deployment CPU利用率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace) (rate(container_cpu_usage_seconds_total{image!=""}[5m])) / sum by(pod,namespace)(container_spec_cpu_quota{image!=""} / 100000)) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_name =~ '{DEPLOYMENTNAME}.*',created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
	pod_cpu_usage	查询Pod CPU利用率	sum(rate(container_cpu_usage_seconds_total{image != ""}[5m])) by (pod, namespace) / (sum(container_spec_cpu_quota{image != ""}/100000) by (pod, namespace)) * 100
		查询单个Pod CPU利用率	sum(rate(container_cpu_usage_seconds_total{image != "",pod="{PODNAME}"}[5m])) by (pod, namespace) / (sum(container_spec_cpu_quota{image != "",pod="{PODNAME}"}/100000) by (pod, namespace)) * 100
	container_cpu_usage	查询Container CPU利用率	sum(rate(container_cpu_usage_seconds_total{image != ""}[5m])) by (pod,namespace,container) / (sum(container_spec_cpu_quota{image != ""}/100000) by (pod,namespace,container)) * 100
		查询单个Container CPU利用率	sum(rate(container_cpu_usage_seconds_total{image != "",pod="{PODNAME}",container="{CONTAINERNAME}"}[5m])) by (pod,namespace,container) / (sum(container_spec_cpu_quota{image != "",pod="{PODNAME}",container="{CONTAINERNAME}"}/ 100000) by (pod,namespace,container)) * 100
内存	deployment_memory_usage	查询Deployment 内存利用率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace)(container_memory_working_set_bytes{image!=""}) / sum by(pod,namespace) (container_spec_memory_limit_bytes{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
		查询单个Deployment内存利用率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace)(container_memory_working_set_bytes{image!=""}) / sum by(pod,namespace) (container_spec_memory_limit_bytes{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_name =~ '{DEPLOYMENTNAME}.*',created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
	pod_memory_usage	查询Pod内存利用率	sum(container_memory_working_set_bytes{container!=""}) by (pod, namespace) / (sum(container_spec_memory_limit_bytes{container!=""}) by (pod, namespace)) * 100

		查询单个Pod 内存利用率	sum(container_memory_working_set_bytes{container!="",pod="{PODNAME}"}) by (pod, namespace) / (sum(container_spec_memory_limit_bytes{container!="",pod="{PODNAME}"}) by (pod, namespace)) * 100
	container_memory_usage	查询Container内存利用率	sum(container_memory_working_set_bytes{container!=""}) by (pod, namespace,container) / (sum(container_spec_memory_limit_bytes{container!=""}) by (pod, namespace,container)) * 100
		查询单个Container内存利用率	sum(container_memory_working_set_bytes{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}) by (pod, namespace,container) / (sum(container_spec_memory_limit_bytes{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}) by (pod, namespace,container)) * 100
文件系统/磁盘	deployment_disk_usage	查询Deployment 磁盘利用率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace)(container_fs_usage_bytes{image!=""}) / sum by(pod,namespace)(container_fs_limit_bytes{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
		查询单个Deployment磁盘利用率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace)(container_fs_usage_bytes{image!=""}) / sum by(pod,namespace)(container_fs_limit_bytes{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_name =~ '{DEPLOYMENTNAME}.*',created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
	pod_disk_usage	查询Pod磁盘利用率	(sum(container_fs_usage_bytes{container!=""}) by (pod)) / (sum(container_fs_limit_bytes{container!=""}) by (pod)) * 100
		查询单个Pod磁盘利用率	(sum(container_fs_usage_bytes{container!="",pod="{PODNAME}"}) by (pod)) / (sum(container_fs_limit_bytes{container!="",pod="{PODNAME}"}) by (pod)) * 100
	container_disk_usage	查询Container磁盘利用率	(sum(container_fs_usage_bytes{container!=""}) by (pod,container)) / (sum(container_fs_limit_bytes{container!=""}) by (pod,container)) * 100
		查询单个Container磁盘利用率	(sum(container_fs_usage_bytes{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}) by (pod,container)) / (sum(container_fs_limit_bytes{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}) by (pod,container)) * 100
	disk_read_kilobytes	查询磁盘的读取速率	sum(rate(container_fs_reads_total{container!=""}[5m])) by(namespace)
	disk_write_kilobytes	查询磁盘的写入速率	sum(rate(container_fs_writes_total{container!=""}[5m])) by(namespace)

网络	deployment_network_transmit_error_packets	查询Deployment 传输错误率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace) (container_network_transmit_packets_dropped_total{image!=""}) / sum by(pod,namespace) (container_network_transmit_packets_total{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
		查询单个Deployment 传输错误率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace) (container_network_transmit_packets_dropped_total{image!=""}) / sum by(pod,namespace) (container_network_transmit_packets_total{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_name =~ '{DEPLOYMENTNAME}.*',created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
	pod_network_transmit_error_packets	查询Pod 传输错误率	sum(rate(container_network_transmit_packets_dropped_total{container!=""}[5m])) by(namespace,pod)/ sum(rate(container_network_transmit_packets_total{container!=""}[5m])) by(namespace,pod)* 100
		查询单个Pod 传输错误率	sum(rate(container_network_transmit_packets_dropped_total{container!="",pod="{PODNAME}"}[5m])) by(namespace,pod)/ sum(rate(container_network_transmit_packets_total{container!="",pod="{PODNAME}"}[5m])) by(namespace,pod)* 100
	container_network_transmit_error_packets	查询Container 传输错误率	sum(rate(container_network_transmit_packets_dropped_total{container!=""}[5m])) by(namespace,pod,container)/ sum(rate(container_network_transmit_packets_total{container!=""}[5m])) by(namespace,pod,container)* 100
		查询单个Container 传输错误率	sum(rate(container_network_transmit_packets_dropped_total{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}[5m])) by(namespace,pod,container)/ sum(rate(container_network_transmit_packets_total{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}[5m])) by(namespace,pod,container)* 100
	deployment_network_receive_error_packets	查询Deployment 丢包率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace) (container_network_receive_packets_dropped_total{image!=""}) / sum by(pod,namespace) (container_network_receive_packets_total{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100
		查询单个Deployment 丢包率	label_replace(sum by (created_by_name,namespace)((sum by (pod,namespace) (container_network_receive_packets_dropped_total{image!=""}) / sum by(pod,namespace) (container_network_receive_packets_total{image!=""})) *on(pod,namespace) group_left(created_by_kind,created_by_name) (count by (pod,created_by_kind,created_by_name,namespace) (kube_pod_info{created_by_name =~ '{DEPLOYMENTNAME}.*',created_by_kind='ReplicaSet'}))),"deployment","$1","created_by_name","(.*)-[^-]*")*100

	pod_network_receive_error_packets	查询Pod丢包率	sum(rate(container_network_receive_packets_dropped_total{container!=""}[5m])) by(namespace,pod)/ sum(rate(container_network_receive_packets_total{container!=""}[5m])) by(namespace,pod)* 100
		查询单个Pod 丢包率	sum(rate(container_network_receive_packets_dropped_total{container!="",pod="{PODNAME}"}[5m])) by(namespace,pod)/ sum(rate(container_network_receive_packets_total{container!="",pod="{PODNAME}"}[5m])) by(namespace,pod)* 100
	container_network_receive_error_packets	查询Container丢包率	sum(rate(container_network_receive_packets_dropped_total{container!=""}[5m])) by(namespace,pod,container)/ sum(rate(container_network_receive_packets_total{container!=""}[5m])) by(namespace,pod,container)* 100
		查询单个Container丢包率	sum(rate(container_network_receive_packets_dropped_total{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}[5m])) by(namespace,pod,container)/ sum(rate(container_network_receive_packets_total{container!="",pod="{PODNAME}",container="{CONTAINERNAME}"}[5m])) by(namespace,pod,container)* 100
	network_transmit_bytes	查询网络发送速率	sum(rate(container_network_transmit_bytes_total{container!=""}[5m])) by(namespace)
	network_receive_bytes	查询网络接收速率	sum(rate(container_network_receive_bytes_total{container!=""}[5m])) by(namespace)

⚠ 注意

- 查询单个Deployment对象资源时，将语句中{DEPLOYMENTNAME}修改为待查询的Deployment名。
- 查询单个Pod对象资源时，将语句中{PODNAME}修改为待查询的Pod名。
- 查询单个Container对象资源时，将语句中{PODNAME}修改为待查询Container所属的Pod名，将{CONTAINERNAME}改为待查询的Container名。

7.2.4 使用 AOM 仪表盘监控 CCI 2.0 指标

AOM支持将多个指标以可视化的方式展示在同一个屏幕上。CCI 2.0支持将重要资源的关键指标添加到仪表盘中，实时监控该指标数据。

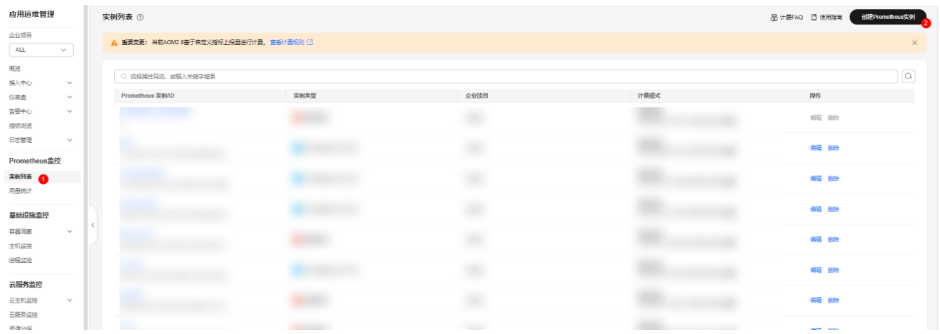
操作步骤

步骤1 在云容器实例 CCI2.0控制台界面创建或修改命名空间处，单击开启对接AOM。

步骤2 选择AOM实例或单击“新建实例”，无需新建则跳到步骤**步骤3**。

1. 新建实例。进入AOM 2.0界面新建实例。

2. 在左侧导航栏选择“Prometheus监控 -> 实例列表”，单击“创建Prometheus实例”。
3. 设置实例名称、企业项目和实例类型信息。



4. 完成后单击“确定”。

步骤3 在云容器实例 CCI2.0控制台界面创建或修改命名空间处，单击刷新，在AOM实例下拉列表中选择已创建的AOM实例。

步骤4 配置其他相关信息，完成后单击“确定”。

步骤5 在AOM 2.0控制台，选择仪表盘，单击“添加仪表盘”。

图 7-1 添加仪表盘

步骤6 填入仪表盘名称，企业项目，分组类型等信息，单击“确定”。

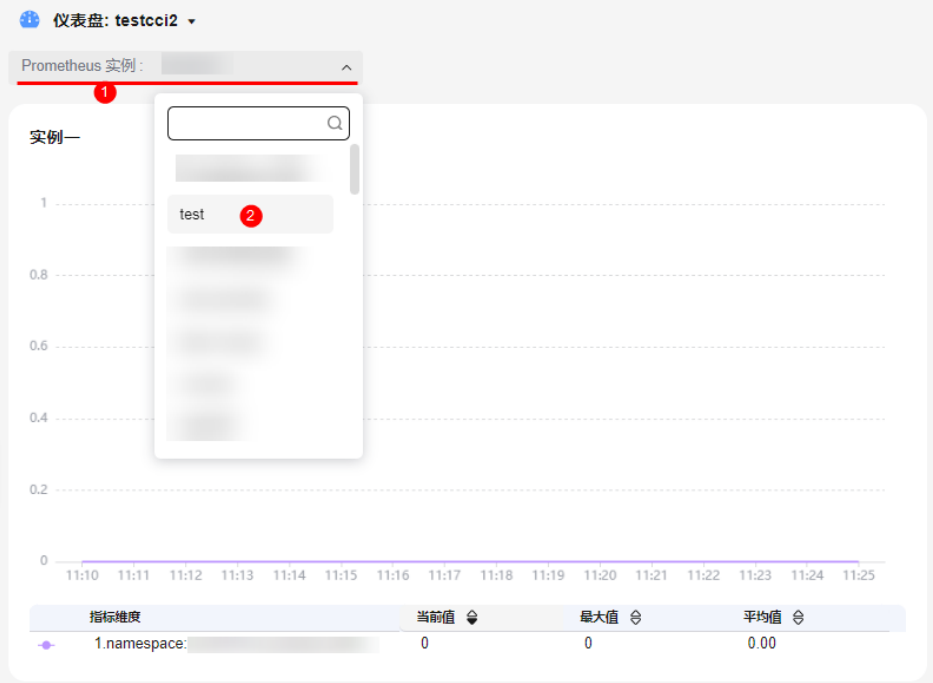
图 7-2 填写仪表盘信息

步骤7 单击创建的仪表盘名称，进入仪表盘详情页面。

步骤8 单击“添加图表”图标，填写图表标题，选择“语法模式（PromQL）”，添加普罗语句，CCI 2.0仪表盘指标可参考**仪表盘指标**，并单击“保存”。

步骤9 您可在AOM 2.0仪表盘详情页面查看该指标数据。

图 7-3 指标数据



----结束

仪表盘指标

维度	类别	指标名	指标含义
deployment	CPU	vCPU Usage	CPU 使用率
		Used vCPUs	CPU 使用量
	mem	Memory Usage	内存使用率
		Used Memory	内存使用量
	net	Network Rate	网络速率
		Packet Loss	网络丢包率
pod	CPU	vCPU Usage	CPU 使用率
		Used vCPUs	CPU 使用量
	mem	Memory Usage	内存使用率
		Used Memory	内存使用量
	net	Network Rate	网络速率
container	CPU	vCPU Usage	CPU 使用率
	mem	Memory Usage	内存使用率

7.2.5 Pod 详情中通过 AOM 查看监控数据

步骤1 登录云容器实例 CCI2.0控制台，在左侧导航栏中选择“负载管理”。

步骤2 切换到“容器组”页面，单击实例名称，进入详情页面。

步骤3 切换到“监控”页面，查看pod的CPU、内存、网络和容器内的相关指标。

----结束