

云容器实例

# 开发指南

文档版本 01  
发布日期 2021-05-06



版权所有 © 华为技术有限公司 2021。保留一切权利。

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

## 商标声明



HUAWEI和其他华为商标均为华为技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

## 注意

您购买的产品、服务或特性等应受华为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，华为公司对本文档内容不做任何明示或默示的声明或保证。

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

---

<b>1 简介</b> .....	<b>1</b>
<b>2 使用 kubectl (推荐)</b> .....	<b>3</b>
<b>3 Namespace 和 Network</b> .....	<b>10</b>
<b>4 Pod</b> .....	<b>16</b>
4.1 Pod.....	16
4.2 环境变量.....	20
4.3 启动命令.....	21
4.4 生命周期管理.....	21
4.5 存活探针 ( liveness probe ) .....	22
<b>5 Label</b> .....	<b>25</b>
<b>6 Deployment</b> .....	<b>28</b>
<b>7 使用 Service 和 Ingress 管理网络访问</b> .....	<b>32</b>
7.1 Service.....	32
7.2 Ingress.....	37
7.3 网络访问场景.....	40
7.4 业务探针 ( Readiness probe ) .....	41
<b>8 使用 PersistentVolumeClaim 申请持久化存储</b> .....	<b>45</b>
<b>9 使用 ConfigMap 和 Secret 提高配置灵活性</b> .....	<b>49</b>
9.1 ConfigMap.....	49
9.2 Secret.....	50
<b>10 使用 Job 和 CronJob 创建任务负载</b> .....	<b>53</b>
<b>A YAML 语法</b> .....	<b>55</b>

# 1 简介

云容器实例（Cloud Container Instance，CCI）服务提供 ServerlessContainer（无服务器容器）引擎，让您无需创建和管理服务器集群即可直接运行容器。云容器实例的 Serverless Container 就是从使用角度，无需创建、管理 Kubernetes 集群，也就是从使用的角度看不见服务器（Serverless），直接通过控制台、kubectl、Kubernetes API 创建和使用容器负载，且只需为容器所使用的资源付费。

本文旨在介绍如何[使用kubectl](#)或调用[云容器实例的API](#)使用云容器实例的各种功能。

## 文档导读

本文档分为如下几个部分。

- 使用kubectl**  
介绍如何在云容器实例中配置kubectl。CCI支持使用原生kubectl或定制的kubectl来创建负载等资源，建议优先选用原生kubectl。
- Namespace和Network**  
介绍云容器实例中Namespace与Network的概念。
- Pod**  
介绍Pod相关概念，以及如何使用Pod。
- Label**  
介绍Label的作用，以及如何使用Label。
- 无状态负载（Deployment）**  
介绍Deployment的使用场景，如何使用Deployment将容器镜像部署到云容器实例中。
- 访问负载**  
介绍Service和Ingress两种管理负载访问的资源对象，使用Service和Ingress解决负载访问的问题。
  - Service：定义一系列Pod以及访问这些Pod的策略的一层抽象。
  - Ingress：管理外部访问的资源对象。
- 使用存储**  
介绍负载中如何使用存储，即如何在容器中如何使用存储卷。包括如何使用云硬盘（EVS）、弹性文件服务（SFS）、对象存储（OBS）。
- 使用ConfigMap和Secret**

介绍如何使用ConfigMap和Secret。

ConfigMap和Secret用于保存配置信息和敏感信息，从而提高负载配置的易用性和灵活性。

9. **使用Job和CronJob**

介绍如何使用Job。Job适用于一次性任务的场景。

# 2 使用 kubectl (推荐)

云容器实例支持使用原生kubectl或定制的kubectl来创建负载等资源，建议优先选用原生kubectl。

## 下载 kubectl

请到[kubernetes版本发布页面](#)下载1.15版本对应的kubectl。

## 下载 cci-iam-authenticator

在CCI官网下载cci-iam-authenticator二进制，最新版本为v2.6.17。

针对不同的操作系统，cci-iam-authenticator的下载地址如[表2-1](#)所示。

表 2-1 下载地址

操作系统	下载地址	查看帮助
Linux AMD 64位	<a href="#">cci-iam-authenticator_linux-amd64</a> <a href="#">cci-iam-authenticator_linux-amd64_sha256</a>	<a href="#">cci-iam-authenticator使用参考</a>

## 安装并设置 kubectl

以下操作以Linux环境为例，更多详情信息请参见[安装和配置kubectl](#)。

**步骤1** 将[下载kubectl](#)中下载的kubectl赋予可执行权限，并放到PATH目录下。

```
chmod +x ./kubectl
```

```
mv ./kubectl $PATH
```

其中，*\$PATH*为PATH路径（如/usr/local/bin），请替换为实际的值。

您还可以通过如下命令查看kubectl的版本，如下所示。

```
kubectl version --client=true
```

```
Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.0",  
GitCommit:"e8462b5b5dc2584fdcd18e6bcfe9f1e4d970a529", GitTreeState:"clean",  
BuildDate:"2019-06-19T16:40:16Z", GoVersion:"go1.12.5", Compiler:"gc", Platform:"linux/amd64"}
```

## 步骤2 配置IAM认证信息并持久化到本地。

1. 将[下载cci-iam-authenticator](#)中下载的cci-iam-authenticator赋予可执行权限，并放到PATH目录下。

```
chmod +x ./cci-iam-authenticator  
mv ./cci-iam-authenticator $PATH
```

2. 初始化cci-iam-authenticator配置。

初始化cci-iam-authenticator提供了AK/SK和用户名/密码两种方式，请选择其中一种执行。

- 使用AK/SK的方式配置IAM认证信息

```
cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://  
$endpoint --ak=xxxxxxx --sk=xxxxxxx
```

其中，endpoint为云容器实例的Endpoint，获取方法请参见[地区和终端节点](#)；ak、sk的获取方法请参见[获取AK/SK](#)，ak为文件中Access Key部分，sk为文件中Secret Key部分。

例如，Endpoint为https://cci.cn-north-4.myhuaweicloud.com，ak的值为my-ak，sk的值为ABCDEFK..，则命令如下所示：

```
cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://  
cci.cn-north-4.myhuaweicloud.com --ak=my-ak --sk=ABCDEFK..
```

执行上述命令后，显示如下类似信息：

```
Switched to context "cci-context-cn-north-4-my-ak"
```

其中，cci-context-cn-north-4-my-ak为context名，可通过[kubectl config get-contexts](#)命令查看。

- 使用用户名/密码的方式配置IAM认证信息

```
cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://  
$endpoint --domain-name=xxxxxxx --user-name=xxxxxx --  
password=xxxxxx
```

其中，endpoint为云容器实例的Endpoint，domain-name为租户名，user-name为子用户名，password为子用户密码，请根据替换为实际的值。

### 📖 说明

- 若无子用户，user-name与domain-name配置一致即可，也可以不添加user-name参数。
- IAM的Endpoint请参见[地区和终端节点](#)，请注意需要使用与云容器实例地区相同的Endpoint。
- 在非安全的环境中使用kubectl，建议您完成此步骤后，使用环境变量的方式重新配置认证信息，具体参考[非安全环境配置kubectl](#)。

## 步骤3 配置完成后，即可通过kubectl命令操作云容器实例的相关资源。

例如，查看北京四的namespace资源。

```
kubectl get ns
```

```
No resources found.
```

您可以从回显中看到北京四没有任何命名空间，在云容器实例中创建资源首先需要创建一个命名空间，具体方法请参见[Namespace](#)和[Network](#)。

----结束

### 警告

- 当通过API访问公有云系统时，需要使用访问用户名密码或者密钥（AK/SK）进行身份认证并对请求进行加密，确保请求的机密性、完整性和请求双方身份的正确性。请妥善保存\$HOME/.kube/config配置文件，确保访问密钥不被非法使用。
- 当开启cache缓存token提高访问性能时，token会以文件的方式保存在\$HOME/.cci/cache的子目录下，请及时清理。
- 当发现访问密钥被非法使用（包括丢失、泄露等情况），可以自行删除或者通知管理员重置访问密钥，重新配置。
- 删除的访问密钥将无法恢复。

## 非安全环境配置 kubectl

**步骤1** 参照上述操作，安装并设置kubectl。

**步骤2** 编辑kubeconfig文件，删除敏感信息参数。

Linux系统，kubeconfig文件默认位于\$HOME/.kube/config。

表 2-2 待删除敏感信息参数

Command Flag	Environment Value	Description
--domain-name	DOMAIN_NAME	租户名
--user-name	USER_NAME	子用户名
--password	PASSWORD	用户密码
--ak	ACCESS_KEY_ID	Access Key
--sk	SECRET_ACCESS_KEY	Secret Key
--cache	CREDENTIAL_CACHE	是否开启缓存Token

更多参数说明请参见[cci-iam-authenticator使用参考](#)。

**步骤3** 配置删除参数相应的环境变量来使用kubectl，以AK/SK为例。

```
export ACCESS_KEY_ID=xxxxxxx
export SECRET_ACCESS_KEY=xxxxxxx
export CREDENTIAL_CACHE=false
kubectl get ns
```

执行上述命令后，提示如下类似信息：



```
No resources found.
```

----结束

## 获取 AK/SK

AK(Access Key ID): 访问密钥ID。与私有访问密钥关联的唯一标识符; 访问密钥ID和私有访问密钥一起使用, 对请求进行加密签名。

SK(Secret Access Key): 与访问密钥ID结合使用的密钥, 对请求进行加密签名, 可标识发送方, 并防止请求被修改。

**步骤1** 登录管理控制台。

**步骤2** 单击用户名, 在下拉列表中单击“我的凭证”。

**步骤3** 在“我的凭证”页面, 单击“访问密钥”页签。

**步骤4** 单击“新增访问密钥”, 输入验证码。

**步骤5** 单击“确定”, 生成并下载访问密钥。

### 📖 说明

为防止访问密钥泄露, 建议您将其保存到安全的位置。

----结束

## 获取云容器实例 Endpoint

Endpoint在[地区和终端节点](#)页面获取, 如下表所示。

表 2-3 地区和终端节点

区域名称	区域	终端节点 (Endpoint)
华北-北京四	cn-north-4	cci.cn-north-4.myhuaweicloud.com
华东-上海一	cn-east-3	cci.cn-east-3.myhuaweicloud.com
华南-广州	cn-south-1	cci.cn-south-1.myhuaweicloud.com

## cci-iam-authenticator 使用参考

cci-iam-authenticator作为k8s client端的认证插件, 主要提供了generate-kubeconfig和token两个子命令。

```
A tool to authenticate to CCI using HuaweiCloud IAM credentials
```

```
Usage:  
cci-iam-authenticator [command]
```

```
Available Commands:  
generate-kubeconfig Generate or modify kubeconfig files based on user configuration  
help Help about any command  
token Authenticate using HuaweiCloud IAM and get token for CCI
```

```

Flags:
  --alsologtostderr log to standard error as well as files
  -h, --help          help for cci-iam-authenticator
  --log_dir string    If non-empty, write log files in this directory
  --log_file string   If non-empty, use this log file
  --logtostderr       log to standard error instead of files (default true)
  -v, --v Level       number for the log level verbosity
  --version           version for cci-iam-authenticator

```

Use "cci-iam-authenticator [command] --help" for more information about a command.

其中，Flags主要为日志选项。

### token

token子命令用于获取用户token，获取token的认证方式有用户名/密码、ak/sk两种，选择其中一种即可。

Authenticate using HuaweiCloud IAM and get token for CCI

```

Usage:
  cci-iam-authenticator token [flags]

```

```

Flags:
  --ak string          IAM access key ID
  --cache              Cache the token credential on disk until it expires (default true)
  --domain-name string IAM domain name, typically your account name
  -h, --help          help for token
  --iam-endpoint string HuaweiCloud IAM endpoint, i.e. https://iam.cn-north-4.myhuaweicloud.com
  (default "https://iam.myhuaweicloud.com")
  --insecure-skip-tls-verify If true, the iam server's certificate will not be checked for validity. (default
  true)
  --password string   IAM user password
  --project-id string IAM project id, project id and project name should not be empty at same time
  --project-name string IAM project name, project id and project name should not be empty at same
  time
  --sk string         IAM secret access key
  --token-only        Return token only for other tool integration
  --user-name string  IAM user name. Same as domain-name when using main account, otherwise
  use iam user name

```

其中，Flags分为用户名密码、AKSK和公共配置。

表 2-4 用户名/密码配置

Command Flag	Environment Value	Description
domain-name	DOMAIN_NAME	租户名，即账号名，详情请参见 <a href="https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html">https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html</a>
user-name	USER_NAME	子用户名，即IAM用户名。若不配置与domain-name一致。详情请参见 <a href="https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html">https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html</a>
password	PASSWORD	用户或子用户密码。

表 2-5 AK/SK 配置

Command Flag	Environment Value	Description
ak	ACCESS_KEY_ID	ak、sk的获取方法请参见 <a href="#">获取AK/SK</a> ，ak为文件中Access Key部分，sk为文件中Secret Key部分。
sk	SECRET_ACCESS_KEY	

表 2-6 公共配置

Command Flag	Environment Value	Description
iam-endpoint	IAM_ENDPOINT	云容器实例的Endpoint，必须配置，详情请参见 <a href="#">获取云容器实例Endpoint</a> 。
project-name	PROJECT_NAME	项目名，详情请参见 <a href="https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html">https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html</a>
project-id	PROJECT_ID	项目ID，详情请参见 <a href="https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html">https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html</a>
insecure-skip-tls-verify	INSECURE_SKIP_TLS_VERIFY	是否跳过对CCI/IAM服务端的校验，默认为true。
cache	CREDENTIAL_CACHE	是否开启将IAM Token缓存到本地，提高访问性能，默认为true。 <b>注意</b> 在非安全环境，建议关闭此选项。

### generate-kubeconfig

为用户直接生成kubeconfig配置，若指定的kubeconfig已存在，则会注入新的server、user、context配置，并将当前的kubeconfig context切换到此次配置的结果。默认情况下会对用户的配置进行校验，尝试访问IAM及CCI，确保用户配置的IAM认证信息、CCI地址可用。

Generate or modify kubeconfig files based on user configuration.

Sets a cluster entry, a user entry and a context entry in kubeconfig and use this context as the current-context.

The loading order follows these rules:

1. If the `--kubeconfig` flag is set, then only that file is loaded. The flag may only be set once and no merging takes place.
2. If `$KUBECONFIG` environment variable is set, then it is used as a list of paths (normal path delimiting rules for your system). These paths are merged. When a value is modified, it is modified in the file that defines the stanza. When a value is created, it is created in the first file that exists. If no files in the chain exist, then it creates the last file in the list.
3. Otherwise, `/${HOME}/.kube/config` is used and no merging takes place.

Examples:

```
# Generate kubeconfig to ${HOME}/.kube/config using aksk
cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://cci.cn-north-4.myhuaweicloud.com --
ak=*** --sk=***
# Generate kubeconfig to ${HOME}/.kube/config using domain name and password
cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://cci.cn-north-4.myhuaweicloud.com --
domain-name=*** --password=***
```

Usage:

```
cci-iam-authenticator generate-kubeconfig [flags]
```

Flags:

```
--ak string          IAM access key ID
--cache              Cache the token credential on disk until it expires (default true)
--cci-endpoint string CCI server endpoint, i.e. https://cci.cn-north-4.myhuaweicloud.com
--domain-name string IAM domain name, typically your account name
-h, --help          help for generate-kubeconfig
--iam-endpoint string HuaweiCloud IAM endpoint, i.e. https://iam.cn-north-4.myhuaweicloud.com
(default "https://iam.myhuaweicloud.com")
--insecure-skip-tls-verify If true, the iam server's certificate will not be checked for validity. (default
true)
--kubeconfig string  use a particular kubeconfig file
--password string    IAM user password
--project-id string  IAM project id, project id and project name should not be empty at same time
--project-name string IAM project name, project id and project name should not be empty at same
time
--sk string          IAM secret access key
--token-only         Return token only for other tool integration
--user-name string   IAM user name. Same as domain-name when using main account, otherwise
use iam user name
--validation         Validate kubeconfig by trying to access CCI with existing config (default true)
```

同一个kubeconfig可以包含多个环境、认证信息，用户可以通过同一份IAM认证配置，仅修改`cci-endpoint`生成多个region的kubeconfig，例如：

```
# 生成北京4的kubeconfig，并切换到对应context
$ cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://cci.cn-north-4.myhuaweicloud.com --
ak=my-ak --sk=xxxxxx
Switched to context "cci-context-cn-north-4-my-ak".
# 生成上海1的kubeconfig，并切换到对应context
$ cci-iam-authenticator generate-kubeconfig --cci-endpoint=https://cci.cn-east-3.myhuaweicloud.com --
ak=my-ak --sk=xxxxxx
Switched to context "cci-context-cn-east-3-my-ak".
# 切换到北京4的context
$ kubectl config use-context cci-context-cn-north-4-my-ak
```

# 3 Namespace 和 Network

Namespace（命名空间）是一种在多个用户之间划分资源的方法。适用于用户中存在多个团队或项目的情况。当前云容器实例提供“通用计算型”和“GPU型”两种类型的资源，创建命名空间时需要选择资源类型，后续创建的负载中容器就运行在此类型的集群上。

- **通用计算型**：支持创建含CPU资源的容器实例及工作负载，适用于通用计算场景。
- **GPU型**：支持创建含GPU资源的容器实例及工作负载，适用于深度学习、科学计算、视频处理等场景。

Network是云容器实例扩展的一种Kubernetes资源对象，用于关联VPC及子网，从而使得容器实例能够使用公有云的网络资源。

## Namespace 与网络的关系

从网络角度看，命名空间对应一个虚拟私有云（VPC）中一个子网，如图3-1所示，在创建命名空间时会关联已有VPC或创建一个新的VPC，并在VPC下创建一个子网。后续在该命名空间下创建Pod、Service等资源时都会在对应的VPC及子网之内，且占用子网中的IP地址。

通常情况下，如果您在同一个VPC下还会使用其他服务的资源，您需要考虑您的网络规划，如子网网段划分、IP数量规划等，确保有可用的网络资源。

图 3-1 命名空间与 VPC 子网的关系



## 哪些情况下适合使用多个命名空间

因为Namespace可以实现部分的环境隔离，当您的项目和人员众多的时候可以考虑根据项目属性，例如生产、测试、开发划分不同的Namespace。

## 创建 Namespace

Namespace下需要有一个Network关联VPC及子网，创建完Namespace后需要创建一个Network。

### 📖 说明

通常情况下，没有频繁创建Namespace的需求，建议通过云容器实例的控制台界面创建Namespace，具体方法请参见[创建命名空间](#)。

以下示例创建一个名为namespace-test的Namespace，指定云容器实例的资源类型为general-computing。

```
apiVersion: v1
kind: Namespace
metadata:
  name: namespace-test
  labels:
    sys_enterprise_project_id: "0"
  annotations:
    namespace.kubernetes.io/flavor: general-computing
spec:
  finalizers:
    - kubernetes
```

这里的定义文件采用YAML格式描述（如果您对YAML格式不了解，可以参考[YAML语法](#)），也是使用JSON格式。

- `sys_enterprise_project_id`：表示企业项目ID，可进入[企业管理](#)的企业项目详情页面获取。未开通企业管理的用户无需配置此参数。不配置时默认为0，表示default企业项目。
- `namespace.kubernetes.io/flavor: general-computing`：指定命名空间类型。命名空间的类型有如下两种：
  - **general-computing**：通用计算型，支持创建含CPU资源的容器实例及工作负载，适用于通用计算场景。
  - **gpu-accelerated**：GPU型，支持创建含GPU资源的容器实例及工作负载，适用于深度学习、科学计算、视频处理等场景。

假如上面Namespace定义的文件名称为ns.yaml，则执行`kubectl create -f ns.yaml`即可创建命名空间，`-f`表示从文件创建。

```
# kubectl create -f ns.yaml
namespace/namespace-test created
```

执行`kubectl get ns`查询namespace是否创建成功，ns为namespace的缩写。

```
# kubectl get ns
NAME          STATUS   AGE
namespace-test Active   23s
```

如上，可以看到namespace-test这个命名空间创建成功，且存在的时长为23秒。

登录云容器实例控制台，单击左侧导航栏“命名空间”，您可以看到命名空间创建成功，但状态为“异常”。这是因为在云容器实例中，您需要为Namespace定义网络策略，具体操作方法请参见[创建Network](#)。

图 3-2 Namespace-异常

名称	类型	状态
namespace-test	通用计算型	异常

## 创建 Network

Namespace创建好后，需要为Namespace创建网络策略，关联VPC及子网。

以下示例创建一个名为test-network的Network。

```
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: cnnorth-1a
  cidr: 192.168.0.0/24
  attachedVPC: vpc-id
  networkID: network-id
  networkType: underlay_neutron
  subnetID: subnet-id
```

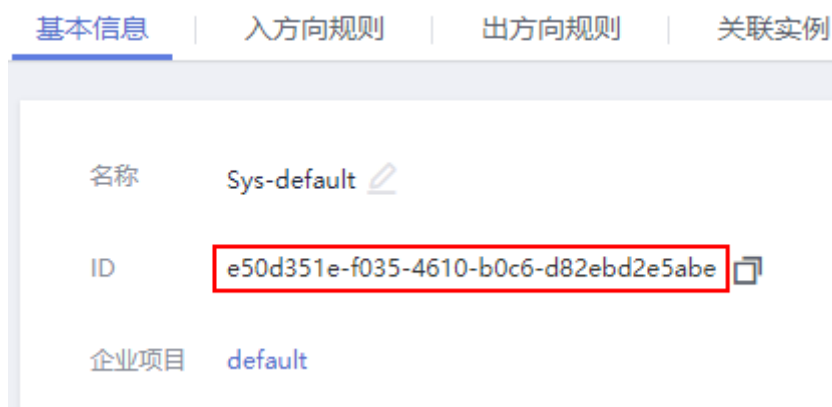
### 说明

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Service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Service访问，而是直接访问Pod，则可以使用此网段。

上面参数获取方法如下：

- network.alpha.kubernetes.io/domain-id：账号ID，可以在[我的凭证](#)获取。
- network.alpha.kubernetes.io/project-id：项目ID，可以在[我的凭证](#)获取。
- network.alpha.kubernetes.io/default-security-group：安全组ID，可以在[安全组控制台](#)获取，如下图。

图 3-3 获取安全组 ID



- availableZone：az名称，当前仅支持“cnnorth-1a”、“cn-north-4a”或“cn-east-3a”。

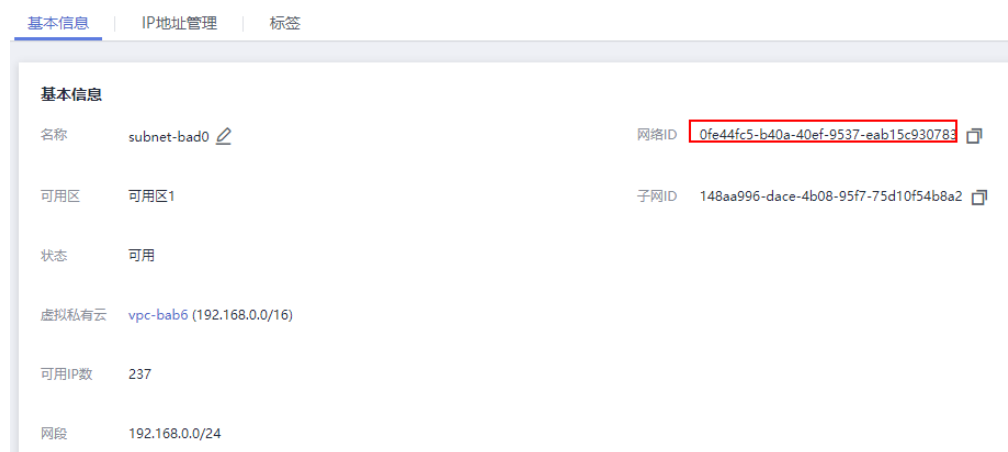
- cidr: 子网网段。
- attachedVPC: 虚拟私有云的ID, 可在VPC控制台获取。

图 3-4 获取 VPC ID



- networkID: 子网的网络ID, 可在VPC控制台 > 子网中获取。

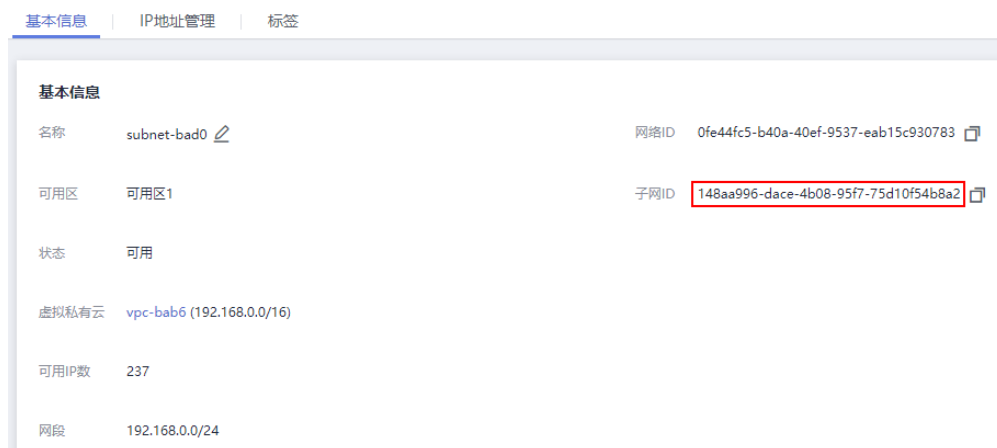
图 3-5 获取子网的网络 ID



- networkType: 网络类型, 当前仅支持underlay\_neutron网络模式。
- subnetID: 子网ID, 可在VPC控制台 > 子网获取。



图 3-6 获取子网 ID



假如上面Network定义的文件名称为network.yaml，则执行kubect create -f network.yaml即可创建命名空间，-f 表示从文件创建。这里--namespace namespace-test是指定在namespace-test这个命名空间下创建。

```
# kubect create -f network.yaml --namespace namespace-test  
network.networking.cci.io/test-network created
```

登录云容器实例控制台，单击左侧导航栏“命名空间”，您可以看到命名空间创建成功，且状态为“正常”。

图 3-7 Namespace-正常

名称	类型	状态
namespace-test	通用计算型	正常

## 为 kubectl 上下文指定 Namespace

上面创建Network是在指定的Namespace下创建的，本文档后续的资源创建都是在某个命名空间下操作，每次都指定命名空间比较麻烦，您可以为kubectl上下文指定命名空间，这样在某个上下文中，创建的资源就都是在某个命名空间下，方便操作。

指定Namespace只需要在设置上下文命令中添加一个“--namespace”选项，如下所示。

```
kubectl config set-context $context --namespace=$ns
```

其中，*\$ns*为Namespace的名称；*\$context* 为上下文的名称，可以自定义，也可执行如下命令获取：

```
# kubectl config get-contexts  
CURRENT NAME CLUSTER AUTHINFO  
NAMESPACE  
cci-context-cn-east-3-1C8PNI0POPPCSFGXPM6S cci-cluster-cn-east-3 cci-user-cn-east-3-1C8PNI0POPPCSFGXPM6S  
* cci-context-cn-east-3-hwuser_xxx cci-cluster-cn-east-3 cci-user-cn-east-3-hwuser_xxx  
kubernetes-admin@kubernetes kubernetes kubernetes-admin
```

假设，上面创建的Namespace名称为namespace-test，则示例如下。

```
# kubectl config set-context cci-context --namespace=namespace-test
```

指定Namespace后，就可以使用 `kubectl` 命令直接操作云容器实例的相关资源。如下所示，执行**`kubectl get pod`**，查看Pod资源，一切正常。

```
# kubectl get pod  
No resources found.
```

# 4 Pod

## 4.1 Pod

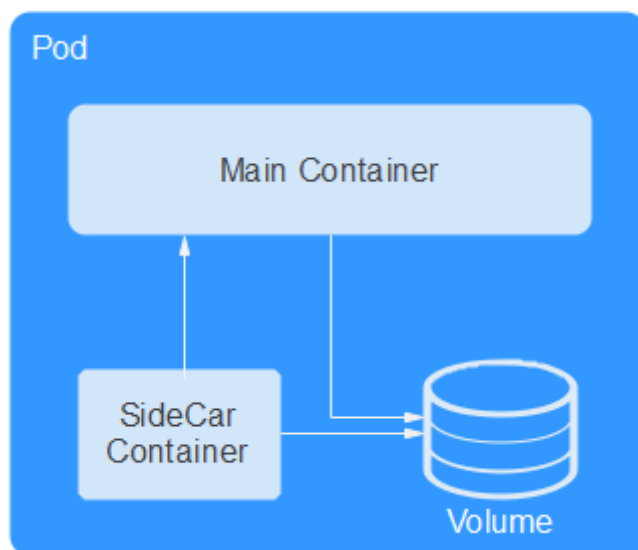
### 什么是 Pod

Pod是Kubernetes创建或部署的最小单位。一个Pod封装一个或多个容器（container）、存储资源（volume）、一个独立的网络IP以及管理控制容器运行方式的策略选项。

Pod使用主要分为两种方式：

- Pod中运行一个容器。这是Kubernetes最常见的用法，您可以将Pod视为单个封装的容器，但是Kubernetes是直接管理Pod而不是容器。
- Pod中运行多个需要耦合在一起工作、需要共享资源的容器。通常这种场景下是应用包含一个主容器和几个辅助容器（SideCar Container），如图4-1所示，例如主容器为一个web服务器，从一个固定目录下对外提供文件服务，而辅助的容器周期性的从外部下载文件存到这个固定目录下。

图 4-1 Pod



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Job。Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力。通常，Controller会使用Pod Template来创建相应的Pod。

## 容器的规格

云容器实例支持使用GPU（必须在GPU类型[命名空间](#)下）或不使用GPU。

当前提供3种类型的Pod，包括通用计算型（通用计算型命名空间下使用）、[RDMA](#)加速型和GPU加速性（GPU型命名空间下使用）。

GPU加速型Pod提供NVIDIA Tesla V100 32G显卡和NVIDIA Tesla V100 16G显卡，具体的规格有如下所示。

- NVIDIA Tesla V100 32G显卡：
  - NVIDIA Tesla V100 32G x 1, CPU 4核, 内存32GB
  - NVIDIA Tesla V100 32G x 2, CPU 8核, 内存64GB
  - NVIDIA Tesla V100 32G x 4, CPU 16核, 内存128GB
  - NVIDIA Tesla V100 32G x 8, CPU 32核, 内存256GB
- NVIDIA Tesla V100 16G显卡：
  - NVIDIA Tesla V100 16G x 1, CPU 4核, 内存32GB
  - NVIDIA Tesla V100 16G x 2, CPU 8核, 内存64GB
  - NVIDIA Tesla V100 16G x 4, CPU 16核, 内存128GB
  - NVIDIA Tesla V100 16G x 8, CPU 32核, 内存256GB

当不使用GPU时，容器规格需满足如下要求：

- Pod的CPU取值范围为0.25核-32核，另外还可选48核和64核，且单个容器的CPU必须为0.25核的整数倍
- Pod的内存取值范围为1GB-512GB，且内存必须为1GB的整数倍
- Pod的CPU/内存配比值必须在1:2到1:8之间
- 一个Pod内最多支持5个容器，单个容器最小配置是0.25核、0.2GB，最大同容器实例的最大配置

## 创建 Pod

kubernetes种资源可以使用YAML描述（如果您对YAML格式不了解，可以参考[YAML语法](#)），也可以使用JSON，如下示例描述了一个名为nginx的Pod，这个Pod中包含一个名为container-0的容器，使用nginx:alpine镜像，使用的资源为0.5核CPU、1024M内存。

```
apiVersion: v1          # Kubernetes的API Version
kind: Pod               # Kubernetes的资源类型
metadata:
  name: nginx           # Pod的名称
spec:                  # Pod的具体规格 ( specification )
  containers:
  - image: nginx:alpine # 使用的镜像为 nginx:alpine
    name: container-0   # 容器的名称
    resources:          # 申请容器所需的资源，云容器实例中limits与requests的值必须相同
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
```

```
memory: 1024Mi
imagePullSecrets: # 拉取镜像使用的证书，必须为imagepull-secret
- name: imagepull-secret
```

如上面YAML的注释，YAML描述文件主要为如下部分：

- **metadata**：一些名称/标签/namespace等信息
- **spec**：Pod实际的配置信息，包括使用什么镜像，volume等

如果去查询Kubernetes的资源，您会看到还有一个**status**字段，status描述kubernetes资源的实际状态，创建时不需要配置。这个示例是一个最小集，其他参数定义后面会逐步介绍。

kubernetes资源的参数定义的解释，您可以通过具体资源的[API参考](#)查询对应的解释。

Pod定义好后就可以使用kubectl创建，如果上面YAML文件名称为nginx.yaml，则创建命令如下所示，-f 表示使用文件方式创建。

```
$ kubectl create -f nginx.yaml -n $namespace_name
pod/nginx created
```

## 使用 GPU

云容器实例支持使用GPU（必须在GPU类型[命名空间](#)下），申请GPU资源的方法非常简单，只需要在容器定制中申请GPU字段即可。

GPU加速型Pod提供NVIDIA Tesla V100 32G显卡和NVIDIA Tesla V100 16G显卡，具体的规格有如下所示。

- NVIDIA Tesla V100 32G显卡：
  - NVIDIA Tesla V100 32G x 1，CPU 4核，内存32GB
  - NVIDIA Tesla V100 32G x 2，CPU 8核，内存64GB
  - NVIDIA Tesla V100 32G x 4，CPU 16核，内存128GB
  - NVIDIA Tesla V100 32G x 8，CPU 32核，内存256GB
- NVIDIA Tesla V100 16G显卡：
  - NVIDIA Tesla V100 16G x 1，CPU 4核，内存32GB
  - NVIDIA Tesla V100 16G x 2，CPU 8核，内存64GB
  - NVIDIA Tesla V100 16G x 4，CPU 16核，内存128GB
  - NVIDIA Tesla V100 16G x 8，CPU 32核，内存256GB

当前支持使用Nvidia GPU的驱动版本为**410.104**和**418.126**，您应用程序中使用的CUDA需满足如[表4-1](#)所示的配套关系。CUDA与驱动的配套关系来源于Nvidia官网，详细信息请参见[CUDA Compatibility](#)。

**表 4-1** NVIDIA GPU 驱动与 CUDA 配套关系

NVIDIA GPU驱动版本	CUDA Toolkit版本
410.104	CUDA 10.0 (10.0.130)及以下
418.126	CUDA 10.1 (10.1.105)及以下

您需要设置Pod的metadata.annotations中添加cri.cci.io/gpu-driver字段，指定使用哪个版本显卡驱动，取值如下：

- gpu-410.104
- gpu-418.126

如下示例创建一个容器规格为NVIDIA V100 16G x 1，CPU 4核，内存32GB的Pod。

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-test
  annotations:
    cri.cci.io/gpu-driver: gpu-418.126      # 指定GPU显卡的驱动版本
spec:
  containers:
  - image: tensorflow:latest
    name: container-0
    resources:
      limits:
        cpu: 4000m
        memory: 32Gi
        nvidia.com/gpu-tesla-v100-16GB: 1   # 申请GPU资源，支持 1、2、4、8，代表几块显卡
      requests:
        cpu: 4000m
        memory: 32Gi
        nvidia.com/gpu-tesla-v100-16GB: 1
  imagePullSecrets:
  - name: imagepull-secret
```

GPU加速型Pod提供了两种规格的显卡，其含义如下：

- nvidia.com/gpu-tesla-v100-16GB：表示NVIDIA Tesla V100 16G显卡。
- nvidia.com/gpu-tesla-v100-32GB：表示NVIDIA Tesla V100 32G显卡。

## 容器镜像

容器镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

华为云[容器镜像服务](#)从镜像中心同步了部分常用镜像，使得您可以在内部网络中直接使用“镜像名称:版本号”，如上述示例中的nginx:alpine，您可以在容器镜像服务控制台中查询同步了哪些镜像。

## 查询 Pod 详情

Pod创建完成后，可以使用kubectl get pods命令查询Pod的状态，如下所示。

```
$ kubectl get pods -n $namespace_name
NAME          READY  STATUS   RESTARTS  AGE
nginx         1/1    Running  0          40s
```

可以看到此处nginx这个Pod的状态为Running，表示正在运行；READY为1/1，表示这个Pod中有1个容器，其中1个容器的状态为Ready。

可以使用kubectl get命令查询具体Pod的配置信息，如下所示，-o yaml表示以YAML格式返回，还可以使用 -o json，以JSON格式返回。

```
$ kubectl get pod nginx -o yaml -n $namespace_name
```

您还可以使用kubectl describe命令查看Pod的详情。

```
$ kubectl describe pod nginx -n $namespace_name
```

## 删除 Pod

删除pod时，Kubernetes终止Pod中所有容器。Kubernetes向进程发送SIGTERM信号并等待一定的秒数（默认为30）让容器正常关闭。如果它没有在这个时间内关闭，Kubernetes会发送一个SIGKILL信号杀死该进程。

Pod的停止与删除有多种方法，比如按名称删除，如下所示。

```
$ kubectl delete po nginx -n $namespace_name
pod "nginx" deleted
```

同时删除多个Pod。

```
$ kubectl delete po pod1 pod2 -n $namespace_name
```

删除所有Pod。

```
$ kubectl delete po --all -n $namespace_name
pod "nginx" deleted
```

根据Label删除Pod，[Label](#)详细内容将会在下一个章节介绍。

```
$ kubectl delete po -l app=nginx -n $namespace_name
pod "nginx" deleted
```

## 4.2 环境变量

环境变量是容器运行环境中设定的一个变量。

环境变量为应用提供极大的灵活性，您可以在应用程序中使用环境变量，在创建容器时为环境变量赋值，容器运行时读取环境变量的值，从而做到灵活的配置，而不是每次都重新编写应用程序制作镜像。

另外，您还可以使用ConfigMap和Secret作为环境变量，详细信息请参见[使用ConfigMap和Secret提高配置灵活性](#)。

环境变量的使用方法如下所示，配置spec.containers.env字段即可。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:1
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    env:
      # 环境变量
      - name: env_key
        value: env_value
      - name: pod_name
        valueFrom:
          # 引用Pod的名称
          fieldRef:
            fieldPath: metadata.name
      - name: pod_ip
        valueFrom:
          # 引用Pod的IP地址
          fieldRef:
            fieldPath: status.podIP
```

```
imagePullSecrets:  
- name: imagepull-secret
```

## 4.3 启动命令

启动容器就是启动主进程，但有些时候，启动主进程前，需要一些准备工作。比如MySQL类的数据库，可能需要一些数据库配置、初始化的工作，这些工作要在最终的MySQL服务器运行之前解决。这些操作，可以在制作镜像时通过在Dockerfile文件中设置ENTRYPOINT或CMD来完成，如下所示的Dockerfile中设置了**ENTRYPOINT ["top", "-b"]**命令，其将会在容器启动时执行。

```
FROM ubuntu  
ENTRYPOINT ["top", "-b"]
```

调用接口时，只需配置pod的containers.command参数，该参数是list类型，第一个参数为执行命令，后面均为命令的参数。

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: Ubuntu  
spec:  
  containers:  
  - image: Ubuntu  
    name: container-0  
    resources:  
      limits:  
        cpu: 500m  
        memory: 1024Mi  
      requests:  
        cpu: 500m  
        memory: 1024Mi  
    command:          # 启动命令  
    - top  
    - "-b"  
  imagePullSecrets:  
  - name: imagepull-secret
```

## 4.4 生命周期管理

云容器实例基于Kubernetes，提供了[容器生命周期钩子](#)，在容器的生命周期的特定阶段执行调用，比如容器在停止前希望执行某项操作，就可以注册相应的钩子函数。目前提供的生命周期钩子函数如下所示。

- 启动后处理（PostStart）：负载启动后触发。
- 停止前处理（PreStop）：负载停止前触发。

调用接口时，只需配置pod的lifecycle.postStart或lifecycle.preStop参数，如下所示。

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx  
spec:  
  containers:  
  - image: nginx:latest  
    name: container-0  
    resources:  
      limits:  
        cpu: 500m  
        memory: 1024Mi  
      requests:
```



```
cpu: 500m
memory: 1024Mi
lifecycle:
  postStart:          # 启动后处理
  exec:
    command:
    - "/postStart.sh"
  preStop:           # 停止前处理
  exec:
    command:
    - "/preStop.sh"
imagePullSecrets:
- name: imagepull-secret
```

## 4.5 存活探针 ( liveness probe )

### 存活探针

Kubernetes提供了自愈的能力，具体就是能感知到容器崩溃，然后能够重启这个容器。但是有时候例如Java程序内存泄漏了，程序无法正常工作，但是JVM进程却一直运行的，对于这种应用本身业务出了问题情况，kubernetes提供了liveness probe机制，通过检测容器响应是否正常来决定是否重启，这是一种很好的健康检查机制。

毫无疑问，每个pod最好都定义liveness probe，否则Kubernetes无法感知Pod是否正常运行。

当前云容器实例支持如下两种探测机制。

- HTTP GET：向容器发送HTTP GET请求，如果probe收到2xx或3xx，说明容器是健康的。
- Exec：probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明健康。

### HTTP GET

HTTP GET方式是最常见的探测方法，其具体机制是向容器发送HTTP GET请求，如果probe收到2xx或3xx，说明容器是健康的，定义方法如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:      # liveness probe
      httpGet:         # HTTP GET定义
        path: /healthz
        port: 8080
```

创建这个Pod。

```
$ kubectl create -f liveness-http.yaml -n $namespace_name
pod/liveness-http created
```

如上，这个probe往容器的8080端口发送HTTP GET请求，上面的程序会在第五次请求会返回500状态码，这时Kubernetes会去重启容器。

查看Pod详情。

```
$ kubectl describe po liveness-http -n $namespace_name
Name:          liveness-http
.....
Containers:
  container-0:
    .....
    State:      Running
      Started:   Mon, 12 Nov 2018 22:57:28 +0800
    Last State: Terminated
      Reason:    Error
      Exit Code: 137
      Started:   Mon, 12 Nov 2018 22:55:40 +0800
      Finished: Mon, 12 Nov 2018 22:57:27 +0800
    Ready:      True
    Restart Count: 1
    Liveness:   http-get http://:8080/ delay=0s timeout=1s period=10s #success=1 #failure=3
    .....
Events:
  Type    Reason      Age          From          Message
  ----    -
  Normal  Scheduled   3m5s        default-scheduler  Successfully assigned default/pod-liveness to node2
  Normal  Pulling     74s (x2 over 3m4s)  kubelet, node2    pulling image "pod-liveness"
  Normal  Killing     74s          kubelet, node2    Killing container with id docker://container-0:Container failed liveness probe.. Container will be killed and recreated.
```

可以看到Pod当前状态是Running，Last State是Terminated，Restart Count为1，说明已经重启1次，另外从事件中也可以看到 Killing container with id docker://container-0:Container failed liveness probe.. Container will be killed and recreated.

另外，容器Kill后会重新创建一个新容器，不只是之前的容器重启。

## Exec

Exec即执行具体命令，具体机制是probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明健康，定义方法如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 60
    livenessProbe:
      # liveness probe
      exec:
        # Exec定义
        command:
        - cat
        - /tmp/healthy
```

上面定义在容器中执行cat /tmp/healthy命令，如果成功执行并返回0，则说明容器是健康的。

## Liveness Probe 高级配置

上面liveness-http的describe命令回显中有如下行。

```
Liveness: http-get http://:8080/ delay=0s timeout=1s period=10s #success=1 #failure=3
```

这一行表示liveness probe的具体参数配置，其含义如下：

- delay=0s 表示容器启动后立即开始探测，没有延迟时间
- timeout=1s 表示容器必须在1s内做出相应反馈给probe，否则视为探测失败
- period=10s 表示每10s探测一次
- #success=1 表示探测连续1次成功表示成功
- #failure=3 表示探测连续3次失败后会重启容器

这些是创建时默认设置的，您也可以手动配置，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - image: k8s.gcr.io/liveness
    livenessProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 10 # 容器启动后多久开始探测
      timeoutSeconds: 2 # 表示容器必须在2s内做出相应反馈给probe，否则视为探测失败
      periodSeconds: 30 # 探测周期，每30s探测一次
      successThreshold: 1 # 连续探测1次成功表示成功
      failureThreshold: 3 # 连续探测3次失败表示失败
```

initialDelaySeconds一般要设置大于0，这是由于很多情况下容器虽然启动成功，但应用就绪也需要一定的时间，需要等就绪时间之后才能返回成功，否则就会导致probe经常失败。

另外failureThreshold可以设置多次循环探测，这样在实际应用中健康检查的程序就不需要多次循环，这一点在开发应用时需要注意。

## 配置有效的 Liveness Probe

- **liveness probe应该检查什么**

一个好的liveness probe应该检查应用内部所有关键部分是否健康，并使用一个专用的URL访问，例如 /health，当访问 /health 时执行这个功能，然后返回对应结果。这里要注意不能做鉴权，不然 probe 就会一直失败导致陷入重启的死循环。

另外检查只能限制在应用内部，不能检查依赖外部的部分，例如当前端web server不能连接数据库时，这个就不能看成web server不健康。

- **liveness probe必须轻量**

liveness probe不能占用过多的资源，且不能占用过长的时间，否则所有资源都在做健康检查，这就没有意义了。例如Java应用，就最好用HTTP GET方式，如果用Exec 方式，JVM启动就占用了非常多的资源。

# 5 Label

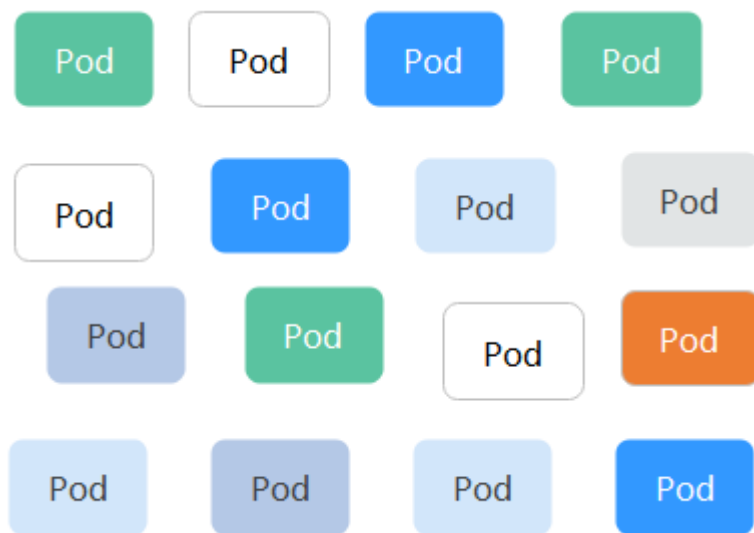
## 为什么需要 Label

当资源变得非常多的时候，如何分类管理就非常重要了，Kubernetes提供了一种机制来为资源分类，那就是Label（标签）。Label非常简单，但是却很强大，Kubernetes中几乎所有资源都可以用Label来组织。

Label的具体形式是key-value的标记对，可以在创建资源的时候设置，也可以在后期添加和修改。

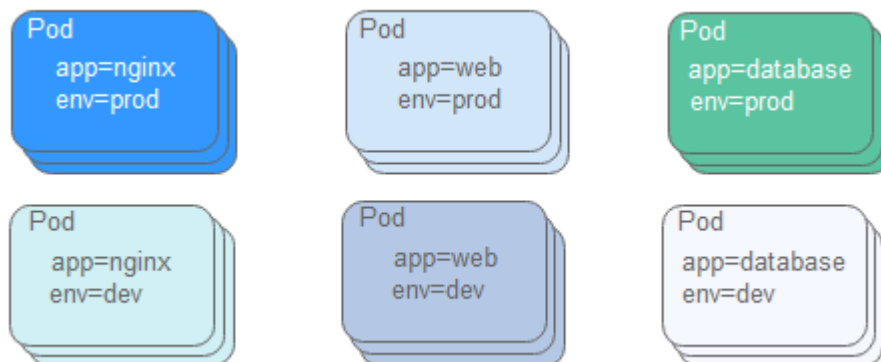
以Pod为例，当Pod变得多起来后，就显得杂乱且难以管理，如下图所示。

图 5-1 没有分类组织的 Pod



如果我们为Pod打上不同标签，那情况就完全不同了，如下图所示。

图 5-2 使用 Label 组织的 Pod



## 添加 Label

Label的形式为key-value形式，使用非常简单，如下，为Pod设置了app=nginx和env=prod两个Label。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:          # 为Pod设置两个Label
    app: nginx
    env: prod
spec:
  containers:
  - image: nginx:latest
    name: container-0
  resources:
    limits:
      cpu: 500m
      memory: 1024Mi
    requests:
      cpu: 500m
      memory: 1024Mi
  imagePullSecrets:
  - name: imagepull-secret
```

Pod有了Label后，在查询Pod的时候带上 `--show-labels` 就可以看到Pod的Label。

```
$ kubectl get pod --show-labels -n $namespace_name
NAME          READY STATUS  RESTARTS  AGE  LABELS
nginx         1/1   Running  0         50s  app=nginx,env=prod
```

还可以使用 `-L` 只查询某些固定的Label。

```
$ kubectl get pod -L app,env -n $namespace_name
NAME          READY STATUS  RESTARTS  AGE  APP  ENV
nginx         1/1   Running  0         1m  nginx  prod
```

对已存在的Pod，可以直接使用 `kubectl label` 命令直接添加Label。

```
$ kubectl label po nginx creation_method=manual -n $namespace_name
pod "nginx" labeled

$ kubectl get pod --show-labels -n $namespace_name
NAME          READY STATUS  RESTARTS  AGE  LABELS
nginx         1/1   Running  0         50s  app=nginx,env=prod,creation_method=manual
```

## 修改 Label

对于已存在的Label，如果要修改的话，需要在命令中带上--overwrite，如下所示。

```
$ kubectl label po nginx env=debug --overwrite -n $namespace_name
pod "nginx" labeled

$ kubectl get pod --show-labels -n $namespace_name
NAME          READY STATUS  RESTARTS  AGE  LABELS
nginx         1/1   Running  0         50s  app=nginx,env=debug,creation_method=manual
```

# 6 Deployment

在Pod这个章节介绍了Pod，Pod是Kubernetes创建或部署的最小单位，但是Pod是被设计为相对短暂的一次性实体，Pod可以被驱逐（当节点资源不足时）、随着集群的节点fail而消失。同时kubernetes提供了Controller（控制器）来管理Pod，Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力，其中最为常用的就是Deployment。

一个Deployment可以包含一个或多个Pod副本，每个Pod副本的角色相同，所以系统会自动为Deployment的多个Pod副本分发请求。

Deployment集成了上线部署、滚动升级、创建副本，恢复上线任务，在某种程度上，Deployment可以帮我们实现无人值守的上线，大大降低我们的上线过程的复杂沟通、操作风险。

## 创建 Deployment

以下示例为创建一个名为nginx的Deployment负载，使用nginx:latest镜像创建两个Pod，每个Pod占用500m core CPU、1G内存。

```
apiVersion: apps/v1 # 注意这里与Pod的区别，Deployment是apps/v1而不是v1
kind: Deployment # 资源类型为Deployment
metadata:
  name: nginx # Deployment的名称
spec:
  replicas: 2 # Pod的数量，Deployment会确保一直有2个Pod运行
  selector: # Label Selector
    matchLabels:
      app: nginx
  template: # Pod的定义，用于创建Pod，也称为Pod template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:latest
          name: container-0
      resources:
        limits:
          cpu: 500m
          memory: 1024Mi
        requests:
          cpu: 500m
          memory: 1024Mi
      imagePullSecrets: # 拉取镜像使用的证书，必须为imagepull-secret
        - name: imagepull-secret
```

从这个定义中可以看到Deployment的名称为nginx，spec.replicas定义了Pod的数量，即这个Deployment控制2个Pod；spec.selector是Label Selector（标签选择器），表示这个Deployment会选择Label为app=nginx的Pod；spec.template是Pod的定义，内容与Pod中的定义完全一致。

将上面Deployment的定义保存到deployment.yaml文件中，使用kubectl创建这个Deployment。

使用kubectl get查看Deployment和Pod，可以看到DESIRED值为2，这表示这个Deployment期望有2个Pod，CURRENT也为2，这表示当前有2个Pod，AVAILABLE为2表示有2个Pod是可用的。

```
$ kubectl create -f deployment.yaml -n $namespace_name

$ kubectl get deployment -n $namespace_name
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
nginx     2        2        2            2           8s
```

## Deployment 如何控制 Pod

继续查询Pod，如下所示。

```
$ kubectl get pods -n $namespace_name
NAME                READY  STATUS   RESTARTS  AGE
nginx-7f98958cdf-tdmqk 1/1    Running  0         13s
nginx-7f98958cdf-txckx 1/1    Running  0         13s
```

如果删掉一个Pod，您会发现立马会有一个新的Pod被创建出来，如下所示，这就是前面所说的Deployment会确保有2个Pod在运行，如果删掉一个，Deployment会重新创建一个，如果某个Pod崩溃或有什么问题，Deployment会自动拉起这个Pod。

```
$ kubectl delete pod nginx-7f98958cdf-txckx -n $namespace_name

$ kubectl get pods -n $namespace_name
NAME                READY  STATUS   RESTARTS  AGE
nginx-7f98958cdf-tdmqk 1/1    Running  0         21s
nginx-7f98958cdf-tesqr 1/1    Running  0         21s
```

看到有如下两个名为nginx-7f98958cdf-tdmqk和nginx-7f98958cdf-tesqr的Pod，其中nginx是直接使用Deployment的名称，-7f98958cdf-tdmqk和-7f98958cdf-tesqr是kubernetes随机生成的后缀。

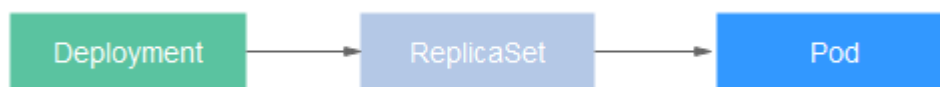
您也许会发现这两个后缀中前面一部分是相同的，都是7f98958cdf，这是因为Deployment不是直接控制Pod的，Deployment是通过一种名为ReplicaSet的控制器控制Pod，通过如下命令可以查询ReplicaSet，其中rs是ReplicaSet的缩写。

```
$ kubectl get rs -n $namespace_name
NAME                DESIRED  CURRENT  READY  AGE
nginx-7f98958cdf   3        3        3      1m
```

这个ReplicaSet的名称为nginx-7f98958cdf，后缀-7f98958cdf也是随机生成的。

Deployment控制Pod的方式如图6-1所示，Deployment控制ReplicaSet，ReplicaSet控制Pod。

图 6-1 Deployment 通过 ReplicaSet 控制 Pod



如果使用kubectl describe命令查看Deployment的详情，您就可以看到ReplicaSet，如下所示，可以看到有一行 NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)，



而且 Events 里面事件确是把ReplicaSet的实例扩容到2个。在实际使用中您也许不会直接操作ReplicaSet，但了解Deployment通过控制ReplicaSet来控制Pod会有助于您定位问题。

```
$ kubectl describe deploy nginx -n $namespace_name
Name:          nginx
Namespace:    default
CreationTimestamp: Sun, 16 Dec 2018 19:21:58 +0800
Labels:       app=nginx
...

NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)
Events:
  Type     Reason          Age    From          Message
  ----     -
  Normal   ScalingReplicaSet 5m     deployment-controller Scaled up replica set nginx-7f98958cdf to 2
```

## 升级

在实际应用中，升级是一个常见的场景，Deployment能够很方便的支撑应用升级。

Deployment 可以设置不同的升级策略，有如下两种。

- RollingUpdate：也就是滚动升级（逐步创建新Pod然后删除旧Pod），也是默认策略
- Recreate：也就是先把当前Pod删掉再重新创建Pod

Deployment的升级可以是声明式的，也就是说只需要修改Deployment的YAML定义即可，比如使用kubectl edit命令将上面Deployment中的镜像修改为nginx:alpine。修改完成后再查询ReplicaSet和Pod，发现创建了一个新的ReplicaSet，Pod也重新创建了。

```
$ kubectl edit deploy nginx -n $namespace_name

$ kubectl get rs -n $namespace_name
NAME          DESIRED  CURRENT  READY  AGE
nginx-6f9f58dff 2        2        2      1m
nginx-7f98958cdf 0        0        0      48m

$ kubectl get pods -n $namespace_name
NAME          READY  STATUS  RESTARTS  AGE
nginx-6f9f58dff-tdmqk 1/1    Running  0         21s
nginx-6f9f58dff-tesqr 1/1    Running  0         21s
```

Deployment可以通过maxSurge 和 maxUnavailable两个参数控制升级过程中同时重新创建Pod的比例，这在很多时候是非常有用，配置如下所示。

```
spec:
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```

- maxSurge：与Deployment中spec.replicas相比，可以有多少个Pod存在，默认值是25%，比如spec.replicas为 4，那升级过程中就不能超过5个 Pod存在，即按1个的步伐升级，实际升级过程中会换算成数字，且换算会向上取整。这个值也可以直接设置成数字。
- maxUnavailable：与Deployment中spec.replicas相比，可以有多少个Pod失效，也就是删除的比例，默认值是25%，比如spec.replicas为 4，那升级过程中就至少有3个Pod存在，即删除Pod 的步伐是 1。同样这个值也可以设置成数字。

在前面的例子中，由于spec.replicas是2，如果maxSurge和maxUnavailable都为默认值25%，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（向上取整，2\*0.75=1.5，取整为2），也就是说在升级过程中，一直会有2个Pod处于运行状态，每次新建一个Pod，等这个Pod创建成功后再删掉一个旧Pod，直至Pod全部为新Pod。

## 回滚

回滚也称为回退，即当发现升级出现问题时，让应用回到老的版本。Deployment可以非常方便的回滚到老版本。

例如上面升级的新版镜像有问题，可以执行kubectl rollout undo命令进行回滚。

```
$ kubectl rollout undo deployment nginx -n $namespace_name  
deployment "nginx" rolled back
```

Deployment之所以能如此容易的做到回滚，是因为Deployment是通过ReplicaSet控制Pod的，升级后之前ReplicaSet都一直存在，Deployment回滚做的就是使用之前的ReplicaSet再次把Pod创建出来。Deployment中保存ReplicaSet的数量可以使用revisionHistoryLimit参数限制，默认值为10。

# 7 使用 Service 和 Ingress 管理网络访问

---

## 7.1 Service

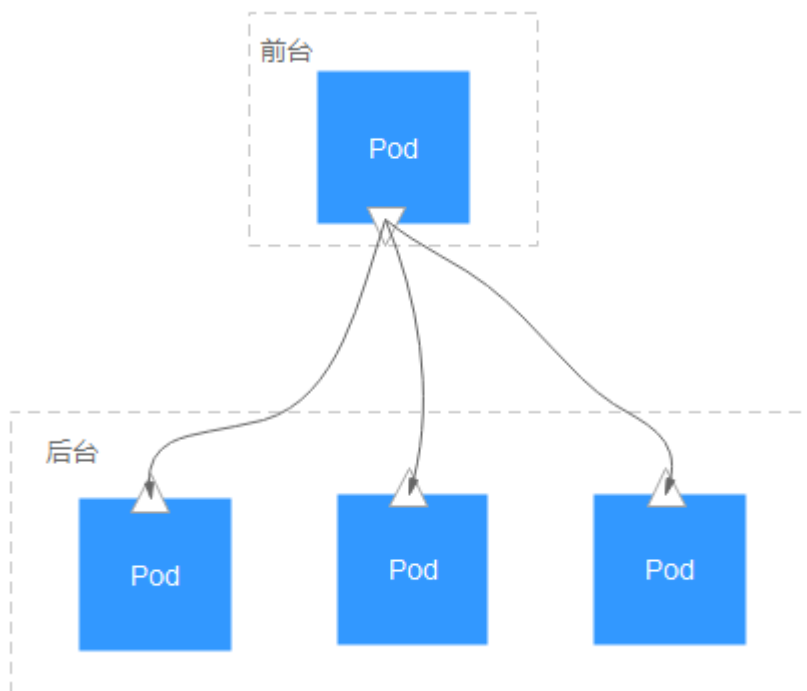
### 直接访问 Pod 的问题

负载创建完成后，如何访问负载呢？访问负载实际上就是访问Pod，但是直接访问Pod会有如下几个问题：

- Pod会随时被Deployment这样的控制器删除重建，那访问Pod的结果就会变得不可预知。
- Pod的IP地址是在Pod启动后才被分配，在启动前并不知道Pod的IP地址。
- 应用往往都是由多个运行相同镜像的一组Pod组成，一个个Pod的访问也变得不现实。

举个例子，假设有这样一个应用程序，使用Deployment创建了前台和后台，前台会调用后台做一些计算处理，如图7-1所示。后台运行了3个Pod，这些Pod是相互独立且可被替换的，当Pod出现状况被重建时，新建的Pod的IP地址是新IP，前台的Pod无法直接感知。

图 7-1 负载间访问

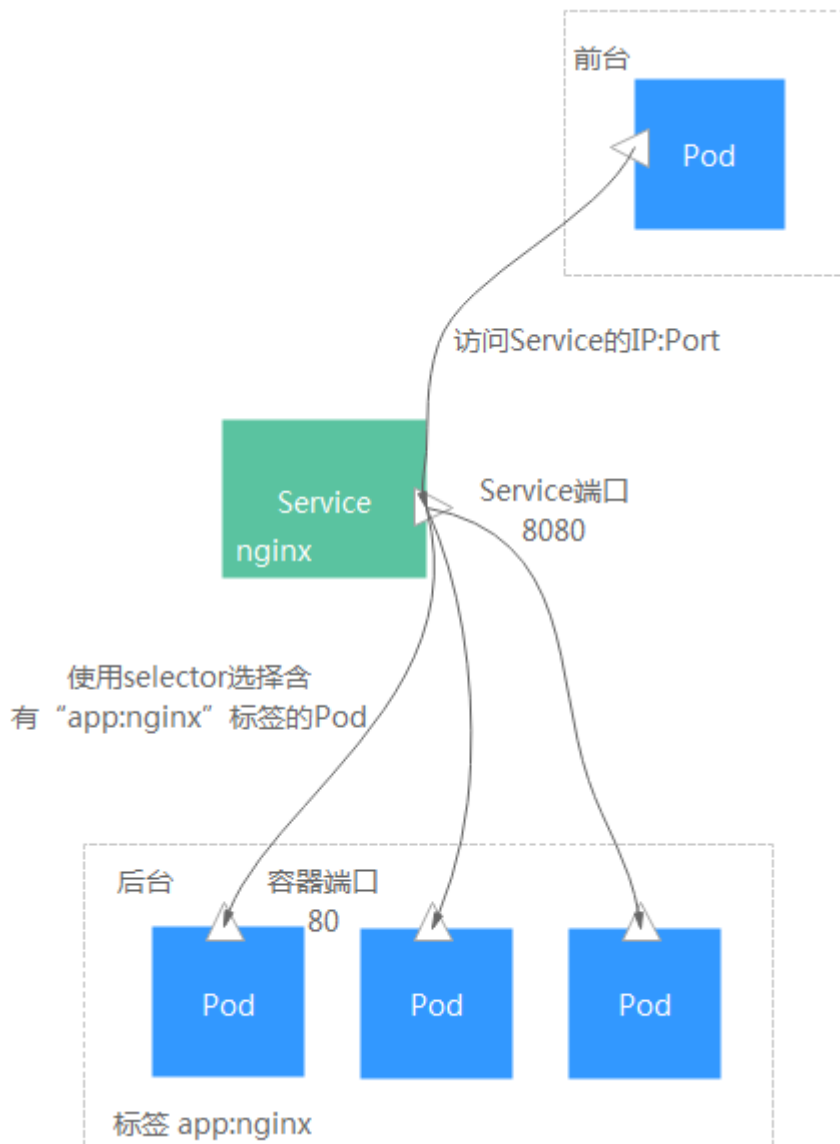


## Service 是如何工作的

Kubernetes中的Service对象就是用来解决上述Pod访问问题的。Service有一个固定IP地址，Service将访问他的流量转发给Pod，具体转发给哪些Pod通过Label来选择，而且Service可以给这些Pod做负载均衡。

那么对于上面的例子，通过为前后台添加两个Service，通过Service来访问Pod，这样前台Pod就无需感知后台Pod的变化，如图7-2所示。

图 7-2 通过 Service 访问 Pod



## 创建 Service

下面示例创建一个名为“nginx”的Service，通过selector选择到标签“app:nginx”的Pod，目标Pod的端口为80，Service对外暴露的端口为8080。

访问服务只需要通过“服务名称:对外暴露的端口”接口，对应本例即“nginx:8080”。这样，在其他负载中，只需要通过“nginx:8080”就可以访问到“nginx”关联的Pod。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx # Service的名称
spec:
  selector: # Label Selector, 选择包含app=nginx标签的Pod
    app: nginx
  ports:
    - name: service0
      targetPort: 80 # Pod的端口
```

```
port: 8080 # Service对外暴露的端口
protocol: TCP # 转发协议类型, 支持TCP和UDP
type: ClusterIP # Service的类型
```

### 📖 说明

在原生kubernetes中, Service还有NodePort类型, 当前云容器实例不支持使用NodePort类型Service。

将上面Service的定义保存到nginx-svc.yaml文件中, 使用kubectl创建这个Service。

```
# kubectl create -f nginx-svc.yaml -n $namespace_name
service/nginx created

# kubectl get svc -n $namespace_name
NAME      TYPE      CLUSTER-IP    EXTERNAL-IP  PORT(S)        AGE
kube-dns  ClusterIP 10.247.9.190  <none>       53/UDP,53/TCP 7m
nginx     ClusterIP 10.247.148.137 <none>       8080/TCP       1h
```

您可以看到Service有个Cluster IP, 这个IP是固定不变的, 除非Service被删除, 所以您也可以使用ClusterIP在内部访问Service。

### 📖 说明

kube-dns是预留给域名解析使用的Service, 云容器实例会自动创建, 域名解析的详细内容请参见[使用ServiceName访问Service](#)。

## 使用 ServiceName 访问 Service

云容器实例中您可以使用[CoreDNS](#)插件给Service做域名解析, 然后使用“ServiceName:Port”访问Service, 这也是Kubernetes中最常用的一种使用方式, CoreDNS的安装请参见[插件管理](#)。

CoreDNS安装成功后会成为DNS服务器, 当创建Service后, CoreDNS会将Service的名称与IP记录起来, 这样Pod就可以通过向CoreDNS查询Service的名称获得Service的IP地址。

访问时通过nginx.<namespace>.svc.cluster.local访问, 其中nginx为Service的名称, <namespace>为命名空间名称, svc.cluster.local为域名后缀, 在实际使用中, 可以省略<namespace>.svc.cluster.local, 直接使用Service的名称即可。

例如上面创建的名为nginx的Service, 直接通过“nginx:8080”就可以访问到Service, 进而访问后台Pod。

使用ServiceName的方式有个主要的优点就是可以在开发应用程序时可以将ServiceName写在程序中, 这样无需感知具体Service的IP地址。

### 须知

CoreDNS插件占用计算资源, 共2个Pod, 每个Pod占用0.5核1G的资源, 您需要为此付费。

## LoadBalancer 类型的 Service

现在您知道可以创建ClusterIP类型的Service, 通过Service的IP可以访问到Service后端的Pod。

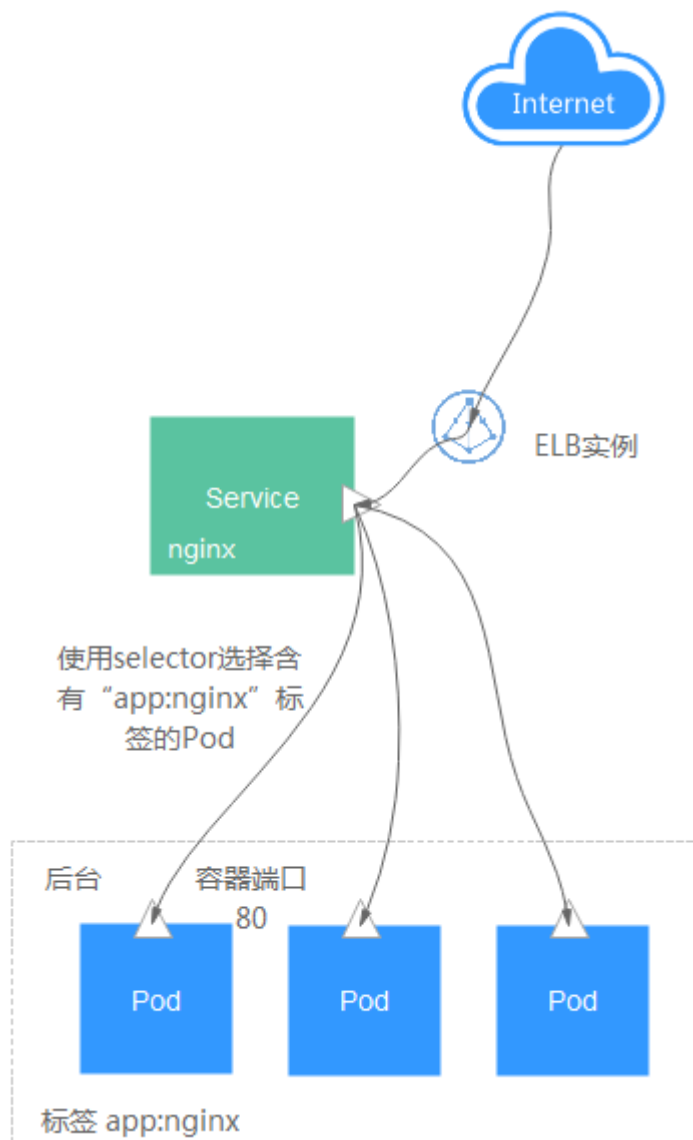
云容器实例同时还支持创建LoadBalancer类型的Service, 将[增强型ELB实例](#)与Service绑定, 这样访问ELB实例的流量就会访问到Service。

ELB实例根据IP地址不同可以分为私网ELB实例和公网ELB实例，区别在于公网ELB实例绑定了一个公网IP，您可以根据需要选择。您可以调用[创建负载均衡器（增强型）](#)创建ELB实例，更方便的方法是通过[ELB控制台](#)创建增强型ELB实例。

#### 说明

ELB实例必须与Service在同一个VPC内，否则无法绑定。

图 7-3 LoadBalancer Service



下面是一个创建LoadBalancer类型的Service。创建完成后，可以通过ELB的IP:Port访问到后端Pod。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    kubernetes.io/elb.id: 77e6246c-a091-xxxx-xxxx-789baa571280 # ELB的ID
spec:
  selector:
```

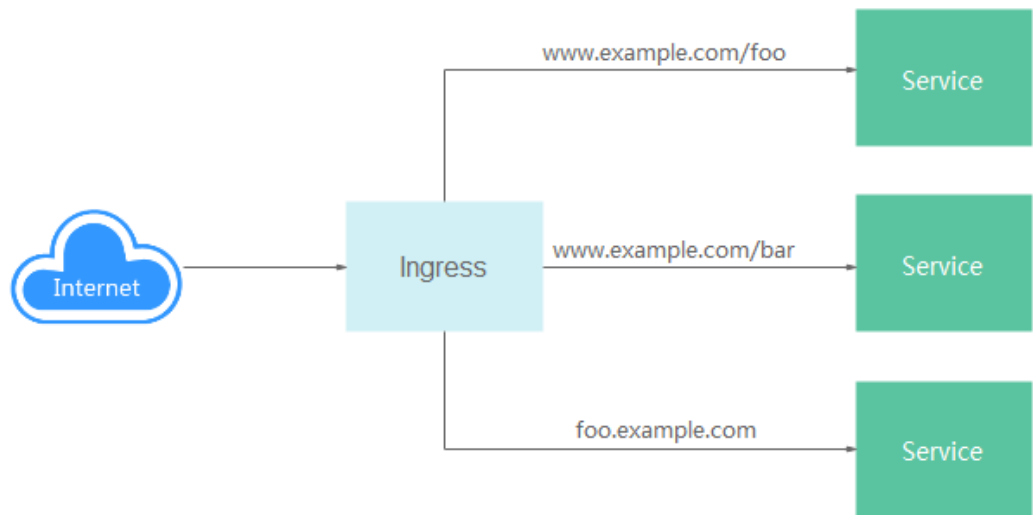
```
app: nginx
ports:
- name: service0
  targetPort: 80
  port: 8080 # ELB访问端口
  protocol: TCP
type: LoadBalancer # Service的类型
```

## 7.2 Ingress

上一节中讲了创建LoadBalancer类型的Service，使用ELB实例从外部访问Pod。

但是Service是基于四层TCP和UDP协议转发的，Ingress可以基于七层的HTTP和HTTPS协议转发，可以通过域名和路径做到更细粒度的划分，如下图所示。

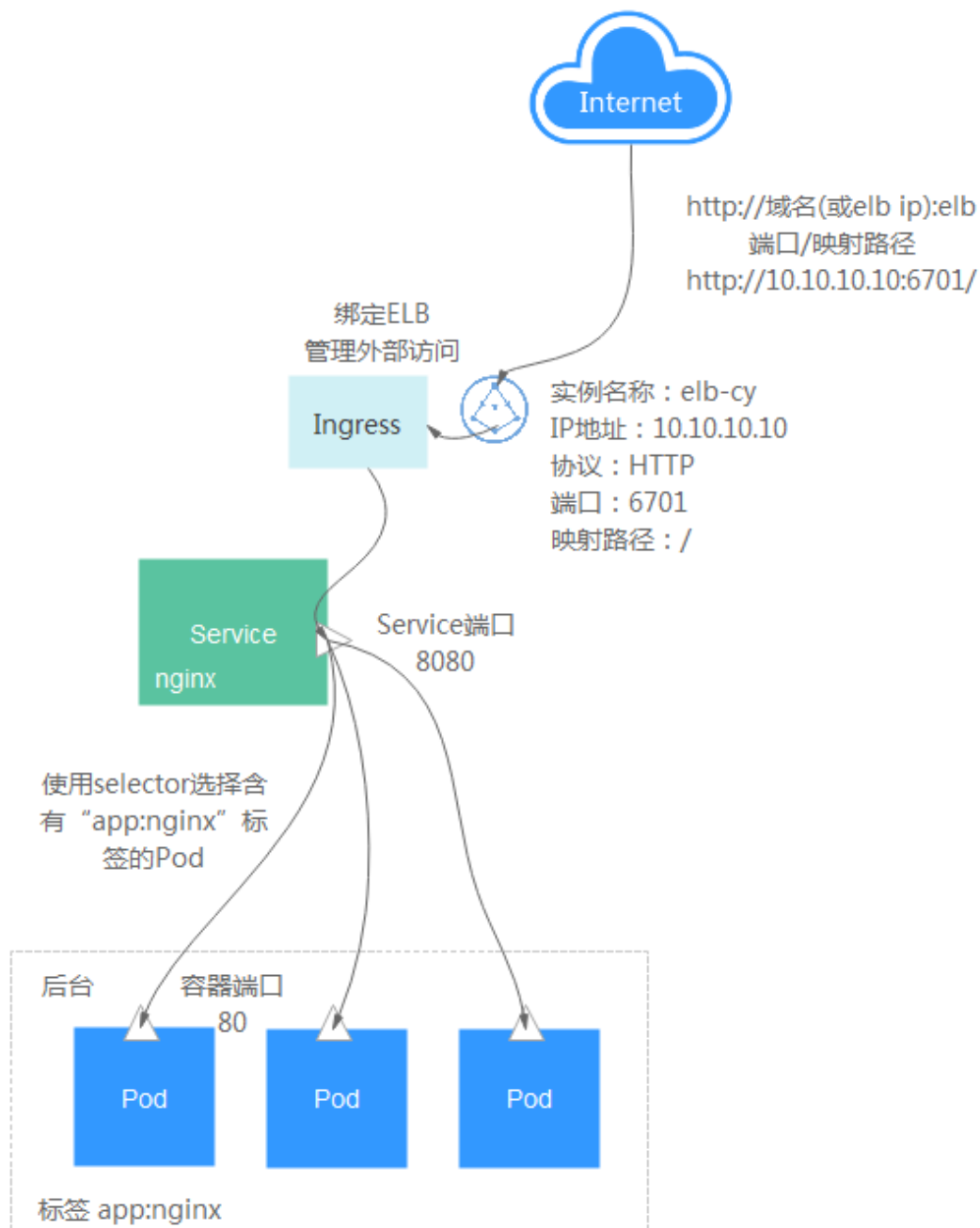
图 7-4 Ingress-Service



在云容器实例中，使用Ingress绑定ELB的IP和端口，实现外部访问，如图7-5所示。



图 7-5 Ingress



## ELB 实例

Ingress支持绑定增强型ELB实例，您可以调用[创建负载均衡器（增强型）](#)创建ELB实例，更方便的方法是通过[ELB控制台](#)创建增强型ELB实例。

ELB实例根据IP地址不同可以分为私网ELB实例和公网ELB实例，区别在于公网ELB实例绑定了一个公网IP，您可以根据需要选择。

## 创建 Ingress

- 使用http协议创建Ingress

下面例子中，关联的backend为“nginx:8080”，当访问“http://10.10.10.10:6071/”时，流量转发“nginx:8080”对应的Service，从而放到对应负载中的Pod。

```
apiVersion: extensions/v1beta1 # Ingress的版本
kind: Ingress
metadata:
  name: nginx
  labels:
    app: nginx
    isExternal: "true" # 必选参数，取值必须为 "true"
    zone: data # 数据平面模式，必选参数，取值必须为data
  annotations:
    kubernetes.io/elb.id: 2d48d034-6046-48db-8bb2-53c67e8148b5 # ELB实例的ID，必选参数
    kubernetes.io/elb.ip: 192.168.137.182 # ELB实例的IP，可选参数
    kubernetes.io/elb.port: '6071' # ELB实例的端口，必选参数
spec:
  rules: # 路由规则
  - http: # 使用http协议
    paths:
    - path: / # 路由
      backend:
        serviceName: nginx # 转发到的Service名称
        servicePort: 8080 # 转发到的Service端口
```

Ingress中还可以设置外部域名，这样您就可以通过域名来访问到ELB，进而访问到后端服务。

### 📖 说明

域名访问依赖于域名解析，需要您将域名解析指向ELB实例的IP地址，例如您可以使用[云解析服务 DNS](#)来实现域名解析。

```
spec:
  rules:
  - host: www.example.com # 域名
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx
          servicePort: 80
```

- 使用https协议创建Ingress

下面例子中，关联的backend为“nginx:8080”，当访问“https://10.10.10.10:6071/”时，流量转发“nginx:8080”对应的Service，从而放到对应负载中的Pod。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/elb.id: 2d48d034-6046-48db-8bb2-53c67e8148b5
    kubernetes.io/elb.ip: 192.168.137.182
    kubernetes.io/elb.port: '6071'
  labels:
    app: nginx
    isExternal: 'true'
    zone: data
  name: nginx
spec:
  rules:
  - http:
    paths:
    - backend:
        serviceName: nginx
        servicePort: 8080
      path: /
  tls:
  - secretName: cci-sslcertificate-20214221 # 上传的SSL证书的名称
```

## 路由到多个服务

Ingress可以同时路由到多个服务，配置如下所示。

- 当访问“http://foo.bar.com/foo”时，访问的是“s1:80”后端。
- 当访问“http://foo.bar.com/bar”时，访问的是“s2:80”后端。

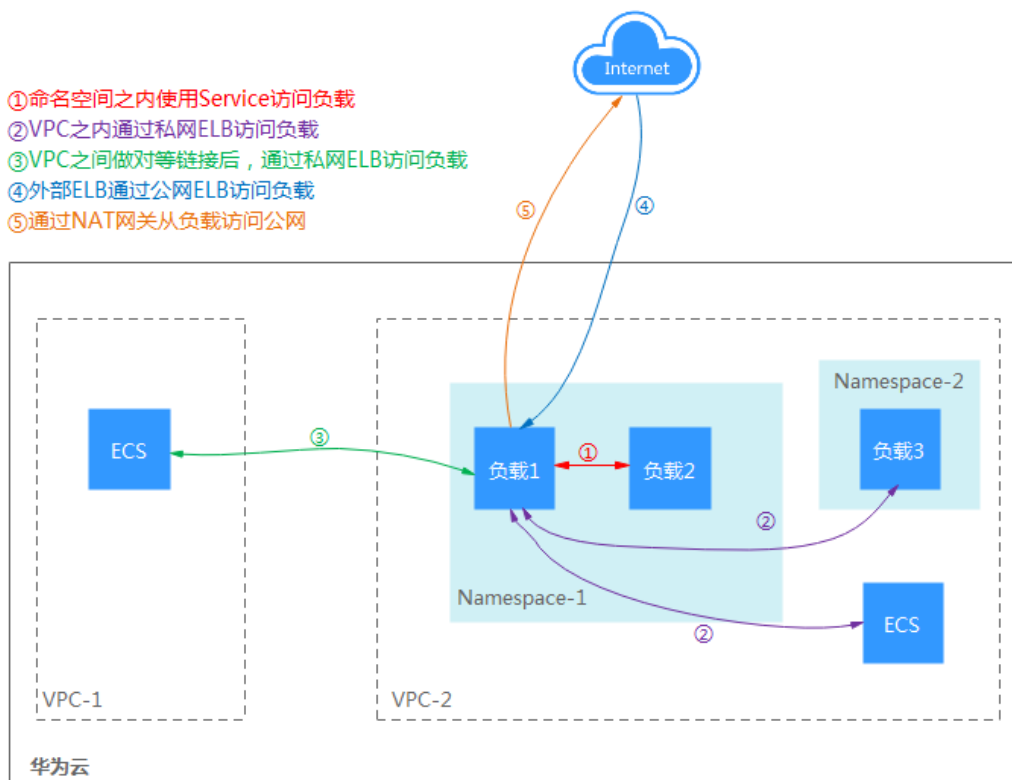
```
spec:
  rules:
  - host: foo.bar.com      # host地址
    http:
      paths:
      - path: "/foo"
        backend:
          serviceName: s1
          servicePort: 80
      - path: "/bar"
        backend:
          serviceName: s2
          servicePort: 80
```

## 7.3 网络访问场景

在前面两节中介绍了如何通过Service和Ingress访问Pod，本节总结一下云容器实例中Pod的访问场景，如图7-6所示，访问负载可以分为如下几种场景，每种场景下可以使用Service和Ingress来解决访问问题。

1. 同一个命名空间中的负载相互访问：只需创建Service，使用“服务名称:服务端口”访问负载。
2. 同一个VPC内资源互相访问：直接访问Service的IP地址，或者通过Ingress绑定私网ELB，使用私网ELB的IP访问负载。
3. 不同VPC内资源互相访问：不同VPC内，可以选择创建VPC对等连接，使得两个VPC之间网络互通，在访问时通过Service的IP地址或私网ELB访问负载。
4. 从公网访问负载：从华为云外部访问负载，需要通过Ingress绑定公网ELB，通过ELB的IP访问负载。
5. 从负载中访问公网：通过在NAT网关服务中配置SNAT规则，使得容器能够访问公网，具体配置方法请参见从容器访问公网。

图 7-6 网络访问示意图



## 7.4 业务探针（Readiness probe）

一个新Pod创建后，Service就能立即选择到它，并会把请求转发给Pod，那问题就来了，通常一个Pod启动是需要时间的，如果Pod还没准备好（可能需要时间来加载配置或数据，或者可能需要执行一个预热程序之类），这时把请求转给Pod的话，Pod也无法处理，造成请求失败。

Kubernetes中解决这个问题的方法就是给Pod加一个业务就绪探针Readiness Probe，当检测到Pod就绪后才允许Service请求转给Pod。

Readiness Probe同样是周期性的检测Pod，然后根据响应来判断Pod是否就绪，与存活探针（liveness probe）相同，云容器实例中也支持两种类型的Readiness Probe。

- HTTP GET：往容器的IP:Port发送HTTP GET请求，如果probe收到2xx或3xx，说明已经就绪。
- Exec：probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明已经就绪。

### Readiness Probe 的工作原理

如果调用kubectl describe命令查看Service的信息，您会看下如下信息。

```
$ kubectl describe svc nginx -n $namespace_name
Name:          nginx
.....
Endpoints:    192.168.113.81:80,192.168.165.64:80,192.168.198.10:80
.....
```

可以看到一个Endpoints，Endpoints同样也是Kubernetes的一种资源对象，可以查询得到。

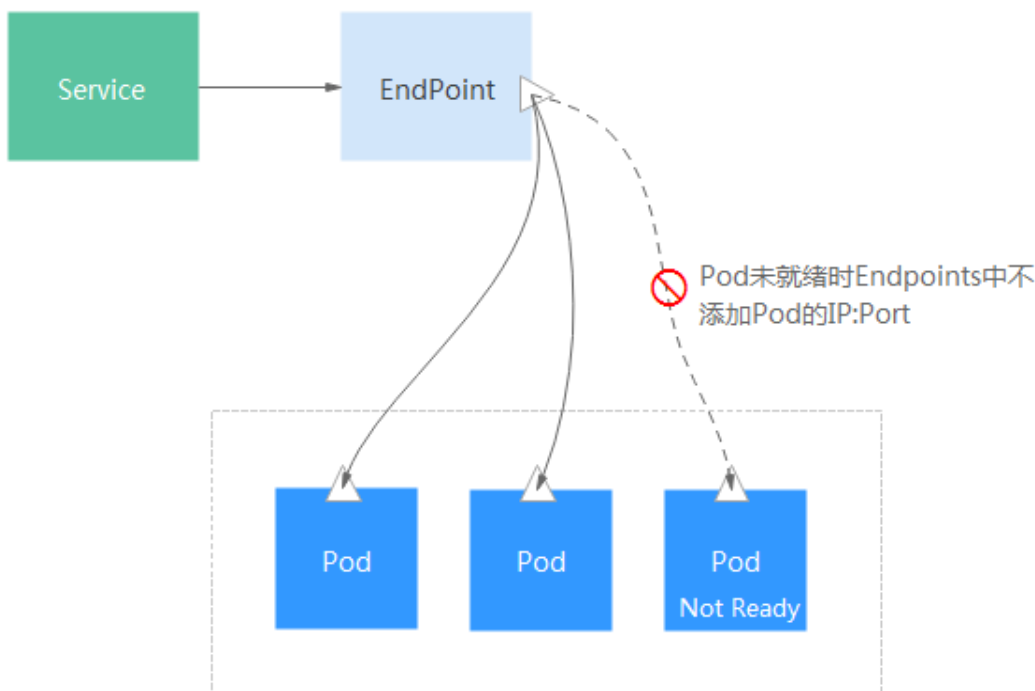
```
$ kubectl get endpoints -n $namespace_name
NAME           ENDPOINTS                                     AGE
nginx          192.168.113.81:80,192.168.165.64:80,192.168.198.10:80  14m
```

这里的192.168.113.81:80是Pod的IP:Port，通过如下命令可以查看到Pod的IP，与上面的IP一致。

```
# kubectl get pods -o wide -n $namespace_name
NAME           READY   STATUS    RESTARTS   AGE   IP
nginx-55c54cc5c7-49chn  1/1     Running   0          1m    192.168.198.10
nginx-55c54cc5c7-x87lb  1/1     Running   0          1m    192.168.165.64
nginx-55c54cc5c7-xp4c5  1/1     Running   0          1m    192.168.113.81
```

通过Endpoints就可以实现Readiness Probe的效果，当Pod还未就绪时，将Pod的IP:Port在Endpoints中删除，Pod就绪后再加入到Endpoints中，如下图所示。

图 7-7 Readiness Probe 的实现原理



## Exec

Exec方式与HTTP GET方式一致，如下所示，这个探针执行**ls /ready**命令，如果这个文件存在，则返回0，说明Pod就绪了，否则返回其他状态码。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
```

```
labels:
  app: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    readinessProbe: # Readiness Probe
      exec: # 定义 ls /ready 命令
        command:
        - ls
        - /ready
    imagePullSecrets:
    - name: imagepull-secret
```

将上面Deployment的定义保存到deploy-read.yaml文件中，删除之前创建的Deployment，用deploy-read.yaml创建这个Deployment。

```
# kubectl delete deploy nginx -n $namespace_name
deployment.apps "nginx" deleted

# kubectl create -f deploy-read.yaml -n $namespace_name
deployment.apps/nginx created
```

这里由于nginx镜像不包含 /ready 这个文件，所以在创建完成后容器不在Ready状态，如下所示，注意READY这一列的值为0/1，表示容器没有Ready。

```
# kubectl get po -n $namespace_name
NAME                READY   STATUS    RESTARTS   AGE
nginx-7955fd7786-686hp 0/1     Running   0          7s
nginx-7955fd7786-9tgwq 0/1     Running   0          7s
nginx-7955fd7786-bqsbj 0/1     Running   0          7s
```

再次查看Service，发现Endpoints一行的值为空，表示没有Endpoints。

```
$ kubectl describe svc nginx -n $namespace_name
Name:          nginx
.....
Endpoints:
.....
```

如果此时给容器中创建一个 /ready 的文件，让Readiness Probe成功，则容器会处于Ready状态。再查看Pod和Endpoints，发现创建了/ready文件的容器已经Ready，Endpoints也已经添加。

```
# kubectl exec -n $namespace_name nginx-7955fd7786-686hp -- touch /ready

# kubectl get po -o wide -n $namespace_name
NAME                READY   STATUS    RESTARTS   AGE   IP
nginx-7955fd7786-686hp 1/1     Running   0          10m   192.168.93.169
nginx-7955fd7786-9tgwq 0/1     Running   0          10m   192.168.166.130
nginx-7955fd7786-bqsbj 0/1     Running   0          10m   192.168.252.160

# kubectl get endpoints -n $namespace_name
NAME      ENDPOINTS          AGE
nginx    192.168.93.169:80 14d
```

## HTTP GET

Readiness Probe的配置与存活探针（[liveness probe](#)）一样，都是在 Pod Template 的 containers 里面，如下所示，这个Readiness Probe向Pod发送HTTP请求，当Probe收到2xx或3xx返回时，说明Pod已经就绪，这

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
      resources:
        limits:
          cpu: 500m
          memory: 1024Mi
        requests:
          cpu: 500m
          memory: 1024Mi
      readinessProbe:
        httpGet:
          path: /read
          port: 80
        # readinessProbe
        # HTTP GET定义
      imagePullSecrets:
      - name: imagepull-secret
```

## Readiness Probe 高级配置

与Liveness Probe相同，Readiness Probe也有同样的高级配置选项，上面nginx Pod的describe命令回显有中有如下行。

```
Readiness: exec [ls /var/ready] delay=0s timeout=1s period=10s #success=1 #failure=3
```

这一行表示readiness probe的具体参数配置，其含义如下：

- delay=0s 表示容器启动后立即开始探测，没有延迟时间
- timeout=1s 表示容器必须在1s内做出相应反馈给probe，否则视为探测失败
- period=10s 表示每10s探测一次
- #success=1 表示探测连续1次成功表示成功
- #failure=3 表示探测连续3次失败后会重启容器

这些是创建时默认设置的，您也可以手动配置，如下所示。

```
readinessProbe: # Readiness Probe
  exec: # 定义 ls /readiness/ready 命令
    command:
      - ls
      - /readiness/ready
  initialDelaySeconds: 10 # 容器启动后多久开始探测
  timeoutSeconds: 2 # 表示容器必须在2s内做出相应反馈给probe，否则视为探测失败
  periodSeconds: 30 # 探测周期，每30s探测一次
  successThreshold: 1 # 连续探测1次成功表示成功
  failureThreshold: 3 # 连续探测3次失败表示失败
```

# 8 使用 PersistentVolumeClaim 申请持久化存储

云容器实例当前支持在容器中使用如下三种持久化存储。

- **云硬盘**（Elastic Volume Service, EVS），EVS是一种块存储服务，提供高I/O（sas）、超高I/O（ssd）和普通I/O（sata）三种类型云硬盘。
- **弹性文件服务**（Scalable File Service, SFS），SFS提供共享的文件存储，支持标准文件协议类型（nfs-rw）。SFS提供了SFS和SFS Turbo两种类型的文件系统。
  - SFS为用户提供一个完全托管的共享文件存储，能够弹性伸缩至PB规模，具备高可用性和持久性，为海量数据、高带宽型应用提供有力支持。适用于多种应用场景，包括HPC、媒体处理、文件共享、内容管理和Web服务等。
  - SFS Turbo为用户提供一个完全托管的共享文件存储，能够弹性伸缩至320TB规模，具备高可用性和持久性，为海量的小文件、低延迟高IOPS型应用提供有力支持。适用于多种应用场景，包括高性能网站、日志存储、压缩解压、DevOps、企业办公、容器应用等。

## 📖 说明

目前“华东-上海一”区域不支持SFS Turbo。

- **对象存储服务**（Object Storage Service, OBS），OBS是一个基于对象的海量存储服务，为客户提供海量、安全、高可靠、低成本的数据存储能力。

上面三种存储中，OBS的使用方式最为直接，云容器实例当前支持直接以SDK方式使用对象存储服务（OBS）。您可以在应用程序中使用SDK方式使用OBS，将应用程序打包成容器镜像，在云容器实例中使用镜像创建负载。OBS的SDK的下载及使用方法请参见<https://developer.huaweicloud.com/sdk?OBS>。

EVS和SFS的使用有个挂载的过程，下面将介绍如何使用EVS和SFS。

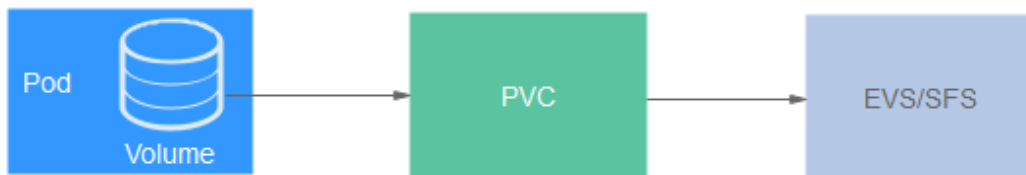
## PersistentVolumeClaim (PVC)

Kubernetes提供PVC专门用于持久化存储的申请，PVC可以让您无需关心底层存储资源如何创建、释放等动作，而只需要申明您需要何种类型的存储资源、多大的存储空间。

在实际使用中，您可以通过Pod中的Volume来关联PVC，通过PVC使用持久化存储，如图8-1所示。



图 8-1 使用持久化存储



## 创建 PVC

- 通过如下定义创建PVC，这个定义申请了一块大小为100G的SATA型云硬盘。如果需要创建加密类型的云硬盘存储卷，在metadata.annotations中增加paas.storage.io/cryptKeyId字段即可。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-eva
  namespace: namespaces-test
  annotations: {
    paas.storage.io/cryptKeyId: ee9b610c-e356-11e9-aadc-d0efc1b3bb6b
  }
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 100Gi
  storageClassName: sata
```

accessModes为存储访问模式，支持如下3种模式：

- ReadWriteOnce：可以被单个节点以读/写模式挂载
- ReadOnlyMany：可以被多个节点以只读模式挂载
- ReadWriteMany：可以被多个节点以读/写模式挂载

storageClassName表示申请的存储类型，当前支持如下4个参数：

- sata：SATA（普通I/O）型EVS硬盘
- sas：SAS（高I/O）型EVS硬盘
- ssd：SSD（超高I/O）型EVS硬盘
- nfs-rw：标准文件协议类型SFS文件存储

- 通过如下定义创建PVC，这个定义申请了一块大小为100G的文件存储。如果需要创建加密类型的文件存储卷，在metadata.annotations中增加paas.storage.io/cryptKeyId、paas.storage.io/cryptAlias和paas.storage.io/cryptDomainId即可。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-sfs
  namespace: namespace-test
  annotations: {
    paas.storage.io/cryptKeyId: ee9b610c-e356-11e9-aadc-d0efc1b3bb6b
    paas.storage.io/cryptAlias: sfs/default
    paas.storage.io/cryptDomainId: d6912480-c3d6-4e9e-8c70-38afeea434c3
    volume.beta.kubernetes.io/storage-provisioner: flexvolume-huawei.com/fuxinfs
  }
spec:
  accessModes:
    - ReadWriteMany
  resources:
```

```
requests:
  storage: 100Gi
storageClassName: nfs-rw
```

## 使用 PVC

使用PVC申请到存储资源后，您可以在Pod中使用Volume来关联PVC，并将Volume挂载到容器中使用。

下面的示例中说明了PVC如何在Pod中使用，这个Pod定义了一个名为“pvc-test-example”的Volume，并将这个Volume挂载到容器的“/tmp/volume0”路径，这样您就可以写入到/tmp的数据就是写到名为pvc-test的PVC中。

- 写入上面申请的sata型云硬盘中

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    volumeMounts:
    - mountPath: "/tmp/volume0" # 将PVC挂载到容器的/tmp/volume0路径
      name: pvc-test-example # Volume的名称
  volumes: # 定义Volume，关联PVC
  - name: pvc-test-example
    persistentVolumeClaim:
      claimName: pvc-test # PVC的名称
  imagePullSecrets:
  - name: imagepull-secret
```

- 写入上面申请的文件存储（storageClassName设置为nfs-rw型）中。  
当创建PVC申请文件存储（storageClassName设置为nfs-rw型）时，在volumeMounts中可设置挂载子路径，即文件存储根路径下子路径。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    volumeMounts:
    - mountPath: "/tmp/volume0" # 将PVC挂载到容器的/tmp/volume0路径
      subPath: "abc" # 文件存储根路径下子路径，如果不存在会自动在文件存储中创建。该子路径必须为相对路径。
      name: pvc-test-example # Volume的名称
  volumes: # 定义Volume，关联PVC
```

```
- name: pvc-test-example
  persistentVolumeClaim:
    claimName: pvc-test # PVC的名称
  imagePullSecrets:
  - name: imagepull-secret
```

# 9 使用 ConfigMap 和 Secret 提高配置灵活性

## 9.1 ConfigMap

ConfigMap是一种用于存储应用所需配置信息的资源类型，用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。

通过ConfigMap可以方便的做到配置解耦，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 ConfigMap

下面示例创建了一个名为configmap-test的ConfigMap，ConfigMap的配置数据在data字段下定义。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-test
data: # 配置数据
  property_1: Hello
  property_2: World
```

### 在环境变量中引用 ConfigMap

ConfigMap最为常见的使用方式就是在环境变量和Volume中引用。

例如下面例子中，引用了configmap-test的property\_1，将其作为环境变量EXAMPLE\_PROPERTY\_1的值，这样容器启动后里面EXAMPLE\_PROPERTY\_1的值就是property\_1的值，即“Hello”。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
  resources:
    limits:
      cpu: 500m
```

```
memory: 1024Mi
requests:
  cpu: 500m
  memory: 1024Mi
env:
- name: EXAMPLE_PROPERTY_1
  valueFrom:
    configMapKeyRef: # 引用ConfigMap
      name: configmap-test
      key: property_1
imagePullSecrets:
- name: imagepull-secret
```

## 在 Volume 中引用 ConfigMap

在Volume中引用ConfigMap，就是通过文件的方式直接将ConfigMap的每条数据填入Volume，每条数据是一个文件，键就是文件名，键值就是文件内容。

如下示例中，创建一个名为vol-configmap的Volume，这个Volume引用名为“configmap-test”的ConfigMap，再将Volume挂载到容器的“/tmp”路径下。Pod创建成功后，在容器的“/tmp”路径下，就有两个文件property\_1和property\_2，他们的值分别为“Hello”和“World”。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    volumeMounts:
    - name: vol-configmap # 挂载名为vol-configmap的Volume
      mountPath: "/tmp1"
  imagePullSecrets:
  - name: imagepull-secret
  volumes:
  - name: vol-configmap
    configMap: # 引用ConfigMap
      name: configmap-test
```

## 9.2 Secret

Secret是一种加密存储的资源对象，您可以将认证信息、证书、私钥等保存在密钥中，从而解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中，只需在容器启动时以环境变量等方式加载到容器中。

Secret与ConfigMap非常像，都是key-value键值对形式，使用方式也相同，不同的是Secret会加密存储，所以适用于存储敏感信息。

### Base64 编码

Secret与ConfigMap相同，是以键值对形式保存数据，所不同的是在创建时，Secret的Value必须使用Base64编码。

对字符串进行Base64编码，可以直接使用“echo -n 要编码的内容 | base64”命令即可，示例如下：

```
root@ubuntu:~# echo -n "3306" | base64
MzMwNg==
```

## 创建 Secret

如下示例中定义的Secret中包含两条Key-Value。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  key1:
VkZNMEOwVlpVbEpQVHpGTFdrSkRWVWhCV2s5T1ZrNUxUVIZNUjBzMFRWcElVMFpVUkVWV1N3PT0= #
Base64编码后的值
  key2: T0Vkr1RGRlZVRlpVU2xCWFdUZFBVRUZCUmtzPQ== # Base64编码后的值
```

## 在环境变量中引用 Secret

Secret最常见的用法是作为环境变量注入到容器中，如下示例。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    env:
    - name: key
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: key1
    imagePullSecrets:
    - name: imagepull-secret
```

## 在 Volume 中引用 Secret

在Volume中引用Secret，就是通过文件的方式直接将Secret的每条数据填入Volume，每条数据是一个文件，键就是文件名，键值就是文件内容。

如下示例中，创建一个名为vol-secret的Volume，这个Volume引用名为“mysecret”的Secret，再将Volume挂载到容器的“/tmp”路径下。Pod创建成功后，在容器的“/tmp”路径下，就有两个文件key1和key2，他们的值分别为“VkZNMEOwVlpVbEpQVHpGTFdrSkRWVWhCV2s5T1ZrNUxUVIZNUjBzMFRWcElVMFpVUkVWV1N3PT0=”和“T0Vkr1RGRlZVRlpVU2xCWFdUZFBVRUZCUmtzPQ==”。

### 说明

此处key1、key2的值为Base64编码后的值。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 500m
        memory: 1024Mi
      requests:
        cpu: 500m
        memory: 1024Mi
    volumeMounts:
      - name: vol-secret          # 挂载名为vol-secret的Volume
        mountPath: "/tmp"
  imagePullSecrets:
  - name: imagepull-secret
  volumes:
  - name: vol-secret
    secret:                      # 引用Secret
      secretName: mysecret
```

# 10 使用 Job 和 CronJob 创建任务负载

任务负载是负责批量处理短暂的一次性任务(short lived one-off tasks)，即仅执行一次的任务，它保证批处理任务的一个或多个 Pod 成功结束。

- 短时任务 (Job)：是Kubernetes用来控制批处理型任务的资源对象。批处理业务与长期服务业务 (Deployment、Statefulset) 的主要区别是批处理业务的运行有头有尾，而长期服务业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出 (Pod自动删除)。
- 定时任务 (CronJob)：是基于时间的Job，就类似于Linux系统的crontab文件中的一行，在指定的时间周期运行指定的Job。

任务负载的这种用完即停止的特性特别适合一次性任务，比如持续集成，配合云容器实例按秒计费，真正意义上做到按需使用。

## 创建 Job

以下是一个Job配置，其计算 $\pi$ 到2000位并打印输出。Job结束需要运行50个Pod，这个示例中就是打印 $\pi$  50次，并行运行5个Pod，Pod如果失败最多重试5次。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
  namespace: cci-namespace-test1
spec:
  completions: 50      # 运行的次数，即Job结束需要成功运行的Pod个数
  parallelism: 5       # 并行运行Pod的数量，默认为1
  backoffLimit: 5      # 表示失败Pod的重试最大次数，超过这个次数不会继续重试。
  activeDeadlineSeconds: 10 # 表示Pod超时时间，一旦达到这个时间，Job即其所有的Pod都会停止。
  template:           # Pod定义
    spec:
      containers:
        - name: pi
          image: perl
          command:
            - perl
            - "-Mbignum=bpi"
            - "-wle"
            - print bpi(2000)
          restartPolicy: Never
```

根据completions和parallelism的设置，可以将Job划分为以下几种类型。



表 10-1 任务类型

Job类型	说明	使用示例
一次性Job	创建一个Pod直至其成功结束	数据库迁移
固定结束次数的Job	依次创建一个Pod运行直至 completions个成功结束	处理工作队列的Pod
固定结束次数的并行Job	依次创建多个Pod运行直至 completions个成功结束	多个Pod同时处理工作队列
并行Job	创建一个或多个Pod直至有一个成功结束	多个Pod同时处理工作队列

## 创建 CronJob

相比Job，CronJob就是一个加了定时的Job，CronJob执行时是在指定的时间创建出Job，然后由Job创建出Pod。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
  namespace: cci-namespace-test1
spec:
  schedule: "0,15,30,45 * * * *"      # 定时相关配置
  jobTemplate:                         # Job的定义
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: main
              image: pi
```

cron的格式从前到后就是：

- Minute
- Hour
- Day of month
- Month
- Day of week

如 "0,15,30,45 \* \* \* \* "，前面逗号隔开的是分钟，后面第一个\* 表示每小时，第二个 \* 表示每个月的哪天，第三个表示每月，第四个表示每周的哪天。

如果你想要每个月的的第一天里面每半个小时执行一次，那就可以设置为 " 0,30 \* 1 \* \* " 如果你想每个星期天的3am执行一次任务，那就可以设置为 "0 3 \* \* 0"。

更详细的cron格式说明请参见<https://zh.wikipedia.org/wiki/Cron>。

# A YAML 语法

YAML 是一种简洁强大的语言，它的设计目标是便于设计和使用人员阅读。

## 基本语法规则

- 大小写敏感。
- 使用缩进表示层级关系。
- 缩进时不允许使用Tab键，只允许使用空格。
- 缩进的空格数目不重要，要求相同层级的元素左侧对齐。
- 使用#表示注释。

## YAML 支持三种数据结构

- 对象：键值对的集合，又称为映射（mapping）/ 哈希（hashes）/ 字典（dictionary）。
- 数组：一组按次序排列的值，又称为序列（sequence）/ 列表（list）。
- 纯量（scalars）：数据最小的单位，单个的、不可再分的值。

## 对象

对象是一组键值对（key: value，冒号后面必须有一个空格或换行），合法的表示方法如下：

```
animal: pets
plant:
  tree
```

也可以将多个键值对写成一个行内对象：

```
hash: {name: Steve, foo: bar}
```

下面这种情况会出错

```
foo: somebody said I should put a colon here: so I did
windows_drive: c:
```

用引号括起来就没有问题，如下所示

```
foo: 'somebody said I should put a colon here: so I did'
windows_drive: 'c:'
```

## 数组

数组使用连字符和空格“- ”表示，合法的表示方法如下：

```
animal:  
- Cat  
- Dog  
- Goldfish
```

也可使用行内表示法：

```
animal: [Cat, Dog, Goldfish]
```

对象和数组可以嵌套使用，形成复合结构：

```
languages:  
- Ruby  
- Perl  
- Python  
websites:  
YAML: yaml.org  
Ruby: ruby-lang.org  
Python: python.org  
Perl: use.perl.org
```

## 纯量

纯量的数据类型有字符串、布尔值、整数、浮点数、Null、时间、日期。

- 字符串表示：

字符串默认不使用引号表示：

```
str: This_is_a_line
```

如果字符串之中包含空格或特殊字符，需要放在引号之中：

```
str: 'content: a string'
```

单引号和双引号都可以使用，两者区别是单引号可以识别转义字符：双引号不会对特殊字符转义：↵

```
s1: 'content:\n a string'  
s2: "content:\n a string"
```

单引号之中如果还有单引号，必须连续使用两个单引号转义。

```
str: 'labor"s day'
```

字符串可以写成多行，从第二行开始，必须有一个单空格缩进。换行符会被转为空格。

```
str: This_is  
 a_multi_line
```

- 整数表示：

```
int_value: 314
```

- 浮点型表示：

```
float_value: 3.14
```

- Null表示：

```
parent: ~
```

- 时间表示：

时间采用ISO8601格式。

```
iso8601: 2018-12-14t21:59:43.10-05:00
```

- 日期表示：

日期采用复合ISO8601格式的年、月、日表示。

```
date: 1976-07-31
```

## 一些特殊符号

- “`---`”表示一个Yaml文件的开始，“`...`”表示一个Yaml文件的结束。

```
---  
# 一个美味水果的列表  
- Apple  
- Orange  
- Strawberry  
- Mango  
...
```

- 对于整数型、浮点型、布尔型数据用两个感叹号“`!!`”进行强制转换：

```
strbool: !!str true  
strint: !!str 10
```

- 多行字符串可以使用“`|`”保留换行符，也可以使用“`>`”折叠换行。这两个符号是Yaml中字符串经常使用的符号。

```
this: |  
Foo  
Bar  
that: >  
Foo  
Bar
```

对应的对象为：

```
{ this: 'Foo\nBar\n', that: 'Foo Bar\n' }
```

一般[YAML 语言教程](#)推荐使用“`|`”就能够满足大多数场景了。

## 注释

YAML支持注释，这是YAML相比JSON的一个优点。

YAML的注释使用“`#`”开头，如下所示。

```
languages:  
- Ruby      # 这是Ruby语言  
- Go        # 这是Go语言  
- Python    # 这是Python语言
```

## 参考文档

- [YAML 1.2 规格](#)
- [Ansible YAML Syntax](#)
- [YAML 语言教程](#)