云容器实例

开发指南

文档版本 01

发布日期 2023-05-30

华为云计算技术有限公司

商标声明


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

注意

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

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

目 录

1 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。

1. **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。
2. **Namespace和Network**

   介绍云容器示例中Namespace与Network的概念。
3. **Pod**

   介绍Pod相关概念，以及如何使用Pod。
4. **Label**

   介绍Label的作用，以及如何使用Label。
5. **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。
6. **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。
7. **使用存储**

   介绍负载中如何使用存储，即如何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）。
8. **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）

2.1 kubectl 配置指南

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1.19版本对应的kubectl。

说明

Apple M1芯片是darwin-arm64架构，如果使用Apple M1芯片的设备，需要**下载**对应架构的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	**cci-iam-authenticator_linux-amd64** **cci-iam-authenticator_linux-amd64_sha256**	**cci-iam-authenticator使用参考**
Darwin AMD 64位	**cci-iam-authenticator_darwin-amd64** **cci-iam-authenticator_darwin-amd64.sha256**	

操作系统	下载地址	查看帮助
Darwin ARM 64位	cci-iam-authenticator_darwin-arm64 cci-iam-authenticator_darwin-arm64.sha256	

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:"19", GitVersion:"v1.19.0",
GitCommit:"e19964183377d0ec2052d1f1fa930c4d7575bd50", GitTreeState:"clean",
BuildDate:"2020-08-26T14:30:33Z", GoVersion:"go1.15", 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=***xxxxxx*

     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ABCDEFAK.. ，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=***ABCDEFAK..*

     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

- 如果无子用户，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。

说明

认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险, 建议在配置文件或者环境变量中密文存放, 使用时解密, 确保安全。

export ACCESS_KEY_ID={Access Key} #替换为HUAWEICLOUD_SDK_AK

export SECRET_ACCESS_KEY={Secret Key} #替换为HUAWEICLOUD_SDK_SK

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
华东-上海二	cn-east-2	cci.cn-east-2.myhuaweicloud.com
华南-广州	cn-south-1	cci.cn-south-1.myhuaweicloud.com
西南-贵阳一	cn-southwest-2	cci.cn-southwest-2.myhuaweicloud.com

2.2 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**https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html**。
user-name	USER_NAME	子用户名，即IAM用户名。如果不配置与domain-name一致。 详情请参见**https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html**。
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**获取AK/SK**，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	IAM的Endpoint，必须配置，详情请参见**https://developer.huaweicloud.com/endpoint?IAM**。
project-name	PROJECT_NAME	项目名，详情请参见**https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html**。
project-id	PROJECT_ID	项目ID，详情请参见**https://support.huaweicloud.com/usermanual-ca/ca_01_0001.html**。
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。 **注意** 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，如果指定的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**），也可使用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。

  命令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”，您可以看到命令空间创建成功，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

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

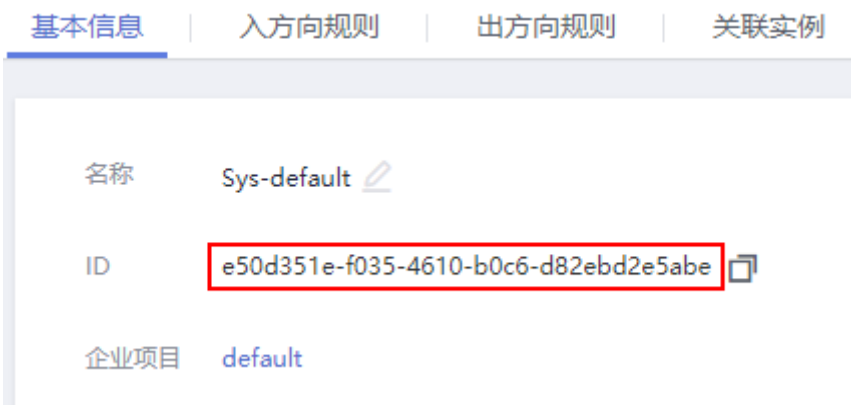
说明

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

- 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kubectl 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。

```
# kubectl 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”，您可以看到命令空间创建成功，且状态为“正常”。

图 3-7 Namespace-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

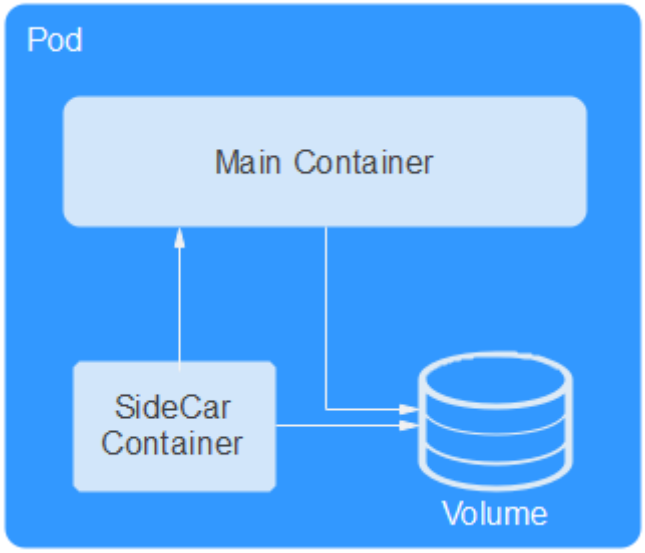
什么是 Pod

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）。具体的规格信息请参考**约束与限制**中的“Pod规格”。

创建 Pod

kubernetes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               # 0.5核
        memory: 1024Mi
      requests:
        cpu: 500m               # 0.5核
        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
```

说明

容器运行的操作系统内核版本由4.18版本升级至5.10版本。

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。

具体的规格信息请参考**约束与限制**中的“Pod规格”。

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-460.106
- 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32GiB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                  # 4核
        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                  # 4核
        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。
- nvidia.com/gpu-tesla-t4：表示NVIDIA Tesla T4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。

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 对容器进行初始化操作

概念

Init-Containers，即初始化容器，顾名思义容器启动的时候，会先启动可一个或多个容器，如果有多个，那么这几个Init Container按照定义的顺序依次执行，只有所有的Init Container执行完后，主容器才会启动。由于一个Pod里的存储卷是共享的，所以Init Container里产生的数据可以被主容器使用到。

Init Container可以在多种K8S资源里被使用到如Deployment、DaemonSet、Job等，但归根结底都是在Pod启动时，在主容器启动前执行，做初始化工作。

使用场景

部署服务时需要做一些准备工作，在运行服务的pod中使用一个init container，可以执行准备工作，完成后Init Container结束退出，再启动要部署的容器。

- **等待其它模块Ready：**比如有一个应用里面有两个容器化的服务，一个是Web Server，另一个是数据库。其中Web Server需要访问数据库。但是当启动这个应用的时候，并不能保证数据库服务先启动起来，所以可能出现在一段时间内Web Server有数据库连接错误。为了解决这个问题，可以在运行Web Server服务的Pod里使用一个Init Container，去检查数据库是否准备好，直到数据库可以连接，Init Container才结束退出，然后Web Server容器被启动，发起正式的数据库连接请求。
- **初始化配置：**比如集群里检测所有已经存在的成员节点，为主容器准备好集群的配置信息，这样主容器起来后就能用这个配置信息加入集群。
- **其它使用场景：**如将pod注册到一个中央数据库、下载应用依赖等。

更多内容请参见**初始容器文档参考**。

操作步骤

步骤1 编辑initcontainer工作负载yaml文件。

vi deployment.yaml

Yaml示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      name: mysql
  template:
    metadata:
      labels:
        name: mysql
    spec:
      initContainers:
      - name: getresource
        image: busybox
        command: ['sleep','20']
      containers:
      - name: mysql
        image: percona:5.7.22
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
        resources:
          limits:
            memory: "500Mi"
            cpu: "500m"
          requests:
            memory: "500Mi"
            cpu: "250m"
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "mysql"
```

步骤2 创建initcontainer工作负载。

kubectl create -f deployment.yaml

命令行终端显示如下类似信息：

```
deployment.apps/mysql created
```

----结束

4.5 Pod 规格计算方式

Pod规格的计算步骤如下：

步骤1 Pod 包含的所有 Init 容器上定义的任何特定资源的约束值 (limit) 或 请求值 (request) 的最大值，作为 Pod 有效初始 request/limit。

步骤2 Pod 对资源的有效 limit/request ，是取如下两项的较大者：

- 所有应用容器对某个资源的 limit/request 之和。
- 对某个资源的有效初始的 limit/request 。

----结束

针对如下实例，计算Pod规格。

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  initContainers:
  - name: config-generator
    image: busybox
    resources:
      requests:
        memory: "256Mi"
        cpu: "250m"
      limits:
        memory: "256Mi"
        cpu: "250m"
  - name: mysql-checker
    image: centos
    resources:
      requests:
        memory: "1Gi"
        cpu: "500m"
      limits:
        memory: "1Gi"
        cpu: "500m"
  containers:
  - name: app
    image: images.my-company.example/app:v4
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "1Gi"
        cpu: "500m"
      limits:
        memory: "1Gi"
        cpu: "500m"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
```

```
      memory: "1Gi"
      cpu: "250m"
    limits:
      memory: "1Gi"
      cpu: "250m"
```

步骤1 Init 容器，request/limit取最大值：memory为1Gi，cpu为500m。

步骤2 所有应用容器request/limit之和：memory为2Gi，cpu为750m。

步骤3 如上两步取较大值，Pod的规格为request/limit：memory 2Gi，cpu 750m。

----结束

4.6 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
```

4.7 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进程却是一直运行的，对于这种应用本身业务出了问题的情况，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。

  说明

  需要为pod配置以下annotation使超时时间(timeoutSeconds)生效：
  cci.io/httpget-probe-timeout-enable:"true"
  具体请参见**Liveness Probe高级配置**样例。

- 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 600
    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:
  template:
    metadata:
      annotations:
        cci.io/httpget-probe-timeout-enable:"true"
  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，并使用一个专有的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。

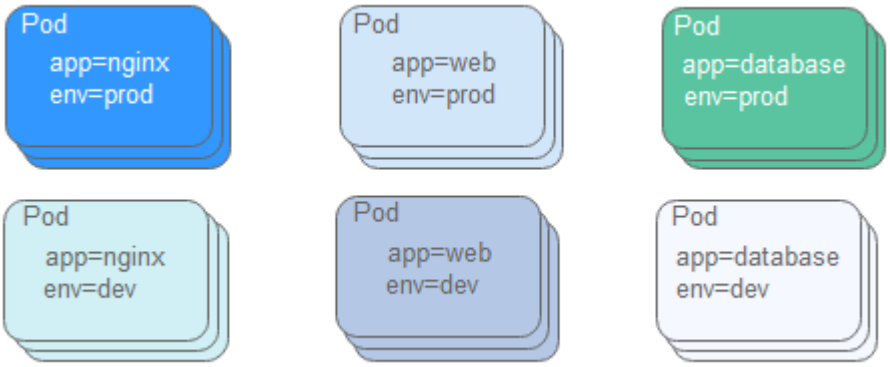
Label的具体形式是key-value的标记对，可以在创建资源的时候设置，也可以在后期添加和修改。

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:
      volumes:
        - name: cci-sfs-test                  # sfs卷的名称
          persistentVolumeClaim:
            claimName: cci-sfs-test
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 500m
            memory: 1024Mi
          requests:
            cpu: 500m
```



```
        memory: 1024Mi
      volumeMounts:
        - name: cci-sfs-test
          mountPath: /tmp/sfs0/krlp2k8j        # sfs卷的容器内挂载路径
    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-6f9f58dffd   2         2         2         1m
nginx-7f98958cdf   0         0         0         48m

$ kubectl get pods -n $namespace_name
NAME                     READY     STATUS    RESTARTS   AGE
nginx-6f9f58dffd-tdmqk   1/1       Running   0          21s
nginx-6f9f58dffd-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 EIPPool

7.1 EIPPool 概述

为方便用户在CCI内直接为Pod关联EIP，CCI新增了名为EIPPool的自定义资源对象，通过EIPPool资源对象，用户可以为Pod自动绑定EIP。EIPPool对象支持两种EIP资源管理方式：动态管理EIP资源（EIP资源由CCI自动创建）、静态管理EIP资源（EIP资源由用户提前创建）。

约束与限制

- 一个Pod只能绑定一个EIP。
- 绑定EIP的Pod，如果要被公网成功访问，需要添加放通相应公网请求流量的安全组规则。
- EIPPool正在被Pod使用时，不支持直接删除EIPPool，需删除关联Pod，再删除EIPPool。
- EIPPool为namespace级别资源，不可跨namespace使用。
- 工作负载滚动升级时，默认策略是逐步创建新Pod然后删除旧Pod（请参见**升级策略**），则可能会由于EIPPool中EIP数量不足而升级失败。建议：EIPPool池的大小略大于使用该EIPPool的所有的Deployment副本数之和，或者maxSurge配置为0，可支持工作负载先减后增滚动升级。

相关操作

您可以参考以下文档， 对EIPPool进行对应的操作：

创建EIPPool

使用EIPPool

管理EIPPool

7.2 创建 EIPPool

7.2.1 创建动态 EIPPool

动态EIPPool，即根据用户在EIPPool中填写的配置，动态创建底层的EIP资源，同时在CCI命名空间下创建相应的EIP对象。

以下示例创建了一个名为eippool-demo1的动态EIPPool，具体字段含义见表7-1。

- 动态创建独占带宽类型的EIPPool，无需指定带宽ID，示例如下：

```
apiVersion: crd.yangtse.cni/v1
kind: EIPPool
metadata:
  name: eippool-demo1
  namespace: xxx                  # EIPPool所在的命名空间，与Pod保持一致
spec:
  amount: 3                      # EIP资源池中的EIP数量
  eipAttributes:
    networkType: 5_bgp
    ipVersion: 4
    bandwidth:
      name: cci-eippool-demo1
      chargeMode: bandwidth
      shareType: PER
      size: 5
```

- 动态创建共享带宽类型的EIPPool，必须指定带宽ID且只需填写该字段，示例如下：

```
apiVersion: crd.yangtse.cni/v1
kind: EIPPool
metadata:
  name: eippool-demo1
  namespace: xxx
spec:
  amount: 3
  eipAttributes:
    networkType: 5_bgp
    ipVersion: 4
    bandwidth:
      id: xxx
      shareType: WHOLE            #带宽类型为共享带宽时，必须指定带宽ID。
```

表 7-1 参数说明

参数	参数含义	约束
name	EIPPool的名称	建议EIPPool的名字长度不超过29个字节，超过字段将被截断，但不影响使用。
namespace	EIPPool所在的命名空间	与Pod的命名空间保持一致。
amount	EIP资源池中的EIP数量	取值范围为0~500。
networkType	EIP的类型	取值范围：5_telcom（电信），5_union（联通），5_bgp（全动态BGP），5_sbgp（静态BGP），5_ipv6。 必须是系统具体支持的类型。

参数	参数含义	约束
ipVersion	弹性公网IP的版本	取值范围：4、6，分别表示创建ipv4和ipv6。 必须是系统具体支持的类型 不填或空字符串时，默认创建ipv4。
chargeMode	按流量计费还是按带宽计费	取值范围：bandwidth，traffic，bandwidth表示按宽带计费，traffic表示按流量计费，不填或者为空时默认是bandwidth。其中IPv6国外默认是bandwidth，国内默认是traffic。
shareType	带宽类型	取值范围：PER，WHOLE（PER为独占带宽，WHOLE是共享带宽）。 该字段为WHOLE时，必须指定带宽ID。
id	带宽ID	取值范围：WHOLE类型的带宽ID。
size	带宽大小	取值范围：1-200。 单位：Mbit/s。 具体范围以各区域配置为准，请参见控制台对应页面显示。

对于以上YAML文件中的EIP相关字段，您还可以在**EIP参数**界面查看参数具体的功能描述，取值范围和约束。

执行以下命令，查看EIPPool详情，-n表示EIPPool所在的命名空间。

回显信息中有名称为eippool-demo1的EIPPool，表示动态EIPPool已创建成功。

```
# kubectl get eippool -n $namespace_name
NAME              EIPS          USAGE        AGE
eippool-demo1                    0/3          39m
```

7.2.2 创建静态 EIPPool

静态EIPPool，即根据用户指定的多个未使用的EIP，静态纳管底层的EIP资源，同时在CCI命名空间下创建相应的EIP对象。如果EIPPool中的EIP已经被NAT或者ELB使用，则会纳管失败。

以下示例创建了一个名为eippool-demo2的静态EIPPool，并在此EIPPool中纳管10.246.173.254和10.246.172.3两个公网IP。示例如下：

```
apiVersion: crd.yangtse.cni/v1
kind: EIPPool                # 创建的对象类别
```

```
metadata:                    # 资源对象的元数据定义
  name:eippool-demo2
spec:                        # EIPPool的配置信息
  eips:                      # 纳管的公网IP
  - 10.246.173.254
  - 10.246.172.3
```

7.3 使用 EIPPool

在命名空间下创建完成EIPPool对象后，用户可在Pod模板中添加指定的Annotation: yangtse.io/eip-pool 使用对应EIPPool中的EIP资源，指定后，Pod在创建时将会自动从EIPPool中获取一个可用的EIP并绑定至Pod。

说明

已经被EIPPool使用的EIP，在VPC界面是无法正常的执行绑定、解绑和删除操作的，因此不建议在VPC界面直接操作已被EIPPool使用的EIP。

以创建的eippool-demo1为例。

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    yangtse.io/eip-pool:eippool-demo1    # 通过指定EIPPool的形式使用EIP
...
```

执行以下命令，查看EIPPool详情，-n表示EIPPool所在的命名空间。

回显信息中名称为eippool-demo1的EIPPool，使用量加1，表示Pod绑定EIP资源成功。

```
# kubectl get eippool -n $namespace_name
NAME                EIPS            USAGE   AGE
eippool-demo1                         1/3     64m
```

Pod启动后，通过Annotation：yangtse.io/allocated-ipv4-eip可查询到Pod当前使用的EIP。

```
apiVersion:v1
kind:Pod
metadata:
  annotations:
    yangtse.io/allocated-ipv4-eip: 116.205.XXX.XXX      # Pod被分配到的EIP
```

说明

如果Pod进行重建，则将会从EIPPool中重新获取一个可用的EIP。

7.4 管理 EIPPool

更新 EIPPool

考虑到用户实际场景，EIPPool对象当前只允许用户调整EIP的数量，即对EIPPool进行扩缩容。用户如果需要调整EIP其他参数，可新建EIPPool对象后在负载配置中替换即可。

EIPPool缩容时，如果EIP资源被占用，则不会删除对应的EIP，直到占用解除。

以动态创建的eippool-demo1为例，将amount：3改为amount: 5。


```
apiVersion: crd.yangtse.cni/v1
kind: EIPPool
metadata:
  name: eippool-demo1
  namespace: xxx
spec:
  amount: 5                    # EIP资源池中的EIP数量
  eipAttributes:               # EIP的一些属性
 ...
```

执行以下命令，查看EIPPool详情，-n表示EIPPool所在的命名空间。

回显信息中名称为eippool-demo1的EIPPool，USAGE由0/3更新为0/5，表示EIPPool更新成功。

```
# kubectl get eippool -n $namespace_name
NAME                EIPS          USAGE        AGE
eippool-demo1                      0/5          39m
```

以静态创建的eippool-demo2为例，更新EIPPool就是增加或减少EIPPool中纳管的公网IP。

```
apiVersion: crd.yangtse.cni/v1
kind: EIPPool                  # 创建的对象类别
metadata:                      # 资源对象的元数据定义
  name:eippool-demo2
spec:                        # EIPPool的配置信息
  eips:                      # 纳管的公网IP
  - 10.246.173.254
  - 10.246.172.3
  - 10.246.172.59
```

删除 EIPPool

直接删除EIPPool时，会级联删除该池下的EIP资源。当有EIP被Pod占用时，无法触发EIPPool的删除，当有EIP被其他资源占用时，EIPPool无法删除成功。

8 EIP

8.1 EIP 概述

为方便用户在CCI内直接为Pod关联弹性公网IP，只需在创建Pod时配置annotation，弹性公网IP就会随Pod自动绑定该Pod。自动绑定弹性公网IP分为两种场景：

表 8-1 自动绑定 EIP 配置 annotation

场景	配置annotation参数
场景一：为Pod自动创建并绑定EIP	**yangtse.io/pod-with-eip**: **"true"**
场景二：为Pod绑定已有EIP	**yangtse.io/eip-id**: "eip-id-xxx"

限制与约束

- 一个Pod只能绑定一个EIP，一个EIP只能被一个Pod绑定。
- 创建Pod时，可指定annotation属性，创建完成后，更新EIP相关的annotation均无效。
- EIP随Pod创建的优先级高于使用EIPPool创建的EIP。
- 绑定已有EIP创建的优先级高于EIP随Pod创建的EIP。
- 绑定EIP的Pod，如果要被公网成功访问，需要添加放通相应公网请求流量的安全组规则。
- 已经被Pod绑定的弹性公网IP，请勿通过弹性公网IP的console或API直接操作（修改别名/删除/解绑/绑定/转包周期等操作），否则可能导致资源残留。
- 绑定已有EIP时，使用负载创建的Pod删除重建后，由于需要等待前一个Pod完成解绑，所以重建的Pod就绪时间会变长。
- 绑定已有的EIP必须是用户手动创建给Pod使用的，不能使用EIPPool生成的EIP，否则会导致EIP状态异常。

相关操作

您可以参考以下文档， 对EIP进行对应的操作：

为Pod动态创建EIP

为Pod绑定已有EIP

8.2 为 Pod 动态创建 EIP

EIP 随 Pod 创建

创建Pod时，填写pod-with-eip的annotation后，EIP会随Pod自动创建并绑定至该Pod。

以下示例创建一个名为nginx的无状态负载，EIP将随Pod自动创建并绑定至Pod。具体字段含义见**表8-2**。

- 创建独占带宽类型的Deployment，无需指定带宽ID，示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "14"
    description: ""
  name: nginx
  namespace: eip
spec:
  ...
  template:
    metadata:
      annotations:
        yangtse.io/pod-with-eip: "true"
        yangtse.io/eip-bandwidth-size: "5"
        yangtse.io/eip-network-type: 5_g-vm
        yangtse.io/eip-charge-mode: bandwidth
        yangtse.io/eip-bandwidth-name: "xxx"
```

- 创建共享带宽类型的Deployment，必须指定带宽ID且只需填写该字段，示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "14"
    description: ""
  name: nginx
  namespace: eip
spec:
  ...
  template:
    metadata:
      annotations:
        yangtse.io/pod-with-eip: "true"
        yangtse.io/eip-bandwidth-id: "xxx"
```

表 8-2 参数说明

参数	参数含义	必选/可选	约束
yangtse.io/pod-with-eip	EIP随Pod创建	必选	字段填写值为"true"时，才会开启该功能。
yangtse.io/eip-bandwidth-size	指定EIP带宽	可选	默认值为5。
yangtse.io/eip-network-type	指定带宽类型	可选	默认值为5_bgp，如果region不支持5_bgp，同时该字段未填写时，pod会报event。
yangtse.io/eip-charge-mode	指定收费类型	可选	默认值bandwidth，可选值traffic。
yangtse.io/eip-bandwidth-id	指定共享带宽id	必选（共享型带宽）	填写该字段时，无需填写其他值。
yangtse.io/eip-bandwidth-name	指定带宽名	可选	默认值同该EIP的名称。

Pod 的 EIP 准备就绪

Pod业务容器的启动时间可能早于EIP分配结果返回成功时间，在Pod启动过程中EIP可能会绑定失败。

通过在init container中可检查EIP是否已经分配成功。容器网络控制器会在Pod IP分配后，为Pod绑定EIP并返回分配结果至Pod的Annotation（yangtse.io/allocated-ipv4-eip），通过Pod配置init container并使用downwardAPI，把yangtse.io/allocated-ipv4-eip annotation通过volume挂载到init container里，可检查EIP是否已经分配成功。具体您可以参考以下示例配置init container：

须知

CCI当前提供Pod自动绑定EIP的能力，实际的EIP分配发生在Pod调度完成之后，因此当前不支持通过ENV将EIP的Annotation注入至Pod中。

```
apiVersion: v1
kind: Pod
metadata:
  name: example
  namespace: demo
  annotations:
    yangtse.io/pod-with-eip: "true"
    yangtse.io/eip-bandwidth-size: "5"
    yangtse.io/eip-network-type: 5_g-vm
    yangtse.io/eip-charge-mode: bandwidth
    yangtse.io/eip-bandwidth-name: "xxx"
spec:
  initContainers:
  - name: init
    image: busybox:latest
    command: ['timeout', '60', 'sh', '-c', "until grep -E '[0-9]+' /etc/eipinfo/allocated-ipv4-eip; do echo waiting
```

```
for allocated-ipv4-eip; sleep 2; done"]
    volumeMounts:
        - name: eipinfo
          mountPath: /etc/eipinfo
  volumes:
    - name: eipinfo
      downwardAPI:
        items:
          - path: "allocated-ipv4-eip"
            fieldRef:
              fieldPath: metadata.annotations['yangtse.io/allocated-ipv4-eip']
```

EIP 随 Pod 删除

当Pod被删除时，EIP会随Pod一起被删除。删除Pod指令如下：

```
# kubectl delete pod nginx -n $namespace_name
```

8.3 为 Pod 绑定已有 EIP

为 Pod 指定 EIP 的 ID

创建Pod时，填写yangtse.io/eip-id的annotation后，EIP会随Pod自动完成绑定。

以下示例创建一个名为nginx的实例数为1的无状态负载，EIP将随Pod自动绑定至Pod。具体字段含义见**表1**。

```
apiVersion: apps/v1
kind: Deployment
metadata:
 annotations:
   deployment.kubernetes.io/revision: "14"
   description: ""
 name: nginx
 namespace: eip
spec:
 ...
 replicas: 1
  template:
   metadata:
    annotations:
     yangtse.io/eip-id: 65eb3679-7a8d-4b24-b681-0b661axxxxcb
```

表 8-3 参数说明

参数	参数含义	必选/可选	约束
yangtse.io/eip-id	弹性公网ID	必选	必须是弹性公网IP页面能查到的ID信息。

Pod 的 EIP 准备就绪

Pod业务容器的启动时间可能早于EIP分配结果返回成功时间，在Pod启动过程中EIP可能会绑定失败。

通过在init container中可检查EIP是否已经分配成功。容器网络控制器会在Pod IP分配后，为Pod绑定EIP并返回分配结果至Pod的Annotation（yangtse.io/allocated-ipv4-eip），通过Pod配置init container并使用downwardAPI，把yangtse.io/allocated-

ipv4-eip annotation通过volume挂载到init container里，可检查EIP是否已经分配成功。具体您可以参考以下示例配置init container：

```
apiVersion: v1
kind: Pod
metadata:
  name: example
  namespace: demo
  annotations:
    yangtse.io/eip-id: 65eb3679-7a8d-4b24-b681-0b661axxxxcb
spec:
  initContainers:
  - name: init
    image: busybox:latest
    command: ['timeout', '60', 'sh', '-c', "until grep -E '[0-9]+' /etc/eipinfo/allocated-ipv4-eip; do echo waiting
for allocated-ipv4-eip; sleep 2; done"]
    volumeMounts:
        - name: eipinfo
          mountPath: /etc/eipinfo
  volumes:
    - name: eipinfo
      downwardAPI:
        items:
          - path: "allocated-ipv4-eip"
            fieldRef:
              fieldPath: metadata.annotations['yangtse.io/allocated-ipv4-eip']
```

9 镜像快照

9.1 创建镜像快照

运行容器需要先拉取指定的容器镜像，但因网络和容器镜像大小等因素，镜像拉取耗时往往成了Pod启动的主要耗时。通过镜像快照功能，可以事先将要使用的镜像制作成快照，基于该快照创建Pod，从而避免镜像下载，提升Pod的启动速度。

要了解镜像快照的工作原理，请参阅**镜像快照概述**。

创建镜像快照

假如镜像快照定义的文件名称为my-imagesnapshot.yaml，则执行kubectl apply -f my-imagesnapshot.yaml即可创建镜像快照，-f 表示从文件创建。

```
# kubectl apply -f my-imagesnapshot.yaml
```

以下示例创建一个名为my-imagesnapshot的镜像快照。

```
apiVersion: imagecache.cci.io/v1
kind: ImageSnapshot
metadata:
  name: "my-imagesnapshot"
spec:
  images:
  - image: serverA.com/xx/redis
  - image: nginx:latest
  - image: redis
  registries:
  - server: serverA.com
    username: userA
    password: pwdA
  - server: serverB.com
    username: userB
    password: pwdB
    plainHTTP: true
  - server: serverC.com
    username: userC
    password: pwdC
    insecureSkipVerify: true
  buildingConfig:
    namespace: my-namespace-a
  evsInfo:
    volumeType: ssd
    volumeSizeGiB: 30
```

```
  filesystemType: ext4
ttlDaysAfterCreated: 20
```

表 9-1 参数说明

名称	类型	必填	示例值	描述
name	string	是	my-imagesnapshot	镜像快照名称。
image	string	是	nginx:latest	用于制作镜像快照的镜像。
server	string	是	serverA.com	不带 http:// 或 https:// 前缀的镜像仓库地址。
username	string	否	userA	镜像仓库用户名。
password	string	否	pwdA	镜像仓库密码。
plainHTTP	boolean	否	true	如果是使用HTTP协议的自建镜像仓库地址，需要设置为true，否则会因协议不同而导致镜像拉取失败。默认值为false。
insecureSkipVerify	boolean	否	true	如果是使用自签发证书的自建镜像仓库地址，需要设置为true来跳过证书认证，否则会因证书认证失败而导致镜像拉取失败。默认值为false。
namespace	string	是	my-namespace-a	用户命名空间。镜像快照制过程需要在用户命名空间创建CCI实例。
evsInfo	EVSInfo	否	/	镜像快照的EVS云盘配置。
volumeType	string	否	ssd	镜像快照的磁盘类型。默认为ssd。
volumeSizeGiB	integer	否	20	镜像快照大小，单位Gi。默认为20。
filesystemType	string	否	ext4	镜像快照的磁盘文件系统类型。当前只支持ext4。

名称	类型	必填	示例值	描述
ttlDaysAfterCreated	integer	否	10	镜像快照保留时间，过期将会被清理。默认值为0，即永不过期。 **说明** 快照过期后，仍会占用配额，需定期审核过期镜像快照后删除。

9.2 使用镜像快照

使用镜像快照创建Pod，支持以下两种方式：

- 自动匹配

自动匹配将从所有用户创建的可用的镜像快照中选择最优的镜像。按以下顺序进行匹配：

a. 镜像匹配度：优先选择匹配度高的镜像快照，匹配度指的是Pod和镜像快照两者在镜像上的匹配情况。

b. 创建时间：优先选择创建时间更新的镜像快照。

- 明确指定

明确指定使用的镜像快照。该镜像缓存必须为创建完成可用（Available）状态。

说明

如果同时使用明确指定和自动匹配两种镜像快照方式创建Pod时，如果两者冲突，会返回400报错。

自动匹配

创建CCI实例时，可以通过注解中添加以下键值来开启自动匹配镜像快照。

名称	示例值	描述
cci.io/image-snapshot-auto-match	"true"	设置是否开启自动匹配镜像快照。

以创建Deployment为例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
```

```
    labels:
      app: redis
    annotations:
      cci.io/image-snapshot-auto-match: "true"
  spec:
    containers:
      - image: redis
        name: container-0
        resources:
          limits:
            cpu: 500m
            memory: 1024Mi
          requests:
            cpu: 500m
            memory: 1024Mi
    imagePullSecrets:
      - name: imagepull-secret
```

明确指定

明确指定和自动匹配同时使用，明确指定具有高优先级。

创建CCI实例时，可以通过注解中添加以下键值来明确指定镜像快照。

名称	示例值	描述
cci.io/image-snapshot-specified-name	"my-imagesnapshot"	指定的镜像快照名称。

以创建Deployment为例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
      annotations:
        cci.io/image-snapshot-auto-match: "false"
        cci.io/image-snapshot-specified-name: "my-imagesnapshot"
    spec:
      containers:
        - image: redis
          name: container-0
          resources:
            limits:
              cpu: 500m
              memory: 1024Mi
            requests:
              cpu: 500m
              memory: 1024Mi
      imagePullSecrets:
        - name: imagepull-secret
```

9.3 管理镜像快照

查看镜像快照

创建镜像快照后，您可以查询镜像快照信息。

- 查询用户创建的所有镜像快照，执行如下命令。
  ```
  kubectl get imagesnapshot -oyaml
  ```
- 查询单个镜像快照，执行如下命令。
  ```
  kubectl get imagesnapshot my-imagesnapshot -oyaml
  ```
  返回结果如下：
  ```
  apiVersion: imagecache.cci.io/v1
  kind: ImageSnapshot
  metadata:
    name: "my-imagesnapshot"
  spec:
    images:
    - image: serverA.com/xx/redis
    - image: nginx:latest
    - image: redis
    ...
  status:
    snapshotID: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    snapshotName: cci-imagesnapshot-my-imagesnapshot
    phase: Available
    reason: xx
    message: xx
    images:
    - image: serverA.com/image-a:v1
    - image: serverB.com/image-b:v2
    - image: serverC.com/image-c:v3
  ```
  从返回结果中可以查看镜像快照名称、状态等信息，其中状态相关主要字段如下：

  表 9-2 参数说明

名称	类型	示例值	描述
phase	string	Available	镜像快照当前所处状态： ● Available：可用 ● Unavailable：不可用 ● Creating：创建中 ● Deleting：删除中
snapshotID	string	/	当前镜像快照对应的evs云盘快照ID。

名称	类型	示例值	描述
snapshotName	string	cci-imagesnapshot-my-imagesnapshot	当前镜像快照对应的evs云盘快照名称，命名格式为cci-imagesnapshot-{imagesnap_name}
reason	string	CreateEVSSnapshotFailed	镜像快照处于当前状态的原因。
message	string	/	镜像快照处于当前状态的原因的具体描述。
images	array of PulledImage	/	镜像快照中缓存的镜像的信息。
image	string	nginx:latest	镜像快照中的镜像。

删除镜像快照

镜像快照对应一份EVS云盘快照，保留镜像快照需要支付相应的EVS云盘快照费用。对于不再使用的镜像快照，如果在创建时未设置保留时长，需手动删除。

使用如下命令删除镜像快照：

```
kubectl delete imagesnapshot my-imagesnapshot --wait=false
```

镜像快照相关API为类kubernetes风格API，并不支持完整的kubernetes API特性，比如“不支持watch”，“不支持fieldSelector、labelSelector”等等。

所以使用kubectl删除镜像快照时，建议额外指定--wait=false。否则，kubectl在删除镜像快照后会发起watch请求以等待镜像快照删除成功。但因镜像快照API不支持watch，此时kubectl工具可能会报错。

10 Pod 资源监控指标

CCI支持Pod资源基础监控能力，提供CPU、内存、磁盘、网络等多种监控指标，满足对Pod资源的基本监控需求。

Pod内置系统agent，默认会以http服务的形式提供Pod和容器的监控指标。agent集成到Pod里面，会占用Pod内资源，建议您预留30MB的内存。

资源监控指标

资源基础监控包含CPU/内存/磁盘等类别，具体请参见资源监控指标。

表 10-1 资源监控指标

监控指标类	指标名称	释义
CPU	container_cpu_system_seconds_total	System CPU累积占用时间（单位：秒）
	container_cpu_usage_seconds_total	容器在所有CPU内核上的累积占用时间 (单位：秒)
	container_cpu_user_seconds_total	User CPU累积占用时间（单位：秒）
	container_cpu_cfs_periods_total	已经执行的CPU时间周期数
	container_cpu_cfs_throttled_periods_total	被限流的CPU时间周期
	container_cpu_cfs_throttled_seconds_total	被限流的CPU时间（单位：秒）
文件系统/磁盘IO	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(单位：字节)
	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

监控指标类	指标名称	释义
	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(单位：字节)
	container_fs_reads_bytes_total	容器累积读取磁盘/文件系统数据的总量(单位：字节)
	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(单位：字节)
	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(单位：字节)
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（单位：字节）
	container_memory_mapped_file	容器内存映射文件的大小（单位：字节）
	container_memory_max_usage_bytes	容器历史最大内存使用量（单位：字节）

监控指标类	指标名称	释义
	container_memory_rss	容器常驻内存集的大小（单位：字节）
	container_memory_swap	容器虚拟内存使用量（单位：字节）
	container_memory_usage_bytes	容器当前的内存使用量（单位：字节）
	container_memory_working_set_bytes	容器工作集内存使用量（单位：字节）
网络	container_network_receive_bytes_total	容器网络累积接收数据总量（单位：字节）
	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（单位：字节）
	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
进程	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	容器当前打开文件描述符的个数
	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_memory_limit_bytes	容器可以使用的总内存量限制

监控指标类	指标名称	释义
	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（单位：秒）
	container_last_seen	最近一次监控采集器感知到容器的时间
gpu	container_accelerator_memory_used_bytes	容器正在使用的GPU加速卡内存量（单位：字节）
	container_accelerator_memory_total_bytes	总GPU加速卡可用内存量（单位：字节）
	container_accelerator_duty_cycle	GPU加速卡实际运行时间百分比

监控指标数总计59个，与cadvisor提供的指标数一致。

指标详细含义，可参考cadvisor文档：**https://github.com/google/cadvisor/blob/v0.39.0/docs/storage/prometheus.md**。

基础配置

以下示例介绍Pod资源监控指标的基础配置方式，提供了Pod级别特性开关和自定义端口的能力。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx-exporter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-exporter
  template:
    metadata:
      labels:
        app: nginx-exporter
      annotations:
        monitoring.cci.io/enable-pod-metrics: "true"
        monitoring.cci.io/metrics-port: "19100"
    spec:
      containers:
        - name: container-0
          image: 'nginx:alpine'
          resources:
            limits:
              cpu: 1000m
              memory: 2048Mi
            requests:
              cpu: 1000m
              memory: 2048Mi
```

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

表 10-2 参数说明

Annotation	功能	可选值	默认值
monitoring.cci.io/enable-pod-metrics	是否开启监控指标特性	true，false（不区分大小写）	true
monitoring.cci.io/metrics-port	指定pod exporter启动监听端口	合法端口（1~65535）	19100

高级配置

创建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。

如下示例中定义的Secret中包含三条Key-Value。

```
apiVersion: v1
kind: Secret
metadata:
  name: cert
type: Opaque
data:
    ca.crt: ...
    server.crt: ...
    server.key: ...
```

配置tls证书

用户可以通过配置annotation指定exporter server的tls证书套件，进行加密通信，并使用文件挂载的方式，关联证书secret。示例如下：

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx-tls
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-tls
  template:
    metadata:
      labels:
        app: nginx-tls
      annotations:
        monitoring.cci.io/enable-pod-metrics: "true"
        monitoring.cci.io/metrics-port: "19100"
        monitoring.cci.io/metrics-tls-cert-reference: cert/server.crt
        monitoring.cci.io/metrics-tls-key-reference: cert/server.key
        monitoring.cci.io/metrics-tls-ca-reference: cert/ca.crt
        sandbox-volume.openvessel.io/volume-names: cert
    spec:
      volumes:
        - name: cert
          secret:
            secretName: cert
```

```
      defaultMode: 384
containers:
  - name: container-0
    image: 'nginx:alpine'
    resources:
      limits:
        cpu: 1000m
        memory: 2048Mi
      requests:
        cpu: 1000m
        memory: 2048Mi
    volumeMounts:
      - name: cert
        mountPath: /tmp/secret0
imagePullSecrets:
  - name: imagepull-secret
```

表 10-3 tls 证书参数说明

Annotation	功能	可选值	默认值
monitoring.cci.io/metrics-tls-cert-reference	tls证书volume引用	${volume-name}/${volume-keyOrPath}（卷/路径）	无（使用http）
monitoring.cci.io/metrics-tls-key-reference	tls私钥volume引用	${volume-name}/${volume-keyOrPath}	无（使用http）
monitoring.cci.io/metrics-tls-ca-reference	tls CA volume引用	${volume-name}/${volume-keyOrPath}	无（使用http）

以上参数的值为tls的证书、私钥、CA文件所在存储卷的“卷名”和“路径”。

获取资源监控指标

配置完上述监控属性后，在能访问Pod IP的VPC内，通过执行如下命令获取Pod的监控数据。

```
curl $podIP:$port/metrics
```

其中<podIP>为Pod的IP地址，<port>为监听端口，例如curl 192.168.XXX.XXX:19100/metrics

11 Pod 日志采集

本章节将介绍在Pod内进行日志采集，允许客户通过配置容器内自定义路径下的日志文件来采集日志，并通过自定义策略处理，上报到用户kafka日志中心。

资源限制

建议您为Fluent Bit预留50MB的内存。

约束与限制

- 当前不支持容器标准输出采集上报到kafka。
- 当前不支持日志轮转，如果需要对日志文件进行大小的控制，请自行处理。
- 单条日志长度限制为250KB，如果超过则会丢弃。
- 不支持指定系统、设备、cgroup、tmpfs、localdir等挂载目录的日志采集，会直接忽略。
- 同一个容器中待采集的日志文件不能重名，如果有重复文件则只会采集到采集器首次感知到的日志文件。
- 日志文件的文件名，最大长度为190，超过长度限制的日志文件将不会被采集。

基础配置

Fluent Bit是一个开源的多平台日志处理器工具，Fluent Bit配置由SERVICE，INPUT，FILTER，PARSER，OUTPUT等模块组成，目前我们只允许定义OUTPUT模块，在OUTPUT中定义日志内容的目的端。

您可以使用下面的ConfigMap将Fluent Bit流程日志发送到kafka。

约束与限制

- output.conf配置内容需要小于1MB。
- 缩进格式需要[OUTPUT]为最外层无缩进，下面配置项为固定4空格缩进。

基础配置

在您的主配置文件中需要配置以下部分：

```
kind: ConfigMap
apiVersion: v1
metadata:
```



```
  name: cci-logging-conf
  labels:
    logconf.k8s.io/discovery: "true"
data:
  output.conf: |
      [OUTPUT]
          Name       kafka
          Match      *
          Brokers    192.168.1.3:9092
          Topics     test
```

表 11-1 参数说明

参数	参数含义	必选/可选	约束
logconf.k8s.io/discovery	标识Configmap为fluent-bit日志配置文件。	必选	必选值：true
Name	插件名称。	必选	必选值：kafka 当前只支持kafka插件。
Match	与传入记录的标签匹配。“*”作为通配符。	可选	如果配置的话，必须是“*”。
Brokers	broker（kafka）地址。可以同时配置多个。	必选	例如：192.168.1.3:9092,192.168.1.4:9092,192.168.1.5:9092
Topics	日志主题。	可选	默认值为fluent-bit 传入的topic必须已经存在。

通过在Pod上配置volume，并配置annotation来指定sandbox volume和对应的日志output配置文件。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: kafka-dey
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
      annotations:
        logpath.k8s.io/container-0: /var/log/*.log;/var/paas/sys/log/virtual-kubelet.log
        logconf.k8s.io/fluent-bit-configmap-reference: cci-logging-conf
    spec:
      containers:
        - name: container-0
          image: 'nginx:alpine'
          resources:
```

```
        limits:
          cpu: 1000m
          memory: 2048Mi
        requests:
          cpu: 1000m
          memory: 2048Mi
    imagePullSecrets:
      - name: default-secret
```

表 11-2 参数说明

Annotation	功能	约束
logpath.k8s.io/$containerName	通过Pod容器环境变量配置采集文件 $containerName为容器名变量。	支持配置多条路径，每个路径配置都要是以“/”开头的绝对路径，各路径配置以“;”分隔。 只支持完整日志文件路径或者使用带“*”通配符的文件名。如果配置带“*”通配符的文件名，则文件所在目录需要在容器启动时就存在。 文件名的最大长度为190。
logconf.k8s.io/fluent-bit-configmap-reference	指定fluent-bit日志采集配置的configmap名称	配置的configmap必须已经存在，并且符合配置fluent-bit中描述的要求。

高级配置

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。

```
apiVersion: v1
kind: Secret
metadata:
  name: cci-sfs-kafka-tls
type: Opaque
data:
   ca.crt: ...
   server.crt: ...
   server.key: ...
```

用户可以通过配置SSL参数，进行加密的安全连接，证书文件等相关的文件引用通过sandbox volume的特性来支持。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cci-logging-conf-tls
  labels:
     logconf.k8s.io/discovery: true
data:
    output.conf: |
        [OUTPUT]
            Name        kafka
            Match       *
            Brokers     192.168.1.3:9092
            Topics      test
            rdkafka.security.protocol ssl
```

```
rdkafka.ssl.certificate.location ${sandbox_volume_kafkatls}/client.crt
rdkafka.ssl.key.location ${sandbox_volume_kafkatls}/client.key
rdkafka.ssl.ca.location ${sandbox_volume_kafkatls}/ca.crt
rdkafka.enable.ssl.certificate.verification true
rdkafka.request.required.acks 1
```

表 11-3 参数说明

参数	参数含义	必选/可选	可选值
rdkafka.security.protocol	用于与代理通信的协议。	开启SSL认证必选	ssl
rdkafka.ssl.certificate.location	SSL公钥路径	开启SSL认证必选	${sandbox_volume_**${VOLUME_NAME}**}/**some.cert**
rdkafka.ssl.key.location	SSL私钥路径	开启SSL认证必选	${sandbox_volume_**${VOLUME_NAME}**}/**some.key**
rdkafka.ssl.ca.location	CA 证书的文件或目录路径	开启对服务端证书认证必选	${sandbox_volume_**${VOLUME_NAME}**}/**some-bundle.crt**
rdkafka.enable.ssl.certificate.verification	是否开始对服务端证书认证	可选	可选true, false。默认为true。

通过在Pod上配置volume，并配置annotation来指定sandbox volume和对应的日志output配置文件。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: kafka-tls
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
      annotations:
        logpath.k8s.io/container-0: /var/log/*.log;/var/paas/sys/log/virtual-kubelet.log
        logconf.k8s.io/fluent-bit-configmap-reference: cci-logging-conf
        sandbox-volume.openvessel.io/volume-names: kafkatls
    spec:
      volumes:
        - name: kafkatls
          secret:
            secretName: cci-sfs-kafka-tls
      containers:
        - name: container-0
          image: 'nginx:alpine'
          resources:
            limits:
              cpu: 1000m
              memory: 2048Mi
```

```
        requests:
          cpu: 1000m
          memory: 2048Mi
      volumeMounts:
        - name: kafkatls
          mountPath: /tmp/sfs
  imagePullSecrets:
    - name: default-secret
```

有关kafka的更多配置项内容，可参考 **https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md**。

12 使用 Service 和 Ingress 管理网络访问

12.1 Service

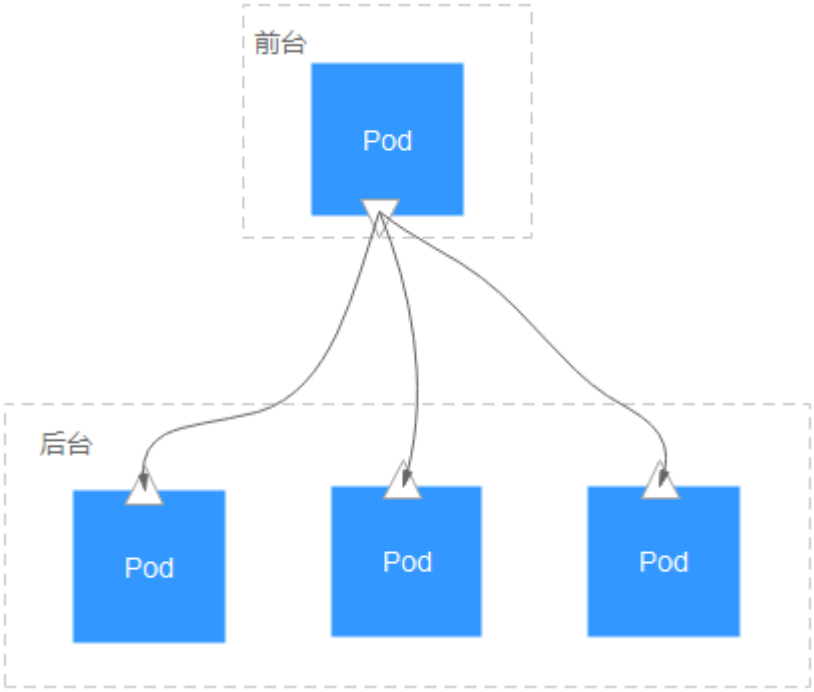
直接访问 Pod 的问题

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12-1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 12-1 负载间访问

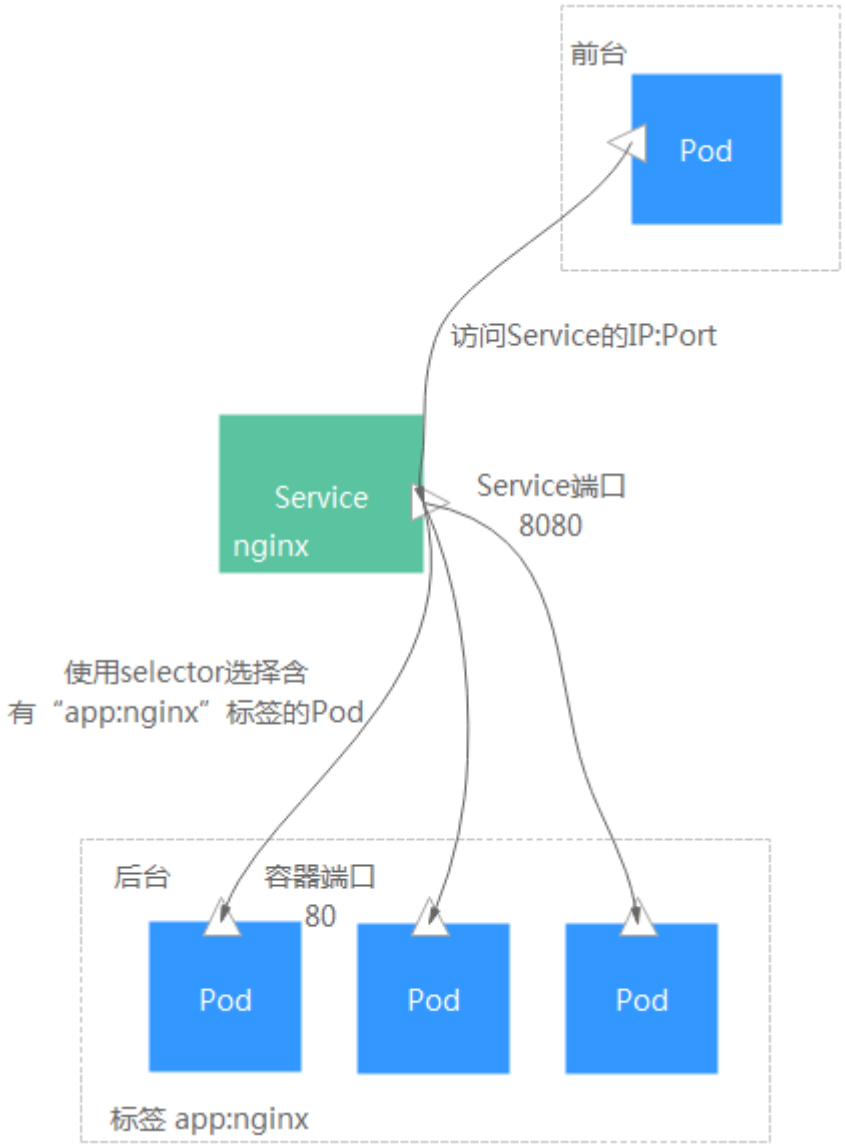


Service 是如何工作的

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12-2所示。

图 12-2 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。

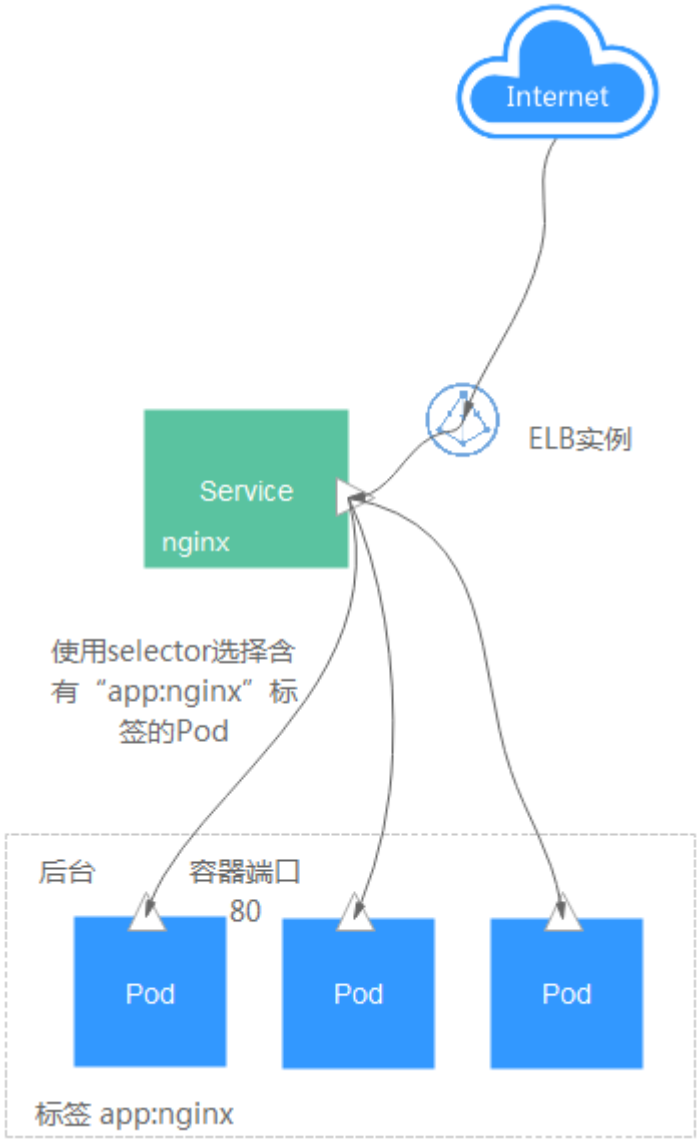
云容器实例同时还支持创建LoadBalancer类型的Service，将**增强型ELB实例**与Service绑定，这样访问ELB实例的流量就会访问到Service。

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。
- 跨namespace不支持service或ELB域名访问，只能通过ELB内网IP:端口访问。

图 12-3 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
```

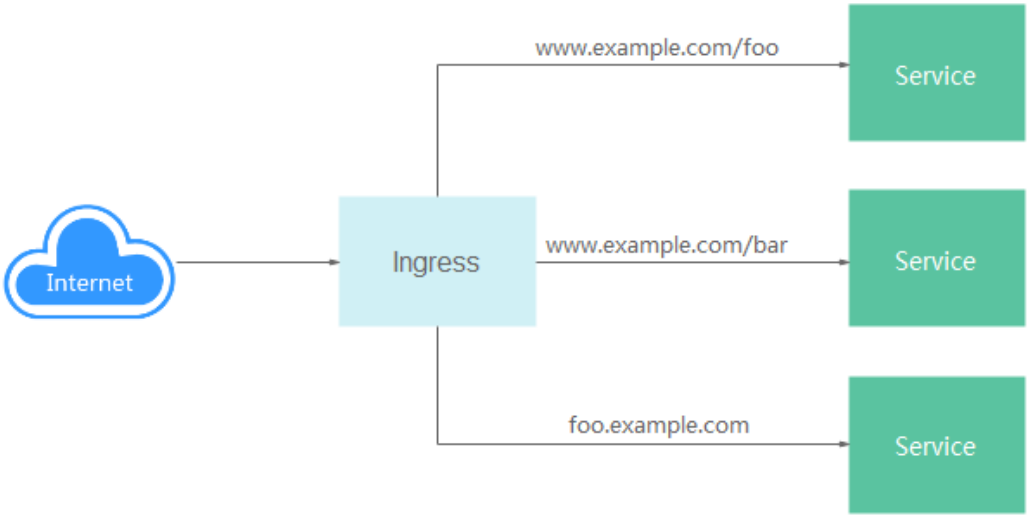
```
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    targetPort: 80
    port: 8080          # ELB访问端口
    protocol: TCP
  type: LoadBalancer    # Service的类型
```

12.2 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。

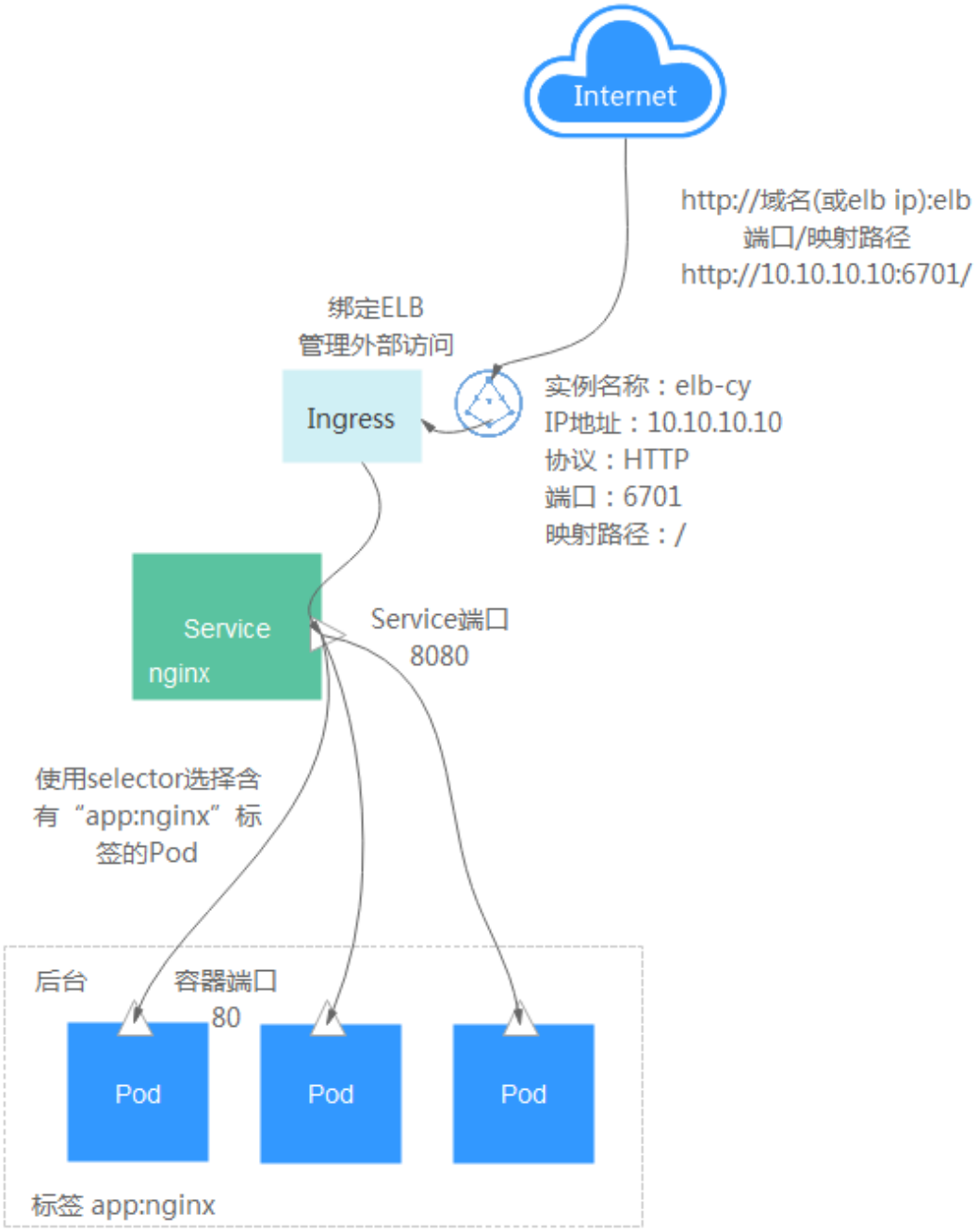
但是Service是基于四层TCP和UDP协议转发的，Ingress可以基于七层的HTTP和HTTPS协议转发，可以通过域名和路径做到更细粒度的划分，如下图所示。

图 12-4 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12-5所示。

图 12-5 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，从而将流量转发到对应负载中的Pod。

```
apiVersion: extensions/v1beta1                          # Ingress的版本
kind: Ingress
metadata:
  name: nginx
  labels:
    app: nginx
    isExternal: "true"   # 系统预留字段，必选参数，取值必须为 "true"
    zone: data         # 系统预留字段，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，从而将流量转发到对应负载中的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
```

配置 URL 重定向的路由服务

以如下模板为例，Ingress对接了名为service-test的后端服务，访问这个Ingress的 "/service-test" 路径会被重定向到后端服务service-test的"/"路径下面。

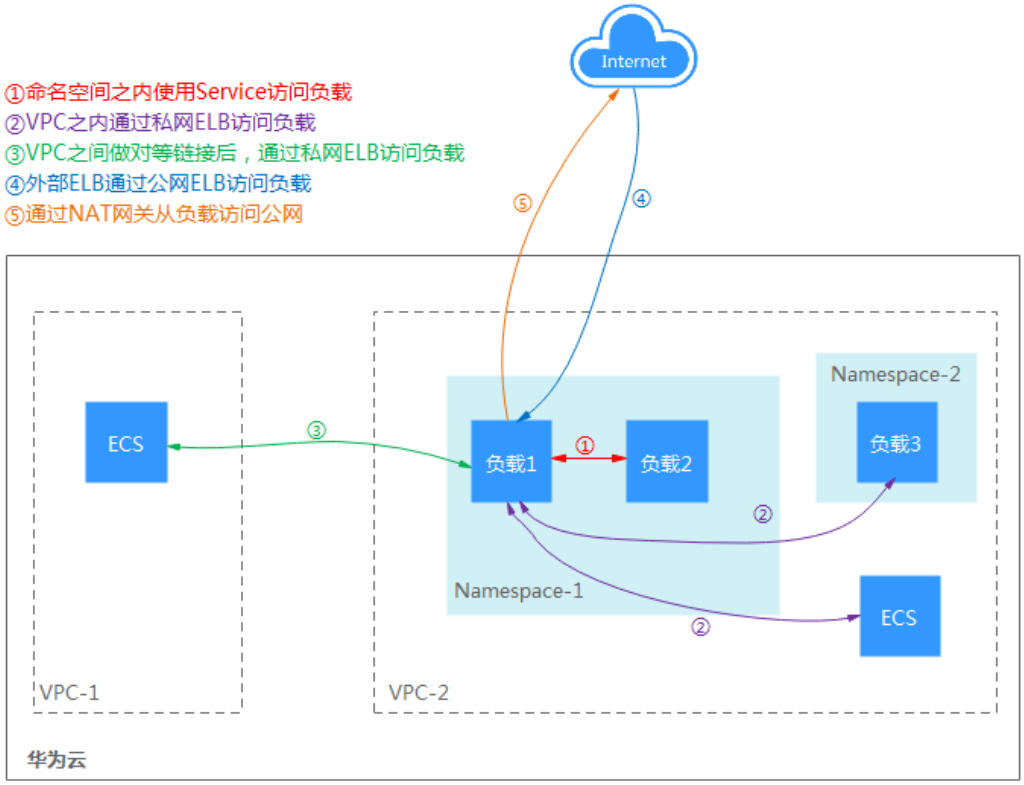
```
cat <<-EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-redirect-test
  namespace: default
 spec:
  rules:
  - host: ingress-test.com
    http:
      paths:
      - path: /
        backend:
          serviceName: service-test
          servicePort: 80
EOF
```

12.3 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12-6**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 12-6 网络访问示意图



12.4 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。

  说明

  需要为pod配置以下annotation使超时时间(timeoutSeconds)生效：
  cci.io/httpget-probe-timeout-enable:"true"
  具体请参见**Liveness Probe高级配置**样例。

- 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，您会看到如下信息。

```
$ 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。

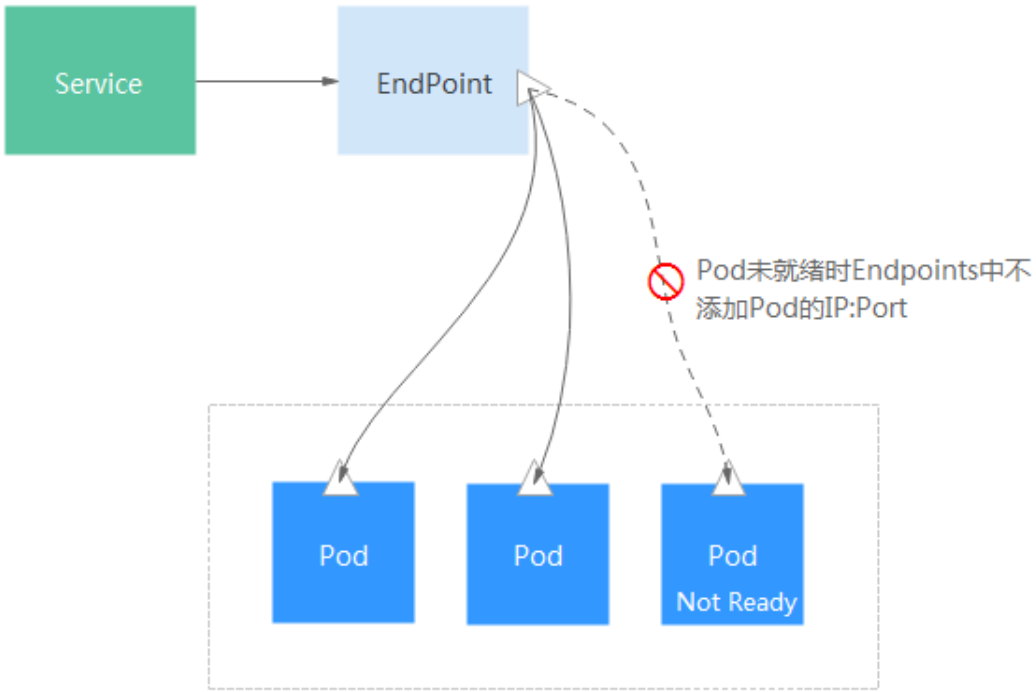
```
$ kubectl get endpoints -n $namespace_name
NAME         ENDPOINTS                                            AGE
nginx        192.168.113.81:80,192.168.165.64:80,192.168.198.10:80   14m
```

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 12-7 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（livness 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。

```
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:           # readinessProbe
          httpGet:                # HTTP GET定义
            path: /read
            port: 80
      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
```


13 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（上一代产品）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、SFS3.0和SFS Turbo三种类型的文件系统。
  - SFS、SFS3.0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。

    说明

    如需在VPC中访问SFS 3.0容量型，请先在VPC中购买SFS 3.0容量型的VPC终端节点，可参考**配置VPC终端节点**。如果已经购买了VPC终端节点，则不需要购买。
  - 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。
- **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://sdkcenter.developer.huaweicloud.com/?product=OBS**。

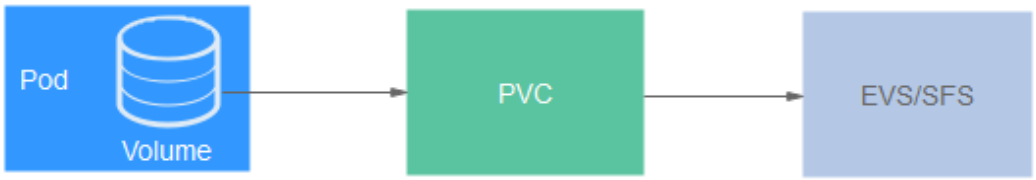
EVS和SFS的使用有个挂载的过程，下面将介绍如何使用EVS和SFS。

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13-1所示。

图 13-1 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SAS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-evs
  namespace: namespaces-test
  annotations: {
  paas.storage.io/cryptKeyId: ee9b610c-e356-11e9-aadc-d0efc1b3bb6b
  }
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 100Gi
  storageClassName: sas
```

  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：
  - 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
  - csi-sfs：SFS 3.0容量型弹性文件服务
- 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。

下面是的示例中说明了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sas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
```


14 使用 ConfigMap 和 Secret 提高配置灵活性

14.1 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
```

14.2 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
type: Opaque
data:
  key1:
VkZNME0wVlpVbEpQVHpGTFdrSkRWVWhCV2s5T1ZrNUxUVlZNUjBzMFRWcElVMFpVUkVWV1N3PT0=   # 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“VkZNME0wVlpVbEpQVHpGTFdrSkRWVWhCV2s5T1ZrNUxUVlZNUjBzMFRWcElVMFpVUkVWV1N3PT0=”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
```

15 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。批处理业务与长期伺服业务（Deployment、Statefulset）的主要区别是批处理业务的运行有头有尾，而长期伺服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π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π 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超期时间，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 15-1 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**