Cloud Container Engine

Kubernetes Basics

Issue 01

Date 2022-09-08

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions

Notice

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

Contents

1 Overview

Kubernetes is an open-source container orchestration system for automating containerized application deployment, scaling, and management across hosts in clouds.

For application developers, Kubernetes can be regarded as a cluster operating system. Kubernetes provides functions such as service discovery, scaling, load balancing, self-healing, and even leader election, freeing developers from infrastructure-related configurations.

You can use CCE, a hosted Kubernetes service provided by HUAWEI CLOUD, by means of the . Before using CCE, you are advised to learn about the following Kubernetes concepts.

Containers and Kubernetes

- **Container**
- **Kubernetes**

Pods, Labels, and Namespaces

- **Pod: the Smallest Scheduling Unit in Kubernetes**
- **Liveness Probe**
- **Label for Managing Pods**
- **Namespace for Grouping Resources**

Pod Orchestration and Scheduling

- **Deployment**
- **StatefulSet**
- **Job and Cron Job**
- **DaemonSet**
- **Affinity and Anti-Affinity Scheduling**

Configuration Management

- **ConfigMap**

- **Secret**

Kubernetes Networking

- **Container Networking**
- **Service**
- **Ingress**
- **Readiness Probe**
- **NetworkPolicy**

Persistent Storage

- **Volume**
- **PersistentVolume, PersistentVolumeClaim, and StorageClass**

Authentication and Authorization

- **ServiceAccount**
- **RBAC**

Auto Scaling

- **Auto Scaling**

2 Container and Kubernetes

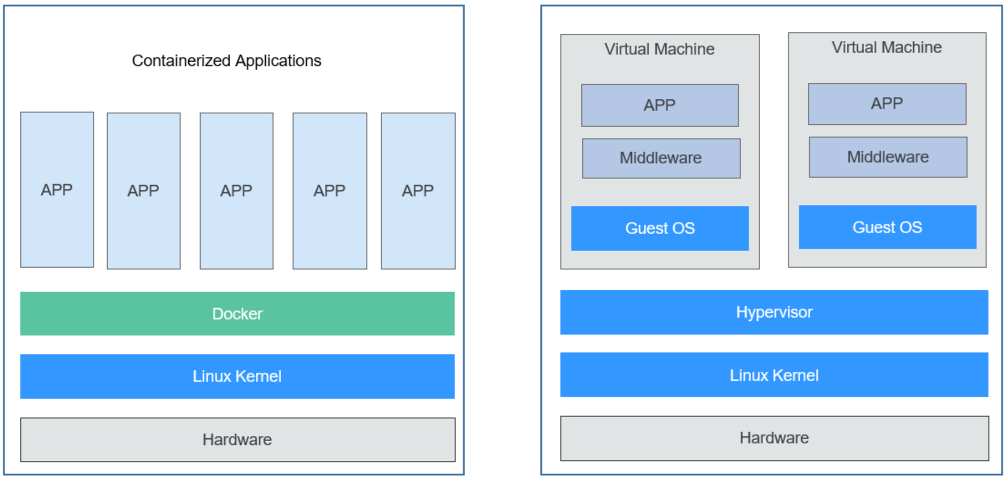
2.1 Container

Container and Docker

Container technologies originate from Linux. Containers provide lightweight virtualization, allow process and resource isolation, and become popular since the emergence of Docker. Docker is the first system that allows containers to be portable in different machines. It simplifies both the application packaging and the application library and dependency packaging. Even the OS file system can be packaged into a simple portable package, which can be used on any other machine that runs Docker.

Except for similar resource isolation and allocation modes as VMs, containers virtualize OSs, making them more portable and efficient.

Figure 2-1 Containers vs VMs



Containers have the following advantages over VMs:

- Higher system resource utilization

  With no overhead for virtualizing hardware and running a complete OS, containers outperform VMs no matter in application execution speed, memory loss, and file storage speed. Therefore, with same configurations, containers can run more applications than VMs.

- Faster startup

  Traditional VMs usually take several minutes to start an application. However, Docker containerized applications run directly on the host kernel with no need to start the entire OS, so they can start within seconds or even milliseconds, greatly saving your time in development, testing, and deployment.

- Consistent running environments

  One of the biggest problems in development is the inconsistency of application running environment. Due to inconsistent development, testing, and production environments, some bugs cannot be discovered prior to rollout. A Docker container image provides a complete runtime to ensure consistency in application running environments.

- Easier migration

  Docker ensures the consistency in execution environment, so migrating applications becomes much easier. Docker can run on many platforms, and no matter on physical machines or virtual ones, its running results remains the same. Therefore, you can easily migrate an application from one platform to another without worrying that the environment change will cause the applications fail to function.

- Easier maintenance and extension

  Tiered storage and image technology applied by Docker facilitate the reuse of applications and simplify application maintenance and update as well as further image extension based on base images. In addition, Docker collaborates with open-source project teams to maintain a large number of high-quality official images. You can directly use them in the production environment or form new images based on them, greatly reducing the image production cost of applications.

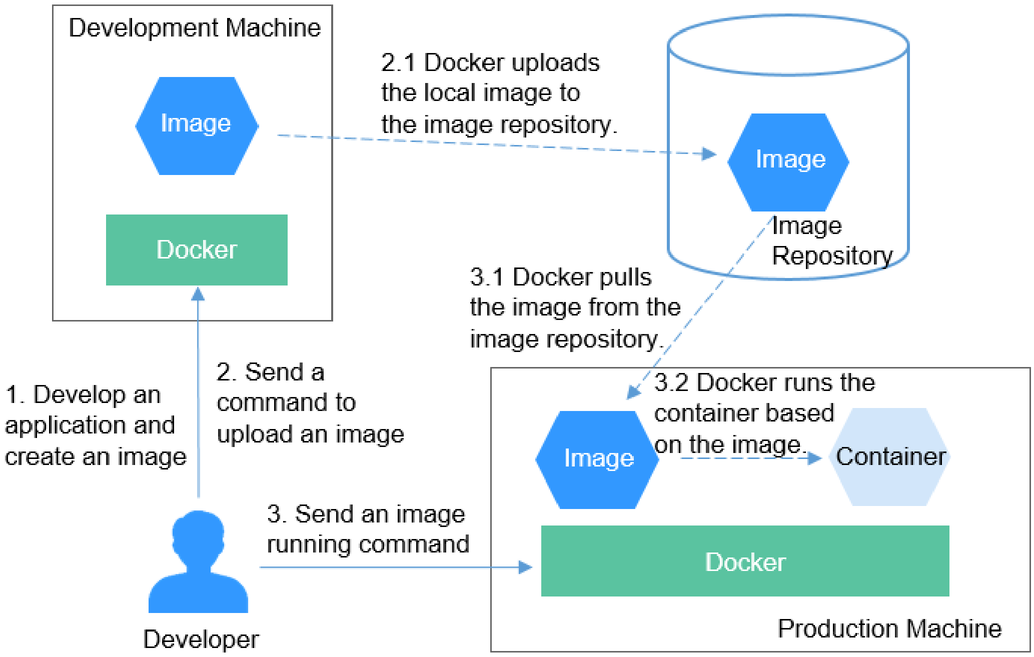
Typical Process of Using Docker Containers

Before using a Docker container, you should know the core components in Docker.

- **Image**: A Docker image is a software package that contains everything needed to run an application, such as the code and the runtime it requires, file systems, and executable file path of the runtime and other metadata.
- **Image repository**: A Docker image repository is used to store Docker images, which can be shared between different users and computers. You can run the image you compiled on the computer where it is compiled, or upload it to an image repository and then download it to another computer and run it. Some repositories are public, allowing everyone to pull images from them. Others are private, which are accessible only to some users and machines.
- **Container**: A Docker container is usually a Linux container created from a Docker image. A running container is a process running on the Docker host. However, it is isolated from the host and all other processes running on the host. The process is also resource-limited, meaning that it can access and use only resources (such as CPU and memory) allocated to it.

Figure 2-2 shows the typical process of using containers.

Figure 2-2 Typical process of using Docker containers



1. A developer develops an application and creates an image in the development machine.

   Docker runs the commands to create an image and store it on the machine.

2. The developer sends a command to upload the image.

   After receiving the command, Docker uploads the local image to the image repository.

3. The developer sends an image running command to the machine.

   After the command is received, Docker pulls the image from the image repository to the machine, and then runs a container based on the image.

Example

In the following example, Docker packages a container image based on the Nginx image, runs an application based on the container image, and pushes the image to the image repository.

Installing Docker

Docker is compatible with almost all operating systems. Select a Docker version that best suits your needs.

In Linux, you can run the following command to install Docker:

```
curl -fsSL get.docker.com -o get-docker.sh
sh get-docker.sh
systemctl restart docker
```

Packaging a Docker Image

Docker provides a convenient way to package your application, which is called Dockerfile.

```
# Use the official Nginx image as the base image.
FROM nginx:alpine

# Run a command to modify the content of the nginx image index.html.
RUN echo "hello world" > /usr/share/nginx/html/index.html

# Permit external access to port 80 of the container.
EXPOSE 80
```

Run the **docker build** command to package the image.

docker build -t *hello* **.**

In the preceding command, **-t** indicates that a tag is added to the image, that is, the image is named. In this example, the image name is **hello**. **.** indicates that the packaging command is executed in the current directory.

Run the **docker images** command to view the image. You can see the hello image has been created successfully. You can also see an Nginx image, which is downloaded from the image repository and used as the base image of the hello image.

```
# docker images
REPOSITORY          TAG              IMAGE ID           CREATED            SIZE
hello             latest           d120ec16dcea       17 minutes ago      158MB
nginx              alpine            eeb27ee6b893       2 months ago        148MB
```

Running the Container Image Locally

After obtaining the image, you can run the **docker run** command on the local host to run the container image.

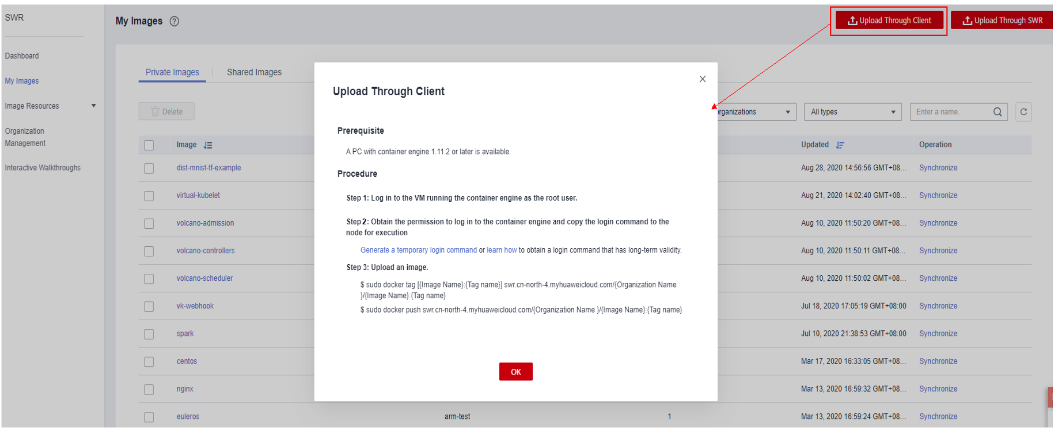
```
#docker run -p 8080:80 hello
```

The **docker run** command will start a container. In the preceding command, **-p** indicates that port 8080 of the local host is mapped to port 80 of the container. That is, the traffic of port 8080 of the local host will be forwarded to port 80 of the container. When you access http://127.0.0.1:8080 on the local host, you can access the container. In this case, the content returned by the browser is **hello world**.

Pushing the Image to the Image Repository

HUAWEI CLOUD provides SoftWare Repository for Container (SWR). You can also upload images to SWR. The following describes how to upload images to SWR. For details, see .

Log in to the . In the navigation pane, choose **My Images**. On the page that is displayed, click **Upload Through Client**. In the dialog box that is displayed, click **Generate a temporary login command**. Then, copy the command and run it on the local host to log in to the SWR image repository.



Before uploading an image, you need to specify a complete name for the image.

```
# docker tag hello swr.cn-east-3.myhuaweicloud.com/container/hello:v1
```

In the preceding command, **swr.cn-east-3.myhuaweicloud.com** indicates the repository address. The address varies depending on the HUAWEI CLOUD region. **v1** indicates the version number allocated to the hello image.

- **swr.cn-east-3.myhuaweicloud.com** indicates the repository address. The address varies with the HUAWEI CLOUD region.
- **container** is the organization name. Generally, an organization is created in SWR. If no organization is created, an organization is automatically created when the image is uploaded for the first time. The organization name is globally unique in a single region. You need to select a proper organization name.
- **v1** is the version number allocated to the hello image.

Run the **docker push** command to upload the image to SWR.

```
# docker push swr.cn-east-3.myhuaweicloud.com/container/hello:v1
```

If you need to use the image, run the **docker pull** command to pull (download) the image.

```
# docker pull swr.cn-east-3.myhuaweicloud.com/container/hello:v1
```

2.2 Kubernetes

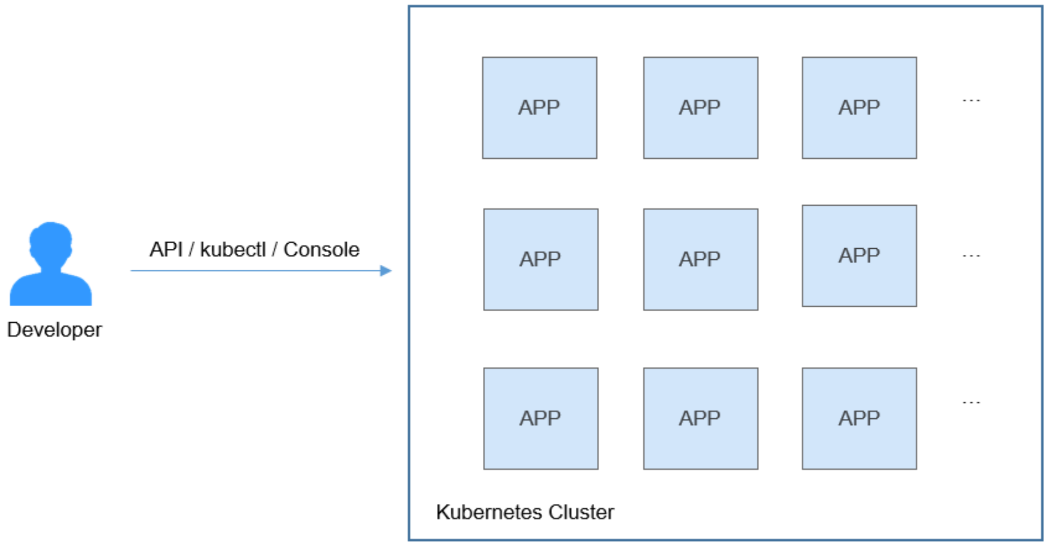
What Is Kubernetes?

Kubernetes is a containerized application software system that can be easily deployed and managed. It facilitates container scheduling and orchestration.

For application developers, Kubernetes can be regarded as a cluster operating system. Kubernetes provides functions such as service discovery, scaling, load balancing, self-healing, and even leader election, freeing developers from infrastructure-related configurations.

When using Kubernetes, it's like you run a large number of servers as one on which your applications run. Regardless of the number of servers in a Kubernetes cluster, the method for deploying applications in Kubernetes is always the same.

Figure 2-3 Running applications in a Kubernetes cluster

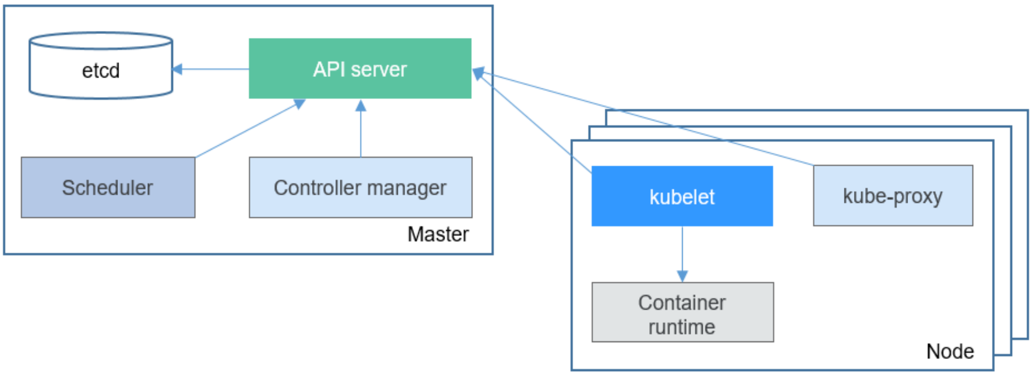


Kubernetes Cluster Architecture

A Kubernetes cluster consists of master nodes (masters) and worker nodes (nodes). Applications are deployed on worker nodes, and you can specify the nodes for deployment.

The following figure shows the architecture of a Kubernetes cluster.

Figure 2-4 Kubernetes cluster architecture



Master node

A master node is the machine where the control plane components run, including API server, Scheduler, Controller manager, and etcd.

- API server: functions as a transit station for components to communicate with each other, receives external requests, and writes information to etcd.
- Controller manager: performs cluster-level functions, such as component replication, node tracing, and node fault fixing.
- Scheduler: schedules containers to nodes based on various conditions (such as available resources and node affinity).

- etcd: serves as a distributed data storage component that stores cluster configuration information.

In the production environment, multiple master nodes are deployed to ensure cluster high availability. For example, you can deploy three master nodes for your CCE cluster.

Worker node

A worker node is a compute node in a cluster, that is, a node running containerized applications. A worker node has the following components:

- kubelet: communicates with the container runtime, interacts with the API server, and manages containers on the node.
- kube-proxy: serves as an access proxy between application components.
- Container runtime: functions as the software for running containers. You can download images to build your container runtime, such as Docker.

Kubernetes Scalability

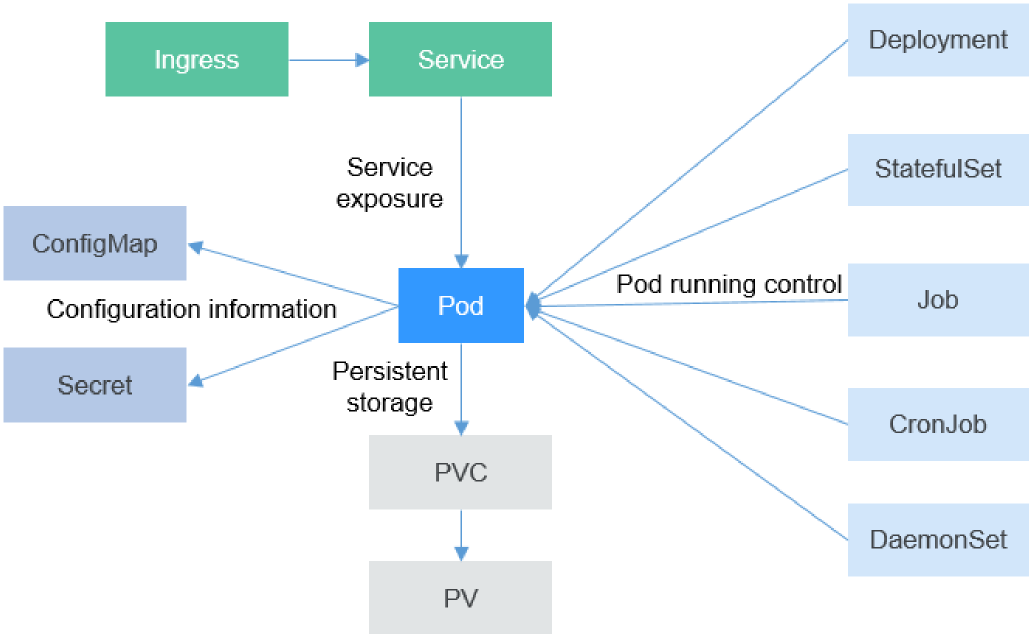
Kubernetes opens the Container Runtime Interface (CRI), Container Network Interface (CNI), and Container Storage Interface (CSI). These interfaces maximize Kubernetes scalability and allow Kubernetes to focus on container scheduling.

- Container Runtime Interface (CRI): provides computing resources when a container is running. It shields differences between container engines and interacts with each container engine through a unified interface.
- Container Network Interface (CNI): enables Kubernetes to support different networking implementations. For example, HUAWEI CLOUD CCE has developed customized CNI plug-ins that allow your Kubernetes clusters to run in HUAWEI CLOUD Virtual Private Cloud (VPC) networks.
- Container Storage Interface (CSI): enables Kubernetes to support various classes of storage. For example, HUAWEI CLOUD CCE can easily interconnect with HUAWEI CLOUD block storage (EVS), file storage (SFS), and object storage (OBS).

Basic Objects in Kubernetes

The following figure describes the basic objects in Kubernetes and the relationships between them.

Figure 2-5 Basic Kubernetes objects



- Pod

  A pod is the smallest and simplest unit that you create or deploy in Kubernetes. A pod encapsulates one or more containers, storage resources, a unique network IP address, and options that govern how the containers should run.

- Deployment

  A Deployment can be viewed as an application encapsulating pods. It can contain one or more pods. Each pod has the same role, and the system automatically distributes requests to the pods of a Deployment.

- StatefulSet

  A StatefulSet is used to manage stateful applications. Like Deployments, StatefulSets manage a group of pods based on an identical container spec. Where they differ is that StatefulSets maintain a fixed ID for each of their pods. These pods are created based on the same declaration but cannot replace each other. Each pod has a permanent ID regardless of how it is scheduled.

- Job

  A job is used to control batch tasks. Jobs are different from long-term servo tasks (such as Deployments). The former can be started and terminated at specific time, while the latter runs unceasingly unless it is terminated. Pods managed by a job will be automatically removed after successfully completing tasks based on user configurations.

- Cron job

  A cron job is a time-based job. Similar to the crontab of the Linux system, it runs a specified job in a specified time range.

- DaemonSet

  A DaemonSet runs a pod on each node in a cluster and ensures that there is only one pod. This works well for certain system-level applications, such as log collection and resource monitoring, since they must run on each node and need only a few pods. A good example is kube-proxy.

- Service

  A Service is used for pod access. With a fixed IP address, a Service forwards access traffic to pods and performs load balancing for these pods.

- Ingress

  Services forward requests based on Layer 4 TCP and UDP protocols. Ingresses can forward requests based on Layer 7 HTTPS and HTTPS protocols and make forwarding more targeted by domain names and paths.

- ConfigMap

  A ConfigMap stores configuration information in key-value pairs required by applications. With a ConfigMap, you can easily decouple configurations and use different configurations in different environments.

- Secret

  A secret lets you store and manage sensitive information, such as password, authentication information, certificates, and private keys. Storing confidential information in a secret is safer and more flexible than putting it verbatim in a pod definition or in a container image.

- PersistentVolume (PV)

  A PV describes a persistent data storage volume. It defines a directory for persistent storage on a host machine, for example, a mount directory of a network file system (NFS).

- PersistentVolumeClaim (PVC)

  Kubernetes provides PVCs to apply for persistent storage. With PVCs, you only need to specify the type and capacity of storage without concerning about how to create and release underlying storage resources.

Setting Up a Kubernetes Cluster

Kubernetes introduces multiple methods for setting up a Kubernetes cluster, such as minikube and kubeadm.

If you do not want to set up a Kubernetes cluster from scratch, you can buy one on the . The following operations will be performed on a purchased cluster.

kubectl

kubectl is a command line tool for Kubernetes clusters. You can install kubectl on any machine and run kubectl commands to operate your Kubernetes cluster.

For details about how to install kubectl, see . After connection, you can run the **kubectl cluster-info** command to view the cluster information, as shown below.

```
# kubectl cluster-info
Kubernetes master is running at https://*.*.*.*:5443
CoreDNS is running at https://*.*.*.*:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Run the **kubectl get nodes** command to view information about nodes in the cluster.

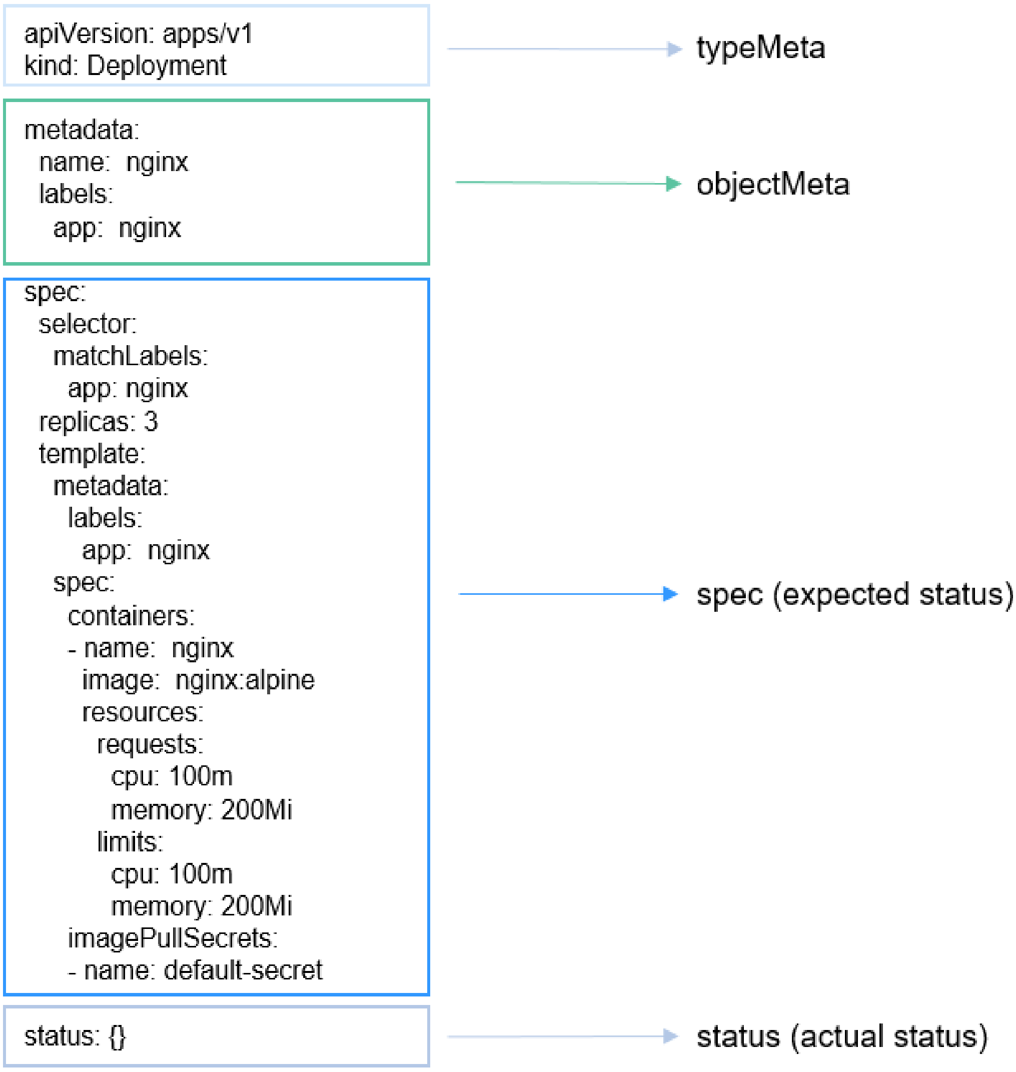

```
NAME           STATUS   ROLES    AGE    VERSION
192.168.0.153  Ready    <none>   7m     v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.207  Ready    <none>   7m     v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.221  Ready    <none>   7m     v1.15.6-r1-20.3.0.2.B001-15.30.2
```

Description of Kubernetes Objects

Resources in Kubernetes can be described in YAML or JSON format. An object description can be divided into the following four parts:

- typeMeta: metadata of the object type, specifying the API version and type of the object.
- objectMeta: metadata about the object, including the object name and used labels.
- spec: expected status of the object, for example, which image the object uses and how many replicas the object has.
- status: actual status of the object, which can be viewed only after the object is created. You do not need to specify the status when creating an object.

Figure 2-6 YAML description file



Running Applications on Kubernetes

Delete **status** from the content in **Figure 2-6** and save it as the **nginx-deployment.yaml** file, as shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  nginx
  labels:
    app:  nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
    app:  nginx
    spec:
      containers:
      - name:  nginx
        image:  nginx:alpine
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
          limits:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

Use kubectl to connect to the cluster and run the following command:

```
# kubectl create -f nginx-deployment.yaml
deployment.apps/nginx created
```

After the command is executed, three pods are created in the Kubernetes cluster. You can run the following command to query the Deployment and pods:

```
# kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   3/3     3            3           9s

# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-685898579b-qrt4d   1/1     Running   0          15s
nginx-685898579b-t9zd2   1/1     Running   0          15s
nginx-685898579b-w59jn   1/1     Running   0          15s
```

By now, we have walked you through the Kubernetes basics of containers and clusters, and provided you an example of how to use kubectl. The following sections will go deeper into Kubernetes objects, such as how they are used and related.

3 Pod, Label, and Namespace

3.1 Pod: the Smallest Scheduling Unit in Kubernetes

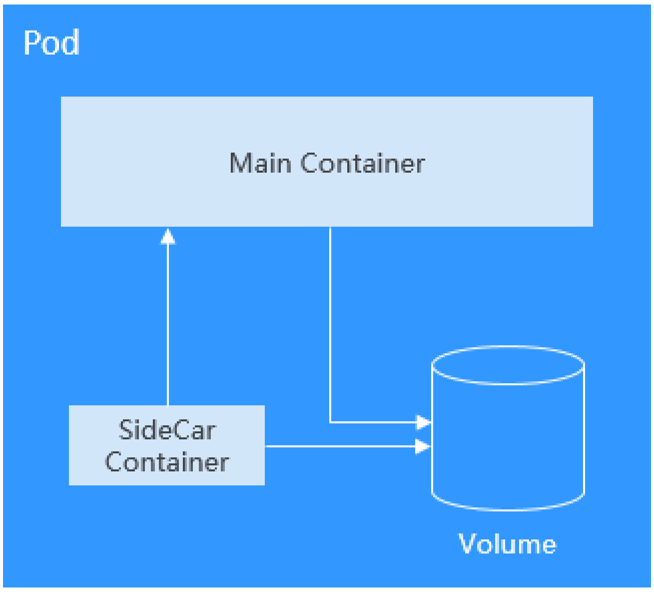
Pod

A pod is the smallest and simplest unit in the Kubernetes object model that you create or deploy. A pod encapsulates one or more containers, storage volumes, a unique network IP address, and options that govern how the containers should run.

Pods can be used in either of the following ways:

- A container is running in a pod. This is the most common usage of pods in Kubernetes. You can view the pod as a single encapsulated container, but Kubernetes directly manages pods instead of containers.
- Multiple containers that need to be coupled and share resources run in a pod. In this scenario, an application contains a main container and several sidecar containers, as shown in **Figure 3-1**. For example, the main container is a web server that provides file services from a fixed directory, and a sidecar container periodically downloads files to the directory.

Figure 3-1 Pod



In Kubernetes, pods are rarely created directly. Instead, controllers such as Deployments and jobs, are used to manage pods. Controllers can create and manage multiple pods, and provide replica management, rolling upgrade, and self-healing capabilities. A controller generally uses a pod template to create corresponding pods.

Creating a Pod

Kubernetes resources can be described using YAML or JSON files. The following example describes a pod named **nginx**. This pod contains a container named **container-0** and uses the **nginx:alpine** image, 100m CPU, and 200 MiB memory.

```
apiVersion: v1                      # Kubernetes API version
kind: Pod                          # Kubernetes resource type
metadata:
  name: nginx                       # Pod name
spec:                              # Pod specifications
  containers:
  - image: nginx:alpine               # The image used is nginx:alpine.
    name: container-0                 # Container name
    resources:                     # Resources required for a container
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
  imagePullSecrets:                  # Secret used to pull the image, which must be default-secret on CCE
  - name: default-secret
```

As shown in the annotation of YAML, the YAML description file includes:

- **metadata**: information such as name, label, and namespace
- **spec**: pod specification such as image and volume used

If you query a Kubernetes resource, you can see the **status** field. This field indicates the status of the Kubernetes resource, and does not need to be set when the resource is created. This example is a minimum set. Other parameter definition will be described later.

After the pod is defined, you can create it using kubectl. Assume that the preceding YAML file is named **nginx.yaml**, run the following command to create the file. **-f** indicates that it is created in the form of a file.

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

After the pod is created, you can run the **kubectl get pods** command to query the pod information, as shown below.

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

The preceding information indicates that the **nginx** pod is in the **Running** state, indicating that the pod is running. **READY** is **1/1**, indicating that there is one container in the pod, and the container is in the **Ready** state.

You can run the **kubectl get** command to query the configuration information about a pod. In the following command, **-o yaml** indicates that the pod is returned in YAML format. **-o json** indicates that the pod is returned in JSON format.

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

You can also run the **kubectl describe** command to view the pod details.

```
$ kubectl describe pod nginx
```

When a pod is deleted, Kubernetes stops all containers in the pod. Kubernetes sends the SIGTERM signal to the process and waits for a period (30 seconds by default) to stop the container. If it is not stopped within the period, Kubernetes sends a SIGKILL signal to kill the process.

You can stop and delete a pod in multiple methods. For example, you can delete a pod by name, as shown below.

```
$ kubectl delete po nginx
pod "nginx" deleted
```

Delete multiple pods at one time.

```
$ kubectl delete po pod1 pod2
```

Delete all pods.

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

Delete pods by labels. For details about labels, see **Labels: Managing Pods**.

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

Environment Variables

Environment variables are set in the container running environment.

Environment variables add flexibility to workload configuration. The environment variables for which you have assigned values during container creation will take effect when the container is running. This saves you the trouble of rebuilding the container image.

The following shows how to use an environment variable. You only need to configure the **spec.containers.env** field.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
    containers:
    - image: nginx:alpine
      name: container-0
      resources:
        limits:
          cpu: 100m
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      env:                          # Environment variable
      - name: env_key
        value: env_value
    imagePullSecrets:
    - name: default-secret
```

Run the following command to check the environment variables in the container. The value of the **env_key** environment variable is **env_value**.

```
$ kubectl exec -it nginx -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=nginx
TERM=xterm
env_key=env_value
```

Environment variables can also reference **ConfigMap** and **secret**. For details, see **Referencing a ConfigMap as an Environment Variable** and **Referencing a Secret as an Environment Variable**.

Setting Container Startup Commands

Starting a container is to start the main process. Some preparations must be made before the main process is started. For example, you may configure or initialize MySQL databases before running MySQL servers. You can set **ENTRYPOINT** or **CMD** in the Dockerfile when creating an image. As shown in the following example, the **ENTRYPOINT ["top", "-b"]** command is set in the Dockerfile. This command will be executed during container startup.

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

When calling an API, you only need to configure the **containers.command** field of the pod. This field is of the list type. The first parameter in the field is the command to be executed, and the subsequent parameters are the command arguments.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
```



```
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    command:                   # Startup command
    - top
    - "-b"
  imagePullSecrets:
  - name: default-secret
```

Container Lifecycle

Kubernetes provides **container lifecycle hooks**. The hooks enable containers to run code triggered by events during their management lifecycle. For example, if you want a container to perform a certain operation before it is stopped, you can register a hook. The following lifecycle hooks are provided:

- **postStart**: triggered immediately after the workload is started
- **preStop**: triggered immediately before the workload is stopped

You only need to set the **lifecycle.postStart** or **lifecycle.preStop** parameter of the pod, as shown in the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    lifecycle:
      postStart:               # Post-start processing
        exec:
          command:
          - "/postStart.sh"
      preStop:                 # Pre-stop processing
        exec:
          command:
          - "/preStop.sh"
  imagePullSecrets:
  - name: default-secret
```

3.2 Liveness Probe

Overview

Kubernetes applications have the self-healing capability, that is, when an application container crashes, the container can be detected and restarted automatically. However, this mechanism does not work for deadlocks. Assume that a Java program is having a memory leak. The program is unable to make any progress, while the JVM process is running. To address this issue, Kubernetes introduces liveness probes to check whether containers response normally and determine whether to restart containers. This is a good health check mechanism.

It is advised to define the liveness probe for every pod to gain a better understanding of pods' running statuses.

Supported detection mechanisms are as follows:

- HTTP GET: The kubelet sends an HTTP GET request to the container. Any 2XX or 3XX code indicates success. Any other code returned indicates failure.
- TCP Socket: The kubelet attempts to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy. If it fails to establish a connection, the container is considered a failure.
- Exec: kubelet executes a command in the target container. If the command succeeds, it returns **0**, and kubelet considers the container to be alive and healthy. If the command returns a non-zero value, kubelet kills the container and restarts it.

In addition to liveness probes, readiness probes are also available for you to detect pod status. For details, see **Readiness Probe**.

HTTP GET

HTTP GET is the most common detection method. An HTTP GET request is sent to a container. Any 2xx or 3xx code returned indicates that the container is healthy. The following example shows how to define such a request:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:          # liveness probe
      httpGet:              #HTTP GET definition
        path: /
        port: 80
  imagePullSecrets:
  - name: default-secret
```

Create pod **liveness-http**.

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

The probe sends an HTTP Get request to port 80 of the container. If the request fails, Kubernetes restarts the container.

View details of pod **liveness-http**.

```
$ kubectl describe po liveness-http
Name:           liveness-http
......
Containers:
  liveness:
    ......
    State:        Running
      Started:     Mon, 03 Aug 2020 03:08:55 +0000
    Ready:         True
    Restart Count:  0
    Liveness:      http-get http://:80/ delay=0s timeout=1s period=10s #success=1 #failure=3
    Environment:    <none>
    Mounts:
```

```
   /var/run/secrets/kubernetes.io/serviceaccount from default-token-vssmw (ro)
......
```

The preceding output reports that the pod is **Running** with **Restart Count** being **0**, which indicates that the container is normal and no restarts have been triggered. If the value of **Restart Count** is not **0**, the container has been restarted.

TCP Socket

TCP Socket: The kubelet attempts to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy. If it fails to establish a connection, the container is considered a failure. For detailed defining method, see the following example.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:           # liveness probe
      tcpSocket:
        port: 80
  imagePullSecrets:
  - name: default-secret
```

Exec

kubelet executes a command in the target container. If the command succeeds, it returns **0**, and kubelet considers the container to be alive and healthy. The following example shows how to define the command.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:           # liveness probe
      exec:                  # Exec definition
        command:
        - cat
        - /tmp/healthy
  imagePullSecrets:
  - name: default-secret
```

In the preceding configuration file, kubelet executes the command **cat /tmp/healthy** in the container. If the command succeeds and returns **0**, the container is considered healthy. For the first 30 seconds, there is a **/tmp/healthy** file. So during the first 30 seconds, the command **cat /tmp/healthy** returns a success code. After 30 seconds, the **/tmp/healthy** file is deleted. The probe will then consider the pod to be unhealthy and restart it.

Advanced Settings of a Liveness Probe

The **describe** command of **liveness-http** returns the following information:

```
Liveness: http-get http://:80/ delay=0s timeout=1s period=10s #success=1 #failure=3
```

This is the detailed configuration of the liveness probe.

- **delay=0s** indicates that the probe starts immediately after the container is started.
- **timeout=1** indicates that the container must respond within one second. Otherwise, the health check is recorded as failed.
- **period=10s** indicates that the probe checks containers every 10 seconds.
- **#success=1** indicates that the operation is recorded as successful if it is successful for once.
- **#failure=3** indicates that a container will be restarted after three consecutive failures.

The preceding liveness probe indicates that the probe checks containers immediately after they are started. If a container does not respond within one second, the check is recorded as failed. The health check is performed every 10 seconds. If the check fails for three consecutive times, the container is restarted.

These are the default configurations when the probe is created. You can customize them as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10    # Liveness probes are initiated after the container has started for 10s.
      timeoutSeconds: 2          # The container must respond within 2s. Otherwise, it is considered as a 
failure.
      periodSeconds: 30          # The probe is performed every 30s.
      successThreshold: 1        # The container is considered healthy as long as the probe succeeds once.
      failureThreshold: 3       # The container is considered unhealthy after three consecutive failures.
```

Normally, the value of **initialDelaySeconds** must be greater than **0**, because it takes a while for the application to be ready. The probe often fails if the probe is initiated before the application is ready.

In addition, you can set the value of **failureThreshold** to be greater than **1**. In this way, the kubelet checks the container for multiple times in one probe rather than performing the probe for multiple times.

Configuring a Liveness Probe

- **What to check**

  An effective liveness probe should check all the key parts of an application and use a dedicated URL, such as **/health**. When the URL is accessed, the probe is triggered and a result is returned. Note that no authentication should be involved. Otherwise, the probe keeps failing and restarting the container.

In addition, a probe must not check parts that have external dependencies. For example, if a frontend web server cannot connect to a database, the web server should not be considered unhealthy for the connection failure.

- **To be lightweight**

  A liveness probe must not occupy too many resources or certain resources for too long. Otherwise, resource shortage may affect service running. For example, the HTTP GET method is recommended for a Java application. If the Exec method is used, the JVM startup process occupies too many resources.

3.3 Label for Managing Pods

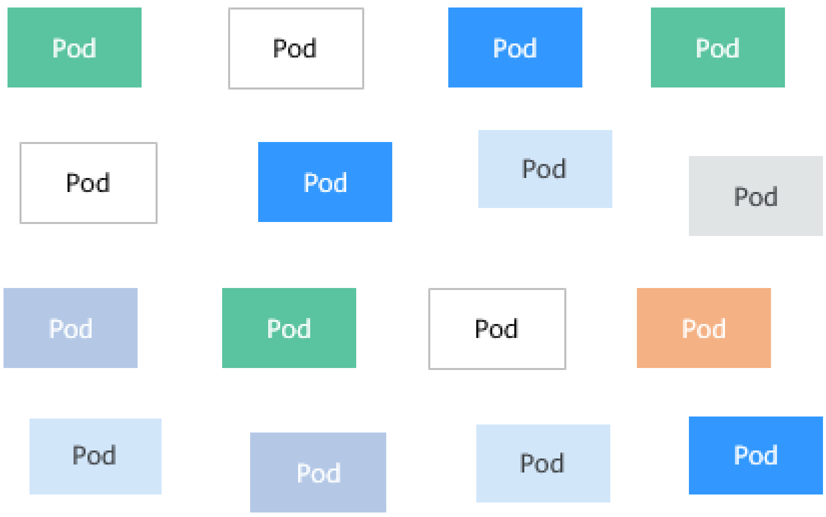
Why We Need Labels

As resources increase, managing resources becomes essential. Labels allow you to easily and efficiently manage almost all the resources in Kubernetes.

A label is a key-value pair. It can be set either during or after resource creation. You can easily modify it when needed at any time.

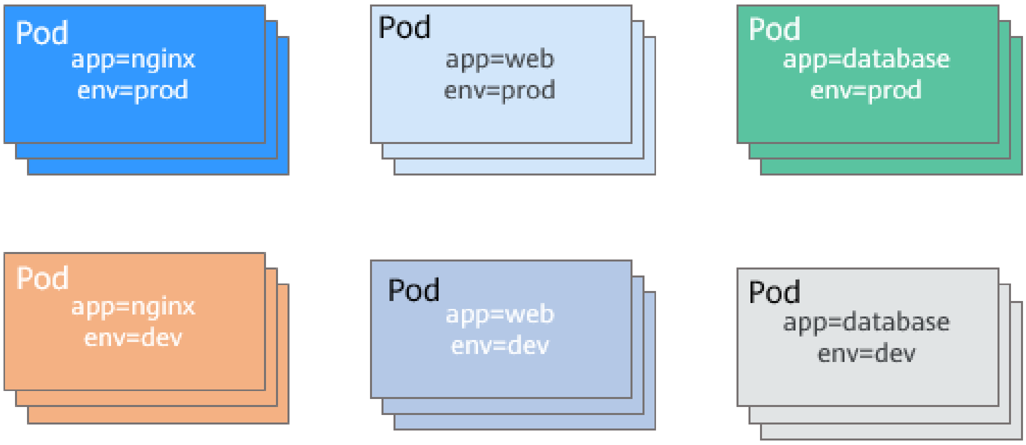
The following figures show how labels work. Assume that you have multiple pods of various kinds. It could be challenging when you manage them.

Figure 3-2 Pods without classification



After we add labels to them. It is much clearer.

Figure 3-3 Pods classified using labels



Adding a Label

The following example shows how to add labels when you are creating a pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:                # Add labels app=nginx and env=prod to the pod.
    app: nginx
    env: prod
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
  imagePullSecrets:
  - name: default-secret
```

After you add labels to a pod, you can view the labels by adding **--show-labels** when querying the pod.

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

You can also use **-L** to query only certain labels.

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

For an existing pod, you can run the **kubectl label** command to add labels.

```
$ kubectl label pod nginx creation_method=manual
pod/nginx labeled

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


Modifying a Label

Add **--overwrite** to the command to modify a label.

```
$ kubectl label pod nginx env=debug --overwrite
pod/nginx labeled

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

3.4 Namespace for Grouping Resources

Why We Need Namespaces

Although labels are simple and efficient, too many labels can cause chaos and make querying inconvenient. Labels can overlap with each other, which is not suitable for certain scenarios. This is where namespace comes in. Namespaces allow you to isolate and manage resources in a more systematic way. Multiple namespaces can divide systems that contain multiple components into different non-overlapped groups. Namespaces also enable you to divide cluster resources between users. In this way, multiple teams can share one cluster.

Resources of the same type can share the same name as long as they are in different namespaces. Unlike most resources in Kubernetes can be managed by namespace, global resources do not belong to a specific namespace. Later sections will discuss this topic in detail.

Run the following command to query namespaces in the current cluster:

```
$ kubectl get ns
NAME              STATUS   AGE
default           Active   36m
kube-node-realease Active   36m
kube-public       Active   36m
kube-system       Active   36m
```

By now, we are performing operations in the default namespace. When **kubectl get** is used but no namespace is specified, the default namespace is used by default.

You can run the following command to view resources in namespace **kube-system**.

```
$ kubectl get po --namespace=kube-system
NAME                                      READY   STATUS    RESTARTS   AGE
coredns-7689f8bdf-295rk                   1/1     Running   0          9m11s
coredns-7689f8bdf-h7n68                   1/1     Running   0          11m
everest-csi-controller-6d796fb9c5-v22df   2/2     Running   0          9m11s
everest-csi-driver-snzrr                  1/1     Running   0          12m
everest-csi-driver-ttj28                  1/1     Running   0          12m
everest-csi-driver-wtrk6                  1/1     Running   0          12m
icagent-2kz8g                             1/1     Running   0          12m
icagent-hjz4h                             1/1     Running   0          12m
icagent-m4bbl                             1/1     Running   0          12m
```

You can see that there are many pods in **kube-system**. **coredns** is used for service discovery, **everest-csi** for connecting with HUAWEI CLOUD storage services, and **icagent** for connecting with HUAWEI CLOUD monitoring system.

These general, must-have applications are put in the **kube-system** namespace to isolate them from other pods. They are invisible to and free from being affected by resources in other namespaces.

Creating a Namespace

Define a namespace.

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

Run the **kubectl** command to create it.

```
$ kubectl create -f custom-namespace.yaml
namespace/custom-namespace created
```

You can also run the **kubectl create namespace** command to create a namespace.

```
$ kubectl create namespace custom-namespace
namespace/custom-namespace created
```

Create resources in the namespace.

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

By now, **custom-namespace** has a pod named **nginx**.

The Isolation function of Namespaces

Namespaces are used to group resources only for organization purposes. Running objects in different namespaces are not essentially isolated. For example, if pods in two namespaces know the IP address of each other and the underlying network on which Kubernetes depends does not provide network isolation between namespaces, the two pods can access each other.

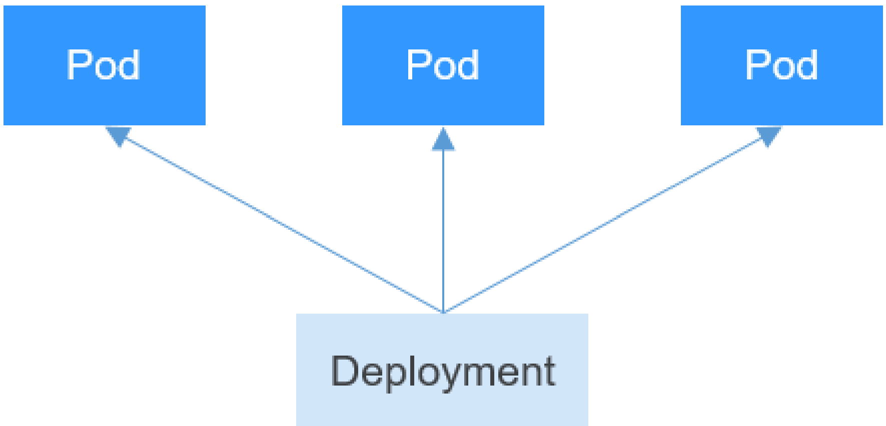
4 Pod Orchestration and Scheduling

4.1 Deployment

Deployment

A pod is the smallest and simplest unit that you create or deploy in Kubernetes. It is designed to be an ephemeral, one-off entity. A pod can be evicted when node resources are insufficient and disappears along with a cluster node failure. Kubernetes provides controllers to manage pods. Controllers can create and manage pods, and provide replica management, rolling upgrade, and self-healing capabilities. The most commonly used controller is Deployment.

Figure 4-1 Relationship between a Deployment and pods



A Deployment can contain one or more pods. These pods have the same role. Therefore, the system automatically distributes requests to multiple pods of a Deployment.

A Deployment integrates a lot of functions, including online deployment, rolling upgrade, replica creation, and restoration of online jobs. To some extent, Deployments can be used to realize unattended rollout, which greatly reduces difficulties and operation risks in the rollout process.

Creating a Deployment

In the following example, a Deployment named **nginx** is created, and two pods are created from the **nginx:latest** image. Each pod occupies 100m CPU and 200 MiB memory.

```
apiVersion: apps/v1      # Note the difference with a pod. It is apps/v1 instead of v1 for a Deployment.
kind: Deployment         # The resource type is Deployment.
metadata:
  name: nginx            # Name of the Deployment
spec:
  replicas: 2            # Number of pods. The Deployment ensures that two pods are running.
  selector:              # Label Selector
    matchLabels:
      app: nginx
  template:              # Definition of a pod, which is used to create pods. It is also known as pod template.
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

In this definition, the name of the Deployment is **nginx**, and **spec.replicas** defines the number of pods. That is, the Deployment controls two pods. **spec.selector** is a label selector, indicating that the Deployment selects the pod whose label is **app=nginx**. **spec.template** is the definition of the pod and is the same as that defined in **Pods**.

Save the definition of the Deployment to **deployment.yaml** and use kubectl to create the Deployment.

Run **kubectl get** to view the Deployment and pods. In the following example, the value of **READY** is **2/2**. The first **2** indicates that two pods are running, and the second 2 indicates that two pods are expected in this Deployment. The value **2** of **AVAILABLE** indicates that two pods are available.

```
$ kubectl create -f deployment.yaml
deployment.apps/nginx created

$ kubectl get deploy
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
nginx    2/2     2            2           4m5s
```

How Does the Deployment Control Pods?

Continue to query pods, as shown below.

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

If you delete a pod, a new pod is immediately created, as shown below. As mentioned above, the Deployment ensures that there are two pods running. If a pod is deleted, the Deployment creates a new pod. If a pod becomes faulty, the Deployment automatically restarts the pod.

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

You see two pods, **nginx-7f98958cdf-tdmqk** and **nginx-7f98958cdf-tesqr**. **nginx** is the name of the Deployment. **-7f98958cdf-tdmqk** and **-7f98958cdf-tesqr** are the suffixes randomly generated by Kubernetes.

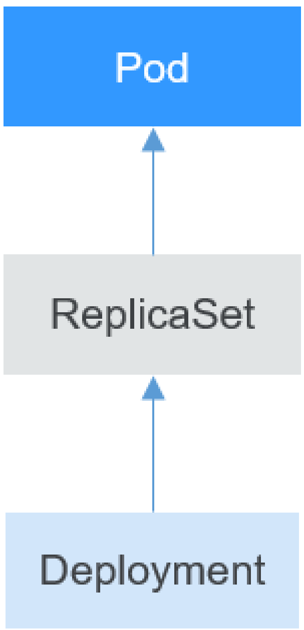
You may notice that the two suffixes share the same content **7f98958cdf** in the first part. This is because the Deployment does not control the pods directly, but through a controller named ReplicaSet. You can run the following command to query the ReplicaSet. In the command, **rs** is the abbreviation of ReplicaSet.

```
$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-7f98958cdf   2         2         2         1m
```

The ReplicaSet is named **nginx-7f98958cdf**, in which the suffix **-7f98958cdf** is generated randomly.

As shown in **Figure 4-2**, the Deployment controls the ReplicaSet, which then controls pods.

Figure 4-2 How does the Deployment control the pod



If you run the **kubectl describe** command to view the details of the Deployment, you can see the ReplicaSet (**NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)**). In **Events**, the number of pods of the ReplicaSet is scaled out to 2. In practice, you may not operate ReplicaSet directly, but understanding that a Deployment controls a pod by controlling a ReplicaSet helps you locate problems.

```
$ kubectl describe deploy nginx
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
```

Upgrade

In actual applications, upgrade is a common operation. A Deployment can easily support application upgrade.

You can set different upgrade policies for a Deployment:

- **RollingUpdate**: New pods are created gradually and then old pods are deleted. This is the default policy.
- **Recreate**: The current pods are deleted and then new pods are created.

The Deployment can be upgraded in a declarative mode. That is, you only need to modify the YAML definition of the Deployment. For example, you can run the **kubectl edit** command to change the Deployment image to **nginx:alpine**. After the modification, query the ReplicaSet and pod. The query result shows that a new ReplicaSet is created and the pod is re-created.

```
$ kubectl edit deploy nginx

$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-6f9f58dffd   2         2         2         1m
nginx-7f98958cdf   0         0         0         48m

$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-6f9f58dffd-tdmqk   1/1       Running   0          1m
nginx-6f9f58dffd-tesqr   1/1       Running   0          1m
```

The Deployment can use the **maxSurge** and **maxUnavailable** parameters to control the proportion of pods to be re-created during the upgrade, which is useful in many scenarios. The configuration is as follows:

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

- **maxSurge** specifies the maximum number of pods that can exist over **spec.replicas** in the Deployment. The default value is 25%. For example, if **spec.replicas** is set to **4**, no more than 5 pods can exist during the upgrade process, that is, the upgrade step is 1. The absolute number is calculated from the percentage by rounding up. The value can also be set to an absolute number.
- **maxUnavailable**: specifies the maximum number of pods that can be unavailable during the update process. The default value is 25%. For example, if **spec.replicas** is set to **4**, at least 3 pods exist during the upgrade process, that is, the deletion step is 1. The value can also be set to an absolute number.

In the preceding example, the value of **spec.replicas** is **2**. If both **maxSurge** and **maxUnavailable** are the default value 25%, **maxSurge** allows a maximum of three pods to exist (2 x 1.25 = 2.5, rounded up to 3), and **maxUnavailable** does not allow a maximum of two pods to be unavailable (2 x 0.75 = 1.5, rounded up to 2). That is, during the upgrade process, there will always be two pods running. Each time a new pod is created, an old pod is deleted, until all pods are new.

Rollback

Rollback is to roll an application back to the earlier version when a fault occurs during the upgrade. A Deployment can be easily rolled back to the earlier version.

For example, if the upgraded image is faulty, you can run the **kubectl rollout undo** command to roll back the Deployment.

```
$ kubectl rollout undo deployment nginx
deployment.apps/nginx rolled back
```

A Deployment can be easily rolled back because it uses a ReplicaSet to control a pod. After the upgrade, the previous ReplicaSet still exists. The Deployment is rolled back by using the previous ReplicaSet to re-create the pod. The number of ReplicaSets stored in a Deployment can be restricted by the **revisionHistoryLimit** parameter. The default value is 10.

4.2 StatefulSet

StatefulSet

All pods under a Deployment have the same characteristics except for the name and IP address. If required, a Deployment can use the pod template to create a new pod. If not required, the Deployment can delete any one of the pods.

However, Deployments cannot meet the requirements in some distributed scenarios when each pod requires its own status or in a distributed database where each pod requires independent storage.

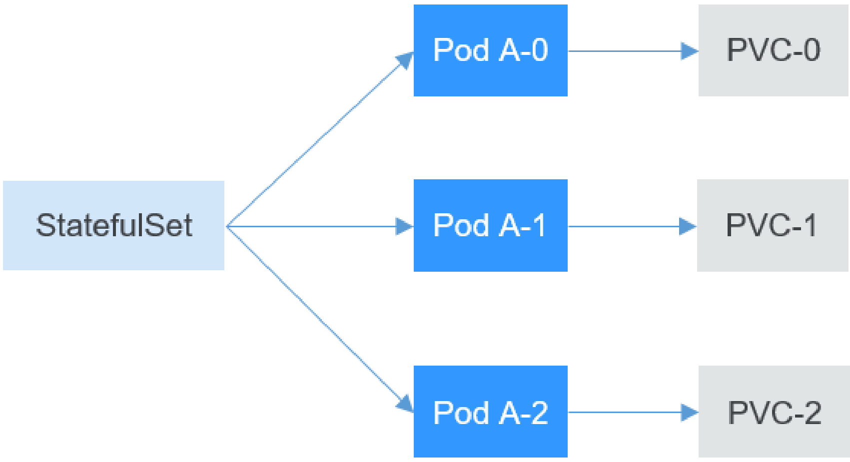
With detailed analysis, it is found that each part of distributed stateful applications plays a different role. For example, the database nodes are deployed in active/standby mode, and pods are dependent on each other. In this case, you need to meet the following requirements for the pods:

- A pod can be recognized by other pods. Therefore, a pod must have a fixed identifier.
- Each pod has an independent storage device. After a pod is deleted and then restored, the data read from the pod must be the same as the previous one. Otherwise, the pod status is inconsistent.

To address the preceding requirements, Kubernetes provides StatefulSets.

1. A StatefulSet provides a fixed name for each pod following a fixed number ranging from 0 to N. After a pod is rescheduled, the pod name and the host name remain unchanged.
2. A StatefulSet provides a fixed access domain name for each pod through the headless Service (described in following sections).

3. The StatefulSet creates PersistentVolumeClaims (PVCs) with fixed identifiers to ensure that pods can access the same persistent data after being rescheduled.



The following describes how to create a StatefulSet and experience its features.

Creating a Headless Service

As described above, a headless Service is required for pod access when a StatefulSet is created. For details about the Service, see **Service**. The following describes how to create a headless Service.

Use the following file to describe the headless Service:

- **spec.clusterIP**: Set it to **None**, which indicates a headless Service is to be created.
- **spec.ports.port**: indicates the number of the port used for communication between pods.
- **spec.ports.name**: indicates the name of the port used for communication between pods.

```
apiVersion: v1
kind: Service       # The object type is Service.
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Name of the port for communication between pods
      port: 80        # Number of the port for communication between pods
  selector:
    app: nginx        # Select the pod whose label is app:nginx.
  clusterIP: None     # Set this parameter to None, indicating that a headless Service is to be created.
```

Run the following command to create a headless Service:

```
# kubectl create -f headless.yaml
service/nginx created
```

After the Service is created, you can query the Service information.

```
# kubectl get svc
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx      ClusterIP   None         <none>        80/TCP    5s
```

Creating a StatefulSet

The YAML definition of StatefulSets is basically the same as that of other objects. The differences are as follows:

- **serviceName** specifies the headless Service used by the StatefulSet. You need to specify the name of the headless Service.
- **volumeClaimTemplates** is used to apply for a **PVC**. A template named **data** is defined, which will create a PVC for each pod. **storageClassName** specifies the persistent storage class. For details, see **PersistentVolume, PersistentVolumeClaim, and StorageClass**. **volumeMounts** is used to mount storage to pods. If no storage is required, you can delete the **volumeClaimTemplates** and **volumeMounts** fields.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                             # Name of the headless Service
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:                          # Storage mounted to the pod
          - name:  data
            mountPath: /usr/share/nginx/html     # Mount the storage to /usr/share/nginx/html.
      imagePullSecrets:
        - name: default-secret
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
      - ReadWriteMany
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-nas                  # Persistent storage class
```

Run the following command to create a StatefulSet:

```
# kubectl create -f statefulset.yaml
statefulset.apps/nginx created
```

After the command is executed, query the StatefulSet and pods. The suffix of the pod names starts from 0 and increases to 2.

```
# kubectl get statefulset
NAME    READY   AGE
nginx   3/3     107s
```

```
# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          112s
nginx-1   1/1     Running   0          69s
nginx-2   1/1     Running   0          39s
```

In this case, if you manually delete the **nginx-1** pod and query the pods again, you can see that a pod with the same name is created. According to **5s** under **AGE**, it is found that the **nginx-1** pod is newly created.

```
# kubectl delete pod nginx-1
pod "nginx-1" deleted

# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          3m4s
nginx-1   1/1     Running   0          5s
nginx-2   1/1     Running   0          1m10s
```

Access the container and check its host names. The host names are **nginx-0**, **nginx-1**, and **nginx-2**.

```
# kubectl exec nginx-0 -- sh -c 'hostname'
nginx-0
# kubectl exec nginx-1 -- sh -c 'hostname'
nginx-1
# kubectl exec nginx-2 -- sh -c 'hostname'
nginx-2
```

In addition, you can view the PVCs created by the StatefulSet. These PVCs are named in the format of ***PVC name*-*StatefulSet name*-*No.*** and are in the **Bound** state.

```
# kubectl get pvc
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-nginx-0   Bound    pvc-f58bc1a9-6a52-4664-a587-a9a1c904ba29   1Gi        RWX            csi-nas        2m24s
data-nginx-1   Bound    pvc-066e3a3a-fd65-4e65-87cd-6c3fd0ae6485   1Gi        RWX            csi-nas        101s
data-nginx-2   Bound    pvc-a18cf1ce-708b-4e94-af83-766007250b0c   1Gi        RWX            csi-nas        71s
```

Network Identifier of a StatefulSet

After a StatefulSet is created, you can see that each pod has a fixed name. The headless Service provides a fixed domain name for pods by using DNS. In this way, pods can be accessed using the domain name. Even if the IP address of the pod changes when the pod is re-created, the domain name remains unchanged.

After a headless Service is created, the IP address of each pod corresponds to a domain name in the following format:

<pod-name>.<svc-name>.<namespace>.svc.cluster.local

For example, the domain names of the three pods are as follows:

- nginx-0.nginx.default.svc.cluster.local
- nginx-1.nginx.default.svc.cluster.local
- nginx-2.nginx.default.svc.cluster.local

In actual access, **.<namespace>.svc.cluster.local** can be omitted.

Create a pod from the **tutum/dnsutils** image. Then, access the container of the pod and run the **nslookup** command to view the domain name of the pod. The IP address of the pod can be parsed. The IP address of the DNS server is **10.247.3.10**. When a CCE cluster is created, the coredns add-on is installed by default to provide the DNS service. The functions of coredns will be described in **Kubernetes Networking**.

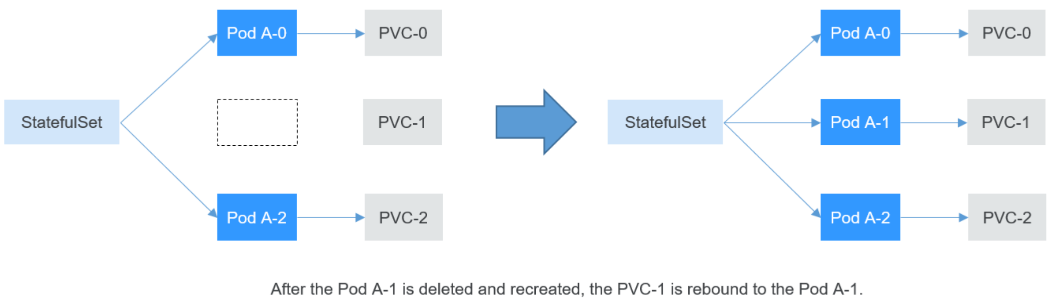
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # nslookup nginx-1.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-1.nginx.default.svc.cluster.local
Address: 172.16.0.18

/ # nslookup nginx-2.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-2.nginx.default.svc.cluster.local
Address: 172.16.0.19
```

In this case, if you manually delete the two pods, query the IP addresses of the pods re-created by the StatefulSet, and run the **nslookup** command to resolve the domain names of the pods, you can still get **nginx-0.nginx** and **nginx-1.nginx**. This ensures that the network identifier of the StatefulSet remains unchanged.

StatefulSet Storage Status

As mentioned above, StatefulSets can use PVCs for persistent storage to ensure that the same persistent data can be accessed after pods are rescheduled. When pods are deleted, PVCs are not deleted.

Figure 4-3 Process for a StatefulSet to re-create a pod



Run the following command to write some data into the **/usr/share/nginx/html** directory of **nginx-1**. For example, change the content of **index.html** to **hello world**.

```
# kubectl exec nginx-1 -- sh -c 'echo hello world > /usr/share/nginx/html/index.html'
```

After the modification, if you access **https://localhost**, **hello world** is returned.

```
# kubectl exec -it nginx-1 -- curl localhost
hello world
```

In this case, if you manually delete the **nginx-1** pod and query the pods again, you can see that a pod with the same name is created. According to **4s** under **AGE**, it is found that the **nginx-1** pod is newly created.

```
# kubectl delete pod nginx-1
pod "nginx-1" deleted

# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          14m
nginx-1   1/1     Running   0          4s
nginx-2   1/1     Running   0          13m
```

Access the **index.html** page of the pod again. **hello world** is still returned, which indicates that the same storage medium is accessed.

```
# kubectl exec -it nginx-1 -- curl localhost
hello world
```

4.3 Job and Cron Job

Job and Cron Job

Jobs and cron jobs allow you to run short lived, one-off tasks in batch. They ensure the task pods run to completion.

- A job is a resource object used by Kubernetes to control batch tasks. Jobs are different from long-term servo tasks (such as Deployments and StatefulSets). The former is started and terminated at specific times, while the latter runs unceasingly unless being terminated. The pods managed by a job will be automatically removed after successfully completing tasks based on user configurations.
- A cron job runs a job periodically on a specified schedule. A cron job object is similar to a line of a crontab file in Linux.

This run-to-completion feature of jobs is especially suitable for one-off tasks, such as continuous integration (CI).

Creating a Job

The following is an example job, which calculates π till the 2000th digit and prints the output. 50 pods need to be run before the job is ended. In this example, print π calculation results for 50 times, and run five pods concurrently. If a pod fails to be run, a maximum of five retries are supported.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  completions: 50            # Number of pods that need to run successfully to end the job
  parallelism: 5            # Number of pods that run concurrently. The default value is 1.
  backoffLimit: 5            # Maximum number of retries performed if a pod fails. When the limit is reached, 
it will not try again.
  activeDeadlineSeconds: 10  # Timeout interval of pods. Once the time is reached, all pods of the job are 
terminated.
  template:                  # Pod definition
    spec:
      containers:
      - name: pi
```

```
      image: perl
      command:
      - perl
      - "-Mbignum=bpi"
      - "-wle"
      - print bpi(2000)
    restartPolicy: Never
```

Based on the **completions** and **Parallelism** settings, jobs can be classified as follows:

Table 4-1 Job types

Job Type	Description	Example
One-off job	One pod runs until it is successfully ends.	Database migration
Jobs with a fixed completion count	One pod runs until the specified completion count is reached.	Pod for processing work queues
Parallel jobs with a fixed completion count	Multiple pods run until the specified completion count is reached.	Multiple pods for processing work queues concurrently
Parallel jobs	One or more pods run until one pod is successfully ended.	Multiple pods for processing work queues concurrently

Creating a Cron Job

Compared with a job, a cron job is a scheduled job. A cron job runs a job periodically on a specified schedule, and the job creates pods.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "0,15,30,45 * * * *"           # Scheduling configuration
  jobTemplate:                             # Job definition
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: main
            image: pi
```

The format of the cron job is as follows:

- Minute
- Hour
- Day of month
- Month

- Day of week

For example, in **0,15,30,45 * * * ***, commas separate minutes, the first asterisk (*) indicates the hour, the second asterisk indicates the day of the month, the third asterisk indicates the month, and the fourth asterisk indicates the day of the week.

If you want to run the job every half an hour on the first day of each month, set this parameter to **0,30 * 1 * ***. If you want to run the job on 3:00 a.m. every Sunday, set this parameter to **0 3 * * 0**.

For details about the cron job format, visit **https://en.wikipedia.org/wiki/Cron**.

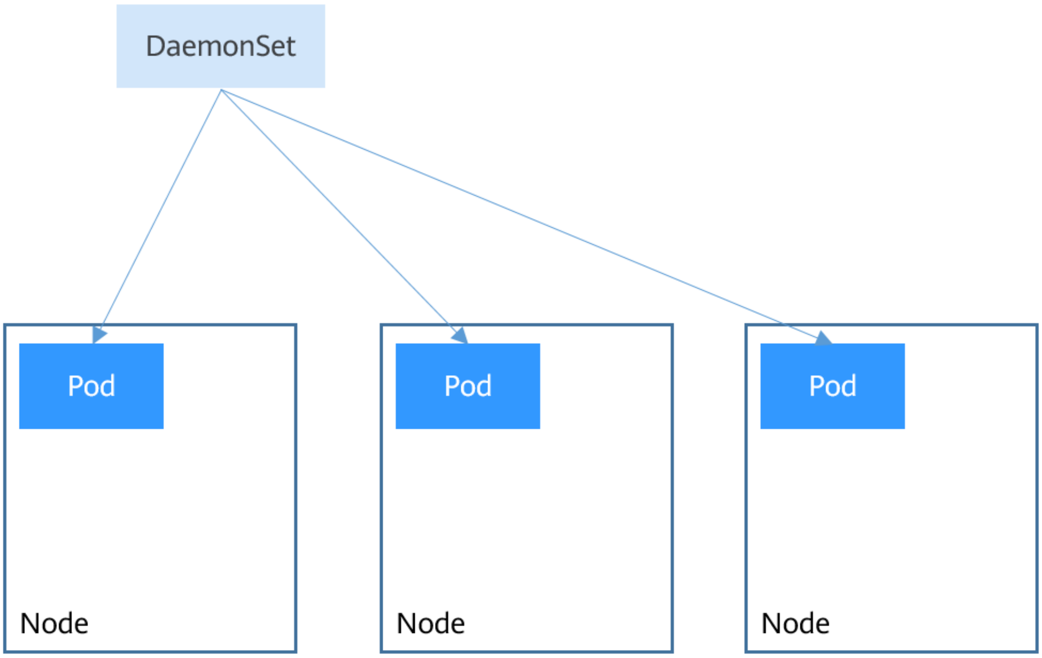
4.4 DaemonSet

DaemonSet

A DaemonSet runs a pod on each node in a cluster and ensures that there is only one pod. This works well for certain system-level applications, such as log collection and resource monitoring, since they must run on each node and need only a few pods. A good example is kube-proxy.

DaemonSets are closely related to nodes. If a node becomes faulty, the DaemonSet will not create the same pods on other nodes.

Figure 4-4 DaemonSet



Creating a DaemonSet

The following is an example of a DaemonSet:

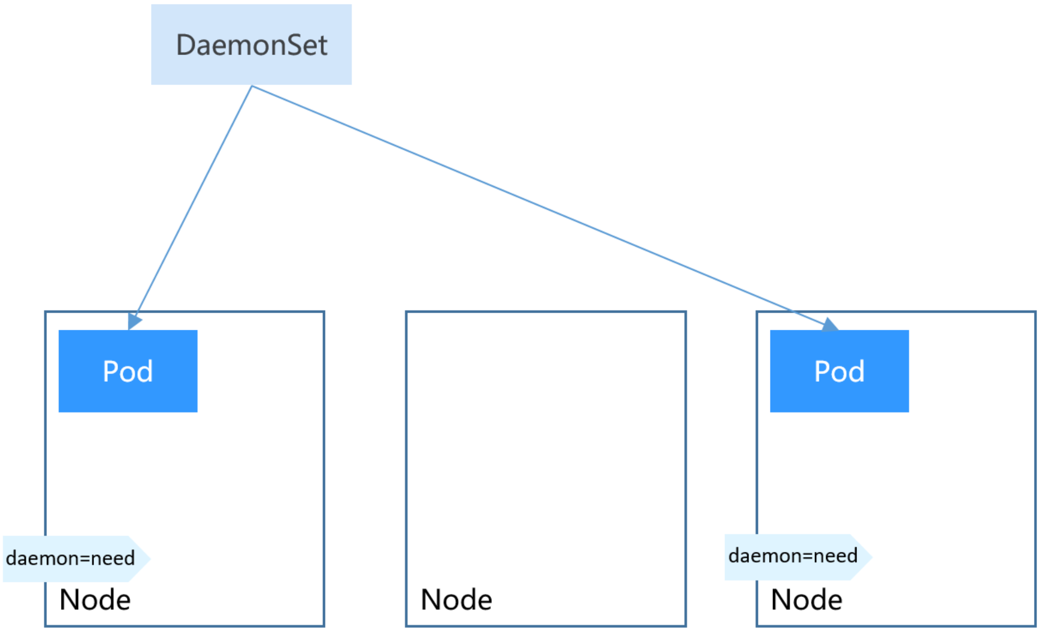
```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
```

```
  labels:
    app: nginx-daemonset
spec:
  selector:
    matchLabels:
      app: nginx-daemonset
  template:
    metadata:
      labels:
        app: nginx-daemonset
    spec:
      nodeSelector:                 #Node selection. A pod is created on a node only when the node meets
daemon=need.
        daemon: need
      containers:
      - name: nginx-daemonset
        image: nginx:alpine
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
      imagePullSecrets:
      - name: default-secret
```

The **replicas** parameter used in defining a Deployment or StatefulSet does not exist in the above configuration for a DaemonSet, because each node has only one replica. It is fixed.

The nodeSelector in the preceding pod template specifies that a pod is created only on the nodes that meet **daemon=need**, as shown in the following figure. If you want to create a pod on each node, delete the label.

Figure 4-5 DaemonSet creating a pod on nodes with a specified label



Create a DaemonSet.

```
$ kubectl create -f daemonset.yaml
daemonset.apps/nginx-daemonset created
```

Run the following command. The output shows that **nginx-daemonset** creates no pods on nodes.

```
$ kubectl get ds
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   0         0         0       0            0           daemon=need     16s

$ kubectl get pods
No resources found in default namespace.
```

This is because no nodes have the **daemon=need** label. Run the following command to query the labels of nodes:

```
$ kubectl get node --show-labels
NAME           STATUS   ROLES    AGE   VERSION                            LABELS
192.168.0.212   Ready    <none>   83m   v1.15.6-r1-20.3.0.2.B001-15.30.2   beta.kubernetes.io/arch=amd64 ...
192.168.0.94    Ready    <none>   83m   v1.15.6-r1-20.3.0.2.B001-15.30.2   beta.kubernetes.io/arch=amd64 ...
192.168.0.97    Ready    <none>   83m   v1.15.6-r1-20.3.0.2.B001-15.30.2   beta.kubernetes.io/arch=amd64 ...
```

Add the **daemon=need** label to node **192.168.0.212**, and then query the pods of **nginx-daemonset** again. It is found that a pod has been created on node **192.168.0.212**.

```
$ kubectl label node 192.168.0.212 daemon=need
node/192.168.0.212 labeled

$ kubectl get ds
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   1         1         0       1            0           daemon=need     116s

$ kubectl get pod -o wide
NAME                    READY   STATUS    RESTARTS   AGE   IP           NODE
nginx-daemonset-g9b7j   1/1     Running   0          18s   172.16.3.0   192.168.0.212
```

Add the **daemon=need** label to node **192.168.0.94**. You can find that a pod is created on this node as well.

```
$ kubectl label node 192.168.0.94 daemon=need
node/192.168.0.94 labeled

$ kubectl get ds
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   2         2         1       2            1           daemon=need     2m29s

$ kubectl get pod -o wide
NAME                    READY   STATUS              RESTARTS   AGE   IP           NODE
nginx-daemonset-6jjxz   0/1     ContainerCreating   0          8s    <none>       192.168.0.94
nginx-daemonset-g9b7j   1/1     Running             0          42s   172.16.3.0   192.168.0.212
```

Modify the **daemon=need** label of node **192.168.0.94**. You can find the DaemonSet deletes its pod from the node.

```
$ kubectl label node 192.168.0.94 daemon=no --overwrite
node/192.168.0.94 labeled

$ kubectl get ds
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   1         1         1       1            1           daemon=need     4m5s

$ kubectl get pod -o wide
NAME                    READY   STATUS    RESTARTS   AGE     IP           NODE
nginx-daemonset-g9b7j   1/1     Running   0          2m23s   172.16.3.0   192.168.0.212
```


4.5 Affinity and Anti-Affinity Scheduling

A nodeSelector provides a very simple way to constrain pods to nodes with particular labels, as mentioned in **DaemonSet**. The affinity and anti-affinity feature greatly expands the types of constraints you can express.

Kubernetes supports node-level and pod-level affinity and anti-affinity. You can configure custom rules to achieve affinity and anti-affinity scheduling. For example, you can deploy frontend pods and backend pods together, deploy the same type of applications on a specific node, or deploy different applications on different nodes.

Node Affinity

Node affinity is conceptually similar to a nodeSelector as it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node. The following output lists the labels of node **192.168.0.212**.

```
$ kubectl describe node 192.168.0.212
Name:               192.168.0.212
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    failure-domain.beta.kubernetes.io/is-baremetal=false
                    failure-domain.beta.kubernetes.io/region=cn-east-3
                    failure-domain.beta.kubernetes.io/zone=cn-east-3a
                    kubernetes.io/arch=amd64
                    kubernetes.io/availablezone=cn-east-3a
                    kubernetes.io/eniquota=12
                    kubernetes.io/hostname=192.168.0.212
                    kubernetes.io/os=linux
                    node.kubernetes.io/subnetid=fd43acad-33e7-48b2-a85a-24833f362e0e
                    os.architecture=amd64
                    os.name=EulerOS_2.0_SP5
                    os.version=3.10.0-862.14.1.5.h328.eulerosv2r7.x86_64
```

These labels are automatically added by CCE during node creation. The following describes a few that are frequently used during scheduling.

- **failure-domain.beta.kubernetes.io/region**: region where the node is located. In the preceding output, the label value is **cn-east-3**, which indicates that the node is located in the CN East-Shanghai1 region.
- **failure-domain.beta.kubernetes.io/zone**: availability zone to which the node belongs.
- **kubernetes.io/hostname**: host name of the node.

In addition to these automatically added labels, you can tailor labels to your service requirements, as introduced in **Label for Managing Pods**. Generally, large Kubernetes clusters have various kinds of labels.

When you deploy pods, you can use a nodeSelector, as described in **DaemonSet**, to constrain pods to nodes with specific labels. The following example shows how to use a nodeSelector to deploy pods only on the nodes with the **gpu=true** label.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
```

```
spec:
  nodeSelector:                #Node selection. A pod is deployed on a node only when the node has the
gpu=true label.
    gpu: true
...
```

Node affinity rules can achieve the same results, as shown in the following example.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  gpu
  labels:
    app:  gpu
spec:
  selector:
    matchLabels:
      app: gpu
  replicas: 3
  template:
    metadata:
      labels:
        app:  gpu
    spec:
      containers:
      - image:  nginx:alpine
        name:  gpu
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
          limits:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: gpu
                operator: In
                values:
                - "true"
```

Even though the node affinity rule requires more lines, it is more expressive, which will be further described later.

requiredDuringSchedulingIgnoredDuringExecution seems to be complex, but it can be easily understood as a combination of two parts.

- requiredDuringScheduling indicates that pods can be scheduled to the node only when all the defined rules are met (required).
- IgnoredDuringExecution indicates that pods already running on the node do not need to meet the defined rules. That is, a label on the node is ignored, and pods that require the node to contain that label will not be re-scheduled.

In addition, the value of **operator** is **In**, indicating that the label value must be in the values list. Other available operator values are as follows:

- **NotIn**: The label value is not in a list.
- **Exists**: A specific label exists.
- **DoesNotExist**: A specific label does not exist.

- **Gt**: The label value is greater than a specified value (string comparison).
- **Lt**: The label value is less than a specified value (string comparison).

Note that there is no such thing as nodeAntiAffinity because operators **NotIn** and **DoesNotExist** provide the same function.

Now, check whether the node affinity rule takes effect. Add the **gpu=true** tag to the **192.168.0.212** node.

```
$ kubectl label node 192.168.0.212 gpu=true
node/192.168.0.212 labeled

$ kubectl get node -L gpu
NAME            STATUS   ROLES    AGE   VERSION                            GPU
192.168.0.212   Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
192.168.0.94    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.97    Ready    <none>   13m   v1.15.6-r1-20.3.0.2.B001-15.30.2
```

Create the Deployment. You can find that all pods are deployed on the **192.168.0.212** node.

```
$ kubectl create -f affinity.yaml
deployment.apps/gpu created

$ kubectl get pod -o wide
NAME                   READY   STATUS    RESTARTS   AGE   IP            NODE
gpu-6df65c44cf-42xw4   1/1     Running   0          15s   172.16.0.37   192.168.0.212
gpu-6df65c44cf-jzjvs   1/1     Running   0          15s   172.16.0.36   192.168.0.212
gpu-6df65c44cf-zv5cl   1/1     Running   0          15s   172.16.0.38   192.168.0.212
```

Node Preference Rule

The preceding **requiredDuringSchedulingIgnoredDuringExecution** rule is a hard selection rule. There is another type of selection rule, that is, **preferredDuringSchedulingIgnoredDuringExecution**. It is used to specify which nodes are preferred during scheduling.

To demonstrate its effect, add a node to the cluster and ensure that the node is not in the same AZ with other nodes. After the node is created, query the AZ of the node. As shown in the following output, the newly added node is in cn-east-3c.

```
$ kubectl get node -L failure-domain.beta.kubernetes.io/zone,gpu
NAME            STATUS   ROLES    AGE     VERSION                            ZONE         GPU
192.168.0.100   Ready    <none>   7h23m   v1.15.6-r1-20.3.0.2.B001-15.30.2   cn-east-3c
192.168.0.212   Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   cn-east-3a   true
192.168.0.94    Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   cn-east-3a
192.168.0.97    Ready    <none>   8h      v1.15.6-r1-20.3.0.2.B001-15.30.2   cn-east-3a
```

Define a Deployment. Use the **preferredDuringSchedulingIgnoredDuringExecution** rule to set the weight of nodes in **cn-east-3a** as **80** and nodes with the **gpu=true** label as **20**. In this way, pods are preferentially deployed on the node in cn-east-3a.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  gpu
  labels:
    app:  gpu
spec:
  selector:
    matchLabels:
      app: gpu
```

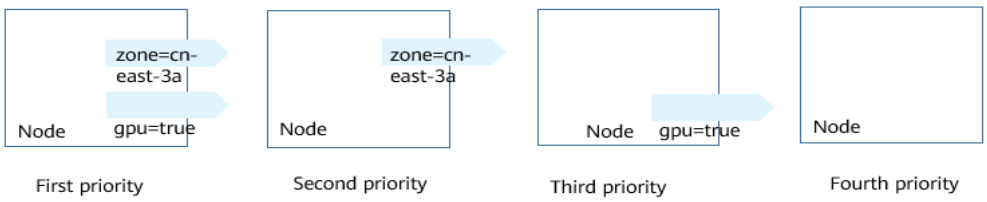
```
replicas: 10
template:
  metadata:
    labels:
      app:  gpu
  spec:
    containers:
    - image:  nginx:alpine
      name:  gpu
      resources:
        requests:
          cpu:  100m
          memory:  200Mi
        limits:
          cpu:  100m
          memory:  200Mi
    imagePullSecrets:
    - name: default-secret
    affinity:
      nodeAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 80
          preference:
            matchExpressions:
            - key: failure-domain.beta.kubernetes.io/zone
              operator: In
              values:
              - cn-east-3a
        - weight: 20
          preference:
            matchExpressions:
            - key: gpu
              operator: In
              values:
              - "true"
```

After the deployment, you can find that five pods are deployed on the **192.168.0.212** node, and two pods are deployed on the **192.168.0.100** node.

```
$ kubectl create -f affinity2.yaml
deployment.apps/gpu created

$ kubectl get po -o wide
NAME                  READY   STATUS    RESTARTS   AGE     IP            NODE
gpu-585455d466-5bmcz   1/1     Running   0          2m29s   172.16.0.44   192.168.0.212
gpu-585455d466-cg2l6   1/1     Running   0          2m29s   172.16.0.63   192.168.0.97
gpu-585455d466-f2bt2   1/1     Running   0          2m29s   172.16.0.79   192.168.0.100
gpu-585455d466-hdb5n   1/1     Running   0          2m29s   172.16.0.42   192.168.0.212
gpu-585455d466-hkgvz   1/1     Running   0          2m29s   172.16.0.43   192.168.0.212
gpu-585455d466-mngvn   1/1     Running   0          2m29s   172.16.0.48   192.168.0.97
gpu-585455d466-s26qs   1/1     Running   0          2m29s   172.16.0.62   192.168.0.97
gpu-585455d466-sxtzm   1/1     Running   0          2m29s   172.16.0.45   192.168.0.212
gpu-585455d466-t56cm   1/1     Running   0          2m29s   172.16.0.64   192.168.0.100
gpu-585455d466-t5w5x   1/1     Running   0          2m29s   172.16.0.41   192.168.0.212
```

In the preceding example, the node scheduling priority is as follows. Nodes with both **cn-east-3a** and **gpu=true** labels have the highest priority. Nodes with the **cn-east-3a** label but no **gpu=true** label have the second priority (weight: 80). Nodes with the **gpu=true** label but no **cn-east-3a** label have the third priority. Nodes without any of these two labels have the lowest priority.

Figure 4-6 Scheduling priority



From the preceding output, you can find that no pods of the Deployment are scheduled to node **192.168.0.94**. This is because the node already has many pods on it and its resource usage is high. This also indicates that the **preferredDuringSchedulingIgnoredDuringExecution** rule defines a preference rather than a hard requirement.

Pod Affinity

Node affinity rules affect only the affinity between pods and nodes. Kubernetes also supports configuring inter-pod affinity rules. For example, the frontend and backend of an application can be deployed together on one node to reduce access latency. There are also two types of inter-pod affinity rules: **requiredDuringSchedulingIgnoredDuringExecution** and **preferredDuringSchedulingIgnoredDuringExecution**.

Assume that the backend of an application has been created and has the **app=backend** label.

```
$ kubectl get po -o wide
NAME                       READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-dlrz8   1/1     Running   0          2m36s   172.16.0.67   192.168.0.100
```

You can configure the following pod affinity rule to deploy the frontend pods of the application to the same node as the backend pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
```

```
      - name: default-secret
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - backend
```

Deploy the frontend and you can find that the frontend is deployed on the same node as the backend.

```
$ kubectl create -f affinity3.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-dlrz8    1/1     Running   0          5m38s   172.16.0.67   192.168.0.100
frontend-67ff9b7b97-dsqzn   1/1     Running   0          6s      172.16.0.70   192.168.0.100
frontend-67ff9b7b97-hxm5t   1/1     Running   0          6s      172.16.0.71   192.168.0.100
frontend-67ff9b7b97-z8pdb   1/1     Running   0          6s      172.16.0.72   192.168.0.100
```

The **topologyKey** field specifies the selection range. The scheduler selects nodes within the range based on the affinity rule defined. The effect of **topologyKey** is not fully demonstrated in the preceding example because all the nodes have the **kubernetes.io/hostname** label, that is, all the nodes are within the range.

To see how **topologyKey** works, assume that the backend of the application has two pods, which are running on different nodes.

```
$ kubectl get po -o wide
NAME                       READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-5bpd6   1/1     Running   0          23m     172.16.0.40   192.168.0.97
backend-658f6cb858-dlrz8   1/1     Running   0          2m36s   172.16.0.67   192.168.0.100
```

Add the **prefer=true** label to nodes **192.168.0.97** and **192.168.0.94**.

```
$ kubectl label node 192.168.0.97 prefer=true
node/192.168.0.97 labeled
$ kubectl label node 192.168.0.94 prefer=true
node/192.168.0.94 labeled

$ kubectl get node -L prefer
NAME            STATUS   ROLES    AGE   VERSION                            PREFER
192.168.0.100   Ready    <none>   44m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.212   Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.94    Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
192.168.0.97    Ready    <none>   91m   v1.15.6-r1-20.3.0.2.B001-15.30.2   true
```

Define **topologyKey** in the **podAffinity** section as **prefer**.

```
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: prefer
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - backend
```

The scheduler recognizes the nodes with the **prefer** label, that is, **192.168.0.97** and **192.168.0.94**, and then find the pods with the **app=backend** label. In this way, all frontend pods are deployed onto **192.168.0.97**.


```
$ kubectl create -f affinity3.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                       READY   STATUS    RESTARTS   AGE     IP            NODE
backend-658f6cb858-5bpd6    1/1     Running   0          26m     172.16.0.40   192.168.0.97
backend-658f6cb858-dlrz8    1/1     Running   0          5m38s   172.16.0.67   192.168.0.100
frontend-67ff9b7b97-dsqzn   1/1     Running   0          6s      172.16.0.70   192.168.0.97
frontend-67ff9b7b97-hxm5t   1/1     Running   0          6s      172.16.0.71   192.168.0.97
frontend-67ff9b7b97-z8pdb   1/1     Running   0          6s      172.16.0.72   192.168.0.97
```

Pod Anti-affinity

Unlike the scenarios in which pods are preferred to be scheduled onto the same node, sometimes, it could be the exact opposite. For example, if certain pods are deployed together, they will affect the performance.

The following example defines an inter-pod anti-affinity rule, which specifies that pods must not be scheduled to nodes that already have pods with the **app=frontend** label, that is, to deploy the pods of the frontend to different nodes with each node has only one replica.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 5
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - frontend
```

Deploy the frontend and query the deployment results. You can find that each node has only one frontend pod and one pod of the Deployment is **Pending**. This is because when the scheduler is deploying the fifth pod, all nodes already have one pod with the **app=frontend** label on them. There is no available node. Therefore, the fifth pod will remain in the **Pending** status.

```
$ kubectl create -f affinity4.yaml
deployment.apps/frontend created

$ kubectl get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP            NODE
frontend-6f686d8d87-8dlsc   1/1     Running   0          18s   172.16.0.76   192.168.0.100
frontend-6f686d8d87-d6l8p   0/1     Pending   0          18s   <none>        <none>
frontend-6f686d8d87-hgcq2   1/1     Running   0          18s   172.16.0.54   192.168.0.97
frontend-6f686d8d87-q7cfq   1/1     Running   0          18s   172.16.0.47   192.168.0.212
frontend-6f686d8d87-xl8hx   1/1     Running   0          18s   172.16.0.23   192.168.0.94
```

5 Configuration Management

5.1 ConfigMap

A ConfigMap is a type of resource used to store the configurations required by applications. It is used to store configuration data or configuration files in key-value pairs.

A ConfigMap allows you to decouple configurations from your environments, so that your environments can use different configurations.

Creating a ConfigMap

In the following example, a ConfigMap named **configmap-test** is created. The ConfigMap configuration data is defined in the **data** field.

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

Referencing a ConfigMap as an Environment Variable

ConfigMaps are usually referenced as environment variables and in volumes.

In the following example, **property_1** of **configmap-test** is used as the value of the environment variable **EXAMPLE_PROPERTY_1**. After the container is started, it will reference the value of **property_1** as the value of **EXAMPLE_PROPERTY_1**, that is, **Hello**.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
```

```
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:
    - name: EXAMPLE_PROPERTY_1
      valueFrom:
        configMapKeyRef:          # Reference the ConfigMap.
          name: configmap-test
          key: property_1
  imagePullSecrets:
  - name: default-secret
```

Referencing a ConfigMap in a Volume

Referencing a ConfigMap in a volume is to fill its data in configuration files in the volume. Each piece of data is saved in a file. The key is the file name, and the key value is the file content.

In the following example, create a volume named **vol-configmap**, reference the ConfigMap named **configmap-test** in the volume, and mount the volume to the **/tmp** directory of the container. After the pod is created, the two files **property_1** and **property_2** are generated in the **/tmp** directory of the container, and the values are **Hello** and **World**.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    volumeMounts:
    - name: vol-configmap          # Mount the volume named vol-configmap.
      mountPath: "/tmp"
  imagePullSecrets:
  - name: default-secret
  volumes:
  - name: vol-configmap
    configMap:                     # Reference the ConfigMap.
      name: configmap-test
```

5.2 Secret

A secret is a resource object that is encrypted for storing the authentication information, certificates, and private keys. The sensitive data will not be exposed in images or pod definitions, which is safer and more flexible.

Similar to a ConfigMap, a secret stores data in key-value pairs. The difference is that a secret is encrypted, and is suitable for storing sensitive information.

Base64 Encoding

A secret stores data in key-value pairs, the same form as that of a ConfigMap. The difference is that the value must be encoded using Base64 when a secret is created.

To encode a character string using Base64, run the **echo -n *to-be-encoded content* | base64** command. The following is an example:

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

Creating a Secret

The secret defined in the following example contains two key-value pairs.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  key1: aGVsbG8gd29ybGQ=   # hello world, a value encoded using Base64
  key2: MzMwNg==           # 3306, a value encoded using Base64
```

Referencing a Secret as an Environment Variable

Secrets are usually injected into containers as environment variables, as shown in the following example.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:
    - name: key
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: key1
  imagePullSecrets:
  - name: default-secret
```

Referencing a Secret in a Volume

Referencing a secret in a volume is to fill its data in configuration files in the volume. Each piece of data is saved in a file. The key is the file name, and the key value is the file content.

In the following example, create a volume named **vol-secret**, reference the secret named **mysecret** in the volume, and mount the volume to the **/tmp** directory of the container. After the pod is created, the two files **key1** and **key2** are generated in the **/tmp** directory of the container.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    volumeMounts:
    - name: vol-secret              # Mount the volume named vol-secret.
      mountPath: "/tmp"
  imagePullSecrets:
  - name: default-secret
  volumes:
  - name: vol-secret
    secret:                         # Reference the secret.
      secretName: mysecret
```

In the pod container, you can find the two files **key1** and **key2** in the **/tmp** directory. The values in the files are the values encoded using Base64, which are **hello world** and **3306**.

6 Kubernetes Networking

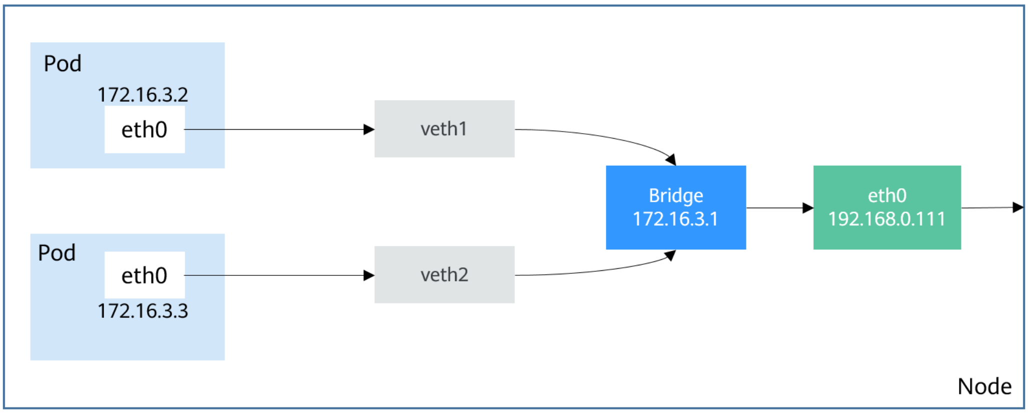
6.1 Container Networking

Networking among pods, clusters, and nodes is not implemented by Kubernetes itself, but by the Container Network Interface (CNI) plug-ins. There are many open source CNI plug-ins, such as Flannel and Calico. HUAWEI CLOUD CCE also provides customized CNI plug-ins (Canal and Yangtse) for you to use HUAWEI CLOUD VPC networks when running Kubernetes.

Kubernetes requires that pods in a cluster can communicate with each other and the pods must be connected through a non-NAT network. That is, the source IP address of the received data packet is that of the pod that sends the data packet. Pods are also required to communicate with nodes through a non-NAT network. However, when the pod accesses an object outside the cluster, the source IP address is changed to the node IP address.

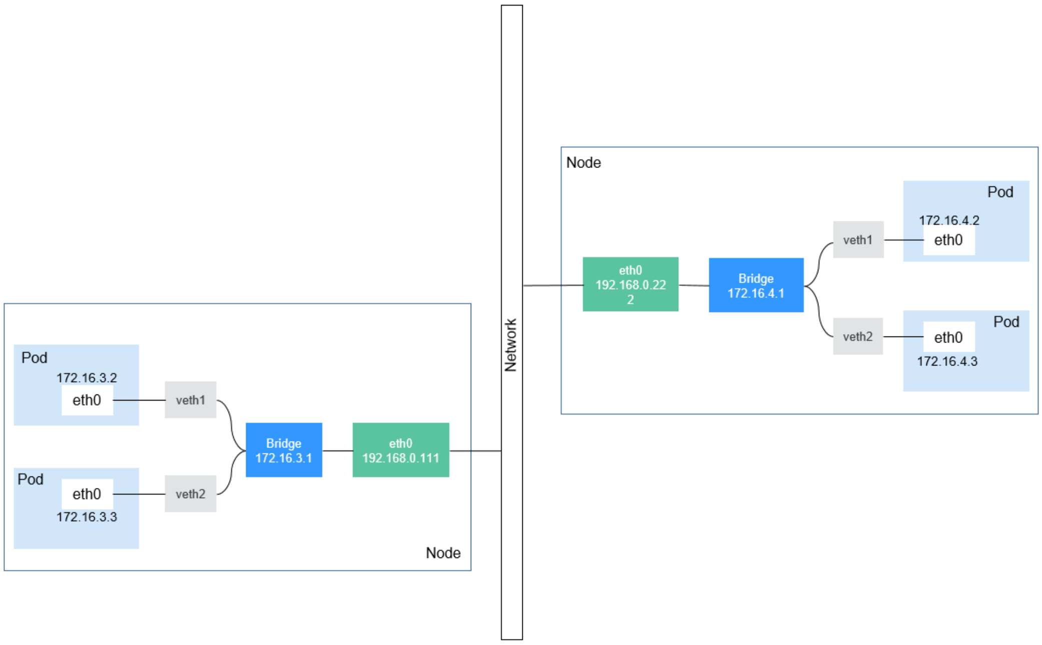
A pod is connected to external systems through a virtual Ethernet interface pair (veth pair). For pods on the same node, they communicate with each other through a Linux bridge, as shown in the following figure.

Figure 6-1 Communication for pods on the same node



Bridges between different nodes can be implemented in multiple modes. However, in a cluster, the pod IP address must be unique. Therefore, cross-node bridges will use different CIDR blocks to prevent duplicate pod IP addresses.

Figure 6-2 Communication for pods on different nodes



The following sections **Service** and **Ingress** will describe how Kubernetes provides access solutions for users based on the container networking.

6.2 Service

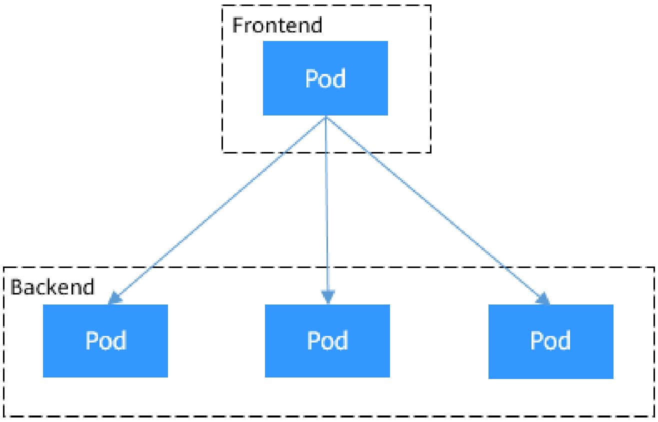
Direct Access to a Pod

After a pod is created, the following problems may occur if you directly access the pod:

- The pod can be deleted and recreated at any time by a controller such as a Deployment, and the result of accessing the pod becomes unpredictable.
- The IP address of the pod is allocated only after the pod is started. Before the pod is started, the IP address of the pod is unknown.
- An application is usually composed of multiple pods that run the same image. Accessing pods one by one is not efficient.

For example, an application uses Deployments to create the frontend and backend. The frontend calls the backend for computing, as shown in **Figure 6-3**. Three pods are running in the backend, which are independent and replaceable. When a backend pod is re-created, the new pod is assigned with a new IP address, of which the frontend pod is unaware.

Figure 6-3 Inter-pod access

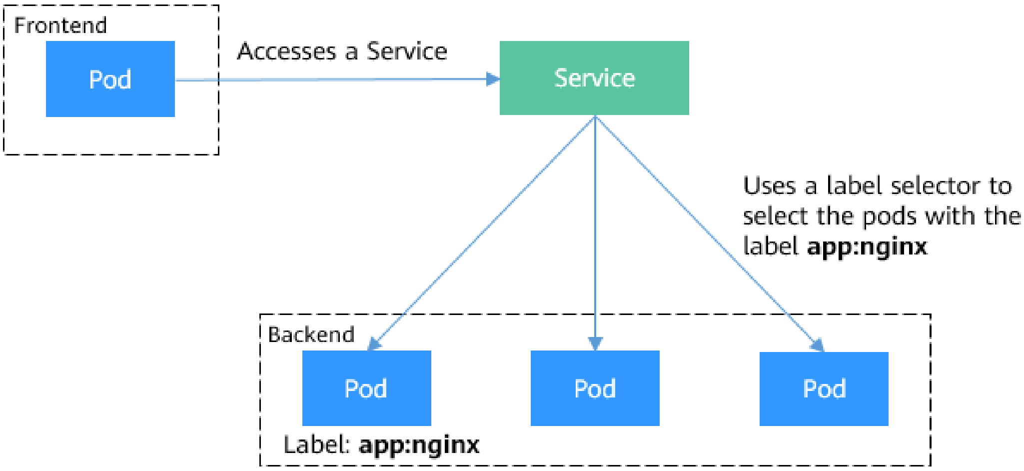


Using Services for Pod Access

Kubernetes Services are used to solve the preceding pod access problems. A Service has a fixed IP address. (When a CCE cluster is created, a Service CIDR block is set, which is used to allocate IP addresses to Services.) A Service forwards requests accessing the Service to pods based on labels, and at the same time, perform load balancing for these pods.

In the preceding example, a Service is added for the frontend pod to access the backend pods. In this way, the frontend pod does not need to be aware of the changes on backend pods, as shown in **Figure 6-4**.

Figure 6-4 Accessing pods through a Service



Creating Backend Pods

Create a Deployment with three replicas, that is, three pods with label **app: nginx**.

```
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: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

Creating a Service

In the following example, we create a Service named **nginx**, and use a selector to select the pod with the label **app:nginx**. The port of the target pod is port 80 while the exposed port of the Service is port 8080.

The Service can be accessed using ***Service name:Exposed port***. In the example, **nginx:8080** is used. In this case, other pods can access the pod associated with **nginx** using **nginx:8080**.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx        #Service name
spec:
  selector:          #Label selector, which selects pods with the label of app=nginx
    app: nginx
  ports:
  - name: service0
    targetPort: 80   #Pod port
    port: 8080       #Service external port
    protocol: TCP    #Forwarding protocol type. The value can be TCP or UDP.
  type: ClusterIP    #Service type
```

Save the Service definition to **nginx-svc.yaml** and use kubectl to create the Service.

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

$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.247.0.1       <none>        443/TCP    7h19m
nginx        ClusterIP   10.247.124.252   <none>        8080/TCP   5h48m
```

You can see that the Service has a ClusterIP, which is fixed unless the Service is deleted. You can use this ClusterIP to access the Service inside the cluster.

Create a pod and use the ClusterIP to access the pod. Information similar to the following is returned.

```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 10.247.124.252:8080
<!DOCTYPE html>
```



```
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Using ServiceName to Access a Service

After the DNS resolves the domain name, you can use ***ServiceName:Port*** to access the Service, the most common practice in Kubernetes. When you are creating a CCE cluster, you are required to install the coredns add-on by default. You can view the pods of CoreDNS in the kube-system namespace.

```
$ kubectl get po --namespace=kube-system
NAME                                READY   STATUS    RESTARTS   AGE
coredns-7689f8bdf-295rk              1/1     Running   0          9m11s
coredns-7689f8bdf-h7n68              1/1     Running   0          11m
```

After coredns is installed, it becomes a DNS. After the Service is created, coredns records the Service name and IP address. In this way, the pod can obtain the Service IP address by querying the Service name from coredns.

nginx.<namespace>.svc.cluster.local is used to access the Service. **nginx** is the Service name, **<namespace>** is the namespace, and **svc.cluster.local** is the domain name suffix. In actual use, you can omit **<namespace>.svc.cluster.local** in the same namespace and use the ServiceName.

For example, if the Service named **nginx** is created, you can access the Service through **nginx:8080** and then access backend pods.

An advantage of using ServiceName is that you can write ServiceName into the program when developing the application. In this way, you do not need to know the IP address of a specific Service.

Now, create a pod and access the pod. Query the IP address of the nginx Service domain name, which is 10.247.124.252. Access the domain name of the pod and information similar to the following is returned.

```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx
Server:         10.247.3.10
Address:    10.247.3.10#53

Name:    nginx.default.svc.cluster.local
Address: 10.247.124.252

/ # curl nginx:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Using Services for Service Discovery

After a Service is deployed, it can discover the pod no matter how the pod changes.

If you run the **kubectl describe** command to query the Service, information similar to the following is displayed:

```
$ kubectl describe svc nginx
Name:              nginx
```

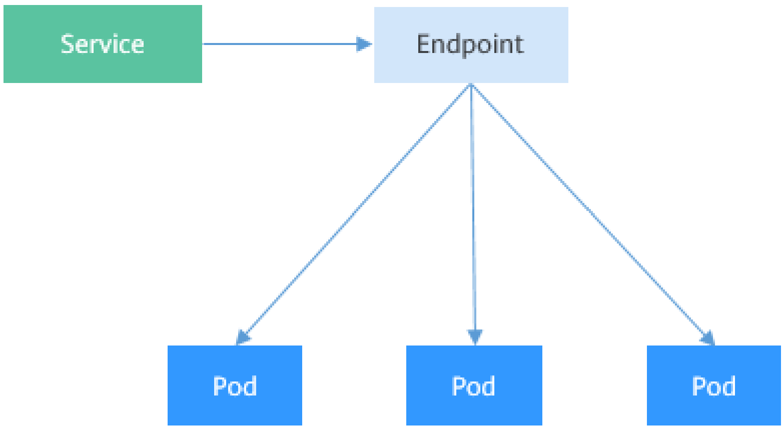
```
......
Endpoints:        172.16.2.132:80,172.16.3.6:80,172.16.3.7:80
......
```

One Endpoints record is displayed. An endpoint is also a resource object in Kubernetes. Kubernetes monitors the pod IP addresses through endpoints so that a Service can discover pods.

```
$ kubectl get endpoints
NAME       ENDPOINTS                                     AGE
nginx      172.16.2.132:80,172.16.3.6:80,172.16.3.7:80   5h48m
```

In this example, **172.16.2.132:80** is the **IP:port** of the pod. You can run the following command to view the IP address of the pod, which is the same as the preceding IP address.

```
$ kubectl get po -o wide
NAME                     READY   STATUS    RESTARTS   AGE     IP             NODE
nginx-869759589d-dnknn   1/1     Running   0          5h40m   172.16.3.7     192.168.0.212
nginx-869759589d-fcxhh   1/1     Running   0          5h40m   172.16.3.6     192.168.0.212
nginx-869759589d-r69kh   1/1     Running   0          5h40m   172.16.2.132   192.168.0.94
```



If a pod is deleted, the Deployment re-creates the pod and the IP address of the new pod changes.

```
$ kubectl delete po nginx-869759589d-dnknn
pod "nginx-869759589d-dnknn" deleted

$ kubectl get po -o wide
NAME                     READY   STATUS    RESTARTS   AGE     IP             NODE
nginx-869759589d-fcxhh   1/1     Running   0          5h41m   172.16.3.6     192.168.0.212
nginx-869759589d-r69kh   1/1     Running   0          5h41m   172.16.2.132   192.168.0.94
nginx-869759589d-w98wg   1/1     Running   0          7s      172.16.3.10    192.168.0.212
```

Check the endpoints again. You can see that the content under **ENDPOINTS** changes with the pod.

```
$ kubectl get endpoints
NAME         ENDPOINTS                                      AGE
kubernetes   192.168.0.127:5444                             7h20m
nginx        172.16.2.132:80,172.16.3.10:80,172.16.3.6:80   5h49m
```

Let's take a closer look at how this happens.

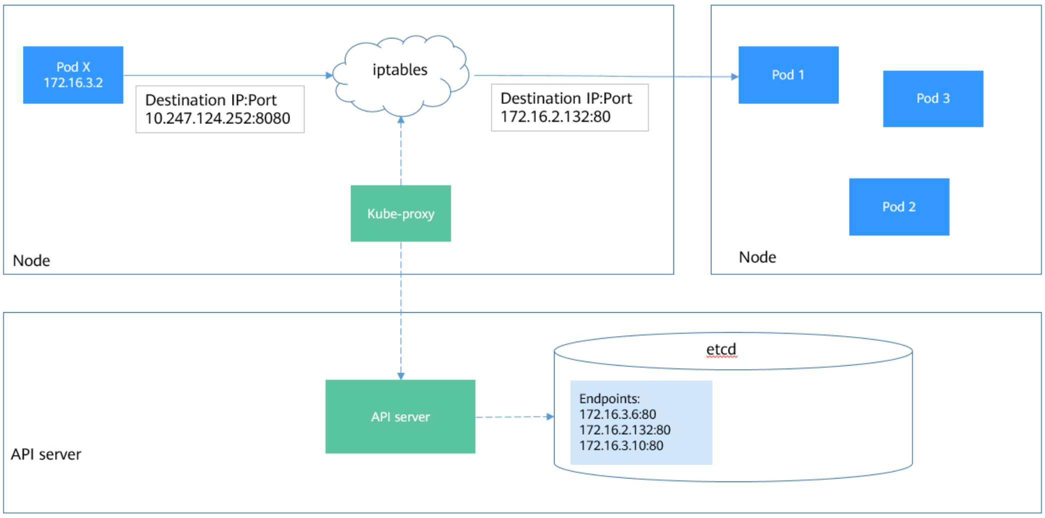
We have introduced kube-proxy on worker nodes in **Kubernetes Cluster Architecture**. Actually, all Service-related operations are performed by kube-proxy. When a Service is created, Kubernetes allocates an IP address to the Service and notifies kube-proxy on all nodes of the Service creation through the API server.

After receiving the notification, each kube-proxy records the relationship between the Service and the IP address/port pair through iptables. In this way, the Service can be queried on each node.

The following figure shows how a Service is accessed. Pod X accesses the Service (10.247.124.252:8080). When pod X sends data packets, the destination IP:Port is replaced with the IP:Port of pod 1 based on the iptables rule. In this way, the real backend pod can be accessed through the Service.

In addition to recording the relationship between Services and IP address/port pairs, kube-proxy also monitors the changes of Services and endpoints to ensure that pods can be accessed through Services after pods are rebuilt.

Figure 6-5 Service access process



Service Types and Application Scenarios

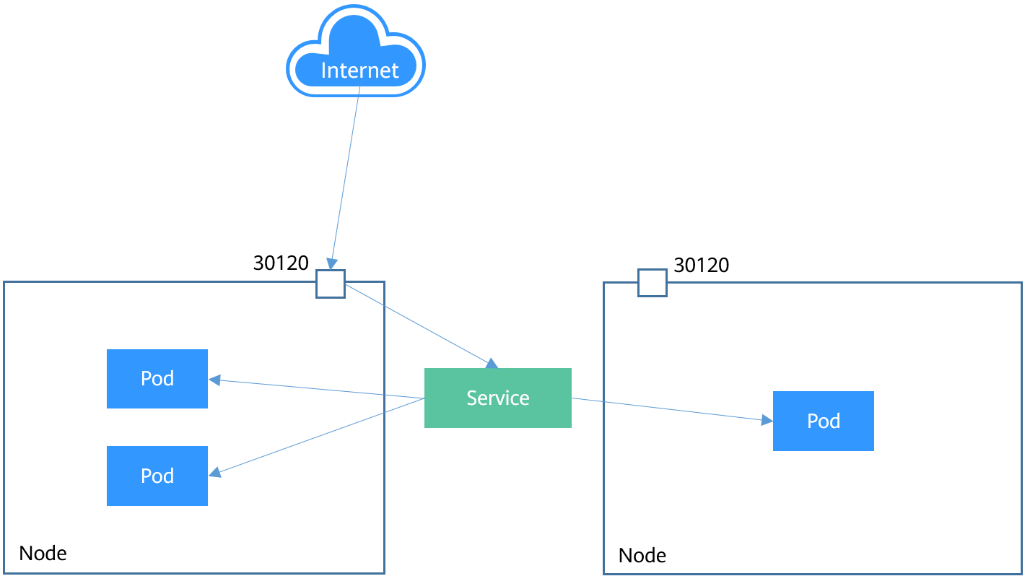
Services of the ClusterIP, NodePort, LoadBalancer, and None types have different functions.

- ClusterIP: used to make the Service only reachable from within a cluster.
- NodePort: used for access from outside a cluster. A NodePort Service is accessed through the port on the node. For details, see **NodePort Services**.
- LoadBalancer: used for access from outside a cluster. It is an extension of NodePort, to which a load balancer routes, and external systems only need to access the load balancer. For details, see **LoadBalancer Services**.
- None: used for mutual discovery between pods. This type of Service is also called headless Service. For details, see **Headless Service**.

NodePort Services

A NodePort Service enables each node in a Kubernetes cluster to reserve the same port. External systems first access the node IP:Port and then the NodePort Service forwards the requests to the pod corresponding to the Service.

Figure 6-6 NodePort Service



The following is an example of creating a NodePort Service. After the Service is created, you can access backend pods through IP:Port of the node.

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    nodePort: 30120
  selector:
    app: nginx
```

Create and view the Service. The value of **PORT** for the NodePort Service is **8080:30120/TCP**, indicating that port 8080 of the Service is mapped to port 30120 of the node.

```
$ kubectl create -f nodeport.yaml
service/nodeport-service created

$ kubectl get svc -o wide
NAME               TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE    SELECTOR
kubernetes         ClusterIP   10.247.0.1       <none>        443/TCP          107m   <none>
nginx              ClusterIP   10.247.124.252   <none>        8080/TCP         16m    app=nginx
nodeport-service   NodePort    10.247.210.174   <none>        8080:30120/TCP   17s    app=nginx
```

Access the Service by using Node IP:Port number to access the pod.

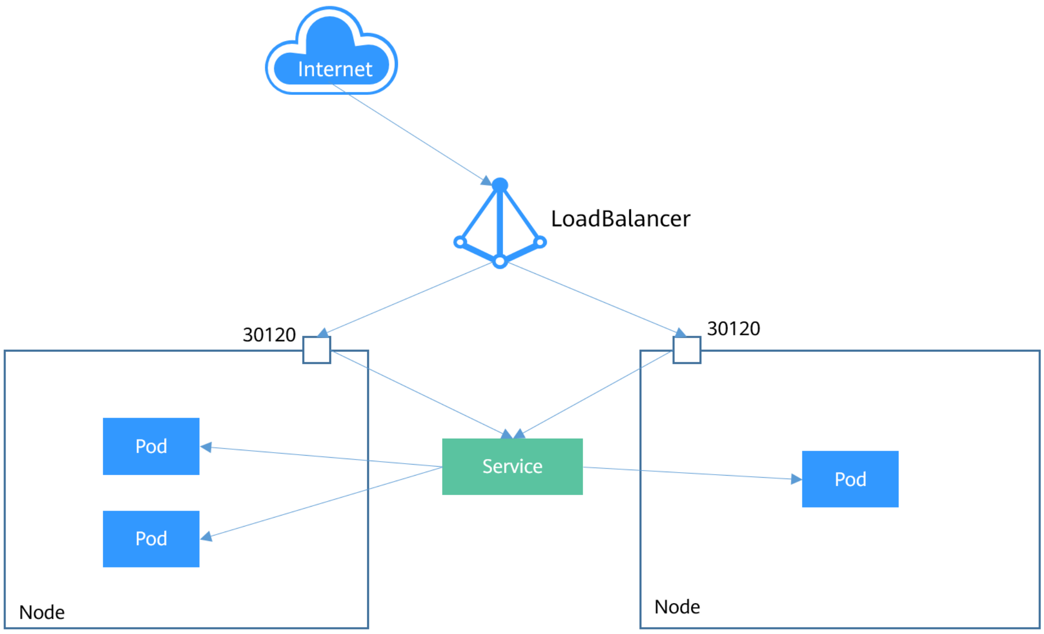
```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 192.168.0.212:30120
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
......
```


LoadBalancer Services

A Service is exposed externally using a load balancer that forwards requests to the NodePort of the node.

Load balancers are not a Kubernetes component. Different cloud service providers have different load balancers. For example, CCE interconnects with HUAWEI CLOUD Elastic Load Balance (ELB). As a result, there are different implementation methods of creating a LoadBalancer Service.

Figure 6-7 LoadBalancer Service



The following is an example of creating a LoadBalancer Service. After the LoadBalancer Service is created, you can access backend pods through IP:Port of the load balancer.

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: 3c7caa5a-a641-4bff-801a-feace27424b6
  labels:
    app: nginx
  name: nginx
spec:
  loadBalancerIP: 10.78.42.242     # IP address of the ELB instance
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30120
  selector:
    app: nginx
  type: LoadBalancer    # Service type (LoadBalancer)
```

The parameters in **annotations** under **metadata** are required for CCE LoadBalancer Services. They specify the ELB instance to which the Service is bound. CCE also allows you to create an ELB instance when creating a LoadBalancer Service. For details, see .

Headless Service

The preceding types of Services allow internal and external pod access, but not the following scenarios:

- Accessing all pods at the same time
- Pods in a Service accessing each other

This is where headless Service come into service. A headless Service does not create a cluster IP address, and the DNS records of all pods are returned during query. In this way, the IP addresses of all pods can be queried. StatefulSets in **StatefulSet** use headless Services to support mutual access between pods.

```
apiVersion: v1
kind: Service       # Object type (Service)
metadata:
  name: nginx-headless
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Name of the port for communication between pods
      port: 80        # Port number for communication between pods
  selector:
    app: nginx        # Select the pod whose label is app:nginx.
  clusterIP: None     # Set this parameter to None, indicating the headless Service.
```

Run the following command to create a headless Service:

```
# kubectl create -f headless.yaml
service/nginx-headless created
```

After the Service is created, you can query the Service.

```
# kubectl get svc
NAME             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx-headless   ClusterIP   None         <none>        80/TCP    5s
```

Create a pod to query the DNS. You can view the records of all pods. In this way, all pods can be accessed.

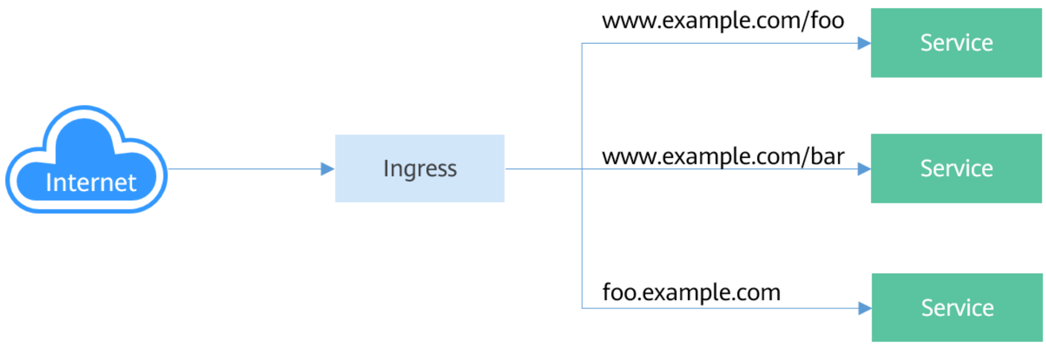
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # nslookup nginx-1.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-1.nginx.default.svc.cluster.local
Address: 172.16.0.18

/ # nslookup nginx-2.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-2.nginx.default.svc.cluster.local
Address: 172.16.0.19
```


6.3 Ingress

Why We Need Ingresses

Services forward requests using layer-4 TCP and UDP protocols. Ingresses forward requests using layer-7 HTTP and HTTPS protocols. Domain names and paths can be used to achieve finer granularities.

Figure 6-8 Ingress and Service

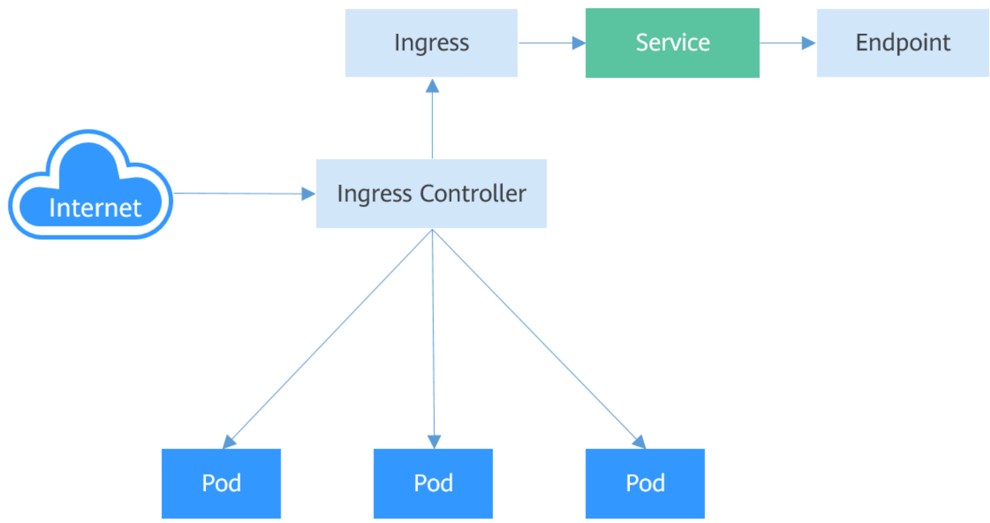


Ingress Working Mechanism

To use ingresses, you must install Ingress Controller on your Kubernetes cluster. Ingress Controller can be implemented in multiple modes. The most common one is **NGINX Ingress Controller** maintained by Kubernetes. In HUAWEI CLOUD, Cloud Container Engine (CCE) works with Elastic Load Balance (ELB) to implement layer-7 load balancing (via ingresses).

An external request is first sent to Ingress Controller. Then, Ingress Controller locates the corresponding Service based on the routing rule of an ingress, queries the IP address of the pod through the Endpoint, and forwards the request to the pod.

Figure 6-9 Ingress working mechanism



Creating an Ingress

In the following example, an ingress that uses the HTTP protocol, associates with backend Service **nginx:8080**, and uses a load balancer (specified by **metadata.annotations**) is created. After the request for accessing **http://192.168.10.155:8080/test** is initiated, the traffic is forwarded to Service **nginx:8080**, which in turn forwards the traffic to the corresponding pod.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: cce
    kubernetes.io/elb.port: '8080'
    kubernetes.io/elb.ip: 192.168.10.155
    kubernetes.io/elb.id: aa7cf5ec-7218-4c43-98d4-c36c0744667a
spec:
  rules:
  - host: ''
    http:
      paths:
      - backend:
          serviceName: nginx
          servicePort: 8080
        path: "/test"
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

You can also set the external domain name in an ingress so that you can access the load balancer through the domain name and then access backend Services.

NOTE

Domain name-based access depends on domain name resolution. You need to point the domain name to the IP address of the load balancer. For example, you can use to resolve domain names.

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

Accessing Multiple Services

An ingress can access multiple Services at the same time. The configuration is as follows:

- When you access **http://foo.bar.com/foo**, the backend Service **s1:80** is accessed.
- When you access **http://foo.bar.com/bar**, the backend Service **s2:80** is accessed.

```
spec:
  rules:
  - host: foo.bar.com          # Host address
    http:
      paths:
      - path: "/foo"
        backend:
          serviceName: s1
```

```
      servicePort: 80
  - path: "/bar"
    backend:
      serviceName: s2
      servicePort: 80
```

6.4 Readiness Probe

After a pod is created, the Service can immediately select it and forward requests to it. However, it takes time to start a pod. If the pod is not ready (it takes time to load the configuration or data, or a preheating program may need to be executed), the pod cannot process requests, and the requests will fail.

Kubernetes solves this problem by adding a readiness probe to pods. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

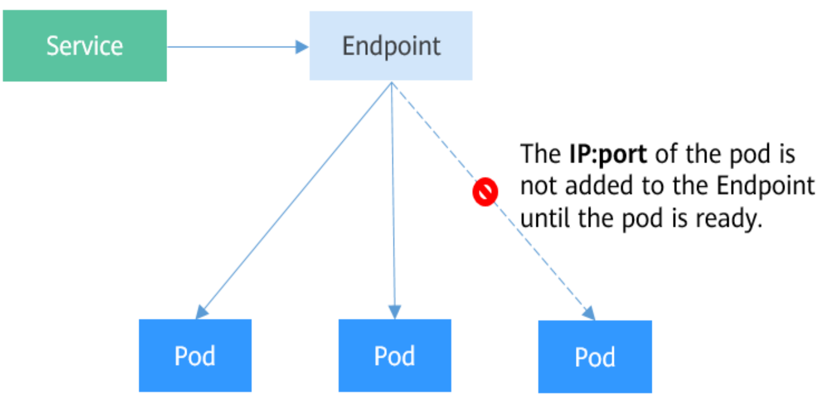
A readiness probe periodically detects a pod and determines whether the pod is ready based on its response. Similar to **Liveness Probe**, there are three types of readiness probes.

- Exec: kubelet executes a command in the target container. If the command succeeds, it returns **0**, and kubelet considers the container to be ready.
- HTTP GET: The probe sends an HTTP GET request to **IP:port** of the container. If the probe receives a 2xx or 3xx status code, the container is considered to be ready.
- TCP Socket: The kubelet attempts to establish a TCP connection with the container. If it succeeds, the container is considered ready.

How Readiness Probes Work

Endpoints can be used as a readiness probe. When a pod is not ready, the **IP:port** of the pod is deleted from the Endpoint and is added to the Endpoint after the pod is ready, as shown in the following figure.

Figure 6-10 How readiness probes work



Exec

The Exec mode is the same as the HTTP GET mode. As shown below, the probe runs the **ls /ready** command. If the file exists, **0** is returned, indicating that the pod is ready. Otherwise, a non-zero status code is returned.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:      # Readiness Probe
          exec:              # Define the ls /ready command.
            command:
            - ls
            - /ready
      imagePullSecrets:
      - name: default-secret
```

Save the definition of the Deployment to the **deploy-read.yaml** file, delete the previously created Deployment, and use the **deploy-read.yaml** file to recreate the Deployment.

```
# kubectl delete deploy nginx
deployment.apps "nginx" deleted

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

The **nginx** image does not contain the **/ready** file. Therefore, the container is not in the **Ready** status after the creation, as shown below. Note that the values in the **READY** column are **0/1**, indicating that the containers are not ready.

```
# kubectl get po
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
```

Create a Service.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
```

```
 - name: service0
   targetPort: 80
   port: 8080
   protocol: TCP
 type: ClusterIP
```

Check the Service. If there are no values in the **Endpoints** line, no Endpoints are found.

```
$ kubectl describe svc nginx
Name:              nginx
......
Endpoints:
......
```

If a **/ready** file is created in the container to make the readiness probe succeed, the container is in the **Ready** status. Check the pod and endpoints. It is found that the container for which the **/ready** file is created is ready and an endpoint is added.

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

# kubectl get po -o wide
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
NAME      ENDPOINTS           AGE
nginx     192.168.93.169:80   14d
```

HTTP GET

The configuration of a readiness probe is the same as that of a **liveness probe**, which is also in the **containers** field of the pod description template. As shown below, the readiness probe sends an HTTP request to the pod. If the probe receives **2xx** or **3xx**, the pod is ready.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:           # readinessProbe
          httpGet:                # HTTP GET definition
            path: /read
            port: 80
```

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

TCP Socket

The following example shows how to define a TCP Socket-type probe.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:             # readinessProbe
          tcpSocket:                # TCP Socket definition
            port: 80
      imagePullSecrets:
      - name: default-secret
```

Advanced Settings of a Readiness Probe

Similar to a liveness probe, a readiness probe also has the same advanced configuration items. The output of the **describe** command of the **nginx** pod is as follows:

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

This is the detailed configuration information of the readiness probe.

- **delay=0s** indicates that the probe starts immediately after the container is started.
- **timeout=1s** indicates that the container must respond to the probe within 1s. Otherwise, it is considered as a failure.
- **period=10s** indicates that the probe is performed every 10s.
- **#success=1** indicates that the probe is considered successful as long as the probe succeeds once.
- **#failure=3** indicates that the probe is considered failed if it fails for three consecutive times.

These are the default configurations when the probe is created. You can customize them as follows:

```
        readinessProbe:      # Readiness Probe
          exec:              # Define the ls /readiness/ready command
```

```
          command:
          - ls
          - /readiness/ready
        initialDelaySeconds: 10    # Readiness probes are initiated after the container has started for 10s.
        timeoutSeconds: 2          # The container must respond within 2s. Otherwise, it is considered as a
failure.
        periodSeconds: 30          # The probe is performed every 30s.
        successThreshold: 1        # The container is considered ready as long as the probe succeeds once.
        failureThreshold: 3        # The probe is considered to be failed after three consecutive failures.
```

6.5 NetworkPolicy

NetworkPolicy is a Kubernetes object used to restrict pod access. In CCE, by setting network policies, you can define ingress rules specifying the addresses to access pods or egress rules specifying the addresses pods can access. This is equivalent to setting up a firewall at the application layer to further ensure network security.

Network policies depend on the networking add-on of the cluster to which the policies apply. For example, CCE clusters support only ingress rules.

By default, if a namespace does not have any policy, pods in the namespace accept traffic from any source and send traffic to any destination.

NetworkPolicy rules are classified into the following types:

- namespaceSelector: This selects particular namespaces for which all pods should be allowed as ingress sources or egress destinations.
- podSelector: This selects particular pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.
- ipBlock: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. (Currently, CCE does not support this mode.)

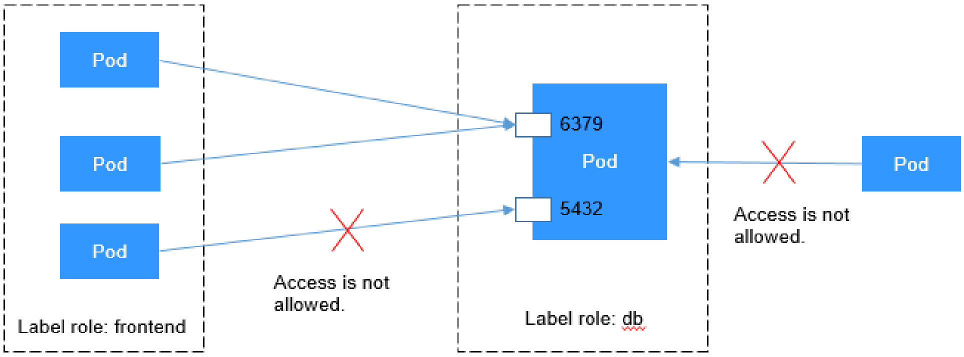
Using Ingress Rules

- **Using podSelector to specify the access scope**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:                  # The rule takes effect for pods with the role=db label.
    matchLabels:
      role: db
ingress:                        #This is an ingress rule.
  - from:
    - podSelector:              #Only traffic from the pods with the role=frontend label is allowed.
        matchLabels:
          role: frontend
ports:                          #Only TCP can be used to access port 6379.
    - protocol: TCP
      port: 6379
```

Diagram:

Figure 6-11 podSelector

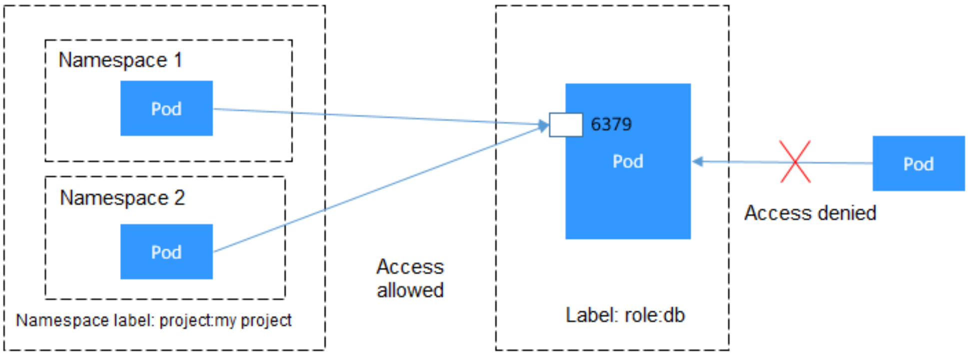


- **Using namespaceSelector to specify the access scope**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector:                  # The rule takes effect for pods with the role=db label.
    matchLabels:
      role: db
ingress:                        #This is an ingress rule.
  - from:
    - namespaceSelector:        # Only traffic from the pods in the namespace with the
"project=myproject" label is allowed.
        matchLabels:
          project: myproject
ports:                          #Only TCP can be used to access port 6379.
    - protocol: TCP
      port: 6379
```

Figure 6-12 shows how namespaceSelector selects ingress sources.

Figure 6-12 namespaceSelector



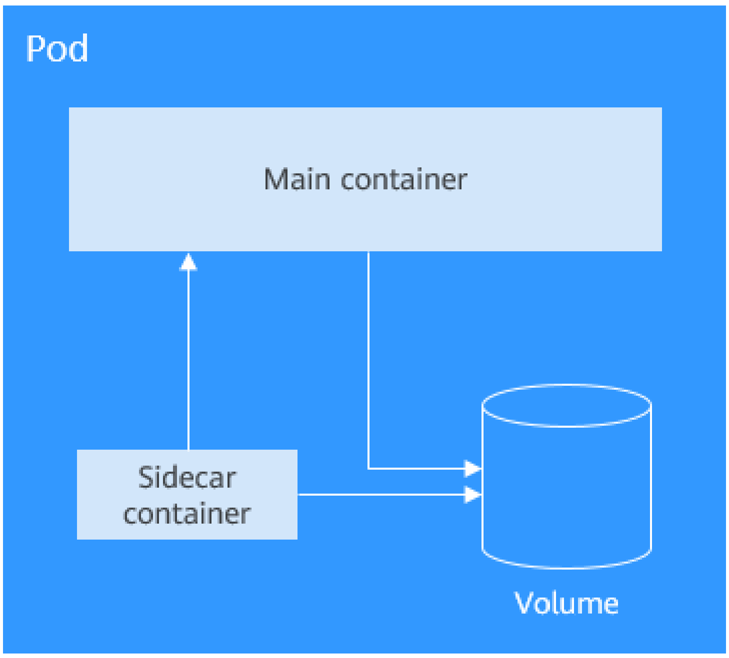
7 Persistent Storage

7.1 Volume

On-disk files in a container are ephemeral. When a container crashes and is then restarted, the files in the container will be lost. When multiple containers run in a pod, files often need to be shared between these containers. Kubernetes provides an abstraction to solve these two problems, that is, storage volumes. Volumes, as part of a pod, cannot be created independently and can only be defined in pods.

All containers in a pod can access its volumes, but the volumes must be attached and can be attached to any directory in the container.

The following figure shows how a storage volume is used between containers in a pod.



A volume will no longer exist if the pod to which it is attached does not exist. However, files in the volume may outlive the volume, depending on the volume type.

Volume Types

Kubernetes supports multiple types of volumes. The most commonly used ones are as follows:

- emptyDir: an empty volume used for temporary storage
- hostPath: a volume that mounts a directory of the host into your pod
- ConfigMap and secret: special volumes that inject or pass information to your pod. For details about how to mount ConfigMaps and secrets, see **ConfigMap** and **Secret**.
- persistentVolumeClaim: Kubernetes persistent storage class. For details, see **PersistentVolume, PersistentVolumeClaim, and StorageClass**.

emptyDir

emptyDir is an empty volume in which your applications can read and write the same files. The lifetime of an emptyDir volume is the same as that of the pod it belongs to. After the pod is deleted, data in the volume is also deleted.

Some uses of an emptyDir volume are as follows:

- scratch space, such as for a disk-based merge sort
- checkpointing a long computation for recovery from crashes

Example emptyDir configuration:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

emptyDir volumes are stored on the disks of the node where the pod is located. You can also set the storage medium to the node memory, for example, by setting **medium** to **Memory**.

```
volumes:
  - name: html
    emptyDir:
      medium: Memory
```

HostPath

hostPath is a persistent storage volume. Data in an emptyDir volume will be deleted when the pod is deleted, but not the case for a hostPath volume. Data in a hostPath volume will still be stored in the node path to which the volume was mounted. If the pod is re-created and scheduled to the same node, after a new hostPath volume is mounted, previous data written by the pod can still be read.

Data stored in hostPath volumes is related to the node. Therefore, hostPath is not suitable for applications such as databases. For example, if the pod in which a database instance runs is scheduled to another node, the read data will be totally different.

Therefore, you are not advised to use hostPath to store cross-pod data, because after the pod is rebuilt, it will be randomly scheduled to a node, which may cause inconsistency when data is written.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: nginx:alpine
    name: hostpath-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      path: /data
```

7.2 PersistentVolume, PersistentVolumeClaim, and StorageClass

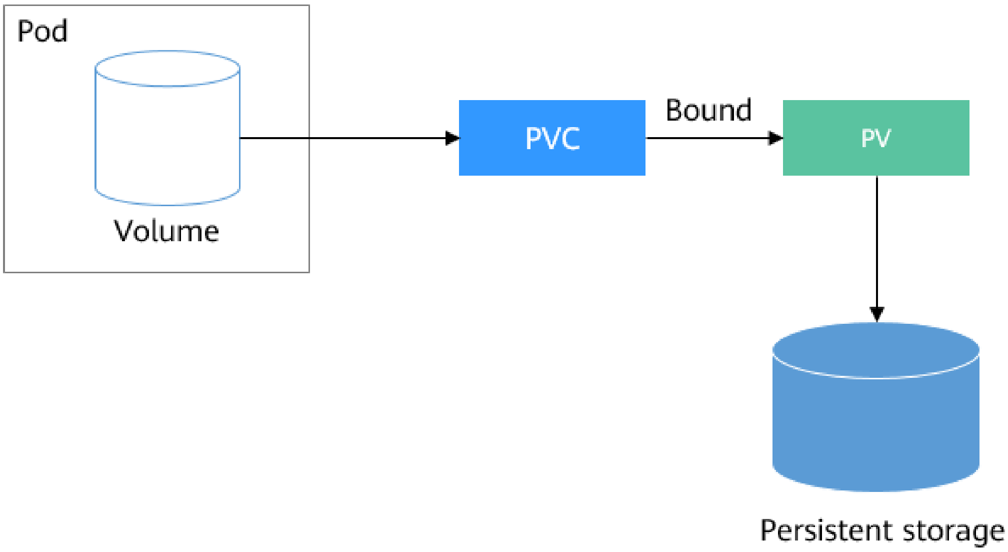
hostPath volumes are used for persistent storage. However, hostPath volumes are node-specific. Writing data into hostPath volumes after a node restart may cause data inconsistency.

If you want to read the previously written data after a pod is rebuilt and scheduled again, you can count on network storage. Typically, a cloud vendor provides at least three classes of network storage: block storage, file storage, and object storage, such as EVS, SFS, and OBS provided by HUAWEI CLOUD. Kubernetes decouples how storage is provided from how it is consumed by introducing two API objects: PersistentVolume (PV) and PersistentVolumeClaim (PVC). You only need to request the storage resources you want, without being exposed to the details of how they are implemented.

- A PV describes a persistent data storage volume. It defines a directory for persistent storage on a host machine, for example, a mount directory of a network file system (NFS).
- A PVC describes the attributes of the PV that a pod wants to use, such as the volume capacity and read/write permissions.

To allow a pod to use PVs, a Kubernetes cluster administrator needs to set the network storage class and provides the corresponding PV descriptors to Kubernetes. You only need to create a PVC and bind the PVC with the volumes in the pod so that you can store data. The following figure shows the relationship between PVs and PVCs.

Figure 7-1 Binding a PVC to a PV



CSI

Kubernetes Container Storage Interface (CSI) can be used to develop plug-ins to support specific storage volumes. For example, in the namespace named kube-system, as shown in **Namespace for Grouping Resources**, **everest-csi-controller-*** and **everest-csi-driver-*** are the storage controllers and drivers developed by HUAWEI CLOUD CCE. With these drivers, you can use cloud storage services on HUAWEI CLOUD, such as EVS, SFS, and OBS.

```
$ kubectl get po --namespace=kube-system
NAME                                      READY   STATUS    RESTARTS   AGE
everest-csi-controller-6d796fb9c5-v22df   2/2     Running   0          9m11s
everest-csi-driver-snzrr                  1/1     Running   0          12m
everest-csi-driver-ttj28                  1/1     Running   0          12m
everest-csi-driver-wtrk6                  1/1     Running   0          12m
```

PV

Each PV contains the specification and status of the volume. For example, a file system is created in HUAWEI CLOUD SFS. The file system ID is **68e4a4fd-d759-444b-8265-20dc66c8c502**, and the mount point is **sfs-nas01.cn-north-4b.myhuaweicloud.com:/share-96314776**. If you want to use this file system in CCE, you need to create a PV to describe the volume. The following is an example PV.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  accessModes:
  - ReadWriteMany                      # Read/write mode
  capacity:
    storage: 10Gi                    # PV capacity
  csi:
    driver: nas.csi.everest.io         # Driver to be used.
    fsType: nfs                      # File system type
    volumeAttributes:
      everest.io/share-export-location: sfs-nas01.cn-north-4b.myhuaweicloud.com:/share-96314776   # Mount
```



```
point
    volumeHandle: 68e4a4fd-d759-444b-8265-20dc66c8c502                                  # File system ID
```

Fields under **csi** in the example above are exclusively used in HUAWEI CLOUD CCE.

Next, create the PV.

```
$ kubectl create -f pv.yaml
persistentvolume/pv-example created

$ kubectl get pv
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM    STORAGECLASS   REASON   AGE
pv-example       10Gi       RWX            Retain           Available                                    4s
```

RECLAIM POLICY indicates the PV reclaim policy. The value **Retain** indicates that the PV is retained after the PVC is released. If the value of **STATUS** is **Available**, the PV is available.

PVC

A PVC can be bound to a PV. The following is an example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi              # Storage capacity
  volumeName: pv-example         # PV name
```

Create the PVC.

```
$ kubectl create -f pvc.yaml
persistentvolumeclaim/pvc-example created

$ kubectl get pvc
NAME          STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-example   Bound    pv-example   10Gi       RWX                           9s
```

The command output shows that the PVC is in **Bound** state and the value of **VOLUME** is **pv-example**, indicating that the PVC is bound to a PV.

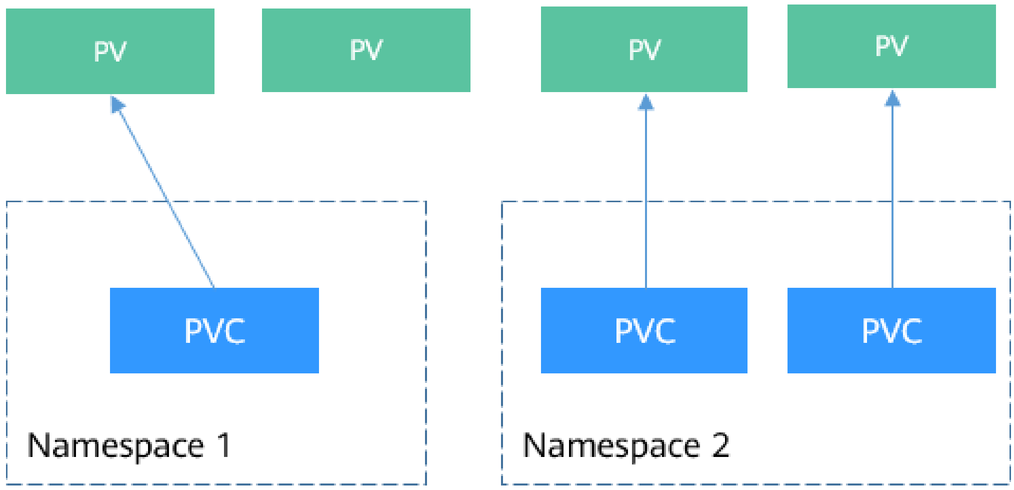
Now check the PV status.

```
$ kubectl get pv
NAME         CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   REASON   AGE
pv-example   10Gi       RWX            Retain           Bound    default/pvc-example                           50s
```

The status of the PVC is also **Bound**. The value of **CLAIM** is **default/pvc-example**, indicating that the PV is bound to the PVC named **pvc-example** in the default namespace.

Note that PVs are cluster-level resources and do not belong to any namespace, while PVCs are namespace-level resources. PVs can be bound to PVCs of any namespace. Therefore, the namespace name "default" is displayed under **CLAIM**.

Figure 7-2 Relationship between PVs and PVCs



StorageClass

Although PVs and PVCs allow you to consume abstract storage resources, you may need to configure multiple fields to create PVs and PVCs (such as the **csi** field structure in the PV), and PVs/PVCs are generally managed by the cluster administrator, which can be inconvenient when you need PVs with varying attributes for different problems.

To solve this problem, Kubernetes supports dynamic PV provisioning to create PVs automatically. The cluster administrator can deploy a PV provisioner and define the corresponding StorageClass. In this way, developers can select the storage class to be created when creating a PVC. The PVC transfers the StorageClass to the PV provisioner, and the provisioner automatically creates a PV. In CCE, storage classes such as csi-disk, csi-nas, and csi-obs are supported. After **storageClassName** is added to a PVC, PVs can be automatically provisioned and underlying storage resources can be automatically created.

NOTE

The following describes how to create a file system in CCE clusters of v1.15 and later, which is different from that for CCE clusters of v1.13 and earlier. For details, see .

Run the following command to query the storage classes that CCE supports. You can use the CSI plug-ins provided by CCE to customize a storage class, which functions similarly as the default storage classes in CCE.

```
# kubectl get sc
NAME              PROVISIONER                AGE
csi-disk          everest-csi-provisioner    17d      # Storage class for EVS disks
csi-disk-topology   everest-csi-provisioner    17d      # Storage class for EVS disks with delayed
association
csi-nas           everest-csi-provisioner    17d      # Storage class for SFS file systems
csi-obs           everest-csi-provisioner    17d      # Storage class for OBS buckets
csi-sfsturbo        everest-csi-provisioner    17d      # Storage class for SFS Turbo file systems
```

Use **storageClassName** to create a PVC.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```



```
  name:  pvc-sfs-auto-example
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas        # StorageClass
```

NOTE

PVCs cannot be directly created by using the storageClassName **csi-sfsturbo**.

Create a PVC and view the PVC and PV details.

```
$ kubectl create -f pvc2.yaml
persistentvolumeclaim/pvc-sfs-auto-example created

$ kubectl get pvc
NAME                  STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
pvc-sfs-auto-example   Bound    pvc-1f1c1812-f85f-41a6-a3b4-785d21063ff3   10Gi       RWX            csi-
nas        29s

$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                          STORAGECLASS   REASON   AGE
pvc-1f1c1812-f85f-41a6-a3b4-785d21063ff3   10Gi       RWO            Delete           Bound    default/pvc-sfs-
auto-example  csi-nas                 20s
```

The command output shows that after a StorageClass is used, a PVC and a PV are created and they are bound to each other.

After a StorageClass is set, PVs can be automatically created and maintained. Users only need to specify StorageClassName when creating a PVC, which greatly reduces the workload.

Note that the types of StorageClassName vary among vendors. In this section, HUAWEI CLOUD SFS is used as an example. For details about other storage classes, see .

Using a PVC in a Pod

After a PVC is available, you can directly bind the PVC to a volume in the pod template and then mount the volume to the pod, as shown in the following example. You can also directly create a PVC in a StatefulSet. For details, see **StatefulSet**.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        volumeMounts:
```



```
    - mountPath: /tmp                          # Mount path
      name: pvc-sfs-example
  restartPolicy: Always
  volumes:
  - name: pvc-sfs-example
    persistentVolumeClaim:
      claimName:  pvc-example                  # PVC name
```

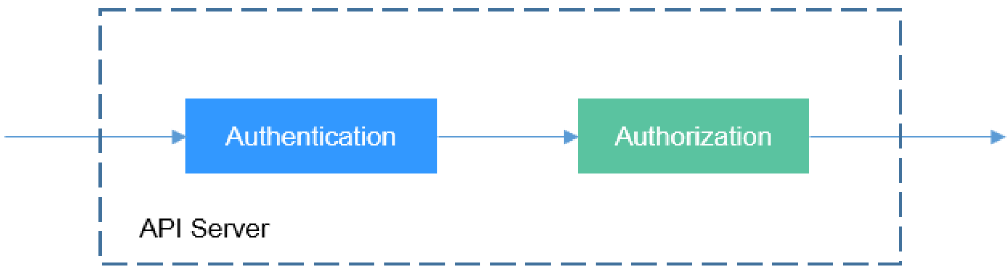
8 Authentication and Authorization

8.1 ServiceAccount

All access requests to Kubernetes resources are processed by the API Server, regardless of whether the requests are from an external system. Therefore, the requests must be authenticated and authorized before they are sent to Kubernetes resources.

- Authentication: used for user identity authentication. Kubernetes uses different authentication mechanisms for external and internal service accounts. For details, see **Authentication and ServiceAccounts**.
- Authorization: used for controlling users' access to resources. Currently, the role-based access control (RBAC) mechanism is used to authorize users to access resources. For details, see **RBAC**.

Figure 8-1 Authentication and authorization of the API Server



Authentication and ServiceAccounts

Kubernetes users are classified into service accounts (ServiceAccounts) and common accounts.

- A ServiceAccount is bound to a namespace and associated with a set of credentials stored in a secret. When a pod is created, the secret is mounted to the pod so that the pod can be called by the API Server.
- Kubernetes does not have objects that represent common accounts. By default, these accounts are managed by external services. For example, CCE

users on HUAWEI CLOUD are managed by Identity and Access Management (IAM).

The following only focuses on ServiceAccounts.

Similar to pods and ConfigMaps, ServiceAccounts are resources in Kubernetes and apply to independent namespaces. That is, a ServiceAccount named **default** is automatically created when a namespace is created.

Run the following command to view ServiceAccounts:

```
$ kubectl get sa
NAME     SECRETS   AGE
default  1         30d
```

In addition, Kubernetes automatically creates a secret for a ServiceAccount. Run the following command to view the secret:

```
$ kubectl describe sa default
Name:                default
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   default-token-vssmw
Tokens:              default-token-vssmw
Events:              <none>
```

In the pod definition file, you can assign a ServiceAccount to a pod by specifying an account name. If no account name is specified, the default ServiceAccount is used. When receiving a request with an authentication token, the API Server uses the token to check whether the ServiceAccount associated with the client that sends the request allows the request to be executed.

Creating a ServiceAccount

Run the following command to create a ServiceAccount:

```
$ kubectl create serviceaccount sa-example
serviceaccount/sa-example created

$ kubectl get sa
NAME         SECRETS   AGE
default      1         30d
sa-example   1         2s
```

The token associated with the ServiceAccount has been created.

```
$ kubectl describe sa sa-example
Name:                sa-example
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   sa-example-token-c7bqx
Tokens:              sa-example-token-c7bqx
Events:              <none>
```

Check the secret content. You can find the **ca.crt**, **namespace**, and **token** data.

```
$ kubectl describe secret sa-example-token-c7bqx
Name:         sa-example-token-c7bqx
...
Data
====
ca.crt:     1082 bytes
```

```
namespace:  7 bytes
token:      <token content>
```

Using a ServiceAccount in a Pod

It is convenient to use a ServiceAccount in a pod. You only need to specify the name of the ServiceAccount.

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-example
spec:
  serviceAccountName: sa-example
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
  imagePullSecrets:
  - name: default-secret
```

Create a pod and view its information. You can see that **sa-example-token-c7bqx** is mounted to the pod, that is, the token corresponding to the ServiceAccount **sa-example**. That is, the pod uses the token for authentication.

```
$ kubectl create -f sa-pod.yaml
pod/sa-example created

$ kubectl get pod
NAME                                    READY   STATUS             RESTARTS   AGE
sa-example                               0/1     running            0          5s

$ kubectl describe pod sa-example
...
Containers:
  sa-example:
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from sa-example-token-c7bqx (ro)
```

You can also view the corresponding file in the pod.

```
$ kubectl exec -it sa-example -- /bin/sh
/ # cd /run/secrets/kubernetes.io/serviceaccount
/run/secrets/kubernetes.io/serviceaccount # ls
ca.crt     namespace  token
```

As shown above, in a containerized application, **ca.crt** and **token** can be used to access the API Server.

Then check whether the authentication takes effect. In a Kubernetes cluster, a Service named **kubernetes** is created for the API Server by default. The API Server can be accessed through this service.

```
$ kubectl get svc
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
kubernetes     ClusterIP   10.247.0.1       <none>        443/TCP         34
```

Go to the pod and run the **curl** command. If the following information is displayed, you do not have the permission.

```
$ kubectl exec -it sa-example -- /bin/sh
/ # curl https://kubernetes
```

```
curl: (60) SSL certificate problem: unable to get local issuer certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
```

Use **ca.crt** and **token** for authentication. The specific procedure is as follows: Place **ca.crt** in the environment variable **CURL_CA_BUNDLE**, and run the **curl** command to specify the certificate using **CURL_CA_BUNDLE**. Place the token content in **TOKEN** and use the token to access the API Server.

```
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:serviceaccount:default:sa-example\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
```

As shown above, the authentication is successful, but the API Server returns **cannot get path \"/\""**, indicating that the API Server can be accessed only after being authorized. For details about the authorization mechanism, see **RBAC**.

8.2 RBAC

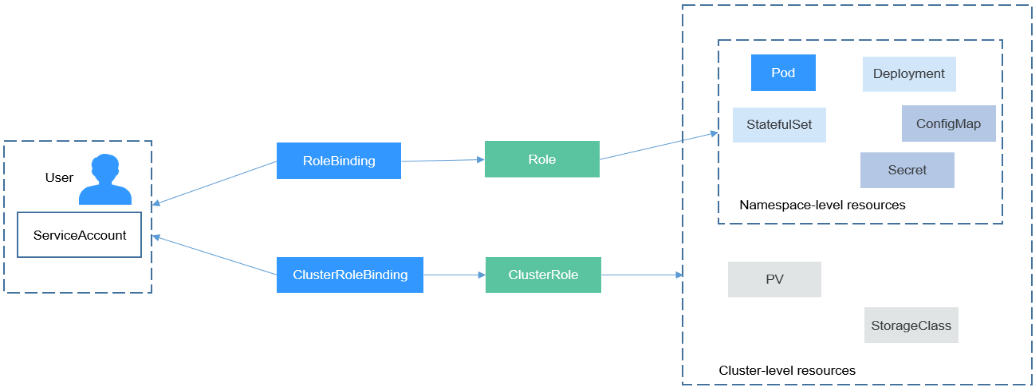
RBAC Resources

In Kubernetes, the RBAC mechanism is used for authorization. RBAC authorization uses four types of resources for configuration.

- Role: defines a set of rules for accessing Kubernetes resources in a namespace.
- RoleBinding: defines the relationship between users and roles.
- ClusterRole: defines a set of rules for accessing Kubernetes resources in a cluster (including all namespaces).
- ClusterRoleBinding: defines the relationship between users and cluster roles.

Role and ClusterRole specify actions that can be performed on specific resources. RoleBinding and ClusterRoleBinding bind roles to specific users, user groups, or ServiceAccounts. See the following figure.

Figure 8-2 Role binding



Creating a Role

The procedure for creating a Role is very simple. To be specific, specify a namespace and then define rules. The rules in the following example are to allow GET and LIST operations on pods in the default namespace.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default                          # Namespace
  name: role-example
rules:
- apiGroups: [""]
  resources: ["pods"]                         # The pod can be accessed.
  verbs: ["get", "list"]                      # The GET and LIST operations can be performed.
```

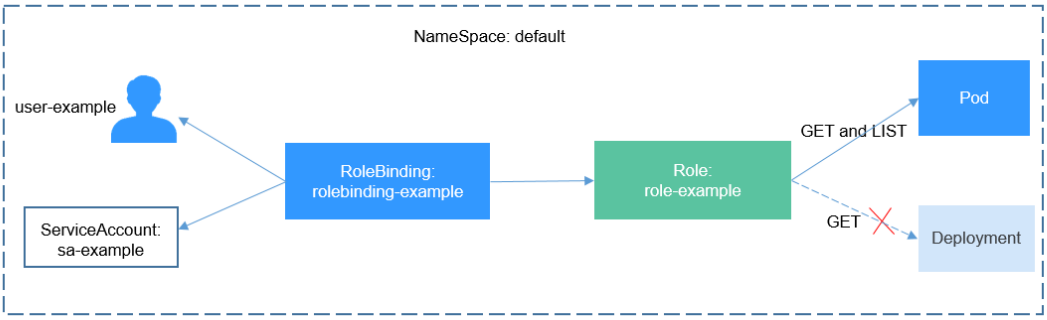
Creating a RoleBinding

After creating a Role, you can bind the Role to a specific user, which is called RoleBinding. The following is an example.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rolebinding-example
  namespace: default
subjects:                                # Specified user
- kind: User                             # Common user
  name: user-example
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount                   # ServiceAccount
  name: sa-example
  namespace: default
roleRef:                                 # Specified Role
  kind: Role
  name: role-example
  apiGroup: rbac.authorization.k8s.io
```

The **subjects** is used to bind the Role to a user. The user can be an external common user or a ServiceAccount. For details about the two user types, see **ServiceAccount**. The following figure shows the binding relationship.

Figure 8-3 A RoleBinding binds the Role to the user.



Then check whether the authorization takes effect.

In **Using a ServiceAccount**, a pod is created and the ServiceAccount **sa-example** is used. The Role **role-example** is bound to **sa-example**. Access the pod and run the **curl** command to access resources through the API Server to check whether the permission takes effect.

Use **ca.crt** and **token** corresponding to **sa-example** for authentication and query all pod resources (**LIST** in **Creating a Role**) in the default namespace.

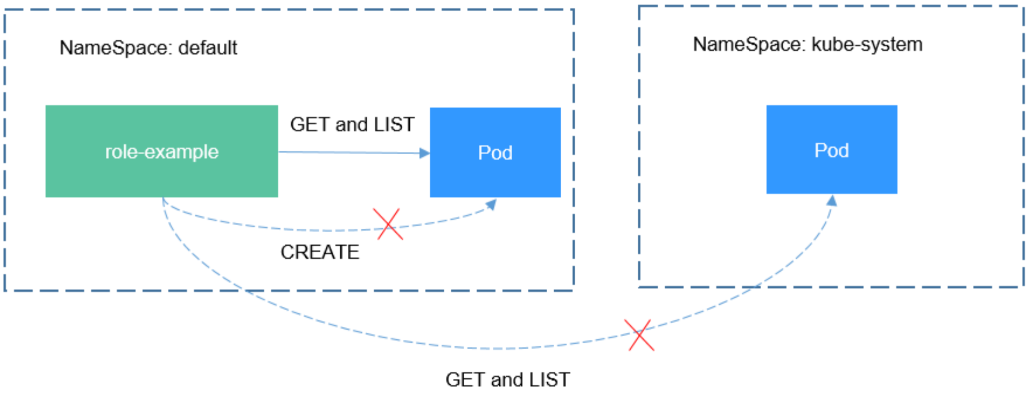
```
$ kubectl exec -it sa-example -- /bin/sh
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "10377013"
  },
  "items": [
    {
      "metadata": {
        "name": "sa-example",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/pods/sa-example",
        "uid": "c969fb72-ad72-4111-a9f1-0a8b148e4a3f",
        "resourceVersion": "10362903",
        "creationTimestamp": "2020-07-15T06:19:26Z"
      },
      "spec": {
...
```

If the returned result is normal, **sa-example** has permission to list pods. Query the Deployment again. If the following information is displayed, you do not have the permission to access the Deployment.

```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/deployments
...
  "status": "Failure",
  "message": "deployments is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list 
resource \"deployments\" in API group \"\" in the namespace \"default\"",
...
```

Role and RoleBinding apply to namespaces and can isolate permissions to some extent. As shown in the following figure, **role-example** defined above cannot access resources in the **kube-system** namespace.

Figure 8-4 Role and RoleBinding applied to namespaces



Continue to access the pod. If the following information is displayed, you do not have the permission.

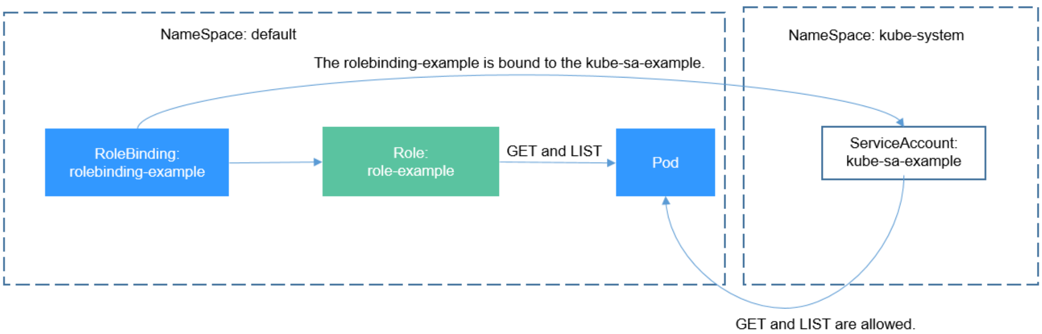
```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/kube-system/pods
...
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list resource \"pods\" in API group \"\" in the namespace \"kube-system\"",
  "reason": "Forbidden",
...
```

In RoleBinding, you can also bind the ServiceAccounts of other namespaces by adding them under the **subjects** field.

```
subjects:                          # Specified user
- kind: ServiceAccount                 # ServiceAccount
  name: kube-sa-example
  namespace: kube-system
```

Then the ServiceAccount **kube-sa-example** in **kube-system** can perform GET and LIST operations on pods in the default namespace, as shown in the following figure.

Figure 8-5 Cross-namespace access



ClusterRole and ClusterRoleBinding

Compared with Role and RoleBinding, ClusterRole and ClusterRoleBinding have the following differences:

- ClusterRole and ClusterRoleBinding do not need to define the **namespace** field.

- ClusterRole can define cluster-level resources.

You can see that ClusterRole and ClusterRoleBinding control cluster-level permissions.

In Kubernetes, many ClusterRoles and ClusterRoleBindings are defined by default.

```
$ kubectl get clusterroles
NAME                                                              AGE
admin                                                             30d
cceaddon-prometheus-kube-state-metrics                            6d3h
cluster-admin                                                     30d
coredns                                                           30d
custom-metrics-resource-reader                                    6d3h
custom-metrics-server-resources                                   6d3h
edit                                                              30d
prometheus                                                        6d3h
system:aggregate-customedhorizontalpodautoscalers-admin           6d2h
system:aggregate-customedhorizontalpodautoscalers-edit            6d2h
system:aggregate-customedhorizontalpodautoscalers-view            6d2h
....
view                                                              30d

$ kubectl get clusterrolebindings
NAME                                                    AGE
authenticated-access-network                            30d
authenticated-packageversion                            30d
auto-approve-csrs-for-group                             30d
auto-approve-renewals-for-nodes                         30d
auto-approve-renewals-for-nodes-server                  30d
cceaddon-prometheus-kube-state-metrics                  6d3h
cluster-admin                                           30d
cluster-creator                                         30d
coredns                                                 30d
csrs-for-bootstrapping                                  30d
system:basic-user                                       30d
system:ccehpa-rolebinding                               6d2h
system:cluster-autoscaler                               6d1h
...
```

The most important and commonly used ClusterRoles are as follows:

- view: has the permission to view namespace resources.
- edit: has the permission to modify namespace resources.
- admin: has all permissions on the namespace.
- cluster-admin: has all permissions on the cluster.

Run the **kubectl describe clusterrole** command to view the permissions of each rule.

Generally, the four ClusterRoles are bound to users to isolate permissions. Note that Roles (rules and permissions) are separated from users. You can flexibly control permissions by combining the two through RoleBinding.

9 Auto Scaling

In **Pod Orchestration and Scheduling**, we introduce controllers such as Deployment to control the number of pod replicas. You can adjust the number of replicas to manually scale your applications. However, manual scaling is sometimes complex and fails to cope with unexpected traffic spikes.

Kubernetes supports auto scaling of pods and cluster nodes. You can set rules to trigger auto scaling when certain metrics (such as CPU usage) reach the configured threshold.

Prometheus and Metrics Server

A prerequisite for auto scaling is that your container running data can be collected, such as number of cluster nodes/pods, and CPU and memory usage of containers. Kubernetes does not provide such monitoring capabilities itself. You can use extensions to monitor and collect your data.

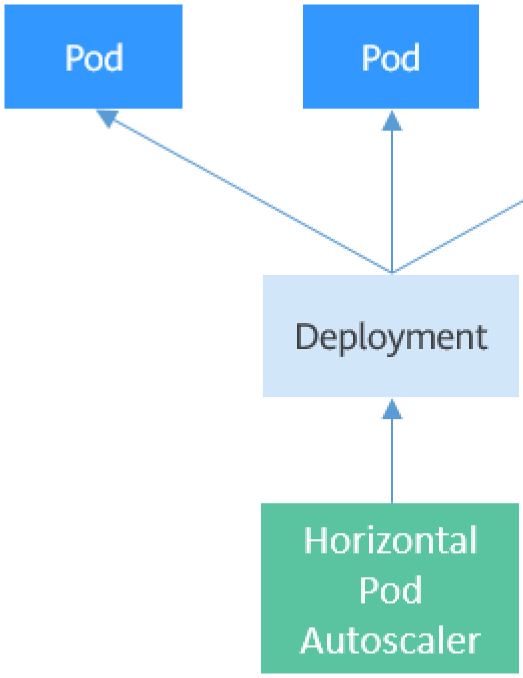
- **Prometheus** is an open source monitoring and alarming framework that can collect multiple types of metrics. Prometheus has been a standard monitoring solution of Kubernetes.
- **Metrics Server** is a cluster-wide aggregator of resource utilization data. Metrics Server collects metrics from the Summary API exposed by kubelet. These metrics are set for core Kubernetes resources, such as pods, nodes, containers, and Services. Metrics Server provides a set of standard APIs for external systems to collect these metrics.

Horizontal Pod Autoscaler (HPA) can work with Metrics Server to implement auto scaling based on the CPU and memory usage. It can also work with Prometheus to implement auto scaling based on custom monitoring metrics.

How HPA Works

HPA is a controller that controls horizontal pod scaling. HPA periodically checks the pod metrics, calculates the number of replicas required to meet the target values configured for HPA resources, and then adjusts the value of the **replicas** field in the target resource object (such as a Deployment).

Figure 9-1 HPA working mechanism



You can configure one or more metrics for the HPA. When configuring a single metric, you only need to sum up the current pod metrics, divide the sum by the expected target value, and then round up the result to obtain the expected number of replicas. For example, if a Deployment controls three pods, the CPU usage of each pod is 70%, 50%, and 90%, and the expected CPU usage configured in the HPA is 50%, the expected number of replicas is calculated as follows: (70 + 50 + 90)/50 = 4.2. The result is rounded up to 5. That is, the expected number of replicas is 5.

If multiple metrics are configured, the expected number of replicas of each metric is calculated and the maximum value will be used.

Using the HPA

The following example demonstrates how to use the HPA. First, use the Nginx image to create a Deployment with four replicas.

```
$ kubectl get deploy
NAME               READY     UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   4/4       4            4           77s

$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-7cc6fd654c-5xzlt   1/1       Running   0          82s
nginx-deployment-7cc6fd654c-cwjzg   1/1       Running   0          82s
nginx-deployment-7cc6fd654c-dffkp   1/1       Running   0          82s
nginx-deployment-7cc6fd654c-j7mp8   1/1       Running   0          82s
```

Create an HPA. The expected CPU usage is 70% and the number of replicas ranges from 1 to 10.

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
```



```
  name: scale
  namespace: default
spec:
  maxReplicas: 10                    # Maximum number of replicas of the target resource
  minReplicas: 1                     # Minimum number of replicas of the target resource
  metrics:                           # Metric. The expected CPU usage is 70%.
  - resource:
      name: cpu
      targetAverageUtilization: 70
    type: Resource
  scaleTargetRef:                    # Target resource
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
```

Query the created HPA.

```
$ kubectl create -f hpa.yaml
horizontalpodautoscaler.autoscaling/celue created

$ kubectl get hpa
NAME     REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
scale    Deployment/nginx-deployment   0%/70%    1         10        4          18s
```

In the command output, the expected value of **TARGETS** is **70%**, but the actual value is **0%**. This means that the HPA will perform scale-in. The expected number of replicas can be calculated as follows: (0 + 0 + 0 + 0)/70 = 0. However, the minimum number of replicas has been set to **1**. Therefore, the number of pods is changed to 1. After a while, the number of pods changes to 1.

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-7cc6fd654c-5xzlt   1/1       Running   0          7m41s
```

Query the HPA again and a record similar to the following is displayed under **Events**. In this example, the record indicates that the HPA successfully performed a scale-in 21 seconds ago and the number of pods is changed to 1, and the scale-in is triggered because the values of all metrics are lower than the target values.

```
$ kubectl describe hpa scale
...
Events:
  Type    Reason             Age   From                       Message
  ----    ------             ----  ----                       -------
  Normal  SuccessfulRescale  21s   horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

If you want to query the Deployment details, you can check the records similar to the following under **Events**. In this example, the second record indicates that the number of replicas of the Deployment is set to **1**, which is the same as that in the HPA.

```
$ kubectl describe deploy nginx-deployment
...
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  7m    deployment-controller  Scaled up replica set nginx-
deployment-7cc6fd654c to 4
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled down replica set nginx-
deployment-7cc6fd654c to 1
```

Cluster AutoScaler

The HPA is designed for pods. However, if the cluster resources are insufficient, you can only add nodes. Scaling of cluster nodes could be laborious. Now with clouds, you can add or delete nodes by simply calling APIs.

Cluster Autoscaler is a component provided by Kubernetes for auto scaling of cluster nodes based on the pod scheduling status and resource usage. You can refer to the API documentation of your cloud service provider to implement auto scaling.

For details about the implementation on HUAWEI CLOUD CCE, see .