Cloud Container Engine

Kubernetes Basics

Issue	01
Date	2025-07-29

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Huawei Cloud Computing Technologies Co., Ltd.

Address:	Huawei Cloud Data Center Jiaoxinggong Road Qianzhong Avenue Gui'an New District Gui Zhou 550029 People's Republic of China
Website:	https://www.huaweicloud.com/intl/en-us/

Contents

1 Overview

Kubernetes is an open source container orchestration platform that manages containerized applications across in-cloud hosts. It offers a framework for deploying, scheduling, updating, and managing applications and makes containerized application deployment easy and efficient.

For application developers, Kubernetes can be thought of 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 access our hosted Kubernetes service, Cloud Container Engine (CCE), by using the CCE console, kubectl, or Kubernetes APIs. Before using CCE, familiarize yourself with the following Kubernetes concepts.

Containers and Kubernetes

- **Containers**
- **Kubernetes**
- **Using kubectl to Operate a Cluster**

Pods, Labels, and Namespaces

- **Pods**
- **Liveness Probes**
- **Labels**
- **Namespaces**

Pod Orchestration and Scheduling

- **Deployments**
- **StatefulSets**
- **Jobs and CronJobs**
- **DaemonSets**
- **Affinity and Anti-Affinity**

Configuration Management

- ConfigMaps
- Secrets

Kubernetes Networks

- Container Networks
- Services
- Ingresses
- Readiness Probes
- Network Policies

Persistent Storage

- Volumes
- PVs, PVCs, and Storage Classes

Authentication and Authorization

- Service Accounts
- RBAC

Auto Scaling

- Auto Scaling

2 Basic Concepts

CCE provides highly scalable, high-performance, and enterprise-class Kubernetes clusters. With CCE, you can easily deploy, manage, and scale containerized applications in the cloud.

CCE provides native Kubernetes APIs, supports kubectl, and provides a graphical console, enabling you to have a complete end-to-end experience. Before using it, familiarize yourself with some related basic concepts.

Cluster

A cluster is a combination of cloud resources, such as cloud servers (nodes) and load balancers, for running containers. In a cluster, one or more elastic cloud servers (also called nodes) are deployed in the same subnet to provide compute resource pools for containers.

CCE supports the cluster types shown in the table below.

Cluster Type	Description
CCE standard cluster	CCE standard clusters are for commercial use, which fully support the standard features of open source Kubernetes clusters. CCE standard clusters offer a simple, cost-effective, highly available solution. There is no need to manually manage and maintain master nodes. You can choose between a container tunnel network or a VPC network depending on your service needs. CCE standard clusters are ideal for typical scenarios that do not need special performance or cluster scale requirements.

Cluster Type	Description
CCE Turbo cluster	CCE Turbo clusters run on the Cloud Native 2.0 infrastructure. They feature hardware and software synergy, zero network performance loss, high security and reliability, and intelligent scheduling. They provide you with one-stop cost-effective container services. The Cloud Native 2.0 networks are good for large-scale, high-performance scenarios. In CCE Turbo clusters, container IP addresses are assigned from VPC CIDR blocks, and containers and nodes can be in different subnets. External networks in a VPC can be directly connected to container IP addresses for high performance.
CCE Autopilot cluster	CCE Autopilot allows you to create serverless clusters that offer optimized Kubernetes compatibility and free you from O&M. CCE Autopilot clusters can be deployed without user nodes, simplifying the application deployment. There is no need to purchase, deploy, or manage nodes or maintain their security. You can just focus on the implementation of application service logic, which greatly reduces your O&M costs and improves the reliability and scalability of applications.

For details, see **Buying a CCE Standard/Turbo Cluster**.

Node

In a Kubernetes cluster, nodes run containerized applications. They can be physical servers or virtual machines (VMs) connected over networks. Each node has necessary components installed, such as a container runtime (Docker for example) and kubelet (used to manage containers). **Pod**s, the smallest deployable units, are deployed and run on nodes, which are centrally scheduled and managed by Kubernetes. Nodes are the basic runtime environments in a cluster, ensuring high availability and scalability of applications.

For details, see **Creating a Node**.

Node Pool

In a Kubernetes cluster, a node pool is a group of nodes that have the same configuration and attributes. These nodes usually have the same hardware specifications, OS version, and configurations. A node pool makes it easier to manage and scale cluster resources in batches. You can create node pools of different sizes and configurations to meet different workload scheduling requirements and ensure efficient resource utilization. In addition, node pools support auto scaling. The number of nodes in a node pool can be scaled automatically based on workloads. This improves the resource utilization, flexibility, and scalability of a cluster.

For details, see **Creating a Node Pool**.

VPC

A VPC provides a secure, logically isolated virtual network environment. VPCs provide the same resources as physical networks, and they also provide various advanced network services, such as elastic IP addresses and security groups.

With VPCs, node networks and container networks in CCE clusters can be isolated. You can also configure EIPs and bandwidths for your clusters for more flexible scalability.

For details, see **Creating a VPC with a Subnet**.

Security Group

A security group is a collection of access rules for Elastic Cloud Servers (ECSs) that have the same security requirements and are mutually trusted in a VPC. After a security group is created, you can create different access rules to control who can access the ECSs that are added to this security group.

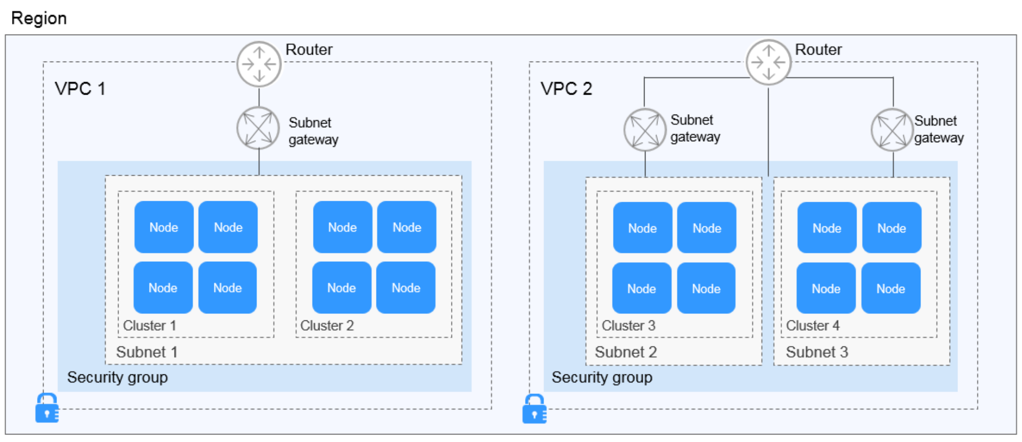
For details, see **Adding a Security Group Rule**.

Relationship Between Clusters, VPCs, Security Groups, and Nodes

As shown in **Figure 2-1**, a region may include multiple VPCs. A VPC consists of one or more subnets. The subnets communicate with each other through subnet gateways. A cluster is created in a subnet. There are the following scenarios:

- Different clusters are created in different VPCs.
- Different clusters are created in the same subnet.
- Different clusters are created in different subnets.

Figure 2-1 Relationship between clusters, VPCs, security groups, and nodes

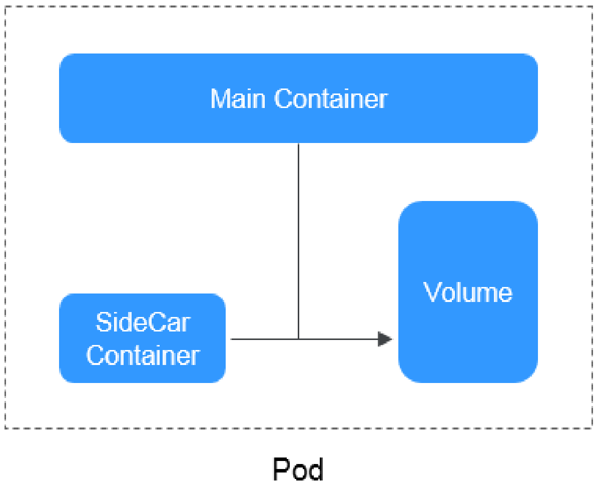


Pod

A pod is the smallest, basic unit for deploying applications or services. It can contain one or more containers, which typically share storage and network resources. Each pod has its own IP address, allowing the containers within the pod to communicate with each other and be accessed by other pods in the same cluster. Kubernetes also offers various policies to manage container execution.

These policies include restart policies, resource requests and limits, and lifecycle hooks.

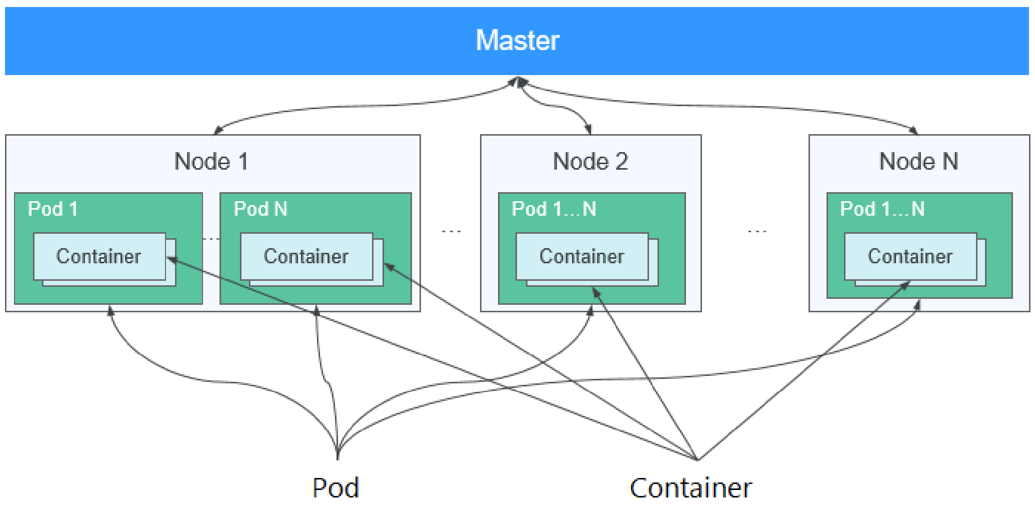
Figure 2-2 Pod



Container

A container is an instance created using a Docker image. Multiple containers can run on the same node (the host). Containers are essentially processes, but they run in their own separate namespaces, unlike actual processes, which run directly on a host machine. Namespaces provide isolation between containers, allowing each container to have its own file system, network API, process ID, and more. This enables OS-level isolation for containers.

Figure 2-3 Relationships between pods, containers, and nodes



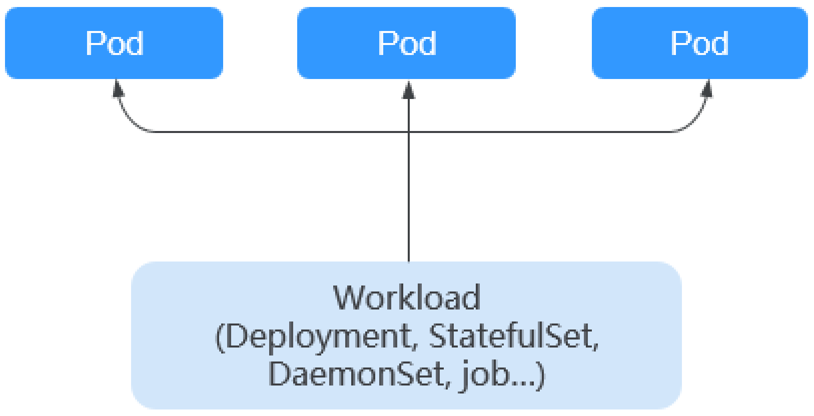
Workload

A workload is an application running in a Kubernetes cluster. No matter how many components are there in your workload, you can run it in a group of pods. A workload is an abstract model of a group of pods. In Kubernetes, there are Deployments, StatefulSets, DaemonSets, jobs, and CronJobs.

- **Deployments** support auto scaling and rolling upgrade. They are ideal for scenarios where pods are completely independent of each other and functionally identical. Typical examples include web applications like Nginx and blog platforms like WordPress.
- **StatefulSets** allow for the organized deployment and removal of pods. Each pod in a StatefulSet has a unique identifier and can communicate with others. StatefulSets are ideal for applications that need persistent storage and communication between pods, like etcd, the distributed key-value store, or MySQL High Availability, the high-availability databases.
- **DaemonSets** guarantee that all or specific nodes have a DaemonSet pod running and automatically deploy DaemonSet pods on newly added nodes in a cluster. They are ideal for services that need to run on every node, like log collection (Fluentd) and monitoring agent (Prometheus node exporter).
- **Jobs** are one-off tasks that guarantee the successful completion of a specific number of pods. They are ideal for one-off tasks, like data backups and batch processing.
- **CronJobs** run tasks on specified schedule. They are ideal for tasks that need to be done regularly, like data synchronization and report generation.

For details, see **Creating a Workload**.

Figure 2-4 Relationship between workloads and pods



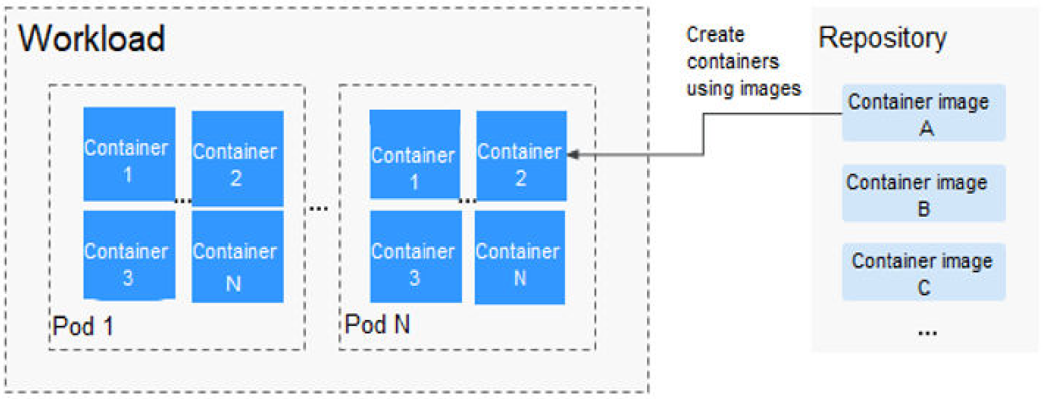
Image

An image is a standard format used to package containerized applications and create containers. Essentially, an image is a specialized file system that includes all the necessary programs, libraries, resources, and configuration files for container runtimes. It also contains configuration parameters like anonymous volumes, environment variables, and users that are required for runtimes. An image does not contain any dynamic data. Once it has been created, the content does not change. When deploying containerized applications, you have the option to use images from Docker Hub, SoftWare Repository for Container (SWR), or your own private image registries. For instance, you can create an image that includes a specific application and all its dependencies, ensuring consistent execution across different environments.

The relationship between an image and a container is akin to that between a class and an instance in object-oriented programming. An image serves as a static blueprint, while a container is its active, running entity. Containers can be created, started, stopped, deleted, and suspended.

For details, see **Pushing an Image**.

Figure 2-5 Relationship between images, containers, and workloads



Namespace

A namespace in Kubernetes is a way to group and organize related resources and objects, such as pods, Services, and Deployments. It logically isolates data from other namespaces, but shares basic resources like CPUs, memory, and storage within the same cluster with them. By deploying different environments in separate namespaces, such as development, testing, and production, you can ensure environmental isolation and simplify management and maintenance tasks.

In Kubernetes, most resource objects, including pods, Services, ReplicationControllers, and Deployments, are associated with the **default** namespace by default. However, there are also cluster-level resources like nodes and PersistentVolumes (PVs) that are not tied to any specific namespace and provide services to resources across all namespaces.

For details, see **Creating a Namespace**.

Service

A Service is used to define access policies for pods. There are different types of Services with their respective values and behaviors:

- ClusterIP: This is the default Service type. Each ClusterIP Service is assigned a unique IP address within the cluster. This IP address is only accessible within the cluster. It cannot be directly accessed from external networks. ClusterIP Services are typically used for internal communications within a cluster.
- NodePort: A NodePort Service opens a static port (NodePort) on all nodes in a cluster. You can access the Service through this port. External systems can contact NodePort Services using the Elastic IPs (EIPs) associated with the nodes over the specified ports.

- LoadBalancer: This type of Service allows you to use the load balancers provided by cloud service providers to expose Services to the Internet. Load balancers can distribute traffic to the NodePort and ClusterIP Services within the cluster.

For details, see **Service Overview**.

Ingress

An ingress controls how Services within a cluster can be accessed from outside the cluster. Ingresses can route traffic based on domain names and paths. They support load balancing, TLS termination, and SSL certificate management. An ingress manages traffic of multiple Services in a unified manner. It acts as an entry point for incoming traffic. This simplifies network configuration, improves cluster scalability and security and is an important way to expose Services in microservices.

For details, see **Ingress Overview**.

Network Policy

Network policies allow you to specify rules for traffic flow between pods. They control whether traffic is allowed or denied to and from a pod based on specified rules to enhance network security for clusters. Network policies allow you to define rules based on pod labels, IP addresses, and ports, limit inbound and outbound traffic, and prevent unauthorized requests, protecting the security of Services in a cluster.

For details, see **Configuring Network Policies to Restrict Pod Access**.

ConfigMap

ConfigMaps are used to store configuration data in key-value pairs. ConfigMaps can decouple configuration details such as configuration files and command-line arguments from pods. With ConfigMaps, you can avoid the need to rebuild container images whenever configurations are shared or updated between pods. ConfigMaps support multiple data formats, such as YAML and JSON. This facilitates application configuration management and ensures maintainability and scalability.

For details, see **Creating a ConfigMap**.

Secret

Secrets store sensitive data, such as passwords, keys, and certificates. Secrets are encrypted to enhance data security. Secrets can be mounted as data volumes or exposed as environment variables to be used in a pod. Secrets can also be used to store authentication information in a cluster. With secrets, you can manage sensitive data separately from the application code to reduce data leakage risks. In addition, you can centrally manage and dynamically update sensitive data to ensure cluster security and flexibility.

For details, see **Creating a Secret**.

Label

Labels are key-value pairs that are attached to objects such as pods, Services, and Deployments. Labels are used to add extra, semantic metadata to objects, enabling users and systems to effortlessly identify, organize, and manage resources.

Label Selector

Label selectors simplify resource management by allowing you to group and select resource objects based on their labels. This enables batch operations, such as traffic distribution, scaling, configuration updates, and monitoring, on the selected resource groups.

Annotation

Annotations are defined as key-value pairs, similar to labels. However, they serve a different purpose and have different constraints.

Labels are used for selecting and managing resources, following strict naming rules and defining metadata for Kubernetes objects. Label selectors use labels to help you select resources.

Annotations, in contrast, are additional information about resources. While Kubernetes does not directly use annotations to control resource behavior, external tools can access the information stored in annotations to extend Kubernetes functions.

PersistentVolume

A PersistentVolume (PV) is a storage resource in a cluster. It can be either a local disk or network storage. It exists independently from pods, so if a pod using a PV is deleted, the data stored in the PV will not be lost.

PersistentVolumeClaim

A PersistentVolumeClaim (PVC) is a request for PVs. It specifies the desired storage size and access mode. Kubernetes will automatically find a suitable PV that meets these requirements.

The relationship between PVCs and PVs is similar to that between pods and nodes. Pods consume node resources and PVCs consume PV resources.

Horizontal Pod Autoscaler for Workload Auto Scaling

Horizontal Pod Autoscaler (HPA) implements horizontal scaling of pods in Kubernetes. HPA enables a Kubernetes cluster to automatically scale in or out pods based on CPU usage, memory usage, or other specified metrics. You can set thresholds for target metrics for HPA to dynamically adjust the pod count to ensure the best application performance.

For details, see **Creating an HPA Policy**.

Cluster Autoscaler for Node Auto Scaling

Node auto scaling refers to automatically adjusting the number of nodes to adapt to changing workloads. Cluster Autoscaler automatically adds nodes when service load increases. As the service load decreases, underutilized nodes are automatically removed to reduce costs. It automatically adjusts the number of nodes in a cluster based on the workloads' resource needs, such as CPU and memory usage, and specified rules. This ensures efficient resource utilization and flexibility.

For details, see **Creating a Node Scaling Policy**.

Affinity and Anti-Affinity

Before an application is containerized, many of its components run on the same VM, and processes need to communicate with each other. During containerization, its processes are packed into different pods and each pod has its own lifecycle. For example, the business process is packed into a pod while the monitoring/logging process or local storage process is packed into another pod. If these pods run on distant nodes, routing between them will be costly and slow.

- Affinity: Pods that closely related to each other are deployed on the same or the nearest node. This can reduce network loss. For instance, if an application needs to frequently communicate with some other application, you can define affinity rules to ensure that these two applications are placed close or even on the same node. By doing so, any potential performance degradation caused by slow routing can be avoided.
- Anti-affinity: Pods of the same application spread across different nodes to achieve higher availability. Once a node is down, the application pods on other nodes are not affected. For example, if an application runs in multiple pods, you can define anti-affinity rules to deploy these pods on different nodes to guarantee the application HA.

For details, see **Overview of Scheduling a Workload**.

Resource Quota

Resource quotas enable administrators to set limits on the overall usage of resources, such as CPU, memory, disk space, and network bandwidth, within namespaces.

Resource Limit (LimitRange)

By default, all containers in Kubernetes have no CPU or memory limit. A LimitRange is a policy used to apply resource limits to objects, like pods, within a namespace.

It offers several constraints that can:

- Restrict the minimum and maximum resource usage for each pod or container in a namespace.
- Set minimum and maximum limits for the storage space that each PVC can request within a namespace.
- Control the ratio between the request and limit for a resource within a namespace.

- Set default requests and limits for compute resources within a namespace and automatically apply them to multiple containers at runtime.

Environment Variable

An environment variable is a variable that is configured in the runtime environment of a container. A maximum of 30 environment variables can be defined in a container template. You can modify environment variables even after workloads are deployed. Workload configuration is quite flexible.

The function of setting environment variables on CCE is the same as that of specifying **ENV** in a Dockerfile.

Chart

For your Kubernetes clusters, you can use **Helm** to manage software packages, which are called charts. Helm is to Kubernetes what **apt** is to Ubuntu or what **yum** is to CentOS. Helm allows you to quickly search for, download, and install charts.

Charts are a packaging format used by Helm. They describe a group of related cluster resource definitions, not an actual container image package. A Helm chart contains a series of YAML files used to deploy Kubernetes applications. You can customize some parameter settings in a Helm chart. When installing a chart, Helm deploys resources in the cluster based on the YAML files defined in the chart. Related container images are not included in the chart. They are pulled from the image repository defined in the YAML files.

Application developers need to push container image packages to the image repository, use Helm charts to package dependencies, and preset some key parameters to simplify application deployment.

Application users can use Helm to search for charts and customize parameter settings. Helm installs applications and their dependencies in the cluster based on the YAML files in a chart. Application users can search for, install, upgrade, roll back, and uninstall applications without defining complex deployment files.

For details, see **Chart Overview**.

3 Containers and Kubernetes

3.1 Containers

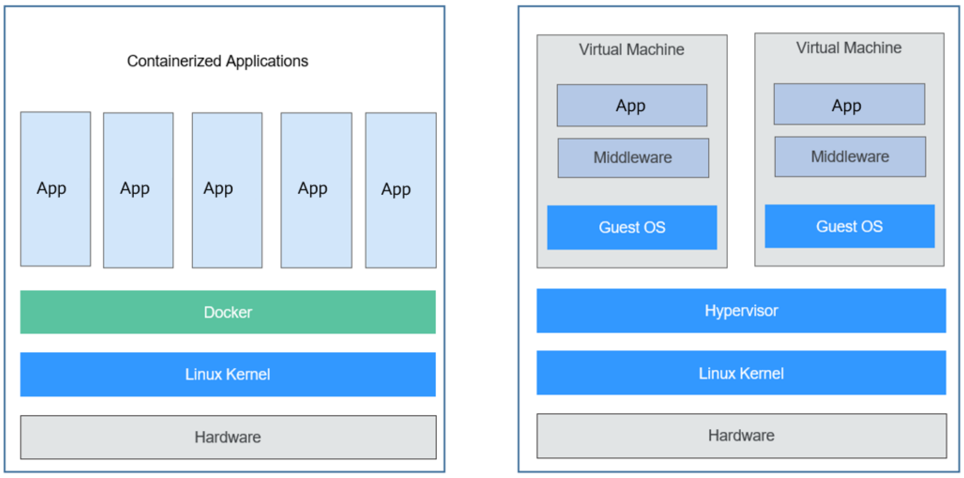
Video Tutorial

Containers and Docker

Containers are a kernel virtualization technology originating with Linux. They provide lightweight virtualization to isolate processes and resources. Containers have become popular since the emergence of Docker. Docker was the first system to make containers fully portable and makes it easy to run your container on any machine you want. It simplifies the packaging of applications, repositories, and dependencies. Even an OS file system can be packaged into a simple portable package that can be used on any machine running Docker.

Containers isolate and allocate resources similarly to VMs. Unlike VMs, however, containers virtualize OSs not hardware, which makes them more portable and efficient.

Figure 3-1 Containers vs VMs



Containers have the following advantages over VMs:

- Better resource utilization

  There is no overhead for virtualizing hardware and running a complete OS, which allows containers to outperform VMs in application execution speed, memory loss, and file storage speed. Compared with a VM with the same configuration, a container can run more applications.
- Faster startup

  A traditional VM typically takes several minutes to start an application, but a Docker container can launch an application in just a few seconds, or even milliseconds. This because containerized applications run directly on the host kernel, and there is no need to load the entire OS. Running applications on containers greatly saves your time in development, testing, and deployment.
- Consistent environment

  Inconsistent development, test, and production environments are a common pain point in application development. As a result, some issues cannot be detected prior to rollout. A Docker container image provides a complete runtime environment except the kernel for applications.
- Easier migration

  Docker provides a consistent execution environment across many platforms, both physical and virtual. Regardless of what platform Docker is running on, the applications run the same way, which makes migrating them much easier. With Docker, you do not have to worry that an application running fine on one platform will fail in a different environment.
- Easier maintenance and extension

  A Docker image is built up from a series of layers that are stacked. When you create a container, you add a container layer on top of image layers. In this way, duplicate layers are reused, which simplifies application maintenance and updates as well as image extension 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 production environments or create your images based on these images. This greatly reduces the image creation costs.

Typical Process of Using Docker Containers

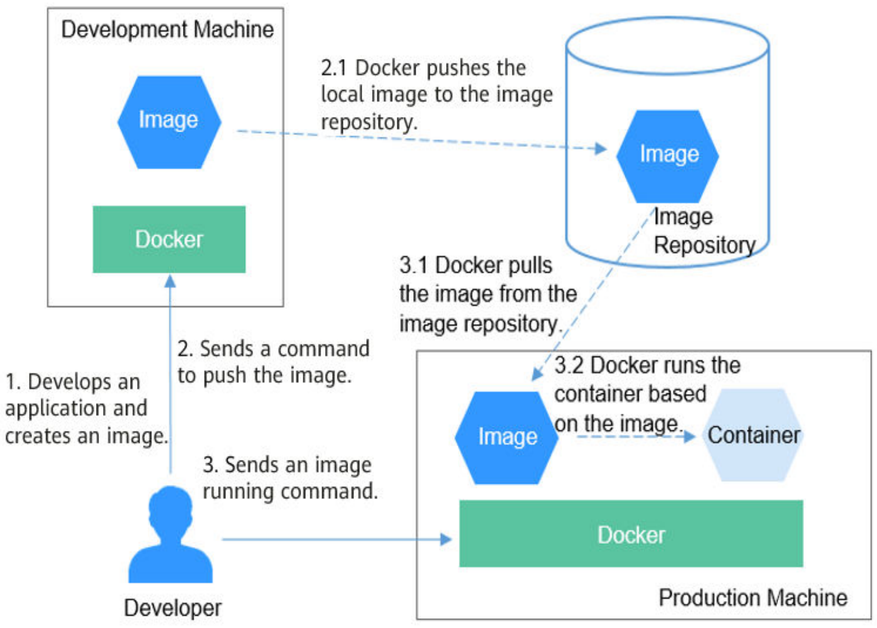
Before using a Docker container, you should know the core components in Docker.

- **Image**: A Docker image is an executable software package that includes the data needed to run an application, including file systems and other metadata, such as executable file path of the runtime.
- **Image repository**: A Docker image repository stores Docker images, which can be shared between different users and computers. You can run an image on the computer where it was created, or you can push it to an image repository, pull it to another computer, and then run it there. Some repositories are public, where anyone can pull images. Some are private, which are only accessible 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, but it is isolated from the host and all other processes running on the host.

The process is also resource-limited, and it can only access and use resources (such as CPU and memory) allocated to it.

Figure 3-2 shows the typical process of using containers.

Figure 3-2 Typical process of using Docker containers


Development Machine
Image
Docker
2.1 Docker pushes the local image to the image repository.
Image
Image Repository
3.1 Docker pulls the image from the image repository.
2. Sends a command to push the image.
1. Develops an application and creates an image.
3.2 Docker runs the container based on the image.
Image
Container
3. Sends an image running command.
Docker
Developer
Production Machine

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

   Docker builds the image and stores it on the machine.
2. The developer sends a command to Docker for pushing the image.

   After receiving the command, Docker pushes the local image to the image repository.
3. The developer sends an image running command to the production machine.

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

Example

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

Installing Docker

Docker is compatible with almost all OSs. Select whichever Docker version best suits your needs.

The following uses CentOS 7.5 64bit (40 GiB) as an example to describe how to quickly install Docker.

1. Add a yum repository.

```
yum install epel-release -y
yum clean all
```

2. Install the required software packages.
```
yum install -y yum-utils device-mapper-persistent-data lvm2
```
3. Configure the Docker yum repository.
```
yum-config-manager --add-repo https://mirrors.huaweicloud.com/docker-ce/linux/centos/docker-
ce.repo
sed -i 's+download.docker.com+mirrors.huaweicloud.com/docker-ce+' /etc/yum.repos.d/docker-ce.repo
```
4. Check the available Docker version.
```
yum list docker-ce --showduplicates | sort -r
```
Information similar to the following is displayed:
```
Loading mirror speeds from cached hostfile
Loaded plugins: fastestmirror
docker-ce.x86_64            3:26.1.4-1.el7            docker-ce-stable
docker-ce.x86_64            3:26.1.3-1.el7            docker-ce-stable
docker-ce.x86_64            3:26.1.2-1.el7            docker-ce-stable
...
```
5. Install Docker of the specified version. To facilitate the subsequent configuration of the image accelerator, use a Docker version from 18.06.0 to 24.0.9.
```
sudo yum install docker-ce-24.0.9 docker-ce-cli-24.0.9 containerd.io
```
Docker 24.0.9 is used in this example. If you choose a different version, replace 24.0.9 with the specific version number.
6. Start Docker.
```
systemctl enable docker  # Set Docker to start automatically upon system boot.
systemctl start docker   # Start Docker.
```
7. Check the installation result.
```
docker --version
```
Information similar to the following is displayed:
```
Docker version 24.0.9, build 2936816
```

Packaging a Docker Image

Docker provides a convenient way to package your application as a Dockerfile, which allows you to create a simple custom Nginx image.

1. To configure an image accelerator, perform the following operations: (An image accelerator can speed up the download of popular open source images, addressing issues with slow or failed downloads from third-party repositories like Docker Hub caused by network problems. Image accelerators are only available in certain regions.)
   a. Log in to the .
   b. In the navigation pane, choose **Image Resources** > **Image Center**. Ensure that Image Center is available in the current region.
   c. Click **Image Accelerator**. In the displayed dialog box, click ❐ to copy the accelerator address.

Figure 3-3 Copying an accelerator address

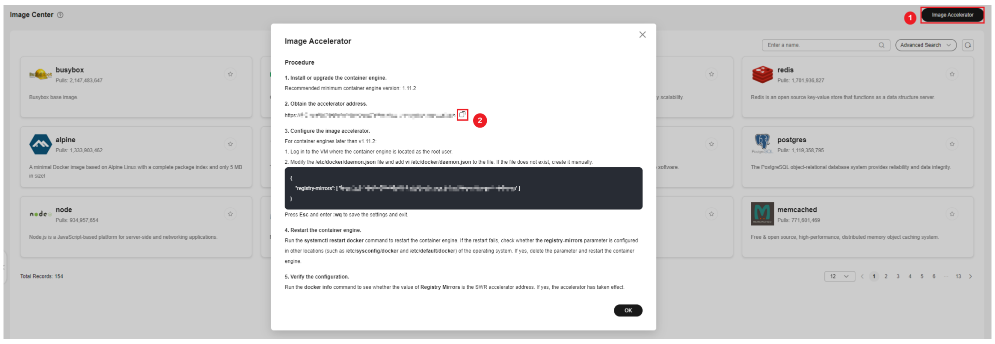


Image Accelerator

Procedure

1. Install or upgrade the container engine.
Recommended minimum container engine version: 1.11.2

2. Obtain the accelerator address.
https://[redacted]

3. Configure the image accelerator.
For container engines later than v1.11.2:
1. Log in to the VM where the container engine is located as the root user.
2. Modify the **/etc/docker/daemon.json** file and add **vi /etc/docker/daemon.json** to the file. If the file does not exist, create it manually.

```
{
  "registry-mirrors": [ "[redacted]" ]
}
```

Press **Esc** and enter **:wq** to save the settings and exit.

4. Restart the container engine.
Run the **systemctl restart docker** command to restart the container engine. If the restart fails, check whether the **registry-mirrors** parameter is configured in other locations (such as **/etc/sysconfig/docker** and **/etc/default/docker**) of the operating system. If yes, delete the parameter and restart the container engine.

5. Verify the configuration.
Run the **docker info** command to see whether the value of **Registry Mirrors** is the SWR accelerator address. If yes, the accelerator has taken effect.

d. Modify the **/etc/docker/daemon.json** file.

```
vim /etc/docker/daemon.json
```

Add the following content to the file:

```
{
    "registry-mirrors": ["accelerator-address"]
}
```

e. Restart the container engine.

```
systemctl restart docker
```

If the restart fails, check whether the **registry-mirrors** parameter is configured in another location of the OS, such as **/etc/sysconfig/docker** or **/etc/default/docker**. If the parameter is configured in another location, delete it from there and restart the container engine.

f. View the Docker details.

```
docker info
```

If the value of **Registry Mirrors** is the accelerator address, the accelerator has been configured.

```
...
Registry Mirrors:
  https://xxx.mirror.swr.myhuaweicloud.com/
...
```

2. Create a file named **Dockerfile** in the **mynginx** directory.

```
mkdir mynginx
cd mynginx
touch Dockerfile
```

3. Edit the **Dockerfile** file.

```
vim Dockerfile
```

Add the following content to **Dockerfile**:

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

# Replace the content of index.html with "hello world".
RUN echo "hello world" > /usr/share/nginx/html/index.html

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

4. Package the image.

```
docker build -t hello .
```

-t is used to add a label to the image to name it. In this example, the image name is **hello**. The period **.** indicates that the packaging command is executed in the current directory.

5. Check whether the image has been created.

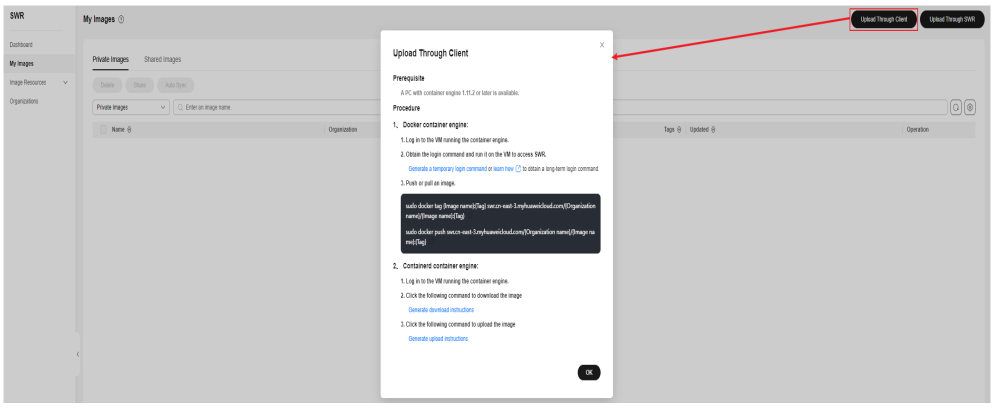
```
docker images
```

If information similar to the following is displayed, the image has been created:

```
REPOSITORY          TAG                  IMAGE ID            CREATED             SIZE
hello               latest               1ff61881be30        10 seconds ago      236MB
```

Pushing the Image to an Image Repository

1. Log in to the SWR console. In the navigation pane, choose **My Images**. On the page displayed, click **Upload Through Client**. In the dialog box 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.



Upload Through Client

Prerequisite

A PC with container engine 1.11.2 or later is available.

Procedure

1、 Docker container engine:

1. Log in to the VM running the container engine.

2. Obtain the login command and run it on the VM to access SWR.

Generate a temporary login command or learn how to obtain a long-term login command.

3. Push or pull an image.

```
sudo docker tag {Image name}:{Tag} swr.cn-east-3.myhuaweicloud.com/{Organization name}/{Image name}:{Tag}

sudo docker push swr.cn-east-3.myhuaweicloud.com/{Organization name}/{Image name}:{Tag}
```

2、 Containerd container engine:

1. Log in to the VM running the container engine.

2. Click the following command to download the image

Generate download instructions

3. Click the following command to upload the image

Generate upload instructions

2. Before pushing an image, specify a complete name for the image.

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

The command details are as follows:

– **swr.cn-east-3.myhuaweicloud.com** is the repository address, which varies with the region.

– **container** is the organization name. An organization is typically created in SWR. If no organizations are available, an organization will be automatically created when an image is pushed to SWR for the first time. Each organization name is unique in a single region.

– **v1** is the tag allocated to the **hello** image.

3. Push the image to SWR.

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

4. Pull the image.

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

3.2 Kubernetes

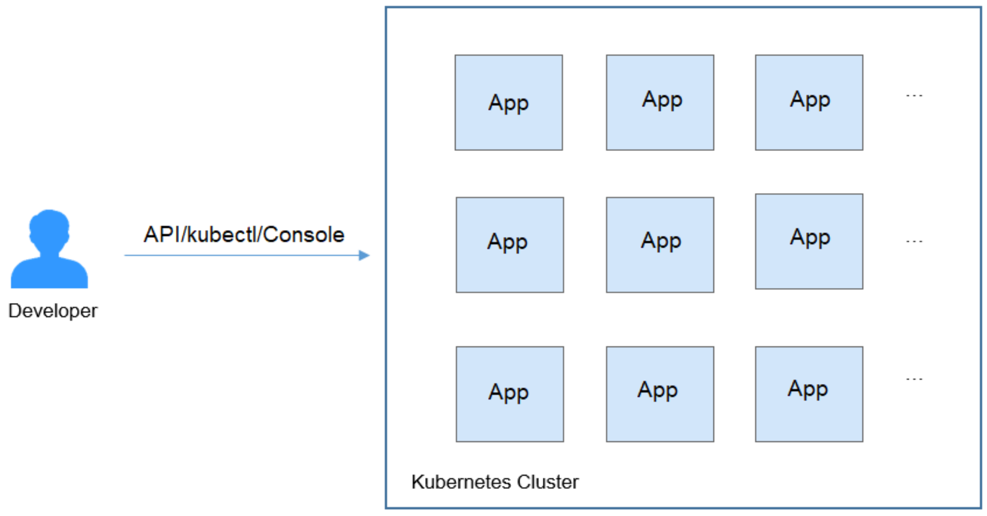
What Is Kubernetes?

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 you use Kubernetes, it is like running a giant number of servers all on the same physical machine, and then running your applications on that single massive platform. Kubernetes enables you to deploy applications always using the same method, regardless of how many servers there are in a cluster.

Figure 3-4 Running applications in a Kubernetes cluster



Kubernetes Cluster Architecture

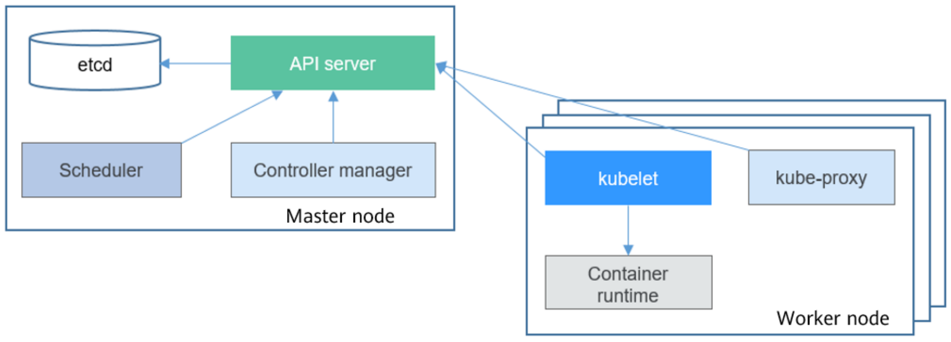
A Kubernetes cluster consists of master nodes and worker nodes. Applications are deployed on worker nodes, and you can specify the nodes for deployment.

NOTE

For CCE clusters, master nodes are managed by CCE. You only need to create worker nodes.

The following figure shows the architecture of a Kubernetes cluster.

Figure 3-5 Kubernetes cluster architecture



Master node

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

- API server: a transit station for components to communicate with each other. It receives external requests and writes data to etcd.

- Controller manager: carries out cluster-level functions, such as component replication, worker node tracing, and node fault troubleshooting.
- Scheduler: schedules pods to nodes based on various conditions (such as available resources and node affinity).
- etcd: provides distributed data storage for cluster configurations.

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

Worker node

A worker node is a compute node for containerized applications in a cluster. A worker node consists of the following components:

- kubelet: communicates with the container runtime, interacts with the API server, and manages pods on the node.
- kube-proxy: serves as an access proxy between application components.
- Container runtime: functions as an engine such as Docker for downloading images and running containers.

Kubernetes Scalability

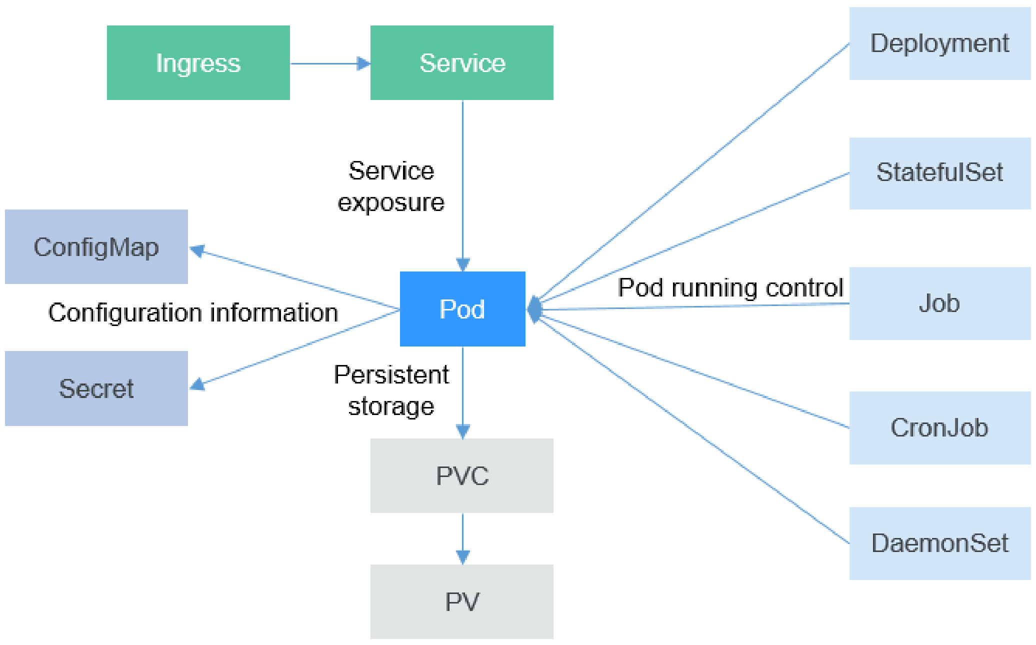
Kubernetes makes the Container Runtime Interface (CRI), Container Network Interface (CNI), and Container Storage Interface (CSI) open-sourced. These interfaces maximize Kubernetes scalability and allow Kubernetes to focus on container scheduling.

- CRI: provides computing resources for a container runtime. It makes differences between container engines irrelevant and interacts with each container engine through a unified interface.
- CNI: enables Kubernetes to support different networking implementations. For example, the custom CNI add-on of CCE allows your Kubernetes clusters to run in VPCs.
- CSI: enables Kubernetes to support various classes of storage. For example, CCE can be interconnected with block storage (EVS), file storage (SFS), and object storage (OBS) services.

Basic Objects in Kubernetes

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

Figure 3-6 Basic objects in Kubernetes



- Pod

  Pods are the smallest deployable units of compute that you can create and manage in Kubernetes. A pod is a group of one or more containers. Pods have shared storage, unique IP addresses, and specifications for how to run containers.
- Deployment

  A Deployment is a service-oriented encapsulation of pods. It can contain one or more pods. These pods have 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 a Deployment, a StatefulSet manages a group of pods that are based on an identical container specification. Unlike a Deployment, a StatefulSet maintains a fixed ID for each of their pods. These pods are created based on the same declaration, but they are not interchangeable. Each pod has a permanent ID regardless of how it was scheduled.
- Job

  A job is used to control batch tasks. Jobs are different from long-term servo tasks (such as Deployments). Jobs are started and terminated at specific times, but long-term servo tasks continue to run until they are specifically terminated. The pods managed by a job will be automatically removed after successfully completing tasks based on user configurations.
- CronJob

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

  A DaemonSet runs only one pod on each node in a cluster. 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 balances load for these pods.

- Ingress

  Services forward requests at Layer 4 using TCP and UDP. Ingresses can forward requests at Layer 7 using HTTP and HTTPS and make forwarding more targeted by domain names and paths.

- ConfigMap

  A ConfigMap stores configurations in key-value pairs required by applications. ConfigMaps allow you to decouple environment-specific configurations from container images, so that different environments can have their own unique configurations.

- Secret

  A secret lets you store and manage sensitive information, such as 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)

  A PVC in Kubernetes is a request for persistent storage. PVCs free you from creating and releasing underlying storage resources. With PVCs, you only need to specify the storage type and capacity.

Setting Up a Kubernetes Cluster

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

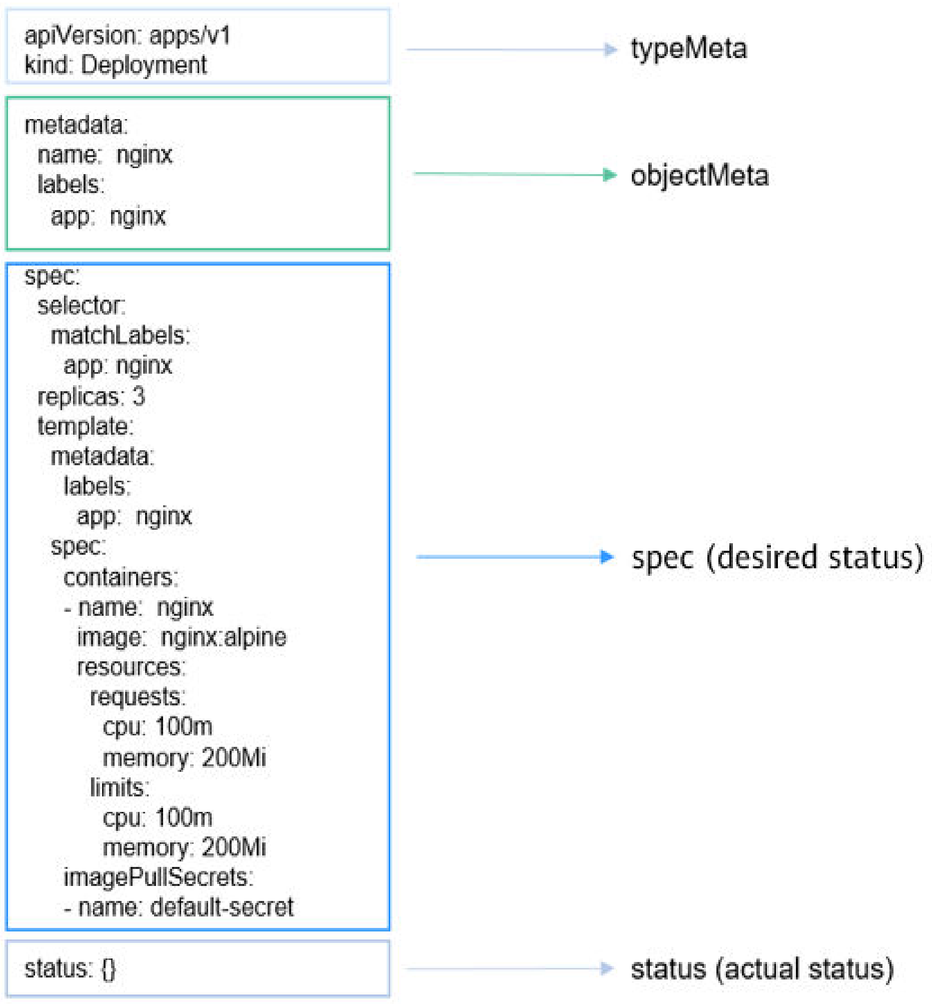
If you do not want to set up a Kubernetes cluster by yourself, you can purchase one on the CCE console. The following uses clusters purchased on the CCE console as examples.

Kubernetes Objects

Resources in Kubernetes can be described in YAML or JSON format. An object consists of the following parts:

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

Figure 3-7 YAML file



```
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
status: {}
```

Running Applications on Kubernetes

Delete **status** from the content in **Figure 3-7** 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 access 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 obtain 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
```

So far, we have walked you through containers and Docker, Kubernetes cluster setup, and basic Kubernetes concepts, and provided you an example of how to use kubectl. The following sections will go deeper into Kubernetes objects and how they are used and related.

3.3 Using kubectl to Operate a Cluster

kubectl

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

For details about how to install kubectl, see **Accessing a Cluster Using kubectl**. After using kubectl to access a cluster, you can run the **kubectl cluster-info** command to view the cluster information. The following shows an example:

```
# 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.

```
# kubectl get nodes
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
```

For more kubectl commands, see **kubectl Quick Reference**.

Basic Commands

get

The **get** command obtains details about one or more resources in a cluster.

This command obtains details about all resources, including cluster nodes, pods, Deployments, and Services.

NOTICE

Multiple namespaces can be created in a cluster. If no namespace is specified in the command, **--namespace=default** is used by default, which means resources in the **default** namespace are obtained.

Examples:

To obtain details of all pods:

```
kubectl get pod -o wide
```

To obtain all pods running in all namespaces:

```
kubectl get pod --all-namespaces
```

To obtain the labels of all pods running in all namespaces:

```
kubectl get pod --show-labels
```

To obtain all namespaces of the node:

```
kubectl get namespace
```

NOTE

Similarly, you can run the **kubectl get svc**, **kubectl get nodes**, and **kubectl get deploy** commands to obtain information about other resources.

To obtain details of pods in the YAML format:

```
kubectl get pod <podname> -o yaml
```

To obtain details of pods in the JSON format:

```
kubectl get pod <podname> -o json
kubectl get pod rc-nginx-2-btv4j -o=custom-columns=LABELS:.metadata.labels.app
```

NOTE

LABELS indicates a comma-separated list of user-defined column titles. **metadata.labels.app** indicates the data to be listed in either YAML or JSON format.

create

The **create** command creates cluster resources based on a file or input.

If you have a YAML or JSON file that defines the desired resource, you can use the following command to create the resource specified in the file:

```
kubectl create -f <filename>
```

expose

The **expose** command creates a Service for a resource such as a pod or Deployment.

```
kubectl expose deployment <deployname> --port=81 --type=NodePort --target-port=80 --node-port=31000
--name=<service-name>
```

NOTE

The **expose** command creates a NodePort Service for the Deployment. **--port** indicates the Service port (used for access inside the cluster), **--type** indicates the Service type, **--target-port** indicates the port of the backend pod associated with the Service, and **--node-port** indicates the node port (used for access outside the cluster). **--node-port** is optional. If it is not specified, the cluster randomly allocates a port from **30000** to **32767**.

run

The **run** command creates a single pod or Deployment, which is suitable for test environments.

Example:

```
kubectl run <deployname> --image=nginx:latest
```

To specify the command that is automatically executed when you create a pod or Deployment:

```
kubectl run <deployname> --image=busybox --command -- ping example.com
```

set

The **set** command configures a function for an object.

Example:

To update the container image of a Deployment to version 1.0 in rolling mode:

```
kubectl set image deployment/<deployname> <containername>=<containername>:1.0
```

edit

The **edit** command provides a way to update a resource.

Example:

To update a pod:

```
kubectl edit pod po-nginx-btv4j
```

The example command yields the same effect as the following commands:

```
kubectl get pod po-nginx-btv4j -o yaml >> /tmp/nginx-tmp.yaml
vim /tmp/nginx-tmp.yaml
# do some changes here
kubectl replace -f /tmp/nginx-tmp.yaml
```

explain

The **explain** command gets documents or other references.

Example:

To get documentation of pods:

```
kubectl explain pod
```

delete

The **delete** command deletes resources by resource name or label.

Example:

To delete a pod immediately:

```
kubectl delete pod <podname> --now
kubectl delete -f nginx.yaml
kubectl delete deployment <deployname>
```

Deployment Commands

rollout

The **rollout** command manages the rollout of a resource.

Examples:

To check the rollout status of a specific resource:

```
kubectl rollout status deployment/<deployname>
```

To view the rollout history of a specific resource:

```
kubectl rollout history deployment/<deployname>
```

To roll back a specific resource(by default, a resource is rolled back to the previous version):

```
kubectl rollout undo deployment/test-nginx
```

scale

The **scale** command increases or reduces the number of replicas when the load increases or decreases.

```
kubectl scale deployment <deployname> --replicas=<newnumber>
```

autoscale

The **autoscale** command automatically adjusts the number of replicas based on the CPU utilization of a workload. The **autoscale** command allows you to define a range of replicas for a workload (such as a Deployment, ReplicaSet, or StatefulSet). The pods will be automatically added or removed within this range based on the average CPU utilization of all pods. If the target utilization is not specified or is set to a negative value, the default auto scaling policy will be applied.

```
kubectl autoscale deployment <deployname> --min=<minnumber> --max=<maxnumber> --cpu-
percent=<cpu>
```

Cluster Management Commands

cordon, **drain**, and **uncordon**

If you need to upgrade a node or if a node becomes unavailable due to a breakdown, you can use these commands to reschedule the pods running on that node to other nodes. The procedure is as follows:

Step 1 Run the **cordon** command to mark a node as unschedulable. This means that new pods will not be scheduled to that node.

```
kubectl cordon <nodename>
```

In CCE, *<nodename>* specifies the private network IP address of a node by default.

Step 2 Run the **drain** command to evict pods on the node and smoothly migrate these pods to other nodes:

```
kubectl drain <nodename> --ignore-daemonsets --delete-emptydir-data
```

--ignore-daemonsets means DaemonSet pods will be ignored. **--delete-emptydir-data** ensures that if there are pods using emptyDir, the node will continue to be drained, and any local data associated with the node will be deleted.

Step 3 Perform maintenance operations on the node, for example, resetting the node.

Step 4 Run the **uncordon** command to mark the node as schedulable.

```
kubectl uncordon <nodename>
```

----End

cluster-info

To display the add-ons running in the cluster:

```
kubectl cluster-info
```

To view details:

```
kubectl cluster-info dump
```

top*

The **top** command shows the usages of resources like CPU, memory, and storage in a cluster. Ensure that the Kubernetes Metrics Server is running normally, or this command may not work.

taint*

The **taint** command updates the taints on one or more nodes.

certificate*

The **certificate** command modifies certificate resources.

Fault Diagnosis and Debugging Commands

describe

The **describe** command is similar to the **get** command. The main distinction is that the **get** command provides details about a resource, while the **describe** command only provides status information about a resource in a cluster. In addition, the **describe** command does not support the **-o** flag. For resources of the same type, the **describe** command provides the same output format and content.

```
kubectl describe pod <podname>
```

NOTE

If you need details about a resource, you can use the **get** command for more information. If you want to check the status of a resource, such as a pod that is not in the running state, you can use the **describe** command to obtain more status information.

logs

The **logs** command prints the standard output of programs running inside a container during pod execution. To display logs in the **tail -f** mode, run this command with the **-f** flag.

```
kubectl logs -f <podname>
```

exec

The **kubectl exec** command functions similarly to the Docker **exec** usage. When multiple containers run in a pod, you can use the **-c** flag to specify the desired container.

```
kubectl exec -it <podname> -- bash
kubectl exec -it <podname> -c <containername> -- bash
```

port-forward*

The **port-forward** command forwards requests from one or more local ports to a pod port.

Example:

To listen to local port 5000 and forward requests to port 6000 used by a pod created in **<my-deployment>**:

```
kubectl port-forward deploy/my-deployment 5000:6000
```

cp

To copy files or directories and paste them to a container:

```
kubectl cp /tmp/foo <podname>:/tmp/bar -c <containername>
```

The local files in **/tmp/foo** are copied and pasted to the **/tmp/bar** directory of a specific container in a pod.

auth*

The **auth** command checks authorization.

attach*

The **attach** command is similar to the **logs -f** command. To exit, run the **ctrl-c** command. If a pod contains multiple containers, to view the output of a specific container, use **-c** *<containername>* following *<podname>* to specify a container.

```
kubectl attach <podname> -c <containername>
```

Advanced Commands

replace

The **replace** command updates or replaces an existing resource. If you need to modify certain attributes of a resource, you can directly edit the YAML file and use the **replace** command to make changes such as adjusting the number of replicas, adding or modifying labels, changing the image version, or modifying the port.

```
kubectl replace -f <filename>
```

NOTICE

Resource names cannot be updated.

apply*

The **apply** command offers stricter control over resource updates than the **patch** and **edit** commands. It allows you to maintain resource configurations in source control. When there is an update, the configuration file is pushed to the server, and the **kubectl apply** command applies the latest configuration to the resource. Kubernetes compares the current configuration file with the applied configuration before applying the update, and then updates only the changed parts. The **apply** command works similarly to the **replace** command, but it does not delete the original resources or recreate new ones. Instead, it updates the existing resources. Additionally, **kubectl apply** adds a comment to the resource, marking the current apply operation, similar to a Git operation.

```
kubectl apply -f <filename>
```

patch

If you want to modify attributes of a running container but you do not want to delete the container first or use the **replace** command, a **patch** command is the rescue. This command changes settings for a running pod. For example, to change a pod label from **app=nginx1** to **app=nginx2** while the pod is running:

```
kubectl patch pod <podname> -p '{"metadata":{"labels":{"app":"nginx2"}}}'
```

convert*

The **convert** command converts configuration files between different API versions.

Configuration Commands

label

The **label** command updates labels on a resource.

```
kubectl label pods my-pod new-label=newlabel
```

annotate

The **annotate** command updates annotations on a resource.

```
kubectl annotate pods my-pod icon-url=http://*****
```

completion

The **completion** command provides autocompletion for kubectl.

Other Commands

api-versions

The **api-versions** command prints the supported API versions.

```
kubectl api-versions
```

api-resources

The **api-resources** command prints the supported API resources.

```
kubectl api-resources
```

config*

The **config** command modifies the kubeconfig file that is used to access APIs. For example, use this command to configure authentication information in API calls.

help

The **help** command gets all command references.

version

The **version** command prints the client and server version information.

```
kubectl version
```

4 Pods, Liveness Probes, Labels, and Namespaces

4.1 Pods

Video Tutorial

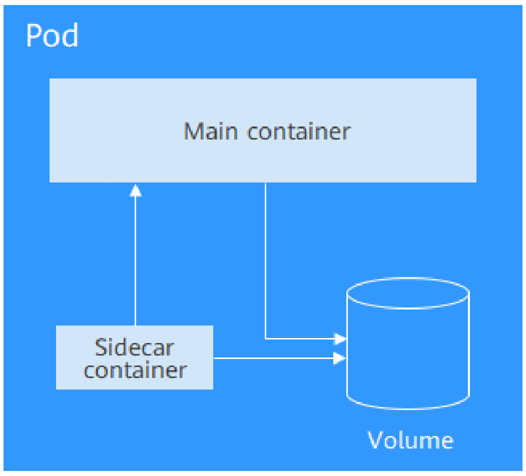
Overview of Pods

Pods are the smallest unit that you can create or deploy in Kubernetes. Each pod comprises one or more containers, shared storage (volumes), a unique IP address, and container runtime policies.

Pods can be used in either of the following ways:

- A pod runs a single container. This is the most common scenario in Kubernetes. In this case, a pod can be thought of as a container, although Kubernetes manages the pod rather than the container itself.
- A pod runs multiple tightly coupled containers that need to share resources. In this case, the pod includes a main container and several sidecar containers, as shown in **Figure 4-1**. For example, the main container might be a web server providing file services from a fixed directory, while sidecar containers periodically download files to that directory.

Figure 4-1 A pod running multiple containers



In Kubernetes, you rarely create pods directly. Instead, controllers like Deployments and jobs create and manage them. These controllers typically use pod templates to create and manage pods, providing features like replica management, rolling upgrades, and self-healing.

Creating a Pod

Kubernetes resources can be described using YAML or JSON files. The following is an example YAML file that describes a pod named **nginx**. This pod contains a container named **container-0** that uses the **nginx:alpine** image. The container requests 100 mCPU and 200 MiB of memory.

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

The above example shows that a YAML file includes:

- **metadata**: information such as name, label, and namespace
- **spec**: the pod specification, including the container image and volumes used

When checking a Kubernetes resource, you will also find the **status** field. It shows the current status of the resource. This field is automatically managed by Kubernetes and does not need to be set during resource creation. This example covers the minimum required parameters. Others will be described later.

In the following command, **-f** indicates that the pod will be created from a file, and **nginx.yaml** is the name of the file.

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

After creating the pod, check the pod status.

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

The command output shows that the **nginx** pod is in the **Running** state. The **READY** state of **1/1** indicates that the pod has one container and that the container is in the **Ready** state.

You can run **kubectl get** to retrieve information about a pod with different output formats. Use **-o yaml** to obtain the information in YAML format or **-o json** to obtain in JSON format.

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

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

```
$ kubectl describe pod nginx
```

Before deleting a pod, Kubernetes terminates all of the containers within it by sending a SIGTERM signal to the main process of each container. It then waits for a grace period (30s by default) for the containers to stop gracefully. If a container does not stop within this period, Kubernetes will send a SIGKILL signal to forcefully terminate it.

There are many ways to delete a pod. For example, you can delete a pod by name using the following command:

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

You can delete multiple pods at once:

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

You can delete all pods at the same time:

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

Delete pods by label (see **Labels**):

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

Environment Variables

You can use environment variables to configure the runtime environment of a container.

They add flexibility to application settings and allow you to customize settings when you create a container. These settings take effect when the container runs, eliminating the need to rebuild the container image.

The following shows an example, in which you only need to configure the environment variable **spec.containers.env**:

```
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:                          # The environment variable
      - name: env_key
        value: env_value
    imagePullSecrets:
    - name: default-secret
```

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 be defined using ConfigMaps or secrets. For details, see **Referencing a ConfigMap as an Environment Variable** or **Referencing a Secret as an Environment Variable**.

Container Startup Commands

Starting a container involves initiating its main process. You need to make some preparations before starting a main process. For example, before running a MySQL server, you may need to configure or initialize the environment. These preparatory steps can be handled by defining the **ENTRYPOINT** or **CMD** in a Dockerfile during image creation. For example, configuring **ENTRYPOINT ["top", "-b"]** in a Dockerfile ensures that CCE automatically performs the necessary preparations during container startup.

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

In practice, you can define the command and its arguments for a container in a pod by setting the **containers.command** field. This field is a list, where the first element is the command, and subsequent elements are 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:                     # The startup command
    - top
  - "-b"
 imagePullSecrets:
  - name: default-secret
```

Container Lifecycle

Kubernetes provides **lifecycle hooks** that allow containers to run custom operations at specific points in their lifecycle. For example, you can create a hook to perform an operation before a container is stopped. The available lifecycle hooks are as follows:

- **postStart**: triggered immediately after a container starts
- **preStop**: triggered immediately before a container stops

To use these hooks, simply configure the **lifecycle.postStart** or **lifecycle.preStop** parameter for a pod. The following shows an 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
    lifecycle:
      postStart:                 # Post-start processing
        exec:
          command:
          - "/postStart.sh"
      preStop:                   # Pre-stop processing
        exec:
          command:
          - "/preStop.sh"
  imagePullSecrets:
   - name: default-secret
```

4.2 Liveness Probes

Overview of Liveness Probes

Kubernetes enables self-healing for applications. If an application container crashes, it is automatically restarted. However, this mechanism does not address deadlocks. For example, if a Java program has a memory leak, it may become unresponsive while the JVM process continues running. To handle such scenarios, Kubernetes uses liveness probes. These probes check whether containers are responding normally and determine whether they need to be restarted. This is an effective health check strategy.

A liveness probe should be defined for each pod to help Kubernetes keep track of pod statuses.

Kubernetes supports the following detection methods:

- HTTP GET: The kubelet sends an HTTP GET request to the container. If the application returns a 2xx or 3xx status code, the container is considered healthy.
- TCP Socket: The kubelet attempts to establish a TCP connection to the target container on a specified port. If the connection is successful, the container is considered healthy. Otherwise, it is considered unhealthy.
- Exec: The kubelet executes a command inside the container. If the command exits with a status code of **0**, the container is considered healthy. If it exits with a non-zero status code, the container is considered unhealthy.

In addition to liveness probes, readiness probes also check pod statuses. For details, see **Readiness Probes**.

HTTP GET

HTTP GET is the most common detection method. In this mode, the kubelet sends an HTTP GET request to the target container. If the application returns a 2xx or 3xx status code, the container is considered healthy. The following shows an example:

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

Create the pod.

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

The kubelet sends an HTTP GET request to port 80 on the container. If the request fails, Kubernetes will restart the container.

View details of the pod.

```
$ 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 above information shows that the pod is **Running** and its **Restart Count** is **0**, indicating there have not been any container restarts. If **Restart Count** is not **0**, the container has been restarted.

TCP Socket

The kubelet attempts to establish a TCP connection to the target container on a specified port. If the connection is successful, the container is considered healthy. Otherwise, it is considered unhealthy. The following shows an example:

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

Exec

The kubelet executes a command inside the target container. If the command exits with a status code of **0**, the container is considered healthy. If it exits with a non-zero status code, the container is considered unhealthy. The following shows an example:

```
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:           # A liveness probe
      exec:                  # Exec definition
        command:
        - cat
        - /tmp/healthy
  imagePullSecrets:
  - name: default-secret
```

According to the Exec definition, the kubelet runs **cat /tmp/healthy** inside the container. If the command exits with a status code of **0**, the container is considered healthy. For the first 30 seconds, the **/tmp/healthy** file exists, causing **cat /tmp/healthy** to return a success code. After 30 seconds, the file is deleted. The kubelet then considers the pod unhealthy and restarts it.

Advanced Settings of a Liveness Probe

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

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

Liveness probe parameters are as follows:

- **delay=0s**: The health check starts immediately after the container starts.
- **timeout=1s**: The container must respond within one second. Otherwise, it is recorded as unhealthy.
- **period=10s**: The probe checks the container every 10 seconds.
- **#success=1**: The container is considered healthy if it succeeds once.
- **#failure=3**: The container is restarted after being recorded as unhealthy for three consecutive failures.

This liveness probe starts immediately after the container starts. If the container does not respond within one second, it is recorded as unhealthy. The probe runs every 10 seconds. If the container is recorded as unhealthy for three consecutive times, it is restarted.

These are the default settings when the probe is created. You can customize them as needed.

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10    # The health check starts 10 seconds after the container starts.
      timeoutSeconds: 2          # The container must respond within 2 seconds. Otherwise, it is considered
unhealthy.
      periodSeconds: 30          # The probe checks the container every 30 seconds.
      successThreshold: 1        # The container is considered healthy if it succeeds once.
      failureThreshold: 3        # The container is considered unhealthy after three consecutive failures.
```

Typically, the **initialDelaySeconds** value must be greater than **0** because it takes time for the application to become ready after the container starts. If the probe is initiated before the application is ready, it may fail.

Additionally, the **failureThreshold** value can be greater than **1**. This allows the kubelet to retry the probe multiple times before considering the container unhealthy, rather than failing the probe immediately after the first failure.

Configuring a Liveness Probe

- **What to check**

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

Additionally, a probe should not check parts with external dependencies. For example, if a frontend web server cannot access a database, the web server should not be considered unhealthy due to the connection failure.

- **To be lightweight**

  A liveness probe must not consume too many resources or hold certain resources for too long, as this could lead to resource shortages and affect service performance. For example, the HTTP GET method is recommended for Java applications. Using the Exec method might cause the JVM startup process to consume excessive resources.

4.3 Labels

Video Tutorial

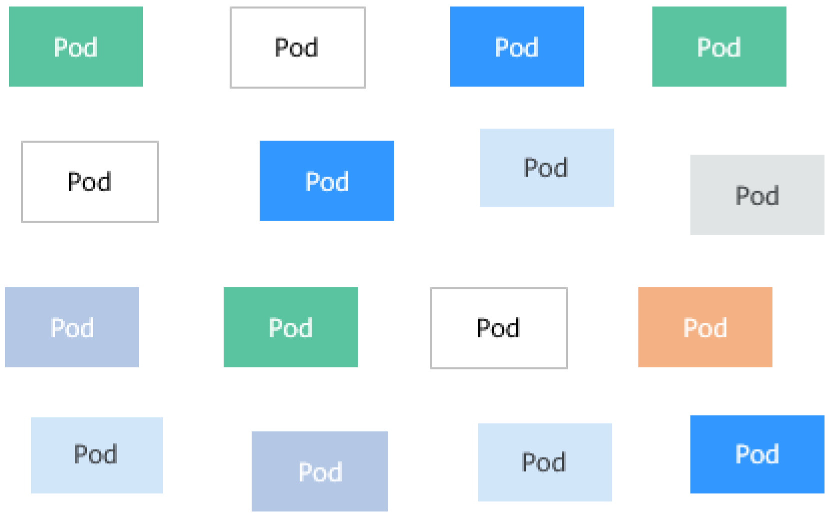
Overview of Labels

As resource volumes increase, efficient classification and management become essential. Kubernetes provides labels to help you manage almost all resources easily.

Labels are key-value pairs that can be set during or after resource creation and modified at any time as needed.

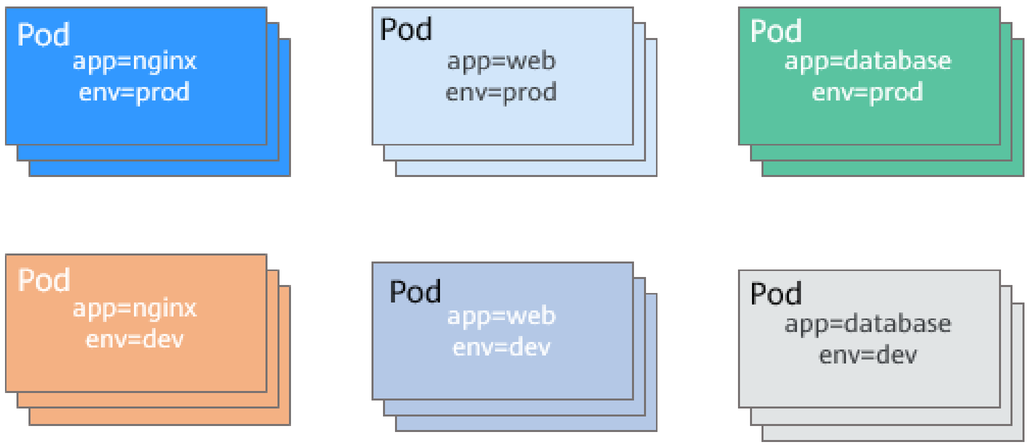
The following figures show how labels work. Managing multiple pods of various types without classification can be challenging.

Figure 4-2 Unclassified pods



Labeling these pods makes management much clearer.

Figure 4-3 Pods classified using labels



Adding a Label

The following example shows how to add labels **app=nginx** and **env=prod** during pod creation:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:                     # Add two labels 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 labeling a pod, you can view its labels by adding **--show-labels** to the pod query command.

```
$ 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 specific labels.

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

For existing pods, you can run **kubectl label** to add labels to them.

```
$ 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

To modify a label, add **--overwrite** to the command for managing labels.

```
$ 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
```

4.4 Namespaces

Video Tutorial

Overview of Namespaces

Labels are simple and efficient, but excessive use can lead to overlap and complicate queries. To address this, Kubernetes provides namespaces to divide systems with multiple components into distinct, non-overlapping groups. Namespaces can also separate resources among tenants, allowing multiple teams to share the same cluster.

Resources can share the same name as long as they are in different namespaces. However, global resources like worker nodes and PVs are not namespace-specific. We will cover this in more detail later.

Run the following command to obtain the namespaces in a cluster:

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

All operations are currently performed in the default namespace. If you run **kubectl get** without specifying a namespace, the default namespace will be used by default.

Run the following command to check 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 many pods in **kube-system**. For example, **coredns** is used for service discovery, **everest-csi** for accessing storage services, and **icagent** for interconnecting with a monitoring system.

These essential pods are placed in the **kube-system** namespace to isolate them from other pods. This ensures they are invisible to and unaffected by resources in other namespaces.

Creating a Namespace

Define a namespace.

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

Run **kubectl** to create it.

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

You can also run **kubectl create namespace** 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
```

The namespace **custom-namespace** now contains a pod named **nginx**.

Isolation of Namespaces

Namespaces are used to group resources for organizational purposes only. Objects running in different namespaces are not inherently isolated. For example, if pods in two namespaces know each other's IP addresses and the underlying network does not provide isolation between namespaces, the pods can still communicate with each other.

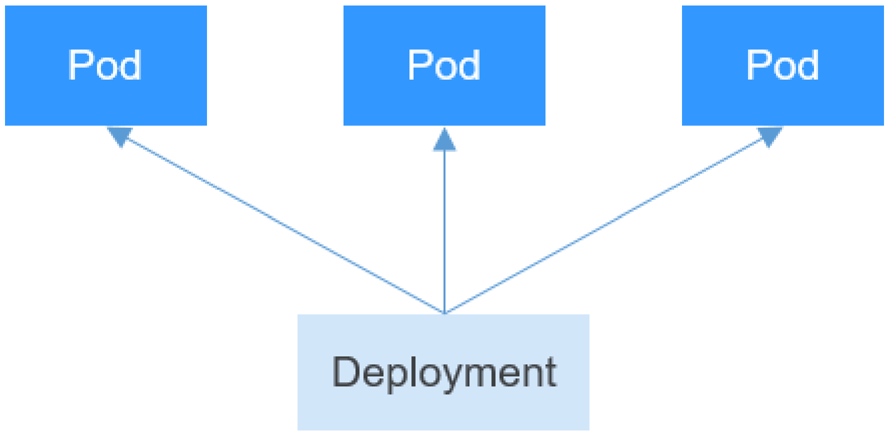
5 Pod Orchestration and Scheduling

5.1 Deployments

Overview of Deployments

A pod is the smallest unit that you create or deploy in Kubernetes. It is evicted when resources are tight and gone if its node fails. Kubernetes provides controllers to manage pods. These controllers create and manage pods, providing features like replica management, rolling upgrades, and self-healing. The most common controller is Deployment.

Figure 5-1 Relationship between a Deployment and pods



A Deployment runs one or more identical pods. Kubernetes load-balances traffic across them.

A Deployment handles rollout, rolling upgrades, scaling, and automatic replacement of failed pods. This enables zero-touch releases with minimal risk.

Creating a Deployment

In the following example, a Deployment named **nginx** launches two pods from **nginx:latest**, each reserving 100 mCPU and 200 MiB of memory.

```
apiVersion: apps/v1      # Use apps/v1 for Deployments. The setting is different from v1 for pods.
kind: Deployment         # The resource type is Deployment.
metadata:
  name: nginx            # The name of the Deployment
spec:
  replicas: 2            # The desired pod count. It indicates that there should be two running pods.
  selector:              # The label selector
    matchLabels:
      app: nginx
  template:              # The pod template. It defines the pods to be created.
    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 example, the Deployment is named **nginx**. **spec.replicas** specifies the number of pods that the Deployment maintains (two in this example). **spec.selector** is a label selector that identifies pods with the label **app=nginx**. **spec.template** defines the pod specifications, which are identical to those 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 its pods. The command output shows that the **READY** value is **2/2**. The first **2** indicates that two pods are running, while the second **2** indicates that the desired number of pods for this Deployment is two. The **AVAILABLE** value of **2** 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 a Deployment Control Pods?

Obtain pods, shown as 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
```

A Deployment ensures that a specified number of pods are running at any given time. If you delete a pod, the Deployment will immediately create a new one to maintain the desired state of two running pods. Similarly, if a pod fails, the Deployment will restart it to ensure the desired number of pods is always met.

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

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

In the example provided, the two pods are named **nginx-7f98958cdf-tdmqk** and **nginx-7f98958cdf-tesqr**. In the two names, **nginx** is the name of the Deployment, while **-7f98958cdf-tdmqk** and **-7f98958cdf-tesqr** are suffixes randomly generated by Kubernetes.

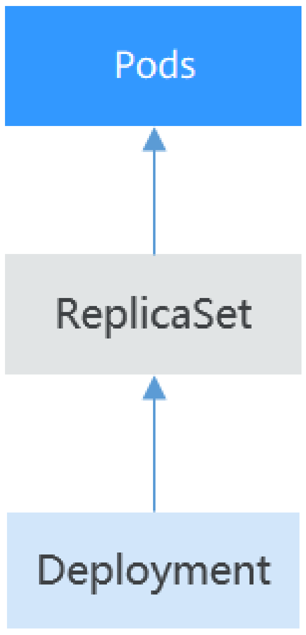
You may notice that the two suffixes share the same content **7f98958cdf**. This is because the Deployment does not directly control the pods. Instead, it manages them through a controller called a ReplicaSet. You can obtain the ReplicaSet using the following command (where **rs** is the abbreviation for ReplicaSet):

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

The ReplicaSet is named **nginx-7f98958cdf**, where the suffix **-7f98958cdf** is randomly generated by Kubernetes.

Figure 5-2 shows how a Deployment controls pods via a ReplicaSet.

Figure 5-2 How does a Deployment control pods



When you run **kubectl describe** to view the details of a Deployment, you will see information about the ReplicaSet it controls, for example, **NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)**. In **Events**, you will see that the number of pods in the ReplicaSet is scaled out to 2. In practice, you typically do not interact with ReplicaSets directly. However, understanding that a Deployment controls pods by managing a ReplicaSet helps you troubleshoot issues more effectively.

```
$ 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 real-world applications, upgrades are a common occurrence, and Deployments make application upgrades seamless and straightforward.

Available upgrade policies for Deployments:

- **RollingUpdate**: New pods are created gradually, and old pods are deleted one by one. This is the default policy.
- **Recreate**: All current pods are deleted first, and then new pods are created.

Deployments are upgraded in a declarative manner. You simply need to modify the YAML definition of the target Deployment. For example, you can run **kubectl edit** to change the image used by the sample Deployment to **nginx:alpine**. After making the modification, you can check the ReplicaSet and pods. The query result shows that a new ReplicaSet has been created, and the pods have been re-created to reflect the updated image.

```
$ 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
```

Each Deployment can control the proportion of pods to be recreated during an upgrade using the **maxSurge** and **maxUnavailable** parameters. This method is useful in a wide range of scenarios. The configuration is as follows:

```
spec:
  strategy:
    rollingUpdate:
      maxSurge: 0.25
      maxUnavailable: 0.25
    type: RollingUpdate
```

- **maxSurge** specifies the maximum number or percentage of pods that can exist above the desired number of pods (**spec.replicas**) during a rolling upgrade. This parameter determines the maximum number of new pods that can be created at a time to replace old pods. The default value is **25%**. During an upgrade, the percentage is converted into an absolute number and **rounded up**.

For example, if **spec.replicas** is set to **2**, a maximum of one pod (2 x 0.25 = 0.5, rounded up to 1) can be created at a time by default. Therefore, during an upgrade, up to 3 pods can exist (2 desired + 1 surge).

- **maxUnavailable** specifies the maximum number or percentage of pods that can be unavailable during a rolling upgrade. This also sets the limit for how many running pods can be below the expected number. The default value is **25%**. During an upgrade, the percentage is converted into an absolute number and **rounded down**.

  For example, if **spec.replicas** is set to **2**, no pods (2 x 0.25 = 0.5, rounded down to 0) can be unavailable. Therefore, during an upgrade, there will always be at least two pods running (2 desired - 0 unavailable). Each old pod is deleted only after a new one is created, ensuring that at least two pods are always running until all pods are updated.

Rollback

Rollback is the process of reverting an application to an earlier version if a fault occurs during an upgrade. Applications managed by Deployments can be easily rolled back to a previous version.

For example, if the image of an upgraded Deployment is faulty, you can run **kubectl rollout undo** 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 pods. After an upgrade, the previous ReplicaSet is retained. The Deployment is rolled back by using the previous ReplicaSet to recreate the pods. The maximum number of ReplicaSets stored in a Deployment can be controlled by the **revisionHistoryLimit** parameter. The default value is **10**.

5.2 StatefulSets

Overview of StatefulSets

All pods under a Deployment are identical except for their names and IP addresses. Deployments can create new pods using a pod template and delete any pod when not needed.

However, Deployments are not suitable for distributed scenarios where each pod requires its own status or independent storage, such as in distributed databases.

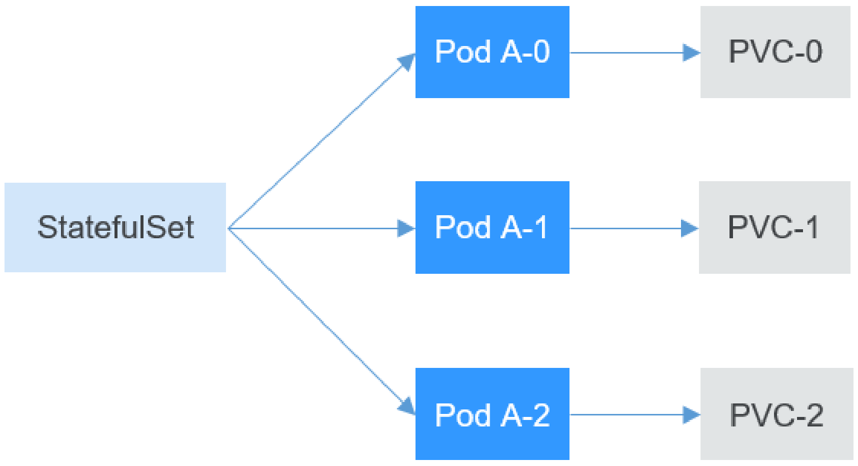
Distributed stateful applications often involve different roles and responsibilities. For example, databases may operate in active/standby mode, and pods may depend on each other. To deploy stateful applications in Kubernetes, pods must meet the following requirements:

- Each pod must have a unique, fixed identifier to be recognized by other pods.
- Each pod should be configured with separate storage resources to ensure data persistence. This allows the original data to be retained and retrieved even after a pod is deleted and recreated. Without dedicated storage, the pod's data will be lost upon deletion, and the new pod will initialize with a different state.

To address these requirements, Kubernetes provides StatefulSets:

1. StatefulSets provide a fixed name for each pod, followed by a sequential numeric suffix (for example, pod-0, pod-1, ..., pod-N). After a pod is rescheduled, its name and hostname remain unchanged.
2. StatefulSets use a headless Service to allocate a fixed domain name for each pod.
3. StatefulSets create PVCs with fixed identifiers. This ensures that pods can access the original persistent data after being rescheduled.

Figure 5-3 StatefulSet



Creating a Headless Service

A **headless Service** is required by a StatefulSet to access its pods.

Define a headless Service as follows:

- **spec.clusterIP**: must be set to **None** to indicate a headless Service.
- **spec.ports.port**: the port for communication between pods.
- **spec.ports.name**: the name of the port for communication between pods.

```
apiVersion: v1
kind: Service       # The object type is Service.
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # The name of the port for communication between pods
      port: 80        # The port for communication between pods
  selector:
    app: nginx        # Select the pods labeled with app:nginx.
  clusterIP: None     # Set this parameter to None to indicate a headless Service.
```

Create the headless Service.

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

After the Service is created, check 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 a StatefulSet is similar to that of other Kubernetes objects. The differences are as follows:

- **serviceName** specifies the headless Service used by the StatefulSet. This parameter is required.
- **volumeClaimTemplates** defines a template for requesting **PVCs**. In this example, a template named **data** is defined, which creates a PVC for each pod. The **storageClassName** field specifies the type of persistent storage. For details, see **PVs, PVCs, and Storage Classes**. The **volumeMounts** field specifies where the storage is mounted on the pods. If no persistent storage is required, you can delete **volumeClaimTemplates** and **volumeMounts**.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                             # The 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:                          # The storage to be mounted on the pods
          - name:  data
            mountPath:  /usr/share/nginx/html    # Mount storage to /usr/share/nginx/html.
      imagePullSecrets:
        - name: default-secret
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
      - ReadWriteMany
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-nas                  # The type of persistent storage
```

Create the StatefulSet.

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

After executing the command, verify the StatefulSet and its pods. Pod names will have sequential numeric suffixes starting from 0 and incrementing up 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
```

Delete the pod named **nginx-1** and recheck the pods. A new pod with the same name (**nginx-1**) will be created. The value **5s** under the **AGE** field indicates that the new pod was recently 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 pods and check their hostnames. They remain as **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
```

Verify the PVCs created by the StatefulSet. These PVCs are named in the format "PVC name-StatefulSet name-Number" 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, each of its pods is assigned a fixed name. The headless Service used by the StatefulSet provides a fixed domain name for each pod via DNS. This allows pods to communicate with each other using their domain names. These names remain unchanged even if a pod's IP address changes after the pod is recreated.

After a headless Service is created, it allocates a domain name in the following format to each pod:

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

For example, the domain names for the preceding 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 practice, **.<namespace>.svc.cluster.local** can be omitted when accessing pods.

To verify DNS resolution, create a pod using the **tutum/dnsutils** image. Access the container and run **nslookup** to resolve the pod's domain name. The pod's IP address should be correctly resolved. 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. For details, see **Kubernetes Networks**.

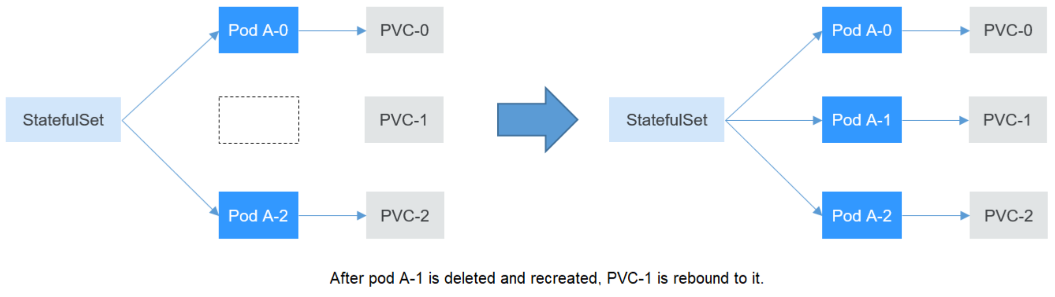
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # 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
```

Manually delete the two pods. After the StatefulSet recreates the pods, check their IP addresses. Then, run **nslookup** to resolve the domain names of the pods. You should still be able to resolve **nginx-0.nginx** and **nginx-1.nginx**. This ensures that the network identity of the StatefulSet remains unchanged.

Storage Status of a StatefulSet

As mentioned above, StatefulSets use PVCs for persistent storage. This ensures that the original data remains accessible even after pods are rescheduled. When pods are deleted, the PVCs are retained.

Figure 5-4 StatefulSet pod recreation process



The following uses an example to illustrate the process. Write data to the **/usr/share/nginx/html** directory of **nginx-1**. For example, modify the content of **index.html** to display **hello world** by running the following command:

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

After modifying the content, access **https://localhost**. The response should be **hello world**.

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

Delete the pod named **nginx-1** and recheck the pods. A new pod with the same name (**nginx-1**) will be created. The value **4s** under the **AGE** field indicates that the new pod was recently 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
```

Re-access the **index.html** of the newly created **nginx-1** pod. The content **hello world** is still returned. This confirms that the new pod continues to access the original storage data.

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

5.3 Jobs and CronJobs

Overview of Jobs and CronJobs

Jobs and CronJobs are Kubernetes resources designed to manage short-lived, one-off tasks that run to completion.

- A job is a resource object used to control batch tasks. Jobs start and terminate at specific times, unlike long-running services such as Deployments and StatefulSets, which run continuously unless terminated. Pods managed by a job are automatically removed after successfully completing their tasks, based on the specified settings.
- A CronJob runs a job periodically on a specified schedule. A CronJob object is similar to a line in a crontab file in Linux.

The run-to-completion feature of workloads makes them particularly suitable for one-off tasks, such as continuous integration (CI) pipelines.

Creating a Job

The following example demonstrates a job that calculates π to the 2000th digit and prints the result. The job is configured to run a total of 50 pods, with up to 5 pods running concurrently. If a pod fails, it can be retried up to 5 times.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  completions: 50            # The total number of pods that must complete successfully for the job to be
considered complete
  parallelism: 5             # The number of pods that can run concurrently. The default value is 1.
  backoffLimit: 5            # The maximum number of retries for a pod if it fails
  activeDeadlineSeconds: 100  # The number of seconds after which the job and all its pods will be
terminated
  template:                  # Define the pods for the job.
    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 into several types.

Table 5-1 Job types

Job Type	Description	Example
One-off job	The job runs a single pod to completion.	Database migration
Non-parallel job with a fixed completion count	The job runs one pod until the specified completion count is reached.	Processing a queue of tasks
Parallel job with a fixed completion count	The job runs multiple pods in parallel until the specified completion count is reached.	Processing a task queue concurrently
Parallel job	The job runs one or more pods in parallel until one pod completes successfully.	Processing a task queue concurrently

Creating a CronJob

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

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "0,15,30,45 * * * *"           # Schedule for the CronJob
  jobTemplate:                              # Define the job.
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: pi
            image: perl
            command:
            - perl
            - "-Mbignum=bpi"
            - "-wle"
            - print bpi(2000)
```

The format of the CronJob is as follows:

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

Take **0,15,30,45 * * * *** as an example. In this example, commas separate the minutes (0, 15, 30, 45). The first asterisk (*) indicates any hour, the second asterisk indicates any day, the third asterisk indicates any month, and the fourth asterisk indicates any day of the week.

If you want to run the job every 30 minutes on the first day of each month, set this parameter to **0,30 * 1 * ***. If you want to run the job at 03:00 a.m. every Sunday, set this parameter to **0 3 * * 0**.

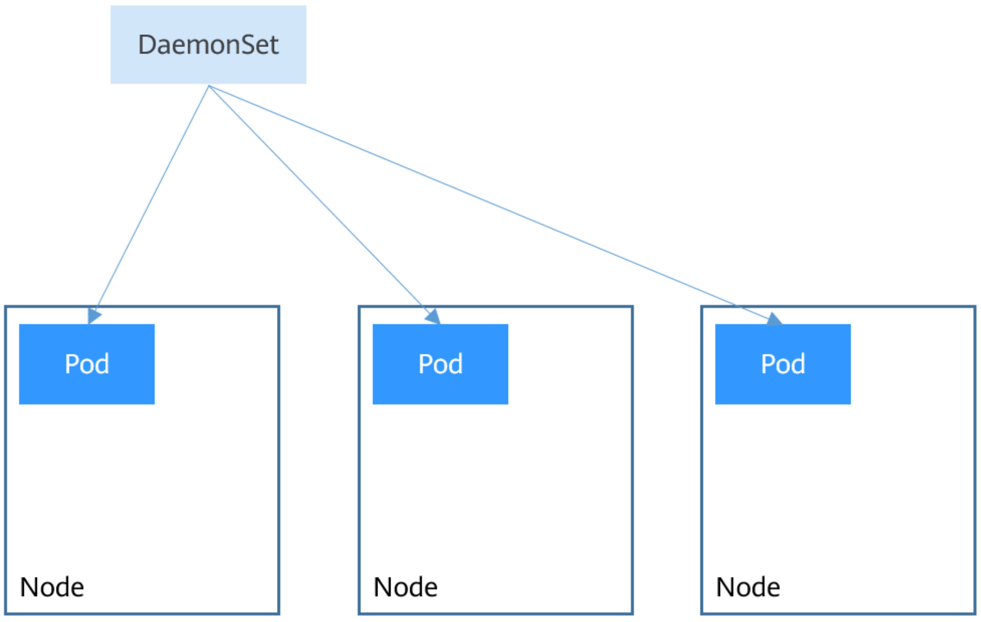
5.4 DaemonSets

Overview of 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. 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 5-5 DaemonSet



Creating a DaemonSet

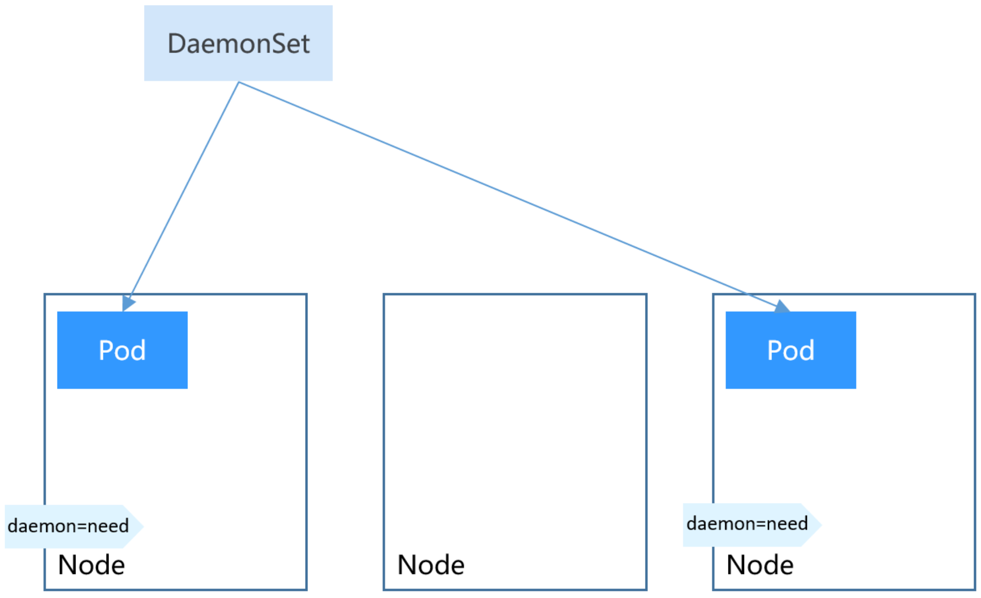
The following is an example of a DaemonSet:

```
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 only created on a node when the node has the
daemon=need label.
        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 DaemonSet pod.

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

Figure 5-6 Creating a DaemonSet pod on the nodes with a specified label



Create the 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 node labels:

```
$ 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 the node **192.168.0.212**, and then check the DaemonSet pods again. In this example, a DaemonSet pod was 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 the node **192.168.0.94**. You can find that a DaemonSet 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
```

Remove the **daemon=need** label of the node **192.168.0.94**. You can see that 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
```

5.5 Affinity and Anti-Affinity

A nodeSelector provides a simple way to assign pods to certain nodes, as mentioned in **DaemonSets**. Kubernetes also supports affinity and anti-affinity for more refined, flexible scheduling.

Kubernetes allows for affinity and anti-affinity for both nodes and pods, allowing you to define strict restrictions or preferences for your workloads. You can, for example, use affinity and anti-affinity rules to group frontend and backend pods together, place similar applications on designated nodes, or distribute applications across different nodes.

Node Affinity

Labels are the foundation of affinity rules in Kubernetes. In a CCE cluster, a node can have the following labels:

```
$ 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 to a node during its creation. The following are a few that are frequently used during scheduling.

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

Additionally, **Labels** describes the custom labels. A large Kubernetes cluster typically has various kinds of labels.

When you deploy pods, you can use a nodeSelector, as described in **DaemonSets**, 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:                 # Select nodes. A pod is deployed on a node that has the gpu=true label.
    gpu: true
...
```

You can also use node affinity rules, 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"
```

Node affinity may seem complex, but it is more expressive, which will be further described later.

In the file, **affinity** represents affinity rules, while **nodeAffinity** specifically defines affinity constraints for nodes. **requiredDuringSchedulingIgnoredDuringExecution** can be broken down into two parts.

- **requiredDuringScheduling** specifies that pods can only be scheduled onto the node when all the defined rules are met (**required**).
- **IgnoredDuringExecution** specifies that pods already running on the node do not need to meet the defined rules. If a label is removed from the node, the pods that require the node to contain that label will not be re-scheduled.

In addition, the value of **operator** is set to **In**. This means 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 the specified value (for strings).
- **Lt**: The label value is less than the specified value (for strings).

There is no node anti-affinity because operators **NotIn** and **DoesNotExist** provide the same function.

Now, verify that the node affinity rule works. Add the **gpu=true** label 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 a Deployment. In this example, all the Deployment pods run 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

requiredDuringSchedulingIgnoredDuringExecution is a hard selection rule. There is also a preferred selection rule **preferredDuringSchedulingIgnoredDuringExecution**, which is used to specify which nodes are preferred during scheduling.

To demonstrate the effect, add a node in a different AZ from other nodes to the cluster and check the AZ of the node. 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 **preferredDuringSchedulingIgnoredDuringExecution** to set the weight of nodes in **cn-east-3a** to **80** and nodes with the **gpu=true** label to **20**. In this way, pods are preferentially deployed onto the nodes 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:
```

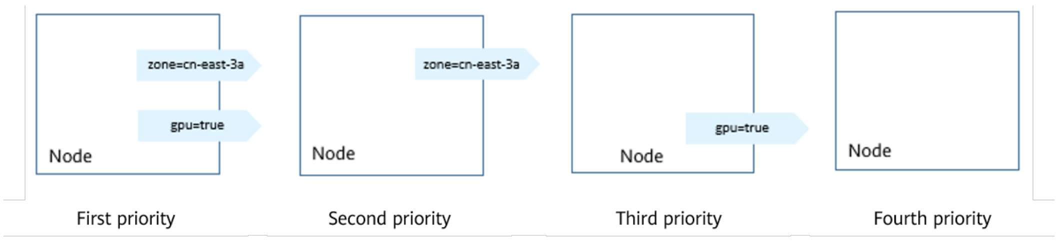
```
containers:
- image:  nginx:alpine
  name:  gpu
  resources:
    requests:
      cpu:  100m
      memory:  200Mi
    limits:
      cpu:  100m
      memory:  200Mi
imagePullSecrets:
- name: default-secret
affinity:
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 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 is created, you can see 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 this example, the node with both **cn-east-3a** and **gpu=true** labels has the highest priority, followed by the node (weight: 80) with only the **cn-east-3a** label, and then the node with only the **gpu=true** label. The node without any of these two labels have the lowest priority.

Figure 5-7 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 means that **preferredDuringSchedulingIgnoredDuringExecution** defines a preference rather than a hard rule.

Workload Affinity

Node affinity affects only the affinity between pods and nodes. Kubernetes also supports inter-pod affinity, which allows you to, for example, deploy the frontend and backend of an application on the same node to reduce access latency. There are also two types of inter-pod affinity rules: **requiredDuringSchedulingIgnoredDuringExecution** and **preferredDuringSchedulingIgnoredDuringExecution**.

NOTE

For workload affinity, the **topologyKey** field cannot be left blank when **requiredDuringSchedulingIgnoredDuringExecution** and **preferredDuringSchedulingIgnoredDuringExecution** are used.

Assume that the backend pod 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
```

Configure the following pod affinity rule to deploy the frontend pods of the application to the same node as its backend pod:

```
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
```

Create the frontend pods and check their nodes. You will find that the frontend pods have been deployed on the same node as the backend pod.

```
$ 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 scheduler first evaluates the **topologyKey** setting, which defines topology domains and determines the selection range for pod placement. Nodes with the specified key and identical values are considered to be in the same topology domain. The scheduler then applies other defined rules to finalize pod placement. Since all nodes in the previous example share the **kubernetes.io/hostname** label, they all fall within the same topology domain, meaning the impact of **topologyKey** cannot be clearly observed.

To see how **topologyKey** works, assume that there are two backend pods of the application and they run 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
```

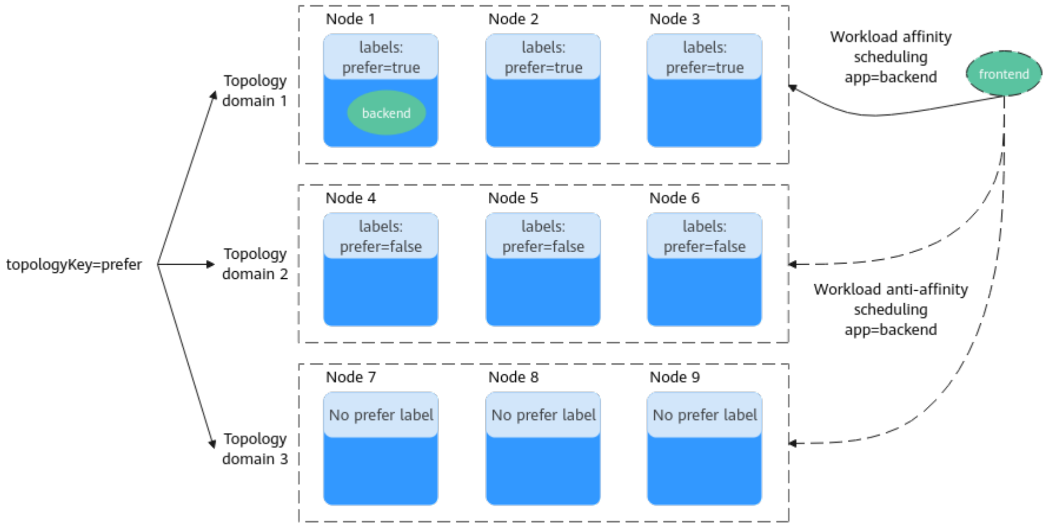
Add the **prefer=true** label to nodes **192.168.0.97** and **192.168.0.94**.

```
$ 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
```

If **topologyKey** in **podAffinity** is set to **prefer**, the node topology domains are divided as shown in **Figure 5-8**.

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

Figure 5-8 Topology domains



During scheduling, node topology domains are determined based on the **prefer** label. In this example, the **192.168.0.97** and **192.168.0.94** nodes belong to the same topology domain. If a pod labeled **app=backend** runs within this topology domain, the frontend is scheduled to the same domain, even if not all nodes within it contain a pod with the **app=backend** label. In this example, only the **192.168.0.97** node hosts a pod labeled **app=backend**, but the scheduling rules ensure that the frontend pods are still deployed within the domain, meaning they can be placed on either the **192.168.0.97** or **192.168.0.94** node.

```
$ 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   192.168.0.94
```

Workload Anti-Affinity

In some cases, instead of grouping pods onto the same node using affinity rules, it is preferable to distribute them across different nodes to prevent some performance issues.

NOTE

For workload anti-affinity, when **requiredDuringSchedulingIgnoredDuringExecution** is used, Kubernetes enforces a restriction through the default admission controller, **LimitPodHardAntiAffinityTopology**. It mandates that **topologyKey** can only be **kubernetes.io/hostname** by default. If a custom topology logic is required, the admission controller must be modified or disabled.

The following is an example of defining an anti-affinity rule. This rule divides node topology domains by the **kubernetes.io/hostname** label. If a pod with the **app=frontend** label already exists on a node in this topology domain, pods with the same label cannot be scheduled to other nodes in the topology domain.

```
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   # Topology domain of a node
            labelSelector:    # Pod label matching rule
              matchExpressions:
              - key: app
                operator: In
                values:
                - frontend
```

Create an anti-affinity rule and view the deployment result. In the example, node topology domains are divided by the **kubernetes.io/hostname** label. The label values of nodes with the **kubernetes.io/hostname** label are different, so there is only one node in a topology domain. If a topology domain contains only one node where a frontend pod already exists, pods with the same label will not be scheduled to that topology domain. In this example, there are only four nodes. Therefore, there is one pod which is in the **Pending** state and cannot be scheduled.

```
$ 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
```

6 Configuration Management

6.1 ConfigMaps

ConfigMaps are a type of resource that is used to store the configurations required by applications. It can store configuration data in key-value pairs or configuration files as files.

ConfigMaps allow you to decouple environment-specific configurations from container images, so that different environments can have their own unique 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 or as configuration files 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, the value of **property_1** is referenced as the value of **EXAMPLE_PROPERTY_1**, which 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 a ConfigMap.
          name: configmap-test
          key: property_1
  imagePullSecrets:
  - name: default-secret
```

Referencing a ConfigMap in a Volume

Referencing a ConfigMap in a volume is when you save the data as configuration files in the volume. Each piece of data is saved as a file. The key is the file name, and the key value is the file content.

In the following example, a volume named **vol-configmap** is created, the ConfigMap named **configmap-test** is referenced in the volume, and the volume is mounted to the **/tmp** directory of the container. After the pod is created, 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 a ConfigMap.
      name: configmap-test
```

6.2 Secrets

Secrets let you store and manage sensitive information, such as authentication details, certificates, and private keys. Storing sensitive information in a secret is safer and more flexible than putting it in a pod specification or a container image.

Similar to ConfigMaps, secrets store data in key-value pairs. The difference is that the secrets are encrypted, so they are 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 *<content-to-be-encoded>* | base64** command. For 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. The following shows an 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 when you save the data as configuration files in the volume. Each piece of data is saved as a file. The key is the file name, and the key value is the file content.

In the following example, a volume named **vol-secret** is created, a secret named **mysecret** is referenced in the volume, and the volume is mounted to the **/tmp** directory of the container. After the pod is created, 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 a secret.
      secretName: mysecret
```

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

7 Kubernetes Networks

7.1 Container Networks

Kubernetes is not responsible for network communications. It only provides the Container Networking Interface (CNI) for networking through CNI plugins. There are many open source CNI plugins, such as Flannel and Calico. CCE offers various network add-ons for clusters that use different network models, enabling seamless network communications within clusters.

Kubernetes requires that cluster networks meet the following requirements:

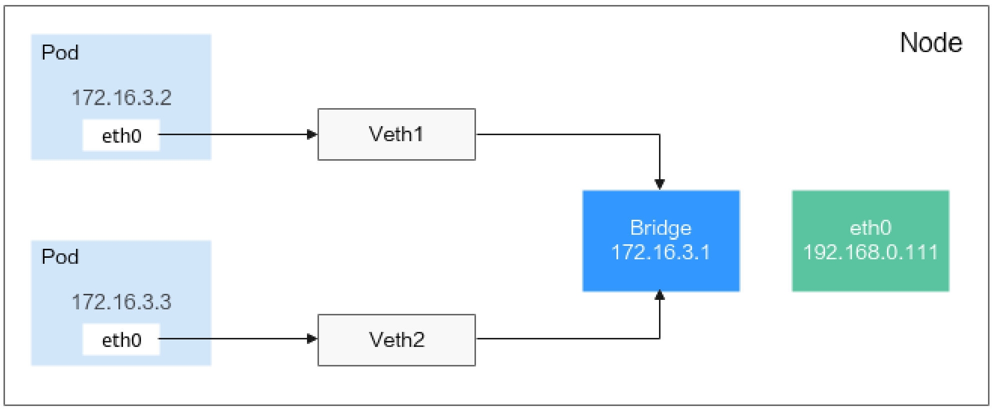
- Pods in a cluster are accessible to each other through a non-NAT network. The source IP addresses of the data packets received by a pod are the IP addresses of the pods that the data packets were sent from.
- Nodes can communicate with each other without NAT.

Pod Communication

Communications Between Pods on the Same Node

A pod communicates with external systems through virtual Ethernet (veth) pairs. A veth pair is like a network cable, with one end inside the pod and the other end outside the pod. The pods on the same node communicate with each other through a Linux bridge.

Figure 7-1 Communications between pods on the same node



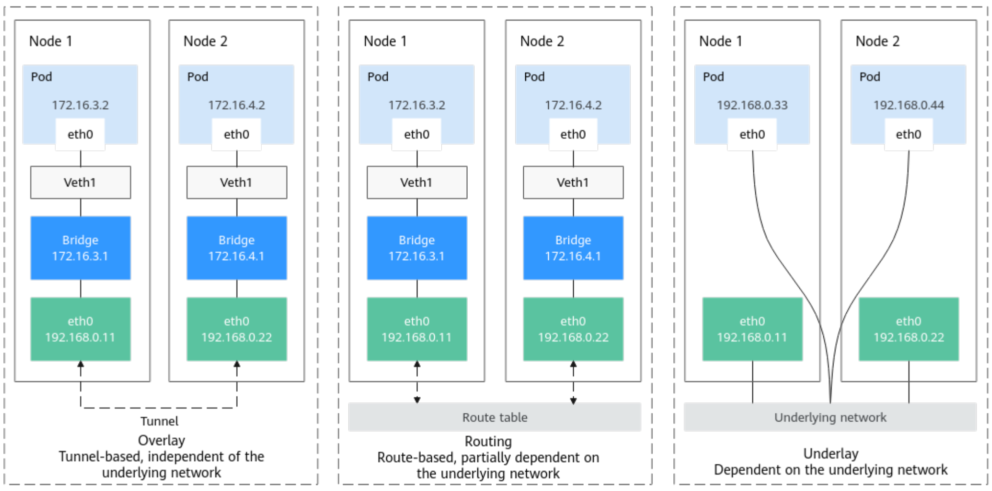
Pods on the same node connect to the bridge through veth devices and obtain IP addresses through the bridge. These IP addresses are from the same CIDR block as the bridge IP address. Additionally, the default routes of all pods on the node point to the bridge, which forwards all traffic from the IP addresses that are not on the local network. The pods on the node can communicate with each other.

Communications Between Pods on Different Nodes

Kubernetes requires the IP address of each pod in a cluster be unique. Each node in the cluster is allocated a subnet to ensure that the IP addresses of the pods are unique within the cluster. Pods running on different nodes communicate with each other through IP addresses. This process is implemented using cluster networking plugins. Pods can communicate with each other using three types of networks: overlay, routing, and underlay:

- An overlay network is set up on the node network using tunnel encapsulation. Such a network has its own IP addresses and IP switching and routing. VXLAN is a mainstream overlay network tunneling protocol.
- In a routing network, a VPC route table is used with the network for communications between pods and nodes. The performance surpasses that of the overlay tunnel encapsulation.
- In an underlay network, drivers expose network interfaces on nodes to pods for high-performance network communications. IP VLANs are commonly used on underlay networks.

Figure 7-2 Communications between pods on different nodes



Sections **Services** and **Ingresses** will describe how Kubernetes provides access solutions for users based on the container networking.

7.2 Services

Video Tutorial

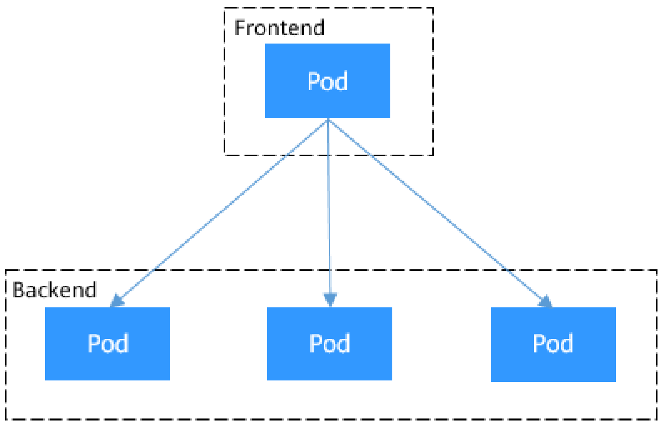
Direct Access to a Pod

After a pod is created, accessing it directly can result in certain problems:

- The pod can be deleted and recreated at any time by a controller such as a Deployment. If the pod is recreated, access to it may fail.
- An IP address cannot be assigned to a pod until the pod is started. Before the pod is started, its IP address is unknown.
- Applications usually run on multiple pods that use the same image. Accessing pods one by one is not efficient.

For example, Deployments are used to deploy the frontend and backend of an application. The frontend calls the backend for computing, as shown in **Figure 7-3**. Three pods are running in the backend, and they are independent and replaceable. When a backend pod is recreated, the new pod is assigned a new IP address, but the frontend pod is unaware of this change.

Figure 7-3 Inter-pod access

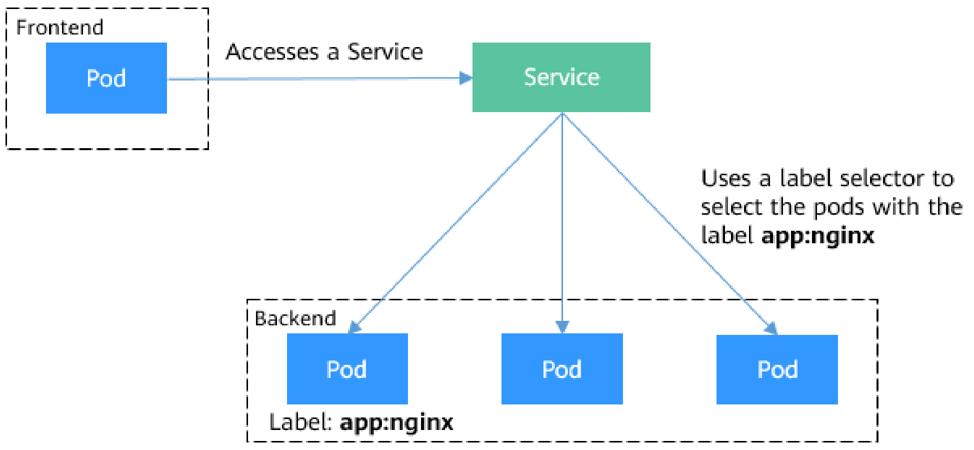


Using Services for Pod Access

Kubernetes Services are used to solve the preceding pod access problems. A Service has a fixed IP address. (When you create a CCE cluster, you need to specify a Service CIDR block, which is used to allocate IP addresses to Services.) A Service distributes requests across pods based on labels and balances the loads for these pods.

In the preceding example, a Service is created 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 7-4**.

Figure 7-4 Accessing pods through a Service



Creating Backend Pods

Create a Deployment with three replicas (three pods) with the 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, a Service named **nginx** is created, and a selector selects the pod with the label **app:nginx**. The pod uses port 80 while the Service access port is 8080.

The Service can be accessed through *<Service-name>:<Service-access-port>*. In this example, the access address is **nginx:8080**. 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 app: nginx
    app: nginx
  ports:
  - name: service0
    targetPort: 80   # Pod port
    port: 8080       # Service access port
    protocol: TCP    # Forwarding protocol. 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 this is a ClusterIP Service, which has a fixed cluster-scoped IP address unless the Service is deleted. You can use this IP address to access the Service within the cluster.

Create a pod and use the IP address (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 the Service Name to Access a Service

With DNS, you can access a Service through *<Service-name>:<port>*. This is the most common practice in Kubernetes. When you are creating a CCE cluster, you are required to install the CoreDNS add-on. 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 the add-on is installed, CoreDNS serves as a DNS server. After the Service is created, CoreDNS records the Service name and IP address. In this way, the pods can obtain the Service IP address by querying the Service name from CoreDNS.

In this example, **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 the same namespace, you can omit *<namespace>***.svc.cluster.local** and only use the Service name.

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

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

Create a pod and enter the container. Then run the **nslookup** command to query the domain name resolution result. The command output shows that the domain name of the Service is **nginx.default.svc.cluster.local**, and the resolved IP address is 10.247.124.252. Run the **curl nginx:8080** command to access the Service. If the page content is returned, the Service 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
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 pods no matter how the pods change.

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

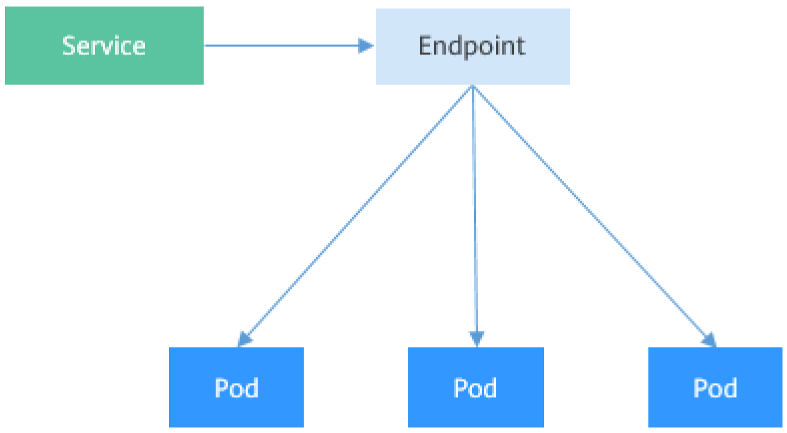
```
$ kubectl describe svc nginx
Name:              nginx
......
Endpoints:         172.16.2.132:80,172.16.3.6:80,172.16.3.7:80
......
```

A record of endpoints is displayed. An endpoint is 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, 172.16.3.6:80, and 172.16.3.7:80 are the IP addresses and ports of pods. You can run the following command to view the IP addresses of the pods, which are the same as the preceding IP addresses:

```
$ 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 recreates the pod, and a new IP address will be assigned to the new pod.

```
$ 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
```

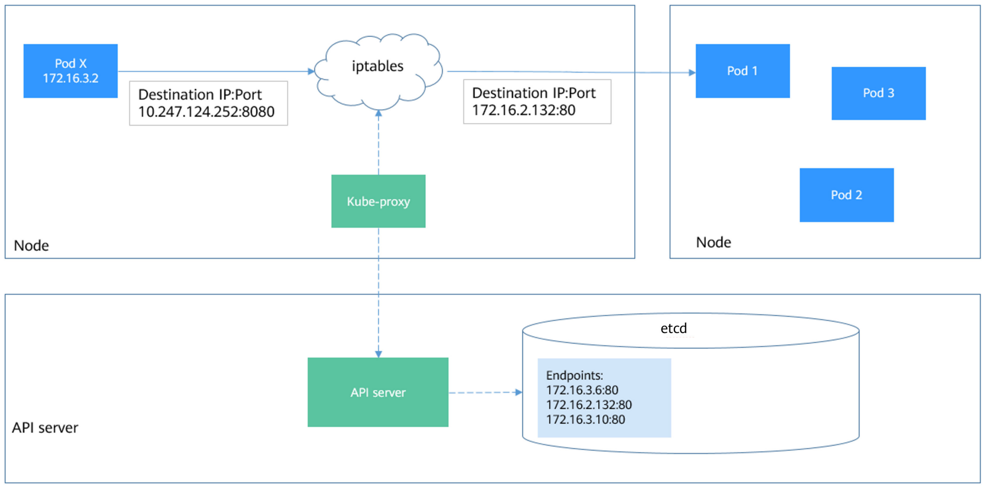
Let's take a closer look at how this happens.

In section **Kubernetes Cluster Architecture**, we have introduced kube-proxy running on worker nodes. 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 worker nodes of the Service creation through the API server. After receiving the notification, kube-proxy records the IP address and port number of the Service through iptables. In this way, the Service can be queried on each node.

The figure below shows how a Service is accessed. When pod X accesses the Service (10.247.124.252:8080), the destination IP address and port are replaced with the IP address and 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 IP address and port of a Service, kube-proxy monitors the changes of the Service and their endpoints to ensure that pods can still be accessed through the Service after the pods are rebuilt.

Figure 7-5 Service access process



Service Types and Application Scenarios

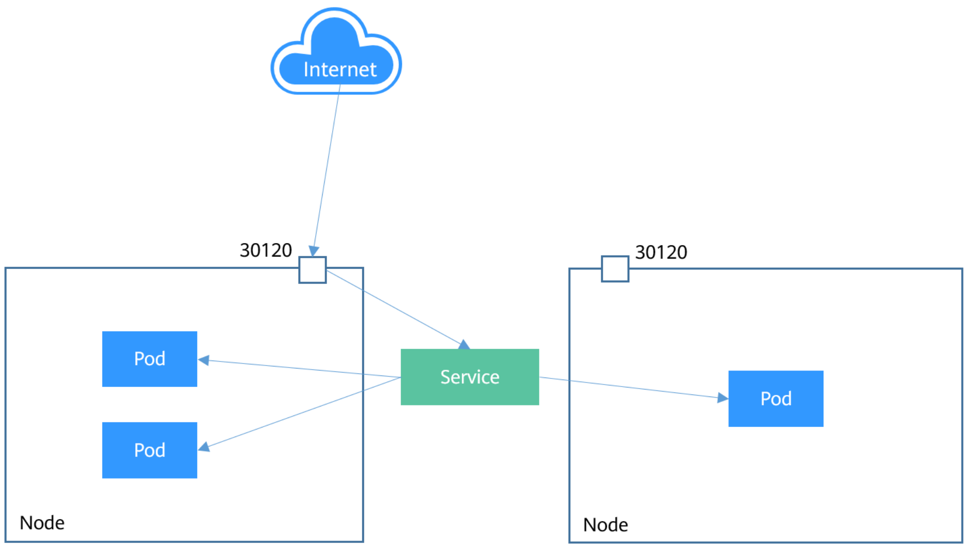
There are several types of Services: ClusterIP, NodePort, LoadBalancer, and Headless Service. Different types of Services offer different functions.

- ClusterIP: The Service is 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, and an external load balancer is used for external systems to access the backend pods. For details, see **LoadBalancer Services**.
- Headless Service: used by pods to discover each other. No separate cluster IP address will be allocated to this type of Service, and the cluster will not balance loads or perform routing for it. You can create a headless Service by setting **spec.clusterIP** to **None**. For details, see **Headless Services**.

NodePort Services

A NodePort Service enables each node in a Kubernetes cluster to reserve the same port. External systems first access the Service through *<node-IP-address>:<node-port>*. The Service then forwards the requests to the pods associated with the Service.

Figure 7-6 A NodePort Service



Below is an example NodePort Service. After the Service is created, you can access backend pods through *<node-IP-address>:<node-port>*.

```
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(S)** 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
```

Accessing the Service using *<node-IP-address>:<node-port>* can access the pod.

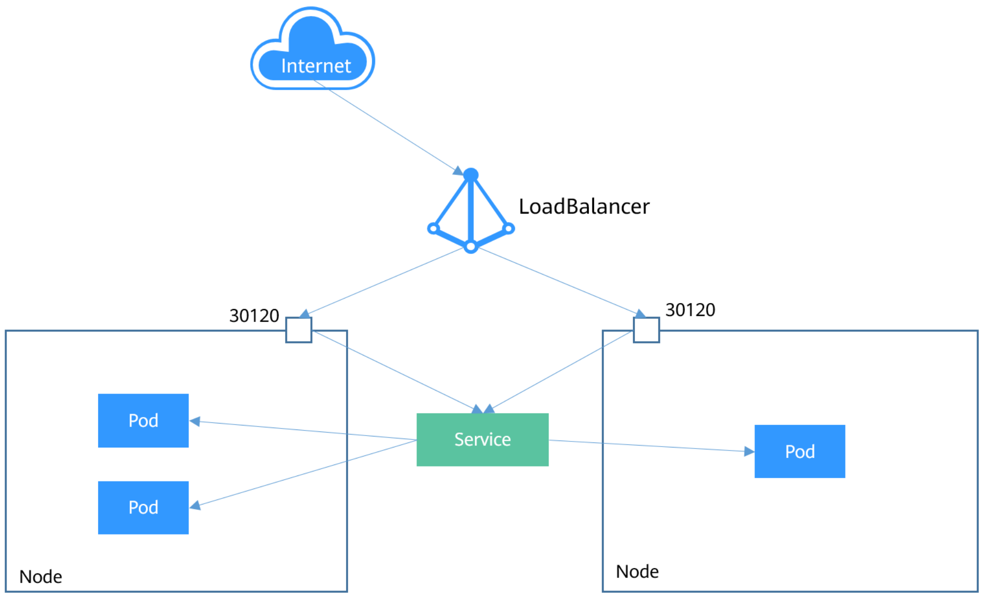
```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 192.168.0.212:30120
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
......
```

LoadBalancer Services

A LoadBalancer Service is exposed externally using a load balancer that forwards requests to a port on the node.

Kubernetes does not directly offer a load balancing component. You can interconnect your Kubernetes cluster with a load balancer of a cloud provider. Cloud providers have different load balancers. For example, CCE interconnects with Elastic Load Balance (ELB). This results in different implementations of LoadBalancer Services.

Figure 7-7 A LoadBalancer Service



The following is an example LoadBalancer Service. After this Service is created, you can access backend pods through *<load-balancer-IP-address>:<load-balancer-listening-port>*.

```
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 load balancer
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30120
  selector:
    app: nginx
  type: LoadBalancer    # Service type. This is a LoadBalancer Service.
```

The parameters in **annotations** under **metadata** are required for CCE LoadBalancer Services. They specify the load balancer that a Service is associated with. When creating a LoadBalancer Service on the CCE console, you can also create a load balancer for the Service. For details, see **LoadBalancer**.

Headless Services

A Service allows a client to access an associated pod for both internal and external network communications. However, there are still the following problems:

- Accessing all pods at the same time
- Allowing pods associated with a Service to access each other

Kubernetes provides headless Services to solve these problems. When a client accesses a non-headless Service, only the cluster IP address of the Service is returned for a DNS query. The cluster forwarding rule (IPVS or iptables) determines which pod will be accessed. A headless Service is not allocated with a separate cluster IP address. During a DNS query, the DNS records of all pods will be returned. In this way, the IP address of each pod can be obtained. **StatefulSets** use headless Services for mutual access between pods.

```
apiVersion: v1
kind: Service       # Object type. This is a Service.
metadata:
  name: nginx-headless
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Name of the port for communications between pods
      port: 80        # Port number for communications between pods
  selector:
    app: nginx        # Select the pod labeled with app:nginx.
  clusterIP: None     # Set this parameter to None, indicating that a headless Service will be created.
```

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 records. 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-headless
Server:         10.247.3.10
Address:        10.247.3.10#53

Name:   nginx-headless.default.svc.cluster.local
Address: 172.16.0.31
Name:   nginx-headless.default.svc.cluster.local
Address: 172.16.0.18
Name:   nginx-headless.default.svc.cluster.local
Address: 172.16.0.19
```

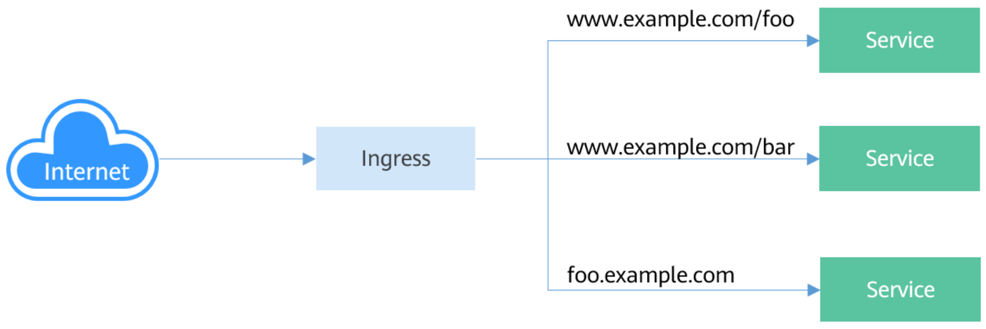
7.3 Ingresses

Video Tutorial

Why Are Ingresses Required?

Services forward requests using TCP and UDP at Layer 4. Ingresses forward requests using HTTP and HTTPS at Layer 7. Domain names and paths can be used for access of finer granularities.

Figure 7-8 An ingress and associated Services

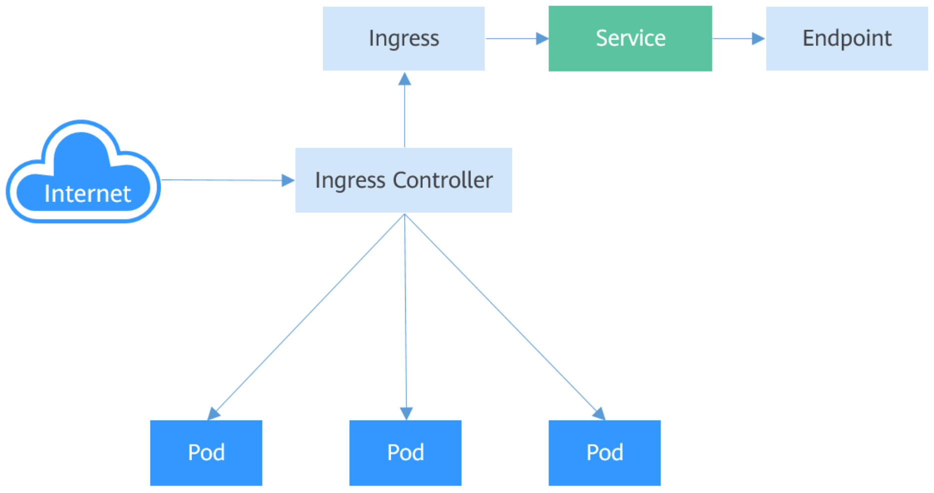


How Ingresses Work

To use Ingresses, you must install an Ingress Controller on your Kubernetes cluster. Cloud providers have different implementations for an Ingress Controller. The most common one is **ingress-nginx**, which is maintained by Kubernetes. CCE works with ELB for load balancing at Layer 7.

The Ingress Controller receives external requests, then finds the corresponding Service based on the routing rule of the Ingress, queries the IP address of the pod through the endpoint, and forwards the requests to the pod.

Figure 7-9 How an Ingress works



Creating an Ingress

In this example, the Ingress uses HTTP for communications and ELB as the Ingress Controller (**metadata.annotations** specifies the load balancer), and associates with a Service (**nginx:8080**). After a request for accessing **http://192.168.10.155:8080/** is initiated, the request is forwarded to the Service (**nginx:8080**) and then to the corresponding pod through the Service.

The following is an example Ingress (only for clusters of v1.23 or later):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.port: '8080'
    kubernetes.io/elb.id: aa7cf5ec-7218-4c43-98d4-c36c0744667a
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: /
            backend:
              service:
                name: nginx
                port:
                  number: 8080
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: cce
```

You can also set an external domain name in an Ingress so that you can access the load balancer through the domain name and then access the associated Service.

NOTE

To use a domain name for access, you need to point the domain name to the IP address of the load balancer. To this end, you can use a DNS service. For example, you can use .

```
...
spec:
  rules:
    - host: www.example.com       # Domain name
      http:
        paths:
          - path: /
            backend:
              service:
                name: nginx
                port:
                  number: 8080
...
```

Routing an Ingress to Multiple Services

An Ingress can be routed to multiple Services at the same time. The following is an example configuration:

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

NOTICE

The paths specified in the Ingress forwarding policy must exist in the backend application. If the paths do not exist, the forwarding fails.

For example, the default path of the Nginx application is **/usr/share/nginx/html**. If you add **/test** in the Ingress forwarding policy, make sure that the Nginx application has the **/usr/share/nginx/html/test** path. If this path does not exist, 404 will return.

```
...
spec:
  rules:
    - host: foo.bar.com          # Host address
      http:
        paths:
          - path: "/foo"
            backend:
              service:
                name: s1
                port:
                  number: 80
          - path: "/bar"
            backend:
              service:
                name: s2
                port:
                  number: 80
...
```

7.4 Readiness Probes

Once a pod is created, it can be immediately selected by a Service, which forwards requests to the pod. However, it takes time to start a pod. If the pod is not ready for some reason (for example, loading the configuration or data or executing a preheating program), it cannot process requests, resulting in request failures.

To solve this problem, Kubernetes provides readiness probes. A readiness probe can be configured for each container in a pod. A pod is only considered ready when the readiness probes configured for all containers are successful. The pod is then added to the endpoint list of the Service and starts to receive traffic.

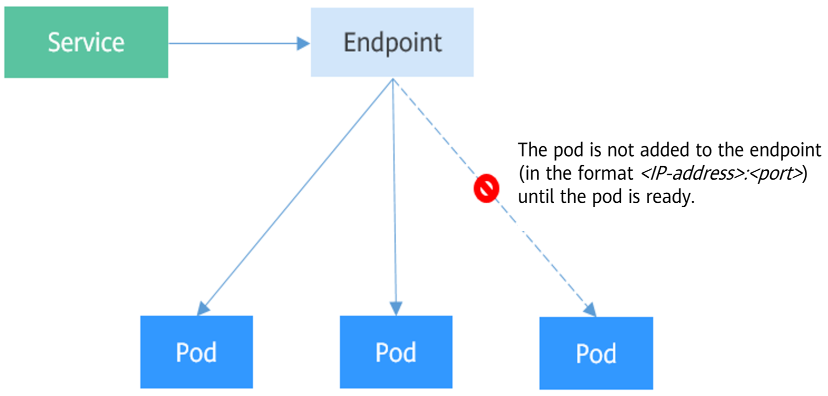
A readiness probe periodically detects a container and determines whether it is ready based on responses. Similar to **Liveness Probes**, there are three kinds of readiness probes.

- Exec: A probe of this kind executes a command in the target container and determines whether the container is ready based on the exit status code. If **0** is returned, the container is considered ready. If a non-zero value is returned, the container is not ready.
- HTTP GET: A probe of this kind uses an HTTP GET request. When a probe of this kind is used, kubelet periodically sends an HTTP GET request to the container port (*<pod-IP>:<container-port>*). If the returned status code is 2xx or 3xx, the probe is successful, and the container is considered ready and can receive traffic. If any other codes are returned, the container is considered not ready, and the Service does not forward traffic to the container.
- TCP socket: A probe of this kind attempts to establish a TCP connection with the target container. If the connection is successful, the container is considered ready.

How Readiness Probes Work

Readiness probes can be implemented using endpoints. Like in the following figure, when a pod is not ready, its IP address and port (in the format *<IP-address>:<port>*) is not in the endpoint list. When the pod is ready, its IP address and port is added to the endpoint list.

Figure 7-10 How readiness probes work


Service
Endpoint
The pod is not added to the endpoint (in the format *<IP-address>:<port>*) until the pod is ready.
Pod
Pod
Pod

Exec

An exec probe is the same as an HTTP GET probe. As shown below, the exec probe runs the **ls /ready** command. If the **/ready** file exists, **0** is returned, indicating that the pod is ready. If the file does not exist, a non-zero status code is returned.

The following uses a Deployment as example. Assume that the **nginx** image used does not contain the **/ready** file. Check whether the pods for the Deployment are ready. The following is an example YAML file:

```
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:              # Defines the ls /ready command.
            command:
            - ls
            - /ready
      imagePullSecrets:
      - name: default-secret
```

Save the above Deployment definition in the **deploy-ready.yaml** file, delete the existing Deployment, and use the **deploy-ready.yaml** file to recreate the Deployment.

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

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

As shown below, because the **nginx** image does not contain the **/ready** file, the value in the **READY** column is **0/1**, indicating that the pods are not in the ready state.

```
# 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. No values are returned for **Endpoints**, indicating that there are no endpoints.

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

If the **/ready** file is created in a container to make the readiness probe work, the container will be in the ready state. If you check the pod and endpoints now, you will find that the pod is already 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

A readiness probe is configured in the same way as a **liveness probe**. It is defined by the **containers** field in the pod description template. As shown below, the workload has three pods, and each pod contains only one container. After the probe for the container in each pod is successful, the pods will be in the ready state.

```
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
          httpGet:                # Defines an HTTP GET request.
            path: /read
            port: 80
      imagePullSecrets:
      - name: default-secret
```

TCP Socket

Another kind of readiness probe uses a TCP socket. The following example shows how to define a readiness probe of this kind.

```
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
          tcpSocket:                # Defines a TCP socket.
            port: 80
      imagePullSecrets:
      - name: default-secret
```

Advanced Settings of Readiness Probes

Similar to a liveness probe, a readiness probe also has the same advanced configuration items, as shown in the output of the **describe** command of the **nginx** pod:

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

The parameters for advanced settings of the readiness probe are detailed as follows:

- **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. If the container fails to respond to the probe, the probe is considered failed.
- **period=10s** indicates that the probe is performed every 10s.
- **#success=1** indicates that the probe is considered successful as long as it succeeds once.
- **#failure=3** indicates that the probe is considered failed after three consecutive times.

These are the default configurations when the probe is created. You can modify them as needed.

```
    readinessProbe:      # Readiness probe
      exec:              # Defines the ls /readiness/ready command.
        command:
        - ls
        - /readiness/ready
      initialDelaySeconds: 10    # The readiness probe is initiated 10s after a container starts.
      timeoutSeconds: 2          # The container must respond within 2s, or the probe is considered failed.
      periodSeconds: 30          # The probe is performed every 30s.
      successThreshold: 1        # The probe is considered successful as long as it succeeds once.
      failureThreshold: 3        # The probe is considered failed after three consecutive failures.
```

7.5 Network Policies

Network policies are designed by Kubernetes to restrict pod access. Like a firewall at the application layer, network policies enhance network security. The capabilities of network policies are determined by the network add-ons available in the cluster.

By default, if a namespace does not have any policies configured, pods in the namespace accept traffic from any ingress sources and send traffic to any egress destinations.

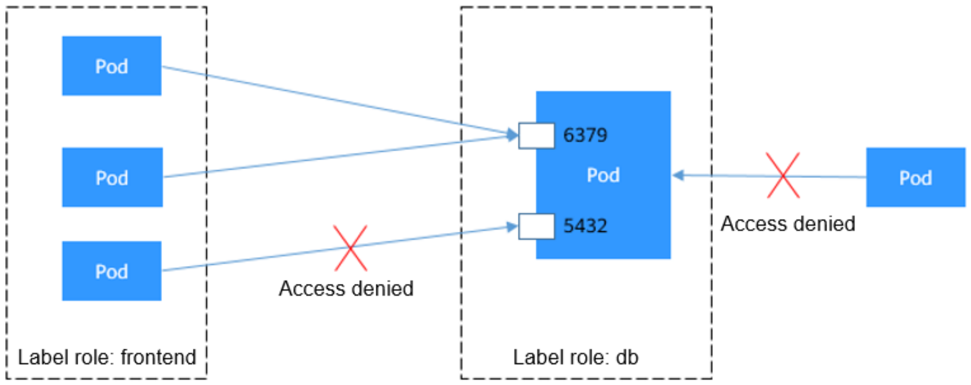
There are three kinds of selectors available for network policies:

- namespaceSelector: selects particular namespaces for which all pods should be allowed as ingress sources or egress destinations.
- podSelector: selects particular pods in the same namespace as the network policy which should be allowed as ingress sources or egress destinations.
- ipBlock: selects particular IP address ranges that are allowed as ingress sources or egress destinations.

Using Ingress Rules Through YAML

- **Scenario 1: Controlled by a preset network policy, a pod can only be accessed by pods with specific labels.**

  Figure 7-11 podSelector

  The pod labeled with **role=db** only permits access to its port 6379 from pods labeled with **role=frontend**. To achieve this, take the following steps:

  a. Create the **access-demo1.yaml** file.

```
vim access-demo1.yaml
```

  File content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-demo1
  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 allows the access of the pods labeled with role=frontend.
        matchLabels:
          role: frontend
    ports:                      # Only TCP can be used to access port 6379.
    - protocol: TCP
      port: 6379
```

  b. Run the following command to create the network policy defined in the **access-demo1.yaml** file:

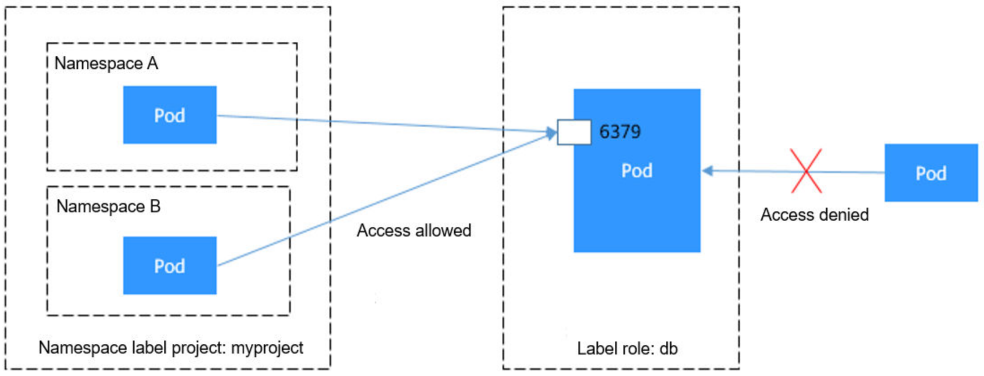
```
kubectl apply -f access-demo1.yaml
```

  Expected output:

```
networkpolicy.networking.k8s.io/access-demo1 created
```

- **Scenario 2: Controlled by a preset network policy, a pod can only be accessed by pods in a specific namespace.**

Figure 7-12 namespaceSelector



The pod labeled with **role=db** only permits access to its port 6379 from pods in the namespace labeled with **project=myproject**. To achieve this, take the following steps:

a. Create the **access-demo2.yaml** file.

```
vim access-demo2.yaml
```

File content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-demo2
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 allows the access of the pods in the namespace labeled
with project=myproject.
        matchLabels:
          project: myproject
    ports:                      # Only TCP can be used to access port 6379.
    - protocol: TCP
      port: 6379
```

b. Run the following command to create the network policy defined in the **access-demo2.yaml** file:

```
kubectl apply -f access-demo2.yaml
```

Expected output:

```
networkpolicy.networking.k8s.io/access-demo2 created
```

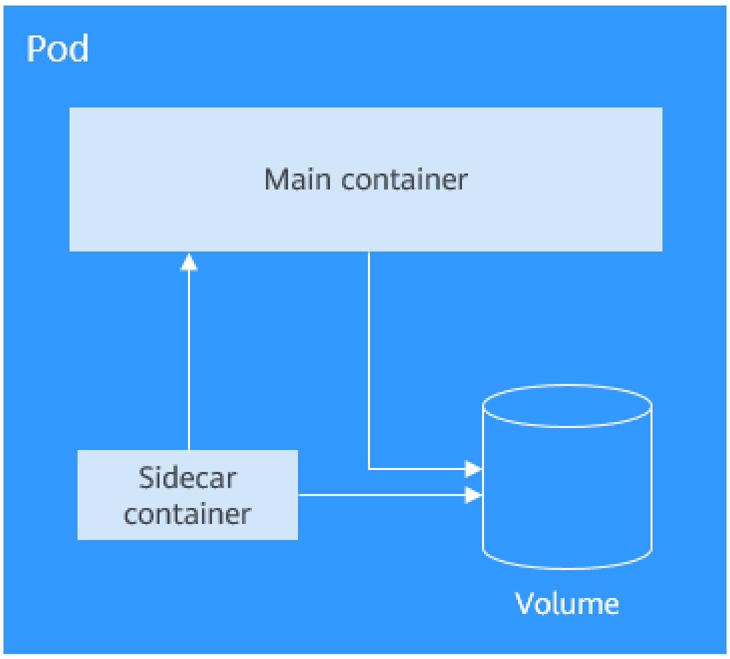
8 Persistent Storage

8.1 Volumes

On-disk files in a container are ephemeral. If the container crashes, the files are lost. Another problem is that on-disk files cannot be easily shared between containers running in a pod. Kubernetes volumes can help solve both problems. Volumes cannot be created independently, but they can be defined in the pod **spec**.

All of the containers in a pod can access the volumes after they are mounted to the pod. A volume can be mounted to any directory in the container.

The following figure shows how a volume is shared by the containers in a pod.



A volume will no longer exist if the pod that it is mounted to ceases to exist. Depending on the volume type, however, files in the volume may outlive the volume itself.

Volume Types

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

- emptyDir: an empty volume used for ephemeral storage
- hostPath: a volume that mounts a directory on the host to a pod
- ConfigMap and secret: special volumes that inject or pass information to the pods. For details about how to use ConfigMaps and secrets as volumes, see **ConfigMaps** and **Secrets**.
- PersistentVolumeClaim: Kubernetes persistent storage class. For details, see **PVs, PVCs, and Storage Classes**.

emptyDir

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

Some uses of an emptyDir volume are as follows:

- An emptyDir volume can provide scratch space, such as for a disk-based merge sort.
- An emptyDir volume can serve as a checkpoint for 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 type of persistent storage volume in Kubernetes. A hostPath volume mounts a file or directory in the file system of the host node to a pod so that the pod can directly access the file resources on the node. Compared with emptyDir that is mainly used for temporary storage, hostPath has the following characteristics:

- Data lifecycle bound to the host node: Data can be restored and used even if the pod is deleted, rebuilt, or scheduled to another node, provided that the node disk is not damaged.
- Cross-pod sharing of node-level resources: Multiple pods on a node can share data through the same hostPath volume (if pods have the permissions to access the file system on the host node).
- Reuse of pre-stored data on the nodes: hostPath can mount existing files or directories (such as system configuration files and hardware drivers) on the nodes to pods, so containers can directly read or modify the pre-stored data.

hostPath is suitable for development and debugging, system component deployment, and scenarios where the host node resources need to be accessed. For example, you may need to mount the log directory, Docker engine, or local cache path of the host node to pods. hostPath has the following limitations:

- hostPath is strongly bound to nodes, which makes it inflexible (lacking portability) and unable to share data across nodes or dynamically allocate storage.
- Security risks are prominent. If sensitive paths (such as **/etc** and **/var/run/docker.sock**) on the nodes are mounted to pods, containers may access node resources without authorization.
- Data persistence is weak. If a node becomes faulty or the disk is damaged, the stored data is easily lost.

For this reason, hostPath is not suitable for production environments that require high data security and reliability, such as databases.

```
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
```

8.2 PVs, PVCs, and Storage Classes

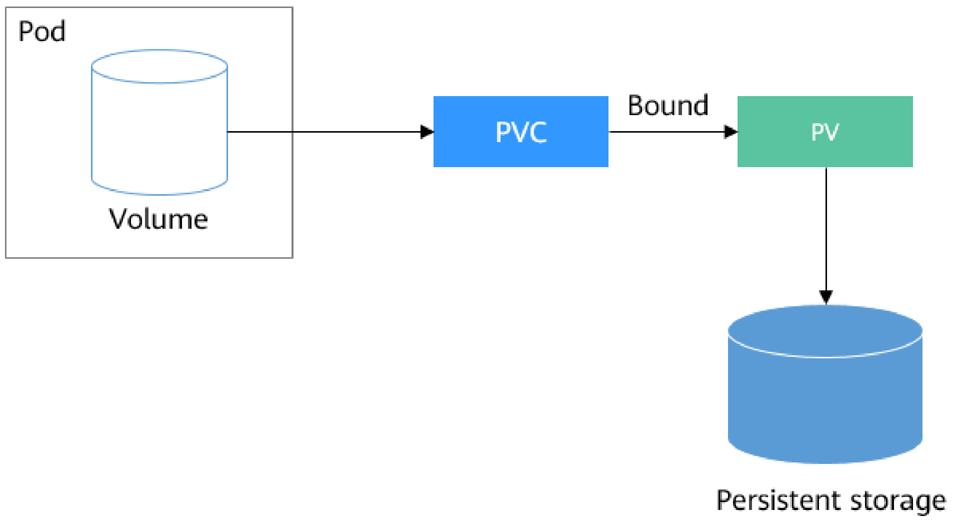
hostPath volumes are a form of persistent storage, as described earlier. However, they are tightly coupled to a specific node. If a node with a hostPath volume becomes faulty and the pods on that node are rescheduled to a different node, the pods may fail to access the original data since that data resides only on the initial node. This could result in lost data.

To ensure pods retain access to original data after rescheduling events (such as node failures or upgrades), Kubernetes requires the use of persistent network storage. Unlike local storage, network storage is not tied to any specific node and continues to provide data access even when pods are relocated. This ensures service continuity. Network storage comes in various forms, including block storage, file storage, and object storage. Cloud service providers typically offer more than three base storage types. To hide these differences, Kubernetes uses PVs and PVCs. They allow developers to request the required storage capacity and access mode, similar to requesting CPUs and memory, without worrying about the specific implementation. Kubernetes then automatically provisions and mounts the appropriate underlying storage. This design decouples storage resources from applications. Users only specify their needs, and the platform handles the allocation. As a result, deployment becomes far more flexible and portable.

- A PV represents a persistent storage volume. It typically defines a directory on the host node, such as a network file system (NFS) mount directory.
- A PVC specifies the attributes of the persistent storage that a pod requires, such as storage volume size and read/write permissions.

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

Figure 8-1 Interaction between a PV and PVC



CSI

Kubernetes offers Container Storage Interface (CSI), which can be used to develop custom CSI add-ons to support specific storage requirements while maintaining decoupling from underlying storage. For example, components developed by CCE such as **everest-csi-controller** and **everest-csi-driver** in the **kube-system** namespace, serve as storage controllers and drivers in **Namespaces**. With **everest-csi-controller** and **everest-csi-driver**, you can use cloud storage services 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
```

PVs

The following shows how to define a PV. In this example, a file system is created in SFS, with the file system ID **68e4a4fd-d759-444b-8265-20dc66c8c502** and the mount point **sfs-nas01.cn-north-4b.myhuaweicloud.com:/share-96314776**. To use this file system in CCE, create a 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                        # Storage class
    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 this example are specifically designed for CCE clusters.

Next, create the PV and view its details.

```
$ 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 defines how a PV is managed after its bound PVC is released. **Retain** means the PV remains in the system, even after the bound PVC is deleted. **Available** indicates that the PV is available.

PVCs

Each PVC can only have one PV bound. 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 and view its details.

```
$ 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 PVC is in the **Bound** state, and the value of **VOLUME** is **pv-example**, indicating that the PVC has a PV bound.

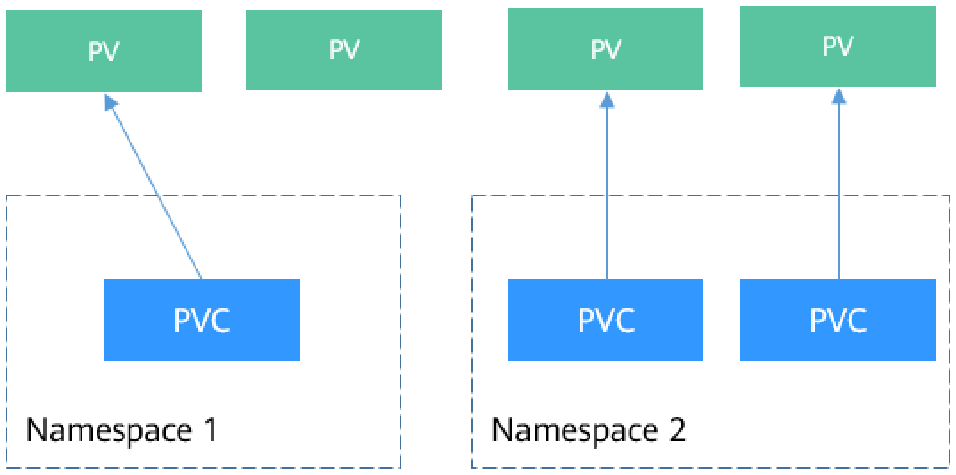
Then, check the PV status.

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

PVs are cluster-level resources and do not belong to any namespace. PVCs, in contrast, are namespace-level resources. PVs can be bound to PVCs in any namespace. In this example, the namespace name **default** followed by the PVC name is displayed under **CLAIM**.

Figure 8-2 Relationship between PVs and PVCs



Storage Classes

PVs and PVCs hide the differences between different types of physical storage, but creating a PV can be complex, especially the configuration of the **csi** field. In addition, PVs and PVCs are generally managed by the cluster administrator. It is inconvenient for you to configure varying attributes for them.

To solve this problem, Kubernetes supports dynamic PV provisioning, which automates the creation of PVs. The cluster administrator can deploy a PV provisioner and define storage classes. In this way, you can select a desired storage class when creating a PVC. The PVC then transfers the storage class to the PV provisioner, which automatically creates a PV. In CCE, storage classes such as csi-disk, csi-nas, and csi-obs are supported. By including the **storageClassName** field in a PVC, CCE ensures that PVs are dynamically provisioned, with underlying storage resources created automatically.

You can run the command below to obtain the storage classes that CCE supports. You can use the CSI add-ons provided by CCE to customize storage classes, which function 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
```

Specify a storage class for creating 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 using the **csi-sfsturbo** storage class. To use SFS Turbo storage, create an SFS Turbo file system and then statically provision a PV and PVC. For details, see **Using an Existing SFS Turbo File System Through a Static PV**.

Create the 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 storage class is specified, a PVC and PV have been created and bound.

After a storage class is specified, PVs can be automatically created and maintained. You only need to specify **StorageClassName** when creating a PVC, which greatly reduces the workload. The types of **StorageClassName** vary by vendor. SFS is only used as an example.

Using a PVC in a Pod

With a PVC, you can easily use persistent storage in a pod. In the pod template, you simply reference the name of the PVC in the **volumes** field and mount it to the pod, as shown in the following example. You can also create a PVC for a StatefulSet. For details, see **StatefulSets**.

```
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
```

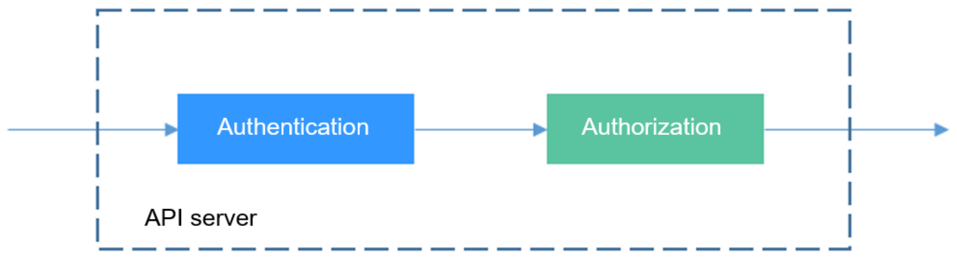
9 Authentication and Authorization

9.1 Service Accounts

All requests to Kubernetes go through the API server, and they require authentication and authorization before access is granted.

- Authentication: authenticates user identities. Kubernetes uses different authentication rules for external and internal service accounts. For details, see **Authentication and Service Accounts**.
- Authorization: controls user access to resources. Role-based access control (RBAC) is used to authorize users to access resources. For details, see **RBAC**.

Figure 9-1 API server authentication and authorization



Authentication and Service Accounts

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

- A service account is bound to a namespace and associated with a set of credentials. When a pod is created, a token is mounted to the pod so that the pod can be called by the API server.
- Kubernetes does not come with pre-built objects for managing common accounts. Instead, external services are used for this purpose. For example, CCE users are managed through Identity and Access Management (IAM).

Service accounts in Kubernetes are a kind of namespace-level resources, just like pods and ConfigMaps. When a namespace is created, the system automatically generates a service account named **default** in the namespace.

You can run the following command to check the service account:

kubectl get sa

```
NAME     SECRETS   AGE
default  1         30d
```

NOTE

- In clusters earlier than v1.21, tokens are obtained by mounting the secret of a service account to a pod. Such tokens are permanent. However, this approach is not recommended in clusters of v1.21 or later. Starting from v1.25, Kubernetes no longer automatically creates secrets for service accounts as part of its community iteration policy.

  Instead, in clusters of v1.21 and later, the recommended approach is to use the **TokenRequest** API to **obtain tokens** and mount them via a projected volume to pods. These tokens remain valid only for a fixed period and become invalid once the pods are deleted.
- If you need a token that never expires, you can **manually manage secrets for service accounts**. Although a permanent service account token can be created manually, you are advised to use a short-lived token by calling the **TokenRequest** API for better security.

In Kubernetes clusters earlier than v1.25, a secret was automatically generated for each service account. However, in clusters of v1.25 and later, no secret is automatically created for each service account. The following describes how to check the statuses of service accounts in clusters earlier than v1.25 and clusters of v1.25 or later.

- In a cluster earlier than v1.25, run the following command to check the status of the **default** service account:

  kubectl describe sa default

  If information similar to the following is displayed, the **default-token-vssmw** secret has been automatically created:

  ```
  Name:                default
  Namespace:           default
  Labels:              <none>
  Annotations:         <none>
  Image pull secrets:  <none>
  Mountable secrets:   default-token-vssmw
  Tokens:              default-token-vssmw
  Events:              <none>
  ```
- In a cluster of v1.25 or later, run the following command to check the status of the **default** service account:

  kubectl describe sa default

  The command output shows no secret is automatically created for the **default** service account.

  ```
  Name:                default
  Namespace:           default
  Labels:              <none>
  Annotations:         <none>
  Image pull secrets:  <none>
  Mountable secrets:   <none>
  Tokens:              <none>
  Events:              <none>
  ```

When defining a pod, you can assign a service account to it by specifying the account name in the file. If no account name is specified, the default service account will be used. When receiving a request with an authentication token, the API server uses the token to check whether the service account associated with the client that sends the request allows the request to be executed.

Creating a Service Account

Step 1 Create a service account in the **default** namespace. A v1.29 cluster is used in this example.

kubectl create serviceaccount *sa-example*

```
serviceaccount/sa-example created
```

Check whether **sa-example** has been created. If **sa-example** is displayed in the **NAME** column, it has been created.

kubectl get sa

```
NAME          SECRETS   AGE
default       1         30d
sa-example    0         2s
```

Because the cluster version used in this example is later than v1.25, the service account will not have a secret created automatically. To check if a secret has been created, use the command below to view the service account details. If the output shows **none** for **Mountable secrets** and **Tokens**, then no secret is automatically created for the service account.

kubectl describe sa *sa-example*

```
Name:                sa-example
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   <none>
Tokens:              <none>
Events:              <none>
```

Step 2 Manually create a secret named **sa-example-token** and associate it with the **sa-example** service account. By manually managing this secret, you can obtain a token that never expires.

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  namespace: default
  name: sa-example-token
  annotations:
    kubernetes.io/service-account.name: sa-example
type: kubernetes.io/service-account-token
EOF
```

Step 3 Check whether **sa-example-token** has been created. If **sa-example-token** is present in the secrets of the **default** namespace, then it has been created.

kubectl get secrets

```
NAME                TYPE                                  DATA   AGE
default-secret      kubernetes.io/dockerconfigjson        1      6d20h
paas.elb            cfe/secure-opaque                     1      6d20h
sa-example-token    kubernetes.io/service-account-token   3      16s
```

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

kubectl describe secret *sa-example-token*

```
Name:         sa-example-token
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: sa-example
              kubernetes.io/service-account.uid: 4b7d3e19-1dfe-4ee0-bb49-4e2e0c3c5e25

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1123 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSU...
```

Step 4 Check whether the service account has been associated with the new secret, meaning if the service account has obtained the token. The command output shows that **sa-example** is associated with **sa-example-token**.

kubectl describe sa *sa-example*

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

----End

Using a Service Account in a Pod

It is convenient to use a service account in a pod. You only need to specify the name of the service account. The following uses **nginx:latest** as an example to describe how to use a service account in a pod.

Step 1 Create a description file named **sa-pod.yaml**. **mysql.yaml** is an example file name. You can rename it to whatever you like.

vim *sa-pod.yaml*

NOTICE

To enable the pod to use the token from the manually created secret, you must mount the secret to the container. For details about how to mount the secret, see the code in bold in the file.

The file content is as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-pod
spec:
  serviceAccountName: sa-example                # Specify sa-example as the service account used by the pod.
  imagePullSecrets:
```

```
  - name: default-secret
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    volumeMounts:                         #  Mount the storage volume named secret-volume to the pod.
    - name: secret-volume
      readOnly: true                      #  The mounted storage volume is read-only.
      mountPath: "/etc/secret-volume"      #  Mount path of the storage volume in the container. You 
can specify the value as needed.
  volumes:                                #  Define a secret volume that can be used by the pod.
  - name: secret-volume                   #   Name of the secret volume. You can specify the value as 
needed.
    secret:                               #  Set the type of the storage volume to secret.
      secretName: sa-example-token         #  Mount sa-example-token to the defined storage volume.
```

Step 2 Create a pod and view its details. You can see that **sa-example-token** is mounted to the pod. The pod uses the token for authentication.

kubectl create -f *sa-pod.yaml*

Information similar to the following is displayed:

```
pod/sa-pod created
```

Check whether the pod has been created.

kubectl get pod

In the command output, if **sa-pod** is in the **Running** state, the pod has been created.

```
NAME                  READY   STATUS            RESTARTS   AGE
sa-pod              1/1     running           0          5s
```

Step 3 View the **sa-pod** pod details and check whether **sa-example-token** has been mounted to the pod.

kubectl describe pod *sa-pod*

Information similar to the following is displayed:

```
...
Containers:
  container-0:
    Container ID:
    Image:          nginx:latest
    Image ID:
    Port:           <none>
    Host Port:      <none>
    State:          Waiting
      Reason:       ImagePullBackOff
    Ready:          False
    Restart Count:  0
    Limits:
      cpu:     100m
      memory:  200Mi
    Requests:
      cpu:        100m
      memory:     200Mi
    Environment:  <none>
    Mounts:
```

```
    # The sa-example-token has been mounted to the pod. The pod can use the token for authentication.
   /etc/secret-volume from secret-volume (ro)
    # Automatically mounted TokenRequest. It can provide a short-term token.
   /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-2s4sw (ro)
...
```

You can also run the command shown here to view the corresponding files in the pod. The path after **cd** needs to be **secret-volume**.

kubectl exec -it *sa-pod* -- /bin/sh

cd */etc/secret-volume*

ls

Information similar to the following is displayed:

```
ca.crt    namespace  token
```

Step 4 Verify that the manually created service account token can work.

1. In a Kubernetes cluster, a Service named **kubernetes** is created for the API server by default. Pods can be accessed through this Service. After exiting the pod by pressing **Ctrl**+**D**, view the Service details.

   kubectl get svc

   Information similar to the following is displayed:

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

2. Access the pod and check whether the pod can access pod resources in the cluster through the API server without using the token.

   kubectl exec -it *sa-pod* -- /bin/sh

   curl *https://10.247.0.1:443*/api/v1/namespaces/default/pods

   If information similar to the following is displayed, the pod cannot directly access pod resources in the cluster through the API server:

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

3. Configure the environment variables of **ca.crt**. Add the path of **ca.crt** to the **CURL_CA_BUNDLE** environment variable. This variable instructs **curl** to use the certificate file as the trust anchor.

   export CURL_CA_BUNDLE = */etc/secret-volume/ca.crt*

4. Add the token content to **TOKEN**.

   TOKEN=$(cat */etc/secret-volume/token*)

   Check whether **TOKEN** has been configured.

   echo $TOKEN

   If information similar to the following is displayed, the **TOKEN** has been configured:

   ```
   eyJhbGciOiJSUzI1NiIsImtpZCI6I...
   ```

5. Access the API server using the configured **TOKEN**.

   curl -H "Authorization: Bearer $TOKEN" *https://10.247.0.1:443*/api/v1/namespaces/default/pods

If information similar to the following is displayed, it means that the pod has been authenticated and the manually created service account token is in use. If the API server returns **cannot get path \"/api/v1/namespaces/default/pods\""**, the pod does not have sufficient permissions. The pod can only access the API server after being authorized. For details about the authorization mechanism, see **RBAC**.

```
"kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "13267712"
  },
  "items": [
    {
      "metadata": {
        "name": "hpa-example-77b9b446f6-nc7b6",
...
```

----End

9.2 RBAC

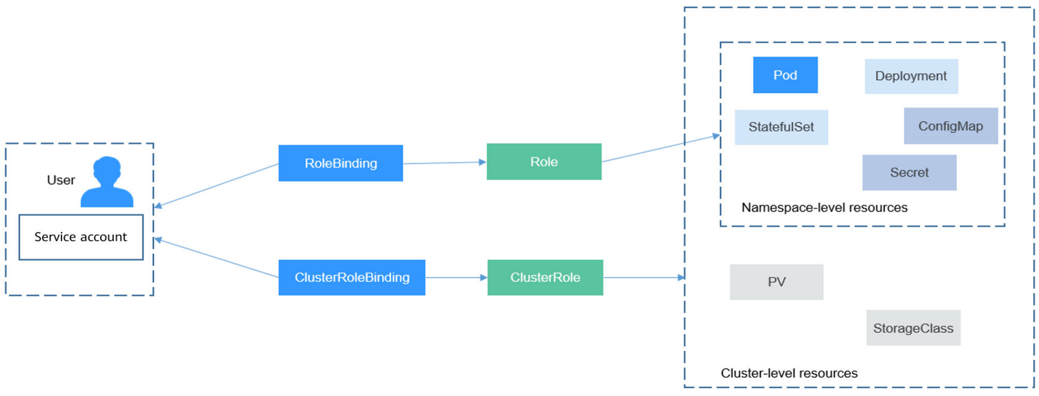
RBAC Resources

In Kubernetes, RBAC is used for authorization. RBAC authorization rules are configured using four types of resources.

- Roles: define a set of rules for accessing Kubernetes resources in a namespace.
- RoleBindings: define the relationship between users and roles.
- ClusterRoles: define a set of rules for accessing Kubernetes resources in a cluster (including all namespaces).
- ClusterRoleBindings: define the relationship between users and cluster roles.

Roles and ClusterRoles specify actions that can be performed on specific resources. RoleBindings and ClusterRoleBindings bind Roles to specific users, user groups, or service accounts.

Figure 9-2 Role binding



Creating a Role

The definition of a Role is simple. You just specify a namespace and some rules. For example, the following rules allow you to perform 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.
```

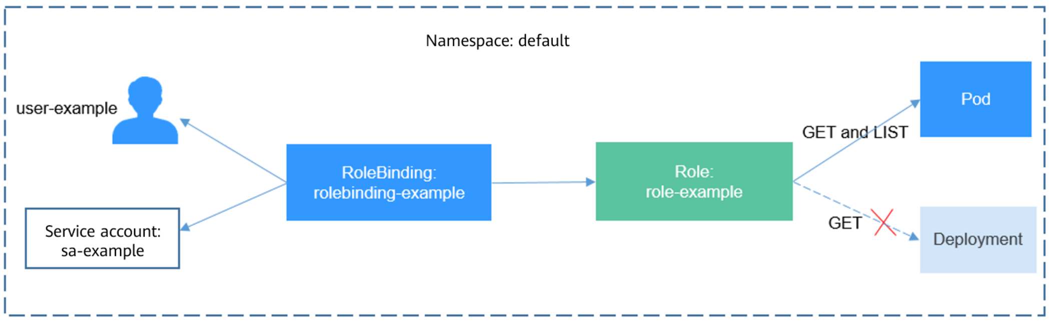
Creating a RoleBinding

After creating a Role, you can bind the Role to a specific user, which is called RoleBinding. The following shows 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                   # Service account
  name: sa-example
  namespace: default
roleRef:                                 # Specified Role
  kind: Role
  name: role-example
  apiGroup: rbac.authorization.k8s.io
```

subjects is used to bind the Role to a user. The user can be an external common user or a service account. For details about the two user types, see **Service Accounts**. The following figure shows the relationships.

Figure 9-3 Binding a Role to a user



Then check whether the authorization takes effect.

In **Using a Service Account in a Pod**, a pod is created, and the **sa-example** service account is used. The **role-example** Role is bound to **sa-example**. Access the pod and run **curl** to access resources through the API server to check whether the permissions have been applied.

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

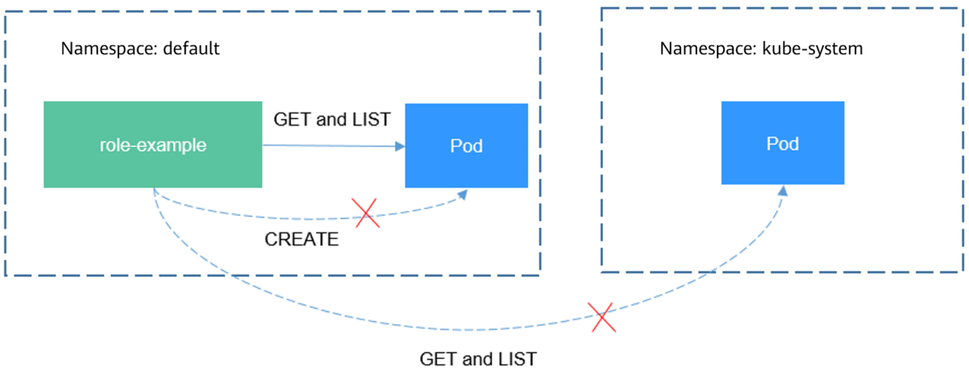
```
$ kubectl exec -it sa-pod -- /bin/sh
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods
{
  "kind": "PodList",
  "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 command output indicates a normal status, **sa-example** has the required permissions to list pods. Obtain the Deployment again. If information similar to the following is displayed, you do not have the permissions 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\"",
...
```

Roles and RoleBindings 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 9-4 Role and RoleBinding applied to namespaces



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

```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/kube-system/pods
...
```

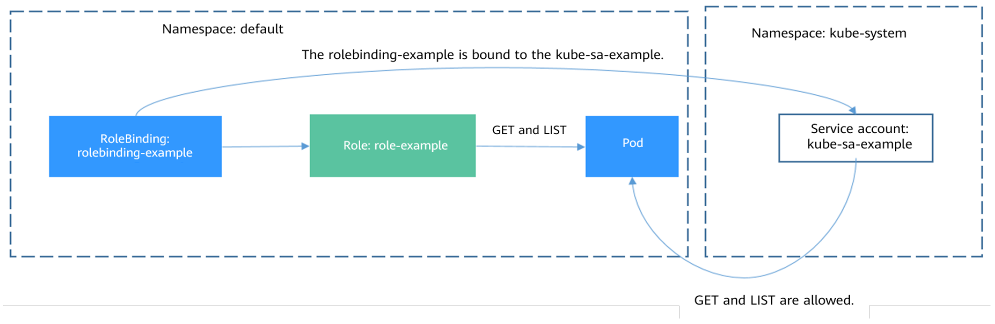
```
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list resource
\"pods\" in API group \"\" in the namespace \"kube-system\"",
  "reason": "Forbidden",
...
```

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

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

Then the **kube-sa-example** service account in **kube-system** can perform GET and LIST operations on pods in the **default** namespace.

Figure 9-5 Cross-namespace access


Namespace: default
The rolebinding-example is bound to the kube-sa-example.
Namespace: kube-system
RoleBinding: rolebinding-example
Role: role-example
GET and LIST
Pod
Service account: kube-sa-example
GET and LIST are allowed.

ClusterRoles and ClusterRoleBindings

Compared with Roles and RoleBindings, ClusterRoles and ClusterRoleBindings have the following differences:

- There is no need to define the **namespace** field for a ClusterRole or a ClusterRoleBinding.
- You can use a ClusterRole to define permissions on cluster-level resources.

ClusterRoles and ClusterRoleBindings are used to control cluster-level permissions.

There are many ClusterRoles and ClusterRoleBindings defined by default in Kubernetes.

```
$ 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**: grants read-only access to the resources in a namespace.
- **edit**: grants read/write access to most resources in a namespace.
- **admin**: grants all permissions for the resources in namespaces.
- **cluster-admin**: grants all permissions for the resources in the cluster.

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

Typically, the four ClusterRoles are bound to users to isolate permissions. The key idea is that Roles, which define rules and permissions, are independent of users. By using RoleBindings, you can associate Roles with users, allowing flexible control over access permissions.

10 Auto Scaling

Pod Orchestration and Scheduling describes how to control the number of pods by using controllers such as Deployments. You can manually scale in or out applications by adjusting the number of pods, but manual scaling can be slow and complex, which is a problem when fast scaling is required to handle traffic surges.

To solve this, Kubernetes supports auto scaling for both pods and nodes. By defining auto scaling rules, Kubernetes can dynamically scale pods and nodes based on metrics like CPU usage.

Prometheus and Metrics Server

To enable auto scaling in Kubernetes, the system must first be able to monitor key performance metrics, such as CPU and memory usage for nodes, pods, and containers. However, Kubernetes does not include built-in monitoring capabilities. It instead relies on external projects to extend its functionality.

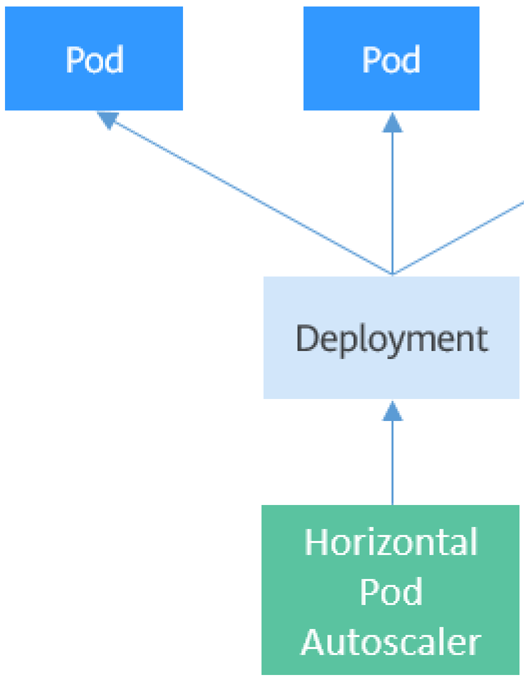
- **Prometheus** is an open-source monitoring and alerting framework that collects a wide range of metrics, making it the standard monitoring solution for Kubernetes.
- **Metrics Server** functions as a resource usage aggregator in Kubernetes clusters, pulling data from the Summary API exposed by kubelet. It provides standardized APIs for external systems, offering insights into core Kubernetes resources such as pods, nodes, containers, and Services.

Horizontal Pod Autoscaler (HPA) integrates with Metrics Server to implement auto scaling based on CPU and memory usage. Additionally, HPA can work with Prometheus to enable auto scaling using custom monitoring metrics.

How HPA Works

An HPA controls horizontal scaling of pods. It periodically checks pod metrics, calculates how many pods are needed to meet target values, and updates the **replicas** field of the associated workload such as a Deployment.

Figure 10-1 HPA working rules



You can configure one or more metrics for an HPA. When only one metric is used, the HPA totals the metric values from the current pods, divides that total by the expected value, and rounds up the result to determine the required number of pods. For example, if a Deployment has three pods with the CPU usage of each pod at 70%, 50%, and 90%, respectively, and the expected CPU usage configured for HPA is 50%, the expected number of pods is calculated as follows: (70 + 50 + 90)/50 = 4.2. The required number of pods is rounded up to 5.

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

Using an HPA

The following example demonstrates how to use an HPA. First, create a Deployment with four pods using an Nginx image.

```
$ 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 pods ranges from 1 to 10.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: scale
  namespace: default
spec:
  scaleTargetRef:                    # Target resource
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
  minReplicas: 1                     # The minimum number of pods for the target resource
  maxReplicas: 10                    # The maximum number of pods for the target resource
  metrics:                           # Metric. The expected CPU usage is 70%.
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Create the HPA and check its details.

```
$ kubectl create -f hpa.yaml
horizontalpodautoscaler.autoscaling/scale 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 scale in some pods. The expected number of pods can be calculated as follows: (0 + 0 + 0 + 0)/70 = 0. However, the minimum number of pods was set to **1**, so the number of pods will be 1. After a while, you can see that there is only one pod.

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

Check the HPA again. You can see that there is a record similar to the following under **Events**. This record shows that 21 seconds ago, the HPA scaled in the Deployment, reducing the total pod count to 1. The adjustment occurred because the number of pods calculated from all metrics fell below the expected value.

```
$ 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 check the Deployment details again, you can see that there is a record similar to the following under **Events**. This record shows that the number of Deployment pods has been adjusted to 1, aligning with the HPA configuration.

```
$ 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

HPAs focus on scaling pods, but when cluster resources become insufficient, the only option is to add nodes. Scaling cluster nodes can be complex, but in cloud-based environments, nodes can be dynamically added or removed using APIs, making the process much more convenient.

Kubernetes offers **Cluster Autoscaler**, a component designed to automatically scale cluster nodes based on pod scheduling demands and resource usage. However, because this relies on cloud provider APIs, the implementation and usage vary across different environments.

For details about the implementation in CCE, see **Creating a Node Scaling Policy**.