

**Cloud Container Engine**

# **Kubernetes Basics**

**Issue**            01  
**Date**             2025-02-11



**Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. All rights reserved.**

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

## **Trademarks and Permissions**



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

## **Notice**

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

## **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

---

<b>1 Overview</b>	<b>1</b>
<b>2 Basic Concepts</b>	<b>3</b>
<b>3 Containers and Kubernetes</b>	<b>12</b>
3.1 Containers	12
3.2 Kubernetes	17
3.3 Using kubectl to Operate a Cluster	23
<b>4 Pods, Labels, and Namespaces</b>	<b>31</b>
4.1 Pod: the Smallest Scheduling Unit in Kubernetes	31
4.2 Liveness Probes	35
4.3 Label for Managing Pods	39
4.4 Namespace for Grouping Resources	41
<b>5 Pod Orchestration and Scheduling</b>	<b>43</b>
5.1 Deployments	43
5.2 StatefulSets	47
5.3 Jobs and CronJobs	52
5.4 DaemonSets	54
5.5 Affinity and Anti-Affinity Scheduling	56
<b>6 Configuration Management</b>	<b>65</b>
6.1 ConfigMaps	65
6.2 Secrets	66
<b>7 Kubernetes Networking</b>	<b>69</b>
7.1 Container Networking	69
7.2 Services	71
7.3 Ingresses	79
7.4 Readiness Probes	82
7.5 Network Policies	86
<b>8 Persistent Storage</b>	<b>91</b>
8.1 Volumes	91
8.2 PersistentVolumes, PersistentVolumeClaims, and StorageClasses	93
<b>9 Authentication and Authorization</b>	<b>98</b>

---

9.1 Service Accounts.....	98
9.2 RBAC.....	104
<b>10 Auto Scaling.....</b>	<b>109</b>

# 1 Overview

---

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications across in-cloud hosts.

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

You can access CCE, a hosted Kubernetes service, using the CCE console, `kubectl`, or Kubernetes APIs. Before using CCE, learn about the following Kubernetes concepts.

## Containers and Kubernetes

- [Containers](#)
- [Kubernetes](#)
- [Using kubectl to Operate a Cluster](#)

## Pods, Labels, and Namespaces

- [Pod: the Smallest Scheduling Unit in Kubernetes](#)
- [Liveness Probes](#)
- [Label for Managing Pods](#)
- [Namespace for Grouping Resources](#)

## Pod Orchestration and Scheduling

- [Deployments](#)
- [StatefulSets](#)
- [Jobs and CronJobs](#)
- [DaemonSets](#)
- [Affinity and Anti-Affinity Scheduling](#)

## Configuration Management

- [ConfigMaps](#)
- [Secrets](#)

## Kubernetes Networking

- [Container Networking](#)
- [Services](#)
- [Ingresses](#)
- [Readiness Probes](#)
- [Network Policies](#)

## Persistent Storage

- [Volumes](#)
- [PersistentVolumes, PersistentVolumeClaims, and StorageClasses](#)

## Authentication and Authorization

- [Service Accounts](#)
- [RBAC](#)

## Auto Scaling

- [Auto Scaling](#)

# 2 Basic Concepts

CCE is a scalable, enterprise-class hosted Kubernetes service. With CCE, you can easily deploy, manage, and scale containerized applications in the cloud.

The graphical CCE console enables E2E user experiences. In addition, CCE supports native Kubernetes APIs and kubectl. Before using CCE, understand related basic concepts.

## Cluster

A cluster is a group of one or more cloud servers (also known as nodes) in the same subnet. It has all the cloud resources (including VPCs and compute resources) required for running containers.

CCE supports the following cluster types.

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

Cluster Type	Description
CCE Turbo cluster	<p>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, offering you a one-stop and cost-effective container service.</p> <p>The Cloud Native 2.0 networks are available 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 directly access container IP addresses for high performance.</p>
CCE Autopilot cluster	<p>CCE Autopilot allows you to create serverless clusters that offer optimized Kubernetes compatibility and free you from O&amp;M.</p> <p>CCE Autopilot clusters can be deployed without user nodes, simplifying the application deployment. There is no need to purchase nodes or maintain the deployment, management, and security of nodes. You only need to focus on the implementation of application service logic, which greatly reduces your O&amp;M costs and improves the reliability and scalability of applications.</p>

## Node

A node is a cloud server (virtual or physical machine) running an instance of the Docker Engine. Containers are deployed, run, and managed on nodes. The node agent (kubelet) runs on each node to manage container instances on the node. The number of nodes in a cluster can be scaled.

## Node Pool

A node pool contains one node or a group of nodes with identical configuration in a cluster.

## VPC

A VPC is a logically isolated virtual network that facilitates secure internal network management and configurations. Resources in the same VPC can communicate with each other, but those in different VPCs cannot communicate with each other by default. VPCs provide the same network functions as physical networks and also advanced network services, such as elastic IP addresses and security groups.

## Security Group

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

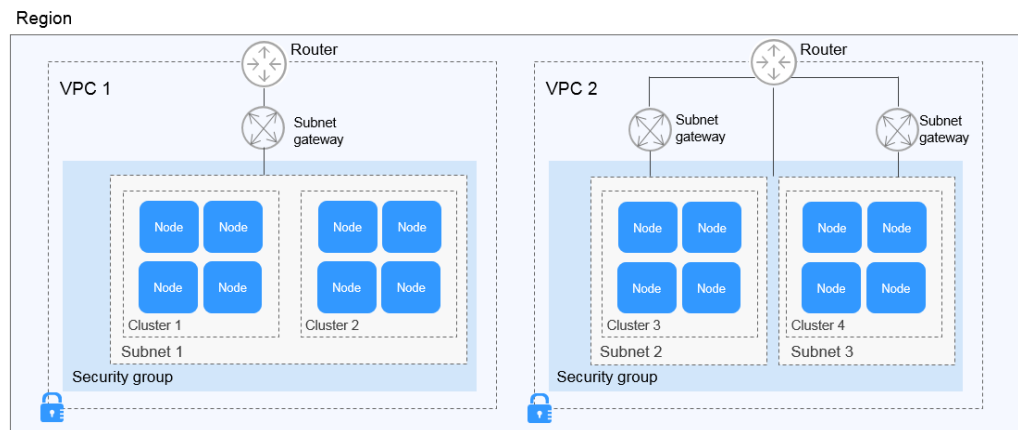


### Relationship Between Clusters, VPCs, Security Groups, and Nodes

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 a subnet gateway. A cluster is created in a subnet. There are three scenarios:

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

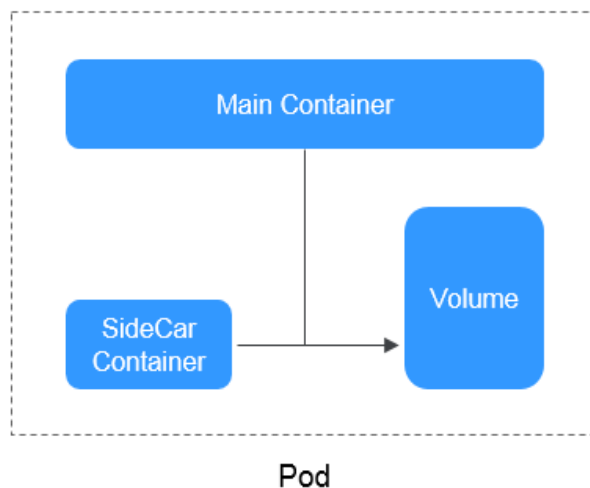
**Figure 2-1** Relationship between clusters, VPCs, security groups, and nodes



## Pod

A pod in Kubernetes is the smallest, basic unit for deploying applications or services. It can contain one or more containers, which typically share storage and networks. 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.

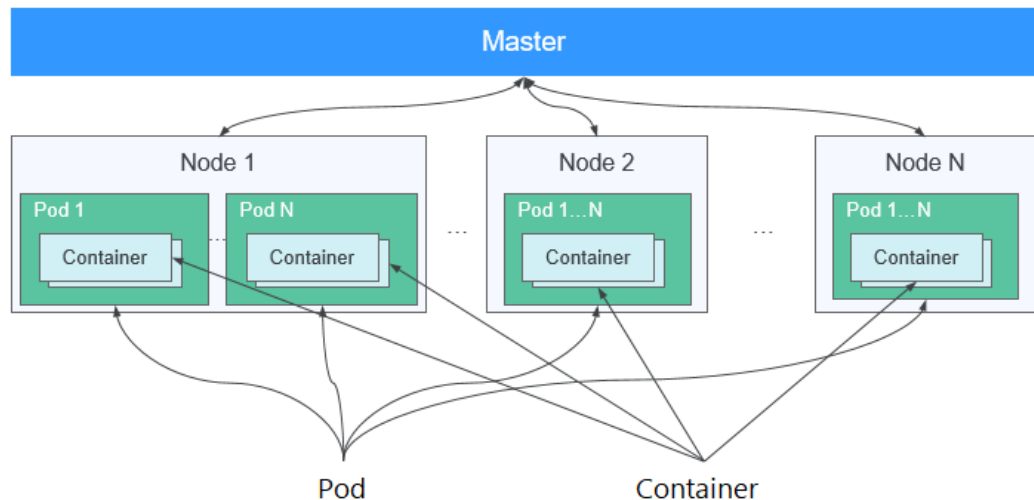
**Figure 2-2** Pod



## Container

A container is a running instance that is created using a Docker image. Multiple containers can run on a node (host). Containers are essentially processes, but they run in their own separate namespaces, unlike processes that directly run on the host machine. These namespaces provide isolation, 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



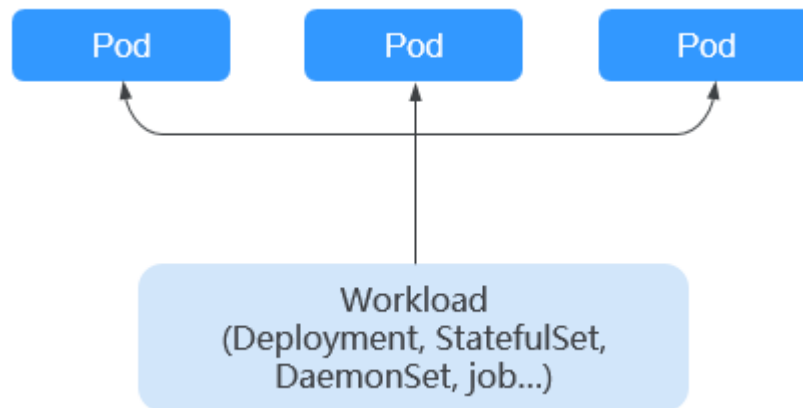
## Workload

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

- **Deployment:** Pods are completely independent of each other and functionally identical. They feature auto scaling and rolling upgrade. Typical examples include web applications like Nginx and blog platforms like WordPress.
- **StatefulSet :** A **StatefulSet** in Kubernetes allows for the organized deployment and removal of pods. Each pod in a StatefulSet has a unique identifier and can communicate with other pods. 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.
- **DaemonSet:** A **DaemonSet** in Kubernetes guarantees that all or specific nodes have a DaemonSet pod running and automatically deploys DaemonSet pods on newly added nodes in a cluster. It is used for services that need to run on every node, like log collection (Fluentd) and monitoring agent (Prometheus Node Exporter) services.
- **Job:** A job in Kubernetes is a task that ensures a specific number of pods are successfully executed. It is used for one-off tasks that need to be performed in a cluster, such as data backup and batch processing.

- **Cron job:** A **cron job** in Kubernetes is a task that runs at a specific time and is used for recurring tasks that need to be executed periodically. It is commonly used for scheduled data synchronization and generating reports on a regular basis.

**Figure 2-4** Relationship between workloads and pods

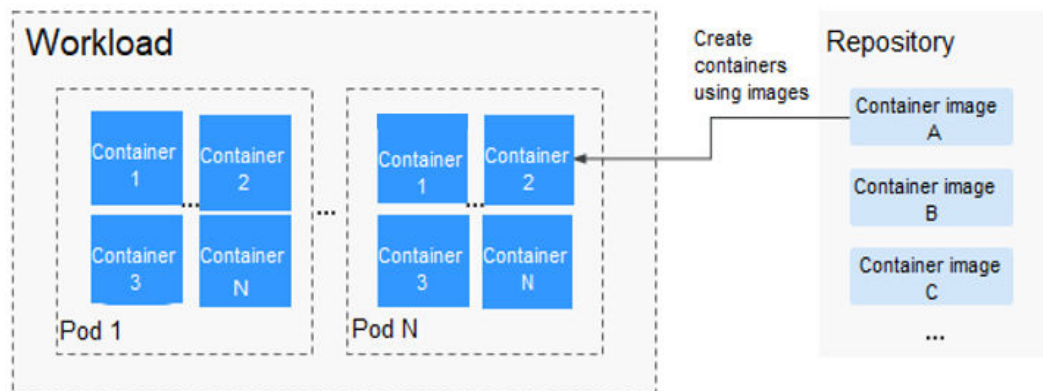


## Image

An image is a standardized 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. Its content remains unchanged after being built. 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.

Images become containers at runtime, that is, containers are created from images. Containers can be created, started, stopped, deleted, and suspended.

**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 provides a logical grouping mechanism where data within different namespaces is isolated from each other, while still allowing them to share basic resources like CPUs, memory, and storage within the same cluster. By deploying different environments in separate namespaces, such as development, testing, and production, you can ensure environment isolation and simplify management and maintenance tasks.

In Kubernetes, most resource objects are associated with a specific namespace (**default**), including pods, Services, ReplicationControllers, and Deployments. 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.

## Service

In Kubernetes, 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. It assigns a unique IP address to the Service within the cluster. This IP address is only accessible within the cluster and cannot be directly accessed from external networks. ClusterIP Services are typically used for internal communication 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. This type of Service allows external traffic to reach the Service by using the Elastic IP (EIP) associated with the node and the specified port.
- **LoadBalancer:** This type of Service uses the load balancer provided by cloud service providers to expose the Service to the Internet. An external load balancer can distribute traffic to the NodePort and ClusterIP Services within the cluster.
- **DNAT:** DNAT translates IP addresses for cluster nodes and enables multiple nodes to share an EIP. This enhances reliability as an EIP does not need to be bound to a single node. Any node failure does not affect access to the cluster.

## Layer-7 Load Balancing (Ingress)

An ingress is a set of routing rules for requests entering a cluster. It provides Services with URLs, load balancing, SSL termination, and HTTP routing for external access to the cluster.

## Network Policy

Network policies provide policy-based network control to isolate applications and reduce the attack surface. A network policy uses label selectors to simulate traditional segmented networks and controls traffic between them and traffic from outside.

## ConfigMap

A ConfigMap is used to store configuration data or configuration files as key-value pairs. ConfigMaps are similar to secrets, but they are specifically designed to handle non-sensitive string data in a more convenient manner.

## Secret

A secret manages sensitive data like passwords, tokens, and keys without exposing them in images or pod specifications. Secrets can be mounted to pods as volumes or injected to pods as environment variables.

## Label

In Kubernetes, a label is a key-value pair that is associated with a resource object like a pod, Service, or Deployment. 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 in Kubernetes simplify resource management and operations by allowing users to group and select resource objects based on their labels. This enables batch operations on the selected resource groups, such as traffic distribution, scaling, configuration updates, and monitoring.

## Annotation

Annotations are defined as key-value pairs and are 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 select resources for users.

On the other hand, annotations provide additional information defined by users. 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 PV is a storage resource in a cluster that can be either a local disk or network storage. It exists independently of pods, so even 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 PVs and PVCs is similar to that between pods and nodes. Just as pods consume resources from nodes, PVCs consume resources from PVs.

## Auto Scaling - HPA

Horizontal Pod Autoscaling (HPA) is a function that implements horizontal scaling of pods in Kubernetes. HPA enables a Kubernetes cluster to automatically scale in or out the number of pods based on CPU usage, memory usage, or other specified metrics. By setting thresholds for target metrics, HPA dynamically adjusts the pod count to ensure the best application performance.

## Affinity and Anti-Affinity

If an application is not containerized, multiple components of the application may run on the same virtual machine and processes communicate with each other. However, in the case of containerization, software processes are packed into different containers and each container has its own lifecycle. For example, the transaction process is packed into a container while the monitoring/logging process and local storage process are packed into other containers. If closely related container processes run on distant nodes, routing between them will be costly and slow.

- **Affinity:** Containers are scheduled onto the nearest node. For instance, when application A and application B have frequent interactions, it is important to use affinity to ensure that these two applications are placed in close proximity or even on the same node. By doing so, any potential performance degradation caused by slow routing can be avoided.
- **Anti-affinity:** Instances of the same application spread across different nodes to achieve higher availability. Once a node is down, instances on other nodes are not affected. For example, if an application has multiple replicas, it is necessary to use the anti-affinity feature to deploy the replicas on different nodes. In this way, no single point of failure will occur.

## Node Affinity

By selecting labels, you can schedule pods to specific nodes.

## Pod Affinity

You can deploy pods onto the same node to reduce consumption of network resources.

## Pod Anti-Affinity

You can deploy pods onto different nodes to reduce the impact of system breakdowns. Anti-affinity deployment is also recommended for workloads that may interfere with each other.

## 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. LimitRange is a feature used to apply resource limits to objects, like pods, within a namespace.

It offers several capabilities, including:

- Restricts the minimum and maximum resource usage for each pod or container in a namespace.
- Sets limits on the storage space that each PVC can request within a namespace.
- Controls the ratio between the request and limit for a resource within a namespace.
- Sets default requests and limits for compute resources within a namespace and automatically applies them to multiple containers during container execution.

## Environment Variable

An environment variable is a variable whose value can affect the way a running container will behave. A maximum of 30 environment variables can be defined at container creation time. You can modify environment variables even after workloads are deployed, increasing flexibility in workload configuration.

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 the apt command is to Ubuntu or what the yum command is to CentOS. Helm can quickly search for, download, and install charts.

Charts are a Helm packaging format. It describes only a group of related cluster resource definitions, not a real container image package. A Helm chart contains only 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 but 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.

Helm directly 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.

# 3 Containers and Kubernetes

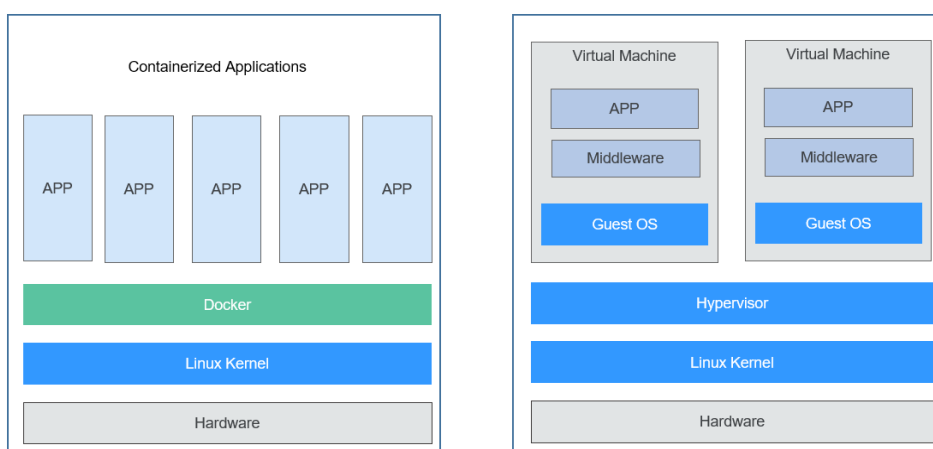
## 3.1 Containers

### Overview

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 is the first system that allows containers to be portable on different machines. It simplifies the packaging of both applications and the applications' repository and dependencies. Even an OS file system can be packaged into a simple portable package that can be used on any other machine that runs Docker.

Containers use similar resource isolation and allocation modes as VMs. The difference between containers and VMs lies in that containers virtualize OSs but not hardware, which makes containers more portable and efficient.

**Figure 3-1** Containers vs VMs



Containers have the following advantages over VMs:



- **Higher system resource utilization**  
With no overhead for virtualizing hardware and running a complete OS, containers outperform VMs in application execution speed, memory loss, and file storage speed. Therefore, with same configurations, containers can run more applications than VMs.
- **Faster startup**  
Traditional VMs usually take several minutes to start an application. However, Docker containerized applications run directly on the host kernel with no need to start the entire OS, so they can start within seconds or even milliseconds, greatly saving your time in development, testing, and deployment.
- **Consistent running environments**  
Inconsistent development, test, and production environments are a common issue in development. As a result, some issues cannot be detected prior to rollout. A Docker container image includes everything (code, runtime, system tools, system libraries, and settings) needed to run an application to ensure consistency in application running environments.
- **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, 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 and these layers are stacked. When you create a new container, you add a container layer on top of image layers. In this way, duplicate layers are reused, which simplify application maintenance and update as well as further 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 custom your images based on these images. This greatly improves the efficiency in creating images for applications.

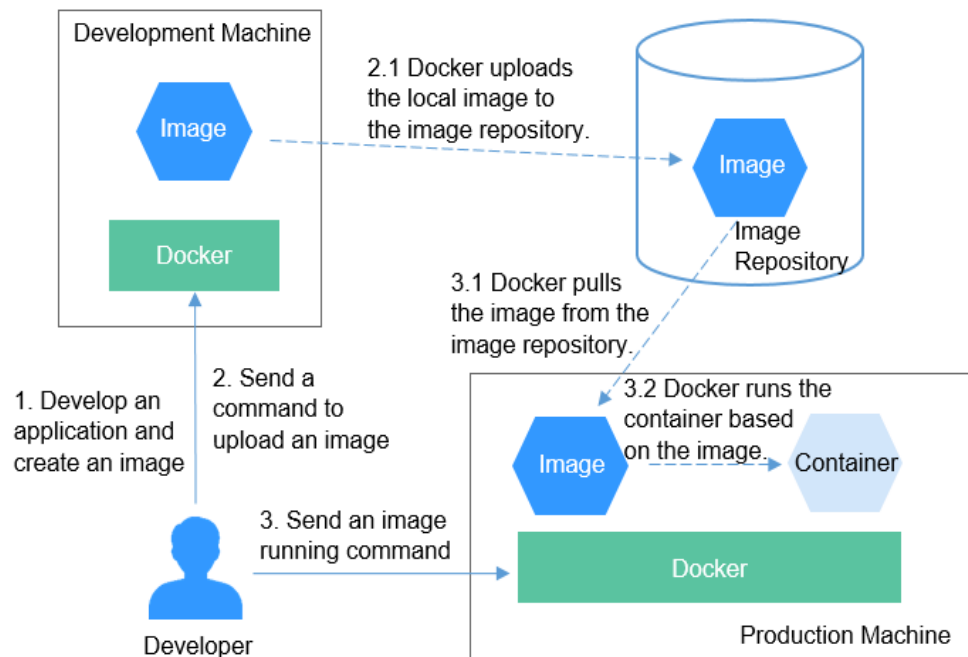
## Typical Process of Using Docker Containers

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

- **Image:** A Docker image is an executable package of software that includes the data needed to run an application, such as code, runtime, file systems, and executable file path of the runtime.
- **Image repository:** A Docker image repository stores Docker images and entities to create and share container images. You can run an image on the computer where it is edited, or you can upload it to an image repository, download it to another computer, and then run it. Some repositories are public, which allow everyone to pull images from them. Some are private, which are accessible only to some users and machines.
- **Container:** A Docker container is usually a Linux container created from a Docker image. A running container is a process running on the Docker host, but it is isolated from the host and all other processes running on the host. The process is also resource-limited, and it can access and use only resources (such as CPUs and memory) allocated to it.

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

Figure 3-2 Typical process of using Docker containers



1. A developer develops an application and creates an image on the development machine.  
Docker creates the image and stores it on the machine.
2. The developer sends a command to Docker for uploading the image.  
After receiving the command, Docker uploads the local image to the image repository.
3. The developer sends an image running command to the production machine.  
After the command is received, Docker pulls the image from the image repository to the machine and then runs a container based on the image.

## Example

In the following example, Docker packages a container image based on 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 operating systems. Select a Docker version that best suits your needs.

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

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. You are advised to install Docker 18.06.0 to 24.0.9 to facilitate the configuration of the image accelerator..

```
sudo yum install docker-ce-24.0.9 docker-ce-cli-24.0.9 containerd.io
```

Docker 24.0.9 is being used as an example. If you choose a different version, simply substitute 24.0.9 with the specific version number.

6. Start Docker:

```
systemctl enable docker # Set Docker to start automatically upon system startup.  
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. Dockerfile allows you to customize a simple Nginx image.


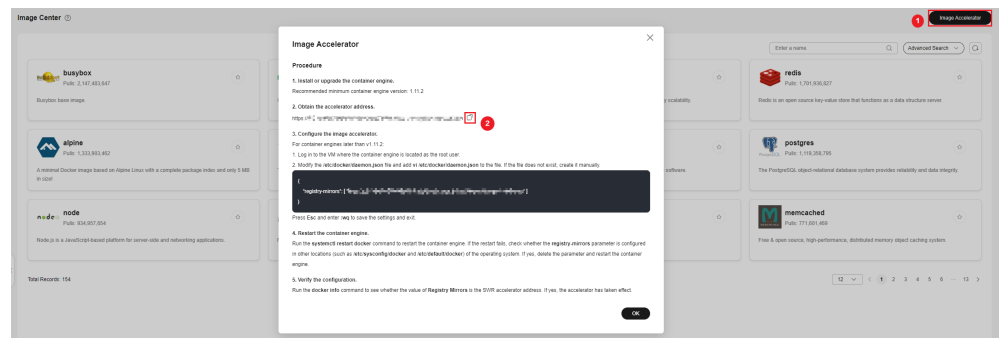
1. To configure an image accelerator, perform the following operations. (Image accelerators can speed up the download of popular open source images, addressing issues with slow or failed downloads from third-party repositories like Docker Hub caused by network problems. These accelerators are available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

Figure 3-3 Copying an accelerator address



- 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. After the configuration is complete, restart the container engine:
- ```
systemctl restart docker
```

If the restart fails, check whether the **registry-mirrors** parameter is configured in other locations within the OS, such as `/etc/sysconfig/docker` and `/etc/default/docker`. If it is, delete the parameter and restart the container engine.

- f. View the Docker details:

```
docker info
```

If the value of the **Registry Mirrors** field 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 the Dockerfile:

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

# Run a command to modify index.html of the Nginx image.
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 .
```

In the preceding command, **-t** is used to add a tag 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 built.

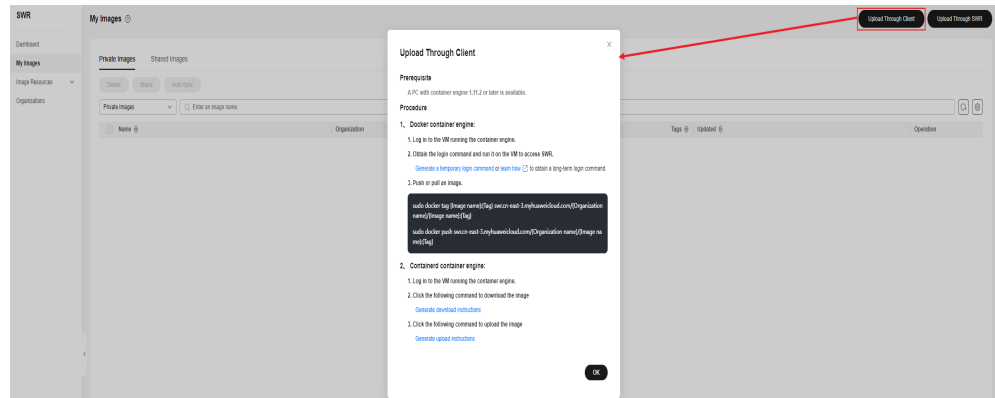
docker images

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

| 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 SWR.



2. Before uploading an image, specify a complete name for the image.  
`docker tag hello swr.cn-east-3.myhuaweicloud.com/container/hello:v1`

In the preceding command, the parameters 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 organization is available, an organization will be automatically created when an image is uploaded to SWR for the first time. Each organization name is unique in a single region.
  - **v1** is the version 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

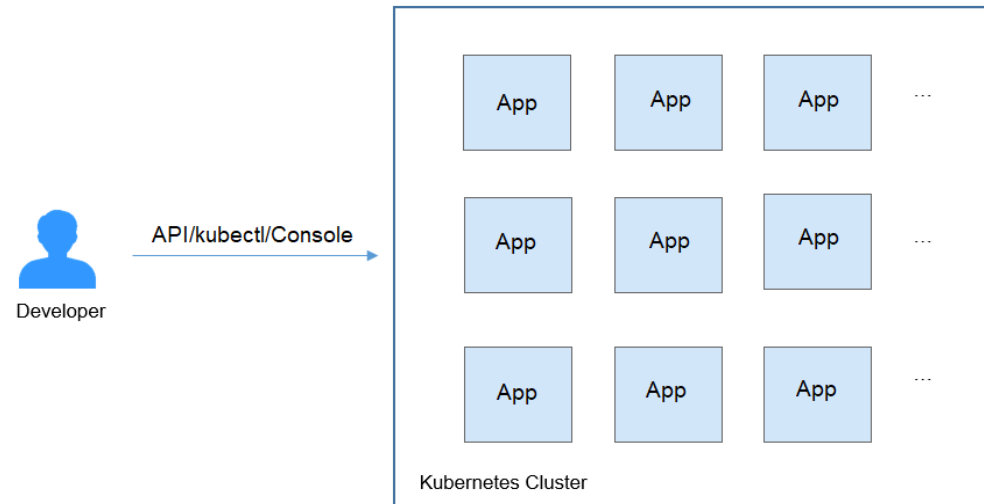
### What Is Kubernetes?

**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 you run a large number of servers as one on which your applications run. Kubernetes enables you to deploy applications always using the same method, regardless of the number of servers in a cluster.

**Figure 3-4** Running applications in a Kubernetes cluster



## Kubernetes Cluster Architecture

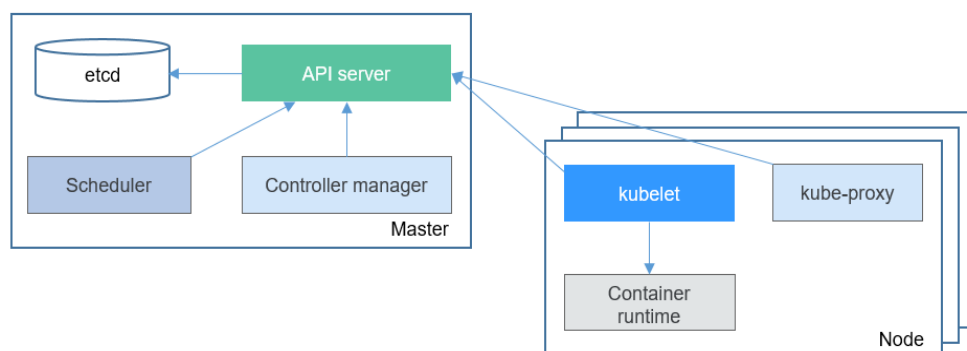
A Kubernetes cluster consists of master nodes (Masters) and worker nodes (Nodes). Applications are deployed on worker nodes, and you can specify the nodes for deployment.

### NOTE

For CCE clusters, master nodes are hosted 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 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 into etcd.

- Controller manager: carries out cluster-level functions, such as component replication, node tracing, and node fault fixing.
- Scheduler: schedules containers to nodes based on various conditions (such as available resources and node affinity).
- etcd: 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 for your CCE cluster.

### **Worker node**

A worker node is a compute node for running 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 containers on the node.
- kube-proxy: an access proxy between application components.
- Container runtime: an engine such as Docker software for downloading images and running containers.

## **Kubernetes Scalability**

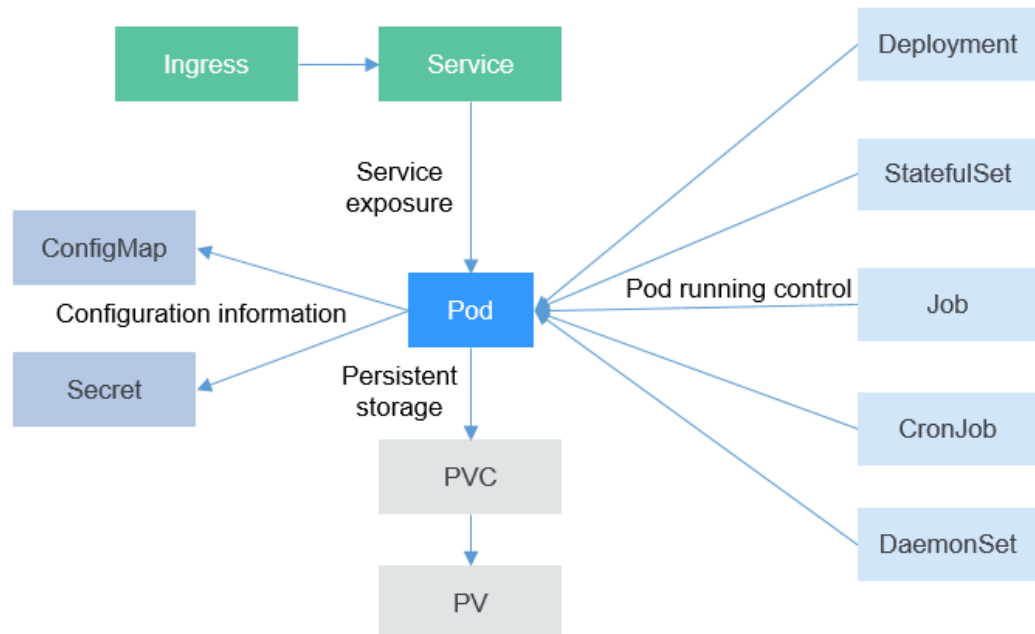
Kubernetes makes the Container Runtime Interface (CRI), Container Network Interface (CNI), and Container Storage Interface (CSI) open sourced. These interfaces maximize Kubernetes scalability and allow Kubernetes to focus on container scheduling.

- CRI: provides computing resources for a container runtime. It shields differences between container engines 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 describes the basic objects in Kubernetes and the relationships between them.

**Figure 3-6** Basic Kubernetes objects



- **Pod**  
Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. Each pod has a separate IP address.
- **Deployment**  
A Deployment manages a set of pods to run an application workload. It can contain one or more pods. Each pod has the same role, and the system automatically distributes requests to the pods of a Deployment.
- **StatefulSet**  
A StatefulSet is used to manage stateful applications. Like Deployments, StatefulSets manage a group of pods based on an identical container spec. Where they differ is that StatefulSets maintain a fixed ID for each of their pods. These pods are created based on the same declaration but cannot replace each other. Each pod has a permanent ID regardless of how it is scheduled.
- **Job**  
A job is used to control batch tasks. Jobs are different from long-term servo tasks (such as Deployments). The former can be started and terminated at specific time, while the latter runs unceasingly unless it is terminated. The pods managed by a job will be automatically removed after successfully completing tasks based on user configurations.
- **Cron job**  
A cron job is a time-based job. Similar to the crontab of 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 based on Layer 4 TCP and UDP protocols. Ingresses can forward requests based on Layer 7 HTTPS and HTTP protocols and make forwarding more targeted by domain names and paths.
- **ConfigMap**  
A ConfigMap stores configurations in key-value pairs required by applications. ConfigMaps allow for the separation of configurations, enabling different environments to 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**  
Kubernetes provides PVCs to apply for persistent storage. With PVCs, you only need to specify the type and capacity of storage without concerning about how to create and release underlying storage resources.

## Setting Up a Kubernetes Cluster

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

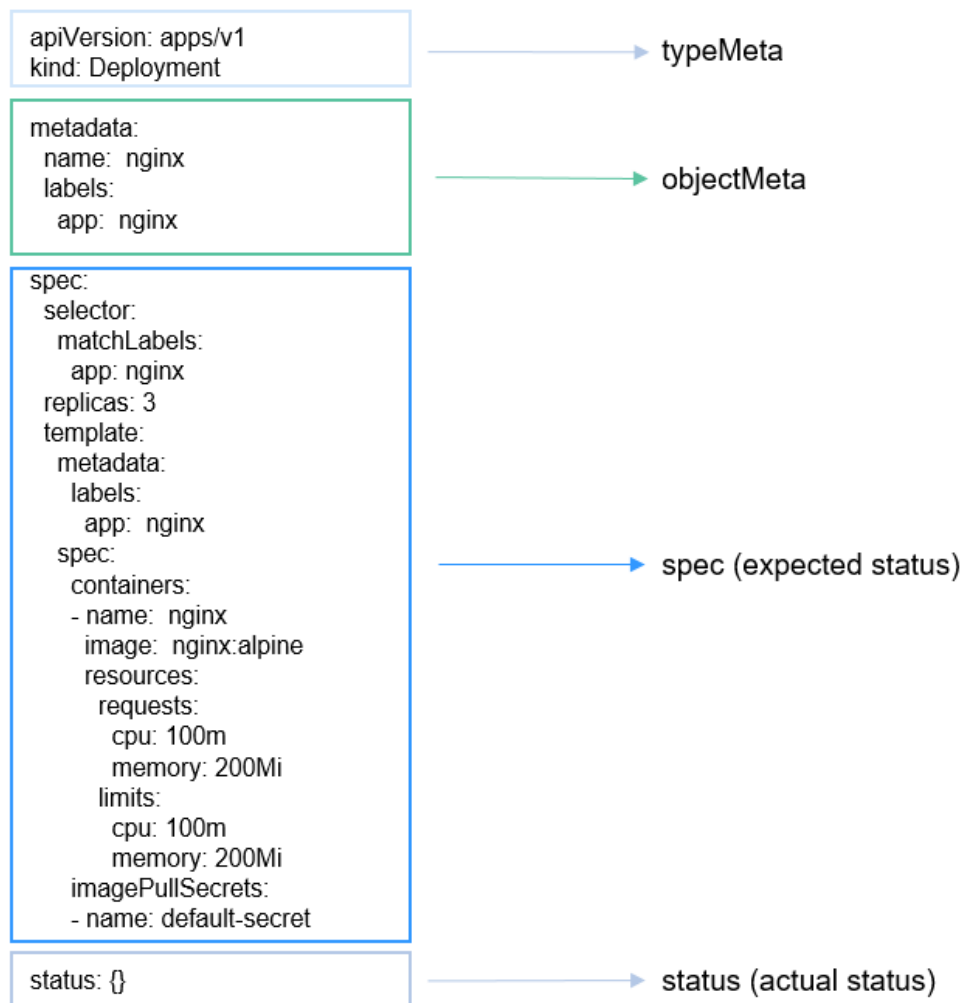
If you do not want to create a Kubernetes cluster by coding, you can create one on the CCE console. The following sections use clusters created on the CCE console as examples.

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

Figure 3-7 YAML description file



## 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
```

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

## 3.3 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 [Connecting to a Cluster Using `kubectl`](#). After connection, 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](#).

## Getting Started

### `get`

The **get** command is used to obtain details about one or more resources in a cluster.

This command prints a table of the most important information about all resources, including cluster nodes, running pods, Deployments, and Services.

---

**NOTICE**

Many 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 all pods with detailed information:

```
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 list 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 the information of pods in the YAML format:

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

To obtain the information 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 is used to create cluster resources based on files 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 exposes a resource as a new Kubernetes service. Possible resources include a pod, Service, and Deployment.

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

#### NOTE

The command above creates a Service for the Deployment. The **--port** parameter specifies the port that the Service will expose, the **--type** parameter determines the type of the Service, and the **--target-port** parameter specifies the port of the backend pod associated with the Service. Visiting *ClusterIP.port* allows you to access the applications in the cluster.

### run

The **run** command runs a particular image in the cluster.

Example:

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

To run a particular image using a specified command:

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

### set

The **set** command configures object resources.

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 edits a resource from the default editor.

Example:

To update a pod:

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

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

```
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 views documents or reference documents.

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 with minimal delay:

```
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 particular deployment:

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

To view the rollout history of a particular deployment:

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

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

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

### scale

The **scale** command sets a new size for a resource by adjusting the number of resource replicas.

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

### autoscale

The **autoscale** command automatically adjusts the number of replicas based on the CPU usage of workloads. The **autoscale** command allows you to define a range of replicas for a workload (such as Deployment, ReplicaSet, StatefulSet, or ReplicationController). During running, the pods will be automatically scaled in or out within this range based on the average CPU usage of all pods. If the target usage is not specified or the parameter 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

### cordons, drains, uncordons\*

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 **cordons** command to mark a node as unschedulable. This means that new pods will not be scheduled to that node.

```
kubectl cordon <nodename>
```

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

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

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

By using **--ignore-daemonsets**, the DaemonSet pods will be ignored. Additionally, **--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, such as resetting the node.

**Step 4** After node maintenance is completed, 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 dump current cluster information to stdout:

```
kubectl cluster-info dump
```

### top\*

The **top** command shows the usage of resources like CPU, memory, and storage in a cluster. Ensure that the Kubernetes Metrics Server is running properly for this command to work.

### taint\*

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

### certificate\*

The **certificate** command modifies the 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 detailed information about a resource, while the **describe** command provides status information about a resource in a cluster. The **describe** command is similar to the **get** command, but it does not support the **-o** option. For resources of the same type, the **describe** command provides the same output format and content.

#### NOTE

If you need specific information about a resource, you can use the **get** command for more detailed information. On the other hand, 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 detailed status information.

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

### 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 handling multiple containers within a pod, you can use the **-c** option to specify the desired container.

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

### port-forward\*

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

Example:

To listen to local port 5000 and forward it to port 6000 in 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 remote pod.

### auth\*

The **auth** command inspects authorization.

### attach\*

The **attach** command is similar to the **logs -f** command and attaches to a process that is already running inside an existing container. 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 original 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 compared to the **patch** and **edit** commands. It allows you to maintain resource configurations in source control. When an update occurs, 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, updating only the changed parts. The **apply** command works similarly to the **replace** command, but it does not delete the original resources and 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 without first deleting the container or using the **replace** command, the **patch** command is to the rescue. The **patch** command updates field(s) of a resource using strategic merge patch, a JSON merge patch, or a JSON patch. 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 update labels on a resource.

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

### annotate

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

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

### completion

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

## 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 kubeconfig files. An example use case of this command is 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 for the current context.

```
kubectl version
```

# 4 Pods, Labels, and Namespaces

---

## 4.1 Pod: the Smallest Scheduling Unit in Kubernetes

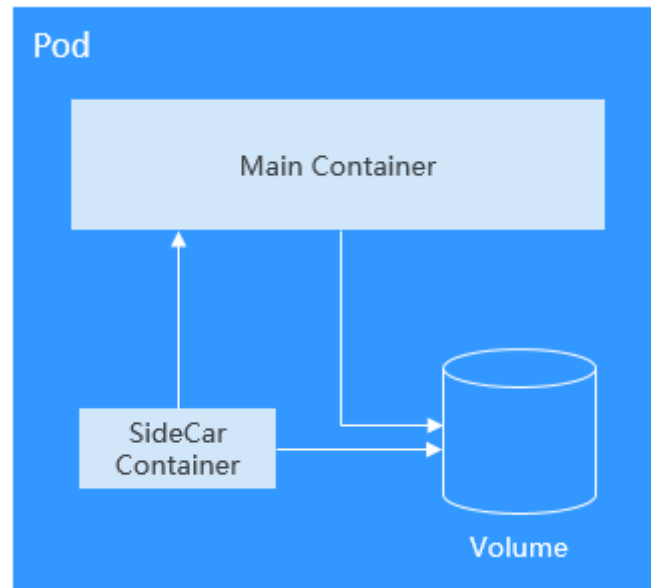
### Overview of Pod

A pod is the smallest, simplest unit in the Kubernetes object model that you create or deploy. A pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. Each pod has a separate IP address.

Pods can be used in either of the following ways:

- A pod runs only one container. This is the most common usage of pods in Kubernetes. You can consider a pod as a container, but Kubernetes directly manages pods instead of containers.
- A pod runs multiple containers that need to be tightly coupled. In this scenario, a pod contains a main container and several sidecar containers, as shown in [Figure 4-1](#). For example, the main container is a web server that provides file services from a fixed directory, and sidecar containers periodically download files to this fixed directory.

**Figure 4-1** Pod running multiple containers



In Kubernetes, pods are rarely created directly. Instead, Kubernetes controller manages pods through pod instances such as Deployments and jobs. A controller typically uses a pod template to create pods. The controller can also manage multiple pods and provide functions such as replica management, rolling upgrade, and self-healing.

## Creating a Pod

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

```

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

As shown in the YAML comments, the YAML file includes:

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

If you check a Kubernetes resource, you can also see the **status** field, which indicates the status of the Kubernetes resource. This field does not need to be set

when the resource is created. This example is a minimum set of parameters. Other parameters will be described later.

After defining the pod, you can use `kubectl` to create the pod. Assume that the preceding YAML file is named **nginx.yaml**, run the following command to create the pod. **-f** indicates that you will create the pod from a file.

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

After the pod is created, you can run the **kubectl get pods** command to obtain the pod status.

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

The preceding command output indicates that the **nginx** pod is in the **Running** state. **READY** is **1/1**, indicating that this pod has one container that is in the **Ready** state.

You can run the **kubectl get** command to obtain the information of a pod. **-o yaml** indicates that the information is returned in YAML format, and **-o json** indicates that the information is returned in JSON format.

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

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

```
$ kubectl describe pod nginx
```

Before deleting a pod, Kubernetes terminates all the containers that are part of that pod. Kubernetes sends a **SIGTERM** signal to the containers' main process and waits a period (30 by default) for it to shut down gracefully. If the process is not shut down during this period, Kubernetes will send a **SIGKILL** signal to stop the process.

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

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

Delete multiple pods at one time:

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

Delete all pods:

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

Delete pods by labels. For details about labels, see [Label for Managing Pods](#).

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

## Environment Variables

You can use environment variables to set up a container runtime environment.

Environment variables add flexibility to configuration. The custom environment variables will take effect when the container is running. This frees you from rebuilding the container image.

In the following shows example, 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:
      # Environment variable
      - name: env_key
        value: env_value
  imagePullSecrets:
  - name: default-secret
```

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

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

Pods can use **ConfigMap** and **secret** as environment variables. For details, see [Referencing a ConfigMap as an Environment Variable](#) and [Referencing a Secret as an Environment Variable](#).

## Setting Container Startup Commands

Starting a container is to start its main process. You need to make some preparations before starting a main process. For example, you may need to configure or initialize MySQL databases before running MySQL servers. All of these operations can be performed by configuring **ENTRYPOINT** or **CMD** in a Dockerfile during image creation. As shown in the following example, configure the **ENTRYPOINT ["top", "-b"]** command in the Dockerfile. Then, the system will automatically perform the preparation operations during container startup.

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

When calling an API, you only need to configure pods' `containers.command` field to define the command and their arguments. The first parameter is the command and the following parameters 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:          # Boot command
- top
- "-b"
imagePullSecrets:
- name: default-secret

```

## Container Lifecycle

Kubernetes provides **container lifecycle hooks** to enable containers to be aware of events in their management lifecycle and run code implemented in a handler when the corresponding lifecycle hook is executed. For example, if you want a container to perform a certain operation before it is stopped, you can register a hook. The following lifecycle hooks are provided:

- **postStart**: triggered immediately after a pod is started
- **preStop**: triggered immediately before a pod is stopped

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

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    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

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

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

Supported detection rules are as follows:

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

In addition to liveness probes, readiness probes are also available for you to detect pod status. For details, see [Readiness Probes](#).

## HTTP GET

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

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

Create pod **liveness-http**.

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

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

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

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

```



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

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

## TCP Socket

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

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

## Exec

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

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

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

## Advanced Settings of a Liveness Probe

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

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

This is the detailed configuration of the liveness probe.

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

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

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

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

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

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

## Configuring a Liveness Probe

- **What to check**

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

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

- **To be lightweight**

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

## 4.3 Label for Managing Pods

### Overview

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

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

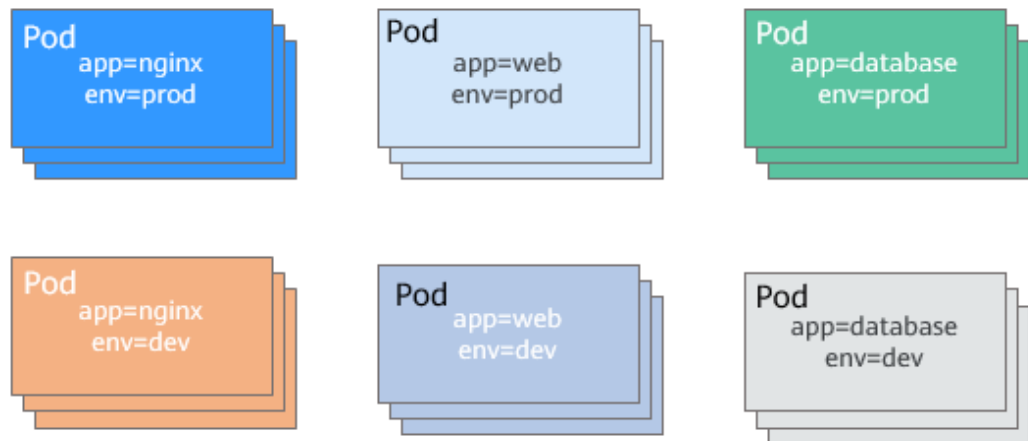
The following figures show how labels work. Assume that you have multiple pods of various kinds. It could be challenging when you manage them.

**Figure 4-2** Pods without classification



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

**Figure 4-3** Pods classified using labels



## Adding a Label

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

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:           # Add 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 you add labels to a pod, you can view the labels by adding **--show-labels** when querying the pod.

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

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

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

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

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

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

## Modifying a Label

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

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

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

## 4.4 Namespace for Grouping Resources

### Overview

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

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

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

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

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

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

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

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

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

## Creating a Namespace

Define a namespace.

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

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

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

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

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

Create resources in the namespace.

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

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

## The Isolation function of Namespaces

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

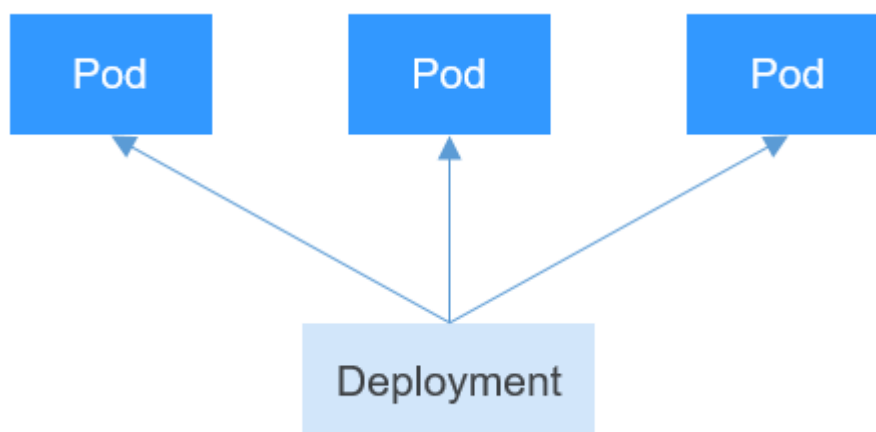
# 5 Pod Orchestration and Scheduling

## 5.1 Deployments

### Overview of Deployment

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

**Figure 5-1** Relationship between a Deployment and pods



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

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

## Creating a Deployment

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

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

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

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

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

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

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

## How Does the Deployment Control Pods?

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
```

If you delete a pod, a new pod is immediately created. As mentioned above, the Deployment ensures that there are two pods running. If a pod is deleted, the



Deployment creates a new pod. If a pod becomes faulty, the Deployment automatically restarts the pod.

```
$ kubectl 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
```

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

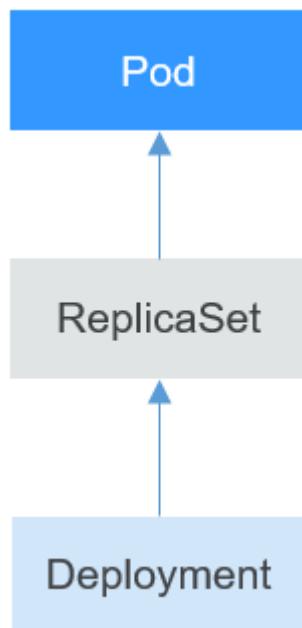
You may notice that the two suffixes share the same content **7f98958cdf** in the first part. This is because the Deployment does not control the pods directly, but through a controller named ReplicaSet. You can run the following command to obtain the ReplicaSet, where **rs** is the abbreviation of ReplicaSet:

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

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

As shown in **Figure 5-2**, the Deployment controls the ReplicaSet, which then controls pods.

**Figure 5-2** How does the Deployment control a pod



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

```
$ kubectl describe deploy nginx
Name:                nginx
```

```

Namespace:      default
CreationTimestamp: Sun, 16 Dec 2018 19:21:58 +0800
Labels:        app=nginx
...
NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)
Events:
  Type     Reason          Age   From              Message
  ----     -
  Normal   ScalingReplicaSet 5m    deployment-controller  Scaled up replica set nginx-7f98958cdf to 2
    
```

## Upgrade

In real-world applications, upgrading is a common occurrence. Deployment effortlessly facilitates application upgrades.

You can set different upgrade policies for a Deployment:

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

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

```

$ kubectrl edit deploy nginx

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

$ kubectrl get pods
NAME                READY  STATUS   RESTARTS  AGE
nginx-6f9f58dff-tdmqk 1/1    Running  0         1m
nginx-6f9f58dff-tesqr 1/1    Running  0         1m
    
```

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

```

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

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

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

## Rollback

Rollback is to roll an application back to the earlier version when a fault occurs during the upgrade. Applications that run in Deployments can be easily rolled back to the earlier version.

For example, if the image of an upgraded Deployment is faulty, run the `kubectl rollout undo` command to roll back the Deployment.

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

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

## 5.2 StatefulSets

### Overview of StatefulSet

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

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

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

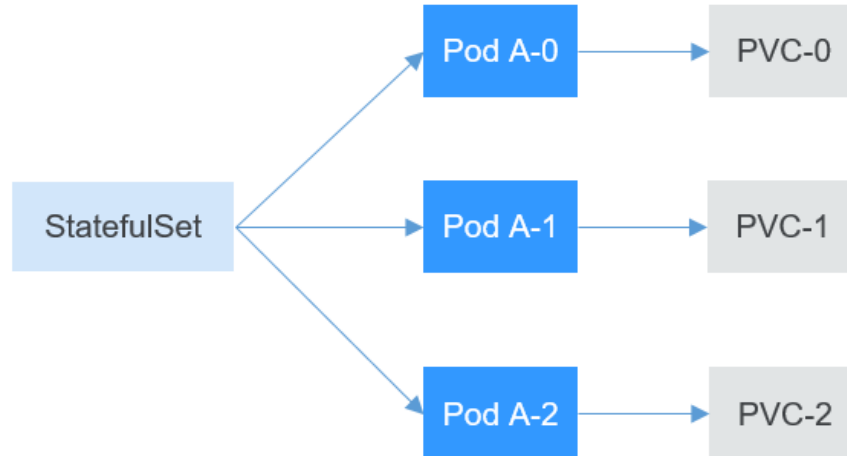
- Each pod must have a fixed identifier so that it can be recognized by other pods.
- Separate storage resources must be configured for each pod. In this way, the original data can be retrieved after a pod is deleted and restored. Otherwise, the pod status will be changed after the pod is rebuilt.

To address the preceding requirements, Kubernetes provides StatefulSets.

1. StatefulSets provide a fixed name for each pod following a fixed number ranging from 0 to N. After a pod is rescheduled, the pod name and the hostname remain unchanged.
2. StatefulSets use a headless Service to allocate a fixed domain name for each pod.

- StatefulSets create PVCs with fixed identifiers to ensure that pods can access the same persistent data after being rescheduled.

Figure 5-3 StatefulSet



## Creating a Headless Service

A **headless Service** is required by a StatefulSet for accessing pods.

Use the following file to describe the headless Service:

- spec.clusterIP:** must be set to **None** to indicate a headless Service.
- spec.ports.port:** number of the port for communication between pods.
- spec.ports.name:** 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 # Name of the port for communication between pods
      port: 80 # Number of the port for communication between pods
  selector:
    app: nginx # Select the pod labeled with app:nginx.
  clusterIP: None # Set this parameter to None, indicating a headless Service.
  
```

Run the following command to create a 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 StatefulSets is basically the same as that of other objects. The differences are as follows:

- **serviceName** specifies the headless Service used by a StatefulSet. You are required to configure this parameter.
- **volumeClaimTemplates** is used to apply for a **PVC**. A template named **data** is defined, which will create a PVC for each pod. **storageClassName** specifies the persistent StorageClass. For details, see [PersistentVolumes, PersistentVolumeClaims, and StorageClasses](#). **volumeMounts** specifies storage to mount to pods. If no storage is required, you can delete the **volumeClaimTemplates** and **volumeMounts** fields.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                # Name of the headless Service
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:              # Storage to be mounted to the pod
            - 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      # Persistent StorageClass
```

Run the following command to create the StatefulSet:

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

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

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

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

Manually delete the **nginx-1** pod and check the pods again. It is found that a pod with the same name is created. According to **5s** under **AGE**, the **nginx-1** pod is newly created.

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

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

Access pods and check their hostnames, which are **nginx-0**, **nginx-1**, and **nginx-2**.

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

Check the PVCs created by the StatefulSet. These PVCs are named in the format of "PVC name-StatefulSet name-No." and are in the **Bound** state.

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

## Network Identifier of a StatefulSet

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

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

Create a pod from the **tutum/dnsutils** image. Then, access the container of the pod and run the **nslookup** command to view the domain name of the pod. The IP address of the pod can be parsed. The IP address of the DNS server is **10.247.3.10**. When a CCE cluster is created, the CoreDNS add-on is installed by default to provide the DNS service. For details, see [Kubernetes Networking](#).

```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
```

```
Server: 10.247.3.10
Address: 10.247.3.10#53
Name: nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

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

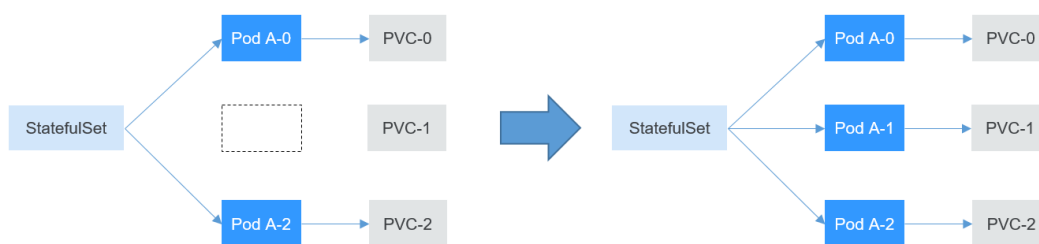
/ # nslookup nginx-2.nginx
Server: 10.247.3.10
Address: 10.247.3.10#53
Name: nginx-2.nginx.default.svc.cluster.local
Address: 172.16.0.19
```

Manually delete the two pods, check the IP addresses of the pods re-created by the StatefulSet, and run the **nslookup** command to resolve the domain names of the pods. You can still get **nginx-0.nginx** and **nginx-1.nginx**. This ensures that the network identifier of the StatefulSet remains unchanged.

### StatefulSet Storage Status

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

**Figure 5-4** Process for a StatefulSet to re-create a pod



After the Pod A-1 is deleted and recreated, the PVC-1 is rebound to the Pod A-1.

Write data into the **/usr/share/nginx/html** directory of **nginx-1**, for example, modify the content of **index.html** to **hello world** by running the following command:

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

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

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

Manually delete the **nginx-1** pod and check the pods again. It is found that a pod with the same name is created. According to **4s** under **AGE**, the **nginx-1** pod is newly created.

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

# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
```

```
nginx-0 1/1 Running 0 14m
nginx-1 1/1 Running 0 4s
nginx-2 1/1 Running 0 13m
```

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

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

## 5.3 Jobs and CronJobs

### Overview of Job and CronJob

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

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

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

### Creating a Job

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

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  completions: 50          # Number of pods that need to run successfully to end the job
  parallelism: 5          # Number of pods that run concurrently. The default value is 1.
  backoffLimit: 5        # Maximum number of retries performed if a pod fails. When the limit is reached,
  it will not try again.
  activeDeadlineSeconds: 100 # Timeout interval of pods. Once the time is reached, all pods of the job are
  terminated.
  template:               # Pod definition
    spec:
      containers:
        - name: pi
          image: perl
          command:
            - perl
            - "-Mbignum=bpi"
            - "-wle"
            - print bpi(2000)
          restartPolicy: Never
```

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



**Table 5-1** Job types

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

## Creating a CronJob

Compared with a job, 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 * * * *"      # Configuration of a scheduled job
  jobTemplate:                        # Job definition
    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

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

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

For details about the CronJob format, visit <https://en.wikipedia.org/wiki/Cron>.

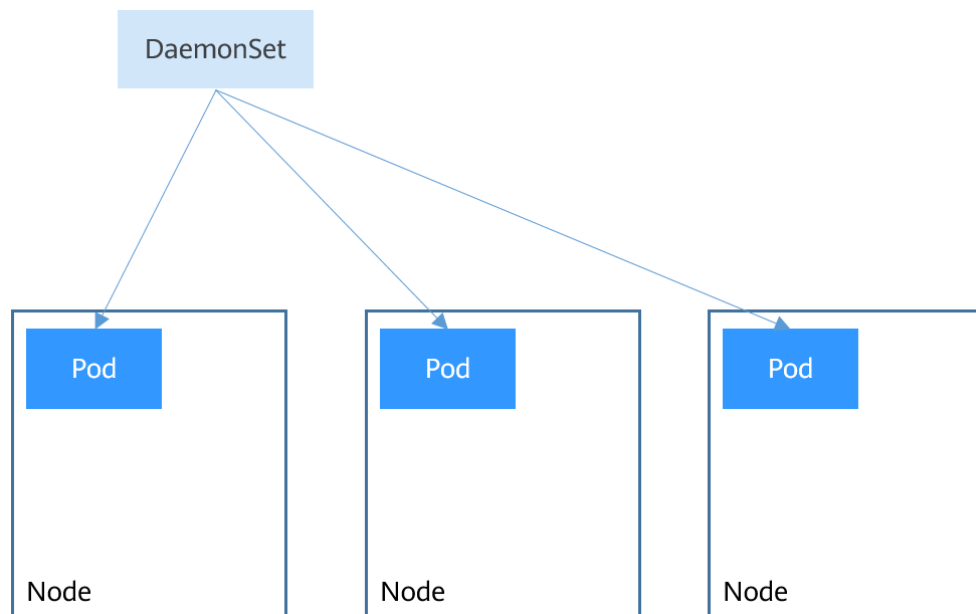
## 5.4 DaemonSets

### 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 and need only a few pods. A good example is kube-proxy.

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

Figure 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:
```

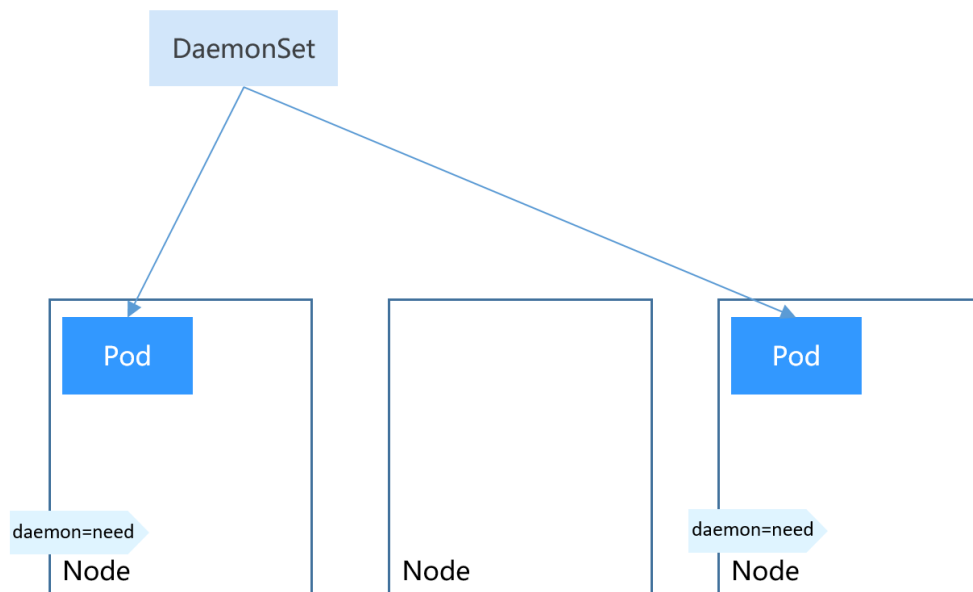
```

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

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

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

**Figure 5-6** DaemonSet creating a pod on nodes with a specified label



Create a DaemonSet.

```

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 5.5 Affinity and Anti-Affinity Scheduling

A nodeSelector provides a very simple way to constrain pods to nodes with specific labels, as mentioned in [DaemonSets](#). Affinity and anti-affinity expands the types of constraints you can define.

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

## Node Affinity

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

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

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

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

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

When you deploy pods, you can use a nodeSelector, as described in [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:      # Node selection. A pod is deployed on a node only when the node is labeled
                    with gpu=true.
                    gpu: true
  ...
```

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

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

```

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

```

Even though the node affinity rules require more lines of code, they are more expressive, which will be further described later.

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

- **requiredDuringScheduling** indicates that pods can be scheduled to the node only when all the defined rules are met (**required**).
- **IgnoredDuringExecution** indicates that pods already running on the node do not need to meet the defined rules. If a label 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 **In**, indicating that the label value must be in the **values** list. Other available **operator** values are as follows:

- **NotIn**: The label value is not in a list.
- **Exists**: A specific label exists.
- **DoesNotExist**: A specific label does not exist.
- **Gt**: The label value is greater than a specified value (string comparison).
- **Lt**: The label value is less than a specified value (string comparison).

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

Now, check whether the node affinity rule takes effect. Add the **gpu=true** 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. You can find that all pods are deployed on the **192.168.0.212** node.

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

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

## Node Preference Rule

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

To demonstrate its effect, add a node in a different AZ from other nodes to the cluster. Then, check the AZ of the node. As shown in the following output, the newly added node is in **cn-east-3c**.

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

Define a Deployment. Use the **preferredDuringSchedulingIgnoredDuringExecution** rule to set the weight of nodes in **cn-east-3a** 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"
```

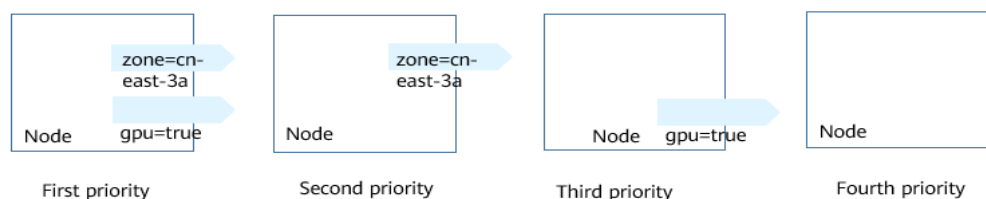
After the Deployment is created, you can find that five pods are deployed on the **192.168.0.212** node, and two pods are deployed on the **192.168.0.100** node.

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

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

In the preceding example, the node with both **cn-east-3a** and **gpu=true** labels has the first (highest) priority, the node with only the **cn-east-3a** label has the second priority (weight: 80), the node with only the **gpu=true** label has the third priority, and the node without any of these two labels have the fourth (lowest) priority.

**Figure 5-7** Scheduling priority



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

## Workload Affinity (podAffinity)

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



**requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution.**

**NOTE**

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

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

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

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - image: nginx:alpine
          name: frontend
          resources:
            requests:
              cpu: 100m
              memory: 200Mi
            limits:
              cpu: 100m
              memory: 200Mi
      imagePullSecrets:
        - name: default-secret
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - topologyKey: kubernetes.io/hostname
              labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - backend
```

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

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

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

```
frontend-67ff9b7b97-hxm5t 1/1 Running 0 6s 172.16.0.71 192.168.0.100
frontend-67ff9b7b97-z8pdb 1/1 Running 0 6s 172.16.0.72 192.168.0.100
```

The **topologyKey** field is used to divide topology keys to specify the selection range. If the label keys and values of nodes are the same, the nodes are considered to be in the same topology key. Then, the contents defined in the following rules are selected. The effect of **topologyKey** is not fully demonstrated in the preceding example because all the nodes have the **kubernetes.io/hostname** label, that is, all the nodes are within the range.

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

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

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

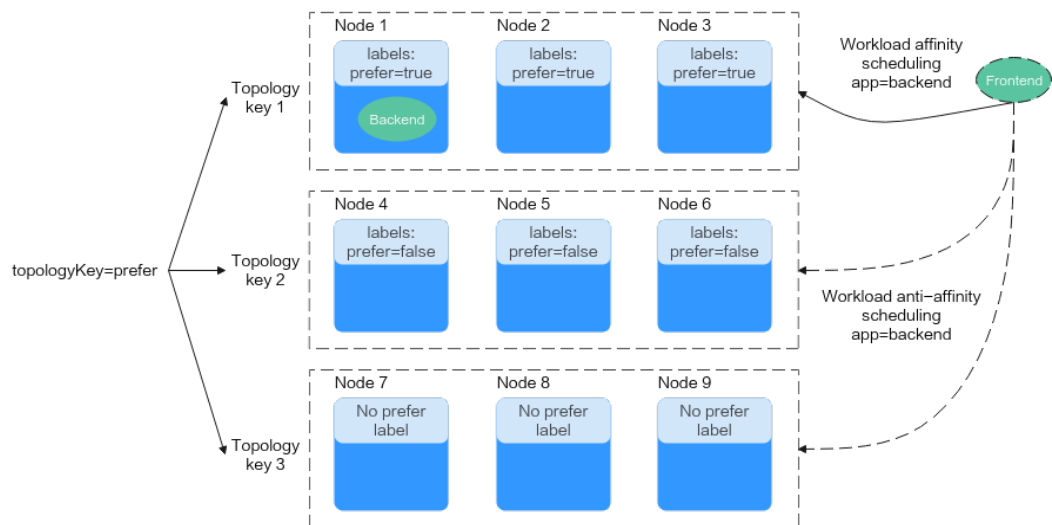
```
$ kubectl label node 192.168.0.97 prefer=true
node/192.168.0.97 labeled
$ kubectl label node 192.168.0.94 prefer=true
node/192.168.0.94 labeled

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

If the **topologyKey** of **podAffinity** is set to **prefer**, the node topology keys 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 keys



During scheduling, node topology keys are divided based on the **prefer** label. In this example, **192.168.0.97** and **192.168.0.94** are divided into the same topology key. If a pod with the **app=backend** label runs in the topology key, even if not all nodes in the topology key run the pod with the **app=backend** label (in this example, only the **192.168.0.97** node has such a pod), **frontend** is also deployed in this topology key (**192.168.0.97** or **192.168.0.94**).

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

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

## Workload Anti-Affinity (podAntiAffinity)

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

### NOTE

For workload anti-affinity, when `requiredDuringSchedulingIgnoredDuringExecution` is used, the default access controller `LimitPodHardAntiAffinityTopology` of Kubernetes requires that `topologyKey` can only be **kubernetes.io/hostname**. To use other custom topology logic, modify or disable the access controller.

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

```
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 key of the 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 keys 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 key. If a topology key contains only one node where a frontend pod already exists, pods with the same label will not be scheduled to that topology key. 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

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

ConfigMaps allow for the separation of configurations, enabling different environments to 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 and in volumes.

In the following example, **property\_1** of **configmap-test** is used as the value of the environment variable **EXAMPLE\_PROPERTY\_1**. After the container is started, it will reference the value of **property\_1** as the value of **EXAMPLE\_PROPERTY\_1**, that is, **Hello**.

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

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

## Referencing a ConfigMap in a Volume

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

In the following example, create a volume named **vol-configmap**, reference the ConfigMap named **configmap-test** in the volume, and mount the volume to the **/tmp** directory of the container. After the pod is created, the two files **property\_1** and **property\_2** are generated in the **/tmp** directory of the container, and the values are **Hello** and **World**.

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

## 6.2 Secrets

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

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

## Base64 Encoding

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

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

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

## Creating a Secret

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

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

## Referencing a Secret as an Environment Variable

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

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

## Referencing a Secret in a Volume

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

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

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

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



# 7 Kubernetes Networking

---

## 7.1 Container Networking

Kubernetes is not responsible for network communication, but it provides the Container Networking Interface (CNI) for networking through CNI plug-ins. There are many open-source CNI plug-ins, such as Flannel and Calico. CCE offers various network add-ons for clusters of different network models, enabling seamless network communication within clusters.

According to Kubernetes, cluster networking must meet the following requirements:

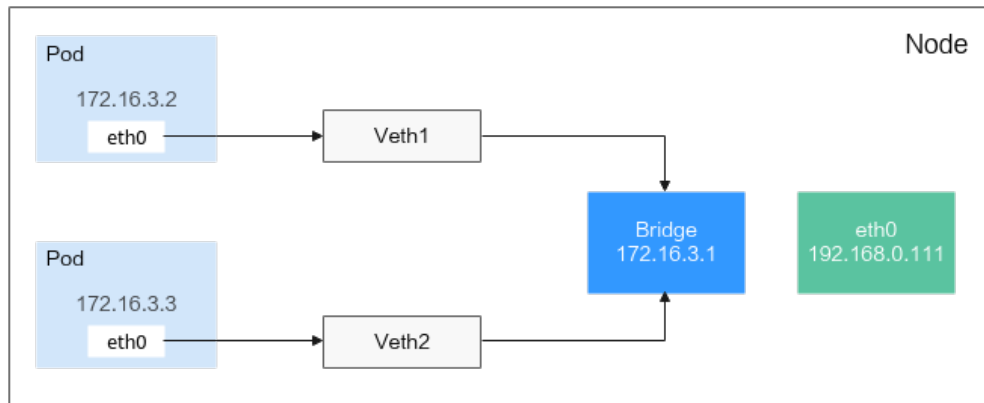
- Pods in a cluster must be accessible to each other through a non-NAT network. In this way, the source IP address of the data packets received by a pod is the IP address of the pod from which the data packets are sent.
- Nodes can communicate with each other without NAT.

### Pod Communication

#### Communication Between Pods on the Same Node

A pod communicates with external systems through veth devices created in interconnected pairs. The veth devices are virtual Ethernet devices acting as tunnels between network namespaces. The pods on the same node communicate with each other through a Linux bridge.

**Figure 7-1** Communication between pods on the same node



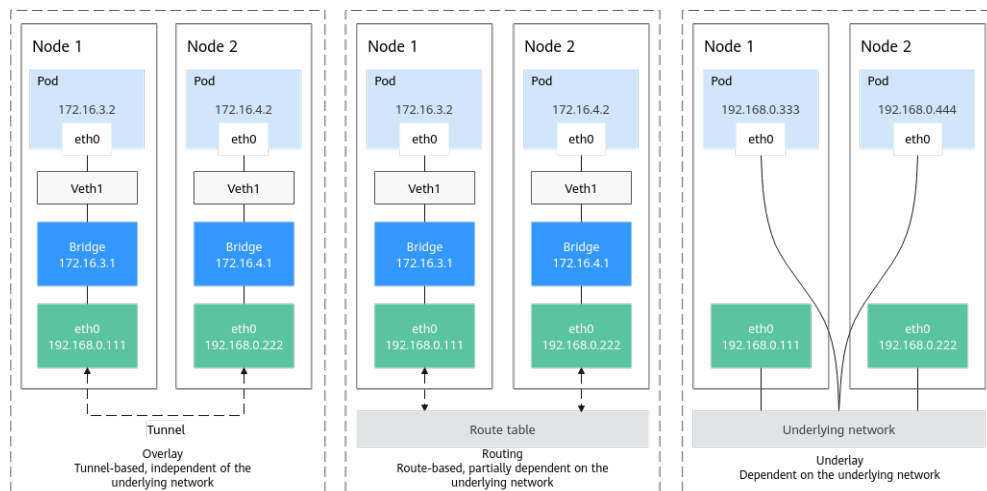
The pods on the same node connect to the bridge through veth devices. The IP addresses of these pods are dynamically obtained through the bridge and belong to the same CIDR block as the bridge IP address. Additionally, the default routes of all pods on the same node point to the bridge, and the bridge forwards all traffic with the source addresses that are not of the local network. In this way, the pods on the same node can directly communicate with each other.

### Communication Between Pods on Different Nodes

According to Kubernetes, the address of each pod in a cluster must be unique. Each node in the cluster is allocated with a subnet to ensure that the IP addresses of the pods are unique in the cluster. Pods running on different nodes communicate with each other through IP addresses in overlay, routing, or underlay networking mode based on the underlying dependency. This process is implemented using cluster networking plug-ins.

- An overlay network is separately constructed using tunnel encapsulation on the node network. Such a network has its own IP address space and IP switching/routing. VXLAN is a mainstream overlay network tunneling protocol.
- In a routing network, a VPC routing table is used with the underlying network for convenient communication between pods and nodes. The performance of routing surpasses that of the overlay tunnel encapsulation.
- In an underlay network, drivers expose underlying network interfaces on nodes to pods for high-performance network communication. IP VLANs are commonly used in underlay networking.

**Figure 7-2** Communication between pods on different nodes



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

## 7.2 Services

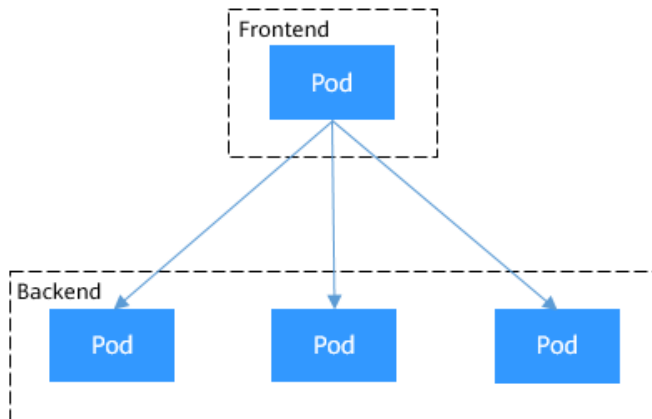
### Direct Access to a Pod

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

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

For example, an application uses Deployments to create the frontend and backend. The frontend calls the backend for computing, as shown in **Figure 7-3**. Three pods are running in the backend, which are independent and replaceable. When a backend pod is re-created, the new pod is assigned with a new IP address, of which the frontend pod is unaware.

Figure 7-3 Inter-pod access

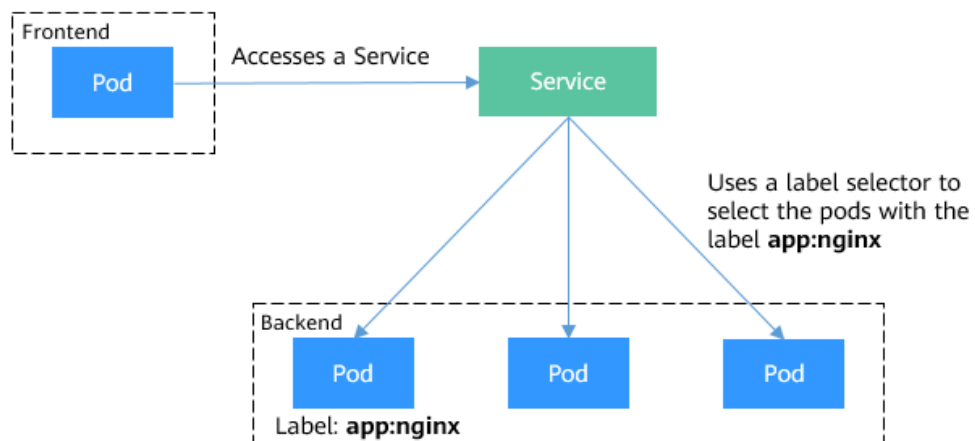


## Using Services for Pod Access

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

In the preceding example, a Service is added for the frontend pod to access the backend pods. In this way, the frontend pod does not need to be aware of the changes on backend pods, as shown in [Figure 7-4](#).

Figure 7-4 Accessing pods through a Service



## Creating Backend Pods

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

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

```
matchLabels:
  app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - image: nginx:latest
        name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    imagePullSecrets:
      - name: default-secret
```

## Creating a Service

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

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

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

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

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

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

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

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

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

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

## Using ServiceName to Access a Service

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

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

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

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

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

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

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

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

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

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

## Using Services for Service Discovery

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

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

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

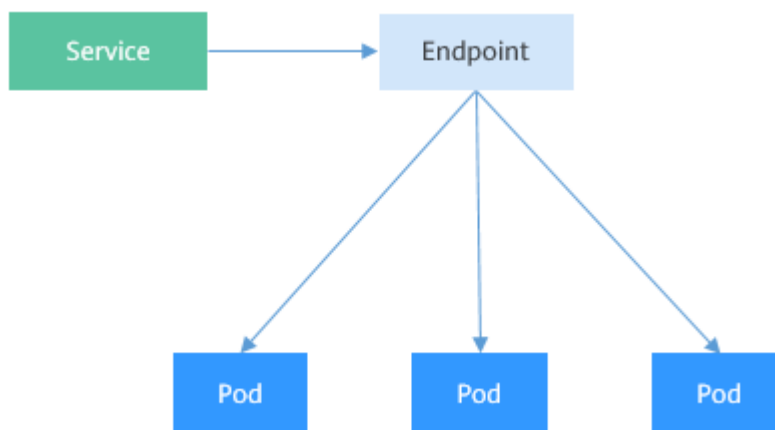
```
.....
Endpoints: 172.16.2.132:80,172.16.3.6:80,172.16.3.7:80
.....
```

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

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

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

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



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

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

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

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

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

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

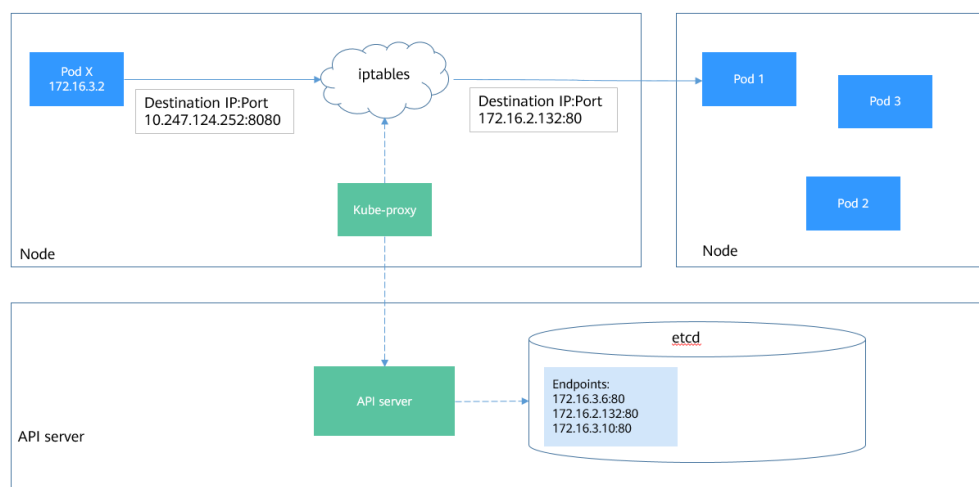
We have introduced kube-proxy on worker nodes in [Kubernetes Cluster Architecture](#). Actually, all Service-related operations are performed by kube-proxy. When a Service is created, Kubernetes allocates an IP address to the Service and notifies kube-proxy on all nodes of the Service creation through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

After receiving the notification, each kube-proxy records the IP address and port number of the Service through iptables. In this way, the Service can be obtained on each node.

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

In addition to recording the IP address and port number 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

Services of the ClusterIP, NodePort, LoadBalancer, and Headless Service types offer different functions.

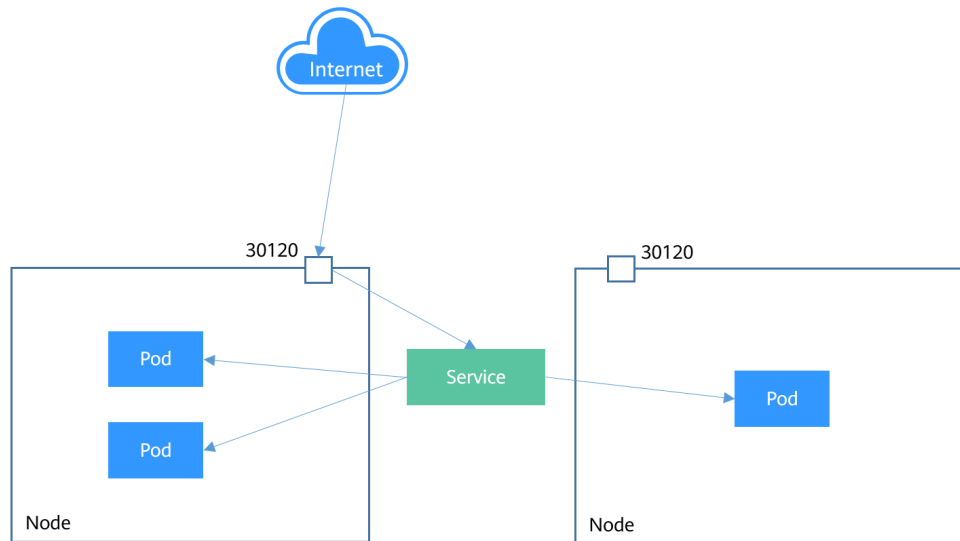
- ClusterIP: used to make the Service only reachable within a cluster.
- NodePort: used for access from outside a cluster. A NodePort Service is accessed through the port on the node. For details, see [NodePort Services](#).
- LoadBalancer: used for access from outside a cluster. It is an extension of NodePort, to which a load balancer routes, and external systems only need to access the load balancer. For details, see [LoadBalancer Services](#).
- 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 the `spec.clusterIP` value 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 *Node IP:Port* and then the NodePort Service forwards the requests to the pod backing the Service.



**Figure 7-6 NodePort Service**



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

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

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

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

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

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

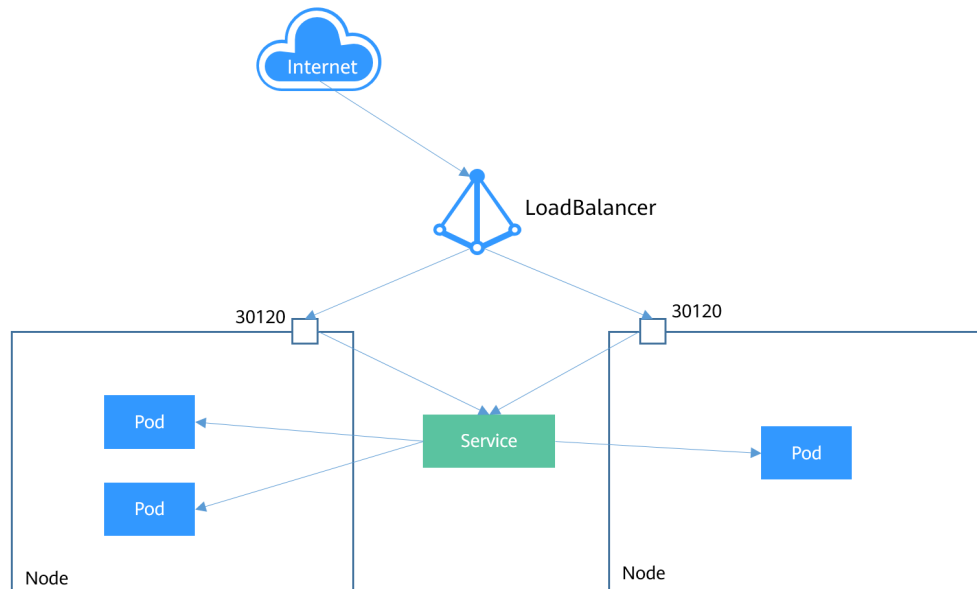
```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 192.168.0.212:30120
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
.....
```

## LoadBalancer Services

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

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

**Figure 7-7** LoadBalancer Service



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

```

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

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

## Headless Services

A Service allows a client to access a pod associated with the Service for both internal and external network communication. However, the following problems persist:

- Accessing all pods at the same time
- Allowing pods in 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 pod to be accessed is determined based on the cluster forwarding rule (IPVS or iptables). 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 in [StatefulSets](#) use headless Services for mutual access between pods.

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

Run the following command to create a headless Service:

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

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

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

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

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

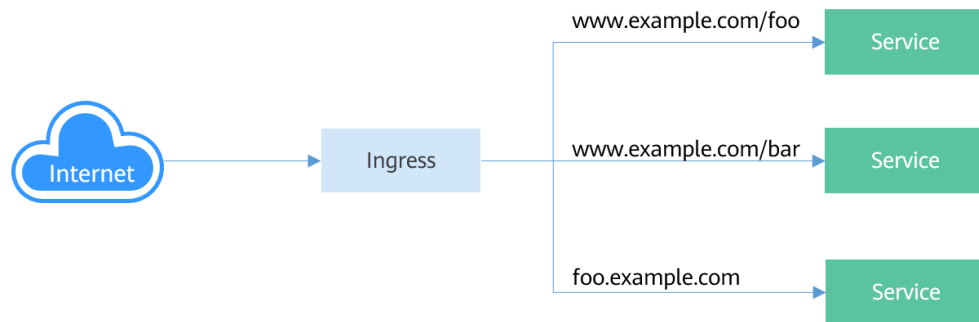
Name:   nginx-headless.default.svc.cluster.local
Address: 172.16.0.31
Name:   nginx-headless.default.svc.cluster.local
Address: 172.16.0.18
Name:   nginx-headless.default.svc.cluster.local
Address: 172.16.0.19
```

## 7.3 Ingresses

### Overview

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

Figure 7-8 Ingress and Service

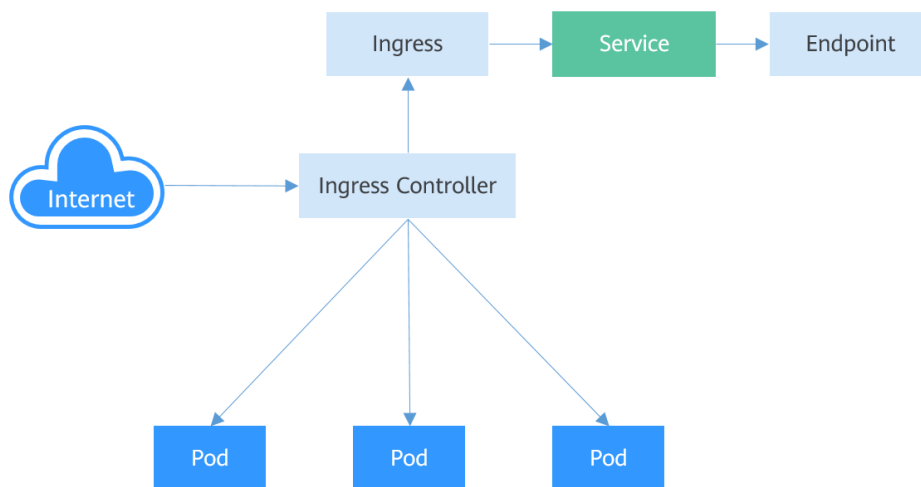


## Ingress Working Rules

To use ingresses, you must install Ingress Controller on your Kubernetes cluster. There are different implementations for an Ingress Controller. The most common one is **Ngix Ingress Controller** maintained by Kubernetes. CCE works with Elastic Load Balance (ELB) to implement layer-7 load balancing (ingresses).

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

Figure 7-9 Ingress working rules



## Creating an Ingress

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

The following is an example (applicable to 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 the external domain name in an ingress so that you can access the load balancer through the domain name and then access backend Services.

#### NOTE

Domain name-based access depends on domain name resolution. You need to point the domain name to the IP address of the load balancer. For example, you can use [Domain Name Service \(DNS\)](#) to resolve domain names.

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

## Accessing Multiple Services

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

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

---

#### NOTICE

The path in the ingress forwarding policy must exist in the backend application. Otherwise, the forwarding fails.

For example, the default access path of the Nginx application is **/usr/share/nginx/html**. If you add **/test** to the ingress forwarding policy, make sure that the access path of your Nginx application includes **/usr/share/nginx/html/test**. Otherwise, you will receive an error 404.

---

```
...
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

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

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

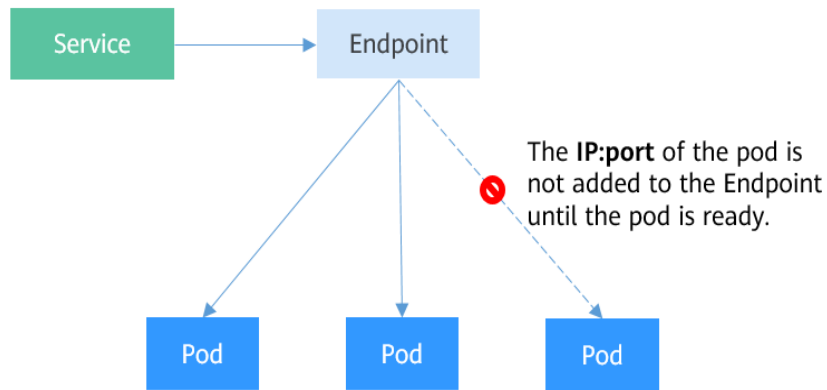
A readiness probe periodically detects a pod and determines whether the pod is ready based on its response. Similar to [Liveness Probes](#), there are three types of readiness probes.

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

### How Readiness Probes Work

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

**Figure 7-10** How readiness probes work



## Exec

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

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

Save the definition of the Deployment to the **deploy-ready.yaml** file, delete the previously created 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
```

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

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

Create a Service.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
  - name: service0
    targetPort: 80
    port: 8080
    protocol: TCP
  type: ClusterIP
```

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

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

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

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

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

# kubectl get endpoints
NAME      ENDPOINTS          AGE
nginx    192.168.93.169:80 14d
```

## HTTP GET

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

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



```
matchLabels:
  app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - image: nginx:alpine
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        readinessProbe:
          httpGet:
            path: /read
            port: 80
          # Readiness probe
          # HTTP GET definition
        imagePullSecrets:
          - name: default-secret
```

## TCP Socket

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

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

## Advanced Settings of a Readiness Probe

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

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

This is the detailed configuration of the readiness probe.

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

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

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

## 7.5 Network Policies

Network policies are designed by Kubernetes to restrict pod access. It is equivalent to a firewall at the application layer to 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 policy, pods in the namespace accept traffic from any source and send traffic to any destination.

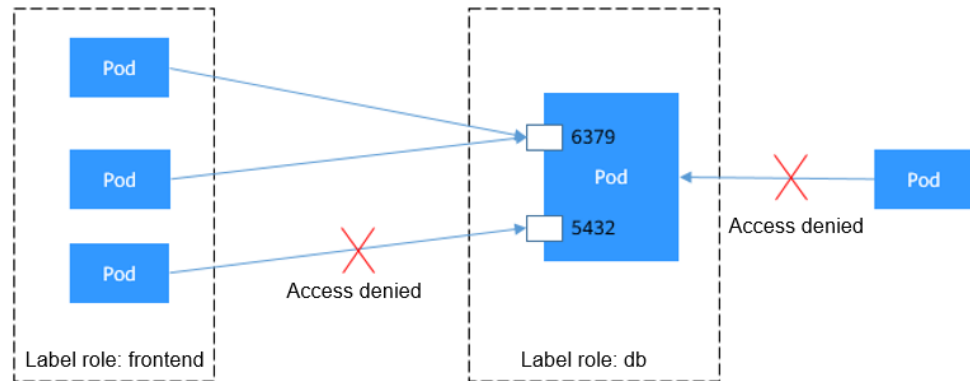
NetworkPolicy rules are classified into the following types:

- **namespaceSelector**: This selects particular namespaces for which all pods should be allowed as ingress sources or egress destinations.
- **podSelector**: This selects particular pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.
- **ipBlock**: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. (Only egress support IP address blocks.)

### Using Ingress Rules Through YAML

- **Scenario 1: Use a network policy to limit access to a pod to only 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 do so, perform the following operations:

- 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 allow 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 based on the **access-demo1.yaml** file:

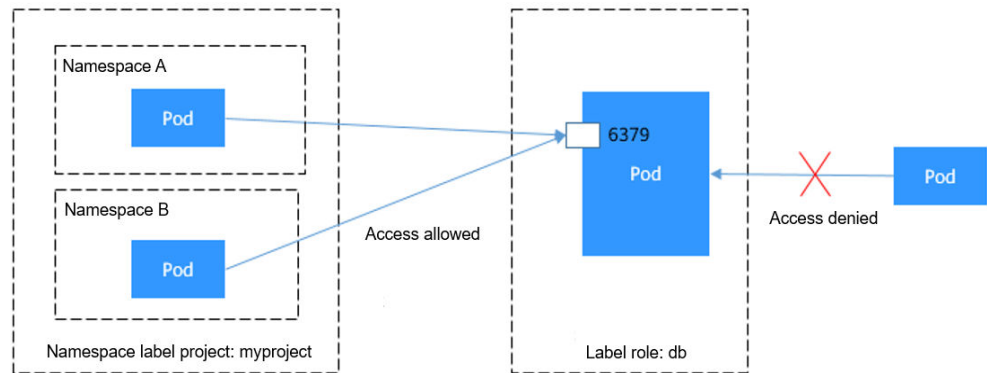
```
kubectl apply -f access-demo1.yaml
```

Expected output:

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

- **Scenario 2: Use a network policy to limit access to a pod to only 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 do so, perform the following operations:

- 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 allow 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 based on the **access-demo2.yaml** file:

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

Expected output:

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

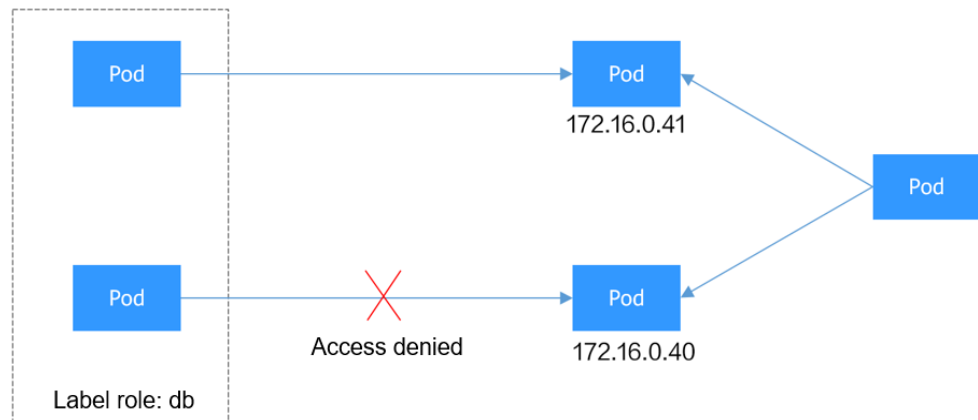
## Using Egress Rules Through YAML

### 📖 NOTE

The clusters of v1.23 or later using a tunnel network support egress rules.

- **Scenario 1: Use a network policy to limit a pod's access to specific addresses.**

**Figure 7-13 IPBlock**



The pod labeled with **role=db** only permits access to the 172.16.0.16/16 CIDR block, excluding 172.16.0.40/32 within it. To do so, perform the following operations:

- a. Create the **access-demo3.yaml** file.

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

File content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-demo3
  namespace: default
spec:
  policyTypes:
    # Must be specified for an egress rule.
    - Egress
  podSelector:
    # The rule takes effect for pods with the role=db label.
    matchLabels:
      role: db
  egress:
    # Egress rule
    - to:
      - ipBlock:
          cidr: 172.16.0.16/16 # Allow access to this CIDR block in the outbound direction.
        except:
          - 172.16.0.40/32 # Block access to this address in the CIDR block.
```

- b. Run the following command to create the network policy based on the **access-demo3.yaml** file:

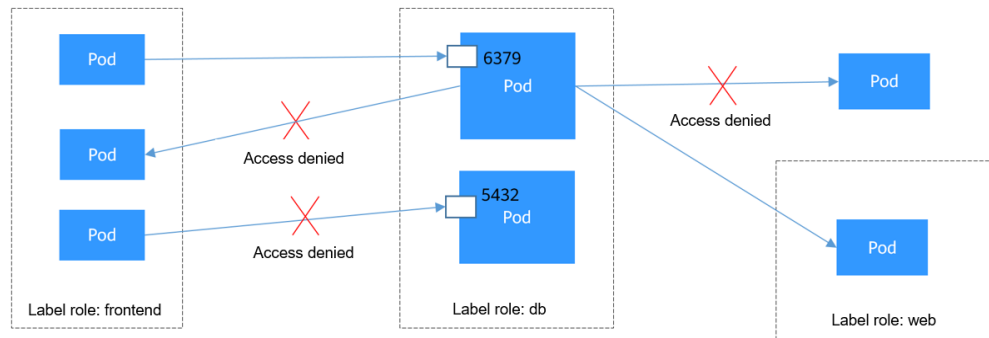
```
kubectl apply -f access-demo3.yaml
```

Expected output:

```
networkpolicy.networking.k8s.io/access-demo3 created
```

- **Scenario 2: Use a network policy to limit access to a pod to only pods with specific labels and this pod can only access specific pods.**

**Figure 7-14** Using both ingress and egress



The pod labeled with **role=db** only permits access to its port 6379 from pods labeled with **role=frontend**, and this pod can only access the pods labeled with **role=web**. You can use the same rule to configure both ingress and egress in a network policy. To do so, perform the following operations:

- a. Create the **access-demo4.yaml** file.

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

File content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-demo4
  namespace: default
spec:
  policyTypes:
    - Ingress
    - Egress
  podSelector:           # The rule takes effect for pods with the role=db label.
    matchLabels:
      role: db
  ingress:               # This is an ingress rule.
    - from:
      - podSelector:     # Only allow 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
  egress:               # Egress rule
    - to:
      - podSelector:     # Only pods with the role=web label can be accessed.
        matchLabels:
          role: web
```

- b. Run the following command to create the network policy based on the **access-demo4.yaml** file:

```
kubectl apply -f access-demo4.yaml
```

Expected output:

```
networkpolicy.networking.k8s.io/access-demo4 created
```

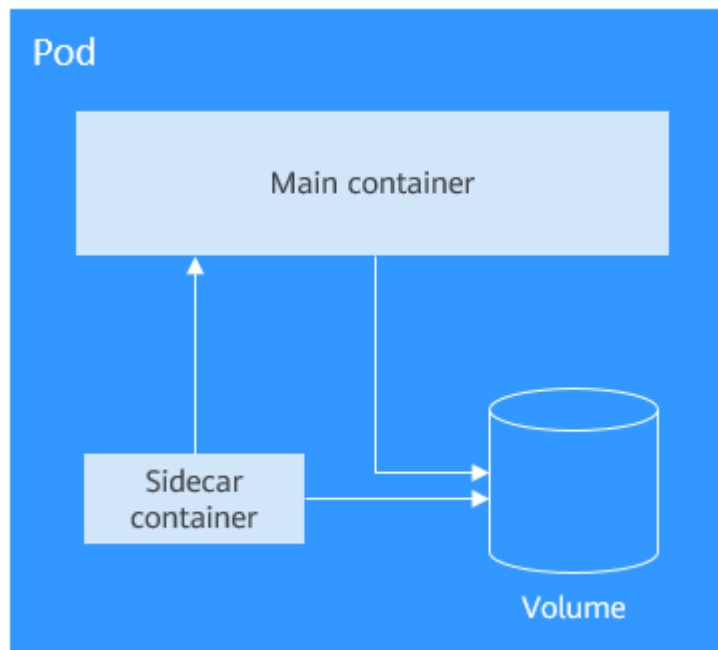
# 8 Persistent Storage

## 8.1 Volumes

On-disk files in a container are ephemeral, which will be lost when the container crashes and are difficult to be shared between containers running together in a pod. The Kubernetes volume abstraction solves both of these problems. Volumes cannot be independently created, but defined in the pod spec.

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

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



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

## Volume Types

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

- `emptyDir`: an empty volume used for temporary storage
- `hostPath`: a volume that mounts a directory of the host into your pod
- `ConfigMap` and `secret`: special volumes that inject or pass information to your pod. For details about how to mount `ConfigMaps` and `secrets`, see [ConfigMaps](#) and [Secrets](#).
- `persistentVolumeClaim`: Kubernetes persistent storage class. For details, see [PersistentVolumes](#), [PersistentVolumeClaims](#), and [StorageClasses](#).

### emptyDir

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

Some uses of an `emptyDir` volume are as follows:

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

Example `emptyDir` configuration:

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

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

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

### HostPath

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



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

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

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

## 8.2 PersistentVolumes, PersistentVolumeClaims, and StorageClasses

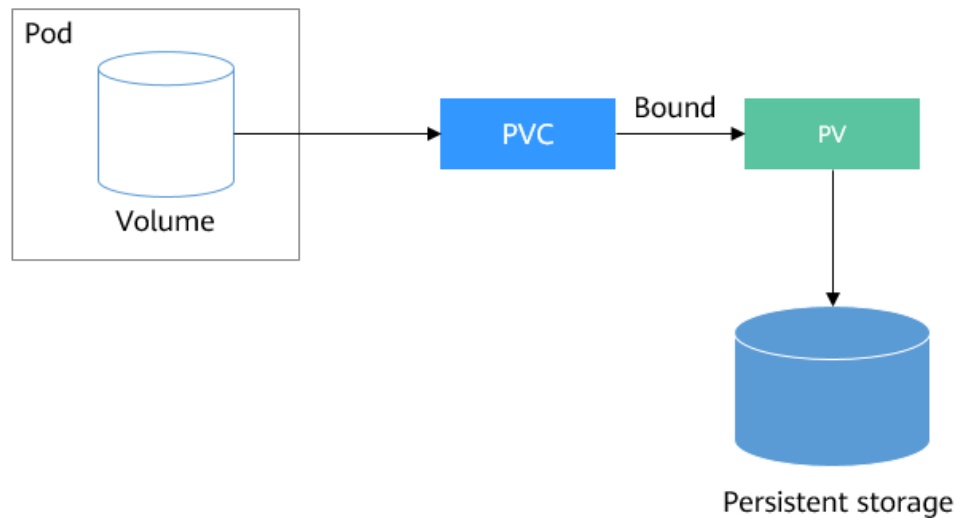
`hostPath` volumes are used for persistent storage. However, such volumes are node-specific. Data written into `hostPath` volumes may be different after a node restart.

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

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

To allow a pod to use a PV, the Kubernetes cluster administrator needs to configure a network `StorageClass` 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 Container Storage Interface (CSI) can be used to develop plug-ins to support specific storage volumes. For example, there are **everest-csi-controller** and **everest-csi-driver** developed by CCE in the **kube-system** namespace in [Namespace for Grouping Resources](#). With these drivers, 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
  
```

## PV

Each PV contains the specification and status of the volume. For 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 to describe the volume, as shown in the following example:

```

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              # StorageClass
    volumeAttributes:
      everest.io/share-export-location: sfs-nas01.cn-north-4b.myhuaweicloud.com:/share-96314776 # Mount point
    volumeHandle: 68e4a4fd-d759-444b-8265-20dc66c8c502 # Storage ID
  
```

Fields under **csi** in this example are dedicated used in CCE.

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  default/pv-example  4s
```

For **RECLAIM POLICY**, the value **Retain** indicates that the PV is retained after the PVC is released.

## PVC

Each PVC can only be bound to one PV. The following is an example:

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

Create the PVC 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           default/pvc-example  9s
```

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

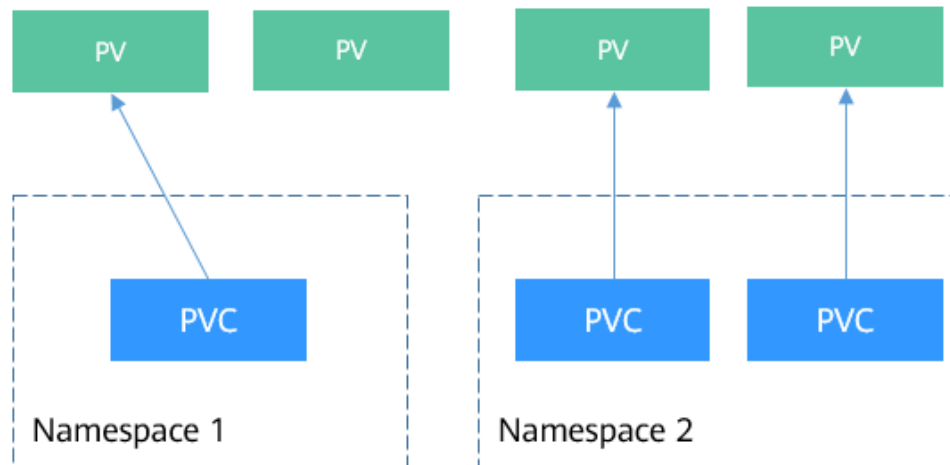
Then, check the PV status.

```
$ 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 named **pvc-example** in the **default** namespace.

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

**Figure 8-2** Relationship between PVs and PVCs



## StorageClass

PVs and PVCs allow you to consume storage resources, but creating them is a complex process, especially the `csi` field in PVs. 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 to create PVs automatically. The cluster administrator can deploy a PV provisioner and define StorageClasses. In this way, you can select a desired StorageClass when creating a PVC. The PVC then transfers the StorageClass to the PV provisioner, and the provisioner automatically creates a PV. In CCE, StorageClasses such as `csi-disk`, `csi-nas`, and `csi-obs` are supported. The `storageClassName` field is added to a PVC so that PVs can be automatically provisioned and underlying storage resources can be automatically created.

Run the following command to obtain the StorageClasses that CCE supports. You can use the CSI add-ons provided by CCE to customize StorageClasses, which function similarly as the default StorageClasses in CCE.

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

Run the following command to specify a StorageClass 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 by using the StorageClass **csi-sfsturbo**. To use SFS Turbo storage, create an SFS Turbo file system and then a PV and PVC through a static PV. For details, see [Using an Existing SFS Turbo File System Through a Static PV](#).

Run the following command to 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 StorageClass is specified, a PVC and a PV are created and bound.

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

## Using a PVC in a Pod

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

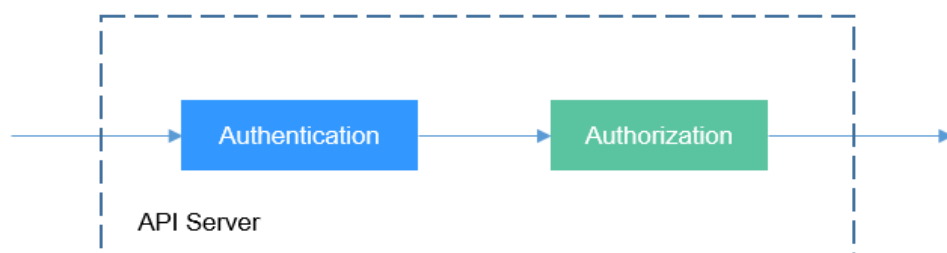
# 9 Authentication and Authorization

## 9.1 Service Accounts

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

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

**Figure 9-1** Authentication and authorization of the API Server



### Authentication and ServiceAccounts

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

- A ServiceAccount is bound to a namespace and associated with a set of credentials. When a pod is created, the 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).

ServiceAccounts in Kubernetes are resources that exist at the namespace level, just like pods and ConfigMaps. When a namespace is created, the system automatically generates a ServiceAccount named **default** in the namespace.

You can run the following command to check ServiceAccounts:

### kubectl get sa

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

#### NOTE

- In clusters earlier than v1.21, a token is obtained by mounting the secret of the service account to a pod. Tokens obtained this way are permanent. This approach is no longer recommended starting from version 1.21. Service accounts will stop auto creating secrets in clusters from version 1.25.  
In clusters of version 1.21 or later, you can use the [TokenRequest](#) API to **obtain the token** and use the projected volume to mount the token to the pod. Such tokens are valid for a fixed period. When the mounting pod is deleted, the token automatically becomes invalid. For details, see [Service Account Token Security Improvement](#).
- If you need a token that never expires, you can also **manually manage secrets for service accounts**. Although a permanent service account token can be manually created, you are advised to use a short-lived token by calling the [TokenRequest](#) API for higher security.

In clusters earlier than 1.25, a secret is automatically created for each ServiceAccount. In clusters 1.25 or later, a secret is not automatically created for each ServiceAccount. The following describes how to check the statuses of ServiceAccounts in clusters earlier than 1.25 and in clusters 1.25 or later.

- In a cluster earlier than 1.25, run the following command to check the status of the **default** ServiceAccount.

#### kubectl describe sa default

If information similar to the following is displayed, the **default-token-vssmw** secret is automatically created for the ServiceAccount.

```
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 1.25 or later, run the following command to check the status of the **default** ServiceAccount.

#### kubectl describe sa default

According to the command output, no secret is automatically created for the **default** ServiceAccount.

```
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 ServiceAccount to it by specifying the account name in the file. If no account name is specified, the default

ServiceAccount will be used. When receiving a request with an authentication token, the API Server uses the token to check whether the ServiceAccount associated with the client that sends the request allows the request to be executed.

## Creating a ServiceAccount

- Step 1** Take a cluster 1.29 as an example. Run the following command to create a ServiceAccount in the **default** namespace:

```
kubectl create serviceaccount sa-example
```

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

Run the following command to 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 case is later than 1.25, the ServiceAccount will not have a secret created automatically. To check if a secret was created, use the following command to view the ServiceAccount details. If the output shows **none** for **Mountable secrets** and **Tokens**, then no secret was automatically created for the ServiceAccount.

```
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** In this example, manually manage the secret to obtain a token that never expires. Use the following command to manually create a secret named **sa-example-token** and associate it with the **sa-example** ServiceAccount.

```
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 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 find the **ca.crt**, **namespace**, and **token** data.

**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 ServiceAccount has been associated with the new secret, meaning if the ServiceAccount 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 ServiceAccount in a Pod

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

**Step 1** Create a description file named **sa-pod.yaml**. **mysql.yaml** is an example file name. You can rename it as required.

**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 description file.

The file content is as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-pod
spec:
  serviceAccountName: sa-example           # Specify sa-example as the ServiceAccount 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, which
      can be customized
  volumes:
    # Define a secret volume that can be used by the pod.
    - name: secret-volume
      # Name of the secret volume, which can be customized
      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**

The command output is as follows:

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

Use the following command to 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** details and check whether **sa-example-token** has been mounted to the pod.

**kubectl describe pod sa-pod**

The command output is as follows:

```
...
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, and the pod can use the token for
      authentication.
```

```
/etc/secret-volume from secret-volume (ro)
# Automatically mounted TokenRequest, which can provide a short-term token
/var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-2s4sw (ro)
...
```

You can also run the following command to view the corresponding files in the pod. (The path after **cd** is the same as the mount path of **secret-volume**.)

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

```
cd /etc/secret-volume
```

```
ls
```

The command output is as follows:

```
ca.crt namespace token
```

#### Step 4 Verify that the manually created ServiceAccount 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**, you can run the following command to view the detailed information about the Service:

```
kubectl get svc
```

The command output is as follows:

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, which instructs the **curl** command 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 as expected.

```
eyJhbGciOiJIUzI1NiIsImtpZCI6I...
```

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 passed authentication and the manually created ServiceAccount token is in effect. (If the API Server returns **cannot get path \"/api/v1/namespaces/default/pods\"**, the pod does not have the access permissions. The pod can access the API Server only 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

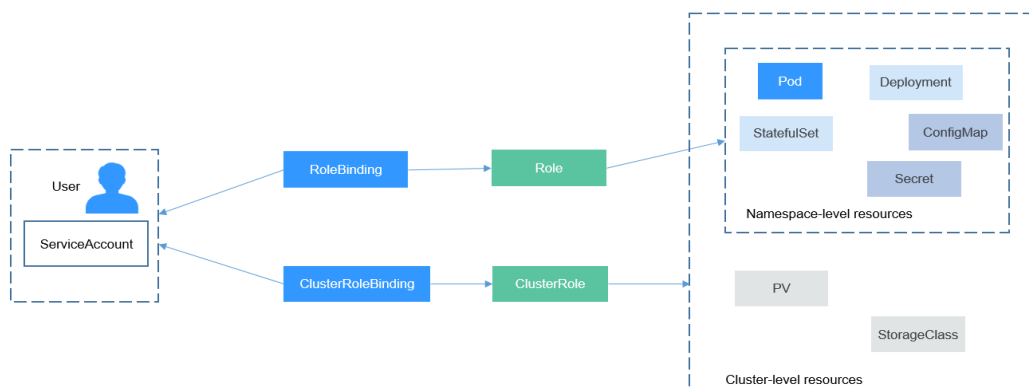
### RBAC Resources

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

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

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

**Figure 9-2** Role binding



## Creating a Role

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

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

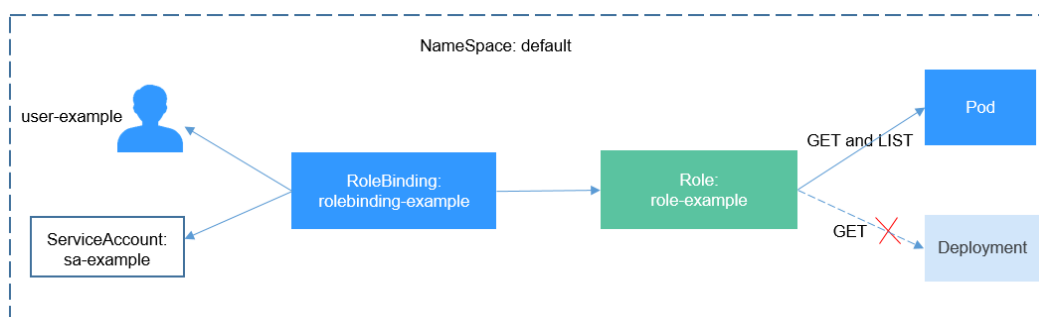
## Creating a RoleBinding

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

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

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

**Figure 9-3** Binding a role to a user



Then check whether the authorization takes effect.

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

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

```
$ kubectl exec -it sa-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": {
        ...

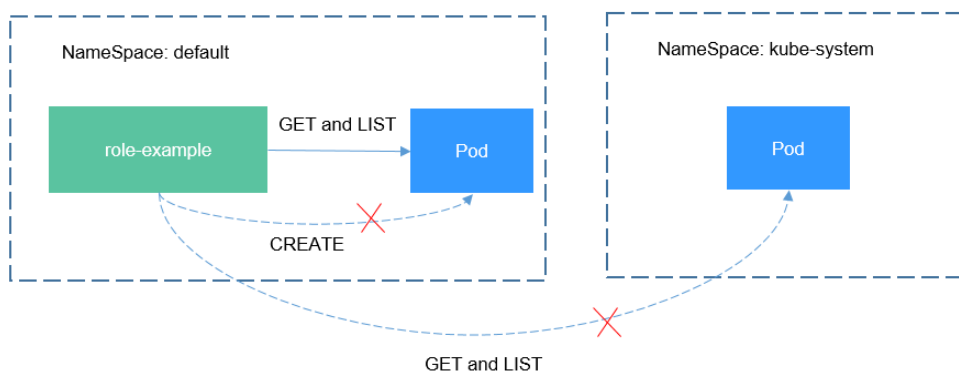
```

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

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

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

**Figure 9-4** Role and RoleBinding applied to namespaces



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

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

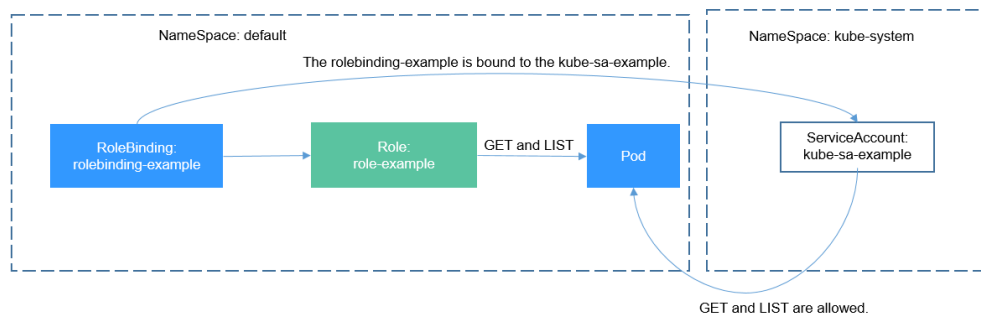
```
"message": "pods is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list resource \"pods\" in API group \"\" in the namespace \"kube-system\",  
\"reason\": \"Forbidden\",  
...
```

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

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

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

Figure 9-5 Cross-namespace access



## ClusterRole and ClusterRoleBinding

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

- ClusterRole and ClusterRoleBinding do not need to define the **namespace** field.
- ClusterRole can define cluster-level resources.

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

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

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

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

The most important and commonly used ClusterRoles are as follows:

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

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

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



# 10 Auto Scaling

---

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

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

## Prometheus and Metrics Server

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

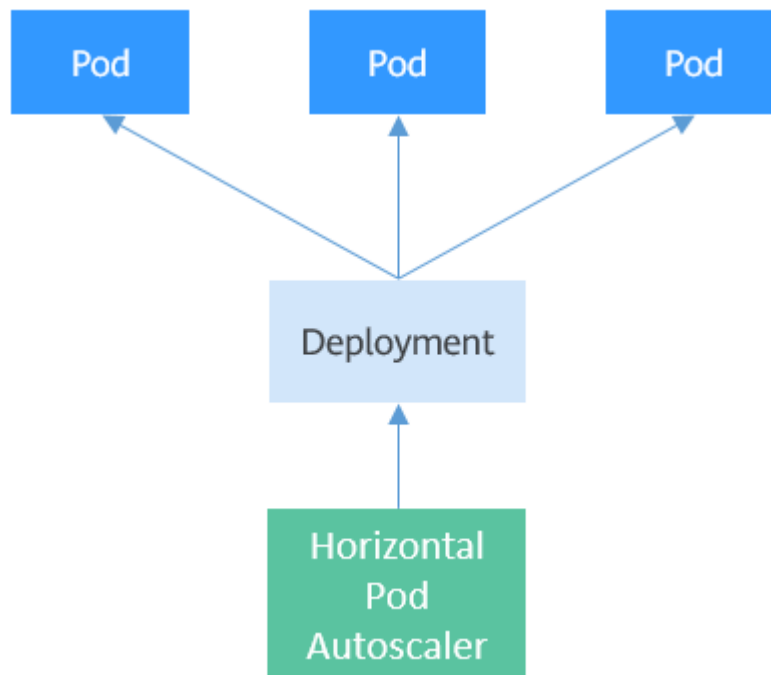
- [Prometheus](#) is an open source monitoring and alarming framework that can collect multiple types of metrics. Prometheus has been a standard monitoring solution of Kubernetes.
- [Metrics Server](#) is a cluster-wide aggregator of resource utilization data. Metrics Server collects metrics from the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

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

## How HPA Works

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

**Figure 10-1** HPA working rules



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

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

## Using the HPA

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

```

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

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

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

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  
```

```

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

```

Query the created HPA.

```

$ 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 perform scale-in. The expected number of replicas can be calculated as follows:  $(0 + 0 + 0 + 0)/70 = 0$ . However, the minimum number of replicas has been set to **1**. Therefore, the number of pods is changed to 1. After a while, the number of pods changes to 1.

```

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

```

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

```

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

```

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

```

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

```

## Cluster Autoscaler

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

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

For details about the implementation in CCE, see [Creating a Node Scaling Policy](#).