Kubernetes 开源知识

文档版本 01

发布日期 2021-10-24

华为技术有限公司

商标声明


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

注意

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

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

目 录

1 概述

Kubernetes是一个开源的容器编排部署管理平台，用于管理云平台中多个主机上的容器化应用。Kubernetes的目标是让部署容器化的应用简单并且高效，Kubernetes提供了应用部署、规划、更新、维护的一种机制。

对应用开发者而言，可以把Kubernetes看成一个集群操作系统。Kubernetes提供服务发现、伸缩、负载均衡、自愈甚至选举等功能，让开发者从基础设施相关配置等解脱出来。

您可以通过**CCE控制台**、**Kubectl命令行**、**Kubernetes API**使用云容器引擎所提供的Kubernetes托管服务。在使用云容器引擎之前，你可以先行了解如下Kubernetes的相关概念，以便您更完整的使用云容器引擎的所有功能。

容器与 Kubernetes

- **2.1 容器**
- **2.2 Kubernetes**

Pod、Label 和 Namespace

- **3.1 Pod：Kubernetes中的最小调度对象**
- **3.2 存活探针（Liveness Probe）**
- **3.3 Label：组织Pod的利器**
- **3.4 Namespace：资源分组**

Pod 的编排与调度

- **4.1 Deployment**
- **4.2 StatefulSet**
- **4.3 Job和CronJob**
- **4.4 DaemonSet**
- **4.5 亲和与反亲和调度**

配置管理

- **5.1 ConfigMap**

2 容器与 Kubernetes

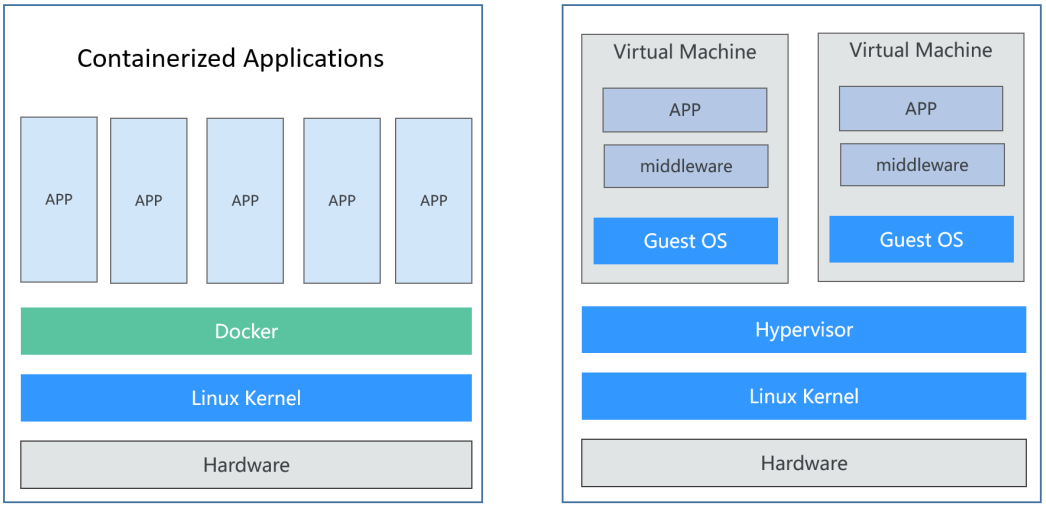
2.1 容器

容器与 Docker

容器技术起源于Linux，是一种内核虚拟化技术，提供轻量级的虚拟化，以便隔离进程和资源。尽管容器技术已经出现很久，却是随着Docker的出现而变得广为人知。Docker是第一个使容器能在不同机器之间移植的系统。它不仅简化了打包应用的流程，也简化了打包应用的库和依赖，甚至整个操作系统的文件系统能被打包成一个简单的可移植的包，这个包可以被用来在任何其他运行Docker的机器上使用。

容器和虚拟机具有相似的资源隔离和分配方式，容器虚拟化了操作系统而不是硬件，更加便携和高效。

图 2-1 容器 vs 虚拟机



相比于使用虚拟机，容器有如下优点：

- 更高效的利用系统资源

  由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，容器对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传

统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

- 更快速的启动时间

  传统的虚拟机技术启动应用服务往往需要数分钟，而Docker容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间，大大节约了开发、测试、部署的时间。

- 一致的运行环境

  开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些问题并未在开发过程中被发现。而Docker的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性。

- 更轻松的迁移

  由于Docker确保了执行环境的一致性，使得应用的迁移更加容易。Docker可以在很多平台上运行，无论是物理机、虚拟机，其运行结果是一致的。因此可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

- 更轻松的维护和扩展

  Docker使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker团队同各个开源项目团队一起维护了大批高质量的官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

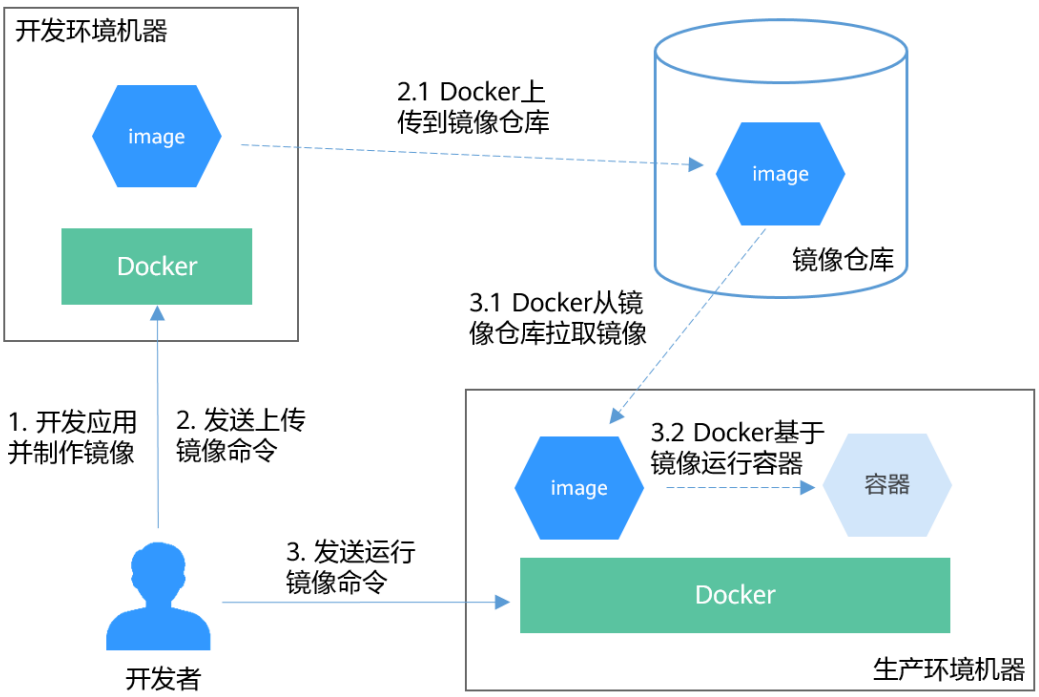
Docker 容器典型使用流程

Docker容器有如下三个主要概念：

- **镜像**：Docker镜像里包含了已打包的应用程序及其所依赖的环境。它包含应用程序可用的文件系统和其他元数据，如镜像运行时的可执行文件路径。
- **镜像仓库**：Docker镜像仓库用于存放Docker镜像，以及促进不同人和不同电脑之间共享这些镜像。当编译镜像时，要么可以在编译它的电脑上运行，要么可以先上传镜像到一个镜像仓库，然后下载到另外一台电脑上并运行它。某些仓库是公开的，允许所有人从中拉取镜像，同时也有一些是私有的，仅部分人和机器可接入。
- **容器**：Docker容器通常是一个Linux容器，它基于Docker镜像被创建。一个运行中的容器是一个运行在Docker主机上的进程，但它和主机，以及所有运行在主机上的其他进程都是隔离的。这个进程也是资源受限的，意味着它只能访问和使用分配给它的资源（CPU、内存等）。

典型的使用流程如图2-2所示：

图 2-2 Docker 容器典型使用流程



1. 首先开发者在开发环境机器上开发应用并制作镜像。
   Docker执行命令，构建镜像并存储在机器上。
2. 开发者发送上传镜像命令。
   Docker收到命令后，将本地镜像上传到镜像仓库。
3. 开发者向生产环境机器发送运行镜像命令。
   生产环境机器收到命令后，Docker会从镜像仓库拉取镜像到机器上，然后基于镜像运行容器。

使用示例

下面使用Docker将基于Nginx镜像打包一个容器镜像，并基于容器镜像运行应用，然后推送到容器镜像仓库。

安装Docker

Docker几乎支持在所有操作系统上安装，用户可以根据需要选择要安装的Docker版本。

在Linux操作系统下，可以使用如下命令快速安装Docker。

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

Docker打包镜像

Docker提供了一种便捷的描述应用打包的方式，叫做Dockerfile，如下所示：

```
# 使用官方提供的Nginx镜像作为基础镜像
FROM nginx:alpine

# 执行一条命令修改Nginx镜像index.html的内容
```



```
RUN echo "hello world" > /usr/share/nginx/html/index.html

# 允许外界访问容器的80端口
EXPOSE 80
```

执行docker build命令打包镜像。

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

其中-t表示给镜像加一个标签，也就是给镜像取名，这里镜像名为hello。. 表示在当前目录下执行该打包命令。

执行docker images命令查看镜像，可以看到hello镜像已经创建成功。您还可以看到一个Nginx镜像，这个镜像是从镜像仓库下载下来的，作为hello镜像的基础镜像使用。

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

本地运行容器镜像

有了镜像后，您可以在本地执行docker run命令运行容器镜像。

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

docker run命令会启动一个容器，命令中-p是将本地机器的8080端口映射到容器的80端口，即本地机器的8080端口的流量会映射到容器的80端口，当您在本地机器访问http://127.0.0.1:8080时，就会访问到容器中，此时浏览器中返回的内容应该就是“hello world”。

把镜像推送到镜像仓库

华为云提供了容器镜像服务SWR，您也可以将镜像上传到SWR，下面演示如何将镜像推送到SWR。详细的方法请参见**客户端上传镜像**，本文档后续的示例中将主要使用SWR作为示例。

首先登录**SWR控制台**，在左侧选择“我的镜像”，然后单击右侧“客户端上传镜像”，在弹出的窗口中单击“生成临时登录指令”，然后复制该指令在本地机器上执行，登录到SWR镜像仓库。

上传镜像前需要给镜像取一个完整的名称，如下所示：

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

这里swr.cn-east-3.myhuaweicloud.com是仓库地址，每个华为云区域的地址不同，v1则是hello镜像分配的版本号。

- swr.cn-east-3.myhuaweicloud.com是仓库地址，每个华为云区域的地址不同。
- container是组织名，组织一般在SWR中创建，如果没有创建则首次上传的时候会自动创建，组织名在单个区域内全局唯一，需要选择合适的组织名称。
- v1则是hello镜像分配的版本号。

然后执行docker push命令就可以将镜像上传到SWR。

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

当需要使用该镜像时，使用docker pull命令拉取（下载）该命令即可。

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

2.2 Kubernetes

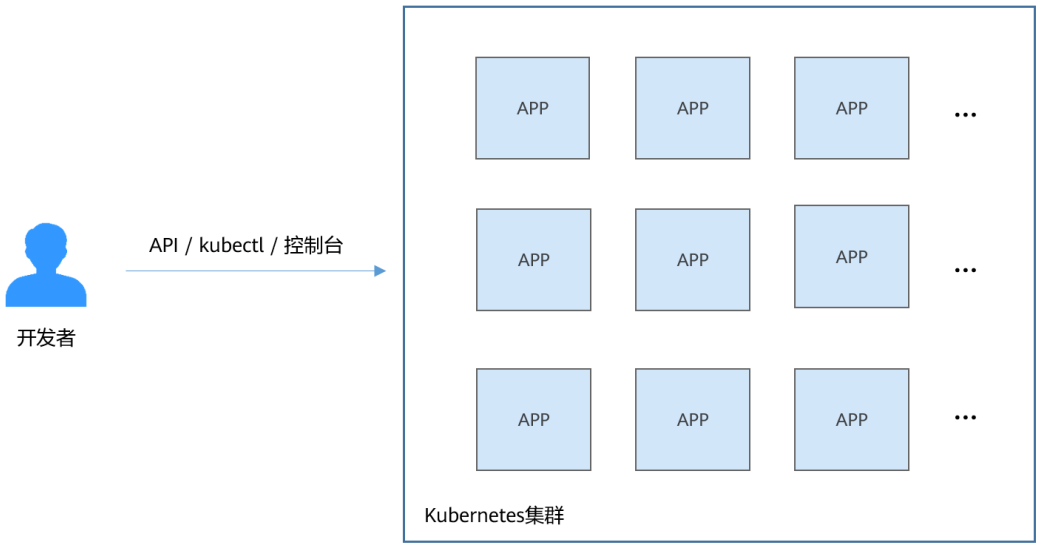
Kubernetes 是什么

Kubernetes是一个很容易地部署和管理容器化的应用软件系统，使用Kubernetes能够方便对容器进行调度和编排。

对应用开发者而言，可以把Kubernetes看成一个集群操作系统。Kubernetes提供服务发现、伸缩、负载均衡、自愈甚至选举等功能，让开发者从基础设施相关配置等解脱出来。

Kubernetes可以把大量的服务器看做一台巨大的服务器，在一台大服务器上面运行应用程序。无论Kubernetes的集群有多少台服务器，在Kubernetes上部署应用程序的方法永远一样。

图 2-3 在 Kubernetes 集群上运行应用程序

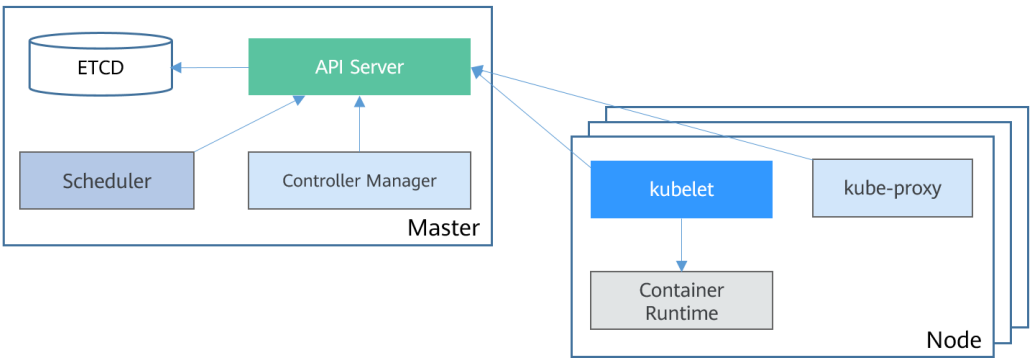


Kubernetes 集群架构

Kubernetes集群包含master节点（控制节点）和node节点（计算节点/工作节点），应用部署在node节点上，且可以通过配置选择应用部署在某些特定的节点上。

Kubernetes集群的架构如下所示：

图 2-4 Kubernetes 集群架构



Master节点

Master节点是集群的控制节点，由API Server、Scheduler、Controller Manager和ETCD四个组件构成。

- API Server：各组件互相通讯的中转站，接受外部请求，并将信息写到ETCD中。
- Controller Manager：执行集群级功能，例如复制组件，跟踪Node节点，处理节点故障等等。
- Scheduler：负责应用调度的组件，根据各种条件（如可用的资源、节点的亲和性等）将容器调度到Node上运行。
- ETCD：一个分布式数据存储组件，负责存储集群的配置信息。

在生产环境中，为了保障集群的高可用，通常会部署多个master，如CCE的集群高可用模式就是3个master节点。

Node节点

Node节点是集群的计算节点，即运行容器化应用的节点。

- kubelet：kubelet主要负责同Container Runtime打交道，并与API Server交互，管理节点上的容器。
- kube-proxy：应用组件间的访问代理，解决节点上应用的访问问题。
- Container Runtime：容器运行时，如Docker，最主要的功能是下载镜像和运行容器。

Kubernetes 的扩展性

Kubernetes开放了容器运行时接口（CRI）、容器网络接口（CNI）和容器存储接口（CSI），这些接口让Kubernetes的扩展性变得最大化，而Kubernetes本身则专注于容器调度。

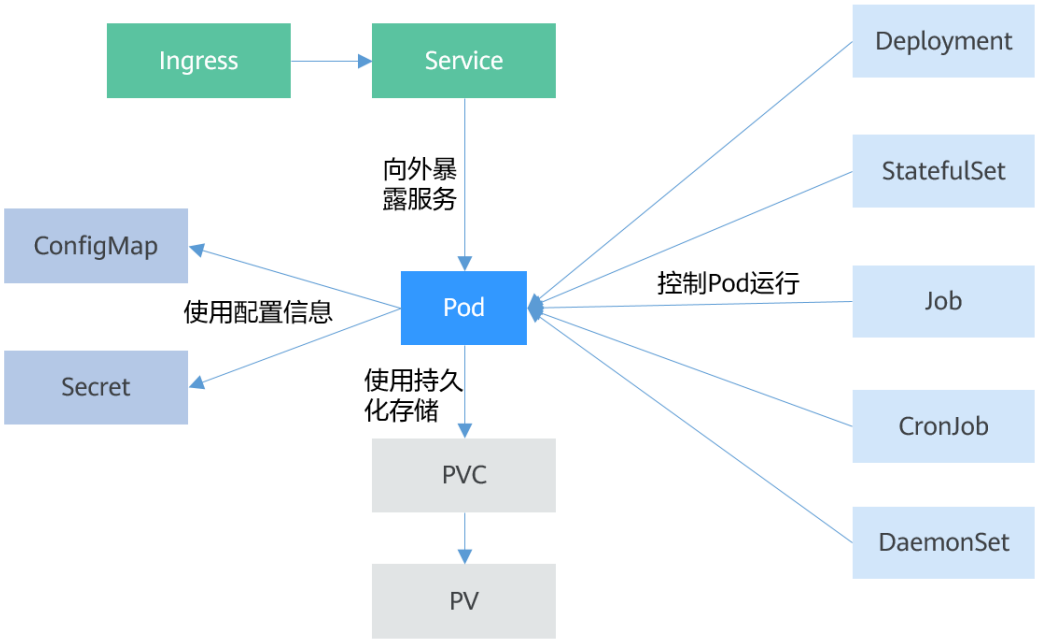
- CRI（Container Runtime Interface）：容器运行时接口，提供计算资源，CRI隔离了各个容器引擎之间的差异，而通过统一的接口与各个容器引擎之间进行互动。
- CNI（Container Network Interface）：容器网络接口，提供网络资源，通过CNI接口，Kubernetes可以支持不同网络环境。例如华为云CCE就是开发的CNI插件支持Kubernetes集群运行在华为云VPC网络中。

- CSI（Container Storage Interface）：容器存储接口，提供存储资源，通过CSI接口，Kubernetes可以支持各种类型的存储。例如华为云CCE就可以方便的对接华为云块存储（EVS）、文件存储（SFS）和对象存储（OBS）。

Kubernetes 中的基本对象

上面介绍Kubernetes集群的构成，下面将介绍Kubernetes中基本对象及它们之间的一些关系。

图 2-5 Kubernetes 基本对象



- Pod

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

- Deployment

  Deployment是对Pod的服务化封装。一个Deployment可以包含一个或多个Pod，每个Pod的角色相同，所以系统会自动为Deployment的多个Pod分发请求。

- StatefulSet

  StatefulSet是用来管理有状态应用的对象。和Deployment相同的是，StatefulSet管理了基于相同容器定义的一组Pod。但和Deployment不同的是，StatefulSet为它们的每个Pod维护了一个固定的ID。这些Pod是基于相同的声明来创建的，但是不能相互替换，无论怎么调度，每个Pod都有一个永久不变的ID。

- Job

  Job是用来控制批处理型任务的对象。批处理业务与长期伺服业务（Deployment）的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出（Pod自动删除）。

- CronJob

CronJob是基于时间控制的Job，类似于Linux系统的crontab，在指定的时间周期运行指定的任务。

- DaemonSet

  DaemonSet是这样一种对象（守护进程），它在集群的每个节点上运行一个Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资源监控等，这类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes的kube-proxy。

- Service

  Service是用来解决Pod访问问题的。Service有一个固定IP地址，Service将访问流量转发给Pod，而且Service可以给这些Pod做负载均衡。

- Ingress

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

- ConfigMap

  ConfigMap是一种用于存储应用所需配置信息的资源类型，用于保存配置数据的键值对。通过ConfigMap可以方便的做到配置解耦，使得不同环境有不同的配置。

- Secret

  Secret是一种加密存储的资源对象，您可以将认证信息、证书、私钥等保存在Secret中，而不需要把这些敏感数据暴露到镜像或者Pod定义中，从而更加安全和灵活。

- PersistentVolume（PV）

  PV指持久化数据存储卷，主要定义的是一个持久化存储在宿主机上的目录，比如一个NFS的挂载目录。

- PersistentVolumeClaim（PVC）

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

搭建 Kubernetes 集群

Kubernetes网站上有多种搭建Kubernetes集群的方法，例如minikube、kubeadm等。

如果不想自行搭建Kubernetes集群，可以在**华为云的CCE服务**中购买，本文后续内容都将在CCE中购买的集群上操作演示。

kubectl

kubectl是Kubernetes集群的命令行工具，您可以将kubectl安装在任意一台机器上，通过kubectl命令操作Kubernetes集群。

CCE集群的kubectl安装请参见**通过kubectl连接集群**。连接后您可以执行**kubectl cluster-info**查看集群的信息，如下所示。

```
# 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'.
```


执行**kubectl get nodes**可以查看集群中的Node节点信息。

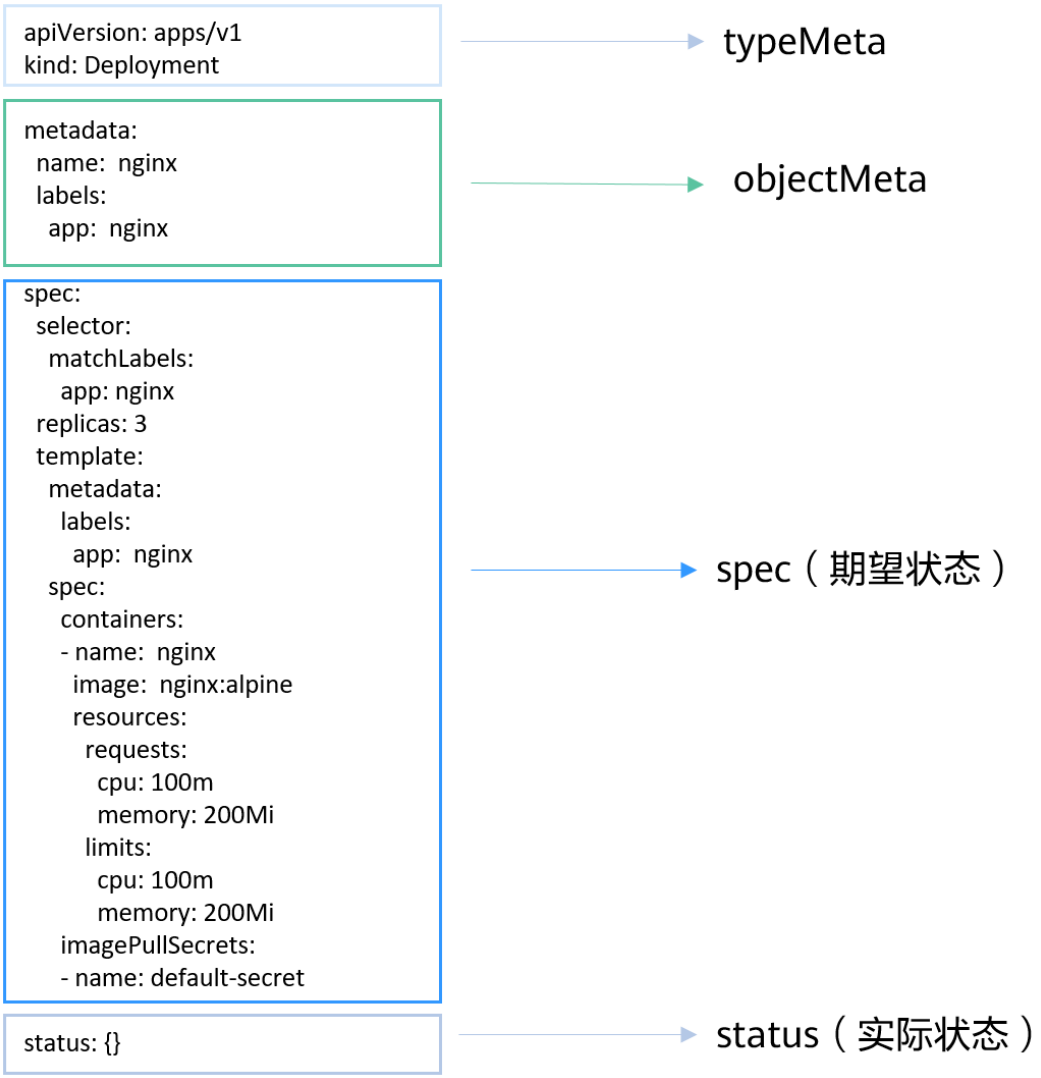
```
# kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
192.168.0.153   Ready    <none>   7m      v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.207   Ready    <none>   7m      v1.15.6-r1-20.3.0.2.B001-15.30.2
192.168.0.221   Ready    <none>   7m      v1.15.6-r1-20.3.0.2.B001-15.30.2
```

Kubernetes 对象的描述

kubernetes中资源可以使用YAML描述（如果您对YAML格式不了解，可以参考**YAML语法**），也可以使用JSON。其内容可以分为如下四个部分：

- typeMeta：对象类型的元信息，声明对象使用哪个API版本，哪个类型的对象。
- objectMeta：对象的元信息，包括对象名称、使用的标签等。
- spec：对象的期望状态，例如对象使用什么镜像、有多少副本等。
- status：对象的实际状态，只能在对象创建后看到，创建对象时无需指定。

图 2-6 YAML 描述文件



在 Kubernetes 上运行应用

将图2-6中的内容去除status存为一个名为nginx-deployment.yaml的文件，如下所示：

```
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
```

使用kubectl连接集群后，执行如下命令：

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

命令执行后，Kubernetes集群中会创建3个Pod，使用如下命令可以查询到Deployment和Pod：

```
# 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
```

到此为止，您了解容器和Docker、Kubernetes集群、Kubernetes基本概念，并通过一个示例了解kubectl的最基本使用，本文后续将向您深入介绍Kubernetes对象的概念以及使用方法，并介绍对象之间的关系。

3 Pod、Label 和 Namespace

3.1 Pod：Kubernetes 中的最小调度对象

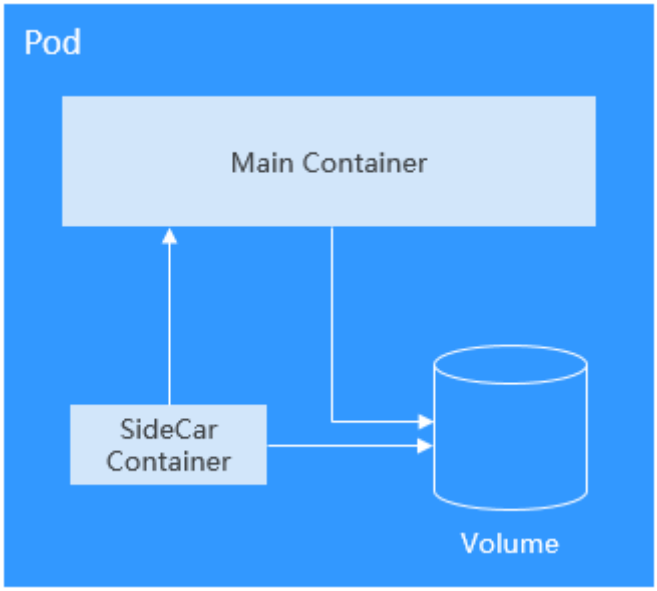
什么是 Pod

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

Pod使用主要分为两种方式：

- Pod中运行一个容器。这是Kubernetes最常见的用法，您可以将Pod视为单个封装的容器，但是Kubernetes是直接管理Pod而不是容器。
- Pod中运行多个需要耦合在一起工作、需要共享资源的容器。通常这种场景下应用包含一个主容器和几个辅助容器（SideCar Container），如图3-1所示，例如主容器为一个web服务器，从一个固定目录下对外提供文件服务，而辅助容器周期性的从外部下载文件存到这个固定目录下。

图 3-1 Pod



实际使用中很少直接创建Pod，而是使用Kubernetes中称为Controller的抽象层来管理Pod实例，例如Deployment和Job。Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力。通常，Controller会使用Pod Template来创建相应的Pod。

创建 Pod

kubernetes中资源可以使用YAML描述（如果您对YAML格式不了解，可以参考**YAML语法**），也可以使用JSON，如下示例描述了一个名为nginx的Pod，这个Pod中包含一个名为container-0的容器，使用nginx:alpine镜像，使用的资源为100m core CPU、200Mi内存。

```
apiVersion: v1                  # Kubernetes的API Version
kind: Pod                       # Kubernetes的资源类型
metadata:
  name: nginx                   # Pod的名称
spec:                           # Pod的具体规格（specification）
  containers:
  - image: nginx:alpine         # 使用的镜像为 nginx:alpine
    name: container-0           # 容器的名称
    resources:                  # 申请容器所需的资源
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
  imagePullSecrets:             # 拉取镜像使用的证书，在CCE上必须为default-secret
  - name: default-secret
```

如上面YAML的注释，YAML描述文件主要为如下部分：

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

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

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

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

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

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

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

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

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

您还可以使用kubectl describe命令查看Pod的详情。

```
$ kubectl describe pod nginx
```

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

Pod的停止与删除有多种方法，比如按名称删除，如下所示。

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

同时删除多个Pod。

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

删除所有Pod。

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

根据Label删除Pod，**Label**详细内容将会在下一个章节介绍。

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

使用环境变量

环境变量是容器运行环境中设定的一个变量。

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

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

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



```
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:                    # 环境变量
    - name: env_key
      value: env_value
  imagePullSecrets:
  - name: default-secret
```

执行如下命令查看容器中的环境变量，可以看到env_key这个环境变量，其值为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
```

环境变量还可以引用ConfigMap和Secret，具体使用方法请参见**在环境变量中引用ConfigMap**和**在环境变量中引用Secret**。

容器启动命令

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

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

实际使用时，只需配置Pod的containers.command参数，该参数是list类型，第一个参数为执行命令，后面均为命令的参数。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    command:                # 启动命令
    - top
    - "-b"
  imagePullSecrets:
   - name: default-secret
```

容器的生命周期

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

- 启动后处理（PostStart）：容器启动后触发。
- 停止前处理（PreStop）：容器停止前触发。

实际使用时，只需配置Pod的lifecycle.postStart或lifecycle.preStop参数，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    lifecycle:
      postStart:                # 启动后处理
       exec:
          command:
          - "/postStart.sh"
      preStop:                  # 停止前处理
       exec:
          command:
          - "/preStop.sh"
  imagePullSecrets:
   - name: default-secret
```

3.2 存活探针（Liveness Probe）

存活探针

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

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

Kubernetes支持如下三种探测机制。

- HTTP GET：向容器发送HTTP GET请求，如果Probe收到2xx或3xx，说明容器是健康的。
- TCP Socket：尝试与容器指定端口建立TCP连接，如果连接成功建立，说明容器是健康的。
- Exec：Probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明容器是健康的。

与存活探针对应的还有一个就绪探针（Readiness Probe），将在**6.4 就绪探针（Readiness Probe）**中会详细介绍。

HTTP GET

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


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

创建这个Pod。

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

如上，这个Probe往容器的80端口发送HTTP GET请求，如果请求不成功，Kubernetes会重启容器。

查看Pod详情。

```
$ 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)
......
```

可以看到Pod当前状态是Running，Restart Count为0，说明没有重启。如果Restart Count不为0，则说明已经重启。

TCP Socket

TCP Socket尝试与容器指定端口建立TCP连接，如果连接成功建立，说明容器是健康的，定义方法如下所示。

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


Exec

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

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

上面定义在容器中执行**cat /tmp/healthy**命令，如果成功执行并返回0，则说明容器是健康的。上面定义中，30秒后命令会删除/tmp/healthy，这会导致Liveness Probe判定Pod处于不健康状态，然后会重启容器。

Liveness Probe 高级配置

上面liveness-http的describe命令回显中有如下行。

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

这一行表示Liveness Probe的具体参数配置，其含义如下：

- delay：延迟，delay=0s，表示在容器启动后立即开始探测，没有延迟时间
- timeout：超时，timeout=1s，表示容器必须在1s内进行响应，否则这次探测记作失败
- period：周期，period=10s，表示每10s探测一次容器
- success：成功，#success=1，表示连续1次成功后记作成功
- failure：失败，#failure=3，表示连续3次失败后会重启容器

以上存活探针表示：容器启动后立即进行探测，如果1s内容器没有给出回应则记作探测失败。每次间隔10s进行一次探测，在探测连续失败3次后重启容器。

这些是创建时默认设置的，您也可以手动配置，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10    # 容器启动后多久开始探测
```



```
timeoutSeconds: 2        # 表示容器必须在2s内做出相应反馈给probe，否则视为探测失败
periodSeconds: 30        # 探测周期，每30s探测一次
successThreshold: 1      # 连续探测1次成功表示成功
failureThreshold: 3      # 连续探测3次失败表示失败
```

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

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

配置有效的 Liveness Probe

- **Liveness Probe应该检查什么**

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

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

- **Liveness Probe必须轻量**

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

3.3 Label：组织 Pod 的利器

为什么需要 Label

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

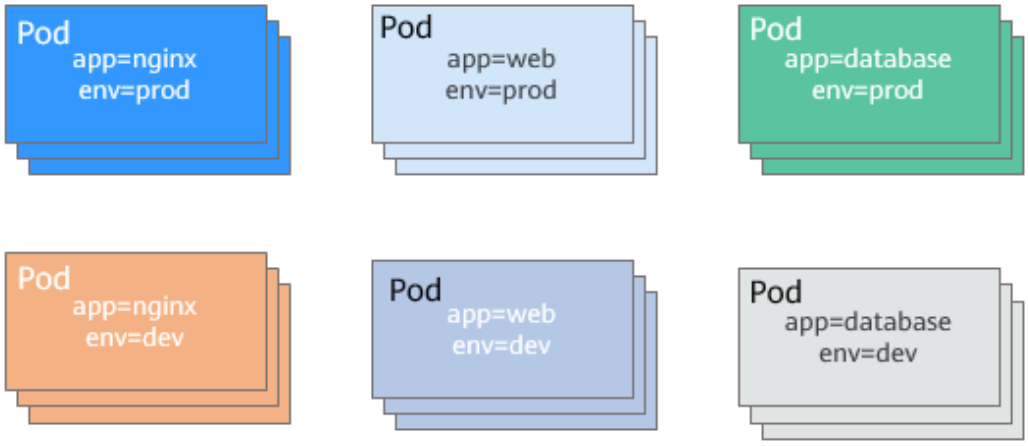
Label的具体形式是key-value的标记对，可以在创建资源的时候设置，也可以在后期添加和修改。

以Pod为例，当Pod变得多起来后，就显得杂乱且难以管理，如下图所示。

图 3-2 没有分类组织的 Pod

如果我们为Pod打上不同标签，那情况就完全不同了，如下图所示。

图 3-3 使用 Label 组织的 Pod



添加 Label

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

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:                  # 为Pod设置两个Label
    app: nginx
    env: prod
spec:
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
```



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

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

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

还可以使用-L只查询某些固定的Label。

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

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

```
$ kubectl label 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
```

修改 Label

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

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

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

3.4 Namespace：资源分组

为什么需要 Namespace

Label虽然好，但只用Label的话，那Label会非常多，有时候会有重叠，而且每次查询之类的动作都带一堆Label非常不方便。Kubernetes提供了Namespace来做资源组织和划分，使用多Namespace可以将包含很多组件的系统分成不同的组。Namespace也可以用来做多租户划分，这样多个团队可以共用一个集群，使用的资源用Namespace划分开。

不同的Namespace下面可以有相同的名字，Kubernetes中大部分资源可以用Namespace划分，不过有些资源不行，它们属于全局资源，不属于某一个Namespace，后面会逐步接触到。

通过如下命令可以查询到当前集群下的Namespace。

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

到目前为止，我们都是在default Namespace下操作，当使用kubectl get而不指定Namespace时，默认为default Namespace。

看下kube-system下面有些什么东西。

```
$ 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
```

可以看到kube-system有很多Pod，其中coredns是用于做服务发现、everest-csi是用于对接华为云存储服务、icagent是用于对接华为云监控系统。

这些通用的、必须的应用放在kube-system这个命名空间中，能够做到与其他Pod之间隔离，在其他命名空间中不会看到kube-system这个命名空间中的东西，不会造成影响。

创建 Namespace

使用如下方式定义Namespace。

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

使用kubectl命令创建。

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

您还可以使用kubectl create namespace命令创建。

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

在指定Namespace下创建资源。

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

这样在custom-namespace下，就创建了一个名为nginx的Pod。

Namespace 的隔离说明

Namespace只能做到组织上划分，对运行的对象来说，它不能做到真正的隔离。举例来说，如果两个Namespace下的Pod知道对方的IP，而Kubernetes依赖的底层网络没有提供Namespace之间的网络隔离的话，那这两个Pod就可以互相访问。

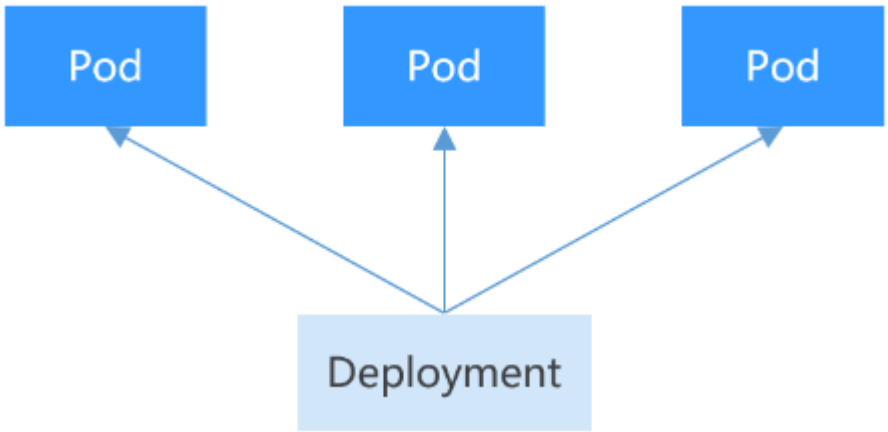
4 Pod 的编排与调度

4.1 Deployment

什么是 Deployment

在3.1 Pod：Kubernetes中的最小调度对象这个章节介绍了Pod，Pod是Kubernetes创建或部署的最小单位，但是Pod是被设计为相对短暂的一次性实体，Pod可以被驱逐（当节点资源不足时）、随着集群的节点崩溃而消失。Kubernetes提供了Controller（控制器）来管理Pod，Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力，其中最为常用的就是Deployment。

图 4-1 Deployment



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

Deployment集成了上线部署、滚动升级、创建副本、恢复上线的功能，在某种程度上，Deployment实现无人值守的上线，大大降低了上线过程的复杂性和操作风险。

创建 Deployment

以下示例为创建一个名为nginx的Deployment负载，使用nginx:latest镜像创建两个Pod，每个Pod占用100m core CPU、200Mi内存。

```
apiVersion: apps/v1      # 注意这里与Pod的区别，Deployment是apps/v1而不是v1
kind: Deployment         # 资源类型为Deployment
metadata:
  name: nginx            # Deployment的名称
spec:
  replicas: 2            # Pod的数量，Deployment会确保一直有2个Pod运行
  selector:              # Label Selector
    matchLabels:
      app: nginx
  template:              # Pod的定义，用于创建Pod，也称为Pod template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

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

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

使用kubectl get查看Deployment和Pod，可以看到**READY**值为2/2，前一个2表示当前有2个Pod运行，后一个2表示期望有2个Pod，**AVAILABLE**为2表示有2个Pod是可用的。

```
$ 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
```

Deployment 如何控制 Pod

继续查询Pod，如下所示。

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

如果删掉一个Pod，您会发现立马会有一个新的Pod被创建出来，如下所示，这就是前面所说的Deployment会确保有2个Pod在运行，如果删掉一个，Deployment会重新创建一个，如果某个Pod故障或有其他问题，Deployment会自动拉起这个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          21s
```

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

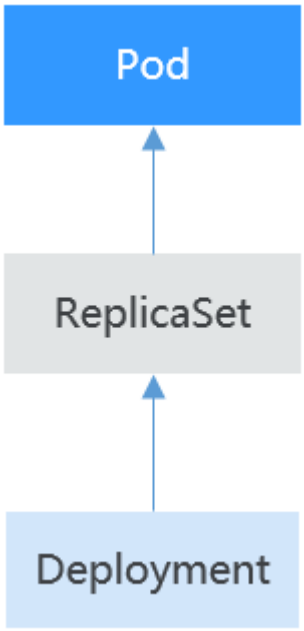
您也许会发现这两个后缀中前面一部分是相同的，都是7f98958cdf，这是因为Deployment不是直接控制Pod的，Deployment是通过一种名为ReplicaSet的控制器控制Pod，通过如下命令可以查询ReplicaSet，其中rs是ReplicaSet的缩写。

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

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

Deployment控制Pod的方式如图4-2所示，Deployment控制ReplicaSet，ReplicaSet控制Pod。

图 4-2 Deployment 通过 ReplicaSet 控制 Pod



如果使用kubectl describe命令查看Deployment的详情，您就可以看到ReplicaSet，如下所示，可以看到有一行NewReplicaSet: nginx-7f98958cdf (2/2 replicas created)，而且Events里面事件确是把ReplicaSet的实例扩容到2个。在实际使用中您也许不会直接操作ReplicaSet，但了解Deployment通过控制ReplicaSet来控制Pod会有助于您定位问题。

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

...
```



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

升级

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

Deployment可以设置不同的升级策略，有如下两种。

- RollingUpdate：滚动升级，即逐步创建新Pod再删除旧Pod，为默认策略。
- Recreate：替换升级，即先把当前Pod删掉再重新创建Pod。

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

```
$ kubectl edit deploy nginx

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

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

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

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

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

在前面的例子中，由于spec.replicas是2，如果maxSurge和maxUnavailable都为默认值25%，那实际升级过程中，maxSurge允许最多3个Pod存在（向上取整，2*1.25=2.5，取整为3），而maxUnavailable则不允许有Pod Unavailable（向上取整，2*0.75=1.5，取整为2），也就是说在升级过程中，一直会有2个Pod处于运行状态，每次新建一个Pod，等这个Pod创建成功后再删掉一个旧Pod，直至Pod全部为新Pod。

回滚

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

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

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

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

4.2 StatefulSet

为什么需要 StatefulSet

在**4.1 Deployment**中讲到了Deployment，Deployment控制器下的Pod都有个共同特点，那就是每个Pod除了名称和IP地址不同，其余完全相同。需要的时候，Deployment可以通过Pod模板创建新的Pod；不需要的时候，Deployment就可以删除任意一个Pod。

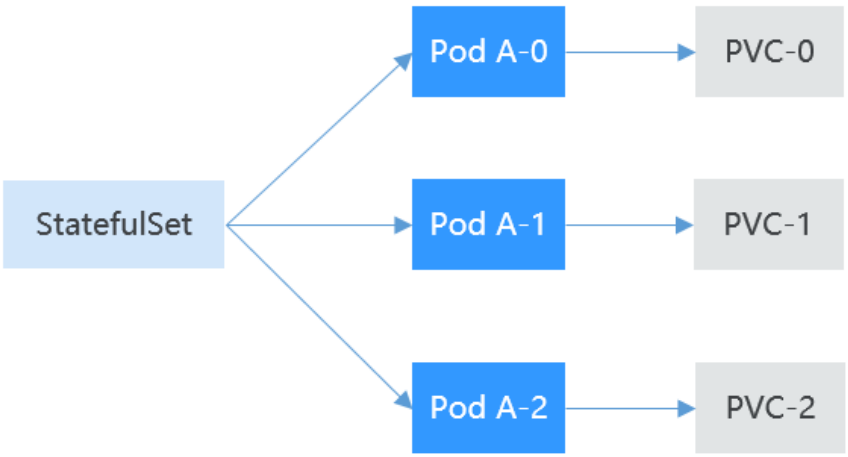
但是在某些场景下，这并不满足需求，比如有些分布式的场景，要求每个Pod都有自己单独的状态时，比如分布式数据库，每个Pod要求有单独的存储，这时Deployment就不能满足需求了。

详细分析下有状态应用的需求，分布式有状态的特点主要是应用中每个部分的角色不同（即分工不同），比如数据库有主备，Pod之间有依赖，对应到Kubernetes中就是对Pod有如下要求：

- Pod能够被别的Pod找到，这就要求Pod有固定的标识。
- 每个Pod有单独存储，Pod被删除恢复后，读取的数据必须还是以前那份，否则状态就会不一致。

Kubernetes提供了StatefulSet来解决这个问题，其具体如下：

1. StatefulSet给每个Pod提供固定名称，Pod名称增加从0-N的固定后缀，Pod重新调度后Pod名称和HostName不变。
2. StatefulSet通过Headless Service给每个Pod提供固定的访问域名，Service的概念会在**6.2 Service**中详细介绍。
3. StatefulSet通过创建固定标识的PVC保证Pod重新调度后还是能访问到相同的持久化数据。



下面将通过创建StatefulSet来体验StatefulSet的这些特性。

创建 Headless Service

如前所述，创建Statefulset需要一个Headless Service用于Pod访问，Service的概念会在6.2 Service中详细介绍，这里先介绍Headless Service的创建方法。

使用如下文件描述Headless Service，其中：

- spec.clusterIP：必须设置为None，表示Headless Service。
- spec.ports.port：Pod间通信端口号。
- spec.ports.name：Pod间通信端口名称。

```
apiVersion: v1
kind: Service      # 对象类型为Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Pod间通信的端口名称
      port: 80        # Pod间通信的端口号
  selector:
    app: nginx        # 选择标签为app:nginx的Pod
  clusterIP: None     # 必须设置为None，表示Headless Service
```

执行如下命令创建Headless Service。

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

创建完成后可以查询Service。

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

创建 Statefulset

Statefulset的YAML定义与其他对象基本相同，主要有两个差异点：

- serviceName指定了Statefulset使用哪个Headless Service，需要填写Headless Service的名称。
- volumeClaimTemplates是用来申请持久化声明PVC ，这里定义了一个名为data的模板，它会为每个Pod创建一个PVC，storageClassName指定了持久化存储的类型，在7.2 PV、PVC和StorageClass会详细介绍；volumeMounts是为Pod挂载存储。当然如果不需要存储的话可以删除volumeClaimTemplates和volumeMounts字段。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                    # headless service的名称
  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:                          # Pod挂载的存储
        - name:  data
          mountPath:  /usr/share/nginx/html     # 存储挂载到/usr/share/nginx/html
    imagePullSecrets:
      - name: default-secret
volumeClaimTemplates:
- metadata:
    name: data
  spec:
    accessModes:
    - ReadWriteMany
    resources:
      requests:
        storage: 1Gi
    storageClassName: csi-nas                  # 持久化存储的类型
```

执行如下命令创建。

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

命令执行后，查询一下StatefulSet和Pod，可以看到Pod的名称后缀从0开始到2，逐个递增。

```
# 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
```

此时如果手动删除nginx-1这个Pod，然后再次查询Pod，可以看到StatefulSet重新创建了一个名称相同的Pod，通过创建时间5s可以看出nginx-1是刚刚创建的。

```
# 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
```

进入容器查看容器的hostname，可以看到同样是nginx-0、nginx-1和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
```

同时可以看一下StatefulSet创建的PVC，可以看到这些PVC，都以“PVC名称-StatefulSet名称-编号”的方式命名，并且处于Bound状态。

```
# 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
```

StatefulSet 的网络标识

StatefulSet创建后，可以看下Pod是有固定名称的，那Headless Service是如何起作用的呢，那就是使用DNS，为Pod提供固定的域名，这样Pod间就可以使用域名访问，即便Pod被重新创建而导致Pod的IP地址发生变化，这个域名也不会发生变化。

Headless Service创建后，每个Pod的IP都会有下面格式的域名。

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

例如上面的三个Pod的域名就是：

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

实际访问时可以省略后面的**.<namespace>.svc.cluster.local**。

下面命令会使用tutum/dnsutils镜像创建一个Pod，进入这个Pod的容器，使用nslookup命令查看Pod对应的域名，可以发现能解析出Pod的IP地址。这里可以看到DNS服务器的地址是10.247.3.10，这是在创建CCE集群时默认安装CoreDNS插件，用于提供DNS服务，后续在**6 Kubernetes网络**会详细介绍CoreDNS的作用。

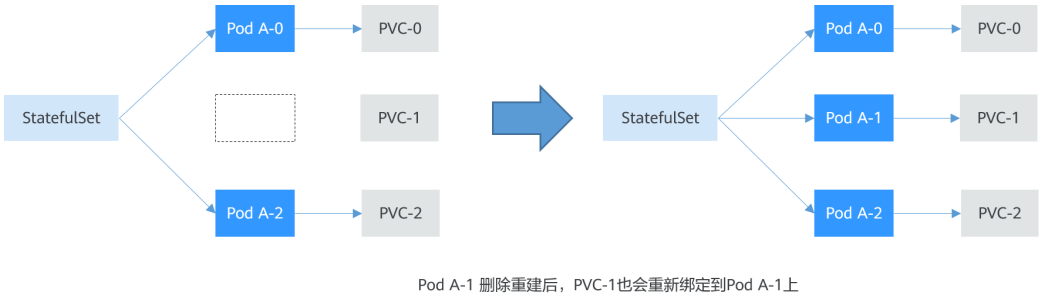
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # 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
```

此时如果手动删除这两个Pod，查询被StatefulSet重新创建的Pod的IP，然后使用nslookup命令解析Pod的域名，可以发现nginx-0.nginx和nginx-1.nginx仍然能解析到对应的Pod。这就保证了StatefulSet网络标识不变。

StatefulSet 存储状态

上面说了StatefulSet可以通过PVC做持久化存储，保证Pod重新调度后还是能访问到相同的持久化数据，在删除Pod时，PVC不会被删除。

图 4-3 StatefulSet 的 Pod 重建过程



下面将通过实际操作验证这一点是如何做到的，执行下面的命令，在nginx-1的目录/usr/share/nginx/html中写入一些内容，例如将index.html的内容修改为“hello world”。

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

修改完后，如果在Pod中访问“http://localhost”，那就会返回“hello world”。

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

此时如果手动删除nginx-1这个Pod，然后再次查询Pod，可以看到StatefulSet重新创建了一个名称相同的Pod，通过创建时间4s可以看出nginx-1是刚刚创建的。

```
# 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
```

再次访问该Pod的index.html页面，会发现仍然返回“hello world”，这说明这个Pod仍然是访问相同的存储。

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

4.3 Job 和 CronJob

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

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

任务负载的这种用完即停止的特性特别适合一次性任务，比如持续集成。

创建 Job

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

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

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

表 4-1 任务类型

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

创建 CronJob

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

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

CronJob的格式从前到后就是：

- Minute

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

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

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

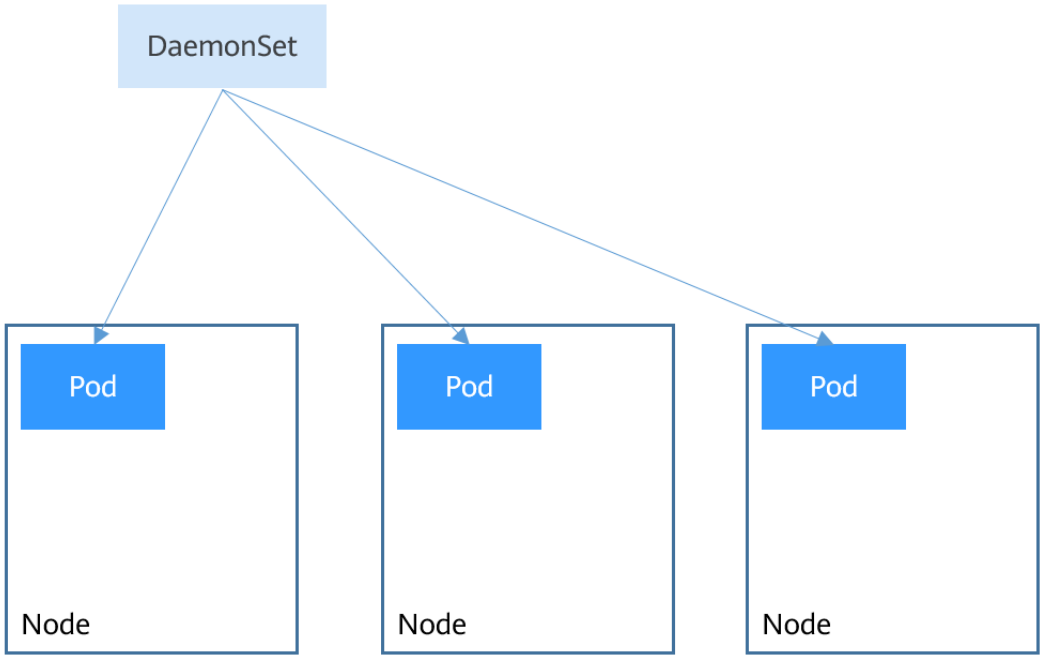
更详细的CronJob格式说明请参见**https://zh.wikipedia.org/wiki/Cron**。

4.4 DaemonSet

DaemonSet是这样一种对象（守护进程），它在集群的每个节点上运行一个Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资源监控等，这类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes的kube-proxy。

DaemonSet跟节点相关，如果节点异常，也不会在其他节点重新创建。

图 4-4 DaemonSet



下面是一个DaemonSet的示例。

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
  labels:
    app: nginx-daemonset
spec:
  selector:
```

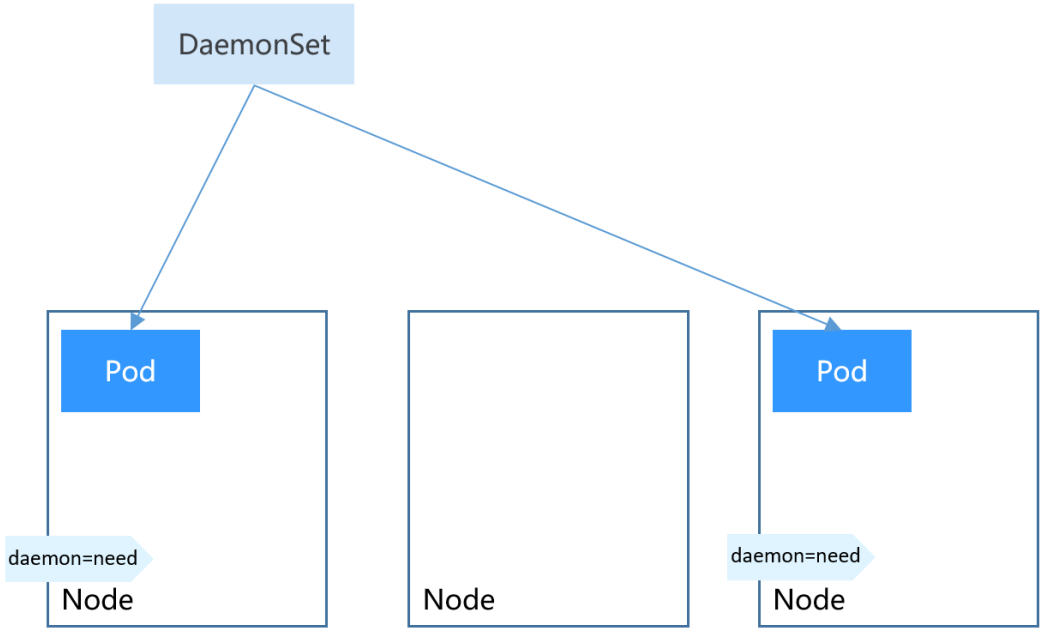


```
  matchLabels:
    app: nginx-daemonset
template:
  metadata:
    labels:
      app: nginx-daemonset
  spec:
    nodeSelector:              # 节点选择，当节点拥有daemon=need时才在节点上创建Pod
      daemon: need
    containers:
    - name: nginx-daemonset
      image: nginx:alpine
      resources:
        limits:
          cpu: 250m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 512Mi
    imagePullSecrets:
    - name: default-secret
```

这里可以看出没有Deployment或StatefulSet中的replicas参数，因为是每个节点固定一个。

Pod模板中有个nodeSelector，指定了只在有“daemon=need”的节点上才创建Pod，如下图所示，DaemonSet只在指定标签的节点上创建Pod。如果需要在每一个节点上创建Pod可以删除该标签。

图 4-5 DaemonSet 在指定标签的节点上创建 Pod



创建DaemonSet：

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

查询发现nginx-daemonset没有Pod创建。

```
$ 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.
```

这是因为节点上没有daemon=need这个标签，使用如下命令可以查询节点的标签。

```
$ 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 ...
```

给192.168.0.212这个节点打上标签，然后再查询，发现已经创建了一个Pod，并且这个Pod是在192.168.0.212这个节点上。

```
$ 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
```

再给192.168.0.94这个节点打上标签，发现又创建了一个Pod：

```
$ 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
```

如果修改掉192.168.0.94节点的标签，可以发现DaemonSet会删除这个节点上的Pod。

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

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

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

4.5 亲和与反亲和调度

在**4.4 DaemonSet**中讲到使用nodeSelector选择Pod要部署的节点，其实Kubernetes还支持更精细、更灵活的调度机制，那就是亲和（affinity）与反亲和（anti-affinity）调度。

Kubernetes支持节点和Pod两个层级的亲和与反亲和。通过配置亲和与反亲和规则，可以允许您指定硬性限制或者偏好，例如将前台Pod和后台Pod部署在一起、某类应用部署到某些特定的节点、不同应用部署到不同的节点等等。

Node Affinity（节点亲和）

您肯定也猜到了亲和性规则的基础肯定也是标签，先来看一下CCE集群中节点上有些什么标签。

```
$ 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
```

这些标签都是在创建节点的时候CCE会自动添加上的，下面介绍几个在调度中会用到比较多的标签。

- failure-domain.beta.kubernetes.io/region：表示节点所在的区域，如果上面这个节点标签值为cn-east-3，表示节点在上海一区域。
- failure-domain.beta.kubernetes.io/zone：表示节点所在的可用区（availability zone）。
- kubernetes.io/hostname：节点的hostname。

另外在**3.3 Label：组织Pod的利器**章节还介绍自定义标签，通常情况下，对于一个大型Kubernetes集群，肯定会根据业务需要定义很多标签。

在**4.4 DaemonSet**中介绍了nodeSelector，通过nodeSelector可以让Pod只部署在具有特定标签的节点上。如下所示，Pod只会部署在拥有gpu=true这个标签的节点上。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeSelector:              # 节点选择，当节点拥有gpu=true时才在节点上创建Pod
    gpu: true
...
```

通过节点亲和性规则配置，也可以做到同样的事情，如下所示。

```
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"
```

看起来这要复杂很多，但这种方式可以得到更强的表达能力，后面会进一步介绍。

这里affinity表示亲和，nodeAffinity表示节点亲和，requiredDuringSchedulingIgnoredDuringExecution非常长，不过可以将这个分作两段来看：

- 前半段requiredDuringScheduling表示下面定义的规则必须强制满足（require）。
- 后半段IgnoredDuringExecution表示不会影响已经在节点上运行的Pod，目前Kubernetes提供的规则都是以IgnoredDuringExecution结尾的，因为当前的节点亲缘性规则只会影响正在被调度的pod，最终，kubernetes也会支持RequiredDuringExecution，即去除节点上的某个标签，那些需要节点包含该标签的pod将会被剔除。

另外操作符operator的值为In，表示标签值需要在values的列表中，其他operator取值如下。

- NotIn：标签的值不在某个列表中
- Exists：某个标签存在
- DoesNotExist：某个标签不存在
- Gt：标签的值大于某个值（字符串比较）
- Lt：标签的值小于某个值（字符串比较）

需要说明的是并没有nodeAntiAffinity（节点反亲和），因为NotIn和DoesNotExist可以提供相同的功能。

下面来验证这段规则是否生效，首先给192.168.0.212这个节点打上gpu=true的标签。

```
$ 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
```

创建这个Deployment，可以发现所有的Pod都部署在了192.168.0.212这个节点上。

```
$ 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
```

节点优先选择规则

上面讲的requiredDuringSchedulingIgnoredDuringExecution是一种**强制**选择的规则，节点亲和还有一种优先选择规则，即preferredDuringSchedulingIgnoredDuringExecution，表示会根据规则**优先**选择哪些节点。

为演示这个效果，先为上面的集群添加一个节点，且这个节点跟另外三个节点不在同一个可用区，创建完之后查询节点的可用区标签，如下所示，新添加的节点在cn-east-3c这个可用区。

```
$ 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
```

下面定义一个Deployment，要求Pod优先部署在可用区cn-east-3a的节点上，可以像下面这样定义，使用preferredDuringSchedulingIgnoredDuringExecution规则，给cn-east-3a设置权重（weight）为80，而gpu=true权重为20，这样Pod就优先部署在cn-east-3a的节点上。

```
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
```



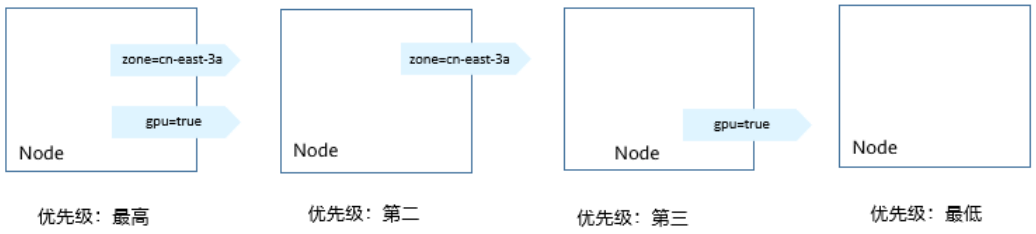
```
            values:
            - cn-east-3a
        - weight: 20
          preference:
            matchExpressions:
            - key: gpu
              operator: In
              values:
              - "true"
```

来看实际部署后的情况，可以看到部署到192.168.0.212这个节点上的Pod有5个，而192.168.0.100上只有2个。

```
$ 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
```

上面这个例子中，对于节点排序优先级如下所示，有个两个标签的节点排序最高，只有cn-east-3a标签的节点排序第二（权重为80），只有gpu=true的节点排序第三，没有的节点排序最低。

图 4-6 优先级排序顺序



这里您看到Pod并没有调度到192.168.0.94这个节点上，这是因为这个节点上部署了很多其他Pod，资源使用较多，所以并没有往这个节点上调度，这也侧面说明preferredDuringSchedulingIgnoredDuringExecution是优先规则，而不是强制规则。

Pod Affinity（Pod 亲和）

节点亲和的规则只能影响Pod和节点之间的亲和，Kubernetes还支持Pod和Pod之间的亲和，例如将应用的前端和后端部署在一起，从而减少访问延迟。Pod亲和同样有requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution两种规则。

来看下面这个例子，假设有个应用的后端已经创建，且带有app=backend的标签。

```
$ 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
```

将前端frontend的pod部署在backend一起时，可以做如下Pod亲和规则配置。


```
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:
              matchLabels:
                app: backend
```

创建frontend然后查看，可以看到frontend都创建到跟backend一样的节点上了。

```
$ 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
```

这里有个topologyKey字段，意思是先圈定topologyKey指定的范围，然后再选择下面规则定义的内容。这里每个节点上都有kubernetes.io/hostname，所以看不出topologyKey起到的作用。

如果backend有两个Pod，分别在不同的节点上。

```
$ 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
```

给192.168.0.97和192.168.0.94打一个prefer=true的标签。

```
$ 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
```

将podAffinity的topologyKey定义为prefer。

```
    affinity:
      podAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - topologyKey: prefer
          labelSelector:
            matchLabels:
              app: backend
```

调度时，先圈定拥有prefer标签的节点，这里也就是192.168.0.97和192.168.0.94，然后再匹配app=backend标签的Pod，从而frontend就会全部部署在192.168.0.97上。

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

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

Pod AntiAffinity（Pod 反亲和）

前面讲了Pod的亲和，通过亲和将Pod部署在一起，有时候需求却恰恰相反，需要将Pod分开部署，例如Pod之间部署在一起会影响性能的情况。

下面例子中定义了反亲和规则，这个规则表示Pod不能调度到拥有app=frontend标签Pod的节点上，也就是下面将frontend分别调度到不同的节点上（每个节点只有一个Pod）。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:   frontend
  labels:
    app:  frontend
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 5
  template:
    metadata:
      labels:
        app:  frontend
    spec:
      containers:
      - image:  nginx:alpine
        name:  frontend
        resources:
          requests:
            cpu:  100m
            memory:  200Mi
          limits:
            cpu:  100m
            memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAntiAffinity:
```



```
      requiredDuringSchedulingIgnoredDuringExecution:
      - topologyKey: kubernetes.io/hostname
        labelSelector:
          matchLabels:
            app: frontend
```

创建并查看，可以看到每个节点上只有一个frontend的Pod，还有一个在Pending，因为在部署第5个时4个节点上都有了app=frontend的Pod，所以第5个一直是Pending。

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

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


5 配置管理

5.1 ConfigMap

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

通过ConfigMap可以方便的做到配置解耦，使得不同环境有不同的配置。

创建 ConfigMap

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

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

在环境变量中引用 ConfigMap

ConfigMap最为常见的使用方式就是在环境变量和Volume中引用。

例如下面例子中，引用了configmap-test的property_1，将其作为环境变量EXAMPLE_PROPERTY_1的值，这样容器启动后里面EXAMPLE_PROPERTY_1的值就是property_1的值，即“Hello”。

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



```
      memory: 200Mi
  env:
  - name: EXAMPLE_PROPERTY_1
    valueFrom:
      configMapKeyRef:          # 引用ConfigMap
        name: configmap-test
        key: property_1
imagePullSecrets:
- name: default-secret
```

在 Volume 中引用 ConfigMap

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

如下示例中，创建一个名为vol-configmap的Volume，这个Volume引用名为“configmap-test”的ConfigMap，再将Volume挂载到容器的“/tmp”路径下。Pod创建成功后，在容器的“/tmp”路径下，就有两个文件property_1和property_2，它们的值分别为“Hello”和“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           # 挂载名为vol-configmap的Volume
      mountPath: "/tmp"
  imagePullSecrets:
  - name: default-secret
  volumes:
  - name: vol-configmap
    configMap:                      # 引用ConfigMap
      name: configmap-test
```

5.2 Secret

Secret是一种加密存储的资源对象，您可以将认证信息、证书、私钥等保存在Secret中，而不需要把这些敏感数据暴露到镜像或者Pod定义中，从而更加安全和灵活。

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

Base64 编码

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

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

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

创建 Secret

如下示例中定义的Secret中包含两条Key-Value。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  key1: aGVsbG8gd29ybGQ=   # "hello world" Base64编码后的值
  key2: MzMwNg==           # "3306" Base64编码后的值
```

在环境变量中引用 Secret

Secret最常见的用法是作为环境变量注入到容器中，如下示例。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx: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
```

在 Volume 中引用 Secret

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

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

```
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          # 挂载名为vol-secret的Volume
```



```
    mountPath: "/tmp"
imagePullSecrets:
- name: default-secret
volumes:
- name: vol-secret
  secret:                    # 引用Secret
    secretName: mysecret
```

进入Pod容器中，可以在/tmp目录下发现key1和key2两个文件，并看到文件中的值是base64解码后的值，分别为“hello world”和“3306”。

6 Kubernetes 网络

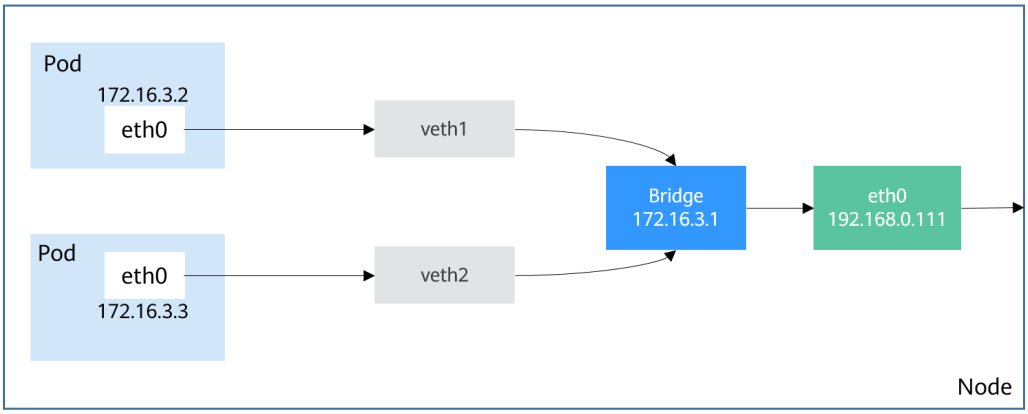
6.1 容器网络

Kubernetes本身并不负责网络通信，Kubernetes提供了容器网络接口CNI（Container Network Interface），具体的网络通信交给CNI插件来负责，开源的CNI插件非常多，像Flannel、Calico等，华为云CCE也专门为Kubernetes定制了CNI插件（Canal和Yangste），使得Kubernetes可以使用华为云VPC网络。

Kubernetes虽然不负责网络，但要求集群中的Pod能够互相通信，且Pod必须通过非NAT网络连接，即收到的数据包的源IP就是发送数据包Pod的IP。同时Pod与节点之间的通信也是通过非NAT网络。但是Pod访问集群外部时源IP会被修改成节点的IP。

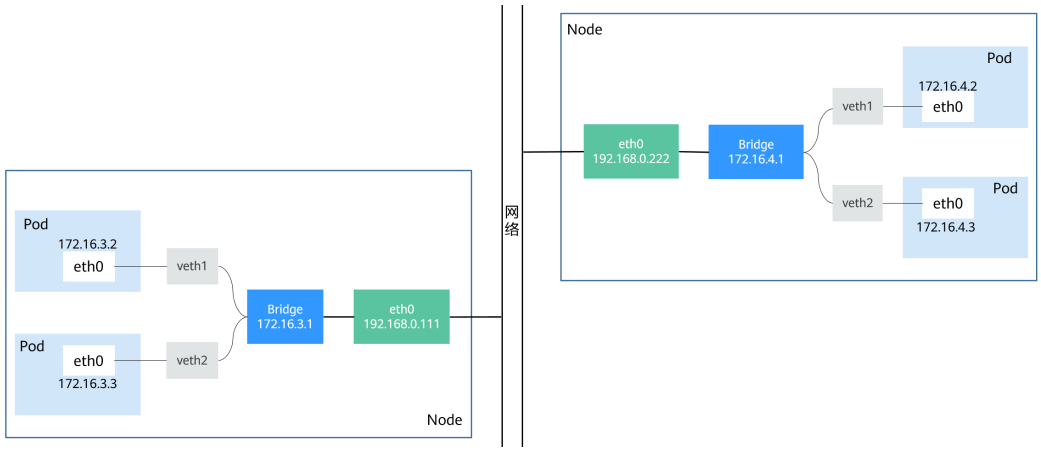
Pod内部是通过虚拟Ethernet接口对（Veth pair）与Pod外部连接，Veth pair就像一根网线，一端留在Pod内部，一端在Pod之外。而同一个节点上的Pod通过网桥（Linux Bridge）通信，如下图所示。

图 6-1 同一个节点中的 Pod 通信



不同节点间的网桥连接有很多种方式，这跟具体实现相关。但集群要求Pod的地址唯一，所以跨节点的网桥通常使用不同的地址段，以防止Pod的IP地址重复。

图 6-2 不同节点上的 Pod 通信



以上就是容器网络底层视图，后面将进一步介绍Kubernetes是如何在此基础上向用户提供访问方案，具体请参见6.2 Service和6.3 Ingress。

6.2 Service

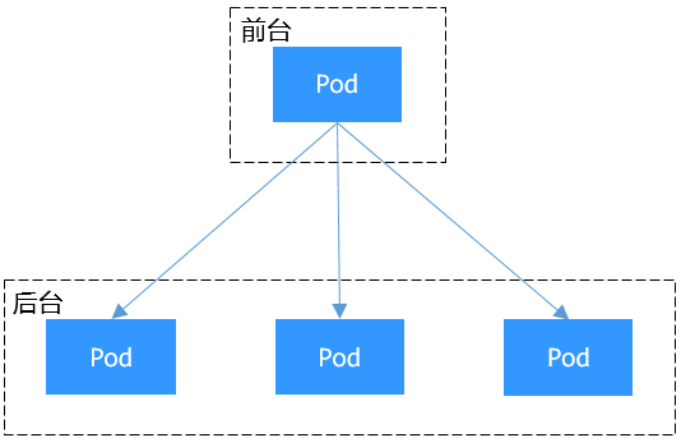
直接访问 Pod 的问题

Pod创建完成后，如何访问Pod呢？直接访问Pod会有如下几个问题：

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

举个例子，假设有这样一个应用程序，使用Deployment创建了前台和后台，前台会调用后台做一些计算处理，如图6-3所示。后台运行了3个Pod，这些Pod是相互独立且可被替换的，当Pod出现状况被重建时，新建的Pod的IP地址是新IP，前台的Pod无法直接感知。

图 6-3 Pod 间访问

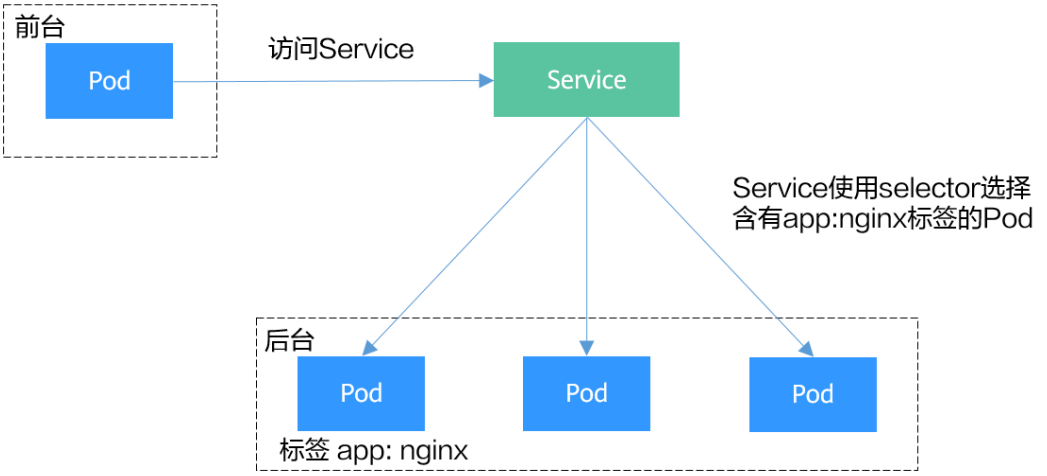


使用 Service 解决 Pod 的访问问题

Kubernetes中的Service对象就是用来解决上述Pod访问问题的。Service有一个固定IP地址（在创建CCE集群时有一个服务网段的设置，这个网段专门用于给Service分配IP地址），Service将访问它的流量转发给Pod，具体转发给哪些Pod通过Label来选择，而且Service可以给这些Pod做负载均衡。

那么对于上面的例子，为后台添加一个Service，通过Service来访问Pod，这样前台Pod就无需感知后台Pod的变化，如图6-4所示。

图 6-4 通过 Service 访问 Pod



创建后台 Pod

首先创建一个3副本的Deployment，即3个Pod，且Pod上带有标签“app: nginx”，具体如下所示。

```
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
```


创建 Service

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

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

```
apiVersion: v1
kind: Service
metadata:
  name: nginx        # Service的名称
spec:
  selector:          # Label Selector，选择包含app=nginx标签的Pod
    app: nginx
  ports:
  - name: service0
    targetPort: 80   # Pod的端口
    port: 8080       # Service对外暴露的端口
    protocol: TCP    # 转发协议类型，支持TCP和UDP
  type: ClusterIP    # Service的类型
```

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

```
$ kubectl create -f nginx-svc.yaml
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
```

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

下面创建一个Pod并进入容器，使用ClusterIP访问Pod，可以看到能直接返回内容。

```
$ 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>
...
```

使用 ServiceName 访问 Service

通过DNS进行域名解析后，可以使用“ServiceName:Port”访问Service，这也是Kubernetes中最常用的一种使用方式。在创建CCE集群的时候，会默认要求安装CoreDNS插件，在kube-system命名空间下可以查看到CoreDNS的Pod。

```
$ 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
```

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

访问时通过nginx.<namespace>.svc.cluster.local访问，其中nginx为Service的名称，<namespace>为命名空间名称，svc.cluster.local为域名后缀，在实际使用中，在同一个命名空间下可以省略<namespace>.svc.cluster.local，直接使用ServiceName即可。

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

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

下面创建一个Pod并进入容器，查询nginx域名的地址，可以发现是解析出nginx这个Service的IP地址10.247.124.252；同时访问Pod的域名，可以看到能直接返回内容。

```
$ 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>
...
```

Service 是如何做到服务发现的

前面说到有了Service后，无论Pod如何变化，Service都能够发现到Pod。

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

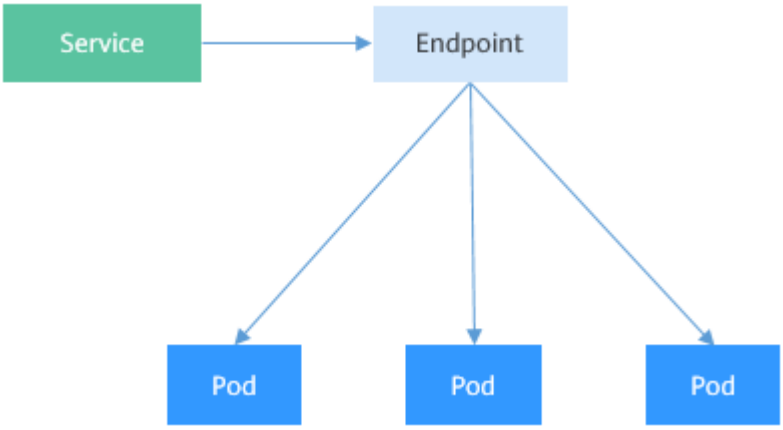
```
$ kubectl describe svc nginx
Name:              nginx
......
Endpoints:         172.16.2.132:80,172.16.3.6:80,172.16.3.7:80
......
```

可以看到一个Endpoints，Endpoints同样也是Kubernetes的一种资源对象，可以查询得到。Kubernetes正是通过Endpoints监控到Pod的IP，从而让Service能够发现Pod。

```
$ kubectl get endpoints
NAME         ENDPOINTS                                      AGE
kubernetes   192.168.0.127:5444                             7h19m
nginx        172.16.2.132:80,172.16.3.6:80,172.16.3.7:80    5h48m
```

这里的172.16.2.132:80是Pod的IP:Port，通过如下命令可以查看到Pod的IP，与上面的IP一致。

```
$ kubectl get 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
```

如果删除一个Pod，Deployment会将Pod重建，新的Pod IP会发生变化。

```
$ 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
```

再次查看Endpoints，会发现Endpoints的内容随着Pod发生了变化。

```
$ 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
```

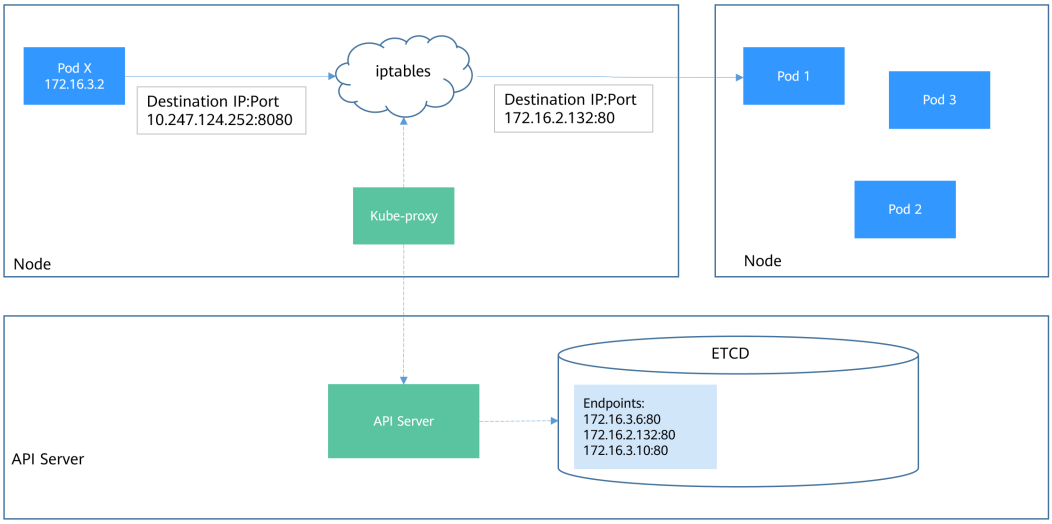
下面进一步了解这又是如何实现的。

在**Kubernetes集群架构**中介绍过Node节点上的kube-proxy，实际上Service相关的事情都由节点上的kube-proxy处理。在Service创建时Kubernetes会分配IP给Service，同时通过API Server通知所有kube-proxy有新Service创建了，kube-proxy收到通知后通过iptables记录Service和IP/端口对的关系，从而让Service在节点上可以被查询到。

下图是一个实际访问Service的图示，Pod X访问Service（10.247.124.252:8080），在往外发数据包时，在节点上根据iptables规则目的IP:Port被随机替换为Pod1的IP:Port，从而通过Service访问到实际的Pod。

除了记录Service和IP/端口对的关系，kube-proxy还会监控Service和Endpoint的变化，从而保证Pod重建后仍然能通过Service访问到Pod。

图 6-5 Pod X 访问 Service 的过程



Service 的类型与使用场景

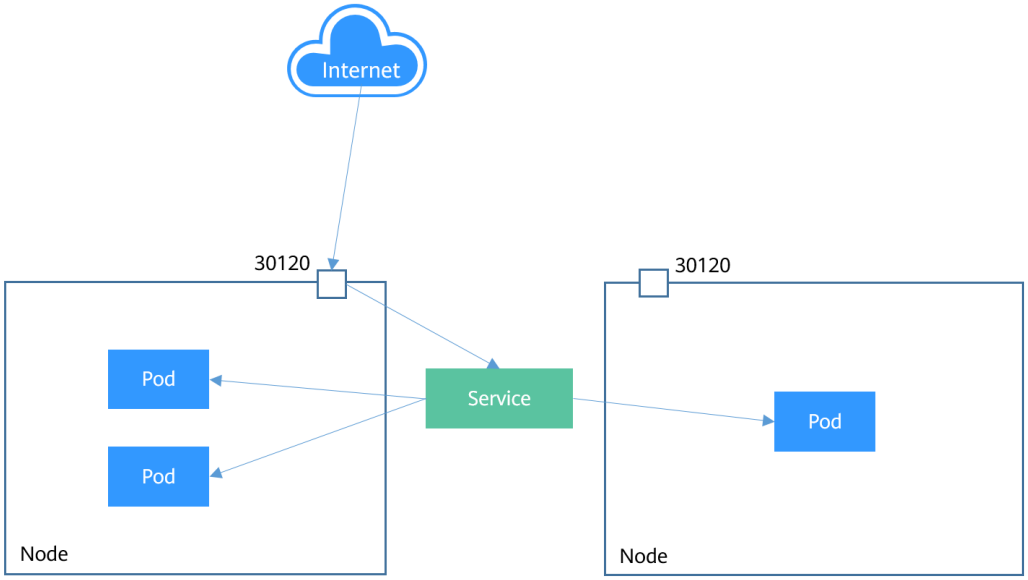
Service的类型除了ClusterIP还有NodePort、LoadBalancer和None，这几种类型的Service有着不同的用途。

- ClusterIP：用于在集群内部互相访问的场景，通过ClusterIP访问Service。
- NodePort：用于从集群外部访问的场景，通过节点上的端口访问Service，详细介绍请参见**NodePort类型的Service**。
- LoadBalancer：用于从集群外部访问的场景，其实是NodePort的扩展，通过一个特定的LoadBalancer访问Service，这个LoadBalancer将请求转发到节点的NodePort，而外部只需要访问LoadBalancer，详细介绍请参见**LoadBalancer类型的Service**。
- None：用于Pod间的互相发现，这种类型的Service又叫Headless Service，详细介绍请参见**Headless Service**。

NodePort 类型的 Service

NodePort类型的Service可以让Kubemetes集群每个节点上保留一个相同的端口， 外部访问连接首先访问节点IP:Port，然后将这些连接转发给服务对应的Pod。如下图所示。

图 6-6 NodePort Service



下面是一个创建NodePort类型的Service。创建完成后，可以通过节点的IP:Port访问到后台Pod。

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

创建并查看，可以看到PORT这一列为8080:30120/TCP，说明Service的8080端口是映射到节点的30120端口。

```
$ 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
```

此时，通过节点IP:端口访问Service可以访问到Pod，如下所示。

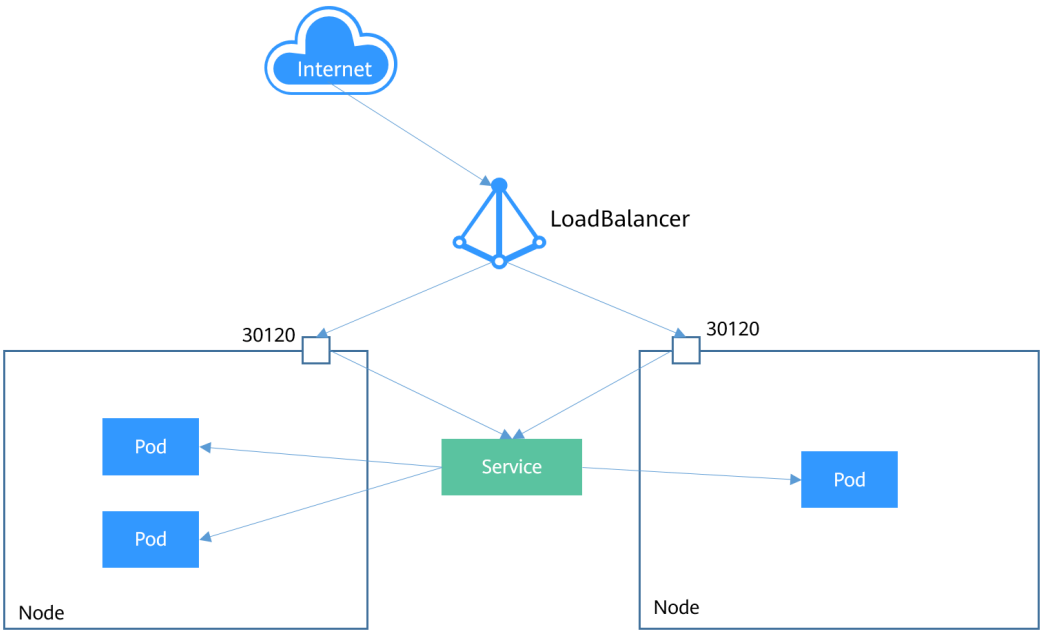
```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 192.168.0.212:30120
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
......
```


LoadBalancer 类型的 Service

LoadBalancer类型的Service其实是NodePort类型Service的扩展，通过一个特定的LoadBalancer访问Service，这个LoadBalancer将请求转发到节点的NodePort。

LoadBalancer本身不是属于Kubernetes的组件，这部分通常是由具体厂商（云服务提供商）提供，不同厂商的Kubernetes集群与LoadBalancer的对接实现各不相同，例如华为云CCE对接了ELB。这就导致了创建LoadBalancer类型的Service有不同的实现。

图 6-7 LoadBalancer Service



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

```
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     #  ELB实例的IP地址
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30120
  selector:
    app: nginx
  type: LoadBalancer    # 类型为LoadBalancer
```

上面metadata.annotations里的参数配置是CCE的LoadBalancer类型Service需要配置的参数，表示这个Service绑定哪个ELB实例。CCE还支持创建LoadBalancer类型Service时新建ELB实例，详细的内容请参见**负载均衡(LoadBalancer)**。

Headless Service

前面讲的Service解决了Pod的内外部访问问题，但还有下面这些问题没解决。

- 同时访问所有Pod
- 一个Service内部的Pod互相访问

Headless Service正是解决这个问题的，Headless Service不会创建ClusterIP，并且查询会返回所有Pod的DNS记录，这样就可查询到所有Pod的IP地址。**4.2 StatefulSet**中StatefulSet正是使用Headless Service解决Pod间互相访问的问题。

```
apiVersion: v1
kind: Service       # 对象类型为Service
metadata:
  name: nginx-headless
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Pod间通信的端口名称
      port: 80        # Pod间通信的端口号
  selector:
    app: nginx        # 选择标签为app:nginx的Pod
  clusterIP: None     # 必须设置为None，表示Headless Service
```

执行如下命令创建Headless Service。

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

创建完成后可以查询Service。

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

创建一个Pod来查询DNS，可以看到能返回所有Pod的记录，这就解决了访问所有Pod的问题了。

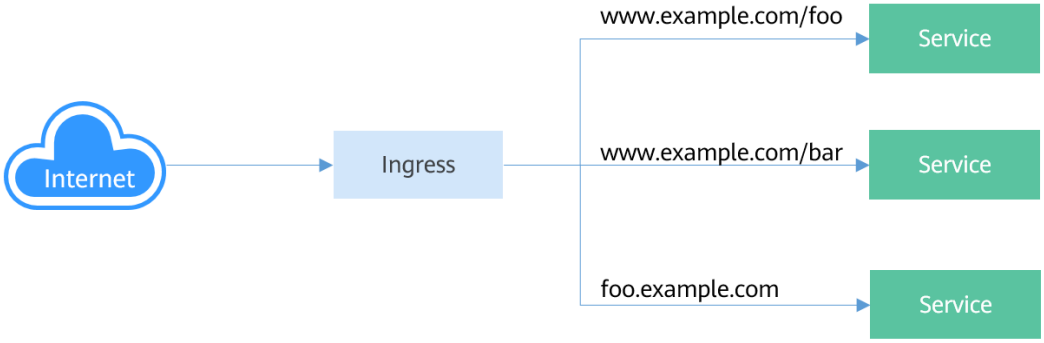
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

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

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


6.3 Ingress

为什么需要 Ingress

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

图 6-8 Ingress-Service

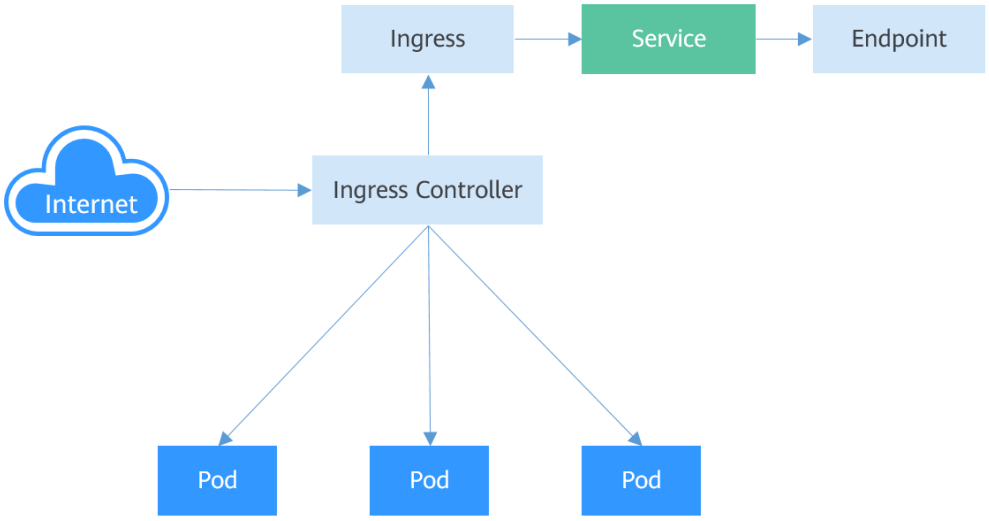


Ingress 工作机制

要想使用Ingress功能，必须在Kubernetes集群上安装Ingress Controller。Ingress Controller有很多种实现，最常见的就是Kubernetes官方维护的**NGINX Ingress Controller**；不同厂商通常有自己的实现，例如华为云CCE使用华为云弹性负载均衡服务ELB实现Ingress的七层负载均衡。

外部请求首先到达Ingress Controller，Ingress Controller根据Ingress的路由规则，查找到对应的Service，进而通过Endpoint查询到Pod的IP地址，然后将请求转发给Pod。

图 6-9 Ingress 工作机制



创建 Ingress

下面例子中，使用http协议，关联的后端Service为“nginx:8080”，使用ELB作为Ingress控制器（metadata.annotations字段都是指定使用哪个ELB实例），当访问“http://192.168.10.155:8080/test”时，流量转发“nginx:8080”对应的Service，从而将流量转发到对应Pod。

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

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

说明

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

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

路由到多个服务

Ingress可以同时路由到多个服务，配置如下所示。

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

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


6.4 就绪探针（Readiness Probe）

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

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

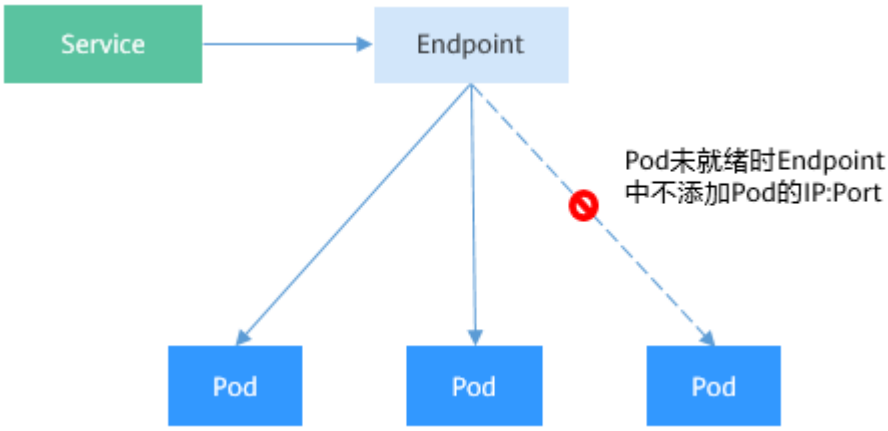
Readiness Probe同样是周期性的检测Pod，然后根据响应来判断Pod是否就绪，与**3.2 存活探针（Liveness Probe）**相同，就绪探针也支持如下三种类型。

- Exec：Probe执行容器中的命令并检查命令退出的状态码，如果状态码为0则说明已经就绪。
- HTTP GET：往容器的IP:Port发送HTTP GET请求，如果Probe收到2xx或3xx，说明已经就绪。
- TCP Socket：尝试与容器建立TCP连接，如果能建立连接说明已经就绪。

Readiness Probe 的工作原理

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

图 6-10 Readiness Probe 的实现原理



Exec

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

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

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - image: nginx:alpine
      name: container-0
      resources:
        limits:
          cpu: 100m
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      readinessProbe:      # Readiness Probe
        exec:              # 定义 ls /ready 命令
         command:
          - ls
          - /ready
    imagePullSecrets:
    - name: default-secret
```

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

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

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

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

```
# kubectl get po
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
```

创建Service。

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

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

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

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


```
# kubectl exec 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

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

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

TCP Socket

同样，TCP Socket类型的探针如下所示。

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



```
  containers:
  - image: nginx:alpine
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    readinessProbe:            # readinessProbe
      tcpSocket:               # TCP Socket定义
       port: 80
  imagePullSecrets:
  - name: default-secret
```

Readiness Probe 高级配置

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

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

这一行表示Readiness Probe的具体参数配置，其含义如下：

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

这些是创建时默认设置的，您也可以手动配置，如下所示。

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

6.5 NetworkPolicy

NetworkPolicy是Kubernetes设计用来限制Pod访问的对象，通过设置NetworkPolicy策略，可以允许Pod被哪些地址访问（即入规则）、或Pod访问哪些地址（即出规则）。这相当于从应用的层面构建了一道防火墙，进一步保证了网络安全。

NetworkPolicy支持的能力取决于集群的网络插件的能力，如CCE的集群只支持设置Pod的入规则。

默认情况下，如果命名空间中不存在任何策略，则所有进出该命名空间中的Pod的流量都被允许。

NetworkPolicy的规则可以选择如下3种：

- namespaceSelector：根据命名空间的标签选择，具有该标签的命名空间都可以访问。

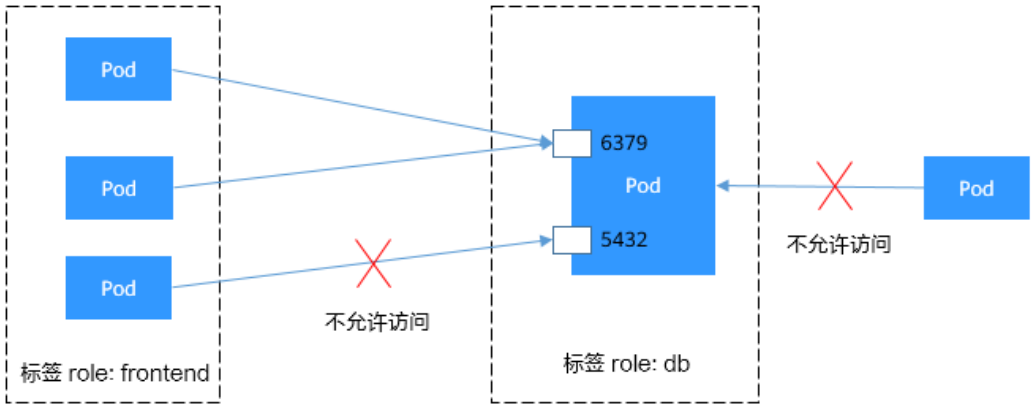
- podSelector：根据Pod的标签选择，具有该标签的Pod都可以访问。
- ipBlock：根据网络选择，网段内的IP地址都可以访问。（CCE当前不支持此种方式）

使用 podSelector 设置访问范围

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:                    # 表示入规则
  - from:
    - podSelector:            # 只允许具有role=frontend标签的Pod访问
        matchLabels:
          role: frontend
    ports:                    # 只能使用TCP协议访问6379端口
    - protocol: TCP
      port: 6379
```

示意图如下所示。

图 6-11 podSelector

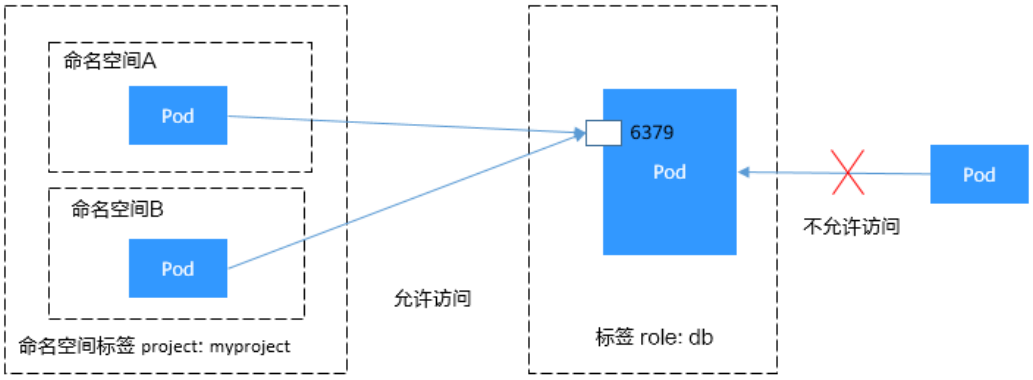


使用 namespaceSelector 设置访问范围

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:                    # 表示入规则
  - from:
    - namespaceSelector:      # 只允许具有project=myproject标签的命名空间中的Pod访问
        matchLabels:
          project: myproject
    ports:                    # 只能使用TCP协议访问6379端口
    - protocol: TCP
      port: 6379
```

示意图如下所示。

图 6-12 namespaceSelector



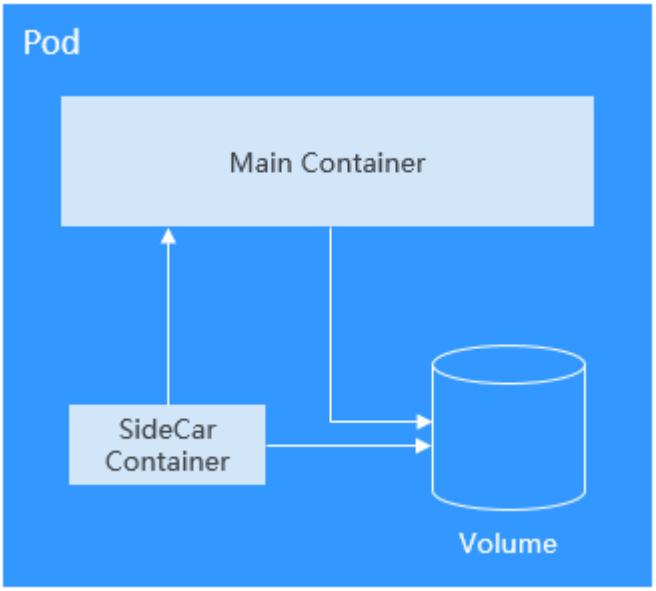
7 持久化存储

7.1 Volume

容器中的文件在磁盘上是临时存放的，当容器重建时，容器中的文件将会丢失，另外当在一个Pod中同时运行多个容器时，常常需要在这些容器之间共享文件，这也是容器不好解决的问题。 Kubernetes抽象出了Volume来解决这两个问题，也就是存储卷，Kubernetes的Volume是Pod的一部分，Volume不是单独的对象，不能独立创建，只能在Pod中定义。

Pod中的所有容器都可以访问Volume，但必须要挂载，且可以挂载到容器中任何目录。

实际中使用容器存储如下图所示，将容器的内容挂载到Volume中，通过Volume两个容器间实现了存储共享。



Volume的生命周期与挂载它的Pod相同，但是Volume里面的文件可能在Volume消失后仍然存在，这取决于Volume的类型。

Volume 的类型

Kubernetes的Volume有非常多的类型，在实际使用中使用最多的类型如下。

- emptyDir：一种简单的空目录，主要用于临时存储。
- hostPath：将主机某个目录挂载到容器中。
- ConfigMap、Secret：特殊类型，将Kubernetes特定的对象类型挂载到Pod，在**5.1 ConfigMap**和**5.2 Secret**章节介绍过如何将ConfigMap和Secret挂载到Volume中。
- persistentVolumeClaim：Kubernetes的持久化存储类型，详细介绍请参考**7.2 PV、PVC和StorageClass**中会详细介绍。

EmptyDir

EmptyDir是最简单的一种Volume类型，根据名字就能看出，这个Volume挂载后就是一个空目录，应用程序可以在里面读写文件，emptyDir Volume的生命周期与Pod相同，Pod删除后Volume的数据也同时删除掉。

emptyDir的一些用途：

- 缓存空间，例如基于磁盘的归并排序。
- 为耗时较长的计算任务提供检查点，以便任务能从崩溃前状态恢复执行。

emptyDir配置示例如下。

```
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实际是将Volume的内容写在Pod所在节点的磁盘上，另外emptyDir也可以设置存储介质为内存，如下所示，medium设置为Memory。

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

HostPath

HostPath是一种持久化存储，emptyDir里面的内容会随着Pod的删除而消失，但HostPath不会，如果对应的Pod删除，HostPath Volume里面的内容依然存在于节点的目录中，如果后续重新创建Pod并调度到同一个节点，挂载后依然可以读取到之前Pod写的内容。

HostPath存储的内容与节点相关，所以它不适合像数据库这类的应用，想象下如果数据库的Pod被调度到别的节点了，那读取的内容就完全不一样了。

记住永远不要使用HostPath存储跨Pod的数据，一定要把HostPath的使用范围限制在读取节点文件上，这是因为Pod被重建后不确定会调度哪个节点上，写文件可能会导致前后不一致。

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

7.2 PV、PVC 和 StorageClass

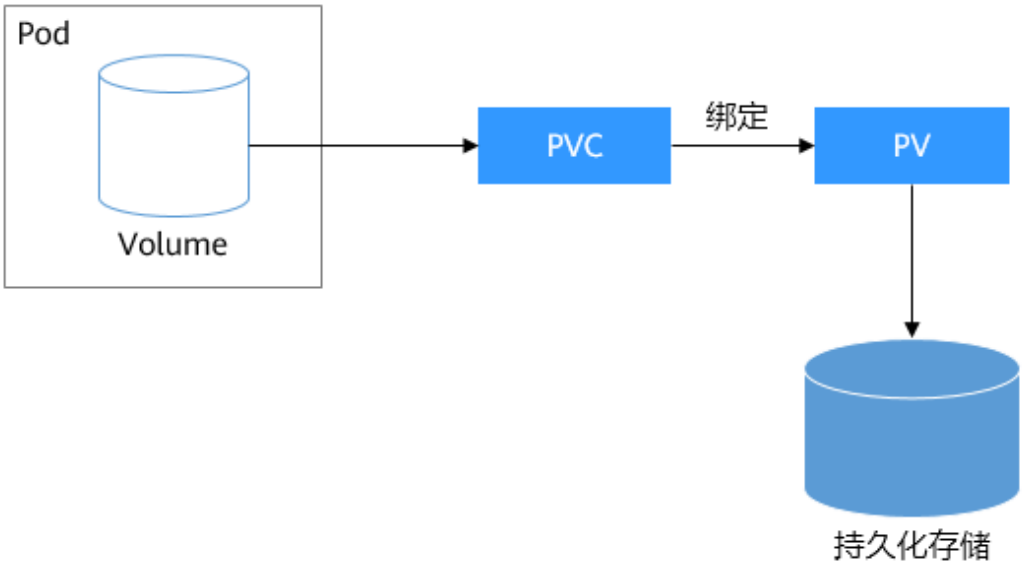
上一章节介绍的HostPath是一种持久化存储，但是HostPath的内容是存储在节点上，导致只适合读取。

如果要求Pod重新调度后仍然能使用之前读写过的数据，就只能使用网络存储了，网络存储种类非常多且有不同的使用方法，通常一个云服务提供商至少有块存储、文件存储、对象存储三种，如华为云的EVS、SFS和OBS。Kubernetes解决这个问题的方式是抽象了PV（PersistentVolume）和PVC（PersistentVolumeClaim）来解耦这个问题，从而让使用者不用关心具体的基础设施，当需要存储资源的时候，只要像CPU和内存一样，声明要多少即可。

- PV：PV描述的是持久化存储卷，主要定义的是一个持久化存储在宿主机上的目录，比如一个NFS的挂载目录。
- PVC：PVC描述的是Pod所希望使用的持久化存储的属性，比如，Volume存储的大小、可读写权限等等。

Kubernetes管理员设置好网络存储的类型，提供对应的PV描述符配置到Kubernetes，使用者需要存储的时候只需要创建PVC，然后在Pod中使用Volume关联PVC，即可让Pod使用到存储资源，它们之间的关系如下图所示。

图 7-1 PVC 绑定 PV



CSI

Kubernetes提供了CSI接口（Container Storage Interface，容器存储接口），基于CSI这套接口，可以开发定制出CSI插件，从而支持特定的存储，达到解耦的目的。例如在**3.4 Namespace：资源分组**中看到的kube-system命名空间下everest-csi-controller和everest-csi-driver就是华为云CCE开发存储控制器和驱动。有了这些驱动就可以使用华为云上的EVS、SFS、OBS存储。

```
$ 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

来看一下PV是如何描述持久化存储，例如在华为云SFS中创建了一个文件存储，这个文件存储ID为68e4a4fd-d759-444b-8265-20dc66c8c502，挂载地址为sfs-nas01.cn-north-4b.myhuaweicloud.com:/share-96314776。如果想在CCE中使用这个文件存储，则需要先创建一个PV来描述这个存储，如下所示。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  accessModes:
  - ReadWriteMany                    # 读写模式
  capacity:
    storage: 10Gi                    # 定义PV的大小
  csi:
    driver: nas.csi.everest.io       # 声明使用的驱动
    fsType: nfs                      # 存储类型
    volumeAttributes:
      everest.io/share-export-location: sfs-nas01.cn-north-4b.myhuaweicloud.com:/share-96314776   # 挂载地址
    volumeHandle: 68e4a4fd-d759-444b-8265-20dc66c8c502                                          # 存储ID
```

这里csi下面的内容就是华为云CCE中特定的字段，在其他地方无法使用。

下面创建这个PV并查看。

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

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

RECLAIM POLICY是指PV的回收策略，Retain表示PVC被释放后PV继续保留。STATUS值为Available，表示PV处于可用的状态。

PVC

PVC可以绑定一个PV，示例如下。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi              # 声明存储的大小
  volumeName: pv-example         # PV的名称
```

创建PVC并查看。

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

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

这里可以看到状态是Bound，VOLUME是pv-example，表示PVC已经绑定了PV。

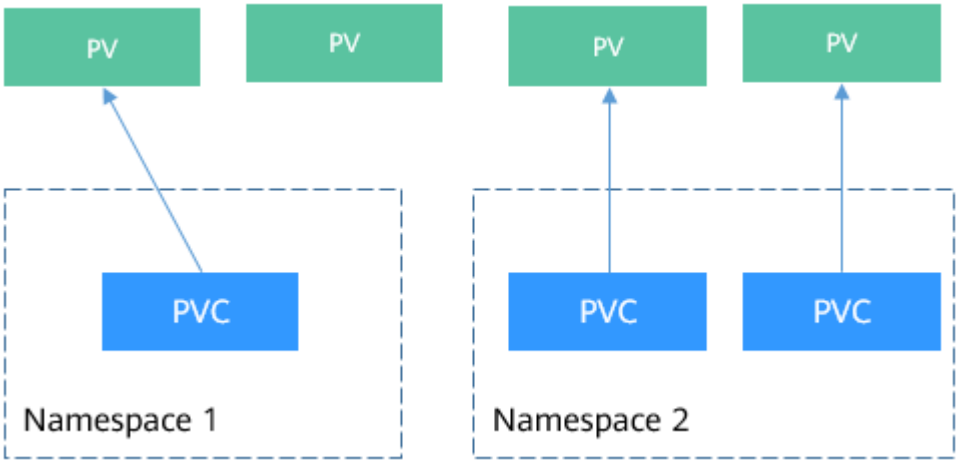
再来看下PV。

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

可以看到状态也变成了Bound，CLAIM是default/pvc-example，表示这个PV绑定了default命名空间下的pvc-example这个PVC。

这里一个比较有意思的地方是CLAIM是default/pvc-example，为什么要显示default呢，这是因为PV是集群级别的资源，并不属于某个命名空间，而PVC是命名空间级别的资源，PV可以与任何命名空间的PVC资源绑定。

图 7-2 PV 与 PVC



StorageClass

上节说的PV和PVC方法虽然能实现屏蔽底层存储，但是PV创建比较复杂（可以看到PV中csi字段的配置很麻烦），通常都是由集群管理员管理，这非常不方便。

Kubernetes解决这个问题的方法是提供动态配置PV的方法，可以自动创PV。管理员可以部署PV配置器（provisioner），然后定义对应的StorageClass，这样开发者在创建PVC的时候就可以选择需要创建存储的类型，PVC会把StorageClass传递给PV provisioner，由provisioner自动创建PV。如CCE就提供csi-disk、csi-nas、csi-obs等StorageClass，在声明PVC时加上StorageClassName，就可以自动创建PV，并自动创建底层的存储资源。

说明

下面是以CCE 1.15及以上版本集群使用方法举例，1.13以及之前版本集群上使用方法有差异，具体请参见**使用kubectl创建文件存储**。

执行如下命令即可查询CCE提供的默认StorageClass。您可以使用CCE提供的CSI插件自定义创建StorageClass，但从功能角度与CCE提供的默认StorageClass并无区别，这里不做过多描述。

```
# kubectl get sc
NAME              PROVISIONER                  AGE
csi-disk          everest-csi-provisioner      17d      # 云硬盘 StorageClass
csi-disk-topology   everest-csi-provisioner      17d      # 延迟绑定的云硬盘 StorageClass
csi-nas           everest-csi-provisioner      17d      # 文件存储 StorageClass
csi-obs           everest-csi-provisioner      17d      # 对象存储 StorageClass
csi-sfsturbo      everest-csi-provisioner      17d      # 极速文件存储 StorageClass
```

使用StorageClassName创建PVC。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name:  pvc-sfs-auto-example
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas       # StorageClass
```

创建PVC并查看PVC和PV。

```
$ 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
```

这可以看到使用StorageClass后，不仅创建了PVC，而且创建了PV，并且将二者绑定了。

定义了StorageClass后，就可以减少创建并维护PV的工作，PV变成了自动创建，作为使用者，只需要在声明PVC时指定StorageClassName即可，这就大大减少工作量。

再次说明，StorageClassName的类型在不同厂商的产品上各不相同，这里只是使用了华为云文件存储作为示例，其余存储类型请参见**存储概述**。

在 Pod 中使用 PVC

有了PVC后，在Pod中使用持久化存储就非常方便了，在Pod Template中的Volume直接关联PVC的名称，然后挂载到容器之中即可，如下所示。甚至在StatefulSet中还可以直接声明PVC，详情请参见**4.2 StatefulSet**。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        volumeMounts:
        - mountPath: /tmp                          # 挂载路径
          name: pvc-sfs-example
      restartPolicy: Always
      volumes:
      - name: pvc-sfs-example
        persistentVolumeClaim:
          claimName:  pvc-example                  # PVC的名称
```

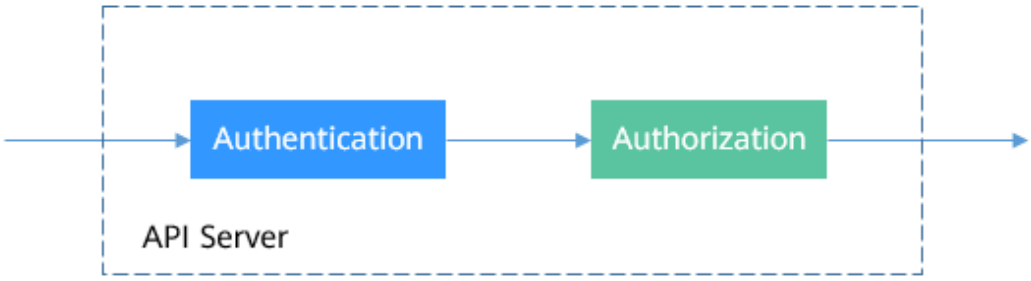
8 认证与授权

8.1 ServiceAccount

Kubernetes中所有的访问，无论外部内部，都会通过API Server处理，访问Kubernetes资源前需要经过认证与授权。

- Authentication：用于识别用户身份的认证，Kubernetes分外部服务帐号和内部服务帐号，采取不同的认证机制，具体请参见**认证与ServiceAccount**。
- Authorization：用于控制用户对资源访问的授权，对访问的授权目前主要使用RBAC机制，将在**8.2 RBAC**介绍。

图 8-1 API Server 的认证授权



认证与 ServiceAccount

Kubernetes的用户分为服务帐户（ServiceAccount）和普通帐户两种类型。

- 服务帐户与Namespace绑定，关联一套凭证，存储在Secret中，Pod创建时挂载Secret，从而允许与API Server之间调用。
- Kubernetes中没有代表普通帐户的对象，这类帐户默认由外部服务独立管理，比如在华为云上CCE的用户是由IAM管理的。

普通帐号并不是这里要讨论的内容，这里主要关注ServiceAccount。

ServiceAccount同样是Kubernetes中的资源，与Pod、ConfigMap类似，且作用于独立的命名空间，也就是ServiceAccount是属于命名空间级别的，创建命名空间时会自动创建一个名为default的ServiceAccount。

使用下面命令可以查看ServiceAccount。

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

同时Kubernetes还会为ServiceAccount自动创建一个Secret，使用下面命令可以查看到。

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

在Pod的定义文件中，可以用指定帐户名称的方式将一个ServiceAccount赋值给一个Pod，如果不指定就会使用默认的ServiceAccount。当API Server接收到一个带有认证Token的请求时，API Server会用这个Token来验证发送请求的客户端所关联的ServiceAccount是否允许执行请求的操作。

创建 ServiceAccount

使用如下命令就可以创建ServiceAccount：

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

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

可以看到已经创建了与ServiceAccount相关联的Token。

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

查看Secret的内容，可以发现ca.crt、namespace和token三个数据。

```
$ kubectl describe secret sa-example-token-c7bqx
Name:         sa-example-token-c7bqx
...
Data
====
ca.crt:     1082 bytes
namespace:  7 bytes
token:      <Token的内容>
```

在 Pod 中使用 ServiceAccount

Pod中使用ServiceAccount非常方便，只需要指定ServiceAccount的名称即可。

```
apiVersion: v1
kind: Pod
metadata:
```



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

创建并查看这个Pod，可以看到Pod挂载了sa-example-token-c7bqx，也就是sa-example这个ServiceAccount对应的Token，即Pod使用这个Token来做认证。

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

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

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

进入Pod内部，还可以看到对应的文件，如下所示。

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

如上，在容器应用中，就可以使用ca.crt和Token来访问API Server。

下面来验证一下认证是否能生效。在Kubernetes集群中，默认为API Server创建了一个名为kubernetes的Service，通过这个Service可以访问API Server。

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

进入Pod，使用curl命令直接访问会得到如下返回信息，表示并没有权限。

```
$ kubectl exec -it sa-example -- /bin/sh
/ # curl https://kubernetes
curl: (60) SSL certificate problem: unable to get local issuer certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

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

使用ca.crt和Token做认证，先将ca.crt放到CURL_CA_BUNDLE这个环境变量中，curl命令使用CURL_CA_BUNDLE指定证书；再将Token的内容放到TOKEN中，然后带上TOKEN访问API Server。

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

```
 "apiVersion": "v1",
 "metadata": {

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

 },
 "code": 403
}
```

可以看到，已经能够通过认证了，但是API Server返回的是cannot get path \"/\""，表示没有权限访问，这说明还需要得到授权后才能访问，授权机制将在8.2 RBAC中介绍。

8.2 RBAC

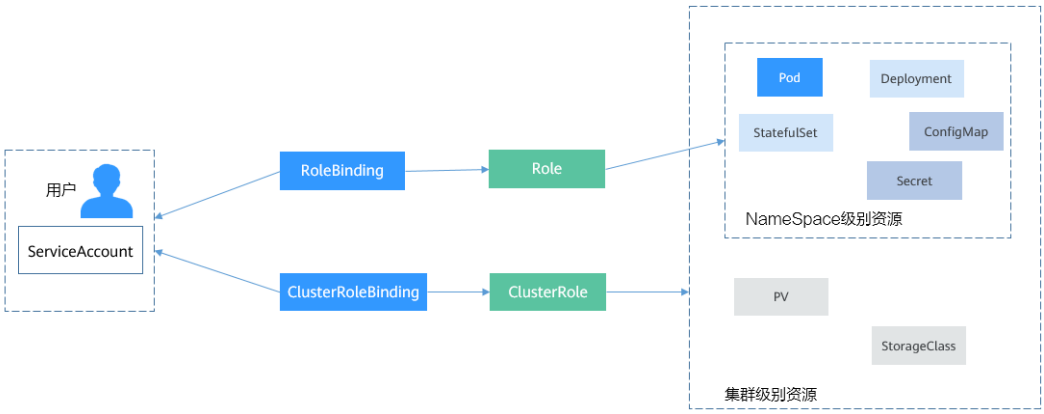
RBAC 资源

Kubernetes中完成授权工作的就是RBAC机制，RBAC授权规则是通过四种资源来进行配置。

- Role：角色，其实是定义一组对Kubernetes资源（命名空间级别）的访问规则。
- RoleBinding：角色绑定，定义了用户和角色的关系。
- ClusterRole：集群角色，其实是定义一组对Kubernetes资源（集群级别，包含全部命名空间）的访问规则。
- ClusterRoleBinding：集群角色绑定，定义了用户和集群角色的关系。

Role和ClusterRole指定了可以对哪些资源做哪些动作，RoleBinding和ClusterRoleBinding将角色绑定到特定的用户、用户组或ServiceAccount上。如下图所示。

图 8-2 角色绑定



创建 Role

Role的定义非常简单，指定namespace，然后就是rules规则。如下面示例中的规则就是允许对default命名空间下的Pod进行GET、LIST操作。

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
```

```
metadata:
  namespace: default                    # 命名空间
  name: role-example
rules:
- apiGroups: [""]
  resources: ["pods"]                   # 可以访问pod
  verbs: ["get", "list"]                # 可以执行GET、LIST操作
```

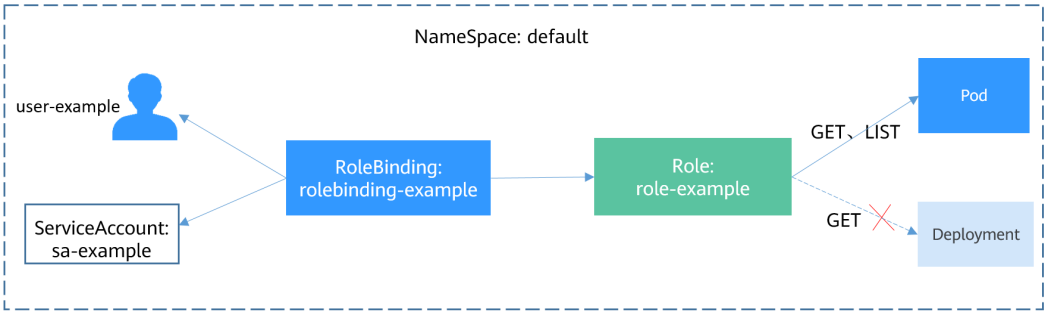
创建 RoleBinding

有了Role之后，就可以将Role与具体的用户绑定起来，实现这个的就是RoleBinding了。如下所示。

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rolebinding-example
  namespace: default
subjects:                               # 指定用户
- kind: User                            # 普通用户
  name: user-example
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount                  # ServiceAccount
  name: sa-example
  namespace: default
roleRef:                                # 指定角色
  kind: Role
  name: role-example
  apiGroup: rbac.authorization.k8s.io
```

这里的subjects就是将Role与用户绑定起来，用户可以是外部普通用户，也可以是ServiceAccount，这两种用户类型在8.1 ServiceAccount有过介绍。绑定后的关系如下图所示。

图 8-3 RoleBinding 绑定 Role 和用户



下面来验证一下授权是否生效。

在前面一个章节使用ServiceAccount中，创建一个Pod，使用了sa-example这个ServiceAccount，而刚刚又给sa-example绑定了role-example这个角色，现在进入到Pod，使用curl命令通过API Server访问资源来验证权限是否生效。

使用sa-example对应的ca.crt和Token认证，查询default命名空间下所有Pod资源，对应创建Role中的LIST。

```
$ kubectl exec -it sa-example -- /bin/sh
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods
{
  "kind": "PodList",
```

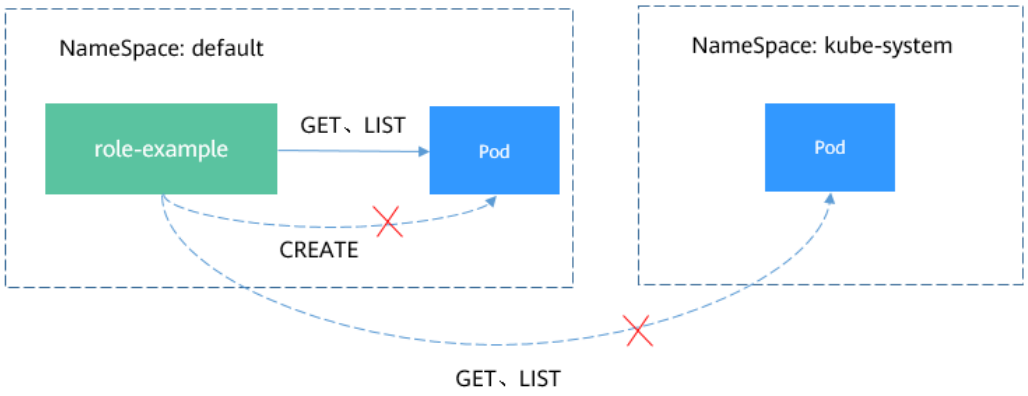
```
 "apiVersion": "v1",
 "metadata": {
   "selfLink": "/api/v1/namespaces/default/pods",
   "resourceVersion": "10377013"
 },
 "items": [
   {
     "metadata": {
       "name": "sa-example",
       "namespace": "default",
       "selfLink": "/api/v1/namespaces/default/pods/sa-example",
       "uid": "c969fb72-ad72-4111-a9f1-0a8b148e4a3f",
       "resourceVersion": "10362903",
       "creationTimestamp": "2020-07-15T06:19:26Z"
     },
     "spec": {
...
```

返回结果正常，说明sa-example是有LIST Pod的权限的。再查询一下Deployment，返回如下，说明没有访问Deployment的权限。

```
# 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和RoleBinding作用的范围是命名空间，能够做到一定程度的权限隔离，如下图所示，上面定义role-example就不能访问kube-system命名空间下的资源。

图 8-4 Role 和 RoleBinding 作用的范围是命名空间



在上面Pod中继续访问，返回如下，说明确实没有权限。

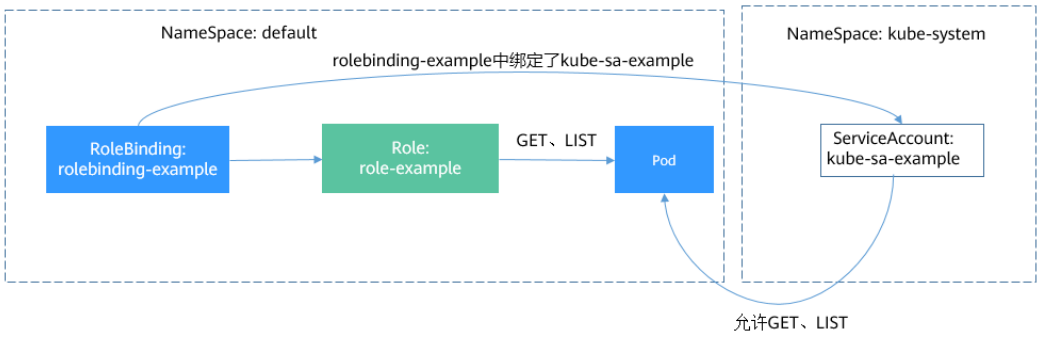
```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/kube-system/pods
...
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list resource
\"pods\" in API group \"\" in the namespace \"kube-system\"",
  "reason":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

在RoleBinding中，还可以绑定其他命名空间的ServiceAccount，只要在subjects字段下添加其他命名空间的ServiceAccount即可。

```
subjects:                       # 指定用户
- kind: ServiceAccount              # ServiceAccount
  name: kube-sa-example
  namespace: kube-system
```


加入之后，kube-system下kube-sa-example这个ServiceAccount就可以GET、LIST命名空间default下的Pod了，如下图所示。

图 8-5 跨命名空间访问



ClusterRole 和 ClusterRoleBinding

相比Role和RoleBinding，ClusterRole和ClusterRoleBinding有如下几点不同：

- ClusterRole和ClusterRoleBinding不用定义namespace字段
- ClusterRole可以定义集群级别的资源

可以看出ClusterRole和ClusterRoleBinding控制的是集群级别的权限。

在Kubernetes中，默认定义了非常多的ClusterRole和ClusterRoleBinding，如下所示。

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

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

其中，最重要最常用的是如下四个ClusterRole。

- view：拥有查看命名空间资源的权限
- edit：拥有修改命名空间资源的权限
- admin：拥有命名空间全部权限
- cluster-admin：拥有集群的全部权限

使用**kubectl describe clusterrole**命令能够查看到各个规则的具体权限。

通常情况下，使用这四个ClusterRole与用户做绑定，就可以很好的做到权限隔离。这里的关键一点是理解到Role（规则、权限）与用户是分开的，只要通过Rolebinding来对这两者进行组合就能做到灵活的权限控制。

9 弹性伸缩

在**4 Pod的编排与调度**章节介绍了Deployment这类控制器来控制Pod的副本数量，通过调整replicas的大小就可以达到给应用手动扩缩容的目的。但是在某些实际场景下，手动调整一是繁琐，二是速度没有那么快，尤其是在应对流量洪峰需要快速弹性时无法做出快速反应。

Kubernetes支持Pod和集群节点的自动弹性伸缩，通过设置弹性伸缩规则，当外部条件（如CPU使用率）达到一定条件时，根据规则自动伸缩Pod和集群节点。

Prometheus 与 Metrics Server

想要做到自动弹性伸缩，先决条件就是能感知到各种运行数据，例如集群节点、Pod、容器的CPU、内存使用率等等。而这些数据的监控能力Kubernetes也没有自己实现，而是通过其他项目来扩展Kubernetes的能力。

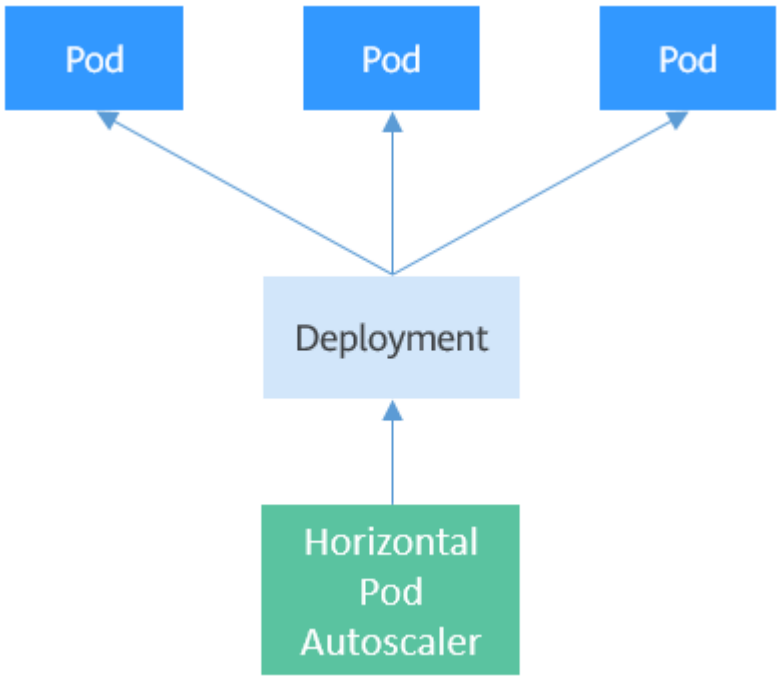
- **Prometheus**是一套开源的系统监控报警框架，能够采集丰富的Metrics（度量数据），目前已经基本是Kubernetes的标准监控方案。
- **Metrics Server**是Kubernetes集群范围资源使用数据的聚合器。Metrics Server从kubelet公开的Summary API中采集度量数据，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据，且对外提供一套标准的API。

使用HPA（Horizontal Pod Autoscaler）配合Metrics Server可以实现基于CPU和内存的自动弹性伸缩，再配合Prometheus还可以实现**自定义监控指标**的自动弹性伸缩。

HPA 工作机制

HPA（Horizontal Pod Autoscaler）是用来控制Pod水平伸缩的控制器，HPA周期性检查Pod的度量数据，计算满足HPA资源所配置的目标数值所需的副本数量，进而调整目标资源（如Deployment）的replicas字段。

图 9-1 HPA 工作机制



HPA可以配置单个和多个度量指标，配置单个度量指标时，只需要对Pod的当前度量数据求和，除以期望目标值，然后向上取整，就能得到期望的副本数。例如有一个Deployment控制有3个Pod，每个Pod的CPU使用率是70%、50%、90%，而HPA中配置的期望值是50%，计算期望副本数=（70 + 50 + 90）/50 = 4.2，向上取整得到5，即期望副本数就是5。

如果是配置多个度量指标，则会分别计算单个度量指标的期望副本数量，然后取其中最大值，就是最终的期望副本数量。

使用 HPA

下面通过示例演示HPA的使用。首先使用Nginx镜像创建一个4副本的Deployment。

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

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

创建一个HPA，期望CPU的利用率为70%，副本数的范围是1-10。

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: scale
  namespace: default
spec:
  maxReplicas: 10                # 目标资源的最大副本数量
  minReplicas: 1                 # 目标资源的最小副本数量
  metrics:                       # 度量指标，期望CPU的利用率为70%
  - resource:
```



```
    name: cpu
    targetAverageUtilization: 70
  type: Resource
scaleTargetRef:                 # 目标资源
  apiVersion: apps/v1
  kind: Deployment
  name: nginx-deployment
```

创建后HPA查看。

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

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

可以看到，TARGETS的期望值是70%，而实际是0%，这就意味着HPA会做出缩容动作，期望副本数量=(0+0+0+0)/70=0，但是由于最小副本数为1，所以Pod数量会调整为1。等待一段时间，可以看到Pod数量变为1。

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

查看HPA详情，可以在Events里面看到这样一条记录。这表示HPA在21秒前成功的执行了缩容动作，新的Pod数量为1，原因是所有度量数量都比目标值低。

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

如果再查看Deployment的详情，可以在Events里面看到这样一条记录。这表示Deployment的副本数量被设置为1了，跟HPA中看到的一致。

```
$ 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

HPA是针对Pod级别的，但是如果集群的资源不够了，那就只能对节点进行扩容了。集群节点的弹性伸缩本来是一件非常麻烦的事情，但是好在现在的集群大多都是构建在云上，云上可以直接调用接口添加删除节点，这就使得集群节点弹性伸缩变得非常方便。

Cluster Autoscaler是Kubernetes提供的集群节点弹性伸缩组件，根据Pod调度状态及资源使用情况对集群的节点进行自动扩容缩容。由于要调用云上接口实现弹性伸缩，这就使得在不同环境上的实现与使用各不相同，这里不详细介绍。

华为云CCE的集群节点弹性伸缩请参见**创建节点伸缩策略**。

