Kubernetes 开源知识

文档版本 01

发布日期 2024-09-24

华为云计算技术有限公司

商标声明


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

注意

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

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

目 录

1 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，您可以先行了解如下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

- 3.1 容器
- 3.2 Kubernetes
- 3.3 使用Kubectl命令操作集群

Pod、Label 和 Namespace

- 4.1 Pod：Kubernetes中的最小调度对象
- 4.2 存活探针（Liveness Probe）
- 4.3 Label：组织Pod的利器
- 4.4 Namespace：资源分组

Pod 的编排与调度

- 5.1 无状态负载（Deployment）
- 5.2 有状态负载（StatefulSet）
- 5.3 普通任务（Job）和定时任务（CronJob）
- 5.4 守护进程集（DaemonSet）
- 5.5 亲和与反亲和调度

配置管理

Kubernetes 网络

持久化存储

认证与授权

弹性伸缩

2 基本概念

云容器引擎（Cloud Container Engine，简称CCE）提供高度可扩展的、高性能的企业级Kubernetes集群。借助云容器引擎，您可以在云上轻松部署、管理和扩展容器化应用程序。

云容器引擎提供Kubernetes原生API，支持使用kubectl，且提供图形化控制台，让您能够拥有完整的端到端使用体验，使用云容器引擎前，建议您先了解相关的基本概念。

集群（Cluster）

集群指容器运行所需要的云资源组合，关联了若干云服务器节点、负载均衡等云资源。您可以理解为集群是“同一个子网中一个或多个弹性云服务器（又称：节点）”通过相关技术组合而成的计算机群体，为容器运行提供了计算资源池。

云容器引擎支持的集群类型如下：

集群类型	描述
CCE Standard集群	CCE Standard集群是云容器引擎服务的标准版本集群，提供商用级容器集群服务，并完全兼容开源 Kubernetes 集群标准功能。 CCE Standard集群为您提供简单、低成本、高可用的解决方案，无需管理和运维控制节点，并且可根据业务场景选择使用容器隧道网络模型或VPC网络模型，适合对性能和规模没有特殊要求的通用场景。
CCE Turbo集群	CCE Turbo集群是基于云原生基础设施构建的云原生2.0容器引擎服务，具备软硬协同、网络无损、安全可靠、调度智能的优势，为用户提供一站式、高性价比的全新容器服务体验。 CCE Turbo集群提供了面向大规模高性能的场景云原生2.0网络，容器直接从VPC网段内分配IP地址，容器和节点可以分属不同子网，支持VPC内的外部网络与容器IP直通，享有高性能。

集群类型	描述
CCE Autopilot集群	CCE Autopilot是云容器引擎服务推出的Serverless版集群，提供免运维的容器服务，并提供经过优化的Kubernetes兼容能力。 CCE Autopilot集群提供了无用户节点的部署方式，简化了应用部署流程。您无需购买节点，也无需对节点的部署、管理和安全性进行维护，只需要关注应用业务逻辑的实现，可以大幅降低您的运维成本，提高应用程序的可靠性和可扩展性。

节点（Node）

每一个节点对应一台服务器（可以是虚拟机实例或者物理服务器），容器应用运行在节点上。节点上运行着Agent代理程序（kubelet），用于管理节点上运行的容器实例。集群中的节点数量可以伸缩。

节点池（NodePool）

节点池是集群中具有相同配置的一组节点，一个节点池包含一个节点或多个节点。

虚拟私有云（VPC）

虚拟私有云是通过逻辑方式进行网络隔离，提供安全、隔离的网络环境。您可以在VPC中定义与传统网络无差别的虚拟网络，同时提供弹性IP、安全组等高级网络服务。

安全组

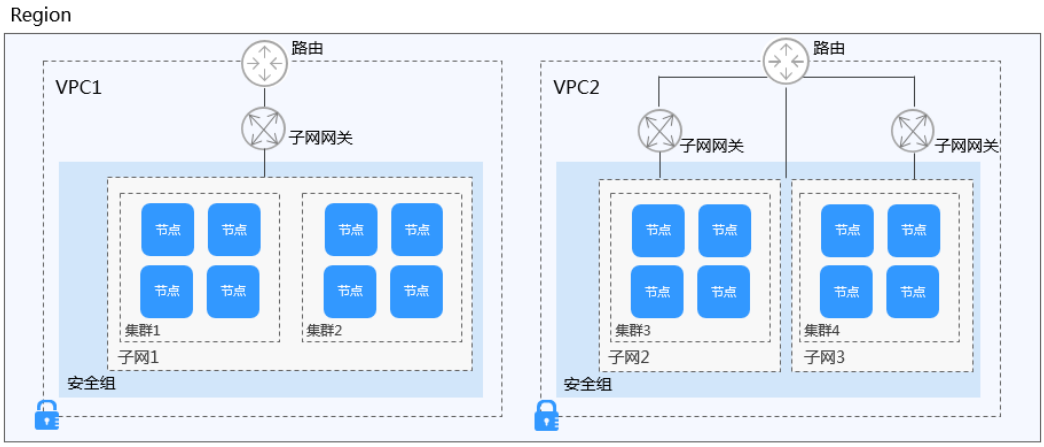
安全组是一个逻辑上的分组，为同一个VPC内具有相同安全保护需求并相互信任的弹性云服务器提供访问策略。安全组创建后，用户可以在安全组中定义各种访问规则，当弹性云服务器加入该安全组后，即受到这些访问规则的保护。

集群、虚拟私有云、安全组和节点的关系

如图2-1，同一个Region下可以有多个虚拟私有云（VPC）。虚拟私有云由一个个子网组成，子网与子网之间的网络交互通过子网网关完成，而集群就是建立在某个子网中。因此，存在以下三种场景：

- 不同集群可以创建在不同的虚拟私有云中。
- 不同集群可以创建在同一个子网中。
- 不同集群可以创建在不同的子网中。

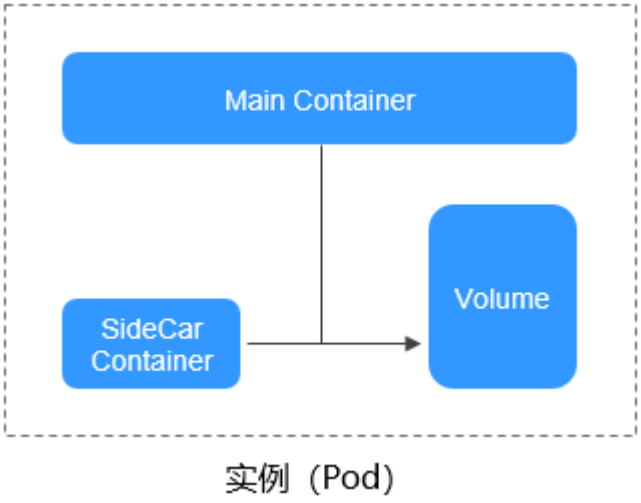
图 2-1 集群、VPC、安全组和节点的关系



实例（Pod）

实例（Pod）是 Kubernetes 部署应用或服务的最小的基本单位。一个Pod 封装多个应用容器（也可以只有一个容器）、存储资源、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。

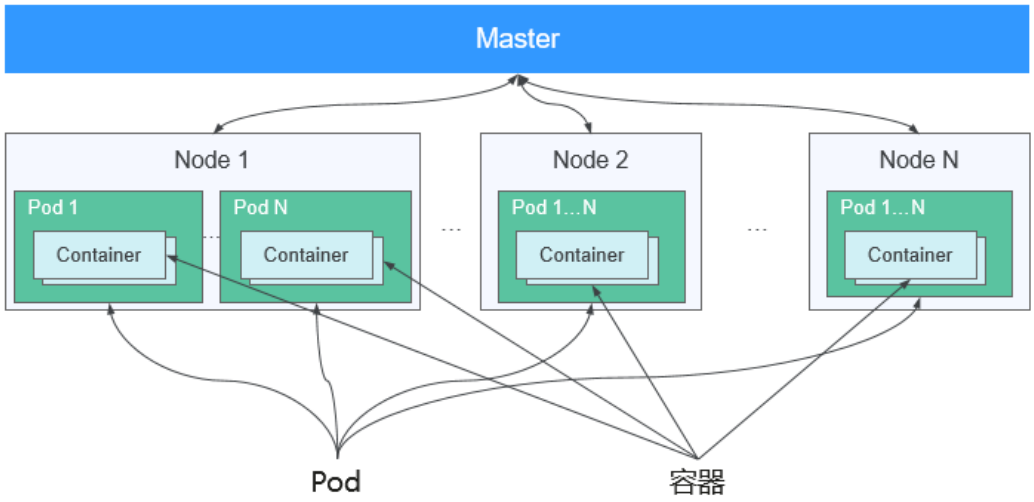
图 2-2 实例（Pod）



容器（Container）

一个通过 Docker 镜像创建的运行实例，一个节点可运行多个容器。容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。

图 2-3 实例 Pod、容器 Container、节点 Node 的关系

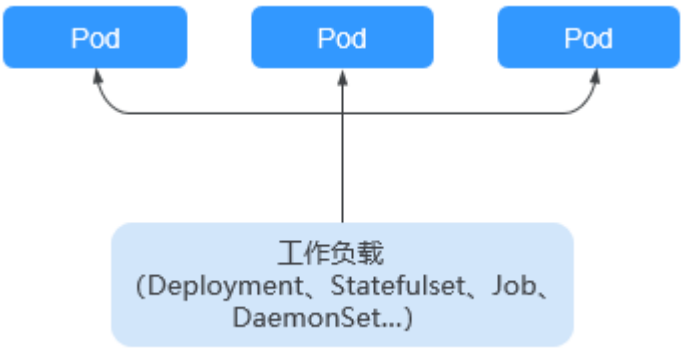


工作负载

工作负载是在Kubernetes上运行的应用程序。无论您的工作负载是单个组件还是协同工作的多个组件，您都可以在Kubernetes上的一组Pod中运行它。在Kubernetes中，工作负载是对一组Pod的抽象模型，用于描述业务的运行载体，包括Deployment、StatefulSet、DaemonSet、Job、CronJob等多种类型。

- **无状态工作负载：**即Kubernetes中的“Deployment”，无状态工作负载支持弹性伸缩与滚动升级，适用于实例完全独立、功能相同的场景，如：nginx、wordpress等。
- **有状态工作负载：**即Kubernetes中的“StatefulSet”，有状态工作负载支持实例有序部署和删除，支持持久化存储，适用于实例间存在互访的场景，如ETCD、mysql-HA等。
- **创建守护进程集：**即Kubernetes中的“DaemonSet”，守护进程集确保全部（或者某些）节点都运行一个Pod实例，支持实例动态添加到新节点，适用于实例在每个节点上都需要运行的场景，如ceph、fluentd、Prometheus Node Exporter等。
- **普通任务：**即Kubernetes中的“Job”，普通任务是一次性运行的短任务，部署完成后即可执行。使用场景为在创建工作负载前，执行普通任务，将镜像上传至镜像仓库。
- **定时任务：**即Kubernetes中的“CronJob”，定时任务是按照指定时间周期运行的短任务。使用场景为在某个固定时间点，为所有运行中的节点做时间同步。

图 2-4 工作负载与 Pod 的关系

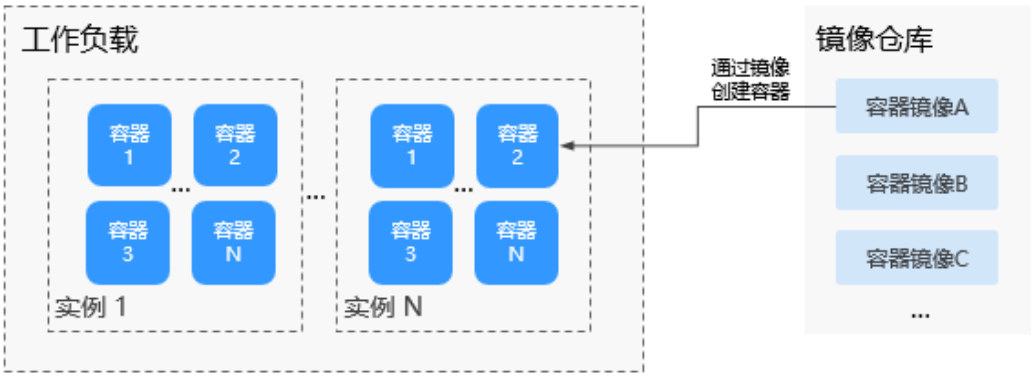


镜像（Image）

Docker镜像是一个模板，是容器应用打包的标准格式，用于创建Docker容器。或者说，Docker镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。在部署容器化应用时可以指定镜像，镜像可以来自于 Docker Hub、容器镜像服务或者用户的私有Registry。例如一个Docker镜像可以包含一个完整的Ubuntu操作系统环境，里面仅安装了用户需要的应用程序及其依赖文件。

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

图 2-5 镜像、容器、工作负载的关系



命名空间（Namespace）

命名空间是对一组资源和对象的抽象整合。在同一个集群内可创建不同的命名空间，不同命名空间中的数据彼此隔离。使得它们既可以共享同一个集群的服务，也能够互不干扰。例如：

- 可以将开发环境、测试环境的业务分别放在不同的命名空间。
- 常见的pods，services，replication controllers和deployments等都是属于某一个namespace的（默认是default），而node，persistentVolumes等则不属于任何namespace。

服务（Service）

Service是将运行在一组 Pods 上的应用程序公开为网络服务的抽象方法。

使用Kubernetes，您无需修改应用程序即可使用不熟悉的服务发现机制。 Kubernetes为Pods提供自己的IP地址和一组Pod的单个DNS名称，并且可以在它们之间进行负载平衡。

Kubernetes允许指定一个需要的类型的Service，类型 的取值以及行为如下：

- ClusterIP：集群内访问。通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 ServiceType。
- NodePort：节点访问。通过每个Node上的 IP 和静态端口（ NodePort ）暴露服务。NodePort服务会路由到ClusterIP服务，这个ClusterIP服务会自动创建。通过请求 <NodeIP>:<NodePort>，可以从集群的外部访问一个 NodePort 服务。
- LoadBalancer：负载均衡。使用云提供商的负载均衡器，可以向外部暴露服务。外部的负载均衡器可以路由到NodePort服务和ClusterIP服务。
- DNAT：DNAT网关。可以为集群节点提供网络地址转换服务，使多个节点可以共享使用弹性IP。与弹性IP方式相比增强了可靠性，弹性IP无需与单个节点绑定，任何节点状态的异常不影响其访问。

七层负载均衡（Ingress）

Ingress是为进入集群的请求提供路由规则的集合，可以给service提供集群外部访问的URL、负载均衡、SSL终止、HTTP路由等。

网络策略（NetworkPolicy）

NetworkPolicy提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。

配置项（Configmap）

ConfigMap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。ConfigMap跟secret很类似，但它可以更方便地处理不包含敏感信息的字符串。

密钥（Secret）

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

标签（Label）

标签其实就一对 key/value，被关联到对象上，比如Pod。标签的使用能够标示对象的特殊特点，并且对用户而言是有意义的，但是标签对内核系统是没有直接意义的。

选择器（LabelSelector）

Label selector是Kubernetes核心的分组机制，通过label selector客户端/用户能够识别一组有共同特征或属性的资源对象。

注解（Annotation）

Annotation与Label类似，也使用key/value键值对的形式进行定义。

Label具有严格的命名规则，它定义的是Kubernetes对象的元数据（Metadata），并且用于Label Selector。

Annotation则是用户任意定义的“附加”信息，以便于外部工具进行查找。

存储卷（PersistentVolume）

PersistentVolume（PV）是集群之中的一块网络存储。跟 Node 一样，也是集群的资源。

存储声明（PersistentVolumeClaim）

PV 是存储资源，而 PersistentVolumeClaim (PVC) 是对 PV 的请求。PVC 跟 Pod 类似：Pod 消费 Node 资源，而 PVC 消费 PV 资源；Pod 能够请求 CPU 和内存资源，而 PVC 请求特定大小和访问模式的数据卷。

弹性伸缩（HPA）

Horizontal Pod Autoscaling，简称HPA，是Kubernetes中实现POD水平自动伸缩的功能。Kubernetes集群可以通过Replication Controller的scale机制完成服务的扩容或缩容，实现具有伸缩性的服务。

亲和性与反亲和性

在应用没有容器化之前，原先一个虚机上会装多个组件，进程间会有通信。但在做容器化拆分的时候，往往直接按进程拆分容器，比如业务进程一个容器，监控日志处理或者本地数据放在另一个容器，并且有独立的生命周期。这时如果分布在网络中两个较远的点，请求经过多次转发，性能会很差。

- 亲和性：可以实现就近部署，增强网络能力实现通信上的就近路由，减少网络的损耗。如：应用A与应用B两个应用频繁交互，所以有必要利用亲和性让两个应用的尽可能的靠近，甚至在一个节点上，以减少因网络通信而带来的性能损耗。
- 反亲和性：主要是出于高可靠性考虑，尽量分散实例，某个节点故障的时候，对应用的影响只是 N 分之一或者只是一个实例。如：当应用采用多副本部署时，有必要采用反亲和性让各个应用实例打散分布在各个节点上，以提高HA。

节点亲和性（NodeAffinity）

通过选择标签的方式，可以限制pod被调度到特定的节点上。

工作负载亲和性（PodAffinity）

指定工作负载部署在相同节点。用户可根据业务需求进行工作负载的就近部署，容器间通信就近路由，减少网络消耗。

工作负载反亲和性（PodAntiAffinity）

指定工作负载部署在不同节点。同个工作负载的多个实例反亲和部署，减少宕机影响；互相干扰的应用反亲和部署，避免干扰。

资源配额（Resource Quota）

资源配额（Resource Quotas）是用来限制用户资源用量的一种机制。

资源限制（Limit Range）

默认情况下，K8s中所有容器都没有任何CPU和内存限制。LimitRange(简称limits)用来给Namespace增加一个资源限制，包括最小、最大和默认资源。在pod创建时，强制执行使用limits的参数分配资源。

环境变量

环境变量是指容器运行环境中设定的一个变量，您可以在创建容器模板时设定不超过30个的环境变量。环境变量可以在工作负载部署后修改，为工作负载提供了极大的灵活性。

在CCE中设置环境变量与Dockerfile中的“ENV”效果相同。

模板（Chart）

Kubernetes集群可以通过**Helm**实现软件包管理，这里的Kubernetes软件包被称为模板（Chart）。Helm对于Kubernetes的关系类似于在Ubuntu系统中使用的apt命令，或是在CentOS系统中使用的yum命令，它能够快速查找、下载和安装模板（Chart）。

模板（Chart）是一种Helm的打包格式，它只是描述了一组相关的集群资源定义，而不是真正的容器镜像包。模板中仅仅包含了用于部署Kubernetes应用的一系列YAML文件，您可以在Helm模板中自定义应用程序的一些参数设置。在模板的实际安装过程中，Helm会根据模板中的YAML文件定义在集群中部署资源，相关的容器镜像并不会包含在模板包中，而是依旧从YAML中定义好的镜像仓库中进行拉取。

对于应用开发者而言，需要将容器镜像包发布到镜像仓库，并通过Helm的模板将安装应用时的依赖关系统一打包，预置一些关键参数，来降低应用的部署难度。

对于应用使用者而言，可以使用Helm查找模板（Chart）包并支持调整自定义参数。Helm会根据模板包中的YAML文件直接在集群中安装应用程序及其依赖，应用使用者不用编写复杂的应用部署文件，即可以实现简单的应用查找、安装、升级、回滚、卸载。

3 容器与 Kubernetes

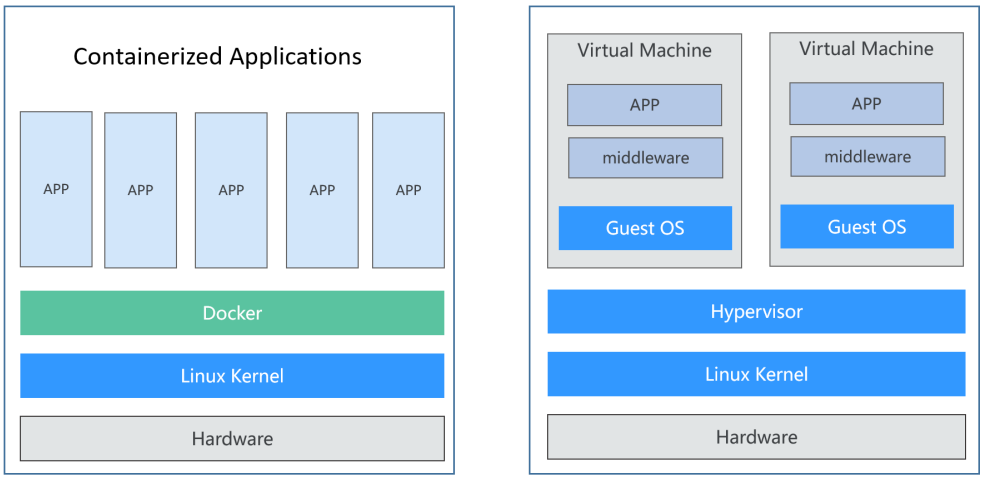
3.1 容器

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 3-1 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。

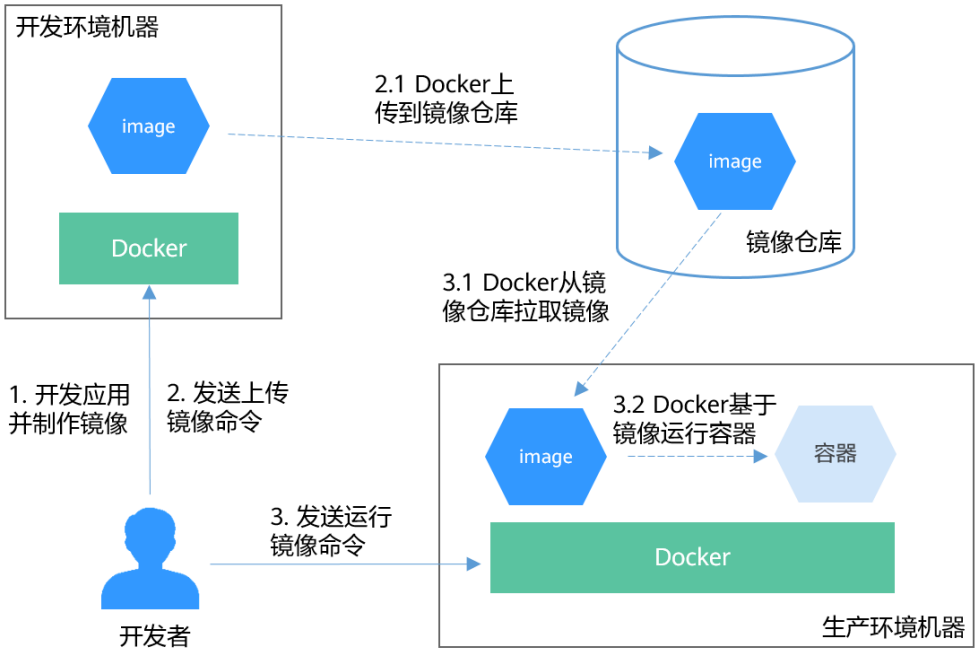
Docker 容器典型使用流程

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3-2所示：

图 3-2 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。

以“CentOS 7.5 64bit(40GiB)”操作系统为例，使用华为云镜像快速安装Docker。

1. 添加yum源。

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

2. 安装yum-util。

   ```
   # yum install -y yum-utils device-mapper-persistent-data lvm2
   ```

3. 设置Docker yum源。

   ```
   # yum-config-manager --add-repo https://mirrors.huaweicloud.com/docker-ce/linux/centos/docker-ce.repo
   # sed -i 's+download.docker.com+mirrors.huaweicloud.com/docker-ce+' /etc/yum.repos.d/docker-ce.repo
   ```


4. 安装并运行Docker。
```
# yum -y install docker-ce
# systemctl enable docker
# systemctl start docker
```
5. 检查安装结果。
```
# docker --version
Docker version 26.1.4, build 5650f9b
```

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。通过Dockerfile定制一个简单的Nginx镜像。

1. 创建一个名为Dockerfile的文件。
```
# mkdir mynginx
# cd mynginx
# touch Dockerfile
```
2. 编辑Dockerfile。
```
# vim Dockerfile
```
增加文件内容如下:
```
# 使用Nginx镜像作为基础镜像
FROM hub.atomgit.com/amd64/nginx:1.25.2-perl

# 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
```
3. 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。结尾的符号. 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    1ff61881be30   10 seconds ago   236MB
```

把镜像推送到镜像仓库

1. 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。

2. 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。

3. 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
```
4. 当需要使用该镜像时，使用docker pull命令拉取（下载）该镜像即可。
```
# docker pull swr.cn-east-3.myhuaweicloud.com/container/hello:v1
```

3.2 Kubernetes

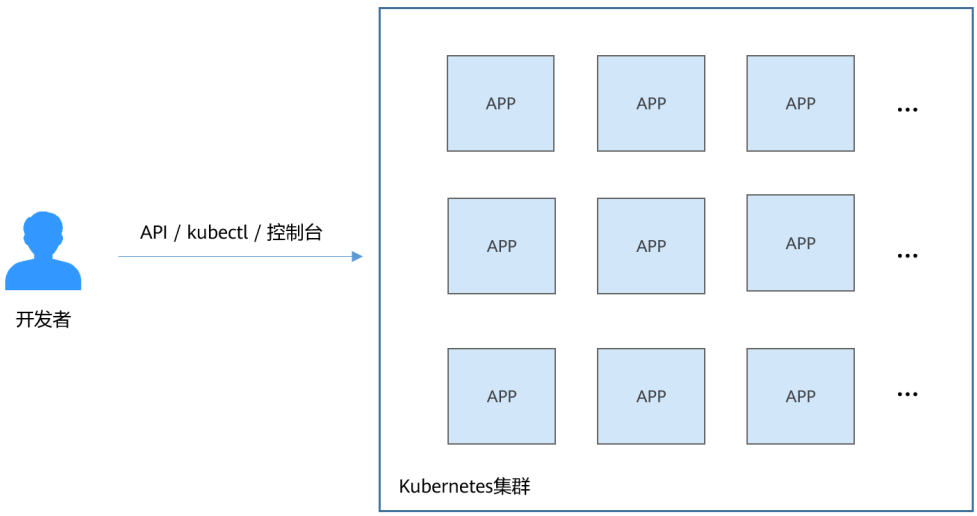
Kubernetes 是什么

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 3-3 在 Kubernetes 集群上运行应用程序



Kubernetes 集群架构

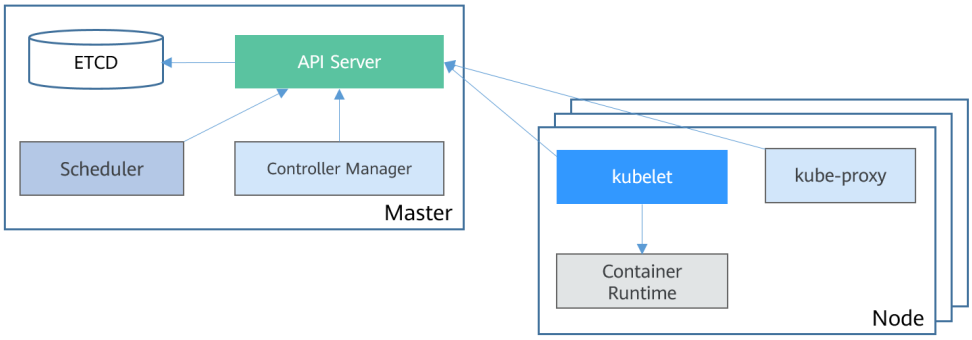
Kubernetes集群包含Master节点（控制节点）和Node节点（计算节点/工作节点），应用部署在Node节点上，且可以通过配置选择应用部署在某些特定的节点上。

说明

通过云容器引擎服务创建的集群，Master节点将由云容器引擎服务托管，您只需创建Node节点。

Kubernetes集群的架构如下所示：

图 3-4 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。

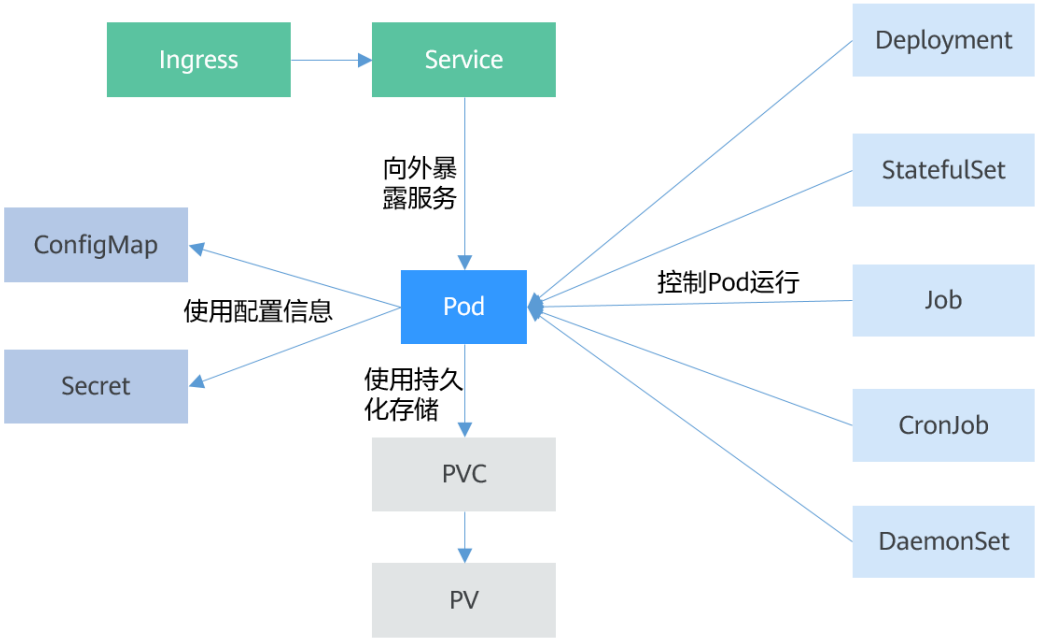
- CRI（Container Runtime Interface）：容器运行时接口，提供计算资源，CRI隔离了各个容器引擎之间的差异，而通过统一的接口与各个容器引擎之间进行互动。

- 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CCE就是开发的CNI插件支持Kubernetes集群运行在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CCE就可以方便的对接块存储（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 3-5 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。

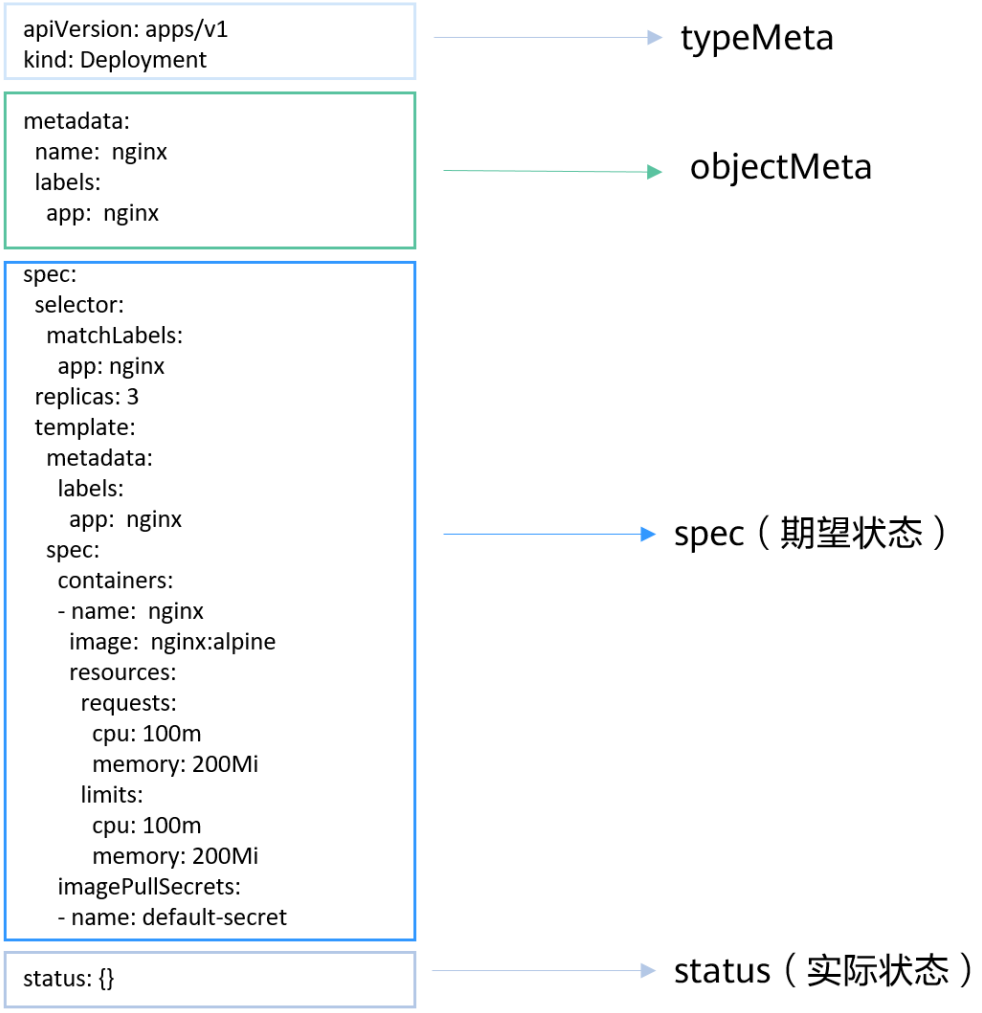
Kubernetes 对象的描述

kubernetes中资源可以使用YAML描述，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 3-6 YAML 描述文件



在 Kubernetes 上运行应用

将图3-6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。

3.3 使用 Kubectl 命令操作集群

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
```

更多kubectl命令请参考**kubectl 快速参考**。

基础命令

get

get命令用于获取集群的一个或一些resource信息。

该命令可以列出集群所有资源的详细信息，resource包括集群节点、运行的Pod、Deployment、Service等。

须知

集群中可以创建多个namespace，未指定namespace的情况下，所有操作都是针对--namespace=default。

例如：

获取所有pod的详细信息：

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

获取所有namespace下的运行的所有pod：

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

获取所有namespace下的运行的所有pod的标签：

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

获取该节点的所有命名空间：

```
kubectl get namespace
```

说明

查询其他节点需要加-s指定节点，类似可以使用“kubectl get svc”，“kubectl get nodes”，“kubectl get deploy”等获取其他resource信息。

以yaml格式输出pod的详细信息：

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

以json格式输出pod的详细信息：

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

说明

其中LABELS为显示的列标题，可以自己设置，“.metadata.labels.app”为查询的数据需要按照之前的yaml或json获取。

create

kubectl命令用于根据文件或输入创建集群resource。

如果已经定义了相应resource的yaml或json文件，直接kubectl create -f filename即可创建文件内定义的resource。

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

expose

expose将一个资源包括Pod、Service、Deployment等公开为一个新的Service。


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

说明

给deployname发布一个服务，--port为服务暴露出去的端口，--type为服务类型，--target-port为服务对应后端Pod的端口，port提供了集群内部访问服务的入口，即ClusterIP:port。

run

在集群中运行一个特定的镜像。

例如：

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

在创建时指定运行的命令：

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

set

在对象上设置特定功能。

例如：

滚动更新一个deployment的容器镜像改为1.0版本：

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

edit

edit提供了另一种更新resource源的操作。

例如：

使用edit直接更新pod的命令为：

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

上面命令的效果等效于：

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

explain

查看文档或参考资料。

例如：

查看pod的相关文档：

```
kubectl explain pod
```

delete

根据resource名或label删除resource。

例如：

立刻删除该pod：

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


部署命令

rollout

管理资源的发布。

例如：

查看指定资源的部署状态：

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

查看指定资源的发布历史：

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

回滚指定资源，默认回滚至上一个版本：

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

scale

scale用于程序在负载加重或缩小时将副本进行扩容或缩小。

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

autoscale

autoscale命令提供了自动根据pod负载对其副本进行扩缩的功能。autoscale命令会给一个rc指定一个副本数的范围，在实际运行中根据pod中运行的程序的负载自动在指定的范围内对pod进行扩容或缩容。

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

集群管理命令

cordon、drain、uncordon*

有时候会遇到这样一个场景，一个node需要升级，但是在该node上又有许多运行的pod，或者该node已经瘫痪，需要保证功能的完善，则需要使用这组命令，使用步骤如下：

步骤1 使用cordon命令将一个node标记为不可调度。这意味着新的pod将不会被调度到该node上。

```
kubectl cordon <nodename>
```

备注：CCE中nodename为节点私网IP。

步骤2 使用drain命令，将运行在该node上运行的pod平滑的搬迁到其他节点上。

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

备注：ignore-emptydir为用户挂载空目录的数据。

步骤3 对该节点进行一些节点维护的操作，如重置节点等。

步骤4 节点维护完后，使用uncordon命令解锁该node，使其重新变得可调度。

```
kubectl uncordon <nodename>
```

----结束

cluster-info

查看在集群中运行的插件：


```
kubectl cluster-info
```

查看详细信息：

```
kubectl cluster-info dump
```

top*

显示资源（CPU/Memory/Storage）使用。需要Heapster运行。

taint*

修改一个或多个节点上的taint。

certificate*

修改证书资源。

故障诊断和调试命令

describe

describe类似于get，同样用于获取resource的相关信息。不同的是，get获得的是更详细的resource个性的详细信息，describe获得的是resource集群相关的信息。describe命令同get类似，但是describe不支持-o选项，对于同一类型resource，describe输出的信息格式，内容域相同。

说明

如果发现是查询某个resource的信息，使用get命令能够获取更加详尽的信息。但是如果想要查询某个resource的状态，如某个pod并不是在running状态，这时需要获取更详尽的状态信息时，就应该使用describe命令。

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

logs

logs命令用于显示pod运行中，容器内程序输出到标准输出的内容。如果要获得tail -f的方式，需使用-f选项。

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

exec

与docker的exec用法相似，如果一个pod中，有多个容器，需要使用-c选项指定容器。

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

port-forward*

转发一个或多个本地端口至一个pod。

例如：

侦听本地端口 5000 并转发到 <my-deployment> 创建的 Pod 里的端口 6000：

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

cp

复制文件或目录到容器：

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

将 /tmp/foo 本地文件复制到远程 Pod 中特定容器的 /tmp/bar 下。

auth*

检查授权。

attach*

attach命令效果类似于logs -f，退出查看使用ctrl-c。如果一个pod中有多个容器，要查看具体的某个容器的输出，需要在pod名后使用-c containername指定运行的容器。

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

高级命令

replace

replace命令用于对已有资源进行更新、替换。当需要更新resource的一些属性的时候，如果修改副本数量，增加、修改label，更改image版本，修改端口等。都可以直接修改原yaml文件，然后执行replace命令。

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

须知

名字不能被更新。

apply*

apply命令提供了比patch，edit等更严格的更新resource的方式。通过apply，用户可以将resource的configuration使用source control的方式维护在版本库中。每次有更新时，将配置文件推送server，然后使用kubectl apply将更新应用到resource。Kubernetes会在引用更新前将当前配置文件中的配置同已经应用的配置做比较，并只更新更改的部分，而不会主动更改任何用户未指定的部分。apply命令的使用方式同replace相同，不同的是，apply不会删除原有resource，然后创建新的。apply直接在原有resource的基础上进行更新。同时kubectl apply还会在resource中添加一条注释，标记当前的apply，类似于git操作。

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

patch

如果一个容器已经在运行，这时需要对一些容器属性进行修改，又不想删除容器，或不方便通过replace的方式进行更新。Kubernetes还提供了一种在容器运行时，直接对容器进行修改的方式，就是patch命令。 例如已存在一个pod的label为app=nginx1，如果需要在运行过程中，将其修改为app=nginx2。

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

convert*

不同的api版本之间转换配置文件。

设置命令

label

更新资源上的标签：

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


annotate

更新资源上的注释：

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

completion

用于实现kubectl工具自动补全。

其他命令

api-versions

打印受支持的api版本：

```
kubectl api-versions
```

api-resources

打印支持的api资源：

```
kubectl api-resources
```

config*

修改kubeconfig文件：用于访问api，比如配置认证信息。

help

所有命令帮助。

version

打印客户端和服务版本信息。

```
kubectl version
```


4 Pod、Label 和 Namespace

4.1 Pod：Kubernetes 中的最小调度对象

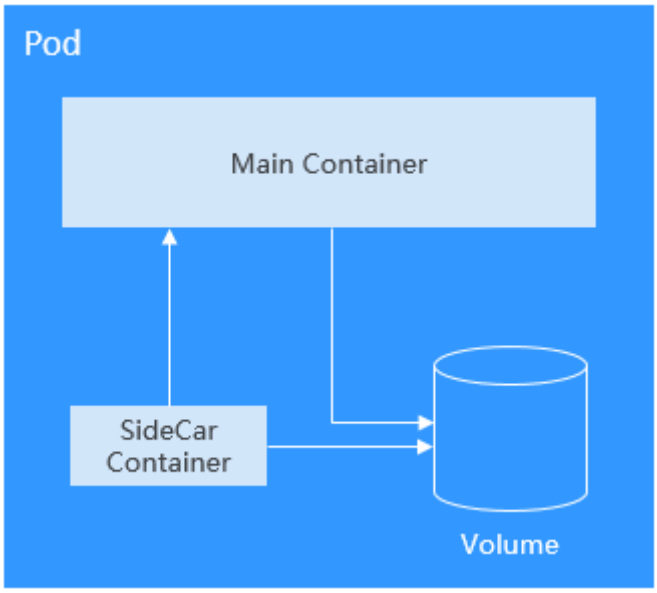
容器组（Pod）

容器组（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4-1所示，例如主容器为一个web服务器，从一个固定目录下对外提供文件服务，而辅助容器周期性的从外部下载文件存到这个固定目录下。

图 4-1 Pod



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

创建 Pod

kubernetes中资源可以使用YAML描述，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 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信号终止该进程。

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
```

4.2 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**7.4 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。

4.3 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。

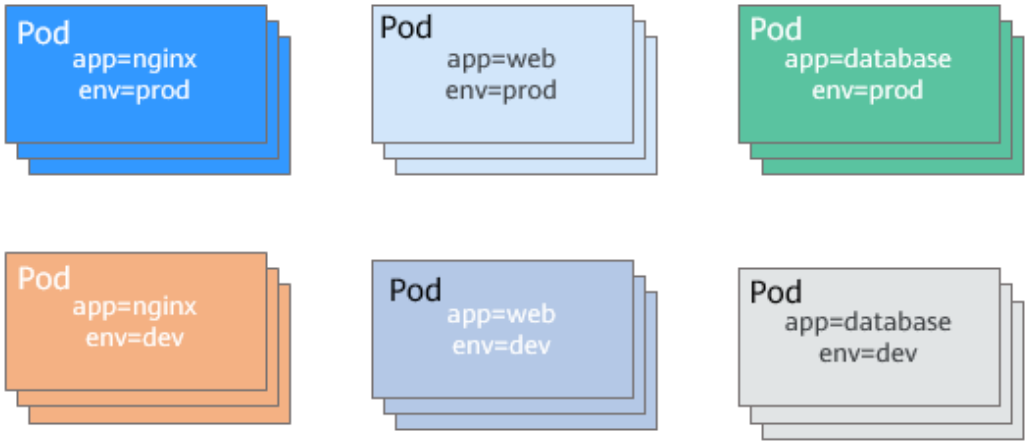
Label的具体形式是key-value的标记对，可以在创建资源的时候设置，也可以在后期添加和修改。

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 4-2 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 4-3 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
```

4.4 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下的资源名称可以相同，Kubernetes中大部分资源可以用Namespace划分，不过有些资源不行，例如Node、PV等，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是用于对接存储服务、icagent是用于对接监控系统。

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。

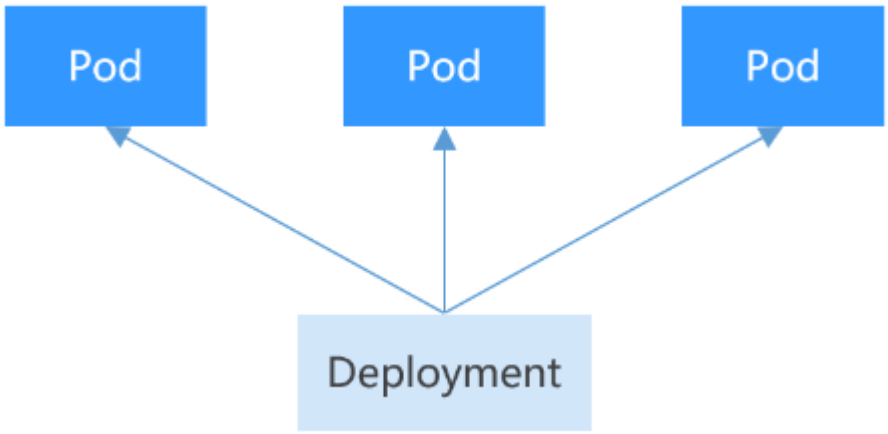
5 Pod 的编排与调度

5.1 无状态负载（Deployment）

无状态负载（Deployment）

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 5-1 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 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          1s
```


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。

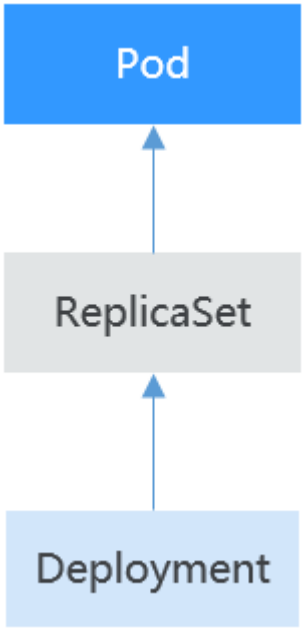
您也许会发现这两个后缀中前面一部分是相同的，都是7f98958cdf，这是因为Deployment不是直接控制Pod的，Deployment是通过一种名为ReplicaSet的控制器控制Pod，通过如下命令可以查询ReplicaSet，其中rs是ReplicaSet的缩写。

```
$ 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5-2所示，Deployment控制ReplicaSet，ReplicaSet控制Pod。

图 5-2 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          1m
nginx-6f9f58dffd-tesqr   1/1       Running   0          1m
```

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。

5.2 有状态负载（StatefulSet）

有状态负载（StatefulSet）

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。

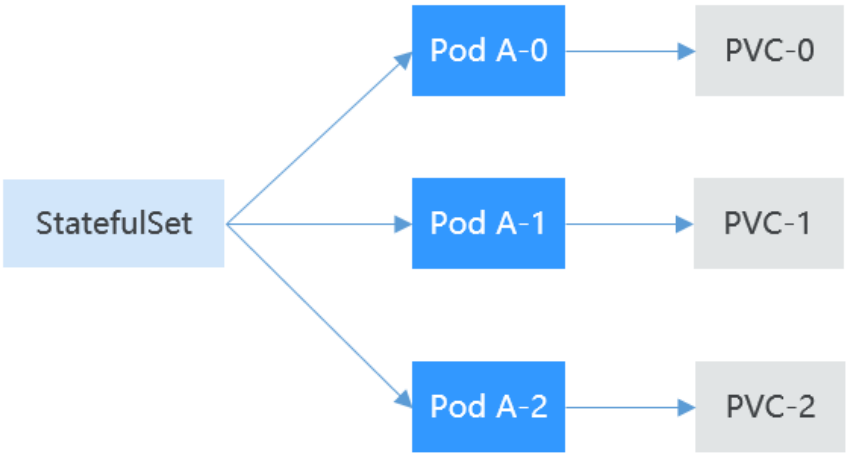
但是在某些场景下，这并不满足需求，比如有些分布式的场景，要求每个Pod都有自己单独的状态时，比如分布式数据库，每个Pod要求有单独的存储，这时Deployment无法满足业务需求。

分布式有状态应用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，在Kubernetes中部署有状态应用对Pod有如下要求：

- Pod能够被别的Pod找到，要求Pod有固定的标识。
- 每个Pod有单独存储，Pod被删除恢复后，必须读取原来的数据，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。
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。



创建 Headless Service

如前所述，创建Statefulset需要一个**Headless Service**用于Pod访问。

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**8.2 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**7 Kubernetes网络**会详细介绍CoreDNS的作用。

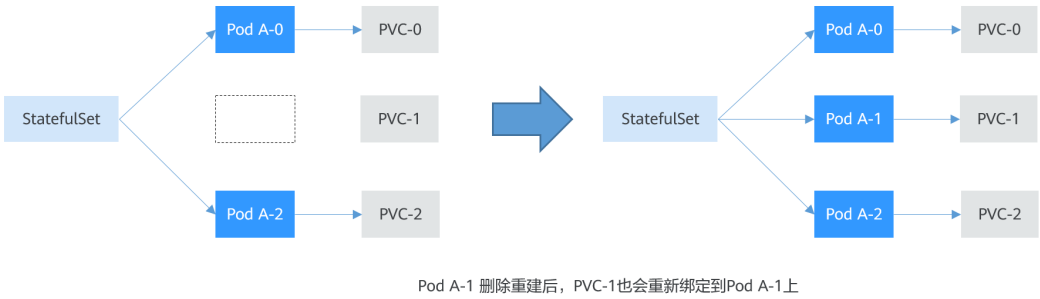
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-0.nginx
Server:         10.247.3.10
Address:        10.247.3.10#53
Name:   nginx-0.nginx.default.svc.cluster.local
Address: 172.16.0.31

/ # 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 5-3 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
```

5.3 普通任务（Job）和定时任务（CronJob）

普通任务（Job）和定时任务（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: 100  # 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 5-1 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/v1
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: pi
            image: perl
            command:
```



```
        - perl
        - "-Mbignum=bpi"
        - "-wle"
        - print bpi(2000)
```

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**。

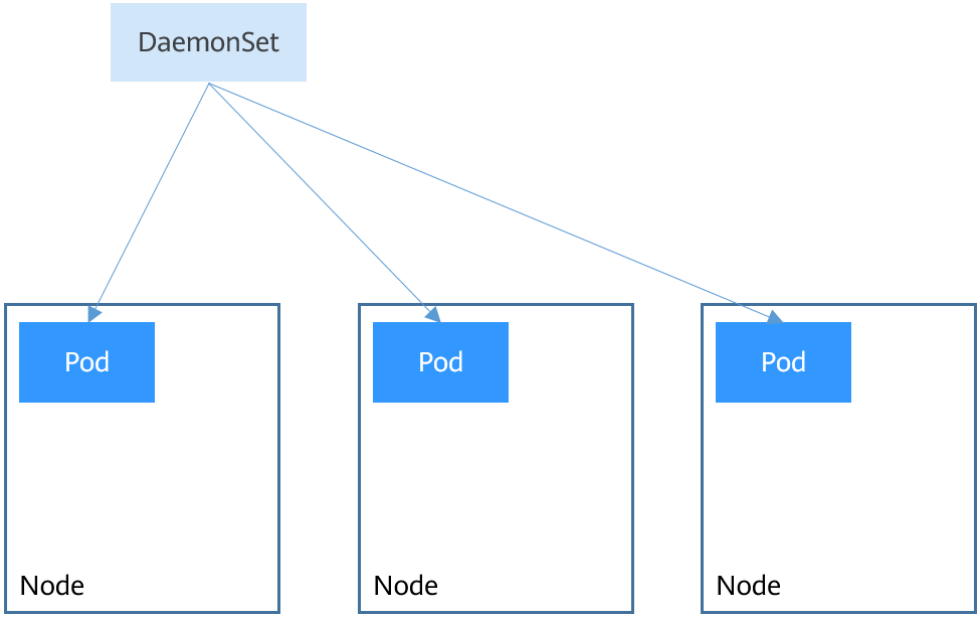
5.4 守护进程集（DaemonSet）

守护进程集（DaemonSet）

DaemonSet（守护进程集）在集群的每个节点上运行一个Pod，且保证只有一个Pod，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 5-4 DaemonSet



创建 DaemonSet

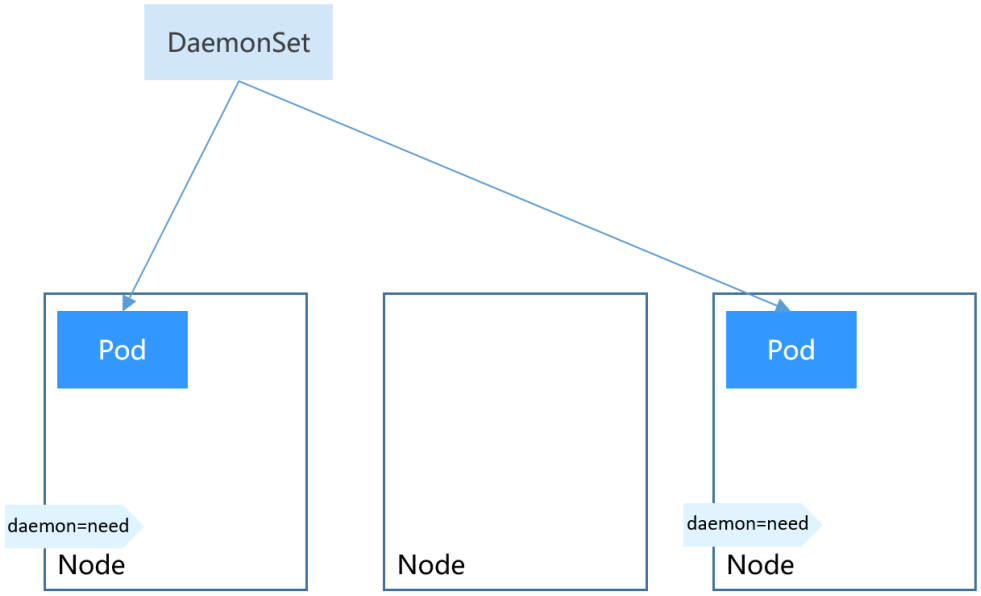
下面是一个DaemonSet的示例。

```
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 5-5 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
```


5.5 亲和与反亲和调度

在**5.4 守护进程集（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**4.3 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**5.4 守护进程集（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）才会调度Pod到节点上。
- 后半段IgnoredDuringExecution表示已经在节点上运行的Pod不需要满足下面定义的规则，即去除节点上的某个标签，那些需要节点包含该标签的Pod不会被重新调度。

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
```



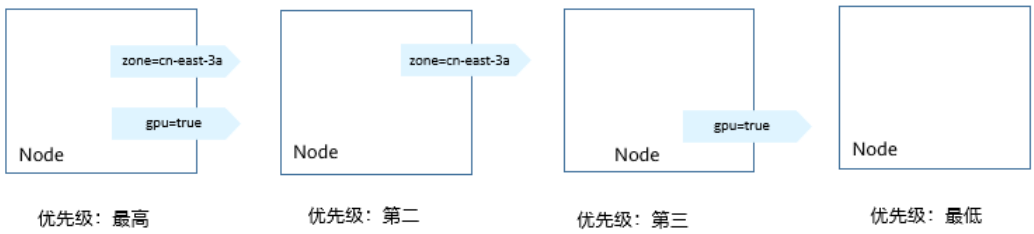
```
        limits:
          cpu:  100m
          memory:  200Mi
      imagePullSecrets:
      - name: default-secret
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 80
            preference:
              matchExpressions:
              - 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 5-6 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。

工作负载亲和（podAffinity）

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。

说明

对于工作负载亲和来说，使用requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution规则时， topologyKey字段不允许为空。

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:
              matchExpressions:
              - key: app
                operator: In
                values:
                - 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（用于划分拓扑域），意思是先圈定topologyKey指定的范围，当节点上的标签键、值均相同时会被认为同一拓扑域，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
```

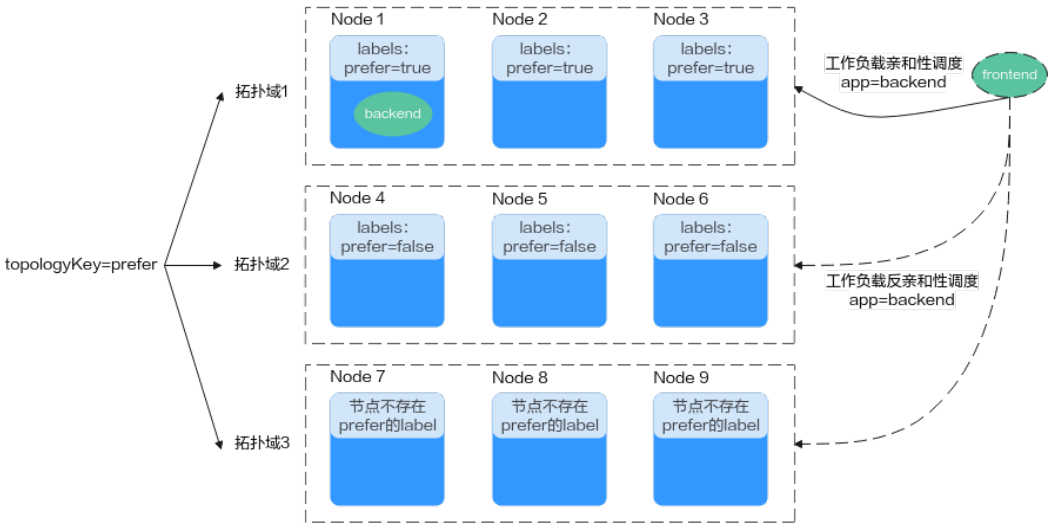
给192.168.0.97和192.168.0.94打上prefer=true的标签。

```
$ 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，则节点拓扑域的划分如图5-7所示。

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

图 5-7 拓扑域示意图



调度时，会根据prefer标签划分节点拓扑域，本示例中192.168.0.97和192.168.0.94被划作同一拓扑域。如果当拓扑域中运行着app=backend的Pod，即使该拓扑域中并非所有节点均运行了app=backend的Pod（本例该拓扑域中仅192.168.0.97节点上存在app=backend的Pod），frontend同样会部署在此拓扑域中（这里的192.168.0.97或192.168.0.94）。


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

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

工作负载反亲和（podAntiAffinity）

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。

说明

对于工作负载反亲和来说，使用requiredDuringSchedulingIgnoredDuringExecution规则时，Kubernetes默认的准入控制器 LimitPodHardAntiAffinityTopology要求topologyKey字段只能是kubernetes.io/hostname。如果您希望使用其他定制拓扑逻辑，可以更改或者禁用该准入控制器。

下面例子中定义了反亲和规则，这个规则表示根据kubernetes.io/hostname标签划分节点拓扑域，且如果该拓扑域中的某个节点上已经存在带有app=frontend标签的Pod，那么拥有相同标签的Pod将不能被调度到该拓扑域内的其他节点上。

```
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    #节点拓扑域
            labelSelector:    #Pod标签匹配规则
              matchExpressions:
              - key: app
                operator: In
                values:
                - frontend
```

创建并查看部署效果，示例中根据kubernetes.io/hostname标签划分节点拓扑域，在拥有kubernetes.io/hostname标签的节点中，每个节点的标签值均不同，因此一个拓扑域中只有一个节点。当一个拓扑域中（此处为一个节点）已经存在frontend标签的Pod时，该拓扑域不会被继续调度具有相同标签的Pod。本例中只有4个节点，因此还有一个Pod处于Pending状态无法调度。

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

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


6 配置管理

6.1 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
```

6.2 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”。

7 Kubernetes 网络

7.1 容器网络

Kubernetes本身并不负责网络通信，但提供了容器网络接口CNI（Container Network Interface），具体的网络通信交由CNI插件来实现。开源的CNI插件非常多，像Flannel、Calico等。针对Kubernetes网络，CCE为不同网络模型的集群提供不同的网络插件实现，用于负责集群内网络通信。

Kubernetes虽然不负责搭建网络模型，但要求集群网络满足以下要求：

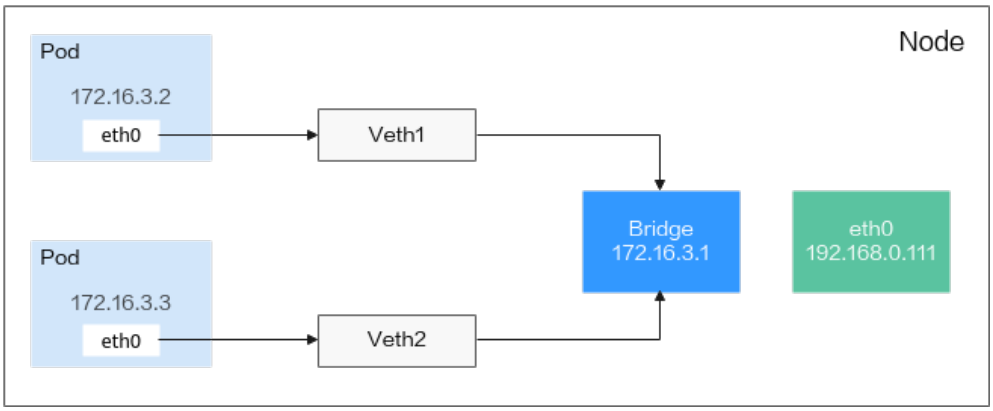
- Pod能够互相通信，且Pod必须通过非NAT网络连接，即收到的数据包的源IP就是发送数据包Pod的IP。
- 节点之间可以在非NAT网络地址转换的情况下通信。

Pod 通信

同一个节点中的Pod通信

Pod通过虚拟Ethernet接口对（Veth Pair）与外部通信，Veth Pair像一根网线，一端在Pod内部，一端在Pod外部。同一个节点上的Pod通过网桥（Linux Bridge）通信，如下图所示。

图 7-1 同一个节点中的 Pod 通信



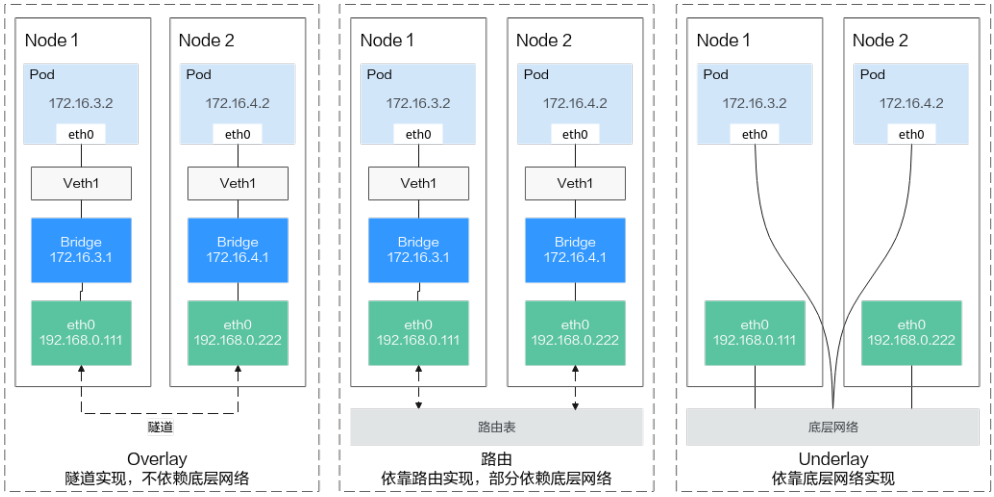
在同一节点上的Pod会通过Veth设备将一端连接到网桥，且它们的IP地址是通过网桥动态获取的，和网桥IP属于同一网段。此外，同一节点上的所有Pod默认路由都指向网桥，网桥会负责将所有非本地地址的流量进行转发。因此，同一节点上的Pod可以直接通信。

不同节点上的Pod通信

Kubernetes要求集群Pod的地址唯一，因此集群中的每个节点都会分配一个子网，以保证Pod的IP地址在整个集群内部不会重复。在不同节点上运行的Pod通过IP地址互相访问，该过程需要通过集群网络插件实现，按照底层依赖大致可分为Overlay模式、路由模式、Underlay模式三类。

- Overlay模式是在节点网络基础上通过隧道封装构建的独立网络，拥有自己独立的IP地址空间、交换或者路由的实现。VXLAN协议是目前最流行的Overlay网络隧道协议之一。
- 路由模式采用VPC路由表的方式与底层网络相结合，能够更加便捷地连接容器和主机，在性能上会优于Overlay的隧道封装。
- Underlay模式是借助驱动程序将节点的底层网络接口直接暴露给容器使用的一种网络构建技术，享有较高的性能，较为常见的解决方案有IP VLAN等。

图 7-2 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7.2 Service和7.3 Ingress。

7.2 Service

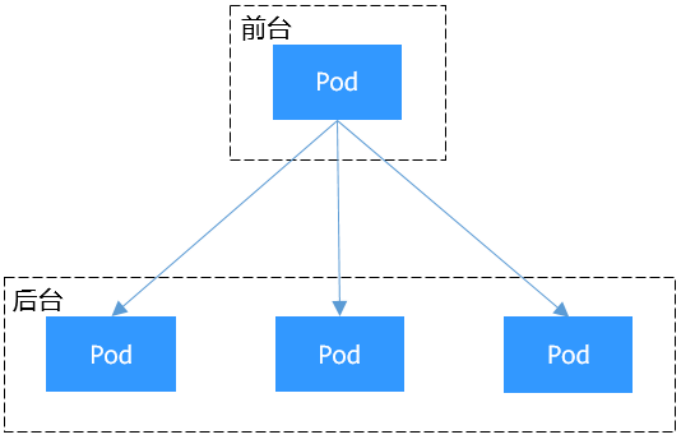
直接访问 Pod 的问题

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7-3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 7-3 Pod 间访问

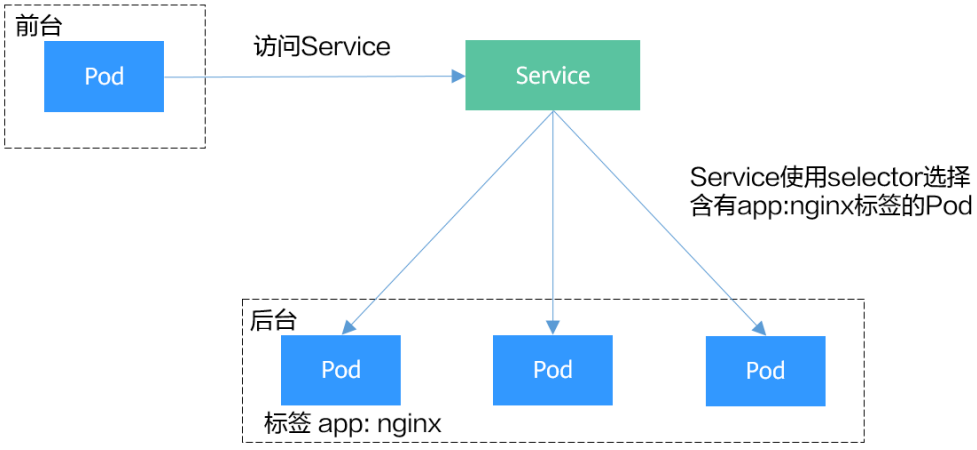


使用 Service 解决 Pod 的访问问题

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7-4所示。

图 7-4 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，您会看到如下信息。

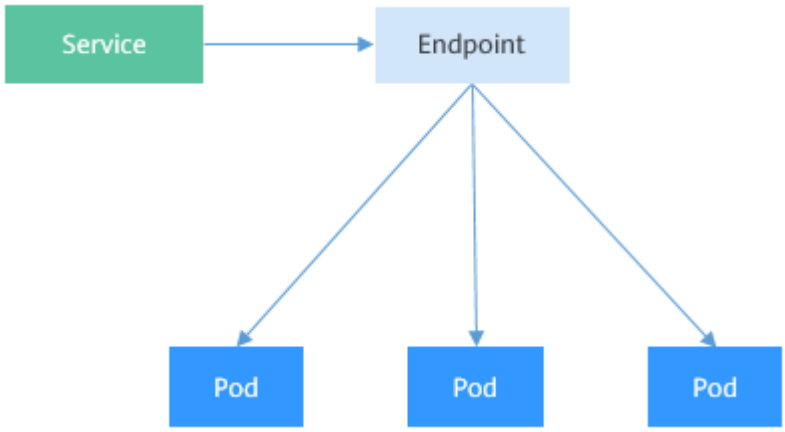
```
$ kubectl describe svc nginx
Name:              nginx
......
Endpoints:         172.16.2.132:80,172.16.3.6:80,172.16.3.7:80
......
```

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
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
```

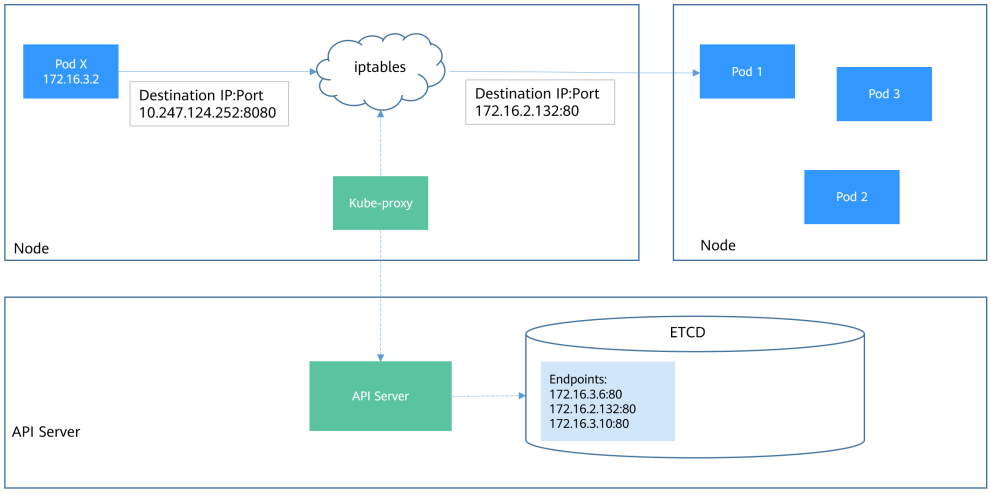
下面进一步了解这又是如何实现的。

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对应的IP和端口，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对应的IP和端口，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 7-5 Pod X 访问 Service 的过程



Service 的类型与使用场景

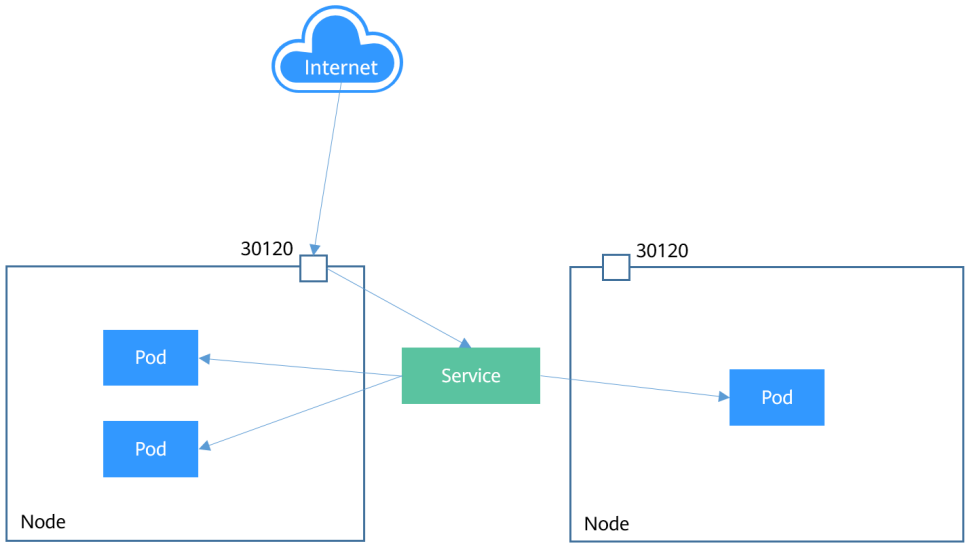
Service的类型除了ClusterIP还有NodePort、LoadBalancer和Headless Service，这几种类型的Service有着不同的用途。

- 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**。
- Headless Service：用于Pod间的互相发现，该类型的Service并不会分配单独的ClusterIP， 而且集群也不会为它们进行负载均衡和路由。您可通过指定spec.clusterIP字段的值为“None”来创建Headless Service，详细介绍请参见**Headless Service**。

NodePort 类型的 Service

NodePort类型的Service可以让Kubernetes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 7-6 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。

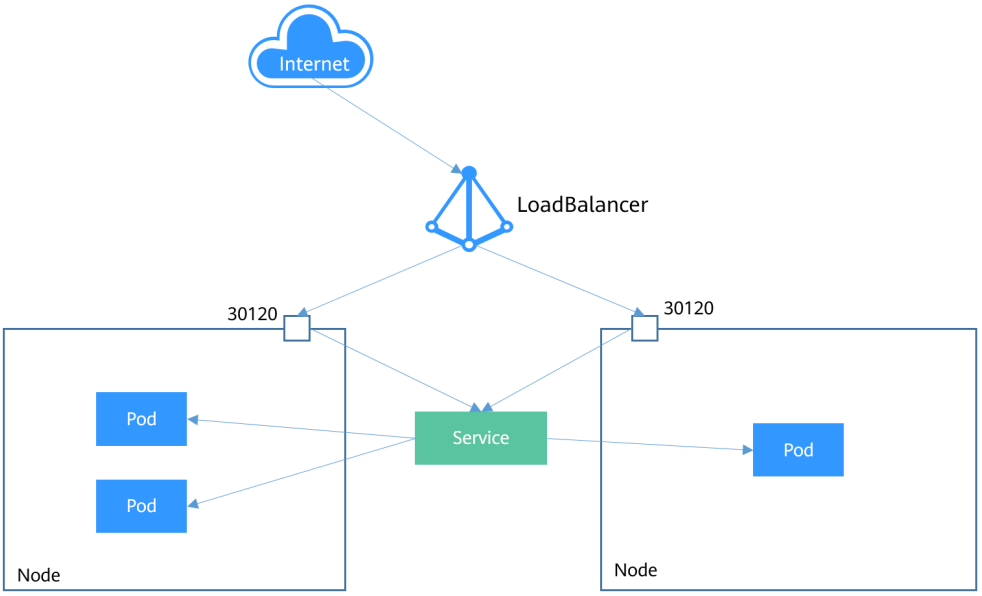
```
$ kubectl run -i --tty --image nginx:alpine test --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl 192.168.0.212:30120
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
.....
```

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CCE对接了ELB。这就导致了创建LoadBalancer类型的Service有不同的实现。

图 7-7 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，允许客户端连接到Service关联的某个Pod。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

为了解决以上问题，Kubernetes提供了另一种较为特殊的Service类型，称为Headless Service。对于其他Service来说，客户端在访问服务时，DNS查询时只会返回Service的ClusterIP地址，具体访问到哪个Pod是由集群转发规则（IPVS或iptables）决定的。而Headless Service并不会分配单独的ClusterIP，在进行DNS查询时会返回所有Pod的DNS记录，这样就可查询到每个Pod的IP地址。**5.2 有状态负载（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。

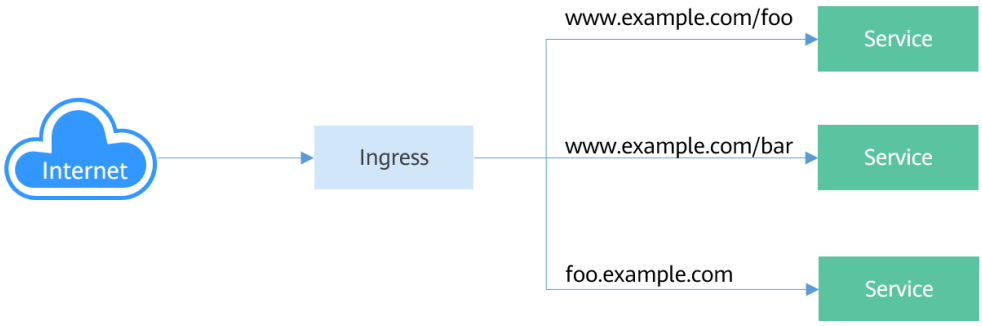
```
$ kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup nginx-headless
Server:         10.247.3.10
Address:        10.247.3.10#53

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

7.3 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 7-8 Ingress-Service

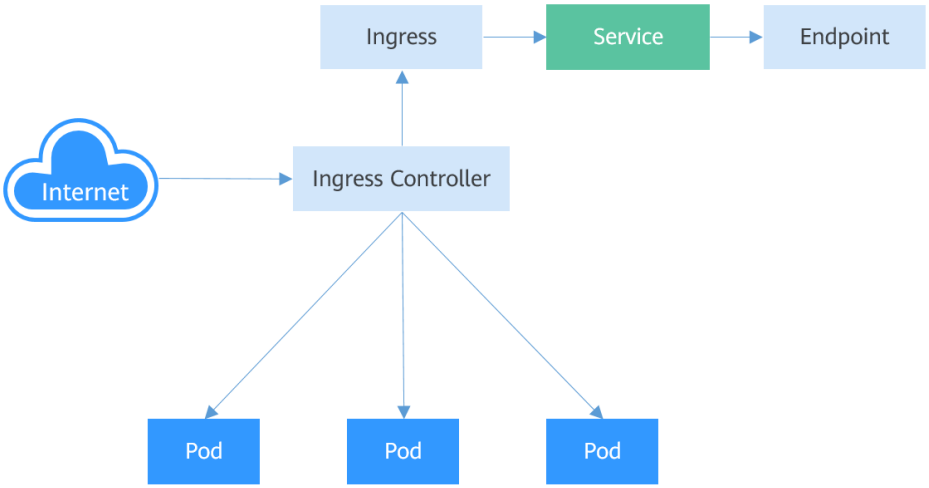


Ingress 工作机制

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CCE使用弹性负载均衡服务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 7-9 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/”时，流量转发“nginx:8080”对应的Service，从而将流量转发到对应Pod。

示例如下（适用于v1.23及以上版本的集群）：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.port: '8080'
```



```
    kubernetes.io/elb.id: aa7cf5ec-7218-4c43-98d4-c36c0744667a
spec:
  rules:
    - host: ''
      http:
        paths:
          - path: /
            backend:
              service:
                name: nginx
                port:
                  number: 8080
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
  ingressClassName: cce
```

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:
              service:
                name: nginx
                port:
                  number: 8080
...
```

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。

须知

Ingress转发策略中的path路径要求后端应用内存在相同的路径，否则转发无法生效。

例如，Nginx应用默认的Web访问路径为“/usr/share/nginx/html”，在为Ingress转发策略添加“/test”路径时，需要应用的Web访问路径下也包含相同路径，即“/usr/share/nginx/html/test”，否则将返回404。

```
...
spec:
  rules:
    - host: foo.bar.com          # host地址
      http:
        paths:
          - path: "/foo"
            backend:
              service:
```



```
            name: s1
            port:
              number: 80
      - path: "/bar"
        backend:
          service:
            name: s2
            port:
              number: 80
...
```

7.4 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。

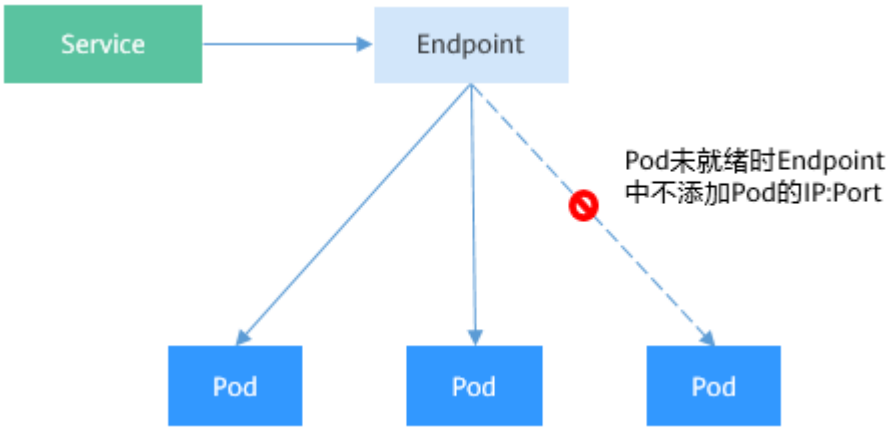
Readiness Probe同样是周期性的检测Pod，然后根据响应来判断Pod是否就绪，与**4.2 存活探针（Liveness Probe）**相同，就绪探针也支持如下三种类型。

- 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 7-10 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-ready.yaml文件中，删除之前创建的Deployment，用deploy-ready.yaml创建这个Deployment。

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

# kubectl create -f deploy-ready.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 探测连续1次成功表示成功
- #failure=3 探测连续3次失败表示失败

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
```

7.5 NetworkPolicy

NetworkPolicy是Kubernetes设计用来限制Pod访问的对象，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。

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。（仅Egress支持IPBlock）

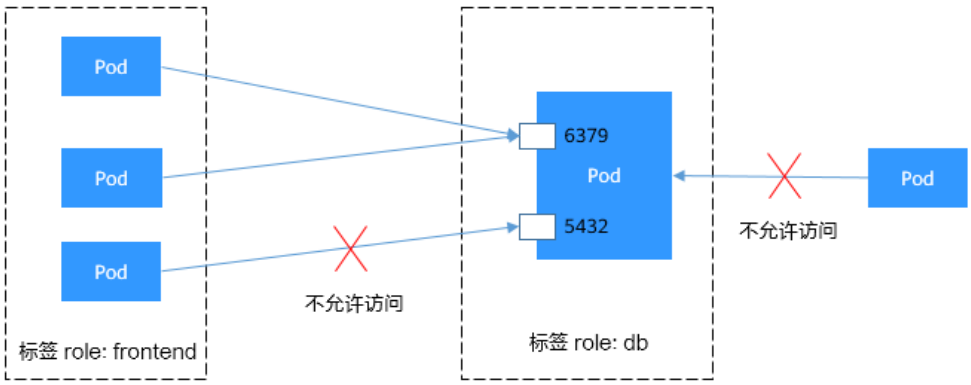
使用 Ingress 规则

- **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:                  # 规则对具有role=db标签的Pod生效
    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 7-11 podSelector

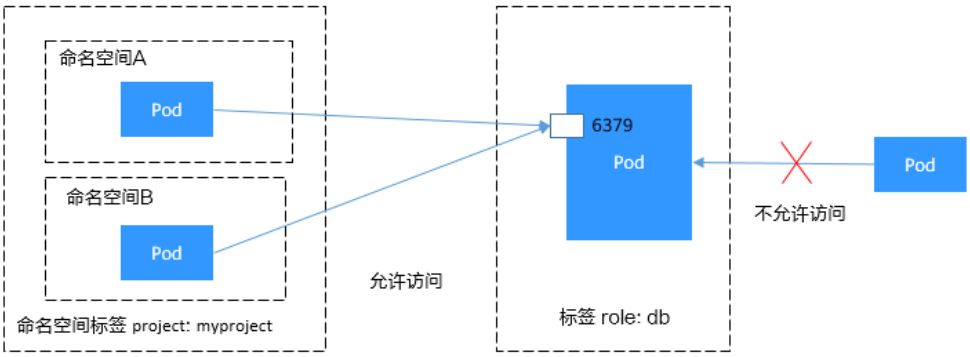


- **使用namespaceSelector设置访问范围**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector:                  # 规则对具有role=db标签的Pod生效
    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 7-12 namespaceSelector



使用 Egress 规则

Egress不仅支持podSelector和namespaceSelector，还支持ipBlock。

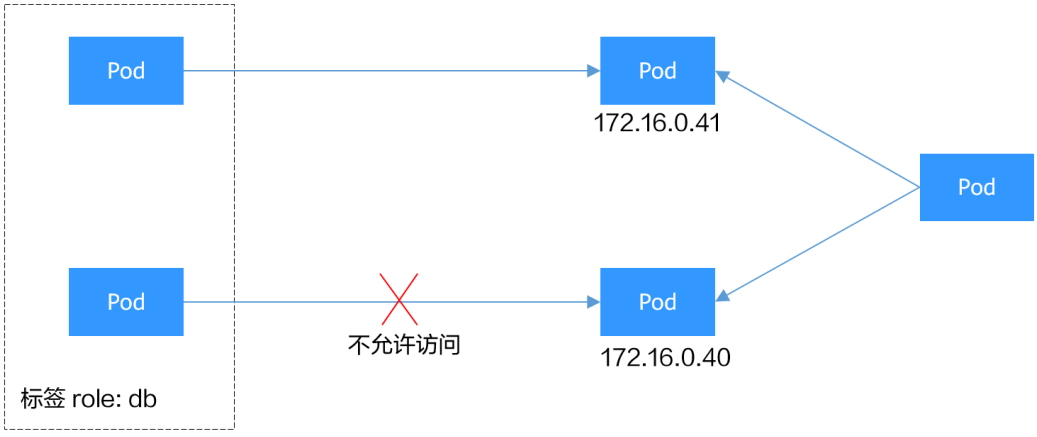
说明

仅1.23及以上版本集群支持Egress规则。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-client-a-via-except-cidr-egress-rule
  namespace: default
spec:
  policyTypes:                  # 使用Egress必须指定policyType
    - Egress
  podSelector:                  # 规则对具有role=db标签的Pod生效
    matchLabels:
      role: db
  egress:                       # 表示出规则
  - to:
    - ipBlock:
        cidr: 172.16.0.16/16    # 允许此网段被访问
        except:
        - 172.16.0.40/32        # 不允许此网段被访问，except需在cidr网段内
```

示意图如下所示。

图 7-13 ipBlock

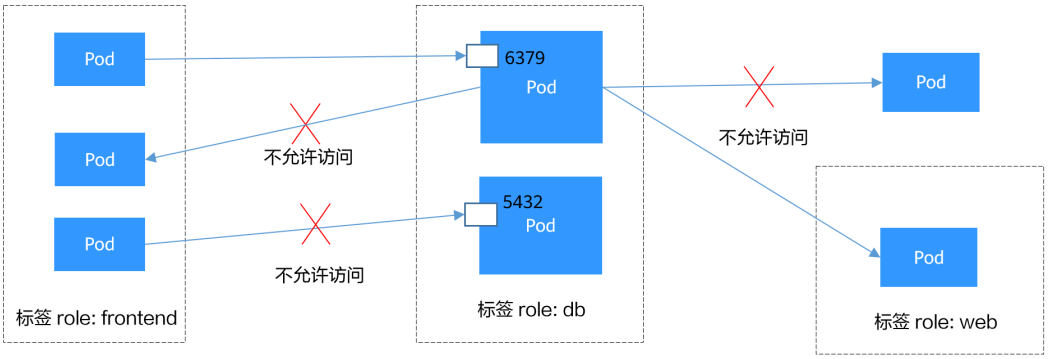


Ingress和Egress可以定义在同一个规则中。


```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  policyTypes:
  - Ingress
  - Egress
  podSelector:                # 规则对具有role=db标签的Pod生效
    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
  egress:                     # 表示出规则
  - to:
    - podSelector:            # 只允许访问具有role=web标签的Pod
        matchLabels:
          role: web
```

示意图如下所示。

图 7-14 同时使用 Ingress 和 Egress



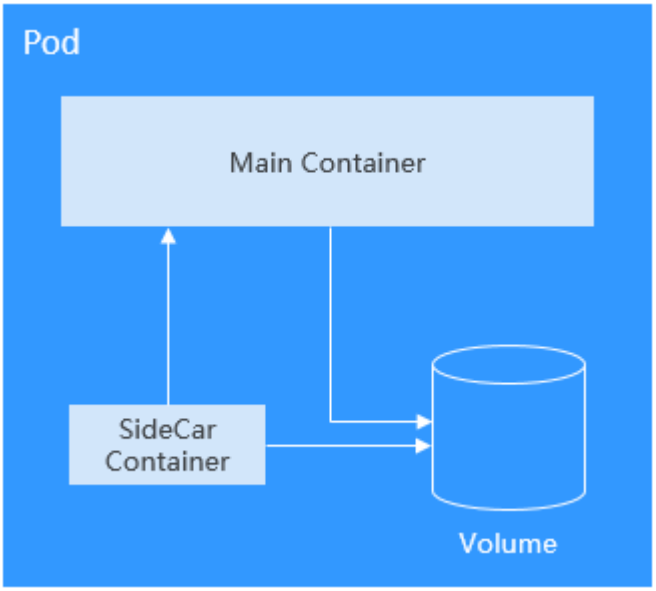
8 持久化存储

8.1 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**6.1 ConfigMap**和**6.2 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**8.2 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
```

8.2 PV、PVC 和 StorageClass

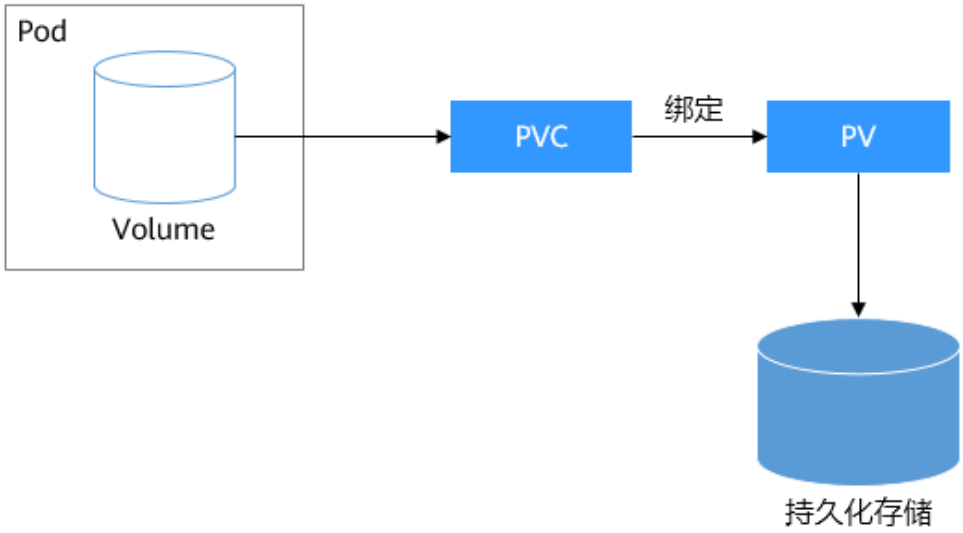
上一章节介绍的HostPath是一种持久化存储，但是HostPath的内容是存储在节点上，导致只适合读取。

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。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 8-1 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**4.4 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CCE开发存储控制器和驱动。有了这些驱动就可以使用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。

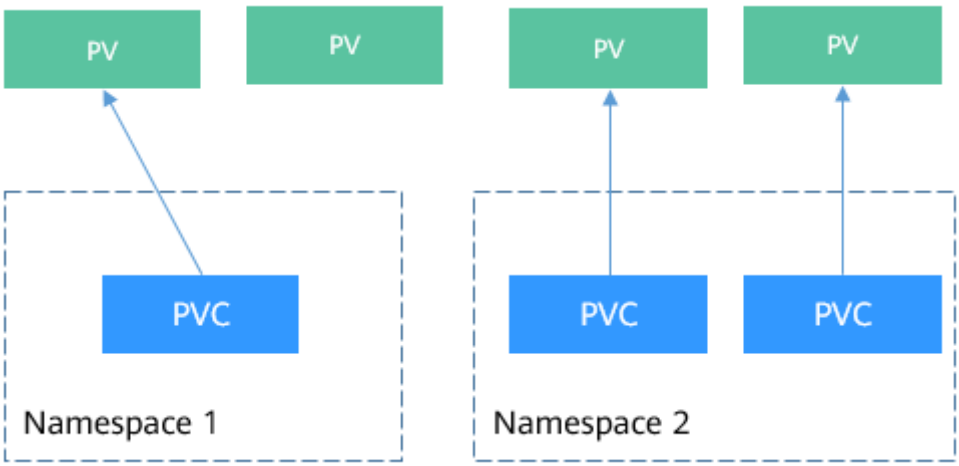
再来看下PV。

```
$ 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 8-2 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。

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StorageClass创建PVC。

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

说明

当前不支持使用csi-sfsturbo类型StorageClass直接创建PVC。如需使用SFS Turbo类型的存储，请提前创建SFS Turbo实例并通过静态存储卷的方式创建PV和PVC，详情请参见**通过静态存储卷使用已有极速文件存储**。

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，这里只是使用了文件存储作为示例。

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**5.2 有状态负载（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
```

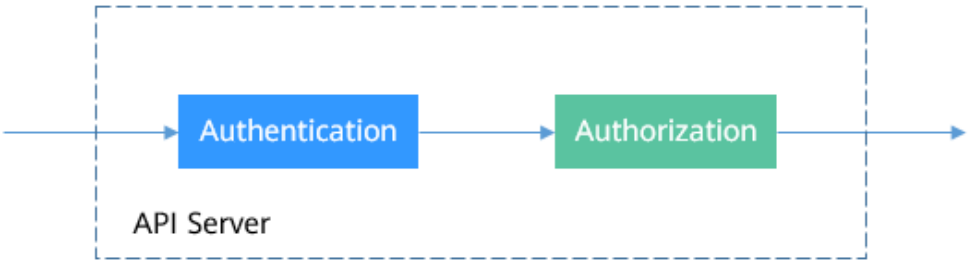

9 认证与授权

9.1 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分外部服务账号和内部服务账号，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**9.2 RBAC**介绍。

图 9-1 API Server 的认证授权



认证与 ServiceAccount

Kubernetes的用户分为服务账户（ServiceAccount）和普通账户两种类型。

- 服务账户与Namespace绑定，关联一套凭证，Pod创建时挂载Token，从而允许与API Server之间调用。
- Kubernetes中没有代表普通账户的对象，这类账户默认由外部服务独立管理，比如在CCE的用户是由IAM管理的。

与Pod、ConfigMap类似，ServiceAccount是Kubernetes中的资源，属于命名空间级别。当创建一个新的命名空间时，系统会自动在其中生成一个名为default的ServiceAccount。

使用下面命令可以查看ServiceAccount。

kubectl get sa

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

说明

- 1.21以前版本的集群中，Pod中获取Token的形式是通过挂载ServiceAccount的Secret来获取Token，这种方式获得的Token是永久的。该方式在1.21及以上的版本中不再推荐使用，并且根据社区版本迭代策略，在1.25及以上版本的集群中，ServiceAccount将不会自动创建对应的Secret。

  1.21及以上版本的集群中，直接使用**TokenRequest** API**获得Token**，并使用投射卷（Projected Volume）挂载到Pod中。使用这种方法获得的Token具有固定的生命周期，并且当挂载的Pod被删除时这些Token将自动失效。详情请参见**Token安全性提升说明**。
- 如果您在业务中需要一个永不过期的Token，您也可以选择**手动管理ServiceAccount的Secret**。尽管存在手动创建永久ServiceAccount Token的机制，但还是推荐使用**TokenRequest**的方式使用短期的Token，以提高安全性。

1.25以前版本的集群中，ServiceAccount会自动创建对应的Secret。1.25及以上版本的集群中，ServiceAccount将不会自动创建对应的Secret。下面分别查看两种集群下的ServiceAccount状态。

- 1.25以前版本集群，查看名为default的ServiceAccount状态。

  kubectl describe sa default

  回显内容如下，说明default自动创建对应的Secret，即default-token-vssmw。

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

- 1.25及以上版本集群，查看名为default的ServiceAccount状态。

  kubectl describe sa default

  由回显内容可知，default未自动创建对应的Secret。

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

在Pod的定义文件中，可以用指定账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

步骤1 以1.29版本集群为例，使用如下命令在default命名空间内创建ServiceAccount。

kubectl create serviceaccount *sa-example*

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

使用以下命令可以检查sa-example是否创建成功，若NAME列出现sa-example则说明创建成功。

kubectl get sa

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

由于本案例使用的集群版本在1.25以上，ServiceAccount将不会自动创建对应的Secret。使用以下命令可以查看创建的ServiceAccount的详细信息，回显中Mountable secrets和Tokens值为none，则说明该ServiceAccount没有自动创建Secret。

kubectl describe sa *sa-example*

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

步骤2 这里选择手动管理Secret，从而得到永不过期的Token。利用以下代码，手动创建名为sa-example-token的Secret，并与名为sa-example的ServiceAccount关联。

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

步骤3 检查sa-example-token是否创建成功。若命名空间default的Secrets中出现sa-example-token，则说明创建成功。

kubectl get secrets

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

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*

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

Type:  kubernetes.io/service-account-token

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

步骤4 检验ServiceAccount与新建的Secret关联是否成功，即检查ServiceAccount是否获取到Token。由回显内容可知，sa-example与sa-example-token关联成功。

kubectl describe sa *sa-example*


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

----结束

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。下面以“nginx:latest”为例，演示具体步骤。

步骤1 创建一个名为sa-pod.yaml的描述文件。其中，mysql.yaml为自定义名称，您可以随意命名。

vim *sa-pod.yaml*

须知

为了确保Pod能够使用手动创建的Secret中的Token，您需要明确地将该Secret挂载到容器中，挂载方式请参见描述文件中的加粗代码。

文件内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-pod
spec:
  serviceAccountName: sa-example                # 指定sa-example为Pod使用的服务账户
  imagePullSecrets:
  - name: default-secret
  containers:
  - image: nginx:latest
    name: container-0
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    volumeMounts:                               # 将名为secret-volume的存储卷挂载到Pod
    - name: secret-volume
      readOnly: true                            # 表示挂载的存储卷是只读的
      mountPath: "/etc/secret-volume"           # 指定存储卷在容器内部的挂载路径，可以自定义
  volumes:                                      # 定义Pod可以使用的Secret存储卷
  - name: secret-volume                         # 指定Secret存储卷的名称，可以自定义
    secret:                                     # 指定这个存储卷的类型为Secret
      secretName: sa-example-token              # 将之前建立的sa-example-token挂载到定义的存储卷
```

步骤2 创建并查看这个Pod，可以看到Pod挂载了sa-example-token，即Pod可以使用这个Token来做认证。

kubectl create -f *sa-pod.yaml*

回显内容如下：

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


使用以下代码，检验Pod是否创建成功。

kubectl get pod

回显内容如下，若sa-pod的状态为Running，则说明Pod创建成功

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

步骤3 查看sa-pod的具体信息，可以检验sa-example-token是否挂载成功。

kubectl describe pod *sa-pod*

回显内容如下：

```
...
Containers:
  container-0:
    Container ID:
    Image:          nginx:latest
    Image ID:
    Port:           <none>
    Host Port:      <none>
    State:          Waiting
      Reason:       ImagePullBackOff
    Ready:          False
    Restart Count:  0
    Limits:
      cpu:     100m
      memory:  200Mi
    Requests:
      cpu:        100m
      memory:     200Mi
    Environment:  <none>
    Mounts:
      # 表示Pod已挂载sa-example-token，即Pod可以使用这个Token来做认证
      /etc/secret-volume from secret-volume (ro)
      # 自动挂载的TokenRequest，可以提供短期的Token
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-2s4sw (ro)
...
```

进入Pod内部，还可以看到对应的文件，具体命令如下。其中，cd后的路径与secret-volume的挂载路径一致。

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

cd */etc/secret-volume*

ls

回显内容如下：

```
ca.crt     namespace  token
```

步骤4 验证手动创建的ServiceAccount Token能否生效。

1. 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可以访问集群内的Pod资源。ctrl+d退出Pod后，使用以下命令可以查看该服务的具体信息。

   kubectl get svc

   回显内容如下：

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

2. 进入Pod，并检查Pod是否能够在不使用Token的情况下通过API Server访问集群内的Pod资源。

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

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

回显结果如下，说明Pod不能直接通过API Server访问集群内的Pod资源。

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

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

3. 设置ca.crt的环境变量。将ca.crt的路径设置到CURL_CA_BUNDLE环境变量中，这将指示curl命令使用该证书文件作为信任锚点。

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

4. 将Token的内容放到TOKEN中。

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

   检验TOKEN是否设置成功。

   echo $TOKEN

   若回显内容如下，则设置成功。

   ```
   eyJhbGciOiJSUzI1NiIsImtpZCI6I...
   ```

5. 利用TOKEN访问API Server。

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

   回显结果如下，则说明Pod能够通过认证，即手动创建的ServiceAccount Token能够生效。若API Server返回的是cannot get path \"/api/v1/namespaces/default/pods\""，则说明没有权限访问，需要得到授权才能访问，授权机制请参见**9.2 RBAC**。

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

----结束

9.2 RBAC

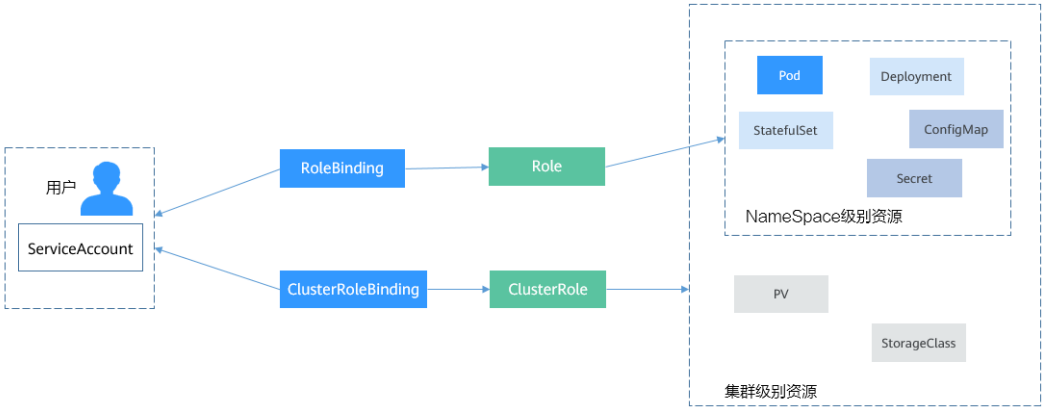
RBAC 资源

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 9-2 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
```

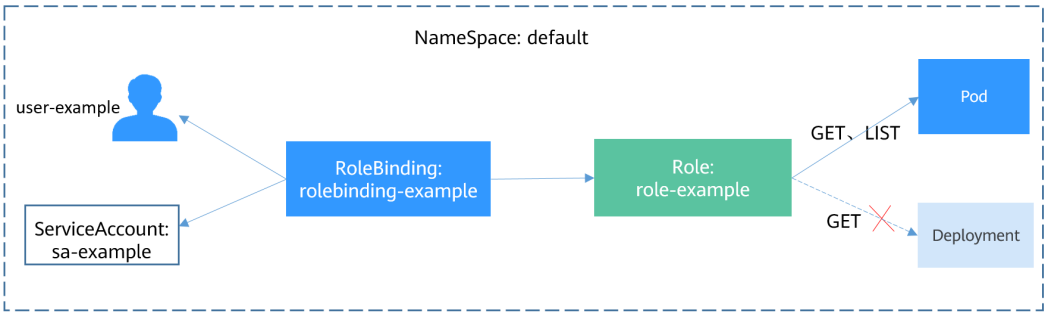
创建 RoleBinding

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**9.1 ServiceAccount**有过介绍。绑定后的关系如下图所示。

图 9-3 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。

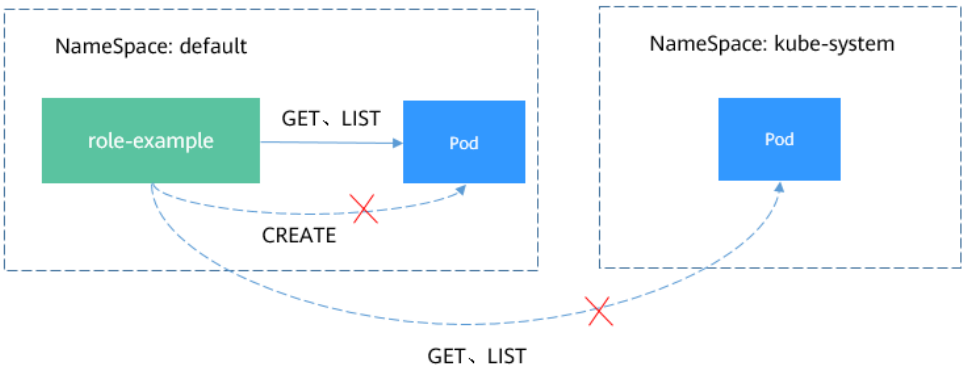
```
$ kubectl exec -it sa-pod -- /bin/sh
# export CURL_CA_BUNDLE=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods
{
  "kind": "PodList",
 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

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 9-4 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。

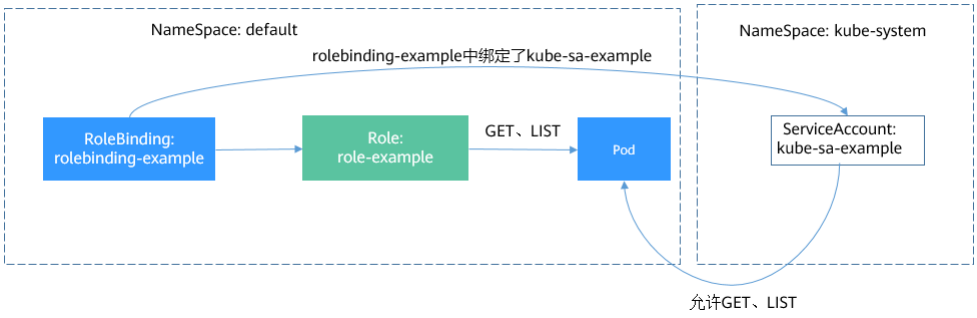
```
# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/kube-system/pods
...
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:sa-example\" cannot list resource \"pods\" in API group \"\" in the namespace \"kube-system\"",
  "reason": "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 9-5 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。

10 弹性伸缩

在**5 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。

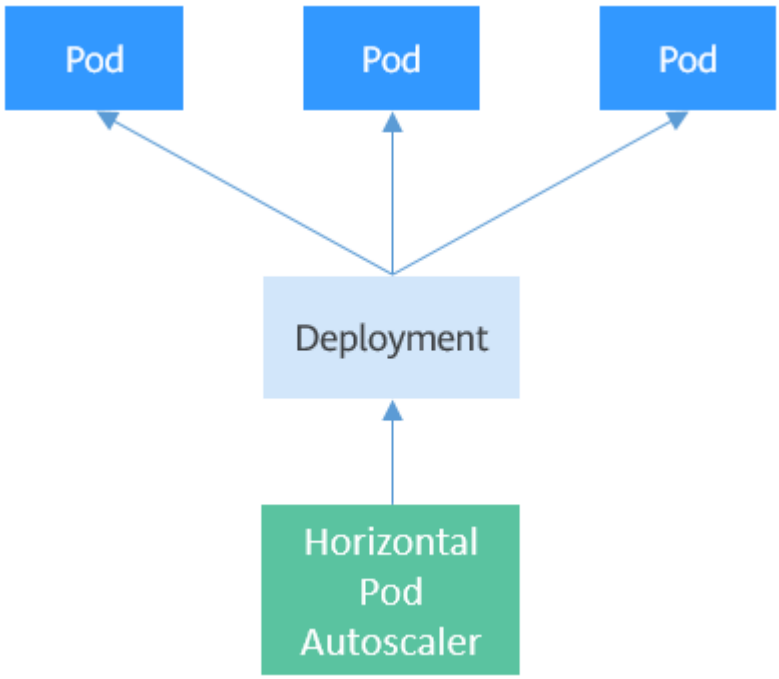
- **Prometheus**是一套开源的系统监控报警框架，能够采集丰富的Metrics（度量数据），目前已经基本是Kubernetes的标准监控方案。
- **Metrics Server**是Kubernetes集群范围资源使用数据的聚合器。Metrics Server从kubelet公开的Summary API中采集度量数据，能够收集包括了Pod、Node、容器、Service等主要Kubernetes核心资源的度量数据，且对外提供一套标准的API。

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 10-1 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/v2
kind: HorizontalPodAutoscaler
metadata:
  name: scale
  namespace: default
spec:
  scaleTargetRef:                 # 目标资源
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
```



```
minReplicas: 1                  # 目标资源的最小副本数量
maxReplicas: 10                  # 目标资源的最大副本数量
metrics:                       # 度量指标，期望CPU的利用率为70%
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 70
```

创建后HPA查看。

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

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

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。

CCE的集群节点弹性伸缩请参见**创建节点伸缩策略**。

