

第六章 容器技术和kubernetes基础

- 6.1 容器技术和Docker
- 6.2 Docker架构
- 6.3 Docker的使用
- 6.4 Ku bernetes概述
- 6.5 Kubernetes编排基础

容器技术概述

什么是容器技术？

容器技术是一种**轻量级的操作系统层虚拟化技术**，它能隔离进程和资源。

核心机制

- **Namespace**：重点功能是隔离资源，每个Namespace下的资源对于其他Namespace都是不可见的
- **Cgroup**：重点在限制资源的使用，提供对CPU、内存、磁盘资源的管理能力

容器技术已经成为云计算产业新的热点，是未来云计算的一个主要发展方向。

应用程序部署的三个时代

1. 物理机时代

- 单体架构
- 追求更快更强的物理机
- 资源隔离不足
- 资源利用率低

2. 虚拟化时代

- 模拟完整操作系统
- 实现资源隔离
- 消耗大量系统资源
- 环境依赖问题

3. 容器化时代

- 轻量级虚拟化
- 完全隔离的运行环境
- 基于预分配资源保证服务质量
- 分层文件系统和镜像仓库

容器技术解决了虚拟机部署应用存在的问题，为应用打造了完全隔离的运行环境。

容器相比传统虚拟机的优势

- **启动速度快、占用资源少**：基于进程而非虚拟机，无须模拟操作系统
- **良好的隔离性**：基于Linux Namespace技术隔离进程
- **资源限制**：基于Linux Control Group技术对用户资源进行限制
- **容器镜像**：支持Dockerfile，面向应用而不是操作系统
- **分层文件结构**：实现一次打包、到处运行
- **增量分发**：基于Digest的文件层，只拉取变更部分
- **跨平台迁移**：镜像可通过不同标签进行版本管理
- **自动化友好**：配置管理方便，让微服务成为可能
- **持续交付**：支持基于流水线的软件交付

Docker架构概述

Docker的重要性

Docker是使用最广、最为成熟的容器技术，为容器技术的快速发展做出了不可磨灭的贡献。

Docker应用的五个组成部分

- **Docker客户端**：接收用户参数转化为HTTP请求
- **Docker守护进程**：Docker引擎的核心组件
- **Docker镜像**：容器技术的核心，多层组成的独立对象
- **Docker容器**：镜像的运行实例
- **镜像仓库**：存储和分发Docker镜像

Docker的镜像格式和运行环境已成为事实上的工业标准。

Docker引擎架构

架构组件

- **Docker Client**: 提供用户接口
- **Docker Daemon**: 提供API和其他特性
- **Containerd**: 管理完整的容器生命周期
- **Shim**: 真实运行容器的载体
- **runC**: 根据OCI标准创建和运行容器

Containerd功能

- 管理容器的生命周期（从创建到销毁）
- 拉取/推送容器镜像
- 存储管理（管理镜像及容器数据的存储）
- 调用runC运行容器
- 管理容器网络接口及网络

Docker镜像系统

镜像特点

- 由多个层组成，每层叠加后形成独立对象
- 内部是精简的操作系统（不包含内核）
- 包含应用运行所需的文件和依赖包
- 通常比较小

写时复制机制（Copy on Write）

- 初始读写层为空
- 修改文件时从只读层复制到读写层
- 只读版本依然存在，被读写层隐藏
- 一个镜像可被多个容器使用，无需多份备份

镜像层可在多个镜像之间共享和复用，有效节省空间并提升性能。

Docker基本操作

镜像操作

```
docker image pull 镜像名称      # 下载镜像
docker image ls                  # 查看镜像列表
docker tag 旧镜像名 新镜像名     # 镜像重命名
docker image rm 镜像名          # 删除镜像
docker image save 镜像名 > file.tar # 导出镜像
docker image load -i file.tar    # 导入镜像
```

容器操作

```
docker run [OPTIONS] IMAGE [COMMAND] # 运行容器
docker ps                               # 查看运行中的容器
docker stop 容器ID                     # 停止容器
docker start 容器ID                    # 启动容器
docker rm 容器ID                       # 删除容器
```


Docker运行参数详解

参数	说明
--name	指定容器名字
-d	后台模式运行
-t -i	分配伪终端并关联标准流
-p	端口映射 hPort:cPort
-v	卷挂载 [src:]dest[:]
-e	设置环境变量
-m	限制内存使用量
--restart	定义退出重启策略

Kubernetes概述

什么是Kubernetes?

Kubernetes (K8s) 是一个**开源的容器编排平台**，用于自动化部署、扩展和管理容器化应用程序。

核心功能

- **容器编排**: 自动化容器的部署、扩展和管理
- **集群管理**: 监控和管理计算节点的健康状况及可用资源
- **PaaS功能**: 提供强大的平台即服务功能
- **轻量级设计**: 遵循微服务架构理念，组件模块化

Kubernetes已然成为云计算行业的实时标准，是容器云的标准化平台。

Kubernetes核心功能

自动化功能

- **自动化上线和回滚**
- **自愈**: 重启失败容器
- **自动装箱**: 根据资源需求自动部署
- **水平扩展**: 根据监控指标自动扩缩容

服务管理

- **服务发现与负载均衡**
- **密钥和配置管理**
- **批量执行**

存储与网络

- **存储编排**: 支持多种存储系统
- **IPv4/IPv6双协议栈**

扩展性

- **扩展设计**: 无须更改源码即可扩展
- **插件化**: 许多功能实现了插件化

Kubernetes提供了完整的容器化应用管理解决方案。

Kubernetes架构

Master节点（管理节点）

- **API Server**: 资源对象的唯一操作入口
- **Controller Manager**: 集群内部管理控制中心
- **Scheduler**: 负责集群资源调度
- **Etcd**: 高可用的分布式键值存储

Node节点（工作节点）

- **Kubelet**: 负责与API Server交互，管理Pod和容器
- **Kube-proxy**: 提供负载均衡，维护网络规则
- **Container Runtime**: 真正管理容器的组件

Master节点是集群的大脑，所有CRUD操作只能通过访问管理节点来操作。

Pod基础概念

什么是Pod?

Pod是Kubernetes里最小的调度单元，也是应用运行的载体。它是一个或多个容器的组合。

Pod特性

- **隔离体**：Pod是一个隔离体，内部包含的多个容器是共享的
- **共享资源**：包括PID、Network、Volume、IPC、UTS等
- **节点绑定**：Pod被分配到一个节点后，直到被删除前都不会离开这个Node
- **故障重建**：Pod失败后，Kubernetes会将其清理，然后重建新Pod

重要提示：重建后Pod的ID会发生变化，这是一个全新的Pod。

Pod生命周期

Pod的生命周期阶段

阶段	描述
Pending	Pod已被创建， 但一个或多个容器还未被创建， 包括调度阶段及镜像下载过程
Running	Pod已被调度到Node， 所有容器已经创建， 并且至少有一个容器在运行或正在重启
Succeeded	Pod中所有容器正常退出
Failed	Pod中所有容器退出， 至少有一个容器是异常退出的
Unknown	Pod状态没有获得， 这种情况发生在和Node通信失败的情况下

Pod创建过程

- **提交请求**: 用户通过kubectl、Rest API向API Server提交创建Pod请求
- **写入etcd**: API Server将Pod对象写入etcd中永久存储
- **反映变化**: API Server处能反映出Pod资源发生的变化
- **调度分配**: Scheduler监听到新Pod, 为其分配最优节点并更新spec.nodeName
- **更新状态**: Scheduler的更新被API Server写回到etcd中
- **kubelet响应**: kubelet监听到Pod被调度到自己节点, 调用CRI申请启动容器
- **创建PodSandbox**: 创建Infra容器, 设置网络命名空间
- **创建容器**: 检查镜像, 下载镜像, 创建并启动容器
- **更新状态**: kubelet将容器状态更新到Pod对象的Status中

Pod基本操作

kubectl命令格式

```
kubectl [command] [options]
```

查看Pod

查看当前命名空间的Pod

```
kubectl get pods
```

查看指定命名空间的Pod

```
kubectl get pod -n kube-system
```

查看所有命名空间的Pod

```
kubectl get pod -A
```

创建Pod

```
kubectl run PodName --image=ImageName:tag
```


Pod创建参数

常用参数

参数	说明
--image	指定pod使用的镜像
--labels	为pod指定标签，格式：--label=key=value
--env	指定容器里的环境变量，格式：--env="变量名"="值"
--port	指定容器里使用的端口
--image-pull-policy	镜像下载策略：Always、IfNotPresent、Never

示例

```
# 创建Pod并指定镜像下载策略
kubectl run pod2 --image=nginx --image-pull-policy=IfNotPresent
```

Pod YAML文件

生成YAML文件

```
# 生成Pod的YAML文件
kubectl run pod1 --image=nginx --dry-run=client -o yaml > pod1.yaml
```

YAML文件结构

一级参数

- . **apiVersion**: API版本 (v1)
- . **kind**: 资源类型 (Pod)
- . **metadata**: 元数据信息
- **spec**: 定义容器及策略

metadata参数

- **name**: Pod名字
- **namespace**: 命名空间
- . **labels**: 标签 (key:value)

使用**kubectl explain pods.spec**查看详细参数说明

Pod管理操作

创建和更新

```
# 创建Pod
kubectl create -f xxx.yaml
kubectl apply -f xxx.yaml
# 编辑运行中的Pod
kubectl edit pod podName
```

执行命令

```
kubectl exec pod1 -- ls /etc/nginx # 在Pod中执行命令
kubectl exec -ti pod1 -- bash # 进入Pod获取bash
```

文件操作

```
# 主机和容器间复制文件
kubectl cp /path1/file1 pod:/path2/
kubectl cp pod:/path2/ /path1/
```

标签Label

标签的作用

标签（Label）的键值对可以附加到系统中的任何API对象，用于标识和选择Kubernetes集群中的任意API对象。

节点标签操作

```
kubectl label node 节点名 key=value # 给节点设置标签
```

```
kubectl label node 节点名 key- # 删除节点标签
```

```
kubectl get nodes --show-labels
```

查看节点标签

nodeSelector使用

通过nodeSelector可以让Pod运行在含有特定标签的节点上：

```
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: k8s3
```

工作负载 (WorkLoad)

为什么需要工作负载?

单个Pod是**不稳定、不健壮**的，Pod被创建后，发生节点故障或其他原因挂掉后不会自动重启。

Kubernetes内置工作负载资源

- **Deployment和ReplicaSet**: 管理无状态应用
- **DaemonSet**: 在每个节点上运行一个Pod
- **StatefulSet**: 管理有状态应用
- **Job和CronJob**: 批处理任务和定时任务

工作负载通过**标签选择器**来管理一组具有相同功能的Pod。

Deployment

Deployment特性

- **副本管理**: 通过ReplicaSet管理Pod副本数量
- **滚动升级**: 支持应用的滚动更新
- **回滚功能**: 支持版本回滚
- **扩缩容**: 支持手动和自动扩缩容
- **故障转移**: 节点故障时自动重建Pod

创建Deployment

```
# 命令行创建
kubectl create deployment dp1 --image=nginx --replicas=3
# 查看Deployment
kubectl get deployments.apps
# 扩缩容
kubectl scale deployment dp1 --replicas=5
```

Deployment升级和回滚

镜像升级

```
# 升级镜像
kubectl set image deployment/dp1 nginx=nginx:1.22 --record
# 查看升级记录
kubectl rollout history deployment dp1
```

回滚操作

```
# 回滚到前一版本
kubectl rollout undo deployment dp1
# 回滚到指定版本
kubectl rollout undo deployment dp1 --to-revision=2
```

滚动更新参数

- **maxSurge**: 指定一次创建几个Pod (可以是百分比或具体数字)
- **maxUnavailable**: 指定最多删除几个Pod (可以是百分比或具体数字)

HPA水平自动扩缩容

HPA的作用

通过检查Pod的**CPU或内存负载情况**通知Deployment变更Pod数量，实现弹性扩展。

资源限制设置

```
resources:  
  requests:  
    cpu: 500m # 0.5个CPU核心
```

创建HPA

```
# 创建HPA  
kubectl autoscale deployment dp1 --min=1 --max=10 --cpu-percent=60  
  
# 查看HPA状态  
kubectl get hpa
```

设置最少运行1个Pod，最多10个Pod，CPU利用率不超过60%时自动扩展。

DaemonSet

DaemonSet特性

- **节点覆盖**：在所有节点上创建一个Pod
- **一对一关系**：有几个节点就创建几个Pod，每个节点只有一个
- **典型应用**：监控、日志收集等系统级服务

DaemonSet与Deployment的区别

特性	DaemonSet	Deployment
副本数	无需设置（自动等于节点数）	需要设置replicas
调度策略	每个节点一个Pod	根据调度算法分布
使用场景	系统级服务	应用级服务

StatefulSet

StatefulSet特性

- **有状态应用**：管理需要持续记录数据的应用
- **唯一标识**：每个Pod有不同的配置、数据、网络标识
- **有序部署**：Pod按顺序创建（0到N-1）和删除（N-1到0）
- **稳定存储**：与PersistentVolume相匹配

适用场景

- 稳定的、唯一的网络标识符
- 稳定的、持久的存储
- 有序的、优雅的部署和扩缩
- 有序的、自动的滚动更新

StatefulSet需要**无头服务（Headless Service）**来负责Pod的网络标识。

Job和CronJob

Job特性

- **批处理任务**：执行一次性任务
- **完成数**：指定需要完成的任务数
- **并行数**：指定同时执行的任务数

CronJob特性

- **定时任务**：通过Cron表达式定义执行周期
- **周期执行**：类似Linux的Crontab

示例

```
# 创建计算圆周率的Job
kubectl create job job2 --image=perl -- perl -Mbignum=bpi -wle 'print bpi(2000)'
# 查看Job状态
kubectl get job
# 查看Job输出
kubectl logs job2-8bq6n
```

Service基础

Service的作用

Service是Kubernetes最核心的概念，为**同一组具有相同功能的容器应用**提供统一的入口地址，并将请求进行负载均衡分发。

为什么需要Service?

- **Pod生命周期短**：Pod状态不稳定，IP会发生变化
- **固定访问点**：Service提供固定的IP和Port
- **自动关联**：通过标签选择器自动关联后端Pod
- **负载均衡**：通过Kube-proxy实现负载均衡

Service核心属性

- **Service Selector**：基于Pod标签的过滤规则
- **Ports**：定义服务端口、协议、目标端口等信息

Service操作

创建Service

为Deployment创建Service

```
kubectl expose deployment dp1 --name=svc1 --port=80 --target-port=80
```

为Pod创建Service

```
kubectl expose pod podname --name=svcname --port=端口 --target-port=端口
```

查看Service

查看Service列表

```
kubectl get svc -o wide
```

查看Service详情

```
kubectl describe svc svc1
```

Service参数说明

- . **--port**: Service的端口，请求转发给后端Pod
- . **--target-port**: 后端Pod里应用的服务端口
- . **--name**: 指定服务名（可选，默认与Deployment同名）

Service访问方式

集群内访问

- **ClusterIP**: 直接访问Service的ClusterIP
- **服务名**: 同一命名空间内直接使用服务名
- **FQDN**: 完全限定域名格式: `$svcname.$namespace.svc.$clusterdomain`

DNS解析

集群内使用服务名访问是通过**coreDNS**解析实现的, 所有Service都会自动注册到coreDNS。

负载均衡

Service会将请求以**负载均衡的方式**转发给后端的多个Pod, 通过Kube-proxy实现。

注意: ClusterIP不会响应ARP, ping请求会失败, 但HTTP请求正常。

Docker与Kubernetes的关系

发展历程

- 2013年Docker出现，彻底改变了容器技术格局
- Google推出Kubernetes，着力于集群管理和容器编排
- 早期Kubernetes将Docker作为默认容器运行时
- 2016年成立OCI（开放容器倡议），推进容器生态发展

容器运行时接口（CRI）

- K8s定义CRI用于与容器运行时交互
- Docker没有实现CRI，需要docker-shim桥接
- K8s v1.24起正式移除docker-shim
- 目前支持containerd、CRI-O、Docker Engine等

如要在K8s中继续使用Docker Engine，需要安装cri-dockerd进行对接。

容器技术的应用场景

适用场景

- **微服务架构**：容器天然适合微服务的部署和管理
- **DevOps流水线**：实现CI/CD自动化部署
- **云原生应用**：支持弹性扩缩容和高可用
- **混合云部署**：实现跨平台应用迁移
- **快速原型开发**：快速搭建开发测试环境

技术优势

- 轻量化特性适合批量快速上线的应用
- 快速规模弹性应用，如互联网Web应用
- 实现操作系统解耦，支持跨平台部署
- 简化应用部署的复杂性

本章总结

关键点

- 容器技术是**轻量级的操作系统层虚拟化技术**
 - Docker是目前最成熟和广泛使用的容器技术
 - Kubernetes是容器编排的事实标准
- 容器技术是**新一代云计算的核心技术**

技术发展趋势

- 从传统虚拟化向容器化技术转变
- 容器编排平台日趋成熟
- 云原生应用架构成为主流
- 容器安全和治理能力不断增强

容器技术是未来云计算的一个主要发展方向，将继续推动云计算产业的创新发展。

本章总结

核心概念

- **Pod**: Kubernetes最小调度单元, 应用运行载体
- **工作负载**: 管理Pod的控制器, 提供高可用和扩展能力
- **Service**: 为Pod提供稳定的网络访问入口
- **标签**: 用于标识和选择Kubernetes对象

工作负载类型

- **Deployment**: 无状态应用, 支持滚动更新和扩缩容
- **DaemonSet**: 每个节点运行一个Pod
- **StatefulSet**: 有状态应用, 提供稳定标识和存储
- **Job/CronJob**: 批处理和定时任务

这些基础概念是使用Kubernetes进行容器编排的核心, 掌握它们是深入学习Kubernetes的基础。