Kubernetes

官网：https://kubernetes.io/zh-cn/

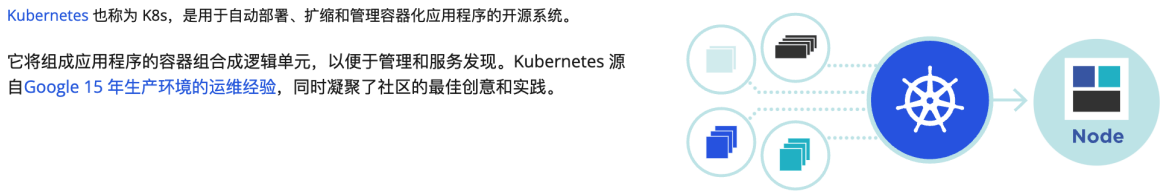
第一章 初识 Kubernetes

- Kubernetes 简介
- 为什么需要 Kubernetes
- Kubernetes 能做什么
- Kubernetes 不是什么？

1 简介

> 摘取官网：https://kubernetes.io/zh-cn/docs/concepts/overview/

Kubernetes 这个名字源于希腊语，意为 舵手 或 飞行员 。k8s 这个缩写是因为 k 和 s 之间有八个字符的关系。 Google 在 2014 年开源了 Kubernetes 项目。 Kubernetes 建立在 Google 大规模运行生产工作负载十几年经验的基础上， 结合了社区中最优秀的想法和实践。

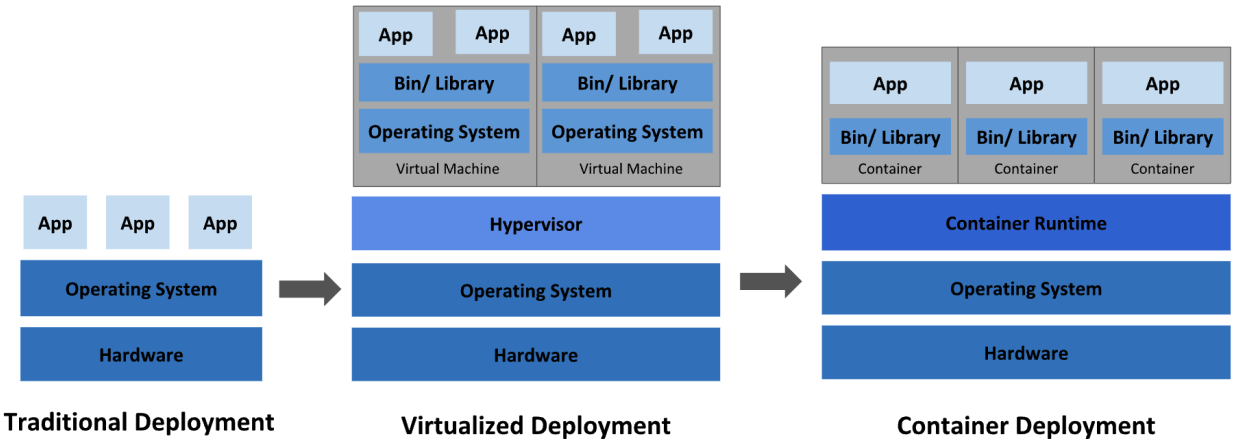


Kubernetes 是一个可移植、可扩展的开源平台，用于 管理容器化的工作负载和服务，可促进声明式配置和自动化 。 Kubernetes 拥有一个庞大且快速增长的生态，其服务、支持和工具的使用范围相当广泛。

从 2014 年第一个版本发布以来，Kuberetes 迅速获得开源社区的追捧，包括 Red Hat、VMware 在内的很多有影响力的公司加入到开发和推广的阵营。目前 Kubernetes 已经成为发展最快、市场占有率最高的容器编排引擎产品。

2 为什么需要 k8s

> 摘取官网：https://kubernetes.io/zh-cn/docs/concepts/overview/



传统部署时代：

早期，各个组织是在物理服务器上运行应用程序。 由于无法限制在物理服务器中运行的应用程序资源使用，因此会导致资源分配问题 。 例如，如果在同一台物理服务器上运行多个应用程序， 则可能会出现一个应用程序占用大部分资源的情况，而导致其他应用程序的性能下降。 一种解决方案是将每个应用程序都运行在不同的物理服务器上， 但是当某个应用程式资源利用率不高时，剩余资源无法被分配给其他应用程式， 而且维护许多物理服务器的成本很高 。

虚拟化部署时代：

因此，虚拟化技术被引入了。虚拟化技术允许你在单个物理服务器的 CPU 上运行多台虚拟机（VM）。 虚拟化能使应用程序在不同 VM 之间被彼此隔离，且能提供一定程度的安全性， 因为一个应用程序的信息不能被另一应用程序随意访问 。

虚拟化技术能够更好地利用物理服务器的资源，并且因为可轻松地添加或更新应用程序， 而因此可以具有更高的可扩缩性，以及降低硬件成本等等的好处。 通过虚拟化，你可以将一组物理资源呈现为可丢弃的虚拟机集群。

每个 VM 是一台完整的计算机，在虚拟化硬件之上运行所有组件，包括其自己的操作系统 。

容器部署时代：

容器类似于 VM，但是更宽松的隔离特性，使容器之间可以共享操作系统（OS）。 因此，容器比起 VM 被认为是更轻量级的。且与 VM 类似，每个容器都具有自己的文件系统、CPU、内存、进程空间等。 由于它们与基础架构分离，因此可以跨云和 OS 发行版本进行移植。 容器的出现解决了应用和基础环境异构的问题，让应用可以做到一次构建，多次部署 。不可否认容器是打包和运行应用程序的好方式，因此容器方式部署变得流行起来。但随着容器部署流行，仅仅是基于容器的部署仍有一些问题没有解决：

- 生产环境中， 你需要管理运行着应用程序的容器，并确保服务不会下线。 例如，如果一个容器发生故障，则你需要启动另一个容器。
- 高并发时，你需要启动多个应用程序容器为系统提高高可用，并保证多个容器能负载均衡。
- 在维护、升级版本时，你需要将运行应用程序容器从新部署，部署时必须对之前应用容器备份，一旦出现错误，需要手动启动之前容器保证系统运行。

如果以上行为交由给系统处理，是不是会更容易一些？那么谁能做到这些？

3 k8s 能做什么？

> 摘取官网：https://kubernetes.io/zh-cn/docs/concepts/overview/

这就是 Kubernetes 要来做的事情！ Kubernetes 为你提供了一个可弹性运行分布式系统的框架 。 Kubernetes 会满足你的扩展要求、故障转移你的应用、提供部署模式等。Kubernetes 为你提供：

- **服务发现和负载均衡**

  Kubernetes 可以使用 DNS 名称或自己的 IP 地址来暴露容器。 如果进入容器的流量很大， Kubernetes 可以负载均衡并分配网络流量，从而使部署稳定。

- **存储编排**

  Kubernetes 允许你自动挂载你选择的存储系统，例如本地存储、公共云提供商等。

- **自动部署和回滚**

  你可以使用 Kubernetes 描述已部署容器的所需状态， 它可以以受控的速率将实际状态更改为期望状态。 例如，你可以自动化 Kubernetes 来为你的部署创建新容器， 删除现有容器并将它们的所有资源用于新容器。

- **自动完成装箱计算/自动资源调度**

  你为 Kubernetes 提供许多节点组成的集群，在这个集群上运行容器化的任务。 你告诉 Kubernetes 每个容器需要多少 CPU 和内存 (RAM)。 Kubernetes 可以将这些容器按实际情况调度到你的节点上，以最佳方式利用你的资源。

- **自我修复/自愈能力**

  Kubernetes 将重新启动失败的容器、替换容器、杀死不响应用户定义的运行状况检查的容器， 并且在准备好服务之前不将其通告给客户端。

- **密钥与配置管理**

  Kubernetes 允许你存储和管理敏感信息，例如密码、OAuth 令牌和 ssh 密钥。 你可以在不重建容器镜像的情况下部署和更新密钥和应用程序配置，也无需在堆栈配置中暴露密钥。

4 k8s 不是什么？

Kubernetes 不是传统的、包罗万象的 PaaS（平台即服务）系统 。 由于 Kubernetes 是在容器级别运行，而非在硬件级别，它提供了 PaaS 产品共有的一些普遍适用的功能， 例如部署、扩展、负载均衡，允许用户集成他们的日志记录、监控和警报方案。 但是， Kubernetes 不是单体式（monolithic）系统 ，那些默认解决方案都是可选、可插拔的。 Kubernetes 为构建开发人员平台提供了基础，但是在重要的地方保留了用户选择权，能有更高的灵活性。

Kubernetes：

- 不限制支持的应用程序类型。 Kubernetes 旨在支持极其多种多样的工作负载，包括无状态、有状态和数据处理工作负载。 如果应用程序可以在容器中运行，那么它应该可以在 Kubernetes 上很好地运行。
- 不部署源代码，也不构建你的应用程序。 持续集成（CI）、交付和部署（CI/CD）工作流取决于组

织的文化和偏好以及技术要求。

- 不提供应用程序级别的服务作为内置服务，例如中间件（例如消息中间件）、 数据处理框架（例如 Spark）、数据库（例如 MySQL）、缓存、集群存储系统 （例如 Ceph）。这样的组件可以在 Kubernetes 上运行，并且/或者可以由运行在 Kubernetes 上的应用程序通过可移植机制（例如开放服务代理）来访问。
- 不是日志记录、监视或警报的解决方案。 它集成了一些功能作为概念证明，并提供了收集和导出指标的机制。
- 不提供也不要求配置用的语言、系统（例如 jsonnet），它提供了声明性 API， 该声明性 API 可以由任意形式的声明性规范所构成。
- 不提供也不采用任何全面的机器配置、维护、管理或自我修复系统。
- 此外，Kubernetes 不仅仅是一个编排系统，实际上它消除了编排的需要。 编排的技术定义是执行已定义的工作流程：首先执行 A，然后执行 B，再执行 C。 而 Kubernetes 包含了一组独立可组合的控制过程，可以连续地将当前状态驱动到所提供的预期状态。 你不需要在乎如何从 A 移动到 C，也不需要集中控制，这使得系统更易于使用且功能更强大、 系统更健壮，更为弹性和可扩展。

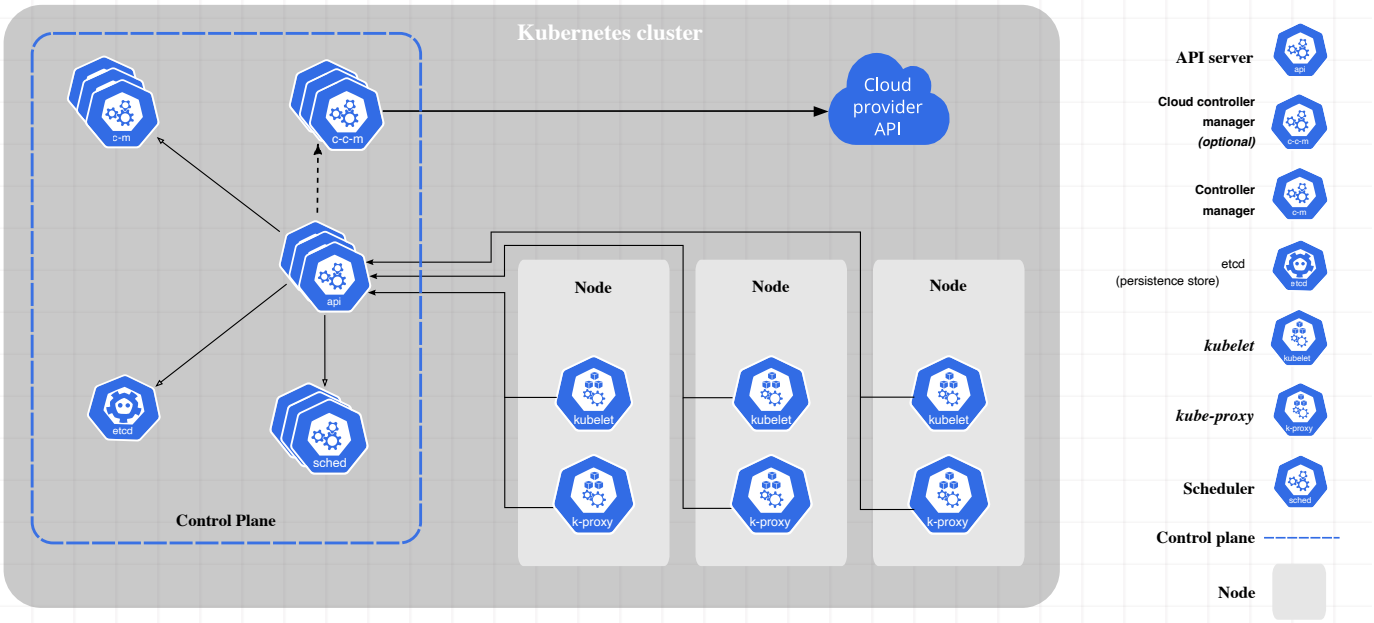
第二章 组件&架构

- 集群组件
- 核心概念
- 集群安装

1 集群组件

- 集群 cluster ： 将同一个软件服务多个节点组织到一起共同为系统提供服务过程称之为该软件的集群。redis 集群、es集群、mongo 等。
- k8s 集群：多个节点：3 个节点 角色： 1.master 节点/control plane 控制节点 2. work node：工作节点(pod 容器:应用程序容器)

当部署完 Kubernetes，便拥有了一个完整的集群。一组工作机器，称为节点， 会运行容器化应用程序。 每个集群至少有一个工作节点 。工作节点会 托管 Pod ，而 Pod 就是作为应用负载的组件。 控制平面管理集群中的工作节点和 Pod 。



1.1 控制平面组件（Control Plane Components）

控制平面组件会为集群做出全局决策，比如资源的调度。 以及检测和响应集群事件，例如当不满足部署的 replicas 字段时， 要启动新的 pod ）。

> 控制平面组件可以在集群中的任何节点上运行。 然而，为了简单起见，设置脚本通常会在同一个计算机上启动所有控制平面组件， 并且不会在此计算机上运行用户容器。

- **kube-apiserver**

  API server是 Kubernetes 控制平面的组件， 该组件负责公开了 Kubernetes API，负责处理接受请求的工作 。 API server 是 Kubernetes 控制平面的前端。Kubernetes API 服务器的主要实现是 kube-apiserver。 kube-apiserver 设计上考虑了水平扩缩，也就是说，它可通过部署多个实例来进行扩缩。 你可以运行 kube-apiserver 的多个实例，并在这些实例之间平衡流量。

- **etcd**

  一致且高度可用的键值存储，用作 Kubernetes 的所有集群数据的后台数据库 。

- **kube-scheduler**

  kube-scheduler 是控制平面的组件， 负责监视新创建的、未指定运行节点 node 的 Pods，并选择节点来让 Pod 在上面运行。调度决策考虑的因素包括单个 Pod 及 Pods 集合的资源需求、软硬件及策略约束、 亲和性及反亲和性规范、数据位置、工作负载间的干扰及最后时限。

- **kube-controller-manager**

  kube-controller-manager 是控制平面的组件， 负责运行控制器进程。从逻辑上讲， 每个控制器都是一个单独的进程， 但是为了降低复杂性，它们都被编译到同一个可执行文件，并在同一个进程中运行。

  这些控制器包括：

- 节点控制器（Node Controller）：负责在节点出现故障时进行通知和响应
- 任务控制器（Job Controller）：监测代表一次性任务的 Job 对象，然后创建 Pods 来运行这些任务直至完成
- 端点分片控制器（EndpointSlice controller）：填充端点分片（EndpointSlice）对象（以提供 Service 和 Pod 之间的链接）。
- 服务账号控制器（ServiceAccount controller）：为新的命名空间创建默认的服务账号（ServiceAccount）。

- **cloud-controller-manager**（optional 可选）

  一个 Kubernetes 控制平面组件， 嵌入了特定于云平台的控制逻辑。 云控制器管理器（Cloud Controller Manager）允许你将你的集群连接到云提供商的 API 之上， 并将与该云平台交互的组件同与你的集群交互的组件分离开来。 `cloud-controller-manager` 仅运行特定于云平台的控制器。 因此如果你在自己的环境中运行 Kubernetes，或者在本地计算机中运行学习环境， 所部署的集群不需要有云控制器管理器。与 `kube-controller-manager` 类似， `cloud-controller-manager` 将若干逻辑上独立的控制回路组合到同一个可执行文件中， 供你以同一进程的方式运行。 你可以对其执行水平扩容（运行不止一个副本）以提升性能或者增强容错能力。

  下面的控制器都包含对云平台驱动的依赖：

  - 节点控制器（Node Controller）：用于在节点终止响应后检查云提供商以确定节点是否已被删除
  - 路由控制器（Route Controller）：用于在底层云基础架构中设置路由
  - 服务控制器（Service Controller）：用于创建、更新和删除云提供商负载均衡器

1.2 Node 组件

> 节点组件会在每个节点上运行，负责维护运行的 Pod 并提供 Kubernetes 运行环境。

- **kubelet**

  kubelet 会在集群中每个节点（node）上运行。 它保证容器（containers）都运行在 Pods 中。

  kubelet 接收一组通过各类机制提供给它的 PodSpecs， 确保这些 PodSpecs 中描述的容器处于运行状态且健康。 kubelet 不会管理不是由 Kubernetes 创建的容器。

- **kube-proxy**

  kube-proxy是集群中每个节点（node）上所运行的网络代理， 实现 Kubernetes 服务（Service）概念的一部分。

  kube-proxy 维护节点上的一些网络规则， 这些网络规则会允许从集群内部或外部的网络会话与 Pod 进行网络通信。

  如果操作系统提供了可用的数据包过滤层，则 kube-proxy 会通过它来实现网络规则。 否则，kube-proxy 仅做流量转发。

- **容器运行时（Container Runtime）**

容器运行环境是负责运行容器的软件。

Kubernetes 支持许多容器运行环境，例如 containerd、 CRI-0、Docker 以及 Kubernetes CRI 的其他任何实现。

1.3 插件（Addons）

- **DNS**

  尽管其他插件都并非严格意义上的必需组件，但几乎所有 Kubernetes 集群都应该有集群 DNS因为很多示例都需要 DNS 服务。

- **Web 界面（仪表盘）**

  Dashboard 是 Kubernetes 集群的通用的、基于 Web 的用户界面。 它使用户可以管理集群中运行的应用程序以及集群本身， 并进行故障排除。

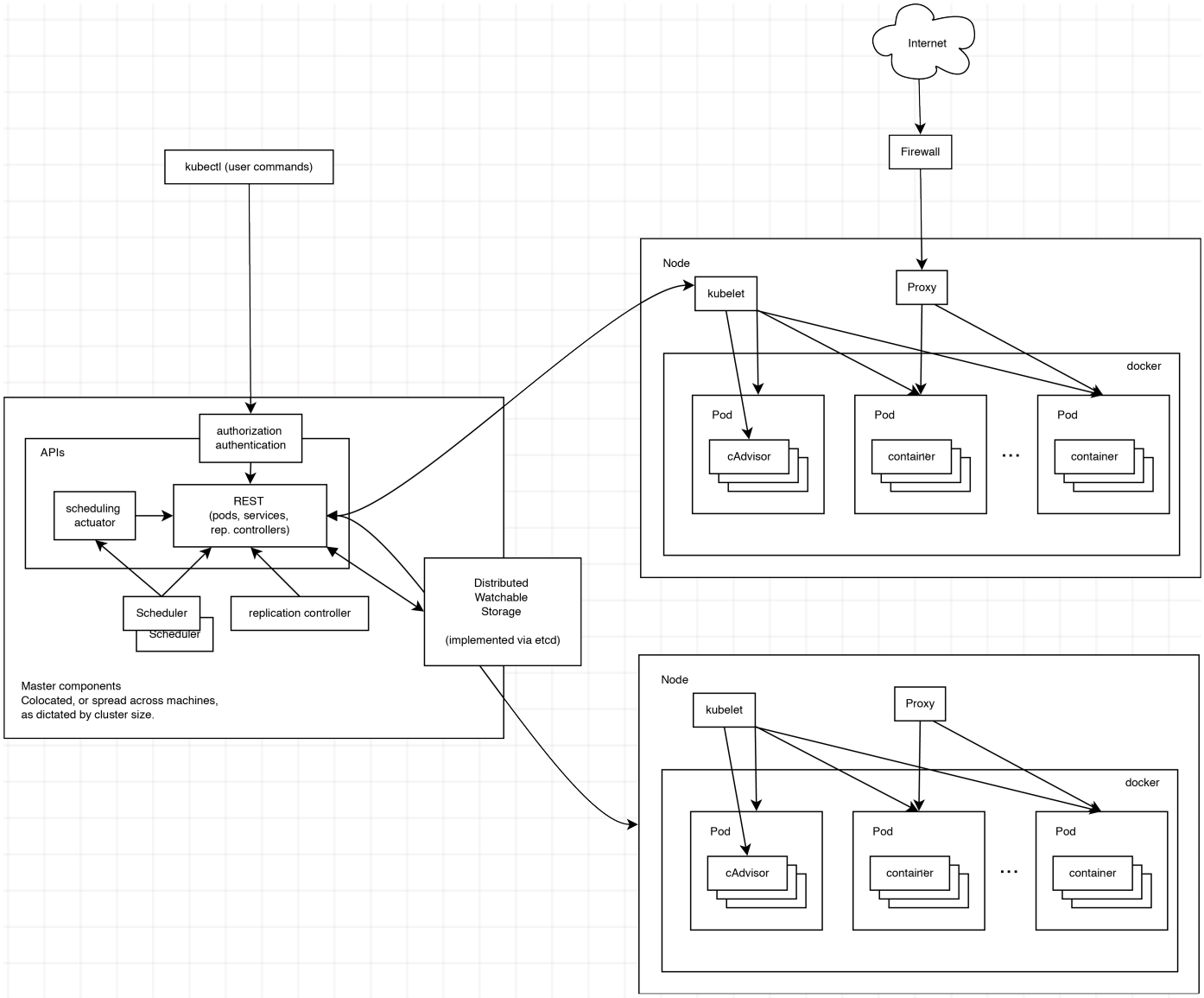
- **容器资源监控**

  容器资源监控将关于容器的一些常见的时间序列度量值保存到一个集中的数据库中， 并提供浏览这些数据的界面。

- **集群层面日志**

  集群层面日志机制负责将容器的日志数据保存到一个集中的日志存储中， 这种集中日志存储提供搜索和浏览接口。

2 集群架构详细



- **总结**
  - Kubernetes 集群由多个节点组成，节点分为两类：一类是属于管理平面的主节点/控制节点（Master Node）；一类是属于运行平面的工作节点（Worker Node）。显然，复杂的工作肯定都交给控制节点去做了，工作节点负责提供稳定的操作接口和能力抽象即可。

3 集群搭建[重点]

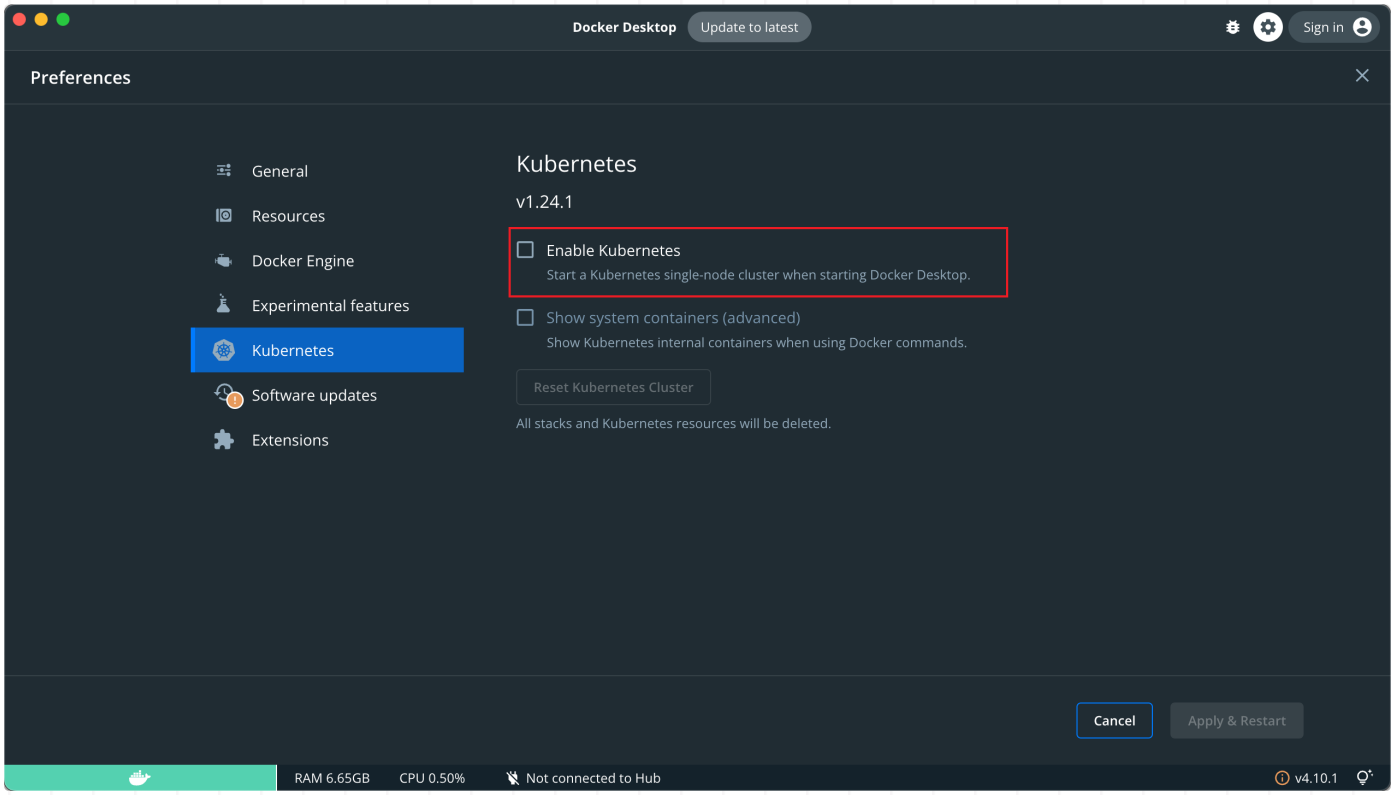
- **minikube**
  只是一个 K8S 集群模拟器，只有一个节点的集群，只为测试用，master 和 worker 都在一起。
- **裸机安装**
  至少需要两台机器（主节点、工作节点个一台），需要自己安装 Kubernetes 组件，配置会稍微麻烦点。
  缺点：配置麻烦，缺少生态支持，例如负载均衡器、云存储。
- **直接用云平台 Kubernetes**
  可视化搭建，只需简单几步就可以创建好一个集群。
  优点：安装简单，生态齐全，负载均衡器、存储等都给你配套好，简单操作就搞定

- **k3s**

  安装简单，脚本自动完成。

  优点：轻量级，配置要求低，安装简单，生态齐全。

3.1 minikube



3.2 裸机安装

0 环境准备

- 节点数量：3 台虚拟机 centos7
- 硬件配置：2G或更多的RAM，2个CPU或更多的CPU，硬盘至少30G 以上
- 网络要求：多个节点之间网络互通，每个节点能访问外网

1 集群规划

- k8s-node1：10.15.0.5
- k8s-node2：10.15.0.6
- k8s-node3：10.15.0.7

2 设置主机名

```
$ hostnamectl set-hostname k8s-node1
$ hostnamectl set-hostname k8s-node2
$ hostnamectl set-hostname k8s-node3
```

3 同步 hosts 文件

> 如果 DNS 不支持主机名称解析，还需要在每台机器的 `/etc/hosts` 文件中添加主机名和 IP 的对应关系：

```
cat >> /etc/hosts <<EOF
10.15.0.5 k8s-node1
10.15.0.6 k8s-node2
10.15.0.7 k8s-node3
EOF
```

4 关闭防火墙

```
systemctl stop firewalld && systemctl disable firewalld
```

5 关闭 SELINUX

> 注意：ARM 架构请勿执行,执行会出现 ip 无法获取问题！

```
setenforce 0 && sed -i 's/SELINUX=enforcing/SELINUX=disabled/g'
/etc/selinux/config
```

6 关闭 swap 分区

```
swapoff -a && sed -ri 's/.*swap.*/#&/' /etc/fstab
```

7 同步时间

```
$ yum install ntpdate -y
$ ntpdate time.windows.com
```

8 安装 containerd

```
# 安装 yum-config-manager 相关依赖
$ yum install -y yum-utils device-mapper-persistent-data lvm2
# 添加 containerd yum 源
$ yum-config-manager --add-repo http://mirrors.aliyun.com/docker-
ce/linux/centos/docker-ce.repo
# 安装 containerd
$ yum install  -y containerd.io cri-tools
# 配置 containerd
$ cat >  /etc/containerd/config.toml <<EOF
disabled_plugins = ["restart"]
[plugins.linux]
shim_debug = true
[plugins.cri.registry.mirrors."docker.io"]
endpoint = ["https://frz7i079.mirror.aliyuncs.com"]
[plugins.cri]
sandbox_image = "registry.aliyuncs.com/google_containers/pause:3.2"
EOF
# 启动 containerd 服务 并 开机配置自启动
$ systemctl enable containerd && systemctl start containerd && systemctl
status containerd

# 配置 containerd 配置
$ cat > /etc/modules-load.d/containerd.conf <<EOF
overlay
br_netfilter
EOF

# 配置 k8s 网络配置
$ cat  > /etc/sysctl.d/k8s.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF

# 加载 overlay br_netfilter 模块
$ modprobe overlay
$ modprobe br_netfilter

# 查看当前配置是否生效
$ sysctl -p /etc/sysctl.d/k8s.conf
```

9 添加源

- 查看源

```
$ yum repolist
```

- 添加源 x86

```
$ cat <<EOF > kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-
x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF
$ mv kubernetes.repo /etc/yum.repos.d/
```

- 添加源 ARM

```
$ cat << EOF > kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-
aarch64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF

$ mv kubernetes.repo /etc/yum.repos.d/
```

11 安装 k8s

```
# 安装最新版本
$ yum install -y kubelet kubeadm kubectl

# 指定版本安装
# yum install -y kubelet-1.26.0 kubectl-1.26.0 kubeadm-1.26.0

# 启动 kubelet
$ sudo systemctl enable kubelet && sudo systemctl start kubelet && sudo
systemctl status kubelet
```

12 初始化集群

- **注意：初始化 k8s 集群仅仅需要再在 master 节点进行集群初始化！**

```
kubeadm init \
--apiserver-advertise-address=10.15.0.5 \
--pod-network-cidr=10.244.0.0/16 \
--image-repository registry.aliyuncs.com/google_containers
```

13 配置集群网络

> 创建配置：kube-flannel.yml ,执行 kubectl apply -f kube-flannel.yml

- **注意：只在主节点执行即可！**

```
---
kind: Namespace
apiVersion: v1
metadata:
  name: kube-flannel
  labels:
    pod-security.kubernetes.io/enforce: privileged
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: flannel
rules:
- apiGroups:
  - ""
  resources:
  - pods
```

```
  verbs:
  - get
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - nodes/status
  verbs:
  - patch
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: flannel
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: flannel
subjects:
- kind: ServiceAccount
  name: flannel
  namespace: kube-flannel
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: flannel
  namespace: kube-flannel
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: kube-flannel-cfg
  namespace: kube-flannel
  labels:
    tier: node
```

```
    app: flannel
data:
  cni-conf.json: |
    {
      "name": "cbr0",
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "flannel",
          "delegate": {
            "hairpinMode": true,
            "isDefaultGateway": true
          }
        },
        {
          "type": "portmap",
          "capabilities": {
            "portMappings": true
          }
        }
      ]
    }
  net-conf.json: |
    {
      "Network": "10.244.0.0/16",
      "Backend": {
        "Type": "vxlan"
      }
    }
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-flannel-ds
  namespace: kube-flannel
  labels:
    tier: node
    app: flannel
spec:
  selector:
    matchLabels:
      app: flannel
  template:
```

```
    metadata:
      labels:
        tier: node
        app: flannel
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/os
                operator: In
                values:
                - linux
      hostNetwork: true
      priorityClassName: system-node-critical
      tolerations:
      - operator: Exists
        effect: NoSchedule
      serviceAccountName: flannel
      initContainers:
      - name: install-cni-plugin
       #image: flannelcni/flannel-cni-plugin:v1.1.0 for ppc64le and mips64le
(dockerhub limitations may apply)
        image: docker.io/rancher/mirrored-flannelcni-flannel-cni-
plugin:v1.1.0
        command:
        - cp
        args:
        - -f
        - /flannel
        - /opt/cni/bin/flannel
        volumeMounts:
        - name: cni-plugin
          mountPath: /opt/cni/bin
      - name: install-cni
       #image: flannelcni/flannel:v0.20.2 for ppc64le and mips64le
(dockerhub limitations may apply)
        image: docker.io/rancher/mirrored-flannelcni-flannel:v0.20.2
        command:
        - cp
        args:
        - -f
```

```
        - /etc/kube-flannel/cni-conf.json
        - /etc/cni/net.d/10-flannel.conflist
        volumeMounts:
        - name: cni
          mountPath: /etc/cni/net.d
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      containers:
      - name: kube-flannel
       #image: flannelcni/flannel:v0.20.2 for ppc64le and mips64le
(dockerhub limitations may apply)
        image: docker.io/rancher/mirrored-flannelcni-flannel:v0.20.2
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
        - --kube-subnet-mgr
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
          limits:
            cpu: "100m"
            memory: "50Mi"
        securityContext:
          privileged: false
          capabilities:
            add: ["NET_ADMIN", "NET_RAW"]
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: EVENT_QUEUE_DEPTH
          value: "5000"
        volumeMounts:
        - name: run
          mountPath: /run/flannel
        - name: flannel-cfg
```

```
          mountPath: /etc/kube-flannel/
        - name: xtables-lock
          mountPath: /run/xtables.lock
      volumes:
      - name: run
        hostPath:
          path: /run/flannel
      - name: cni-plugin
        hostPath:
          path: /opt/cni/bin
      - name: cni
        hostPath:
          path: /etc/cni/net.d
      - name: flannel-cfg
        configMap:
          name: kube-flannel-cfg
      - name: xtables-lock
        hostPath:
          path: /run/xtables.lock
          type: FileOrCreate
```

14 查看集群状态

```
# 查看集群节点状态 全部为 Ready 代表集群搭建成功
$ kubectl get nodes
NAME        STATUS   ROLES           AGE   VERSION
k8s-node1   Ready    control-plane   21h   v1.26.0
k8s-node2   Ready    <none>          21h   v1.26.0
k8s-node3   Ready    <none>          21h   v1.26.0

# 查看集群系统 pod 运行情况,下面所有 pod 状态为 Running 代表集群可用
$ kubectl get pod -A
NAMESPACE      NAME                                READY   STATUS
RESTARTS   AGE
default        nginx                               1/1     Running   0
   21h
kube-flannel   kube-flannel-ds-gtq49               1/1     Running   0
   21h
kube-flannel   kube-flannel-ds-qpdl6               1/1     Running   0
   21h
kube-flannel   kube-flannel-ds-ttxjb               1/1     Running   0
   21h
```

```
kube-system    coredns-5bbd96d687-p7q2x            1/1     Running   0
   21h
kube-system    coredns-5bbd96d687-rzcnz            1/1     Running   0
   21h
kube-system    etcd-k8s-node1                      1/1     Running   0
   21h
kube-system    kube-apiserver-k8s-node1            1/1     Running   0
   21h
kube-system    kube-controller-manager-k8s-node1   1/1     Running   0
   21h
kube-system    kube-proxy-mtsbp                    1/1     Running   0
   21h
kube-system    kube-proxy-v2jfs                    1/1     Running   0
   21h
kube-system    kube-proxy-x6vhn                    1/1     Running   0
   21h
kube-system    kube-scheduler-k8s-node1            1/1     Running   0
   21h
```

第三章 Pod & Container

- 什么是 Pod
- Pod 基本操作
- Pod 的 Labels
- Pod 的 生命周期
- Container 特性
- Pod 的资源限制
- Pod 中 Init 容器
- 节点亲和性分配 Pod

1 什么是 Pod

> 摘取官网：https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/#working-with-pods

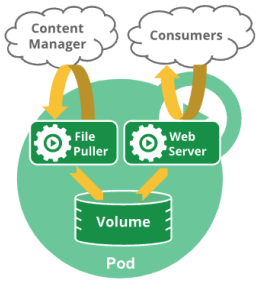
1.1 简介

Pod 是可以在 Kubernetes 中**创建和管理的、最小的可部署的计算单元。Pod**（就像在鲸鱼荚或者豌豆荚中）**是一组（一个或多个）容器**； 这些容器共享存储、网络、以及怎样运行这些容器的声明。 Pod 中的**内容总是并置（colocated）的并且一同调度，在共享的上下文中运行**。简言之如果用 Docker 的术语来描述，**Pod 类似于共享名字空间并共享文件系统卷的一组容器。**

定义：Pod 就是用来管理一组(一个|多个)容器的集合 特点：共享网络 共享存储 共享上下文环境

1.2 Pod 怎样管理多个容器?

Pod 中的容器被自动安排到集群中的同一物理机或虚拟机上，并可以一起进行调度。 容器之间可以共享资源和依赖、彼此通信、协调何时以及何种方式终止自身。例如，你可能有一个容器，为共享卷中的文件提供 Web 服务器支持，以及一个单独的 "边车（sidercar)" 容器负责从远端更新这些文件，如下图所示：



1.3 如何使用 Pod?

通常你不需要直接创建 Pod，甚至单实例 Pod。 相反，你会使用诸如 Deployment 或 Job 这类工作负载资源来创建 Pod。 如果 Pod 需要跟踪状态，可以考虑 StatefulSet 资源。

Kubernetes 集群中的 Pod 主要有两种用法：

- **运行单个容器的 Pod**。"每个 Pod 一个容器" 模型是最常见的 Kubernetes 用例； 在这种情况下，可以将 Pod 看作单个容器的包装器，并且 Kubernetes 直接管理 Pod，而不是容器。
- **运行多个协同工作的容器 的 Pod**。 Pod 可能封装由多个紧密耦合且需要共享资源的共处容器组成的应用程序。 这些位于同一位置的容器可能形成单个内聚的服务单元 —— 一个容器将文件从共享卷提供给公众， 而另一个单独的 “边车”（sidecar）容器则刷新或更新这些文件。 Pod 将这些容器和存储资源打包为一个可管理的实体。

说明：

- 将多个并置、同管的容器组织到一个 Pod 中是一种相对高级的使用场景。 只有在一些场景中，容器之间紧密关联时你才应该使用这种模式。
- 每个 Pod 都旨在运行给定应用程序的单个实例。如果希望横向扩展应用程序 （例如，运行多个实例以提供更多的资源），则应该使用多个 Pod，每个实例使用一个 Pod。 在 Kubernetes 中，这通常被称为**副本（Replication）**。 通常使用一种工作负载资源及其控制器来创建和管理一组 Pod 副本。

2 Pod 基本操作

2.1 查看 pod

```
# 查看默认命名空间的 pod
$ kubectl get pods|pod|po

# 查看所有命名空间的 pod
$ kubectl get pods|pod -A
$ kubectl get pods|pod|po -n 命名空间名称

# 查看默认命名空间下 pod 的详细信息
$ kubectl get pods -o wide

# 查看所有命名空间下 pod 的详细信息
$ kubectl get pods -o wide -A

# 实时监控 pod 的状态
$ kubectl get pod -w
```

2.2 创建 pod

> 官网参考地址：https://kubernetes.io/zh-cn/docs/reference/kubernetes-api/workload-resources/pod-v1/

```
# nginx-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.19
      ports:
        - containerPort: 80
```

```
# 使用 kubectl apply/create -f 创建 pod
$ kubectl create -f nginx-pod.yml
$ kubectl apply -f nginx-pod.yml
```

> 注意：create 仅仅是不存在时创建,如果已经存在则报错！apply 不存在创建，存在更新配置。推荐使用 apply!

2.3 删除 pod

```
$ kubectl delete -f pod.yml
```

2.4 进入 pod 中容器

```
$ kubectl exec -it nginx(pod名称) --(固定写死) bash(执行命令)
```

2.5 查看 pod 日志

```
$ kubectl logs -f(可选,实时) nginx(pod 名称)
```

2.6 查看 pod 描述信息

```
$ kubectl describe pod nginx(pod名称)
```

3 Pod 运行多个容器

3.1 创建 pod

```yaml
# myapp-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
    - name: nginx
      image: nginx:1.19
      ports:
        - containerPort: 80
      imagePullPolicy: IfNotPresent

    - name: redis
      image: redis:5.0.10
      ports:
```

```
    - containerPort: 6379
  imagePullPolicy: IfNotPresent
```

```
$ kubectl apply -f myapp-pod.yml
```

3.2 查看指定容器日志

```
# 查看日志（默认只查看第一个容器日志，这里是展示 nginx 日志）
$ kubectl logs -f myapp

# 查看 pod 中指定容器的日志
$ kubectl logs -f myapp -c nginx(容器名称)
$ kubectl logs -f myapp -c redis(容器名称)
```

3.3 进入容器

```
# 进入 pod 的容器（默认进入第一个容器内部，这里会进入 nginx 容器内部）
$ kubectl exec -it myapp -- sh

# 进入 pod 中指定容器内部
$ kubectl exec -it myapp -c nginx -- sh
$ kubectl exec -it myapp -c redis -- sh
```

4 Pod 的 Labels(标签)

标签（Labels） 是附加到 Kubernetes 对象（比如 Pod）上的键值对。 标签旨在用于指定对用户有意义且相关的对象的标识属性。标签可以在创建时附加到对象，随后可以随时添加和修改。每个对象都可以定义一组键(key)/值(value)标签，但是每个键(key)对于给定对象必须是唯一的。

4.1 语法

标签由键值对组成 ，其有效标签值：

- 必须为 63 个字符或更少（可以为空）
- 除非标签值为空，必须以字母数字字符（ `[a-z0-9A-Z]` ）开头和结尾
- 包含破折号（ `-` ）、下划线（ `_` ）、点（ `.` ）和字母或数字

4.2 示例

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    name: myapp #创建时添加
spec:
  containers:
    - name: nginx
      image: nginx:1.21
      imagePullPolicy: IfNotPresent

    - name: redis
      image: redis:5.0.10
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

4.3 标签基本操作

```shell
# 查看标签
$ kubectl get pods --show-labels

# kubectl label pod pod名称 标签键值对
$ kubectl label pod myapp env=prod

# 覆盖标签 --overwrite
$ kubectl label --overwrite pod myapp env=test

# 删除标签 -号代表删除标签
$ kubectl label pod myapp env-

# 根据标签筛选 env=test/env
$ kubectl get po -l env=test
$ kubectl get po -l env
$ kubectl get po -l '!env' # 不包含的 pod
$ kubectl get po -l 'env in (test,prod)' #选择含有指定值的 pod
$ kubectl get po -l 'env notin (test,prod)' #选择含有指定值的 pod
```

5 Pod 的生命周期

> 摘自官网：https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/pod-lifecycle/

Pod 遵循预定义的生命周期，起始于 `Pending` 阶段， 如果至少其中有一个主要容器正常启动，则进入 `Running`，之后取决于 Pod 中是否有容器以失败状态结束而进入 `Succeeded` 或者 `Failed` 阶段。与此同时Pod 在其生命周期中只会被调度一次。 一旦 Pod 被调度（分派）到某个节点，Pod 会一直在该节点运行，直到 Pod 停止或者被终止。

5.1 生命周期

和一个个独立的应用容器一样，Pod 也被认为是相对临时性（而不是长期存在）的实体。 Pod 会被创建、赋予一个唯一的 ID(UID)， 并被调度到节点，并在终止（根据重启策略）或删除之前一直运行在该节点。如果一个节点死掉了，调度到该节点的 Pod 也被计划在给定超时期限结束后删除。

Pod 自身不具有自愈能力。如果 Pod 被调度到某节点而该节点之后失效， Pod 会被删除；类似地，Pod 无法在因节点资源耗尽或者节点维护而被驱逐期间继续存活。 Kubernetes 使用一种高级抽象来管理这些相对而言可随时丢弃的 Pod 实例， 称作控制器。

任何给定的 Pod （由 UID 定义）从不会被“重新调度（rescheduled）”到不同的节点； 相反，这一 Pod 可以被一个新的、几乎完全相同的 Pod 替换掉。 如果需要，新 Pod 的名字可以不变，但是其 UID 会不同。

如果某物声称其生命期与某 Pod 相同，例如存储卷， 这就意味着该对象在此 Pod （UID 亦相同）存在期间也一直存在。 如果 Pod 因为任何原因被删除，甚至某完全相同的替代 Pod 被创建时， 这个相关的对象（例如这里的卷）也会被删除并重建。

5.2 pod 阶段

Pod 阶段的数量和含义是严格定义的。 除了本文档中列举的内容外，不应该再假定 Pod 有其他的 `phase` 值。

取值	描述
`Pending`（悬决）	Pod 已被 Kubernetes 系统接受，但有一个或者多个容器尚未创建亦未运行。此阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间。
`Running`（运行中）	Pod 已经绑定到了某个节点，Pod 中所有的容器都已被创建。至少有一个容器仍在运行，或者正处于启动或重启状态。
`Succeeded`（成功）	Pod 中的所有容器都已成功终止，并且不会再重启。
`Failed`（失败）	Pod 中的所有容器都已终止，并且至少有一个容器是因为失败终止。也就是说，容器以非 0 状态退出或者被系统终止。
`Unknown`（未知）	因为某些原因无法取得 Pod 的状态。这种情况通常是因为与 Pod 所在主机通信失败。

> **说明：**
>
> 1. 当一个 Pod 被删除时，执行一些 kubectl 命令会展示这个 Pod 的状态为 `Terminating`（终止）。 这个 `Terminating` 状态并不是 Pod 阶段之一。 Pod 被赋予一个可以体面终止的期限，默认为 30 秒。 你可以使用 `--force` 参数来强制终止 Pod。
> 2. 如果某节点死掉或者与集群中其他节点失联，Kubernetes 会实施一种策略，将失去的节点上运行的所有 Pod 的 `phase` 设置为 `Failed` 。

6 Contrainer 特性

6.1 容器生命周期

Kubernetes 会跟踪 Pod 中每个容器的状态，就像它跟踪 Pod 总体上的阶段一样。 你可以使用容器生命周期回调来在容器生命周期中的特定时间点触发事件。

一旦调度器将 Pod 分派给某个节点， `kubelet` 就通过容器运行时开始为 Pod 创建容器。容器的状态有三种： `Waiting`（等待）、 `Running`（运行中）和 `Terminated`（已终止）。

要检查 Pod 中容器的状态，你可以使用 `kubectl describe pod <pod 名称>` 。 其输出中包含 Pod 中每个容器的状态。

每种状态都有特定的含义：

- `Waiting`（等待）

如果容器并不处在 `Running` 或 `Terminated` 状态之一，它就处在 `Waiting` 状态。 处于 `Waiting` 状态的容器仍在运行它完成启动所需要的操作：例如， 从某个容器镜像仓库拉取容器镜像，或者向容器应用 Secret 数据等等。 当你使用 `kubectl` 来查询包含 `Waiting` 状态的容器的 Pod 时，你也会看到一个 Reason 字段，其中给出了容器处于等待状态的原因。

- `Running`（运行中）

`Running` 状态表明容器正在执行状态并且没有问题发生。 如果配置了 `postStart` 回调，那么该回调已经执行且已完成。 如果你使用 `kubectl` 来查询包含 `Running` 状态的容器的 Pod 时，你也会看到关于容器进入 `Running` 状态的信息。

- `Terminated`（已终止）

处于 `Terminated` 状态的容器已经开始执行并且或者正常结束或者因为某些原因失败。 如果你使用 `kubectl` 来查询包含 `Terminated` 状态的容器的 Pod 时， 你会看到容器进入此状态的原因、退出代码以及容器执行期间的起止时间。

如果容器配置了 `preStop` 回调，则该回调会在容器进入 `Terminated` 状态之前执行。

6.2 容器生命周期回调

类似于许多具有生命周期回调组件的编程语言框架，例如 Angular、Vue、Kubernetes 为容器提供了生命周期回调。 回调使容器能够了解其管理生命周期中的事件，并在执行相应的生命周期回调时运行在处理程序中实现的代码。

有两个回调暴露给容器：

- `PostStart` 这个回调在容器被创建之后立即被执行。 但是，不能保证回调会在容器入口点（ENTRYPOINT）之前执行。 没有参数传递给处理程序。
- `PreStop` 在容器因 API 请求或者管理事件（诸如存活态探针、启动探针失败、资源抢占、资源竞争等） 而被终止之前，此回调会被调用。 如果容器已经处于已终止或者已完成状态，则对 preStop 回调的调用将失败。 在用来停止容器的 TERM 信号被发出之前，回调必须执行结束。 Pod 的终止宽限周期在 `PreStop` 回调被执行之前即开始计数， 所以无论回调函数的执行结果如何，容器最终都会在 Pod 的终止宽限期内被终止。 没有参数会被传递给处理程序。
- **使用容器生命周期回调**

```
# nginx-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.19
      lifecycle:
        postStart:
          exec:
            command: ["/bin/sh","-c","echo postStart >> /start.txt"]
        preStop:
          exec:
```

```
            command: ["/bin/sh","-c","echo postStop >> /stop.txt && sleep
5"]
      ports:
        - containerPort: 80
```

6.3 容器重启策略

Pod 的 `spec` 中包含一个 `restartPolicy` 字段，其可能取值包括 `Always(总是重启)、OnFailure(容器异常退出状态码非 0,重启) 和 Never` 。默认值是 `Always` 。

`restartPolicy` 适用于 Pod 中的所有容器。`restartPolicy` 仅针对同一节点上 `kubelet` 的容器重启动作。当 Pod 中的容器退出时，`kubelet` 会按指数回退方式计算重启的延迟（10s、20s、40s、...），其最长延迟为 5 分钟。 一旦某容器执行了 10 分钟并且没有出现问题，`kubelet` 对该容器的重启回退计时器执行重置操作。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.19
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

6.4 自定义容器启动命令

和 Docker 容器一样,k8s中容器也可以通过command、args 用来修改容器启动默认执行命令以及传递相关参数。**但一般推荐使用 command 修改启动命令，使用 args 为启动命令传递参数。**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
    app: redis
spec:
  containers:
    - name: redis
      image: redis:5.0.10
```

```
      command: ["redis-server"] #用来指定启动命令
      args: ["--appendonly yes"] # 用来为启动命令传递参数
      #args: ["redis-server","--appendonly yes"] # 单独使用修改启动命令并传递参
数
      #args:                                    # 另一种语法格式
      #  - redis-server
      #  - "--appendonly yes"
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

6.4 容器探针

probe 是由 kubelet对容器执行的定期诊断。 要执行诊断，kubelet 既可以在容器内执行代码，也可以发出一个网络请求。

探测类型

针对运行中的容器， `kubelet` 可以选择是否执行以下三种探针，以及如何针对探测结果作出反应：

- `livenessProbe` **指示容器是否正在运行**。如果存活态探测失败，则 kubelet 会杀死容器， 并且容器将根据其重启策略决定未来。如果容器不提供存活探针， 则默认状态为 `Success` 。
- `readinessProbe` **指示容器是否准备好为请求提供服**。如果就绪态探测失败， 端点控制器将从与 Pod 匹配的所有服务的端点列表中删除该 Pod 的 IP 地址。 初始延迟之前的就绪态的状态值默认为 `Failure` 。 如果容器不提供就绪态探针，则默认状态为 `Success` 。
- `startupProbe 1.7+` **指示容器中的应用是否已经启动**。如果提供了启动探针，则所有其他探针都会被 禁用，直到此探针成功为止。如果启动探测失败， `kubelet` 将杀死容器， 而容器依其重启策略进行重启。 如果容器没有提供启动探测，则默认状态为 `Success` 。

探针机制

使用探针来检查容器有四种不同的方法。 每个探针都必须准确定义为这四种机制中的一种：

- `exec`

  在容器内执行指定命令。如果命令退出时返回码为 0 则认为诊断成功。
- `grpc`

  使用 gRPC 执行一个远程过程调用。 目标应该实现 gRPC健康检查。 如果响应的状态是 "SERVING"，则认为诊断成功。 gRPC 探针是一个 Alpha 特性，只有在你启用了 "GRPCContainerProbe" 特性门控时才能使用。
- `httpGet`

  对容器的 IP 地址上指定端口和路径执行 HTTP `GET` 请求。如果响应的状态码大于等于 200 且小于 400，则诊断被认为是成功的。
- `tcpSocket`

  对容器的 IP 地址上的指定端口执行 TCP 检查。如果端口打开，则诊断被认为是成功的。 如果远程系统（容器）在打开连接后立即将其关闭，这算作是健康的。

探针结果

每次探测都将获得以下三种结果之一：

- `Success`（成功）容器通过了诊断。
- `Failure`（失败）容器未通过诊断。
- `Unknown`（未知）诊断失败，因此不会采取任何行动。

探针参数

```
initialDelaySeconds: 5   #初始化时间5s
periodSeconds: 4    #检测间隔时间4s
timeoutSeconds: 1   #默认检测超时时间为1s
failureThreshold: 3   #默认失败次数为3次，达到3次后重启pod
successThreshold: 1   #默认成功次数为1次，1次监测成功代表成功
```

使用探针

- **exec**

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
  labels:
    exec: exec
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
    args:
    - /bin/sh
    - -c
    - sleep 7;nginx -g "daemon off;" #这一步会和初始化同时开始运行，也就是在初始化5s后和7秒之间，会检测出一次失败，7秒后启动后检测正常，所以pod不会重启
    imagePullPolicy: IfNotPresent
    livenessProbe:
      exec:    #这里使用 exec 执行 shell 命令检测容器状态
        command:
        - ls
        - /var/run/nginx.pid  #查看是否有pid文件
      initialDelaySeconds: 5   #初始化时间5s
      periodSeconds: 4    #检测间隔时间4s
```

```
      timeoutSeconds: 1    #默认检测超时时间为1s
      failureThreshold: 3    #默认失败次数为3次，达到3次后重启pod
      successThreshold: 1    #默认成功次数为1次，1 次代表成功
```

说明：

```
1. 如果 sleep 7s，第一次检测发现失败，但是第二次检测发现成功后容器就一直处于健康状态不会重启。
1. 如果 sleep 30s，第一次检测失败，超过 3 次检测同样失败，k8s 就回杀死容器进行重启，反复循环这个过程。
```

- **tcpSocket**

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcpsocket
  labels:
    tcpsocket: tcpsocket
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
    args:
    - /bin/sh
    - -c
    - sleep 7;nginx -g "daemon off;"  #这一步会和初始化同时开始运行，也就是在初始化5s后和7秒之间，会检测出一次失败，7秒后启动后检测正常，所以pod不会重启
    imagePullPolicy: IfNotPresent
    livenessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 5    #初始化时间5s
      periodSeconds: 4     #检测间隔时间4s
      timeoutSeconds: 1    #默认检测超时时间为1s
      failureThreshold: 3    #默认失败次数为3次，达到3次后重启pod
      successThreshold: 1    #默认成功次数为1次，1 次代表成功
```

- **httpGet**

```
# probe-liveness-httget.yml
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-httpget
  labels:
    httpget: httpget
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
    args:
    - /bin/sh
    - -c
    - sleep 7;nginx -g "daemon off;" #这一步会和初始化同时开始运行，也就是在初始化5s后和7秒之间，会检测出一次失败，7秒后启动后检测正常，所以pod不会重启
    imagePullPolicy: IfNotPresent
    livenessProbe:
      httpGet:      #httpget
        port: 80    #访问的端口
        path: /index.html    #访问的路径
      initialDelaySeconds: 5    #初始化时间5s
      periodSeconds: 4     #检测间隔时间4s
      timeoutSeconds: 1    #默认检测超时时间为1s
      failureThreshold: 3    #默认失败次数为3次，达到3次后重启pod
      successThreshold: 1    #默认成功次数为1次，1 次代表成功
```

- GRPC 探针

> 官方参考地址：https://kubernetes.io/zh-cn/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

6.6 资源限制

在k8s中对于容器资源限制主要分为以下两类：

- 内存资源限制：内存**请求**（request）和内存**限制**（limit）分配给一个容器。 我们保障容器拥有它请求数量的内存，但不允许使用超过限制数量的内存。
  - 官网参考地址：https://kubernetes.io/zh-cn/docs/tasks/configure-pod-container/assign-memory-resource/

- CPU 资源限制：为容器设置 CPU **request（请求）** 和 CPU **limit（限制）**。 容器使用的 CPU 不能超过所配置的限制。 如果系统有空闲的 CPU 时间，则可以保证给容器分配其所请求数量的 CPU 资源。
  - 官网参考地址：https://kubernetes.io/zh-cn/docs/tasks/configure-pod-container/assign-cpu-resource/

1 metrics-server

> 官网地址：https://github.com/kubernetes-sigs/metrics-server

Kubernetes Metrics Server（Kubernetes指标服务器），它是一个**可扩展的、高效的容器资源度量源**。Metrics Server 用于监控每个 Node 和 Pod 的负载（用于Kubernetes内置自动扩缩管道）。Metrics Server 从Kubelets 收集资源指标，并通过 Metrics API 在Kubernetes apiserver 中公开，供 Horizontal Pod Autoscaler 和 Vertical Pod Autoscaler 使用。Metrics API 也可以通过 kubectl top 访问，使其更容易调试自动扩缩管道。

- 查看 metrics-server（或者其他资源指标 API `metrics.k8s.io` 服务提供者）是否正在运行， 请键入以下命令：

```
kubectl get apiservices
```

- 如果资源指标 API 可用，则会输出将包含一个对 `metrics.k8s.io` 的引用。

```
NAME
v1beta1.metrics.k8s.io
```

- 安装 metrics-server

```
# components.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    k8s-app: metrics-server
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
```

```yaml
    rbac.authorization.k8s.io/aggregate-to-view: "true"
  name: system:aggregated-metrics-reader
rules:
- apiGroups:
  - metrics.k8s.io
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    k8s-app: metrics-server
  name: system:metrics-server
rules:
- apiGroups:
  - ""
  resources:
  - nodes/metrics
  verbs:
  - get
- apiGroups:
  - ""
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server-auth-reader
  namespace: kube-system
```

```yaml
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-reader
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server:system:auth-delegator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: system:metrics-server
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:metrics-server
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: v1
kind: Service
metadata:
  labels:
```

```yaml
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
spec:
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: https
  selector:
    k8s-app: metrics-server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: metrics-server
  strategy:
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      labels:
        k8s-app: metrics-server
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=15s
        - --kubelet-insecure-tls #修改去掉证书验证
        image: dyrnq/metrics-server:v0.6.2 #修改官方无法下载
        imagePullPolicy: IfNotPresent
        livenessProbe:
          failureThreshold: 3
```

```
          httpGet:
            path: /livez
            port: https
            scheme: HTTPS
          periodSeconds: 10
        name: metrics-server
        ports:
        - containerPort: 4443
          name: https
          protocol: TCP
        readinessProbe:
          failureThreshold: 3
          httpGet:
            path: /readyz
            port: https
            scheme: HTTPS
          initialDelaySeconds: 20
          periodSeconds: 10
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
        securityContext:
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true
          runAsNonRoot: true
          runAsUser: 1000
        volumeMounts:
        - mountPath: /tmp
          name: tmp-dir
      nodeSelector:
        kubernetes.io/os: linux
      priorityClassName: system-cluster-critical
      serviceAccountName: metrics-server
      volumes:
      - emptyDir: {}
        name: tmp-dir
---
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  labels:
    k8s-app: metrics-server
```

```
  name: v1beta1.metrics.k8s.io
spec:
  group: metrics.k8s.io
  groupPriorityMinimum: 100
  insecureSkipTLSVerify: true
  service:
    name: metrics-server
    namespace: kube-system
  version: v1beta1
  versionPriority: 100
```

```
$ kubectl appply -f components.yaml
```

2 指定内存请求和限制

为容器指定内存请求，请在容器资源清单中包含 `resources: requests` 字段。 同理，要指定内存限制，请包含 `resources: limits` 。

```
# nginx-memory-demo.yaml
#内存资源的基本单位是字节（byte）。你可以使用这些后缀之一，将内存表示为 纯整数或定点整数：E、P、T、G、M、K、Ei、Pi、Ti、Gi、Mi、Ki。 例如，下面是一些近似相同的值：
128974848, 129e6, 129M, 123Mi
apiVersion: v1
kind: Pod
metadata:
  name: nginx-memory-demo
spec:
  containers:
  - name: nginx-memory-demo
    image: nginx:1.19
    resources:
      requests:
        memory: "100Mi"
      limits:
        memory: "200Mi"
```

- 查看容器内存使用情况

```
$ kubectl get pod nginx-memory-demo --output=yaml
```

- 查看容器正在使用内存情况

```
$ kubectl top pod nginx-memory-demo
```

- 内存请求和限制的目的

  通过为集群中运行的容器配置内存请求和限制，你可以有效利用集群节点上可用的内存资源。 通过将 Pod 的内存请求保持在较低水平，你可以更好地安排 Pod 调度。 通过让内存限制大于内存请求，你可以完成两件事：

  - Pod 可以进行一些突发活动，从而更好的利用可用内存。
  - Pod 在突发活动期间，可使用的内存被限制为合理的数量。

- 没有指定内存限制

  如果你没有为一个容器指定内存限制，则自动遵循以下情况之一：

  - 容器可无限制地使用内存。容器可以使用其所在节点所有的可用内存， 进而可能导致该节点调用 OOM Killer。 此外，如果发生 OOM Kill，没有资源限制的容器将被杀掉的可行性更大。
  - 运行的容器所在命名空间有默认的内存限制，那么该容器会被自动分配默认限制。

3 指定 CPU 请求和限制

为容器指定 CPU 请求，请在容器资源清单中包含 `resources: requests` 字段。 要指定 CPU 限制，请包含 `resources:limits` 。

```
# nginx-cpu-demo.yaml
#CPU 资源以 CPU 单位度量。小数值是可以使用的。一个请求 0.5 CPU 的容器保证会获得请求 1 个 CPU 的容器的 CPU 的一半。 你可以使用后缀 m 表示毫。例如 100m CPU、100 milliCPU 和 0.1 CPU 都相同。 CPU 请求只能使用绝对数量，而不是相对数量。0.1 在单核、双核或 48 核计算机上的 CPU 数量值是一样的。
apiVersion: v1
kind: Pod
metadata:
  name: nginx-cpu-demo
spec:
  containers:
  - name: nginx-cpu-demo
    image: nginx:1.19
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
```

- 显示 pod 详细信息

```
$ kubectl get pod nginx-cpu-demo --output=yaml
```

- 显示 pod 运行指标

```
$ kubectl top pod nginx-cpu-demo
```

- `CPU 请求和限制的初衷`

  通过配置你的集群中运行的容器的 CPU 请求和限制，你可以有效利用集群上可用的 CPU 资源。 通过将 Pod CPU 请求保持在较低水平，可以使 Pod 更有机会被调度。 通过使 CPU 限制大于 CPU 请求，你可以完成两件事：

  - Pod 可能会有突发性的活动，它可以利用碰巧可用的 CPU 资源。
  - Pod 在突发负载期间可以使用的 CPU 资源数量仍被限制为合理的数量。

- `如果不指定 CPU 限制`

  如果你没有为容器指定 CPU 限制，则会发生以下情况之一：

  - 容器在可以使用的 CPU 资源上没有上限。因而可以使用所在节点上所有的可用 CPU 资源。
  - 容器在具有默认 CPU 限制的名字空间中运行，系统会自动为容器设置默认限制。

- `如果你设置了 CPU 限制但未设置 CPU 请求`

  如果你为容器指定了 CPU 限制值但未为其设置 CPU 请求，Kubernetes 会自动为其 设置与 CPU 限制相同的 CPU 请求值。类似的，如果容器设置了内存限制值但未设置 内存请求值，Kubernetes 也会为其设置与内存限制值相同的内存请求。

7 Pod 中 init 容器

Init 容器是一种特殊容器，在Pod 内的应用容器启动之前运行。Init 容器可以包括一些应用镜像中不存在的实用工具和安装脚本。

7.1 init 容器特点

init 容器与普通的容器非常像，除了如下几点：

- 它们总是运行到完成。如果 Pod 的 Init 容器失败，kubelet 会不断地重启该 Init 容器直到该容器成功为止。 然而，如果 Pod 对应的 `restartPolicy` 值为 "Never"，并且 Pod 的 Init 容器失败， 则 Kubernetes 会将整个 Pod 状态设置为失败。
- 每个都必须在下一个启动之前成功完成。
- 同时 Init 容器不支持 `lifecycle` 、 `livenessProbe` 、 `readinessProbe` 和 `startupProbe` ， 因为它们必须在 Pod 就绪之前运行完成。
- 如果为一个 Pod 指定了多个 Init 容器，这些容器会按顺序逐个运行。 每个 Init 容器必须运行成功，下一个才能够运行。当所有的 Init 容器运行完成时， Kubernetes 才会为 Pod 初始化应用容器并像平常一样运行。
- Init 容器支持应用容器的全部字段和特性，包括资源限制、数据卷和安全设置。 然而，Init 容器对资源请求和限制的处理稍有不同。

7.2 使用 init 容器

> 官网地址：https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/

在 Pod 的规约中与用来描述应用容器的 `containers` 数组平行的位置指定 Init 容器。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'echo init-myservice is running! && sleep 5']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'echo init-mydb is running! && sleep 10']
```

- 查看启动详细

```
$ kubectl describe pod init-demo

# 部分结果
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  2m16s  default-scheduler  Successfully assigned
default/init-demo to k8s-node2
  Normal  Pulling    2m16s  kubelet            Pulling image "busybox:1.28"
  Normal  Pulled     118s   kubelet            Successfully pulled image
"busybox:1.28" in 17.370617268s (17.370620685s including waiting)
  Normal  Created    118s   kubelet            Created container init-
myservice
  Normal  Started    118s   kubelet            Started container init-
myservice
  Normal  Pulled     112s   kubelet            Container image
"busybox:1.28" already present on machine
```

```
  Normal  Created    112s   kubelet            Created container init-mydb
  Normal  Started    112s   kubelet            Started container init-mydb
  Normal  Pulled     101s   kubelet            Container image
"busybox:1.28" already present on machine
  Normal  Created    101s   kubelet            Created container myapp-
container
  Normal  Started    101s   kubelet            Started container myapp-
container
```

8 节点亲和性分配 Pod

> 官方地址：https://kubernetes.io/zh-cn/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/

使用节点亲和性可以把 Kubernetes Pod 分配到特定节点。

8.1 给节点添加标签

- 列出集群中的节点及其标签：

```
$ kubectl get nodes --show-labels
#输出类似于此：
NAME        STATUS   ROLES           AGE   VERSION   LABELS
k8s-node1   Ready    control-plane   10d   v1.26.0
beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux...
k8s-node2   Ready    <none>          10d   v1.26.0
beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux...
k8s-node3   Ready    <none>          10d   v1.26.0
beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux...
```

- 选择一个节点，给它添加一个标签：

```
kubectl label nodes k8s-node1(节点名称) disktype=ssd
```

- 验证你所选节点具有 `disktype=ssd` 标签：

```
$ kubectl get nodes --show-labels
#输出类似于此：
NAME        STATUS   ROLES           AGE   VERSION   LABELS
k8s-node1   Ready    control-plane   10d   v1.26.0
beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux,disktype=ssd..
.
k8s-node2   Ready    <none>          10d   v1.26.0
beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux...
k8s-node3   Ready    <none>          10d   v1.26.0
beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux...
```

8.2 依据强制的节点亲和性调度 Pod

```yaml
# affintiy-require-demo.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx:1.19
    imagePullPolicy: IfNotPresent
```

8.3 使用首选的节点亲和性调度 Pod

```yaml
# affintiy-preferre-demo.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
```

```
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx:1.19
    imagePullPolicy: IfNotPresent
```

第四章 Controller 控制器

- 什么是 Controller 以及作用
- 常见的 Controller 控制器
- Controller 如何管理 Pod
- Deployment 基本操作与应用
- 升级回滚和弹性伸缩

1 Controller 控制器

1.1 什么是 Controller

Kubernetes 通常不会直接创建 Pod， 而是通过 Controller 来管理 Pod 的。 **Controller 中定义了 Pod 的部署特性，比如有几个副本、在什么样的 Node 上运行等**。通俗的说可以认为 Controller 就是用来管理 Pod 一个对象。

1.2 常见的 Controller 控制器

- Deployment 是最常用的 Controller，比如在线教程中就是通过创建 Deployment 来部署应用的。Deployment 可以管理 Pod 的多个副本，并确保 Pod 按照期望的状态运行。
- ReplicaSet 实现了 Pod 的多副本管理。使用 Deployment 时会自动创建 ReplicaSet,也就是说 Deployment 是通过 ReplicaSet 来管理 Pod 的多个副本的，我们通常不需要直接使用 ReplicaSet。
- Daemonset 用于每个Node 最多只运行一个 Pod 副本的场景。正如其名称所揭示的，DaemonSet 通常用于运行 daemon。
- Statefuleset 能够保证 Pod 的每个副本在整个生命周期中名称是不变的，而其他Controller 不提供这个功能。当某个 Pod 发生故障需要删除并重新启动时，Pod 的名称会发生变化，同时

StatefuleSet 会保证副本按照固定的顺序启动、更新或者删除。

- Job用于运行结束就州除的应用，而其他 Controller 中的 Pod 通常是长期持续运行。

1.3 Controller 如何管理 Pod

2 Deployment 控制器

2.1 创建 deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.19
          ports:
            - containerPort: 80
```

2.2 查看 deployment

```
# kubectl get deployment | deployments
```

- `NAME` 列出了名字空间中 Deployment 的名称。
- `READY` 显示应用程序的可用的“副本”数。显示的模式是“就绪个数/期望个数”。
- `UP-TO-DATE` 显示为了达到期望状态已经更新的副本数。
- `AVAILABLE` 显示应用可供用户使用的副本数。

- `AGE` 显示应用程序运行的时间。

> 请注意期望副本数是根据 `.spec.replicas` 字段设置 3。

2.3 查看 deployment 详细

```
# kubectl describe deployment nginx-deployment(控制器名称)
```

2.4 查看 deployment 副本

```
# kubectl get rs|replicaset
```

5 查看 deployment

> **说明:**
>
> 仅当 Deployment Pod 模板（即 `.spec.template`）发生改变时，例如模板的标签或容器镜像被更新， 才会触发 Deployment 上线。其他更新（如对 Deployment 执行扩缩容的操作）不会触发上线动作。

```
# kubectl rollout status deployment/nginx-deployment
```

5 删除 deployment

```
# kubectl delete deployment nginx-deployment
# kubect delete -f nginx-deployment.yml
```

Service