

kubernetes 篇

基础篇：Kubernetes 简介与核心概念

Kubernetes 简介

什么是 Kubernetes?

Kubernetes（常简称为 K8s）是一个开源的容器编排和管理平台，用于自动化部署、扩展和操作应用程序容器。它提供了一个强大的工具集，可以简化容器化应用的部署、维护和扩展。

Kubernetes 的特点和优势

- 自动化容器部署和扩展：Kubernetes 可以自动处理容器的创建、部署和伸缩，使得应用程序的管理变得更加高效和简化。
- 跨平台和云原生：Kubernetes 支持在各种云平台和本地环境中运行，是构建**云原生应用**的理想选择。
- 服务发现和负载均衡：Kubernetes 提供内置的**服务发现**和**负载均衡**功能，使得应用程序可以轻松地进行相互通信和平衡负载。
- 自愈和自恢复能力：Kubernetes 可以监控和管理应用程序的状态，并在出现故障时自动进行恢复，确保应用的高可用性。
- 灵活的存储管理：Kubernetes 提供了灵活的存储管理机制，可以方便地挂载存储卷，并支持各种存储后端和插件。
- 跨团队协作：Kubernetes 支持多团队共享和管理应用程序，通过**命名空间**等机制，实现了资源的隔离和安全性。

Kubernetes 的核心组件

Kubernetes 由多个核心组件组成，每个组件都有特定的职责和功能：

- **Master节点**：包括 **API Server**、**Controller Manager**、**Scheduler** 和 **etcd**（键值存储系统），用于管理整个集群的状态和控制平面。
- **Node节点**：也称为工作节点，运行着应用程序容器。每个节点上都有 **kubelet** 代理、容器运行时和网络插件等组件。
- **Pod**：是 Kubernetes 的最小部署单元，包含一个或多个容器。它是应用程序的运行实例，具有独立的 IP 地址和存储卷。
- **控制器**：如 **ReplicaSet** 和 **Deployment**，用于定义和管理应用程序的期望状态，并确保集群中的 Pod 数量和状态符合预期。
- **服务**：提供了一种稳定的网络访问方式，用于将流量路由到一组具有相同标签的 Pod。

Kubernetes 示例

下面是一个简单的 Kubernetes 示例，用于部署一个 Web 应用程序：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: web-pod
5  spec:
6    containers:
7      - name: web-container
8        image: nginx:latest
9        ports:
10       - containerPort: 80
```

上述示例定义了一个 Pod，其中包含一个名为 "web-container" 的容器，使用 **nginx:latest** 镜像，并将容器的端口映射到主机的端口80。

架构和组件概述

Kubernetes 架构

Kubernetes 采用了一种分布式的架构，由多个节点协同工作来管理和运行应用程序容器。这些节点包括 Master 节点和 Node 节点。

Master 节点

Master 节点是 Kubernetes 集群的控制平面，它负责管理和监控整个集群。主要组件包括：

- **API Server**：作为 Kubernetes 集群的入口，处理来自用户、CLI 工具和其他组件的请求。它负责验证请求、修改集群状态，并将结果返回给请求方。
- **Controller Manager**：负责运行各种控制器，监控集群状态并进行调整。例如，ReplicaSet 控制器负责确保指定数量的 Pod 在集群中运行。
- **Scheduler**：负责将新创建的 Pod 调度到集群的合适节点上。它根据节点的资源、亲和性和其他约束条件来做出决策。
- **etcd**：是一个分布式键值存储系统，用于保存集群的配置信息和状态。所有的集群数据都存储在 etcd 中，包括节点信息、Pod 状态、配置信息等。

Node 节点

Node 节点是 Kubernetes 集群中的工作节点，用于运行容器化的应用程序。每个节点上都包含以下组件：

- **kubelet**：作为节点上的代理，负责与 Master 节点通信，并管理节点上的 Pod。它接收来自 Master 的指令，启动、停止和监控 Pod 的运行状态。
- **容器运行时**：Kubernetes 支持多种容器运行时，如 Docker、Containerd 等。容器运行时负责管理和运行容器，提供隔离、资源管理和安全性等功能。
- **网络插件**：用于提供 Pod 之间和与外部网络的通信。不同的网络插件可以实现不同的网络模型，如基于虚拟网络的 Overlay 网络、主机网络等。
- **Proxy**：运行在每个节点上的网络代理，负责转发和负载均衡 Pod 的网络请求。
- **Pod**：是 Kubernetes 的最小调度单位，包含一个或多个容器。每个节点上运行的容器都包含在一个 Pod 中，Pod 有自己的 IP 地址和存储卷。

Kubernetes 组件

Kubernetes 的各个组件之间通过 API Server 进行交互，实现集群的管理和协调。以下是一些常见的组件交互关系：

- **用户和 CLI 工具**：通过 API Server 发送请求，例如创建、更新或删除资源的操作。
- **Controller Manager**：通过 API Server 监听集群状态的变化，根据需要启动或停止相关的控制器。
- **Scheduler**：通过 API Server 获取待调度的 Pod 信息，根据节点的资源情况和调度策略进行调度决策。
- **kubelet**：通过 API Server 获取指令和 Pod 配置信息，将指令传达给容器运行时，启动和管理 Pod。
- **Proxy**：负责转发和负载均衡 Pod 的网络请求，通过与 API Server 通信获取服务和端口信息。
- **etcd**：作为集群的分布式键值存储系统，保存着集群的配置和状态信息。各个组件通过 etcd 进行数据的读写和同步。

这些组件的协作和交互构成了 Kubernetes 的完整体系结构，支持各种容器化应用的部署、管理和扩展。

Kubernetes 的安装和配置

Kubernetes 部署前期准备

Docker 和 Kubernetes 版本对应关系

[对应关系](#)：Docker 的版本是 v19.x，对应的 Kubernetes 的版本是 v1.21。

关闭&禁用防火墙

```
1 | systemctl stop firewalld
2 | systemctl disable firewalld
```

升级系统内核（非必须）

- 查看当前系统的版本

```
1 | cat /etc/redhat-release
```

- 看当前系统的内核

```
1 | uname -sr
```

- 查看启动顺序

```
1 | # 查看启动顺序
2 | yum install -y grub2-pc
3 | grub2-editenv list
```

- 查看可用内核版本及启动顺序

```
1 | sudo awk -F\ ' '$1=="menuentry " {print i++ " : " $2}' /boot/grub2/grub.cfg
```

- 在 CentOS 7.x 上启用 ELRepo 仓库

```
1 rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org
2 rpm -Uvh https://www.elrepo.org/elrepo-release-7.0-4.el7.elrepo.noarch.rpm
```

提升下载速度，换源

- `sed -i "s/mirrorlist=/#mirrorlist=g" /etc/yum.repos.d/elrepo.repo`
- `sed -i "s#elrepo.org/linux#mirrors.tuna.tsinghua.edu.cn/elrepo#g" /etc/yum.repos.d/elrepo.repo`

- 查看可用的系统内核相关包

```
1 yum --disablerepo="*" --enablerepo="elrepo-kernel" list available
```

- 安装最新主线内核版本

```
1 yum -y --enablerepo=elrepo-kernel install kernel-ml
```

- 设置默认的内核版本

```
1 vim /etc/default/grub
```

```
1 GRUB_TIMEOUT=5
2 GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
3 GRUB_DEFAULT=0 # 修改此处，原来是 saved
4 GRUB_DISABLE_SUBMENU=true
5 GRUB_TERMINAL_OUTPUT="console"
6 GRUB_CMDLINE_LINUX="crashkernel=auto rd.lvm.lv=centos/root
   rd.lvm.lv=centos/swap rhgb quiet"
7 GRUB_DISABLE_RECOVERY="true"
```

或者使用如下命令

```
1 sed -i 's/^GRUB_DEFAULT=saved$/GRUB_DEFAULT=0/' /etc/default/grub
```

- 重新创建内核配置

```
1 grub2-mkconfig -o /boot/grub2/grub.cfg
```

- 重启系统

```
1 reboot
```

- 查看当前系统的内核

```
1 uname -sr
```

设置主机名 (部署集群需要)

```
1 # 192.168.183.101
2 hostnamectl set-hostname master
3 # 192.168.183.102
4 hostnamectl set-hostname node1
5 # 192.168.183.103
6 hostnamectl set-hostname node2
```

主机名解析 (部署集群需要)

为了方便后面集群节点间的直接调用, 需要配置一下主机名解析。

```
1 cat >> /etc/hosts << EOF
2 127.0.0.1    $(hostname)
3 192.168.183.101 master
4 192.168.183.102 node1
5 192.168.183.103 node2
6 EOF
```

时间同步 (部署集群需要)

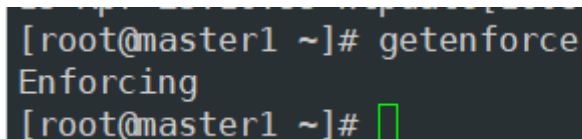
Kubernetes 要求集群中的节点时间必须精确一致, 所以在每个节点上添加时间同步

```
1 yum install ntpdate -y && ntpdate time.windows.com
```

关闭 SELinux

- 查看 SELinux 是否开启

```
1 getenforce
```



A terminal window showing the output of the 'getenforce' command. The prompt is '[root@master1 ~]#'. The output is 'Enforcing'. The prompt is '[root@master1 ~]#' followed by a green cursor.

- 永久关闭 SELinux, 需要重启

```
1 sed -i 's/enforcing/disabled/' /etc/selinux/config
```

- 关闭当前会话的 SELinux, 重启之后无效

```
1 setenforce 0
```

关闭 swap 分区

- 永久关闭 swap, 需要重启

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

- 关闭当前会话的 swap, 重启之后无效

```
1 | swapoff -a
```

将桥接的 IPv4 流量传递到 iptables 的链

- 修改 /etc/sysctl.conf 文件

```
1 # 如果有配置, 则修改
2 sed -i "s#^net.ipv4.ip_forward.*#net.ipv4.ip_forward=1#g" /etc/sysctl.conf
3 sed -i "s#^net.bridge.bridge-nf-call-ip6tables.*#net.bridge.bridge-nf-call-
4 ip6tables=1#g" /etc/sysctl.conf
5 sed -i "s#^net.bridge.bridge-nf-call-iptables.*#net.bridge.bridge-nf-call-
6 iptables=1#g" /etc/sysctl.conf
7 sed -i "s#^net.ipv6.conf.all.disable_ipv6.*#net.ipv6.conf.all.disable_ipv6=1#g"
8 /etc/sysctl.conf
9 sed -i "s#^net.ipv6.conf.default.disable_ipv6.*#net.ipv6.conf.default.disable_ipv6=1
10 #g" /etc/sysctl.conf
11 sed -i "s#^net.ipv6.conf.lo.disable_ipv6.*#net.ipv6.conf.lo.disable_ipv6=1#g"
12 /etc/sysctl.conf
13 sed -i "s#^net.ipv6.conf.all.forwarding.*#net.ipv6.conf.all.forwarding=1#g"
14 /etc/sysctl.conf
15 sed -i "s#^net.ipv4.ip_nonlocal_bind.*#net.ipv4.ip_nonlocal_bind=1#g"
16 /etc/sysctl.conf
```

- 可能没有, 追加

```
1 # 可能没有, 追加
2 echo "net.ipv4.ip_forward = 1" >> /etc/sysctl.conf
3 echo "net.bridge.bridge-nf-call-ip6tables = 1" >> /etc/sysctl.conf
4 echo "net.bridge.bridge-nf-call-iptables = 1" >> /etc/sysctl.conf
5 echo "net.ipv6.conf.all.disable_ipv6 = 1" >> /etc/sysctl.conf
6 echo "net.ipv6.conf.default.disable_ipv6 = 1" >> /etc/sysctl.conf
7 echo "net.ipv6.conf.lo.disable_ipv6 = 1" >> /etc/sysctl.conf
8 echo "net.ipv6.conf.all.forwarding = 1" >> /etc/sysctl.conf
9 echo "net.ipv4.ip_nonlocal_bind=1" >> /etc/sysctl.conf
```

- 加载 br_netfilter 模块

```
1 | modprobe br_netfilter
```

- 持久化修改 (保留配置包本地文件, 重启系统或服务进程仍然有效)

```
1 | sysctl -p
```

开启 ipvs

- 在 Kubernetes 中 service 有两种代理模型, 一种是基于 iptables, 另一种是基于 ipvs 的。ipvs 的性能要高于 iptables 的, 但是如果使用它, 需要手动载入 ipvs 模块。
- 在所有机器安装 ipset 和 ipvsadm

```
1 | yum -y install ipset ipvsadm
```

- 在所有机器执行如下脚本

```
1 cat > /etc/sysconfig/modules/ipvs.modules <<EOF
2 #!/bin/bash
3 modprobe -- ip_vs
4 modprobe -- ip_vs_rr
5 modprobe -- ip_vs_wrr
6 modprobe -- ip_vs_sh
7 modprobe -- nf_conntrack
8 EOF
```

- 授权、运行、检查是否加载

```
1 chmod 755 /etc/sysconfig/modules/ipvs.modules && bash
   /etc/sysconfig/modules/ipvs.modules && lsmod | grep -e ip_vs -e
   nf_conntrack_ipv4
```

重启

```
1 reboot
```

安装 docker

- 卸载原有版本，准备安装对应版本

```
1 sudo yum remove docker \
2     docker-client \
3     docker-client-latest \
4     docker-common \
5     docker-latest \
6     docker-latest-logrotate \
7     docker-logrotate \
8     docker-engine
```

- yum 安装 gcc 相关

```
1 yum -y install gcc
2 yum -y install gcc-c++
```

- 安装所需要的软件包

```
1 yum -y install yum-utils
```

- 设置 stable 镜像仓库

```
1 yum-config-manager --add-repo https://mirrors.aliyun.com/docker-
   ce/linux/centos/docker-ce.repo
```

- 更新 yum 软件包索引

```
1 yum makecache fast
```

- 查看存储库中 Docker 的版本

```
1 | yum list docker-ce --showduplicates | sort -r
```

- 安装指定版本的 Docker

```
1 | yum install -y docker-ce-19.03.14 docker-ce-cli-19.03.14 containerd.io
```

- 启动 Docker

```
1 | # 启动 Docker
2 | systemctl start docker
```

- 设置为开机自启动

```
1 | # 开启自动启动
2 | systemctl enable docker
```

- 验证 Docker 是否安装成功

```
1 | docker version
```

- 阿里云镜像加速

```
1 | sudo mkdir -p /etc/docker
```

```
1 | sudo tee /etc/docker/daemon.json <<- 'EOF'
2 | {
3 |     "exec-opts": ["native.cgroupdriver=systemd"],
4 |     "registry-mirrors": [
5 |         "https://du3ia00u.mirror.aliyuncs.com",
6 |         "https://hub-mirror.c.163.com",
7 |         "https://mirror.baidubce.com",
8 |         "https://registry.docker-cn.com",
9 |         "https://docker.mirrors.sjtug.sjtu.edu.cn"
10 |    ],
11 |     "dns": ["8.8.8.8", "114.114.114.114"],
12 |     "live-restore": true,
13 |     "log-driver": "json-file",
14 |     "log-opts": {"max-size": "500m", "max-file": "3"},
15 |     "max-concurrent-downloads": 10,
16 |     "max-concurrent-uploads": 5,
17 |     "storage-driver": "overlay2"
18 | }
19 | EOF
```

```
1 | sudo systemctl daemon-reload
2 | sudo systemctl restart docker
```


添加阿里云的 Kubernetes 的 YUM 源

由于 Kubernetes 的镜像源在国外，非常慢，这里切换成国内的阿里云镜像源（所有机器均需执行下面命令）

```
1 cat > /etc/yum.repos.d/kubernetes.repo << EOF
2 [kubernetes]
3 name=Kubernetes
4 baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
5 enabled=1
6 gpgcheck=0
7 repo_gpgcheck=0
8 gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
  https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
9 EOF
```

下载 kubekey

```
1 mkdir tmp && cd tmp
2
3 export KKZONE=cn
4 curl -sL https://get-kk.kubesphere.io | VERSION=v2.0.0 sh -
5
6 wget https://github.com/kubesphere/kubekey/releases/download/v2.0.0/kubekey-
  v2.0.0-linux-amd64.tar.gz
7
8 tar -zxvf kubekey-v2.0.0-linux-amd64.tar.gz
9
10 # 添加可执行权限
11 chmod +x kk
```

安装依赖或者相应工具

```
1 yum install socat -y
2 yum install conntrack -y
3 yum install ebtables -y
4 yum install ipset -y
```

单机部署

安装

```
1 ./kk create cluster --with-kubernetes v1.20.10 --with-kubesphere v3.1.1
```

- 安装 KubeSphere 3.2.1 的建议 Kubernetes 版本：1.19.x、1.20.x、1.21.x 或 1.22.x（实验性支持）。如果不指定 Kubernetes 版本，KubeKey 将默认安装 Kubernetes v1.21.5。有关受支持的 Kubernetes 版本的更多信息，请参见[支持矩阵](#)。
- 一般来说，对于 All-in-One 安装，无需更改任何配置。
- 如果在这一步的命令中不添加标志 `--with-kubesphere`，则不会部署 KubeSphere，KubeKey 将只安装 Kubernetes。如果添加标志 `--with-kubesphere` 时不指定 KubeSphere 版本，则会安装最新版本的 KubeSphere。
- KubeKey 会默认安装 [OpenEBS](#) 为开发和测试环境提供 LocalPV 以方便新用户。对于其他存储类型，请参见[持久化存储配置](#)。

验证

```
1 kubectl logs -n kubespere-system $(kubectl get pod -n kubespere-system -l 'app in (ks-install, ks-installer)' -o jsonpath='{.items[0].metadata.name}') -f
```

卸载

```
1 ./kk delete cluster
```

集群部署

机器要求

本示例，创建8台 Centos Linux release 7.6.1810 (Core) 的虚拟机部署默认的最小化安装。其中，主机配置为2Core，4 GB，40 G，从机配置4Core，8GB，300G

序号	主机IP	主机名称	角色
1	10.39.174.127	master1	master, etcd
2	10.39.174.205	master2	master, etcd
3	10.39.174.109	master3	master, etcd
4	10.39.174.51	node1	worker
5	10.39.174.161	node2	worker
6	10.39.174.138	node3	worker
7	10.39.174.128	vip	虚拟IP（不是实际的主机）
8	10.39.174.126	lb-0	lb （Keepalived + HAProxy）
9	10.39.174.33	lb-1	lb （Keepalived + HAProxy）

部署 keepalived 和 haproxy

安装

在主机为 lb-0 和 lb-1 中部署 Keepalived + HAProxy 即 IP 为 10.39.174.126 与 10.39.174.33 的服务器上安装部署 HAProxy 和 psmisc。

```
1 yum install keepalived haproxy psmisc -y
```

配置 haproxy

在主机为 lb-0 和 lb-1 的服务器上按如下参数配置 HAProxy (两台 lb 机器配置一致即可，注意后端服务地址)。

```
1 cat <<END>/etc/haproxy/haproxy.cfg
2
3 # HAProxy Configure /etc/haproxy/haproxy.cfg
4 global
5
```

```
6      # 设置日志输出的目标地址和日志级别
7      log      127.0.0.1 local2
8
9      # 将HAProxy进程chroot到指定的目录
10     chroot    /var/lib/haproxy
11
12     # 指定HAProxy进程的PID文件位置
13     pidfile   /var/run/haproxy.pid
14
15     # 设置最大并发连接数
16     maxconn    4000
17
18     # 指定运行HAProxy进程的用户和组
19     user       haproxy
20     group      haproxy
21
22     # 以守护进程方式运行HAProxy
23     daemon
24
25     # 打开统计信息的Unix套接字
26     stats socket /var/lib/haproxy/stats
27
28     #-----
29
30     # common defaults that all the 'listen' and 'backend' sections will
31
32     # use if not designated in their block
33
34     #-----
35
36     defaults
37         # 默认使用全局日志设置
38         log                        global
39
40         # 设置一些默认选项，如HTTP日志和不记录空连接
41         option                     httplog
42         option                     dontlognull
43
44         # 设置一些默认选项，如HTTP日志和不记录空连接
45         timeout connect            5000
46         timeout client             5000
47         timeout server             5000
48
49     #-----
50
51     # main frontend which proxys to the backends
52
53     #-----
54
55     frontend kube-apiserver
56
57         # 设置一些默认选项，如HTTP日志和不记录空连接
58         bind *:6443
59
60         # 使用TCP模式
```

```

61     mode tcp
62
63     # 启用TCP日志
64     option tcplog
65
66     # 默认后端为kube-apiserver
67     default_backend kube-apiserver
68
69     #-----
70
71     # round robin balancing between the various backends
72
73     #-----
74
75     backend kube-apiserver
76         # 使用TCP模式
77         mode tcp
78
79         # 启用TCP日志
80         option tcplog
81
82         # 使用轮询算法进行负载均衡
83         balance      roundrobin
84
85         # 设置默认服务器参数，包括健康检查和连接限制
86         default-server inter 10s downinter 5s rise 2 fall 2 slowstart 60s
87         maxconn 250 maxqueue 256 weight 100
88
89         # 定义kube-apiserver 服务器（master），检查服务器健康状态
90         server kube-apiserver-1 10.39.174.127:6443 check
91
92         server kube-apiserver-2 10.39.174.205:6443 check
93
94         server kube-apiserver-3 10.39.174.109:6443 check
95
96     END

```

启动之前检查语法是否有问题

```
1 haproxy -f /etc/haproxy/haproxy.cfg -c
```

启动 Haproxy, 并设置开机自启动

```
1 systemctl restart haproxy && systemctl enable haproxy
```

停止 Haproxy

```
1 systemctl stop haproxy
```

配置 keepalived

在主机为 lb-0 的服务器上按如下参数配置 keepalived。

```
1 cat <<END>/etc/keepalived/keepalived.conf
2
3 global_defs {
4
5     notification_email {
6         # 在此处填入通知邮件的收件人地址
7     }
8
9     # 设置SMTP连接的超时时间为30秒
10    smtp_connect_timeout 30
11
12    # 设置该服务器的标识符为LVS_DEVEL01，相当于给服务器起了一个别名
13    router_id LVS_DEVEL01
14
15    # 跳过对对等地址的检查
16    vrrp_skip_check_adv_addr
17
18    # 设置GARP (Gratuitous ARP) 通告的间隔时间为0，即不发送GARP通告
19    vrrp_garp_interval 0
20    vrrp_gna_interval 0
21
22 }
23
24 # 定义一个检查HAProxy进程的脚本，每2秒运行一次，权重为20
25 vrrp_script chk_haproxy {
26     script "killall -0 haproxy"
27     interval 2
28     weight 20
29 }
30
31 vrrp_instance haproxy-vip {
32     # 设置该实例为主服务器
33     state MASTER
34
35     # 设置主服务器的优先级为100，比备用服务器高
36     priority 100
37
38     # 设置实例绑定的网络接口为 eth0
39     interface eth0
40
41     # 定义一个虚拟路由器组，虚拟路由器组的标识符为60
42     virtual_router_id 60
43
44     # 设置通告间隔为1秒，用于检查对方是否存活
45     advert_int 1
46
47     # 设置认证类型为PASS，并指定认证密码为1111（这些相当于认证的密码）
48     authentication {
49         auth_type PASS
50         auth_pass 1111
51     }
```

```

52
53     # 设置当前机器的IP地址为源地址
54     unicast_src_ip 10.39.174.126
55
56     # 设置对等机器的IP地址（其他机器的地址）
57     unicast_peer {
58         10.39.174.33
59     }
60
61     # 设置虚拟IP地址 - vip地址
62     virtual_ipaddress {
63         10.38.174.128/24
64     }
65     # 设置要跟踪的脚本为chk_haproxy
66     track_script {
67         chk_haproxy
68     }
69 }
70
71 END

```

在主机为 lb-0 的服务器上按如下参数配置 keepalived。

```

1  cat <<END>/etc/keepalived/keepalived.conf
2
3  global_defs {
4
5      notification_email {
6
7      }
8
9      router_id LVS_DEVELO2 ##相当于给这个服务器起个昵称
10
11      vrrp_skip_check_adv_addr
12
13      vrrp_garp_interval 0
14
15      vrrp_gna_interval 0
16
17  }
18
19  vrrp_script chk_haproxy {
20
21      script "killall -0 haproxy"
22
23      interval 2
24
25      weight 20
26
27  }
28
29  vrrp_instance haproxy-vip {
30
31      state BACKUP #备份服务器 是 backup
32

```

```

33     priority 90 #优先级要低（把备份的90修改为100）
34
35     interface eth0    #实例绑定的网卡
36
37     virtual_router_id 60
38
39     advert_int 1
40
41     authentication {
42
43         auth_type PASS
44
45         auth_pass 1111
46
47     }
48
49     unicast_src_ip 10.39.174.33    #当前机器地址
50
51     unicast_peer {
52
53         10.39.174.126    #peer 中其它机器地址
54
55     }
56
57     virtual_ipaddress {
58
59         #加/24
60
61         10.38.174.128/24
62
63     }
64
65     track_script {
66
67         chk_haproxy
68
69     }
70
71 }
72
73
74 END

```

启动 keepalived，设置开机自启动

```

1  systemctl restart keepalived && systemctl enable keepalived
2  systemctl stop keepalived

```

开启 keepalived服务

```

1  systemctl start keepalived

```

验证可用性

使用 `ip a s` 查看各 lb 节点 vip 绑定情况

```
1 | ip a s
```

暂停 vip 所在节点 HAProxy

```
1 | systemctl stop haproxy
```

再次使用 `ip a s` 查看各 lb 节点 vip 绑定情况，查看 vip 是否发生漂移

```
1 | ip a s
```

或者使用下面命令查看

```
1 | systemctl status -l keepalived
```

#####

创建配置文件

每个主从机器都要配置，最重要的是一个一个部署，并发容易出现错误。

```
1 | ./kk create config --with-kubernetes v1.20.10 --with-kubesphere v3.1.1
```

ssh 互相免密

方便后续，配置文件中的时候，不需要设置密码了。

- 生成密钥

```
1 | ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
```

- host验证

`vim /etc/ssh/ssh_config` 在最后添加

```
1 | StrictHostKeyChecking no
2 | UserKnownHostsFile /dev/null
```

按大写G，直接跳到最后一行。

- 将密钥分别拷贝给自己和别人

```
1 | ssh-copy-id -i ~/.ssh/id_rsa.pub root@10.39.174.127
2 |
3 | ssh-copy-id -i ~/.ssh/id_rsa.pub root@10.39.174.205
4 |
5 | ...
```


修改默认配置文件

```
1  apiVersion: kubekey.kubesphere.io/v1alpha2
2  kind: Cluster
3  metadata:
4    name: sample
5  spec:
6    hosts: # 修改
7    - {name: master1, address: 10.39.174.127, internalAddress: 10.39.174.127,
      password: "123456"}
8    - {name: master2, address: 10.39.174.205, internalAddress: 10.39.174.205,
      password: "123456"}
9    - {name: master3, address: 10.39.174.109, internalAddress: 10.39.174.109,
      password: "123456"}
10   - {name: node1, address: 10.39.174.51, internalAddress: 10.39.174.51,
      password: "123456"}
11   - {name: node2, address: 10.39.174.161, internalAddress: 10.39.174.161,
      password: "123456"}
12   - {name: node3, address: 10.39.174.138, internalAddress: 10.39.174.138,
      password: "123456"}
13   roleGroups:
14     etcd:
15     - master1
16     - master2
17     - master3
18     control-plane:
19     - master1
20     - master2
21     - master3
22     worker:
23     - node1
24     - node2
25     - node3
26   controlPlaneEndpoint:
27     domain: lb.kubesphere.local
28     # vip, 这里配置的vip
29     address: "10.38.174.128"
30     port: 6443
31   kubernetes:
32     version: v1.20.10
33     imageRepo: kubesphere
34     clusterName: cluster.local
35     masqueradeAll: false # masqueradeAll tells kube-proxy to SNAT
36     everything if using the pure iptables proxy mode. [Default: false]
37     maxPods: 110 # maxPods is the number of pods that can run on this
38     Kubelet. [Default: 110]
39     nodeCidrMaskSize: 24 # internal network node size allocation. This is
40     the size allocated to each node on your network. [Default: 24]
41     proxyMode: ipvs # mode specifies which proxy mode to use. [Default:
42     ipvs]
43   network:
44     plugin: calico
45     calico:
```

```
42     ipipMode: Always # IPIP Mode to use for the IPv4 POOL created at
start up. If set to a value other than Never, vxlanMode should be set to
"Never". [Always | CrossSubnet | Never] [Default: Always]
43     vxlanMode: Never # VXLAN Mode to use for the IPv4 POOL created at
start up. If set to a value other than Never, ipipMode should be set to
"Never". [Always | CrossSubnet | Never] [Default: Never]
44     vethMTU: 1440 # The maximum transmission unit (MTU) setting
determines the largest packet size that can be transmitted through your
network. [Default: 1440]
45     kubePodsCIDR: 10.233.64.0/18
46     kubeServiceCIDR: 10.233.0.0/18
47     registry:
48       registryMirrors: []
49       insecureRegistries: []
50     addons: []
51
52 ---
53 apiVersion: installer.kubesphere.io/v1alpha1
54 kind: ClusterConfiguration
55 metadata:
56   name: ks-installer
57   namespace: kubesphere-system
58   labels:
59     version: v3.1.1
60 spec:
61   persistence:
62     storageClass: ""
63   authentication:
64     jwtSecret: ""
65   zone: ""
66   local_registry: ""
67   etcd:
68     monitoring: false
69     endpointIps: localhost
70     port: 2379
71     tlsEnable: true
72   common:
73     redis:
74       enabled: false
75       redisVolumeSize: 2Gi
76     openldap:
77       enabled: false
78       openldapVolumeSize: 2Gi
79     minioVolumeSize: 20Gi
80     monitoring:
81       endpoint: http://prometheus-operated.kubesphere-monitoring-
system.svc:9090
82     es:
83       elasticsearchMasterVolumeSize: 4Gi
84       elasticsearchDataVolumeSize: 20Gi
85       logMaxAge: 7
86       elkPrefix: logstash
87     basicAuth:
88       enabled: false
89       username: ""
```

```
90     password: ""
91     externalElasticsearchUrl: ""
92     externalElasticsearchPort: ""
93 console:
94     enableMultiLogin: true
95     port: 30880
96 alerting:
97     enabled: false
98     # thanosruler:
99     #   replicas: 1
100    #   resources: {}
101 auditing:
102     enabled: false
103 devops:
104     enabled: false
105     jenkinsMemoryLim: 2Gi
106     jenkinsMemoryReq: 1500Mi
107     jenkinsVolumeSize: 8Gi
108     jenkinsJavaOpts_Xms: 512m
109     jenkinsJavaOpts_Xmx: 512m
110     jenkinsJavaOpts_MaxRAM: 2g
111 events:
112     enabled: false
113     ruler:
114         enabled: true
115         replicas: 2
116 logging:
117     enabled: false
118     logsidecar:
119         enabled: true
120         replicas: 2
121 metrics_server:
122     enabled: false
123 monitoring:
124     storageClass: ""
125     prometheusMemoryRequest: 400Mi
126     prometheusVolumeSize: 20Gi
127 multicluster:
128     clusterRole: none
129 network:
130     networkpolicy:
131         enabled: false
132     ippool:
133         type: none
134     topology:
135         type: none
136 openpitrix:
137     store:
138         enabled: false
139 servicemesh:
140     enabled: false
141 kubeedge:
142     enabled: false
143     cloudCore:
144         nodeSelector: {"node-role.kubernetes.io/worker": ""}
```

```

145     tolerations: []
146     cloudhubPort: "10000"
147     cloudhubQuicPort: "10001"
148     cloudhubHttpsPort: "10002"
149     cloudstreamPort: "10003"
150     tunnelPort: "10004"
151     cloudHub:
152       advertiseAddress:
153         - ""
154       nodeLimit: "100"
155     service:
156       cloudhubNodePort: "30000"
157       cloudhubQuicNodePort: "30001"
158       cloudhubHttpsNodePort: "30002"
159       cloudstreamNodePort: "30003"
160       tunnelNodePort: "30004"
161     edgeWatcher:
162       nodeSelector: {"node-role.kubernetes.io/worker": ""}
163       tolerations: []
164       edgeWatcherAgent:
165         nodeSelector: {"node-role.kubernetes.io/worker": ""}
166         tolerations: []

```

安装

```
1 | ./kk create cluster -f config-sample.yaml
```

查看安装进度

```
1 | kubectl logs -n kubesphere-system $(kubectl get pod -n kubesphere-system -l
  app=ks-install -o jsonpath='{.items[0].metadata.name}') -f
```

问题出现

The connection to the server localhost:8080 was refused - did you specify the right host or port?

- 主节点出现该问题

```

1 | sudo mkdir ~/.kube
2 | sudo cp /etc/kubernetes/admin.conf ~/.kube/
3 |
4 | cd ~/.kube
5 |
6 | sudo mv admin.conf config
7 | sudo service kubelet restart

```

- 从节点出现该问题：权限不足，需要使用上述配置文件，从主节点的 `~/.kube\` 下的配置文件拷贝到从节点中。

Kubernetes 对象概述

什么是kubernetes对象

在 Kubernetes 中，对象是用于描述和管理集群中各种资源的抽象概念。它们是 Kubernetes API 的一部分，通过定义对象的规范和配置来创建、更新和删除集群中的资源。每个对象都有一个唯一的标识符（例如名称和命名空间），并包含一组属性和规格，用于定义对象的行为和状态。

通过定义和创建对象，我们可以管理 Kubernetes 集群中的各种资源。这些资源可以是应用程序的容器、负载均衡服务、存储卷、配置信息等等。每个对象都有特定的用途和行为，可以相互关联和协同工作以构建和管理复杂的应用程序。

在实践中，我们通常使用 YAML 文件来定义和配置 Kubernetes 对象。这些文件包含对象的属性和规格，并使用 `kubectl apply` 命令将其应用到 Kubernetes 集群中。

描述 Kubernetes 对象

Kubernetes 中的对象是通过 YAML 或 JSON 文件定义的。每个对象都具有以下属性：

- **apiVersion**：指定所使用的 Kubernetes API 版本。
- **kind**：定义对象的类型，例如 Pod、Deployment、Service 等。
- **metadata**：包含对象的元数据，如名称、命名空间、标签等。
- **spec**：定义对象的规格和配置，例如容器镜像、端口、存储卷等。
- **status**：包含对象的当前状态信息，由 Kubernetes 自动更新。

可以使用 `kubectl` 命令行创建对象，也可以使用 `yaml` 格式的文件进行创建。

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    namespace: default
6    labels:
7      app: nginx
8  spec:
9    selector:
10     matchLabels:
11       app: nginx
12    replicas: 3
13    template:
14     metadata:
15       labels:
16         app: nginx
17     spec:
18     containers:
19     - name: nginx
20       image: nginx
21       ports:
22     - containerPort: 80
```

使用 Kubernetes 对象

使用 Kubernetes 对象，你可以通过将对象配置文件应用到 Kubernetes 集群来创建、更新和管理对象。你可以使用 `kubectl apply` 命令来应用配置文件。

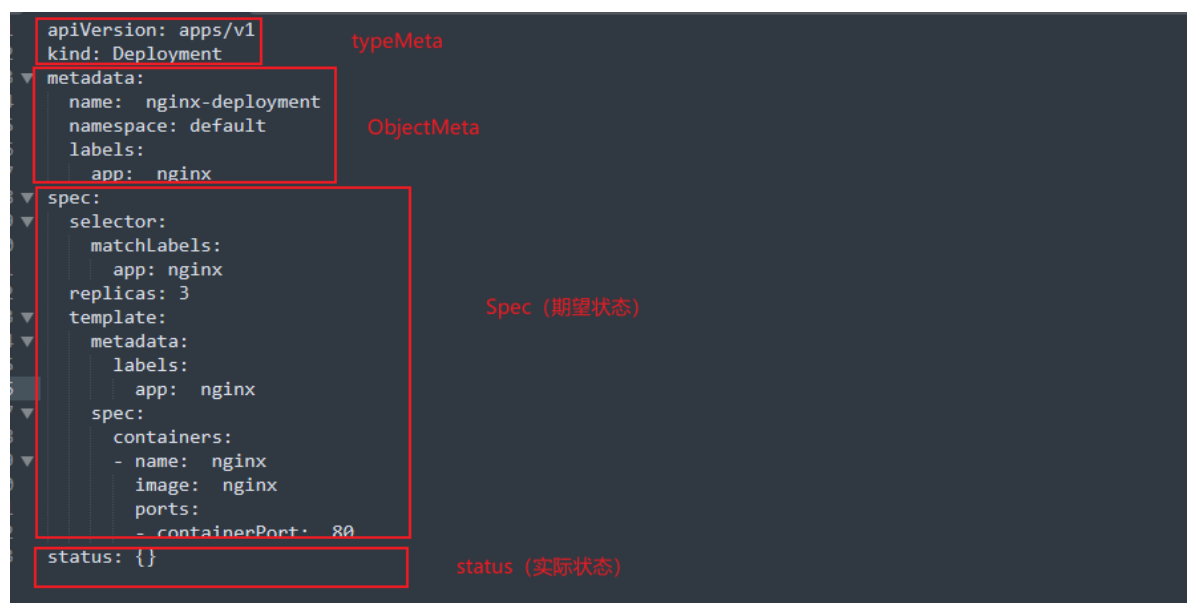
例如，要创建一个 Pod，可以使用以下命令：

```
1 kubectl apply -f pod.yaml
```

要获取现有对象的信息，可以使用以下命令：

```
1 kubectl get pods
```

Kubernetes 对象的 yaml 格式



逐个解释每个部分的含义：

- `apiVersion`：指定所使用的 Kubernetes API 版本。它定义了对对象的属性和规范的结构。例如，`v1` 表示使用的是 Kubernetes 核心 API 的版本1。
- `kind`：定义对象的类型，它指示 Kubernetes 如何处理对象。常见的对象类型包括 Pod、Service、Deployment、ConfigMap 等。
- `metadata`：包含对象的元数据，用于唯一标识和组织对象，并提供额外的信息。
 - `name`：指定对象的名称，用于在集群中唯一标识对象。
 - `namespace`：可选项，指定对象所属的命名空间。命名空间提供了资源隔离和访问控制的机制。
 - `labels`：可选项，为对象添加标签。标签是键值对的形式，用于对对象进行分类、组织和选择。
- `spec`：定义对象的规范和配置，它描述了对对象应该具有的期望状态。
- `status`：反映了对对象的当前状态，提供关于对象实际状态的信息。

根据对象的类型和用途，`spec` 部分会有不同的属性和配置。例如，在 Pod 对象中，`spec` 包含容器的配置信息，如容器镜像、端口等。而在 Deployment 对象中，`spec` 包含副本数、升级策略等配置信息。

实际中如何创建 Kubernetes 对象的 yaml

- 如果 Kubernetes 集群中已经存在了要创建的对象，那么可以使用 `kubectl get` 直接输出 yaml，然后去除 status 即可

```
1 kubectl get pod xxx -o yaml > demo.yaml
```

- 如果 Kubernetes 集群中不存在了要创建的对象，那么可以使用类似 `kubectl run xxx --dry-run=client` 输出 yaml

```
1 # --dry-run=client 用于模拟执行命令而不进行实际的修改操作
2 kubectl run nginx-pod --image=nginx --dry-run=client -o yaml > demo.yaml
```

常见的 Kubernetes 对象类型

在 Kubernetes 中，有许多不同类型的对象，用于管理和配置不同层面的资源。以下是一些常见的 Kubernetes 对象类型：

- **Pod**：Pod 是 Kubernetes 中最小的调度单位，它包含一个或多个容器。Pod 提供了容器的运行环境和资源。
- **ReplicaSet**：ReplicaSet 用于确保指定数量的 Pod 副本在集群中运行。它负责自动扩缩容和故障恢复。
- **Deployment**：Deployment 是管理应用程序部署和更新的控制器。它可以创建和管理 ReplicaSet，并提供滚动更新功能。
- **DaemonSet**：DaemonSet 用于确保在集群的每个节点上运行一个 Pod 的副本。它通常用于运行在每个节点上必要的系统服务或日志收集器。
- **StatefulSet**：StatefulSet 用于管理有状态应用程序，它为每个 Pod 提供唯一的标识和稳定的网络标识符。
- **Service**：Service 提供稳定的网络访问方式，用于将流量路由到一组具有相同标签的 Pod。它实现负载均衡和服务发现。
- **ConfigMap**：ConfigMap 用于存储应用程序的配置数据，如环境变量、配置文件等。
- **Secret**：Secret 用于安全地存储敏感信息，如密码、API 密钥等。
- **Job**：Job 是一次性任务，它创建一个或多个 Pod 来完成任务，并确保任务成功完成后自动终止。
- **CronJob**：CronJob 是定时任务，它基于 Cron 表达式创建一个 Job，用于在指定的时间间隔内执行任务。

标签、选择器

对象名称规范

Kubernetes REST API 中，所有的对象都是通过 `name` 和 `UID` 唯一性的确定。

可以通过 namespace + name 唯一性的确定一个 RESTful 对象，如：

```
1 /api/v1/namespaces/{namespace}/pods/{name}
```

Name

在同一名称空间下，同一个类型的对象，可以通过 name 来确定唯一性。如果删除该对象之后，可以重新创建一个同名对象。

根据命名规则，Kubernetes 对象的名称应该是：

- 最长不超过 253 个字符。
- 必须由小写字母、数字、减号 -、小数点 . 组成。
- 某些资源类型有更具体的要求。

示例：下面的配置文件定义了一个 name 为 nginx-demo 的 Pod，该 Pod 包含一个 name 为 nginx 的容器

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-demo ##pod的名字
5  spec: containers:
6    - name: nginx ##容器的名字
7      image: nginx:1.7.9
8      ports:
9        - containerPort: 80
```

UID

- UID 是由 Kubernetes 系统生成的，唯一标识某个 Kubernetes 对象的字符串。
- Kubernetes 集群中，每创建一个对象，都有一个唯一的 UID。用于区分多次创建的同名对象（如前面所述，按照名字删除对象后，重新再创建同名对象时，两次创建的对象 name 相同，但是 UID 不同。）

标签和选择器

当使用标签 (labels) 和选择器 (selectors) 来管理 Kubernetes 对象时，它们能够提供更好的组织和分类方式，同时也提供了更灵活的对象选择机制。下面是一些示例来帮助你更好地理解它们的作用和用法。

标签的语法

标签是一组键值对 (key/value)，标签的 key 有两个部分：可选的前缀和标签名，通过 / 分隔。

标签前缀：

- 标签前缀部分是可选的。
- 如果指定，必须是一个 DNS 的子域名，如：k8s.eip.work。
- 不能多于 253 个字符。
- 使用 / 和标签名分隔。

标签名：

- 标签名部分是必须的。
- 不能多余 63 个字符。
- 必须由字母、数字开始和结尾。
- 可以包含字母、数字、减号 (-)、下划线 (_)、小数点 (.)。

如果省略标签前缀，则标签的 key 就被认为是专属于用户的。Kubernetes 的系统组件（如：kube-scheduler、kube-controller-manager、kube-apiserver、kubectl 或其他第三方组件）向 Kubernetes 对象添加标签的时候，必须指定一个前缀。kubernetes.io/ 和 k8s.io/ 这两个前缀是 Kubernetes 核心组件预留的。

标签的 value：

- 不能多于 63 个字符。

- 可以为空字符串。
- 如果不为空，则必须由字母、数字开始和结尾。
- 如果不为空，可以包含字母、数字、减号 (-)、下划线 (_)、小数点 (.)。

两种Label Selector

常用的Label 示例如下：

- 版本标签: "version": "release", "version": "stable".....
- 环境标签: "environment": "dev", "environment": "test", "environment": "pro"
- 架构标签: "tier": "frontend", "tier": "backend"

当前有两种Label Selector：

- **基于等式的Label Selector**
 - `name = slave` 选择所有包含Label中key="name"且value="slave"的对象
 - `env != production` 选择所有包括Label中的key="env"且value不等于"production"的对象
- **基于集合的Label Selector**
 - `name in (master, slave)` 选择所有包含Label中的key="name"且value="master"或"slave"的对象
 - `name not in (frontend)` 选择所有包含Label中的key="name"且value不等于"frontend"的对象

标签的选择条件可以使用多个，此时将多个Label Selector进行组合，使用逗号","进行分隔即可。例如：

```
1 name=slave,env!=production
2 name not in (frontend),env!=production
```

标签示例

在 Kubernetes 中，你可以将标签附加到对象的元数据中，用于标识和描述对象的特征和属性。以下是一个示例，展示了如何使用标签：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5   labels:
6     app: frontend
7     environment: production
8 spec:
9   containers:
10    - name: nginx
11      image: nginx:latest
```

在这个示例中，我们为 Pod 对象添加了两个标签：`app` 和 `environment`。通过标签，我们可以轻松地对具有相同 `app=frontend` 和 `environment=production` 标签的 Pod 进行分类和组织。

Label相关命令

为pod资源打标签

```
1 kubectl label pod nginx version=1.0 -n dev
```

为pod资源更新标签

```
1 kubectl label pod nginx version=2.0 -n dev --overwrite
```

查看标签

```
1 kubectl get pod nginx -n dev --show-labels
```

筛选标签

```
1 kubectl get pod -n dev -l version=2.0 --show-labels
2 kubectl get pod -n dev -l version!=2.0 --show-labels
```

删除标签

```
1 kubectl label pod nginx version- -n dev
```

选择器示例

选择器允许你根据标签的键值对条件来选择和筛选 Kubernetes 对象。以下是一个示例，展示了如何使用选择器：

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-service
5 spec:
6   selector:
7     app: frontend
8   ports:
9     - protocol: TCP
10     port: 80
11     targetPort: 8080
```

在这个示例中，我们创建了一个名为 `my-service` 的 Service 对象，并使用选择器指定了要与该 Service 关联的 Pod。选择器 `app: frontend` 表示选择具有 `app=frontend` 标签的 Pod。

注解 annotations

注解 (annotation) 可以用来向 Kubernetes 对象的 `meta.annotations` 字段添加任意的信息。Kubernetes 的客户端或者自动化工具可以存取这些信息以实现自定义的逻辑。

```
1 metadata:
2   annotations:
3     key1: value1
4     key2: value2
```

#####

标签和选择器的好处

标签和选择器的使用有助于提高应用程序的管理和组织效率，同时也提供了更灵活的对象选择机制。以下是一些标签和选择器的好处：

- **分类和组织**：通过为对象添加标签，可以将它们分组为逻辑上相关的集合，便于管理和操作。
- **环境区分**：使用标签可以区分不同的环境，例如开发、测试和生产环境，有助于在不同环境中进行应用程序部署和管理时的识别和区分。
- **关联对象**：使用选择器可以实现对象之间的关联和联系，例如将 Service 和特定标签的 Pod 关联起来。
- **对象组合**：通过使用选择器，可以将多个标签组合起来，以选择满足特定条件的对象，从而更灵活地定义选择规则。

重新认识 kubectl 和 kubelet

- 核心文件夹：`/etc/kubernetes`。
- kubelet 额外参数配置：`/etc/sysconfig/kubelet`。
- kubelet配置位置：`/var/lib/kubelet/config.yaml`。

命名空间和资源限制

命名空间

概述

在 Kubernetes 中名称空间是用来对象资源进行隔离的。默认情况下，Kubernetes 会初始化四个名称空间：

```
1 kubectl get ns
```

- default：所有没有指定 namespace 的对象都会被分配到此名称空间中。
- kube-node-lease：Kubernetes 集群节点之间的心跳维护，V 1.13 开始引入。
- kube-system：Kubernetes 系统创建的对象放在此名称空间中。
- kube-public：此名称空间是 Kubernetes 集群安装时自动创建的，并且所有的用户都可以访问（包括未认证的用户），主要是为集群预留的，如：在某些情况中，某些 Kubernetes 对象应用应该能被所有集群用户访问到。

名称空间在实际开发中如何划分

- 基于环境隔离，如：dev（开发）、test（测试）、prod（生产）等。
- 基于产品线隔离，如：前端、后端、中间件、大数据、Android、iOS、小程序等。
- 基于团队隔离，如：企业发展事业部、技术工程事业部、云平台事业部等。

名称空间的特点

- 名称空间资源隔离、网络不隔离，如：配置文件不可以跨名称空间访问，但是网络访问可以跨名称空间访问。
- 默认情况下，安装 Kubernetes 集群的时候，会初始化一个 `default` 名称空间，用来承载那些没有指定名称空间的 Pod、Service、Deployment 等对象。

名称空间的命名规则

- 不能带小数点（.）。
- 不能带下划线（_）。
- 使用数字、小写字母或减号（-）组成的字符串。

名称空间的操作

- 示例：创建和删除名称空间（yaml）

```
1 vim k8s-namespace.yaml
```

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: demo # 名称空间的名字
5 spec: {}
6 status: {}
```

```
1 # 创建名称空间
2 kubectl apply -f k8s-namespace.yaml
```

```
1 kubectl delete -f k8s-namespace.yaml
```

- 示例：创建和删除名称空间（命令行）

```
1 # 创建名称空间
2 kubectl create ns demo
```

```
1 kubectl delete ns demo
```

- 示例：创建 Pod 的同时，指定自定义的名称空间（yaml）

```
1 vim k8s-pod.yaml
```

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: demo # 名称空间的名字
5 spec: {} # 默认为空，其实可以不写
6 status: {} # 默认为空，其实可以不写
7
8 # 以上是 namespace
9 ---
10 apiVersion: v1
11 kind: Pod
12 metadata:
13   name: nginx-pod
14   namespace: demo # 指定自定义的名称空间，如果不写，默认为 default
15   labels:
16     app: nginx
```

```
17 spec:
18   containers:
19   - name: nginx
20     image: nginx
21     resources: # 后面会讲
22       limits:
23         cpu: 200m
24         memory: 500Mi
25       requests:
26         cpu: 100m
27         memory: 200Mi
28     ports:
29     - containerPort: 80
30       name: http
31     volumeMounts:
32     - name: localtime
33       mountPath: /etc/localtime
34   volumes:
35   - name: localtime
36     hostPath:
37       path: /usr/share/zoneinfo/Asia/Shanghai
38   restartPolicy: Always
39
40   # 以上的 Pod
```

```
1 kubectl apply -f k8s-pod.yaml
```

Service 与 Pod 的 DNS

- 当创建一个 Service 的时候，Kubernetes 会创建一个相应的 [DNS 条目](#)。
- 该条目的形式是 `<service-name>.<namespace-name>.svc.cluster.local`，这意味着如果容器中只使用 `<服务名称>`，它将被解析到本地名称空间的服务器。这对于跨多个名字空间（如开发、测试和生产）使用相同的配置非常有用。如果你希望跨名字空间访问，则需要使用完全限定域名（FQDN）。

注意事项

大多数的 Kubernetes 资源（如：Pod、Service、副本控制器等）都位于某些名称空间中，但是名称空间本身并不在名称空间中，而且底层资源（如：node 和持久化卷）不属于任何命名空间。

- 查看在名称空间中的资源：

```
1 kubectl api-resources --namespaced=true
```

- 查看不在名称空间中的资源：

```
1 kubectl api-resources --namespaced=false
```

资源限制

什么是 Kubernetes 资源限制

Kubernetes 资源限制是指在 Pod 或容器级别设置的 CPU、内存、存储和网络带宽等计算资源限制。资源限制通过 Kubernetes API 对象中的资源限制字段来设置。

Kubernetes 中有哪些资源可以限制

在 Kubernetes 中，可以设置的资源限制包括：

- CPU：表示容器可以使用的 CPU 时间片，通常以 CPU 核数为单位。
- 内存：表示容器可以使用的内存量，通常以字节或者以可读性更好的单位（例如 MB 或 GB）表示。
- 存储：表示容器可以使用的存储量，可以设置存储容量上限和存储类型等参数。
- 网络带宽：表示容器可以使用的网络带宽上限。

为什么需要资源限制

在 Kubernetes 集群中，可能会有多个应用程序和服务共享同一组计算资源。如果某个应用程序或服务没有限制资源使用，它可能会占用大量的 CPU、内存或存储等资源，导致其他应用程序或服务无法正常运行。

资源限制可以帮助 Kubernetes 管理者控制容器使用的计算资源，避免资源竞争问题，从而提高集群的可靠性和稳定性。

如何设置资源限制

在 Kubernetes 中，可以通过 Pod 或容器的 YAML 文件来设置资源限制。以下是一个 Pod YAML 文件的示例，其中设置了 CPU 和内存的资源限制：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7    - name: my-container
8      image: nginx
9      resources:
10       limits:
11         cpu: "1"
12         memory: "512Mi"
13       requests:
14         cpu: "0.5"
15         memory: "256Mi"
```

在上面的 YAML 文件中，使用 `resources` 字段来设置资源限制。其中，`limits` 字段用于设置资源的上限值，`requests` 字段用于设置容器对资源的最小需求值。

如何检查资源限制是否生效

可以使用以下命令来检查 Kubernetes 中的资源限制是否生效：

```
1  kubectl describe pod <pod-name>
```

运行上述命令后，将会输出 Pod 的详细信息，其中包括容器的资源限制信息。如果资源限制设置正确并生效，将会在输出中看到类似以下的内容：

```
1 Limits:
2   cpu:      1
3   memory:   512Mi
4 Requests:
5   cpu:      500m
6   memory:   256Mi
```

这表示容器被限制在使用 1 个 CPU 核心和最多 512 MB 的内存，同时请求了至少 500m CPU 核心和 256 MB 的内存。如果资源限制设置有误或者没有生效，将会在输出中看到相应的错误或者警告信息。

容器配置与资源管理

镜像和私有镜像库的使用

下载和使用镜像

在 Kubernetes 的 Pod 定义容器的时候，必须指定容器所使用的镜像，容器中的 image 字段支持的语法和 docker 命令是一样的，包括私有镜像仓库和标签，如：

```
1 # 192.168.65.100:5000/xudaxian/ubuntu:1.0
2 my-registry.example.com:5000/example/web-example:v1.0
```

注意：在生产环境中，建议锁定镜像的版本。

- 示例：

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: demo
5 spec: {}
6 status: {}
7
8 # 以上是 namespace
9 ---
10 apiVersion: v1
11 kind: Pod
12 metadata:
13   name: nginx-pod
14   namespace: demo
15   labels:
16     app: nginx
17 spec:
18   containers:
19   - name: nginx
20     image: nginx:1.20.2 # Docker 的镜像名称，和 Docker 命令一样，my-
    registry.example.com:5000/example/web-example:v1.0，实际开发中，建议锁定镜像的版
    本。
21   ports:
22   - containerPort: 80
23
```

Kubernetes 中的镜像拉取策略：

- IfNotPresent（默认）：只有当镜像在本地不存在时才会拉取。
- Always：每当 kubelet 启动一个容器时，kubelet 会查询容器的镜像仓库，将名称解析为一个镜像摘要。如果 kubelet 有一个容器镜像，并且对应的摘要已在本地缓存，kubelet 就会使用其缓存的镜像；否则，kubelet 就会使用解析后的摘要拉取镜像，并使用该镜像来启动容器。
- Never：Kubelet 不会尝试获取镜像。如果镜像已经以某种方式存在本地，kubelet 会尝试启动容器；否则，会启动失败。

```

1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: demo
5  spec: {}
6  status: {}
7
8  # 以上是 namespace
9  ---
10 apiVersion: v1
11 kind: Pod
12 metadata:
13   name: nginx-pod
14   namespace: demo
15   labels:
16     app: nginx
17 spec:
18   containers:
19   - name: nginx
20     image: nginx:1.20.2 # Docker 的镜像名称，和 Docker 命令一样，my-
registry.example.com:5000/example/web-example:v1.0，实际开发中，建议锁定镜像的版
本。
21     imagePullPolicy: Always # 镜像拉取策略：IfNotPresent（默认）、Always、Never
22     ports:
23     - containerPort: 80
24   # 以上的 Pod

```

私有镜像库的设置和使用

使用阿里云容器镜像的私有仓库，阿里云要求进行登录，如果是 docker 拉取镜像，那么只需要 docker login 之类的就可以了；但是，如果使用 Kubernetes 该怎么办？

- 创建 secret

```

1  # -n demo：表示该密钥将只在指定的名称空间 demo 中生效
2  # docker-registry aliyun：指定 Docker 镜像仓库的名称
3  # --docker-server: Docker 镜像仓库的地址
4  # --docker-username: Docker 镜像仓库的用户名
5  # --docker-password: Docker 镜像仓库的密码
6  kubectl create secret -n demo docker-registry aliyun \
7    --docker-server=registry.cn-shanghai.aliyuncs.com \
8    --docker-username=xudaxian \
9    --docker-password=123456

```


- 在 yaml 中拉取镜像的时候设置镜像拉取的密钥 (secret)

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: demo
5  spec: {}
6  status: {}
7
8  # 以上是 namespace
9  ---
10 apiVersion: v1
11 kind: Pod
12 metadata:
13   name: demo-pod
14   namespace: demo
15   labels:
16     app: nginx
17 spec:
18   containers:
19     imagePullSecrets: # Pull 镜像时使用的 secret 名称, 以 key: secretkey 格式指定
20       - name: aliyun
21       - name: nginx
22     image: nginx:1.20.2 # Docker 的镜像名称, 和 Docker 命令一样, my-
23       registry.example.com:5000/example/web-example:v1.0, 实际开发中, 建议锁定镜像的版
24       本。
25     imagePullPolicy: Always # 镜像拉取策略: IfNotPresent (默认)、Always、Never
26     - name: arcgis
27       image: registry.cn-shanghai.aliyuncs.com/xudaxian/arcgis/v1.0
28       imagePullPolicy: Always
```

注意: 需要将 secret 的用户名和密码设置为自己的, 而且在拉取阿里云私有镜像的时候设置为自己的镜像。

环境变量和启动命令的配置

容器的环境变量配置

设置容器的环境变量: 你可以在 Pod 的配置中使用 `env` 字段设置环境变量。例如, 下面是一个 Pod 配置文件中设置环境变量的示例。

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: my-image
9        env:
10          - name: ENV_VAR_NAME
11            value: "value"
```

使用 ConfigMap 集中管理环境变量和配置：当需要管理多个环境变量或配置时，可以使用 Kubernetes 的 ConfigMap 对象来集中管理。你可以创建一个 ConfigMap，并在 Pod 的配置中引用该 ConfigMap，从而将多个环境变量传递给容器。

启动命令的定制化

Docker 的镜像拥有存储镜像信息的相关元数据，如果不设置生命周期命令和参数，容器运行时会运行镜像制作时提供的默认的命令和参数，Docker 原生定义这两个字段为 `ENTRYPOINT` 和 `CMD`。

如果在创建工作负载时填写了容器的运行命令和参数，将会覆盖镜像构建时的默认命令 `Entrypoint`、`CMD`，规则如下：

镜像 Entrypoint	镜像CMD	容器 command	容器 args	最终执行
[touch]	[/root/test]	未设置	未设置	[touch /root/test]
[touch]	[/root/test]	[mkdir]	未设置	[mkdir]
[touch]	[/root/test]	未设置	[/opt/test]	[touch /opt/test]
[touch]	[/root/test]	[mkdir]	[/opt/test]	[mkdir /opt/test]

换言之，如果在 Kubernetes 的 yaml 中定义了 `comand` 和 `args`，那么就会覆盖 Dockerfile 中的 `ENTRPOINT` 和 `CMD`。

例如，下面是一个 Pod 配置文件中定制启动命令的示例：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: my-image
9        command: ["echo"]
10       args: ["Hello, kubernetes!"]
```

容器资源限额管理和配额设置

资源限额管理是 Kubernetes 中重要的一部分，它可以帮助你控制和管理容器的资源使用量。下面我将为你重新讲解资源限额管理，并提供具体的示例来帮助你更好地理解。

Pod 资源限额

容器中的程序要运行，肯定会占用一定的资源，比如 CPU 和内存等，如果不对某个容器的资源做限制，那么它就可能吃掉大量的资源，导致其他的容器无法运行。

针对上面的情况，Kubernetes 提供了对内存和 CPU 的资源进行配额的机制，这种机制主要通过 `resources` 选项实现，它有两个子选项：

- `limits`：用于限制运行的容器的最大占用资源，当容器占用资源超过 `limits` 时会被终止，并进行重启。
- `requests`：用于设置容器需要的最小资源，如果环境资源不够，容器将无法启动。

注意，对于资源限制的配置，在 YAML 文件中，将数值用引号包裹是可选的。因此，可以选择是否在 `cpu` 的值周围使用引号。

下面是一个示例 Pod 配置，展示了如何设置 CPU 和内存的限额和请求：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: my-image
9        resources:
10         limits:
11           cpu: "2"
12           memory: "2Gi"
13         requests:
14           cpu: "1"
15           memory: "1Gi"
```

在上述示例中，我们为 Pod 中的容器设置了以下资源限额和请求：

- `limits` 定义了容器的资源限制，即容器最多可以使用的资源量。
 - `cpu` 限制为 2 个核心。
 - `memory` 限制为 2GiB 内存。
- `requests` 定义了容器对资源的请求，即期望分配的资源量。
 - `cpu` 请求为 1 个核心。
 - `memory` 请求为 1GiB 内存。

通过设置这些限额和请求，Kubernetes 可以根据资源需求进行资源分配和调度，以确保各个容器在集群中得到适当的资源分配。

命名空间资源配额

除了针对单个 Pod 的资源限额，你还可以为命名空间设置资源配额，以控制整个命名空间中的资源使用量。

示例: 设置命名空间的资源配额

```
1  apiVersion: v1
2  kind: ResourceQuota
3  metadata:
4    name: my-namespace-quota
5  spec:
6    hard:
7      cpu: "2"
8      memory: 2Gi
```

在上述示例中，我们创建了一个 ResourceQuota 对象来设置命名空间的资源配额。这个配额限制了该命名空间中的总 CPU 使用量为 2 个核心，总内存使用量为 2GiB。

常见 Kubernetes 操作和配置

指定输出格式

kubernetes支持的格式有很多，比较常见的是wide、json、yaml。

- 通过 `-o` 格式参数 命令使用

```
1 | kubectl get ns default -o yaml
```

命名空间操作

在 Kubernetes 中，命名空间用于将集群划分为多个虚拟环境，帮助进行资源隔离和管理。

默认创建的namespace

NAME	STATUS	AGE	
default	Active	45h	所有未指定Namespace的对象都会被分配在default命名空间
kube-node-lease	Active	45h	集群节点之间的心跳维护，v1.13开始引入
kube-public	Active	45h	此命名空间下的资源可以被所有人访问（包括未认证用户）
kube-system	Active	45h	所有由Kubernetes系统创建的资源都处于这个命名空间

注意，后续的 namespace 等价于 ns。

创建和查看命名空间

使用命令或配置文件创建命名空间，并查看已存在的命名空间。示例：

```
1 | # 创建命名空间
2 | kubectl create namespace my-namespace
3 |
4 | # 查看命名空间
5 | kubectl get namespaces
6 | # kubectl get namespace
7 | # kubectl get ns
```

切换命名空间

在操作时切换当前所使用的命名空间。示例：

```
1 | # 切换命名空间
2 | kubectl config set-context --current --namespace=my-namespace
```

删除命名空间

删除不再需要的命名空间。示例：

```
1 # 删除命名空间
2 kubectl delete namespace my-namespace
```

默认命名空间

Kubernetes 还提供了一个默认的命名空间，默认情况下，所有未指定命名空间的资源都会被放置在默认命名空间中。示例：

```
1 # 查看默认命名空间
2 kubectl config view --minify --output 'jsonpath={..namespace}'
```

查看命名空间日志

查看指定命名空间详细信息。示例：

```
1 kubectl describe namespace my-namespace
```

YAML 配置信息

```
1 # ns-dev.yaml
2 apiVersion: v1
3 kind: Namespace
4 metadata:
5   name: dev
```

Pod 操作

Pod是kubernetes集群进行管理的最小单元，程序要运行必须部署在容器中，而容器必须存在于Pod中。

Pod可以认为是容器的封装，一个Pod中可以存在一个或者多个容器。

创建和查看 Pod

使用配置文件或命令创建 Pod，并查看已存在的 Pod。示例：

```
1 # 创建 Pod，配置看后面
2 kubectl create -f pod.yaml
3
4 # 查看 所有Pod
5 kubectl get pods
6
7 # 等价于，也就是当前空间
8 kubectl get pods -n default
9
10 # 如果想看其他空间的 Pod
11 kubectl get pods -n my-namespace
12
13 # 查看指定 Pod
14 kubectl get pod nginx -n dev
15 kubectl get pod/nginx -n dev
```

```
16
17 # 创建nginx
18 kubectl run nginx --image=nginx:latest --port=80
```

删除 Pod

根据名称或标识符删除 Pod。示例：

```
1 # 删除 Pod
2 kubectl delete pod nginx
```

查看 Pod 日志

查看 Pod 的日志输出。示例：

```
1 # 查看 Pod 日志
2 kubectl logs nginx
```

默认 Pod 命名空间

如果在创建 Pod 时未指定命名空间，Pod 将被创建在默认命名空间中。示例：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx
5 spec:
6   containers:
7     - name: nginx
8       image: nginx
```

描述 Pod

使用 `kubectl describe` 命令查看 Pod 的详细信息，包括 Pod 的状态、事件和关联的资源。示例：

```
1 # 描述 Pod
2 kubectl describe pod nginx
```

查询一下指定namespace下的Pod控制器

```
1 kubectl get deploy -n dev
```

进入 Pod 容器内部

- 使用 `kubectl exec` 命令

```
1 kubectl exec -it <pod-name> -n <namespace> -- <command>
```

将 `<pod-name>` 替换为您要进入的 Pod 的名称，`<namespace>` 替换为 Pod 所在的命名空间，`<command>` 替换为要在容器内部执行的命令。这将在 Pod 中找到一个容器，并在其中执行指定的命令。

- 使用 `kubectl exec` 命令并指定容器名称

```
1 | kubectl exec -it <pod-name> -c <container-name> -n <namespace> -- <command>
```

除了上述命令，还需要指定 `-c <container-name>` 来明确要进入的容器名称，**特别是当 Pod 中有多个容器时**。

在这两种方式中，`-it` 选项用于以交互模式进入容器，即允许与容器进行交互。可以在容器内部执行命令，例如 `bash` 或 `sh`，以进入容器的命令行终端。

注意，请确保将 `<pod-name>` 替换为您实际的 Pod 名称，`<namespace>` 替换为适当的命名空间，`<command>` 替换为要执行的命令。

- 使用 `kubectl attach` 命令

```
1 | kubectl attach -it <pod-name> -c <container-name> -n <namespace>
```

这个命令会将当前终端附加到正在运行的容器中，类似于通过 SSH 连接到远程服务器。可以与容器交互并查看容器的输出。

- 使用 `kubectl run` 命令：

```
1 | kubectl run -it --rm <temp-pod-name> --image=<container-image> -n <namespace> -- <command>
```

这个命令会创建一个临时的 Pod 并进入其容器。可以指定一个临时 Pod 的名称 `<temp-pod-name>`、容器镜像 `<container-image>` 和要执行的命令 `<command>`。

YAML 配置信息

```
1 | # pod.yaml
2 | apiVersion: v1
3 | kind: Pod
4 | metadata:
5 |   name: nginx
6 |   namespace: dev
7 | spec:
8 |   containers:
9 |   - image: nginx:latest
10 |     name: pod
11 |     ports:
12 |     - name: nginx-port
13 |       containerPort: 80
14 |       protocol: TCP
```

部署 (Deployment) 操作

在kubernetes中，Pod是最小的控制单元，但是kubernetes很少直接控制Pod，**一般都是通过Pod控制器来完成的**。Pod控制器用于pod的管理，确保pod资源符合预期的状态，当pod的资源出现故障时，会尝试进行重启或重建pod。

注意，后续的 deployment 等价于 deploy。

创建 Deployment

可以通过使用配置文件或命令创建 Deployment，这里使用命令创建。示例：

```
1 kubectl create deploy nginx --image=nginx:latest --port=80 --replicas=3 -n dev
```

- `--image`：指定pod的镜像
- `--port`：指定端口
- `--replicas`：指定创建pod数量
- `--namespace/-n`：指定namespace

查看 Deployment

查看已存在的 Deployment。示例：

```
1 kubectl get pods -n dev
```

更新 Deployment

修改 Deployment 的配置，如镜像版本或副本数量。示例：

```
1 # 更新 Deployment 镜像版本
2 kubectl set image deployment/nginx nginx=redis -n dev
3
4 # 等价于 ==> deployment/nginx ==> deployment nginx
5 kubectl set image deployment nginx nginx=redis -n dev
6
7 # 等价于 ==> deployment ==> deploy
8 kubectl set image deploy nginx nginx=redis -n dev
9
10 # kubectl set image deployment/my-deployment my-container=my-image:2.0
```

扩缩 Deployment

修改 Deployment 的配置，如镜像版本或副本数量。示例：

```
1 # 扩展/收缩 Deployment
2 kubectl scale deployment/nginx --replicas=2 -n dev
3
4 # kubectl scale deployment/my-deployment --replicas=3
```

描述 Deployment

查看 Deployment 的配置。示例：

```
1 kubectl describe deployment/nginx -n dev
```

删除 Deployment

删除 Deployment。示例：

```
1 kubectl delete deploy nginx -n dev
```


YAML 配置信息

```
1 # deploy-nginx.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: nginx
6   namespace: dev
7
8 spec:
9   replicas: 3
10  selector:
11    matchLabels:
12      run: nginx
13
14  template:
15    metadata:
16      labels:
17        run: nginx
18    spec:
19      containers:
20      - image: nginx:latest
21        name: nginx
22        ports:
23        - containerPort: 80
24          protocol: TCP
```

服务 (Service) 操作

服务是 Kubernetes 中用于提供稳定网络访问的抽象层，借助Service应用可以方便地实现服务发现和负载均衡。

创建和查看服务

使用配置文件或命令创建服务，并查看已存在的服务。示例：

```
1 # 创建服务
2 kubectl create -f service.yaml
3
4 # 查看服务
5 kubectl get services
6
7 # 等价于
8 kubectl get svc
```

更新服务

修改服务的配置，如端口映射或负载均衡器设置。示例：

```
1 # 更新服务
2 kubectl apply -f service.yaml
```

删除服务

根据名称或标识符删除服务。示例：

```
1 # 删除服务
2 kubectl delete service my-service
```

集群内部访问服务

```
1 # 暴露Service
2 kubectl expose deploy nginx --name=svc-nginx1 --type=ClusterIP --port=80 --
  target-port=80 -n dev
3
4 # 查看service
5 kubectl get svc -n dev -o wide
6
7 # 这里产生了一个CLUSTER-IP，这就是service的IP，在Service的生命周期中，这个地址是不会变
  动的
8 # 可以通过这个IP访问当前service对应的POD
9 curl 10.233.43.85:80
```

外部访问服务

```
1 # 上面创建的Service的type类型为ClusterIP，这个ip地址只用集群内部可访问
2 # 如果需要创建外部也可以访问的Service，需要修改type为NodePort
3 kubectl expose deploy nginx --name=svc-nginx2 --type=NodePort --port=80 --
  target-port=80 -n dev
4
5 # 此时查看，会发现出现了NodePort类型的Service，而且有一对Port（80:31790/TCP）
6 kubectl get svc svc-nginx2 -n dev -o wide
7
8 # 接下来就可以通过集群外的主机访问 节点IP:31790访问服务了
9 # 例如在的电脑主机上通过浏览器访问下面的地址
10 http://10.39.177.71:31790/ # http://主机IP:31790/
11
12 #删除Service
13 kubectl delete svc svc-nginx1 -n dev
```

YAML 配置信息

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: svc-nginx
5   namespace: dev
6 spec:
7   clusterIP: 10.233.43.85 #固定svc的内网ip
8   ports:
9     - port: 80
10     protocol: TCP
11     targetPort: 80
12   selector:
13     run: nginx
14   type: ClusterIP
```

常用命令文档

kubectll命令的对应关系：

命令	简写	含义
deployment	deploy	部署 (Deployment)
pod	po	Pod
service	svc	服务 (Service)
replica set	rs	副本集 (ReplicaSet)
daemon set	ds	守护进程集 (DaemonSet)
stateful set	sts	有状态副本集 (StatefulSet)
job	-	作业 (Job)
cron job	cj	定时任务 (CronJob)
config map	cm	配置映射 (ConfigMap)
secret	-	密文 (Secret)
ingress	-	入口 (Ingress)
namespace	ns	命名空间 (Namespace)
persistent volume	pv	持久卷 (PersistentVolume)
persistent volume claim	pvc	持久卷声明 (PersistentVolumeClaim)
node	no	节点 (Node)
service account	sa	服务账户 (ServiceAccount)
role	-	角色 (Role)
role binding	-	角色绑定 (RoleBinding)
cluster role	-	集群角色 (ClusterRole)
cluster role binding	-	集群角色绑定 (ClusterRoleBinding)
context	-	上下文 (Context)
component status	cs	组件状态 (ComponentStatus)
event	ev	事件 (Event)

一些常用的kubectll命令的表格形式呈现，包括语法、含义和参数含义：

命令	语法	含义	参数含义
----	----	----	------

命令	语法	含义	参数含义
kubectl get	<code>kubectl get <资源类型></code>	获取指定类型的资源列表	<code><资源类型></code> ：要获取的资源类型，例如 pods、services、deployments 等
kubectl describe	<code>kubectl describe <资源类型> <资源名称></code>	显示指定资源的详细信息	<code><资源类型></code> ：要描述的资源类型，例如 pods、services、deployments 等； <code><资源名称></code> ：要描述的资源名称
kubectl create	<code>kubectl create <资源类型> <资源配置></code>	创建指定类型的资源	<code><资源类型></code> ：要创建的资源类型，例如 pod、service、deployment 等； <code><资源配置></code> ：要使用的配置文件或配置参数
kubectl apply	<code>kubectl apply -f <文件路径></code>	应用指定的配置文件来创建或更新资源	<code>-f <文件路径></code> ：指定要应用的配置文件的路径
kubectl delete	<code>kubectl delete <资源类型> <资源名称></code>	删除指定的资源	<code><资源类型></code> ：要删除的资源类型，例如 pod、service、deployment 等； <code><资源名称></code> ：要删除的资源名称
kubectl edit	<code>kubectl edit <资源类型> <资源名称></code>	编辑指定资源的配置	<code><资源类型></code> ：要编辑的资源类型，例如 pod、service、deployment 等； <code><资源名称></code> ：要编辑的资源名称
kubectl exec	<code>kubectl exec -it <pod名称> -- <命令></code>	在指定 Pod 内的容器中执行命令	<code>-it <pod名称></code> ：要执行命令的 Pod 的名称； <code>-- <命令></code> ：要在容器内部执行的命令
kubectl logs	<code>kubectl logs <pod名称></code>	获取指定 Pod 的日志	<code><pod名称></code> ：要获取日志的 Pod 的名称
kubectl port-forward	<code>kubectl port-forward <pod名称> <本地端口>:<远程端口></code>	将本地端口转发到 Pod 的端口	<code><pod名称></code> ：要进行端口转发的 Pod 的名称； <code><本地端口>:<远程端口></code> ：本地端口和 Pod 端口之间的映射关系
kubectl scale	<code>kubectl scale <资源类型>/<资源名称> -- replicas=<副本数量></code>	调整部署的副本数量	<code><资源类型>/<资源名称></code> ：要调整副本数量的资源的类型和名称； <code>--replicas=<副本数量></code> ：要调整的副本数量
kubectl rollout	<code>kubectl rollout status <资源类型>/<资源名称></code>	显示部署的滚动更新状态	<code><资源类型>/<资源名称></code> ：要显示滚动更新状态的部署的类型和名称

命令	语法	含义	参数含义
kubectl expose	<code>kubectl expose</code> <资源类型> <资源名称> --port=<端口号>	为资源创建服务	<资源类型>：要创建服务的资源类型，例如 deployment、pod 等；<资源名称>：要创建服务的资源的名称；--port=<端口号>：服务的端口号
kubectl get namespaces	<code>kubectl get namespaces</code>	获取所有命名空间的列表	-
kubectl config use-context	<code>kubectl config use-context</code> <上下文名称>	切换到指定的上下文	<上下文名称>：要切换到的上下文的名称

请注意，这里列出的命令只是一些常见的kubectl命令示例，还有更多的kubectl命令和选项可用。可以通过运行 `kubectl --help` 命令或访问 [Kubernetes 官方文档](#) 了解更多详细信息和命令选项。

核心篇：构建高可用、高性能的容器化平台

Pod：最小调度单位

Pod 概述

Pod 是 Kubernetes 中的基本调度单位，它是一个可以包含一个或多个相关容器的组合。在 Pod 内部的容器共享相同的网络命名空间和存储卷，它们可以通过本地主机上的 IPC（进程间通信）机制进行通信。Pod 提供了一种在容器之间共享资源和紧密协作的方式。

什么是Pod

Pod 是一组相关容器的集合，它们共享相同的网络命名空间和存储卷。Pod 是 Kubernetes 调度器所管理的最小调度单位，它表示在同一个主机上运行的一组容器。

Pod 的作用

Pod 的主要作用是将相关的容器组合在一起，以便它们可以共享资源、协同工作和共享网络。Pod 提供了一个抽象层，将多个容器作为一个单元进行管理，从而方便了容器间的通信和协作。

Pod 的结构定义

Pod 的定义包括以下几个主要部分：

- metadata：用于描述 Pod 的元数据信息，如名称、命名空间、标签等。
- spec：定义 Pod 中的容器和相关配置信息，如容器镜像、端口映射、环境变量等。
- status：表示当前 Pod 的状态信息，如运行状态、IP 地址、重启次数等。

查看 pod定义信息

```

1 #小提示:
2 # 在这里, 可通过一个命令来查看每种资源的可配置项
3 # kubectl explain 资源类型          查看某种资源可以配置的一级属性
4 # kubectl explain 资源类型.属性      查看属性的子属性
5 kubectl explain pod
6 # apiVersion <string>
7 # kind <string>
8 # metadata <Object>
9 # spec <Object>
10 # status <Object>
11 kubectl explain pod.metadata

```

示例：多容器协同

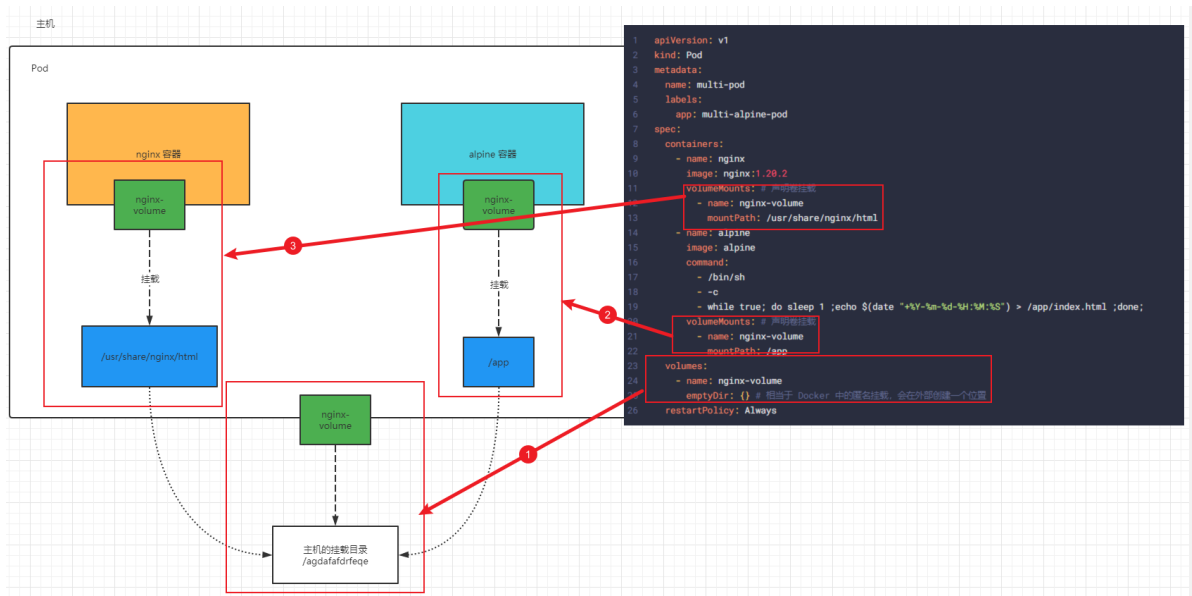
下面是一个示例，展示了一个 Pod 中包含多个协同工作的容器：

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: multi-pod
5   labels:
6     app: multi-alpine-pod
7 spec:
8   containers:
9     - name: nginx
10       image: nginx:1.20.2
11       volumeMounts: # 声明卷挂载
12         - name: nginx-volume
13           mountPath: /usr/share/nginx/html
14     - name: alpine
15       image: alpine
16       command:
17         - /bin/sh
18         - -c
19         - while true; do sleep 1 ;echo $(date "+%Y-%m-%d-%H:%M:%S") >
/app/index.html ;done;
20       volumeMounts: # 声明卷挂载
21         - name: nginx-volume
22           mountPath: /app
23   volumes:
24     - name: nginx-volume
25       emptyDir: {} # 相当于 Docker 中的匿名挂载, 会在外部创建一个位置
26   restartPolicy: Always

```

在上面的示例中，Pod `multi-pod` 包含了两个容器：`nginx` 和 `alpine`。



温馨提示：如果想进多容器协同的 Pod 中查看指定容器，使用这样的命令：`kubectl exec -it Pod的名称 -c Pod中的容器名 -- COMMAND`。

初始化容器

什么是初始化容器

初始化容器是在 Pod 的主容器启动之前运行的一种特殊类型的容器。它用于执行一些预加载任务或准备工作，以确保主容器在启动时能够顺利运行。

初始化容器的作用

初始化容器在Kubernetes中扮演着重要的角色，其作用包括但不限于以下几个方面：

- 预加载依赖项：**初始化容器可以用于下载或准备应用程序所需的依赖项，以确保主容器在启动时能够顺利运行。例如，可以使用初始化容器下载配置文件、静态文件、数据库初始化脚本等。
- 数据库初始化：**如果你的应用程序依赖于数据库，初始化容器可以用于初始化数据库并确保数据库的正确配置和状态。它可以在主容器启动之前，确保数据库的准备工作已经完成，主容器可以正常连接和使用数据库。
- 环境配置：**有时候，应用程序可能需要在启动之前进行一些环境配置，例如加载环境变量、生成配置文件等。初始化容器可以负责执行这些环境配置任务，确保主容器在启动时拥有正确的环境。
- 启动顺序控制：**如果你有多个容器之间存在启动顺序的依赖关系，初始化容器可以用来控制容器的启动顺序。通过定义多个初始化容器，并设置它们的顺序，你可以确保容器按照指定的顺序依次启动，从而满足启动顺序的要求。

初始化容器的两大特征

- 必须运行完成直至结束：**初始化容器必须运行成功，如果某个初始化容器运行失败，Kubernetes会自动重启它，直至成功完成。
- 必须按照定义的顺序执行：**在一个 Pod 的配置中，可以定义多个初始化容器，并且它们按照顺序依次执行。每个初始化容器必须等待前一个初始化容器成功完成之后才能开始运行。

示例：使用初始化容器进行依赖项预加载

在下面的示例中，我们将展示如何使用初始化容器进行依赖项预加载。

- 创建一个包含初始化容器的 Pod 配置文件（例如，`init-container.yaml`）：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp
5  spec:
6    initContainers:
7      - name: setup
8        image: busybox
9        command: ['sh', '-c', 'wget http://example.com/data.tgz']
10   containers:
11     - name: mainapp
12       image: myapp:latest
13       # 主容器的配置...

```

- 应用配置文件以创建 Pod：

```
1 | kubectl apply -f init-container.yaml
```

在上面的示例中，我们创建了一个名为 `setup` 的初始化容器，它使用 `busybox` 镜像并执行命令 `wget http://example.com/data.tgz`，从而下载 `data.tgz` 文件作为主应用程序的依赖项。

这样，当 Pod 启动时，初始化容器会在主容器之前执行，并完成依赖项的预加载。

临时容器

临时容器是一种临时性的容器实例，用于在已运行的 Pod 内部执行命令、调试或获取信息，而无需直接修改 Pod 的配置。

什么是临时容器

临时容器是在已运行的 Pod 内部临时启动的容器。它与 Pod 共享相同的网络和存储空间，可以在 Pod 内部执行命令、查看日志文件、调试应用程序等操作，而无需直接修改 Pod 的配置或重新启动 Pod。

临时容器的用途

当由于容器奔溃或容器镜像不包含调试工具而导致 `kubectl exec` 无用的时候，临时容器对于交互式故障排查非常有用。

比如，像 `distroless` 镜像允许用户部署最小的容器镜像，从而减少攻击面并减少故障和漏洞的暴露。由于 `distroless` 镜像不包含 Shell 或任何的调试工具，因此很难单独使用 `kubectl exec` 命令进行故障排查。

使用临时容器的时候，启用 [进程名字空间共享](#) 很有帮助，可以查看其他容器中的进程。

开启临时容器

- 查询临时容器是否开启

```
1 | kubelet -h | grep EphemeralContainers
```

```
[root@master1 busybox-1.28.3]# kubelet -h | grep EphemeralContainers
EphemeralContainers=true|false (ALPHA - default=false)
```

默认是关闭的。

- 修改 `kubectl` 的参数


```
1 | vi /etc/sysconfig/kubelet
```

```
1 | # 修改增加--feature-gates EphemeralContainers=true
2 | KUBELET_EXTRA_ARGS="--cgroup-driver=systemd --feature-gates
   | EphemeralContainers=true"
3 | KUBE_PROXY_MODE="ipvs"
```

```
1 | vi /var/lib/kubelet/config.yaml
```

```
1 | # 修改部分
2 | featureGates:
3 |   EphemeralContainers: true
```

- 加载配置文件以便重启 kubelet

```
1 | systemctl daemon-reload
2 |
3 | systemctl stop kubelet
4 |
5 | systemctl start kubelet
```

- 在 Master 节点 修改 kube-apiserver.yaml 和 kube-scheduler.yaml

```
1 | vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

```
1 | apiVersion: v1
2 | kind: Pod
3 | metadata:
4 |   annotations:
5 |     kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint:
192.168.65.100:6443
6 |   creationTimestamp: null
7 |   labels:
8 |     component: kube-apiserver
9 |     tier: control-plane
10 |   name: kube-apiserver
11 |   namespace: kube-system
12 | spec:
13 |   containers:
14 |   - command:
15 |     - kube-apiserver
16 |     ...
17 |     - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
18 |     - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
19 |     # 修改部分
20 |     - --feature-gates=EphemeralContainers=true
```

```
1 | vi /etc/kubernetes/manifests/kube-scheduler.yaml
```

```
1 | apiVersion: v1
```

```

2  kind: Pod
3  metadata:
4    creationTimestamp: null
5    labels:
6      component: kube-scheduler
7      tier: control-plane
8    name: kube-scheduler
9    namespace: kube-system
10 spec:
11   containers:
12   - command:
13     - kube-scheduler
14     - --authentication-kubeconfig=/etc/kubernetes/scheduler.conf
15     - --authorization-kubeconfig=/etc/kubernetes/scheduler.conf
16     - --bind-address=127.0.0.1
17     - --kubeconfig=/etc/kubernetes/scheduler.conf
18     - --leader-elect=true
19     - --port=0
20     # 修改部分
21     - --feature-gates=EphemeralContainers=true

```

示例: debug 调试

- 创建一个 Pod

```
1 vim k8s-pod.yaml
```

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    shareProcessNamespace: true # 这个配置非常重要，一定要配置
7    containers:
8    - name: nginx
9      image: nginx:1.17.1

```

```
1 kubectl apply -f k8s-pod.yaml
```

- 创建临时容器

```
1 kubectl debug -it nginx --image=busybox:1.28
```

- 查看nginx这个pod是否已经有临时容器

```
1 kubectl describe pods nginx
```

或者通过如下

- 创建 ec.json 文件，内容如下

```
1 vi ec.json
```

```

1 {
2   "apiVersion": "v1",
3   "kind": "EphemeralContainers",
4   "metadata": {
5     "name": "nginx"
6   },
7   "ephemeralContainers": [{
8     "command": [
9       "sh"
10    ],
11    "image": "busybox",
12    "imagePullPolicy": "IfNotPresent",
13    "name": "debugger",
14    "stdin": true,
15    "tty": true,
16    "targetContainerName": "nginx",
17    "terminationMessagePolicy": "File"
18  }]
19 }

```

注意: json 文件目前是没有注释的。

- 应用临时容器

```

1 # /api/v1/namespaces/default/pods/[pod的名称]/ephemeralcontainers
2 kubectl replace --raw
  /api/v1/namespaces/default/pods/nginx/ephemeralcontainers -f ec.json

```

- 进入到临时容器上去

```

1 kubectl debug nginx --image=nginx --share-processes --copy-to=myapp-debug
2
3 kubectl exec -it myapp-debug bash

```

重启策略

重启策略的概念

重启策略定义了当 Pod 的容器退出后, Kubernetes 如何处理容器的重启。它确定了容器在退出后是否应该被自动重启, 并且在何种情况下应该进行重启。

重启策略的类型

重启策略是 Pod 配置中的一个重要概念。它用于定义容器退出后的处理方式。Kubernetes 提供了三种重启策略:

1. Always (始终重启): 当容器退出时, Kubernetes 会自动重启容器。这是**默认**的重启策略。
2. OnFailure (仅在失败时重启): 当容器以非零状态退出时, Kubernetes 会自动重启容器。
3. Never (不重启): 当容器退出时, Kubernetes 不会自动重启容器。如果需要重新启动, 需要手动进行操作。

重启策略适用于 Pod 对象中的所有容器, 首次需要重启的容器, 将在其需要的时候立即进行重启, 随后再次重启的操作将由 kubelet 延迟一段时间后进行, 且反复的重启操作的延迟时长以此为 10s、20s、40s、80s、160s 和 300s, 300s 是最大的延迟时长。

示例：处理故障和重启策略的设置

以下是一个示例，展示如何设置 Pod 的重启策略以处理故障：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp
5  spec:
6    restartPolicy: OnFailure
7    containers:
8      - name: myapp-container
9        image: myapp-image
```

在上述示例中，重启策略被设置为 `OnFailure`，这意味着当容器以非零状态退出时，Kubernetes 会自动重启容器。

通过设置适当的重启策略，可以确保在容器故障时自动进行重启，提高应用程序的可靠性和稳定性。

钩子函数

钩子函数的概念

钩子函数是一种在 Pod 生命周期中执行特定操作的机制。它允许在容器的生命周期事件发生时运行预定义的脚本或命令。

钩子函数的类型

Kubernetes 提供了两种类型的钩子函数：

- `PostStart`（启动后）钩子函数：在容器启动后立即执行的操作。它适用于在容器启动后进行一些初始化配置、数据准备或启动应用程序等操作。
- `PreStop`（停止前）钩子函数：在容器停止之前执行的操作。它适用于在容器停止之前进行一些清理工作、保存状态或停止应用程序等操作。

钩子处理器的定义方式

钩子处理器支持以下三种方式定义动作：

- `ExecAction`：使用 `exec` 执行命令的方式来定义动作。可以在钩子处理器中指定要执行的命令及参数。

```
1  lifecycle:
2    postStart:
3      exec:
4        command: ["/bin/sh", "-c", "echo 'PostStart hook executed!'"]
5    preStop:
6      exec:
7        command: ["/bin/sh", "-c", "echo 'PreStop hook executed!'"]
```

上述示例中，`postStart` 钩子函数使用 `exec` 方式定义了一个动作，在容器启动后执行 `/bin/sh -c "echo 'PostStart hook executed!'"` 命令。`preStop` 钩子函数也使用 `exec` 方式定义了一个动作，在容器停止前执行 `/bin/sh -c "echo 'PreStop hook executed!'"` 命令。

- `HTTPGetAction`：使用 `HTTP GET` 请求的方式来定义动作。可以指定要访问的 URL，并定义期望的 HTTP 状态码。

```

1  lifecycle:
2    postStart:
3      httpGet:
4        path: /health
5        port: 8080
6        httpHeaders:
7          - name: Authorization
8            value: Bearer <token>
9        initialDelaySeconds: 5
10       periodSeconds: 10

```

上述示例中，`postStart` 钩子函数使用 `HTTPGetAction` 方式定义了一个动作，每隔 10 秒钟发送一个 HTTP GET 请求到 `/health` 路径，端口为 8080，并带有自定义的 HTTP 请求头。可以根据需要修改路径、端口、请求头等配置。

- `TCPSocketAction`：使用 `TCP` 套接字的方式来定义动作。可以指定要连接的主机和端口。

```

1  lifecycle:
2    postStart:
3      tcpSocket:
4        host: localhost
5        port: 5432

```

上述示例中，`postStart` 钩子函数使用 `TCPSocketAction` 方式定义了一个动作，尝试连接到 `localhost` 主机的 5432 端口。

示例：在 Pod 生命周期中运行脚本

以下是一个示例，展示如何在 Pod 生命周期中使用钩子函数运行脚本：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp
5  spec:
6    containers:
7      - name: myapp-container
8        image: myapp-image
9        lifecycle:
10       postStart:
11         exec:
12           command:
13             - /bin/sh
14             - -c
15             - echo 'Container started!'
16       preStop:
17         exec:
18           command:
19             - /bin/sh
20             - -c
21             - echo 'Container stopping!'

```

在上述示例中，`postStart` 钩子函数使用 `exec` 方式定义了一个动作，即在容器启动后执行一个命令，输出 "Container started!"。 `preStop` 钩子函数也使用 `exec` 方式定义了一个动作，即在容器停止之前执行一个命令，输出 "Container stopping!"。

探针机制

什么是探针机制

探针机制是一种用于监测容器内应用程序的健康状态的机制。通过定期发送请求或执行命令，探针可以判断应用程序是否正常运行，并根据结果采取相应的操作。

探针机制的类型

在 Kubernetes 中，有三种类型的探针：

1. 存活探针 (Liveness Probe)：用于检测容器是否仍然运行，并且在容器不可用时触发自动重启。
2. 就绪探针 (Readiness Probe)：用于检测容器是否已经准备好接收流量，并且在容器未准备好时阻止将流量转发到该容器。
3. 启动探针 (Startup Probe)：用于检测容器是否已经启动完成并准备好接收流量。与就绪探针不同的是，启动探针只在容器启动期间进行检测，一旦探测成功，则不再进行后续的检测。

探针机制的配置项

在 Pod 的配置中，可以使用以下字段来定义探针的配置项：

- `initialDelaySeconds`：探针开始检查之前的初始延迟时间，默认值为 0。
- `periodSeconds`：两次连续探测之间的时间间隔，默认值为 10。
- `timeoutSeconds`：探测超时时间，如果在此时间内未收到响应，则认为探测失败，默认值为 1。
- `successThreshold`：连续成功探测的阈值，达到该阈值则认为探测成功，默认值为 1。
- `failureThreshold`：连续失败探测的阈值，达到该阈值则认为探测失败，默认值为 3。

探针机制的探测方式

在 Kubernetes 中，探针机制可以使用以下探测方式（类似钩子函数）：

1. HTTP 探测：通过发送 HTTP 请求并检查响应的状态码来判断应用程序的健康状态。
2. EXEC 探测：通过在容器内执行命令并检查命令的退出代码来判断应用程序的健康状态。
3. TCP 探测：通过建立 TCP 连接并检查连接是否成功来判断应用程序的健康状态。

示例：使用探针机制监测应用程序健康状态

以下是一个示例，展示如何使用探针机制来监测应用程序的健康状态：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp
5  spec:
6    containers:
7      - name: myapp-container
8        image: myapp-image
9        readinessProbe:
10         httpGet:
11           path: /health
12           port: 8080
13         initialDelaySeconds: 5
14         periodSeconds: 10
```

```

15     livenessProbe:
16         exec:
17             command:
18                 - /bin/sh
19                 - -c
20                 - ps aux | grep myapp
21         initialDelaySeconds: 10
22         periodSeconds: 15
23     startupProbe:
24         tcpSocket:
25             port: 8080
26         initialDelaySeconds: 20
27         periodSeconds: 5

```

在上述示例中，定义了一个就绪探针、一个存活探针和一个启动探针。就绪探针通过发送 HTTP GET 请求到 `/health` 路径检查应用程序是否准备好接收流量。存活探针通过执行 `/bin/sh -c "ps aux | grep myapp"` 命令来检查应用程序是否正常运行。启动探针通过尝试连接到 8080 端口的 TCP 套接字来检测容器是否已经启动完成。每个探针都设置了初始延迟和检查周期。

通过使用探针机制，可以及时发现应用程序的健康状况，并在需要时采取相应的操作，如自动重启、阻止流量转发或等待容器启动完成。

示例：使用探针机制监测微服务

SpringBoot (2.3+) 已经支持了 Kubernetes 的探针机制，只需要添加 `spring-boot-starter-actuator` 依赖，并在 `application.yml` 中配置如下内容：

```

1  server: # 服务器配置
2      shutdown: graceful # 开启优雅停机
3  spring:
4      lifecycle:
5          timeout-per-shutdown-phase: 30s #设置缓冲时间 默认30s
6  management:
7      endpoint:
8          health:
9              probes:
10                 enabled: true
11                 show-details: always
12      endpoints:
13          web:
14              exposure:
15                  include: "*"
16      health:
17          livenessState:
18              enabled: true
19          readinessState:
20              enabled: true

```

- 存活探针的路径是：`/actuator/health/liveness`。
- 就绪探针的路径是：`/actuator/health/readiness`。

```

1  apiVersion: v1
2  kind: Pod
3  metadata:

```

```
4   name: nginx-prode
5   labels:
6     app: nginx-prode
7   spec:
8     terminationGracePeriodSeconds: 30 # 优雅停机时间
9     containers:
10      - name: nginx
11        image: nginx:1.20.2
12        imagePullPolicy: IfNotPresent
13        livenessProbe: # 就绪探针
14          httpGet:
15            path: /actuator/health/liveness
16            port: 8080
17            scheme: HTTP
18        readinessProbe:
19          httpGet:
20            path: /actuator/health/readiness
21            port: 8080
22            scheme: HTTP
```

Pod 调度

Pod 调度的概念

Pod 调度是 Kubernetes 中的一个关键概念，用于将 Pod 分配到集群中的节点上运行。调度器 (Scheduler) 是 Kubernetes 系统的组件之一，负责执行 Pod 调度的过程。

什么是 Pod 调度

Pod 调度是指将创建的 Pod 实例分配到集群中的合适节点上，以便运行应用程序。调度器根据一系列算法和策略，考虑节点的资源利用率、节点的标签、节点的负载均衡情况等因素，选择最适合的节点来运行 Pod。调度过程中，调度器会考虑每个节点的可用资源，例如 CPU、内存和存储等，以确保节点能够满足 Pod 的资源需求。

Pod 调度是一个动态的过程，当有新的 Pod 创建请求时，调度器会尽力将其分配到最佳的节点上。同时，调度器还会监控节点的状态和资源使用情况，以便在节点不可用或资源不足的情况下重新调度 Pod 到其他可用的节点上。

调度器的作用

调度器是 Kubernetes 系统的一个关键组件，主要负责以下任务：

1. 节点选择：调度器根据节点的可用资源、标签和其他条件，选择最佳的节点来运行 Pod。它会考虑节点的负载均衡，避免某些节点过载而导致性能下降。
2. 资源分配：调度器会根据 Pod 的资源需求（如 CPU、内存）和节点的资源利用率，对节点进行资源分配，以确保每个 Pod 能够得到足够的资源来正常运行。
3. 故障处理：调度器会监控节点的状态，如果某个节点不可用或发生故障，调度器会重新调度该节点上的 Pod 到其他可用节点上，以保证应用程序的高可用性。
4. 扩展性和负载均衡：调度器支持水平扩展和负载均衡。当集群中新增节点时，调度器可以将 Pod 均匀地分配到这些节点上，以实现集群的负载均衡。

总之，调度器的作用是通过智能的算法和策略，将 Pod 分配到合适的节点上，以实现资源的有效利用和应用程序的高可用性。

调度策略和调度器

调度策略是 Kubernetes 中决定 Pod 如何被调度的规则集合，而调度器是负责执行 Pod 调度过程的组件。在 Kubernetes 中，默认的调度器是 kube-scheduler，它根据一系列算法和策略来选择最适合的节点进行调度。此外，Kubernetes 还支持自定义调度器，允许用户根据自己的需求定义和使用自己的调度器。

默认调度器的行为

默认调度器 kube-scheduler 的行为是基于调度策略来进行 Pod 的调度。它首先考虑节点的资源利用率，选择资源满足要求的节点。如果有多个节点满足要求，调度器会根据负载均衡的原则来选择一个节点。同时，调度器还会根据 Pod 的亲性和、反亲性和、容忍和污点等配置进行调度决策。

自定义调度器的配置和使用

Kubernetes 提供了自定义调度器的功能，允许用户根据自己的需求定义和使用自己的调度器。自定义调度器可以根据用户自定义的调度策略来进行 Pod 的调度。

要配置和使用自定义调度器，需要以下步骤：

1. 创建调度器配置文件：创建一个调度器的配置文件，其中包含调度器的名称、调度策略、优先级等信息。
2. 部署调度器：使用 Kubernetes 的资源定义文件，将自定义调度器部署到集群中。
3. 配置 Pod 的调度策略：在 Pod 的配置文件中，通过设置 `schedulerName` 字段，指定要使用的调度器名称。

使用自定义调度器的好处是可以根据自己的业务需求定义灵活的调度策略，以实现更精确和个性化的 Pod 调度。

标签和选择器的作用

标签 (Labels) 和选择器 (Selectors) 是 Kubernetes 中用于标识和选择对象的重要概念。它们在 Pod 调度和资源管理中起着关键的作用。

标签的定义和作用

标签是用于对 Kubernetes 对象进行分类和标识的键值对。它们可以附加到各种对象，如 Pod、节点、服务等。每个标签都由一个键 (key) 和一个值 (value) 组成，形式为 `key=value`。

标签的作用有以下几个方面：

1. 分组和分类：通过为对象添加标签，我们可以对它们进行分组和分类。标签可以根据应用程序、环境、用途等不同维度来定义，以便更好地组织和管理对象。
2. 选择和过滤：标签提供了一种灵活的方式来选择和过滤对象。通过使用选择器 (Selectors)，我们可以根据标签的键值对进行筛选，从而选择满足特定条件的对象。
3. 关联和关系建立：标签可以用于建立对象之间的关联关系。通过在不同对象上使用相同的标签，我们可以将它们关联起来，从而方便进行跨对象的操作和管理。

选择器的使用和匹配规则

选择器是用于根据标签来选择对象的一种机制。它定义了一组匹配规则，用于筛选具有特定标签的对象。

在 Kubernetes 中，有两种常见的选择器：

- **等式选择器 (Equality-based Selectors)**：等式选择器通过指定标签的键值对来进行选择。例如，可以使用 `key=value` 的形式来选择具有指定键值对的对象。

- **集合选择器 (Set-based Selectors)**：集合选择器通过使用一组标签来进行选择。它包括以下几种匹配规则：

- `in`：选择具有给定标签值之一的对象。
- `notin`：选择不具有给定标签值之一的对象。
- `exists`：选择具有指定键的对象。
- `notexists`：选择不具有指定键的对象。
- `!`：取反操作，选择不匹配给定条件的对象。

选择器可以在各种 Kubernetes 对象的配置文件中使用时，例如 Pod 的配置文件。通过将选择器与标签配对使用，可以实现对对象的精确选择和筛选。

标签的实践

在 YAML 配置文件中，可以使用 `metadata` 字段为对象添加标签。以下是一个 Pod 配置文件的示例，展示了如何为 Pod 添加标签：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5    labels:
6      app: my-app
7      environment: production
8  spec:
9    containers:
10     - name: my-container
11       image: my-image:latest
```

在上述示例中，我们为 Pod 添加了两个标签，分别是 `app: my-app` 和 `environment: production`。这些标签可以根据实际需求定义，以便对 Pod 进行分类、选择和过滤。

选择器的实践

在 YAML 配置文件和命令操作中，可以使用选择器来选择具有特定标签的对象。以下是一些示例：

- YAML 配置文件示例：

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-deployment
5  spec:
6    selector:
7      matchLabels:
8        app: my-app
9    template:
10     metadata:
11       labels:
12         app: my-app
13     spec:
14       containers:
15         - name: my-container
16           image: my-image:latest
```

在上述示例中，我们使用选择器来选择具有标签 `app: my-app` 的 Pod。

命令操作示例

- 添加标签

要为已经存在的对象添加标签，可以使用以下命令：

```
1 # 为 Pod 添加标签
2 kubectl label pods <pod-name> <label-key>=<label-value>
3
4 # 为 Deployment 添加标签
5 kubectl label deployment <deployment-name> <label-key>=<label-value>
```

- 删除标签

要删除已经存在的对象上的标签，可以使用以下命令：

```
1 # 删除 Pod 上的标签
2 kubectl label pods <pod-name> <label-key>-
3
4 # 删除 Deployment 上的标签
5 kubectl label deployment <deployment-name> <label-key>-
6 注意，使用 - 表示删除指定的标签。
```

- 使用选择器进行匹配

要使用选择器进行匹配，可以在 `kubectl` 命令中使用 `-l` 或 `--selector` 参数。以下是一些常见的使用选择器的命令示例：

```
1 # 获取具有特定标签的 Pod
2 kubectl get pods -l <label-key>=<label-value>
3
4 # 获取具有多个标签的 Pod
5 kubectl get pods -l <label-key1>=<label-value1>,<label-key2>=<label-value2>
6
7 # 删除具有特定标签的 Pod
8 kubectl delete pods -l <label-key>=<label-value>
9
10 # 获取具有特定标签的 Deployment
11 kubectl get deployments -l <label-key>=<label-value>
12
```

通过使用 `-l` 或 `--selector` 参数，我们可以根据选择器的条件来选择和操作具有特定标签的对象。

- 更新标签

要更新已经存在对象上的标签，可以使用以下命令：

```
1 # 更新 Pod 上的标签
2 kubectl label pods <pod-name> <label-key>=<new-label-value> --overwrite
3
4 # 更新 Deployment 上的标签
5 kubectl label deployment <deployment-name> <label-key>=<new-label-value> --
  overwrite
```

注意，使用 `--overwrite` 参数来覆盖原有标签的值。

- 查看标签

要查看对象的标签，可以使用以下命令：

```
1 # 查看 Pod 上的标签
2 kubectl get pods --show-labels
3
4 # 查看 Deployment 上的标签
5 kubectl get deployments --show-labels
```

使用 `--show-labels` 参数可以显示对象的标签信息。

- 筛选标签

要使用选择器进行标签筛选，可以在 `kubectl` 命令中使用 `-l` 或 `--selector` 参数。以下是一些常见的使用选择器进行标签筛选的命令示例：

```
1 # 获取具有特定标签的 Pod
2 kubectl get pods -l <label-key>=<label-value>
3
4 # 获取具有多个标签的 Pod
5 kubectl get pods -l <label-key1>=<label-value1>,<label-key2>=<label-value2>
6
7 # 删除具有特定标签的 Pod
8 kubectl delete pods -l <label-key>=<label-value>
9
10 # 获取具有特定标签的 Deployment
11 kubectl get deployments -l <label-key>=<label-value>
```

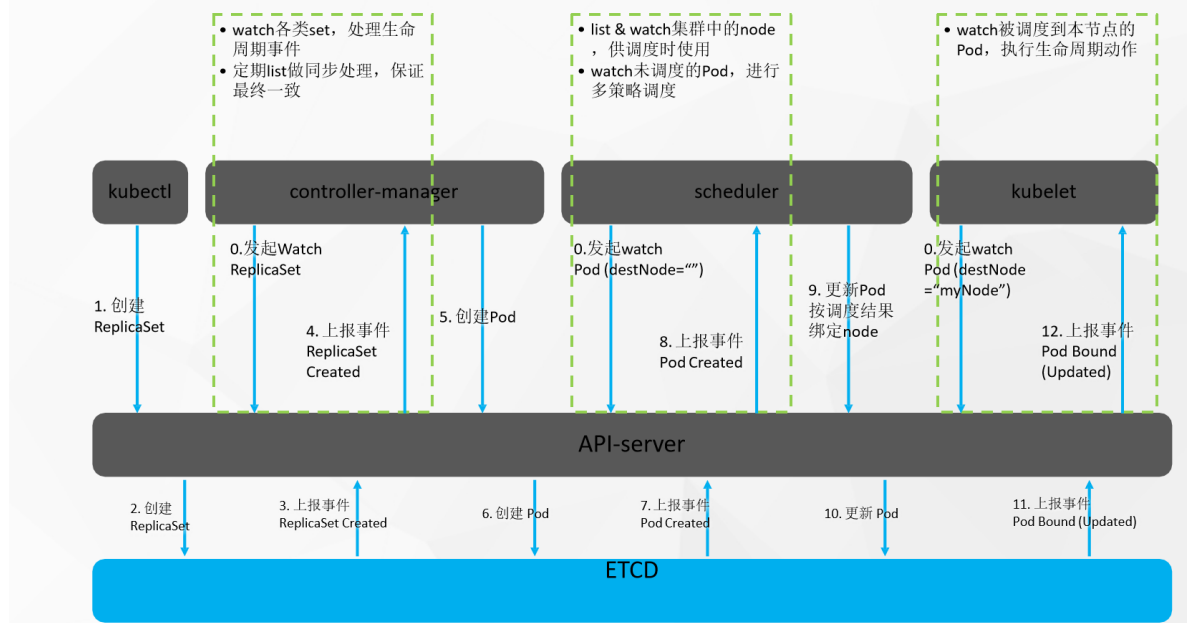
自动调度

自动调度是 Kubernetes 中的默认调度方式，它基于调度器的算法和策略来决定将 Pod 分配到哪个节点上运行。

自动调度的原理

1. **调度器接收到创建 Pod 的请求**：用户提交创建 Pod 的请求，该请求发送给调度器进行处理。
2. **调度器进行节点选择**：调度器根据一系列算法和策略选择最适合的节点进行调度。这些算法和策略考虑了节点的资源利用率、标签匹配、负载均衡等因素。
3. **调度器进行节点评分**：调度器对每个节点进行评分，计算每个节点的适合度得分。评分通常基于节点资源的可用性、节点和 Pod 之间的亲和性和反亲和性、节点上的污点和 Pod 的容忍等条件。
4. **调度器选择最高得分的节点**：调度器选择得分最高的节点作为最终的调度结果。如果有多个节点得分相同，则调度器会根据负载均衡原则选择一个节点。
5. **调度器发送调度结果**：调度器将选择的节点信息作为调度结果返回给 Kubernetes API Server。
6. **Kubernetes API Server更新 Pod 状态**：Kubernetes API Server接收到调度结果后，会更新 Pod 的状态，并将该信息发送给相应的节点进行调度。
7. **节点上运行 Pod**：被选中的节点会根据调度结果，运行相应的 Pod。

Kubernetes基于list-watch机制的控制器架构



通过Kubernetes 部署是一个 Tomcat 应用，了解组件交互原理。

- 0: 开机默认所有节点的 kubelet、master 节点的 scheduler（调度器）、controller-manager（控制管理器）一直**监听** master 的 api-server 发来的事件变化。
- 1: 程序员使用命令行工具：kubectl；**kubectl create deploy tomcat --image=tomcat8**（告诉 master 让集群使用 tomcat8 镜像，部署一个 tomcat 应用）。
- 2: kubectl 命令行内容发给 api-server，api-server 保存此次创建信息到 etcd。
- 3: etcd 给 api-server 上报事件，说刚才有人给我里面保存一个信息。（部署Tomcat[deploy]）
- 4: controller-manager 监听到 api-server 的事件，是（部署Tomcat[deploy]）。
- 5: controller-manager 处理这个（部署Tomcat[deploy]）的事件。controller-manager 会**生成 Pod 的部署信息【pod信息】**。
- 6: controller-manager 把 Pod 的信息交给 api-server，再保存到 etcd。
- 7: etcd 上报事件【pod信息】给 api-server。
- 8: scheduler 专门监听【pod信息】，拿到【pod信息】的内容，计算，看哪个节点合适部署这个 Pod【pod 调度过后的信息（node: node-02）】。
- 9: scheduler 把【pod 调度过后的信息（node: node-02）】交给 api-server 保存给 etcd。
- 10: etcd 上报事件【pod调度过后的信息（node: node-02）】，给 api-server。
- 11: 其他节点的 kubelet 专门监听【pod 调度过后的信息（node: node-02）】事件，集群所有节点 kubelet 从 api-server 就拿到了【pod调度过后的信息（node: node-02）】事件。
- 12: 每个节点的 kubelet 判断是否属于自己的事情；node-02 的 kubelet 发现是它的事情。
- 13: node-02 的 kubelet 启动这个 pod。汇报给 master 当前启动好的所有信息。

调度器的算法和策略

调度器的算法和策略决定了节点选择和评分的方式，以及最终选择哪个节点进行调度。

Kubernetes 中的调度器（kube-scheduler）采用了以下算法和策略：

- 优先级和资源配额**：调度器考虑每个节点的资源利用率和资源配额情况，选择满足要求的节点。
- 节点亲和性和反亲和性**：调度器支持亲和性和反亲和性，可以根据节点和 Pod 之间的标签匹配情况来调度。
- Pod 优先级和抢占**：调度器支持 Pod 的优先级和抢占，确保高优先级的 Pod 能够优先运行。

- **污点和容忍**：调度器支持节点的污点（Taints）和 Pod 的容忍（Tolerations），可以通过污点和容忍的设置来控制 Pod 是否能够调度到特定节点。

调度器的算法和策略可以根据用户的需求进行定制和扩展，以满足不同场景的调度需求。

定向调度

使用 `nodeName` 字段进行定向调度

定向调度是一种指定 Pod 运行在特定节点上的方式。其中一种方式是使用 `nodeName` 字段，在 Pod 的配置中直接指定节点名称。

示例 YAML 配置文件：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    nodeName: node-1
7    containers:
8      - name: my-container
9        image: my-image:latest
```

在上述示例中，我们通过在 Pod 的配置中指定 `nodeName: node-1`，将 Pod 定向调度到名为 `node-1` 的节点上运行。

使用 `nodeSelector` 字段进行定向调度

另一种常见的定向调度方式是使用 `nodeSelector` 字段。`nodeSelector` 允许你根据节点的标签选择合适的节点来运行 Pod。

示例 YAML 配置文件：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    nodeSelector:
7      app: backend
8    containers:
9      - name: my-container
10       image: my-image:latest
```

在上述示例中，我们通过设置 `nodeSelector: app: backend`，将 Pod 定向调度到具有标签 `app: backend` 的节点上运行。

亲和性调度

亲和性调度的概念和作用

亲和性调度是一种调度策略，用于将 Pod 调度到与其指定的节点或其他 Pod 相关联的节点上。它可以通过提高节点的资源利用率、改善应用性能、实现节点间的协作等方式发挥作用。

亲和性调度可以分为节点亲和性调度和 Pod 亲和性调度。

- **nodeAffinity(节点亲和性)** : 以node为目标, 解决pod可以调度到哪些node的问题
- **podAffinity(pod亲和性)** : 以pod为目标, 解决pod可以和哪些已存在的pod部署在同一个拓扑域中的问题
- **podAntiAffinity(pod反亲和性)** : 以pod为目标, 解决pod不能和哪些已存在pod部署在同一个拓扑域中的问题

使用 NodeAffinity 字段进行节点亲和性调度

使用 `NodeAffinity` 字段可以定义节点亲和性规则, 将 Pod 调度到与其指定的节点标签匹配的节点上。

示例 YAML 配置文件:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    affinity:
7      nodeAffinity:
8        requiredDuringSchedulingIgnoredDuringExecution:
9          nodeSelectorTerms:
10           - matchExpressions:
11             - key: app
12               operator: In
13               values:
14                 - backend
15    containers:
16     - name: my-container
17       image: my-image:latest
```

在上述示例中, 我们使用 `nodeAffinity` 字段定义了节点亲和性规则。该规则要求 Pod 被调度到具有标签 `app: backend` 的节点上。

使用 PodAffinity 字段进行 Pod 亲和性调度

使用 `PodAffinity` 字段可以定义 Pod 亲和性规则, 控制 Pod 与其他 Pod 的关联和调度位置。

示例 YAML 配置文件:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    affinity:
7      podAffinity:
8        requiredDuringSchedulingIgnoredDuringExecution:
9          - labelSelector:
10              matchExpressions:
11                - key: app
12                  operator: In
13                  values:
14                    - backend
15            topologyKey: kubernetes.io/hostname
```

```
16     containers:
17       - name: my-container
18         image: my-image:latest
```

在上述示例中，我们使用 `podAffinity` 字段定义了 Pod 亲和性规则。该规则要求 Pod 被调度到与具有标签 `app: backend` 的其他 Pod 所在的节点上。

使用 PodAntiAffinity 字段进行 Pod 反亲和性调度

使用 `PodAntiAffinity` 字段可以定义 Pod 反亲和性规则，控制 Pod 避免与其他 Pod 调度到同一节点。

示例 YAML 配置文件：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    affinity:
7      podAntiAffinity:
8        preferredDuringSchedulingIgnoredDuringExecution:
9          - weight: 100
10            podAffinityTerm:
11              labelSelector:
12                matchExpressions:
13                  - key: app
14                    operator: In
15                    values:
16                      - frontend
17              topologyKey: kubernetes.io/hostname
18    containers:
19      - name: my-container
20        image: my-image:latest
```

在上述示例中，我们使用 `podAntiAffinity` 字段定义了 Pod 反亲和性规则。该规则将 Pod 避免调度到与具有标签 `app: frontend` 的其他 Pod 所在的节点上。

关于亲和性(反亲和性)使用场景的说明

- **亲和性**：如果两个应用频繁交互，那就有必要利用亲和性让两个应用的尽可能的靠近，这样可以减少因网络通信而带来的性能损耗。
- **反亲和性**：当应用的采用多副本部署时，有必要采用反亲和性让各个应用实例打散分布在各个 node 上，这样可以提高服务的高可用性。

污点调度

污点调度的概念和作用

污点调度是一种调度策略，用于标记节点上的污点（Taints），以限制哪些 Pod 可以调度到该节点上。污点可以用于排除一些节点，保护特定的节点资源，或者进行节点故障模拟等操作。

使用 字段给节点添加污点

通过使用 `Taints` 字段，可以为节点添加污点，以限制 Pod 的调度。

污点的格式为 `key=value:effect`，其中 `key` 和 `value` 是污点的标签，`effect` 描述了污点的作用。污点的 `effect` 支持以下三个选项：

- `NoSchedule`：不允许调度新的 Pod 到带有该污点的节点上。
- `PreferNoSchedule`：尽量避免调度新的 Pod 到带有该污点的节点上，但不是强制性的。
- `NoExecute`：在运行中的 Pod 如果不符合容忍规则，则会被从带有该污点的节点上驱逐。

示例 YAML 配置文件：

```
1  apiVersion: v1
2  kind: Node
3  metadata:
4    name: node-1
5  spec:
6    taints:
7      - key: key1
8        value: value1
9        effect: NoSchedule
```

在上述示例中，我们为名为 `node-1` 的节点添加了一个污点。该污点的键为 `key1`，值为 `value1`，作用效果为 `NoSchedule`，表示不允许调度新的 Pod 到该节点上。

Taints 命令操作

- 设置节点污点

要设置节点的污点，可以使用以下命令：

```
1  kubectl taint nodes <node-name> <taint-key>=<taint-value>:<taint-effect>
```

示例：

```
1  kubectl taint nodes node-1 key1=value1:NoSchedule
```

上述命令将在名为 `node-1` 的节点上添加一个污点。该污点的键为 `key1`，值为 `value1`，作用效果为 `NoSchedule`。

- 去除节点污点

要去除节点的污点，可以使用以下命令：

```
1  kubectl taint nodes <node-name> <taint-key>-
```

示例：

```
1  kubectl taint nodes node-1 key1-
```

上述命令将从名为 `node-1` 的节点上移除键为 `key1` 的污点。

- 查看节点污点

要查看节点的污点信息，可以使用以下命令：

```
1 kubectl describe node <node-name> | grep Taints
```

示例：

```
1 kubectl describe node node-1 | grep Taints
```

上述命令将显示名为 `node-1` 的节点上的污点信息。

使用 Tolerations 字段声明 Pod 对污点的容忍

通过使用 `Tolerations` 字段，可以在 Pod 的配置中声明对污点的容忍，使得 Pod 仍然可以调度到带有污点的节点上。

要为 Pod 添加对污点的容忍，可以在 Pod 的配置文件中添加 `tolerations` 字段。示例 YAML 配置文件：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5 spec:
6   tolerations:
7     - key: key1
8       value: value1
9       operator: Equal
10      effect: NoSchedule
11   containers:
12     - name: my-container
13       image: my-image:latest
```

在上述示例中，我们在 Pod 的配置中添加了一个 `tolerations` 字段，声明对具有键为 `key1`、值为 `value1`、作用效果为 `NoSchedule` 的污点的容忍。这将允许 Pod 被调度到带有该污点的节点上。

Tolerations 命令操作

使用 `kubectl` 命令可以对 Pod 的 `tolerations` 进行操作，声明 Pod 对污点的容忍。以下是一些常用的命令操作示例：

- 添加 Pod 对污点的容忍

要为 Pod 添加对污点的容忍，可以使用 `kubectl patch` 命令：

```
1 kubectl patch pod <pod-name> -p '{"spec":{"tolerations":[{"key":"<taint-key>", "operator":"<operator>", "value":"<taint-value>", "effect":"<taint-effect>"}]}}'
```

示例：

```
1 kubectl patch pod my-pod -p '{"spec":{"tolerations":[{"key":"key1", "operator":"Equal", "value":"value1", "effect":"NoSchedule"}]}}'
```

上述命令将为名为 `my-pod` 的 Pod 添加对键为 `key1`、值为 `value1`、作用效果为 `NoSchedule` 的污点的容忍。

- 删除 Pod 的容忍规则

要删除 Pod 的容忍规则，可以使用 `kubectl patch` 命令将 `tolerations` 字段设置为空数组：

```
1 kubectl patch pod <pod-name> -p '{"spec":{"tolerations":[]}]'
```

示例：

```
1 kubectl patch pod my-pod -p '{"spec":{"tolerations":[]}]'
```

上述命令将删除名为 `my-pod` 的 Pod 的所有容忍规则。

- 查看 Pod 的容忍规则

要查看 Pod 的容忍规则，可以使用以下命令：

```
1 kubectl get pod <pod-name> -o=jsonpath='{.spec.tolerations}'
```

示例：

```
1 kubectl get pod my-pod -o=jsonpath='{.spec.tolerations}'
```

上述命令将显示名为 `my-pod` 的 Pod 的容忍规则信息。

示例场景：隔离敏感数据

在某些情况下，你可能希望将处理敏感数据的应用程序与其他应用程序隔离开来，以确保数据的安全性和保密性。使用 Pod 调度的功能，你可以实现对敏感数据应用程序的隔离。

一种常见的方法是通过节点亲和性调度将敏感数据应用程序调度到特定的节点上，而其他非敏感数据应用程序则调度到其他节点上。这样可以确保敏感数据应用程序在与其他应用程序隔离的环境中运行。

示例 YAML 配置文件：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: sensitive-app
5 spec:
6   affinity:
7     nodeAffinity:
8       requiredDuringSchedulingIgnoredDuringExecution:
9         nodeSelectorTerms:
10          - matchExpressions:
11            - key: app
12              operator: In
13              values:
14                - sensitive
15   containers:
16     - name: sensitive-container
17       image: sensitive-app:latest
```

在上述示例中，我们使用节点亲和性调度将名为 `sensitive-app` 的敏感数据应用程序调度到具有标签 `app: sensitive` 的节点上。

示例场景：平衡节点负载

在 Kubernetes 集群中，节点的负载均衡是非常重要的，它确保各个节点之间的资源利用均衡，提高整个集群的性能和稳定性。通过使用 Pod 调度功能，可以实现节点负载的平衡。

一种常见的做法是使用节点亲和性调度和标签选择器来将 Pod 均匀地分布在不同的节点上。可以使用标签选择器和亲和性规则来确保 Pod 在调度时考虑节点的负载情况，并选择负载较低的节点。

示例 YAML 配置文件：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    affinity:
7      nodeAffinity:
8        preferredDuringSchedulingIgnoredDuringExecution:
9          - weight: 100
10             preference:
11               matchExpressions:
12                 - key: app
13                   operator: In
14                   values:
15                     - frontend
16   containers:
17     - name: my-container
18       image: my-image:latest
```

在上述示例中，我们使用节点亲和性调度将 Pod 调度到具有标签 `app: frontend` 的节点上。通过设置 `weight` 属性为 100，确保优先选择负载较低的节点。

示例场景：避免单节点故障导致应用不可用

当你部署的应用程序对高可用性和容错性要求很高时，避免单个节点故障对应用程序的可用性产生影响是至关重要的。使用 Pod 调度的功能，你可以实现对单节点故障的容错。

一种常见的方法是使用 Pod 反亲和性调度将相同应用程序的多个副本调度到不同的节点上。这样，在某个节点发生故障时，其他节点上的副本仍然可以提供服务，确保应用程序的可用性。

示例 YAML 配置文件：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    affinity:
7      podAntiAffinity:
8        preferredDuringSchedulingIgnoredDuringExecution:
9          - weight: 100
10             podAffinityTerm:
11               labelSelector:
```

```
12         matchExpressions:
13           - key: app
14             operator: In
15             values:
16               - my-app
17         topologyKey: kubernetes.io/hostname
18     containers:
19       - name: my-container
20         image: my-image:latest
```

在上述示例中，我们使用 Pod 反亲和性调度将相同标签为 `app: my-app` 的多个副本调度到不同的节点上。通过设置 `weight` 属性为 100，确保优先选择不在同一节点上的副本。

控制器：应用程序的调度与管理

控制器概述

什么是 Kubernetes 控制器

在 Kubernetes 中，控制器是负责管理和控制应用程序的关键组件。它们用于确保所管理的资源处于期望的状态，并根据实际情况进行调整，以实现应用程序的预期行为。在之前的章节中，我们已经学习了 Pod，它是 Kubernetes 中的最小调度单位。现在，我们将进一步学习各种控制器，它们用于**管理和调度 Pod 资源**。

控制器的作用和重要性

控制器的主要作用是管理和控制应用程序的生命周期。它们负责创建、部署、调度、更新和删除应用程序的相关资源，以确保应用程序在集群中的运行状态符合预期。控制器可以根据需求自动进行资源的扩缩，以满足应用程序的负载变化。它们还负责监控应用程序的健康状态，并根据需要进行故障恢复。

控制器的分类和特点

控制器根据其功能和使用方式可以分为多种类型。常见的控制器类型包括 ReplicaSet、Deployment、DaemonSet、StatefulSet、Job 和 CronJob 等。每种控制器都有其独特的特点和适用场景，可以满足不同应用程序的需求。

以下是一些常见控制器的特点：

- **ReplicaSet**：用于创建和管理多个 Pod 的副本，确保指定数量的 Pod 在集群中运行。
- **Deployment**：提供了对应用程序的声明式管理，支持滚动更新、回滚和扩缩容等操作。
- **DemonSet**：确保集群中的每个节点运行一个 Pod 实例，常用于运行守护进程应用程序。
- **StatefulSet**：用于管理有状态应用程序，提供唯一标识和稳定网络标识符。
- **Job**：用于运行一次性任务或批处理作业，并确保任务完成后终止。
- **CronJob**：基于时间表调度任务的控制器，支持按照指定时间间隔运行任务。

控制器的核心概念和机制

控制器的核心概念包括目标对象、期望状态和实际状态的对比以及调谐机制。控制器通过监控目标对象的变化，并根据期望状态和实际状态的对比来决定是否需要采取调整措施。调谐机制用于控制器的自动调节和自我修复能力，确保应用程序持续处于所期望的状态。

ReplicaSet 控制器

ReplicaSet 控制器介绍

ReplicaSet 控制器是 Kubernetes 中的一个重要控制器，用于管理和控制 Pod 的副本集。它确保指定数量的 Pod 在集群中运行，以满足应用程序的负载需求。

ReplicaSet 的作用和功能

ReplicaSet 的主要作用是维护指定数量的 Pod 副本，并根据需要进行副本的创建、调度和删除。它可以确保应用程序的副本数量始终符合预期，实现应用程序的高可用性和负载均衡。

管理和操作 ReplicaSet

ReplicaSet 控制器提供了多种操作来管理和操作 Pod 的副本集。

创建和部署 ReplicaSet

要创建和部署一个 ReplicaSet，需要定义一个包含 Pod 模板和副本数量的 YAML 配置文件。在配置文件中指定所需的 Pod 模板和副本数量后，可以使用 `kubectl create` 命令将 ReplicaSet 配置应用到集群中。

示例 YAML 配置文件：

```
1  apiVersion: apps/v1
2  kind: ReplicaSet
3  metadata:
4    name: my-replicaset
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16       - name: my-container
17         image: nginx
```

使用 `kubectl create` 命令创建 ReplicaSet：

```
1  kubectl create -f replicaset.yaml
```

扩容和缩容 ReplicaSet (后续类似)

要扩容或缩容一个 ReplicaSet，有以下三种方式：

- 方式一：可以更新 ReplicaSet 的 YAML 配置文件中的 `spec.replicas` 字段的值，然后使用 `kubectl apply` 命令将更新的配置应用到集群中。

```
1  # replicaset.yaml
2  spec:
3    replicas: 5  # 更新副本数量为 5
```

使用 `kubectl apply` 命令扩容或缩容 ReplicaSet:

```
1 kubectl apply -f replicaset.yaml
```

- 方式二: 使用 `scale` 命令使用扩缩容, 后面加上 `--replicas=n` 直接指定目标数量即可。

```
1 kubectl scale rs my-replicaset --replicas=2
```

- 方式三: 使用 `kubectl edit` 命令, 修改 `spec:replicas:n` 即可

```
1 kubectl edit rs my-replicaset
```

更新 ReplicaSet (后续类似)

要更新一个 ReplicaSet, 可以修改 ReplicaSet 的 YAML 配置文件中的 Pod 模板或其他字段的值, 然后使用 `kubectl apply` 命令将更新的配置应用到集群中。Kubernetes 控制器会启动新的 Pod, 并逐步停止旧的 Pod, 实现滚动更新。

示例 YAML 配置文件:

```
1 apiVersion: apps/v1
2 kind: ReplicaSet
3 metadata:
4   name: my-replicaset
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: my-app
10  template:
11    metadata:
12      labels:
13        app: my-app
14    spec:
15      containers:
16      - name: my-container
17        image: nginx:1.2 # 更新容器镜像为新版本
```

使用 `kubectl apply` 命令更新 ReplicaSet:

```
1 kubectl apply -f replicaset.yaml
```

删除和清理 ReplicaSet (后续类似)

- 方式一: 通过 `kubectl delete` 命令

要删除一个 ReplicaSet, 可以使用 `kubectl delete` 命令, 并指定 ReplicaSet 的名称。

```
1 kubectl delete replicaset my-replicaset
2
3 # 等价于
4
5 kubectl delete replicaset my-replicaset --cascade=true
```

当删除一个 ReplicaSet 时，其管理的 Pod 也会被自动删除。如果不想自动删除其管理的 Pod，可以使用 `kubectl delete` 命令的 `--cascade` 参数。

```
1 kubectl delete replicaset my-replicaset --cascade=false
```

- 方式二：通过 yaml 文件

```
1 kubectl delete -f replicaset.yaml
```

示例：replicaSet 控制器管理应用程序副本集

假设我们有一个应用程序，需要确保始终有 3 个 Pod 副本运行。我们可以使用 ReplicaSet 控制器来管理这个副本集。

首先，创建一个包含 Pod 模板和副本数量的 YAML 配置文件（例如 `replicaset.yaml`）：

```
1 apiVersion: apps/v1
2 kind: ReplicaSet
3 metadata:
4   name: my-replicaset
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: my-app
10  template:
11    metadata:
12      labels:
13        app: my-app
14    spec:
15      containers:
16      - name: my-container
17        image:
```

Deployment 控制器

Deployment 控制器介绍

Deployment 控制器是 Kubernetes 中最常用的控制器之一，它提供了对应用程序的声明性定义、自动化部署、滚动更新和回滚等功能。通过使用 Deployment 控制器，您可以轻松管理应用程序的副本集，并确保应用程序始终处于所需的状态。

Deployment 的作用和功能

Deployment 的主要作用是定义和管理应用程序的部署。它可以根据指定的副本数自动创建和管理 Pod 的副本集，并确保副本集中的 Pod 按照所需的状态运行。

Deployment 提供了以下关键功能：

- **声明性定义**：通过 YAML 文件或命令行参数，您可以声明性地定义 Deployment 的配置。这样，Kubernetes 将根据这些配置自动管理应用程序的部署状态。
- **自动化部署**：Deployment 控制器会根据您定义的配置，自动创建并管理 Pod 的副本集。它会确保所需的副本数一直保持在运行状态，并在需要时自动进行扩容和缩容。
- **滚动更新**：Deployment 支持滚动更新策略，允许您在不中断应用程序服务的情况下，逐步更新 Pod 的版本。您可以指定更新策略、版本号以及更新的时间间隔和速率等参数。

- **回滚操作**：如果应用程序更新出现问题或不符合预期，您可以使用 Deployment 控制器进行快速的回滚操作，将应用程序恢复到之前的稳定版本。

管理和操作 Deployment

创建和部署 Deployment

要创建和部署一个 Deployment，您需要提供一个包含 Deployment 配置的 YAML 文件。示例配置文件如下：

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16       - name: my-container
17         image: nginx:latest
18         ports:
19         - containerPort: 80
```

通过使用 `kubectl create` 命令并指定 YAML 文件，可以创建并部署 Deployment，例如：

```
1  kubectl apply -f deployment.yaml
```

该命令会将配置文件中定义的 Deployment 部署到 Kubernetes 集群中。Kubernetes 控制器会自动创建和管理指定数量的 Pod 副本。

扩容和缩容 Deployment

要扩容或缩容一个 ReplicaSet，有以下三种方式：

- 方式一：可以更新 ReplicaSet 的 YAML 配置文件中的 `spec.replicas` 字段的值，然后使用 `kubectl apply` 命令将更新的配置应用到集群中。

```
1  # deployment.yaml
2  spec:
3    replicas: 5 # 更新副本数量为 5
```

使用 `kubectl apply` 命令扩容或缩容 ReplicaSet：

```
1  kubectl apply -f deployment.yaml
```

- 方式二：如果您需要增加或减少 Deployment 中运行的 Pod 副本的数量，可以使用 `kubectl scale` 命令进行扩缩容。例如，要将 Deployment 的副本数扩展到 5 个，可以运行以下命令：

```
1 kubectl scale deployment my-deployment --replicas=5
```

- 方式三：使用 `kubectl edit` 命令，修改 `spec:replicas:n` 即可

```
1 kubectl edit deployment my-deployment
```

更新 Deployment

Deployment 是 Kubernetes 中负责管理应用程序副本集的控制面，可以通过更新 Deployment 来进行应用程序的更新。在更新 Deployment 时，可以选择两种镜像更新策略：重建更新和滚动更新。可以通过 `strategy` 字段进行配置：

```
1 spec:
2   strategy:
3     type: RollingUpdate
4     rollingUpdate:
5       maxUnavailable: 25%
6       maxSurge: 25%
```

- `strategy`：指定新的Pod替代旧的Pod的策略，支持两个属性
- `type`：指定策略类型，支持两种策略
 - 重建更新（Recreate）：重建更新策略会先删除旧的 Deployment 对象中的所有 Pod，然后根据更新后的配置创建新的 Pod。这种策略会导致应用程序短暂的停机时间，因为在删除旧的 Pod 和创建新的 Pod 之间会有一段时间的间隙。
 - 滚动更新（Rolling）：滚动更新策略允许逐步更新 Deployment 中的 Pod，以确保在整个更新过程中应用程序保持可用状态。滚动更新会逐步替换旧的 Pod，同时保持应用程序的可用性，确保在更新过程中始终有足够的实例在运行。
- `rollingUpdate`：当type为RollingUpdate的时候生效，用于为rollingUpdate设置参数，支持两个属性
 - `maxUnavailable`：用来指定在升级过程中不可用的Pod的最大数量，默认为25%。
 - `maxSurge`：用来指定在升级过程中可以超过期望的Pod的最大数量，默认为25%。

Deployment 的默认的镜像更新策略是 RollingUpdate（滚动更新），**实际开发的时候，使用默认镜像更新策略即可。**

示例：将镜像更新

- 创建 `nginx-deployment.yaml` 用作测试

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: nginx
10  strategy: # 更新策略
11    type: Recreate
12  template:
13    metadata:
```

```

14     name: nginx
15   spec:
16     containers:
17     - name: nginx
18       image: nginx:1.17.1 # 镜像
19       imagePullPolicy: IfNotPresent
20     restartPolicy: Always

```

- 方式一：使用 `kubectl set image` 命令

```
1 kubectl set image deployment nginx-deployment nginx=nginx:1.20.2
```

- 方式二：修改 yaml 文件，使用 `kubectl apply -f nginx-deployment.yaml` 命令即可

```

1   ...
2   spec:
3     containers:
4     - name: nginx
5       image: nginx:1.20.2 # 镜像更新
6       imagePullPolicy: IfNotPresent
7     restartPolicy: Always
8   ...

```

回滚 Deployment

如果更新导致问题，可以使用 `kubectl rollout undo` 命令回滚到先前的版本。但是在进行回滚之前，请确保 Deployment 曾经进行过至少一个更新。

当使用 `kubectl apply` 命令或者将更新的配置文件直接传递给 `kubectl` 命令时，更新操作会被记录到 Deployment 的历史记录中。但是命令操作一般默认不会的。

而Deployment 之所以能够实现版本的回退，就是通过记录下历史的 ReplicaSet 来实现的，一旦想回滚到那个版本，只需要将当前版本的 Pod 数量降为 0，然后将回退版本的 Pod 提升为目标数量即可。

- 通过命令将更新操作添加到 Deployment 的历史记录中，通过 `--record` 参数用于记录历史。

```
1 kubectl set image deployment/nginx-deployment nginx=nginx:1.20.2
```

- 要检查是否有回滚历史记录

```
1 kubectl rollout history deployment nginx-deployment
```

- 如果输出中显示有历史记录

```
1 kubectl rollout undo deployment nginx-deployment
```

这将回滚到先前成功部署的版本，以恢复应用程序到稳定状态。

- 如果输出中显示有历史记录，回滚指定版本

```

1 # 可以使用-to-revision=1回退到1版本，如果省略这个选项，就是回退到上个版本，即2版本
2 kubectl rollout undo deployment nginx-deployment --to-revision=1

```

- `kubectl rollout` 命令操作

命令	语法
<code>kubectl rollout status deployment/<deployment-name></code>	查看 Deployment 的滚动更新状态。
<code>kubectl rollout history deployment/<deployment-name></code>	查看 Deployment 的历史版本。
<code>kubectl rollout undo deployment/<deployment-name></code>	回滚到上一个版本。
<code>kubectl rollout undo deployment/<deployment-name> --to-revision=<revision></code>	回滚到指定的版本。
<code>kubectl rollout pause deployment/<deployment-name></code>	暂停滚动更新操作。
<code>kubectl rollout resume deployment/<deployment-name></code>	恢复滚动更新操作。

请将 `<deployment-name>` 替换为您实际使用的 Deployment 名称。

暂停和继续 Deployment

在某些情况下，您可能需要暂停 Deployment 的进行。例如，您可能需要暂停更新操作，以便进行故障排除或其他维护工作。您可以使用 `kubectl rollout pause` 命令暂停 Deployment 的进行，例如：

```
1 kubectl rollout pause deployment my-deployment
```

这将暂停 Deployment 的进程，阻止任何新的副本部署或更新操作。

要继续进行 Deployment 的操作，可以使用 `kubectl rollout resume` 命令，例如：

```
1 kubectl rollout resume deployment my-deployment
```

这将恢复 Deployment 的正常操作，使其继续进行新的副本部署或更新操作。

删除和清理 Deployment

- 方式一：通过 `kubectl delete` 命令

要删除一个 ReplicaSet，可以使用 `kubectl delete` 命令，并指定 ReplicaSet 的名称。

```
1 kubectl delete deployment my-deployment
2
3 # 等价于
4
5 kubectl delete deployment my-deployment --cascade=true
```

当删除一个 Deployment 时，其管理的 Pod 也会被自动删除。如果不想自动删除其管理的 Pod，可以使用 `kubectl delete` 命令的 `--cascade` 参数。

```
1 kubectl delete deployment my-deployment --cascade=false
```

- 方式二：通过 yaml 文件

```
1 | kubectl delete -f deployment.yaml
```

金丝雀发布 Deployment

示例：使用 Deployment 控制器进行应用程序管理和版本控制

假设我们有一个名为 `my-app` 的应用程序，我们可以创建一个 Deployment 来管理它的副本集。以下是一个示例的 Deployment 的 YAML 配置文件：

```
1 | apiVersion: apps/v1
2 | kind: Deployment
3 | metadata:
4 |   name: my-deployment
5 | spec:
6 |   replicas: 3
7 |   selector:
8 |     matchLabels:
9 |       app: my-app
10 |  template:
11 |    metadata:
12 |      labels:
13 |        app: my-app
14 |    spec:
15 |      containers:
16 |        - name: my-app-container
17 |          image: my-app-image:1.0.0
18 |          ports:
19 |            - containerPort: 8080
```

通过将上述 YAML 文件保存为 `deployment.yaml`，然后运行以下命令创建和部署 Deployment：

```
1 | kubectl create -f deployment.yaml
```

Deployment 将根据定义的配置信息创建 3 个副本的 Pod，并确保它们一直保持在运行状态。您可以使用其他命令来扩容、更新、回滚或删除该 Deployment。

DaemonSet 控制器

DaemonSet 控制器介绍

在 Kubernetes 中，DaemonSet 控制器是一种用于在集群中每个节点上运行一个副本的控制器。它确保在集群的每个节点上都有一个 Pod 实例运行。与 ReplicaSet 和 Deployment 不同，DaemonSet 不关心副本数量，它只关心每个节点是否有一个 Pod 在运行。

DaemonSet 控制器通常用于在整个集群中运行一些特殊的系统级任务或守护进程应用程序，例如日志收集器、监控代理或网络插件。

DaemonSet 的作用和功能

DaemonSet 的作用是在每个节点上运行一个 Pod 实例，确保集群中的每个节点都执行相应的任务或运行指定的应用程序。它的功能包括：

- 在集群的每个节点上创建和管理 Pod 的生命周期。
- 监控节点的变化，并根据需要在新节点上自动部署或删除 Pod 实例。
- 提供节点级别的服务或应用程序，确保每个节点都有相应的 Pod 实例在运行。

管理和操作 DaemonSet

创建和部署 DaemonSet

要创建和部署 DaemonSet，可以使用以下示例的 YAML 配置文件：

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: my-daemonset
5  spec:
6    selector:
7      matchLabels:
8        app: my-app
9    template:
10     metadata:
11       labels:
12         app: my-app
13     spec:
14       containers:
15         - name: my-app
16           image: nginx:1.21.1
```

将上述配置保存为 `daemonset.yaml` 文件，并使用以下命令创建 DaemonSet：

```
1  kubectl apply -f daemonset.yaml
```

这将创建一个名为 `my-daemonset` 的 DaemonSet，并在每个节点上运行一个 Pod 实例。

更新 DaemonSet

1.6版本开始支持滚动更新。如下两种策略

- OnDelete：默认升级策略，在创建好新的DaemonSet配置之后，新的Pod不会被自动创建，用户需要手动删除旧版本的Pod，才触发新建操作。
- RollingUpdate：旧版本的POD 将被自动杀掉，然后自动创建新版的DaemonSet Pod。与 Deployment 不同为不支持查看和管理DaemonSet的更新记录；回滚操作是通过再次提交旧版本配置而不是 rollback命令实现

要更新 DaemonSet 的策略，可以编辑原始的 YAML 配置文件并更新相关字段。以下是一个示例，展示如何使用 `rollingUpdate` 策略进行滚动更新：

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: my-daemonset
5  spec:
6    selector:
7      matchLabels:
8        app: my-app
9    updateStrategy:
10     type: RollingUpdate
11     rollingUpdate:
12       maxUnavailable: 1
13       maxSurge: 1
```

```

14   template:
15     metadata:
16       labels:
17         app: my-app
18     spec:
19       containers:
20         - name: my-app
21           image: nginx:1.22.1

```

在上述示例中，我们添加了 `updateStrategy` 字段，并将 `type` 设置为 `RollingUpdate`，表示使用滚动更新策略。

- `maxUnavailable` 定义了在进行滚动更新期间允许不可用的最大 Pod 数量。这里设置为 `1`，表示每次更新只允许一个 Pod 不可用。
- `maxSurge` 定义了在进行滚动更新期间允许超出所需副本数的最大 Pod 数量。这里设置为 `1`，表示每次更新最多允许一个 Pod 超出所需副本数。

要应用更新策略，可以使用以下命令：

```
1 kubectl apply -f daemonset.yaml
```

这将更新 DaemonSet 的配置，并根据新的策略进行滚动更新。在滚动更新期间，将逐个替换旧版本的 Pod 实例，并确保在更新过程中保持可用性。

请注意，根据实际情况和需求，你可以调整 `maxUnavailable` 和 `maxSurge` 的值，以满足特定的更新需求。

删除和清理 DaemonSet

要删除和清理 DaemonSet，可以使用以下命令：

```
1 kubectl delete daemonset my-daemonset
```

示例：使用 DaemonSet 在每个节点上运行守护进程

例如，假设你有一个日志收集器，你希望在集群的每个节点上都运行该收集器的实例。你可以创建一个 DaemonSet 配置来实现这一目标。以下是一个示例的 DaemonSet 配置文件：

```

1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: log-collector
5  spec:
6    selector:
7      matchLabels:
8        app: log-collector
9    template:
10     metadata:
11       labels:
12         app: log-collector
13     spec:
14       containers:
15         - name: log-collector
16           image: log-collector-image:tag
17       # 容器配置和其他设置

```

在上面的示例中，`log-collector` 是 DaemonSet 的名称，并且 `app: log-collector` 是选择器（selector），用于标识要在哪些节点上运行 DaemonSet 中的 Pod。在 `template` 部分中，你可以定义要运行的容器，以及其他相关的配置。

使用上述配置，Kubernetes 将在集群的每个节点上自动创建一个 Pod 实例，其中运行着 `log-collector` 容器。这样，你就可以在每个节点上收集日志并执行其他相关任务。

StatefulSet 控制器

StatefulSet 控制器介绍

在 Kubernetes 中，StatefulSet 是一种控制器，用于管理有状态应用程序的部署。与无状态应用程序不同，有状态应用程序需要稳定的网络标识和有序的部署与扩展。StatefulSet 提供了这些功能，确保有状态应用程序的稳定性和可靠性。

StatefulSet 的作用和功能

StatefulSet 的作用是确保有状态的应用程序在 Kubernetes 集群中的可靠运行。它为每个 Pod 提供唯一的标识符，并为这些 Pod 分配稳定的网络标识符，以便可以持久地访问它们。这使得有状态的应用程序可以在扩容、升级或故障恢复等场景下保持数据的一致性和稳定性。

StatefulSet 控制器的主要功能包括：

- 为每个 Pod 分配稳定的网络标识符和主机名。
- 提供有序的创建、更新和删除操作，以确保应用程序的有状态特性得到保持。
- 集成持久化存储，为有状态应用程序提供持久化的数据存储能力。
- 支持有状态应用程序的扩容和缩容操作。

管理和操作 StatefulSet

创建和部署 StatefulSet

要创建和部署 StatefulSet，需要使用 YAML 配置文件来定义 StatefulSet 的规范。以下是一个示例的 StatefulSet 配置文件：

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: my-statefulset
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16       - name: my-container
17         image: nginx:latest
```

保存以上配置到名为 `my-statefulset.yaml` 的文件中，并使用以下命令来创建 StatefulSet：


```
1 | kubectl apply -f my-statefulset.yaml
```

这将创建一个名为 `my-statefulset` 的 StatefulSet，并部署 3 个 Pod。

扩容和缩容 StatefulSet

要扩容 StatefulSet 中的 Pod 数量，可以使用以下命令：

```
1 | kubectl scale statefulset my-statefulset --replicas=5
```

这会将 `my-statefulset` 的副本数量扩展到 5 个。

要缩容 StatefulSet，可以使用相同的命令并将副本数量减少到所需的数量。

更新 StatefulSet

比如，要更新 StatefulSet 中的容器镜像。

方式一：使用 `kubectl set image` 命令

```
1 | kubectl set image statefulset my-statefulset my-container=nginx:1.19
```

这会将 `my-statefulset` 中的 `my-container` 容器的镜像更新为 `nginx:1.19`。

- 方式二：修改 yaml 文件，使用 `kubectl apply -f my-statefulset.yaml` 命令即可

```
1 spec:
2   replicas: 3
3   selector:
4     matchLabels:
5       app: my-app
6   template:
7     metadata:
8       labels:
9         app: my-app
10    spec:
11      containers:
12        - name: my-container
13          image: nginx:1.19
```

删除和清理 StatefulSet

- 方式一：通过 `kubectl delete` 命令

要删除一个 ReplicaSet，可以使用 `kubectl delete` 命令，并指定 ReplicaSet 的名称。

```
1 | kubectl delete statefulset my-statefulset
2
3 # 等价于
4
5 | kubectl delete statefulset my-statefulset --cascade=true
```

当删除一个 Deployment 时，其管理的 Pod 也会被自动删除。如果不想自动删除其管理的 Pod，可以使用 `kubectl delete` 命令的 `--cascade` 参数。

```
1 | kubectl delete statefulset my-statefulset --cascade=false
```

- 方式二：通过 yaml 文件

```
1 | kubectl delete -f my-statefulset.yaml
```

使用 StatefulSet 管理有状态应用程序

通过使用 StatefulSet，你可以管理有状态应用程序的生命周期。StatefulSet 会根据配置在集群中创建指定数量的 Pod，并为每个 Pod 分配一个稳定的网络标识。这使得有状态应用程序可以保持稳定的网络连接，并保持数据的一致性。

稳定的网络标识

每个 StatefulSet 的 Pod 都有一个唯一的网络标识。这个标识是通过 Headless Service（无头服务）来提供的，它为每个 Pod 提供 DNS 解析条目。

无头服务的 DNS 条目通常为 `$(service name).$(namespace).svc.cluster.local`，因此，Pod 的解析条目为 `$(pod name).$(service name).$(namespace).svc.cluster.local`。

稳定的网络标识对于有状态应用程序非常重要，它可以用于建立稳定的网络连接、进行服务发现和通信。

有序的部署与扩展

StatefulSet 提供了有序的部署与扩展功能，通过 `podManagementPolicy` 参数来控制 Pod 的管理策略。`podManagementPolicy` 有两个选项：OrderedReady（有序启动，默认值）和 Parallel（并发一起启动）。

- OrderedReady：StatefulSet 将按照索引顺序逐个启动和更新 Pod。每个 Pod 在启动之前必须先达到 Ready 状态。这种管理策略适用于有状态应用程序，其中每个实例在启动之前可能需要一些初始化操作或依赖其他实例的状态。
- Parallel：StatefulSet 可以并发地启动和更新 Pod，而不需要等待之前的 Pod 达到 Ready 状态。这种管理策略适用于无状态应用程序或无依赖关系的有状态应用程序。

你可以通过在 StatefulSet 的配置中添加 `podManagementPolicy` 参数来指定管理策略。例如：

```
1 | apiVersion: apps/v1
2 | kind: StatefulSet
3 | metadata:
4 |   name: my-statefulset
5 | spec:
6 |   replicas: 3
7 |   podManagementPolicy: OrderedReady
8 |   # 其他配置项...
```

Job 控制器

Job 的概述

在 Kubernetes 中，Job 控制器是一种控制器对象，用于管理批量作业。Job 控制器负责确保指定数量的 Pod 在集群中运行，并且在所有 Pod 完成任务后将它们删除。

Job 控制器的主要组成部分包括以下几个方面：

- 需要执行的任务：任务可以是任何可以在容器中运行的操作，如 Shell 脚本、命令程序等。

- Pod 模板：Job 控制器使用 Pod 模板来创建 Pod。Pod 模板定义了要使用的容器镜像、容器的命令和参数等。
- 完成策略：完成策略定义了 Job 完成后如何处理 Pod。默认情况下，Job 控制器会在所有 Pod 完成任务后将它们删除。但是，您也可以将完成策略设置为保留所有 Pod 或仅保留最后一个 Pod。

创建和配置

要创建一个 Job，你需要创建一个 Job 配置文件。以下是一个示例的 Job 配置文件：

```
1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: my-job
5  spec:
6    template:
7      spec:
8        containers:
9          - name: my-container
10            image: my-image:tag
11            # 容器配置和其他设置
12    backoffLimit: 3
```

在上面的配置文件中，你需要指定 Job 的名称和 Pod 的模板。在模板中，你可以定义要运行的容器的镜像、环境变量、资源限制等。

`backoffLimit` 是一个可选参数，它指定了在任务或作业失败时的重试次数。在上述示例中，`backoffLimit` 设置为 3，意味着任务最多会重试 3 次。

Kubernetes Job 控制器的详细参数表格：

参数	描述	默认值
apiVersion	Job 对象的 API 版本	batch/v1
kind	Job 对象的类型	Job
metadata	Job 对象的元数据，包括名称、标签等	
spec.completions	Job 完成的 Pod 数量	1
spec.parallelism	同时运行的 Pod 数量	1
spec.backoffLimit	重试失败的 Pod 的次数	6
spec.activeDeadlineSeconds	Job 运行的最长时间，超过该时间后，所有未完成的 Pod 将被终止	
spec.selector	用于选择要包含在 Job 中的 Pod 的标签选择器	
spec.template.metadata	Pod 的元数据，包括名称、标签等	
spec.template.spec.containers	Pod 中的容器列表，包括容器的名称、镜像、命令和参数等	

参数	描述	默认值
spec.template.spec.restartPolicy	Pod 的重启策略。通常应将其设置为 Never，因为 Job 控制器将在所有 Pod 完成任务后删除它们	Never

CronJob 控制器

CronJob 的概述

CronJob 是基于 Cron 定时任务的 Kubernetes 资源类型。类似于传统的 Cron 任务，CronJob 允许您在特定时间或时间间隔内运行一次或多次任务。Kubernetes CronJob 控制器会监视 CronJob 对象，并在指定的时间表上创建和删除 Pod。

创建和配置

要使用 Kubernetes CronJob 控制器，您需要创建一个 CronJob 对象并将其提交给 Kubernetes API 服务器。以下是一个简单的 CronJob 对象示例：

```
1  apiVersion: batch/v1beta1
2  kind: CronJob
3  metadata:
4    name: my-cronjob
5  spec:
6    schedule: "*/1 * * * *"
7    jobTemplate:
8      spec:
9        template:
10       spec:
11         containers:
12         - name: my-container
13           image: my-image
14           command: ["echo", "Hello, world!"]
15         restartPolicy: OnFailure
```

在以上示例中，CronJob 控制器将创建一个名为 my-cronjob 的 CronJob 对象，该 CronJob 将每分钟运行一次一个简单的容器，该容器将运行一个命令 `echo "Hello, world!"`。该 CronJob 的 schedule 字段设置为 Cron 时间表，即 `*/1 * * * *`，表示该 CronJob 每分钟运行一次。

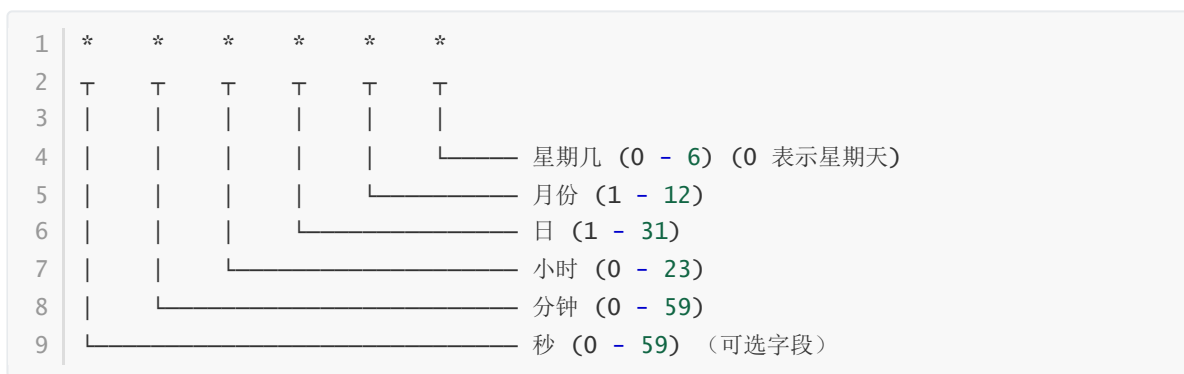
要创建此 CronJob，您可以使用以下命令：

```
1  kubectl create -f my-cronjob.yaml
```

运行该命令后，Kubernetes 将创建一个名为 my-cronjob 的 CronJob 对象并将其提交到 Kubernetes API 服务器。

运行定期任务

Kubernetes CronJob 控制器允许您在指定的时间表上定期运行任务。您可以使用 Cron 时间表语法指定任务应在哪些时间运行。以下是 Cron 表达式语法：



Cron 表达式的每个字段都有一组允许的值。以下是各个字段的含义：

- 秒（秒钟）：0 - 59
- 分钟（分钟）：0 - 59
- 小时（小时）：0 - 23
- 日（日期）：1 - 31
- 月（月份）：1 - 12 或 JAN - DEC
- 星期几：0 - 6 或 SUN - SAT

在以上每个字段中，您可以使用以下特殊字符：

- 星号（*）：表示该字段的所有可能值。例如，* 在分钟字段中表示每分钟都运行任务。
- 逗号（,）：用于分隔多个值。例如，1,3,5 在月份字段中表示一月、三月和五月运行任务。
- 中划线（-）：用于表示一个范围。例如，1-5 在日期字段中表示 1 到 5 号运行任务。
- 斜杠（/）：用于表示一个增量。例如，*/15 在分钟字段中表示每 15 分钟运行任务。

Cron 表达式的含义是根据各个字段的值来确定任务的运行时间。例如，以下 Cron 表达式：

```
1 | 0 0 * * * *
```

表示每天 0 点整开始运行任务。其中，第一个字段 0 表示秒钟，第二个字段 0 表示分钟，第三个字段 * 表示所有小时数，第四个字段 * 表示所有日期，第五个字段 * 表示所有月份，第六个字段 * 表示所有星期几。

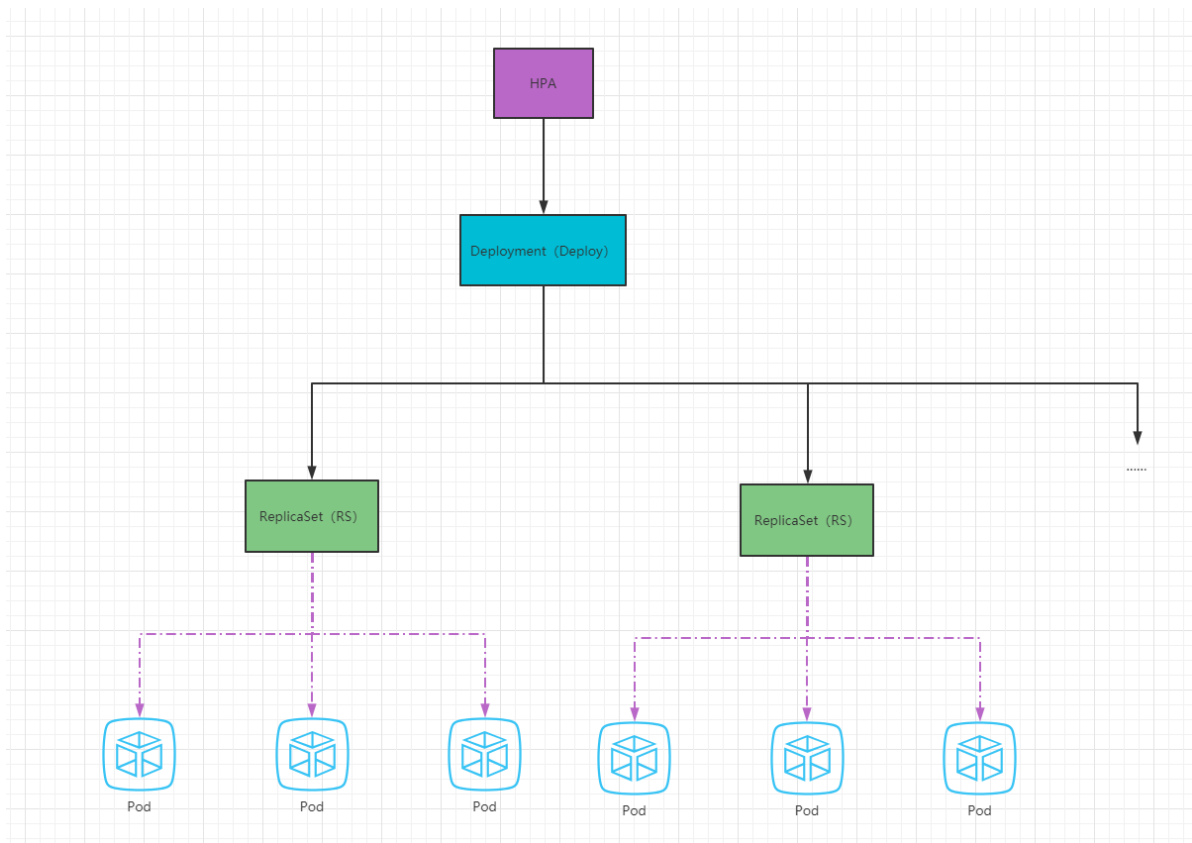
以下是一些 Cron 表达式示例：

- 0 0 * * * *：每小时运行一次。
- 0 0 12 * * *：每天中午 12 点运行一次。
- 0 0 * * * 1-5：每周一到周五每小时运行一次。
- 0 0 0 1 * *：每月的第一天午夜运行一次。
- 0 0 0 * * 1：每周一午夜运行一次。
- 0 0 0 * * 1-5：每周一到周五午夜运行一次。
- 0 0 0 1-15 * *：每月的第一天到第 15 天午夜运行一次。
- 0 0 0 1,15 * *：每月的第一天和第 15 天午夜运行一次。
- 0 0 0 * * 0：每周日午夜运行一次。

水平自动伸缩（HPA）控制器

HPA 控制器的介绍

水平自动伸缩（Horizontal Pod Autoscaler, HPA）是 Kubernetes 中的一个控制器，可以根据应用程序的负载情况自动扩展或缩小 Pod 的数量。HPA 控制器可以通过监视指标（例如 CPU 使用率或内存使用率）来发现负载变化，并根据预定义的规则调整 Pod 的数量，以确保应用程序具有所需的资源。



HPA 的作用和功能

HPA 控制器的作用是自动根据应用程序负载情况来调整 Pod 的数量，以确保应用程序具有所需的资源。HPA 控制器的功能包括：

- 监视指标：HPA 控制器可以监视指标，例如 CPU 使用率或内存使用率，以发现负载变化。
- 自动伸缩 Pod 数量：HPA 控制器可以根据预定义的规则自动扩展或缩小 Pod 的数量，以确保应用程序具有所需的资源。
- 避免资源浪费：通过自动调整 Pod 的数量，HPA 控制器可以避免资源浪费，从而提高资源利用率。
- 提高应用程序的可用性：通过确保应用程序具有足够的资源，HPA 控制器可以提高应用程序的可用性，并降低应用程序因资源不足而崩溃的风险。

使用 metrics-server 监控 Kubernetes 资源。

- 获取 metrics-server v0.6.1（网速不行，请点这里 [🔗 components.yaml](#)）

```
1 | wget https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.6.1/components.yaml
```

- 修改 components.yaml

```
1 | vim components.yaml
```

```
1 | apiVersion: apps/v1
2 | kind: Deployment
3 | metadata:
4 |   labels:
5 |     k8s-app: metrics-server
6 |   name: metrics-server
7 |   namespace: kube-system
```

```

8 spec:
9   selector:
10     matchLabels:
11       k8s-app: metrics-server
12   strategy:
13     rollingUpdate:
14       maxUnavailable: 0
15   template:
16     metadata:
17       labels:
18         k8s-app: metrics-server
19     spec:
20       containers:
21       - args:
22         - --cert-dir=/tmp
23         - --secure-port=4443
24         - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
25         - --kubelet-use-node-status-port
26         - --metric-resolution=15s
27         # 修改部分
28         - --kubelet-insecure-tls # 使用非安全的协议
29       image: bitnami/metrics-server:0.6.1 # k8s.gcr.io/metrics-
server/metrics-server:v0.6.1

```

- 安装

```
1 kubectl apply -f components.yaml
```

- 查看pod运行情况

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

- 使用kubectl top node 查看资源使用情况

```
1 kubectl top node [--use-protocol-buffers]
```

```
1 kubectl top pod -n kube-system [--use-protocol-buffers]
```

管理和操作 HPA

创建和配置 HPA

要创建 HPA 控制器，您需要提供以下信息：

1. 监视的指标：例如 CPU 使用率或内存使用率。
2. 要自动扩展或缩小的目标对象：例如 Deployment 或 ReplicaSet。
3. 要保持的最小和最大 Pod 数量。
4. 触发自动伸缩的阈值。

以下是一个示例 HPA 控制器的 YAML 文件：

```

1 apiVersion: autoscaling/v2beta2
2 kind: HorizontalPodAutoscaler
3 metadata:

```

```

4   name: example-app
5   spec:
6     scaleTargetRef:
7       apiVersion: apps/v1
8       kind: Deployment
9       name: example-app
10  minReplicas: 2
11  maxReplicas: 10
12  metrics:
13  - type: Resource
14    resource:
15      name: cpu
16      targetAverageUtilization: 50

```

在上面的示例中，HPA 控制器将监视名为 `example-app` 的 Deployment 的 CPU 使用率，并在平均使用率达到 50% 时自动扩展 Pod 数量，以保持最小 2 个 Pod，最大 10 个 Pod。

监控和自动伸缩 Pod 数量

一旦创建了 HPA 控制器，它将开始监视指定的指标，并根据规则自动扩展或缩小 Pod 数量。您可以使用以下命令来查看 HPA 控制器的状态：

```
1 | kubectl get hpa
```

此命令将返回 HPA 控制器的当前状态，包括监视的指标、目标对象、当前 Pod 数量和自动伸缩规则。

更新和删除 HPA

要更新 HPA 控制器的规则，您可以使用以下命令：

```
1 | kubectl edit hpa example-app
```

此命令将打开 HPA 控制器的 YAML 文件，您可以在其中更新监视的指标、Pod 数量等信息。

要删除 HPA 控制器，您可以使用以下命令：

```
1 | kubectl delete hpa example-app
```

这将删除名为 `example-app` 的 HPA 控制器。

示例：使用 HPA 控制器进行自动伸缩和负载管理

以下是一个使用 HPA 控制器进行自动伸缩和负载管理的示例：

- 创建一个 Deployment：

```
1 | kubectl create deployment example-app --image=nginx
```

- 创建一个 HPA 控制器：

```

1   apiVersion: autoscaling/v2beta2
2   kind: HorizontalPodAutoscaler
3   metadata:
4     name: example-app
5   spec:

```



```
6   scaleTargetRef:
7     apiVersion: apps/v1
8     kind: Deployment
9     name: example-app
10  minReplicas: 2
11  maxReplicas: 10
12  metrics:
13  - type: Resource
14    resource:
15      name: cpu
16      targetAverageUtilization: 50
```

- 使用 `kubectl get hpa` 命令查看 HPA 控制器的状态。
- 使用 `kubectl run` 命令创建一个负载测试：

```
1 kubectl run -i --tty load-generator --image=busybox /bin/sh
```

- 在负载测试中的 shell 中运行以下命令来生成负载：

```
1 while true; do wget -q -O- http://example-app; done
```

这将使用 `wget` 命令不断地向 `example-app` 发送请求，从而增加负载。

- 在另一个 shell 中使用 `kubectl get pods` 命令来查看 Pod 的数量是否自动扩展。
- 在负载测试中的 shell 中使用 `Ctrl+C` 停止负载生成，并使用 `kubectl delete` 命令删除负载测试：

```
1 kubectl delete pod load-generator
```

- 在 HPA 控制器中更新规则，将目标 CPU 使用率从 50% 提高到 80%：

```
1 kubectl edit hpa example-app
```

- 再次开始负载测试，并使用 `kubectl get pods` 命令查看 Pod 的数量是否自动扩展。
- 在负载测试中的 shell 中使用 `Ctrl+C` 停止负载生成，并使用 `kubectl delete` 命令删除负载测试。

服务发现与负载均衡

服务发现的概念和实现

在分布式系统中，服务发现是自动识别和定位可用服务实例的过程。它允许其他服务或组件动态地找到和访问所需的服务，而无需显式配置具体的网络地址或位置。

服务发现的实现通常涉及以下关键组件：

1. 服务注册：服务在启动时将自身信息注册到服务注册表中。注册表记录了服务的网络位置、元数据和其他有关服务实例的信息。
2. 服务发现：其他服务或客户端通过查询服务注册表来获取所需服务的信息。它可以根据服务名称、标签或其他属性进行查找。
3. 心跳和健康检查：服务实例定期发送心跳信号，以向注册表明它们的健康状态。服务发现机制可以使用健康检查来确定服务实例是否可用。

常见的服务发现解决方案包括基于 DNS 的解决方案、专用的服务发现工具（如Consul）和容器编排平台（如Kubernetes）提供的内置服务发现机制。

什么是服务发现和负载均衡？

在分布式系统中，服务发现是自动识别和定位可用服务实例的过程。它允许其他服务或组件动态地找到和访问所需的服务，而无需显式配置具体的网络地址或位置。

服务发现的实现通常涉及以下关键组件：

1. 服务注册：服务在启动时将自身信息注册到服务注册表中。注册表记录了服务的网络位置、元数据和其他有关服务实例的信息。
2. 服务发现：其他服务或客户端通过查询服务注册表来获取所需服务的信息。它可以根据服务名称、标签或其他属性进行查找。
3. 心跳和健康检查：服务实例定期发送心跳信号，以向注册表表明它们的健康状态。服务发现机制可以使用健康检查来确定服务实例是否可用。

常见的服务发现解决方案包括基于 DNS 的解决方案、专用的服务发现工具（如Consul）和容器编排平台（如Kubernetes）提供的内置服务发现机制。

为什么需要 Kubernetes 的服务发现和负载均衡？

Kubernetes 提供了内置的服务发现和负载均衡功能，带来了许多好处：

1. 动态环境：Kubernetes 环境中的服务实例可以动态启动、停止或扩展。服务发现机制可以自动感知这些变化，并确保其他服务能够及时找到和与目标服务通信。
2. 透明的服务访问：通过 Kubernetes 的服务发现机制，服务可以使用统一的 DNS 名称或虚拟 IP 地址来访问其他服务，而无需了解具体的服务实例。这简化了应用程序的配置和维护，提供了更灵活和透明的服务访问方式。
3. 负载均衡：Kubernetes 提供了内置的负载均衡功能，通过将请求自动分发到可用的服务实例，提高了应用程序的可扩展性和性能。负载均衡算法可以根据实际需求进行配置，以适应不同的负载均衡策略。
4. 健康检查和故障转移：Kubernetes 通过定期的健康检查来监控服务实例的状态，并自动将请求从不健康或故障的实例转移到其他健康的实例上。这提高了应用程序的可用性和容错性，减少了由于服务实例故障而导致的服务中断。
5. 无缝的服务扩展：Kubernetes 允许根据负载情况动态

Kubernetes 服务发现

在分布式系统中，服务发现是一项关键功能，它允许应用程序自动发现和其他服务进行通信。在 Kubernetes 中，服务发现是通过一些机制来实现的，包括基于环境变量和基于 DNS 的服务发现。本文将介绍 Kubernetes 的服务概述，以及如何在 Kubernetes 中发现和使用服务。

Kubernetes 的服务概述

在 Kubernetes 中，服务是一组提供相同功能的 Pod 的抽象。它们具有唯一的名称和虚拟 IP（ClusterIP），可以由其他应用程序使用该 IP 来访问服务。服务对象是 Kubernetes 中的资源对象，用于定义服务的属性，如名称、选择器、端口等。

Kubernetes 如何发现服务

在 Kubernetes 中，有多种方式可以发现和访问服务。下面介绍了两种常用的方式：基于环境变量和基于 DNS。

基于环境变量的服务发现

在 Kubernetes 中，每个容器都可以通过环境变量来获取与其关联的服务的信息。当创建服务时，Kubernetes 会自动创建相关的环境变量，并将服务的名称、IP 地址和端口等信息注入到容器的环境变量中。通过读取这些环境变量，应用程序可以发现并与所需的服务进行通信。

例如，在容器的配置文件中可以使用环境变量来指定服务的访问地址和端口：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp-pod
5  spec:
6    containers:
7      - name: myapp-container
8        image: myapp-image
9        env:
10         - name: DATABASE_HOST
11           value: $(MYAPP_SERVICE_HOST)
12         - name: DATABASE_PORT
13           value: $(MYAPP_SERVICE_PORT)
```

对于上述示例中的环境变量 `DATABASE_HOST` 和 `DATABASE_PORT`，它们的值将根据实际情况而定。下面是对应情况的解释：

- `DATABASE_HOST`：该环境变量的值是服务的 IP 地址。在 Kubernetes 内部，服务的 IP 地址将用于容器内部的通信，应用程序可以直接使用这个 IP 地址来访问服务。
- `DATABASE_PORT`：该环境变量的值是服务的端口号。它指定了应用程序可以使用的端口，通过这个端口可以与服务建立连接并进行通信。

在示例中，`$(MYAPP_SERVICE_HOST)` 和 `$(MYAPP_SERVICE_PORT)` 是环境变量的占位符，它们会被 Kubernetes 替换为实际的服务信息。在容器启动时，Kubernetes 会将这些占位符替换为实际的服务 IP 地址和端口号。

基于 DNS 的服务发现

Kubernetes 使用内置的 DNS 服务来提供基于 DNS 的服务发现功能。每个服务都有一个 DNS 记录，可以通过服务名称进行解析。应用程序可以使用服务名称作为主机名来访问其他服务，无需关注具体的 IP 地址。

让我们以一个具体的示例来说明。假设有一个名为 `myapp-service` 的服务，它所在的命名空间是 `default`。在这种情况下，Kubernetes 会为该服务分配一个 DNS 名称：`myapp-service.default.svc.cluster.local`。

Service 域名格式：`$(service name).$(namespace).svc.cluster.local`，其中 `cluster.local` 为指定的集群的域名

例如，在容器中使用服务名称来访问其他服务：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp-pod
5  spec:
6    containers:
7      - name: myapp-container
8        image: myapp-image
9        command: ["sh", "-c", "curl http://myapp-
    service.default.svc.cluster.local:8080"]

```

通过使用服务名称作为主机名，容器可以直接访问与 `myapp-service` 关联的服务。

在 Kubernetes 中使用服务发现

- 创建服务：首先，您需要创建一个服务来公开您的应用程序或微服务。使用 Kubernetes 的 Service 资源来定义服务。在 Service 中指定选择器以关联服务和相应的 Pod。

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: myapp-service
5  spec:
6    selector:
7      app: myapp
8    ports:
9      - name: http
10        protocol: TCP
11        port: 8080
12        targetPort: 8080

```

上述示例中创建了一个名为 `myapp-service` 的服务，它使用标签选择器 `app: myapp` 关联到具有相同标签的 Pod。服务将在端口 8080 上监听请求，并将其转发到 Pod 的端口 8080。

- 定义环境变量：在您的应用程序容器中，您可以定义环境变量，使用服务名称作为主机名来访问其他服务。这些环境变量将由 Kubernetes 自动创建并注入到容器中。

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp-pod
5  spec:
6    containers:
7      - name: myapp-container
8        image: myapp-image
9        env:
10          - name: DATABASE_HOST
11            value: myapp-service.default.svc.cluster.local
12          - name: DATABASE_PORT
13            value: "8080"

```

在上述示例中，`myapp-container` 容器中的环境变量 `DATABASE_HOST` 设置为 `myapp-service.default.svc.cluster.local`，`DATABASE_PORT` 设置为端口号 8080。这样，您的应用程序容器可以使用这些环境变量来访问服务。

- 服务间通信：在您的应用程序中，使用服务名称作为主机名和相应的端口号来访问其他服务。这样，Kubernetes 的内置 DNS 服务会将服务名称解析为实际的 IP 地址和端口，并将请求转发到相应的服务。例如，在应用程序的代码中，您可以使用以下方式来访问服务：

```
1 import requests
2
3 database_host = os.environ.get("DATABASE_HOST")
4 database_port = os.environ.get("DATABASE_PORT")
5
6 response = requests.get(f"http://{database_host}:{database_port}")
```

这里的 `database_host` 将解析为服务的实际 IP 地址，`database_port` 将是端口号 `8080`。您可以使用这些值构建请求 URL 来与服务进行通信。

综上所述，在这个示例中，`myapp-service.default.svc.cluster.local` 是服务的完整 DNS 名称，可以通过环境变量传递给应用程序，应用程序可以使用这些环境变量来构建正确的服务访问 URL。

Kubernetes 负载均衡

在 Kubernetes 中，负载均衡是一种关键的功能，用于分发流量到后端应用程序的多个副本（Pod）之间，以提高可靠性和性能。Kubernetes 提供了多种负载均衡的方式和策略，可以根据需求进行选择。

Kubernetes 负载均衡概述

负载均衡是将流量均匀地分发到多个后端服务副本，以避免单个副本过载并提供高可用性。在 Kubernetes 中，负载均衡是通过 Service 对象来实现的。Service 定义了一个虚拟的稳定网络终结点，它通过选择器（Selector）将流量路由到一组相应的 Pod 上。

Kubernetes 如何实现负载均衡

Kubernetes 实现负载均衡的关键在于 Service 对象和其后端 Pod 的关联。Service 使用标签选择器来选择一组 Pod，并为它们分配一个虚拟 IP 地址和端口。当流量到达 Service 的虚拟 IP 地址和端口时，Kubernetes 会将请求转发给选择器所选择的 Pod，从而实现负载均衡。

Kubernetes 负载均衡的类型

在 Kubernetes 中，有几种常见的负载均衡类型，具体取决于使用的负载均衡算法和策略。

Round-robin 负载均衡

Round-robin 是 Kubernetes 默认的负载均衡算法。它按照顺序将请求依次分配给后端 Pod，确保每个 Pod 都有机会处理请求。这种方式适用于大部分的负载均衡场景。以下是一个示例：

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: myapp-service
5 spec:
6   selector:
7     app: myapp
8   ports:
9     - protocol: TCP
10     port: 8080
11     targetPort: 8080
```

在这个示例中，`myapp-service` 是一个 Service 资源，它将流量分发到带有标签 `app: myapp` 的后端 Pod 上。

Session-based 负载均衡

Session-based 负载均衡算法基于会话（Session）来进行请求分发。Kubernetes 提供了 Session Affinity（会话亲和）的功能来实现此类型的负载均衡。以下是一个示例：

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: myapp-service
5  spec:
6    selector:
7      app: myapp
8    sessionAffinity: ClientIP
9    ports:
10     - protocol: TCP
11       port: 8080
12       targetPort: 8080
```

在这个示例中，我们通过将 `sessionAffinity` 设置为 `ClientIP`，启用了基于客户端 IP 地址的会话亲和。这意味着来自同一 IP 地址的请求会被发送到同一个后端 Pod，以保持会话的一致性。

IP-based 负载均衡

IP-based 负载均衡算法根据客户端的 IP 地址来进行请求分发。在 Kubernetes 中，可以使用 `ExternalTrafficPolicy`（外部流量策略）来实现该功能。以下是一个示例：

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: myapp-service
5  spec:
6    selector:
7      app: myapp
8    externalTrafficPolicy: Local
9    ports:
10     - protocol: TCP
11       port: 8080
12       targetPort: 8080
```

在这个示例中，我们通过将 `externalTrafficPolicy` 设置为 `Local`，启用了基于客户端 IP 地址的负载均衡。这样，负载均衡器会将流量直接发送到目标节点，而不是通过集群内部的 IPVS 负载均衡。

Weighted 负载均衡

Weighted 负载均衡算法根据权重分配请求给后端 Pod。您可以使用不同的工具和插件来实现权重负载均衡，具体取决于您的使用场景和需求。以下是一个示例：

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: myapp-service
5  spec:
```

```

6   selector:
7     app: myapp
8   ports:
9     - protocol: TCP
10      port: 8080
11      targetPort: 8080
12   externalTrafficPolicy: Local
13   sessionAffinity: None
14   loadBalancerIP: 10.0.0.100
15   loadBalancerSourceRanges:
16     - 192.168.0.0/16
17   loadBalancerBackendWeights:
18     app-pod-1: 50
19     app-pod-2: 30
20     app-pod-3: 20

```

在这个示例中，我们使用了 `loadBalancerBackendWeights` 注解来配置权重负载均衡。根据配置，50% 的请求将发送到 `app-pod-1`，30% 的请求将发送到 `app-pod-2`，20% 的请求将发送到 `app-pod-3`。

Kubernetes 服务发现和负载均衡的具体实现

Kubernetes Service 和 Endpoint

Kubernetes Service是一种抽象，用于定义一组应用程序的逻辑集合。它为这组应用程序提供了一个稳定的入口，并可通过该入口进行访问。而Endpoint是Service所代理的真实后端Pod的IP地址和端口。

当你创建一个Service时，Kubernetes会自动创建一个对应的Endpoint。**Endpoint包含了Service所指向的Pod的IP地址和端口，这样当Service接收到请求时，它会将请求转发到这些Endpoint所代表的Pod上，从而实现了服务发现和负载均衡。**

Kubernetes Service 的类型

ClusterIP

假设你有一个简单的应用程序，它运行在一个Deployment中，包含两个Pod，并且暴露端口为8080。你可以创建一个ClusterIP类型的Service来暴露这个应用程序：**【集群内部访问】**

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-app-service
5  spec:
6    selector:
7      app: my-app
8    ports:
9      - protocol: TCP
10        port: 80
11        targetPort: 8080

```

在这个例子中，我们创建了一个名为 `my-app-service` 的Service，并指定了它的选择器（selector）为 `app: my-app`，这样它就会代理带有标签 `app=my-app` 的Pod。通过Service，你可以通过 `my-app-service` 的ClusterIP地址（例如：`10.0.0.1`）访问这两个Pod。

NodePort

假设你希望将上述的 `my-app-service` 在集群外部暴露，使得外部用户可以通过节点的IP地址和指定的端口来访问该服务。你可以创建一个NodePort类型的Service：【外网访问内部】

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-app-service
5  spec:
6    selector:
7      app: my-app
8    ports:
9      - protocol: TCP
10       port: 80
11       targetPort: 8080
12    type: NodePort
```

Kubernetes会自动在集群的每个节点上打开一个随机端口（例如：32000），然后将这个端口映射到Service的端口（在这个例子中为80）。现在，你可以通过任何节点的IP地址加上映射的端口（例如：`http://<Node-IP>:32000`）来访问该Service。

LoadBalancer

假设你在云服务提供商上部署你的Kubernetes集群，并希望将你的应用程序暴露给公网。你可以创建一个LoadBalancer类型的Service：

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-app-service
5  spec:
6    selector:
7      app: my-app
8    ports:
9      - protocol: TCP
10       port: 80
11       targetPort: 8080
12    type: LoadBalancer
```

在这个例子中，Kubernetes会与云服务提供商协作，在该云平台上创建一个外部负载均衡器，并将该负载均衡器配置到Service的ClusterIP地址上。现在，你可以通过负载均衡器公开的IP地址来访问你的应用程序。

ExternalName

假设你的Kubernetes集群中有一个应用程序，它需要访问集群外部的某个服务或域名，例如数据库、消息队列或外部API。在这种情况下，你可以使用ExternalName类型的Service来创建一个Kubernetes Service，将Service名称映射到集群外部的DNS名称。

示例：假设你有一个外部服务（例如外部数据库），其DNS名称为 `db.example.com`，并且你希望在Kubernetes集群内部通过 `my-db-service` 这个Service名称来访问它。

你可以使用ExternalName类型的Service来实现这个映射：


```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-db-service
5  spec:
6    type: ExternalName
7    externalName: db.example.com
```

在这个示例中，我们创建了一个名为 `my-db-service` 的 Service，并将其类型设置为 `ExternalName`。同时，我们将 `externalName` 字段设置为 `db.example.com`，这样当你在集群内部通过 `my-db-service` 来访问该 Service 时，Kubernetes 会将其解析为 `db.example.com`。

ClusterIP: None (Headless Service)

"ClusterIP: None" 类型的 Service 是一种特殊的 Service 类型，它在一定程度上与其他类型的 Service 不同。在 "ClusterIP: None" 类型的 Service 中，Kubernetes 不会为 Service 分配一个 ClusterIP，而是会将 DNS 域名直接指向后端 Pod 的 IP 地址。

- 示例配置：假设你有一个 Deployment，其中包含了一组 Pod，并且你希望将这些 Pod 通过 Service 暴露出去，但不需要为 Service 分配 ClusterIP。你可以创建一个 "ClusterIP: None" 类型的 Service：

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-app-service
5  spec:
6    selector:
7      app: my-app
8    clusterIP: None
9    ports:
10     - protocol: TCP
11       port: 80
12       targetPort: 8080
```

在这个例子中，我们创建了一个名为 `my-app-service` 的 Service，并将其 `clusterIP` 设置为 `None`，从而声明它为 "ClusterIP: None" 类型的 Service。这样，Kubernetes 不会分配 ClusterIP，并且 DNS 域名将直接指向后端 Pod 的 IP 地址。

- Headless Service 的 DNS 解析：

在 "ClusterIP: None" 类型的 Service 中，DNS 解析将返回所有后端 Pod 的 IP 地址列表。每个 Pod 的 DNS 记录将采用以下格式：

```
1  <pod-name>.<service-name>.<namespace>.svc.cluster.local
```

这将返回一个 A 记录（IPv4 地址）或 AAAA 记录（IPv6 地址），指向相应的 Pod 的 IP 地址。

- 使用 Headless Service：

"ClusterIP: None" 类型的 Service 通常用于以下场景：

1. StatefulSets：当使用 StatefulSets 来运行有状态的应用程序时，可以使用 "ClusterIP: None" 类型的 Service 来为每个 Pod 分配一个唯一的 DNS 名称，以实现有状态应用程序的访问和服务发现。
2. 自定义 DNS："ClusterIP: None" 类型的 Service 也可以用于实现自定义的 DNS 服务，将特定域名解析为一组 Pod 的 IP 地址。

3. 网络代理：在某些情况下，你可能需要直接访问后端Pod的IP地址，而不经Service的ClusterIP。这时可以使用"ClusterIP: None"类型的Service来实现。

Pod 的 hostname 和 subdomain 字段

hostname

`hostname` 字段用于为Pod设置一个特定的主机名。当你设置了 `hostname` 字段后，该Pod的主机名将被设置为 `hostname` 字段指定的值。这样，在Pod内部，你可以通过该主机名来访问自己。请注意，Pod的主机名必须符合DNS标准的主机名格式。示例：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    hostname: my-pod-hostname
7    containers:
8      - name: my-container
9        image: nginx:latest
```

在这个示例中，我们创建了一个名为 `my-pod` 的Pod，并设置了 `hostname` 字段为 `my-pod-hostname`。在Pod内部，你可以使用 `my-pod-hostname` 这个主机名来访问自己。

subdomain

`subdomain` 字段用于将Pod的主机名添加到Service的DNS解析中，以实现跨命名空间的服务发现。

`subdomain` 的值必须与Service的 `name` 字段相同。这样，如果你在Pod中访问Service的名称，它将被解析为 `<service-name>.<subdomain>.<namespace>.svc.cluster.local`。示例：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    subdomain: my-subdomain
7    containers:
8      - name: my-container
9        image: nginx:latest
```

在这个示例中，我们创建了一个名为 `my-pod` 的Pod，并设置了 `subdomain` 字段为 `my-subdomain`。这样，如果在Pod中访问名为 `my-service` 的Service，它将被解析为 `my-service.my-subdomain.<namespace>.svc.cluster.local`。

需要注意的是，`subdomain` 字段只有在Pod是由Deployment、StatefulSet或DaemonSet创建的时候才会生效。对于直接创建的Pod，`subdomain` 字段会被忽略。

综上所述，`hostname` 字段用于设置Pod的主机名，`subdomain` 字段用于在Service的DNS解析中添加子域名，以实现跨命名空间的服务发现。

注意的是，一旦设置了 `hostname`，那么该 Pod 的主机名就被设置为 `hostname` 的值，而 `subdomain` 需要和 `svc` 中的 `name` 相同。

Kubernetes Ingress

为什么需要Ingress

- Service 可以使用 NodePort 暴露集群外访问端口，但是性能低、不安全并且端口的范围有限。
- Service 缺少七层（OSI 网络模型）的统一访问入口，负载均衡能力很低，不能做限流、验证非法用户、链路追踪等等。
- Ingress 公开了从集群外部到集群内服务的 HTTP 和 HTTPS 路由。流量路由由 Ingress 资源上定义的规则控制。
- 使用 Ingress 作为整个集群统一的入口，配置 Ingress 规则转发到对应的 Service 。

ingress 安装

- 自建集群采用[裸金属安装方式](#)。
- 下载

```
1 | wget https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.2/deploy/static/provider/baremetal/deploy.yaml
```

- 需要做如下修改：
 - 修改 `k8s.gcr.io/ingress-nginx/controller:v1.1.2` 镜像为 `registry.cn-hangzhou.aliyuncs.com/google_containers/nginx-ingress-controller:v1.1.2`。
 - 修改 `k8s.gcr.io/ingress-nginx/kube-webhook-certgen:v1.1.1` 镜像为 `registry.cn-hangzhou.aliyuncs.com/google_containers/kube-webhook-certgen:v1.1.1`。
 - 修改 Service 为 ClusterIP，无需 NodePort 模式。
 - 修改 Deployment 为 DaemonSet。
 - 修改 Container 使用主机网络，直接在主机上开辟 80、443 端口，无需中间解析，速度更快，所有各个节点的 80 和 443 端口不能被其它进程占用。

```
app.kubernetes.io/name: ingress-nginx
```
- spec:
 - `dnsPolicy: ClusterFirstWithHostNet` # dns 调整为主机网络，原先为 `ClusterFirst`
 - `hostNetwork: true` # 直接让 nginx 占用本机的 80 和 443 端口，这样就可以使用主机网络
- containers:
 - containers 使用主机网络，对应的 dnsPolicy 策略也需要改为主机网络的。
 - 修改 DaemonSet 的 `nodeSelector: ingress-node=true`。这样只需要给 node 节点打上 `ingress-node=true` 标签，即可快速的加入或剔除 ingress-controller 的数量。

```
nodeSelector:
  node-role: ingress # 以后只需要给某个 node 打上这个标签就可以部署 ingress-nginx 到这个节点上了
```

- 创建文件

```
1 | vi deploy.yaml
```

- deploy.yaml 修改的内容如下

```
1 | #GENERATED FOR K8S 1.20
2 | apiVersion: v1
3 | kind: Namespace
4 | metadata:
5 |   labels:
6 |     app.kubernetes.io/instance: ingress-nginx
7 |     app.kubernetes.io/name: ingress-nginx
8 |   name: ingress-nginx
9 | ---
```

```
10 apiVersion: v1
11 automountServiceAccountToken: true
12 kind: ServiceAccount
13 metadata:
14   labels:
15     app.kubernetes.io/component: controller
16     app.kubernetes.io/instance: ingress-nginx
17     app.kubernetes.io/managed-by: Helm
18     app.kubernetes.io/name: ingress-nginx
19     app.kubernetes.io/part-of: ingress-nginx
20     app.kubernetes.io/version: 1.1.2
21     helm.sh/chart: ingress-nginx-4.0.18
22   name: ingress-nginx
23   namespace: ingress-nginx
24 ---
25 apiVersion: v1
26 kind: ServiceAccount
27 metadata:
28   annotations:
29     helm.sh/hook: pre-install,pre-upgrade,post-install,post-upgrade
30     helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
31   labels:
32     app.kubernetes.io/component: admission-webhook
33     app.kubernetes.io/instance: ingress-nginx
34     app.kubernetes.io/managed-by: Helm
35     app.kubernetes.io/name: ingress-nginx
36     app.kubernetes.io/part-of: ingress-nginx
37     app.kubernetes.io/version: 1.1.2
38     helm.sh/chart: ingress-nginx-4.0.18
39   name: ingress-nginx-admission
40   namespace: ingress-nginx
41 ---
42 apiVersion: rbac.authorization.k8s.io/v1
43 kind: Role
44 metadata:
45   labels:
46     app.kubernetes.io/component: controller
47     app.kubernetes.io/instance: ingress-nginx
48     app.kubernetes.io/managed-by: Helm
49     app.kubernetes.io/name: ingress-nginx
50     app.kubernetes.io/part-of: ingress-nginx
51     app.kubernetes.io/version: 1.1.2
52     helm.sh/chart: ingress-nginx-4.0.18
53   name: ingress-nginx
54   namespace: ingress-nginx
55 rules:
56 - apiGroups:
57   - ""
58   resources:
59   - namespaces
60   verbs:
61   - get
62 - apiGroups:
63   - ""
64   resources:
```

```
65 - configmaps
66 - pods
67 - secrets
68 - endpoints
69 verbs:
70 - get
71 - list
72 - watch
73 - apiGroups:
74 - ""
75 resources:
76 - services
77 verbs:
78 - get
79 - list
80 - watch
81 - apiGroups:
82 - networking.k8s.io
83 resources:
84 - ingresses
85 verbs:
86 - get
87 - list
88 - watch
89 - apiGroups:
90 - networking.k8s.io
91 resources:
92 - ingresses/status
93 verbs:
94 - update
95 - apiGroups:
96 - networking.k8s.io
97 resources:
98 - ingressclasses
99 verbs:
100 - get
101 - list
102 - watch
103 - apiGroups:
104 - ""
105 resourceName:
106 - ingress-controller-leader
107 resources:
108 - configmaps
109 verbs:
110 - get
111 - update
112 - apiGroups:
113 - ""
114 resources:
115 - configmaps
116 verbs:
117 - create
118 - apiGroups:
119 - ""
```

```
120 resources:
121   - events
122 verbs:
123   - create
124   - patch
125 ---
126 apiVersion: rbac.authorization.k8s.io/v1
127 kind: Role
128 metadata:
129   annotations:
130     helm.sh/hook: pre-install,pre-upgrade,post-install,post-upgrade
131     helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
132   labels:
133     app.kubernetes.io/component: admission-webhook
134     app.kubernetes.io/instance: ingress-nginx
135     app.kubernetes.io/managed-by: Helm
136     app.kubernetes.io/name: ingress-nginx
137     app.kubernetes.io/part-of: ingress-nginx
138     app.kubernetes.io/version: 1.1.2
139     helm.sh/chart: ingress-nginx-4.0.18
140   name: ingress-nginx-admission
141   namespace: ingress-nginx
142 rules:
143   - apiGroups:
144     - ""
145     resources:
146     - secrets
147     verbs:
148     - get
149     - create
150   ---
151 apiVersion: rbac.authorization.k8s.io/v1
152 kind: ClusterRole
153 metadata:
154   labels:
155     app.kubernetes.io/instance: ingress-nginx
156     app.kubernetes.io/managed-by: Helm
157     app.kubernetes.io/name: ingress-nginx
158     app.kubernetes.io/part-of: ingress-nginx
159     app.kubernetes.io/version: 1.1.2
160     helm.sh/chart: ingress-nginx-4.0.18
161   name: ingress-nginx
162 rules:
163   - apiGroups:
164     - ""
165     resources:
166     - configmaps
167     - endpoints
168     - nodes
169     - pods
170     - secrets
171     - namespaces
172     verbs:
173     - list
174     - watch
```

```
175 - apiGroups:
176   - ""
177   resources:
178   - nodes
179   verbs:
180   - get
181 - apiGroups:
182   - ""
183   resources:
184   - services
185   verbs:
186   - get
187   - list
188   - watch
189 - apiGroups:
190   - networking.k8s.io
191   resources:
192   - ingresses
193   verbs:
194   - get
195   - list
196   - watch
197 - apiGroups:
198   - ""
199   resources:
200   - events
201   verbs:
202   - create
203   - patch
204 - apiGroups:
205   - networking.k8s.io
206   resources:
207   - ingresses/status
208   verbs:
209   - update
210 - apiGroups:
211   - networking.k8s.io
212   resources:
213   - ingressclasses
214   verbs:
215   - get
216   - list
217   - watch
218 ---
219 apiVersion: rbac.authorization.k8s.io/v1
220 kind: ClusterRole
221 metadata:
222   annotations:
223     helm.sh/hook: pre-install,pre-upgrade,post-install,post-upgrade
224     helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
225   labels:
226     app.kubernetes.io/component: admission-webhook
227     app.kubernetes.io/instance: ingress-nginx
228     app.kubernetes.io/managed-by: Helm
229     app.kubernetes.io/name: ingress-nginx
```

```
230     app.kubernetes.io/part-of: ingress-nginx
231     app.kubernetes.io/version: 1.1.2
232     helm.sh/chart: ingress-nginx-4.0.18
233     name: ingress-nginx-admission
234 rules:
235 - apiGroups:
236   - admissionregistration.k8s.io
237   resources:
238   - validatingwebhookconfigurations
239   verbs:
240   - get
241   - update
242 ---
243 apiVersion: rbac.authorization.k8s.io/v1
244 kind: RoleBinding
245 metadata:
246   labels:
247     app.kubernetes.io/component: controller
248     app.kubernetes.io/instance: ingress-nginx
249     app.kubernetes.io/managed-by: Helm
250     app.kubernetes.io/name: ingress-nginx
251     app.kubernetes.io/part-of: ingress-nginx
252     app.kubernetes.io/version: 1.1.2
253     helm.sh/chart: ingress-nginx-4.0.18
254   name: ingress-nginx
255   namespace: ingress-nginx
256 roleRef:
257   apiGroup: rbac.authorization.k8s.io
258   kind: Role
259   name: ingress-nginx
260 subjects:
261 - kind: ServiceAccount
262   name: ingress-nginx
263   namespace: ingress-nginx
264 ---
265 apiVersion: rbac.authorization.k8s.io/v1
266 kind: RoleBinding
267 metadata:
268   annotations:
269     helm.sh/hook: pre-install,pre-upgrade,post-install,post-upgrade
270     helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
271   labels:
272     app.kubernetes.io/component: admission-webhook
273     app.kubernetes.io/instance: ingress-nginx
274     app.kubernetes.io/managed-by: Helm
275     app.kubernetes.io/name: ingress-nginx
276     app.kubernetes.io/part-of: ingress-nginx
277     app.kubernetes.io/version: 1.1.2
278     helm.sh/chart: ingress-nginx-4.0.18
279   name: ingress-nginx-admission
280   namespace: ingress-nginx
281 roleRef:
282   apiGroup: rbac.authorization.k8s.io
283   kind: Role
284   name: ingress-nginx-admission
```



```
285 subjects:
286 - kind: ServiceAccount
287   name: ingress-nginx-admission
288   namespace: ingress-nginx
289 ---
290 apiVersion: rbac.authorization.k8s.io/v1
291 kind: ClusterRoleBinding
292 metadata:
293   labels:
294     app.kubernetes.io/instance: ingress-nginx
295     app.kubernetes.io/managed-by: Helm
296     app.kubernetes.io/name: ingress-nginx
297     app.kubernetes.io/part-of: ingress-nginx
298     app.kubernetes.io/version: 1.1.2
299     helm.sh/chart: ingress-nginx-4.0.18
300   name: ingress-nginx
301 roleRef:
302   apiGroup: rbac.authorization.k8s.io
303   kind: ClusterRole
304   name: ingress-nginx
305 subjects:
306 - kind: ServiceAccount
307   name: ingress-nginx
308   namespace: ingress-nginx
309 ---
310 apiVersion: rbac.authorization.k8s.io/v1
311 kind: ClusterRoleBinding
312 metadata:
313   annotations:
314     helm.sh/hook: pre-install,pre-upgrade,post-install,post-upgrade
315     helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
316   labels:
317     app.kubernetes.io/component: admission-webhook
318     app.kubernetes.io/instance: ingress-nginx
319     app.kubernetes.io/managed-by: Helm
320     app.kubernetes.io/name: ingress-nginx
321     app.kubernetes.io/part-of: ingress-nginx
322     app.kubernetes.io/version: 1.1.2
323     helm.sh/chart: ingress-nginx-4.0.18
324   name: ingress-nginx-admission
325 roleRef:
326   apiGroup: rbac.authorization.k8s.io
327   kind: ClusterRole
328   name: ingress-nginx-admission
329 subjects:
330 - kind: ServiceAccount
331   name: ingress-nginx-admission
332   namespace: ingress-nginx
333 ---
334 apiVersion: v1
335 data:
336   allow-snippet-annotations: "true"
337 kind: ConfigMap
338 metadata:
339   labels:
```

```
340     app.kubernetes.io/component: controller
341     app.kubernetes.io/instance: ingress-nginx
342     app.kubernetes.io/managed-by: Helm
343     app.kubernetes.io/name: ingress-nginx
344     app.kubernetes.io/part-of: ingress-nginx
345     app.kubernetes.io/version: 1.1.2
346     helm.sh/chart: ingress-nginx-4.0.18
347     name: ingress-nginx-controller
348     namespace: ingress-nginx
349 ---
350 apiVersion: v1
351 kind: Service
352 metadata:
353   labels:
354     app.kubernetes.io/component: controller
355     app.kubernetes.io/instance: ingress-nginx
356     app.kubernetes.io/managed-by: Helm
357     app.kubernetes.io/name: ingress-nginx
358     app.kubernetes.io/part-of: ingress-nginx
359     app.kubernetes.io/version: 1.1.2
360     helm.sh/chart: ingress-nginx-4.0.18
361     name: ingress-nginx-controller
362     namespace: ingress-nginx
363 spec:
364   ipFamilies:
365   - IPV4
366   ipFamilyPolicy: SingleStack
367   ports:
368   - appProtocol: http
369     name: http
370     port: 80
371     protocol: TCP
372     targetPort: http
373   - appProtocol: https
374     name: https
375     port: 443
376     protocol: TCP
377     targetPort: https
378   selector:
379     app.kubernetes.io/component: controller
380     app.kubernetes.io/instance: ingress-nginx
381     app.kubernetes.io/name: ingress-nginx
382   type: ClusterIP # NodePort
383 ---
384 apiVersion: v1
385 kind: Service
386 metadata:
387   labels:
388     app.kubernetes.io/component: controller
389     app.kubernetes.io/instance: ingress-nginx
390     app.kubernetes.io/managed-by: Helm
391     app.kubernetes.io/name: ingress-nginx
392     app.kubernetes.io/part-of: ingress-nginx
393     app.kubernetes.io/version: 1.1.2
394     helm.sh/chart: ingress-nginx-4.0.18
```

```
395     name: ingress-nginx-controller-admission
396     namespace: ingress-nginx
397 spec:
398   ports:
399   - appProtocol: https
400     name: https-webhook
401     port: 443
402     targetPort: webhook
403   selector:
404     app.kubernetes.io/component: controller
405     app.kubernetes.io/instance: ingress-nginx
406     app.kubernetes.io/name: ingress-nginx
407   type: ClusterIP
408 ---
409 apiVersion: apps/v1
410 kind: DaemonSet # Deployment
411 metadata:
412   labels:
413     app.kubernetes.io/component: controller
414     app.kubernetes.io/instance: ingress-nginx
415     app.kubernetes.io/managed-by: Helm
416     app.kubernetes.io/name: ingress-nginx
417     app.kubernetes.io/part-of: ingress-nginx
418     app.kubernetes.io/version: 1.1.2
419     helm.sh/chart: ingress-nginx-4.0.18
420   name: ingress-nginx-controller
421   namespace: ingress-nginx
422 spec:
423   minReadySeconds: 0
424   revisionHistoryLimit: 10
425   selector:
426     matchLabels:
427       app.kubernetes.io/component: controller
428       app.kubernetes.io/instance: ingress-nginx
429       app.kubernetes.io/name: ingress-nginx
430   template:
431     metadata:
432       labels:
433         app.kubernetes.io/component: controller
434         app.kubernetes.io/instance: ingress-nginx
435         app.kubernetes.io/name: ingress-nginx
436     spec:
437       dnsPolicy: ClusterFirstWithHostNet # dns 调整为主机网络，原先为
ClusterFirst
438       hostNetwork: true # 直接让 nginx 占用本机的 80 和 443 端口，这样就可以使用
主机网络
439     containers:
440     - args:
441       - /nginx-ingress-controller
442       - --election-id=ingress-controller-leader
443       - --controller-class=k8s.io/ingress-nginx
444       - --ingress-class=nginx
445       - --report-node-internal-ip-address=true
446       - --configmap=$(POD_NAMESPACE)/ingress-nginx-controller
447       - --validating-webhook=:8443
```

```
448 - --validating-webhook-certificate=/usr/local/certificates/cert
449 - --validating-webhook-key=/usr/local/certificates/key
450 env:
451 - name: POD_NAME
452   valueFrom:
453     fieldRef:
454       fieldPath: metadata.name
455 - name: POD_NAMESPACE
456   valueFrom:
457     fieldRef:
458       fieldPath: metadata.namespace
459 - name: LD_PRELOAD
460   value: /usr/local/lib/libmimalloc.so
461 image: registry.cn-hangzhou.aliyuncs.com/google_containers/nginx-
ingress-controller:v1.1.2 # 修改 k8s.gcr.io/nginx-
nginx/controller:v1.1.2@sha256:28b11ce69e57843de44e3db6413e98d09de0f6688e33
d4bd384002a44f78405c
462 imagePullPolicy: IfNotPresent
463 lifecycle:
464   preStop:
465     exec:
466       command:
467         - /wait-shutdown
468 livenessProbe:
469   failureThreshold: 5
470   httpGet:
471     path: /healthz
472     port: 10254
473     scheme: HTTP
474   initialDelaySeconds: 10
475   periodSeconds: 10
476   successThreshold: 1
477   timeoutSeconds: 1
478 name: controller
479 ports:
480 - containerPort: 80
481   name: http
482   protocol: TCP
483 - containerPort: 443
484   name: https
485   protocol: TCP
486 - containerPort: 8443
487   name: webhook
488   protocol: TCP
489 readinessProbe:
490   failureThreshold: 3
491   httpGet:
492     path: /healthz
493     port: 10254
494     scheme: HTTP
495   initialDelaySeconds: 10
496   periodSeconds: 10
497   successThreshold: 1
498   timeoutSeconds: 1
499 resources: # 资源限制
```

```

500         requests:
501             cpu: 100m
502             memory: 90Mi
503         limits:
504             cpu: 500m
505             memory: 500Mi
506     securityContext:
507         allowPrivilegeEscalation: true
508         capabilities:
509             add:
510                 - NET_BIND_SERVICE
511             drop:
512                 - ALL
513         runAsUser: 101
514     volumeMounts:
515     - mountPath: /usr/local/certificates/
516       name: webhook-cert
517       readOnly: true
518     nodeSelector:
519         node-role: ingress # 以后只需要给某个 node 打上这个标签就可以部署
ingress-nginx 到这个节点上了
520         # kubernetes.io/os: linux
521     serviceAccountName: ingress-nginx
522     terminationGracePeriodSeconds: 300
523     volumes:
524     - name: webhook-cert
525       secret:
526         secretName: ingress-nginx-admission
527 ---
528 apiVersion: batch/v1
529 kind: Job
530 metadata:
531     annotations:
532         helm.sh/hook: pre-install,pre-upgrade
533         helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
534     labels:
535         app.kubernetes.io/component: admission-webhook
536         app.kubernetes.io/instance: ingress-nginx
537         app.kubernetes.io/managed-by: Helm
538         app.kubernetes.io/name: ingress-nginx
539         app.kubernetes.io/part-of: ingress-nginx
540         app.kubernetes.io/version: 1.1.2
541         helm.sh/chart: ingress-nginx-4.0.18
542     name: ingress-nginx-admission-create
543     namespace: ingress-nginx
544 spec:
545     template:
546         metadata:
547             labels:
548                 app.kubernetes.io/component: admission-webhook
549                 app.kubernetes.io/instance: ingress-nginx
550                 app.kubernetes.io/managed-by: Helm
551                 app.kubernetes.io/name: ingress-nginx
552                 app.kubernetes.io/part-of: ingress-nginx
553                 app.kubernetes.io/version: 1.1.2

```

```

554     helm.sh/chart: ingress-nginx-4.0.18
555     name: ingress-nginx-admission-create
556     spec:
557       containers:
558       - args:
559         - create
560         - --host=ingress-nginx-controller-admission,ingress-nginx-
controller-admission.$(POD_NAMESPACE).svc
561         - --namespace=$(POD_NAMESPACE)
562         - --secret-name=ingress-nginx-admission
563       env:
564       - name: POD_NAMESPACE
565         valueFrom:
566           fieldRef:
567             fieldPath: metadata.namespace
568       image: registry.cn-hangzhou.aliyuncs.com/google_containers/kube-
webhook-certgen:v1.1.1 # k8s.gcr.io/ingress-nginx/kube-webhook-
certgen:v1.1.1@sha256:64d8c73dca984af206adf9d6d7e46aa550362b1d7a01f3a0a91b2
0cc67868660
569       imagePullPolicy: IfNotPresent
570       name: create
571       securityContext:
572         allowPrivilegeEscalation: false
573       nodeSelector:
574         kubernetes.io/os: linux
575       restartPolicy: OnFailure
576       securityContext:
577         fsGroup: 2000
578         runAsNonRoot: true
579         runAsUser: 2000
580       serviceAccountName: ingress-nginx-admission
581     ---
582     apiVersion: batch/v1
583     kind: Job
584     metadata:
585       annotations:
586         helm.sh/hook: post-install,post-upgrade
587         helm.sh/hook-delete-policy: before-hook-creation,hook-succeeded
588       labels:
589         app.kubernetes.io/component: admission-webhook
590         app.kubernetes.io/instance: ingress-nginx
591         app.kubernetes.io/managed-by: Helm
592         app.kubernetes.io/name: ingress-nginx
593         app.kubernetes.io/part-of: ingress-nginx
594         app.kubernetes.io/version: 1.1.2
595         helm.sh/chart: ingress-nginx-4.0.18
596     name: ingress-nginx-admission-patch
597     namespace: ingress-nginx
598     spec:
599       template:
600         metadata:
601           labels:
602             app.kubernetes.io/component: admission-webhook
603             app.kubernetes.io/instance: ingress-nginx
604             app.kubernetes.io/managed-by: Helm

```

```

605     app.kubernetes.io/name: ingress-nginx
606     app.kubernetes.io/part-of: ingress-nginx
607     app.kubernetes.io/version: 1.1.2
608     helm.sh/chart: ingress-nginx-4.0.18
609     name: ingress-nginx-admission-patch
610 spec:
611   containers:
612   - args:
613     - patch
614     - --webhook-name=ingress-nginx-admission
615     - --namespace=$(POD_NAMESPACE)
616     - --patch-mutating=false
617     - --secret-name=ingress-nginx-admission
618     - --patch-failure-policy=Fail
619     env:
620     - name: POD_NAMESPACE
621       valueFrom:
622         fieldRef:
623           fieldPath: metadata.namespace
624     image: registry.cn-hangzhou.aliyuncs.com/google_containers/kube-
webhook-certgen:v1.1.1 # k8s.gcr.io/ingress-nginx/kube-webhook-
certgen:v1.1.1@sha256:64d8c73dca984af206adf9d6d7e46aa550362b1d7a01f3a0a91b2
0cc67868660
625     imagePullPolicy: IfNotPresent
626     name: patch
627     securityContext:
628       allowPrivilegeEscalation: false
629     nodeSelector:
630       kubernetes.io/os: linux
631     restartPolicy: OnFailure
632     securityContext:
633       fsGroup: 2000
634       runAsNonRoot: true
635       runAsUser: 2000
636     serviceAccountName: ingress-nginx-admission
637 ---
638 apiVersion: networking.k8s.io/v1
639 kind: IngressClass
640 metadata:
641   labels:
642     app.kubernetes.io/component: controller
643     app.kubernetes.io/instance: ingress-nginx
644     app.kubernetes.io/managed-by: Helm
645     app.kubernetes.io/name: ingress-nginx
646     app.kubernetes.io/part-of: ingress-nginx
647     app.kubernetes.io/version: 1.1.2
648     helm.sh/chart: ingress-nginx-4.0.18
649   name: nginx
650 spec:
651   controller: k8s.io/ingress-nginx
652 ---
653 apiVersion: admissionregistration.k8s.io/v1
654 kind: ValidatingWebhookConfiguration
655 metadata:
656   labels:

```

```

657     app.kubernetes.io/component: admission-webhook
658     app.kubernetes.io/instance: ingress-nginx
659     app.kubernetes.io/managed-by: Helm
660     app.kubernetes.io/name: ingress-nginx
661     app.kubernetes.io/part-of: ingress-nginx
662     app.kubernetes.io/version: 1.1.2
663     helm.sh/chart: ingress-nginx-4.0.18
664     name: ingress-nginx-admission
665     webhooks:
666     - admissionReviewVersions:
667       - v1
668       clientConfig:
669         service:
670           name: ingress-nginx-controller-admission
671           namespace: ingress-nginx
672           path: /networking/v1/ingresses
673       failurePolicy: Fail
674       matchPolicy: Equivalent
675       name: validate.nginx.ingress.kubernetes.io
676       rules:
677       - apiGroups:
678         - networking.k8s.io
679         apiVersions:
680         - v1
681         operations:
682         - CREATE
683         - UPDATE
684         resources:
685         - ingresses
686       sideEffects: None

```

- 给 Node 节点打标签

```
1 | kubectl label node node1 node-role=ingress
```

```
1 | kubectl label node node2 node-role=ingress
```

当然，也可以给 Master 节点打标签，但是 **kubeadm 安装 k8s 集群的时候，会给 Master 节点打上污点**，即使打上标签，也不会进行 Pod 的调度；换言之，Ingress 也不会在 Master 节点上安装。

- 安装 Ingress（需要关闭 Node 节点的 80 和 443 端口，不能有其他进程占用）

```
1 | kubectl apply -f deploy.yaml
```

- 验证是否安装成功：只需要在部署了 ingress 的主机，执行如下的命令

```
1 | netstat -ntlp | grep 80
```

```
1 | netstat -ntlp | grep 443
```

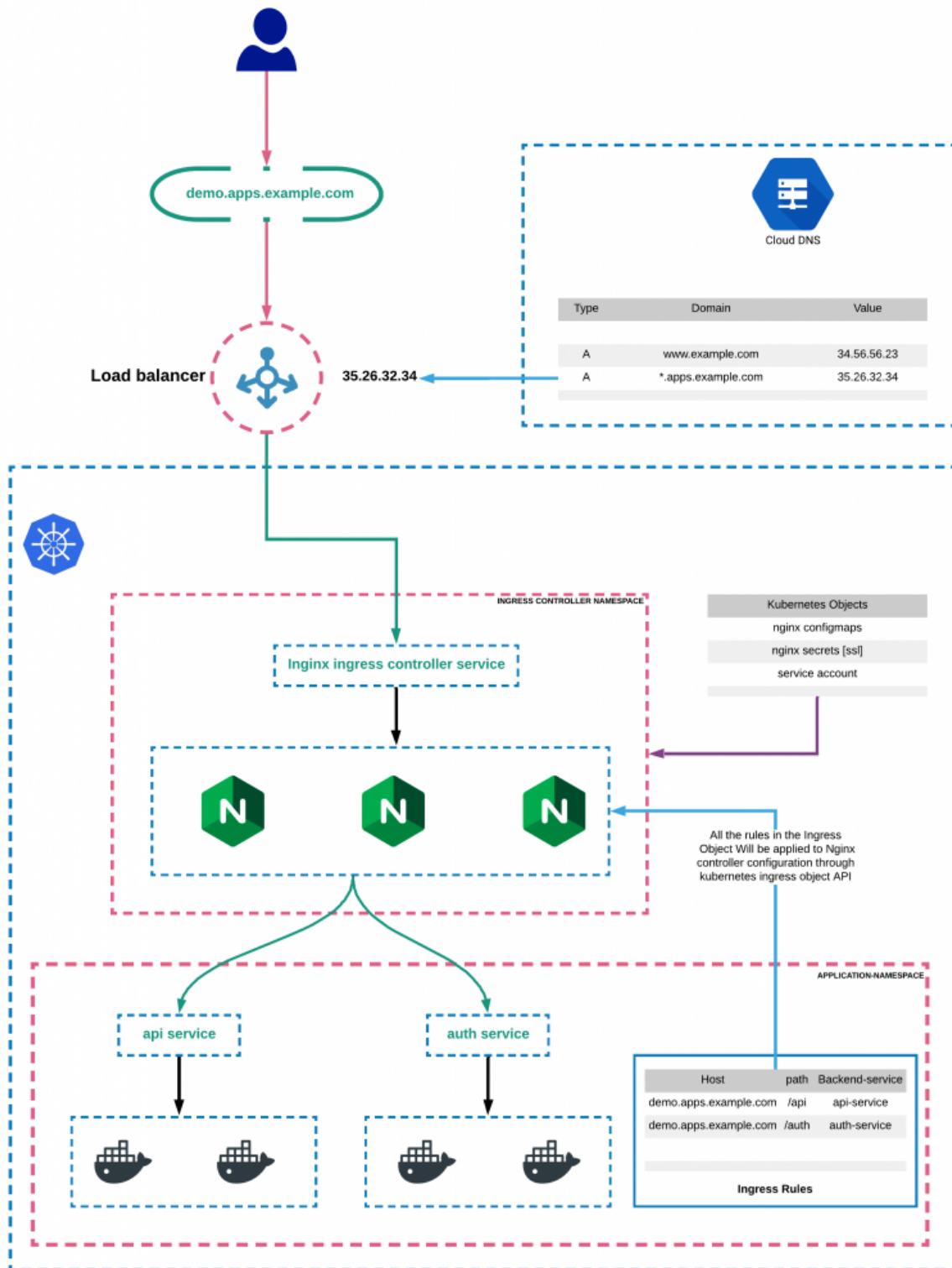


```
[root@node2 ~]# netstat -ntlp | grep :80
tcp        0      0 0.0.0.0:80 0.0.0.0:*    LISTEN     5800/nginx: master
[root@node2 ~]# netstat -ntlp | grep :443
tcp        0      0 0.0.0.0:443 0.0.0.0:*    LISTEN     5800/nginx: master
[root@node2 ~]#
```

- 卸载

```
1 | kubectl delete -f deploy.yaml
```

Ingress 原理



- 用户编写 Ingress 规则，说明哪个域名对应 k8s 集群中的哪个 Service。
- Ingress 控制器动态感知 Ingress 服务规则的变化，然后生成一段对应的 Nginx 的反向代理配置。

- Ingress 控制器会将生成的 Nginx 配置写入到一个运行着的 Nginx 服务中，并动态更新。

Ingress 路由规则

在使用Ingress之前，需要先安装一个Ingress Controller，如Nginx Ingress Controller。然后，你可以定义Ingress资源，并在其中指定路由规则。

示例配置：假设你有两个应用程序，`app1` 和 `app2`，它们分别运行在不同的Deployment中，并且你想通过不同的域名将流量路由到它们。

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: my-ingress
5  spec:
6    rules:
7      - host: app1.example.com
8        http:
9          paths:
10           - path: /foo
11             pathType: Prefix
12             backend:
13               service:
14                 name: app1-service
15                 port:
16                   number: 80
17      - host: app2.example.com
18        http:
19          paths:
20           - path: /bar
21             pathType: Prefix
22             backend:
23               service:
24                 name: app2-service
25                 port:
26                   number: 80
```

在这个例子中，我们创建了一个名为 `my-ingress` 的Ingress资源，并定义了两个规则。第一个规则将来自 `app1.example.com` 的流量路由到名为 `app1-service` 的Service，这个Service代理了 `app1` 应用程序的Pod。同样地，第二个规则将来自 `app2.example.com` 的流量路由到名为 `app2-service` 的Service，代理了 `app2` 应用程序的Pod。

在上述示例中，当来自 `app1.example.com/foo` 的请求到达Ingress时，它将被路由到 `app1-service` 上的80端口。类似地，当来自 `app2.example.com/bar` 的请求到达Ingress时，它将被路由到 `app2-service` 上的80端口。

Ingress的路由规则详解：

字段	含义	默认值
<code>host</code>	指定请求的主机名（域名）。	None
<code>http.paths</code>	定义对应主机名的请求路径与后端Service的映射规则。	None

字段	含义	默认值
<code>http.paths.path</code>	指定请求的URL路径。	None (必需字段)
<code>http.paths.pathType</code>	指定路径匹配类型，可以是 <code>Exact</code> （精确匹配）或 <code>Prefix</code> （前缀匹配）。	Prefix
<code>http.paths.backend.service.name</code>	指定后端Service的名称。	None
<code>http.paths.backend.service.port.number</code>	指定后端Service的端口号。	None

在这个表格中，我们详细列出了每个Ingress路由规则字段的名称、含义以及默认值（如果有的话）。需要注意的是，`http.paths.path` 字段是必需字段，没有默认值，因为它用于指定请求的URL路径，决定了哪些请求会被路由到相应的后端Service。

TLS 加密支持

Kubernetes Ingress提供了TLS（传输层安全）加密支持，通过TLS加密，可以保护在网络上传输的数据的安全性和完整性。这允许你为Ingress资源配置HTTPS路由规则，为你的服务启用安全的加密通信。

- **创建 Secret 对象：** 在Kubernetes中创建一个包含TLS证书和私钥的Secret对象。这些证书和私钥将用于加密和解密HTTPS通信。

```

1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: tls-secret
5  type: kubernetes.io/tls
6  data:
7    tls.crt: BASE64_ENCODED_CERTIFICATE
8    tls.key: BASE64_ENCODED_PRIVATE_KEY

```

在上面的配置中，`tls.crt` 和 `tls.key` 字段是Base64编码的TLS证书和私钥。

或者

```

1  openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out
   tls.crt -subj "/CN=example.com"
2
3  kubectl create secret tls angle-tls --key tls.key --cert tls.crt

```

- **配置 Ingress 对象**

在Ingress资源中定义TLS配置，指定哪些域名需要启用TLS加密。

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: my-ingress
5  annotations:

```

```

6      nginx.ingress.kubernetes.io/rewrite-target: / # 防止http访问跳转到https访问
7 spec:
8   tls:
9     - hosts:
10         - example.com
11       secretName: tls-secret
12 rules:
13   - host: example.com
14     http:
15       paths:
16         - path: /
17           pathType: Prefix
18           backend:
19             service:
20               name: my-service
21             port:
22               number: 80

```

在上面的配置中，我们通过 `tls` 字段定义了TLS配置，指定了需要启用TLS加密的域名和相应的Secret对象。在这个例子中，`example.com` 将使用名为 `tls-secret` 的Secret对象中的TLS证书和私钥来实现TLS加密。

TLS Secret必须位于与Ingress资源相同的命名空间中

当Ingress和Service位于不同的命名空间时，为了确保正确的路由，需要在Ingress中使用完全限定的Service名称，即 `<service-name>.<namespace>`。这样Ingress Controller才能正确地找到后端Service并将请求路由到相应的Pod。

例如，如果Ingress在 `ingress-namespace` 命名空间中，但要路由到 `my-service` 在 `my-namespace` 命名空间中的Pod时，需要在Ingress规则中使用 `my-service.my-namespace` 来指定完全限定的Service名称。

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: my-ingress
5    namespace: ingress-namespace
6  spec:
7    rules:
8      - host: example.com
9        http:
10          paths:
11            - path: /
12              pathType: Prefix
13              backend:
14                service:
15                  name: my-service.my-namespace
16                  port:
17                    number: 80

```

[Nginx Ingress](#)

- 下载kubernetes-ingress

```
1 git clone https://github.com/nginxinc/kubernetes-ingress.git --branch v2.2.0
2
3 cd kubernetes-ingress/deployments
```

- Configure RBAC

```
1 kubectl apply -f common/ns-and-sa.yaml
2 kubectl apply -f rbac/rbac.yaml
3 kubectl apply -f rbac/ap-rbac.yaml
4 kubectl apply -f rbac/apdos-rbac.yaml
```

- 创建公共资源

```
1 openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out
  tls.cert -subj "/CN=test.test.com"
2 kubectl create secret tls ingress-tls --key tls.key --cert tls.cert
3
4
5 kubectl apply -f default-server-secret.yaml
6 kubectl apply -f common/nginx-config.yaml
7 kubectl apply -f common/ingress-class.yaml
```

其中，编辑 default-server-secret.yaml

```
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: ingress-tls
5   namespace: kubescape-system
6   annotations:
7     kubernetes.io/ingress.class: "nginx"
8 spec:
9   tls:
10    - hosts:
11      - test.test.com # 通过浏览器访问
12      secretName: ingress-tls
13   rules:
14    - host: test.test.com
15      http:
16        paths:
17        - path: /
18          pathType: Prefix
19          backend:
20            service:
21              name: ks-console
22              port:
23                number: 80
```

- 部署 NGINX 入口控制器

```
1 kubectl apply -f deployment/nginx-ingress.yaml
2 kubectl apply -f daemon-set/nginx-ingress.yaml
```

- 访问 NGINX 入口控制器

```
1 kubectl create -f service/nodeport.yaml
```

- 卸载 NGINX 入口控制器

```
1 kubectl delete namespace nginx-ingress
2 kubectl delete clusterrole nginx-ingress
3 kubectl delete clusterrolebinding nginx-ingress
4 kubectl delete -f common/crds/
```

Ingress 配置

Ingress中的nginx全局配置

- 方式一：在安装的时候，添加配置。

```
1 apiVersion: v1
2 data:
3   allow-snippet-annotations: "true" # Nginx 的全局配置
4 kind: ConfigMap
5 metadata:
6   labels:
7     app.kubernetes.io/component: controller
8     app.kubernetes.io/instance: ingress-nginx
9     app.kubernetes.io/managed-by: Helm
10    app.kubernetes.io/name: ingress-nginx
11    app.kubernetes.io/part-of: ingress-nginx
12    app.kubernetes.io/version: 1.1.2
13    helm.sh/chart: ingress-nginx-4.0.18
14   name: ingress-nginx-controller
15   namespace: ingress-nginx
```

- 方式二：编辑 cm

```
1 kubectl edit cm ingress-nginx-controller -n ingress-nginx
```

```
1 # 配置项加上
2 data:
3   map-hash-bucket-size: "128" # Nginx 的全局配置
4   ssl-protocols: SSLv2
```

限流配置

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: rate-ingress
5    kubernetes.io/ingress.class: "nginx"
6    nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
7    nginx.ingress.kubernetes.io/limit-rps: "1" # 限流
8  spec:
9    rules:
10   ...

```

默认后端配置

Ingress允许您为未匹配任何已定义规则请求配置默认后端。以下是一个示例：

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ingress-http
5    namespace: default
6    annotations:
7      kubernetes.io/ingress.class: "nginx"
8      nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
9  spec:
10   defaultBackend: # 指定所有未匹配的默认后端
11     service:
12       name: nginx-svc
13       port:
14         number: 80
15   rules:
16     - host: tomcat.com
17       http:
18         paths:
19           - path: /abc
20             pathType: Prefix
21             backend:
22               service:
23                 name: tomcat-svc
24                 port:
25                   number: 8080

```

- tomcat.com 域名的非 /abc 开头的请求，都会转到 defaultBackend。
- 非 tomcat.com 域名下的所有请求，也会转到 defaultBackend。

路径重写配置

Ingress允许您重写传入请求的路径。以下是一个示例：

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: example-ingress
5    annotations:
6      nginx.ingress.kubernetes.io/rewrite-target: /$1
7  spec:

```

```

8   rules:
9   - http:
10      paths:
11      - path: /foo/(?.* )
12        backend:
13          serviceName: foo-service
14          servicePort: 80

```

在上面的示例中，我们将传入请求的路径重写为 `/foo/$1`。匹配路径 `/foo/*` 的请求将被转发到名称为 `foo-service`，端口为 80 的服务。

基于Cookie的配置

Ingress允许您基于Cookie配置路由。以下是一个示例：

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: example-ingress
5    annotations:
6      nginx.ingress.kubernetes.io/affinity: "cookie"
7      nginx.ingress.kubernetes.io/session-cookie-name: "route"
8      # 直接使用 nginx.ingress.kubernetes.io/affinity: "cookie"
9  spec:
10     rules:
11     - host: example.com
12       http:
13         paths:
14         - path: /
15           backend:
16             serviceName: example-service
17             servicePort: 80

```

在上面的示例中，我们将 `affinity` 设置为 `cookie`，将 `session-cookie-name` 设置为 `route`。这意味着请求将根据 `route` cookie 的值路由到相同的后端。

Ingress 示例

以下是一个Ingress的示例，根据主机头将流量路由到两个不同的服务：

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: example-ingress
5  spec:
6    rules:
7    - host: foo.example.com
8      http:
9        paths:
10        - path: /
11          backend:
12            serviceName: foo-service
13            servicePort: 80
14    - host: bar.example.com
15      http:

```



```

16     paths:
17     - path: /
18       backend:
19         serviceName: bar-service
20         servicePort: 80

```

在上面的示例中，具有主机头 `foo.example.com` 的请求将路由到端口为 80 的 `foo-service`，而具有主机头 `bar.example.com` 的请求将路由到端口为 80 的 `bar-service`。

配置 SSL

- [官方地址](#)。
- 生成证书语法

```

1 openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ${KEY_FILE} -out
  ${CERT_FILE} -subj "/CN=${HOST:baidu.com}/O=${HOST:baidu.com}"

```

```

1 kubectl create secret tls ${CERT_NAME:baidu-tls} --key ${KEY_FILE:tls.key} --
  cert ${CERT_FILE:tls.cert}

```

- 示例：生成证书

```

1 openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out
  tls.cert -subj "/CN=nginx.angle.com/O=tomcat.angle.com"

```

```

1 kubectl create secret tls angle-tls --key tls.key --cert tls.cert

```

```

1 kubectl get secrets angle-tls -o yaml

```

- 示例

```

1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: ingress-tls
5   namespace: default
6   annotations:
7     kubernetes.io/ingress.class: "nginx"
8 spec:
9   tls:
10  - hosts:
11    - nginx.angle.com # 通过浏览器访问 https://nginx.angle.com
12    - tomcat.angle.com # 通过浏览器访问 https://tomcat.angle.com
13    secretName: angle-tls
14  rules:
15  - host: nginx.angle.com
16    http:
17      paths:
18      - path: /
19        pathType: Prefix
20        backend:
21          service:

```

```

22         name: nginx-svc
23         port:
24             number: 80
25     - host: tomcat.angle.com
26     http:
27         paths:
28         - path: /
29           pathType: Prefix
30         backend:
31             service:
32                 name: tomcat-svc
33             port:
34                 number: 8080

```

- 测试

```
1 | curl https://nginx.angle.com -k
```

```

d:\Users\Bamboo\Desktop>curl https://nginx.angle.com -k
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>

d:\Users\Bamboo\Desktop>

```

注意：实际开发的时候，需要自己购买证书。

金丝雀发布

什么是金丝雀发布

金丝雀发布是一种流行的部署策略，用于逐步将新版本的应用程序部署到生产环境中。在Kubernetes中，使用Ingress对象可以实现金丝雀发布。

具体来说，可以使用Ingress对象的多个规则来将流量路由到不同版本的应用程序。例如，可以将10%的流量路由到新版本的应用程序，而将90%的流量路由到旧版本的应用程序。在经过一段时间的测试和验证后，可以逐步将流量比例提高到100%，从而完成金丝雀发布。

前提工作

- 部署 Service 和 Deployment

```
1 | vi k8s-ingress-canary-deploy.yaml
```

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: v1-deployment
5    labels:
6      app: v1-deployment
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       app: v1-pod
12   template:
13     metadata:
14       name: nginx
15       labels:
16         app: v1-pod
17     spec:
18       initContainers:
19         - name: alpine
20           image: alpine
21           imagePullPolicy: IfNotPresent
22           command: ["/bin/sh", "-c", "echo v1-pod > /app/index.html;"]
23           volumeMounts:
24             - mountPath: /app
25               name: app
26       containers:
27         - name: nginx
28           image: nginx:1.17.2
29           imagePullPolicy: IfNotPresent
30           ports:
31             - containerPort: 80
32           resources:
33             requests:
34               cpu: 100m
35               memory: 100Mi
36             limits:
37               cpu: 250m
38               memory: 500Mi
39           volumeMounts:
40             - name: app
41               mountPath: /usr/share/nginx/html
42       volumes:
43         - name: app
44           emptyDir: {}
45       restartPolicy: Always
46 ---
47 apiVersion: v1
48 kind: Service
49 metadata:
```

```
50   name: v1-service
51   namespace: default
52 spec:
53   selector:
54     app: v1-pod
55   type: ClusterIP
56   ports:
57   - name: nginx
58     protocol: TCP
59     port: 80
60     targetPort: 80
61 ---
62 apiVersion: apps/v1
63 kind: Deployment
64 metadata:
65   name: v2-deployment
66   labels:
67     app: v2-deployment
68 spec:
69   replicas: 3
70   selector:
71     matchLabels:
72       app: v2-pod
73   template:
74     metadata:
75       name: v2-pod
76     labels:
77       app: v2-pod
78     spec:
79       containers:
80       - name: nginx
81         image: nginx:1.17.2
82         imagePullPolicy: IfNotPresent
83         ports:
84         - containerPort: 80
85       resources:
86         requests:
87           cpu: 100m
88           memory: 100Mi
89         limits:
90           cpu: 250m
91           memory: 500Mi
92       restartPolicy: Always
93 ---
94 apiVersion: v1
95 kind: Service
96 metadata:
97   name: v2-service
98   namespace: default
99 spec:
100   selector:
101     app: v2-pod
102   type: ClusterIP
103   ports:
104   - name: nginx
```

```
105 | protocol: TCP
106 | port: 80
107 | targetPort: 80
```

```
1 | kubectl apply -f k8s-ingress-canary-deploy.yaml
```

- 部署普通的 ingress

```
1 | vi k8s-ingress-v1.yaml
```

```
1 | apiVersion: networking.k8s.io/v1
2 | kind: Ingress
3 | metadata:
4 |   name: ingress-v1
5 |   namespace: default
6 | spec:
7 |   ingressClassName: "nginx"
8 |   rules:
9 |   - host: nginx.angle.com
10 |     http:
11 |       paths:
12 |       - path: /
13 |         pathType: Prefix
14 |         backend:
15 |           service:
16 |             name: v1-service
17 |             port:
18 |               number: 80
```

```
1 | kubectl apply -f k8s-ingress-v1.yaml
```

- 测试 ingress

```
1 | # curl 来模拟请求
2 | curl -H "Host: canary.example.com" http://EXTERNAL-IP # EXTERNAL-IP 替换为
   | Nginx Ingress 自身对外暴露的 IP
```

```
1 | curl -H "Host: nginx.angle.com" http://192.168.183.102
```

基于服务权重的流量切分

以下是一个简单的示例Ingress对象，用于实现金丝雀发布：

```
1 | apiVersion: networking.k8s.io/v1
2 | kind: Ingress
3 | metadata:
4 |   name: ingress-canary
5 |   namespace: default
6 |   annotations:
7 |     nginx.ingress.kubernetes.io/canary: "true" # 开启金丝雀
8 |     nginx.ingress.kubernetes.io/canary-weight: "10" # 基于服务权重
```

```

9 spec:
10   ingressClassName: "nginx"
11   rules:
12   - host: nginx.xudaxian.com
13     http:
14       paths:
15       - path: /
16         pathType: Prefix
17         backend:
18           service:
19             name: v2-service
20             port:
21               number: 80

```

在上述示例中，Ingress对象包含两个规则，每个规则将请求路由到不同版本的应用程序。第一个规则将以 /v1 为前缀的请求路由到旧版本的应用程序，第二个规则将以 /v2 为前缀的请求路由到新版本的应用程序。通过在Ingress对象的注释中设置 `nginx.ingress.kubernetes.io/canary: "true"`，启用了金丝雀发布策略，`nginx.ingress.kubernetes.io/canary-weight: "10"` 表示将新版本应用的程序的流量权重设置为10%。

基于Header的流量切分

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ingress-canary
5    namespace: default
6    annotations:
7      nginx.ingress.kubernetes.io/canary: "true" # 开启金丝雀
8
9      nginx.ingress.kubernetes.io/canary-by-header: "Region" # 基于请求头
10     # 如果 请求头 Region = always ，就路由到金丝雀版本；如果 Region = never ，就永
      远不会路由到金丝雀版本。
11
12     nginx.ingress.kubernetes.io/canary-by-header-value: "sz" # 自定义值
13     # 如果 请求头 Region = sz ，就路由到金丝雀版本；如果 Region != sz ，就永远不会路
      由到金丝雀版本。
14     # nginx.ingress.kubernetes.io/canary-by-header-pattern: "sh|sz"
15     # 如果 请求头 Region = sh 或 Region = sz ，就路由到金丝雀版本；如果 Region !=
      sz 并且 Region != sz ，就永远不会路由到金丝雀版本。
16 spec:
17   ingressClassName: "nginx"
18   rules:
19   - host: nginx.angle.com
20     http:
21       paths:
22       - path: /
23         pathType: Prefix
24         backend:
25           service:
26             name: v2-service
27             port:
28               number: 80

```

基于 Cookie 的流量切分

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ingress-canary
5    namespace: default
6    annotations:
7      nginx.ingress.kubernetes.io/canary: "true" # 开启金丝雀
8      nginx.ingress.kubernetes.io/canary-by-cookie: "vip" # 如果 cookie 是 vip
= always ，就会路由到到金丝雀版本；如果 cookie 是 vip = never ，就永远不会路由到金丝
雀的版本。
9  spec:
10    ingressClassName: "nginx"
11    rules:
12      - host: nginx.xudaxian.com
13        http:
14          paths:
15            - path: /
16              pathType: Prefix
17              backend:
18                service:
19                  name: v2-service
20                  port:
21                    number: 80
```

存储管理和卷配置

配置信息

ConfigMap

在 Kubernetes 中，ConfigMap 是一种用于存储非敏感配置数据的对象。ConfigMap 可以存储键值对、属性文件、JSON 配置等数据，这些数据可以在容器中使用，从而将应用程序与配置数据分离开来。在本节中，我们将介绍 ConfigMap 的概念、创建 ConfigMap 的方法以及在 Pod 和容器中使用 ConfigMap 的方法。

ConfigMap 的概念

ConfigMap 是 Kubernetes 中的一种对象，用于存储非敏感的配置数据。ConfigMap 可以存储键值对、属性文件、JSON 配置等数据。ConfigMap 通常用于将应用程序与配置数据分离，从而使应用程序更易于管理和维护。

创建 ConfigMap

在 Kubernetes 中，可以通过多种方式创建 ConfigMap，如使用 kubectl 命令行工具、使用 YAML 文件、使用 Helm 等。下面我们将介绍使用 kubectl 命令行工具创建 ConfigMap 的方法。

1. 使用命令行创建 ConfigMap

使用 kubectl 命令行工具可以创建 ConfigMap。下面是创建 ConfigMap 的基本语法：

```
1  kubectl create configmap <configmap-name> <data-source>
```

其中 `<configmap-name>` 是要创建的 ConfigMap 的名称, `<data-source>` 是要从中创建 ConfigMap 的数据源, 可以是文件、目录、环境变量等。

例如, 我们可以使用以下命令从文件创建 ConfigMap:

```
1 kubectl create configmap my-config --from-file=config-file.txt
```

这将创建一个名为 `my-config` 的 ConfigMap, 其中包含 `config-file.txt` 文件的内容。

我们还可以从目录创建 ConfigMap。例如, 我们可以使用以下命令从目录创建 ConfigMap:

```
1 kubectl create configmap my-config --from-file=config-dir/
```

这将创建一个名为 `my-config` 的 ConfigMap, 其中包含 `config-dir/` 目录中的所有文件的内容。

我们还可以从环境变量创建 ConfigMap。例如, 我们可以使用以下命令从环境变量创建 ConfigMap:

```
1 kubectl create configmap my-config --from-literal=foo=bar --from-literal=baz=qux
```

这将创建一个名为 `my-config` 的 ConfigMap, 其中包含两个键值对: `foo=bar` 和 `baz=qux`。

2. 使用 YAML 文件创建 ConfigMap

我们还可以使用 YAML 文件创建 ConfigMap。下面是一个示例 YAML 文件:

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: my-config
5 data:
6   config-file.txt: |
7     This is the content of the config file.
```

该文件包含了一个名为 `my-config` 的 ConfigMap, 其中包含一个键值对, 键为 `config-file.txt`, 值为 `This is the content of the config file.`。

我们可以使用以下命令将该文件中的配置应用于 Kubernetes:

```
1 kubectl apply -f configmap.yaml
```

这将创建一个名为 `my-config` 的 ConfigMap, 并将其中的数据应用于 Kubernetes。

在 Pod 中使用 ConfigMap

在 Pod 中使用 ConfigMap 可以使用环境变量或者配置文件的方式。下面我们将介绍如何在 Pod 中使用 ConfigMap。

• 使用环境变量

我们可以在 Pod 中使用 ConfigMap 中的数据作为环境变量。以下是一个示例 YAML 文件:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
```



```

4   name: my-pod
5   spec:
6     containers:
7     - name: my-container
8       image: nginx
9       env:
10      - name: MY_CONFIG
11        valueFrom:
12          configMapKeyRef:
13            name: my-config
14            key: config-file.txt

```

该文件包含了一个名为 `my-pod` 的 Pod，其中包含一个名为 `my-container` 的容器。该容器使用 nginx 镜像，并将 ConfigMap `my-config` 中 `config-file.txt` 键值对的值作为名为 `MY_CONFIG` 的环境变量的值。

我们可以使用以下命令将该文件中的配置应用于 Kubernetes：

```
1 kubectl apply -f pod.yaml
```

这将创建一个名为 `my-pod` 的 Pod，并将其中的数据应用于 Kubernetes。

- **使用配置文件**

我们还可以在 Pod 中使用 ConfigMap 中的数据作为配置文件。以下是一个示例 YAML 文件：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7    - name: my-container
8      image: nginx
9      volumeMounts:
10     - name: config-volume
11       mountPath: /etc/my-config
12  volumes:
13  - name: config-volume
14    configMap:
15      name: my-config

```

该文件包含了一个名为 `my-pod` 的 Pod，其中包含一个名为 `my-container` 的容器。该容器使用 nginx 镜像，并将 ConfigMap `my-config` 中的数据作为配置文件挂载到 `/etc/my-config` 目录中。

我们可以使用以下命令将该文件中的配置应用于 Kubernetes：

```
1 kubectl apply -f pod.yaml
```

这将创建一个名为 `my-pod` 的 Pod，并将其中的数据应用于 Kubernetes。

在容器中使用 ConfigMap

在容器中使用 ConfigMap 可以使用环境变量或者配置文件的方式。下面我们将介绍如何在容器中使用 ConfigMap。

注意，镜像是在配置了 configmap 的 k8s 集群中使用

- 使用环境变量

我们可以在容器中使用 ConfigMap 中的数据作为环境变量。以下是一个示例 Dockerfile 文件：

```
1 FROM nginx
2 COPY nginx.conf /etc/nginx/
3 ENV MY_CONFIG=$(my-config)
```

该文件使用 nginx 镜像作为基础镜像，并将 my-config ConfigMap 中的数据作为名为 MY_CONFIG 的环境变量的值。

我们可以使用以下命令将该 Dockerfile 构建为镜像：

```
1 docker build -t my-image .
```

这将构建名为 my-image 的 Docker 镜像。

- 使用配置文件

我们还可以在容器中使用 ConfigMap 中的数据作为配置文件。以下是一个示例 Dockerfile 文件：

```
1 FROM nginx
2 COPY my-config /etc/my-config/
```

该文件使用 nginx 镜像作为基础镜像，并将 my-config ConfigMap 中的数据作为配置文件复制到 /etc/my-config/ 目录中。

我们可以使用以下命令将该 Dockerfile 构建为镜像：

```
1 docker build -t my-image .
```

这将构建名为 my-image 的 Docker 镜像。

示例：将配置文件作为 ConfigMap 使用

假设我们有一个名为 my-config 的 ConfigMap，其中包含一个名为 config-file.txt 的键值对，其值为：

```
1 # This is the content of the config file.
2 server {
3     listen 80;
4     server_name example.com;
5     root /usr/share/nginx/html;
6     index index.html;
7 }
```

我们可以在 Pod 中使用该 ConfigMap 中的数据作为配置文件。以下是一个示例 YAML 文件：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: nginx
9        volumeMounts:
10       - name: config-volume
11         mountPath: /etc/nginx/conf.d
12    volumes:
13      - name: config-volume
14        configMap:
15          name: my-config

```

该文件包含了一个名为 `my-pod` 的 Pod，其中包含一个名为 `my-container` 的容器。该容器使用 nginx 镜像，并将 ConfigMap `my-config` 中的 `config-file.txt` 键值对的值作为配置文件挂载到 `/etc/nginx/conf.d` 目录中。

我们可以使用以下命令将该文件中的配置应用于 Kubernetes：

```
1  kubectl apply -f pod.yaml
```

这将创建一个名为 `my-pod` 的 Pod，并将其中的数据应用于 Kubernetes。

示例：在容器中使用 ConfigMap

假设我们有一个名为 `my-config` 的 ConfigMap，其中包含一个名为 `my-key` 的键值对，其值为 `my-value`。我们可以在容器中使用该 ConfigMap 中的数据作为环境变量。以下是一个示例 Deployment YAML 文件：

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-app
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16         - name: my-container
17           image: nginx
18           env:
19             - name: MY_KEY
20               valueFrom:
21                 configMapKeyRef:
22                   name: my-config

```

该文件创建一个名为 `my-app` 的 Deployment，其中包含一个名为 `my-container` 的容器。在该容器中，我们定义了一个环境变量 `MY_KEY`，它的值来自于 `my-config` ConfigMap 中的 `my-key` 键值对。

在这个例子中，我们使用 `configMapKeyRef` 引用了 ConfigMap 中的数据，`name` 字段指定 ConfigMap 的名称，`key` 字段指定要使用的键名。

通过这种方式，我们可以将 ConfigMap 中的配置数据注入到容器中，以便容器中的应用程序可以使用这些配置数据。

Secret

Secret 的概念

在 Kubernetes 中，Secret 是一种**用于存储敏感数据的对象**。与 ConfigMap 类似，Secret 也可以用于将配置数据注入到 Pod 中。但是，与 ConfigMap 不同的是，Secret 存储的是敏感数据，例如密码、证书等，这些数据需要被加密并安全地存储。Kubernetes 会自动为 Secret 进行加密，并且只有被授权的用户才能访问 Secret。

创建 Secret

与创建 ConfigMap 类似，创建 Secret 也可以使用 `kubectl` 命令行工具、使用 YAML 文件等多种方式。以下是一个使用 `kubectl` 命令行工具创建 Secret 的示例：

```
1 kubectl create secret generic my-secret --from-literal=username=myuser --  
  from-literal=password=mypass
```

该命令会创建一个名为 `my-secret` 的 Secret，其中包含两个键值对：`username` 和 `password`，它们的值分别为 `myuser` 和 `mypass`。这两个键值对都是使用 `--from-literal` 参数指定的。

创建 Secret 还可以使用 YAML 文件，以下是一个示例 YAML 文件：

```
1 apiVersion: v1  
2 kind: Secret  
3 metadata:  
4   name: my-secret  
5 type: Opaque  
6 data:  
7   username: bXl1c2Vy  
8   password: bXlwYXNz
```

在这个例子中，我们创建了一个名为 `my-secret` 的 Secret，其中包含两个键值对：`username` 和 `password`。这两个值都是经过 base64 编码的，这样可以保证敏感数据在存储时不会被明文存储。`type` 字段指定了 Secret 的类型，`data` 字段包含了 Secret 中的键值对。

在 Pod 中使用 Secret

在 Pod 中使用 Secret 时，我们可以使用环境变量或者 Volume 的方式。以下是一个使用环境变量的示例 Deployment YAML 文件：

```
1 apiVersion: apps/v1  
2 kind: Deployment  
3 metadata:  
4   name: my-app
```

```

5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: my-app
10  template:
11    metadata:
12      labels:
13        app: my-app
14    spec:
15      containers:
16        - name: my-container
17          image: nginx
18          env:
19            - name: USERNAME
20              valueFrom:
21                secretKeyRef:
22                  name: my-secret
23                  key: username
24            - name: PASSWORD
25              valueFrom:
26                secretKeyRef:
27                  name: my-secret
28                  key: password

```

该文件创建一个名为 `my-app` 的 Deployment，其中包含一个名为 `my-container` 的容器。在该容器中，我们定义了两个环境变量 `USERNAME` 和 `PASSWORD`，它们的值分别来自于 `my-secret` Secret 中的 `username` 和 `password` 键值对。

在容器中使用 Secret

除了使用环境变量的方式，在容器中还可以使用 Volume 的方式使用 Secret。以下是一个使用 Volume 的示例 Deployment YAML 文件：

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-app
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16         - name: my-container
17           image: nginx
18           volumeMounts:
19             - name: my-secret
20               mountPath: /etc/my-secret
21               readOnly: true

```

```

22     volumes:
23       - name: my-secret
24         secret:
25           secretName: my-secret

```

该文件创建一个名为 `my-app` 的 Deployment，其中包含一个名为 `my-container` 的容器。在该容器中，我们定义了一个 Volume `my-secret`，它将 `my-secret` Secret 挂载到容器的 `/etc/my-secret` 目录中。

在容器中，我们可以使用 `/etc/my-secret` 目录中的文件来访问 Secret 中的键值对。例如，我们可以使用以下命令来读取 `username` 键的值：

```

1 cat /etc/my-secret/username

```

示例：将敏感数据作为 Secret 使用

假设我们有一个包含敏感数据的 YAML 文件，例如：

```

1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: my-secret
5 type: Opaque
6 data:
7   username: bX1lc2Vy
8   password: bX1wYXNz

```

我们可以使用 `kubectl` 命令行工具将其创建为一个名为 `my-secret` 的 Secret：

```

1 kubectl apply -f my-secret.yaml

```

此时，该 Secret 将被加密并安全地存储在 Kubernetes 集群中。

示例：在容器中使用 Secret

假设我们有一个名为 `my-container` 的容器，我们想要在该容器中使用 `my-secret` Secret 中的 `username` 和 `password` 键值对。我们可以使用以下 YAML 文件创建一个使用 Volume 的 Deployment：

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: my-app
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: my-app
10  template:
11    metadata:
12      labels:
13        app: my-app
14    spec:

```

```

15     containers:
16       - name: my-container
17         image: nginx
18         volumeMounts:
19           - name: my-secret
20             mountPath: /etc/my-secret
21             readOnly: true
22     volumes:
23       - name: my-secret
24         secret:
25           secretName: my-secret

```

在该 Deployment 中，我们创建了一个名为 `my-secret` 的 Volume，它使用 `my-secret` Secret 中的键值对。在容器中，我们可以使用 `/etc/my-secret` 目录中的文件来访问 Secret 中的键值对。例如，我们可以使用以下命令来读取 `username` 键的值：

```

1 cat /etc/my-secret/username

```

在这个例子中，我们使用了 Volume 的方式来将 Secret 中的数据挂载到容器中。除此之外，我们还可以使用环境变量的方式来使用 Secret 中的数据，具体方式与在 Pod 中使用 Secret 类似。

临时存储

在 Kubernetes 中，有多种方式可以在 Pod 中存储临时数据，例如使用 `EmptyDir`、`hostPath` 或者卷挂载（Volume Mounts）。这些存储方式各有优缺点，我们可以根据具体的需求选择合适的方式。

卷挂载 (Volume Mounts)

卷挂载的概念

卷挂载（Volume Mounts）是 Kubernetes 中的一种存储方式，它可以将一个或多个卷（Volume）挂载到容器中。卷可以存储持久化数据，也可以存储临时数据。卷挂载通常用于存储需要共享的数据或者需要持久化存储的数据。

在 Pod 中使用卷挂载

在 Pod 中使用卷挂载是指将一个或多个卷（Volume）挂载到一个 Pod 中，Pod 中的多个容器可以访问这些卷中存储的数据。在 Pod 的 YAML 文件中，我们可以使用 `volumes` 字段来指定要使用的卷。

以下是一个示例 YAML 文件，展示如何在 Pod 中使用卷挂载：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container-1
8        image: nginx
9        volumeMounts:
10         - name: my-volume
11           mountPath: /data
12      - name: my-container-2
13        image: nginx
14        volumeMounts:

```

```

15     - name: my-volume
16       mountPath: /data
17   volumes:
18     - name: my-volume
19       emptyDir: {}

```

在这个例子中，我们创建了一个名为 `my-volume` 的 EmptyDir 卷，并将它挂载到名为 `my-container-1` 和 `my-container-2` 的容器中。这个示例展示了如何在一个 Pod 中使用卷挂载，并在多个容器之间共享数据。

在容器中使用卷挂载

在容器中使用卷挂载时，我们可以在容器的 YAML 文件中将 Volume 挂载到容器的某个目录中。以下是一个示例 YAML 文件：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: nginx
9        volumeMounts:
10         - name: my-volume
11           mountPath: /data
12   volumes:
13     - name: my-volume
14       emptyDir: {}

```

在这个例子中，我们创建了一个名为 `my-volume` 的 Volume，并将它挂载到名为 `my-container` 的容器的 `/data` 目录中。该 Volume 使用了 EmptyDir 存储类型。

EmptyDir

EmptyDir 的概念

EmptyDir 是 Kubernetes 中的一种存储类型，它可以用于存储临时数据。EmptyDir 存储是基于 Pod 的生命周期的，当 Pod 被删除时，EmptyDir 中的数据也会被删除。EmptyDir 存储通常在容器之间共享数据时使用，例如在一个 Pod 中运行多个容器，它们需要共享一些数据时。

使用 EmptyDir 存储临时数据

以下是一个使用 EmptyDir 存储临时数据的示例 YAML 文件：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container-1
8        image: nginx
9        volumeMounts:
10         - name: my-volume

```



```

11     mountPath: /data
12     command: ["/bin/bash", "-c", "echo 'hello world' > /data/hello.txt &&
sleep 1d"]
13   - name: my-container-2
14     image: nginx
15     volumeMounts:
16       - name: my-volume
17         mountPath: /data
18     command: ["/bin/bash", "-c", "cat /data/hello.txt"]
19   volumes:
20     - name: my-volume
21       emptyDir: {}

```

在这个例子中，我们创建了一个名为 `my-volume` 的 EmptyDir Volume，并将它挂载到名为 `my-container-1` 和 `my-container-2` 的容器中。`my-container-1` 容器会将 `hello world` 写入 `/data/hello.txt` 文件中，并在之后等待一天。`my-container-2` 容器会从 `/data/hello.txt` 中读取数据并输出到标准输出。这个示例展示了如何使用 EmptyDir 存储临时数据，并在多个容器之间共享数据。

hostPath

hostPath 的概念

hostPath 是 Kubernetes 中的一种存储类型，它可以用于将主机上的文件系统挂载到 Pod 中。

hostPath 存储通常用于存储临时数据或者需要持久化存储的数据。但是，使用 hostPath 存储有一定的安全风险，因为容器可以访问主机上的所有文件系统。

使用 hostPath 存储临时数据

以下是一个使用 hostPath 存储临时数据的示例 YAML 文件：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container-1
8        image: nginx
9        volumeMounts:
10         - name: my-volume
11           mountPath: /data
12         command: ["/bin/bash", "-c", "echo 'hello world' > /data/hello.txt &&
sleep 1d"]
13      - name: my-container-2
14        image: nginx
15        volumeMounts:
16         - name: my-volume
17           mountPath: /data
18         command: ["/bin/bash", "-c", "cat /data/hello.txt"]
19    volumes:
20      - name: my-volume
21        hostPath:
22          path: /tmp/my-data

```

在这个例子中，我们创建了一个名为 `my-volume` 的 `hostPath` Volume，并将它挂载到名为 `my-container-1` 和 `my-container-2` 的容器中。`my-container-1` 容器会将 `hello world` 写入 `/tmp/my-data/hello.txt` 文件中，并在之后等待一天。`my-container-2` 容器会从 `/tmp/my-data/hello.txt` 中读取数据并输出到标准输出。这个示例展示了如何使用 `hostPath` 存储临时数据，并在多个容器之间共享数据。注意，使用 `hostPath` 存储有一定的安全风险，因为容器可以访问主机上的所有文件系统。

持久化存储

PersistentVolume

PersistentVolume 的概念

PersistentVolume 是 Kubernetes 中用于表示持久化存储资源的 API 对象。它允许 Kubernetes 集群中的 Pod 访问独立于 Pod 生命周期的持久化存储。PersistentVolume 可以基于不同的存储后端（如本地存储、网络存储和云存储）创建，并且可以在不同的 Pod 之间共享和重用。

创建 PersistentVolume

可以通过 Kubernetes 中的 YAML 文件定义 `PersistentVolume`。下面是一个例子：

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: my-pv
5  spec:
6    storageClassName: fast
7    capacity:
8      storage: 1Gi
9    accessModes:
10     - ReadWriteOnce
11    hostPath:
12      path: /data
```

在这个例子中，我们创建了一个名为 `my-pv` 的 `PersistentVolume`，使用了 `fast` 存储类别，容量为 1GB，访问模式为 `ReadWriteOnce`，使用了 `hostPath` 存储后端。其中，`hostPath` 存储后端将数据存储在节点上的本地磁盘上。在实际生产环境中，通常会使用一些网络存储或云存储后端。

注意，`fast` 存储类别这个是由 `pvc` 定义的。

PersistentVolume 的回收策略

`PersistentVolume` 的回收策略定义了当 `PersistentVolumeClaim` 被删除时，`PersistentVolume` 中数据的处理方式。Kubernetes 支持四种回收策略：

- **Retain**：保留 `PersistentVolume` 中的数据，不进行删除。
- **Delete**：删除 `PersistentVolume` 中的数据，并将 `PersistentVolume` 标记为可用。
- **Recycle**：删除 `PersistentVolume` 中的数据，并将 `PersistentVolume` 标记为可用。但是，数据并不会被完全清除，只会被清除到一个空目录中。
- **Delete 和 Retain**：将数据从 `PersistentVolume` 中删除，并将 `PersistentVolume` 标记为不可用。但是，数据不会被彻底删除，可以手动进行恢复。

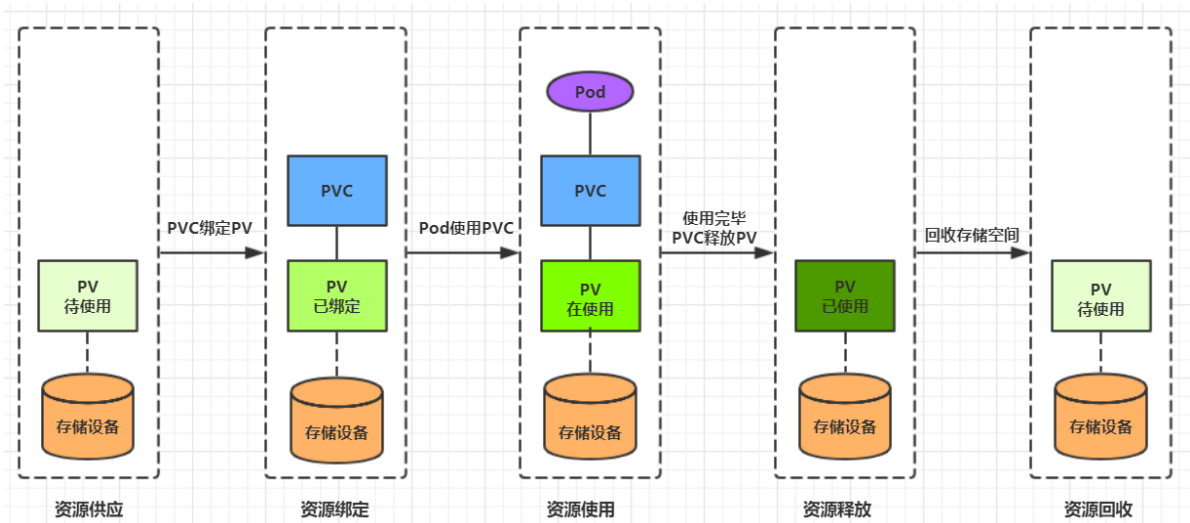
在 `PersistentVolume` 的 YAML 文件中，可以通过 `spec.persistentVolumeReclaimPolicy` 字段指定回收策略。例如：

```

1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: my-pv
5  spec:
6    capacity:
7      storage: 1Gi
8    accessModes:
9      - ReadWriteOnce
10   persistentVolumeReclaimPolicy: Retain
11   hostPath:
12     path: /data

```

PersistentVolume 的生命周期



PersistentVolume 的生命周期可以分为三个阶段：创建、使用和删除。

- 在创建阶段，管理员通过定义 PersistentVolume 的 YAML 文件来创建 PersistentVolume。创建后，PersistentVolume 进入可用状态，并可以被 PersistentVolumeClaim 使用。
- 在使用阶段，PersistentVolumeClaim 可以通过定义 YAML 文件来请求 PersistentVolume。如果存在符合要求的 PersistentVolume，则将其绑定到 PersistentVolumeClaim 上，并将其标记为正在使用。在 PersistentVolumeClaim 的 Pod 被删除之前，PersistentVolume 将一直保持绑定状态。
- 在删除阶段，管理员可以删除 PersistentVolume，或者删除与 PersistentVolume 相关联的 PersistentVolumeClaim。当一个 PersistentVolumeClaim 被删除时，PersistentVolume 可以按照预定义的回收策略进行处理。

PersistentVolumeClaim

PersistentVolumeClaim 的概念

PersistentVolumeClaim (PVC) 是 Kubernetes 中用于声明要求持久卷的 API 对象。PVC 用于将存储需求与存储资源分离，Pod 可以通过 PVC 来请求持久化存储资源而不关心它们的实现。PVC 可以在多个 Pod 之间共享，使得多个 Pod 可以访问同一份数据，从而提高了数据的可靠性和可用性。

创建 PersistentVolumeClaim

在 Kubernetes 中，可以通过 YAML 文件创建 `PersistentVolumeClaim`。以下是一个 PVC 的 YAML 文件示例：

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: my-pvc
5  spec:
6    accessModes:
7      - ReadWriteOnce
8    resources:
9      requests:
10       storage: 5Gi
```

在上面的示例中，`apiVersion` 和 `kind` 分别指定了 PVC 的 API 版本和类型。`metadata` 部分包含 PVC 的元数据信息，例如 PVC 的名称。`spec` 部分定义了 PVC 的规格，包括访问模式、存储容量等信息。

在 Pod 中使用 PersistentVolumeClaim

要在 Pod 中使用 `PersistentVolumeClaim`，需要在 Pod 的 YAML 文件中指定 `volumes` 和 `volumeMounts`。以下是一个 Pod 的 YAML 文件示例：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: my-image
9        volumeMounts:
10         - name: my-pvc-storage
11           mountPath: /data
12    volumes:
13      - name: my-pvc-storage
14        persistentVolumeClaim:
15          claimName: my-pvc
```

在上面的示例中，`volumes` 指定了 Pod 中要挂载的卷，`volumeMounts` 则指定了要将卷挂载到哪个容器的哪个路径下。在 `volumes` 中，我们指定了要挂载的 PVC 的名称，使用 `persistentVolumeClaim` 字段引用 PVC。在 `volumeMounts` 中，我们将卷挂载到了 `/data` 目录下。

理解，pv是磁盘空间，pvc用于申请请求，pod使用的是pvc。

PersistentVolume 和 PersistentVolumeClaim 之间的关系（重要）

`PersistentVolume` (PV) 是 Kubernetes 中用于表示持久化存储的 API 对象。PV 是存储资源的实际卷，而 PVC 是 Pod 中声明要使用的存储资源的声明。PVC 用于请求 PV，从而将存储需求与存储资源分离。

在创建 PVC 时，需要指定 `accessModes` 和 `resources.requests.storage` 字段。`accessModes` 指定了 PVC 的访问模式，例如 `ReadWriteOnce`、`ReadWriteMany` 或 `ReadOnlyMany`。`resources.requests.storage` 指定了 PVC 所需的存储容量。当创建 PVC 时，Kubernetes 会查找匹配存储容量和访问模式的 PV，并将它们绑定在一起。如果找不到匹配的 PV，PVC 将处于 Pending 状态，直到新的 PV 可用为止。

以下是一个完整的 `PersistentVolume` 和 `PersistentVolumeClaim` 的示例：

首先，我们创建一个 `PersistentVolume` 的 YAML 文件，定义一个名为 `my-pv` 的持久化卷，类型为 `hostPath`，并指定路径为 `/mnt/data`。该 YAML 文件如下所示：

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: my-pv
5  spec:
6    storageClassName: "fast-storage" # 用于分组
7    capacity:
8      storage: 1Gi
9    accessModes:
10     - ReadWriteOnce
11    hostPath:
12      path: "/mnt/data"
```

接下来，我们创建一个 `PersistentVolumeClaim` 的 YAML 文件，请求 500MB 的存储资源，并指定访问模式为 `ReadWriteOnce`。该 YAML 文件如下所示：

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: my-pvc
5  spec:
6    storageClassName: "fast-storage"
7    accessModes:
8     - ReadWriteOnce
9    resources:
10     requests:
11       storage: 500Mi
```

然后，我们创建一个使用 `PersistentVolumeClaim` 的 Pod 的 YAML 文件，挂载刚才创建的 PVC 到 `/data` 目录下。该 YAML 文件如下所示：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7     - name: my-container
8       image: nginx
9       volumeMounts:
10        - name: my-pvc-volume
11          mountPath: /data
```

```
12 volumes:
13   - name: my-pvc-volume
14     persistentVolumeClaim:
15       claimName: my-pvc
```

在上面的 YAML 文件中，我们指定了一个叫做 `my-pvc-volume` 的卷，使用 `persistentVolumeClaim` 字段引用了 PVC。`volumeMounts` 部分将卷挂载到了 `/data` 目录下。

最后，我们将上面的三个 YAML 文件应用到 Kubernetes 中：

```
1 kubectl apply -f pv.yaml
2 kubectl apply -f pvc.yaml
3 kubectl apply -f pod.yaml
```

现在，我们可以检查 Pod 是否正常运行，并使用 `kubectl exec` 命令进入容器：

```
1 # kubectl get pvc # 可以通过这个命令查看 pvc 是否挂载正常
2 kubectl get pods
3 kubectl exec -it my-pod -- /bin/bash
```

在容器中，我们可以检查 `/data` 目录是否正确挂载了 PVC：

```
1 root@my-pod:/# df -h /data
2 Filesystem              Size  Used Avail Use% Mounted on
3 /dev/mapper/vg01-lv01   493M   10M  458M   3% /data
```

如上所示，容器中的 `/data` 目录挂载了一个大小为 500MB 的卷。这是因为我们在 PVC 中请求了 500MB 的存储空间。如果我们尝试在容器中写入超过 500MB 的数据，将会失败。这表明 PVC 可以帮助我们管理容器中的数据存储配额。

PersistentVolumeClaim 的访问模式

`PersistentVolumeClaim` (PVC) 的访问模式指定了 PVC 可以被多少个 Pod 同时访问。以下是 PVC 的访问模式：

- `ReadWriteOnce`：一个 Pod 可以以读写模式挂载 PVC。
- `ReadOnlyMany`：多个 Pod 可以以只读模式挂载 PVC。
- `ReadWriteMany`：多个 Pod 可以以读写模式挂载 PVC。

需要注意的是，不是所有的存储后端都支持所有的访问模式。在选择存储后端时，需要根据您的需求确定所需的访问模式。

分布式存储

StorageClass

StorageClass 的概念

`StorageClass` 是 Kubernetes 中用于动态分配存储资源的机制。通过定义不同的 `StorageClass`，Kubernetes 可以使用不同的存储后端来满足不同的应用需求，并支持动态的存储资源分配。

每个 `StorageClass` 都定义了一组参数，包括存储后端类型、卷类型、卷访问模式、存储配额等。当 Kubernetes 集群中的 Pod 请求动态分配存储资源时，Kubernetes 将根据 `StorageClass` 的定义动态创建一个 `PersistentVolume` 对象，并将其绑定到请求的 `PersistentVolumeClaim` 对象上，从而满足 Pod 对持久化存储的需求。

`StorageClass` 主要用于管理动态分配的持久化存储资源，它允许 Kubernetes 集群管理员为不同的应用或不同的命名空间定义不同类型、不同规格的存储资源。因此，当你需要使用第三方存储提供商的存储后端或者定制的存储资源时，你可以使用 `StorageClass` 来定义这些存储资源。

在 Kubernetes 中使用 `StorageClass`

要在 Kubernetes 中使用 `StorageClass`，您需要首先创建一个 `StorageClass` 对象。下面是一个使用 `hostPath` 存储后端的 `StorageClass` 的示例：

```
1  apiVersion: storage.k8s.io/v1
2  kind: StorageClass
3  metadata:
4    name: my-storage-class
5  provisioner: kubernetes.io/no-provisioner
6  volumeBindingMode: WaitForFirstConsumer
7  allowVolumeExpansion: true
8  parameters:
9    type: local
```

在上面的示例中，我们定义了一个名为 `my-storage-class` 的 `StorageClass` 对象。`provisioner` 字段指定了用于管理存储资源的插件，这里我们指定为 `kubernetes.io/no-provisioner`，表示不使用任何默认的存储插件。`volumeBindingMode` 字段指定了卷绑定模式，这里我们将其设置为 `WaitForFirstConsumer`，表示卷将在第一个使用它的 Pod 被调度后才会被绑定。`allowVolumeExpansion` 字段指定了卷是否允许扩容。`parameters` 字段指定了一组键值对，用于定义存储后端的参数，这里我们将 `type` 设置为 `local`，表示使用 `hostPath` 存储后端。

创建 `StorageClass` 对象后，您可以在 `PersistentVolumeClaim` 中指定该 `StorageClass`，以便 Kubernetes 使用该 `StorageClass` 分配存储资源。例如，下面是一个使用 `my-storage-class` 的 `PersistentVolumeClaim` 的示例：

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: my-pvc
5  spec:
6    storageClassName: my-storage-class
7    accessModes:
8      - ReadWriteOnce
9    resources:
10     requests:
11       storage: 1Gi
```

在上面的示例中，我们在 `PersistentVolumeClaim` 中指定了 `storageClassName` 为 `my-storage-class`，这表示 Kubernetes 应该使用 `my-storage-class` 分配存储资源。

NFS

安装 NFS

使用 NFS

使用 `StorageClass` 配置动态存储

CephFS

安装 CephFS

使用 CephFS

使用 `StorageClass` 配置动态存储

动态供应

动态供应的概念

Kubernetes 中的动态供应

kubernetes 调度原理

Kubernetes 调度器概述

调度器的角色和功能

Kubernetes 调度器是 Kubernetes 集群的控制平面组件之一，主要负责将 Pod 调度到集群中的节点上。调度器的主要角色和功能包括：

- 监视集群中未调度的 Pod。
- 根据 Pod 的需求和节点的资源情况，选择最佳的节点进行调度。
- 向 API Server 发送调度请求，将 Pod 调度到选定的节点上。
- 监视和处理调度失败、节点故障等异常情况。

调度器的架构和组件

Kubernetes 调度器的架构和组件主要包括以下几个部分：

- 队列：调度器会将需要调度的 Pod 放入队列中，按照优先级和时间顺序进行调度。
- 预选器（Predicate）：调度器会对队列中的 Pod 进行预选，筛选出满足 Pod 要求的节点，例如 CPU、内存等资源的要求，节点上已有的 Pod 的数量等。
- 优选器（Priority）：调度器会对经过预选的节点进行评分，并根据评分结果选择最佳的节点进行调度。
- 绑定器（Binding）：调度器将调度成功的 Pod 绑定到选定的节点上，并向 API Server 发送绑定请求。

调度器的工作流程

Pod 的调度流程

Kubernetes 调度器的工作流程包括以下三个步骤：

预选和优先级

调度器首先会对待调度的 Pod 进行预选，筛选出满足 Pod 要求的节点，例如 CPU、内存等资源的要求，节点上已有的 Pod 的数量等。然后，调度器会对经过预选的节点进行评分，并根据评分结果选择最佳的节点进行调度。

选定节点

调度器选定节点的过程包括以下几个步骤：

- 获取所有未被标记为 unschedulable 的节点。
- 根据节点的资源 and Pod 的要求进行匹配。
- 对符合要求的节点进行评分，选择最佳的节点。
- 如果有多个节点得分相同，则按照一定的规则进行选择（例如随机选择）。

绑定 Pod

调度器将调度成功的 Pod 绑定到选定的节点上，并向 API Server 发送绑定请求。

调度器的算法和策略

调度器的默认算法

Kubernetes 调度器默认使用的是最高得分优先（Highest Score First）的算法，即对节点进行评分，并选择得分最高的节点进行调度。评分的过程包括以下几个因素：

- 节点资源的匹配度。
- Pod 与节点亲和性的匹配度。
- 节点上已有 Pod 的数量。
- Pod 的优先级和 QoS 类别。

调度器的可扩展算法

除了默认算法之外，Kubernetes 调度器还支持自定义算法，包括：

- 基于深度学习的算法。
- 基于遗传算法的算法。
- 基于模拟退火的算法。
- 基于贪心算法的算法。

可扩展算法需要通过调度器插件机制进行实现。

常见的调度方式

Kubernetes 提供了以下四种常见的调度方式：

1. **基于 Pod 的亲和性调度 (Pod Affinity)**：可以将同一个 Pod 或者同一个副本集的多个 Pod 调度到同一个节点或者一组节点，以提高应用性能和可靠性。
2. **基于节点的亲和性调度 (Node Affinity)**：可以将具有特定标签或者标签组合的 Pod 调度到具有相同标签或者标签组合的节点上，以满足应用的特殊需求。
3. **污点和容忍 (Taints and Tolerations)**：可以将节点标记为不适合运行某些 Pod，然后通过为 Pod 添加容忍标签来使其在这些节点上运行。这种方式可以防止不合适的应用程序在错误的节点上运行，从而提高应用程序的可靠性。
4. **基于节点选择器的定向调度 (Node Selector)**：可以将具有特定标签或者标签组合的 Pod 调度到具有相同标签或者标签组合的节点上，以满足应用的特殊需求。这种方式与基于节点的亲和性调度非常相似，但是更加灵活，可以更精确地控制 Pod 的调度。

这些调度方式可以帮助 Kubernetes 用户更好地管理和控制应用程序的调度，从而提高应用程序的性能、可靠性和可扩展性。

调度器的高级特性

自定义调度器和扩展点

Kubernetes 调度器允许用户通过自定义插件和扩展点来实现自定义调度器。用户可以使用调度器插件机制来实现自定义的调度算法、策略和扩展点，以满足特定的应用需求。

一些常见的调度器插件包括：

- 预选器插件：可以自定义预选器的行为，例如添加新的过滤规则、修改过滤规则的权重等。
- 优选器插件：可以自定义优选器的行为，例如添加新的评分规则、修改评分规则的权重等。
- 绑定器插件：可以自定义绑定器的行为，例如在绑定 Pod 之前执行某些操作、修改绑定的结果等。

调度器的调试和故障排除

在 Kubernetes 中，调度器是应用程序的核心组件之一，因此调度器故障可能会导致应用程序无法正常运行。为了及时发现和解决调度器问题，可以采用以下调试和故障排除方法：

- 查看调度器的日志文件，分析日志中的错误和警告信息。
- 使用 `kubectl` 命令查看未调度的 Pod 和调度器的状态。
- 检查节点的状态和资源使用情况，查看是否存在节点故障或资源耗尽的情况。
- 使用 `kubectl` 命令手动调度 Pod，检查调度器是否能够正常工作。
- 使用 `kubectl` 命令创建和删除节点，检查调度器是否能够正常响应变化。

调度器的性能和优化

为了最大限度地提高 Kubernetes 调度器的性能和可靠性，可以采用以下优化措施：

- 使用节点亲和性调度和定向调度，将 Pod 调度到最合适的节点上，减少不必要的调度操作。
- 避免在节点上运行过多的 Pod，以避免资源耗尽和冲突。
- 将 Pod 分配到不同的命名空间中，以避免不同应用程序之间的冲突和干扰。
- 使用调度器插件机制，实现自定义调度策略和算法，以满足特定的应用需求。
- 定期检查调度器的日志文件和性能指标，及时发现和解决问题。

亲和性调度

什么是亲和性调度

亲和性调度是 Kubernetes 中的一种调度策略，它可以根据 Pod 和 Node 之间的关系，将 Pod 调度到特定的节点上。亲和性调度主要有三种类型：Node 亲和性调度、Pod 亲和性调度和 Pod 亲和性反调度（也称为 Pod 亲和性的反向策略）。

为什么需要亲和性调度

亲和性调度可以实现多种调度策略，例如将相同应用程序的多个 Pod 调度到相同节点上，或将同一数据库集群的多个 Pod 调度到相同节点上，从而提高了应用程序的性能和可靠性。此外，通过将 Pod 调度到相同节点上，可以减少网络通信和数据传输的延迟，提高应用程序的处理速度。

Node 亲和性调度

NodeSelector

假设有一个 Kubernetes 集群，其中有多节点，每个节点都有不同的标签。现在我想将一个具有特定标签的 Pod 调度到一个具有特定标签的节点上。这时，可以使用 NodeSelector 进行调度。以下是一个示例：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    containers:
7      - name: my-container
8        image: nginx
9    nodeSelector:
10      disktype: ssd # 指定调度到具有 disktype=ssd 标签的节点上
```

在上面的示例中，我将 Pod 的 disktype 标签设置为 ssd，而且节点也有 disktype 标签，因此 Pod 可以通过 NodeSelector 调度到拥有相同标签的节点上。

Inter-Pod Affinity

假设有一个 Kubernetes 集群，其中有多节点，每个节点都有不同的标签。现在我想**将多个具有相同标签的 Pod 调度到同一节点上**，以提高应用程序的性能。这时，可以使用 Inter-Pod Affinity 进行调度。以下是一个示例：

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16         - name: my-container
17           image: nginx
18       affinity:
19         podAffinity:
20           requiredDuringSchedulingIgnoredDuringExecution:
21             - labelSelector:
22                 matchExpressions:
23                   - key: app
24                     operator: In
25                     values:
26                       - my-app
27             topologyKey: "kubernetes.io/hostname"
```

在上面的示例中，我将 Pod 的 app 标签设置为 my-app，然后使用 PodAffinity 将它们调度到同一节点上。在这个示例中，我使用了 requiredDuringSchedulingIgnoredDuringExecution 选项，这意味着只有当满足 Pod 的标签选择器时，才会将 Pod 调度到同一节点上。同时，我还指定了 topologyKey，这样 Pod 会被调度到具有相同标签的节点上。

Inter-Pod Anti-Affinity

假设有一个 Kubernetes 集群，其中有多个节点，每个节点都有不同的标签。现在我想**将具有不同标签的多个 Pod 调度到不同节点上**，以提高应用程序的可靠性。这时，可以使用 Inter-Pod Anti-Affinity 进行调度。以下是一个示例：

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16       - name: my-container
17         image: nginx
18       affinity:
19         podAntiAffinity:
20           requiredDuringSchedulingIgnoredDuringExecution:
21             - labelSelector:
22                 matchExpressions:
23                   - key: app
24                     operator: NotIn
25                     values:
26                       - my-app
27             topologyKey: "kubernetes.io/hostname"
```

在上面的示例中，我将 Pod 的 app 标签设置为 my-app，然后使用 PodAntiAffinity 将它们调度到不同节点上。在这个示例中，我使用了 requiredDuringSchedulingIgnoredDuringExecution 选项，这意味着只有当满足 Pod 的标签选择器时，才会将 Pod 调度到不同节点上。同时，我还指定了 topologyKey，这样 Pod 会被调度到不同标签的节点上。

Node Affinity

Node Affinity 是 Kubernetes 中的一种调度策略，它允许您根据节点的属性或状态，**将 Pod 调度到特定的节点上**。与 NodeSelector 不同，Node Affinity 具有更强的表达能力，可以通过使用节点标签、节点标注、节点资源等属性来实现更复杂的调度策略。

以下是一个示例，演示如何使用 Node Affinity 将 Pod 调度到拥有特定标签的节点上：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
```

```

4   name: my-pod
5   spec:
6     containers:
7     - name: my-container
8       image: nginx
9     affinity:
10      nodeAffinity:
11        requiredDuringSchedulingIgnoredDuringExecution:
12          nodeSelectorTerms:
13          - matchExpressions:
14            - key: disktype
15              operator: In
16              values:
17            - ssd

```

在上面的示例中，我将 Pod 的 disktype 标签设置为 ssd，然后使用 Node Affinity 将它们调度到拥有相同标签的节点上。在这个示例中，我使用了 requiredDuringSchedulingIgnoredDuringExecution 选项，这意味着只有当满足节点的标签选择器时，才会将 Pod 调度到特定的节点上。同时，我还使用了 matchExpressions，这样可以更详细地描述节点选择器的匹配规则。

除了 requiredDuringSchedulingIgnoredDuringExecution 外，还有 preferredDuringSchedulingIgnoredDuringExecution 选项，它允许您指定当没有满足 requiredDuringSchedulingIgnoredDuringExecution 的节点时，可以选择哪些节点进行调度。此外，您还可以使用 nodeSelectorTerms 来指定多个节点选择器，以实现更复杂的调度策略。

Pod 亲和性调度

Pod 亲和性调度是 Kubernetes 中的一种调度策略，它可以根据 Pod 之间的关系，将 Pod 调度到特定的节点上。

Pod Affinity

Pod Affinity 是一种 Pod 亲和性调度策略，它可以根据 Pod 之间的关系，将具有相同标签的多个 Pod 调度到同一节点上，并且可以根据节点的标签来选择调度的节点。以下是一个 Pod Affinity 的示例：

```

1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: my-deployment
5   spec:
6     replicas: 3
7     selector:
8       matchLabels:
9       app: my-app
10    template:
11      metadata:
12        labels:
13        app: my-app
14      spec:
15        containers:
16        - name: my-container
17          image: nginx
18        affinity:
19          podAffinity:
20            requiredDuringSchedulingIgnoredDuringExecution:

```

```

21         - labelSelector:
22             matchExpressions:
23             - key: app
24               operator: In
25               values:
26             - my-app
27         topologyKey: "kubernetes.io/hostname"
28     podAntiAffinity:
29         requiredDuringSchedulingIgnoredDuringExecution:
30         - labelSelector:
31             matchExpressions:
32             - key: app
33               operator: NotIn
34               values:
35             - my-app
36         topologyKey: "kubernetes.io/hostname"

```

在上面的示例中，我将 Pod 的 app 标签设置为 my-app，并使用 Pod Affinity 将具有相同标签的多个 Pod 调度到同一节点上，并且使用 Pod Anti-Affinity 将具有不同标签的多个 Pod 调度到不同的节点上。同时，我还使用了 topologyKey 来指定节点的标签，以更精确地选择调度的节点。

Inter-Pod Anti-Affinity

Inter-Pod Anti-Affinity 是一种 Pod 亲和性调度策略，它可以根据 Pod 之间的关系，**将具有不同标签的多个 Pod 调度到不同的节点上**。以下是一个 Inter-Pod Anti-Affinity 的示例：

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4      name: my-deployment
5  spec:
6      replicas: 3
7      selector:
8          matchLabels:
9          app: my-app
10     template:
11         metadata:
12             labels:
13             app: my-app
14         spec:
15             containers:
16             - name: my-container
17               image: nginx
18             affinity:
19                 podAntiAffinity:
20                     requiredDuringSchedulingIgnoredDuringExecution:
21                     - labelSelector:
22                         matchExpressions:
23                         - key: app
24                           operator: NotIn
25                           values:
26                         - my-app
27                     topologyKey: "kubernetes.io/hostname"

```

在上面的示例中，我将 Pod 的 app 标签设置为 my-app，并使用 Inter-Pod Anti-Affinity 将具有不同标签的多个 Pod 调度到不同的节点上。

Inter-Pod Affinity

Inter-Pod Affinity 是一种 Pod 亲和性调度策略，它可以根据 Pod 之间的关系，**将具有相同标签的多个 Pod 调度到同一节点上**。以下是一个 Inter-Pod Affinity 的示例：

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: my-app
10   template:
11     metadata:
12       labels:
13         app: my-app
14     spec:
15       containers:
16       - name: my-container
17         image: nginx
18       affinity:
19         podAffinity:
20           requiredDuringSchedulingIgnoredDuringExecution:
21             - labelSelector:
22                 matchExpressions:
23                   - key: app
24                     operator: In
25                     values:
26                       - my-app
27               topologyKey: "kubernetes.io/hostname"
```

在上面的示例中，我将 Pod 的 app 标签设置为 my-app，并使用 Inter-Pod Affinity 将具有相同标签的多个 Pod 调度到同一个节点上。

亲和性调度的优化和注意事项

避免节点资源耗尽

在使用节点亲和性调度时，确保节点资源能够满足 Pod 的需求。例如，如果设置了节点亲和性调度，但节点资源已经耗尽，那么新的 Pod 将无法调度。因此，在设置节点亲和性调度时，需要仔细考虑节点资源。

避免节点单点故障

在使用节点亲和性调度时，确保不要将相邻的 Pod 调度到同一个节点上，以避免节点单点故障导致 Pod 不可用。例如，如果将多个具有相同标签的 Pod 调度到同一个节点上，那么如果该节点发生故障，所有 Pod 将不可用。

注意亲和性调度的优先级

在使用亲和性调度时，需要注意调度策略的优先级。例如，如果使用 Pod 亲和性调度和节点亲和性调度，那么节点亲和性调度将具有更高的优先级。因此，在设置亲和性调度时，需要了解调度策略的优先级，并根据需要进行调整。

关注亲和性调度的性能和可靠性

在使用亲和性调度时，需要关注其性能和可靠性。例如，如果使用节点亲和性调度来提高网络性能，但实际上在某些情况下会导致网络拥塞或延迟，那么可能需要重新考虑亲和性调度的策略，以提高应用程序的性能和可靠性。

定向调度

定向调度的概念和原理

定向调度是 Kubernetes 集群中的一种调度方式，它可以将某些特定的 Pod 调度到指定的节点上。在某些场景下，我们需要将一些 Pod 调度到特定的节点上，例如需要与某些节点上的硬件设备进行通信，或者需要在某些节点上运行特定的应用程序。

在 Kubernetes 中，定向调度可以通过 nodeName、nodeSelector、taints 和 tolerations 等方式实现。

使用 nodeName 实现定向调度

使用 nodeName 实现定向调度是最简单的一种方式，它允许我们直接指定某个节点的名称，将 Pod 调度到该节点上。使用 nodeName 实现定向调度需要在 Pod 的 YAML 配置文件中添加以下字段：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    nodeName: node1 # 指定调度到node1节点上
7    containers:
8      - name: my-container
9        image: nginx
```

在上面的例子中，我们将 Pod 调度到名为 node1 的节点上。

使用 nodeSelector 实现定向调度

使用 nodeSelector 实现定向调度需要先在节点上添加标签，然后在 Pod 的 YAML 配置文件中指定相应的标签选择器。只有满足标签选择器条件的节点才会被考虑进行 Pod 调度。使用 nodeSelector 实现定向调度需要在 Pod 的 YAML 配置文件中添加以下字段：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: my-pod
5  spec:
6    nodeSelector:
7      disk: ssd
8    containers:
9      - name: my-container
10       image: nginx
```


在上面的例子中，我们使用了一个名为 disk 的标签选择器，只有具有标签 disk=ssd 的节点才会被考虑进行 Pod 调度。

```
1 | kubectl label nodes node1 disk=ssd
```

污点和容忍

污点

污点的概念和作用

在 Kubernetes 中，**污点是一种机制，用于标记节点上不允许运行某些 Pod 的情况**。污点可以被视为一种负面标记，它表示节点上存在一些问题，如运行的服务已满负荷或需要进行维护等。

污点可以帮助 Kubernetes 集群管理者更好地掌握节点的状态，有助于提高集群的安全性和稳定性。当节点出现问题时，Kubernetes 可以自动将 Pod 调度到其他可用的节点上，以确保集群的正常运行。

污点的标签和选择器

在 Kubernetes 中，污点是通过节点的标签和选择器来管理的。节点的标签可以用来描述节点的状态和属性，选择器可以用来选择满足特定要求的节点。

要为节点设置污点，您需要在节点上添加一个 taints 字段。该字段包含一个或多个污点，每个污点由三个部分组成：键、值和效果。

键和值用于标识污点，效果用于描述污点对 Pod 调度的影响。效果有三种类型：

- NoSchedule：表示节点上不允许运行任何未经容忍的 Pod。
- PreferNoSchedule：表示节点上不鼓励运行未经容忍的 Pod，但仍可运行。
- NoExecute：表示节点上的已有 Pod 将被驱逐，并阻止运行未经容忍的新 Pod。

污点的常见场景和示例

污点的常见使用场景和示例包括：

- 防止 Pod 调度到过载或维护中的节点上。
- 在集群中保障特定类型的 Pod 运行在特定的节点上。
- 为集群中的敏感或关键任务分配专用的节点。

下面是一个 YAML 示例代码，用于为节点添加污点：

```
1 | apiVersion: v1
2 | kind: Node
3 | metadata:
4 |   name: node1
5 | spec:
6 |   taints:
7 |   - key: workload
8 |     value: high
9 |     effect: NoSchedule
```

在上面的示例中，我们为名为 node1 的节点添加了一个名为 workload 的污点，它的值为 high，效果为 NoSchedule。这意味着该节点不允许运行未经容忍的 Pod。

污点常用命令

- 为node设置污点(PreferNoSchedule)

```
1 kubectl taint nodes node1 tag=heima:PreferNoSchedule
```

- 取消污点PreferNoSchedule, 创建NoSchedule污点

```
1 kubectl taint nodes all-in-one tag:PreferNoSchedule-
2 kubectl taint nodes all-in-one tag-heima:NoSchedule
```

容忍

容忍的概念和作用

在 Kubernetes 中，**容忍是一种机制，用于允许某些 Pod 运行在不满足要求的节点上**。这些节点可能有污点，或者不满足其他特定的条件，但如果 Pod 配置了相应的容忍规则，它们仍然可以被调度到这些节点上。

容忍可以帮助 Kubernetes 集群管理者更灵活地管理集群中的 Pod。当集群中的节点出现问题或需要维护时，容忍可以允许一些 Pod 在不满足要求的节点上运行，以确保集群的正常运行。

容忍的标签和选择器

在 Kubernetes 中，容忍是通过 Pod 的标签和选择器来管理的。Pod 的标签可以用来描述 Pod 的状态和属性，选择器可以用来选择满足特定要求的节点。

要为 Pod 设置容忍规则，您需要在 Pod 的规范中添加一个 tolerations 字段。该字段包含一个或多个容忍规则，每个规则由三个部分组成：键、操作符和值。

键和值用于标识节点的标签或污点，操作符用于描述容忍规则的类型。操作符有三种类型：

- Equal：表示容忍规则的值与节点的标签或污点的值相等。
- Exists：表示容忍规则的键存在于节点的标签或污点的键中。
- NotEqual：表示容忍规则的值与节点的标签或污点的值不相等。

容忍的常见场景和示例

容忍的常见使用场景和示例包括：

- 允许一些 Pod 运行在节点上，即使它们被标记为污点。
- 允许一些 Pod 运行在不满足要求的节点上，以保证集群的稳定性。
- 允许一些 Pod 在节点维护期间运行，以确保集群的正常运行。

下面是一个 YAML 示例代码，用于为 Pod 设置容忍规则：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx
5 spec:
6   containers:
7   - name: nginx
8     image: nginx:1.21.1
9     ports:
10    - containerPort: 80
11   tolerations:
```

```
12 - key: workload
13   operator: Equal
14   value: high
15   effect: NoSchedule # 添加容忍的规则，这里必须和标记的污点规则相同
```

在上面的示例中，我们为名为 nginx 的 Pod 添加了一个容忍规则，它的键为 workload，操作符为 Equal，值为 high，效果为 NoSchedule。这意味着该 Pod 可以运行在被标记为 workload=high 的节点上。

kubernetes 网络架构

Kubernetes 网络模型和设计原则

Kubernetes 网络模型的概念和意义

Kubernetes 网络模型是指 Kubernetes 集群中的 Pod 网络互联模型。Kubernetes 网络模型的设计旨在提供一种灵活、可扩展、安全和高效的网络架构，使得应用程序可以在 Kubernetes 集群中更加方便地进行开发、部署和运行。

Kubernetes 网络模型的主要目标包括：

1. 提供可扩展、高效、动态的网络互联机制，使得 Pod 可以在集群内相互通信。
2. 提供多种网络插件和驱动程序，使得 Kubernetes 可以支持不同的网络架构和技术。
3. 提供网络安全和隔离机制，保护 Kubernetes 集群中的应用程序和数据不受恶意攻击和安全漏洞的影响。
4. 提供网络质量保障和服务质量管理机制，使得应用程序可以获得足够的带宽、低延迟和高可靠性的网络服务。

Kubernetes 网络设计的原则和考虑因素

在设计 Kubernetes 网络架构时，需要考虑以下因素：

1. 网络拓扑和结构：Kubernetes 网络架构应该基于集群的物理拓扑和逻辑结构进行设计，以确保网络互联的高效性和可扩展性。
2. 网络插件和驱动程序：Kubernetes 网络架构应该支持多种网络插件和驱动程序，以满足不同的网络架构和技术需求。
3. 网络安全和隔离：Kubernetes 网络架构应该提供网络安全和隔离机制，以保护应用程序和数据的安全性和隐私性。
4. 网络质量和服务质量：Kubernetes 网络架构应该提供网络质量保障和服务质量管理机制，以确保应用程序可以获得足够的带宽、低延迟和高可靠性的网络服务。
5. 可扩展性和灵活性：Kubernetes 网络架构应该具有良好的可扩展性和灵活性，以适应不同规模和复杂度的应用场景。

Kubernetes 网络模型的体系结构和组件

Kubernetes 网络模型的体系结构和组件包括：

1. Pod 网络：Kubernetes 中的 Pod 网络是指 Pod 之间的网络互联机制。Pod 网络可以使用多种网络插件和驱动程序，例如 Flannel、Calico、Cilium、Weave Net 等。
2. Service 网络：Kubernetes 中的 Service 网络是指 Service 对象的网络互联机制。Service 网络可以使用 kube-proxy 等网络组件来实现负载均衡和流量转发功能。
3. Ingress 网络：Kubernetes 中的 Ingress 网络是指 Ingress 对象的网络互联机制。Ingress 网络可以使用 Ingress Controller 等网络组件来实现 HTTP(S) 流量的路由和负载均衡功能。
4. Network Policy：Kubernetes 中的 Network Policy 是一种网络安全和隔离机制，可以通过标签选择器来限制来自不同来源和目标的网络流量。Network Policy 可以实现 Pod 级别和 Namespace

级别的网络隔离和安全策略。

Kubernetes 网络模型的可扩展性和插件化

Kubernetes 网络模型的可扩展性和插件化是其设计的重要特点。通过使用不同的网络插件和驱动程序，可以支持多种不同的网络架构和技术，例如 Overlay 网络、VXLAN、BGP、IPsec 等。这些网络插件和驱动程序可以根据不同的应用场景和需求进行选择和配置，以满足不同的网络连接和传输需求。

Kubernetes 网络模型的安全性和隔离性

Kubernetes 网络模型的安全性和隔离性是其设计的另一个重要特点。通过使用 Network Policy 等网络安全机制，可以实现对 Pod 和 Service 的网络访问进行限制和控制，以保护应用程序和数据的安全性和隐私性。此外，还可以使用 TLS 和 RBAC 等安全机制，进一步加强 Kubernetes 集群的安全性和隔离性。

Kubernetes 网络模型的质量保障和服务质量管理

Kubernetes 网络模型的质量保障和服务质量管理是其设计的另一个重要特点。通过使用 QoS 和 Pod Anti-Affinity 等机制，可以保证应用程序可以获得足够的带宽、低延迟和高可靠性的网络服务。此外，还可以使用 L7 负载均衡器和 DNS 解析器等网络组件，进一步提高 Kubernetes 集群的网络性能和可用性。

Kubernetes 网络组件和功能介绍

kube-proxy

kube-proxy 是 Kubernetes 中一个重要的网络组件，它负责实现 Kubernetes Service 的网络代理和负载均衡功能。kube-proxy 可以将 Service 对象转换为 iptables 规则，使得 Service 中定义的 Pod 可以通过虚拟 IP 地址进行访问。

kube-proxy 的工作原理如下：

1. 监听 Kubernetes API Server 上的 Service 和 Endpoints 对象变化。
2. 根据 Service 和 Endpoints 的变化，生成或更新 iptables 规则。
3. 根据 iptables 规则，将流量转发到 Service 中定义的后端 Pod。

kube-proxy 支持三种代理模式：Userspace、Iptables 和 IPVS。其中，Iptables 和 IPVS 是 Kubernetes 官方推荐的代理模式。kube-proxy 的默认代理模式是 Iptables。

kube-dns/coredns

kube-dns/coredns 是 Kubernetes 中的一个 DNS 解析组件，它使得在 Kubernetes 集群中的 Pod 和服务可以通过 DNS 名称进行解析和访问。

kube-dns/coredns 的工作原理如下：

1. 监听 Kubernetes API Server 上的 Service 和 Endpoints 对象变化。
2. 将 Service 和 Endpoints 转换为 DNS 记录，存储在 etcd 中。
3. 在 Kubernetes 集群中，Pod 和服务可以通过 DNS 名称进行解析和访问。

kube-dns/coredns 可以为 Kubernetes 集群中的 Pod 和服务提供稳定的 DNS 解析服务，使得应用程序可以更方便地进行服务发现和访问。

Ingress Controller

Ingress Controller 是 Kubernetes 中的一个网络组件，它可以实现 HTTP(S) 流量的路由和负载均衡功能。Ingress Controller 可以通过 Ingress 对象对流量进行路由和负载均衡，支持多种路由策略和负载均衡算法。

Ingress Controller 的工作原理如下：

1. 监听 Kubernetes API Server 上的 Ingress 对象变化。
2. 根据 Ingress 对象中定义的规则，生成或更新负载均衡器的配置。
3. 通过负载均衡器，将流量转发到 Ingress 对象中定义的后端服务。

Kubernetes 支持多种 Ingress Controller 插件和驱动程序，例如 Nginx、Traefik、HAProxy 等。可以根据不同的需求和场景进行选择和配置。

Network Policy

Network Policy 是 Kubernetes 中的一个网络安全和隔离机制，可以通过标签选择器来限制来自不同来源和目标的网络流量。Network Policy 可以实现 Pod 级别和 Namespace 级别的网络隔离和安全策略。

Network Policy 的工作原理如下：

1. 定义 Network Policy 对象，指定源和目标 Pod 之间的网络流量策略。
2. 将 Network Policy 对象应用到指定的 Pod 或 Namespace 上。
3. 根据 Network Policy 中定义的规则，过滤和限制流入和流出的网络流量。

通过使用 Network Policy，可以实现 Kubernetes 集群中的网络安全和隔离，保护应用程序和数据的安全性和隐私性。

Kubernetes 网络插件介绍

Flannel

Flannel 是 Kubernetes 中最常用的网络插件之一，它可以实现 Pod 的跨主机网络通信。Flannel 的工作原理是在每个节点上运行一个代理程序，该代理程序会为每个 Pod 分配一个唯一的 IP 地址，并使用 VXLAN 或者 UDP Tunnel 等技术将 Pod 的网络流量封装在不同的隧道中。

Flannel 的优点是简单易用，适合对网络要求不高的场景。但是，由于 Flannel 使用的是网络隧道技术，可能会对网络性能产生一定的影响。

Calico

Calico 是一个开源的网络和安全解决方案，它可以为 Kubernetes 提供网络和安全策略。Calico 的工作原理是通过 BGP 协议实现网络路由和负载均衡，通过网络策略实现安全隔离和流量控制。

Calico 的优点是高性能、高可靠和可扩展性强，适合对网络和安全要求较高的场景。但是，Calico 的配置和管理比较复杂，需要一定的技术和人力资源支持。

Weave Net

Weave Net 是一个开源的网络插件，它可以为 Kubernetes 提供网络通信和安全隔离。Weave Net 的工作原理是使用虚拟网络技术将 Pod 的网络流量封装在不同的隧道中，从而实现跨主机的网络通信和安全隔离。

Weave Net 的优点是易于使用、性能稳定和可扩展性强，适合对网络和安全要求较高的场景。但是，Weave Net 的网络隧道可能会对网络性能产生一定的影响。

Cilium

Cilium 是一个开源的网络和安全解决方案，它可以为 Kubernetes 提供网络通信和安全策略。Cilium 的工作原理是通过 Linux 内核的 eBPF 技术实现网络路由和负载均衡，通过网络策略实现安全隔离和流量控制。

Cilium 的优点是高性能、高可靠和可扩展性强，同时具有低延迟和低资源占用的特点，适合对网络和安全要求较高的场景。但是，Cilium 的配置和管理比较复杂，需要一定的技术和人力资源支持。

Kubernetes 网络安全性和实践建议

网络隔离和安全组

网络隔离和安全组是 Kubernetes 中保障网络安全的重要手段。可以通过定义 Network Policy 和安全组等方式限制 Pod 和服务之间的网络通信，保护应用程序和数据的安全性和隐私性。

在实践中，建议使用 Network Policy 和安全组等技术实现 Kubernetes 集群中的网络隔离和安全策略，限制网络流量和访问权限，防止网络攻击和数据泄露。

使用 TLS 加密通信

使用 TLS 加密通信是保障 Kubernetes 集群网络安全的重要措施。通过在 Kubernetes 集群中启用 TLS 加密，可以保护 Pod 和服务之间的网络通信，防止敏感信息被窃取或篡改。

在实践中，建议为 Kubernetes 集群中的所有网络通信启用 TLS 加密，使用证书和密钥等方式保护 TLS 的安全性和可靠性。

使用 RBAC 和网络策略控制访问权限

使用 RBAC 和网络策略是实现 Kubernetes 集群网络安全的重要手段。可以通过定义 RBAC 角色和网络策略等方式限制用户和应用程序的访问权限，保护 Kubernetes 集群中的资源和数据安全。

在实践中，建议使用 RBAC 和网络策略等技术实现 Kubernetes 集群中的访问控制和权限管理，限制用户和应用程序的访问权限，防止未经授权的访问和操作。

网络故障排除和调试技巧

网络故障排除和调试技巧是维护 Kubernetes 集群网络稳定和可靠的重要方法。在网络故障发生时，需要使用一些常用的故障排除和调试工具，例如 ping、tracert、tcpdump、nslookup 等，对网络故障进行定位和分析。

在实践中，建议使用网络故障排除和调试工具对 Kubernetes 集群中的网络故障进行定位和分析，找出故障原因并进行修复和优化。同时，也需要定期进行网络性能和负载测试，评估 Kubernetes 集群网络的可靠性和性能，保障应用程序和数据的正常运行。

kubernetes安全架构

Kubernetes 安全模型和设计原则

最小权限原则

最小权限原则是 Kubernetes 安全设计中的基本原则之一，它要求为每个用户和应用程序分配最小的访问权限，避免不必要的权限和访问。

在 Kubernetes 中，可以通过 RBAC 和网络策略等方式实现最小权限原则。通过定义 RBAC 角色和网络策略等方式限制用户和应用程序的访问权限，保护 Kubernetes 集群中的资源和数据安全。

基于角色的访问控制

基于角色的访问控制是 Kubernetes 安全设计中的重要手段之一，它可以为每个用户和应用程序分配不同的访问权限和角色，保护 Kubernetes 集群中的资源和数据安全。

在 Kubernetes 中，可以通过定义 RBAC 角色和 ClusterRole 等方式实现基于角色的访问控制。通过为每个用户和应用程序分配不同的访问权限和角色，限制用户和应用程序的访问权限，保护 Kubernetes 集群中的资源和数据安全。

安全审计和日志记录

安全审计和日志记录是 Kubernetes 安全设计中的重要手段之一，它可以记录和监控 Kubernetes 集群中的所有访问和操作，实现安全审计和日志分析。

在 Kubernetes 中，可以通过定义 Pod 和容器的日志记录、使用安全审计工具和实时监控等方式实现安全审计和日志记录。通过记录和监控 Kubernetes 集群中的所有访问和操作，及时发现和防止未经授权的访问和操作，保护 Kubernetes 集群中的资源和数据安全。

安全加固和漏洞管理

安全加固和漏洞管理是 Kubernetes 安全设计中的重要手段之一，它可以及时发现和修复 Kubernetes 集群中的安全漏洞和风险，保护 Kubernetes 集群中的资源和数据安全。

在 Kubernetes 中，可以通过定期更新和升级 Kubernetes 的组件和插件、使用安全扫描工具和漏洞管理工具等方式实现安全加固和漏洞管理。通过定期检查和修复 Kubernetes 集群中的安全漏洞和风险，保护 Kubernetes 集群中的资源和数据安全。

Kubernetes 安全认证

访问控制

客户端

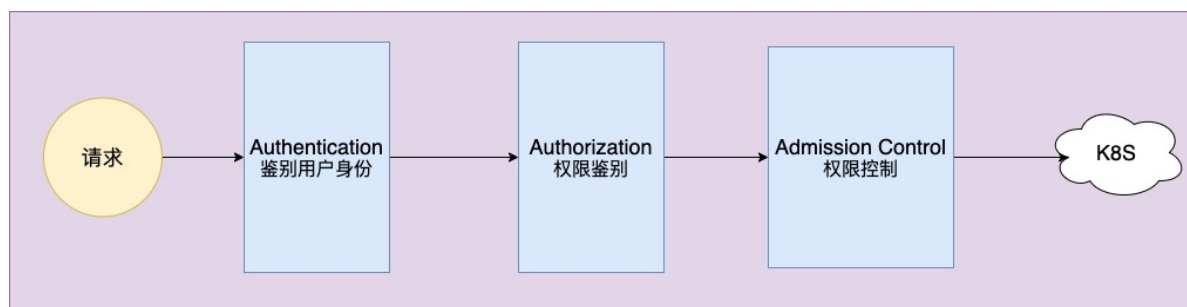
在 Kubernetes 集群中，客户端通常由两类：

- User Account：一般是独立于 Kubernetes 之外的其他服务管理的用户账号。
- Service Account：Kubernetes 管理的账号，用于为 Pod 的服务进程在访问 Kubernetes 时提供身份标识

认证/授权/准入控制

API Service 是访问和管理资源的唯一入口,任何一个请求都需要访问 API Service,经历如下流程

- Authentication(认证)：身份鉴别,只有正确的账号才能访问
- Authorization(授权)：判断账号是否有权限执行相应的动作
- Admission Control（注入控制）：用于补充授权机制以实现更加精细的访问控制功能

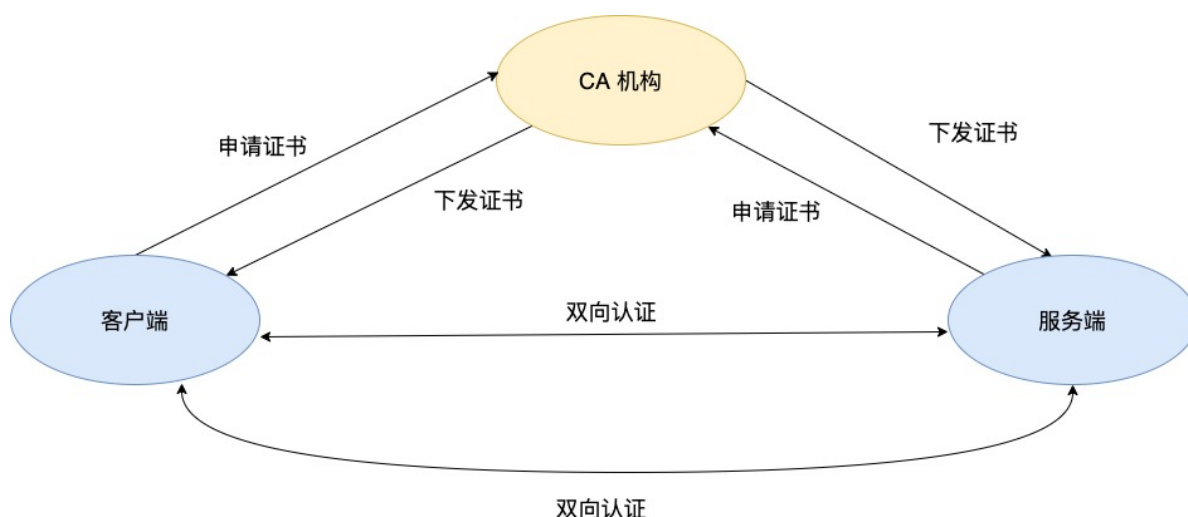


认证管理

K8S 客户端认证方式

K8S 需要对账户进行认证，在 K8S 中提供了三种认证方式

- HTTP Base 认证
 - 通过用户名 + 密码进行认证
 - 该方式通过将用户名与密码通过 base64 进行编码之后放入 HTTP 请求的 Header 的 Authorization 域里面发送给服务端,服务端接受到信息,提取用户名与密码之后在进行解码,之后在进行用户认证
 - 由于使用 base64 进行编码可以通过反编码得到用户名密码因此该方式不安全
- HTTP Token
 - 通过一个 Token 来标识一个合法的用户
 - 在该方式中每个用户有一个唯一的 Token 用来标识用户,当客户端发起请求的时候在 HTTP 的 Header 中放入 Token,当服务端收到客户端请求之后提取客户端的 Token 信息,然后与服务器中保存的 Token 进行对比,进而验证用户信息
- HTTPS 证书认证
 - 基于 CA 根证书签名的双向证书认证方式
 - 这种认证方式是安全性最高的一种方式，但是同时也是操作起来最麻烦的一种方式。



HTTPS 认证

- 证书申请和下发,HTTPS 通信双方向 CA 机构进行证书申请,CA机构下发根证书，服务端证书以及私钥给申请者
- 客户端和服务端进行双向认证
 - 客户端向服务端发起请求,服务端向客户端发送自己的证书,客户端通过自己的私钥对服务端的证书进行解密获取服务端的公钥,客户端利用服务端的公钥认证证书信息,如果信息一致则信任服务端
 - 客户端发送自己的证书给服务端,服务端接受到客户端的证书之后使用私钥解密获取客户端的公钥,之后使用客户端的公钥进行认证证书信息,如果一致则认可客户端
- 服务端和客户端进行通信
 - 服务端和客户端协商解密方案,并且产生一个随机的密钥进行加密
 - 客户端将密钥发给服务端,之后双方使用该加密密钥进行数据通信

注意: Kubernetes允许同时配置多种认证方式，只要其中任意一个方式认证通过即可

HTTP Token 认证

Token 认证是 Kubernetes 中另一种常用的客户端身份认证方式，它使用基于令牌的身份验证验证客户端的身份。Token 需要存储在 Kubernetes 集群中。

这种认证方式是用一个很长的难以被模仿的字符串--Token 来表明客户端身份的一种方式。每个 Token 对应一个用户名，当客户端发起 API 调用请求的时候，需要在 HTTP 的 Header 中放入 Token，API Server 接受到 Token 后会和服务器中保存的 Token 进行比对，然后进行用户身份认证的过程。

- `--token-auth-file=/etc/kubernetes/pki/token.csv` 指定了静态token文件，这个文件的格式如下：

```
1 token,user,uid,"group1,group2,group3"
```

- 生成token方式如下：

```
1 export BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x | tr -d ' ')
2 cat > token.csv <<EOF
3 ${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-bootstrap"
4 EOF
```

- 请求Api时只要要在Authorization头中加入Bearer Token即可

```
1 [root@localhost pki]# curl -k --header "Authorization: Bearer
2 7861c93d890ae64287b0717111f2a869" https://localhost:6443/api
3 {
4   "kind": "APIVersions",
5   "versions": [
6     "v1"
7   ],
8   "serverAddressByClientCIDRs": [
9     {
10      "clientCIDR": "0.0.0.0/0",
11      "serverAddress": "192.168.116.128:6443"
12    }
13  ]
14 }
```

授权管理 (HTTPS认证)

概述

- 授权发生在认证成功之后，通过认证就可以知道请求用户是谁，然后kubernetes会根据事先定义的授权策略来决定用户是否有权限访问，这个过程就称为授权。
- 每个发送到API Server的请求都带上了用户和资源的信息：比如发送请求的用户、请求的路径、请求的动作等，授权就是根据这些信息和授权策略进行比较，如果符合策略，则认为授权通过，否则会返回错误。

授权策略

- AlwaysDeny：表示拒绝所有请求，一般用于测试。
- AlwaysAllow：允许接收所有的请求，相当于集群不需要授权流程（kubernetes默认的策略）。
- ABAC：基于属性的访问控制，表示使用用户配置的授权规则对用户请求进行匹配和控制。
- Webhook：通过调用外部REST服务对用户进行授权。

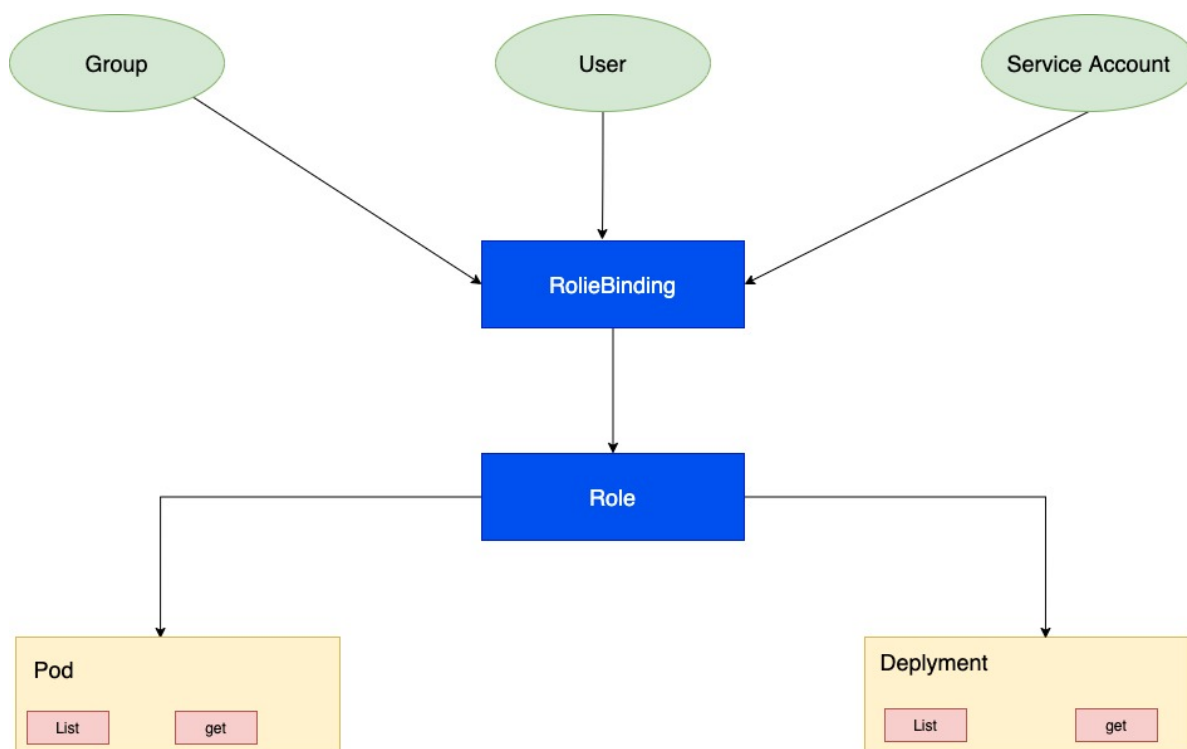
- Node：是一种专用模式，用于对kubelet发出的请求进行访问控制。
- RBAC：基于角色的访问控制（kubeadm安装方式下的默认选项）。

RBAC

RBAC (Role Based Access Control)：基于角色的访问控制，主要是在描述一件事情：给哪些对象授权了哪些权限。

RBAC涉及到了下面几个概念：

- 对象：User、Groups、ServiceAccount。
- 角色：代表着一组定义在资源上的可操作的动作（权限）的集合。
- 绑定：将定义好的角色和用户绑定在一起。



RBAC还引入了4个顶级资源对象

- Role、ClusterRole：角色，用于指定一组权限。
- RoleBinding、ClusterRoleBinding：角色绑定，用于将角色（权限的集合）赋予给对象。

Role/ClusterRole

一个角色就是一组权限的集合，这里的权限都是许可形式的（白名单）。

role 资源清单文件

```

1  # Role只能对命名空间的资源进行授权，需要指定namespace
2  apiVersion: rbac.authorization.k8s.io/v1
3  kind: Role
4  metadata:
5    name: authorization-role
6    namespace: dev
7  rules:
8    - apiGroups: [""] # 支持的API组列表，""空字符串，表示核心API群
9      resources: ["pods"] # 支持的资源对象列表
10     verbs: ["get","watch","list"]
  
```

ClusterRole的资源清单文件：

```

1 # ClusterRole可以对集群范围内的资源、跨namespace的范围资源、非资源类型进行授权
2 apiVersion: rbac.authorization.k8s.io/v1
3 kind: ClusterRole
4 metadata:
5   name: authorization-clusterrole
6 rules:
7   - apiGroups: ["" ] # 支持的API组列表, ""空字符串, 表示核心API群
8     resources: ["pods"] # 支持的资源对象列表
9     verbs: ["get","watch","list"]

```

rules中的参数说明:

apiGroups: 支持的API组列表。

- "", "apps", "autoscaling", "batch".

resources: 支持的资源对象列表。

- "services", "endpoints", "pods", "secrets", "configmaps", "crontabs", "deployments", "jobs", "nodes", "rolebindings", "clusterroles", "daemonsets", "replicasets", "statefulsets", "horizontalpodautoscalers", "replicationcontrollers", "cronjobs".

verbs: 对资源对象的操作方法列表。

- "get", "list", "watch", "create", "update", "patch", "delete", "exec".

RoleBinding、ClusterRoleBinding

RoleBinding、ClusterRoleBinding

角色绑定用来把一个角色绑定到一个目标对象上, 绑定目标可以是User、Group或者ServiceAccount。

RoleBinding的资源清单文件:

```

1 # RoleBinding可以将同一namespace中的subject对象绑定到某个Role下, 则此Subject具有该
  Role定义的权限
2 apiVersion: rbac.authorization.k8s.io/v1
3 kind: RoleBinding
4 metadata:
5   name: authorization-role-binding
6   namespace: dev
7 subjects:
8   - kind: User
9     name: SR
10    apiGroup: rbac.authorization.k8s.io
11 roleRef:
12   apiGroup: rbac.authorization.k8s.io
13   kind: Role
14   name: authorization-role

```

ClusterRoleBinding的资源清单文件:

```

1 # ClusterRoleBinding在整个集群级别和所有namespaces将特定的subject与ClusterRole绑定，授予权限
2 apiVersion: rbac.authorization.k8s.io/v1
3 kind: ClusterRoleBinding
4 metadata:
5   name: authorization-clusterrole-binding
6 subjects:
7   - kind: User
8     name: SR
9     apiGroup: rbac.authorization.k8s.io
10 roleRef:
11   apiGroup: rbac.authorization.k8s.io
12   kind: ClusterRole
13   name: authorization-clusterrole

```

RoleBinding引用ClusterRole进行授权

- RoleBinding可以引用ClusterRole，对属于同一命名空间内ClusterRole定义的资源主体进行授权。
- 一种很常用的做法是，集群管理员为集群范围预定义好一组角色（ClusterRole），然后在多个命名空间中重复使用这些ClusterRole。这样可以大幅度提高授权管理工作效率，也使得各个命名空间下的基础性授权规则和使用体验保持一致。

```

1 # 虽然authorization-clusterrole是一个集群角色，但是因为使用了RoleBinding
2 # 所以SR只能读取dev命名空间中的资源
3 apiVersion: rbac.authorization.k8s.io/v1
4 kind: RoleBinding
5 metadata:
6   name: authorization-clusterrole-binding
7 subjects:
8   - kind: User
9     name: SR
10    apiGroup: rbac.authorization.k8s.io
11 roleRef:
12   apiGroup: rbac.authorization.k8s.io
13   kind: ClusterRole
14   name: authorization-clusterrole

```

RBAC实战：创建账号

- 创建一个只能管理dev命名空间下Pods资源的账号。

```

1 # 创建证书
2 cd /etc/kubernetes/pki/ && (umask 077;openssl genrsa -out devman.key 2048)
3
4 # 签名申请 申请的用户是devman，组是devgroup
5 openssl req -new -key devman.key -out devman.csr -subj
6   "/CN=devman/O=devgroup"
7
8 # 签署证书
9 openssl x509 -req -in devman.csr -CA ca.crt -CAkey ca.key -CAcreateserial -
10 out devman.crt -days 3650

```

```

1 # 设置集群/用户/上下文
2 kubectl config set-cluster kubernetes --embed-certs=true --certificate-
  authority=/etc/kubernetes/pki/ca.crt --server=https://192.168.116.128:6443
3
4 kubectl config set-credentials devman --embed-certs=true --client-
  certificate=/etc/kubernetes/pki/devman.crt --client-
  key=/etc/kubernetes/pki/devman.key
5
6 kubectl config set-context devman@kubernetes --cluster=kubernetes --
  user=devman
7
8
9 # 切换devman 账号
10 kubectl config use-context devman@kubernetes
11
12 # 查看
13 kubectl get pods -n dev

```

- 切换到 admin 账户

```

1 kubectl config use-context kubernetes-admin@kubernetes

```

可以通过 `kubectl config view` 查看用户/集群配置信息。

创建Role和RoleBinding, 为devman授权

```

1 # 创建配置文件
2 cat > dev-role.yaml << EOF
3 apiVersion: rbac.authorization.k8s.io/v1
4 kind: Role
5 metadata:
6   name: dev-role
7   namespace: dev
8 rules:
9   - apiGroups: ["" ] # 支持的API组列表, ""空字符串, 表示核心API群
10     resources: ["pods"] # 支持的资源对象列表
11     verbs: ["get", "watch", "list"]
12
13 ---
14
15 kind: RoleBinding
16 apiVersion: rbac.authorization.k8s.io/v1
17 metadata:
18   name: authorization-role-binding
19   namespace: dev
20 subjects:
21   # 针对用户类型
22   - kind: User
23     name: devman
24     apiGroup: rbac.authorization.k8s.io
25 roleRef:
26   # 针对 role
27   kind: Role
28   name: dev-role

```

```
29     apiGroup: rbac.authorization.k8s.io
30 EOF
31
32 # 加载配置文件
33 [root@master k8s]# kubectl create -f dev-role.yaml
34
35 # 切换用户再次验证
36 kubectl config use-context devman@kubernetes
37
38 [root@master k8s]# kubectl get -n dev pod -o wide
```

准入控制

- 通过了前面的认证和授权之后，还需要经过准入控制通过之后，API Server才会处理这个请求。
- 准入控制是一个可配置的控制器列表，可以通过在API Server上通过命令行设置选择执行哪些注入控制器。

```
1 --enable-admission-
  plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,PersistentVolumeLabel,DefaultStorageClass,ResourceQuota,DefaultTolerationSeconds
```

当前可配置的Admission Control (准入控制)

- AlwaysAdmit: 允许所有请求。
- AlwaysDeny: 禁止所有请求，一般用于测试。
- AlwaysPullImages: 在启动容器之前总去下载镜像。
- DenyExecOnPrivileged: 它会拦截所有想在Privileged Container上执行命令的请求。
- ImagePolicyWebhook: 这个插件将允许后端的一个Webhook程序来完成admission controller的功能。
- Service Account: 实现ServiceAccount实现了自动化。
- SecurityContextDeny: 这个插件将使用SecurityContext的Pod中的定义全部失效。
- ResourceQuota: 用于资源配额管理目的，观察所有请求，确保在namespace上的配额不会超标。
- LimitRanger: 用于资源限制管理，作用于namespace上，确保对Pod进行资源限制。
- InitialResources: 为未设置资源请求与限制的Pod，根据其镜像的历史资源的使用情况进行设置。
- NamespaceLifecycle: 如果尝试在一个不存在的namespace中创建资源对象，则该创建请求将被拒绝。当删除一个namespace时，系统将会删除该namespace中所有对象。
- DefaultStorageClass: 为了实现共享存储的动态供应，为未指定StorageClass或PV的PVC尝试匹配默认StorageClass，尽可能减少用户在申请PVC时所需了解的后端存储细节。
- DefaultTolerationSeconds: 这个插件为那些没有设置forgiveness tolerations并具有notready:NoExecute和unreachable:NoExecute两种taints的Pod设置默认的“容忍”时间，为5min。
- PodSecurityPolicy: 这个插件用于在创建或修改Pod时决定是否根据Pod的security context和可用的PodSecurityPolicy对Pod的安全策略进行控制

Kubernetes 安全实践建议

硬化 Kubernetes 集群

在安全 Kubernetes 集群时，需要对集群中的主机和网络进行硬化，以保护 Kubernetes 集群免受未经授权的访问和攻击。具体的措施包括：

- 禁用不必要的服务和端口。
- 配置防火墙，限制网络流量。

- 配置操作系统和主机的安全设置。
- 配置 Kubernetes API Server、etcd、kubelet 等组件的安全设置。

安全配置 Kubernetes 组件

在安全 Kubernetes 集群时，需要对 Kubernetes 组件进行安全配置，以防范可能的安全漏洞和攻击。具体的措施包括：

- 配置 Kubernetes API Server、etcd、kubelet 等组件的安全设置。
- 配置 Kubernetes 组件的访问控制和认证设置。
- 对 Kubernetes 组件进行审计和日志记录。

安全配置容器镜像和容器运行时

在安全 Kubernetes 集群时，需要对容器镜像和容器运行时进行安全配置，以防范可能的安全漏洞和攻击。具体的措施包括：

- 使用安全的容器镜像，避免使用带有已知漏洞的镜像。
- 配置容器运行时的安全设置，例如限制容器的资源使用、限制容器的网络访问等。
- 使用容器运行时提供的安全功能。

使用网络策略控制网络访问

在安全 Kubernetes 集群时，需要使用网络策略控制网络访问，以防范未经授权的访问和攻击。具体的措施包括：

- 针对 Kubernetes 集群中的各个命名空间和服务，使用网络策略控制入站和出站流量。
- 使用网络策略控制 Pod 之间的通信，限制 Pod 之间的网络访问。

使用 TLS 加密通信

在安全 Kubernetes 集群时，需要使用 TLS 加密通信，以保护 Kubernetes 集群中的数据和通信安全。具体的措施包括：

- 配置 Kubernetes 组件使用 TLS 加密通信。
- 配置容器镜像库使用 TLS 加密通信。
- 配置 Pod 使用 TLS 加密通信。

使用 RBAC 和 Pod Security Policies 控制访问权限

在安全 Kubernetes 集群时，需要使用 RBAC 和 Pod Security Policies 控制访问权限，以防范未经授权的访问和攻击。具体的措施包括：

- 使用 RBAC 限制用户和服务账号的访问权限。
- 使用 Pod Security Policies 控制容器的安全策略，例如限制容器的资源使用、限制容器的网络访问等。

使用安全扫描和漏洞管理工具

在安全 Kubernetes 集群时，需要使用安全扫描和漏洞管理工具，及时发现和修复 Kubernetes 集群中的安全漏洞和风险。具体的措施包括：

- 使用安全扫描工具扫描容器镜像，及时发现和修复镜像中的安全漏洞和风险。
- 使用漏洞管理工具管理 Kubernetes 集群中的安全漏洞和风险，及时修复和更新 Kubernetes 组件和插件。

监控和日志管理

监控基础知识

什么是监控？

监控的目的是什么？

监控的类型

基础设施监控

应用程序监控

网络监控

监控的指标

性能指标

容量指标

可用性指标

监控工具和技术

基础设施监控工具

Nagios

Zabbix

应用程序监控工具

New Relic

AppDynamics

日志收集工具

Logstash

Fluentd

Filebeat

数据库监控工具

Prometheus

InfluxDB

日志管理基础知识

什么是日志？

日志的目的是什么？

日志类型

访问日志

错误日志

调试日志

日志级别

错误

警告

信息

调试

日志管理工具和技术

日志收集工具

Logstash

Fluentd

Filebeat

日志分析工具

ELK Stack

Splunk

Graylog

日志存储技术

Elasticsearch

Cassandra

Hadoop

监控和日志管理的最佳实践

架构设计原则

配置和部署最佳实践

警报和响应最佳实践

高级监控和日志管理

监控自动化

自动化监控配置

自动化警报和响应

监控可视化

仪表板

报表

警报

日志分析

日志搜索

过滤

分析

安全监控

入侵检测

安全事件管理

应用实践篇：构建可靠、安全的容器化应用生态

部署一个简单的应用程序

构建多层次的应用程序架构

水平扩展和自动伸缩

灰度发布和滚动更新

故障排除和日志分析

监控和告警设置

备份和恢复策略

高可用和容灾设计

安全加固和访问控制

多集群部署和联邦集群

实战演练篇：构建高可用集群系统

部署 MySQL

单机 MySQL

一主多从 MySQL

多主多从 MySQL

部署 Redis

单机 Redis

集群-主从模式

集群-Redis Cluster模式

部署 RabbitMQ

部署 MongoDB

部署 fastdfs（部署高可用版本）

思考提升篇：为什么使用 Kubernetes

注意事项篇：在容器化平台中的关键注意事项

源码解析篇：深入学习 Kubernetes 源代码
