

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
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)

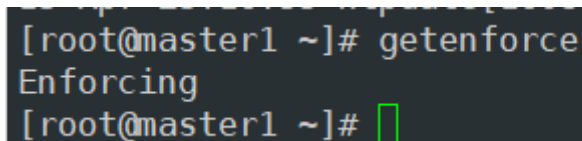
Kubernetes 要求集群中的节点时间必须精确一致, 所以在每个节点上添加时间同步

```
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 ~]#'.

- 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
8 "s#^net.ipv6.conf.all.disable_ipv6.*#net.ipv6.conf.all.disable_ipv6=1#g"
9 /etc/sysctl.conf
10 sed -i
11 "s#^net.ipv6.conf.default.disable_ipv6.*#net.ipv6.conf.default.disable_ipv6=1
12 #g" /etc/sysctl.conf
13 sed -i "s#^net.ipv6.conf.lo.disable_ipv6.*#net.ipv6.conf.lo.disable_ipv6=1#g"
14 /etc/sysctl.conf
15 sed -i "s#^net.ipv6.conf.all.forwarding.*#net.ipv6.conf.all.forwarding=1#g"
16 /etc/sysctl.conf
17 sed -i "s#^net.ipv4.ip_nonlocal_bind.*#net.ipv4.ip_nonlocal_bind=1#g"
18 /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     kubePodCIDR: 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?

```

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

```

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。

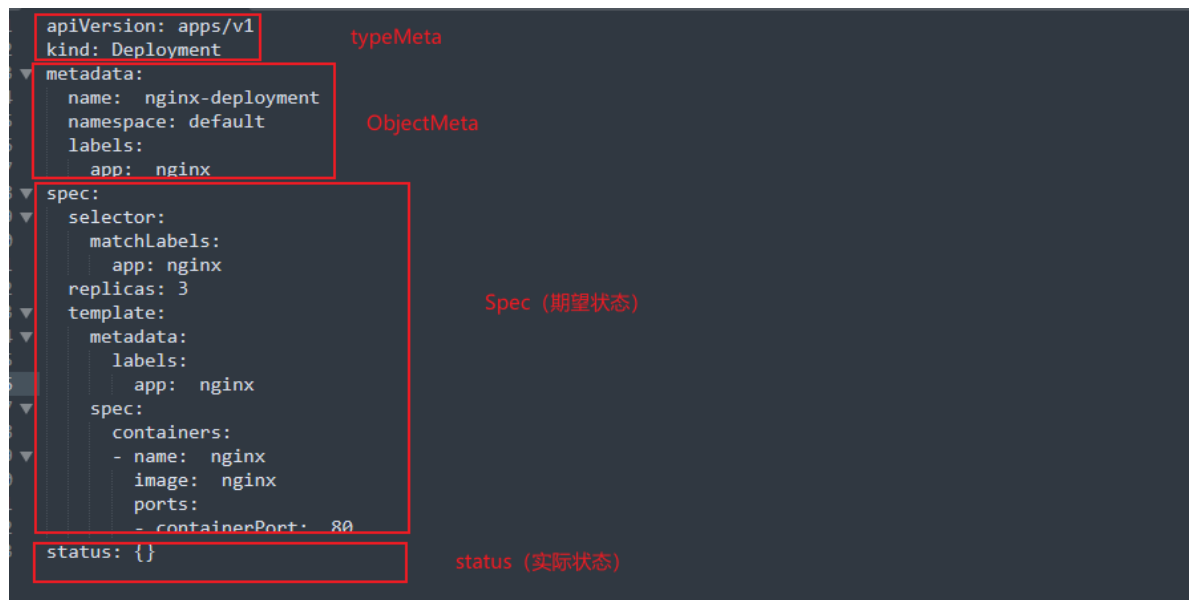
例如，要创建一个 Pod，可以使用以下命令：

```
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、选择器和命名空间

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，可以再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。）

名称空间（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
```

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`。

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-
21       registry.example.com:5000/example/web-example:v1.0，实际开发中，建议锁定镜像的版
22       本。
23     ports:
24     - containerPort: 80
25
26 # 以上的 Pod
```

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-
registry.example.com:5000/example/web-example:v1.0，实际开发中，建议锁定镜像的版
本。
23     imagePullPolicy: Always # 镜像拉取策略：IfNotPresent（默认）、Always、Never
24   - name: arcgis
25     image: registry.cn-shanghai.aliyuncs.com/xudaxian/arcgis/v1.0
26     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]

镜像 Entrypoint	镜像CMD	容器 command	容器 args	最终执行
[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

NAME	STATUS	AGE	
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}'
```

查看命令空间日志

查看指定命令空间详细信息。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  shellCopy code# 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

kubectl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）

命令	简写	含义
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kubectl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
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

命令	语法	含义	参数含义
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
kubectl expose	<code>kubectl expose <资源类型> <资源名称> --port=<端口号></code>	为资源创建服务	<code><资源类型></code> ：要创建服务的资源类型，例如 deployment、pod 等； <code><资源名称></code> ：要创建服务的资源的名称； <code>--port=<端口号></code> ：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 <上下文名称></code>	切换到指定的上下文	<code><上下文名称></code> ：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。

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：

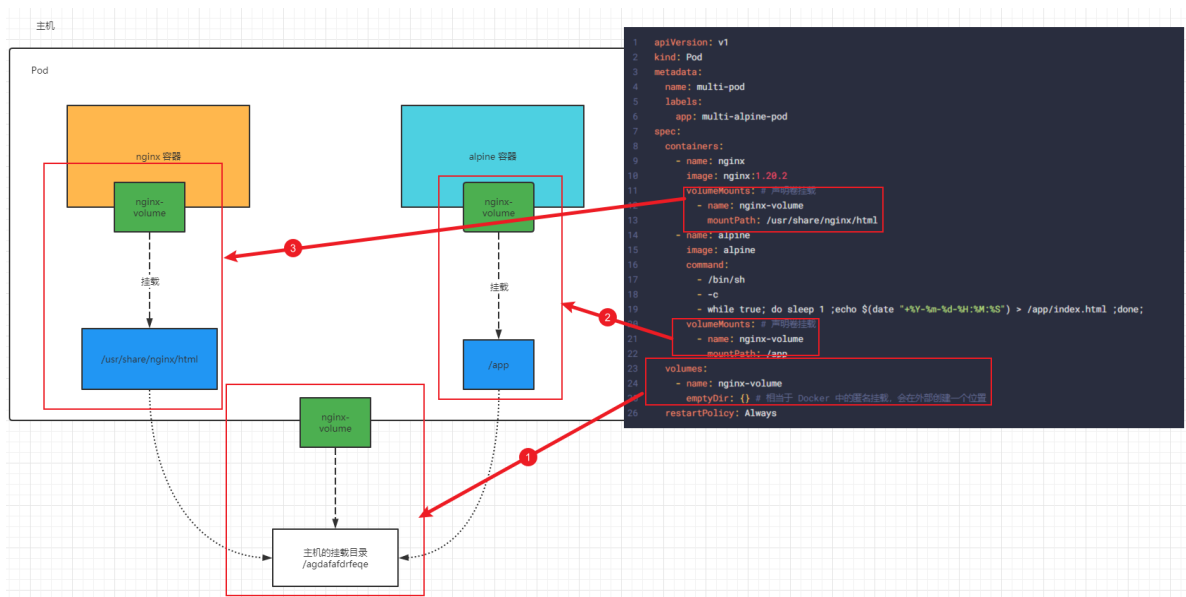
```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: multi-pod
5   labels:
6     app: multi-alpine-pod
7 spec:
8   containers:
9     - name: nginx
10       image: nginx:1.20.2
11       volumeMounts: # 声明卷挂载
12         - name: nginx-volume
```

```

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。

4. **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

1. 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。
2. 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 and 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 | kubectl -h | grep EphemeralContainers
```

```
[root@master1 busybox-1.28.3]# kubectl -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 的亲中性、反亲中性、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 shellCopy code# 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 shellCopy code# 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  shellCopy code# 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>

```

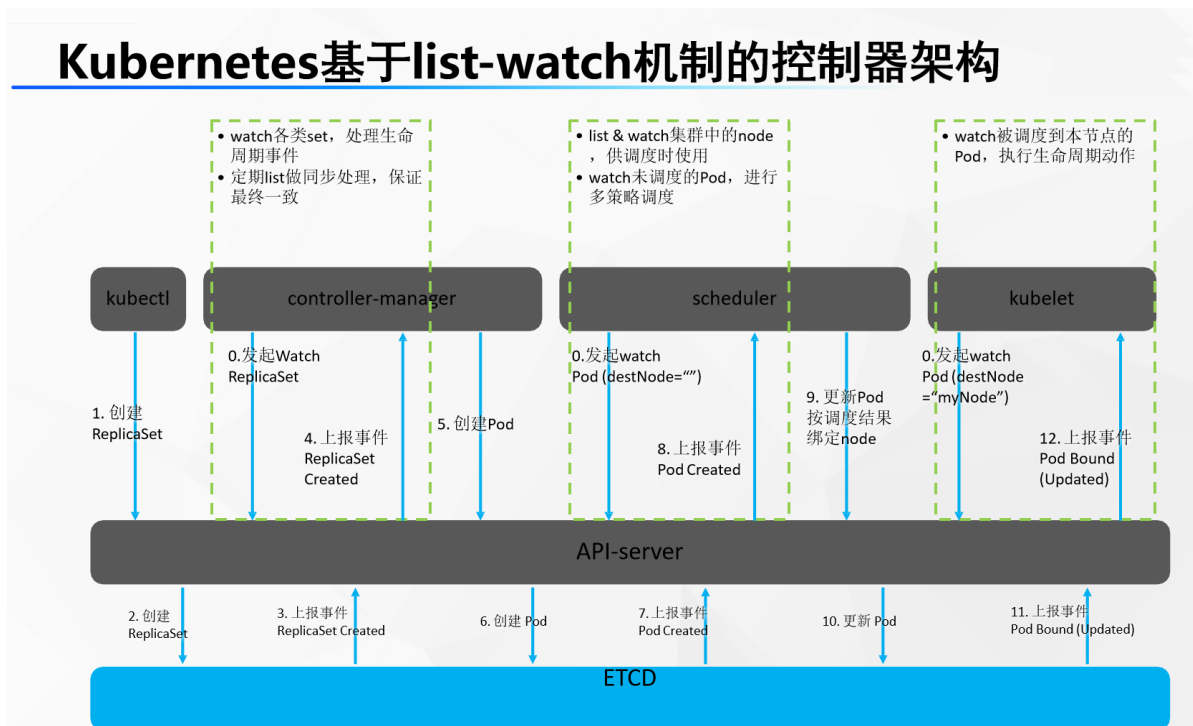
自动调度

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

- 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。
- 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。
- 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。
- 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。
- 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。
- 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。
- 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。

调度流程



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 Toleration 字段声明 Pod 对污点的容忍

通过使用 `Toleration` 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。

Toleration 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。

Kubernetes 控制器：应用程序的调度与管理

控制器概述

控制器的作用和功能

控制器的分类和特点

控制器的核心概念和机制

ReplicaSet 控制器

ReplicaSet 控制器的介绍和功能

管理和操作 ReplicaSet

创建和部署 ReplicaSet

扩容和缩容 ReplicaSet

更新和滚动更新 ReplicaSet

删除和清理 ReplicaSet

示例：使用 ReplicaSet 控制器管理应用程序副本集

Deployment 控制器

Deployment 控制器的介绍和功能

管理和操作 Deployment

创建和部署 Deployment

扩容和缩容 Deployment

更新和回滚 Deployment

暂停和继续 Deployment

删除和清理 Deployment

示例：使用 Deployment 控制器进行应用程序管理和版本控制

DaemonSet 控制器

DaemonSet 控制器的介绍和功能

管理和操作 DaemonSet

创建和部署 DaemonSet

更新和滚动更新 DaemonSet

删除和清理 DaemonSet

示例：使用 DaemonSet 控制器管理守护进程应用程序

StatefulSet 控制器

StatefulSet 控制器的介绍和功能

管理和操作 StatefulSet

创建和部署 StatefulSet

扩容和缩容 StatefulSet

更新和回滚 StatefulSet

删除和清理 StatefulSet

示例：使用 StatefulSet 控制器管理有状态应用程序

Job 控制器

Job 控制器的介绍和功能

管理和操作 Job

创建和运行 Job

查看和监控 Job 状态

清理和删除 Job

示例：使用 Job 控制器运行一次性任务和批处理作业

CronJob 控制器

CronJob 控制器的介绍和功能

管理和操作 CronJob

创建和调度 CronJob

查看和监控 CronJob 状态

更新和删除 CronJob

示例：使用 CronJob 控制器定时调度任务

水平自动伸缩（HPA）控制器

HPA 控制器的介绍和功能

管理和操作 HPA

创建和配置 HPA

监控和自动伸缩 Pod 数量

更新和删除 HPA

示例：使用 HPA 控制器进行自动伸缩和负载管理

服务发现与负载均衡

服务发现的概念和实现

使用 Service 进行内部和外部访问

存储管理和卷配置

存储的概念和类型

使用 ConfigMap 进行配置管理

使用 Secret 进行密钥管理

网络 and 安全性

Kubernetes 网络和通信的配置

安全性和权限控制的实施

监控和日志管理

监控 Kubernetes 集群和应用程序

日志管理和分析的方法

扩展和自动化

扩展 Kubernetes 集群的方法和工具

自动化运维和管理 Kubernetes

核心篇

命名空间和资源限制

自定义资源和扩展

状态管理和滚动更新

高可用和容错

网络策略和入口控制

存储卷插件和 CSI

容器运行时和 CRI

集群自动化和自动伸缩

服务网格和 Istio

进阶篇

高级调度器和定制调度器

自定义控制器和运算符

运维工具和生态系统

多集群管理和联邦集群

安全加固和审计

网络插件和 CNI

存储插件和 CSI 开发

容器镜像和 OCI 标准

应用实践篇

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

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）

扩展篇

etcd 篇

注意事项篇

源码解析篇
