### k8s操作手册

**前言：**

1.蓝色字体表示命令行命令，正式执行时不要复制前面的#号，#号只是提示应该使用root权限操作

2.绿色字体表示注释，有时注释太多就不用绿色表示了

3.注意：本文档的所有操作请先在测环境进行实践，请不要直接在真实的服务器中操作！

**版权声明**：

本文档以开源的形式发布，所有条款如下：

（1）无担保：作者不保证文档内容的准确无误，亦不承担由于使用此文档所导致的任何后果

（2）自由使用：任何人可以出于任何目的而自由地 阅读/链接/打印/转载/引用/分发/再创作 此文档，无需任何附加条件

若您 阅读/链接/打印/转载/引用/分发/再创作 本文档，则说明接受以上2个条款。

作者：李茂福

更新日期：2023-12-29

**★第0章、K8S集群搭建准备工作**

**相关单词原义：**

docker 码头工人

pod 集装箱

kubernetes 舵手，领航员

helm 舵轮，驾驶盘

chart 图表，海图

**①k8s对系统要求**

linux内核在3.10及以上，服务器规格2核cpu，2G内存及以上，可以装在虚拟机里，也可以装在实体机上

**②规划主机名及ip**

k8s的服务器使用固定ip地址，**配置主机名**，要求能解析相应的主机名（master结点）到对应的ip地址，可以使用内网集群的dns服务器或**写入/etc/hosts**文件里。如：

|  |  |
| --- | --- |
| 主机名 | ip地址 |
| k8s-master1.cof-lee.com | 10.99.1.51 |
| k8s-master2.cof-lee.com | 10.99.1.52 |
| k8s-master3.cof-lee.com | 10.99.1.53 |
| k8s-node01.cof-lee.com | 10.99.1.61 |
| k8s-node02.cof-lee.com | 10.99.1.62 |

规划Pod网络： 10.244.0.0/16

规划Service网络： 10.7.0.0/16

# pod网络和service网络都要求为16位的地址块，且不能与环境中其他网络地址段冲突

# hostnamectl set-hostname k8s-master1.cof-lee.com #设置主机名称，所有结点分别设置

# cat >> /etc/hosts <<EOF

10.99.1.51 k8s-master1.cof-lee.com k8s-master1

10.99.1.52 k8s-master2.cof-lee.com k8s-master2

10.99.1.53 k8s-master3.cof-lee.com k8s-master3

10.99.1.61 k8s-node01.cof-lee.com k8s-node01

10.99.1.62 k8s-node02.cof-lee.com k8s-node02

EOF

★k8s初始化时要求系统里有/etc/resolv.conf文件及系统对外通信网口上配置有默认路由；根据实际情况添加

# cat >> /etc/resolv.conf <<EOF

nameserver 10.99.1.1

EOF

# nmcli con mod ens33 ipv4.gateway 10.99.1.1

# nmcli con up ens33

**③关闭seLinux**

若不会配置selinux，则可关闭SELinux，若对安全性要求较高，则需自行配置

# sed -i '/^SELINUX/s/enforcing/disabled/' /etc/selinux/config

# setenforce 0 #关闭selinux

**④ulimit设置**

# cat >> vi /etc/security/limits.conf <<EOF

\* soft nofile 1000000

\* hard nofile 1000000

\* soft core unlimited

\* hard core unlimited

\* soft memlock unlimited

\* soft memlock unlimited

\* soft nproc 40960

\* hard nproc 40960

EOF

#要重启系统才生效

**⑤关闭swap**

# cat >> /etc/sysctl.conf <<EOF

vm.swappiness = 0

EOF

# swapon -s #查看swap挂载情况，如果有swap挂载，则需要去/etc/fstab里取消挂载

# vi /etc/fstab

# swapoff -a #取消所有swap挂载

**⑥NTP服务需要开启**，使用集群内的ntp server，确保集群时间的一致性

# yum install chrony -y

# systemctl enable chronyd

# systemctl start chronyd

# cat > /etc/chrony.conf <<EOF

server **10.99.1.1** iburst prefer

driftfile /var/lib/chrony/drift

makestep 1.0 3

rtcsync

keyfile /etc/chrony.keys

leapsectz right/UTC

logdir /var/log/chrony

EOF

# timedatectl set-timezone Asia/Shanghai #设置时区

# systemctl restart chronyd

# chronyc sources -v

**⑦设置网络参数**

**#首先加载模块**

# cat > /etc/modules-load.d/br\_netfilter.conf <<EOF

overlay

br\_netfilter

EOF

# modprobe overlay

# modprobe br\_netfilter

# cat >> /etc/sysctl.conf <<EOF

net.bridge.bridge-nf-call-ip6tables = 1

net.bridge.bridge-nf-call-iptables = 1

net.bridge.bridge-nf-call-arptables = 1

net.ipv4.ip\_forward = 1

EOF

#前3行表示bridge设备在二层转发时也去调用iptables配置的三层规则（包含conntrack）

# sysctl -p #加载配置

**⑧防火墙放行端口**

TCP: 6443，2379，2380，10250~10252，30000~32767

UDP: 8285，8472

★最好是允许整个k8s的node网段以及pod网段入站

# firewall-cmd --add-rich-rule='rule family="ipv4" source address="10.99.1.0/24" accept'

# firewall-cmd --add-rich-rule='rule family="ipv4" source address="10.244.0.0/16" accept'

# firewall-cmd --runtime-to-permanent

# firewall-cmd --list-all

★如果有硬件交换机做ACL或基于云的安全组做访问控制，则可关闭服务器上的防火墙软件

**⑨加载ipvs模块**

# cat > /etc/modules-load.d/k8s-ipvs.conf <<EOF

ip\_vs

ip\_vs\_sh

ip\_vs\_rr

ip\_vs\_wrr

nf\_conntrack\_ipv4

EOF

# modprobe ip\_vs

# modprobe ip\_vs\_sh

# modprobe ip\_vs\_rr

# modprobe ip\_vs\_wrr

# modprobe nf\_conntrack\_ipv4

#一般默认只用ip\_vs\_rr

# lsmod | grep -e ip\_vs -e nf\_conntrack #检查是否已加载ip\_vs模块

**★最后重启操作系统**

# reboot

**★第1章、部署k8s版本<=1.23**

k8s在1.23及之前版本默认是调用docker作为底层的容器运行时，从1.24版本开始移除了dockerShim组件，不再支持docker，从而默认使用containerd作为底层的容器运行时。

**k8s 1.23及之前版本：**

kubelet→dockerShim→dockerd→containerd→runC

默认调用的cri-socket： unix:///var/run/dockershim.sock

要求docker<=20.10

**k8s 1.24及之后版本：**

kubelet→cri-containerd→containerd→runC

后来cri-containerd重构进containerd中（CRI Plugin），合为一个containerd进程

默认调用的cri-socket： unix:///var/run/containerd/containerd.sock

**本小节讲解k8s v1.23及之前版本的安装；首先按照第0章的步骤完成准备工作。**

**①安装并配置docker-ce**

# yum install docker-ce docker-ce-cli containerd.io -y #k8s v1.23支持的docker最新版本为20.10.x

# systemctl enable docker

# systemctl start docker

# docker info

★配置docker服务使用systemd去管理（以及信任本地镜像仓库）

# vi /etc/docker/daemon.json

{

"data-root": "/docker\_data",

"registry-mirrors": [ "https://cof-lee.com:5443" ],

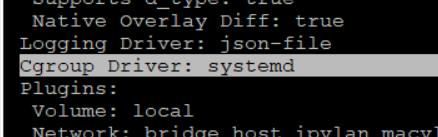
"insecure-registries": [ "cof-lee.com:5443" ],

**"exec-opts": [ "native.cgroupdriver=systemd" ]**

}

# mkdir -p /etc/systemd/system/docker.service.d

# docker info



**★docker会修改防火墙规则，导致pod网络不通**

# vi /usr/lib/systemd/system/docker.service

#在[Service]下的ExecStart=/usr/bin/dockerd -H fd:// 这行下面再添加一行：

ExecStartPost=/usr/sbin/iptables -P FORWARD ACCEPT

# systemctl daemon-reload

# systemctl restart docker

★默认还加了DOCKER-USER这个forward链，默认全部return，导致不通，也得放开，具体得看下iptables规则），以下操作目的为 在系统启动后等待60秒待k8s把iptables规则设置完毕再在以下几个chain里放通所有流量，如果对防火墙有自定义规则或对安全性要求较高场景无需配置以下这段，防火墙相关操作自行按需处理！

# cat >> /etc/rc.d/rc.local <<EOF

sleep 60

/usr/sbin/iptables -I DOCKER 1 -s 0.0.0.0/0 -j ACCEPT

/usr/sbin/iptables -I DOCKER-USER 1 -s 0.0.0.0/0 -j ACCEPT

/usr/sbin/iptables -I DOCKER-ISOLATION-STAGE-1 1 -s 0.0.0.0/0 -j ACCEPT

/usr/sbin/iptables -I DOCKER-ISOLATION-STAGE-2 1 -s 0.0.0.0/0 -j ACCEPT

EOF

**★最好的做法是让docker不生成默认的docker0网桥，不修改iptables规则**

# cat > /etc/docker/daemon.json <<EOF

{

"bridge": "none",

"iptables": false,

"experimental": true,

"data-root": "/docker\_data",

"registry-mirrors": [ "https://cof-lee.com:5443" ],

"insecure-registries": [ "cof-lee.com:5443" ],

**"exec-opts": [ "native.cgroupdriver=systemd" ]**

}

EOF

# systemctl daemon-reload

# systemctl restart docker

**②安装k8s二进制组件**

#使用aliyun的源

# cat >> /etc/yum.repos.d/k8s-ali.repo <<EOF

[k8s-ali]

name=k8s-ali

baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86\_64/

enabled=1

gpgcheck=0

EOF

#或者使用google的源：

# cat >> /etc/yum.repos.d/k8s-google.repo <<EOF

[k8s-google]

name=k8s-google

baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86\_64

enabled=1

gpgcheck=0

EOF

# yum install kubelet kubeadm kubectl kubernetes-cni -y #安装k8s二进制组件（<=1.23版本）

# systemctl enable kubelet

# systemctl start kubelet

**③k8s集群初始化**

# kubeadm version #先查看k8s版本



# GitVersion:"v1.19.4"

# kubeadm config images list #查看k8s其他组件的docker镜像名，默认用k8s.gcr.io/的镜像源地址

k8s.gcr.io/kube-apiserver:v1.19.4

k8s.gcr.io/kube-controller-manager:v1.19.4

k8s.gcr.io/kube-scheduler:v1.19.4

k8s.gcr.io/kube-proxy:v1.19.4

k8s.gcr.io/pause:3.2

k8s.gcr.io/etcd:3.4.13-0

k8s.gcr.io/coredns:1.7.0

#可以先下载以上7个镜像，传到每台k8s服务器上，再docker load导入；或者使用内部registry仓库（内部registry镜像仓库里要有以上7个镜像）

★**直接使用命令行方式初始化集群**

（**以下**是**非HA模式**的master初始化，如果要部署高可用集群，则参考第4章）

|  |  |
| --- | --- |
| kubeadm init --kubernetes-version=v1.19.4 \ |  |
| --apiserver-advertise-address=10.99.1.51 \ | # api server地址 |
| --pod-network-cidr=10.244.0.0/16 \ | # pod容器网段 |
| --service-cidr=10.7.0.0/16 \ | # service网段，即cluster ip网段 |
| --ignore-preflight-errors=Swap \ | #忽略swap未关闭而导致的检查错误 |
| --image-repository="cof-lee.com:5443/k8s" | #指定为集群内部的docker镜像源 |

如果指定使用集群内部的docker镜像仓库，要提前在docker的daemon.json文件里配置信任此镜像仓库地址！

# cat >> /etc/hosts <<EOF

10.99.1.248 cof-lee.com

EOF

**★使用配置文件方式去初始化集群（和上面命令行方式二选一）**

# kubeadm config print init-defaults > /etc/kubeadm-init.yaml #输出初始化配置文件并编辑

# vi /etc/kubeadm-init.yaml

apiVersion: kubeadm.k8s.io/v1beta2

bootstrapTokens:

- groups:

- system:bootstrappers:kubeadm:default-node-token

token: abcdef.0123456789abcdef

ttl: 24h0m0s

usages:

- signing

- authentication

kind: InitConfiguration

localAPIEndpoint:

advertiseAddress: 10.99.1.51

bindPort: 6443

nodeRegistration:

criSocket: **/var/run/dockershim.sock**

name: k8s-master1

taints:

- effect: NoSchedule

key: node-role.kubernetes.io/master

---

apiServer:

timeoutForControlPlane: 4m0s

apiVersion: kubeadm.k8s.io/v1beta2

certificatesDir: /etc/kubernetes/pki

clusterName: kubernetes

controllerManager: {}

dns:

type: CoreDNS

etcd:

local:

dataDir: /var/lib/etcd

imageRepository: cof-lee.com:5443/k8s

kind: ClusterConfiguration

kubernetesVersion: v1.19.4

networking:

dnsDomain: cluster.local

serviceSubnet: 10.7.0.0/16

podSubnet: 10.244.0.0/16

scheduler: {}

---

apiVersion: kubelet.config.k8s.io/v1beta1

kind: KubeletConfiguration

cgroupDriver: systemd

failSwapOn: False

#保存，（修改蓝色字体的参数值及新增最后5行）

# kubeadm config images list --config /etc/kubeadm-init.yaml #查看需要的镜像

cof-lee.com:5443/k8s/kube-apiserver:v1.19.4

cof-lee.com:5443/k8s/kube-controller-manager:v1.19.4

cof-lee.com:5443/k8s/kube-scheduler:v1.19.4

cof-lee.com:5443/k8s/kube-proxy:v1.19.4

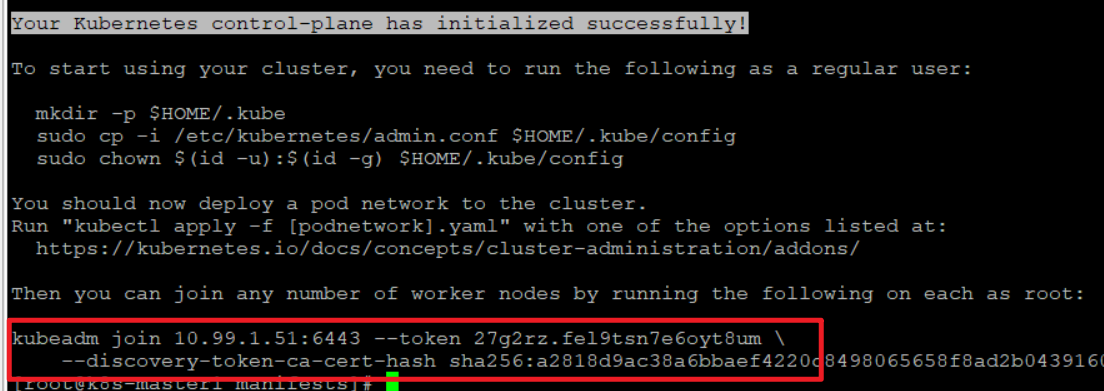
cof-lee.com:5443/k8s/pause:3.2

cof-lee.com:5443/k8s/etcd:3.4.13-0

#可见镜像名已由默认的k8s.gcr.io/换成了配置文件里指定的docker镜像源

# kubeadm config images pull --image-repository="cof-lee.com:5443/k8s" #提前下载需要的镜像

# kubeadm init --config /etc/kubeadm-init.yaml #初始化集群



当出现Your Kubernetes control-plane has initialized successfully!这行时说明初始化k8s成功了

记住最后2行命令，是用来让node结点加入集群的命令（含token）

**★第2章、部署k8s版本>=1.24**

本小节讲解k8s v1.24及之后版本的安装；首先按照第0章的步骤完成准备工作。

**①安装并配置containerd**

**★安装containerd.io软件**

可以使用阿里的源或docker官方的软件源

# cd /etc/yum.repos.d/

# wget https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo #aliyun的源

或者： https://download.docker.com/linux/centos/docker-ce.repo #官方的源



然后在/etc/yum.repos.d/目录下多了一个名为docker-ce.repo的文件

如果下载不了，可以在此目录下自己创建一个docker-ce.repo文件，内容如下：

[docker-ce-stable]

name=Docker CE Stable - $basearch

baseurl=https://mirrors.aliyun.com/docker-ce/linux/centos/7/$basearch/stable

enabled=1

gpgcheck=1

gpgkey=https://mirrors.aliyun.com/docker-ce/linux/centos/gpg

# yum install containerd.io -y

# containerd --version #查看版本

containerd containerd.io 1.6.24 61f9fd88f79f081d64d6fa3bb1a0dc71ec870523

**★配置containerd服务**

# mkdir -p /etc/containerd

# containerd config default > /etc/containerd/config.toml #生成containerd的配置文件

# vi /etc/containerd/config.toml #修改以下2行配置

SystemdCgroup = true

sandbox\_image = "cof-lee.com:5443/k8s/pause:3.9" #和k8s需要的pause镜像版本保持一致

#如果要启用CRI-Plugin，注释掉其中的 disabled\_plugins = ["cri"]

#再重启containerd即可有 unix:///run/containerd/containerd.sock 接口

**#信任私有镜像仓库ssl证书，添加或修改以下几行配置**

[plugins."io.containerd.grpc.v1.cri".registry.configs]

[plugins."io.containerd.grpc.v1.cri".registry.configs."cof-lee.com:5443".tls]

insecure\_skip\_verify = true #跳过安全认证

#如果下载镜像需要身份验证则配置下面3行，不需要身份验证则不用配置

[plugins."io.containerd.grpc.v1.cri".registry.configs."cof-lee.com:5443".auth]

username = "*userxx*"

password = "*passwdxx*"

/etc/containerd/config.toml 这个配置文件是给crictl和kubelet使用的，ctr不使用这个配置文件，因为ctr不使用CRI，所以它不读取 plugins."io.containerd.grpc.v1.cri" 配置。

# systemctl enable containerd

# systemctl restart containerd

**★如果不想配置信任私有镜像仓库，也可将服务器证书添加到操作系统的ca证书库里**

# cat ca.com.crt >> /etc/pki/tls/certs/ca-bundle.crt

#将ca证书添加到centos系统证书信任列表中，链接到：

/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem

**②安装k8s二进制组件**

#使用aliyun的源（如果用的是RHEL8系列的系统，也是用的el7的仓库源，因为k8s组件是用go语言写的，直接二进制文件就可用，不需要特殊的库依赖，于是和系统弱相关，8或9系列的系统也直接可用这些rpm软件）

# cat >> /etc/yum.repos.d/k8s-ali.repo <<EOF

[k8s-ali]

name=k8s-ali

baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86\_64/

enabled=1

gpgcheck=0

EOF

#或者使用google的源：

# cat >> /etc/yum.repos.d/k8s-google.repo <<EOF

[k8s-google]

name=k8s-google

baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86\_64

enabled=1

gpgcheck=0

EOF

# yum install kubelet kubeadm kubectl kubernetes-cni -y #安装k8s二进制组件（>=1.24版本）

# systemctl enable kubelet

# systemctl start kubelet

**③k8s集群初始化**

# kubeadm version #先查看k8s版本

kubeadm version: &version.Info{Major:"1", Minor:"28", GitVersion:"v1.28.2", GitCommit:"89a4ea3e1e4ddd7f7572286090359983e0387b2f", GitTreeState:"clean", BuildDate:"2023-09-13T09:34:32Z", GoVersion:"go1.20.8", Compiler:"gc", Platform:"linux/amd64"}

# kubeadm config images list #查看k8s其他组件的docker镜像名，默认用k8s.gcr.io/的镜像源地址

registry.k8s.io/kube-apiserver:v1.28.2

registry.k8s.io/kube-controller-manager:v1.28.2

registry.k8s.io/kube-scheduler:v1.28.2

registry.k8s.io/kube-proxy:v1.28.2

registry.k8s.io/pause:3.9

registry.k8s.io/etcd:3.5.9-0

registry.k8s.io/coredns/coredns:v1.10.1

#可以先下载以上7个镜像，传到每台k8s服务器上，再ctr -n k8s.io images import导入；或者使用内部registry仓库（内部registry镜像仓库里要有以上7个镜像）

★**直接使用命令行方式初始化集群**

（**以下**是**非HA模式**的master初始化，如果要部署高可用集群，则参考第4章）

|  |  |
| --- | --- |
| kubeadm init --kubernetes-version=v1.28.2 \ |  |
| --apiserver-advertise-address=10.99.1.51 \ | # api server地址 |
| --pod-network-cidr=10.244.0.0/16 \ | # pod容器网段 |
| --service-cidr=10.7.0.0/16 \ | # service网段，即cluster ip网段 |
| --ignore-preflight-errors=Swap \ | #忽略swap未关闭而导致的检查错误 |
| --image-repository="cof-lee.com:5443/k8s" | #指定为集群内部的docker镜像源 |

如果指定使用集群内部的docker镜像仓库，要提前在containerd的配置文件里配置信任此镜像仓库地址！

# cat >> /etc/hosts <<EOF

10.99.1.248 cof-lee.com

EOF

**★使用配置文件方式去初始化集群（和上面命令行方式二选一）**

# kubeadm config print init-defaults > /etc/kubeadm-init.yaml #输出初始化配置文件并编辑

# vi /etc/kubeadm-init.yaml

apiVersion: kubeadm.k8s.io/v1beta3

bootstrapTokens:

- groups:

- system:bootstrappers:kubeadm:default-node-token

token: abcdef.0123456789abcdef

ttl: 24h0m0s

usages:

- signing

- authentication

kind: InitConfiguration

localAPIEndpoint:

advertiseAddress: 10.99.1.51

bindPort: 6443

nodeRegistration:

criSocket: **unix:///var/run/containerd/containerd.sock**

imagePullPolicy: IfNotPresent

name: k8s-master1

taints: null

---

apiServer:

timeoutForControlPlane: 4m0s

apiVersion: kubeadm.k8s.io/v1beta3

certificatesDir: /etc/kubernetes/pki

clusterName: kubernetes

controllerManager: {}

dns: {}

etcd:

local:

dataDir: /var/lib/etcd

imageRepository: cof-lee.com:5443/k8s

kind: ClusterConfiguration

kubernetesVersion: 1.28.2

networking:

dnsDomain: cluster.local

serviceSubnet: 10.7.0.0/16

podSubnet: 10.244.0.0/16

scheduler: {}

---

apiVersion: kubelet.config.k8s.io/v1beta1

kind: KubeletConfiguration

cgroupDriver: systemd

failSwapOn: False

#保存，（修改蓝色字体的参数值及新增最后5行）

# kubeadm config images list --config /etc/kubeadm-init.yaml #查看需要的镜像

cof-lee.com:5443/k8s/kube-apiserver:v1.28.2

cof-lee.com:5443/k8s/kube-controller-manager:v1.28.2

cof-lee.com:5443/k8s/kube-scheduler:v1.28.2

cof-lee.com:5443/k8s/kube-proxy:v1.28.2

cof-lee.com:5443/k8s/pause:3.9

cof-lee.com:5443/k8s/etcd:3.5.9-0

cof-lee.com:5443/k8s/coredns:v1.10.1

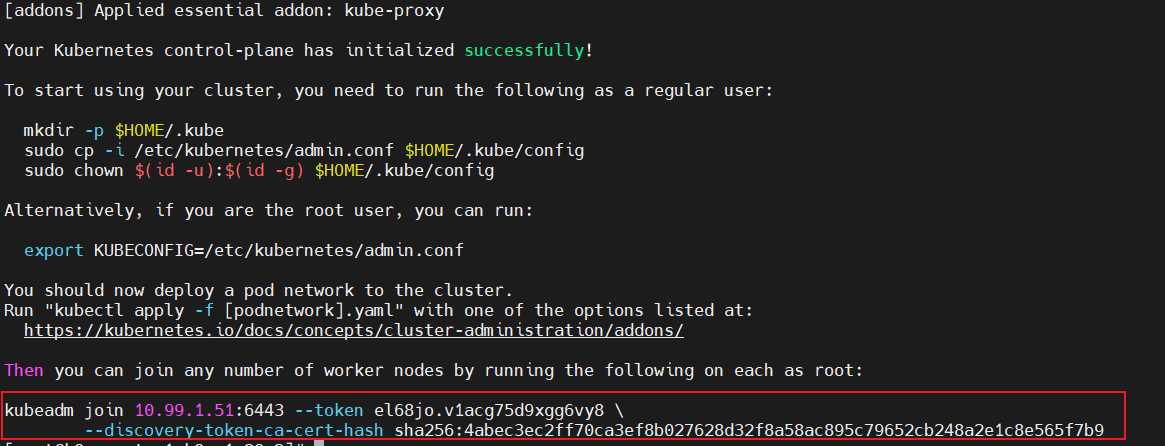
#可见镜像名已由默认的k8s.gcr.io/换成了配置文件里指定的docker镜像源

# kubeadm config images pull --image-repository="cof-lee.com:5443/k8s" #提前下载需要的镜像

# ctr -n k8s.io image ls #查看下载的镜像，k8s默认使用k8s.io的命名空间



# kubeadm init --config /etc/kubeadm-init.yaml #初始化集群



当出现Your Kubernetes control-plane has initialized successfully!这行时说明初始化k8s成功了

记住最后2行命令，是用来让node结点加入集群的命令（含token）

**★附、crictl命令**

v1.24及之后的k8s默认带有crictl命令，可用于查看containerd的镜像及容器等信息

安装了containerd后，containerd的命令行工具为ctr，而k8s v1.24之后版本又新增额外的辅助工具crictl

crictl是遵循CRI接口规范的一个命令行工具，通常用它来检查和管理kubelet节点上的容器运行时和镜像。（支持containerd/docker/crio）

**★安装crictl工具**

默认随kubeadm一起安装了，在cri-tools-xxxx.x86\_64.rpm这个包里，如果想单独安装，则直接装这个rpm包或者到github上下载此软件

# wget https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.28.0/crictl-v1.28.0-linux-amd64.tar.gz

# tar -xf crictl-v1.28.0-linux-amd64.tar.gz -C /usr/bin/

# crictl -v #查看crictl版本

crictl version v1.28.0

# crictl images #查看底层镜像，同ctr -n k8s.io images ls

WARN[0000] image connect using default endpoints: [unix:///var/run/dockershim.sock unix:///run/containerd/containerd.sock unix:///run/crio/crio.sock]. As the default settings are now deprecated, you should set the endpoint instead. #crictl默认会依次读取以上3个接口

**★设置crictl连接containerd的cri接口**

# cat > /etc/crictl.yaml <<EOF

runtime-endpoint: unix:///run/containerd/containerd.sock

image-endpoint: unix:///run/containerd/containerd.sock

timeout: 10

debug: false

EOF

**★crictl常用操作**

crictl只支持pull拉取镜像及rmi删除镜像操作，不支持push及tag打标签

# crictl images #查看镜像，默认查询k8s.io命名空间，且不可指定命名空间

# crictl ps #查看容器，默认固定为k8s.io的命名空间

# crictl logs xxxx #查看容器日志，支持带-f参数

# crictl inspect xxx #查看镜像或容器信息

**★附、安装cri-dockerd**

Kubernetes自v1.24移除了对docker-shim的支持，而Docker Engine默认又不支持CRI规范，所以k8s-v1.24开始不直接支持docker。为此，Mirantis和Docker联合创建了cri-dockerd项目，用于为Docker Engine提供一个能够支持CRI规范的垫片，从而能够让Kubernetes基于CRI控制Docker,但是这种架构缺点也很明显，调用链更长，效率更低。

即 unix:///var/run/cri-dockerd.sock 相当于v1.23及之前版本的的 unix:///var/run/dockershim.sock

不过，不建议使用了，调用层次太多，本小节仅供学习测试

**首先安装docker**

# yum install docker-ce

# systemctl enable docker

# systemctl start docker

# cat > /etc/docker/daemon.json <<EOF

{

"bridge": "none",

"iptables": false,

"experimental": true,

"insecure-registries": [ "cof-lee.com" ],

"exec-opts": [ "native.cgroupdriver=systemd" ]

}

EOF

# systemctl daemon-reload

# systemctl restart docker

**再安装cri-dockerd**

下载地址： https://github.com/Mirantis/cri-dockerd

# wget https://github.com/Mirantis/cri-dockerd/releases/download/v0.3.8/cri-dockerd-0.3.8-3.el8.x86\_64.rpm

# yum install cri-dockerd-0.3.8-3.el8.x86\_64.rpm

# vi /usr/lib/systemd/system/cri-docker.service #修改cri-dockerd服务配置

#修改ExecStart行如下

ExecStart=/usr/bin/cri-dockerd --container-runtime-endpoint fd:// --pod-infra-container-image=cof-lee.com/k8s/pause:3.9

#重启cri-docker

# systemctl daemon-reload

# systemctl enable cri-docker.service

# systemctl restart cri-docker.service

最后修改k8s配置，使用cri-dockerd.sock

# vi /var/lib/kubelet/kubeadm-flags.env #修改--container-runtime-endpoint=的值为新的cri socket

#KUBELET\_KUBEADM\_ARGS="--container-runtime-endpoint=unix:///var/run/containerd/containerd.sock --pod-infra-container-image=cof-lee.com/k8s/pause:3.9"

KUBELET\_KUBEADM\_ARGS="--container-runtime-endpoint=unix:///var/run/cri-dockerd.sock --pod-infra-container-image=cof-lee.com/k8s/pause:3.9"

# systemctl daemon-reload

# systemctl restart kubelet

所有节点都要安装docker及cri-docker

已加入集群的，所有节点都要修改 /var/lib/kubelet/kubeadm-flags.env 内容

未加入集群的，执行加入命令时添加--cri-socket选项：

# kubeadm join x.x.x.x:6443 --token xxxx \

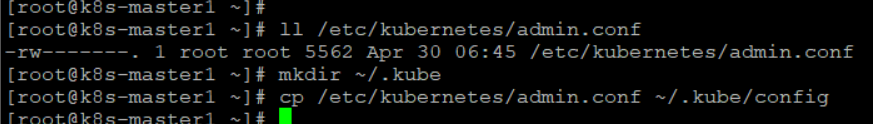
--discovery-token-ca-cert-hash xxxxxxxx --cri-socket unix:///var/run/cri-dockerd.sock

**★第3章、安装后续步骤**

**④k8s认证文件.kube/config**

在刚刚安装好的master结点上有操作整个k8s集群的认证文件/etc/kubernetes/admin.conf

把它复制到需要使用kubectl命令的节点上的$HOME/.kube/目录下并命令为config即可

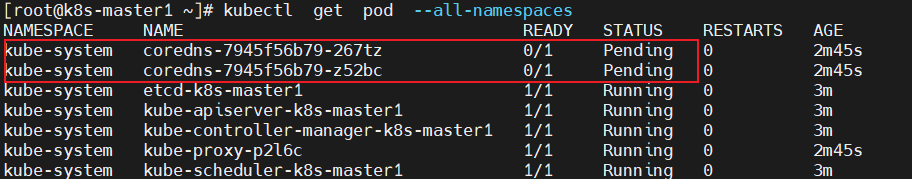


# mkdir ~/.kube

# cp /etc/kubernetes/admin.conf ~/.kube/config

# chown $(id -u):$(id -g) ~/.kube/config

# kubectl get pod --all-namespaces #查看容器运行状态

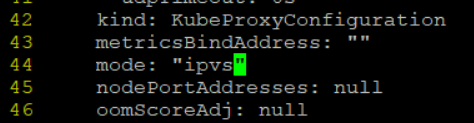


如果有2个coredns的pod未启动就绪，因为pod网络组件还未部署，部署后再查看；如果部署网络组件之后还未启动，则可能是网络不通，防火墙要放通pod容器网段入站。

**⑤service使用ipvs模式**

# kubectl edit cm kube-proxy -n kube-system #编辑configMap

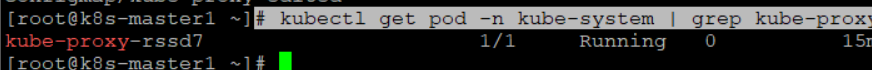
找到kind: KubeProxyConfiguration这行往下第2行的mode: ""，在""里写入ipvs，保存退出



（要求在做准备工作时已加载ipvs内核模块）

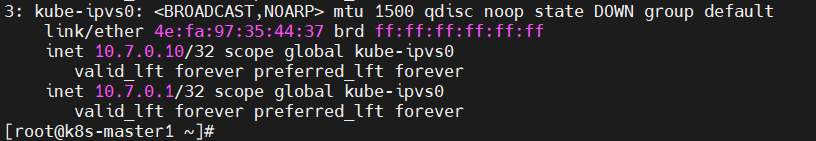
再删除kube-proxy的所有pod（即重启这些pod）

# kubectl get pod -n kube-system | grep kube-proxy #查看名为kube-proxy\*的pod



# kubectl -n kube-system delete pod kube-proxy-rssd7 #删除pod

开启了ipvs模式的k8s结点上执行ip addr命令会看到名为kube-ipvs0的接口名



**⑥部署flannel网络插件**

flannel只是pod容器网络的一种实现方式，还可使用calico，canal等其他网络方案，一套k8s集群只可使用一种网络插件

flannel使得跨主机的Pod之间能通信，底层实现是使用了vxlan

flannel源码地址： https://github.com/flannel-io/flannel

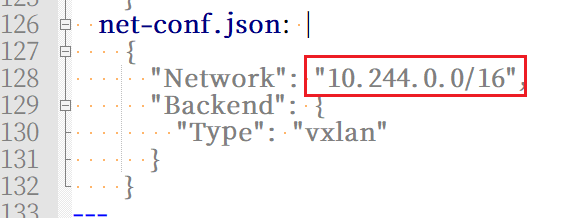
#先下载flannel插件的k8s部署yml文件

# wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

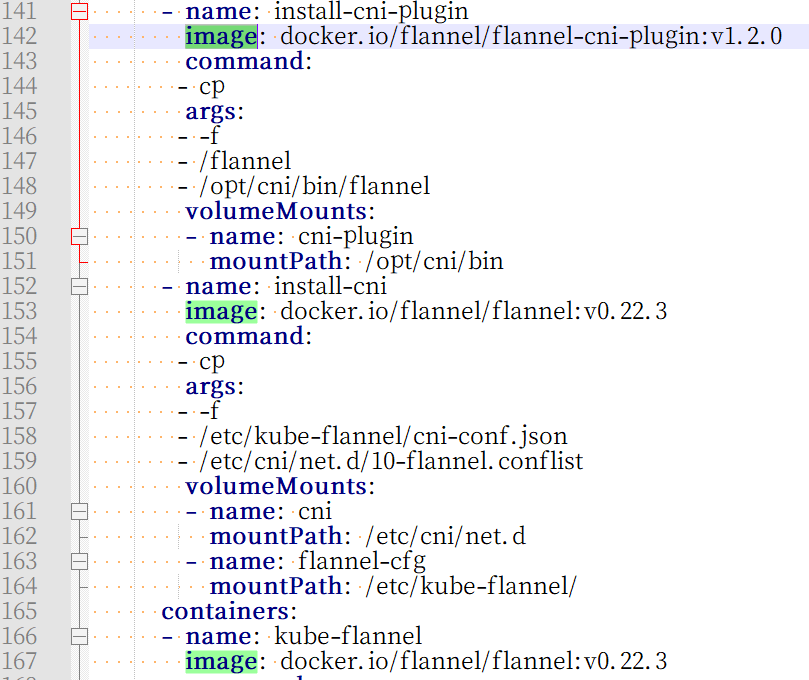
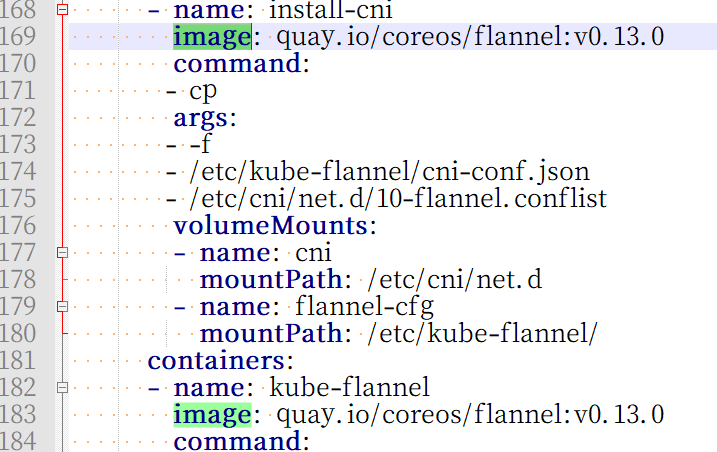
或者 wget https://limaofu.github.io/scripts/kube-flannel-v0.13.0.yml

# vi kube-flannel.yml

#将里面的net-conf.json下面的Network网段改为规划的pod网段

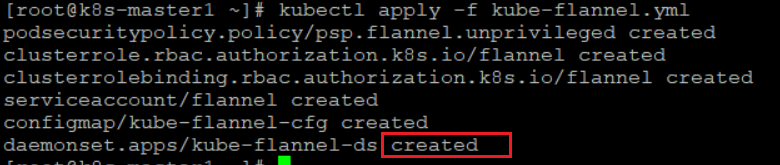


#默认使用的docker镜像是quay.io/coreos/的仓库，可改为自己集群的docker仓库



保存，退出

# kubectl apply -f kube-flannel.yml #应用flannel配置



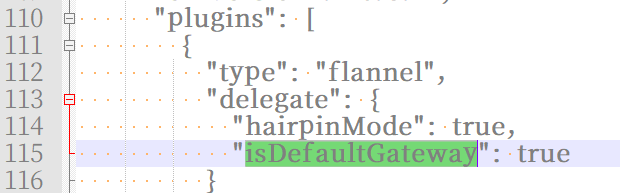
如果pod启动失败了，查看日志

# kubectl logs kube-flannel-ds-24tfm -n kube-system

I0430 11:16:34.505952 1 main.go:518] Determining IP address of default interface

E0430 11:16:34.506062 1 main.go:204] Failed to find any valid interface to use: failed to get default interface: Unable to find default route

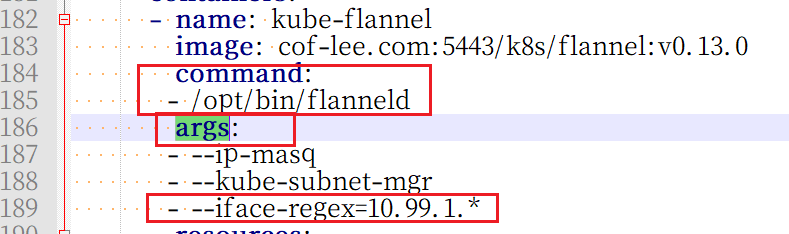
原因是没有找到有效的网卡，因为默认没有在kube-flannel.yml指定vxlan使用的底层网络接口，所以它根据ip route show去查找default via这行的网卡（有默认路由的网络接口），但我们测试环境的服务器没有配置网关，所以它找不到默认路由，导致flannel启动失败



解决方法是给服务加个网关（默认路由）或者在kube-flannel.yml里指定vxlan绑定的网卡设备（不建议直接绑定网络接口名，因为不同的服务器接口名称可能不一样，可匹配ip网段所在的网络接口）

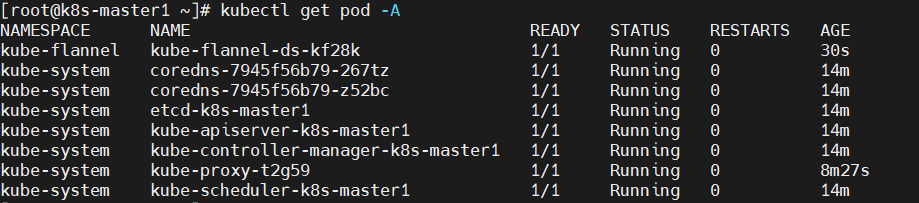
# kubectl delete -f kube-flannel.yml #先删除旧的部署

#再在kube-flannel.yml部署配置文件里的- /opt/bin/flanneld下面的args:这行下面再添加一行配置 **- --iface-regex=10.99.1.\*** 匹配要通信的ip网段（为node结点的通信ip网段，即pod容器流量使用此ip网络对应的网口进行vxlan封装）



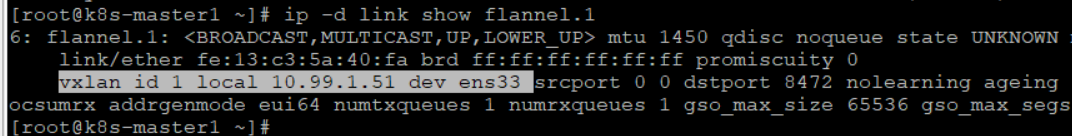
#最后重新部署flannel的deployment文件

# kubectl apply -f kube-flannel.yml



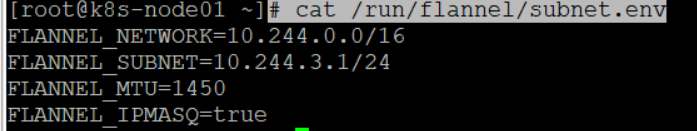
#可见flannel的pod启动了（名称为kube-flannel-xxxx）

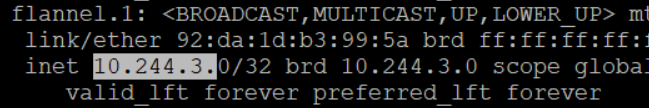
# ip -d link show flannel.1 #查看vxlan信息，默认使用8472/udp端口



最后，k8s集群中的每台成员结点都会向podSubnet中注册一个24位掩码的子网，把16位的pod网段划分为24位的子网，每台成员结点分配一个24位的子网，

# cat /run/flannel/subnet.env #查看当前结点注册的pod子网信息





**⑦部署calico网络插件（与flannel二选一）**

官方指导： https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart

# mkdir ~/calico

# cd ~/calico

# wget https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/tigera-operator.yaml

# wget https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/custom-resources.yaml

#创建控制器 Install the Tigera Calico operator and custom resource definitions

# kubectl create -f tigera-operator.yaml #如果需要更改镜像，可编辑此文件

# more tigera-operator.yaml | grep -i image:

image: quay.io/tigera/operator:v1.30.4

**#修改pod网段**

# sed -i 's/192.168.0.0\/16/10.244.0.0\/16/g' custom-resources.yaml

#创建资源 Install Calico by creating the necessary custom resource

# kubectl create -f custom-resources.yaml #如果需要更改镜像，只能部署后，只能更改相应的daemonset及deployment

#查看calico节点状态，状态全部为Running则为启动成功

# kubectl get pods -n tigera-operator

NAME READY STATUS RESTARTS AGE

tigera-operator-59b69c49dc-t6crt 1/1 Running 1 (10m ago) 51m

# kubectl get pods -n calico-system

NAME READY STATUS RESTARTS AGE

calico-kube-controllers-56cf5967cb-x8nsv 1/1 Running 1 (10m ago) 11m

calico-node-qn7cp 1/1 Running 1 (10m ago) 11m

calico-typha-84cb54bfd-hh57t 1/1 Running 1 (10m ago) 11m

csi-node-driver-fn7zd 2/2 Running 2 (10m ago) 11m

# Remove the taints on the control plane so that you can schedule pods on it.

# kubectl describe node master1.cof-lee.com | grep -i Taints #查看master结点的污点

# kubectl taint nodes --all node-role.kubernetes.io/control-plane-

# kubectl taint nodes --all node-role.kubernetes.io/master-

**要用到的镜像：**

quay.io/tigera/operator:v1.30.4

docker.io/calico/typha:v3.26.1

docker.io/calico/pod2daemon-flexvol:v3.26.1

docker.io/calico/node-driver-registrar:v3.26.1

docker.io/calico/csi:v3.26.1

docker.io/calico/kube-controllers:v3.26.1

#以下2个要求上传到harbor，必须为在线获取

docker.io/calico/cni:v3.26.1

docker.io/calico/node:v3.26.1

# kubectl get daemonset -A

NAMESPACE NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

calico-system calico-node 2 2 2 2 2 kubernetes.io/os=linux 4h24m

calico-system csi-node-driver 2 2 2 2 2 kubernetes.io/os=linux 4h24m

# kubectl get deploy -A

NAMESPACE NAME READY UP-TO-DATE AVAILABLE AGE

calico-system calico-kube-controllers 1/1 1 1 4h25m

calico-system calico-typha 1/1 1 1 4h25m

tigera-operator tigera-operator 1/1 1 1 5h5m

**★指定底层网络接口（node ip）**

k8s calico组件daemonset设置：

- name: IP\_AUTODETECTION\_METHOD

valuse: "interface=ens3"

**★安装calicoctl命令工具**

官方指导文档： https://docs.tigera.io/calico/latest/operations/calicoctl/install

下载地址： https://github.com/projectcalico/calicoctl

# curl -L https://github.com/projectcalico/calicoctl/releases/download/v3.20.6/calicoctl-linux-amd64 -o calicoctl

# chmod +x ./calicoctl

**⑧node结点安装k8s组件并加入集群**

node结点同master结点一样也要做准备工作，都是第0章的内容

准备工作做好后，安装k8s二进制组件

# yum install kubelet kubeadm kubectl kubernetes-cni -y #安装k8s二进制组件

# systemctl enable kubelet

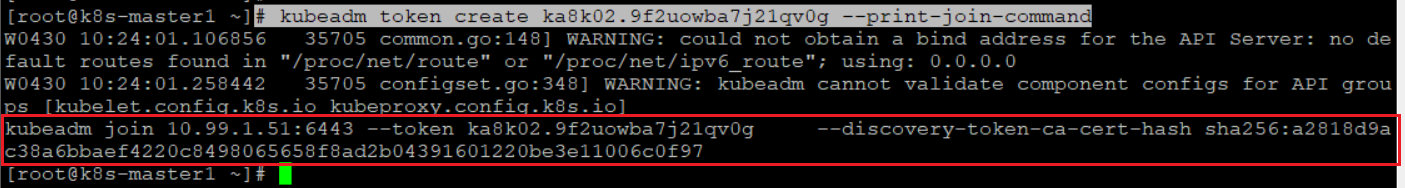
# systemctl start kubelet

再加入K8S集群（使用 安装第一个k8s的master结点时生成的最后2行命令kubeadm join xxx）不过这命令的token默认是2小时有效期，超时后可重新生成token再去node结点上执行

master1结点# kubeadm token generate #重新生成token，下面这一串字符

ka8k02.9f2uowba7j21qv0g

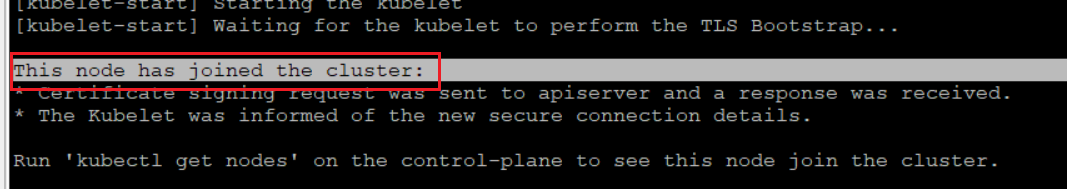
master1结点# kubeadm token create ka8k02.9f2uowba7j21qv0g --print-join-command



再复制（kubeadm join xxxxx）命令到node结点上执行：

node结点执行# kubeadm join 10.99.1.51:6443 --token ka8k02.9f2uowba7j21qv0g \

--discovery-token-ca-cert-hash sha256:a2818d9ac38a6bbaef4220c8498065658f8ad2b04391601220be3e11006c0f97



出现： This node has joined the cluster: 这句提示说明加入成功，再去master结点上查看

[root@k8s-master1 ~]# kubectl get node #查看集群的所有结点

NAME STATUS ROLES AGE VERSION

k8s-master1 Ready master 3h49m v1.19.4

k8s-node01 Ready <none> 16s v1.19.4 #刚刚加入集群的node结点

**★将node结点移出k8s集群**

**master结点上**# kubectl taint node *结点名* keyrm:NoExecute #先打上污点，驱逐工作负载pod，等待几分钟，确认目标node结点上没有工作pod运行后，再删除

**master结点上**# kubectl delete nodes *结点名* #将目标结点从集群里删除

**node结点上**# kubeadm reset #node结点上重置kubeadm配置，请勿操作错了，“kubeadm reset”这条命令是**在要移除的node结点**上执行，不可在master上执行！！

**★k8s初始化报错（树莓派）**

CGROUPS\_MEMORY: missing

[ERROR SystemVerification]: missing required cgroups: memory

因为树莓派系统默认未启动cgroup\_memory，解决方法：

# vi /boot/firmware/cmdline.txt #如果有这个文件就编辑此文件

# vi /boot/cmdline.txt #如果有这个文件就编辑此文件

#在原有的**console=**ttyAMA0,115200 console=tty1 root=/dev/mmcblk0p3 rootfstype=ext4 elevator=deadline rootwait 这行后面添加

cgroup\_enable=memory cgroup\_memory=1

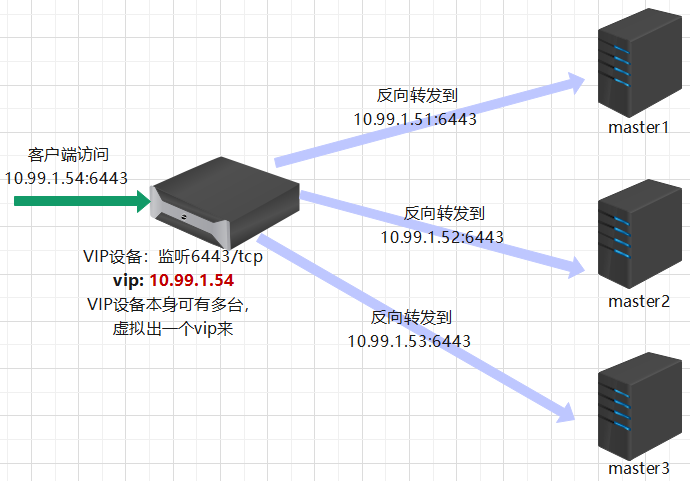
#是在行尾添加，不是另起一行

# reboot #重启后生效

**★第4章、K8S高可用集群搭建**

前面几章只装了一台master结点，如果它出故障了，则整个k8s集群都不可用了，需要部署至少3台master结点，准备工作一样都同第0章，只是还要规划一个vip（虚拟ip）在master的集群初始化这步需要指定control-plane-endpoint地址为规划的vip，前端的vip可以使用任何负载工具软件，反向代理到后端的每台master结点的6443/tcp端口即可。

**高可用集群拓扑图：**



**★先配置HA高可用的反向代理**

本例中vip为**10.99.1.54**（三台master ip为10.99.1.51~53）使用haproxy做反向代理

frontend k8s\_api\_tcp\_6443

bind \*:6443

mode tcp

default\_backend my\_k8s\_cluster\_6443

backend my\_k8s\_cluster\_6443

mode tcp

balance roundrobin

server s1 10.99.1.51:6443 check weight 1 inter 2000 rise 2 fall 2

server s2 10.99.1.52:6443 check weight 1 inter 2000 rise 2 fall 2

server s3 10.99.1.53:6443 check weight 1 inter 2000 rise 2 fall 2

#如果vip设备就在这3台master结点上，则Haproxy监听的端口不能为6443，否则与后端api监听的端口冲突了，可改为其他端口号

#本例使用单独的一台服务器来配置vip及反向代理，vip使用keepalived软件来生成，配置省略。

**★先安装master1**

★**使用命令行方式初始化集群**

|  |  |
| --- | --- |
| kubeadm init --kubernetes-version=v1.19.4 \ |  |
| --apiserver-advertise-address=10.99.1.51 \ | # api server地址，master1的node ip |
| --pod-network-cidr=10.244.0.0/16 \ | # pod容器网段 |
| --service-cidr=10.7.0.0/16 \ | # service网段，即cluster ip网段 |
| --image-repository="cof-lee.com:5443/k8s" \ | #指定镜像源为集群可访问的源地址 |
| --control-plane-endpoint=**10.99.1.54:6443** \ | #**指定控制面板vip与端口号** |
| --ignore-preflight-errors=Swap \ | #忽略swap未关闭而导致的检查错误 |
| --upload-certs | #将控制平面证书上传到kubeadm-certs Secret；并在后续添加节点时自动分发证书文件 |

**★使用配置文件方式去初始化集群（和上面命令行方式二选一）**

# kubeadm config print init-defaults > /etc/kubeadm-init.yaml #输出初始化配置文件并编辑

# vi /etc/kubeadm-init.yaml

apiVersion: kubeadm.k8s.io/v1beta2

bootstrapTokens:

- groups:

- system:bootstrappers:kubeadm:default-node-token

token: abcdef.0123456789abcdef

ttl: 24h0m0s

usages:

- signing

- authentication

kind: InitConfiguration

localAPIEndpoint:

advertiseAddress: 10.99.1.51

bindPort: 6443

nodeRegistration:

criSocket: /var/run/dockershim.sock

name: k8s-master1

taints:

- effect: NoSchedule

key: node-role.kubernetes.io/master

---

apiServer:

timeoutForControlPlane: 4m0s

apiVersion: kubeadm.k8s.io/v1beta2

certificatesDir: /etc/kubernetes/pki

clusterName: kubernetes

controlPlaneEndpoint: "10.99.1.54:6443"

controllerManager: {}

dns:

type: CoreDNS

etcd:

local:

dataDir: /var/lib/etcd

imageRepository: cof-lee.com:5443/k8s

kind: ClusterConfiguration

kubernetesVersion: v1.19.4

networking:

dnsDomain: cluster.local

serviceSubnet: 10.7.0.0/16

podSubnet: 10.244.0.0/16

scheduler: {}

---

apiVersion: kubelet.config.k8s.io/v1beta1

kind: KubeletConfiguration

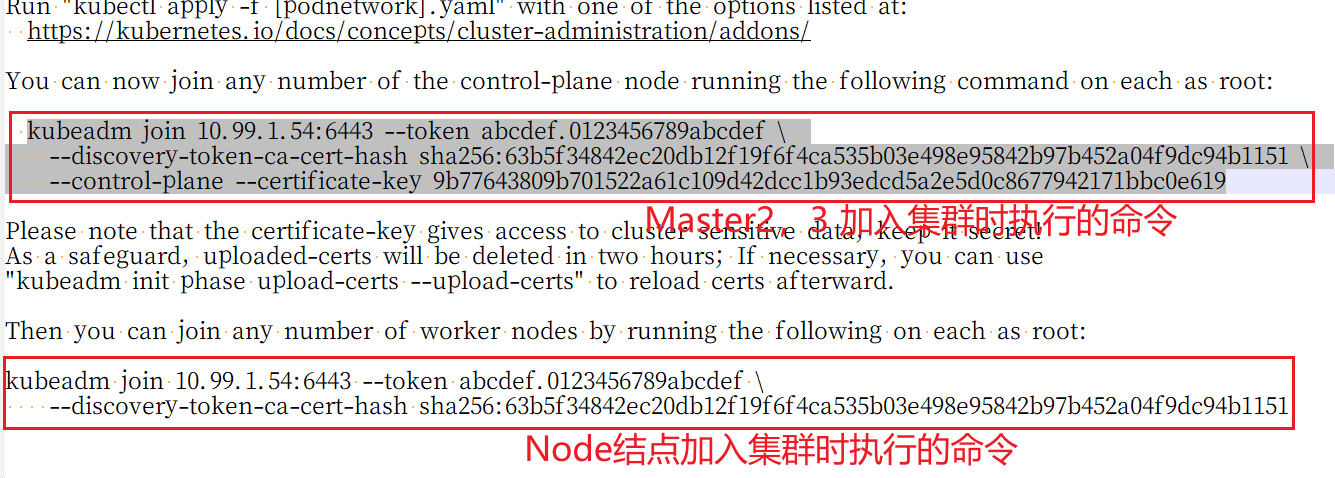
cgroupDriver: systemd

failSwapOn: False

#保存

# kubeadm config images list --config /etc/kubeadm-init.yaml #查看需要的镜像

# kubeadm init --config /etc/kubeadm-init.yaml --upload-certs #初始化集群



kubeadm join 10.99.1.54:6443 --token abcdef.0123456789abcdef \

--discovery-token-ca-cert-hash sha256:63b5f34842ec20db12f19f6f4ca535b03e498e95842b97b452a04f9dc94b1151 \

--control-plane --certificate-key 9b77643809b701522a61c109d42dcc1b93edcd5a2e5d0c8677942171bbc0e619

kubeadm join 10.99.1.54:6443 --token abcdef.0123456789abcdef \

--discovery-token-ca-cert-hash sha256:63b5f34842ec20db12f19f6f4ca535b03e498e95842b97b452a04f9dc94b1151

# mkdir ~/.kube

# cp /etc/kubernetes/admin.conf ~/.kube/config

**★再参考第3章的以下2步:**

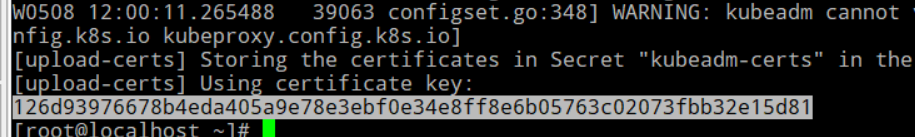
**⑤service使用ipvs模式**

**⑥部署flannel网络插件 或⑦部署calico网络插件**

**完成以上配置，再将master2, master3加入集群**

# 如果执行加入命令时提示certificate-key过期了，执行以下命令更新：

# kubeadm init phase upload-certs --upload-certs



将新生成的certificate key替换旧的，再去执行加入节点命令

# kubectl -n kube-system get cm kubeadm-config -oyaml #查看集群配置

**★etcd的配置更新**

第3个master加入集群后，其etcd的配置是有3台成员结点，而前面的master1上面只有一台的信息，master2上面有2台(master1,master2）成员信息，得修改master1及master2上的etcd配置

# vi /etc/kubernetes/manifests/etcd.yaml #先修改master1上的

#在 - --initial-cluster=k8s-master1=https://10.99.1.51:2380这行后面添加另2台master的信息

,k8s-master3=https://10.99.1.53:2380,k8s-master2=https://10.99.1.52:2380 #用逗号隔开

保存退出（master2上面只用添加master3的信息；master3不用修改）

★修改完etcd.yaml配置文件后，kubelet会自动重启此etcd的pod容器（不可使用kubectl delete pod命令去手动删除etcd的pod）只需等待2到3分钟即可，

★如果etcd的pod重启失败，可先移动/etc/kubernetes/manifests/etcd.yaml文件到其他地方，这时kubelet会自动删除本结点上的etcd pod，再把此文件移回去，又会重新生成etcd pod

# kubectl exec -it etcd-k8s-master3 sh -n kube-system #进入etcd容器执行以下命令

**# 查看集群成员**

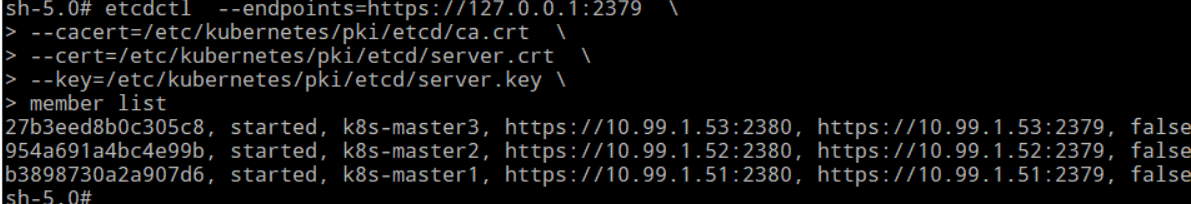
etcdctl --endpoints=https://127.0.0.1:2379 \

--cacert=/etc/kubernetes/pki/etcd/ca.crt \

--cert=/etc/kubernetes/pki/etcd/server.crt \

--key=/etc/kubernetes/pki/etcd/server.key \

member list



**# 查看集群状态**

etcdctl --endpoints=https://10.99.1.51:2379,https://10.99.1.52:2379,https://10.99.1.53:2379 \

--cacert=/etc/kubernetes/pki/etcd/ca.crt \

--cert=/etc/kubernetes/pki/etcd/server.crt \

--key=/etc/kubernetes/pki/etcd/server.key \

endpoint status --write-out=table



**# 查看集群健康状态**

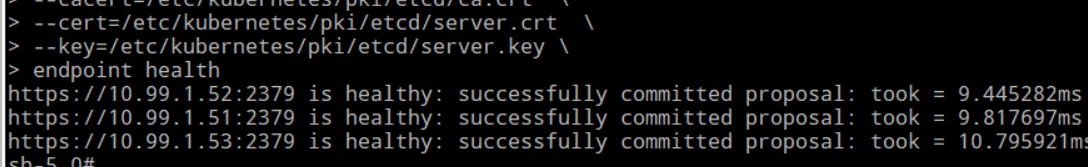
etcdctl --endpoints=https://10.99.1.51:2379,https://10.99.1.52:2379,https://10.99.1.53:2379 \

--cacert=/etc/kubernetes/pki/etcd/ca.crt \

--cert=/etc/kubernetes/pki/etcd/server.crt \

--key=/etc/kubernetes/pki/etcd/server.key \

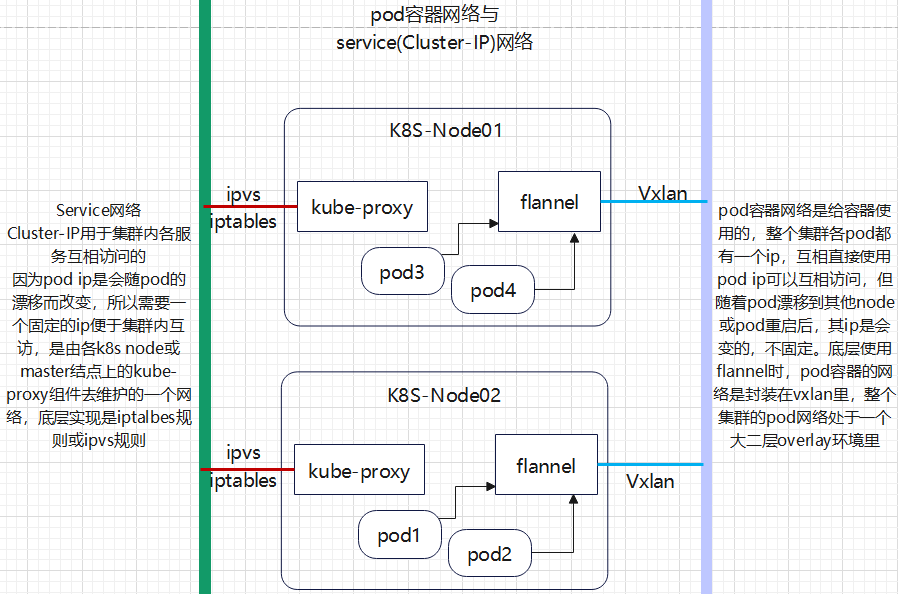
endpoint health



**# 至此，k8s高可用集群搭建完成**

**★第5章、K8S的网络原理**

**★pod网络与Service网络**



**整个K8S集群有3种网络ip：**

|  |  |
| --- | --- |
| Node-ip | 各k8s服务器的通信ip，是服务器本身的通信/管理ip |
| Pod-ip | 各pod容器组的ip，pod容器网络是给容器使用的，整个集群各pod都有一个ip，直接使用pod ip可以互相访问，但随着pod漂移到其他node或pod重启后，其ip是会变的，不固定。底层使用flannel时，pod容器的网络是封装在vxlan里，整个集群的pod网络处于一个大二层overlay环境里 |
| Cluster-ip | Service网络，Cluster-IP用于集群内各服务互相访问的，因为pod ip是会随pod的漂移而改变，所以需要一个固定的ip便于集群内互访，是由各k8s成员结点上的kube-proxy组件去维护的一个网络，底层实现是iptalbes规则或ipvs规则 |

**★K8S服务器上的网络接口信息**

# ip addr #查看ip网卡信息

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN #本地环回口

inet 127.0.0.1/8 scope host lo

2: ens33: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP #k8s服务器本身用于通信的网口

inet 10.99.1.61/24 brd 10.99.1.255 scope global

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN #docker默认的容器网络

inet 172.17.0.1/16 brd 172.17.255.255 #如果不使用docker，则没有此接口

5: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN #Service网络，使用ipvs实现

inet 10.7.0.1/32 scope global kube-ipvs0

valid\_lft forever preferred\_lft forever

inet 10.7.0.10/32 scope global kube-ipvs0 #当访问这些ip加某端口时转为真实的pod ip

valid\_lft forever preferred\_lft forever

6: flannel.1: <BROADCAST,MULTICAST,UP,LOWER\_UP> **mtu 1450** qdisc noqueue state UNKNOWN #vxlan设备

inet 10.244.0.0/32 brd 10.244.0.0 scope global flannel.1

7: cni0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 #网桥设备，是本服务器上的pod网络的网关

inet 10.244.3.1/24 brd 10.244.3.255 scope global cni0

8: veth57d7a776@if3: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 #veth设备，没有配置ip地址，它与某个pod网络连通，且这个veth加入了cni0的网桥里，于是容器pod里的网卡连通了宿主机的cni网桥

# yum install bridge-utils #安装网桥命令工具

[root@k8s-node01 ~]# brctl show #查看网桥（虚拟交换机）

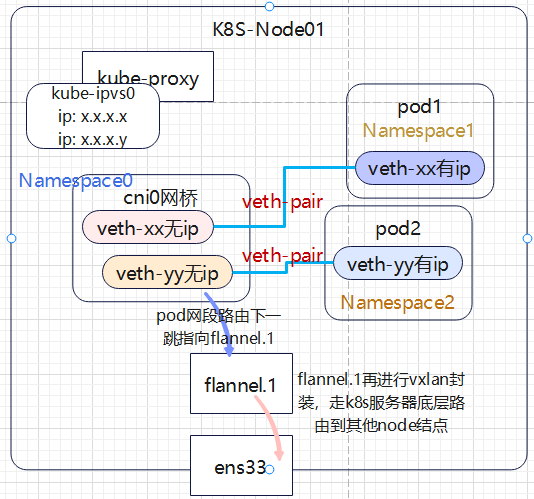
bridge name bridge id STP enabled interfaces

cni0 8000.9e3551d62fed no veth57d7a776

docker0 8000.0242f0503f0d no

#可见k8s服务器上只有2个网桥，一个名为cni0，是k8s的pod网络用的，本k8s服务器上的所有pod的网卡都连通到这个网桥里，即同一台k8s服务器上的所有pod容器处于同一个二层广播域

**★k8s服务器上的各网卡关系图**



[root@k8s-node01 ~]# ip route #查看k8s node结点上的路由表

default via 10.99.1.1 dev ens33 proto static metric 100 #k8s服务器底层默认路由

10.99.1.0/24 dev ens33 proto kernel scope link src 10.99.1.61 metric 100

10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink #去往10.244.0/24的pod网段下一跳指向flannel.1

10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink #去往10.244.1/24的pod网段下一跳指向flannel.1

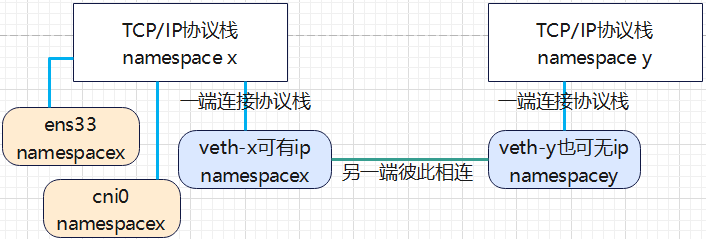
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink #去往10.244.2/24的pod网段下一跳指向flannel.1

10.244.3.0/24 dev cni0 proto kernel scope link src 10.244.3.1 #本node结点上的pod网段

#由上面的路由表可得知，各k8s服务器上的pod容器并不是在同一个二层广播域里，虽然底层是vxlan隧道，所有的pod处于一个overlay网络里，但为了避免二层的广播流量占用大量的网络带宽，所以k8s把pod网段进一步细分了，各服务器占一个子网段，然后各pod子网之间是走路由转发的，路由下一跳为flannel.1设备，它再进行vxlan封装。

**★veth-pair虚拟网络接口**

veth pair是成对的虚拟网络设备，一端连接自己的namespace的tcp/ip协议栈，另一端互相连接；所以veth-pair常用于充当桥梁，连通2个不同的NameSpace网络



NameSpace是Linux 2.6.x之后的内核版本支持的特性，主要用于资源的隔离，有了NameSpace，一个Linux系统可以抽象出多个网络子系统，各子系统都有自己的网络设备、协议栈等，且互不影响；Docker容器服务给每个容器组创建一个NameSpace，以及一个veth-pair设备，其中一半veth-x放在容器的NameSpace里，另一半veth-y放在宿主系统的NameSpace里，再把位于宿主系统的这块veth放到某个网桥里（如Docker0或cni0这些网桥）

**★VxLAN大二层网络隧道**

VxLAN是一种隧道技术，能在三层网络的基础上建立二层以太网络，vxlan的实现方法是将原始的以太网帧加上8字节的vxlan头部，再封装在一个UDP报文里，此报文目的端口为8472/udp，即对端的隧道终端服务监听8472/udp端口。

所以vxlan可以理解为 MAC帧over UDP，Linux内核在3.7.0版本才开始支持vxlan，但linux 3.7发布时，相关的vxlan标准未规定使用哪个端口号，所以大多数厂商使用8472/udp端口，后来IANA分配了4789/udp作为vxlan的目的端口

**VTEP**（Vxlan Tunnel End Point）vxlan隧道终端节点，实现vxlan报文的封装/解封装

**VxLAN头部：**

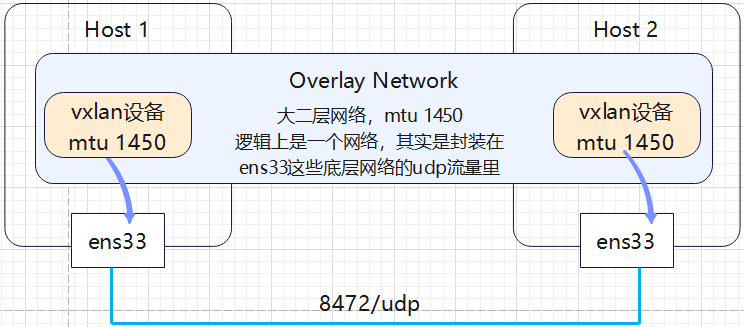
|  |  |  |  |
| --- | --- | --- | --- |
| 8 bit | 24 bit | 24 bit | 8 bit |
| vxlan tag | Reserved | VNI | Reserved |

VNI：（Vxlan Network Identifier），类似于vlan的id，vni区分一个vxlan虚拟overlay网络，vni有24bit，可表示16M个vxlan网络（即16777216个）

vxlan隧道底层网络的mtu如果是1500的话，那么vxlan要求被封装的报文网络mtu为1450

1500 - ( 20+8+8+14 ) = 1450 **所以vxan设备的mtu一般为1450**

20表示ip头，8为udp头，8为vxlan头，14为被封装流量的mac帧头（不带vlan tag）



**★kube-proxy的Service网络服务**

k8s的Service是提供访问到（真实pod容器里面的服务）的一种服务，它是由kube-proxy组件实现的，创建一个service资源时，先注册此服务的dns条目，让服务名称解析到service-ip，kube-proxy会想办法让访问service-ip的流量转发到真实pod里面去，如果一个service服务后端对应多个pod，则kube-proxy会使用负载均衡机制。

★每创建一个service资源，就会创建一个同名的EndPoints

Endpoints：记录了service的targetPort与目标pod的 ip:port映射关系，当目标pod漂移或重启时，endpoints会自动更新

service-ip：是虚拟的ip，是由kube-proxy去建立相应的iptables/ipvs规则进行流量的转发

**★kube-proxy的代理规则模式有：**

|  |  |
| --- | --- |
| 代理模式 | k8s版本要求 |
| User Space proxy mode | v1.0 + |
| iptables proxy mode | v1.1 + |
| ipvs proxy mode | v1.8 + 需要在所有k8s服务器上加载ipvs内核模块 |

效率：ipvs > iptables > UserSpace

**iptables模式**默认只有round robin负载均衡方式，如果第一个pod无响应，则连接失败，可配置pod就绪检查（readiness probe）确保后端pod正常工作，这样iptables模式下kube-proxy只使用健康的后端pod

**ipvs模式**支持更多的负载均衡模式：

|  |  |
| --- | --- |
| rr（round-robin） | wrr（加权轮询） |
| lc（least connection） | wlc（加权最少连接） |
| dh（destination hashing） | lblc 基于局部性最少连接 |
| sh（source hashing） | lblcr 带复制的基于局部性最少连接 |
| sed（shortest expected delay） | nq（never queue） |

默认使用rr

**★k8s的service类型**：

|  |  |
| --- | --- |
| ClusterIp | 只能用于集群内部，iptables,ipvs  cluster-ip:port --> real pod ip:port |
| NodePort | 通过每个node ip上的静态端口 映射到内部自动创建的一个cluster ip，  NodePort默认的有效端口范围是 30000-32768 |
| LoadBalancer | k8s创建NodePort与Cluster-ip；云服务商云上的负载均衡器去监测此k8s集群当发现有创建LoadBalancer时，就自动添加相应的监听器（将用户的请求流量转发到pod所处node结点上的目标service Port或node port，目标node上的ipvs规则是直接将此node port/servce port转发到此node上的pod里） |
| ExternalName | 将服务映射到集群外部的某个资源，要求v1.7及以上版本 |

例：

kind: Service

spec:

type: NodePort

ports:

- port: 443 # service ip 的port

targetPort: 8443 # pod容器里的port

nodePort: 30001 # Node IP对外的port

#

pod所处node结点上：

NodeIP:30001 --> PodIP:8443

ServiceIP:443 --> PodIP:8443

非pod所处node结点上这些规则还会再转一次，转发到目标node上

**★第6章、pod的创建**

**★★创建pod**

创建单个pod，在可操作k8s集群的结点上创建一个yaml文件

# vi pod-nginx-web1-v1.19.5.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-nginx-web1-v1.19.5 #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: lbvalue-nginx-web1

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，**不可带小数点**

image: cof-lee.com/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

#command: [ "/bin/sh", "-ce", "ping -c 3 1.1.1.1" ] #容器入口程序及参数

resources:

requests: #资源需求，定义了容器所需要的最小资源量，便于调度，不能小于jvm中的Xms

memory: "256Mi"

cpu: "200m"

limits: #资源限制，定义了容器最大可以消耗的资源上限，对应jvm中的Xmx

memory: "256Mi"

cpu: "200m" #1核cpu为 1000m，这个m是mili（千分之一）的意思

ports: #端口暴露，仅作提示信息

- name: http

containerPort: 80

protocol: TCP

env: #给容器传递的环境变量

- name: TZ

value: Asia/Shanghai

#securityContext: #安全上下文

# runAsNonRoot: true

# runAsUser: 1000 #指定容器里面的运行服务的用户id

# allowPrivilegeEscalation: false

livenessProbe: #pod存活性探测，对应STATUS，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

timeoutSeconds: 2 #检测超时2秒，默认1秒

periodSeconds: 3 #隔3秒再进行下一次检测，默认1秒

failureThreshold: 2 #2次失败，则判定pod启动失败，默认3次

successThreshold: 1 #1次检测成功，则判定pod启动成功了，默认1次

readinessProbe: #pod就绪性探测，对应READY，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

initialDelaySeconds: 10

timeoutSeconds: 2

periodSeconds: 3

restartPolicy: Always #重启策略，默认Always

#selector: #根据标签选择目标资源

# matchLabels:

# lbnamexx: lbvaluexx

#nodeSelector: #选择特定node结点去运行此容器

# nodelbxx: lbvaluexxxx

---

apiVersion: v1

kind: Service #创建service资源

metadata:

name: svc-nginx-web1 #其他pod可直接访问此名称:servicePort，如 curl -kv http://svc-nginx-web1:2380/

spec:

selector: **#通过标签选择目标pod**

lbname: lbvalue-nginx-web1 #与目标pod的label相同，如果目标pod有多个，则自动实现负载均衡

type: NodePort

ports:

- port: 2380 # service ip 的port

targetPort: 80 # pod容器里的port

nodePort: 30001 # Node IP对外的port

#

# kubectl apply -f pod-nginx-web1-v1.19.5.yaml #应用此文件，创建文件里定义的所有资源

# kubectl get pod -owide #查看pod运行情况

# kubectl get svc #查看service

k8s-node01# docker ps | grep nginx #查看容器

k8s\_nginx-v1-19-5\_pod-nginx-web1-v1.19.5\_default\_41b8e6f3-6b7e-4e72-9fe5-9d8e12b1eebe\_1

k8s\_POD\_pod-nginx-web1-v1.19.5\_default\_41b8e6f3-6b7e-4e72-9fe5-9d8e12b1eebe\_5 （pause容器）

#命名方式为： k8s\_容器名\_pod名\_命名空间\_uuid

# kubectl delete -f pod-nginx-web1-v1.19.5.yaml #删除文件里定义的所有资源

**★容器探测其他配置：**

★HTTP探测

livenessProbe: #pod存活性探测

httpGet: #使用http

path: /xx/cxxx/healthz #请求的文件

port: 80

scheme: HTTP #协议，可为HTTP, HTTPS

**★执行shell命令进行探测**

livenessProbe: #pod存活性探测

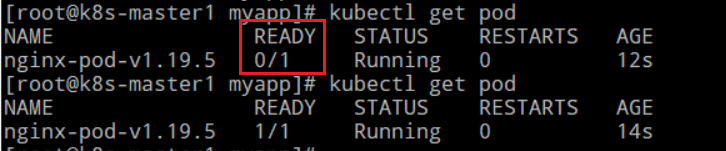
exec: #使用exec，在容器里执行shell命令，返回0则成功

command: ["test", "-e", "/tmp/xxhealthy"] #执行的命令

# kubectl get pod #查看pod，

READY为就绪状态，x/y，x表示就绪的容器数，y为总容器数

STATUS表示容器运行状态，Running为运行中



**★通过kind: Pod创建的容器组，在kubectl delete pod xxx时，它就真的被删除了，不会重新拉起一个新的pod，建议使用pod控制器去创建pod**

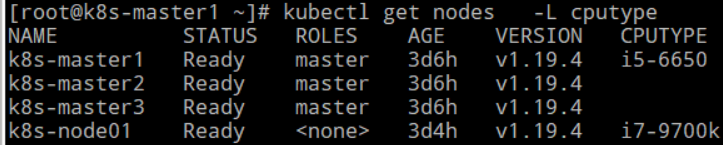
**★给目标node打标签**

# kubectl label nodes node名称 标签名=标签值

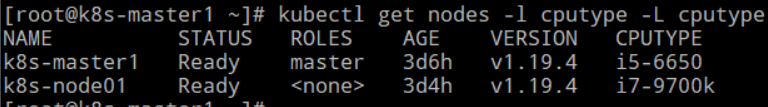
# kubectl label nodes k8s-node01 cputype=i7-9700k #给指定的node结点打标签

# kubectl label nodes --all=true cputype=i7-9700k #给所有node结点打标签

# kubectl get nodes -L cputype #查看所有结点，显示cputype属性列



# kubectl get nodes -l cputype -L cputype #只显示带有标签名为cputype的结点



# kubectl label nodes k8s-node01 cputype- #删除标签，标签名后加个减号

**★查看结点标签**

# kubectl get nodes --show-labels #查看所有结点标签

# kubectl get nodes k8s-node01 --show-labels #查看指定结点标签

# master结点默认标签

beta.kubernetes.io/arch=amd64, beta.kubernetes.io/os=linux, kubernetes.io/arch=amd64,

kubernetes.io/hostname=k8s-master2, kubernetes.io/os=linux,

node-role.kubernetes.io/master=

# node结点默认标签

beta.kubernetes.io/arch=amd64, beta.kubernetes.io/os=linux, kubernetes.io/arch=amd64,

kubernetes.io/hostname=k8s-node03, kubernetes.io/os=linux

**★namespace**

创建命名空间

# cat > namespace-testxx.yaml <<EOF

apiVersion: v1

kind: Namespace

metadata:

name: testxx

labels:

labelnamexx: valuexxx

EOF

# kubectl apply -f namespace-testxx.yaml

**★镜像获取方式imagePullPolicy**

在containers/- name/下面，与image标签同级

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，**不可带小数点**

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取，默认

imagePullPolicy: Never #只从node结点本地获取，本地没有则报错

imagePullPolicy: Always #只从网络获取

**★列出k8s所有资源类型名称**

# kubectl api-resources #列出所有资源类型详名称及缩写名称

# kubectl api-resources --verbs=list --namespaced -o name #列出所有资源类型名称

**★LimitRange**

LimitRange资源在每个命名空间中为每个容器/pod指定最小及最大计算资源/内存使用量限制，任何违反LimitRange定义的资源使用最大用量的请求都将被直接拒绝

LimitRange对pod和容器的资源限制为cpu和内存使用量

LimitRange对PVC的资源限制为存储空间的使用量

资源限额只对新创建的资源生效，对于已经存在的对象不产生任何限制

limitrange和resourcequota都是有命名空间之分的

# cat > limitRange-testxx.yaml <<EOF

apiVersion: v1

kind: LimitRange

metadata:

name: limitrange-testxx

namespace: default

spec:

limits:

- type: Container #对每个容器的限制，也可指定type为Pod

default: #定义默认的**资源限制**，同创建pod时指定的resources/limits

cpu: 200m #200m表示0.2个vcpu，1vcpu==1000m

memory: "256Mi"

defaultRequest: #定义默认的**资源需求**，同创建pod时指定的resources/requests

cpu: 200m

memory: "256Mi"

min: #定义最小的资源用量

cpu: 50m

memory: "128Mi"

max: #定义最大资源用量

cpu: 2000m

memory: "4096Mi"

maxLimitRequestRatio: #最大资源用量（同max，二选一）只是指定的资源量不是具体值，而是min的倍数

cpu: 10 #这个是指10倍的最小量（min），而不是10个cpu

EOF

# kubectl apply -f limitRange-testxx.yaml

**★ResourceQuota**

LimitRange只能限制单个容器/Pod/PVC的资源使用量，无法限制创建的容器个数，用户可能会创建数量众多的pod而消耗完所有资源，可以使用ResourceQuota资源限制每个命名空间的对象数量及资源限额

ResourceQuota支持限制每种资源类型的对象总数，以及所有对象消耗的资源问量

资源限额只对新创建的资源生效，对于已经存在的对象不产生任何限制

# cat > resourceQuota-testxx.yaml <<EOF

apiVersion: v1

kind: ResourceQuota

metadata:

name: resourcequota-testxx

namespace: default

spec:

hard:

pods: "10"

requests.cpu: "4" #同cpu: 4，cpu资源需求的总量限额

requests.memory: 2Gi #同memory: 2Gi，内存资源需求的总量限额

limits.cpu: "8" #cpu资源限制的总量限额

limits.memory: 4Gi #内存资源限制的总量限额

count/deployments.apps: "2" #可以创建的deployment总数

count/deployments.extensions: "2"

persistentvolumeclaims: "2" #可以创建的pvc总数

requests.storage: 20Gi #可以创建的pvc存储需求存储空间总量限额

EOF

# kubectl apply -f resourceQuota-testxx.yaml

**★第7章、pod控制器**

pod控制器是由kube-controller-manager组件提供的一些资源，负责控制pod的创建、删除、重新调度、运行时间等。常用的pod控制器有ReplicaSet, Deployment, DaemonSet, Job, CronJob等，如果pod被删除，则可以重新拉起一个

★通过kind: Pod创建的容器组，在kubectl delete pod xxx时，它就真的被删除了，不会重新拉起一个新的pod，建议使用pod控制器去创建pod

**★控制器原理：**

for {

实际状态 = 获取集群中对象X的实际状态（Actual State）

期望状态 = 获取集群中对象X的期望状态（Desired State）

if 实际状态 == 期望状态 {

什么都不做

} else {

执行编排动作，将实际状态调整为期望状态

}

}

**★Deployment控制器**

deployment构建于**ReplicaSet控制器**之上，可为pod和replicaset资源提供声明式更新。

# vi nginx-deployment.yml #内容如下

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment #Deployment名，也是pod名，不同于容器名

spec:

replicas: 2 #pod运行副本数（实例数）

selector: #选择相应pod模板

matchLabels:

podlable: nginx-deploy-pod

template: #定义一个pod模板，具体要运行多少个实例由上面的replicas决定

metadata:

labels:

podlable: nginx-deploy-pod #pod模板标签

spec:

containers: #定义一组容器

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "200m"

limits: #资源限制

memory: "256Mi"

cpu: "200m" # 1核cpu为1000m

ports: #端口暴露，仅作提示信息，无实际影响

- name: http

containerPort: 80

protocol: TCP

env: #给容器传递的环境变量

- name: TZ

value: Asia/Shanghai

livenessProbe: #pod存活性探测，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

readinessProbe: #pod就绪性探测，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

initialDelaySeconds: 10

restartPolicy: Always #重启策略，默认Always，deploy不支持Never

---

apiVersion: v1

kind: Service #创建service资源，不归deployment管理

metadata:

name: nginx-deploy-svc

spec:

selector: #通过标签选择目标pod

podlable: nginx-deploy-pod #与目标pod的labels相同

type: NodePort

ports:

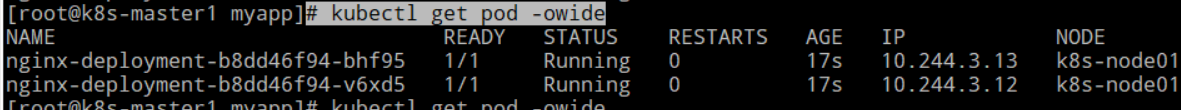
- port: 2380 # service ip 的port

targetPort: 80 # pod容器里的port

nodePort: 30002 # Node IP对外的port

# kubectl apply -f nginx-deployment.yml #应用deployment

# kubectl get pod -owide #查看pod，AGE后的IP为pod-ip



# kubectl get deployment #查看定义的deployment

**★DaemonSet控制器**

DaemonSet用于在集群中的全部节点上同时运行一份指定的pod副本（master打了污点，所以默认不运行，可以设置容忍度）后续新加入集群的工作节点也会自动创建一个相关的pod副本，这些pod一般是执行系统级操作任务的服务，如应用代理，监控，日志收集等

# vi monit.daemonset.yml #内容如下

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: mymonit-ds #daemonset名

labels:

ds-lb: dslabel #打标签

spec:

selector: #选择相应pod模板

matchLabels:

podlable: ds-pod

template:

metadata:

labels:

podlable: ds-pod #pod模板标签

spec:

containers: #定义一组容器

- name: dsxx #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/xxx #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "200m"

limits: #资源限制

memory: "256Mi"

cpu: "200m" #1核cpu为 1000m

livenessProbe: #pod存活性探测，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

readinessProbe: #pod就绪性探测，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

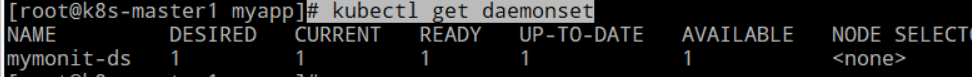
initialDelaySeconds: 10

restartPolicy: Always #重启策略，默认Always

#

# kubectl apply -f monit.daemonset.yml #应用

# kubectl get daemonset #查看



**★Job控制器**

job控制器用于运行一次性pod（如执行sql脚本或一次性备份任务等）容器中的进程在正常运行结束后，置为completed状态，且不会再重启，如果进程运行失败，则根据配置决定是否要重启

# vi xx-job.yml #内容如下

apiVersion: batch/v1

kind: Job

metadata:

name: myjob #Job名

spec:

completions: 3 #一共起1个pod（默认1）

parallelism: 2 #可同时运行的pod数（默认1）

template:

spec:

containers: #定义一组容器

- name: jobxx #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/fdsfsd #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "200m"

limits: #资源限制

memory: "256Mi"

cpu: "200m" #1核cpu为 1000m

restartPolicy: Never #重启策略，默认Always

#

**★CronJob控制器**

cronjob控制器用于运行周期性pod，定时执行

# vi xx-cronjob.yml #内容如下

apiVersion: batch/v1beta1

kind: CronJob

metadata:

name: mycronjob #cronJob名

labels:

cjlb: xxx #cronjob的标签

spec:

schedule: "\*/2 \* \* \* \*" #分时日月周

concurrencyPolicy: Allow #前一次作业未完成时，下一次作业是否允许运行/如何运行

# Allow允许下一次作业运行，Forbid禁止，Replace替换

failedJobsHistoryLimit: 1 #失败的任务 历史记录数，默认1

successfulJobsHistoryLimit: 3 #成功的任务 历史记录数，默认3

jobTemplate:

metadata:

labels:

joblb: jbxxx #job的标签

spec:

template:

spec:

containers: #定义一组容器

- name: jobxx #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/fdsew #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "200m"

limits: #资源限制

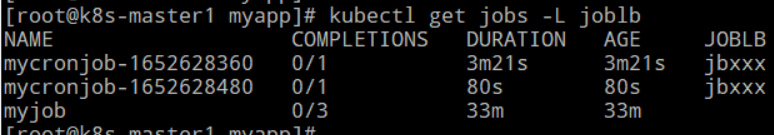
memory: "256Mi"

cpu: "200m" #1核cpu为 1000m

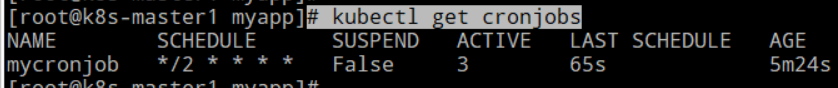
restartPolicy: Never #重启策略，默认Always

#

# kubectl get jobs -l joblb: jbxxx #查看cronjob控制器创建的job资源，数量由successfulJobsHistoryLimit决定



# kubectl get cronjobs #查看cronjobs



**★StatefulSet控制器**

在使用deployment时，创建的pod名称是没有顺序的，是随机字符串，在用statefulset管理pod时要求pod名称必须是有序的，每一个pod不能被随意取代，pod重建后pod名称还是一样的（pod ip会变）。

Deployment称为无状态应用，StatefulSet称为有状态应用

statefulset管理pod的启停顺序：

|  |  |
| --- | --- |
| 有序部署 | 部署StatefulSet时，如果有多个Pod副本，它们会被顺序地创建（从0到N-1）  并且在下一个Pod运行之前，前面的所有Pod都必须是Running和Ready状态 |
| 有序删除 | 当Pod被删除时，它们被终止的顺序是从N-1到0 |
| 有序扩展 | 当对Pod执行扩展操作时，与部署一样，它前面的Pod必须都处于Running和Ready状态 |

# vi nginx-statefulset.yml #内容如下

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: nginx-statefulset #StatefulSet名，也是pod名，不同于容器名

spec:

replicas: 2 #pod运行副本数（实例数）

selector: #选择相应pod模板

matchLabels:

podlable: nginx-pod-sts

serviceName: nginx-sts-svc #必须指定serviceName关联svc

template: #定义一个pod模板，具体要运行多少个实例由上面的replicas决定

metadata:

labels:

podlable: nginx-pod-sts #pod模板标签

spec:

containers: #定义一组容器

- name: nginx-v1-19-5-sts #容器名，不可带小数点

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

limits: #资源限制

memory: "256Mi"

cpu: "200m" # 1核cpu为1000m

livenessProbe: #pod存活性探测，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

readinessProbe: #pod就绪性探测，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

initialDelaySeconds: 10

restartPolicy: Always #重启策略，默认Always，deploy不支持Never

---

apiVersion: v1

kind: Service #创建service资源，不归deployment管理

metadata:

name: nginx-sts-svc

spec:

selector: #通过标签选择目标pod

podlable: nginx-pod-sts #与目标pod的labels相同

type: NodePort

ports:

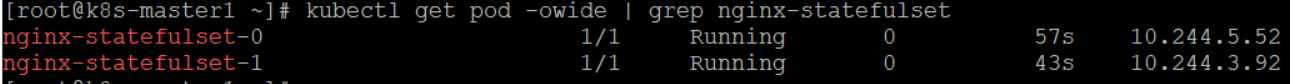
- port: 2680 # service ip 的port

targetPort: 80 # pod容器里的port

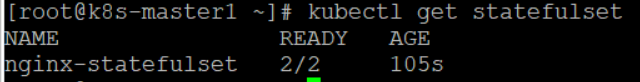
nodePort: 30031 # Node IP对外的port

# kubectl apply -f nginx-statefulset.yml #应用deployment

# kubectl get pod -owide | grep nginx-statefulset #查看pod



# kubectl get statefulset #查看statefulset



**★强制删除pod**

# kubectl delete pod podNamexx --force --grace-period=0

**★scale调整replicas副本数**

# kubectl scale deployment deployName --replicas=5

# kubectl scale deployment/deployName --replicas=5

# kubectl scale statefulset/nginx-statefulset --replicas=3

#scale命令调整的副本数会写入相应的dep/sts配置清单中

**★第8章、Service和Ingress**

**★创建Service**

**①ClusterIP类型**

# vi mynginx-svc.yml #内容如下

apiVersion: v1

kind: Service #创建service资源

metadata:

name: mynginx-svc #其他pod可直接访问此名称加servicePort

# 如 curl -kv http://mynginx-svc:1380/

spec:

selector: #通过标签选择目标pod

podlable: ds-pod #与目标pod的labels相同，如果目标pod有多个，则自动实现负载均衡

type: ClusterIP

ports:

- protocol: TCP #默认为TCP，

port: 1380 # service ip 的port

targetPort: 80 # pod容器里的port

sessionAffinity: ClientIP #把来自同一源ip的请求始终转发到同一pod

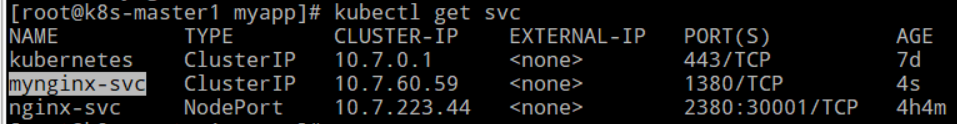
sessionAffinityConfig:

clientIP:

timeoutSeconds: 3600 #默认10800秒（3小时）

# kubectl apply -f mynginx-svc.yml #应用

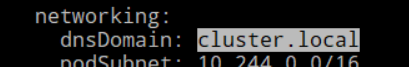
# kubectl get svc #查看



★当创建service资源对象时，coreDNS会自动创建相应的名称解析，集群的各pod可直接使用service的name加service的port 去访问目标服务

名称解析默认后缀为集群初始化时--cluster-domain指定的域名后缀，所以如果pod里的/etc/resolv.conf的search后得指定默认查询的域，如果不对，则域名解析会失败

search default.svc.cluster.local svc.cluster.local cluster.local



**②NodePort类型**

# vi nginx-svc.yml #内容如下

apiVersion: v1

kind: Service #创建service资源

metadata:

name: nginx-svc **#其他pod可直接访问此名称:servicePort**

# 如 curl -kv http://nginx-svc:2380/

spec:

selector: **#通过标签选择目标pod**

lbname: lbvalue #与目标pod的labels相同，如果目标pod有多个，则自动实现负载均衡

type: NodePort #默认为 ClusterIP

ports:

- port: 2380 # service ip 的port

targetPort: 80 # pod容器里的port

nodePort: 30001 # Node IP对外的port

**③LoadBalancer类型**

# vi mynginx-svc-lb.yml #内容如下

apiVersion: v1

kind: Service #创建service资源

metadata:

name: mynginx-svc-lb #其他pod可直接访问此名称:servicePort

# 如 curl -kv http://mynginx-svc-lb:4380/

spec:

selector: #通过标签选择目标pod

podlable: nginx-pod #与目标pod的labels相同，如果目标pod有多个，则自动实现负载均衡

type: LoadBalancer

ports:

- protocol: TCP

port: 4380 # service ip 的port

targetPort: 80 # pod容器里的port

nodePort: 30003 # Node IP对外的port

sessionAffinity: ClientIP #把来自同一源ip的请求始终转发到同一pod

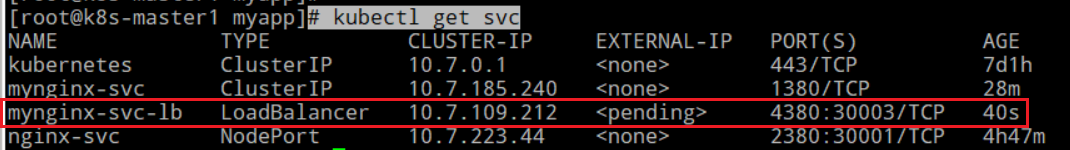
sessionAffinityConfig:

clientIP:

timeoutSeconds: 3600 #默认10800秒（3小时）

# kubectl apply -f mynginx-svc-lb.yml #应用

# kubectl get svc #查看



**★expose暴露端口**

kubectl expose命令会将目标deployment/pod里声明的端口暴露出来，即自动创建NodePort类型的svc

# kubectl expose deployment dep-xxx --type=NodePort --name=svc-test1 #将deploy里的端口暴露出来

# kubectl expose pod pod-nginx-web1 --type=NodePort --name=svc-test2 #将pod里的端口暴露出来

# kubectl get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes ClusterIP 10.7.0.1 <none> 443/TCP 34d

my-service-1 NodePort 10.7.35.6 <none> 80:31796/TCP 40m

# curl http://node1.cof-lee.com:31796/ #测试

**★修改NodePort端口范围**

NodePort默认的有效端口范围是 30000-32768

# vi /etc/kubernetes/manifests/kube-apiserver.yaml

spec:

containers:

- command:

- kube-apiserver #在这行下面添加一行

- --service-node-port-range=20000-21000

保存退出之后，kube-apiserver的pod将自动更新，更新期间kube-apiserver将不可用，在生产环境谨慎操作！

# kubectl get pod #执行kubectl命令报错，是正常的，kube-apiserver在重启期间不可用

The connection to the server 10.99.1.245:6443 was refused - did you specify the right host or port?

等待几分钟恢复

对于旧的svc，端口范围不在刚刚设置的范围内，也是生效的，新创建的svc端口范围必须在设置的范围内

**★创建ingress**

**①部署ingress控制器**

ingress控制器（Ingress Controller）得单独安装，Ingress控制器可基于某ingress资源定义的规则将客户端的请求流量直接转发至与Service对应的后端pod资源上，绕过service直接转发到真实pod上。Ingress资源是基于http的host名或url的转发规则

k8s-ingress-nginx官网地址 https://kubernetes.github.io/ingress-nginx/deploy/

# wget https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.2.0/deploy/static/provider/cloud/deploy.yaml

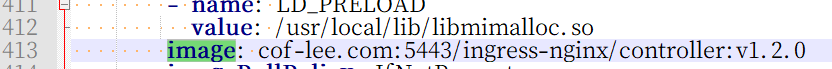
或者 wget https://limaofu.github.io/scripts/ingress-nginx-controller-v1.2.0.yaml

# 需要提前下载以下2个镜像

k8s.gcr.io/ingress-nginx/controller:v1.2.0

k8s.gcr.io/ingress-nginx/kube-webhook-certgen:v1.1.1

# vi ingress-nginx-controller-v1.2.0.yaml #可以指定image为环境里的镜像仓库地址





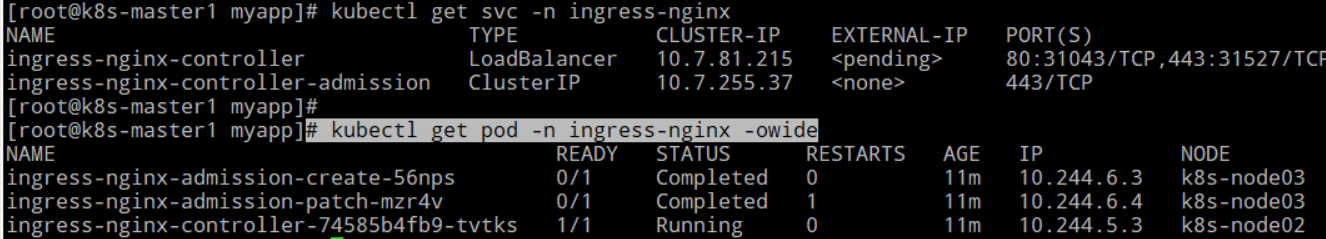


# kubectl apply -f ingress-nginx-controller-v1.2.0.yaml #创建ingress-controller服务

#是以Deployment控制器去管理ingress控制器的pod，并通过LoadBalancer类型的service向集群外部提供服务

# kubectl get svc -n ingress-nginx #查看svc

# kubectl get pod -n ingress-nginx -owide #查看pod



**# ingress-nginx-controller默认的Service**

apiVersion: v1

kind: Service

metadata:

#略

name: ingress-nginx-controller

namespace: ingress-nginx

spec:

externalTrafficPolicy: Local

ports:

- appProtocol: http

name: http

port: 80 #service port

protocol: TCP

targetPort: http #pod容器里的port, 80

- appProtocol: https

name: https

port: 443 #service port

protocol: TCP

targetPort: https #pod容器里的port, 443

selector:

#略

type: LoadBalancer

**# Deployment里选择目标node**

nodeSelector:

kubernetes.io/os: linux #默认所有linux服务器都有这标签

#默认并未指定固定的nodePort，实际部署时可指定固定nodePort以及添加nodeSelector选择指定node结点去运行此容器，然后在那指定的node上要有eip（公网ip），就可以提供外部访问

**②创建http的ingress**

ingress资源仅通过Service资源匹配后端pod，ingress控制器直接转发流量到后端的pod上，不经过service

# vi myweb-ingress.yml #内容如下

apiVersion: networking.k8s.io/v1beta1 **# k8s v1.14 to v1.18**

kind: Ingress

metadata:

name: myweb-ingress

namespace: default

annotations: #资源注解，仅起提示作用

kubernetes.io/ingress.class: "nginx"

spec:

rules: #以下是各转发规则，根据匹配的hostName去转发

- host: "web.xxx.com"

http:

paths:

- path: / #若不指定url，则默认就是/根路径，全部转发

backend:

serviceName: mynginx-svc

servicePort: 1380

- host: "web2.xxx.com"

http:

paths: # host下还可根据不同的url去转发到不同svc

- path: /apixx #转发到后端时也是带这个url的

backend:

serviceName: mynginx-svc2

servicePort: 2390

- path: /staticxx

backend:

serviceName: mynginx-svc

servicePort: 1380

#

# vi myweb-ingress.yml #内容如下

apiVersion: networking.k8s.io/v1 **# k8s v1.19+**

kind: Ingress

metadata:

name: myweb-ingress

namespace: default

annotations: #资源注解，仅起提示作用

kubernetes.io/ingress.class: "nginx"

spec:

rules: #以下是各转发规则，根据匹配的hostName去转发

- host: "web.xxx.com"

http:

paths:

- path: / #若不指定url，则默认就是/根路径，全部转发

pathType: Prefix

backend:

service:

name: mynginx-svc

port:

number: 1380

- host: "\*.yyy.com" #支持\*通配符域名，得放在双引号里

http:

paths: #host下还可根据不同的url去转发到不同svc

- path: /apixx/ #转发到后端时也是带这个url的

pathType: Prefix

backend:

service:

name: mynginx-svc2

port:

number: 2390

- path: /staticxx #转发到后端时也是带这个url的

pathType: Prefix

backend:

service:

name: mynginx-svc

port:

number: 1380

# kubectl apply -f myweb-ingress.yml #应用

**③创建tls类型的ingress资源**

先创建tls证书，再把证书做成secret资源，tls类型的ingress不能直接使用私钥和证书文件，可使用secret资源

# kubectl create secret tls web-inress-secret1 \

--cert=web.xxx.com.crt \

--key=web.xxx.com.key -n default

#创建一个名为web-ingress-secret1的secret资源，在default命名空间里

# kubectl get secret #查看

# vi myweb-ingress-tls.yml #内容如下

apiVersion: networking.k8s.io/v1 **# k8s v1.19+**

kind: Ingress

metadata:

name: myweb-ingress-tls

namespace: default

annotations: #资源注解，仅起提示作用

kubernetes.io/ingress.class: "nginx"

spec:

tls:

- hosts:

- "web.xxx.com"

secretName: web-inress-secret1 #使用指定的secret资源

rules: #以下是各转发规则，根据匹配的hostName去转发

- host: "web.xxx.com"

http:

paths:

- path: / #若不指定url，则默认就是/根路径，全部转发

pathType: Prefix

backend:

service:

name: mynginx-svc

port:

number: 1380

#

# kubectl apply -f myweb-ingress-tls.yml #应用

然后客户端就可使用https://web.xxx.com:<ingress-https-node-port> 去访问目标服务了

**★第9章、存储卷**

pod里的容器在运行结束后，其数据无法持久存在，得额外挂载一个外部存储文件系统，即存储卷，存储卷是pod内所有容器都可挂载的共享目录。k8s支持node结点本地存储以及网络存储

node本地存储：emptyDir hostPath

网络存储：NFS Ceph GlusterFS 等

k8s集群级别的存储资源：PV（PersistentVolume）

**①node结点存储卷hostPath**

在创建pod时直接指定volumes为hostPath（node结点上的本地目录）

# vi pod-nginx-use-volume.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-nginx-use-volume #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: lbvalue-nginx-use-volume

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

livenessProbe: #pod存活性探测，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

readinessProbe: #pod就绪性探测，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

initialDelaySeconds: 10

volumeMounts: #pod容器要使用的存储卷声明

- name: alog #根据此名称去寻找对应的volume存储卷

mountPath: /var/log/nginx #容器里的挂载点

readOnly: false #非只读（即可读写）

volumes:  **#存储卷定义，与containers同级**

- name: alog #存储卷名称

hostPath: #存储卷类型，hostPath表示node结点本地目录

path: /nginxlog/ #node结点上的本地目录

type: DirectoryOrCreate #默认，指定目录不存在时则自动创建，权限0755

#type: Directory #指定的目录必须存在，不存在时则报错，容器无法启动

restartPolicy: Always #pod重启策略，默认Always

---

apiVersion: v1

kind: Service #创建service资源

metadata:

name: svc-nginx-use-volume #其他pod可直接访问此名称:servicePort

spec:

selector: #通过标签选择目标pod

lbname: lbvalue-nginx-use-volume #与目标pod的labels相同，如果目标pod有多个，则自动实现负载均衡

type: NodePort

ports:

- port: 2382 # service ip 的port（servicePort）

targetPort: 80 # pod容器里的port

nodePort: 30071 # Node IP对外的port

# kubectl apply -f pod-nginx-use-volume.yml

**②nfs存储卷**

在创建pod时，直接在volumes里指定使用nfs共享目录

spec:

containers:

- name: nginx-v1-19-5

#xxx省略其他配置项

volumeMounts:

- name: alog

mountPath: /var/log/nginx

volumes:

- name: alog

nfs:

server: 10.99.1.71 # nfs server ip/域名

path: /nfs-share #nfs serv导出的共享目录

readOnly: false #默认false

★要求所有node结点上安装nfs客户端

# yum install rpcbind nfs-utils -y #安装nfs客户端

**③PVC**

PV是要用户了解所用到的网络存储系统的具体细节才可使用相关存储卷，而pvc使得用户可以以抽象的方式去使用k8s集群的PV资源，pvc是pv的消费者

PVC（PersistentVolumeClaim）向PV申请特定大小的空间，从而创建出pvc存储卷，再由pod通过pvc存储卷去关联使用具体的PV，pvc与pv是一一对应关系，每个PV只可给一个pvc使用

pv和pvc的存储空间只是申请的大小，不是后端存储实际大小，后端存储实际大小可小于pv的大小

pv要大于等于pvc的大小，pvc申请到pv绑定到pv后，大小也变为pv的大小了

PV是集群级别的资源，不属于任何名称空间；用户需要通过PVC向PV提出使用申请，最终的容器是与pvc关联的

pvc有命名空间之分，pv不分命名空间

**★创建存储类SC**

StorageClass存储类是一个逻辑上的分组，根据后端存储的特性、性能、用作等将PV分到不同的类里，创建PVC时可指定使用某个存储类的PV，方便管理。

只有属于同一StorageClass和PVC和PV才能产生绑定关系，即没有指定StorageClass的PVC只和没有指定StorageClass的PV绑定

# cat > storage-class-localdir.yaml <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: storage-class-localdir

namespace: default

provisioner: kubernetes.io/no-provisioner #供给方，以kubenetes.io为前缀

#volumeBindingMode: VolumeBindingImmediate #定义如何为PVC完成供给和绑定，默认VolumeBindingImmediate

volumeBindingMode: WaitForFirstConsumer #等待第一个消费者使用时才绑定（有pod使用pvc时才绑定）

EOF

# kubectl create -f storage-class-localdir.yaml

# kubectl get sc

**★创建PV（使用NFS后端）**

# yum install rpcbind nfs-utils -y #所有node节点上都要安装nfs客户端

不同的PV可以使用同一个nfs后端

k8s默认挂载nfs时，默认使用root用户挂载，如果nfs服务端未配置no\_root\_squash，则会被压缩为nobody，配置了no\_root\_squash后，不会被压缩；当有pod使用此nfs的pvc后，在pod所在的node节点上查看挂载情况如下：

# cat /proc/mounts | grep nfs-share

10.99.1.246:/nfs-share /var/lib/kubelet/pods/708e1945-9cba-48fe-bf53-c42e9bcc5126/volumes/kubernetes.io~nfs/pv-nfs-01 nfs4 rw,relatime,vers=4.1,rsize=262144,wsize=262144,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=sys,clientaddr=10.99.1.242,local\_lock=none,addr=10.99.1.246 0 0

# cat > pv-nfs-share1.yaml <<EOF

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-nfs-share1 #此PV的名称

labels:

pvlab: pvlab-nfs-share1

spec:

capacity: #此PV的容量

storage: 10Gi

volumeMode: Filesystem #卷模型，指定此卷是文件系统还是裸格式的块设备，默认为文件系统

accessModes: #访问模式

- ReadWriteMany #可同时被多个node结点读写挂载

#- ReadWriteOnce #同时只可被一个node结点读写挂载

#- ReadOnlyMany #可同时被多个node结点只读挂载

persistentVolumeReclaimPolicy: Retain #PV空间被释放时的处理机制，Retain保持不动，Recycle回收

#，Retain保持不动，Recycle或Delete回收

storageClassName: slow #当前PV所属的存储类名称，默认为空（不属于任何存储类）

mountOptions: #挂载nfs的参数

- hard #如 ro, soft, hard, sync

- nfsvers=4.1

nfs: #使用NFS存储

path: "/nfs-share"

server: 10.99.1.71

EOF

# kubectl apply -f pv-nfs-share1.yaml

**★创建PV（使用RBD后端）**

# cat > pv-rbd-image-test1.yaml <<EOF

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-rbd-image-test1 #此PV的名称

labels:

pvlab: pvlab-rbd-image-test1

spec:

capacity: #此PV的容量

storage: 10Gi

accessModes: #访问模式

- ReadWriteOnce #同时只可被一个node结点读写挂载

#- ReadOnlyOnce #同时只可被一个node结点只读挂载?

persistentVolumeReclaimPolicy: Retain #PV空间被释放时的处理机制，Retain保持不动，Recycle回收

storageClassName: slow #当前PV所属的存储类名称，默认为空（不属于任何存储类）

rbd: #使用RBD磁盘

monitors:

- ceph1.cof-lee.com:6789

- ceph2.cof-lee.com:6789

- ceph3.cof-lee.com:6789

pool: k8s

image: image-test1

user: admin

secretRef:

name: ceph-secret

fsType: ext4

readOnly: false

EOF

# kubectl apply -f pv-rbd-image-test1.yaml

**★创建PV（使用node节点本地目录）**

# cat > pv-local-dir1.yaml <<EOF

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-local-dir1 #此PV的名称

labels:

pvlab: pvlab-local-dir1

spec:

capacity: #此PV的容量

storage: 10Gi

volumeMode: Filesystem #卷模型，指定此卷是文件系统还是裸格式的块设备，默认为文件系统

accessModes: #访问模式

- ReadWriteMany #可同时被多个node结点读写挂载

#- ReadWriteOnce #同时只可被一个node结点读写挂载

#- ReadOnlyMany #可同时被多个node结点只读挂载

persistentVolumeReclaimPolicy: Retain #PV空间被释放时的处理机制，Retain保持不动，Recycle回收

storageClassName: storage-class-localdir #当前PV所属的存储类名称，默认为空（不属于任何存储类）

local: #使用node节点上的本地目录

path: /dirxx

nodeAffinity: #节点亲和性，指定在某些节点上创建，也可不指定

required:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/hostname

operator: In

values:

- node1.cof-lee.com #指定node结点的hostname

- node2.cof-lee.com #指定node结点的hostname

EOF

# kubectl apply -f pv-local-dir1.yaml

**★创建PVC**

PVC（PersistentVolumeClaim）通过申请占用某个PV而创建，它与pv是一一对应的，用户无需关心其底层实现细节，申请时，用户只需要指定目标空间大小，访问模式，pv标签选择器和StorageClass等相关信息即可。

# cat > pvc-nfs-share1.yaml <<EOF

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc-nfs-share1

labels:

pvclab: pvclab-nfs-share1

spec:

accessModes: #访问模式，同PV的访问模式

- ReadWriteMany

volumeMode: Filesystem

resources: #当前PVC需要占用的最小资源量

requests:

storage: 10Gi

storageClassName: *slow* #存储类，要和PV的存储类保持一致，若pv未指定存储类，则pvc也不指定

volumeName: pv-nfs-share1 #指定使用此pv，与下面的selector二选一，

selector: #用于挑选特定的PV，可通过标签选择器或匹配条件表达式

matchLabels:

pvlab: pvlab-nfs-share1 #通过labels去匹配pv，若未指定volumeName也未指定标签选择器,则从存储类名相同的PV里随便挑一个大小符合的

EOF

# kubectl apply -f pvc-nfs-share1.yaml

**★pod使用pvc**

# vi pod-nginx-use-pvc.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-nginx-use-pvc #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: lbvalue-nginx-use-pvc

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

livenessProbe: #pod存活性探测，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

readinessProbe: #pod就绪性探测，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

initialDelaySeconds: 10

volumeMounts: #pod容器要使用的存储卷声明

- name: logdir #根据此名称去寻找对应的volume存储卷

mountPath: /var/log/nginx #容器里的挂载点

readOnly: false #非只读（即可读写）

volumes:  **#存储卷定义，与containers同级**

- name: logdir #存储卷名称

persistentVolumeClaim:

claimName: pvc-nfs-share1 #指定要使用的pvc

ReadOnly: false #默认就是非只读

restartPolicy: Always #pod重启策略，默认Always

---

apiVersion: v1

kind: Service #创建service资源

metadata:

name: svc-nginx-use-pvc #其他pod可直接访问此名称:servicePort

spec:

selector: #通过标签选择目标pod

lbname: lbvalue-nginx-use-pvc #与目标pod的labels相同，如果目标pod有多个，则自动实现负载均衡

type: NodePort

ports:

- port: 2382 # service ip 的port（servicePort）

targetPort: 80 # pod容器里的port

nodePort: 30091 # Node IP对外的port

# kubectl apply -f pod-nginx-use-volume.yml

**④存储类与动态pv供给**

PV动态供给表示：当没有匹配的PV时，k8s集群会尝试为PVC动态创建符合需求的PV，此方式依赖于存储类的辅助，PVC必须向一个已申明的存储类发起动态分配PV的请求，没有指定存储类的PVC请求会被禁止动态创建PV

**★第10章、ConfigMap和Secret**

ConfigMap对象就是一系列配置数据的集合，这些数据以键值对的形式进行存储，可“注入”到pod对象中，并为容器应用所使用。注入方式有挂载为存储卷和传递为环境变量两种。

configmap是区分命名空间的

**①创建ConfigMap对象**

★命令行直接创建

# kubectl create configmap cm-name1 \ #cm名称不能带\_，得和dns名称规则一样

--from-literal=key1=value1 \

--from-literal=key2=value2

★使用清单文件创建

# cat > cm-name2.yaml <<EOF

apiVersion: v1

kind: ConfigMap

metadata:

name: cm-name2

namespace: default

data:

key1: value1

key2: value2

key3: value3

EOF

# kubectl apply -f cm-name2.yaml #创建

**②pod使用configMap**

# vi nginx-pod-configmap-tz.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod-cm-tz #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: lbvalue-cm-tz

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "200m"

limits: #资源限制

memory: "256Mi"

cpu: "200m" #1核cpu为 1000m

**env:** #给容器传递的环境变量

- name: TZ #环境变量名，有限个数的变量

valueFrom: #环境变量值取自：

configMapKeyRef:

name: cm-name-tz #使用的configmap名称

key: TZ #configmap的键名

livenessProbe: #pod存活性探测，失败后根据restartPolicy决定是否重启

tcpSocket: #检测tcp端口是否可连接

port: 80

initialDelaySeconds: 10 #pod刚拉起时延迟10秒再检测，默认0秒

readinessProbe: #pod就绪性探测，失败后不会重启pod，只是服务不可访问

tcpSocket:

port: 80

initialDelaySeconds: 10

restartPolicy: Always #重启策略，默认Always

#

#如果configmap里有多个键值对，则直接全部注入pod中，注入时可指定键名前缀，防止从多个cm中导入的键名有冲突（如果cm中的键名中有-减号，则自动转为\_下划线）

**envFrom:** #给容器传递的环境变量，从cm中全部导入，和env同级

- prefix: XXCFG\_ #给cm中的键名加上此前缀

configMapKeyRef:

name: cm-name-tz #使用的configmap名称

optional: false #此cm里的键“是否可选”，false表示为必选

#

# kubectl exec nginx-pod-cm-tz printenv #查看pod里的环境变量

**③创建Secret**

secret资源是区分命名空间的

**★命令行方式创建secret**

创建账号密码验证secret

# kubectl create secret generic database-auth --from-literal=username=root --from-literal=password=passwd123

创建存储于某文件的认证secret

# kubectl create secret generic myssh-key-secret --from-file=ssh-privatekey=${HOME}/.ssh/id\_rsa \

--from-file=ssh-publickey=${HOME}/.ssh/id\_rsa.pub

创建ssl证书secret

# kubectl create secret tls cof-lee.com-ssl --key=./cof-lee.com.key --cert=./cof-lee.com.crt

# kubectl get secret

NAME TYPE DATA AGE

cof-lee.com-ssl kubernetes.io/tls 2 7s

database-auth Opaque 2 10m

myssh-key-secret Opaque 2 3m43s

# kubectl get secret database-auth -oyaml #secret的文本信息均以base64编码存储

apiVersion: v1

data:

password: cGFzc3dkMTIz #base64编码

username: cm9vdA== #base64编码

kind: Secret

metadata:

creationTimestamp: "2023-12-05T21:19:48Z"

name: database-auth

namespace: default

resourceVersion: "1167"

uid: 38790a7e-30ee-4b75-8132-18bca66ca512

type: Opaque

**★基于清单文件创建secret**

# cat > database-auth2.yaml <<EOF

apiVersion: v1

kind: Secret

metadata:

name: database-auth2

namespace: default

stringData: #可基于明文方式提供键值对信息，如果直接写data: 则下面的信息值得事先转为base64

username: root

password: passwd123

type: Opaque

EOF

# kubectl apply -f database-auth2.yaml

**④pod使用secret**

# vi pod-nginx-use-secret.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-nginx-use-secret #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: lbvalue-nginx-use-secret

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: nginx-v1-19-5 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com/k8s/nginx:1.21.6 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

volumeMounts: #pod容器要使用的存储卷声明

- name: cert #根据此名称去寻找对应的volume存储卷

mountPath: /etc/nginx/ssl/ #容器里的挂载点

readOnly: true #只读

volumes:  **#存储卷定义，与containers同级**

- name: cert #存储卷名称

secret:

secretName: cof-lee.com-ssl #指定要使用的secret

restartPolicy: Always #pod重启策略，默认Always

# kubectl apply -f pod-nginx-use-secret.yml

# kubectl exec pod-nginx-use-secret -- ls -lh /etc/nginx/ssl

lrwxrwxrwx 1 root root 14 Dec 5 22:20 tls.crt -> ..data/tls.crt

lrwxrwxrwx 1 root root 14 Dec 5 22:20 tls.key -> ..data/tls.key

**⑤创建imagePullSecret**

imagePullSecret资源保存了docker-registry的账号密码等信息，可在pod下载镜像时使用

命令行方式创建：

# kubectl create secret docker-registry mylocal-registry1 --docker-server=cof-lee.com \

--docker-username=user1 --docker-password=passWdxx123 --docker-email=user1@cof-lee.com

**⑥使用imagePullSecret**

# cat > pod-busybox-test.yml <<EOF

apiVersion: v1

kind: Pod

metadata:

name: pod-busybox-test #pod名，这个名称可带小数点

spec:

imagePullSecrets:

- name: mylocal-registry1

containers: #在spec次级，定义一组容器，-表示数组

- name: busybox #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com/tools/busybox:latest #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

command: [ "/bin/sh", "-ce", "sleep 180" ] #容器入口程序及参数

restartPolicy: Never #重启策略，默认Always

EOF

# kubectl apply -f pod-busybox-test.yml

**④创建imagePullSecret**

**★第11章、RBAC权限配置**

k8s的账号有2类，UserAccount用户账号和ServiceAccount服务账号。

k8s中不直接保存 UserAccount用户账号，在k8s系统里看不到，服务账号需要绑定一个命名空间。

ServiceAccount服务账号是让pod内的容器进程访问其他服务时提供身份认证信息的账户。每个pod都只有一个服务账户，若创建pod时未指定要使用的服务账号，则k8s的准入控制器ServiceAccount会为此pod自动附相应namespace的默认服务账号，默认服务账号名为default

# kubectl get serviceaccount #查看默认命名空间的默认服务账号

NAME SECRETS AGE

default 1 30d

# kubectl describe serviceaccount default

Name: default

Namespace: default

Labels: <none>

Annotations: <none>

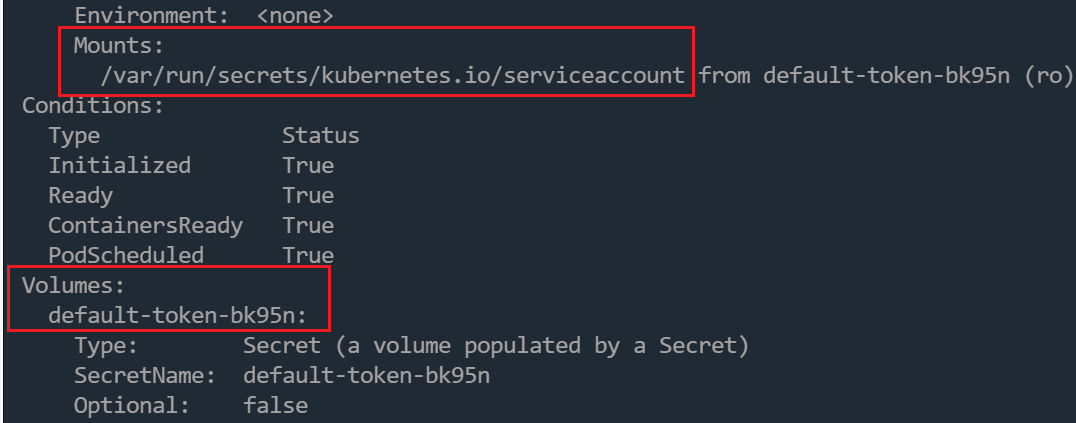
Image pull secrets: <none>

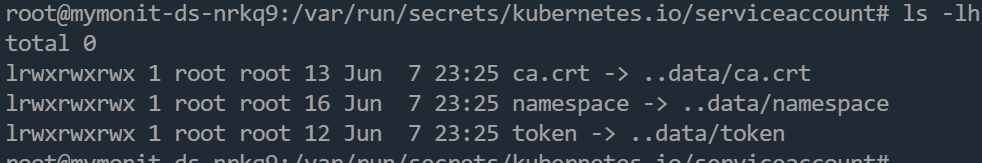
Mountable secrets: default-token-bk95n

Tokens: default-token-bk95n

Events: <none>

#以下是某容器默认挂载的一个存储卷，里面一般有3个文件





**①创建用户账号**

# cd /etc/kubernetes/pki

# openssl genrsa -out kube-user01.key 2048 #创建用户的私钥

# openssl req -new -key kube-user01.key -out kube-user01.csr \

-subj "/CN=kube-user01/O=kubeusers" #创建证书申请文件

# openssl x509 -req -in kube-user01.csr -CA ca.crt -CAkey ca.key -CAcreateserial \

-days 3600 -out kube-user01.crt #使用ca签名，创建用户证书

#以默认管理员 为新建的kube-user01设定kube-conifg配置文件，配置结果默认保存在当前系统用户的.kube/config文件中

# kubectl config set-credentials kube-user01 --embed-certs=true \

--client-certificate=/etc/kubernetes/pki/kube-user01.crt \

--client-key=/etc/kubernetes/pki/kube-user01.key

**#创建context上下文**

# kubectl config set-context kube-user01@kubernetes --cluster=kubernetes --user=kube-user01

# kubectl config use-context kube-user01@kubernetes #切换上下文（指定使用kube-user01用户去访问k8s集群，此用户目前没有访问集群的权限

# kubectl config use-context kubernetes-admin@kubernetes #切换回管理员用户

# kubectl --context=kube-user01@kubernetes get pods #临时使用某用户上下文

**②创建服务账号ServiceAccount**

# cat > sa-test.yaml <<EOF

apiVersion: v1

kind: ServiceAccount

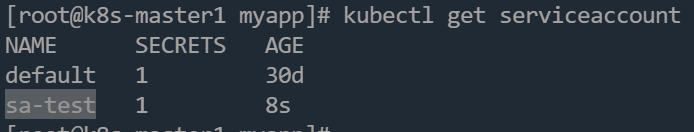
metadata:

name: sa-test

namespace: default

EOF

# kubectl apply -f sa-test.yaml



**★服务账号关联imagePullSecret**

# cat > sa-test.yaml <<EOF

apiVersion: v1

kind: ServiceAccount

metadata:

name: sa-test

namespace: default

imagePullSecrets:

- name: docker-user01-secret #此Secret资源为docker-registry类型的imagePullSecret

EOF

**③RBAC基于角色的访问控制**

RBAC（Role-Based Access Control）基于角色的访问控制，它将权限授予角色Role之上，然后将角色绑定到某用户或用户组上，这样用户就有了所绑定角色的权限

**k8s的RBAC的角色有2种，**

|  |  |
| --- | --- |
| Role | role作用于名称空间，有命名空间之分  将Role绑定到某用户(组)需要使用RoleBinding资源 |
| ClusterRole | ClusterRole作用于集群及所有名称空间，无命名空间之分，（集群级别的资源有nodes，非资源类型的端点）。将ClusterRole绑定到某用户(组)需要使用ClusterRoleBinding资源 |

**★创建Role**

# cat > role-pods-reader.yaml <<EOF

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

name: role-pods-reader #role名称

namespace: default

rules:

- apiGroups: #api组名称

- "" # "" 表示核心组

resources: #目标资源

- pods

- pods/log

verbs: #对目标资源的操作权限

- get

- list

- watch

EOF

# kubectl apply -f role-pods-reader.yaml

**★创建RoleBinding**

# cat > rolebinding-pods-reader.yaml <<EOF

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: rolebinding-pods-reader #RoleBinding的名称

namespace: default #RoleBinding的命名空间

subjects: #绑定到此对象

- kind: User

name: kube-user01 #绑定到名为kube-user01的User

apiGroup: rbac.authorization.k8s.io

roleRef: #将此对象绑定到subjects

kind: Role

name: role-pods-reader #将名为pods-reader的Role绑定到上面的subjects

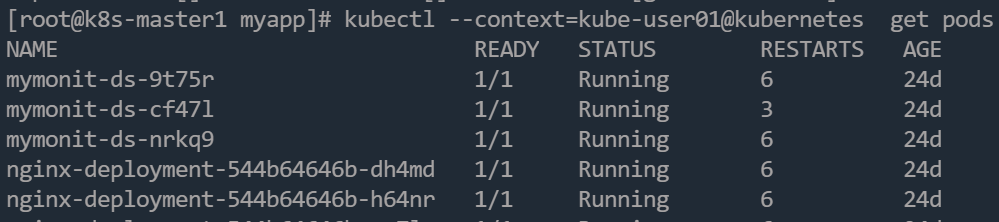
apiGroup: rbac.authorization.k8s.io

EOF

# kubectl apply -f rolebinding-pods-reader.yaml

然后kube-user01用户就有了读取default名称空间中的pods资源的权限

# kubectl --context=kube-user01@kubernetes get pods #此用户有了获取pods的权限



★同一用户可以通过多个rolebinding绑定多个role

**★创建ClusterRole**

# cat > clusterrole-pods-reader.yaml <<EOF

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: clusterrole-pods-reader # ClusterRole名称

rules:

- apiGroups: #api组名称

- "" # "" 表示核心组

resources: #目标资源

- pods

- pods/log

- nodes

verbs: #对目标资源的操作权限

- get

- list

- watch

EOF

# kubectl apply -f clusterrole-pods-reader.yaml

**★创建ClusterRoleBinding**

# cat > clusterrolebinding-pods-reader.yaml <<EOF

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: clusterrolebinding-pods-reader #ClusterRoleBinding的名称

subjects: #绑定到此对象

- kind: User # User或者ServiceAccount，绑定到名为kube-user01的User

name: kube-user01

apiGroup: rbac.authorization.k8s.io #User所拥有的权限，ServiceAccount为空""

roleRef: #将此对象绑定到subjects

kind: ClusterRole

name: clusterrole-pods-reader #将名为pods-reader-cr的ClusterRole绑定到上面的subjects

apiGroup: rbac.authorization.k8s.io

EOF

# kubectl apply -f clusterrolebinding-pods-reader.yaml

然后kube-user01用户就有了读取非名称空间资源nodes资源的权限

# kubectl --context=kube-user01@kubernetes get nodes

**★聚合型ClusterRole**

k8s从1.9版本开始支持在ClusterRole中引用其他的ClusterRole

# cat > xx-clusterrole.yaml <<EOF

aggregationRule:

clusterRoleSelectors:

- matchLabels:

rbac.authorization.k8s.io/aggregate-to-admin: "true"

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: xx

rules: []

EOF

#然后其他要被引用的clusterrole要在metadata中带labels标签：

rbac.authorization.k8s.io/aggregate-to-admin: "true"

**★第12章、K8S Dashboard**

k8s官方提供了一个可视化界面管理k8s集群，开源地址：

https://github.com/kubernetes/dashboard

可在我们的k8s集群里直接应用：

先提前下载以下2个镜像（具体版本根据实际使用的配置yaml文件来决定）

docker.io/kubernetesui/dashboard:v2.6.0

docker.io/kubernetesui/metrics-scraper:v1.0.8

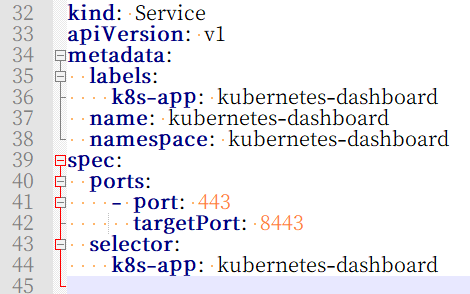
# wget https://raw.githubusercontent.com/kubernetes/dashboard/v2.6.0/aio/deploy/recommended.yaml

# kubectl apply -f recommended.yaml

# wget https://limaofu.github.io/scripts/k8s-dashboard-v2.6.0.yaml

或者先把下载的镜像重新打tag为自己环境的repo源，再修改recommended.yaml文件里的镜像地址为自己环境的repo源，最后再应用

**★dashboard默认监听端口类型为Service**



集群外部无法访问，可增加一个NodePort类型的Service

# vi kubernetes-dashboard-nodeport.yaml

kind: Service

apiVersion: v1

metadata:

labels:

k8s-app: kubernetes-dashboard

name: kubernetes-dashboard-nodeport

namespace: kubernetes-dashboard

spec:

type: NodePort

ports:

- port: 443

targetPort: 8443

nodePort: 30081

selector:

k8s-app: kubernetes-dashboard

#

# kubectl apply -f kubernetes-dashboard-nodeport.yaml

# kubernetes-dashboard默认的服务账号权限有限，如果要访问整个集群的资源，我们可以再创建一个ServiceAccount，并绑定cluster-admin的ClusterRole角色

# vi sa-k8s-dashboard.yaml #内容如下

apiVersion: v1

kind: ServiceAccount

metadata:

name: sa-k8s-dashboard

namespace: default

# kubectl apply -f sa-k8s-dashboard.yaml

# vi admin-clusterrolebinding-k8s-dashboard.yaml #内容如下

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: admin-clusterrolebinding-k8s-dashboard

subjects:

- kind: ServiceAccount

name: sa-k8s-dashboard

namespace: default

apiGroup: ""

roleRef:

kind: ClusterRole

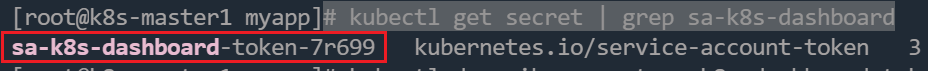
name: cluster-admin

apiGroup: ""

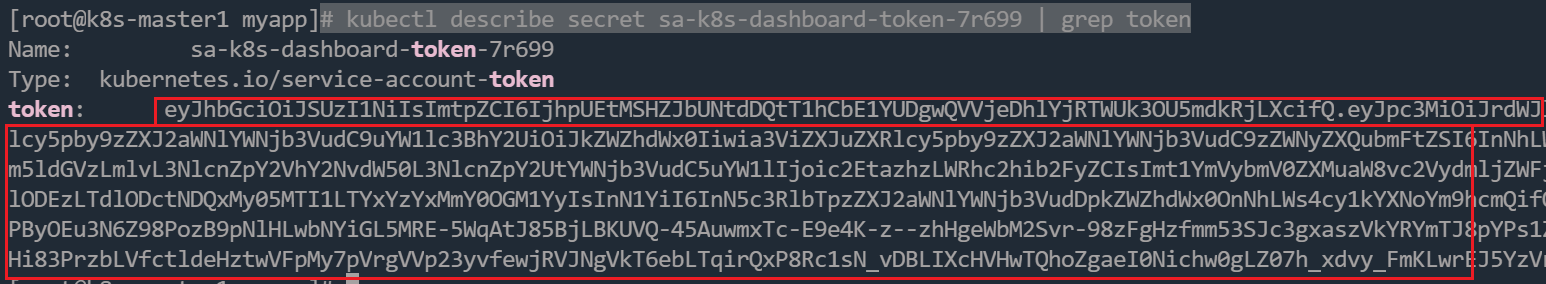
# kubectl apply -f admin-clusterrolebinding-k8s-dashboard.yaml

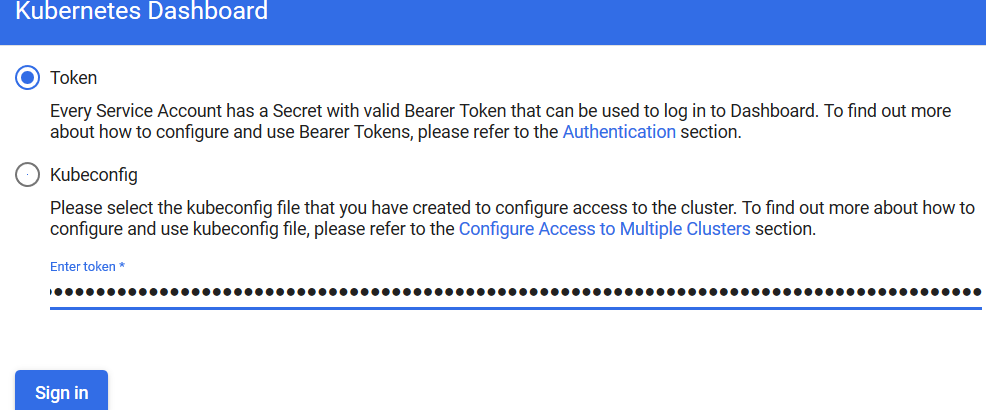
# 最后获取sa-k8s-dashboard服务账号的token，就可去访问k8s-dashboard界面了

# kubectl get secret | grep sa-k8s-dashboard



# kubectl describe secret sa-k8s-dashboard-token-7r699 | grep token





# 或者使用kubeconfig文件登录，先创建sa-k8s-dashboard服务账号的config认证文件

#获取集群初始化信息

kubectl config set-cluster kubernetes --embed-certs=true \

--server="https://10.99.1.54:6443" \

--certificate-authority=/etc/kubernetes/pki/ca.crt \

--kubeconfig=./sa-k8s-dashboard.config

#获取sa-k8s-dashboard服务账号的token并放入config文件里

DASHBOARD\_SECRET=$(kubectl get secret | grep sa-k8s-dashboard | awk '{print $1}')

DASHBOARD\_TOKEN=$(kubectl get secret ${DASHBOARD\_SECRET} -o jsonpath={.data.token} | base64 -d)

kubectl config set-credentials sa-k8s-dashboard --token=${DASHBOARD\_TOKEN} \

--user=sa-k8s-dashboard --kubeconfig=./sa-k8s-dashboard.config

#设置context上下文

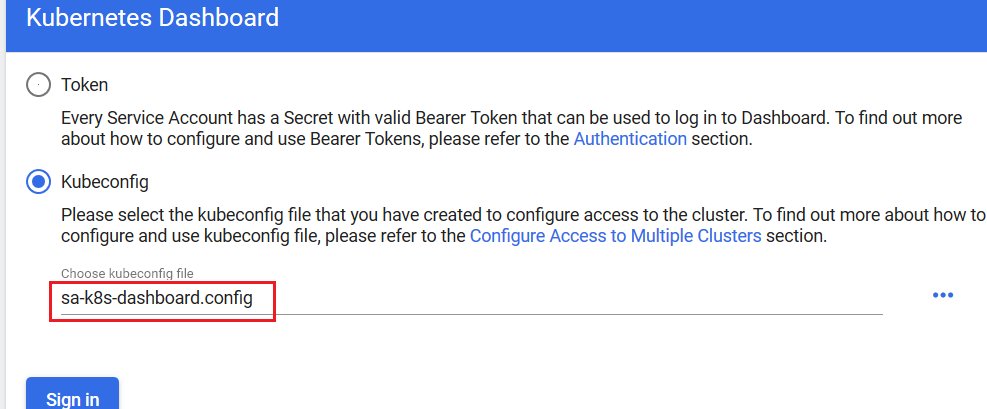
kubectl config set-context sa-k8s-dashboard --cluster=kubernetes \

--user=sa-k8s-dashboard --kubeconfig=./sa-k8s-dashboard.config

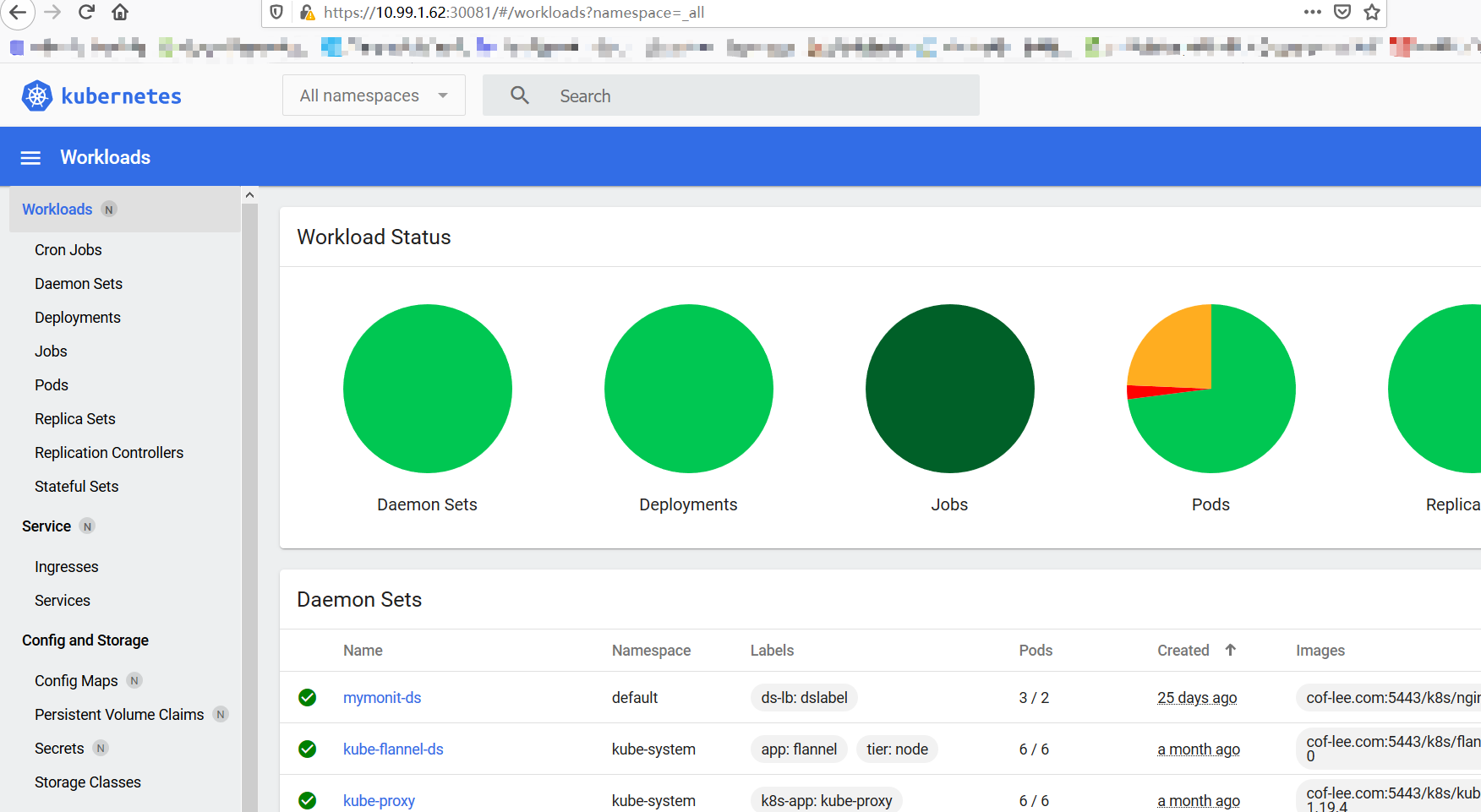
kubectl config use-context sa-k8s-dashboard --cluster=kubernetes \

--user=sa-k8s-dashboard --kubeconfig=./sa-k8s-dashboard.config

#最后下载此sa-k8s-dashboard.config文件去登录dashboard



kubernetes-dashboard界面：



**★第13章、pod资源调度**

**①节点亲和性调度**

节点亲和性调度是指 根据某些规则确定要创建的pod在哪个node结点上运行

**★节点硬亲和性调度**

# vi node-affinity-test1.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: node-affinity-test1 #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: node-affinity-test1

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: node-affinity-test1 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/docker/busybox:latest #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

command: [ "/bin/sh", "-ce", "/bin/ping -c 3 10.99.1.248" ]

resources:

requests: #资源需求

memory: "256Mi"

cpu: "100m"

limits: #资源限制

memory: "256Mi"

cpu: "100m" #1核cpu为 1000m

affinity: #亲和性设置

nodeAffinity: #结点亲和性

requiredDuringSchedulingIgnoredDuringExecution: #硬亲和性

nodeSelectorTerms: #基于结点标签进行匹配

- matchExpressions:

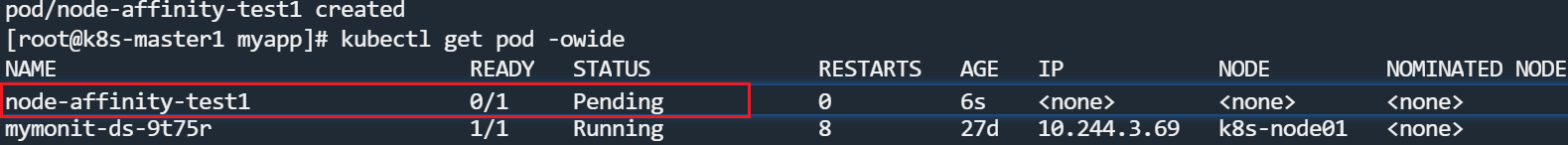
- key: disktype #匹配标签名为 disktype 值为 ssd 的node节点，多个-key:之间是 与 关系

operator: In

values: ["ssd"]

restartPolicy: Never #重启策略设置为Never，默认Always，根据实际需求设置

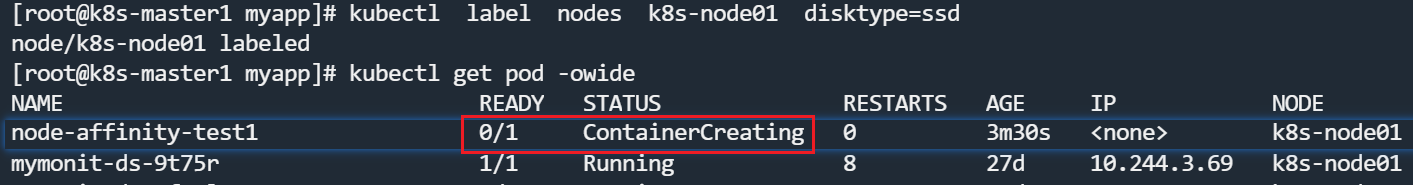
# kubectl apply -f node-affinity-test1.yml



刚刚创建的pod处于Pending状态，因为目前的node结点没有一个是打了disktype=ssd的标签，可找一台node打上此标签再查看此pod状态

# kubectl label nodes k8s-node01 disktype=ssd #给指定的node结点打标签

# kubectl get nodes -L disktype #查看所有结点，-L显示disktype属性列



上图可见刚刚的pod已在创建了，调度到k8s-node01上了

**★节点软亲和性调度**

节点软亲和性表示想让创建的pod优先调度到匹配的node上，如果没有匹配的node，也可调度到其他的node上运行

# vi node-affinity-test2.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: node-affinity-test2 #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: node-affinity-test2

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: node-affinity-test2 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/docker/busybox:latest #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

command: [ "/bin/sh", "-ce", "/bin/ping -c 3 10.99.1.248" ]

resources:

requests: #资源需求

memory: "256Mi"

cpu: "100m"

limits: #资源限制

memory: "256Mi"

cpu: "100m" #1核cpu为 1000m

affinity: #亲和性设置

nodeAffinity: #结点亲和性

preferredDuringSchedulingIgnoredDuringExecution: #软亲和性

- weight: 60 #取值1~100，越大越优先

preference:

matchExpressions:

- {key: disktype, operator: In, values: ["nvme"]} #匹配标签名为disktype值为nvme的node

- weight: 30

preference:

matchExpressions:

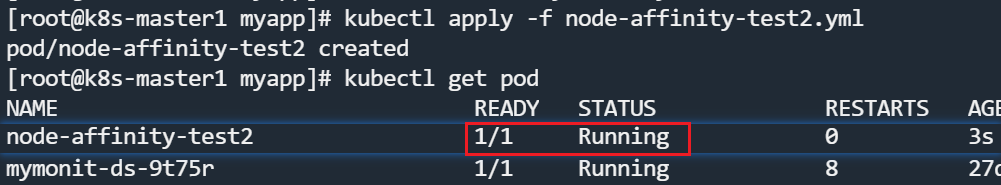
- {key: disktype, operator: In, values: ["xxx"]} #此key的值为xxx

- {key: disktype, operator: NotIn, values: ["hdd"]} #此key的值不为hdd

- {key: highio, operator: Exists, values: []} #存在此key即可

restartPolicy: Never

# kubectl apply -f node-affinity-test2.yml



#没有匹配的node也可调度

★xxxDuringSchedulingIgnoredDuringExecution中的IgnoredDuringExecution表示此pod成功调度到某node运行后，当所在node的标签改变了而不再符合此亲和性匹配规则时，调度器不会将此pod从当前node移出

**②pod亲和性调度**

pod亲和性调度表示要创建的pod要求和某些已运行的pod在同一node上（亲和性），或者不要与某些pod在同一node上（反亲和性）

**★pod硬亲和性调度**

# vi pod-affinity-test1.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-affinity-test1 #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: pod-affinity-test1

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: pod-affinity-test1 #容器名，小写字母开头，后可接数字-减号

image: cof-lee.com:5443/docker/busybox:latest #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

command: [ "/bin/sh", "-ce", "/bin/ping -c 3 10.99.1.248" ]

resources:

requests: #资源需求

memory: "256Mi"

cpu: "100m"

limits: #资源限制

memory: "256Mi"

cpu: "100m" #1核cpu为 1000m

affinity: #亲和性设置

podAffinity: #结点亲和性

requiredDuringSchedulingIgnoredDuringExecution: #硬亲和性

- labelSelector:

matchExpressions:

- {key: lbname, operator: In, values: ["node-affinity-test1"]} #此pod要求为running状态

topologyKey: kubernetes.io/hostname

restartPolicy: Never

#

# kubectl apply -f pod-affinity-test1.yml

**★pod软亲和性调度**

# vi pod-affinity-test2.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-affinity-test2 #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: pod-affinity-test2

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: pod-affinity-test2 #容器名，小写字母开头，后可接数字-减号，不可带小数点

image: cof-lee.com:5443/docker/busybox:latest #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

command: [ "/bin/sh", "-ce", "/bin/ping -c 3 10.99.1.248" ]

resources:

requests: #资源需求

memory: "256Mi"

cpu: "100m"

limits: #资源限制

memory: "256Mi"

cpu: "100m" #1核cpu为 1000m

affinity: #亲和性设置

podAffinity: #结点亲和性

preferredDuringSchedulingIgnoredDuringExecution: #软亲和性

- weight: 70

podAffinityTerm:

labelSelector:

matchExpressions:

- {key: lbname, operator: In, values: ["node-affinity-test1"]}

topologyKey: kubernetes.io/hostname

- weight: 30

podAffinityTerm:

labelSelector:

matchExpressions:

- {key: lbname, operator: In, values: ["node-affinity-test2"]}

topologyKey: kubernetes.io/hostname

restartPolicy: Never #重启策略，默认Always

#所有的labelSelector规则都匹配不上，则也可调度（前提是有可用结点）

# kubectl apply -f pod-affinity-test2.yml

**★pod反亲和性调度**

把上面2例子的podAffinity替换为podAntiAffinity即可，反亲和性一般用于分散同一类应用（同一label）的pod资源

# vi dep-anti-affinity-test1.yml #内容如下

apiVersion: apps/v1

kind: Deployment

metadata:

name: dep-anti-affinity-test1 #deploy名，这个名称可带小数点

spec:

replicas: 3

selector:

matchLabels:

lbname: pod-anti-affinity-test1 #匹配pod标签

template:

metadata:

name: pod-anti-affinity-test1 #pod名称

labels: #给pod打标签，便于其他资源对它的选择

lbname: pod-anti-affinity-test1

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: pod-anti-affinity-test1 #容器名，小写字母开头，后可接数字-减号

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "100m"

limits: #资源限制

memory: "256Mi"

cpu: "100m" #1核cpu为 1000m

affinity: #亲和性设置

podAntiAffinity: #结点 反亲和性

requiredDuringSchedulingIgnoredDuringExecution: #硬 反亲和性

- labelSelector:

matchExpressions:

- {key: lbname, operator: In, values: ["pod-anti-affinity-test1"]} #匹配本pod自己

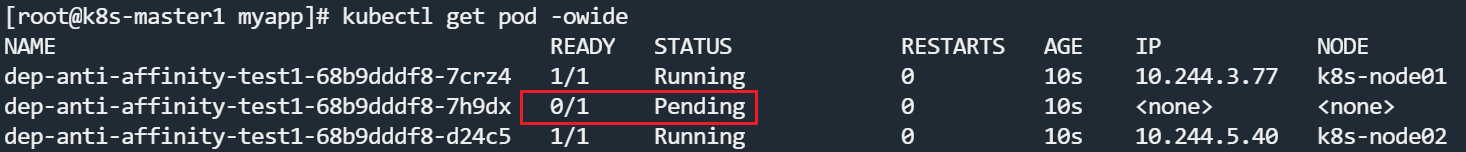
topologyKey: kubernetes.io/hostname

restartPolicy: Always

#

# kubectl apply -f dep-anti-affinity-test1.yml

下图可见要求3个副本数，结果只有2个可用node，导致有一个pod处于Pending状态



**③污点和容忍度**

在某node上（含master结点）打上污点，可以驱逐其上运行的pod

pod上的容忍度如果匹配了某些污点，则就算node上有此污点，也可调度此pod

# kubectl taint node *结点名* keyxx=valuexx:NoSchedule #给结点打上污点

# kubectl taint node *结点名* keyyy:NoExecute #给结点打上污点

# kubectl taint node *结点名* keyyy**-** #去除此key的所有污点，在key名后加 -减号

# kubectl taint node *结点名* keyyy:NoExecute**-** #去除此key的NoExecute效应的污点

污点由三部分组成：key加value加额外的effect说明，value可省略

effect效应说明主要包含以下三种类型：

|  |  |
| --- | --- |
| NoSchedule | 不能容忍此污点的新pod不会调度到此Node上，已有的pod不影响 |
| NoExecute | 不能容忍此污点的新pod不会调度到此Node上，已有的pod被驱逐 |
| PreferNoSchedule | 不能容忍此污点的新pod尽量不调度到此Node上，已有的pod不影响  若无其他结点可调度，则也可调度到此Node上 |

**★master结点默认的污点：**

# kubectl describe node k8s-master1 | grep -i Taints #查看master结点的污点

node-role.kubernetes.io/master:NoSchedule

node-role.kubernetes.io/control-plane:NoSchedule

#master结点默认打上了以上污点，如果想让某pod调度到master结点上，需要设置pod容忍此污点（不同版本的k8s，设置的污点可能不一样）

★k8s v1.28.2不可去除管理节点上的污点：node-role.kubernetes.io/control-plane:NoSchedule-

去除后，整个集群管理服务都挂了，原因暂未知

# kubectl taint node k8s-node02 keyxx=valuexx:NoSchedule #打污点

# vi pod-toleration-test1.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: pod-toleration-test1 #pod名，这个名称可带小数点

labels: #给pod打标签，便于其他资源对它的选择

lbname: pod-toleration-test1

spec:

containers: #在spec次级，定义一组容器，-表示数组

- name: pod-toleration-test1 #容器名，小写字母开头，后可接数字-减号

image: cof-lee.com:5443/k8s/nginx:v1.19.5 #容器使用的镜像

imagePullPolicy: IfNotPresent #镜像获取方式，优先本地，后网络拉取

resources:

requests: #资源需求

memory: "256Mi"

cpu: "100m"

limits: #资源限制

memory: "256Mi"

cpu: "100m" #1核cpu为 1000m

restartPolicy: Never #重启策略，默认Always

tolerations:

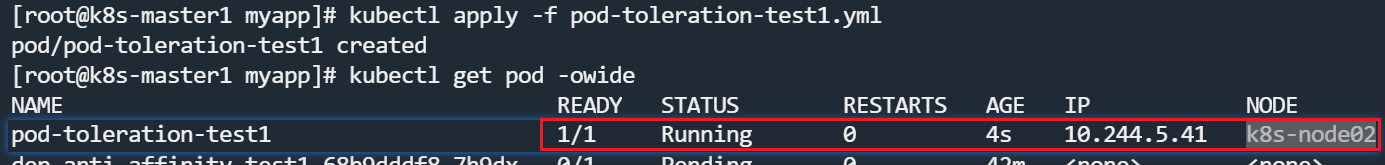
- key: "keyxx"

operator: "Equal" #key,value,effect三者必须都匹配上

value: "valuexx"

effect: "NoSchedule"

# kubectl apply -f pod-toleration-test1.yml



- key: "keyxx"

operator: "Exists" #key,effect两者必须都匹配上

effect: "NoExecute"

tolerationSeconds: 600 #达到600秒后，也会被驱逐

**★第14章、Helm包管理**

helm就是k8s的应用程序包管理器，helm将k8s应用的相关资源配置文件组织为Charts

Charts就是一个helm程序包，

**★安装helm**

**①helm v3**

★Helm v3不再使用tiller，直接使用helm命令即可

helm开源地址： https://github.com/helm/helm

下载压缩包到linux服务器上，并解压，把解压目录下的helm可执行文件复制到/usr/bin/目录下添加可执行权限即可

helm和kubectl一样默认读取$HOME/.kube/config配置文件去获取操作k8s集群的权限

**②helm v2**

helm v2需要使用tiller,运行于k8s集群上，需要使用一个ServiceAccount

# vi tiller-cluster-admin.yaml #内容如下

apiVersion: v1

kind: ServiceAccount

metadata:

name: tiller

namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: tiller-clusterrolebinding

subjects:

- kind: ServiceAccount

name: tiller

namespace: kube-system

apiGroup: ""

roleRef:

kind: ClusterRole

name: cluster-admin

apiGroup: rbac.authorization.k8s.io

#

# kubectl apply -f tiller-cluster-admin.yaml

# helm init --service-account tiller --skip-refresh #初始化tiller server环境

**★helm常用命令**

#添加helm软件仓库，仓库名为awx-operator，后面为仓库url，要求仓库地址能访问才添加成功

# helm repo add awx-operator https://ansible.github.io/awx-operator/

# helm repo list #列出所有repo仓库

NAME URL

awx-operator https://ansible.github.io/awx-operator/

# helm repo update #更新仓库

# helm repo remove awx-operator #删除仓库

# helm pull chatName #下载chart包，默认为压缩包，可用--version=指定版本

# helm install chartName repoName/chartPkgName #安装指定仓库里的chart包

# helm dependency list chartName.tgz #查看chart的依赖包

# helm install -f values\_xxx.yaml chartName ./pathTo/chartPkg #安装离线chart包，-f指定参数文件

# helm upgrade -f values\_yy.yaml chartName ./pathTo/chartPkg #升级chart包版本，-f指定参数文件

# helm install --set key=valuexx chartName ./pathTo/chartPkg #安装时指定参数

# helm delete chartName #取消部署

# helm list -A #查看已发布的chart（运行了的chart），可用-n xxx 查看指定的命名空间

# helm history chartName #查看发布chart的历史版本

# helm history awx-operator -n awx

REVISION UPDATED STATUS CHART APP VERSION DESCRIPTION

1 Thu Oct 19 15:44:46 2023 deployed awx-operator-2.7.0 2.7.0 Install complete

# helm rollback chartName 2 -n xxx #回滚到指定历史版本

**★制作包**

# helm create test-chart #创建一个chart项目目录，默认结构如下：

# tree test-chart/

test-chart/

├── charts

├── Chart.yaml

├── templates

│ ├── deployment.yaml

│ ├── \_helpers.tpl

│ ├── hpa.yaml

│ ├── ingress.yaml

│ ├── NOTES.txt

│ ├── serviceaccount.yaml

│ ├── service.yaml

│ └── tests

│ └── test-connection.yaml

└── values.yaml

# helm package --version 1.0.2 test-chart #将chart项目目录打包为一个tgz压缩文件

Successfully packaged chart and saved it to: /root/test-chart-1.0.2.tgz

**★搭建私有helm仓库**

安装web服务器，创建一个根目录/charts

将所有chart压缩包复制到/charts目录下，并生成库的index文件

# yum install httpd -y

# mkdir /charts

# cp test-chart-1.0.2.tgz /charts #如果有多个包，则都可复制过去

# helm repo index /charts/ --url http://cof-lee.com/charts

# more /charts/index.yaml

apiVersion: v1

entries:

test-chart:

- apiVersion: v2

appVersion: 1.16.0

created: "2023-10-19T16:45:42.589450935+08:00"

description: A Helm chart for Kubernetes

digest: 1eb546ef8d5585f256c8c93c741a8da73867f4eec21e1f9631081969efc8b0d3

name: test-chart

type: application

urls:

- http://cof-lee.com/charts/test-chart-1.0.2.tgz

version: 1.0.2

generated: "2023-10-19T16:45:42.587716651+08:00"

# ln -s /charts /var/www/html/charts

# systemctl --now enable httpd

**#客户端测试**

# helm repo add test-repo http://cof-lee.com/charts

# helm repo list

NAME URL

test-repo http://cof-lee.com/charts

# helm search repo test-chart

NAME CHART VERSION APP VERSION DESCRIPTION

test-repo/test-chart 1.0.2 1.16.0 A Helm chart for Kubernetes

# helm pull test-repo/test-chart #下载包

# ls -lh

-rw-r--r-- 1 root root 3960 Oct 19 16:53 test-chart-1.0.2.tgz

**★上传chart包到harbor（未测试成功）**

首先要安装helm-push插件，在线安装方式：

helm plugin install https://github.com/chartmuseum/helm-push

离线安装方式：

# helm env | grep HELM\_PLUGINS #查看helm环境变量，查找helm插件路径

HELM\_PLUGINS="/root/.local/share/helm/plugins"

https://github.com/chartmuseum/helm-push查看版本，并下载到本地，上传到服务器上

wget https://github.com/chartmuseum/helm-push/releases/download/v0.10.4/helm-push\_0.10.4\_linux\_amd64.tar.gz

# mkdir -p /root/.local/share/helm/plugins/helm-push

# tar -xvf helm-push\_0.10.4\_linux\_amd64.tar.gz -C /root/.local/share/helm/plugins/helm-push

# helm plugin list

NAME VERSION DESCRIPTION

cm-push 0.10.4 Push chart package to ChartMuseum

# helm添加harbor的repo

# helm repo add harbor-246 http://cof-lee.com/chartrepo/test-chart --username=admin --password=xx

#推送在压缩包到harbor仓库

# helm cm-push test-chart-1.0.2.tgz harbor-246 --host=cof-lee.com --username=admin --password=xx

#推送目录到harbor仓库

# helm cm-push test-chart/ harbor-246 --host=cof-lee.com --username=admin --password=xxx

#推送目录及指定的版本

# helm cm-push test-chart/ --version="1.2.2" harbor-246 --host=cof-lee.com --username=admin --password=xx

**★第15章、prometheus监控系统**

**★Prometheus简介**

Prometheus是一款基于时序数据库的开源监控告警系统，Prometheus的基本原理是通过HTTP协议周期性抓取被监控组件的状态，任意组件只要提供对应的HTTP接口就可以接入监控。不需要任何SDK或者其他的集成过程。这样做非常适合做虚拟化环境监控系统，比如VM、Docker、Kubernetes等。输出被监控组件信息的HTTP接口被叫做exporter

Promethus特点：

支持多维数据模型：由度量名和键值对组成的时间序列数据

内置时间序列数据库TSDB

支持PromQL查询语言，可以完成非常复杂的查询和分析，对图表展示和告警非常有意义

支持HTTP的Pull方式采集时间序列数据

支持PushGateway采集瞬时任务的数据

支持服务发现和静态配置两种方式发现目标

支持接入Grafana

Prometheus Server三个组件

①Retrieval: 数据采集，获取被监控对象的相关数据

②TSDB: 时间序列数据库（Time Series Database）用来处理时间序列数据的数据库，并且数据中的数组是由时间进行索引的。具备以下特点：

大多数数据都是按时间顺序写入，很少涉及修改数据

删除操作都是删除一段时间的数据，而不涉及到删除无规律数据

读操作一般都是升序或者降序

③HTTP Server: 为告警和绘图提供查询接口

Promethus通过PromQL语句查询指标信息，并在页面展示。虽然Prometheus自带UI界面，但是大多数是使用Grafana进行绘图展示。另外也可通过API接口来给第三方软件提供监控指标。alertmanager组件负责邮件报警。

**★部署Prometheus**

部署清单文件下载： https://github.com/prometheus-operator/kube-prometheus

Operator部署器是根据已经编写好的yaml文件，可以将prometheus server，alertmanager，grafana，node-exporter等组件部署在k8s集群里。根据k8s版本选择Operator版本，版本适配关系查看github主页的Compatibility部分：

kube-prometheus版本 适配的k8s版本

release-0.8 1.20 1.21

release-0.9 1.21 1.22

release-0.10 1.22 1.23

release-0.11 1.23 1.24

release-0.12 1.24 1.25

release-0.13 1.26 1.27 1.28

本小节以k8s v1.23版本为例，部署release-0.10版本的prometheus，先下载部署清单文件到服务器上

# tar -xf kube-prometheus-0.10.0.tar.gz

# cd kube-prometheus-0.10.0/manifests

# grep -i 'image:' \*.yaml #查看要用到的镜像

alertmanager-alertmanager.yaml: image: quay.io/prometheus/alertmanager:v0.23.0

blackboxExporter-deployment.yaml: image: quay.io/prometheus/blackbox-exporter:v0.19.0

blackboxExporter-deployment.yaml: image: jimmidyson/configmap-reload:v0.5.0

blackboxExporter-deployment.yaml: image: quay.io/brancz/kube-rbac-proxy:v0.11.0

grafana-deployment.yaml: image: grafana/grafana:8.3.3

kubeStateMetrics-deployment.yaml: image: k8s.gcr.io/kube-state-metrics/kube-state-metrics:v2.3.0

kubeStateMetrics-deployment.yaml: image: quay.io/brancz/kube-rbac-proxy:v0.11.0

kubeStateMetrics-deployment.yaml: image: quay.io/brancz/kube-rbac-proxy:v0.11.0

nodeExporter-daemonset.yaml: image: quay.io/prometheus/node-exporter:v1.3.1

nodeExporter-daemonset.yaml: image: quay.io/brancz/kube-rbac-proxy:v0.11.0

prometheusAdapter-deployment.yaml: image: k8s.gcr.io/prometheus-adapter/prometheus-adapter:v0.9.1

prometheusOperator-deployment.yaml: image: quay.io/prometheus-operator/prometheus-operator:v0.53.1

prometheusOperator-deployment.yaml: image: quay.io/brancz/kube-rbac-proxy:v0.11.0

prometheus-prometheus.yaml: image: quay.io/prometheus/prometheus:v2.32.1

# grep -i 'prometheus-config-reloader' \*.yaml #查看要用到的镜像

prometheusOperator-deployment.yaml: \

- --prometheus-config-reloader=quay.io/prometheus-operator/prometheus-config-reloader:v0.53.1

若所有k8s节点都能访问互联网，则直接部署，

若不能，则需要提前下载所需镜像到所有节点/或者指定使用内部镜像仓库

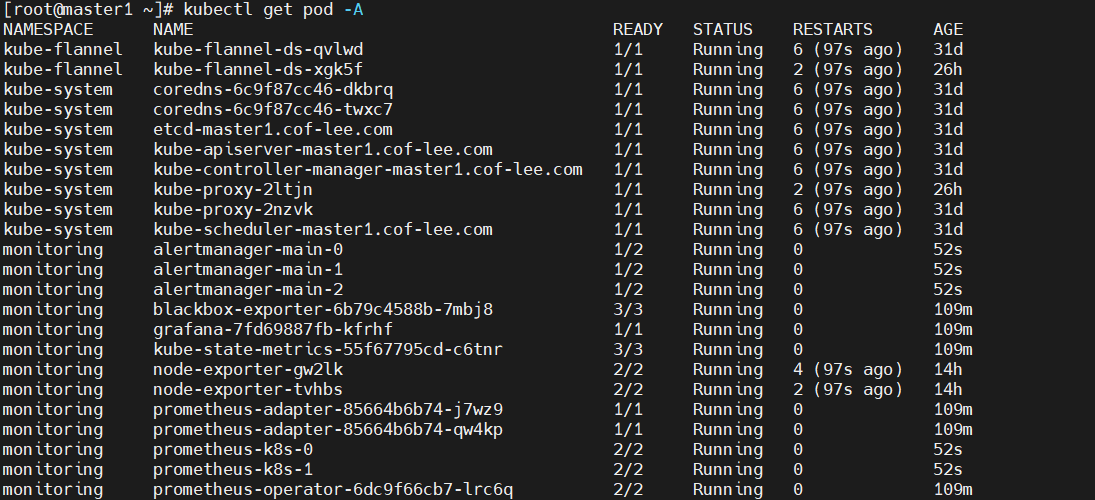
# kubectl apply --server-side -f ./setup/ #初始化设置， k8s v1.22版本开始支持此命令

# kubectl create -f ./setup/ #初始化设置，适用于k8s所有版本，与上面二选一

# kubectl wait --for condition=Established --all CustomResourceDefinition --namespace=monitoring

# kubectl apply -f ./ #正式部署

# kubectl get pod --all-namespaces



**#下载镜像**

docker pull quay.io/prometheus/alertmanager:v0.23.0

docker pull quay.io/prometheus/blackbox-exporter:v0.19.0

docker pull jimmidyson/configmap-reload:v0.5.0

docker pull quay.io/brancz/kube-rbac-proxy:v0.11.0

docker pull grafana/grafana:8.3.3

docker pull quay.io/prometheus/node-exporter:v1.3.1

docker pull quay.io/prometheus-operator/prometheus-operator:v0.53.1

docker pull quay.io/prometheus/prometheus:v2.32.1

docker pull quay.io/prometheus-operator/prometheus-config-reloader:v0.53.1

#以下2个镜像下载不了，

docker pull k8s.gcr.io/kube-state-metrics/kube-state-metrics:v2.3.0

docker pull k8s.gcr.io/prometheus-adapter/prometheus-adapter:v0.9.1

#替换为：

docker pull docker.io/lbbi/kube-state-metrics:v2.3.0

docker pull docker.io/lbbi/prometheus-adapter:v0.9.1

**#导出镜像到文件，复制到所有节点并导入**

mkdir kube-prometheus-0.10.0-all

cd kube-prometheus-0.10.0-all

docker save quay.io/prometheus/alertmanager:v0.23.0 -o alertmanager\_\_v0.23.0.tar

docker save quay.io/prometheus/blackbox-exporter:v0.19.0 -o blackbox-exporter\_\_v0.19.0.tar

docker save jimmidyson/configmap-reload:v0.5.0 -o configmap-reload\_\_v0.5.0.tar

docker save quay.io/brancz/kube-rbac-proxy:v0.11.0 -o kube-rbac-proxy\_\_v0.11.0.tar

docker save grafana/grafana:8.3.3 -o grafana\_\_8.3.3.tar

docker save quay.io/prometheus/node-exporter:v1.3.1 -o node-exporter\_\_v1.3.1.tar

docker save quay.io/prometheus-operator/prometheus-operator:v0.53.1 -o prometheus-operator\_\_v0.53.1.tar

docker save quay.io/prometheus/prometheus:v2.32.1 -o prometheus\_\_v2.32.1.tar

docker save quay.io/prometheus-operator/prometheus-config-reloader:v0.53.1 -o prometheus-config-reloader\_\_v0.53.1.tar

docker tag docker.io/lbbi/kube-state-metrics:v2.3.0 k8s.gcr.io/kube-state-metrics/kube-state-metrics:v2.3.0

docker tag docker.io/lbbi/prometheus-adapter:v0.9.1 k8s.gcr.io/prometheus-adapter/prometheus-adapter:v0.9.1

docker save k8s.gcr.io/kube-state-metrics/kube-state-metrics:v2.3.0 -o kube-state-metrics\_\_v2.3.0.tar

docker save k8s.gcr.io/prometheus-adapter/prometheus-adapter:v0.9.1 -o prometheus-adapter\_\_v0.9.1.tar

scp kube-prometheus-0.10.0-all目录到k8s所有节点

**#所有k8s节点导入镜像**

# cd kube-prometheus-0.10.0-all/

docker load -i alertmanager\_\_v0.23.0.tar

docker load -i blackbox-exporter\_\_v0.19.0.tar

docker load -i configmap-reload\_\_v0.5.0.tar

docker load -i kube-rbac-proxy\_\_v0.11.0.tar

docker load -i grafana\_\_8.3.3.tar

docker load -i node-exporter\_\_v1.3.1.tar

docker load -i prometheus-operator\_\_v0.53.1.tar

docker load -i prometheus\_\_v2.32.1.tar

docker load -i prometheus-config-reloader\_\_v0.53.1.tar

docker load -i kube-state-metrics\_\_v2.3.0.tar

docker load -i prometheus-adapter\_\_v0.9.1.tar

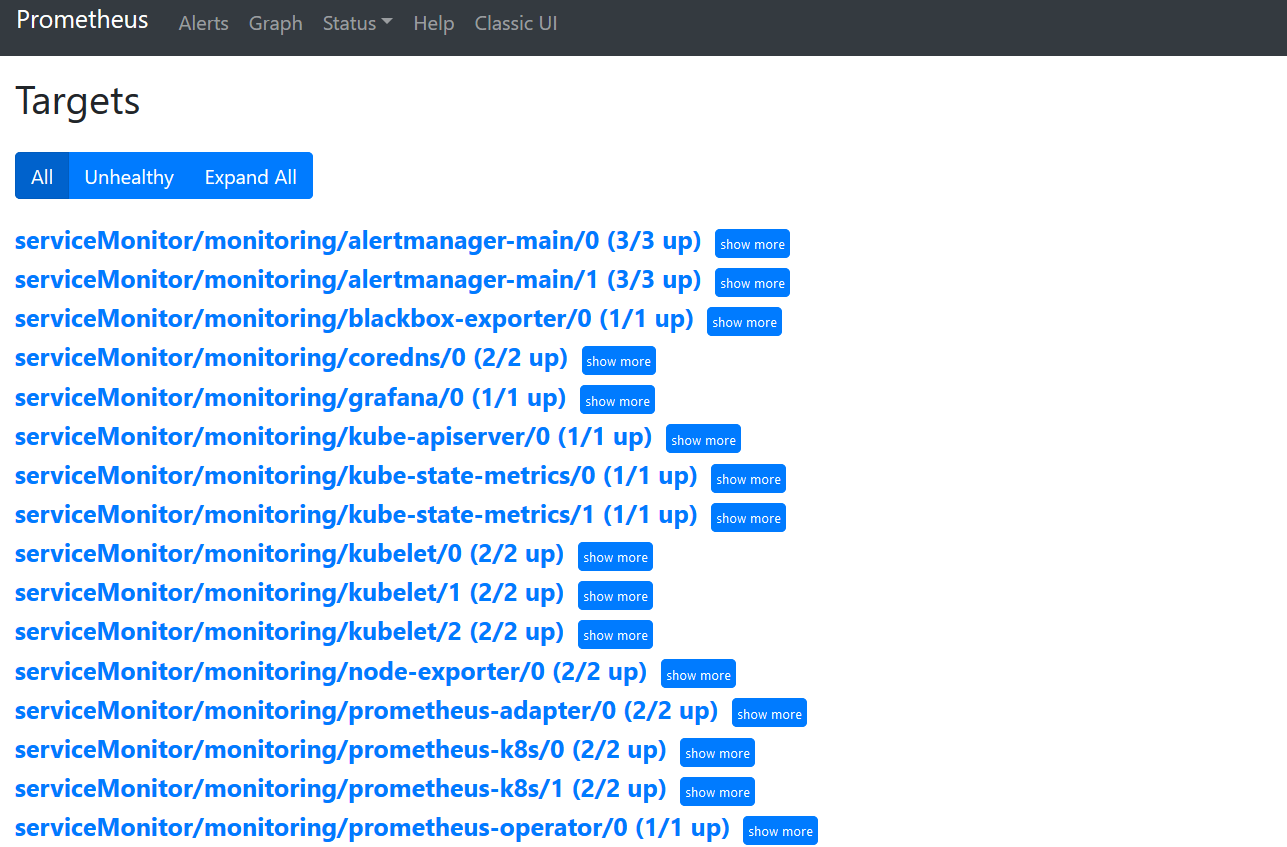
**★暴露prometheus的svc**

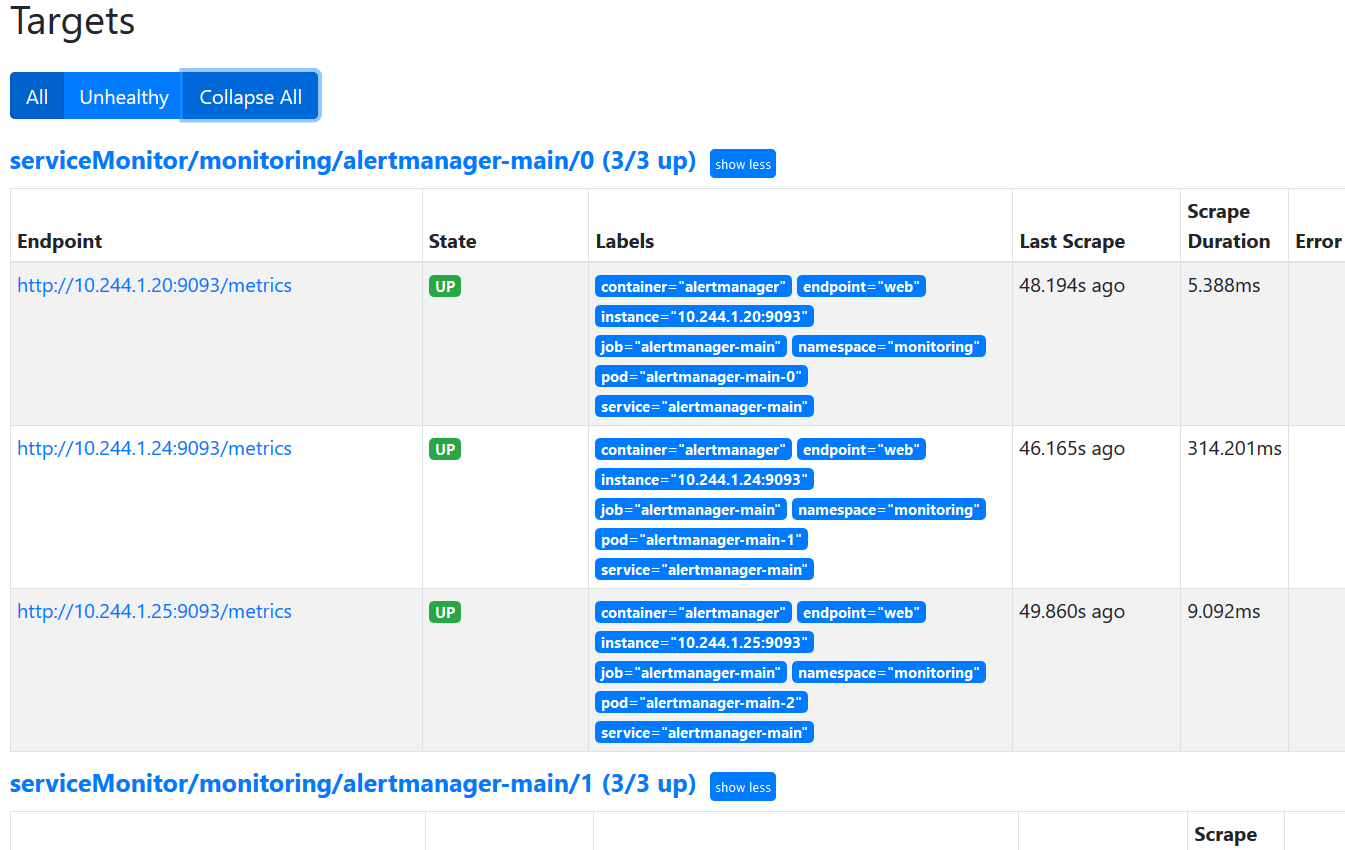
# cat manifests/prometheus-service.yaml #查看prometheus的svc配置情况

# kubectl expose svc prometheus-k8s --type=NodePort --name=prometheus-k8s-nodeport -n monitoring

monitoring prometheus-k8s-nodeport NodePort 10.7.153.25 <none> 9090:30324/TCP,8080:30835/TCP 2s

浏览器访问 http://node-ip:30324/targets





**★暴露grafana的svc**

# cat manifests/grafana-service.yaml #查看grafana的svc配置情况

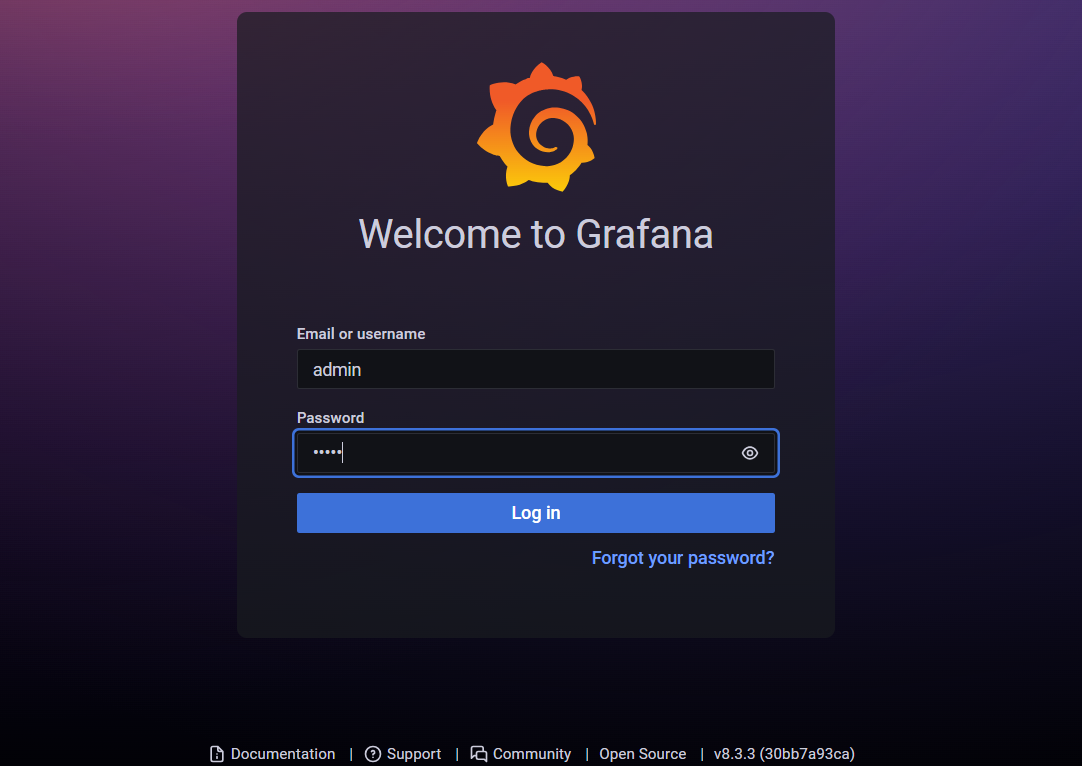
# kubectl expose svc grafana --type=NodePort --name=grafana-nodeport -n monitoring

monitoring grafana-nodeport NodePort 10.7.101.247 <none> 3000:32089/TCP 14h

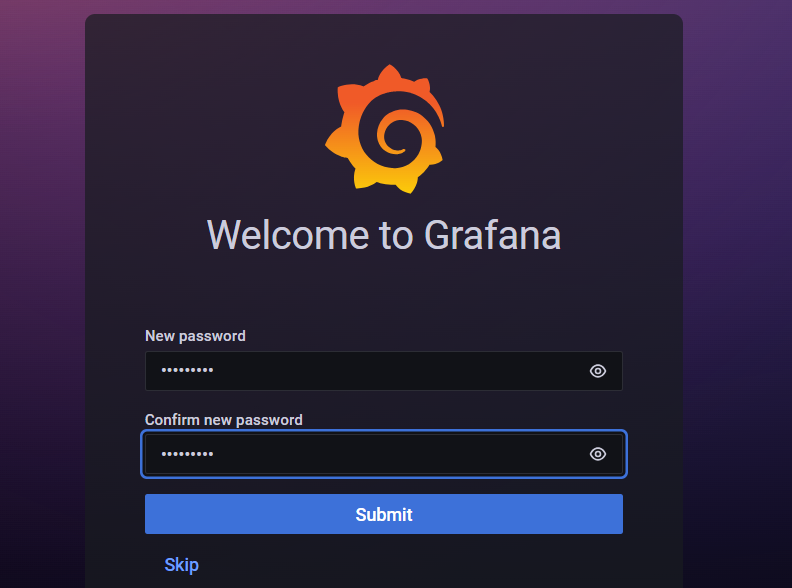
浏览器访问 http://node-ip:32089/

初始默认账号密码为 admin admin

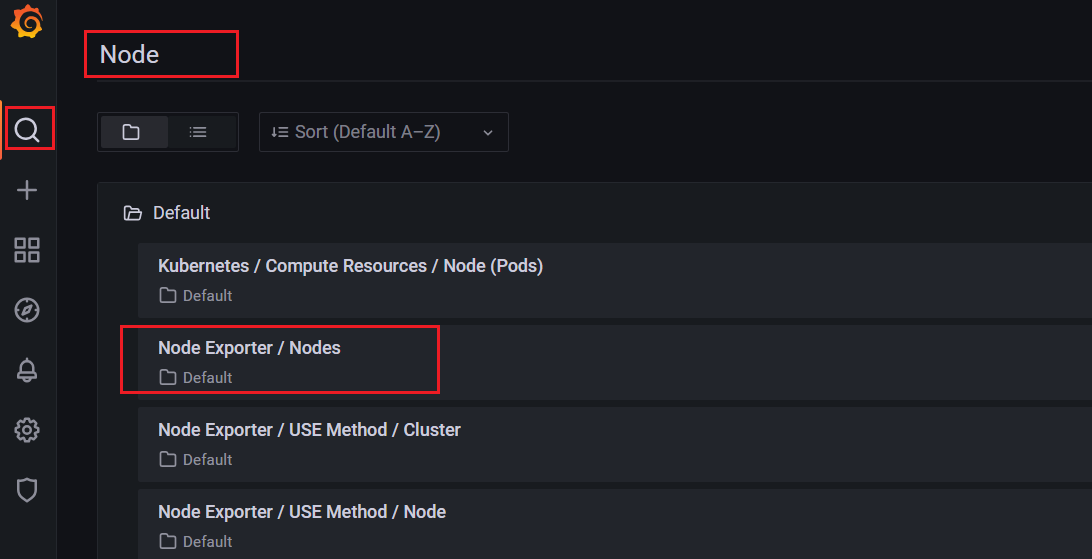
登录后自动弹出改密码窗口，设置新的密码即可



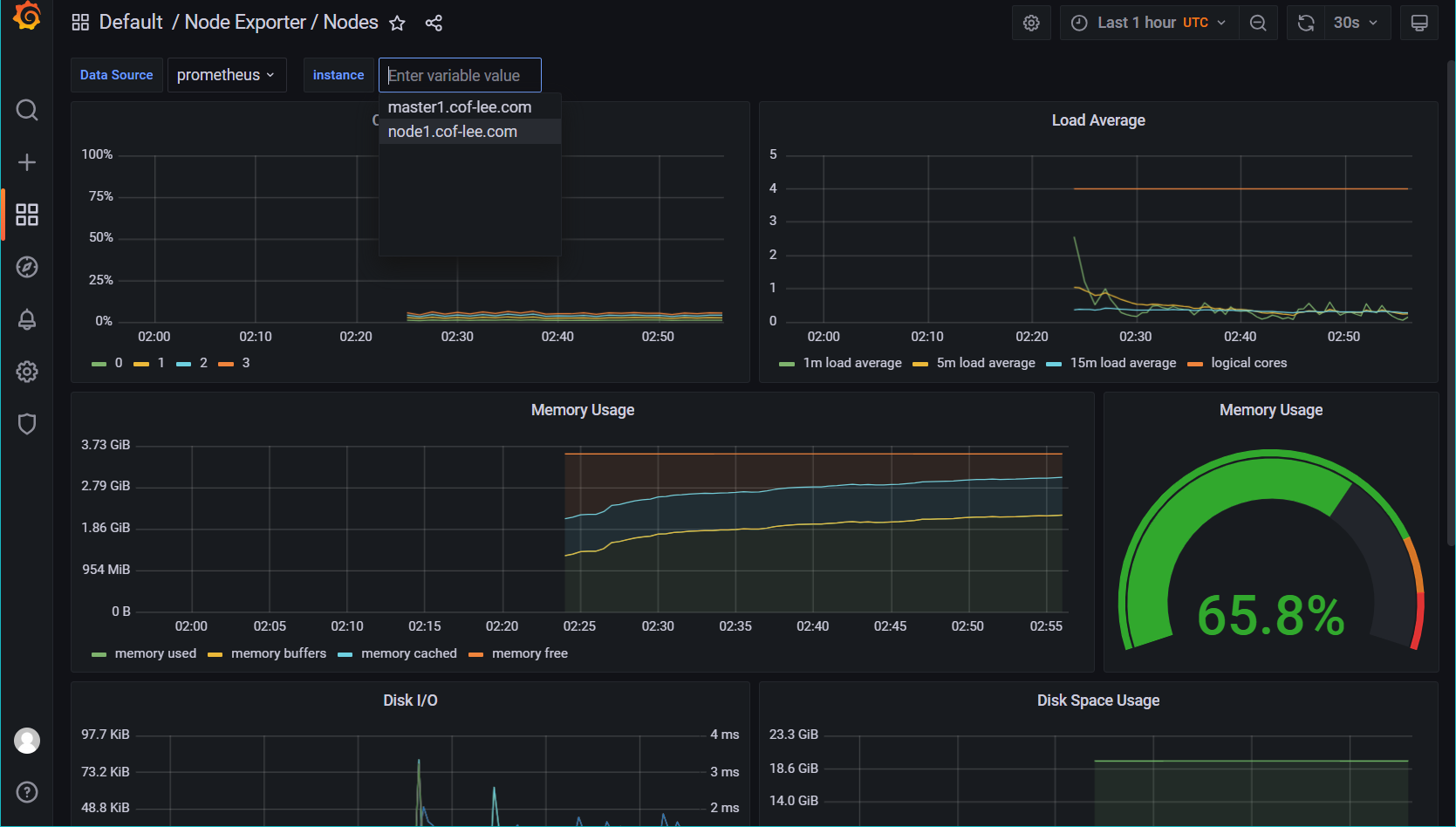
首先使用默认账号admin默认密码admin登录，然后根据提示设置新密码



登录后，到搜索框搜索相关监控目标，比如输入“Node”，点击“Node Exporter / Nodes”



监控信息界面如下：

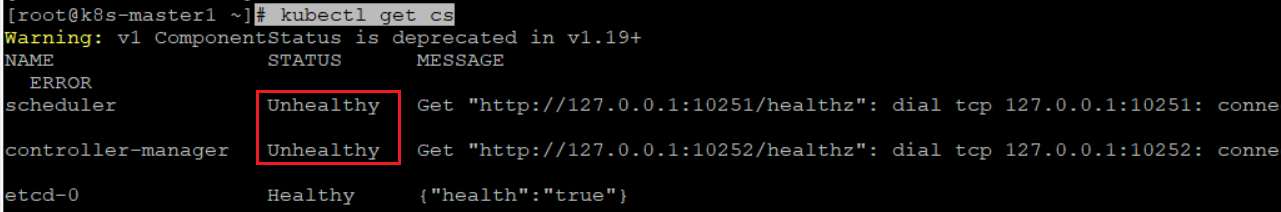


**★第n章、其他**

在

**①检查集群健康状态为Unhealthy**

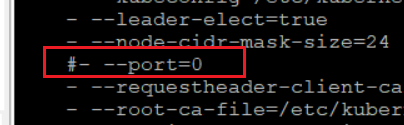
# kubectl get cs



# vi /etc/kubernetes/manifests/kube-controller-manager.yaml

# vi /etc/kubernetes/manifests/kube-scheduler.yaml

#编辑所有master结点里的以上2个文件，把- --port=0这行注释掉



# systemctl restart kubelet #再重启kubelet服务

# kubectl get cs

