### 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/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/再创作 本文档，则说明接受以上2个条款。

作者：李茂福

时间：2022-05-08

**章0、K8S集群搭建准备工作**

**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 | 10.99.1.51 |
| k8s-master2 | 10.99.1.52 |
| k8s-master3 | 10.99.1.53 |
| k8s-node01 | 10.99.1.61 |
| k8s-node02 | 10.99.1.62 |

规划Pod网络： 10.244.0.0/16

规划Service网络： 10.7.0.0/16

**③关闭seLinux，**若不会配置selinux，则可关闭

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

# setenforce 0

**④ulimit设置**

# vi /etc/security/limits.conf

\* soft nofile 1000000

\* hard nofile 1000000

\* soft core unlimited

\* hard core unlimited

\* soft memlock unlimited

\* soft memlock unlimited

保存，退出

要重启系统才生效

**⑤关闭swap**

# vi /etc/sysctl.conf #文档末尾添加以下一行

vm.swappiness = 0

保存，退出

# 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

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

**⑦设置网络参数**

# vi /etc/sysctl.conf #文档末尾添加以下4行

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

保存，退出

# 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 node ip网段以及pod ip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

# iptables -I DOCKER-USER 1 -s 10.99.1.0/24 -j ACCEPT

# iptables -I DOCKER-USER 1 -s 10.244.0.0/16 -j ACCEPT

# iptables -I DOCKER-USER 1 -s 10.7.0.0/16 -j ACCEPT

#iptables-save //保存配置

**⑨安装docker-ce**并设置为systemd去管理服务

# yum install docker-ce docker-ce-cli containerd.io -y



# systemctl enable docker

# systemctl start docker

# docker info

配置docker服务使用systemd去管理

# 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" ]

}

保存

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

★docker从1.13版本开始自动将iptables的Forward策略改为drop，改回accept

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

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

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

保存，（还加了DOCKER-USER这个forward，默认全部return，导致不通，也得放开，具体的看下iptables规则）

vi /etc/rc.d/rc.local

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

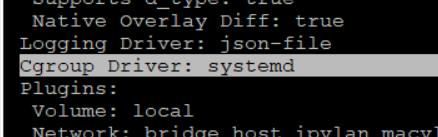
/usr/sbin/iptables -I DOCKER-ISOLATION-STAGE-1 1 -s 0.0.0.0/0 -j ACCEPT

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

# systemctl daemon-reload

# systemctl restart docker

再查看 docker info



**⑩加载ipvs模块**

# vi /etc/rc.d/rc.local #添加以下几行

modprobe -- ip\_vs

modprobe -- ip\_vs\_sh

modprobe -- ip\_vs\_rr

modprobe -- ip\_vs\_wrr

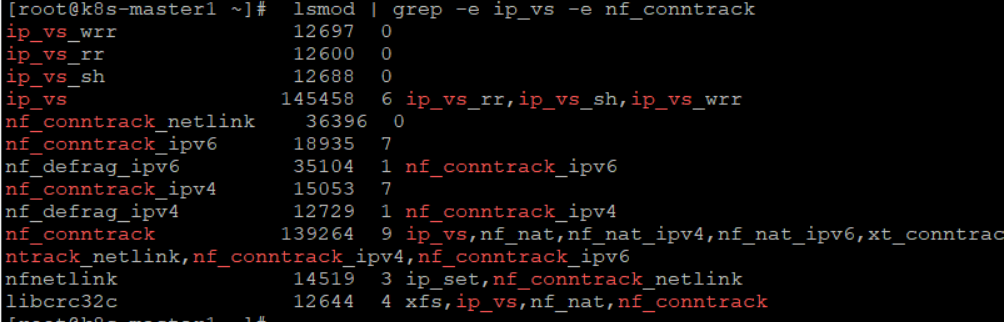
modprobe -- nf\_conntrack\_ipv4

保存

# chmod +x /etc/rc.d/rc.local #添加可执行权限

★最后重启服务器 init 6

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



**章一、master结点安装k8s组件**

k8s的组件有部分是以二进制文件形式安装在服务器上的，如kubectl，kubeadm，

kubelet，kubernets-cni之类的，k8s官方已打包为rpm软件包，可配置国内的yum源去安装；而其他的组件则做成了docker镜像包了，先安装二进制rpm软件包，再docker pull下载当前k8s版本需要的镜像包

**①安装k8s二进制组件**

# vi /etc/yum.repos.d/k8s-ali.repo

[k8s-ali]

name=k8s-ali

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

enabled=1

gpgcheck=0

保存。或者使用google的源

# vi /etc/yum.repos.d/k8s-google.repo

[k8s-google]

name=k8s-google

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

enabled=1

gpgcheck=0

保存

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

# systemctl enable kubelet

# systemctl start kubelet

**②k8s集群初始化**

# kubeadm version #先查看k8s版本

看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

可以先想办法下载以上7个镜像，再传到k8s服务器上，再docker load

★**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**章四**）

# 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

可使用以下参数**指定使用的镜像源**：（可指定为集群内部的docker镜像源地址，要提前在docker的daemon.json文件里配置信任）

--image-repository="cof-lee.com:5443/k8s"

**★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

# 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

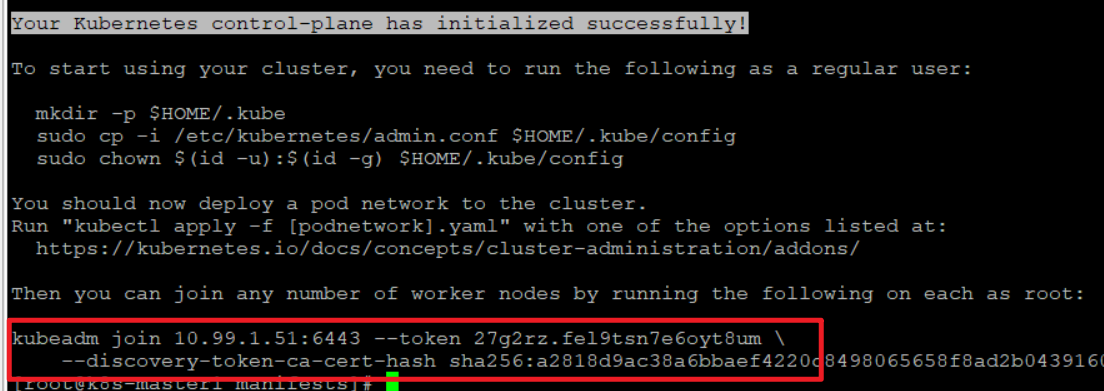
cof-lee.com:5443/k8s/kube-proxy:v1.19.4

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



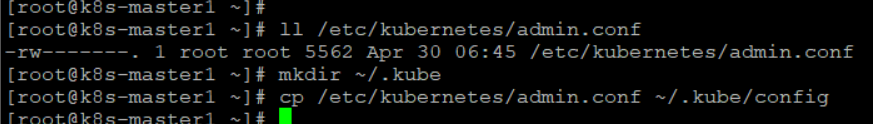
当出现Your Kubernetes control-plane has initialized successfully!这行时说明初始化k8s成功了

记住最后2行命令，是用来让node结点加入集群的token命令

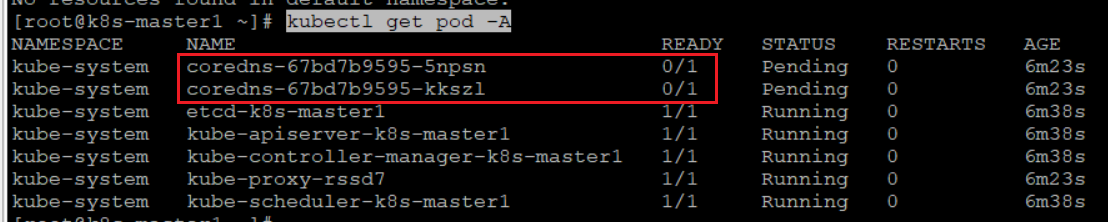
**③k8s认证文件.kube/config**

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



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

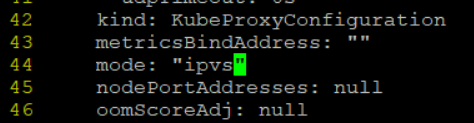


如果有2个coredns的pod未启动就绪，因为pod网络组件还未部署，部署后再查看，如果还未启动，则可能是网络不通，防火墙要放通pod容器网段入站。

**④service使用ipvs模式**

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

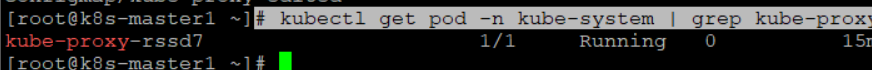
找到kind: KubeProxyConfiguration 这行下面的mode: "" ，在""写入ipvs，保存退出



（前提是在准备工作时已加载ipvs内核模块）

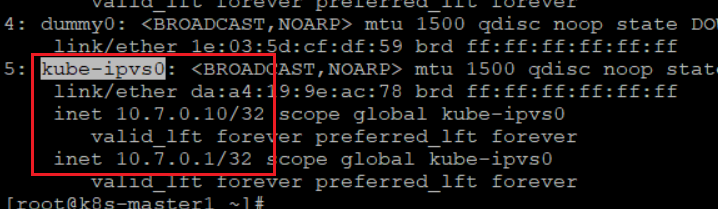
再删除kube-proxy的所有pod（即重启这些pod）

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



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

开启了ipvs模式的k8s结点上ip add会看到名为kube-ipvs0的接口名



**⑤部署flannel网络插件**

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

flannel使得跨主机的Pod之间能通信，底层实现是使用了vxlan

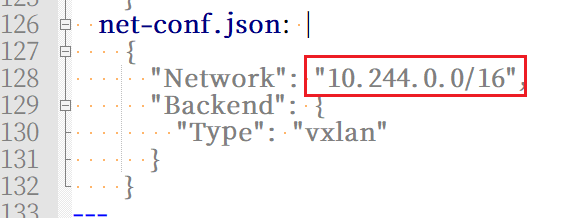
# wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

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

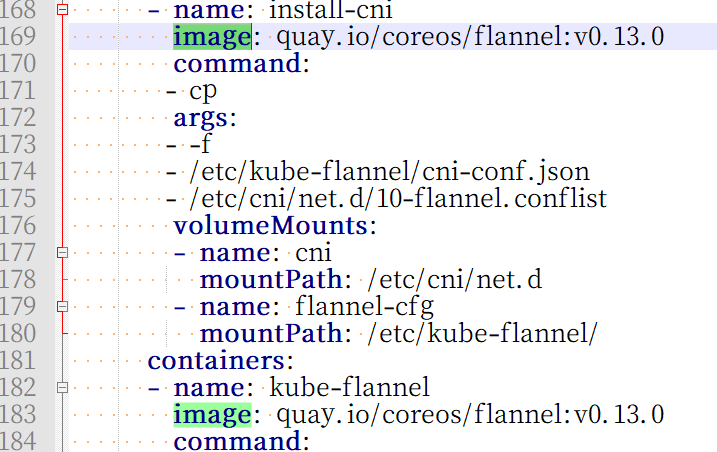
先下载插件的k8s部署yml文件

# 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

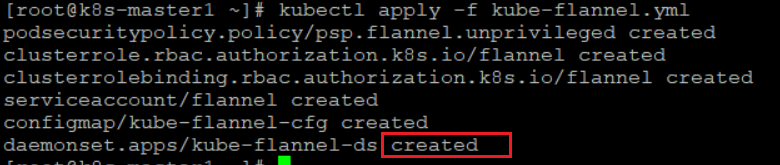


#默认使用的docker镜像是quay.io/coreos/的仓库，可改为自己集群的docker仓库



保存，退出

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



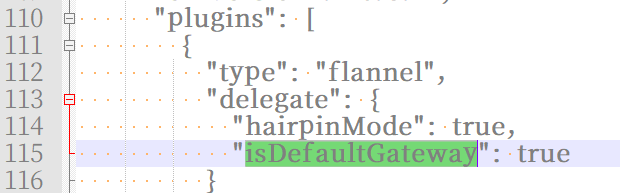
如果pod启动失败了，查看日志

# 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的网卡，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忘记配置网关了，所以它找不到这行，导致flannel启动失败

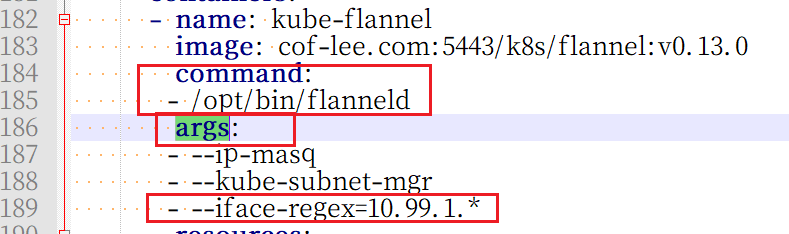


解决方法是给服务加个网关（默认路由）或者在kube-flannel.yml里指定vxlan绑定的网卡设备（不建议直接绑定网络接口名，因为不同的服务器接口名不太一样，可匹配ip网段所在的网络接口）

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

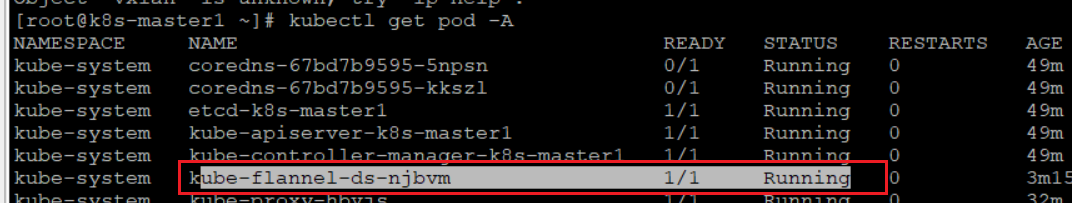
再在kube-flannel.yml部署配置文件里的- /opt/bin/flanneld下面的

args下面添加一行配置 - --iface-regex=10.99.1.\* 匹配要通信的ip网段



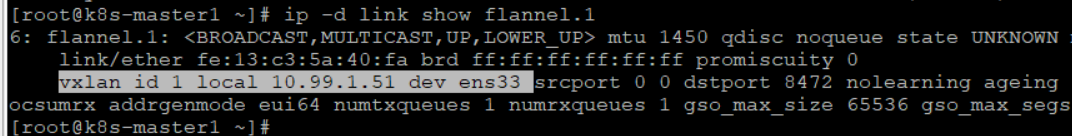
最后重新部署flannel的deployment文件

# kubectl apply -f kube-flannel.yml



可见flannel的pod启动了

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



**章三、Node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，都是章0的内容。准备工作做好后，安装k8s二进制包

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

# systemctl daemon-reload

# systemctl enable kubelet

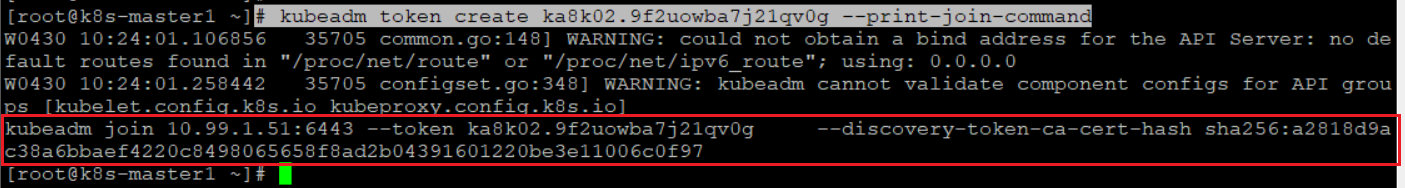
# systemctl start kubelet

再使用安装第一个k8s的master结点时的最后2行命令（kubeadm join xxxx）加入集群，不过这命令里的token默认是2小时有效期，可以重新生成token命令再去node结点上执行

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

ka8k02.9f2uowba7j21qv0g

# kubeadm token create ka8k02.9f2uowba7j21qv0g --print-join-command #生成加入集群的命令



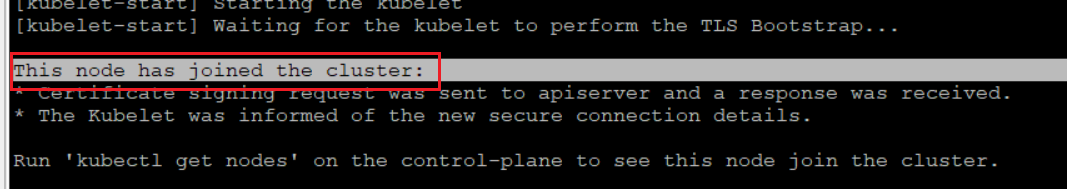
kubeadm join 10.99.1.51:6443 --token ka8k02.9f2uowba7j21qv0g \

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

再复制以上命令到node结点上执行：

node# 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: 这句说明加入成功

可以去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

**★将node结点移出k8s集群**

master# kubectl taint node *结点名* keyrm:NoExecute #先打上污点，驱逐pod

等几分钟，确认目标node结点上没有工作pod运行后，再删除

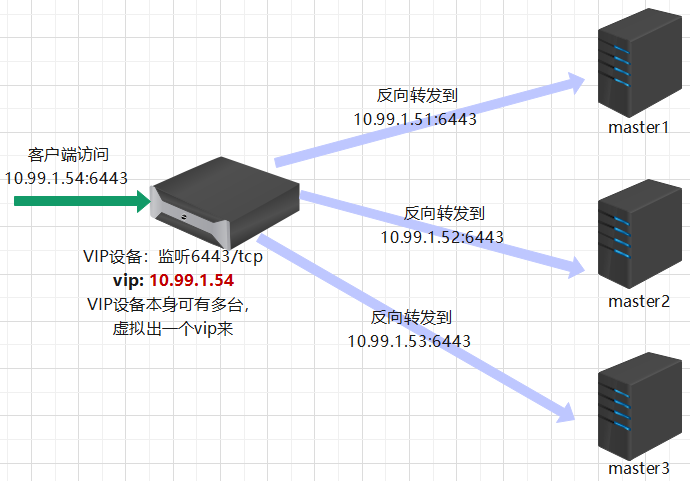
master# kubectl delete nodes *结点名* #将目标结点从集群里删除

node# kubeadm reset #node结点上重置kubeadm配置

**章四、K8S高可用集群搭建**

前面只装了一台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，只是要规划一个虚拟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，反向代理到后端的多台master结点6443端口即可。

高可用集群拓扑图：



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\_6443

backend my\_k8s\_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

**①先装master1**

★**直接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

--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

--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 #在后续执行加入节点时自动分发同步证书文件

**★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

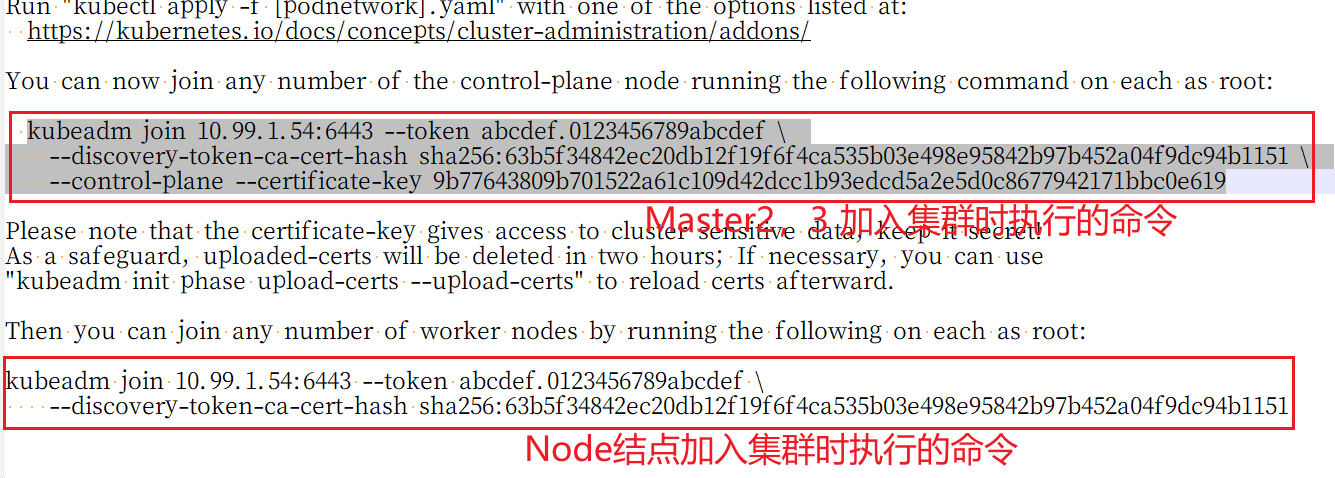
cgroupDriver: systemd

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

**再参考章一的以下2步**

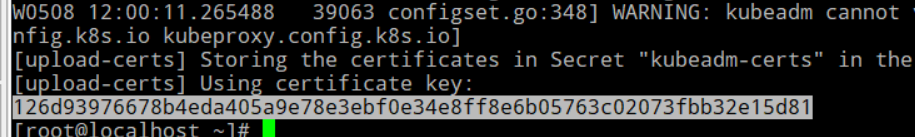
**④service使用ipvs模式**

**⑤部署flannel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

# 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 #编辑master1上的

在 - --initial-cluster=k8s-master1=https://10.99.1.51:2380这行后面添加

,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）

#docker ps | grep etcd #将本master结点上的etcd容器清除（master1,2两台都执行）

#docker stop 4699e15cf918

#docker stop d85674105805

#docker container prune

y

#再重启本master结点

# reboot

# 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**

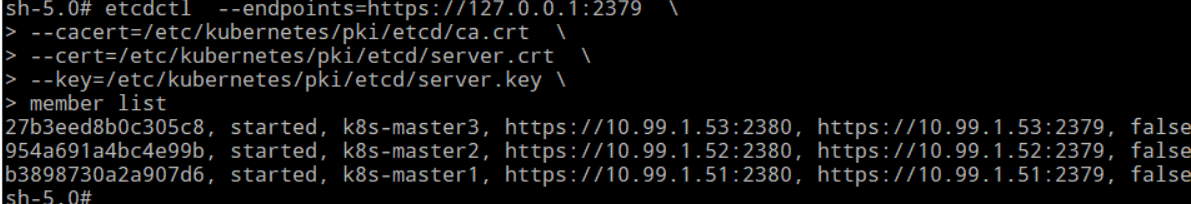
etcdctl --endpoints=https://127.0.0.1:2379 \

--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**

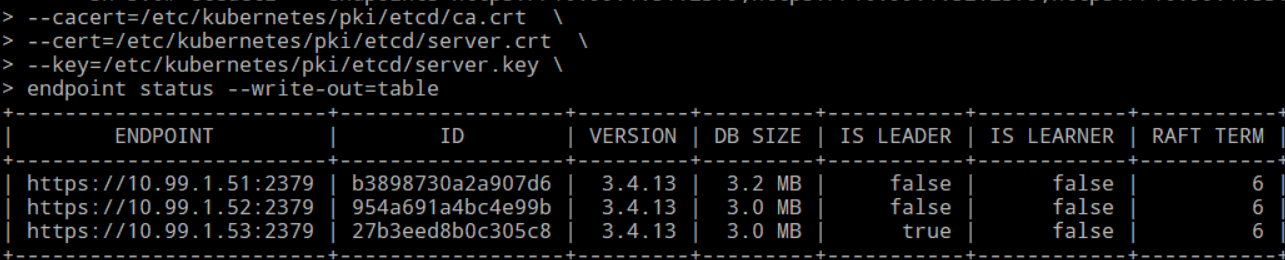
etcdctl --endpoints=https://10.99.1.51:2379,https://10.99.1.52:2379,https://10.99.1.53:2379 \

--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**

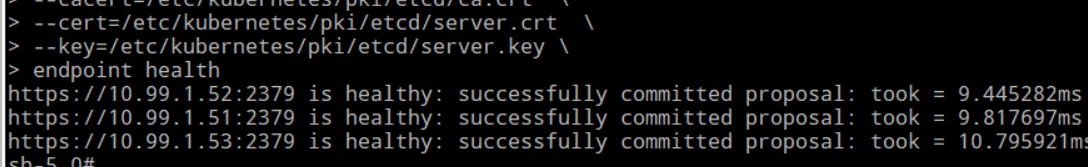
etcdctl --endpoints=https://10.99.1.51:2379,https://10.99.1.52:2379,https://10.99.1.53:2379 \

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

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

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

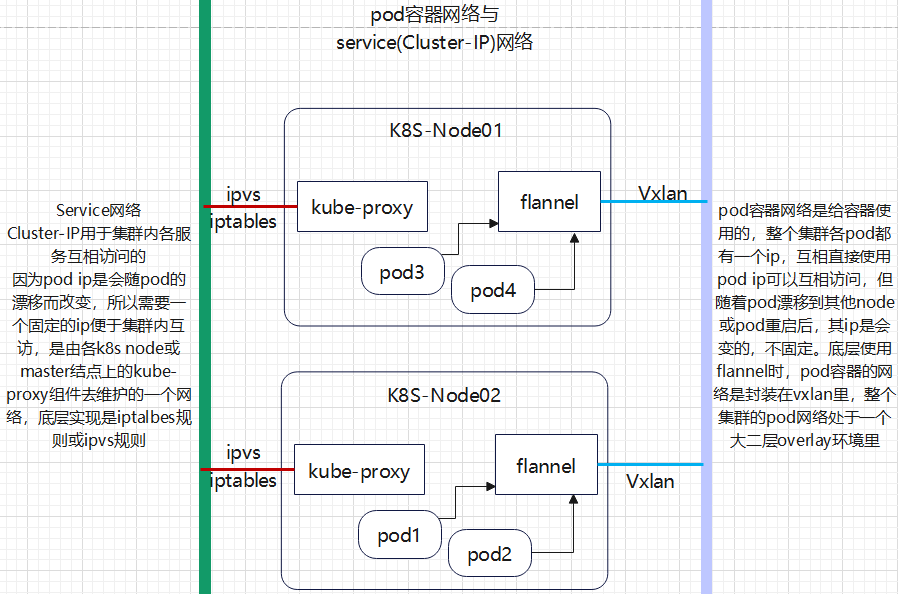
endpoint health



# 至此，k8s高可用集群搭建完成

**章五、K8S的网络原理**

**★pod网络与Service网络**



整个K8S集群有3种网络：

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容器组的真实ip，pod容器网络是给容器使用的，整个集群各pod都有一个 ip，互相直接使用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 node或master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

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN

inet 127.0.0.1/8 scope host lo #本地环回口

2: ens33: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP

inet 10.99.1.61/24 brd 10.99.1.255 scope global #k8s服务器本身用于通信的网口

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN

inet 172.17.0.1/16 brd 172.17.255.255 #docker默认的容器网络，k8s不用它

5: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN

inet 10.7.0.1/32 scope global kube-ipvs0 #Service网络，使用ipvs实现

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

inet 10.244.0.0/32 brd 10.244.0.0 scope global flannel.1 #flannel.1是vxlan设备

7: cni0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 #cni0是一个网桥设备，

inet 10.244.3.1/24 brd 10.244.3.255 scope global cni0 #是本服务器上的pod网络的网关

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

#

# 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，默认一个广播域）

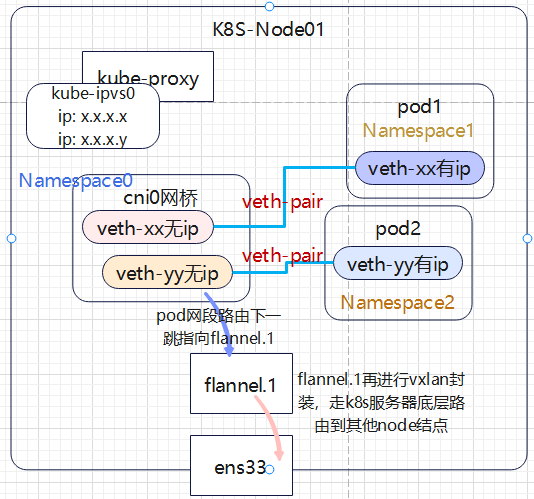
bridge name bridge id STP enabled interfaces

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

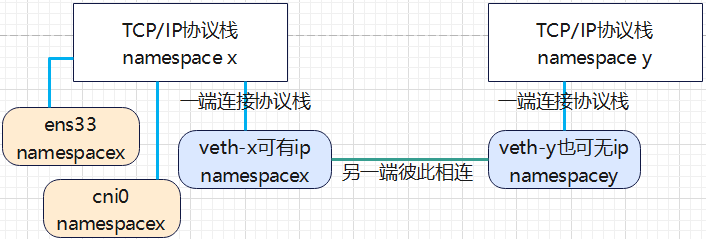
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink #去往10.244.2/24的pod网段下一跳指向flannel.1

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：**

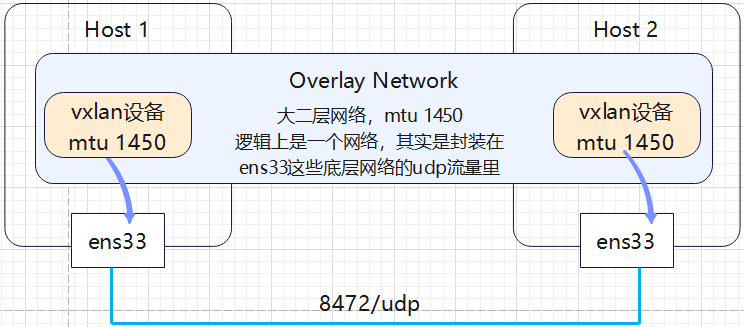
|  |  |  |  |
| --- | --- | --- | --- |
| 8 bit | 24 bit | 24 bit | 8 bit |
| vxlan tag | Reserved | VNI | Reserved |

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）



**章六、kube-proxy的Service网络服务**

k8s的Service就是访问真实pod里面的服务端口的一种服务，它是由kube-proxy组件实现的，创建一个service资源时，kube-proxy会想办法让访问service的流量转发到真实pod里面去，如果一个service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的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） |

**k8s的service类型**：

|  |  |
| --- | --- |
| Cluster Ip | 只能用于集群内部，iptables,ipvs  clusterip: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， |
| LoadBalancer | 通过使用云服务商的负载均衡器对集群外部提供服务，自动创建NodePort与Cluster ip；集群外部的负载均衡器最终请用户请求流量路由到目标NodePort再到Cluster ip |
| 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

#

NodeIP:30001 --> PodIP:8443

ServiceIP:443 --> PodIP:8443

**章七、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 nginx-pod.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod-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

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

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

#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，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，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:

# lbname: lbvalue

#nodeSelector: #选择特定node结点去运行此容器

# nodelbxx: lbvalue

---

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

ports:

- port: 2380 # service ip 的port

targetPort: 80 # pod容器里的port

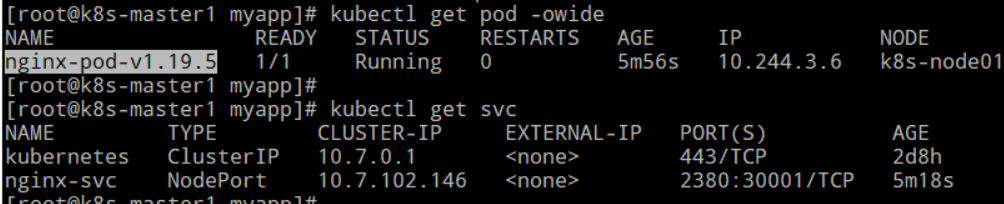
nodePort: 30001 # Node IP对外的port

#

# kubectl apply -f nginx-pod.yml #应用此文件，创建文件里的资源

# 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\_nginx-pod-v1.19.5\_default\_41b8e6f3-6b7e-4e72-9fe5-9d8e12b1eebe\_1

k8s\_POD\_nginx-pod-v1.19.5\_default\_41b8e6f3-6b7e-4e72-9fe5-9d8e12b1eebe\_5 （pause）

可见命名方式为： k8s\_容器名\_pod名\_命名空间\_uuid

# kubectl delete -f nginx-pod.yml #删除文件里的资源

**★容器探测其他配置：**

livenessProbe: #pod存活性探测

httpGet: #使用http

path: /xx/cxxx/healthz #请求的文件

port: 80

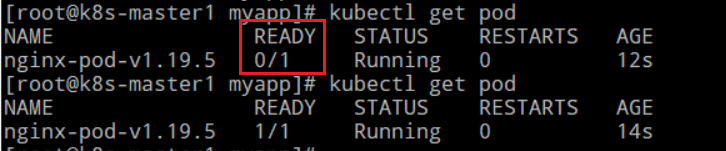
scheme: HTTP #协议，可为HTTP, HTTPS

#

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时，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

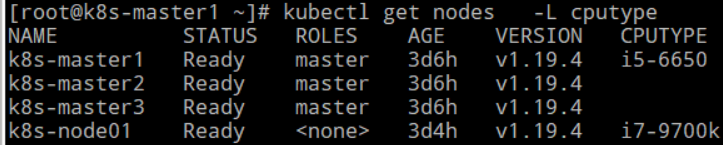
**★通过kind: Pod创建的容器组，在kubectl delete pod xxx时，它就真的被删除了，不会重新拉起一个新的pod，建议使用pod控制器去创建pod**

**★给目标node打标签**

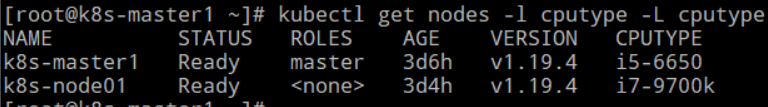
# kubectl label nodes k8s-node01 cputype=i7-9700k #给指定的node结点打标签

# 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



**★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

**★limitRange**

在

**★namespace**

创建命名空间

# vi testxx-namespace.yaml

apiVersion: v1

kind: Namespace

metadata:

name: testxx

labels:

labelnamexx: valuexxx

#

# kubectl apply -f testxx-namespace.yaml

**章八、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

**★Deployment控制器**

# 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-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-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-svc

spec:

selector: #通过标签选择目标pod

podlable: nginx-pod #与目标pod的labels相同

type: NodePort

ports:

- port: 2380 # service ip 的port

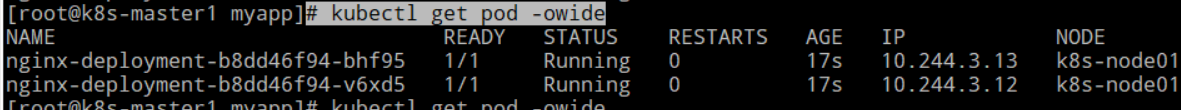
targetPort: 80 # pod容器里的port

nodePort: 30001 # Node IP对外的port

#

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

# kubectl get pod -owide #查看



\*AGE后的IP为pod-ip

# kubectl get deployment #查看

**★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）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应用，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

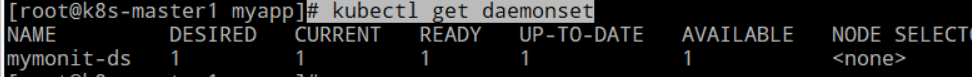
initialDelaySeconds: 10

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

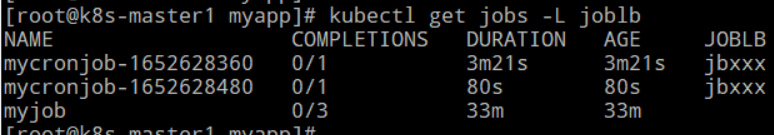
memory: "256Mi"

cpu: "200m" #1核cpu为 1000m

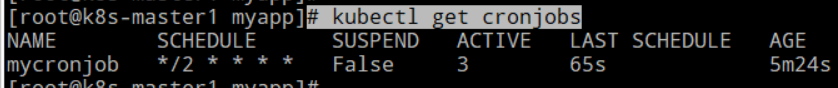
restartPolicy: Never #重启策略，默认Always

#

# 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



**章九、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:

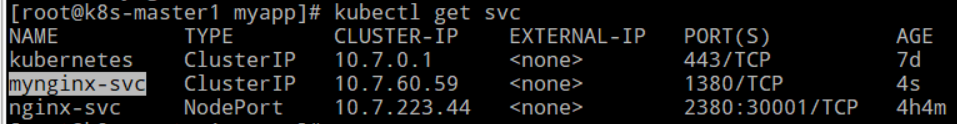
clientIP:

timeoutSeconds: 3600 #默认10800秒（3小时）

#

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

# kubectl get svc #查看

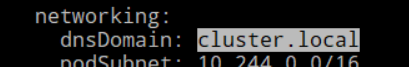


#

★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:

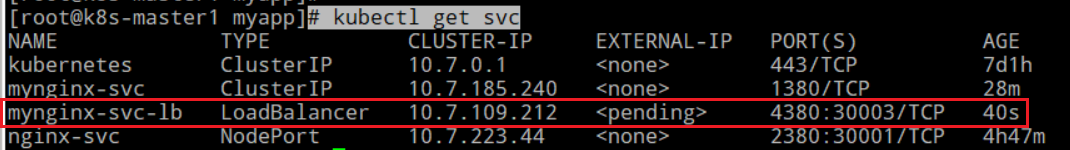
clientIP:

timeoutSeconds: 3600 #默认10800秒（3小时）

#

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

# kubectl get svc #查看



**★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

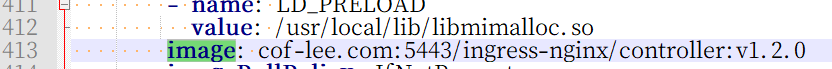
或者 wget https://limaofu.github.io/scripts/ingress-nginx-controller-v1.2.0.yaml

# 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





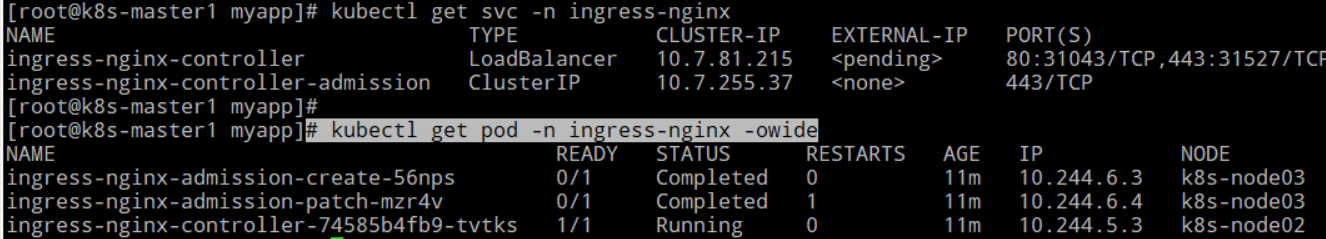


# kubectl apply -f ingress-nginx-controller-v1.2.0.yaml #创建ingress-controller服务

#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:**

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

**章十、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**

# vi nginx-pod-volume.yml #内容如下

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod-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-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

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

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

volumeMounts: #容器里指定 要使用的存储卷

- name: alog #根据此名称去寻找对应的存储卷

mountPath: /var/log/nginx #容器里的挂载点

- name: xxlog #根据此名称去寻找对应的存储卷

mountPath: /var/log/xxx #容器里的挂载点

readOnly: true #只读

volumes: #存储卷定义，与containers 同级

- name: alog #存储卷名称

hostPath: #存储卷类型，hostPath为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

- name: xxlog #存储卷名称

hostPath: #存储卷类型，hostPath为node结点本地目录

path: /var/log/ #node结点上的本地目录

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

restartPolicy: Always #重启策略，默认Always

---

apiVersion: v1

kind: Service #创建service资源

metadata:

name: nginx-svc-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-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

targetPort: 80 # pod容器里的port

nodePort: 30051 # Node IP对外的port

#

**②nfs存储卷**

在

spec:

containers:

- name: nginx-v1-19-5

#xxx略

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

**⑤PVC**

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

★创建PV

# vi pv-nfs-01.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-nfs-01

labels:

pvlab: pvlab

spec:

capacity:

storage: 10Gi

volumeMode: Filesystem #默认，文件系统

accessModes:

- ReadWriteMany #可被多个node结点同时读写挂载

persistentVolumeReclaimPolicy: Retain #保持不动，或 Recycle回收

storageClassName: slow

mountOptions: #挂载nfs的参数

- hard #如 ro, soft, hard

- nfsvers=4.1

nfs:

path: "/nfs-share"

server: 10.99.1.71

#

★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。

# vi pvc-nfs-01.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc-nfs-01

labels:

pvclab: pvclab

spec:

accessModes:

- ReadWriteMany

volumeMode: Filesystem

resources:

requests:

storage: 10Gi

storageClassName: slow #存储类名

selector:

matchLabels:

pvlab: pvlab #通过labels去匹配pv

#

**★pod使用pvc**

#

volumeMounts:

- name: alog

mountPath: /var/log/nginx

volumes:

- name: alog

persistentVolumeClaim:

claimName: pvc-nfs-01

ReadOnly: false #默认

restartPolicy: Always #重启策略，默认Always

#

**⑥存储类与动态pv供给**

在

**章十一、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。

**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

# vi cm-name2.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: cm-name2

namespace: default

data:

key1: value1

key2: value2

key3: value3

#

# 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:

configMapKeyRef:

name: cm-name-tz #使用的configmap名称

key: TZ #cm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为必选

#

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

**③创建Secret**

在

④创建imagePullSecret

**章十二、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 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>

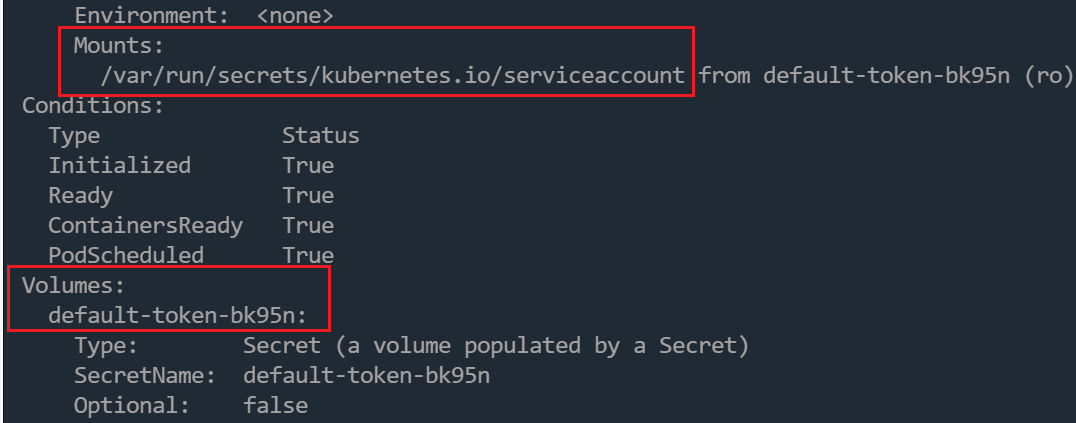
Image pull secrets: <none>

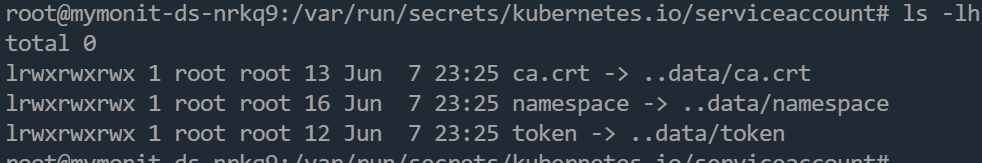
Mountable secrets: default-token-bk95n

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**

# vi sa-test.yaml #内容如下

apiVersion: v1

kind: ServiceAccount

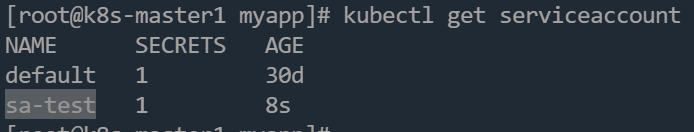
metadata:

name: sa-test

namespace: default

#

# kubectl apply -f sa-test.yaml



★服务账号可指定关联一个imagePullSecret资源

apiVersion: v1

kind: ServiceAccount

metadata:

name: sa-test

namespace: default

imagePullSecrets:

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

#

**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作用于名称空间，ClusterRole作用于集群及所有名称空间

（集群级别的资源有nodes，非资源类型的端点）

将Role绑定到某用户(组)需要使用RoleBinding资源，

将ClusterRole绑定到某用户(组)需要使用ClusterRoleBinding资源

**★创建Role**

# vi pods-reader-role.yaml #内容如下

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

name: 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

#

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

**★创建RoleBinding**

# vi pods-reader-rolebinding.yaml #内容如下

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: pods-reader-rolebinding

namespace: default

subjects:

- kind: User

name: kube-user01

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pods-reader

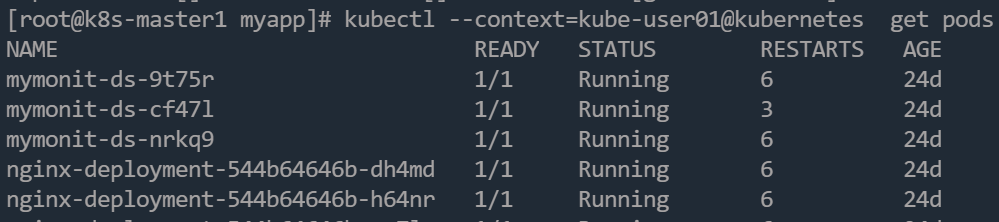
apiGroup: rbac.authorization.k8s.io

#

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

# vi pods-reader-clusterrole.yaml #内容如下

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: pods-reader-cr # 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

#

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

**★创建ClusterRoleBinding**

# vi pods-reader-clusterrolebinding.yaml #内容如下

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: pods-reader-clusterrolebinding

subjects:

- kind: User

name: kube-user01

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: pods-reader-cr

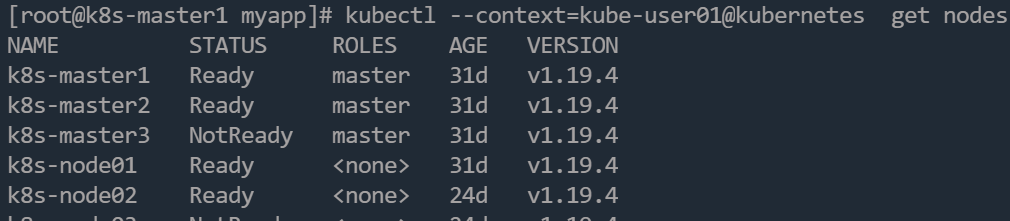
apiGroup: rbac.authorization.k8s.io

#

# kubectl apply -f pods-reader-clusterrolebinding.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

# vi xx-clusterrole.yaml

aggregationRule:

clusterRoleSelectors:

- matchLabels:

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

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: xx

rules: []

#

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"

**章十三、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：

先提前下载下面2个镜像：

docker.io/kubernetesui/dashboard:v2.6.0

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

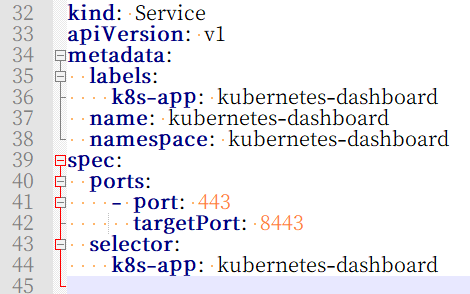
# wget https://raw.githubusercontent.com/kubernetes/dashboard/v2.6.0/aio/deploy/recommended.yaml

# 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为我们环境的repo源，再修改recommended.yaml文件里的镜像地址，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

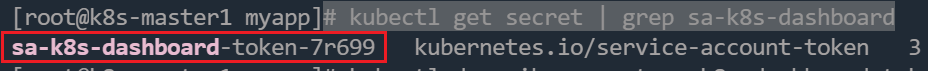
apiGroup: ""

#

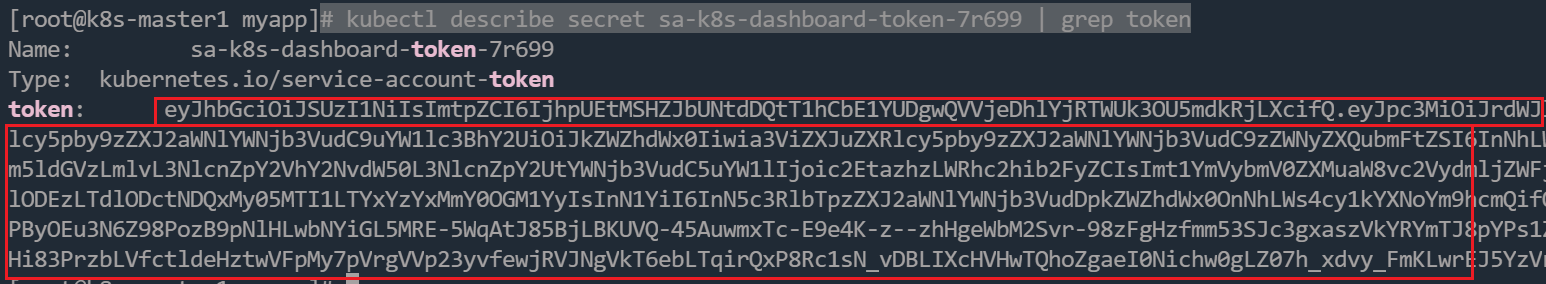
# kubectl apply -f admin-clusterrolebinding-k8s-dashboard.yaml

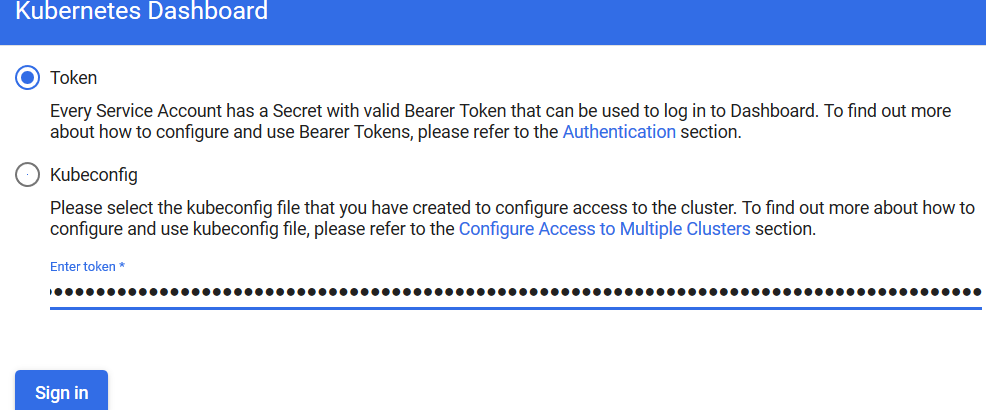
最后获取sa-k8s-dashboard服务账号的token，就可去访问k8s-dashboard界面了

# 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

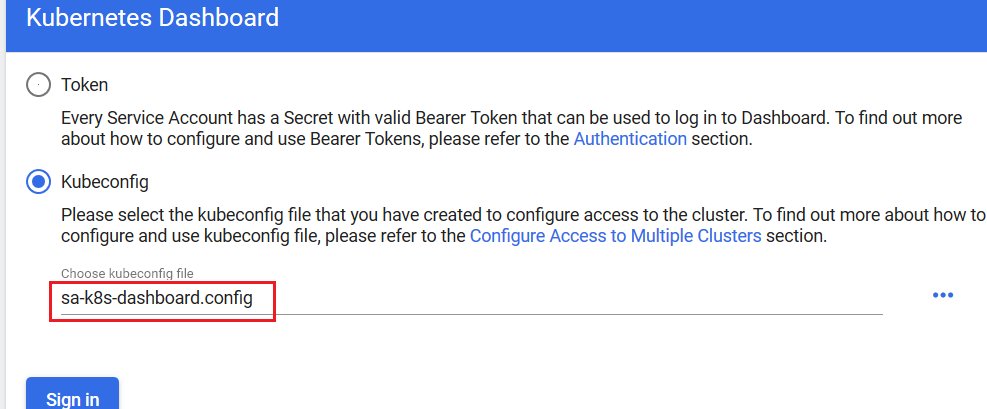
kubectl config set-context sa-k8s-dashboard --cluster=kubernetes \

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

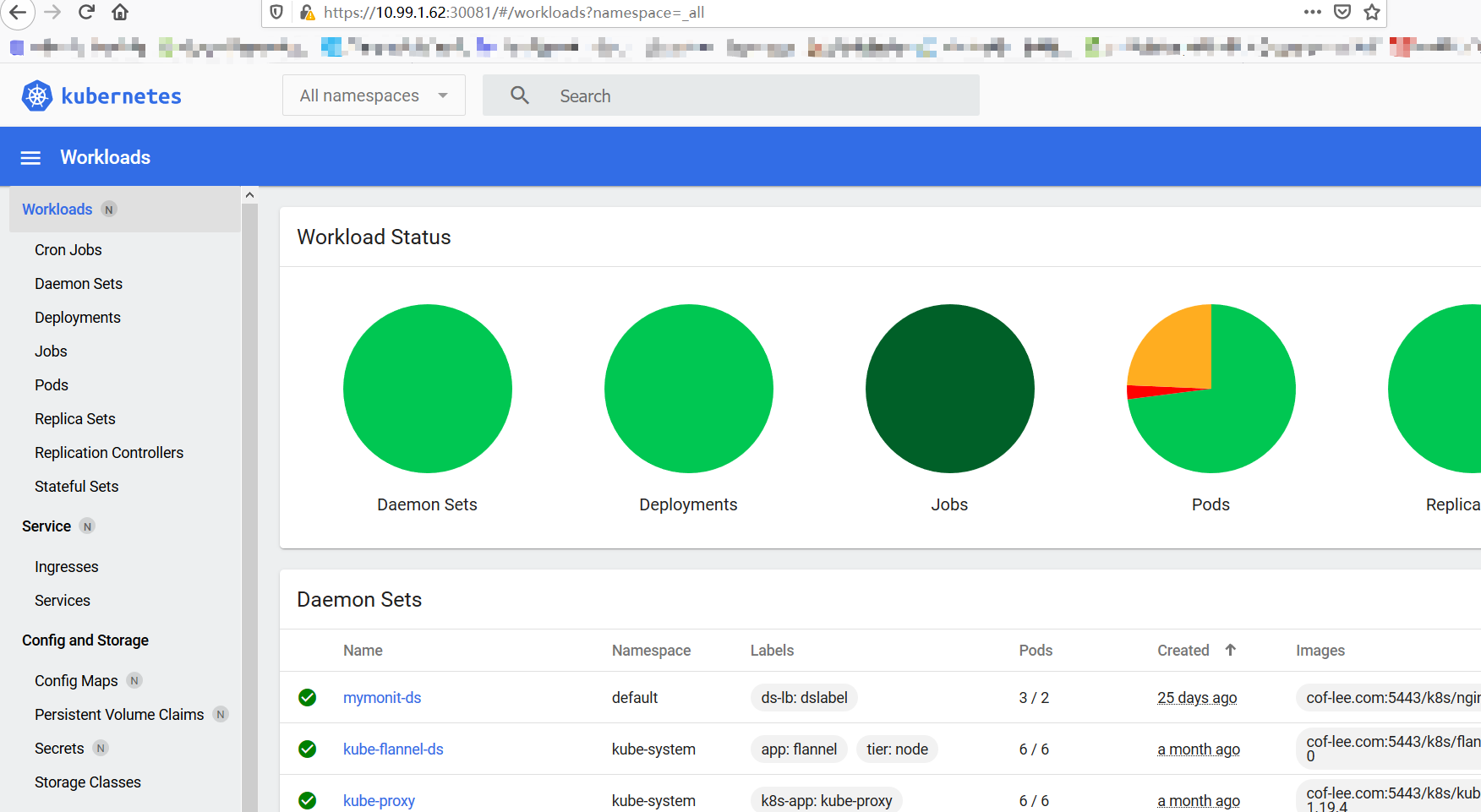
kubectl config use-context sa-k8s-dashboard --cluster=kubernetes \

--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：



**章十四、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

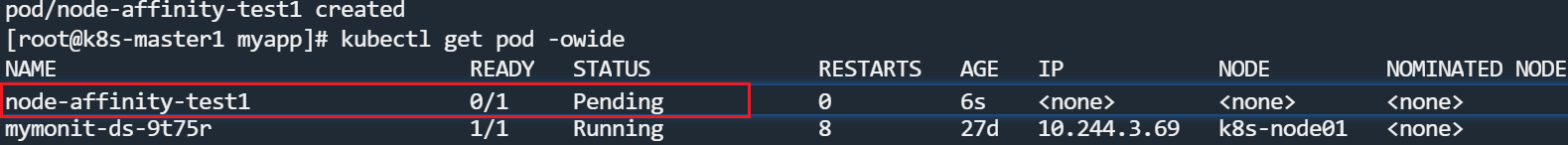
- matchExpressions:

- {key: disktype, operator: In, values: ["ssd"]} #匹配标签名为 disktype 值为 ssd 的node，多行 -{key:xx} 之间是 与 关系

restartPolicy: Never #重启策略，默认Always

#

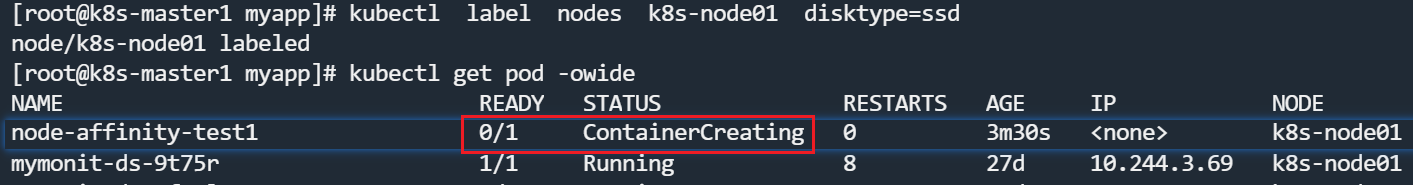
# kubectl apply -f node-affinity-test1.yml



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，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

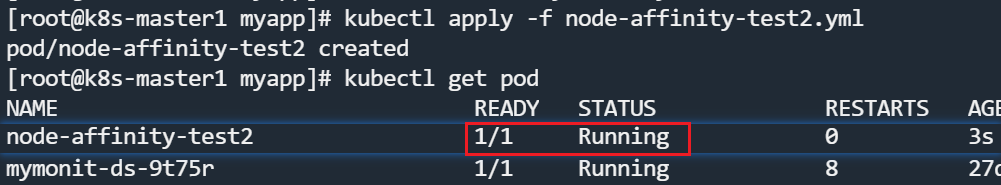
- {key: disktype, operator: NotIn, values: ["hdd"]} #此key的值不为hdd

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

restartPolicy: Never #重启策略，默认Always

#

# 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 #重启策略，默认Always

#

# 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"]} #匹配自己

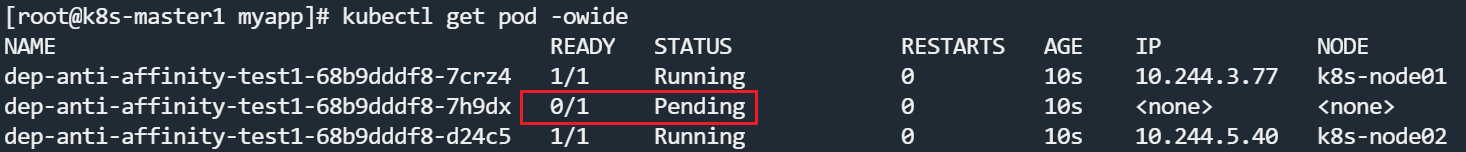
topologyKey: kubernetes.io/hostname

restartPolicy: Always #重启策略，默认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：

node-role.kubernetes.io/master:NoSchedule #master结点默认打上了此污点，如果想让某pod调度到master结点上，需要设置pod容忍此污点

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

# 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"

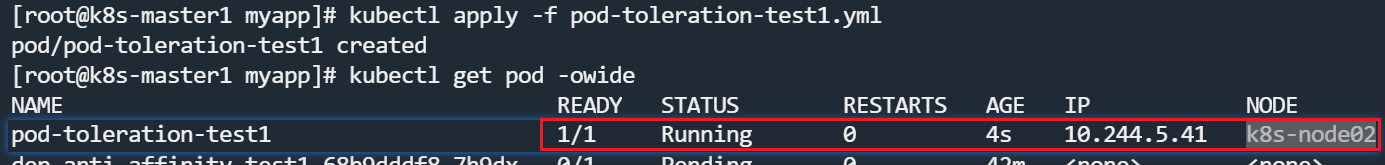
operator: "Equal" #key,value,effect三者必须都匹配上

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

**章十五、prometheus监控系统**

在

**章十六、Helm包管理**

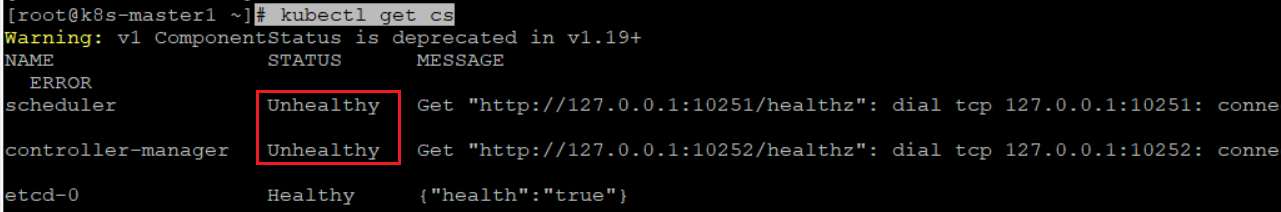
在

**章n、其他**

在

**①检查集群健康状态为Unhealthy**

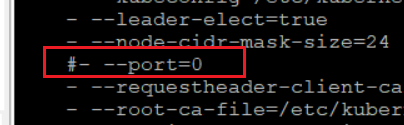
# kubectl get cs



# vi /etc/kubernetes/manifests/kube-controller-manager.yaml

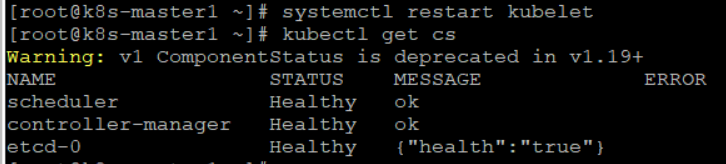
# vi /etc/kubernetes/manifests/kube-scheduler.yaml

#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



在

**标准色：**

