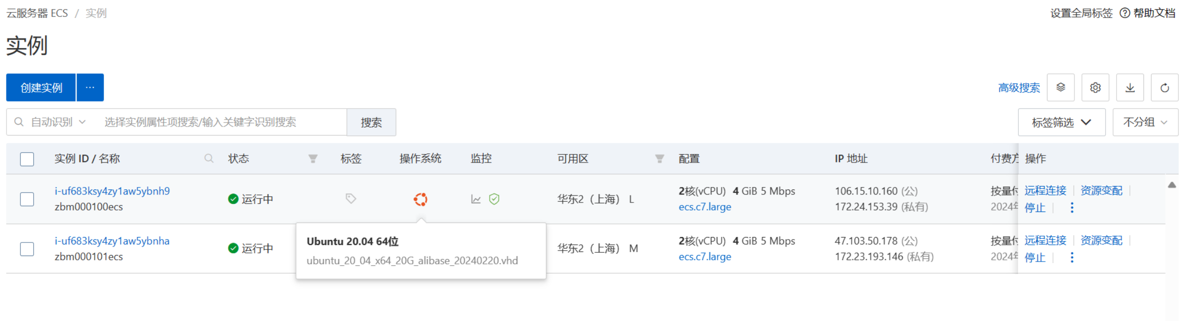
K8s实践作业

本次事件作业采用的是阿里云上的云服务器，云服务器的配置如下图。



□	实例 ID / 名称	状态	标签	操作系统	监控	可用区	配置	IP 地址	付费方	操作
□	i-uf683ksy4zy1aw5ybnh9 zbm000100ecs	✓ 运行中				华东2（上海） L	2核(vCPU) 4 GiB 5 Mbps ecs.c7.large	106.15.10.160 (公) 172.24.153.39 (私有)	按量付 2024年	远程连接 | 资源变配 | 停止 | ⋮
□	i-uf683ksy4zy1aw5ybnha zbm000101ecs	✓ 运行中				华东2（上海） M	2核(vCPU) 4 GiB 5 Mbps ecs.c7.large	47.103.50.178 (公) 172.23.193.146 (私有)	按量付 2024年	远程连接 | 资源变配 | 停止 | ⋮

Ubuntu 20.04 64位
ubuntu_20_04_x64_20G_alibase_20240220.vhd

1.设置主机名

```
sudo hostnamectl set-hostname k8s-master
sudo hostnamectl set-hostname k8s-worker
```

可以通过hostname查看是否设置正确

2.修改本地DNS，让两台机器之间可以相互ping通

```
sudo nano /etc/hosts
```

在其中添加这两行

172.24.153.39   k8s-master k8s-master
172.23.193.146 k8s-worker k8s-worker

尝试是否能够ping通

```
root@k8s-worker:~# ping k8s-master
PING k8s-master (172.24.153.39) 56(84) bytes of data.
64 bytes from k8s-master (172.24.153.39): icmp_seq=1 ttl=64 time=2.38 ms
64 bytes from k8s-master (172.24.153.39): icmp_seq=2 ttl=64 time=2.27 ms

root@k8s-master:~# ping k8s-worker
PING k8s-worker (172.23.193.146) 56(84) bytes of data.
64 bytes from k8s-worker (172.23.193.146): icmp_seq=1 ttl=64 time=2.76 ms
64 bytes from k8s-worker (172.23.193.146): icmp_seq=2 ttl=64 time=2.62 ms

```

出现如上即代表可以ping通。

3.配置内核路由转发及网桥过滤

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter
# 设置所需的 sysctl 参数，参数在重新启动后保持不变
```

```
cat > /etc/sysctl.d/k8s.conf << EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
# 查看是否加载
# lsmod | grep br_netfilter
# br_netfilter           22256  0
# bridge                151336  1 br_netfilter
# 应用 sysctl 参数而不重新启动
sudo sysctl --system
```

通过运行以下指令确认 `net.bridge.bridge-nf-call-iptables`、`net.bridge.bridge-nf-call-ip6tables` 和 `net.ipv4.ip_forward` 系统变量在你的 `sysctl` 配置中被设置为 1：

```
root@k8s-master:/etc/containerd# sysctl net.bridge.bridge-nf-call-iptables net.bridge.bridge-nf-call-ip6tables net.ipv4.ip_forward
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
```

代表设置成功

4.下载容器运行时

我们这里选择下载`containerd`作为我们的容器运行时，具体下载方案参考`containerd`的官方文档，如下：这里简单就以我们的k8s-master节点举例

```
root@k8s-master:/home# ls
cni-plugins-linux-amd64-v1.4.1.tgz  containerd-1.7.16-linux-amd64.tar.gz  runc.amd64
```

将安装必要的压缩包文件传入服务器的/home文件夹下。

安装`containerd`

```
root@k8s-master:/home# tar Cxzvf /usr/local containerd-1.7.16-linux-amd64.tar.gz
bin/
bin/containerd-shim-runc-v2
bin/containerd-stress
bin/containerd
bin/containerd-shim-runc-v1
bin/ctr
bin/containerd-shim
```

然后我们将`containerd.service`这个文件从https://raw.githubusercontent.com/containerd/containerd/main/containerd.service 拷贝到服务器的 `/usr/lib/systemd/system/containerd.service`然后运行下面指令

```
systemctl daemon-reload
systemctl enable --now containerd
```

安装`runc`

```
root@k8s-master:/home# install -m 755 runc.amd64 /usr/local/sbin/runc
```

安装`cni`

```
root@k8s-master:/home# mkdir -p /opt/cni/bin
root@k8s-master:/home# tar Cxzvf /opt/cni/bin cni-plugins-linux-amd64-v1.4.1.tgz
./
./LICENSE
./host-device
./dummy
./README.md
./firewall
./macvlan
./bridge
./dhcp
./bandwidth
./tuning
./vlan
./ipvlan
./ptp
./static
./loopback
./tap
./host-local
./sbr
./portmap
./vrf
```

生成`containerd`的配置

`containerd` 默认配置文件在 `/etc/containerd` 目录下，名称为 `config.toml`。

可以通过如下命令生成默认配置：

```
root@k8s-master:/etc# mkdir -p /etc/containerd
root@k8s-master:/etc# containerd config default > /etc/containerd/config.toml
```

注意需要修改下面两个地方：

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  ...
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    SystemdCgroup = true
```

`SystemdCgroup`这里需要设为true

另外需要设置沙盒镜像为阿里云。

`sandbox_image = "registry.aliyuncs.com/google_containers/pause:3.9"`

设置完之后重启`containerd`

```
sudo systemctl restart containerd
```

5.下载`kubelet kubeadm kubectl`

这里我们参考阿里云上提供的官方下载方法，设置阿里镜像加速安装。

```
# 1. 更新软件源并安装基础依赖
sudo apt-get update && sudo apt-get install -y apt-transport-https ca-certificates curl

# 2. 添加Kubernetes GPG密钥（通过阿里云镜像）
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://mirrors.aliyun.com/kubernetes-new/core/stable/v1.28/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

# 3. 添加Kubernetes APT源（阿里云镜像）
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://mirrors.aliyun.com/kubernetes-new/core/stable/v1.28/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list

# 4. 更新软件源并安装组件
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl  # 防止自动升级
```

6.然后我们调用如下指令初始化master节点

```
# 使用阿里云镜像源初始化Kubernetes集群 (v1.28.9)
sudo kubeadm init \
  --kubernetes-version=v1.28.9 \
  --pod-network-cidr=10.244.0.0/16 \
  --image-repository=registry.aliyuncs.com/google_containers
```

最后可以看到终端有下面这行指令

```
kubeadm join 172.24.153.39:6443 --token yb3v37.o8p3emy5aie3yro4 \
    --discovery-token-ca-cert-hash sha256:cc3365e7c219893c51ca3c3278a41cf762339100c209c6b8c10a0de213300b0a
```

同时我们看到下面的提示

```
To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

执行要求的上述三条指令。

这个时候我们看到

```
root@k8s-master:/home# kubectl get node
NAME         STATUS     ROLES           AGE   VERSION
k8s-master   NotReady   control-plane   36m   v1.28.9
```

还没有ready，需要安装网络插件，这里我们选择安装flannel插件

从https://github.com/flannel-io/flannel/blob/master/Documentation/kube-flannel.yml处下载`kube-flannel.yml`

```
#安装插件
kubectl create -f kube-flannel.yml
```

一段时间过后调用`kubectl get node`

```
root@k8s-master:/home# kubectl get node
NAME         STATUS   ROLES           AGE   VERSION
k8s-master   Ready    control-plane   37m   v1.28.9
```

master节点已经就绪了。

7.将worker节点也加入集群

调用我们上面的指令

```
root@k8s-worker:/etc/containerd# kubeadm join 172.24.153.39:6443 \
  --token yb3v37.o8p3emy5aie3yro4 \
  --discovery-token-ca-cert-hash sha256:cc3365e7c219893c51ca3c3278a41cf762339100c209c6b8c10a0de213300b0asha256:cc3365e7c219893c51ca3c3278a41cf762339100c209c6b8c10a0de213300b0a
accepts at most 1 arg(s), received 2
To see the stack trace of this error execute with --v=5 or higher
root@k8s-worker:/etc/containerd# kubeadm join 172.24.153.39:6443 \
  --token yb3v37.o8p3emy5aie3yro4 \
  --discovery-token-ca-cert-hash sha256:cc3365e7c219893c51ca3c3278a41cf762339100c209c6b8c10a0de213300b0asha256:cc3365e7c219893c51ca3c3278
accepts at most 1 arg(s), received 2
To see the stack trace of this error execute with --v=5 or higher
root@k8s-worker:/etc/containerd# kubeadm join 172.24.153.39:6443 \
  --token yb3v37.o8p3emy5aie3yro4 \
  --discovery-token-ca-cert-hash sha256:cc3365e7c219893c51ca3c3278a41cf762339100c209c6b8c10a0de213300b0asha256:cc3365e7c219893c51ca3c3278
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

```

```
Run 'kubectl get nodes' on the control-plane to see this node join the
cluster.
```

成功加入集群。

一段时间后，worker节点也就绪了。

```
root@k8s-master:/home# kubectl get node
NAME         STATUS   ROLES           AGE   VERSION
k8s-master   Ready    control-plane   44m   v1.28.9
k8s-worker   Ready    <none>          2m    v1.28.9
```

添加master标签

```
root@k8s-master:/home# kubectl label nodes k8s-master node-
role.kubernetes.io/master=
node/k8s-master labeled
root@k8s-master:/home# kubectl get node
NAME         STATUS   ROLES                  AGE     VERSION
k8s-master   Ready    control-plane,master   47m     v1.28.9
k8s-worker   Ready    <none>                 4m29s   v1.28.9
```

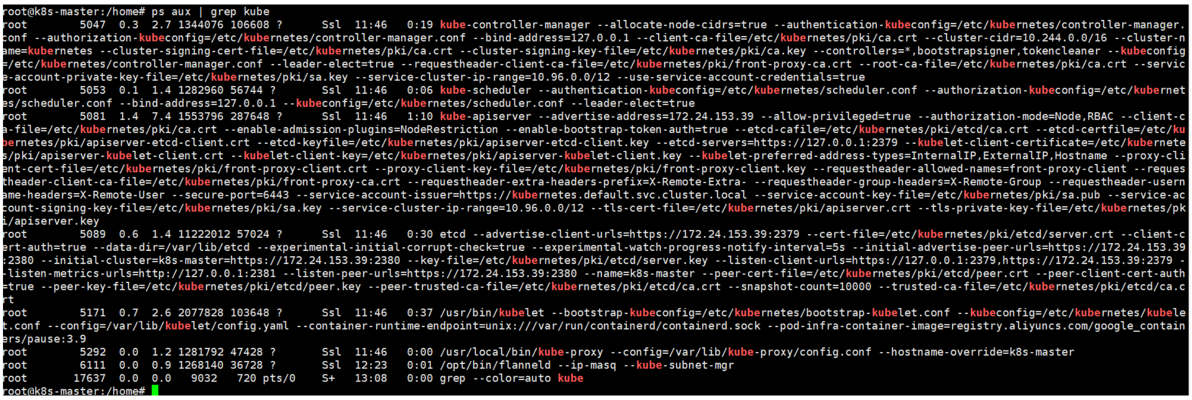
这样就跟作业文档里要求的一致了。

Q1: 请记录所有安装步骤的指令，并简要描述其含义

回答：具体的详实安装教程都已经在上面了。

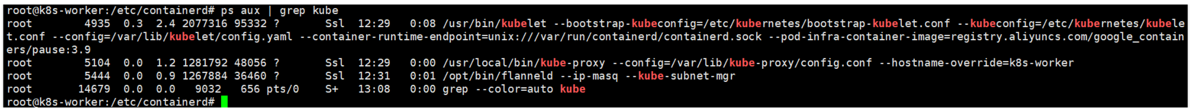
Q2:在两个节点上分别使用 ps aux | grep kube 列出所有和k8s相关的进程，记录其输出，并简 要说明各个进程的作用

回答：在master节点上输出如下：



```
root@k8s-master:/home# ps aux | grep kube
root        5047  0.3  2.7 1344076 106608 ?      Ssl  11:46   0:19 kube-controller-manager --allocate-node-cidrs=true --authentication-kubeconfig=/etc/kubernetes/controller-manager.
conf --authorization-kubeconfig=/etc/kubernetes/controller-manager.conf --bind-address=127.0.0.1 --client-ca-file=/etc/kubernetes/pki/ca.crt --cluster-cidr=10.244.0.0/16 --cluster-n
ame=kubernetes --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt --cluster-signing-key-file=/etc/kubernetes/pki/ca.key --controllers=*,bootstrapsigner,tokencleaner --kubeconfig
=/etc/kubernetes/controller-manager.conf --leader-elect=true --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --root-ca-file=/etc/kubernetes/pki/ca.crt --servic
e-account-private-key-file=/etc/kubernetes/pki/sa.key --service-cluster-ip-range=10.96.0.0/12 --use-service-account-credentials=true
root        5053  0.1  1.4 1282960 56744 ?      Ssl  11:46   0:06 kube-scheduler --authentication-kubeconfig=/etc/kubernetes/scheduler.conf --authorization-kubeconfig=/etc/kubernet
es/scheduler.conf --bind-address=127.0.0.1 --kubeconfig=/etc/kubernetes/scheduler.conf --leader-elect=true
root        5081  1.4  7.4 1553796 287648 ?      Ssl  11:46   1:10 kube-apiserver --advertise-address=172.24.153.39 --allow-privileged=true --authorization-mode=Node,RBAC --client-c
a-file=/etc/kubernetes/pki/ca.crt --enable-admission-plugins=NodeRestriction --enable-bootstrap-token-auth=true --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt --etcd-certfile=/etc/ku
bernetes/pki/apiserver-etcd-client.crt --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key --etcd-servers=https://127.0.0.1:2379 --kubelet-client-certificate=/etc/kubernete
s/pki/apiserver-kubelet-client.crt --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname --proxy-cli
ent-cert-file=/etc/kubernetes/pki/front-proxy-client.crt --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key --requestheader-allowed-names=front-proxy-client --reques
theader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --requestheader-extra-headers-prefix=X-Remote-Extra- --requestheader-group-headers=X-Remote-Group --requestheader-usern
ame-headers=X-Remote-User --secure-port=6443 --service-account-issuer=https://kubernetes.default.svc.cluster.local --service-account-key-file=/etc/kubernetes/pki/sa.pub --service-ac
count-signing-key-file=/etc/kubernetes/pki/sa.key --service-cluster-ip-range=10.96.0.0/12 --tls-cert-file=/etc/kubernetes/pki/apiserver.crt --tls-private-key-file=/etc/kubernetes/pk
i/apiserver.key
root        5089  0.6  1.4 11222012 57024 ?      Ssl  11:46   0:30 etcd --advertise-client-urls=https://172.24.153.39:2379 --cert-file=/etc/kubernetes/pki/etcd/server.crt --client-c
ert-auth=true --data-dir=/var/lib/etcd --experimental-initial-corrupt-check=true --experimental-watch-progress-notify-interval=5s --initial-advertise-peer-urls=https://172.24.153.39
:2380 --initial-cluster=k8s-master=https://172.24.153.39:2380 --key-file=/etc/kubernetes/pki/etcd/server.key --listen-client-urls=https://127.0.0.1:2379,https://172.24.153.39:2379 -
-listen-metrics-urls=http://127.0.0.1:2381 --listen-peer-urls=https://172.24.153.39:2380 --name=k8s-master --peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt --peer-client-cert-auth
=true --peer-key-file=/etc/kubernetes/pki/etcd/peer.key --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt --snapshot-count=10000 --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.c
rt
root        5171  0.7  2.6 2077828 103648 ?      Ssl  11:46   0:37 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubele
t.conf --config=/var/lib/kubelet/config.yaml --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock --pod-infra-container-image=registry.aliyuncs.com/google_contain
ers/pause:3.9
root        5292  0.0  1.2 1281792 47428 ?      Ssl  11:46   0:00 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/config.conf --hostname-override=k8s-master
root        6111  0.0  0.9 1268140 36728 ?      Ssl  12:23   0:01 /opt/bin/flanneld --ip-masq --kube-subnet-mgr
root       17637  0.0  0.0   9032   720 pts/0    S+   13:08   0:00 grep --color=auto kube
root@k8s-master:/home# 
```

在worker节点上输出如下：



```
root@k8s-worker:/etc/containerd# ps aux | grep kube
root        4935  0.3  2.4 2077316 95332 ?      Ssl  12:29   0:08 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubele
t.conf --config=/var/lib/kubelet/config.yaml --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock --pod-infra-container-image=registry.aliyuncs.com/google_contain
ers/pause:3.9
root        5104  0.0  1.2 1281792 48056 ?      Ssl  12:29   0:00 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/config.conf --hostname-override=k8s-worker
root        5444  0.0  0.9 1267884 36460 ?      Ssl  12:31   0:01 /opt/bin/flanneld --ip-masq --kube-subnet-mgr
root       14679  0.0  0.0   9032   656 pts/0    S+   13:08   0:00 grep --color=auto kube
root@k8s-worker:/etc/containerd# 
```

具体功能如下：

1.master节点上的`kube-controller-manager`：Kube Controller Manager是一个守护进程，内嵌随Kubernetes一起发布的核心控制回路。`Kube Controller Manager`通过API服务器监控集群的状态，确保集群处于预期的工作状态。`Kube Controller Manager`由负责不同资源的多个控制器构成。目前，Kubernetes自带的控制器包括副本控制器、节点控制器、命名空间控制器和服务账号控制器等。

（阿里云上的解释）

2.master节点上的`kube-scheduler`：Kubernetes 调度器是一个控制面进程，负责将 Pods 指派到节点上。 调度器基于约束和可用资源为调度队列中每个 Pod 确定其可合法放置的节点。 调度器之后对所有合法的节点进行排序，将 Pod 绑定到一个合适的节点。

3.master节点上的`kube-apiserver`：这个就是我们课上所解释的`api-server`，API 服务器验证并配置 API 对象的数据， 这些对象包括 pods、services、replicationcontrollers 等。 API 服务器为 REST 操作提供服务，并为集群的共享状态提供前端， 所有其他组件都通过该前端进行交互。

4.master节点上的`etcd`进程：`etcd` 是一个分布式键值存储系统，通常用于存储 Kubernetes 集群的配置信息、状态信息等数据。

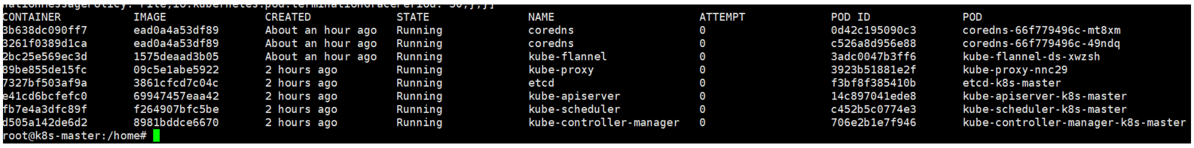
5.`kubelet`：kubelet 是在每个 Node 节点上运行的主要 “节点代理”。它可以使用以下之一向`apiserver` 注册： 主机名（hostname）； 覆盖主机名的参数；某云驱动的特定逻辑。

6.`kube-proxy`：这是k8s的网络代理，在每个节点上运行。网络代理反映了每个节点上 Kubernetes API 中定义的服务，并且可以执行简单的 TCP、UDP 和 SCTP 流转发，或者在一组后端进行循环 TCP、UDP 和 SCTP 转发。

7.flanneld 进程：`flanneld` 是 Kubernetes 集群中一个网络管理组件，主要负责网络的管理和配置，与我们下载的flannel插件有关。

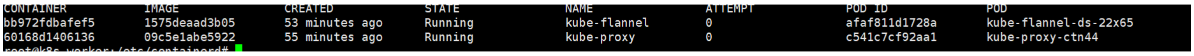
Q3: 在两个节点中分别使用 crictl ps 显示所有正常运行的containerd容器，记录其输出，并简要 说明各个容器所包含的k8s组件，以及那些k8s组件未运行在容器中

回答：master节点的显示：



```
CONTAINER           IMAGE               CREATED             STATE               NAME                      ATTEMPT             POD ID              POD
3b638dc090ff7       ead0a4a53df89       About an hour ago   Running             coredns                   0                   0d42c195090c3       coredns-66f779496c-mt8xm
3261f0389d1ca       ead0a4a53df89       About an hour ago   Running             coredns                   0                   c526a8d956e88       coredns-66f779496c-49ndq
2bc25e569ec3d       1575deaad3b05       About an hour ago   Running             kube-flannel              0                   3adc0047b3ff6       kube-flannel-ds-xwzsh
89be855de15fc       09c5e1abe5922       2 hours ago         Running             kube-proxy                0                   3923b51881e2f       kube-proxy-nnc29
7327bf503af9a       3861cfcd7c04c       2 hours ago         Running             etcd                      0                   f3bf8f385410b       etcd-k8s-master
e41cd6bcfefc0       69947457eaa42       2 hours ago         Running             kube-apiserver            0                   14c897041ede8       kube-apiserver-k8s-master
fb7e4a3dfc89f       f264907bfc5be       2 hours ago         Running             kube-scheduler            0                   c452b5c0774e3       kube-scheduler-k8s-master
d505a142de6d2       8981bddce6670       2 hours ago         Running             kube-controller-manager   0                   706e2b1e7f946       kube-controller-manager-k8s-master
root@k8s-master:/home#
```

worker节点的显示：



```
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT             POD ID              POD
bb972fdbafef5       1575deaad3b05       53 minutes ago      Running             kube-flannel        0                   afaf811d1728a       kube-flannel-ds-22x65
60168d1406136       09c5e1abe5922       55 minutes ago      Running             kube-proxy          0                   c541c7cf92aa1       kube-proxy-ctn44
root@k8s-worker:/etc/containerd#
```

`coredns`组件：`CoreDNS` 是 Kubernetes 集群中一个常用的 DNS 服务器组件，用于解析主机名到 IP 地址。它主要负责为集群内部服务提供 DNS 解析服务，以便实现服务之间的互相发现和通信。

`kube-flannel`组件：`kube-flannel` 是 Kubernetes 集群中一个常用的网络插件组件，用于实现网络的配置和管理。它通常与 `flanneld`、`etcd` 等其他组件配合使用，以确保集群内网络的正确运行和通信。

其它组件的说明与Q2中的类似，这里不重复赘述。

像`kubelet`这种就没有运行在容器中，通常是直接运行在 Kubernetes 集群的节点上，它会监听 API 服务器的指令，执行 Pod 的创建、销毁、重启等操作，并监控容器的健康状态等。

Q4: 请采用声明式接口对Pod进行部署，并将部署所需的yaml文件记录在实践文档中

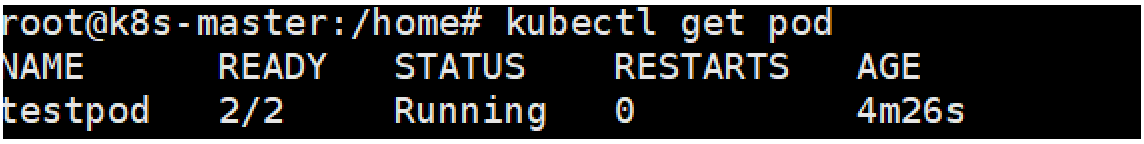
我们的test_k8s.yaml文件如下：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  labels:
    app: testpod
spec:
  containers:
    - name: fileserver
      image: 7143192/fileserver:latest
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: test-volume
          mountPath: /usr/share/files
    - name: downloader
      image: 7143192/downloader:latest
      ports:
        - containerPort: 3000
      volumeMounts:
        - name: test-volume
          mountPath: /data
  volumes:
    - name: test-volume
      hostPath:
        path: /home/data
        type: DirectoryOrCreate
```

我们调用指令

```
kubectl apply -f test_k8s.yaml
```

一段时间后查看pod状态

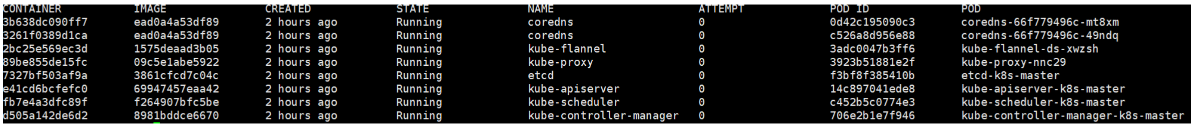


```
root@k8s-master:/home# kubectl get pod
NAME       READY   STATUS    RESTARTS   AGE
testpod    2/2     Running   0          4m26s
```

可以看到有两个pod正在运行。

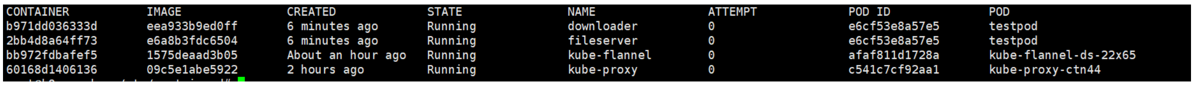
Q5: 请在worker节点上，在部署Pod的前后分别采用 crictl ps 查看所有运行中的容器并对比两者 的区别。请将创建该Pod所创建的全部新容器列举出来，并一一解释其作用

回答：master节点上



```
CONTAINER       IMAGE           CREATED         STATE     NAME                      ATTEMPT   POD ID          POD
3b638dc090ff7   ead0a4a53df89   2 hours ago     Running   coredns                   0         0d42c195090c3   coredns-66f779496c-mt8xm
3261f0389d1ca   ead0a4a53df89   2 hours ago     Running   coredns                   0         c526a8d956e88   coredns-66f779496c-49ndq
2bc25e569ec3d   1575deaad3b05   2 hours ago     Running   kube-flannel              0         3adc0047b3ff6   kube-flannel-ds-xwzsh
89be855de15fc   09c5e1abe5922   2 hours ago     Running   kube-proxy                0         3923b51881e2f   kube-proxy-nnc29
7327bf503af9a   3861cfcd7c04c   2 hours ago     Running   etcd                      0         f3bf8f385410b   etcd-k8s-master
e41cd6bcfefc0   69947457eaa42   2 hours ago     Running   kube-apiserver            0         14c897041ede8   kube-apiserver-k8s-master
fb7e4a3dfc89f   f264907bfc5be   2 hours ago     Running   kube-scheduler            0         c452b5c0774e3   kube-scheduler-k8s-master
d505a142de6d2   8981bddce6670   2 hours ago     Running   kube-controller-manager   0         706e2b1e7f946   kube-controller-manager-k8s-master
```

worker节点上



```
CONTAINER       IMAGE           CREATED             STATE     NAME           ATTEMPT   POD ID          POD
b971dd036333d   eea933b9ed0ff   6 minutes ago       Running   downloader     0         e6cf53e8a57e5   testpod
2bb4d8a64ff73   e6a8b3fdc6504   6 minutes ago       Running   fileserver     0         e6cf53e8a57e5   testpod
bb972fdbafef5   1575deaad3b05   About an hour ago   Running   kube-flannel   0         afaf811d1728a   kube-flannel-ds-22x65
60168d1406136   09c5e1abe5922   2 hours ago         Running   kube-proxy     0         c541c7cf92aa1   kube-proxy-ctn44
```

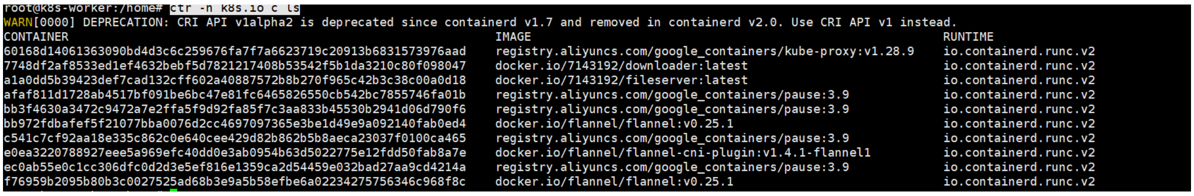
与我们在Q3中打印出的信息对比，发现master节点上没有变化，新创建的pod运行在了worker节点上。容器的作用在前面已经解释过，这里只解释worker节点新增的两个节点的作用：

如这两个容器的名字所展示的一样，一个是downloader，另外一个是fileserver。

这里很奇怪的一点是没有pause容器，Pause容器是Kubernetes系统自动生成的一个特殊容器，通常用于实现Pod的网络命名空间共享和挂载共享卷，在Kubernetes中起到连接和协调其他容器的作用，确保Pod能够正常运行并实现容器之间的共享和通信。

经过网上的查阅，在https://www.saoniuhuo.com/question/detail-2150987.html中查到了原因，

我们用`ctr -n k8s.io c ls`查看所有运行在k8s上的`containerd`容器，显示如下：



```
root@k8s-worker:/home# ctr -n k8s.io c ls
WARN[0000] DEPRECATION: CRI API v1alpha2 is deprecated since containerd v1.7 and removed in containerd v2.0. Use CRI API v1 instead.
CONTAINER                                                           IMAGE                                                     RUNTIME
60168d14061363090bd4d3c6c259676fa7f7a6623719c20913b6831573976aad    registry.aliyuncs.com/google_containers/kube-proxy:v1.28.9    io.containerd.runc.v2
7748df2af8533ed1ef4632bebf5d7821217408b53542f5b1da3210c80f098047    docker.io/7143192/downloader:latest                       io.containerd.runc.v2
a1a0dd5b39423def7cad132cff602a40887572b8b270f965c42b3c38c00a0d18    docker.io/7143192/fileserver:latest                       io.containerd.runc.v2
afaf811d1728ab4517bf091be6bc47e81fc6465826550cb542bc7855746fa01b    registry.aliyuncs.com/google_containers/pause:3.9         io.containerd.runc.v2
bb3f4630a3472c9472a7e2ffa5f9d92fa85f7c3aa833b45530b2941d06d790f6    registry.aliyuncs.com/google_containers/pause:3.9         io.containerd.runc.v2
bb972fdbafef5f21077bba0076d2cc4697097365e3be1d49e9a092140fab0ed4    docker.io/flannel/flannel:v0.25.1                         io.containerd.runc.v2
c541c7cf92aa18e335c862c0e640cee429d82b862b5b8aeca23037f0100ca465    registry.aliyuncs.com/google_containers/pause:3.9         io.containerd.runc.v2
e0ea3220788927eee5a969efc40dd0e3ab0954b63d5022775e12fdd50fab8a7e    docker.io/flannel/flannel-cni-plugin:v1.4.1-flannel1      io.containerd.runc.v2
ec0ab55e0c1cc306dfc0d2d3e5ef816e1359ca2d54459e032bad27aa9cd4214a    registry.aliyuncs.com/google_containers/pause:3.9         io.containerd.runc.v2
f76959b2095b80b3c0027525ad68b3e9a5b58efbe6a02234275756346c968f8c    docker.io/flannel/flannel:v0.25.1                         io.containerd.runc.v2
```

这样就看到了我们的pause容器，所以pause容器也是存在的。

Q6: 请结合博客 https://blog.51cto.com/u_15069443/4043930 的内容，将容器中的veth与host机器 上的veth匹配起来，并采用 ip link 和 ip addr 指令找到位于host机器中的所有网络设备及其之 间的关系。结合两者的输出，试绘制出worker节点中涉及新部署Pod的所有网络设备和其网络结构， 并在图中标注出从master节点中使用pod ip访问位于worker节点中的Pod的网络路径

回答：通过以下指令进入容器

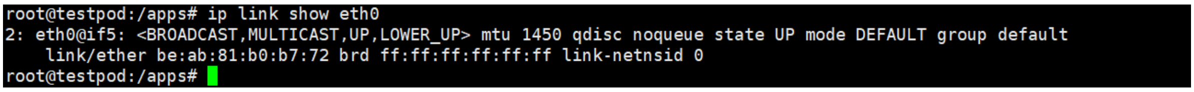
```
# 这个是downloader容器
crictl exec -it b971dd036333d /bin/bash
# 这个是fileserver容器
crictl exec -it 2bb4d8a64ff73 /bin/bash
```

进入后执行

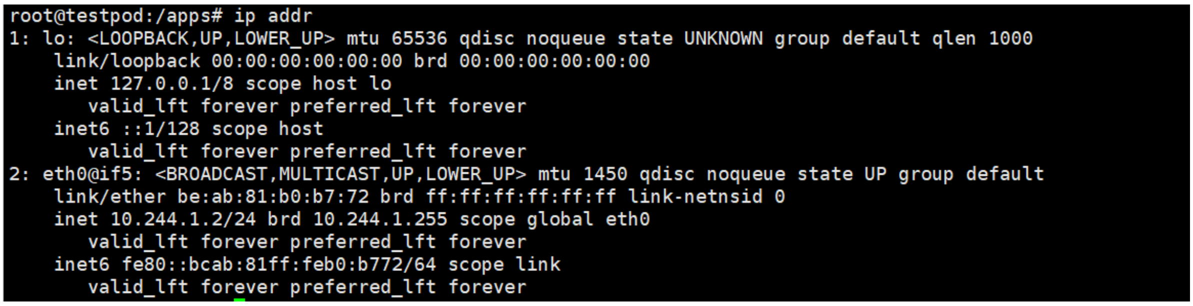
```
apt-get update
apt-get install -y iproute2
```

安装对应的工具，然后参照博客

downloader中：

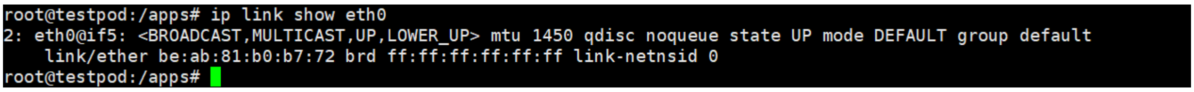


```
root@testpod:/apps# ip link show eth0
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default
    link/ether be:ab:81:b0:b7:72 brd ff:ff:ff:ff:ff:ff link-netnsid 0
root@testpod:/apps#
```

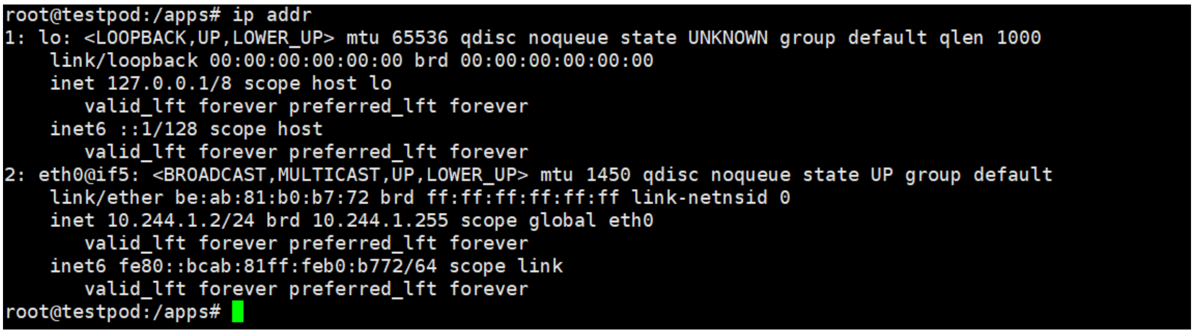


```
root@testpod:/apps# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether be:ab:81:b0:b7:72 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.1.2/24 brd 10.244.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::bcab:81ff:feb0:b772/64 scope link
       valid_lft forever preferred_lft forever
```

fileserver中：

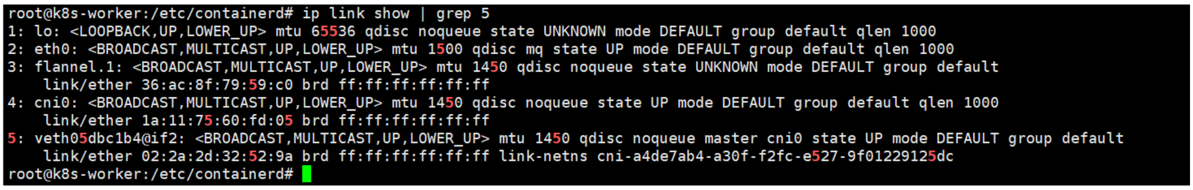


```
root@testpod:/apps# ip link show eth0
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default
    link/ether be:ab:81:b0:b7:72 brd ff:ff:ff:ff:ff:ff link-netnsid 0
root@testpod:/apps#
```

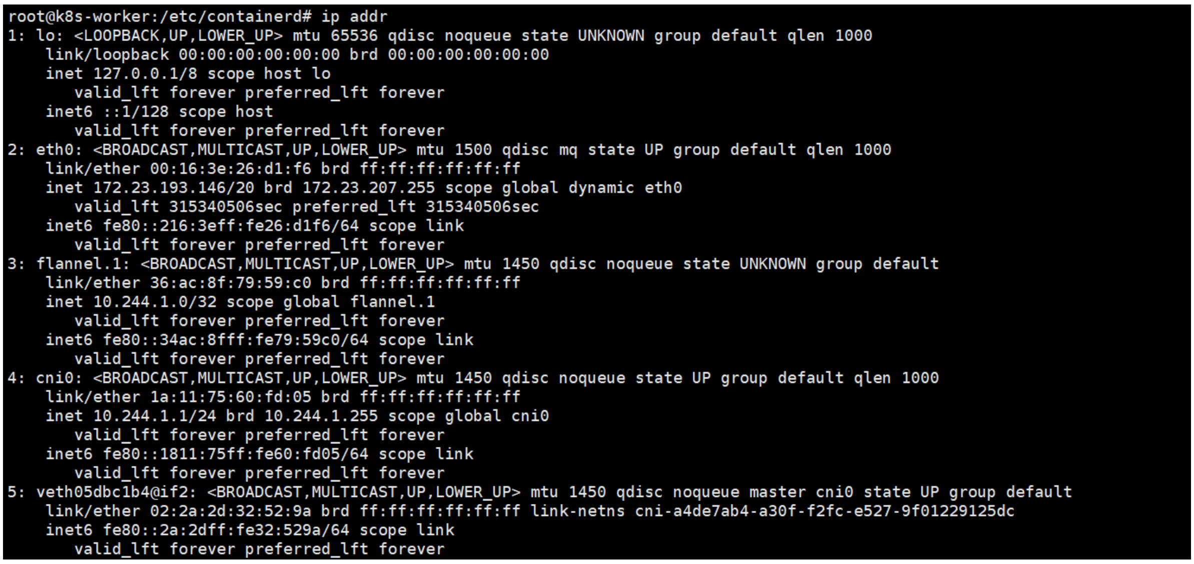


```
root@testpod:/apps# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether be:ab:81:b0:b7:72 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.1.2/24 brd 10.244.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::bcab:81ff:feb0:b772/64 scope link
       valid_lft forever preferred_lft forever
root@testpod:/apps#
```

在host主机中：



```
root@k8s-worker:/etc/containerd# ip link show | grep 5
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/ether 36:ac:8f:79:59:c0 brd ff:ff:ff:ff:ff:ff
4: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 1a:11:75:60:fd:05 brd ff:ff:ff:ff:ff:ff
5: veth05dbc1b4@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether 02:2a:2d:32:52:9a brd ff:ff:ff:ff:ff:ff link-netns cni-a4de7ab4-a30f-f2fc-e527-9f01229125dc
root@k8s-worker:/etc/containerd#
```



```
root@k8s-worker:/etc/containerd# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:16:3e:26:d1:f6 brd ff:ff:ff:ff:ff:ff
    inet 172.23.193.146/20 brd 172.23.207.255 scope global dynamic eth0
       valid_lft 315340506sec preferred_lft 315340506sec
    inet6 fe80::216:3eff:fe26:d1f6/64 scope link
       valid_lft forever preferred_lft forever
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 36:ac:8f:79:59:c0 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::34ac:8fff:fe79:59c0/64 scope link
       valid_lft forever preferred_lft forever
4: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether 1a:11:75:60:fd:05 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.1/24 brd 10.244.1.255 scope global cni0
       valid_lft forever preferred_lft forever
    inet6 fe80::1811:75ff:fe60:fd05/64 scope link
       valid_lft forever preferred_lft forever
5: veth05dbc1b4@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP group default
    link/ether 02:2a:2d:32:52:9a brd ff:ff:ff:ff:ff:ff link-netns cni-a4de7ab4-a30f-f2fc-e527-9f01229125dc
    inet6 fe80::2a:2dff:fe32:529a/64 scope link
       valid_lft forever preferred_lft forever
```

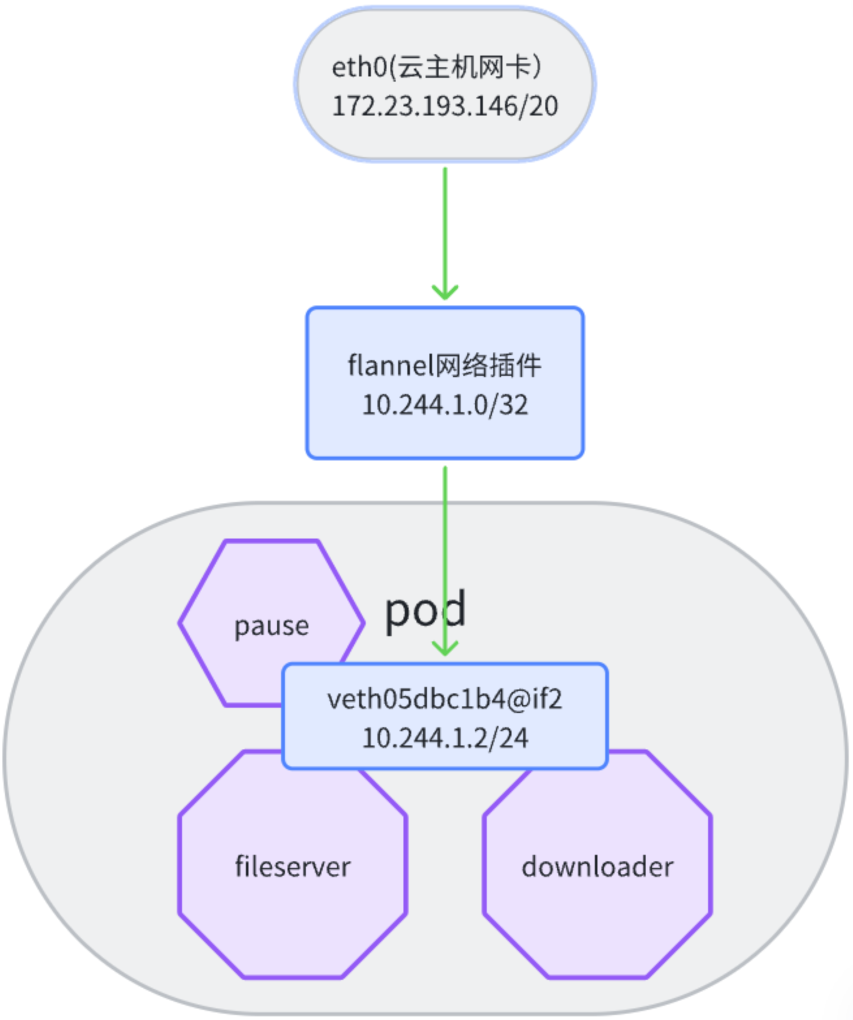
查看对应5的`vethinterface`是哪一个

好，这里我们就知道这是`veth-pair`技术，是一对虚拟的网络设备接口，成对出现。一端链接主机，一端链接容器，这样可以让它们之间直接通讯。

我们在上面主机执行的时候，看到的是`5: veth05dbc1b4@if2`,2是它pair配对的`veth`的索引index，这里就是我们之前在两个容器之中看到的`2: eth0@if5`,刚好我们发现它配对的veth的索引是5，与上面的恰好顺序相反。

同一个网段间的设备之间可以相互通信，我们在主机中可以很轻松找到flannel网络插件对应的虚拟网卡，另外，流量的转发也一定离不开服务器自带的虚拟网卡eth0(节点的网络接口），对应服务器的内网`ip`

拓扑图大致如下：



Q7: 请采用声明式接口对Deployment进行部署，并将Deployment所需要的yaml文件记录在文档中

回答：我们的`test_k8s_deployment.yaml`如下写法：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: testpod
  template:
    metadata:
      labels:
        app: testpod
    spec:
      containers:
        - name: fileserver
          image: 7143192/fileserver:latest
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: test-volume
              mountPath: /usr/share/files
        - name: downloader
          image: 7143192/downloader:latest
          ports:
            - containerPort: 3000
          volumeMounts:
            - name: test-volume
              mountPath: /data
      volumes:
        - name: test-volume
          hostPath:
            path: /home/data2
            type: DirectoryOrCreate
```

我们在master节点上使用如下指令：

```
root@k8s-master:/home# kubectl apply -f test_k8s_deployment.yaml
deployment.apps/test-deployment created
root@k8s-master:/home# kubectl get deployment
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
test-deployment   3/3     3            3           23s
```

我们可以看到成功创建

通过get pod

```
root@k8s-master:/home# kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
test-deployment-754f9f879-pzlv7   2/2     Running   0          34s
test-deployment-754f9f879-rfm6b   2/2     Running   0          34s
test-deployment-754f9f879-tprpj   2/2     Running   0          34s
testpod                           2/2     Running   0          94m
```

因为我们设置的replicas为3，所以会产生3个pod。

Q8: 在该使用Deployment的部署方式下，不同Pod之间的文件是否共享？该情况会在实际使用文件下 载与共享服务时产生怎样的实际效果与问题？应如何解决这一问题？

回答：默认情况下容器的文件系统是互相隔离的，但是我们在`yaml`文件中声明了需要绑定的Volume，并且我们将其挂载在了主机下，所以最后会导致挂载的那一部分目录内容是共享的。

这可能会导致冲突产生，如文件名冲突（你写的一个文件名可能已经被别人写了，那么就不能以这个文件名在作为文件了），甚至一个用户（对于某个特定的downloader）可以下载到别人的文件（可能是别的fileserver写的），因为数据共享，导致安全性问题。所以我们需要使用PV（持久化卷）解决上面的问题，它是对底层共享存储的一种抽象，将共享存储定义为一种资源，不属于任何的namespace，属于集群级别资源。用户使用PV需要通过`PersistentVolumeClaim`（PVC），我们在使用PV可以通过设置不同的参数来解决共享同一个volume的问题。

例如下面这样：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: unique-volume-deployment
spec:
  replicas: 2
```

```yaml
  selector:
    matchLabels:
      app: unique-volume
  template:
    metadata:
      labels:
        app: unique-volume
    spec:
      containers:
        - name: unique-volume-container
          image: your-image
          volumeMounts:
            - name: unique-volume
              mountPath: /path/to/data
      volumes:
        - name: unique-volume
          persistentVolumeClaim:
            claimName: unique-pvc-$(podname) # 使用 podname 动态生成不同的 PVC 名称
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: unique-pvc-$(podname) # 使用 podname 动态生成不同的 PVC 名称
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

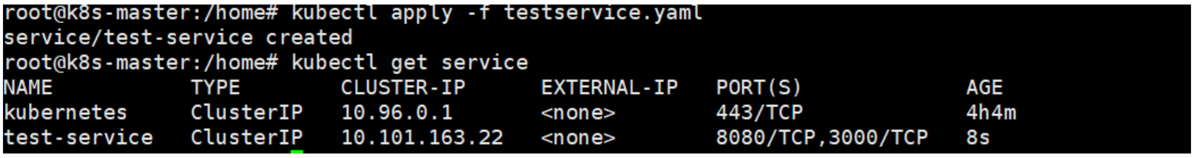
这样就能根据不同的podname生成不同的PVC从而每个pod共享自己独有的volume

Q9: 请采用声明式接口对Service进行部署，并将部署所需的yaml文件记录在实践文档中

回答：我们的testservice.yaml文件如下：

```yaml
apiVersion: v1
kind: Service
metadata:
  name: test-service
spec:
  selector:
    app: testpod
  ports:
    - port: 8080
      targetPort: 8080
      name: fileserver
    - port: 3000
      targetPort: 3000
      name: downloader
```

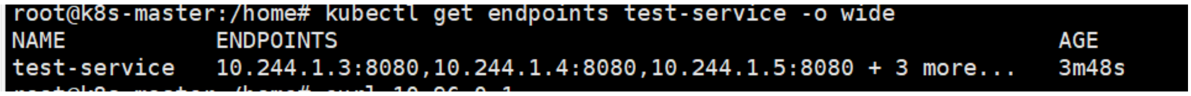
在master节点上运行如下指令：



```
root@k8s-master:/home# kubectl apply -f testservice.yaml
service/test-service created
root@k8s-master:/home# kubectl get service
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP             4h4m
test-service   ClusterIP   10.101.163.22   <none>        8080/TCP,3000/TCP   8s
```

我们完成了名为test-service服务的部署。

通过

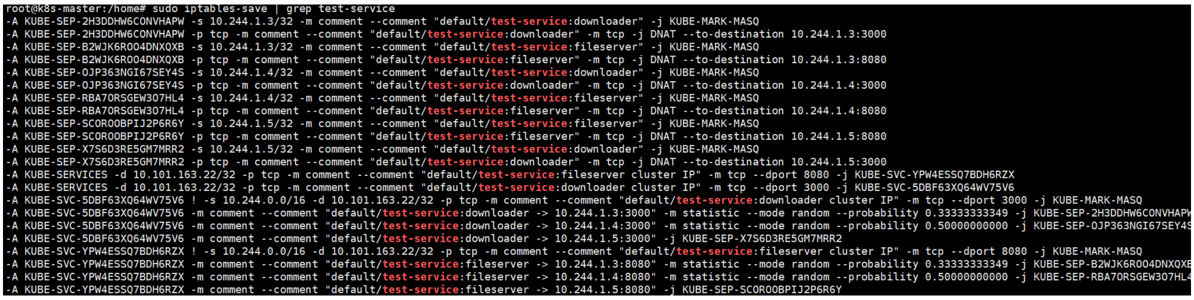


```
root@k8s-master:/home# kubectl get endpoints test-service -o wide
NAME           ENDPOINTS                                                    AGE
test-service   10.244.1.3:8080,10.244.1.4:8080,10.244.1.5:8080 + 3 more...   3m48s
```

也可以看到绑定对应的endpoints成功。

Q10: 请在master节点中使用 iptables-save 指令输出所有的iptables规则，将其中与Service访问 相关的iptable规则记录在实践文档中，并解释网络流量是如何采用基于iptables的方式被从对Service 的clusterIP的访问定向到实际的Pod中的，又是如何实现负载均衡到三个pod的。

回答：调用指令终端打印如下：



```
root@k8s-master:/home# sudo iptables-save | grep test-service
-A KUBE-SEP-2H3DDHW6CONVHAPW -s 10.244.1.3/32 -m comment --comment "default/test-service:downloader" -j KUBE-MARK-MASQ
-A KUBE-SEP-2H3DDHW6CONVHAPW -p tcp -m comment --comment "default/test-service:downloader" -m tcp -j DNAT --to-destination 10.244.1.3:3000
-A KUBE-SEP-B2WJK6ROO4DNXQXB -s 10.244.1.3/32 -m comment --comment "default/test-service:fileserver" -j KUBE-MARK-MASQ
-A KUBE-SEP-B2WJK6ROO4DNXQXB -p tcp -m comment --comment "default/test-service:fileserver" -m tcp -j DNAT --to-destination 10.244.1.3:8080
-A KUBE-SEP-OJP363NGI67SEY4S -s 10.244.1.4/32 -m comment --comment "default/test-service:downloader" -j KUBE-MARK-MASQ
-A KUBE-SEP-OJP363NGI67SEY4S -p tcp -m comment --comment "default/test-service:downloader" -m tcp -j DNAT --to-destination 10.244.1.4:3000
-A KUBE-SEP-RBA7ORSGEW3O7HL4 -s 10.244.1.4/32 -m comment --comment "default/test-service:fileserver" -j KUBE-MARK-MASQ
-A KUBE-SEP-RBA7ORSGEW3O7HL4 -p tcp -m comment --comment "default/test-service:fileserver" -m tcp -j DNAT --to-destination 10.244.1.4:8080
-A KUBE-SEP-SCOROOBPIJ2P6R6Y -s 10.244.1.5/32 -m comment --comment "default/test-service:fileserver" -j KUBE-MARK-MASQ
-A KUBE-SEP-SCOROOBPIJ2P6R6Y -p tcp -m comment --comment "default/test-service:fileserver" -m tcp -j DNAT --to-destination 10.244.1.5:8080
-A KUBE-SEP-X7S6D3RE5GM7MRR2 -s 10.244.1.5/32 -m comment --comment "default/test-service:downloader" -j KUBE-MARK-MASQ
-A KUBE-SEP-X7S6D3RE5GM7MRR2 -p tcp -m comment --comment "default/test-service:downloader" -m tcp -j DNAT --to-destination 10.244.1.5:3000
-A KUBE-SERVICES -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:fileserver cluster IP" -m tcp --dport 8080 -j KUBE-SVC-YPW4ESSQ7BDH6RZX
-A KUBE-SERVICES -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:downloader cluster IP" -m tcp --dport 3000 -j KUBE-SVC-5DBF63XQ64WV75V6
-A KUBE-SVC-5DBF63XQ64WV75V6 ! -s 10.244.0.0/16 -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:downloader cluster IP" -m tcp --dport 3000 -j KUBE-MARK-MASQ
-A KUBE-SVC-5DBF63XQ64WV75V6 -m comment --comment "default/test-service:downloader -> 10.244.1.3:3000" -m statistic --mode random --probability 0.33333333349 -j KUBE-SEP-2H3DDHW6CONVHAPW
-A KUBE-SVC-5DBF63XQ64WV75V6 -m comment --comment "default/test-service:downloader -> 10.244.1.4:3000" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-OJP363NGI67SEY4S
-A KUBE-SVC-5DBF63XQ64WV75V6 -m comment --comment "default/test-service:downloader -> 10.244.1.5:3000" -j KUBE-SEP-X7S6D3RE5GM7MRR2
-A KUBE-SVC-YPW4ESSQ7BDH6RZX ! -s 10.244.0.0/16 -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:fileserver cluster IP" -m tcp --dport 8080 -j KUBE-MARK-MASQ
-A KUBE-SVC-YPW4ESSQ7BDH6RZX -m comment --comment "default/test-service:fileserver -> 10.244.1.3:8080" -m statistic --mode random --probability 0.33333333349 -j KUBE-SEP-B2WJK6ROO4DNXQXB
-A KUBE-SVC-YPW4ESSQ7BDH6RZX -m comment --comment "default/test-service:fileserver -> 10.244.1.4:8080" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-RBA7ORSGEW3O7HL4
-A KUBE-SVC-YPW4ESSQ7BDH6RZX -m comment --comment "default/test-service:fileserver -> 10.244.1.5:8080" -j KUBE-SEP-SCOROOBPIJ2P6R6Y
```

这里我们可以看到有三个比较关键的部分：`KUBE_SVC`,`KUBE_SEP`,`KUBE_SERVICES`

首先`KUBE_SERVICES`是位于最外层，暴露服务的端口，两行规则如下：

```
-A KUBE-SERVICES -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:fileserver cluster IP" -m tcp --dport 8080 -j KUBE-SVC-YPW4ESSQ7BDH6RZX
-A KUBE-SERVICES -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:downloader cluster IP" -m tcp --dport 3000 -j KUBE-SVC-5DBF63XQ64WV75V6
```

第一行规则指定了当目标IP地址为10.101.163.22且目标TCP端口为8080时，应该将流量重定向到名为KUBE-SVC-YPW4ESSQ7BDH6RZX的规则进行处理。这个规则对应着test-service的fileserver服务。第二行规则则指定了当目标IP地址为10.101.163.22且目标TCP端口为3000时，应该将流量重定向到名为KUBE-SVC-5DBF63XQ64WV75V6的规则进行处理。这个规则对应着test-service的downloader服务。

其次是`KUBE_SVC`，它的八行规则如下：

```
-A KUBE-SVC-5DBF63XQ64WV75V6 ! -s 10.244.0.0/16 -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:downloader cluster IP" -m tcp --dport 3000 -j KUBE-MARK-MASQ
-A KUBE-SVC-5DBF63XQ64WV75V6 -m comment --comment "default/test-service:downloader -> 10.244.1.3:3000" -m statistic --mode random --probability 0.33333333349 -j KUBE-SEP-2H3DDHW6CONVHAPW
-A KUBE-SVC-5DBF63XQ64WV75V6 -m comment --comment "default/test-service:downloader -> 10.244.1.4:3000" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-OJP363NGI67SEY4S
-A KUBE-SVC-5DBF63XQ64WV75V6 -m comment --comment "default/test-service:downloader -> 10.244.1.5:3000" -j KUBE-SEP-X7S6D3RE5GM7MRR2
-A KUBE-SVC-YPW4ESSQ7BDH6RZX ! -s 10.244.0.0/16 -d 10.101.163.22/32 -p tcp -m comment --comment "default/test-service:fileserver cluster IP" -m tcp --dport 8080 -j KUBE-MARK-MASQ
-A KUBE-SVC-YPW4ESSQ7BDH6RZX -m comment --comment "default/test-service:fileserver -> 10.244.1.3:8080" -m statistic --mode random --probability 0.33333333349 -j KUBE-SEP-B2WJK6ROO4DNXQXB
-A KUBE-SVC-YPW4ESSQ7BDH6RZX -m comment --comment "default/test-service:fileserver -> 10.244.1.4:8080" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-RBA7ORSGEW3O7HL4
-A KUBE-SVC-YPW4ESSQ7BDH6RZX -m comment --comment "default/test-service:fileserver -> 10.244.1.5:8080" -j KUBE-SEP-SCOROOBPIJ2P6R6Y

```

这里上面四条是处理downloader的，后面四条是处理fileserver的。

我们可以看到的是，流量转发到不同pod上的概率，就以前四条处理downloader的来说明，当流量到来时，转移到10.244.1.3:3000的概率为0.333（由probability 0.33333333349可知，因为有三个节点，所以概率是1/3），剩下0.67的概率会继续执行下面的规则，接着是转发到10.244.1.4：3000的概率为0.5（这里是我们的前一条规则以0.67的概率继续向下执行，这个时候有两个节点，所以概率是0.5），剩下0.5的概率继续执行下面的规则，最后是转发到10.244.1.5：3000的概率，这个时候已经经过前面两个规则（已经不会转发到10.244.1.3:3000和10.244.1.4：3000），剩下的概率自然是100%，上面的过程基本上保证了每个pod的负载均衡，基本是1/3，**这种均衡是随机的负载均衡，不是基于IP-Hash，也不是round-robin**。在决定好转发到那个pod后，可以看到这三行规则每个后面都有一个类似`-j KUBE-SEP-2H3DDHW6CONVHAPW`，也就是根据具体的pod转发到对应的`KUBE-SEP`,`KUBE-SEP`是 `KUBE-SVC` 对应的终端,它后面跟着具体的DNAT规则，例如：

```
-A KUBE-SEP-2H3DDHW6CONVHAPW -p tcp -m comment --comment "default/test-service:downloader" -m tcp -j DNAT --to-destination 10.244.1.3:3000

```

当流量匹配到这条规则时，会触发对应的处理动作。处理动作是进行目的地址网络地址转换（DNAT），将流量重定向到目标地址为10.244.1.3，目标端口为3000的目的IP地址。

简单来说，就是下面这样：

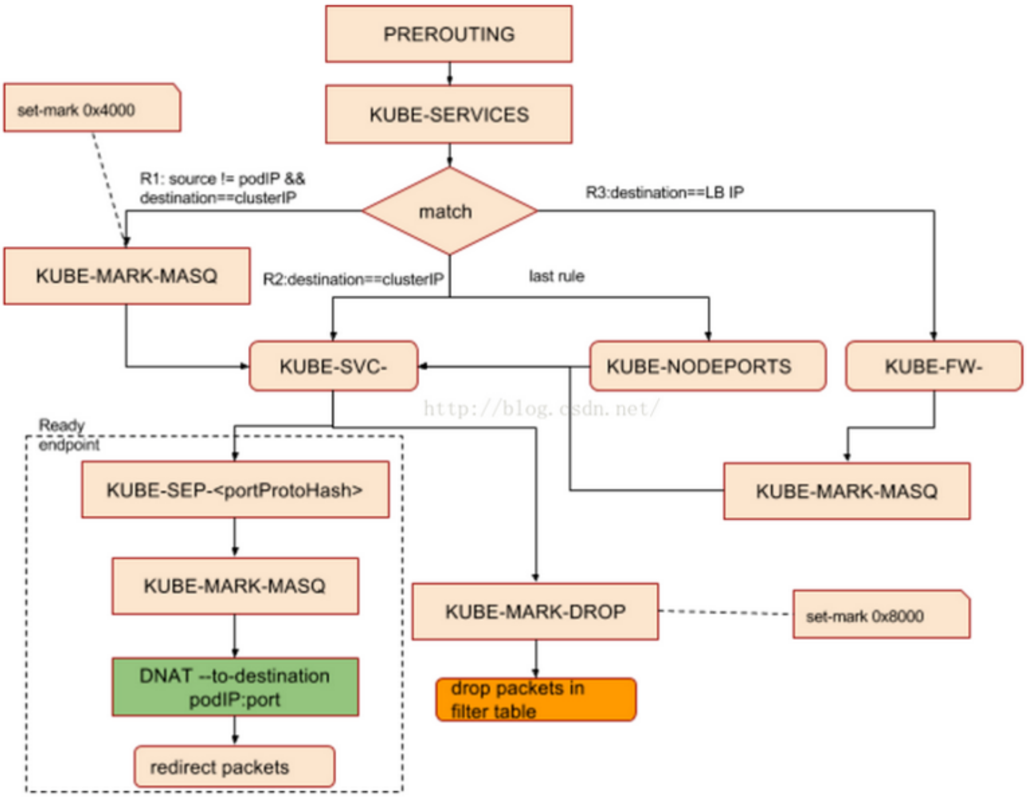
1.每个Service的每个服务端口都会在Chain KUBE-SERVICES中有一条对应的规则，发送到`clusterIP`的报文，将会转发到对应的Service的规则链，没有命中`ClusterIP`的，转发到KUBE-NODEPORTS。

只有发送到被`kubernetes`占用的端口的报文才会进入KUBE-MARK-MASQ打上标记，并转发到对应的服务规则链。

2.每一个KUBE_SERVICE，又将报文提交到了各自的KUBE-SEP-XXX

3.最后在KUBE-SEP-XX中完整了最终的DNAT，将目的地址转换成了POD的IP和端口。
这里的KUBE-MARK-MASQ为报文打上了标记，表示这个报文是由`kubernetes`管理的，`Kubernetes`将会对它进行NAT转换。

流程图像这样：



Q11: kube-proxy组件在整个service的定义与实现过程中起到了什么作用？请自行查找资料，并解释 在iptables模式下，kube-proxy的功能

回答：`kube-proxy`会部署在k8s集群中的每个node节点上，它能够实现k8s service的通信和负载均衡，`kube-proxy`负责帮助pod创建代理服务，从k8s的`api-server`获取server信息，并且根据信息创建代理服务，实现了从服务器端到pod的请求路由与转发，在iptables模式下，`kube-proxy`的功能如下：

1. 实现Service的负载均衡：`kube-proxy`会在节点上为每个Service创建相应的iptables规则，从而实现Service的负载均衡功能。这些规则会根据Service的类型（`ClusterIP`、`NodePort`、`LoadBalancer`）来进行配置，确保流量能够正确地路由到后端Pod上。
2. 实现Service的访问策略：`kube-proxy`也会根据Service的定义，实现相应的访问策略，如`SessionAffinity`策略，确保请求会被定向到相同的后端Pod上。
3. 实现Service的可达性：`kube-proxy`会监视后端Pod的健康状态，并根据实际情况更新相应的iptables规则，以确保流量不会被发送到不可达的后端Pod上。
4. 实现Service的服务发现：`kube-proxy`会监视Service和Endpoints的变化，随时更新iptables规则以确保新的Pod能够被正确地访问到。
5. 定时从`etcd`服务获取到service信息来做相应的策略，维护网络规则和四层负载均衡工作
6. 如同我们上面的例子，`kube-proxy`管理service的endpoints，并且让这个service对外暴露一个虚拟`ip`，也就是`CLUSTER-IP`，集群内部可以通过访问这个`ip`对应的端口就能顺利转发访问到集群内对应service的pod

Q12: 请在上面部署的Deployment的基础上为其配置HPA，并将部署所需的yaml文件记录在实践文档 中，如果对上面的Deployment配置有修改也请在文档中说明。具体参数为最大副本数为6，最小副本 数为3，目标cpu平均利用率为40%。

回答：首先我们在使用`hpa`之前需要先安装 Metrics Service插件。

安装过程如下：

```
root@k8s-master:/home# wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

```

这里我们使用`wget`拉取`components.yaml`文件，我们需要对里面的一些内容做一些修改（最主要的是修改镜像源）。

修改这个文件的对应内容如下（只展示修改部分）：

```
  spec:
   containers:
   - args:
    - --cert-dir=/tmp
    - --secure-port=10250
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --kubelet-use-node-status-port
    - --kubelet-insecure-tls
    - --metric-resolution=15s
    image: registry.cn-hangzhou.aliyuncs.com/google_containers/metrics-server:v0.7.1
```

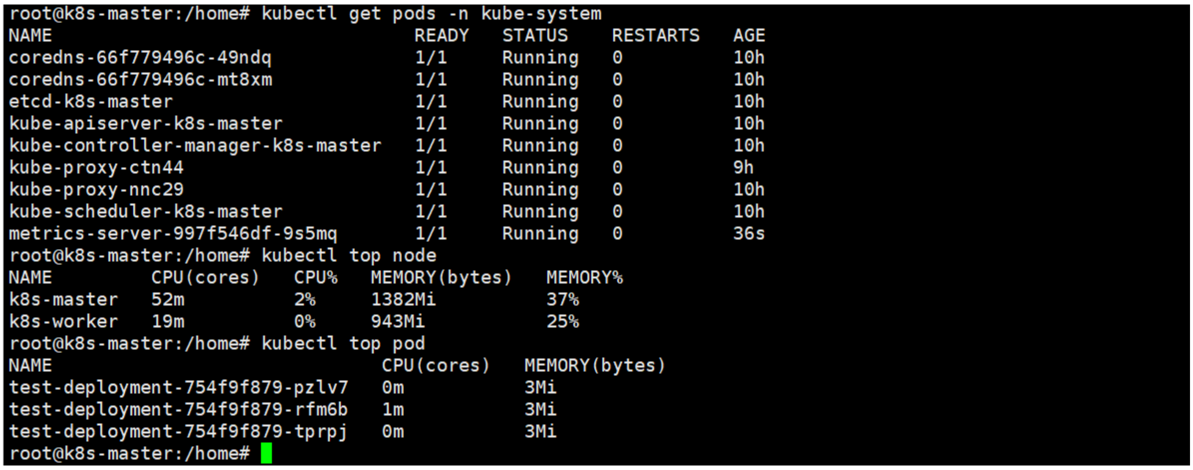
修改源为阿里，同时需要加上`kubelet-insecure-tls`不验证客户端证书。

接着运行：

```
root@k8s-master:/home# kubectl apply -f components.yaml
serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created

```

然后我们用get pod查看并且利用top node和top pod查看是否安装成功



```
root@k8s-master:/home# kubectl get pods -n kube-system
NAME                                  READY   STATUS    RESTARTS   AGE
coredns-66f779496c-49ndq              1/1     Running   0          10h
coredns-66f779496c-mt8xm              1/1     Running   0          10h
etcd-k8s-master                       1/1     Running   0          10h
kube-apiserver-k8s-master             1/1     Running   0          10h
kube-controller-manager-k8s-master    1/1     Running   0          10h
kube-proxy-ctn44                      1/1     Running   0          9h
kube-proxy-nnc29                      1/1     Running   0          10h
kube-scheduler-k8s-master             1/1     Running   0          10h
metrics-server-997f546df-9s5mq        1/1     Running   0          36s
root@k8s-master:/home# kubectl top node
NAME         CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
k8s-master   52m          2%     1382Mi          37%
k8s-worker   19m          0%     943Mi           25%
root@k8s-master:/home# kubectl top pod
NAME                               CPU(cores)   MEMORY(bytes)
test-deployment-754f9f879-pzlv7    0m           3Mi
test-deployment-754f9f879-rfm6b    1m           3Mi
test-deployment-754f9f879-tprpj    0m           3Mi
root@k8s-master:/home# 
```

至此Metrics Service插件安装成功。

下面来执行本部分的任务，我们修改原来的`test_k8s_deployment.yaml`文件如下：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deployment
spec:
  # replicas: 3
  selector:
    matchLabels:
      app: testpod
  template:
    metadata:
      labels:
        app: testpod
    spec:
      containers:
        - name: fileserver
          image: 7143192/fileserver:latest
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: test-volume
              mountPath: /usr/share/files
          resources:
            limits:
              cpu: "500m"
            requests:
              cpu: "200m"
        - name: downloader
          image: 7143192/downloader:latest
          ports:
            - containerPort: 3000
          volumeMounts:
            - name: test-volume
              mountPath: /data
          resources:
            limits:
              cpu: "500m"
            requests:
              cpu: "200m"
      volumes:
        - name: test-volume
          hostPath:
            path: /home/data2
            type: DirectoryOrCreate
```

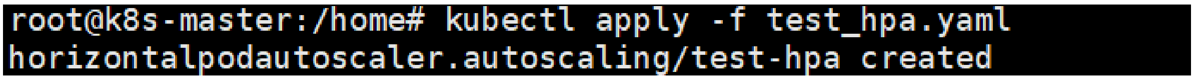
本质上的修改是我去掉了replicas字段，同时给每个container的资源分配做了一个要求和限制。

下面是test_hpa.yaml文件的内容：

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: test-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: test-deployment
  minReplicas: 3
  maxReplicas: 6
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 40
  behavior:
    scaleDown:
      policies:
        - type: Percent
          value: 10
          periodSeconds: 60
```

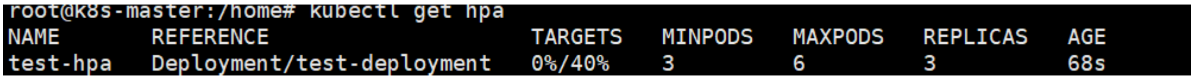
这里我们已经对Q12和Q13两个问题的要求都已经写入`yaml`文件中，其中Q12的为具体参数为最大副本数为6，最小副本 数为3，目标`cpu`平均利用率为40%，已经写在`scaleTargetRef`字段下，Q13的为`hpa`配置缩容的速率限制为每分钟 10%，已经写在behavior字段下。

运行时重新apply之前deployment的`yaml`文件后运行



```
root@k8s-master:/home# kubectl apply -f test_hpa.yaml
horizontalpodautoscaler.autoscaling/test-hpa created
```

最后通过get hpa查看



```
root@k8s-master:/home# kubectl get hpa
NAME       REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
test-hpa   Deployment/test-deployment 0%/40%    3         6         3          68s
```

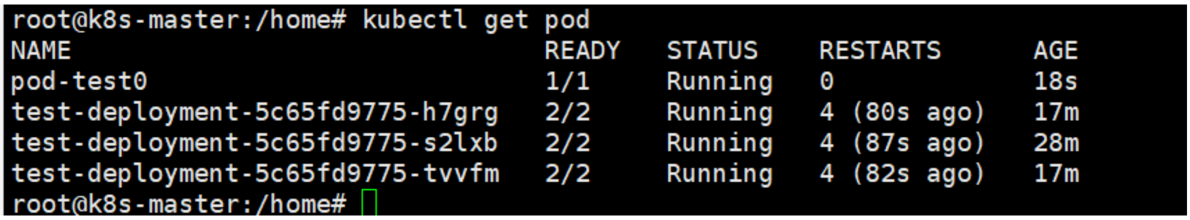
Q13:小明发现，hpa缩容过快在负载抖动的情况下会引起pod被不必要地删除和再次创建，因此他决定 限制缩容的速率，请为hpa配置缩容的速率限制为每分钟10%，并将部署所需的yaml文件记录在实践 文档中

回答：配置文件`test_hpa.yaml` Q12已经给出，相应设置已经在其中。现在我们想要感受一下HPA带来的水平伸缩，所以我们需要给我们的集群上一些压力。

我们运行下面的指令：

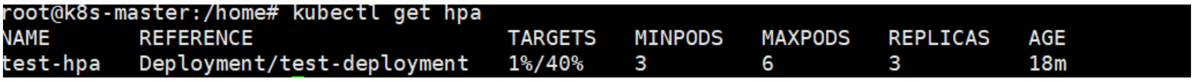
```
kubectl run "pod-test0" --image=busybox:latest --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://10.101.163.22:8080; done"
```

这个命令将创建一个临时的Pod对象，名称为"pod-test0"，使用`busybox:latest`镜像，并且设置不重新启动。这个Pod将会执行一个命令，不停地访问http://10.101.163.22:8080这个URL，可以测试http://10.101.163.22:8080上的负载。



```
root@k8s-master:/home# kubectl get pod
NAME                              READY   STATUS    RESTARTS      AGE
pod-test0                         1/1     Running   0             18s
test-deployment-5c65fd9775-h7grg  2/2     Running   4 (80s ago)   17m
test-deployment-5c65fd9775-s2lxb  2/2     Running   4 (87s ago)   28m
test-deployment-5c65fd9775-tvvfm  2/2     Running   4 (82s ago)   17m
root@k8s-master:/home#
```

可以看到的时已经在跑了，



```
root@k8s-master:/home# kubectl get hpa
NAME       REFERENCE                    TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
test-hpa   Deployment/test-deployment   1%/40%    3         6         3          18m
```

我们发现CPU利用率有所变化，但是显然还不够明显。

所以我们用下面的脚本test_add.sh：

```
for i in {1..80}
do
  podname="test$i"
  kubectl run "pod-$podname" --image=busybox:latest --restart=Never -- /bin/sh -c "while sleep 0.01;do wget -q -O- http://10.101.163.22:8080; done"
done
```

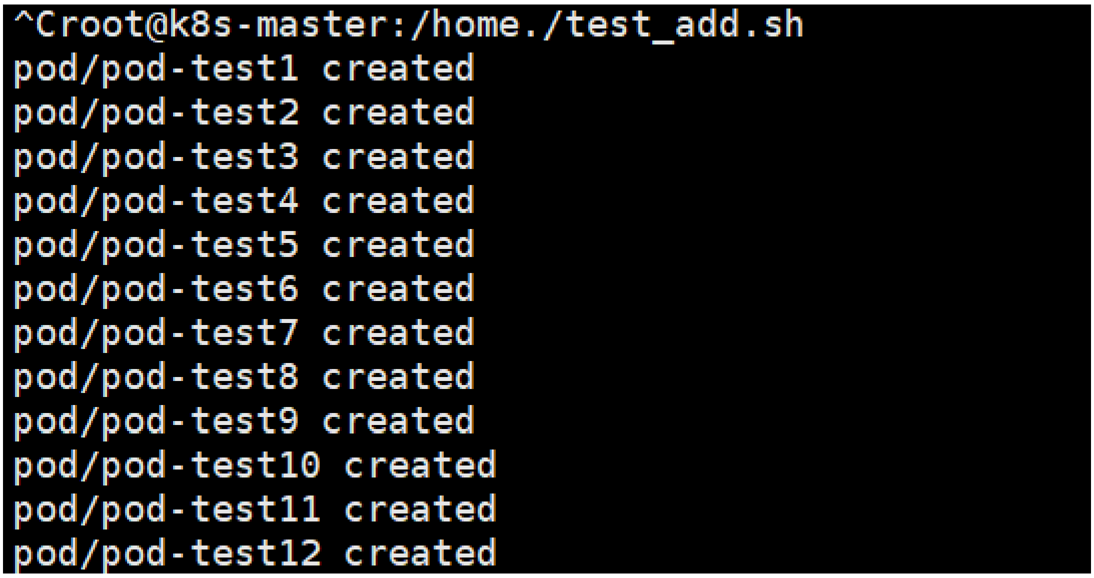
逻辑也非常简单，就是创建80个节点不停地访问http://10.101.163.22:8080这个URL，增加上面的负载。

同时我们修改`test_hpa.yaml`,修改如下：

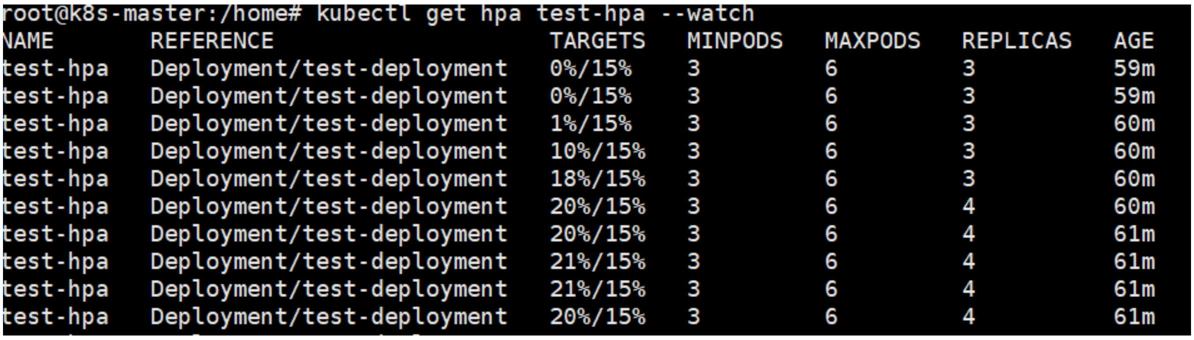
```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: test-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: test-deployment
  minReplicas: 3
  maxReplicas: 6
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 15
  behavior:
    scaleDown:
      policies:
        - type: Percent
          value: 10
          periodSeconds: 60
```

原来的40%要求优点高，我们尝试降低到15%，这样观察的效果更明显

效果如下：



```
^Croot@k8s-master:/home./test_add.sh
pod/pod-test1 created
pod/pod-test2 created
pod/pod-test3 created
pod/pod-test4 created
pod/pod-test5 created
pod/pod-test6 created
pod/pod-test7 created
pod/pod-test8 created
pod/pod-test9 created
pod/pod-test10 created
pod/pod-test11 created
pod/pod-test12 created
```



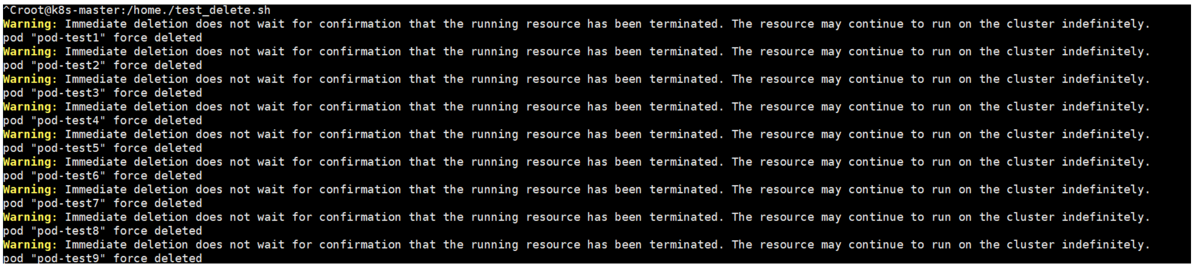
```
root@k8s-master:/home# kubectl get hpa test-hpa --watch
NAME       REFERENCE                    TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
test-hpa   Deployment/test-deployment   0%/15%    3         6         3          59m
test-hpa   Deployment/test-deployment   0%/15%    3         6         3          59m
test-hpa   Deployment/test-deployment   1%/15%    3         6         3          60m
test-hpa   Deployment/test-deployment   10%/15%   3         6         3          60m
test-hpa   Deployment/test-deployment   18%/15%   3         6         3          60m
test-hpa   Deployment/test-deployment   20%/15%   3         6         4          60m
test-hpa   Deployment/test-deployment   20%/15%   3         6         4          61m
test-hpa   Deployment/test-deployment   21%/15%   3         6         4          61m
test-hpa   Deployment/test-deployment   21%/15%   3         6         4          61m
test-hpa   Deployment/test-deployment   20%/15%   3         6         4          61m
```

我们可以看到对应的负载逐渐变高，当CPU利用率达到20时，发生了自动扩容，replicas的数量从3变到了4.

我们同样编写删除脚本test_delete.sh，内容如下：

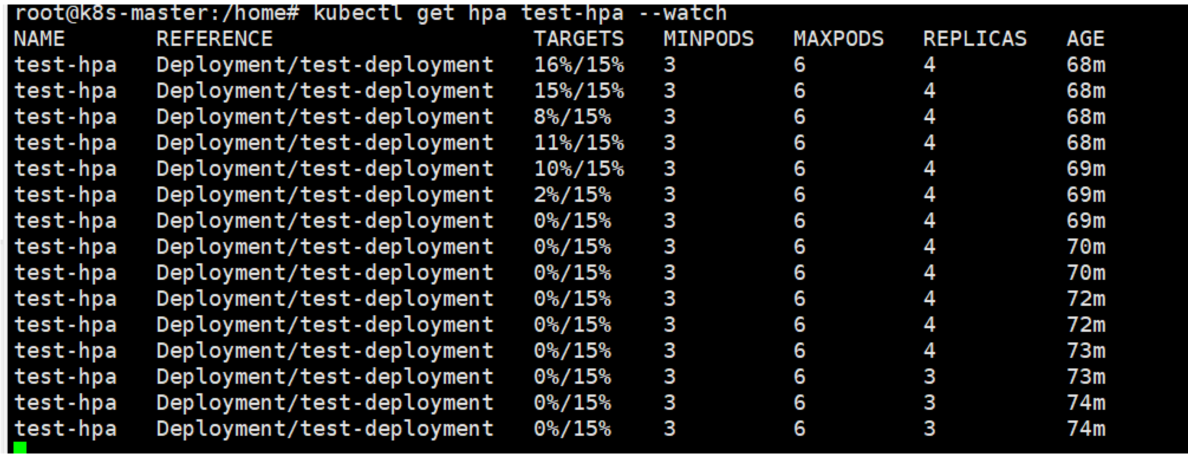
```
for i in {1..80}
do
  podname="pod-test${i}"
  kubectl delete pod ${podname} --grace-period=0 --force
done
```

也就是删除80个创建的pod，我们接着运行如下的指令。



```
^Croot@k8s-master:/home./test_delete.sh
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test1" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test2" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test3" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test4" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test5" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test6" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test7" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test8" force deleted
Warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-test9" force deleted
```

删除80个pod。



```
root@k8s-master:/home# kubectl get hpa test-hpa --watch
NAME       REFERENCE                    TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
test-hpa   Deployment/test-deployment   16%/15%   3         6         4          68m
test-hpa   Deployment/test-deployment   15%/15%   3         6         4          68m
test-hpa   Deployment/test-deployment   8%/15%    3         6         4          68m
test-hpa   Deployment/test-deployment   11%/15%   3         6         4          68m
test-hpa   Deployment/test-deployment   10%/15%   3         6         4          69m
test-hpa   Deployment/test-deployment   2%/15%    3         6         4          69m
test-hpa   Deployment/test-deployment   0%/15%    3         6         4          69m
test-hpa   Deployment/test-deployment   0%/15%    3         6         4          70m
test-hpa   Deployment/test-deployment   0%/15%    3         6         4          70m
test-hpa   Deployment/test-deployment   0%/15%    3         6         4          72m
test-hpa   Deployment/test-deployment   0%/15%    3         6         4          72m
test-hpa   Deployment/test-deployment   0%/15%    3         6         4          73m
test-hpa   Deployment/test-deployment   0%/15%    3         6         3          73m
test-hpa   Deployment/test-deployment   0%/15%    3         6         3          74m
test-hpa   Deployment/test-deployment   0%/15%    3         6         3          74m
```

CPU利用率逐渐降低，最后降到了0，这个时候`hpa`缩容了，也就是replicas的数量重新由4变回了3。至此已经可以说明我们的hpa配置部署成功。