K8S v1.12高可用集群搭建

目录

[开始之前 2](#_Toc530144995)

[环境 2](#_Toc530144996)

[架构 3](#_Toc530144997)

[特别说明 3](#_Toc530144998)

[准备环境 3](#_Toc530144999)

[创建虚拟机 3](#_Toc530145000)

[关闭swap 4](#_Toc530145001)

[重启生效 5](#_Toc530145002)

[安装kubeadm 5](#_Toc530145003)

[安装docker 5](#_Toc530145004)

[安装kubeadm 6](#_Toc530145005)

[搭建k8s高可用集群 6](#_Toc530145006)

[确定网络插件 7](#_Toc530145007)

[搭建高可用etcd集群 7](#_Toc530145008)

[生成根证书CA 8](#_Toc530145009)

[用kubeadm签发etcd证书 9](#_Toc530145010)

[安装etcd集群 11](#_Toc530145011)

[k8s master高可用架构原理 13](#_Toc530145012)

[搭建haproxy 14](#_Toc530145013)

[启动第1个master 15](#_Toc530145014)

[搭建第2个master 20](#_Toc530145015)

[搭建第3个master 24](#_Toc530145016)

[验证集群 25](#_Toc530145017)

[部署服务 25](#_Toc530145018)

[允许master运行pod 26](#_Toc530145019)

[验证pod网络 28](#_Toc530145020)

[验证高可用 29](#_Toc530145021)

[搭建成功 30](#_Toc530145022)

# 开始之前

## 环境

* Ubuntu16.04 虚拟机3台，1G内存\*1核
* K8s的1.12版本

## 架构

* 搭建3个节点的etcd集群
* 搭建3个节点的k8s高可用集群，均充当master+slave角色
* 使用flannel网络
* 搭建1个节点的haproxy充当k8s master集群的负载均衡

## 特别说明

* 以k8s官方教程为依据。
* 使用kubeadm搭建。

# 准备环境

官方要求每台机器至少2G内存2核，我这里凑活用1核1G的虚拟机搞一下，大家机器好最好把虚拟机资源给够。

## 创建虚拟机

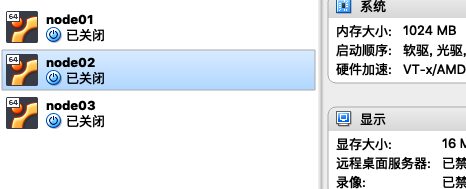
我们需要3个虚拟机。

通过virtualbox安装ubuntu 16.04虚拟机，安装的过程中，我指定了帐号与密码都是k8s。

安装好1个之后，配置virtualbox的网络模式为桥接到物理机网卡，这样虚拟机就会在物理机的局域网中分配到物理IP。



然后拷贝这个虚拟机，改一下名字，得到3个节点：



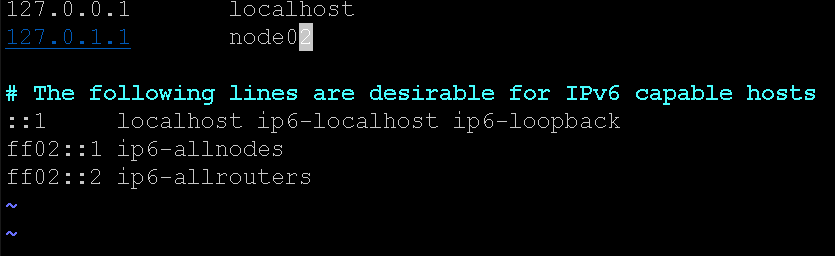
接下来我们启动这3个虚拟机，登录后发现它们的hostname都一样，ifconfig可以查看到它们的IP。

分别给3个虚拟机安装ssh服务端：

sudo apt-get install ssh -y

以后我们使用ssh终端连到3个虚拟机上操作，因为直接在虚拟机里操作很不方便。

K8s要求每台机器有唯一的Host，所以我们先分别修改3台机器的/etc/hostname，分别叫做node01，node02，node03，同时记得对应改一下它们的/etc/hosts文件中的解析：



## 关闭swap

K8s不允许系统用swap特性（磁盘当内存用），我们编辑一下/etc/fstab，注释掉swap盘的挂载配置：



## 重启生效

执行：sudo reboot，三台机器重启后上述修改全部生效。

# 安装kubeadm

接下来这一段步骤主要是搞定kubeadm工具，它用来简化后续我们k8s高可用集群的部署，参考的是官方文档：<https://kubernetes.io/docs/setup/independent/install-kubeadm/>。

请先sudo su root 切到root用户，方便我们安装。

## 安装docker

K8s基于CRI（Container Runtime Interface）规范支持多种不同的容器运行时实现。

我们就是用docker，k8s默认也是用docker，完整官方安装教程：<https://kubernetes.io/docs/setup/cri/>。

我们给3台机器安装一下docker，ubuntu repo默认版本就可以，最新的docker一般k8s是不支持的，k8s官方指导安装方法：

***# Install Docker from Ubuntu's repositories:***

**apt-get update**

**apt-get install -y docker.io**

验证一下安装效果，执行：

docker ps

有输出就成功了。

## 安装kubeadm

安装需要用apt-get完成，但是官方给的k8s源是google域下的，被墙了。

我们不能用官方的源，我们用阿里云的源：

先安装个这个：

apt-get update && apt-get install -y apt-transport-https

再加个阿里云的key：

curl https://mirrors.aliyun.com/kubernetes/apt/doc/apt-key.gpg | apt-key add –

把阿里云的k8s源保存到apt配置中（注意别拷贝EOF后面的空格）：

cat <<EOF >/etc/apt/sources.list.d/kubernetes.list

deb https://mirrors.aliyun.com/kubernetes/apt/ kubernetes-xenial main

EOF

拉一下阿里云的K8s列表：

apt-get update

安装kubeadm，kubelet，kubectl：

**apt-get install -y kubelet kubeadm kubectl**

kubeadm是搭建k8s集群的工具，kubelet是每台k8s节点上的第一个守护进程，kubectl是管理k8s集群的命令行工具。

下面这个命令可以不用执行，它是用来锁住这几个apt包的更新的，如果一旦手误更新了这些包，K8s集群就会因为版本不兼容挂了：

**apt-mark hold kubelet kubeadm kubectl**

# 搭建k8s高可用集群

为了真正能上生产，不仅k8s要搭多个master的高可用集群，k8s依赖的etcd也得是高可用集群。

官方基于kubeadm提供了2种搭建高可用k8s集群的方法，一种是k8s master和etcd放在一套机器上，另外一种是etcd集群单独抽出来部署，放到哪里都可以。

官方把k8s master叫做control plane，翻译叫飞机驾驶员？

经过研究，还是把etcd集群抽出来单独部署的方法更利于隔离与维护，所以我会采用第二种方案。

对于试图不用kubeadm搭建k8s的想法，我建议早点放弃。

## 确定网络插件

为了实现容器间的虚拟网络，K8S基于CNI规范接入了很多网络方案，比如calico，flannel，weave等，等我们的k8s集群搭建起来之后，只需要安装一下插件，k8s就会自动把网络环境生效到集群，非常简单。

现在我们还没搭起k8s集群，所以无法安装网络插件，但我们需要先选好插件，因为不同插件提供的虚拟网段不同，这会影响到我们对kubelet守护进程的配置。

打开官方链接：<https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/#pod-network>

我们用flannel即可，它要求把**--pod-network-cidr=10.244.0.0/16传给kubeadm init，我们只需要记住这个10.244.0.0/16网段即可，稍后我们在用kubeadm安装集群时会用到。**

## 搭建高可用etcd集群

因为k8s基于etcd保存所有信息，所以得先搭建etcd集群。

我只有3台虚拟机，所以etcd的高可用集群也会搭在上面，真正生产环境最好和k8s机器分开。

官方文档在搭建etcd这块具有一定误导性，不知道为什么给3个etcd节点签发了3套证书，这让k8s用哪个做client cert呢？所以接下来会和官方文档略有不同，但核心思路相同。（安装参考的官方文档：<https://kubernetes.io/docs/setup/independent/setup-ha-etcd-with-kubeadm/>）

我们安装etcd的机器也要安装一下kubeadm，我们需要用这个工具帮我们搞定etcd证书的签发工作，主要实现3个目的：

1. 保障k8s访问etcd时可以验证etcd是可信的，这一套证书是etcd服务端的cert。
2. etcd可以验证来访者的确是k8s，这一套证书是给apiserver提供的客户端cert。
3. etcd节点之间可以互相确认身份，这一头证书叫做peer Cert。

在签发这3套证书前，kubeadm会生成一个ca，用这个ca来签这3套证书，同样的ca最终会配置给k8s和etcd，这样证书才能基于这个ca做校验。

### 生成根证书CA

K8s配置里到处都是证书，这里讲一下怎么快速理解这个事情的作用。

世界上有一些权威机构，它们拥有一个自签的无敌根证书叫ca，全世界都信任它，这个证书可以用于生成其他证书，叫做cert。

开启https的网站（比如baidu）会去找权威用ca签一个证书cert给你，以后客户端访问你的网站，你就把证书cert返回客户端，客户端用ca证书就能校验一下你的cert合不合法，是不是真的baidu，这就可以防止钓鱼网站问题。

我们k8s要访问etcd，所以得确认etcd是不是我们搭的真货。我们也可以自签一个无敌根证书ca，然后用ca签发一个server cert给etcd配置上，这样k8s访问etcd时会收到etcd发来的cert，然后因为k8s配置保存了无敌证书ca，所以这时候用ca验一下cert就知道etcd是真的（因为是ca签的）。

反过来，etcd也只允许k8s访问，所以我们用这个ca再签一个client cert，当k8s访问etcd时会先根据server cert与ca验证etcd是自己人，然后k8s会把client cert发给etcd，然后etcd用ca校验一下client cert也是自己签的，就可以知道客户端也是自己人，所以就是双向验证了。

明白原理了，接下来做的事情就很容易理解，我们后面边做边讲，现在先自签一个ca根证书。

root@node01:~# kubeadm alpha phase certs etcd-ca

I1115 17:48:48.825368 6867 version.go:93] could not fetch a Kubernetes version from the internet: unable to get URL "https://dl.k8s.io/release/stable-1.txt": Get https://storage.googleapis.com/kubernetes-release/release/stable-1.txt: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)

I1115 17:48:48.825441 6867 version.go:94] falling back to the local client version: v1.12.2

[certificates] Generated etcd/ca certificate and key.

上述报错没有影响，ca证书与对应的密钥都放在**/etc/kubernetes/pki/etcd/**目录了，专门用来围绕etcd签发各种证书。

root@node01:~# ll /etc/kubernetes/pki/etcd/

total 16

drwxr-xr-x 2 root root 4096 Nov 15 17:48 ./

drwxr-xr-x 3 root root 4096 Nov 15 17:48 ../

-rw-r--r-- 1 root root 1017 Nov 15 17:48 ca.crt

-rw------- 1 root root 1679 Nov 15 17:48 ca.key

Ca.crt就是根证书，ca.key是ca证书的私钥，其本质是RSA加密算法。

### 用kubeadm签发etcd证书

生成ca，生成各种证书的过程对我们没什么意义，kubeadm提供了内置的功能完成etcd的证书签发。

我们先写一个kubeadm的签发etcd证书配置：

apiVersion: "kubeadm.k8s.io/v1alpha3"

kind: ClusterConfiguration

etcd:

local:

serverCertSANs:

- "localhost"

- "node01"

- "node02"

- "node03"

- "127.0.0.1"

- "172.18.10.177"

- "172.18.10.178"

- "172.18.10.179"

peerCertSANs:

- "localhost"

- "node01"

- "node02"

- "node03"

- "127.0.0.1"

- "172.18.10.177"

- "172.18.10.178"

- "172.18.10.179"

只有服务端cert才需要配置host，所以这里只有server cert和peer cert两种证书，我们需要指定服务端的有效HOST，也就是etcd的服务地址。

当客户端随机访问某个etcd节点时，客户端会用ca根证书校验服务端发来的cert，并且看一下当前请求的host是不是在cert的hosts配置范围内，这里serverCertSANs和peerCertSANs就是告诉kubeadm生成证书时用这些服务端的host，覆盖所有etcd节点的所有访问方式（IP，域名），其中peer是etcd节点间互联时用的证书，server cert是客户端来访的时候返回的证书，其实peer和server cert一样也无所谓，只是kubeadm会分别生成两套。

接着，我们写个脚本，把所有围绕etcd的证书生成出来：

#!/bin/bash

kubeadm alpha phase certs etcd-server --config=./kubeadmcfg.yaml

kubeadm alpha phase certs etcd-peer --config=./kubeadmcfg.yaml

kubeadm alpha phase certs etcd-healthcheck-client --config=./kubeadmcfg.yaml

kubeadm alpha phase certs apiserver-etcd-client --config=./kubeadmcfg.yaml

脚本用kubeadm，传入刚才的配置文件，就生成了4套证书，其中-client结尾的都是客户端证书，剩余两套是服务端证书。这些证书都是用我们最开始自签的/etc/kubernetes/pki/etcd/ca.crt根证书签发的，所以可以用这个ca来校验。

现在所有证书如下：

root@node01:~# find /etc/kubernetes/pki

/etc/kubernetes/pki

/etc/kubernetes/pki/etcd

/etc/kubernetes/pki/etcd/server.crt

/etc/kubernetes/pki/etcd/ca.crt

/etc/kubernetes/pki/etcd/peer.key

/etc/kubernetes/pki/etcd/ca.key

/etc/kubernetes/pki/etcd/peer.crt

/etc/kubernetes/pki/etcd/healthcheck-client.crt

/etc/kubernetes/pki/etcd/server.key

/etc/kubernetes/pki/etcd/healthcheck-client.key

/etc/kubernetes/pki/apiserver-etcd-client.crt

/etc/kubernetes/pki/apiserver-etcd-client.key

我们只需要把ca以及client证书配置到k8s，其他的配置给etcd，双向验证就可以work了。

把这个目录拷贝到其他2个机器上的同样位置，一会我们搭建etcd集群要用。

（如果你不想用kubeadm签证书，那就看etcd官方的手册：http://play.etcd.io/install）

### 安装etcd集群

到etcd的github下载amd64版本的二进制包：<https://github.com/etcd-io/etcd/releases> ，因为被墙了，所以用宿主机翻墙下载了再scp到3个虚拟机里吧。

把etcd和etcdctl两个二进制mv到/usr/local/bin里。

我们需要配置systemd来拉起每台机器上的etcd服务，主要依据官方配置：<https://github.com/etcd-io/etcd/tree/master/contrib/systemd/etcd3-multinode>， 额外增加了一些证书配置，原理之前都说的很明白了：

配置模板：

[Unit]

Description=etcd

Documentation=https://github.com/coreos/etcd

Conflicts=etcd.service

Conflicts=etcd2.service

[Service]

Type=notify

Restart=always

RestartSec=5s

LimitNOFILE=40000

TimeoutStartSec=0

ExecStart=/usr/local/bin/etcd \

--name my-etcd-1 \

--data-dir /var/lib/etcd \

--listen-client-urls https://${IP\_1}:2379 \

--advertise-client-urls https://${IP\_1}:2379 \

--listen-peer-urls https://${IP\_1}:2380 \

--initial-advertise-peer-urls https://${IP\_1}:2380 \

--initial-cluster my-etcd-1=https://${IP\_1}:2380,my-etcd-2=https://${IP\_2}:2380,my-etcd-3=https://${IP\_3}:2380 \

--initial-cluster-token my-etcd-token \

--initial-cluster-state new \

--client-cert-auth=true \

--peer-client-cert-auth=true \

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

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

--peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt \

--peer-key-file=/etc/kubernetes/pki/etcd/peer.key \

--trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt \

--peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt

[Install]

WantedBy=multi-user.target

注意到所有URL都是https的，全部加密通讯。

每台节点的—name不同，以及IP不同，做替换即可。

3台机器的配置文件都放到/etc/systemd/system/etcd.service。

把数据目录建出来：

mkdir -p /var/lib/etcd

然后每台机器启动etcd：

systemctl enable etcd

systemctl start etcd

3个节点都启动后，查看systemctl status etcd确认状态正常。

然后我们可以用etcdctl访问一下（替换node\_ip）：

etcdctl --endpoints=https://${NODE\_IP}:2379 --ca-file=/etc/kubernetes/pki/etcd/ca.crt --cert-file=/etc/kubernetes/pki/apiserver-etcd-client.crt --key-file=/etc/kubernetes/pki/apiserver-etcd-client.key cluster-health

我得到的结果：

root@node01:~# etcdctl --endpoints=https://172.18.10.177:2379 --ca-file=/etc/kubernetes/pki/etcd/ca.crt --cert-file=/etc/kubernetes/pki/apiserver-etcd-client.crt --key-file=/etc/kubernetes/pki/apiserver-etcd-client.key cluster-health

member ea38486c711249f is healthy: got healthy result from https://172.18.10.179:2379

member 994eaeaa12aed915 is healthy: got healthy result from https://172.18.10.177:2379

member fa5ae984ea630d26 is healthy: got healthy result from https://172.18.10.178:2379

cluster is healthy

这里客户端使用了apiserver-etcd-client.crt和key，其实就是我们即将给k8s使用的客户端证书，即etcd只允许可信的client访问自己，k8s就是用这套证书证明自己的。反过来，客户端也需要确认服务端身份，所以客户端会保存ca根证书，校验一下服务端是不是可信的，所以ca未来也会配置给k8s。

至此，etcd集群搭建完成。

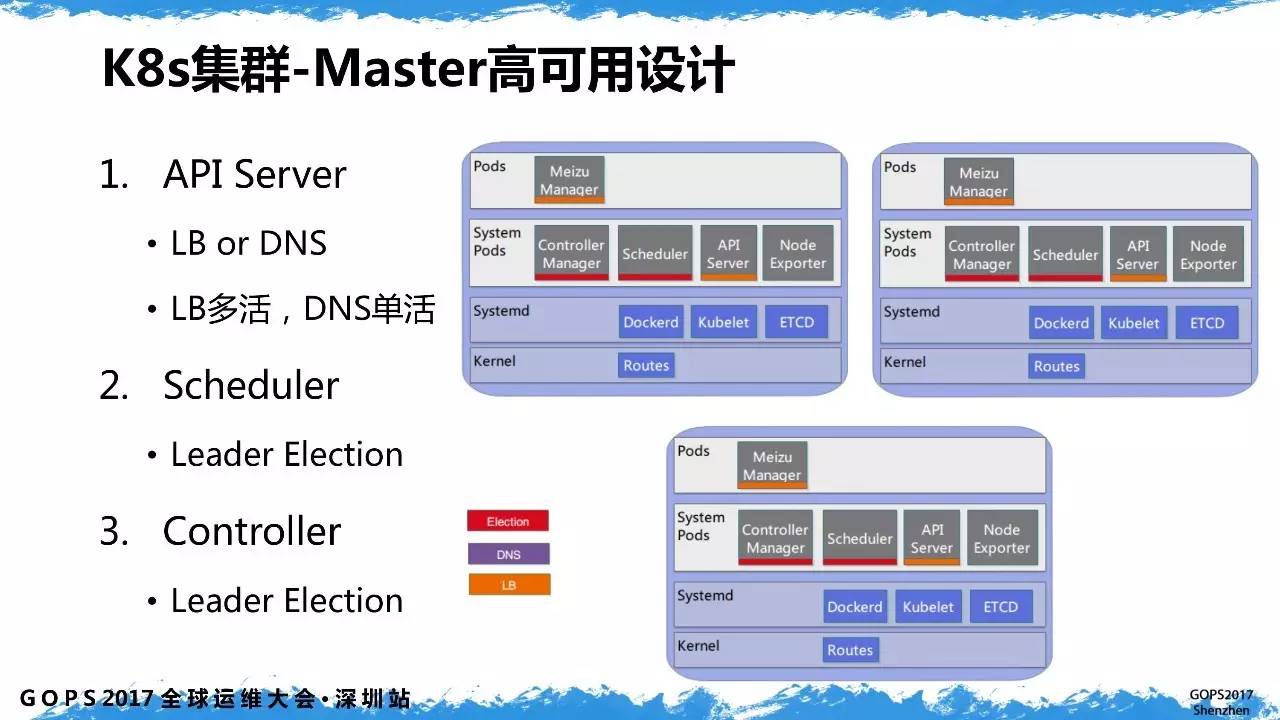
## k8s master高可用架构原理

接下来搭建多个k8s master，保障高可用。

步骤参考官方：<https://kubernetes.io/docs/setup/independent/high-availability/#common-tasks-after-bootstrapping-control-plane> 的external etcd方案。

K8s的高可用就是实现apiserver高可用，apiserver是k8s对外和对内服务的入口。

k8s让我们自己搭一个负载均衡代理作为多个apiserver的反向代理，然后k8s的其他组件都通过这个负载均衡访问到某个apiserver，架构图这样：



Apiserver需要负载均衡，scheduler和controller则通过集群中选举出1个工作。

实际上云的话，可以买云上的负载均衡做反向代理。现在我电脑资源有限，索性就在node01上启动一个haproxy充当负载均衡做4层转发，并把master部署到node01,node02,node03三个节点上作高可用部署。

## 搭建haproxy

我们在node01部署haproxy做3个apiserver的反向代理。

Apiserver默认端口官方有写：<https://kubernetes.io/docs/setup/independent/install-kubeadm/#check-required-ports> ， 是6443。

我们先在node01安装haproxy：apt-get install haproxy -y

然后vim /etc/haproxy/haproxy.cfg 配置一下haproxy做4层反向代理就行（因为真实环境一般不会用这个，我就简单配置一下能用即可）：

global

user haproxy

group haproxy

daemon

defaults

timeout server 5s

timeout client 5s

timeout connect 5s

frontend k8s-load-balance

bind \*:5000

mode tcp

default\_backend k8s-api-server

backend k8s-api-server

balance roundrobin

server node01 172.18.10.177:6443 check inter 1000 fall 3 rise 3

server node02 172.18.10.178:6443 check inter 1000 fall 3 rise 3

server node03 172.18.10.179:6443 check inter 1000 fall 3 rise 3

负载均衡部署在node01，监听在5000端口，对后端3个k8s apiserver每隔1秒做一次健康检查。

root@node01:~# netstat -tanlp|grep 5000

tcp 0 0 0.0.0.0:5000 0.0.0.0:\* LISTEN 20445/haproxy

## 启动第1个master

因为我把etcd部署在了k8s的3个机器上，所以etcd的ca和client证书都在机器上了。

真实环境，需要把证书从etcd机器上拷贝过来，配置给k8s用。

我们现在node01上配置第一个Master，使用工具仍旧是kubeadm，参考文档：<https://kubernetes.io/docs/setup/independent/high-availability/#external-etcd>。

编辑一个文件kubeadm-config.yaml：

**apiVersion: kubeadm.k8s.io/v1alpha3**

**kind: ClusterConfiguration**

**kubernetesVersion: stable**

**apiServerCertSANs:**

**- "LOAD\_BALANCER\_DNS"**

**controlPlaneEndpoint: "LOAD\_BALANCER\_DNS:LOAD\_BALANCER\_PORT"**

**etcd:**

**external:**

**endpoints:**

**- https://ETCD\_0\_IP:2379**

**- https://ETCD\_1\_IP:2379**

**- https://ETCD\_2\_IP:2379**

**caFile: /etc/kubernetes/pki/etcd/ca.crt**

**certFile: /etc/kubernetes/pki/apiserver-etcd-client.crt**

**keyFile: /etc/kubernetes/pki/apiserver-etcd-client.key**

**networking:**

**# This CIDR is a calico default. Substitute or remove for your CNI provider.**

**podSubnet: "192.168.0.0/16"**

apiServerCertSANS是kubeadm帮我们apiserver生成对外服务的证书用的。因为外部访问apiserver是通过负载均衡实现的，所以作为服务端提供的证书中应该写的hosts是负载均衡的地址。

controlPlaneEndpoint是apiserver的服务地址，同样是负载均衡的host:port。

Etcd使用外部集群，我们给k8s配置etcd集群地址，还有ca以及客户端cert。

Networking是根据我们选择的网络插件决定的子网段，我选的flannel就配flannel的：**10.244.0.0/16**。

我替换后的文件是这样的：

apiVersion: kubeadm.k8s.io/v1alpha3

kind: ClusterConfiguration

kubernetesVersion: stable

apiServerCertSANs:

- "172.18.10.177"

controlPlaneEndpoint: "172.18.10.177:5000"

etcd:

external:

endpoints:

- https://172.18.10.177:2379

- https://172.18.10.178:2379

- https://172.18.10.179:2379

caFile: /etc/kubernetes/pki/etcd/ca.crt

certFile: /etc/kubernetes/pki/apiserver-etcd-client.crt

keyFile: /etc/kubernetes/pki/apiserver-etcd-client.key

networking:

podSubnet: "10.244.0.0/16"

接下来用kubeadm来init一个master出来，传入这个配置文件：

**kubeadm init --config ./kubeadm-config.yaml**

但实际情况会发现命令卡住了，因为kubeadm要下载apiserver,scheduler,core-dns等docker镜像回来，而这些镜像都在谷歌的docker仓库里，全都被墙了。

大家把下面的命令放到一个xxx.sh中执行，脚本会从一个镜像的docker仓库拉下所有k8s用到的image，然后通过docker tag命令改名到k8s的原始image名：

#!/bin/bash

docker pull anjia0532/google-containers.kube-apiserver-amd64:v1.12.2

docker pull anjia0532/google-containers.kube-controller-manager-amd64:v1.12.2

docker pull anjia0532/google-containers.kube-scheduler-amd64:v1.12.2

docker pull anjia0532/google-containers.kube-proxy-amd64:v1.12.2

docker pull anjia0532/google-containers.pause:3.1

docker pull anjia0532/google-containers.etcd-amd64:3.2.24

docker pull anjia0532/google-containers.coredns:1.2.2

docker tag anjia0532/google-containers.kube-apiserver-amd64:v1.12.2 k8s.gcr.io/kube-apiserver:v1.12.2

docker tag anjia0532/google-containers.kube-controller-manager-amd64:v1.12.2 k8s.gcr.io/kube-controller-manager:v1.12.2

docker tag anjia0532/google-containers.kube-scheduler-amd64:v1.12.2 k8s.gcr.io/kube-scheduler:v1.12.2

docker tag anjia0532/google-containers.kube-proxy-amd64:v1.12.2 k8s.gcr.io/kube-proxy:v1.12.2

docker tag anjia0532/google-containers.pause:3.1 k8s.gcr.io/pause:3.1

docker tag anjia0532/google-containers.etcd-amd64:3.2.24 k8s.gcr.io/etcd:3.2.24

docker tag anjia0532/google-containers.coredns:1.2.2 k8s.gcr.io/coredns:1.2.2

docker rmi anjia0532/google-containers.kube-apiserver-amd64:v1.12.2

docker rmi anjia0532/google-containers.kube-controller-manager-amd64:v1.12.2

docker rmi anjia0532/google-containers.kube-scheduler-amd64:v1.12.2

docker rmi anjia0532/google-containers.kube-proxy-amd64:v1.12.2

docker rmi anjia0532/google-containers.pause:3.1

docker rmi anjia0532/google-containers.etcd-amd64:3.2.24

docker rmi anjia0532/google-containers.coredns:1.2.2

我怎么知道要下载哪些k8s镜像呢？ 执行一条命令就知道了：

root@node01:~# kubeadm config images list

k8s.gcr.io/kube-apiserver:v1.12.2

k8s.gcr.io/kube-controller-manager:v1.12.2

k8s.gcr.io/kube-scheduler:v1.12.2

k8s.gcr.io/kube-proxy:v1.12.2

k8s.gcr.io/pause:3.1

k8s.gcr.io/etcd:3.2.24

k8s.gcr.io/coredns:1.2.2

好了，现在就可以再次执行kubeadm init了。

最后输出这段信息：

Your Kubernetes master has initialized successfully!

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

You should now deploy a pod network to the cluster.

Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:

https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node

as root:

kubeadm join 172.18.10.177:5000 --token m1omoh.wfreawaz74d4w5a0 --discovery-token-ca-cert-hash sha256:a2f6c04e2940e149f9bedd144230de36ad5dd861c2c293853a9bde42a17a5db5

第一部分提示让我们切换回普通用户才能使用k8s，所以我们从root切换回k8s帐号。

然后在HOME下创建一个.kube目录，然后cp /etc/kubernetes/admin.conf到.kube/config中，改成k8s的权限：

k8s@node01:~$ pwd

/home/k8s

k8s@node01:~$ ll .kube/

total 16

drwxrwxr-x 2 k8s k8s 4096 Nov 16 13:12 ./

drwxr-xr-x 4 k8s k8s 4096 Nov 16 13:12 ../

-rw------- 1 k8s k8s 5453 Nov 16 13:12 config

验证一下k8s正常工作：

k8s@node01:~$ kubectl version

Client Version: version.Info{Major:"1", Minor:"12", GitVersion:"v1.12.2", GitCommit:"17c77c7898218073f14c8d573582e8d2313dc740", GitTreeState:"clean", BuildDate:"2018-10-24T06:54:59Z", GoVersion:"go1.10.4", Compiler:"gc", Platform:"linux/amd64"}

Server Version: version.Info{Major:"1", Minor:"12", GitVersion:"v1.12.2", GitCommit:"17c77c7898218073f14c8d573582e8d2313dc740", GitTreeState:"clean", BuildDate:"2018-10-24T06:43:59Z", GoVersion:"go1.10.4", Compiler:"gc", Platform:"linux/amd64"}

再看看集群信息，的确发现master跑在负载均衡后面：

k8s@node01:~$ kubectl cluster-info

Kubernetes master is running at https://172.18.10.177:5000

KubeDNS is running at https://172.18.10.177:5000/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump

然后看看现在k8s的核心组件都跑起来了没？

k8s@node01:~$ kubectl get pod --namespace kube-system

NAME READY STATUS RESTARTS AGE

coredns-576cbf47c7-gpt29 0/1 Pending 0 6m59s

coredns-576cbf47c7-hl4d8 0/1 Pending 0 6m59s

kube-apiserver-node01 1/1 Running 0 6m20s

kube-controller-manager-node01 1/1 Running 0 6m14s

kube-proxy-zwwjp 1/1 Running 0 6m59s

kube-scheduler-node01 1/1 Running 0 6m18s

发现coredns的2个pod还没拉起来，scheduler,apiserver,controller,kube-proxy这些核心组件都启动了。

我们耐心等一下coredns的初始化。

第二部分提示我们如何将其他节点加入到k8s集群，我们先记录下来这条命令。

## 搭建第2个master

刚才搭建第1个master的过程中，kubeadm又自签了几个CA根证书，用来给apiserver等节点签发证书，注意这些CA和之前etcd用的CA不一样。

然后kubeadm又用新CA签发了一些证书供k8s组件间，以及供客户端访问k8s使用。

我们只需把这些新ca拷贝到node02上，至于新ca在node01上签发的服务端or客户端证书不需要拷贝到Node02，kubeadm会在第2个master上重新签证书，这是因为master集群对外是基于负载均衡IP服务的，所以每个节点签发证书时的hosts都会使用负载均衡IP，所以客户端拿任意一个节点签发的client cert都可以通过服务端的验证。

具体要拷贝这些ca到node02：

**/etc/kubernetes/pki/ca.crt**

**/etc/kubernetes/pki/ca.key**

**/etc/kubernetes/pki/sa.key**

**/etc/kubernetes/pki/sa.pub**

**/etc/kubernetes/pki/front-proxy-ca.crt**

**/etc/kubernetes/pki/front-proxy-ca.key**

压缩一下，scp到node02，解压到对应路径即可。

我们还得手动把k8s镜像下载到node02上：

然后，我们把刚才的join命令粘贴出来：

kubeadm join 172.18.10.177:5000 --token m1omoh.wfreawaz74d4w5a0 --discovery-token-ca-cert-hash sha256:a2f6c04e2940e149f9bedd144230de36ad5dd861c2c293853a9bde42a17a5db5

在末尾加一个–experimental-control-plane选项，然后再执行：

kubeadm join 172.18.10.177:5000 --token m1omoh.wfreawaz74d4w5a0 --discovery-token-ca-cert-hash sha256:a2f6c04e2940e149f9bedd144230de36ad5dd861c2c293853a9bde42a17a5db5 ***--experimental-control-plane***

你会发现，join的地址也是现在的master负载均衡地址，目前就是代理到了node01。

输出的关键信息如下：

This node has joined the cluster and a new control plane instance was created:

\* Certificate signing request was sent to apiserver and approval was received.

\* The Kubelet was informed of the new secure connection details.

\* Master label and taint were applied to the new node.

\* The kubernetes control plane instances scaled up.

To start administering your cluster from this node, 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

Run 'kubectl get nodes' to see this node join the cluster.

提示我们，给master集群加入了一个新的control plane instance。

同样的，我们把/etc/kubernetes/admin.conf拷给k8s用户，今后用k8s用户操作集群。

查看集群已经有2个master了：

k8s@node02:~$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

node01 NotReady master 34m v1.12.2

node02 NotReady master 3m40s v1.12.2

查看一下master中运行了哪些组件：

k8s@node02:~$ kubectl get pods --namespace kube-system -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

coredns-576cbf47c7-gpt29 0/1 ContainerCreating 0 42m <none> node02 <none>

coredns-576cbf47c7-hl4d8 0/1 ContainerCreating 0 42m <none> node02 <none>

kube-apiserver-node01 1/1 Running 0 42m 172.18.10.177 node01 <none>

kube-apiserver-node02 1/1 Running 0 11m 172.18.10.178 node02 <none>

kube-controller-manager-node01 1/1 Running 0 42m 172.18.10.177 node01 <none>

kube-controller-manager-node02 1/1 Running 0 11m 172.18.10.178 node02 <none>

kube-proxy-45jjd 1/1 Running 0 11m 172.18.10.178 node02 <none>

kube-proxy-zwwjp 1/1 Running 0 42m 172.18.10.177 node01 <none>

kube-scheduler-node01 1/1 Running 0 42m 172.18.10.177 node01 <none>

kube-scheduler-node02 1/1 Running 0 11m 172.18.10.178 node02 <none>

可见，每个节点跑了一套apiserver+scheduler+controller-manager+kube-proxy，这些节点虽然是docker拉起的，但网络都host在宿主机上，整个集群就一套core-dns做pod DNS服务。

然而，Core-dns一直没初始化成功，可以看一下原因：

执行下面命令查看pod的日志：

kubectl describe pod coredns-576cbf47c7-gpt29 -n kube-system

发现了：

Warning FailedScheduling 14m (x184 over 44m) default-scheduler 0/1 nodes are available: 1 node(s) had taints that the pod didn't tolerate.

Warning NetworkNotReady 3m32s (x46 over 13m) kubelet, node02 network is not ready: [runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni config uninitialized]

意思是k8s的网络插件还没安装，所有core-dns没法和容器通讯之类的，所以我们把flannel安一下（在node01或者node02都可以）：

**kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml**

一旦安装上，整个k8s集群所有节点都会自动拉起一个flannel的docker进程负责网络分配，执行命令可以看到对应的Pod：

kubectl get pods -n kube-system -o wide

启动比较慢，因为要拉镜像：

Events:

Type Reason Age From Message

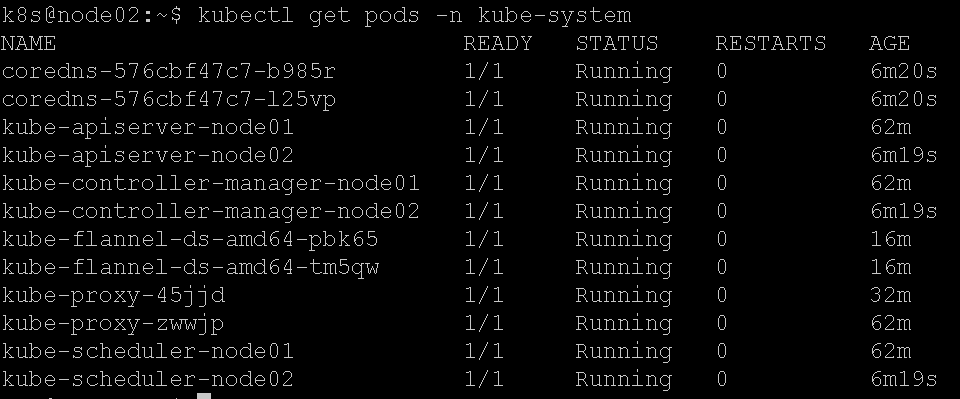
---- ------ ---- ---- -------

Normal Scheduled 118s default-scheduler Successfully assigned kube-system/kube-flannel-ds-amd64-pbk65 to node01

Normal Pulling 117s kubelet, node01 pulling image "quay.io/coreos/flannel:v0.10.0-amd64"

这个flannel镜像没有完全被墙，只是很慢，如果实在不行大家自己去找替代的镜像下载，然后docker tag改名吧。

等完成之后，我们很快可以看到Coredns正常服务，并且master状态为ready：



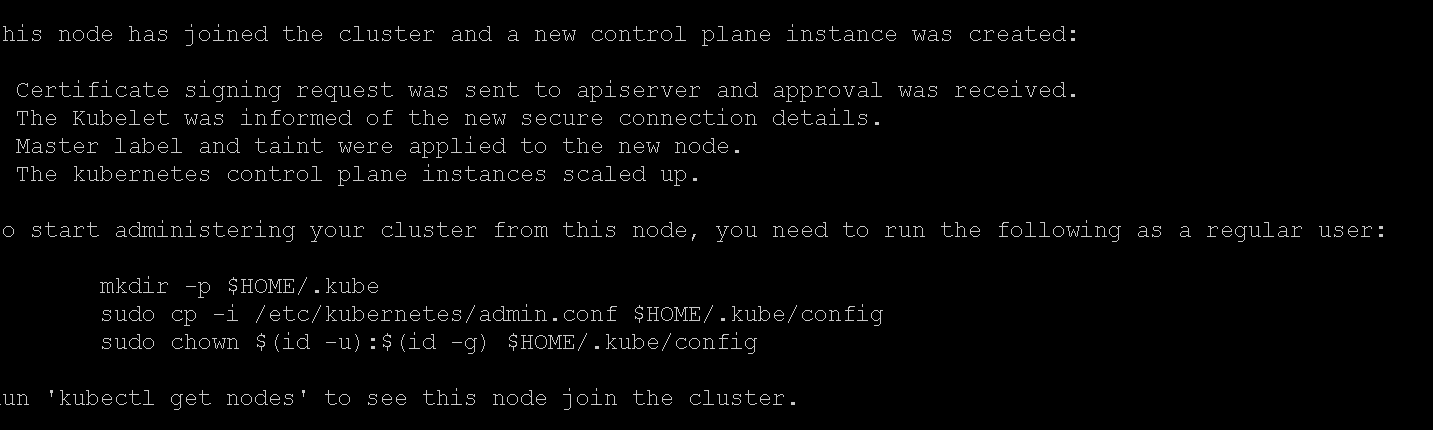


## 搭建第3个master

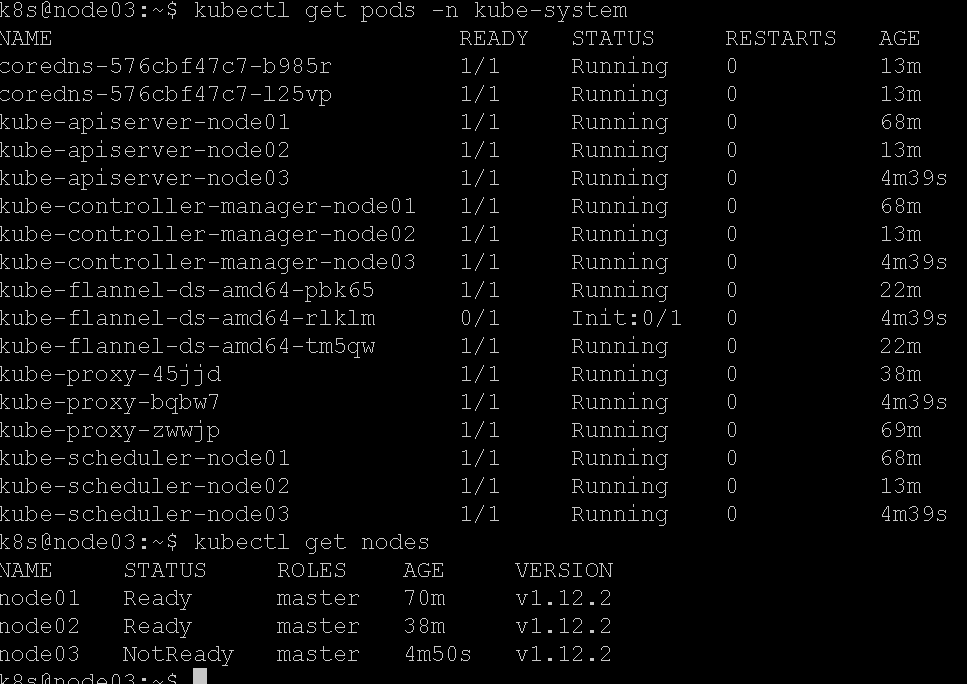
记得把ca拷过来，然后记得给join命令加参数，然后执行即可：

kubeadm join 172.18.10.177:5000 --token m1omoh.wfreawaz74d4w5a0 --discovery-token-ca-cert-hash sha256:a2f6c04e2940e149f9bedd144230de36ad5dd861c2c293853a9bde42a17a5db5 ***--experimental-control-plane***

成功：



把admin.conf拷到k8s用户/home/k8s/.kube/config（用户属主记得改为k8s），然后等待node03上的master生效。



还是卡在flannel下载镜像上，多等等吧。

# 验证集群

## 部署服务

我们编写一个nginx的deployment部署，看看是否正常调度。

编辑nginx.yaml：

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: nginx

spec:

replicas: 3

template:

metadata:

labels:

app: nginx

spec:

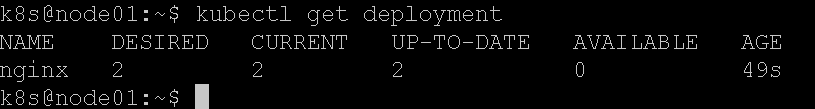
containers:

- name: nginx

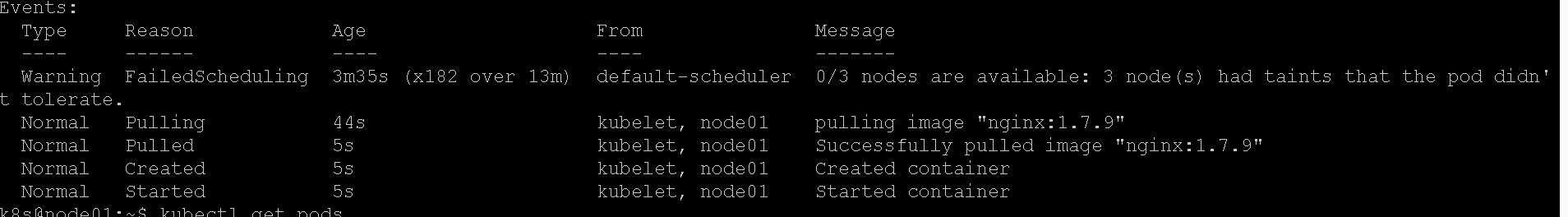
image: nginx:1.7.9

这个deployment部署任务，会启动包含3个pods的replicaset副本集。

我们执行kubectl apply -f nginx.yaml就触发了部署，稍后会看到deployment与pod的状态：



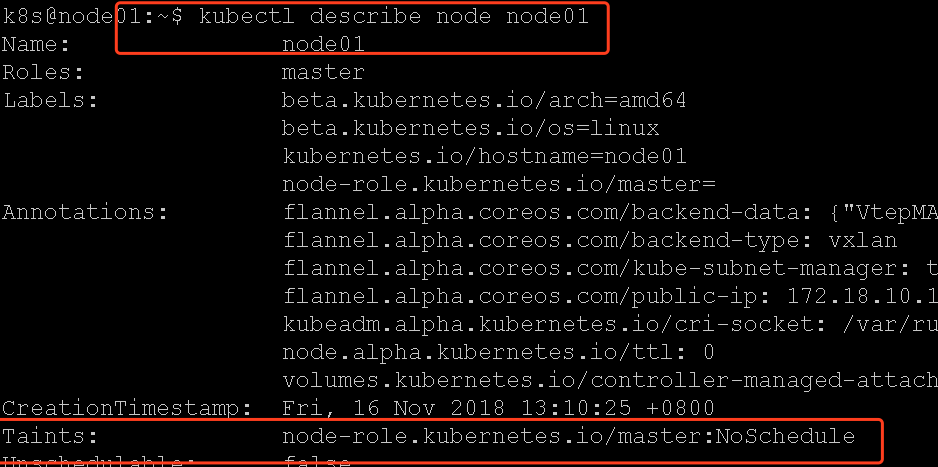
Avaliable始终为0，这里原因是默认k8s不会调度用户pod到master节点：



我们需要给3个master节点解禁:

## 允许master运行pod

我们可以查看node01的节点配置：



它有一个污点taints（参考：https://blog.frognew.com/2018/05/taint-and-toleration.html），其中NoSchedule标识该节点禁止调度POD。

我们稍微改一下，把这个effect改成PreferNoSchedule，就是尽量不调度到该节点，如果没有普通slave节点就调度到master上：

先删掉原先的污点：

kubectl taint nodes node01 node-role.kubernetes.io/master:NoSchedule-

加上新的污点：

kubectl taint nodes node01 node-role.kubernetes.io/master=:PreferNoSchedule

nginx调度立即完成了：



我们用同样的方法，给其他2个节点解禁：

kubectl taint nodes node02 node-role.kubernetes.io/master:NoSchedule-

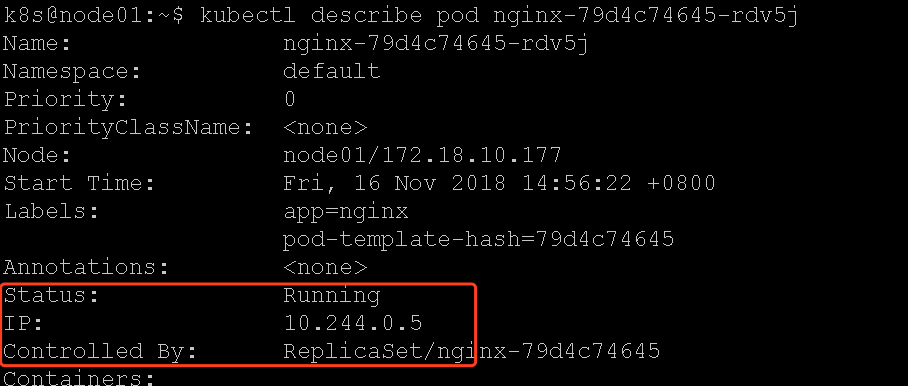
kubectl taint nodes node02 node-role.kubernetes.io/master=:PreferNoSchedule

kubectl taint nodes node03 node-role.kubernetes.io/master:NoSchedule-

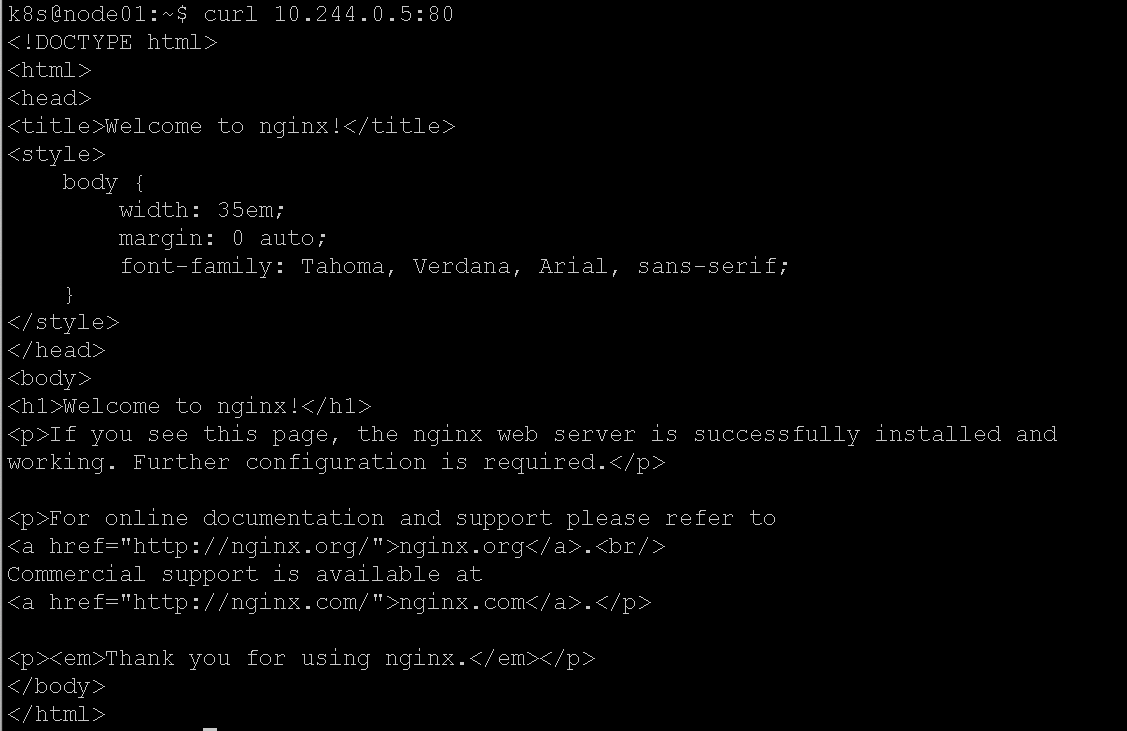
kubectl taint nodes node03 node-role.kubernetes.io/master=:PreferNoSchedule

## 验证pod网络

K8s集群中的宿主机和容器网络是互通的，我们先看一下某个POD的IP：



看到了基于flannel分配的POD IP，现在我们可以直接在宿主机上请求它：



请求成功，说明k8s集群网络没有问题。

## 验证高可用

我们现在关掉node03节点,等了一会发现node03已经挂了：

k8s@node01:~$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

node01 Ready master 114m v1.12.2

node02 Ready master 83m v1.12.2

node03 NotReady master 49m v1.12.2

我们修改nginx.yaml的replicas数量为5，并apply扩容：

k8s@node01:~$ kubectl apply -f nginx.yaml

deployment.extensions/nginx configured

观察pods：



Pods只会被调度到node01和node02，我们的kubectl命令行操作也没有出现任何报错。

很快，5个pod全部正常：



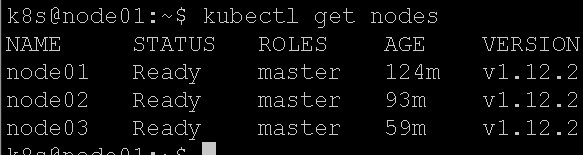
现在杀死node02节点，只留下node01节点。

再次请求发现超时了，**这并不是因为k8s不行了，是因为etcd集群已经只剩下1个节点了，raft协议无法工作导致的。**

实际我们分离部署的话，只剩下一台k8s master也是可以work的。

## 搭建成功

最后，我们把node02和node03重新启动，k8s集群自动恢复了正常，自愈能力很强。



我什么也没做，nginx服务全部恢复了正常：

