# Kubernetes 概述

## 背景

kubernetes(以下简称k8s)是google基于Blog进行改进后，开源出来的一款“容器管理应用”。由于近几年来容器技术的火爆，许许多多的服务都不会直接部署在linux主机或各大云厂商的虚拟机上；利用Docker，将每个服务做成一个Image，把他们跑在各自的Container中。

这样做的好处有非常多，比如环境配置隔离、服务启动快、移植便捷等等。但是使用的Container多到一定程度，就会带来容器管理上的问题：使用docker ps命令之后有一大堆Container，如果标识的不清楚也很容易混淆；某些分布式服务，需要将Docker部署到许多不同的机器上，这也会增加我们运维的难度。因此，我们现在需要一款“专门管理容器”的平台，为我们提供可视化界面，方便我们对各个容器进行管理。

k8s就是这样一款辅助我们管理容器的平台，支持管理在分布式环境(即多台服务器上)启动的Container。

## 基础概念

* Maste**r**

Cluster的大脑，主要职责是调度，可以运行多个master来保证高可用。

* **Node**：

职责是运行容器应用，Node由Master管理，负责监控并汇报容器的状态，同时根据Master的要求管理容器的生命周期。

* Pod

Pod 是K8s的最小工作单元。每个Pod包含一个或多个容器。

* 有些容器天生就是需要紧密联系，一起工作。Pod提供了比容器更高层次的抽象，K8s以Pod为最小单位进行调度、扩展、共享资源、管理生命周期。
* Pod中的所有容器使用同一个网络的namespace，即相同的IP地址和Port空间。它们可以直接用localhost通信。同样的，这些容器可以共享存储，当K8s挂载Volume到Pod上，本质上是将volume挂载到Pod中的每一个容器。
* Pod控制器

K8s通常不直接创建Pod，而是通过Controller来管理Pod。Controller中定义了pod的部署属性，比如几个副本、在什么样的Node上运行等。

K8s提供了多种Controller，包括Deployment、ReplicaSet、DaemonSet、StatefuleSet、Job等。

* **ReplicationController** （副本控制器），确保Pod的数量始终保持设定的个数。也支持Pod的滚动更新。
* **ReplicaSet** （副本集），它不直接使用，有一个声明式更新的控制器叫**Deployment**来负责管理。但是Deployment只能负责管理那些无状态的应用。
* **StatefulSet** （有状态副本集），负责管理有状态的应用。
* **DaemonSet** ，如果需要在每一个Node上只运行一个副本，而不是随意运行，就需要DaemonSet。
* **Job**，运行作业，对于时间不固定的操作，比如：某个应用生成了一大堆数据集，现在需要临时启动一个Pod去清理这些数据集，清理完成后，这个Pod就可以结束了。 这些不需要一直处于运行状态的应用，就用Job这个类型的控制器去控制。如果Pod运行过程中意外中止了，Job负责重启Pod。如果Pod任务执行完了，就不需要再启动了。
* **Cronjob**，周期性作业。
* Service：

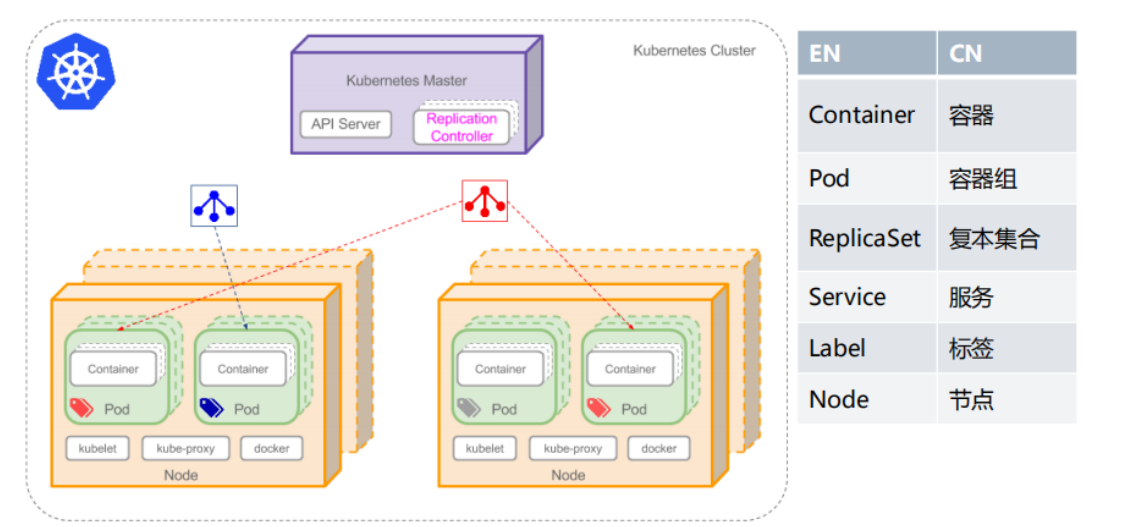
Deployement可以部署多个副本，每个Pod都有自己的副IP，外界如何访问这些副本。Pod会被频繁的销毁和重启，IP实时变化，不能用IP， 答案是通过service。K8s service 定义了外界访问一组特定Pod的方式。service有自己的IP和端口，service为Pod提供了负载均衡。

* Namespace:

Namespace将物理的Cluster逻辑上划分成多个虚拟Cluster，每个Cluster就是一个Namespace。不同的Namespace里的资源是完全隔离的。

* default：默认的namespace
* kube-system:  K8s自己创建的的系统资源放到这个namespace

## K8S的架构

Kubernetes集群包含有节点代理Kubelet 和Master组件（APIs，scheduler，etc.），下面是K8S的架构图。

Master节点包含API Server、Scheduler（调度器）、ControllerManager（控制器管理器）这三个核心的组件。

Node节点包含的核心组件有Kubelet、Docker容器引擎、Kube-proxy

* **API Server（kube-apiserver）**

提供了HTTP/HTTPS RESTful API,即Kubernetes API。 API server是Kubernetes Cluster的前端接口。其他客户端工具（CLI或UI）以及K8S其它组件可以通过它管理Cluster资源。

* **Scheduler**(kube-scheduler)

调度器，它负责决定将Pod放在哪个Node上运行。调度时候考虑Cluster拓扑，各个节点的负载，以及应用对高可用、性能、数据亲和性的需求。

* **Controller-Manager**

负责监控每一个Controller（控制器）的健康状态，并确保控制器是健康的。而控制器是确保Pod健康的组件。

* **etcd**

负责保存K8s Cluster的配置信息和各种资源的状态信息。当数据发生变化时，etcd可以快速的通知K8s 组件。

* **Pod网络**

Pod能通信，k8s cluster必须部署Pod网络（比如flannel是其中一个方案）

* **Kubelet**

是Node的agent,当scheduler确定在某个Node上运行Pod后，会将Pod的具体配置信息（image、volume等）发送给该节点的kubelet，kubelet根据这些信息创建和运行容器，并向Master报告运行状态。

* **Kube-proxy**

service在逻辑上代表了后端的多个Pod，外界通过service访问Pod。service接收到的请求是如何转发到Pod的呢？这就是kube-proxy要完成的工作。每个node都运行kube-proxy服务，它负责将访问service的TCP/UDP数据流转发到后端容器。如果有多个副本，kube-proxy实现负载均衡。

# Kubernetes 集群搭建

## 环境准备

本次教程，使用docker 18.09.9和kubelet-1.16.4，要求centos7.6以上版本

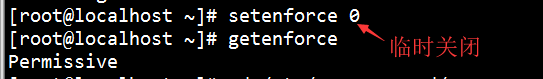


### 关闭selinux

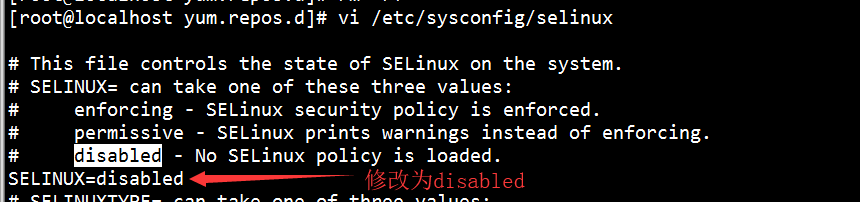
查看selinux是否关闭



先设置临时关闭



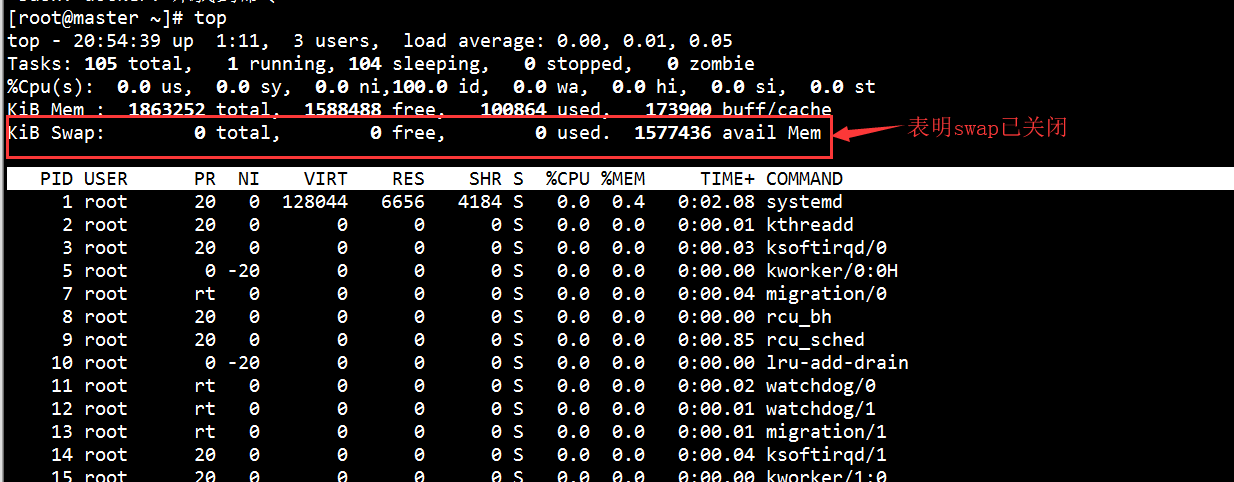
永久关闭



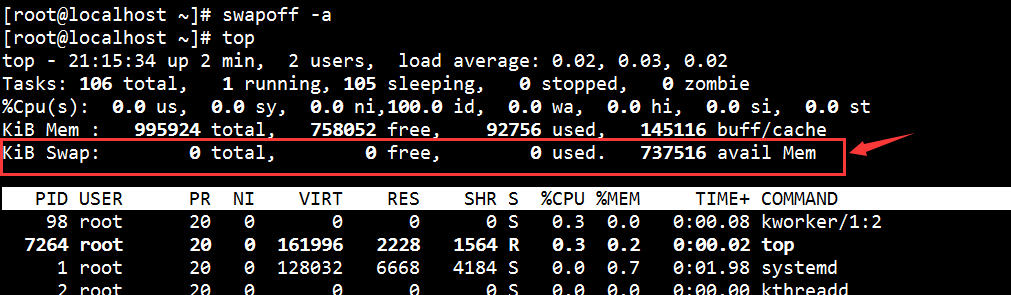
### 关闭swap

**k8s要求系统关闭，否则安装过程会报错**

**查看系统是否关闭了swap**

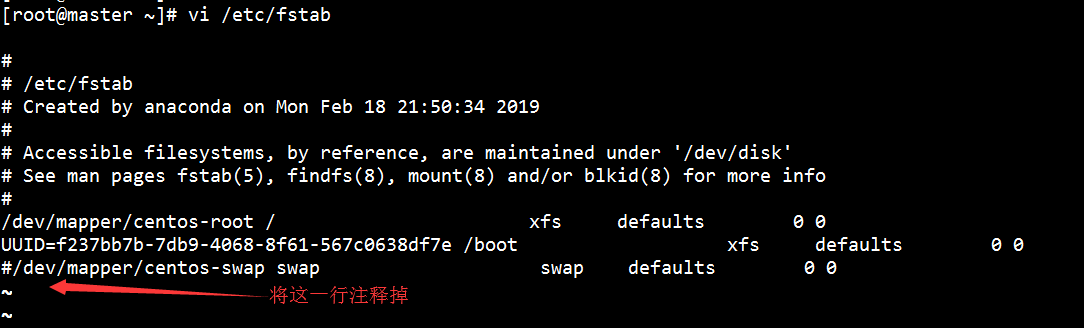
****

**临时禁用：swapoff -a**

****

**永久禁用：sed -i.bak '/swap/s/^/#/' /etc/fstab ##注释掉swap那一行**

**作用就是修改/etc/fstab配置为如下：**

****

### 配置ip\_forward转发

ip\_forward 配置文件当前内容为0，表示禁止数据包转发，将其修改为1表示允许

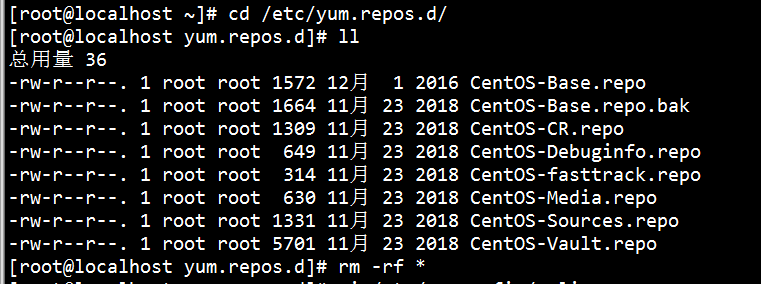
echo "1" > /proc/sys/net/ipv4/ip\_forward



### 更新yum源

为了一次性配置好下载源，我们一次性修改好centos7软件源，docker源，k8s源

先清除掉系统自带配置



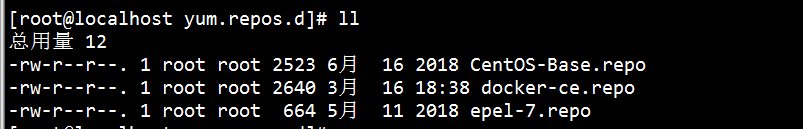
下载centos7的源和docker源

wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.repo

wget -P /etc/yum.repos.d/ http://mirrors.aliyun.com/repo/epel-7.repo

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

结果如下：



配置k8s源

cat <<EOF > /etc/yum.repos.d/kubernetes.repo

[kubernetes]

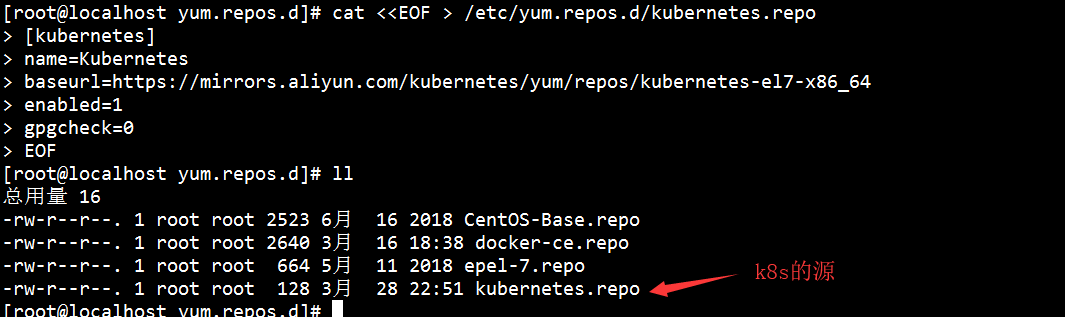
name=Kubernetes

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

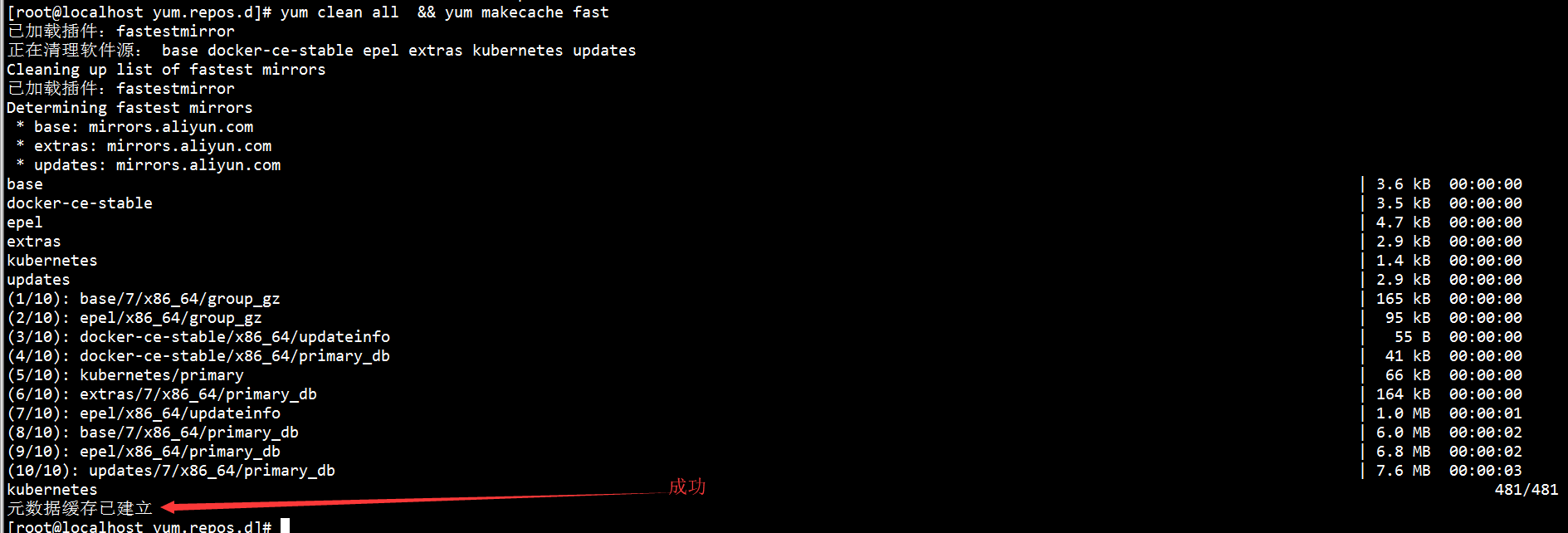
enabled=1

gpgcheck=0

EOF



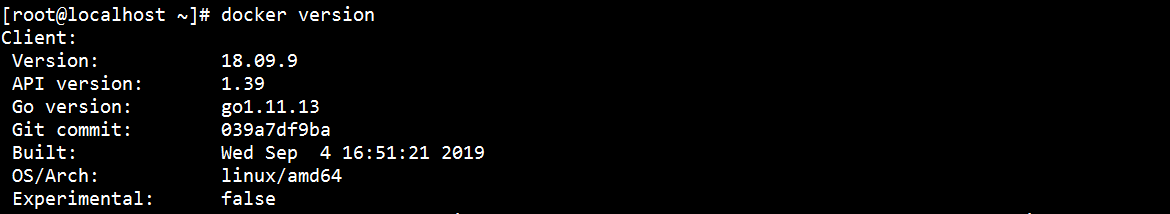
刷新yum缓存



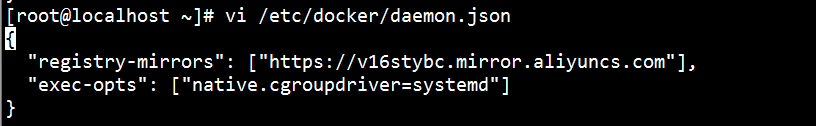
### 安装docker

docker使用版本 18.09.9

yum install docker-ce-18.09.9 docker-ce-cli-18.09.9 containerd.io -y

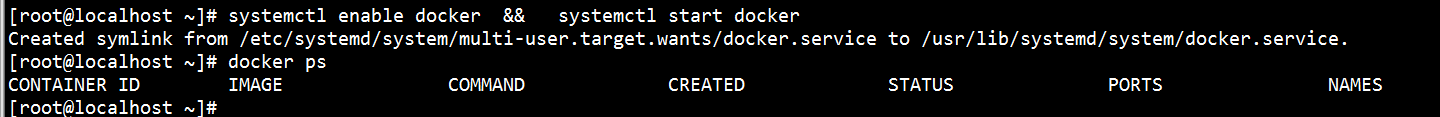


k8s运行要求docker的--cgroup-driver=systemd



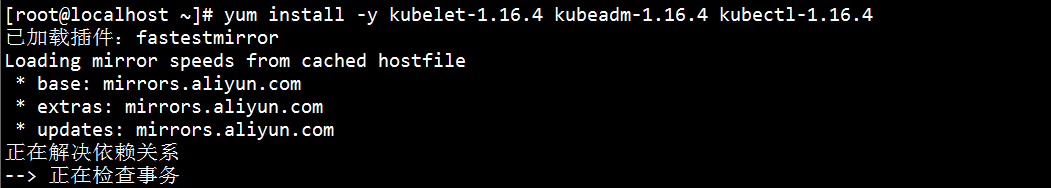
启动docker并设置开机启动

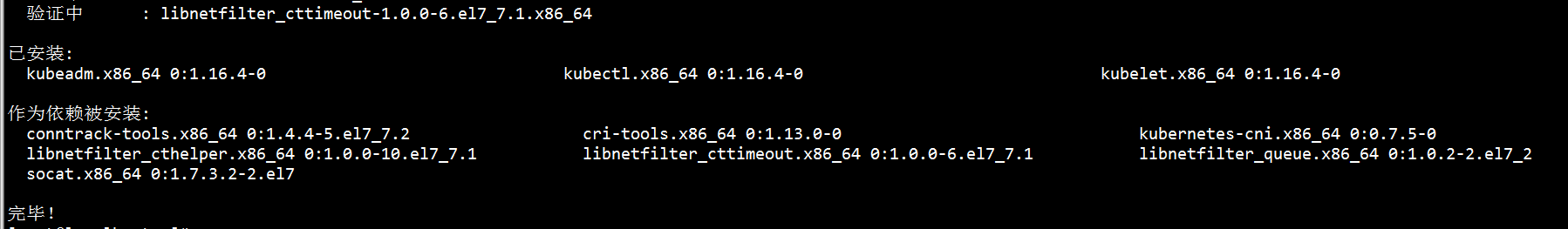
systemctl enable docker && systemctl start docker



### 安装k8s组件：

yum install -y kubelet-1.16.4 kubeadm-1.16.4 kubectl-1.16.4





设置开机启动：systemctl enable kubelet && systemctl start kubelet

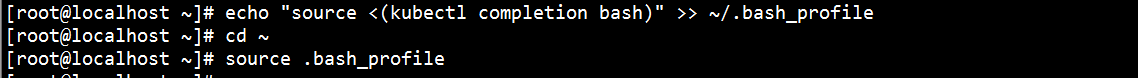


添加kubectl上下文到环境中

echo "source <(kubectl completion bash)" >> ~/.bash\_profile

source .bash\_profile

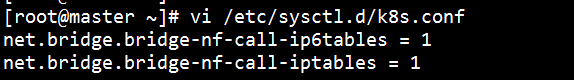
在家目录中，配置生效



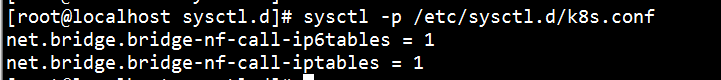
### 内核参数修改

k8s网络一般使用flannel，该网络需要设置内核参数bridge-nf-call-iptables=1

添加参数配置文件：



执行：sysctl -p /etc/sysctl.d/k8s.conf



至此，环境准备工作完毕

#### 内核参数修改失败常见问题

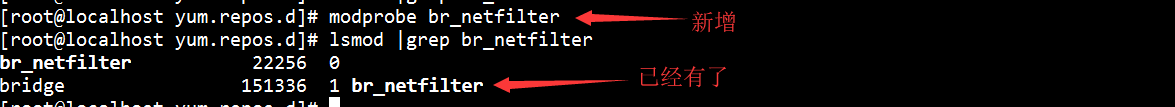
有些系统执行sysctl -p /etc/sysctl.d/k8s.conf会报异常，一般是因为

修改这个参数需要系统有br\_netfilter模块

使用lsmod |grep br\_netfilter命令，查看系统里是否有br\_netfilter模块

****

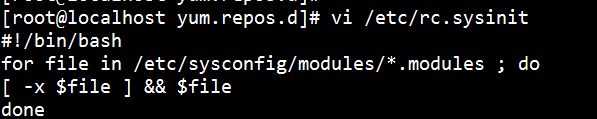
**新增br\_netfilter模块:**



**上述方式重启后无效。需要配置系统启动加载脚本使其永久生效：**

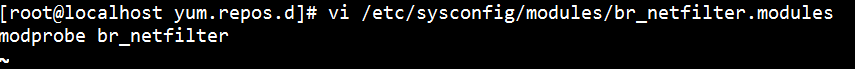
* **先加开机启动动作**

**vi /etc/rc.sysinit**



* **再做加载模块动作**

**vi /etc/sysconfig/modules/br\_netfilter.modules**



* **再增加执行权限**

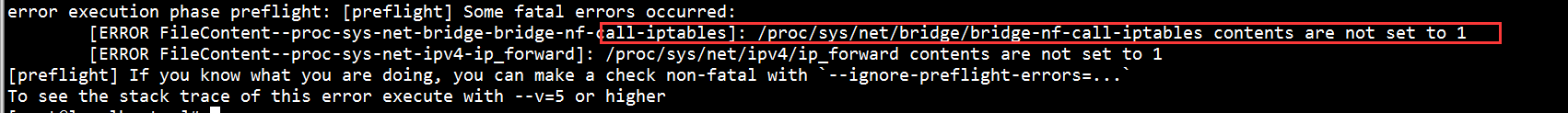


## Master节点配置

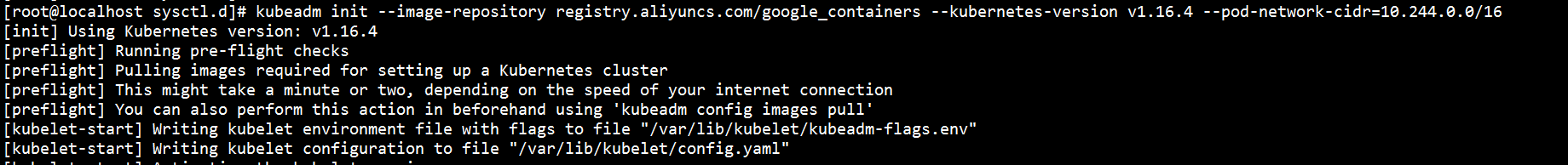
### Master节点初始化

kubeadm init --image-repository registry.aliyuncs.com/google\_containers --kubernetes-version v1.16.4 --pod-network-cidr=10.244.0.0/16

这一步，如果出现下面错误，则是前面的2.1.6节点内核没有配置



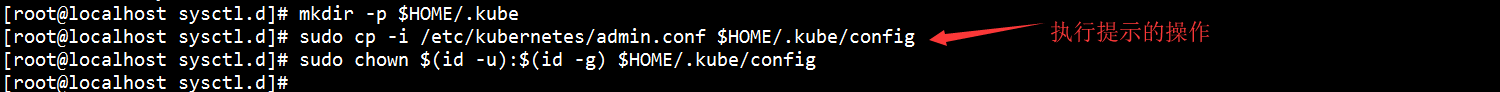
正常情况，如下：





出现这一步，恭喜你，已经成功一大半了！

接下来，我们按它的提示执行操作



### 添加flannel的网络

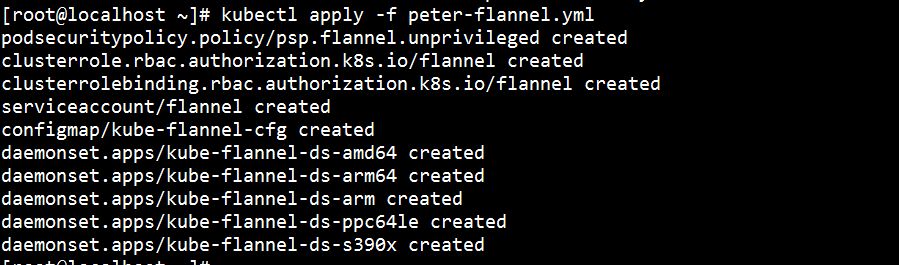
按照master的提示，我们接下来应该配置一个pod network。

但是，因为国内网络不通的原因，此操作无法完成。

你只能选择peter老师为你定制的下面这个文件来完成

上传文件到你的系统后，使用下面命令

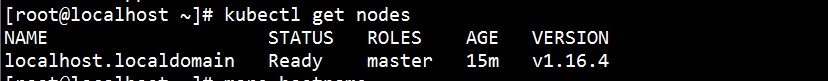
kubectl apply -f peter-flannel.yml



至此，大功告成

### 查看集群

查看你k8s集群



## work节点初始化

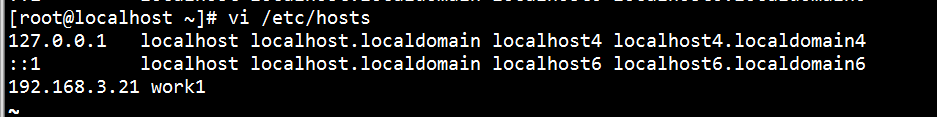
**work节点的配置，相对master来说简单许多，只需要规划好节点的名称即可**

### 设置机器名

设置一个机器名为work1

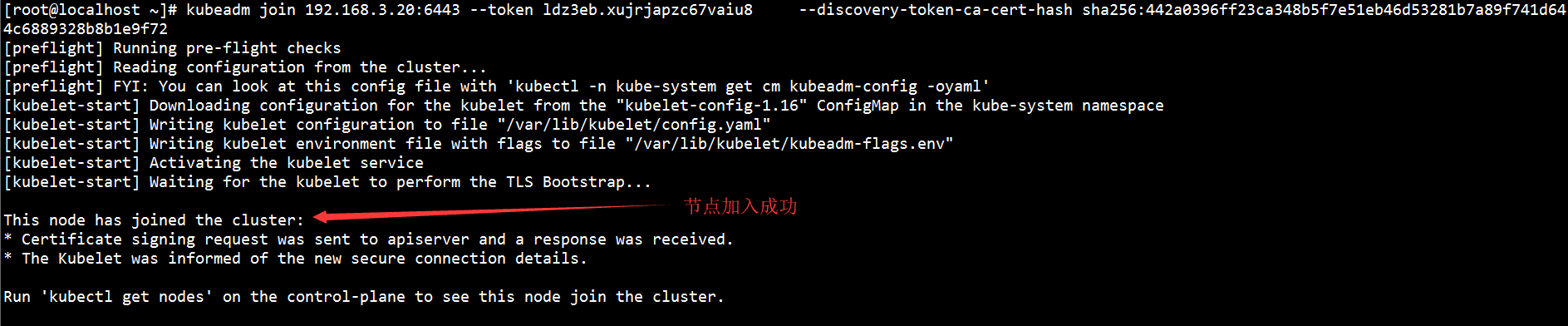


配置对应的ip

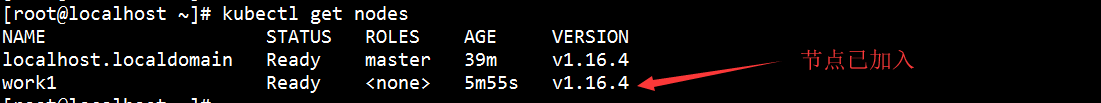
****

### 加入集群

在要加入的工作节点机器上，执行master 初始化时提示的join语句，即加到master的管辖内

****

**回到master节点再次查看集群**

****

# K8S基础操作

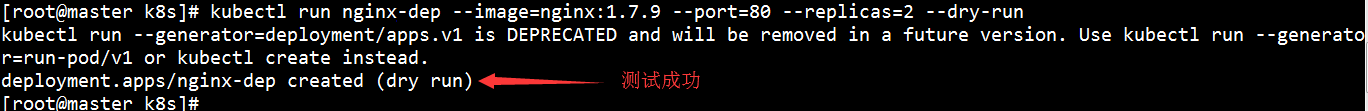
## kubectl的命令用法

可以借助kubectl -h命令学习用法，下面介绍常用的一些命令使用：

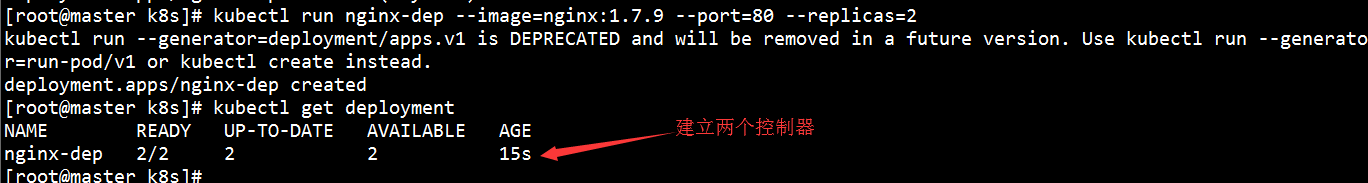
### kubectl run，创建一个应用程序

kubectl run nginx-dep --image=nginx:1.7.9 --port=80 --replicas=2

可以先测试：--dry-run



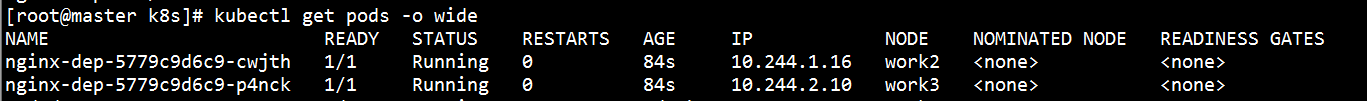
正式创建：



查看服务信息：

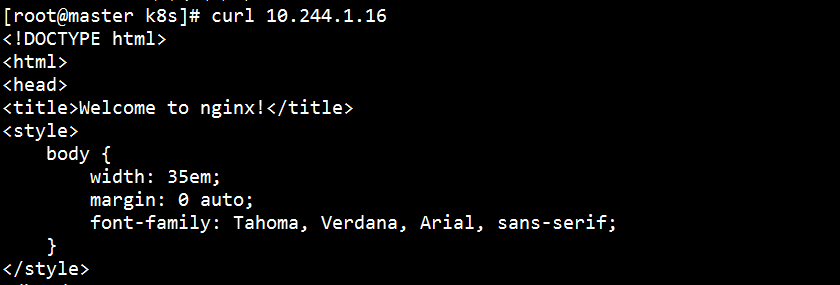
kubectl get pods -o wide

#获取pod的信息，-o wide 表示更详细的显示信息



看到系统里启动了两个pod服务（运行nginx），分别运行在节点node2和node3上

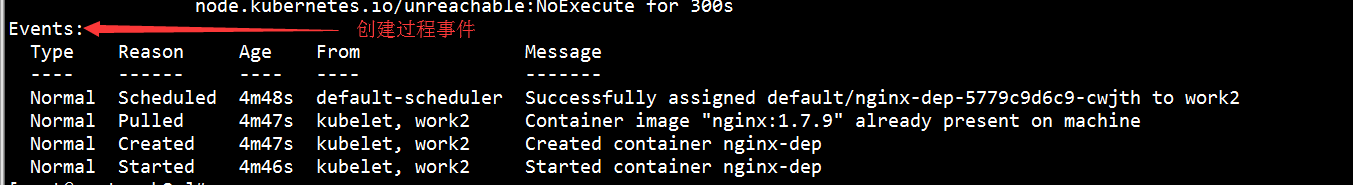
测试nginx服务，服务ok



### 探究pod详情：

kubectl describe pod nginx-dep-5779c9d6c9-cwjth

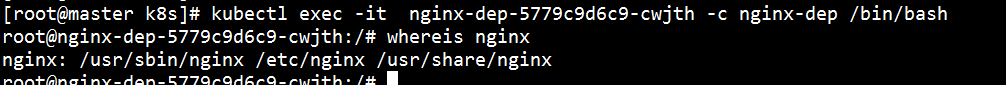




进入容器查看：

格式：kubectl exec -it podName  -c  containerName -n namespace -- shell comand

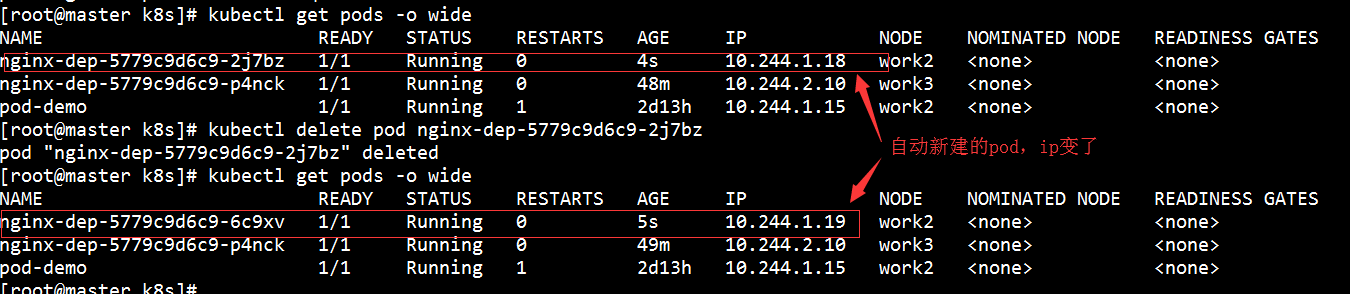
kubectl exec -it nginx-dep-5779c9d6c9-cwjth -c nginx-dep /bin/bash



### 暴露服务到外网

将pod创建完成后，访问该pod内的服务只能在集群内部通过pod的的地址去访问该服务；当该pod出现故障后，该pod的控制器会重新创建一个包括该服务的pod,此时访问该服务须要获取该服务所在的新的pod的地址ip去访问。

#删除当前的pod:



如何保持pod的故障恢复，对调用者无影响呢？

可以创建一个service，当新的pod的创建完成后，service会通过pod的label连接到该服务，只需通过service即可访问该服务。

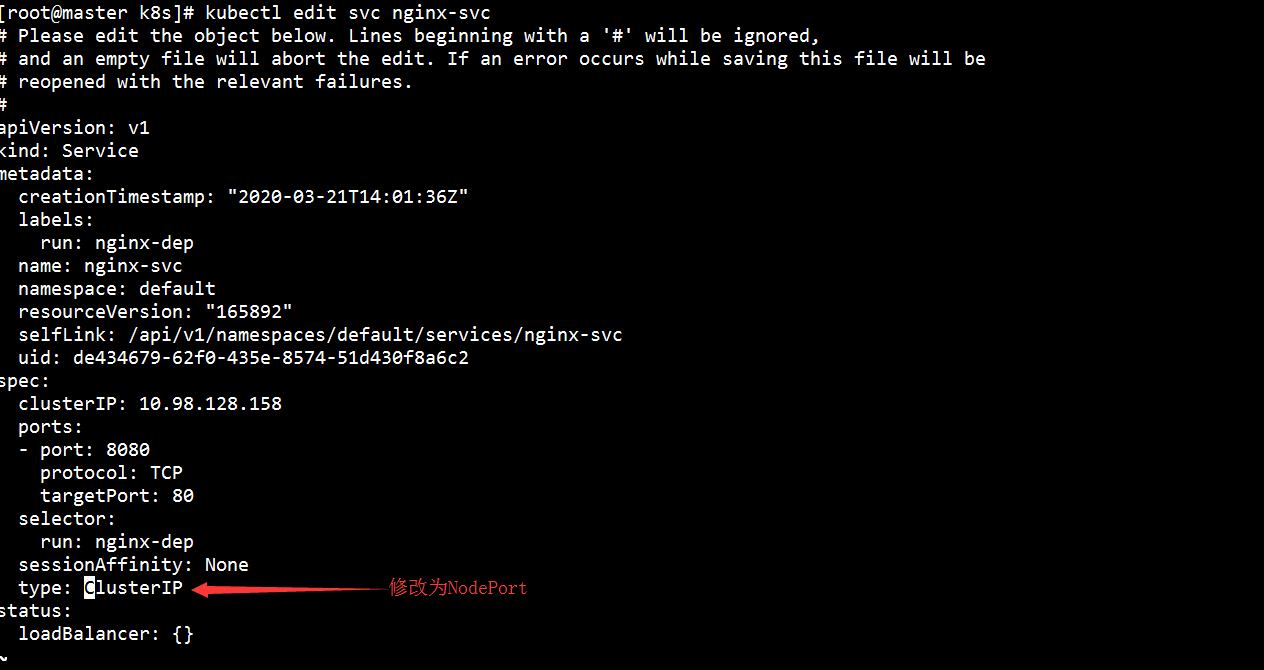


查看svc的label配置

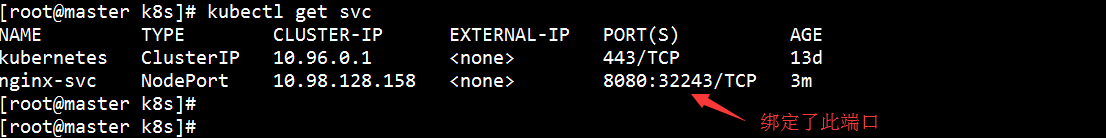


上述方式，虽然能够通过service访问到pod服务。但在集群外部，是无法访问到的，如在本地windows机器上，想要访问到nginx服务，网络是不通的。我们可以修改service的类型的NodePort。

kubectl edit svc nginx-svc



查看绑定端口

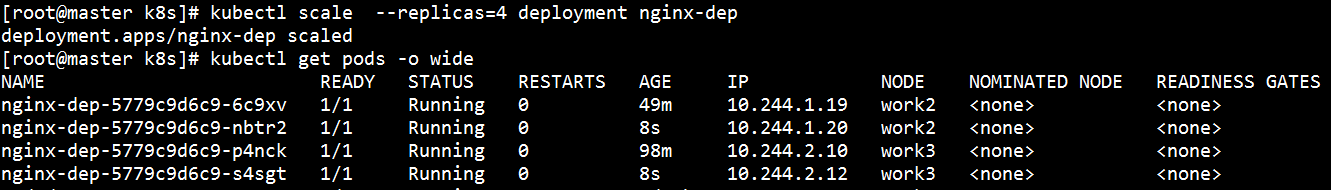


在外部可以通过node节点的地址及该端口访问pod内的服务。



### 服务的伸缩

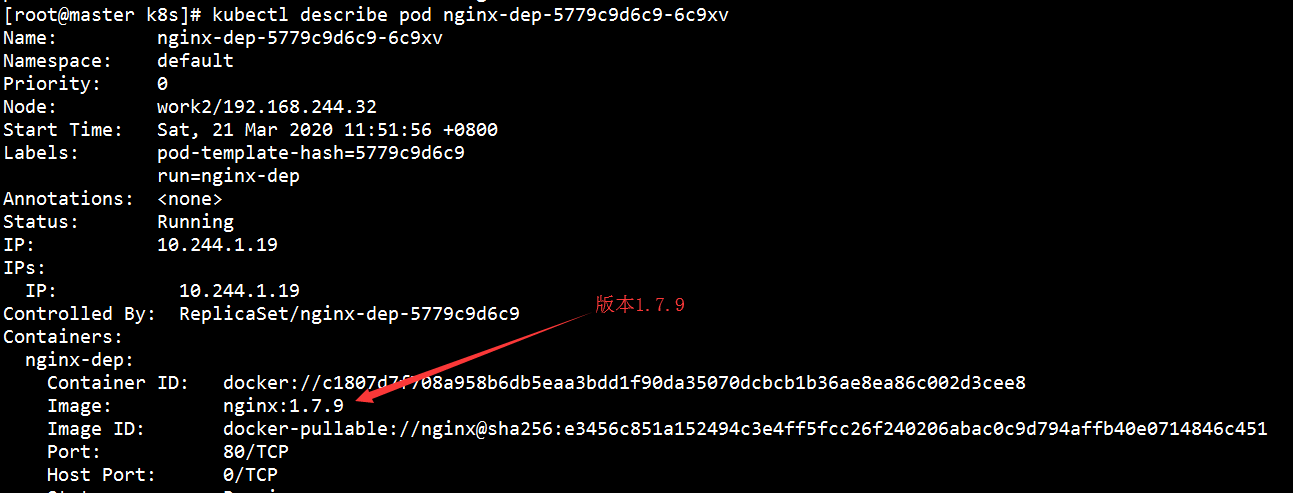
Pod创建完成后，当服务的访问量过大时，可以对pod的进行扩展让pod中的服务处理更多的请求；当访问量减小时，可以缩减pod数量，以节约资源。 这些操作都可以在线完成，并不会影响现有的服务。



缩减服务雷同

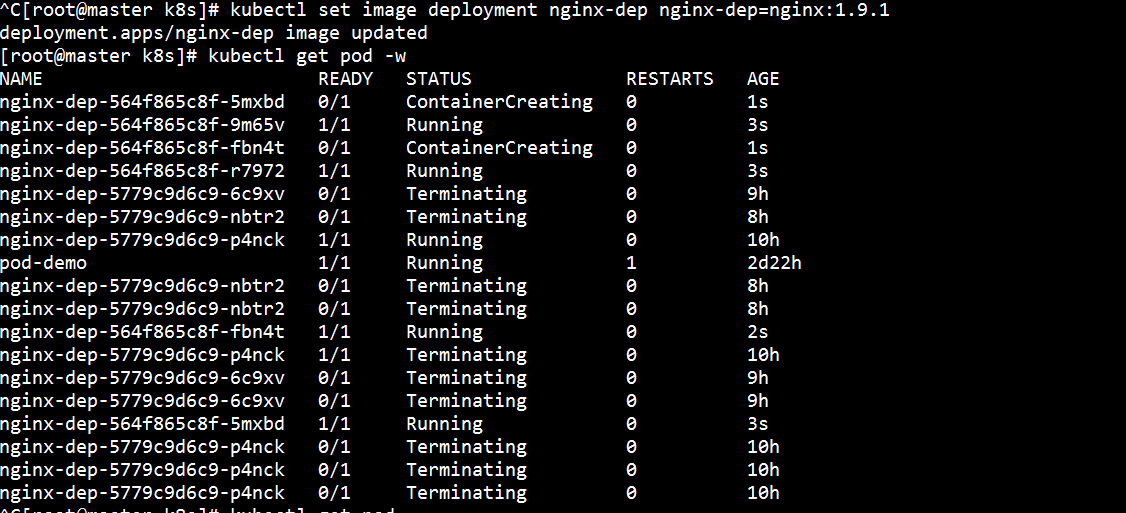
### 服务的在线升级与回滚

在kubernetes服务中部署完服务后，对服务的升级可以在线完成，升级出问题后，也可以在线完成回滚。、



可以看到滚动更新的过程：

kubectl get pod -w ##w参数是watch，持续执行，并观察改变

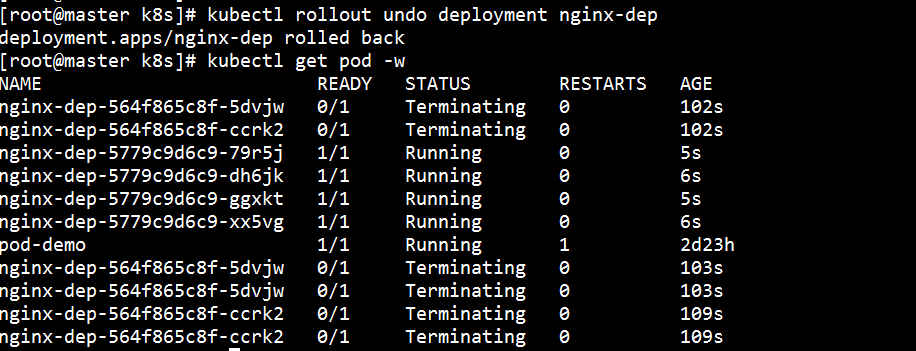


再次查看镜像版本



还可以再回滚回原来的版本：

kubectl rollout undo deployment nginx-dep



查看版本，又回到1.7.9的版本



## YAML文件管理资源

K8s两种创建资源的方式：

　　（1）用kubectl命令的方式直接创建：比如前面的创建deployment

　　（2）通过配置文件和kubectl apply创建，

正式的使用，一般采用第二种方式

### 资源清单的格式

apiVersion: group/apiversion # 不指定定group，默认为croe

kind: #资源类别

metadata： #资源元数据

name

namespace #k8s自身的namespace

lables

annotations #主要目的是方便用户阅读查找

spec:期望的状态（disired state）

status：当前状态，本字段有kubernetes自身维护，用户不能去定义

•缩进表示层级关系

•不支持制表符“tab”缩进，使用空格缩进

•通常开头缩进2 个空格

•字符后缩进1 个空格，如冒号、逗号等

•“---” 表示YAML格式，一个文件的开始

•“#”注释

### 创建deployment

# vi nginx-deployment.yaml

apiVersion: apps/v1 # 配置格式的版本

kind: Deployment # 创建的资源类型，这里是deployment

metadata: # 元数据

name: nginx-deployment

namespace: default

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

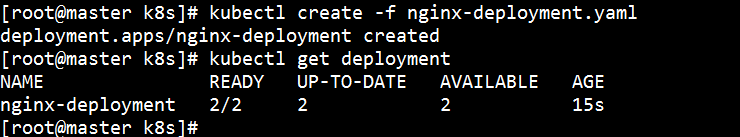
image: nginx:1.7.9

ports:

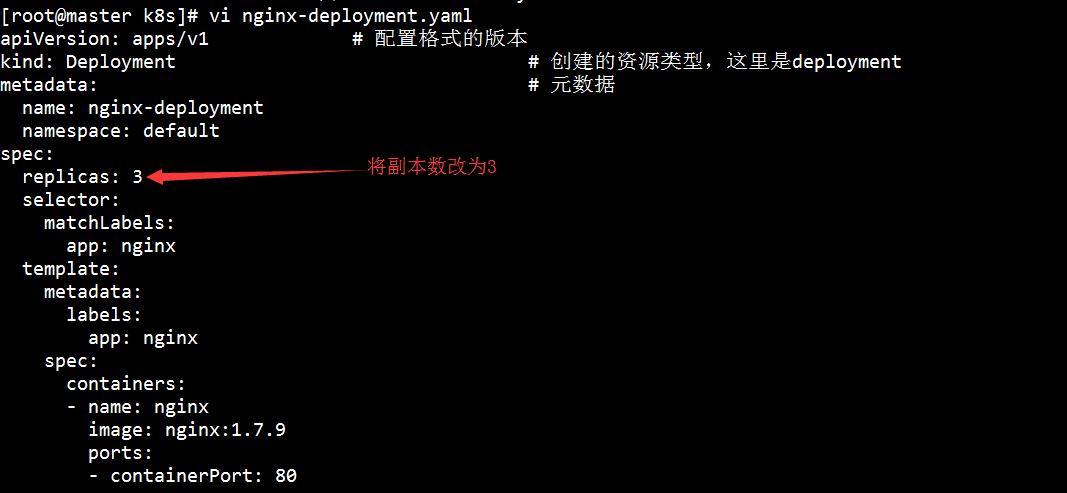
- containerPort: 80

此文件定义内容，效果与上一节中的命令方式相同。但配置更为详尽

#使用create 子命令以yaml文件的方式启动

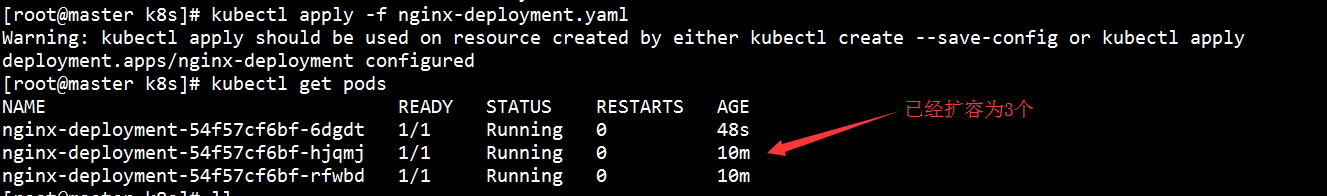
kubectl create -f nginx-deployment.yaml 

### 扩容伸缩



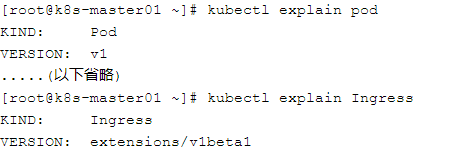
使用kubectl apply生效

kubectl apply -f nginx-deployment.yaml



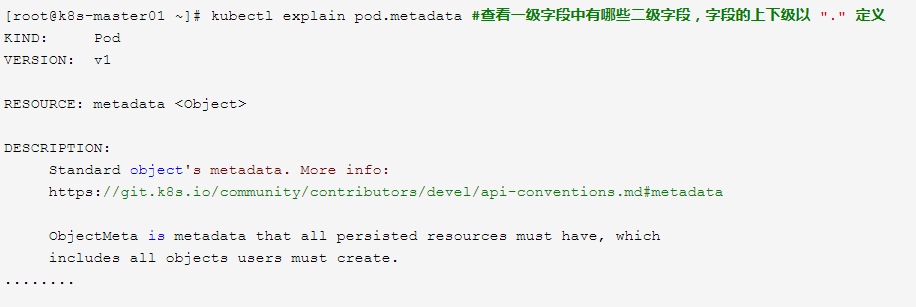
### 获取资源的apiVersion版本及资源配置的帮助

k8s中， 可以使用kubectl api-versions 获取当前k8s版本上所有的apiVersion版本信息(每个版本可能不同)

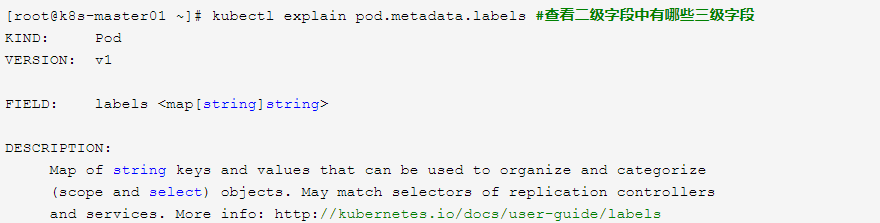


**可以看到出来，不同的资源属于不同的apiVersion版本**

你还可以查看资源的配置清单的二级级别的字段



三级字段



### 使用service提供外部访问

vim nginx-service.yaml

apiVersion: v1

kind: Service

metadata:

name: nginx-service

labels:

app: nginx

spec:

type: NodePort

ports:

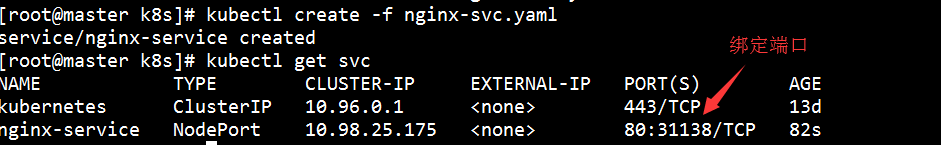
- port: 80

targetPort: 80

selector:

app: nginx

创建：kubectl create -f nginx-svc.yaml



浏览器访问ok



## [Pod控制器(kube-controller-manager)](https://www.cnblogs.com/panwenbin-logs/p/9907847.html)

kube-controller-manager 是Kubernetes 的大脑， 它通过 apiserver 监控整个集群的状态， 并确保集群处于预期的工作状态。

### ReplicaSets控制器

k8s中最初有个Replication Controller ，它保证了在所有时间内，都有特定数量的Pod副本正在运行，如果太多了，Replication Controller就杀死几个，如果太少了，Replication Controller会新建几个，和直接创建的pod不同的是，Replication Controller会替换掉那些删除的或者被终止的pod，不管删除的原因是什么（维护阿，更新啊，Replication Controller都不关心）。基于这个理由，我们建议即使是只创建一个pod，我们也要使用Replication Controller。

Replication Controller 就像一个进程管理器，监管着不同node上的多个pod,而不是单单监控一个node上的pod,Replication Controller 会委派本地容器来启动一些节点上服务（Kubelet ,Docker）。对于服务和用户来说，Replication Controller是通过一种无形的方式来维持着服务的状态

ReplicaSet是加强版的Replication Controller，其功能只是扩展了Replication Controller的selector，支持基于集合的selector（version in (v1.0, v2.0)或env notin (dev, qa)），K8s中一般我们不再使用Replication Controller。

### Deployment控制器

Deployment为Pod和Replica Set（下一代Replication Controller）提供声明式更新。意思就是，Replica Set一是在Deployment里使用的。

你只需要在Deployment中描述你想要的目标状态是什么，Deployment controller就会帮你将Pod和Replica Set的实际状态改变到你的目标状态。你可以定义一个全新的Deployment，也可以创建一个新的替换旧的Deployment。

典型的用法：

1.使用Deployment来创建ReplicaSet。ReplicaSet在后台创建pod。检查启动状态，看它是成功还是失败。

2.然后，通过更新Deployment的PodTemplateSpec字段来声明Pod的新状态。这会创建一个新的ReplicaSet，Deployment会按照控制的速率将pod从旧的ReplicaSet移动到新的ReplicaSet中。

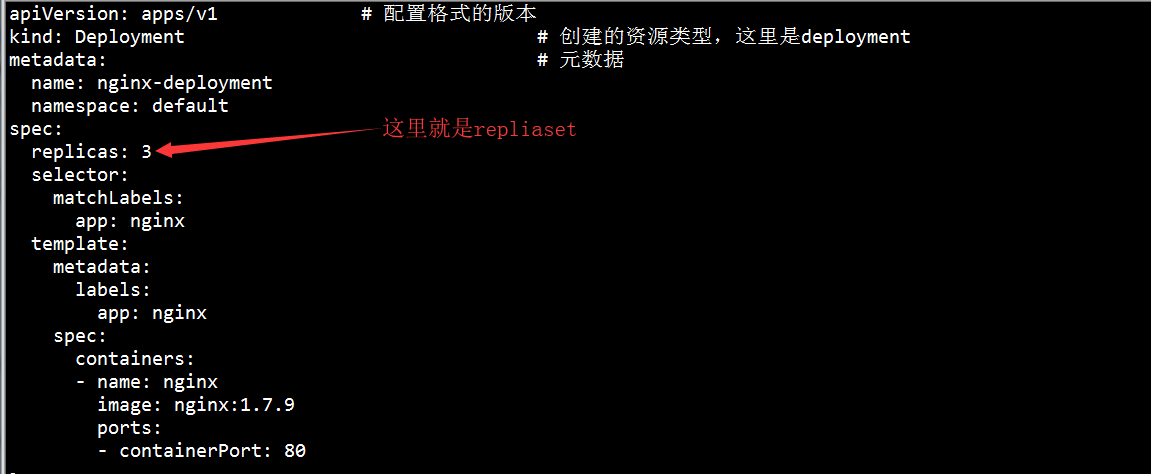
3.如果当前状态不稳定，回滚到之前的Deployment revision。每次回滚都会更新Deployment的revision。

4.扩容Deployment以满足更高的负载。

5.暂停Deployment来应用PodTemplateSpec的多个修复，然后恢复上线。

6.根据Deployment 的状态判断上线是否hang住了。

7.清除旧的不必要的ReplicaSet。



### DaemonSet

DaemonSet保证在每个Node上都运行一个容器副本，常用来部署一些集群的日志、监控或者其他系统管理应用。k8s内部就启用了这个，一般运维监控才用到它。日志收集，系统监控等

系统程序，比如kube-proxy, kube-dns, glusterd, ceph等都是DaemonSet

### StatefulSet

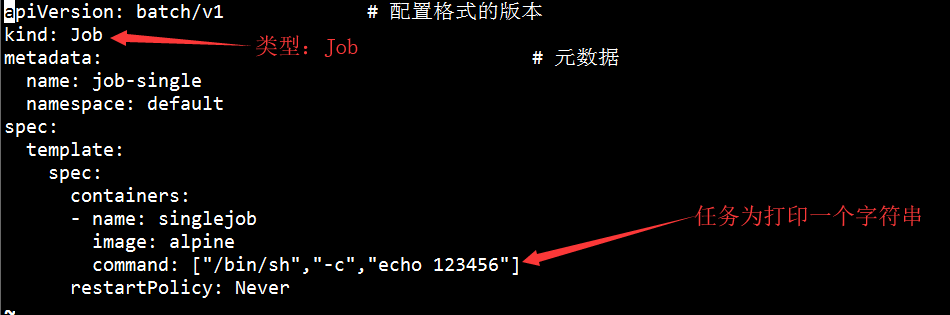
StatefulSet是为了解决有状态服务的问题（对应Deployments和ReplicaSets是为无状态服务而设计），我们后续讲完了K8s存储后，再回头来讲它的使用

### [Job控制器](https://www.cnblogs.com/caibao666/p/11207677.html)

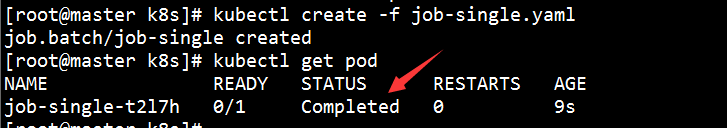
Job控制器用于调配pod对象运行一次性任务，容器中的进程在正常运行结束后不会对其进行重启，而是将pod对象置于completed状态。若容器中的进程因错误而终止，则需要依据配置确定重启与否，未运行完成的pod对象因其所在的节点故障而意外终止后会被重新调度。

job控制器对象有两种：

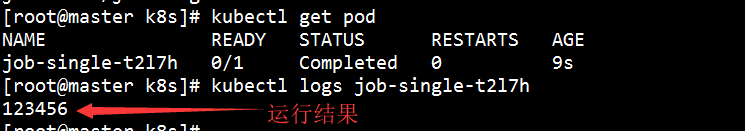
* 单工作队列的串行式job：即以多个一次性的作业方式串行执行多次作业，直至满足期望的次数



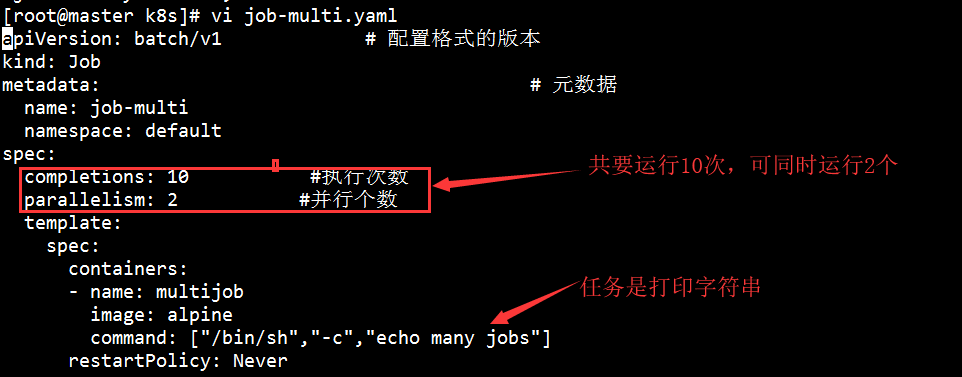
运行后，pod的状态：



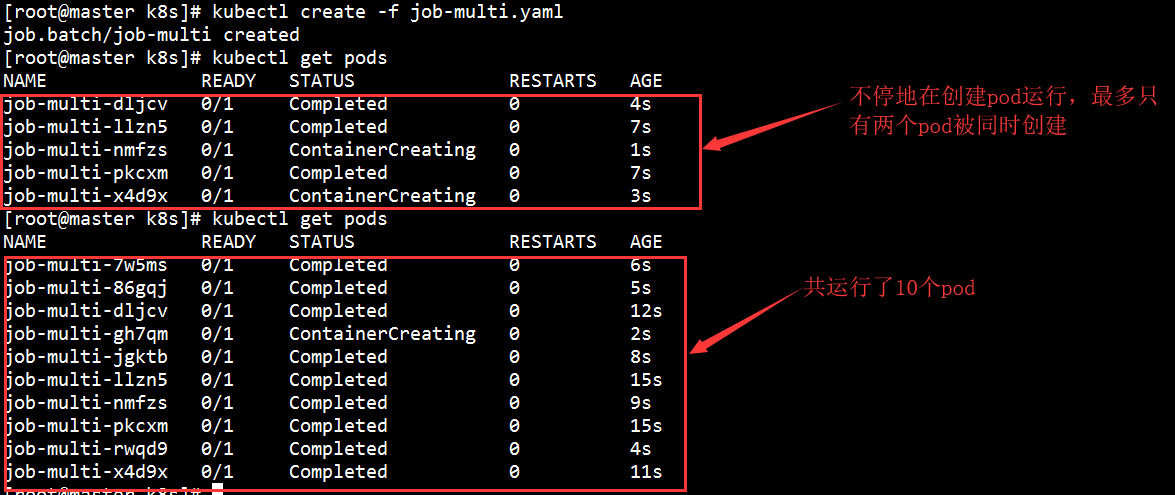
查看pod的运行日志



* 多工作队列的并行式job：这种方式可以设置工作队列数，即作业数，每个队列仅负责运行一个作业。

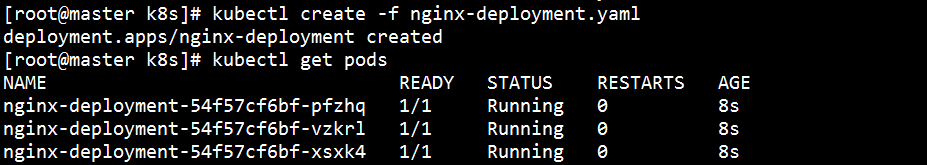


此任务运行后，查看其执行的pod个数

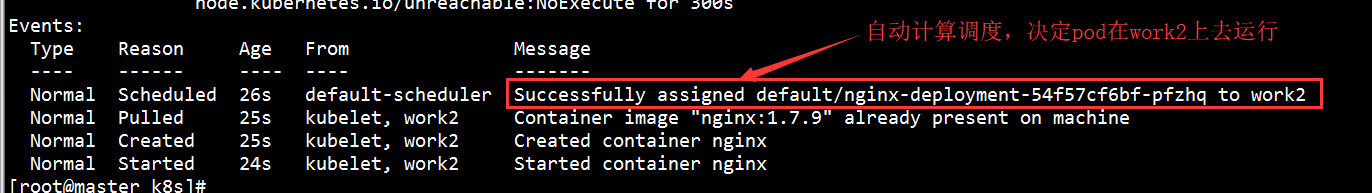


## Pod的调度

在k8s中当定义某个Pod对象时，若没有特定调度规则设定，则k8s本身会调用GenericScheduler通过预选优选算法来为该Pod选择一个最优Node节点，即最终Node节点是谁是不确定的。例如，我们前面经常用一deployment的过程：

比如这个创建:

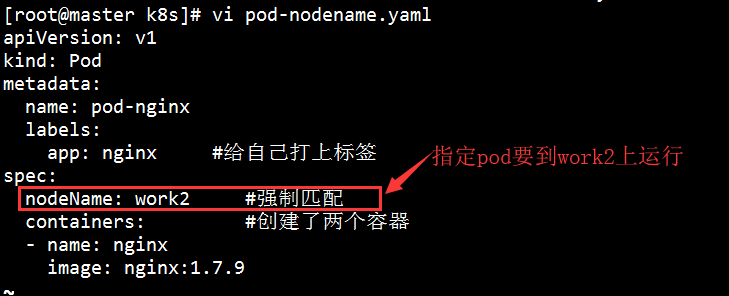
查看其中一个pod的事件列表，第一步都是计算调度的assigned决定：



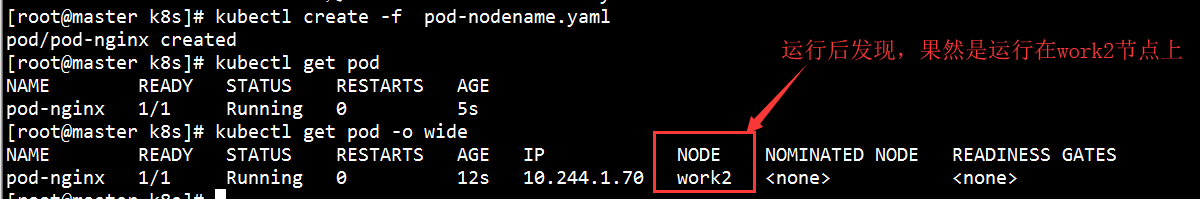
但在实际工作中，我们可能需要指向性地将某个Pod定向调度到某个Node中，K8s为我们准备了几种方式来实现这种需求。

### NodeName强制约束

我们可以使用NodeName指定，来强制约束pod要在某个node上运行

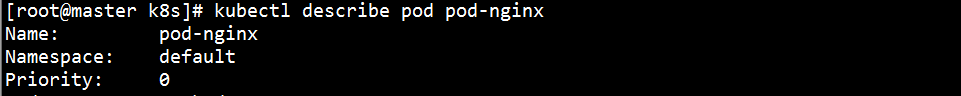


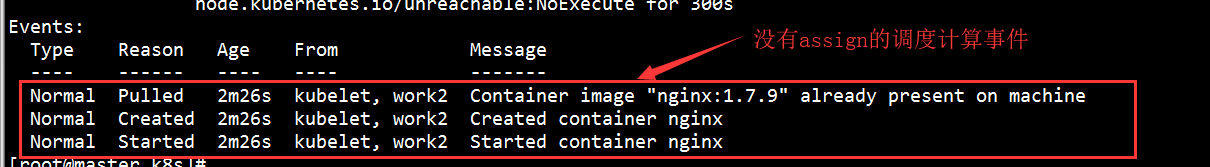
运行查验



实际上，这种node的指定是强制的，是直接取消掉的k8s的调度计算的

查看pod的事件列表：





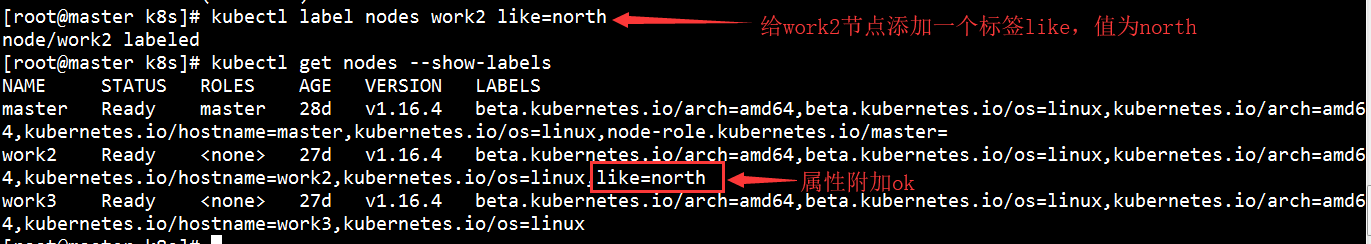
### NodeSelector定向调度

通过kubernetes的label-selector机制进行节点选择，由scheduler调度策略MatchNodeSelector进行label匹配，调度pod到目标节点，该匹配规则是强制约束，使用示例如下：

1. 给目标node打上一些标签，示例如下：

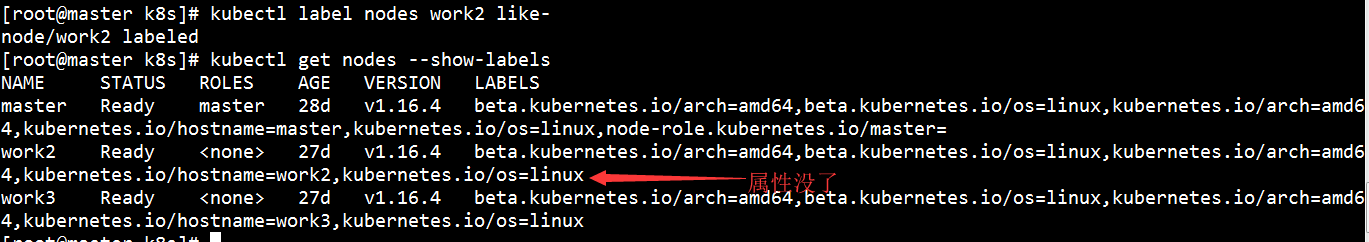
kubectl label nodes work2 like=north

其实就是为work2节点加一个属性，key和value值都随意

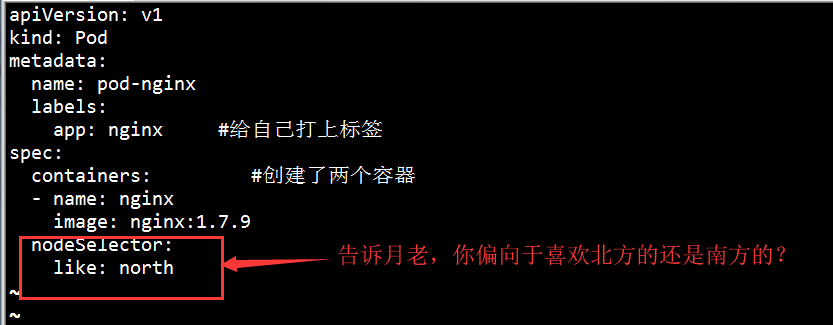


删除这个属性，只需要将key紧接一个“-”字符即可

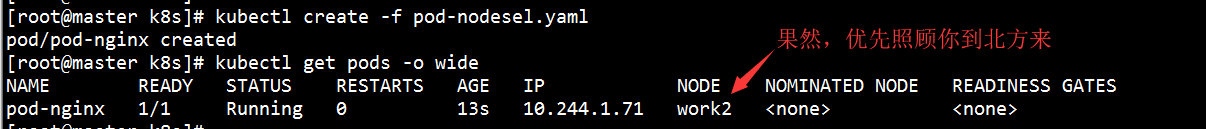
kubectl label nodes work2 like-



1. 在pod的定义加上nodeSelector设置（pod的偏向喜好）



运行这个pod，看它是否能得偿所愿



### NodeAffinity调度

NodeSelector调度算法比较简单，只是调度pod到某个拥有特定标签的Node上。而现实世界复杂的多，我们需要的是一系列的策略来决定，于是NodeAffinity出场，扩展了策略

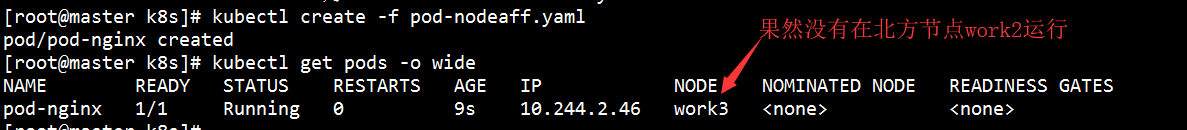
NodeAffinity 亲和性有两种表达方式：

* RequiredDuringSchedulingIgnoredDuringExecution ：必须满足指定的规则才可以调度Pod到Node上，相当于硬限制。
* PreferredDuringSchedulingIgnoredDuringExecution：强调优先满足指定的规则，相当于软限制，并不强求。

示例：



运行尝试，查验结果



PreferredDuringSchedulingIgnoredDuringExecution的用法，依次类推，希望你能举一反三。此外，此使用匹配语法灵活，你完全可以配出一系列实际效果，如pod节点的互斥性等等

## K8S的label作用

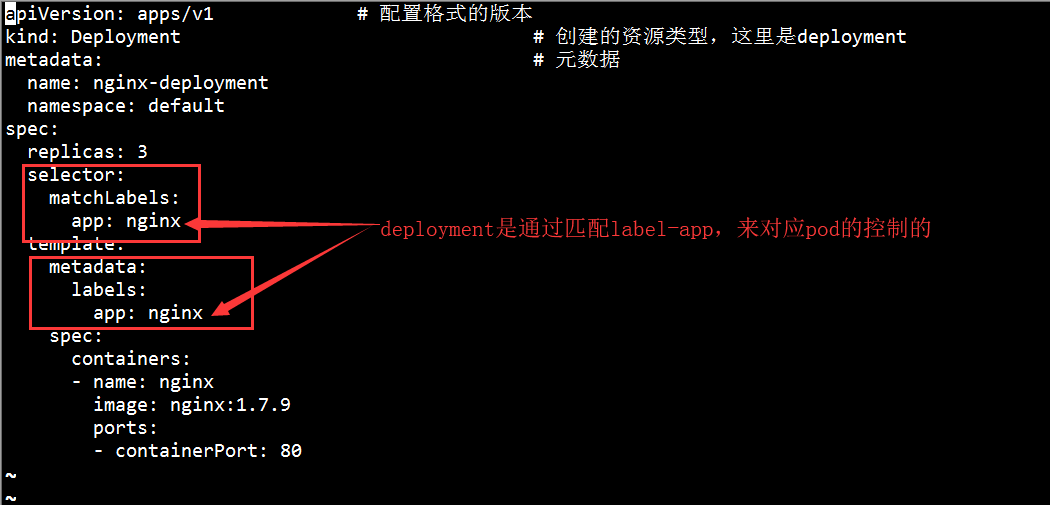
我们在上述的章节中，看到了k8s中一个特殊的存在：label。

在这里特别强调一下：

标签是一种简单却又功能强大的kubernetes特性，不仅可以组织pod，也可以组织所有其他的kubernetes资源，标签是可以附加到资源的任意键值对，用以选择具有该确切标签的资源，只要标签的key在资源内是唯一的，一个资源便可以拥有多个标签。

k8s除了使用标签来控制pod的调度之外，还有两个功能，也是使用label来实现的。

1.deploy控制器通过label找到对应控制的pods集群



2.Service通过label，来绑定port到对应的pods集群，从而提供稳定的服务



## Pod的健康检查

Kubernetes会维持Pod的状态及个数，而Pod内容器失败后往往也会导致pod退出，这么看来，K8s在某种程度上已经帮我们保证了容器服务的安全性。但也有很多场景，仅仅这样远远不够，比如某个时候容器服务假死，或者容器服务已经出错但并未退出。这些情况k8s的策略是无能为力的。

K8S提供了一处机制，来帮助我们来检查容器的健康的程度。

健康检查（Health Check）用于检测您的应用实例是否正常工作，是保障业务可用性的一种传统机制，一般用于负载均衡下的业务，如果实例的状态不符合预期，将会把该实例“摘除”，不承担业务流量。Kubernetes中的健康检查使用存活性探针（liveness probes）和就绪性探针（readiness probes）来实现，service即为负载均衡，k8s保证 service 后面的 pod 都可用，是k8s中自愈能力的主要手段。

目前支持的探测方式包括：

* HTTP
* TCP
* Exec命令

我们主要讲一下常见易用的Exec方式和Http方式

### 通过exec方式做健康探测

对于Exec探针，Kubernetes则只是在容器内运行命令。 如果命令以退出代码0返回，则容器标记为健康。 否则，它被标记为不健康。

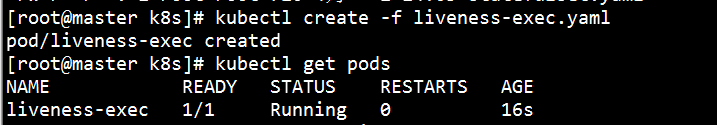
当您不能或不想运行HTTP服务时，此类型的探针则很有用，但是必须是运行可以检查您的应用程序是否健康的命令。

用法如以下示例：

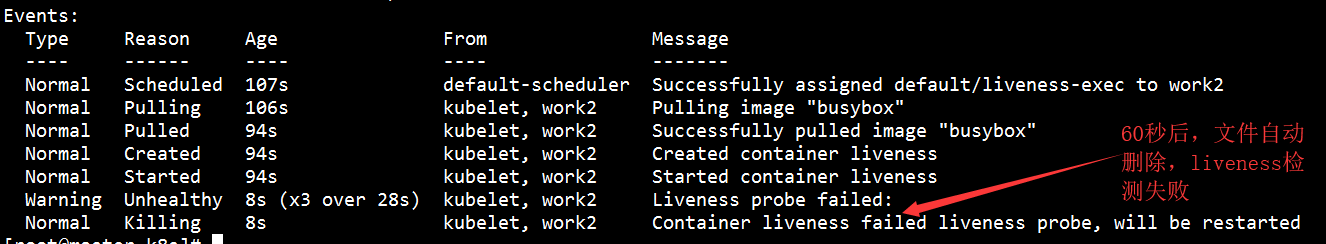


测试如下：

创建pod



查看pod事件



可以看到，在容器刚启动的60秒内，服务是正常的。此时healthy文件也在容器里。但容器启动后60秒后，healthy文件自动被删除，此时liveness检测无法探测到healthy文件，于是触发K8S机制剔除掉服务，并重建容器。

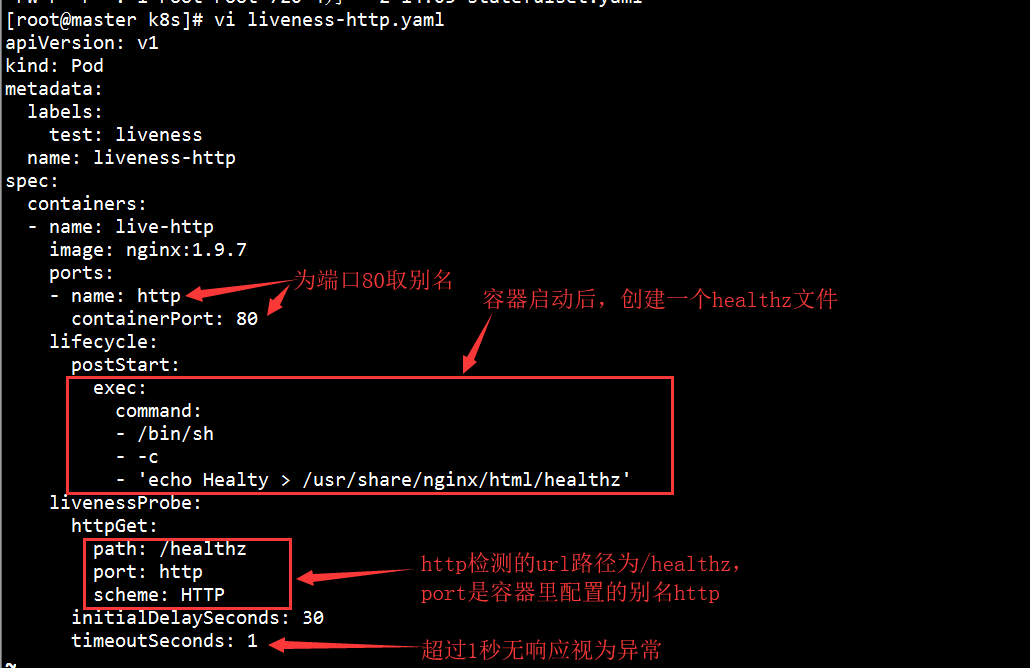
### 通过HTTP方式做健康探测

HTTP探针可能是最常见的自定义Liveness探针类型。 即使您的应用程序不是HTTP服务，您也可以在应用程序内创建轻量级HTTP服务以响应Liveness探针。 Kubernetes去访问一个路径，如果它得到的是200或300范围内的HTTP响应，它会将应用程序标记为健康。 否则它被标记为不健康。

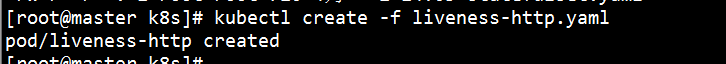
httpGet配置项：

* host：连接的主机名，默认连接到pod的IP。你可能想在http header中设置"Host"而不是使用IP。
* scheme：连接使用的schema，默认HTTP。
* path: 访问的HTTP server的path。
* httpHeaders：自定义请求的header。HTTP运行重复的header。
* port：访问的容器的端口名字或者端口号。端口号必须介于1和65535之间。

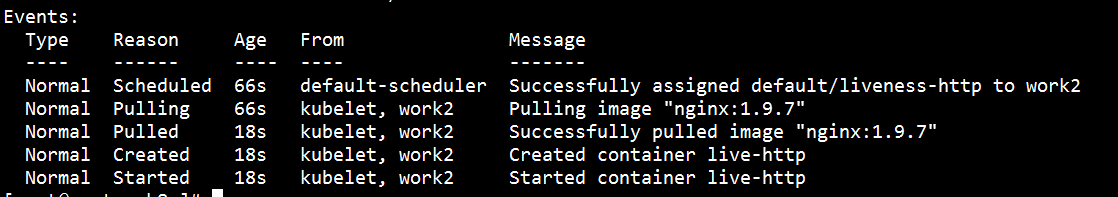
示例：



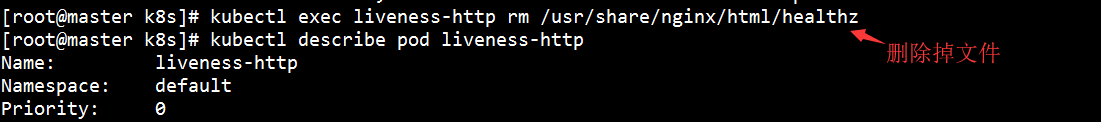
运行此pod实例

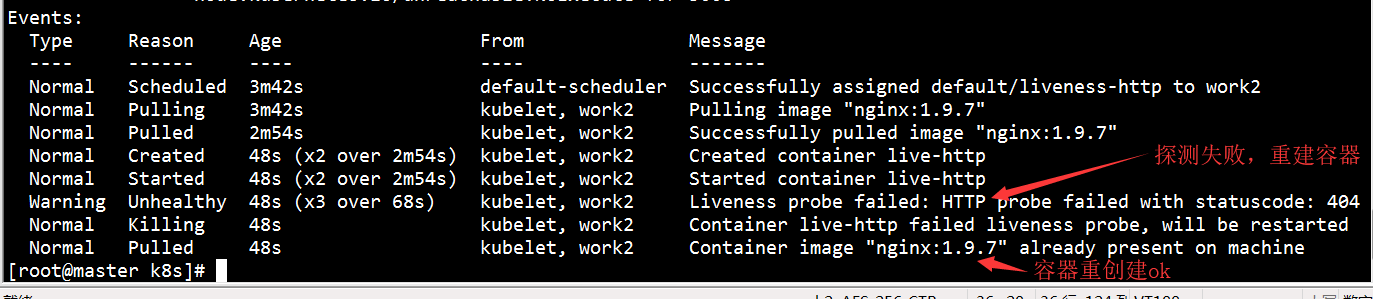


查看事件列表



此时，我们执行命令删除掉容器内的文件，看容器事件





# K8s存储

## 存储卷基础

在docker容器中，为了实现数据的持久性存储，在宿主机和容器内做映射，可以保证在容器的生命周期结束，数据依旧可以实现持久性存储。

但是在k8s中，由于pod分布在各个不同的节点之上，并不能实现不同节点之间持久性数据的共享，并且，在节点故障时，可能会导致数据的永久性丢失。为此，k8s就引入了外部存储卷的功能。

### emptyDir目录

emptyDir：Pod挂载在本地的磁盘或者内存，被称为emptyDir ，称为临时空目录，随着Pod删除，也会被删除。注意：一个容器崩溃了不会导致数据的丢失，因为容器的崩溃并不移除pod.

做一个empty数据卷示例：



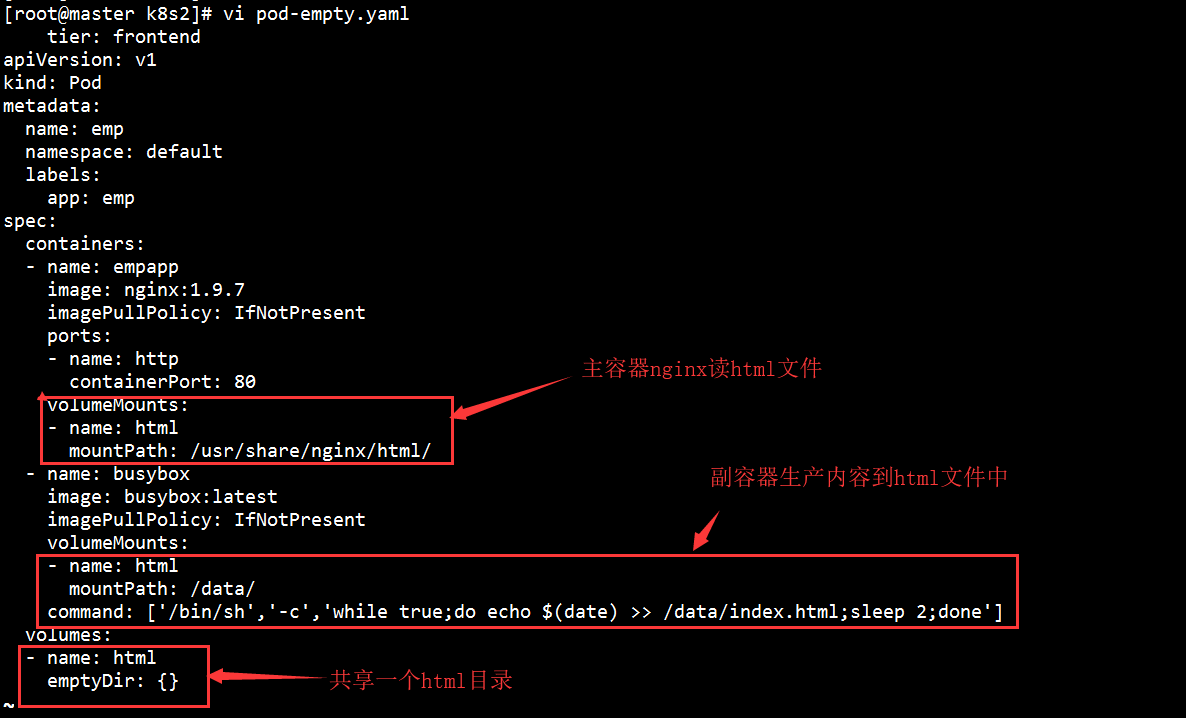
运行后查看，挂载描述如下：



使用场景，容器共享文件。在一个pod中有两类容器，一个是主容器，一个是辅助容器，两个容器使用同一个存储卷。

辅助容器用来生成新的内容，主容器加载使用。

示例2



运行后，可看到页面内容不断增加



### gitrepo目录

gitrepo实际上并不是一个新的存储类型，只是emptydir上补添一个git命令来拉取文件而已。

当pod创建时候，会拉取git(依赖于宿主机git命令驱动)仓库中数据克隆到本地，并且作为存储卷定义在pod之上。gitrepo基于emptyDir,此存储卷是emptyDir，git仓库中拉取的代码存放在emptyDir后被定义到pod。

因为拉取git仓库的动作，仅仅是在pod创建时触发一次，并不会待续更新文件，因此在工作中基本上无用处。有兴趣的同学可以自行研究一下。

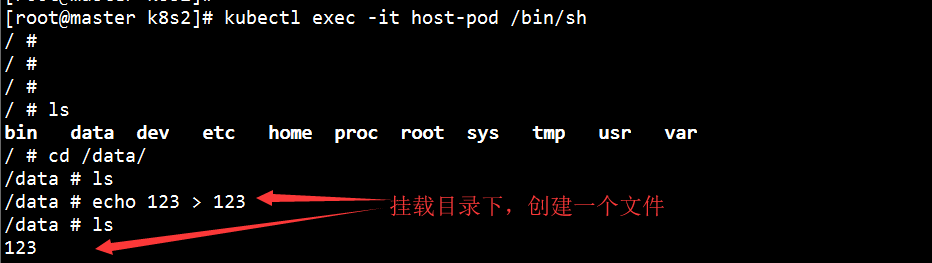
### hostPath目录

hostPath类型则是映射node文件系统中的文件或者目录到pod里。可实现针对某一节点的数据持久化，如果节点宕机了，那数据就丢失了

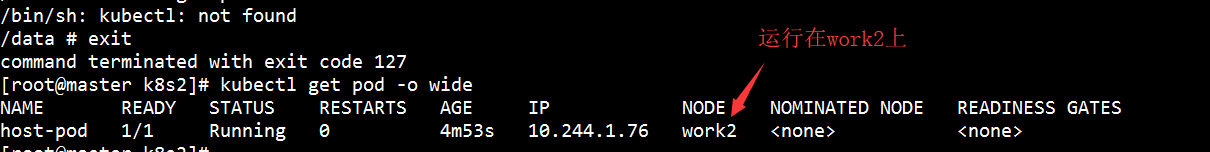
示例：



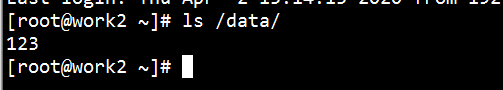
进入pod中创建一个测试文件



回到pod运行的node机器上查看，是否存在了这个测试文件



果然有此文件



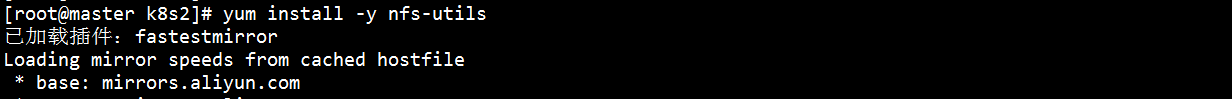
此后，删除掉pod，此文件也依然存在，同学们可自行去验证

### nfs共享存储卷

nfs使的我们可以挂在已经存在的共享到的我们的Pod中，nfs不会被删除，仅仅是解除挂在状态而已，这就意味着NFS能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间相互传递.并且，nfs可以同时被多个pod挂在并进行读写

#### 节点安装配置nfs

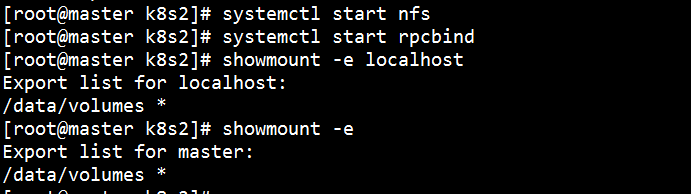
安装nfs：



配置nfs:



启动并查看：



#### node节点挂载

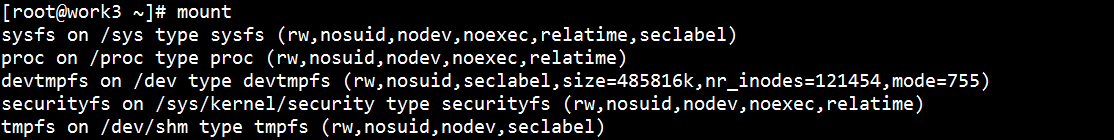
需要在所有节点安装 `nfs-utils` 组件，否则当Pod被分配到没有组件的节点，会启动失败，因为没有`mount.nfs`

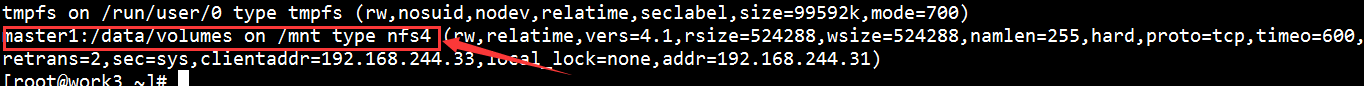
安装nfs：yum install -y nfs-utils

挂载nfs：mount -t nfs master1:/data/volumes /nfs（没有则先新建nfs目录）

解除挂载：umount /nfs

查看挂载效果：mount





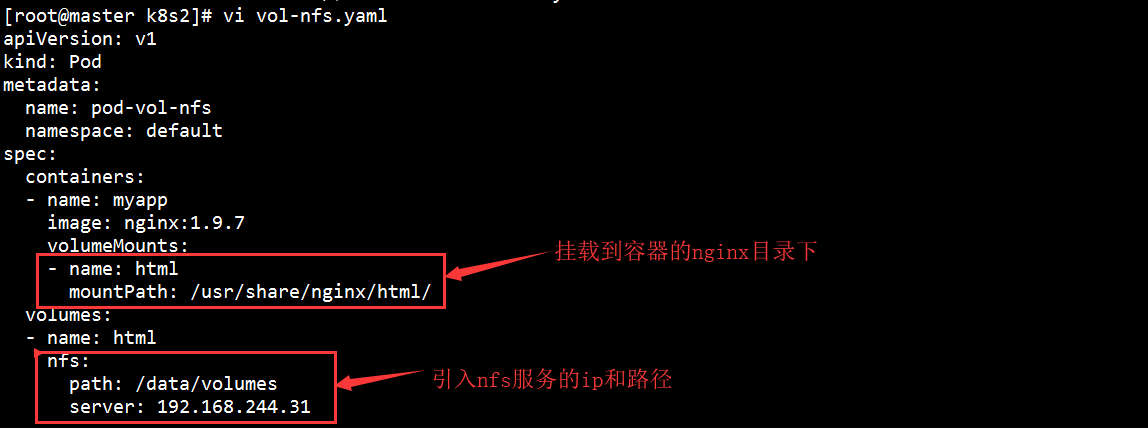
进入node节点，可查看/mnt目录下文件与master已互通

#### 测试NFS

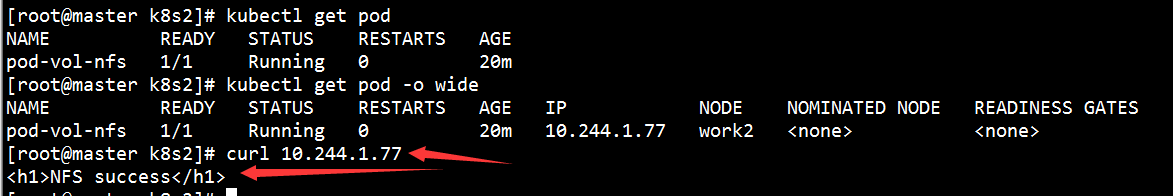
先在nfs服务内，加一个html测试页面

echo '<h1>NFS success</h1>' > /data/volumes/index.html

创建一个pod测试：



运行pod后测试，可看到nginx输出页面



你还可以测试pod的删除重建，丝毫不影响pod服务效果

## k8s的namespace概念

一个namespaces可以理解为kubernetes集群中的一个虚拟化集群。

在一个Kubernetes集群中可以拥有多个命名空间，它们在逻辑上彼此隔离。 这样可以方便我们区分各个团队的不同服务，增加安全甚至性能方面的帮助！

K8S默认已经为了我们初始化了三个namespace：

* default：你的service和app默认被创建于此。
* kube-system：kubernetes系统组件使用。
* kube-public：公共资源使用。但实际上现在并不常用。

总而言之，namespace是一种为了你方便管理服务的辅助，你可以根据需要创建若干个namespace供业务服务分组治理。从另一方面来说，K8s的任何一个服务，都是归属于某一个namespace的。

## PVC和PV

通过上面的存储卷的基础知识，我们大概了解了在K8S中，要持久化存储数据，最好还是存入到第三方服务里更安全，如NFS等。而在业界，这样的存储很多，使用方式五花八门，这就要求运维工程师必须要精通这些存储才能用好它。

PersistentVolume（pv）和PersistentVolumeClaim（pvc）就是k8s提供的两种API资源，用于抽象存储细节：

* 管理员关注于如何通过pv提供存储功能而无需如何使用
* 用户只需要挂载pvc到容器中而不需要关注存储卷采用何种技术实现。

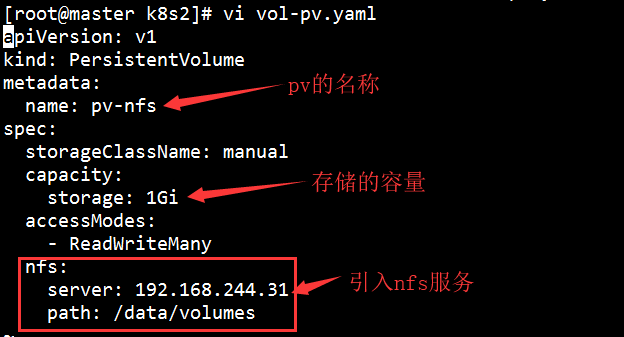
pvc和pv的关系与pod和node关系类似，前者消耗后者的资源。pvc可以向pv申请指定大小的存储资源并设置访问模式。

接下来我们使用PV与PVC的方式，实现上面nfs模式的资源挂载效果。

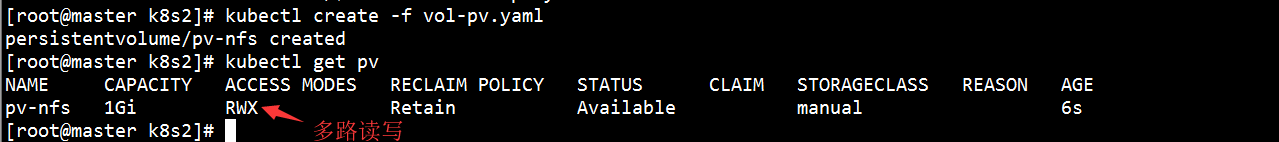
定义一个PV资源

### PV概念与使用

PersistentVolume（**PV**）是集群中由管理员配置的**一段网络存储**。 它是集群中的资源，就像节点是集群资源一样。因此PV是全局资源，不归属于任何namespace的。



创建PV



此处的RWX是PV资源的AccessModes （**访问模式）**，有以下几种模式。

* ReadWriteOnce（RWO）：读写权限，但是只能被单个节点挂载
* ReadOnlyMany（ROX）：只读权限，可以被多个节点挂载
* ReadWriteMany（RWX）：读写权限，可以被多个节点挂载

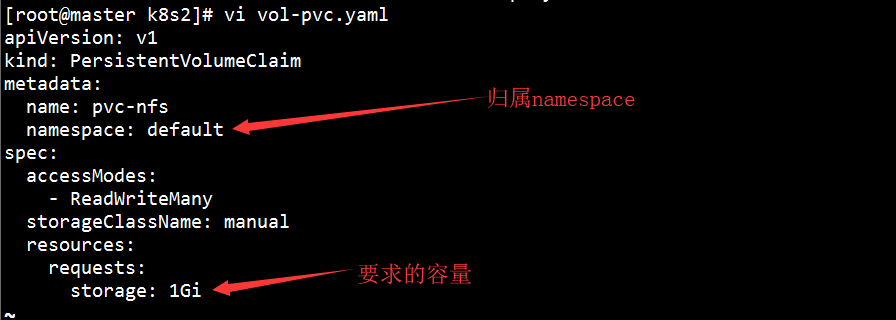
此处的NFS资源，是支持多个pod同时挂载使用的。

**同时，PV同一时间只能被一个PVC申请使用，这里有个状态变迁过程：**

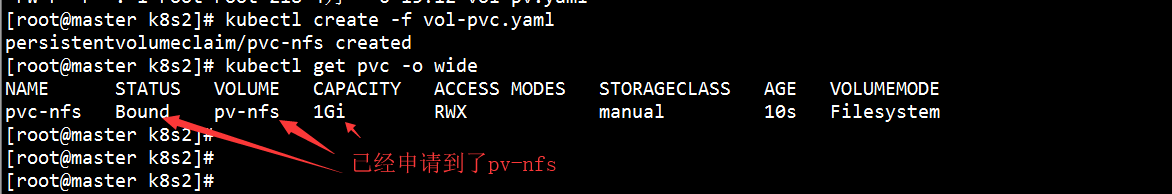
* Available（可用）：表示可用状态，还未被任何 PVC 绑定
* Bound（已绑定）：表示 PV 已经被 PVC 绑定
* Released（已释放）：PVC 被删除，但是资源还未被集群重新声明
* Failed（失败）： 表示该 PV 的自动回收失败

### PVC的概念与使用

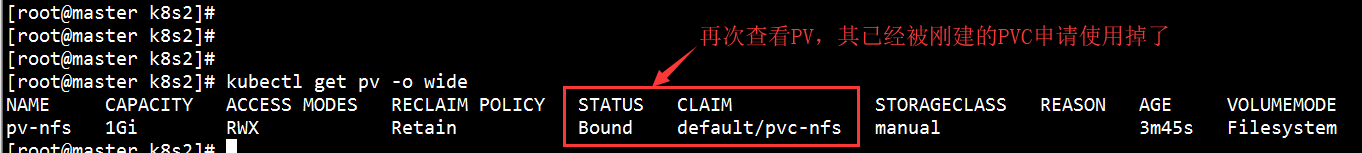
PersistentVolumeClaim（**PVC**）是由**用户进行存储的请求**。 它类似于pod。 Pod消耗节点资源，PVC消耗PV资源。因此PVC是归属于某个namespace的。



创建PVC



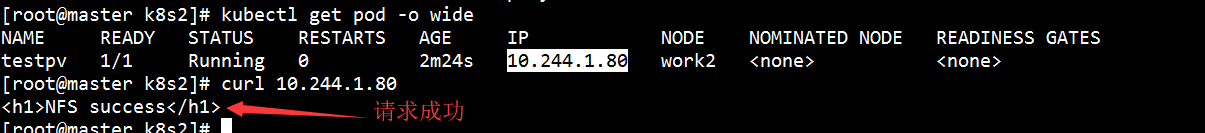
现次查看PV的状态，发现其已经被绑定使用了



测试一个pod的使用效果，将上面的pvc引入这里使用

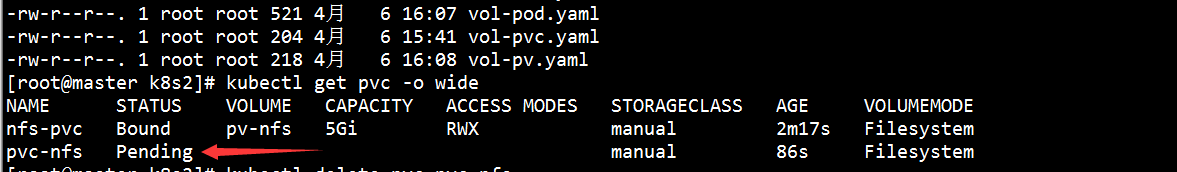


查看效果

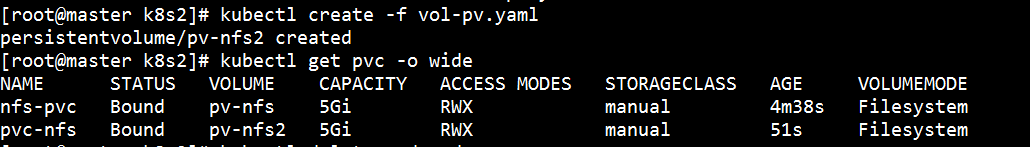


### PVC的阻塞

当PVC没有足够的PV来满足时，状态为挂起pending



如果有PV被释放，或者新建了PV能满足请求时，自动绑定占用



## statefulset控制器

现实工作中，我们的应用，分两类，**有状态和无状态**

* 无状态的, 关注的是群体，如nginx，tomcat等
* 有状态的, 关注的是个体，如redis，mysql等

我们前面的控制器，对无状态的应用操作很好，但对有状态的应用完全不实用。

**有状态应用集的特点:**

1. 稳定且需要唯一的网络标识符;

如: Redis集群,  
在Redis集群中，它是通过槽位来存储数据的，假如:第一个节点是01000,第二个节点是10012000,第三个节点2001~3000...等等，这就使得Redis集群中每个节点要通过ID来标识自己，如:  
第二个节点宕机了，重建后它必须还叫第二个节点，或者说第二个节点叫R2，它必须还叫R2，这样在获取1001~2000槽位的数据时，才能找到数据，否则Redis集群将无法找到这段数据。

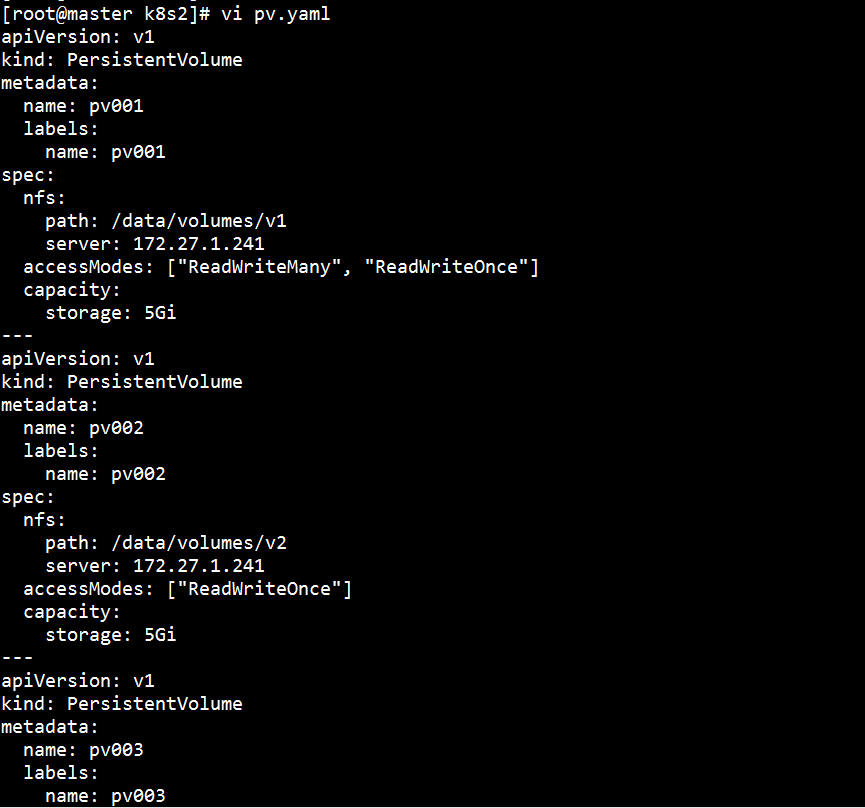
1. 稳定且持久的存储;

可实现持久存储，新增或减少pod，存储不会随之发生变化。

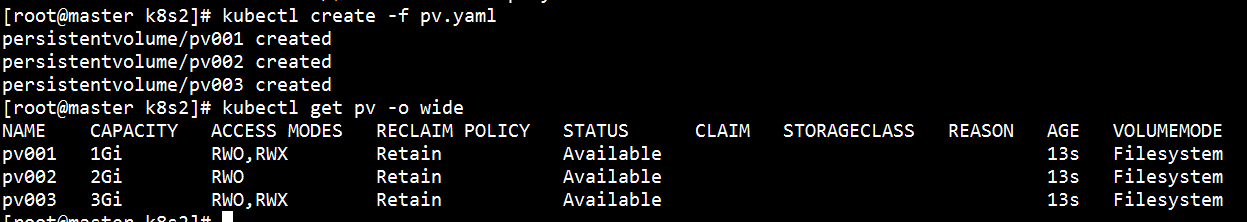
针对这样的特点，k8s推出StatefulSet

### PV资源配置

制做三个基础PV资源



创建PV

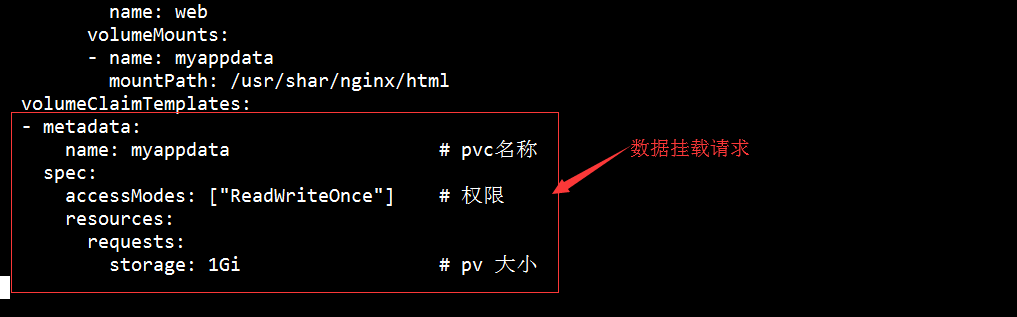


### 创建StatefulSet服务

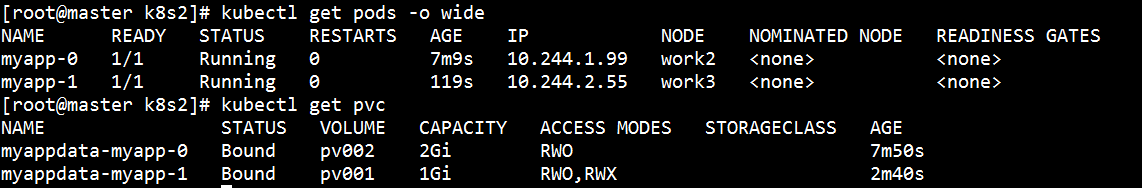
StatefulSet资源清单包含三个部分：

1. headless service 用于定义网络标识（DNS）
2. StatefulSet 控制器,用于定义具体应用
3. volumeClaimTemplate 存储卷申请模板,用于创建PVC





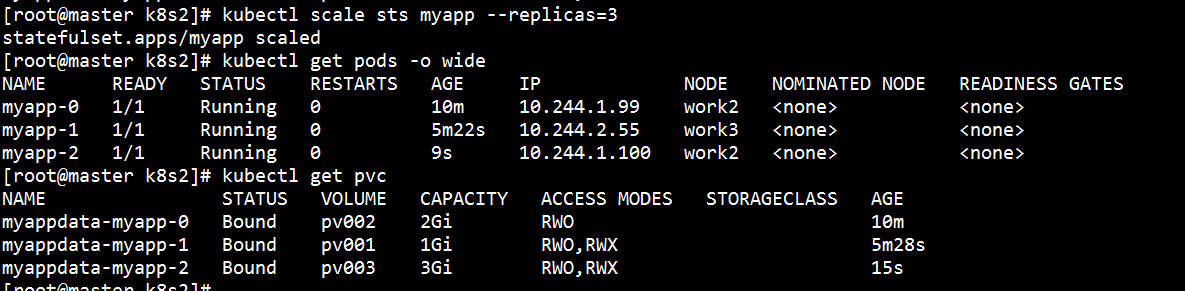
创建服务：



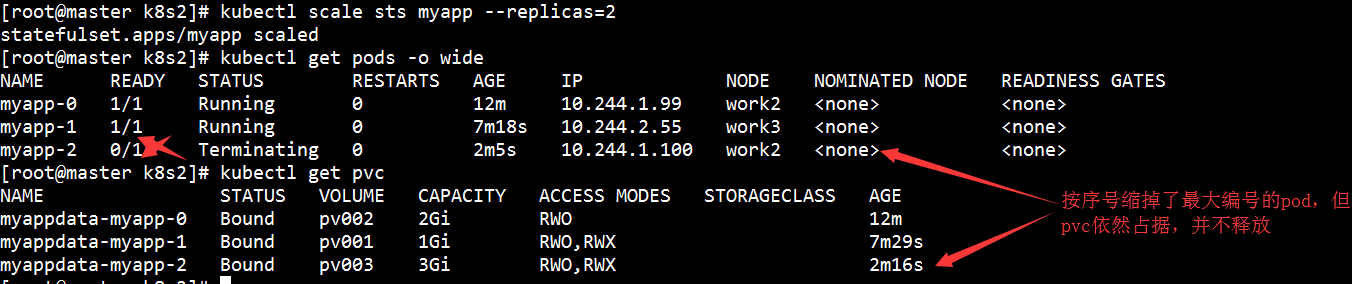
查看pod，发现sts创建的pod集群，是有序编号的，从第0个开始，依次创建。

查看pvc列表（sts自动创建管理的），它与上面的pod是序号一一对应的。也就是说，此pod重启重建，对应的pvc都是一样的

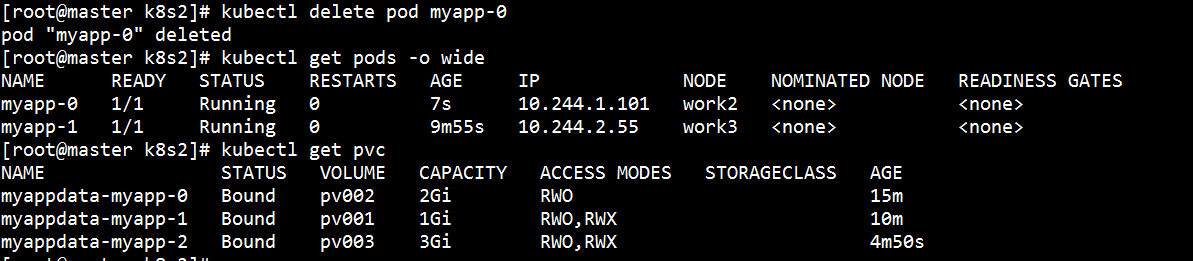
服务扩容：



可以看到，pod扩展，会连带pvc一起扩容。但是缩容却不会



删除pod：



可以看到，新建出来的pod名称/node都是不变的，对应 的PVC当然也不变。

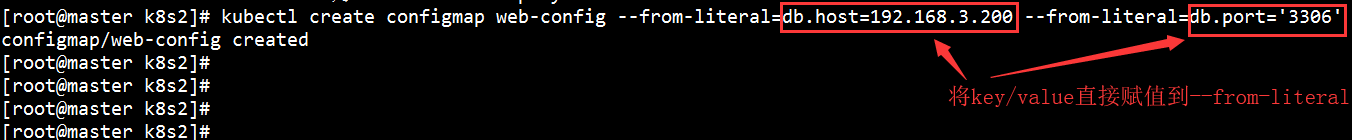
## configmap与secret

secret和configmap可以理解为特殊的存储卷，但是它们不是给Pod提供存储功能的，而是提供了从集群外部向集群内部的应用注入配置信息的功能。ConfigMap扮演了K8S集群中配置中心的角色。

ConfigMap定义了Pod的配置信息，可以以存储卷的形式挂载至Pod中的应用程序配置文件目录，从configmap中读取配置信息；也可以基于环境变量的形式，从ConfigMap中获取变量注入到Pod容器中使用。

### 命令行方式创建ConfigMap

* --from-literal=key=value格式直接赋值，示例如下：

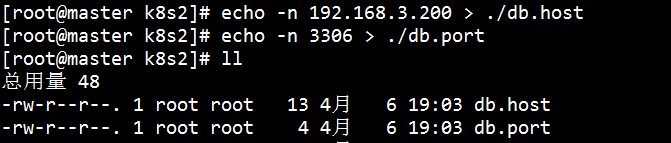


查看这个configmap信息：

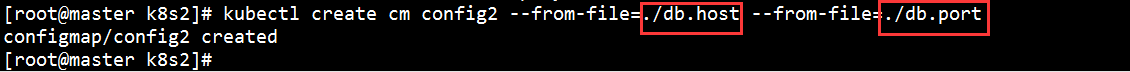


* --from-file=path方式，将文件中信息存入configmap

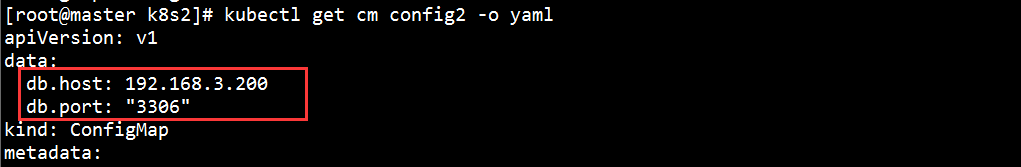
先将值放入文件中



再合建configmap

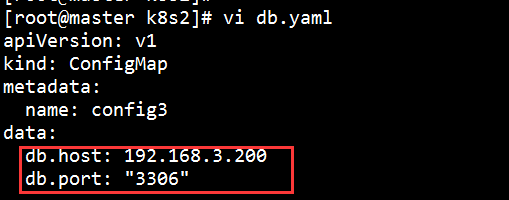


查看信息，效果一样

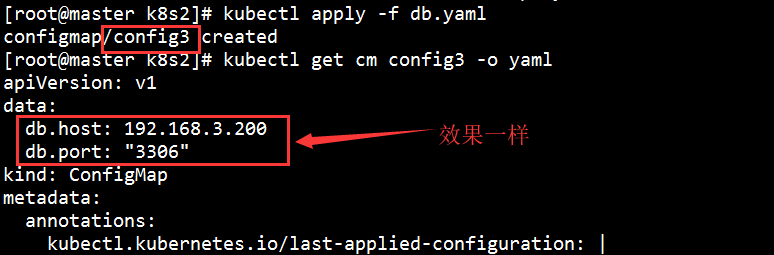


### YAML 配置文件方式创建ConfigMap

直接创建一个yaml文件，配置好信息

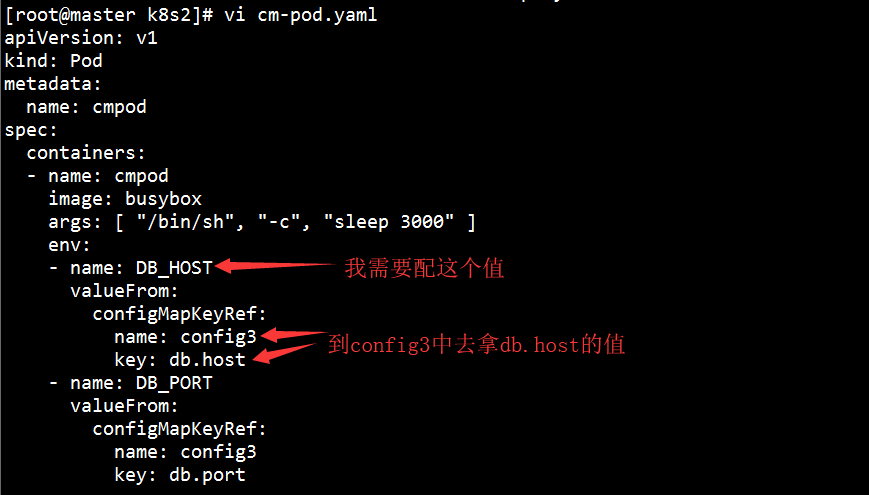


创建并查验信息：

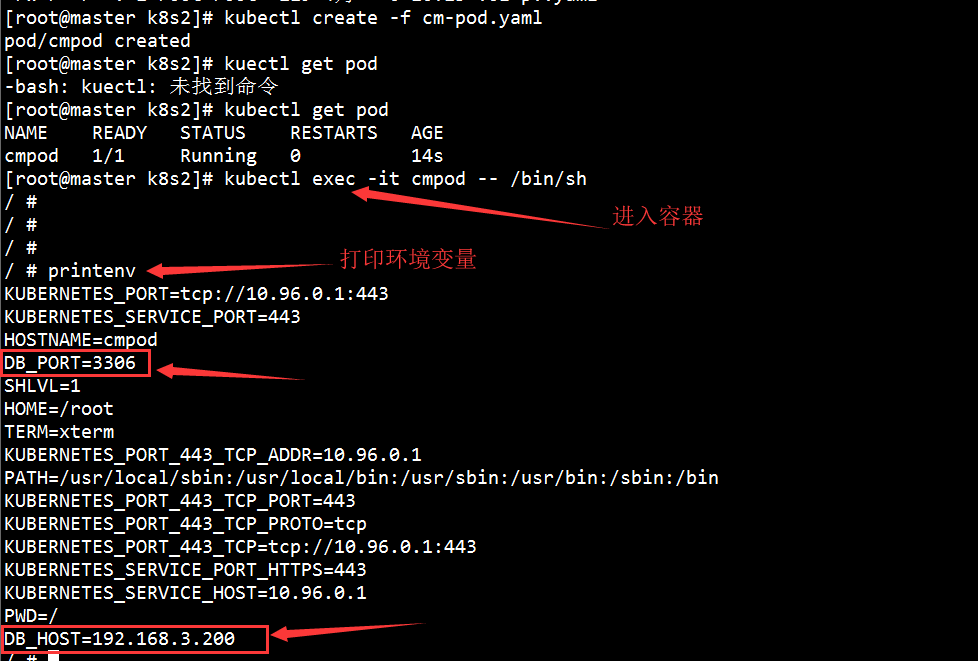


### 环境变量方式使用configmap

使用valueFrom、configMapKeyRef、name、key指定要用的key。

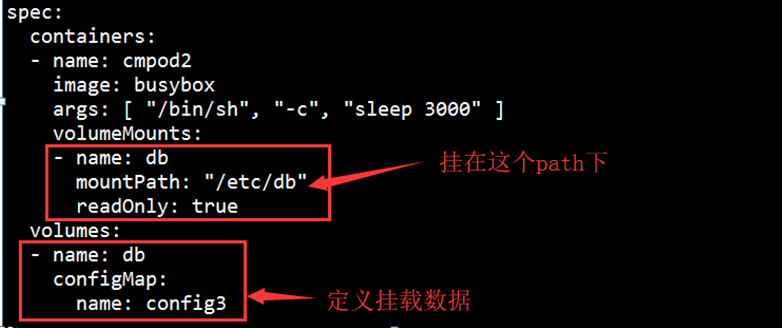


运行pod后，进入pod容器中，查看环境信息已经引入



### volume挂载使用

把configmap当作文件挂载进pod容器





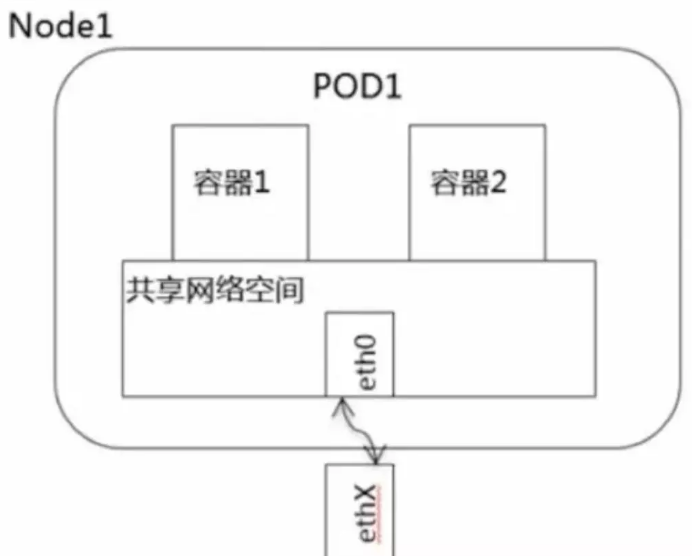
# K8S网络

K8S中包含三种网络：

* container之间的通信；
* Pod网络：Pod之间通信的网络。
* 节点网络： 各个节点主机之间的通信网络；

## Pod内容器间的通讯

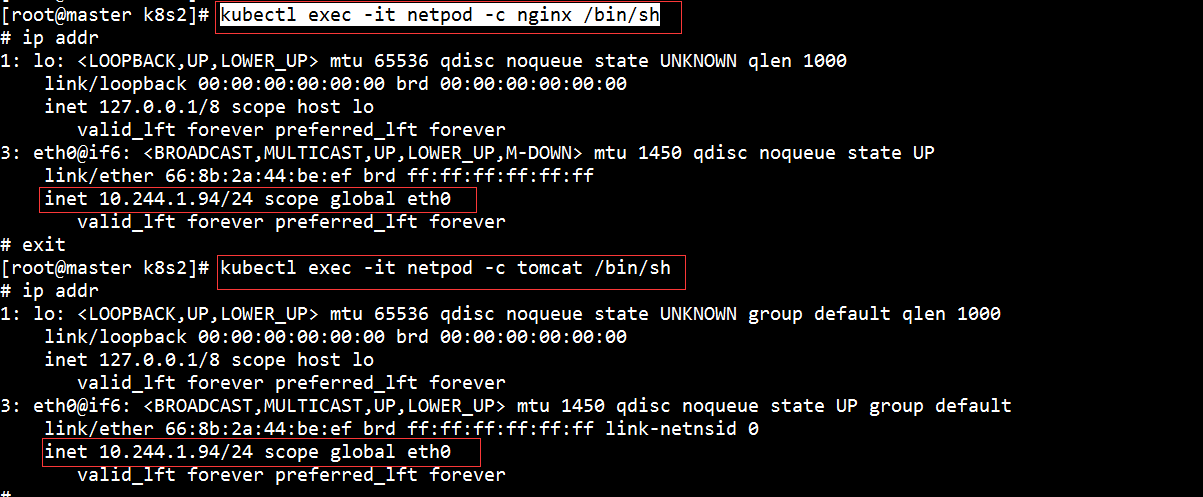
K8s的用处是容器的编排和管理，但最小组成却不是容器，是pod。物理机或者虚拟机叫node，pod是基础单元，pod里可以有多个容器，也可以只有一个容器，同一个pod的容器彼此是共享网络和主机配置的，换句话说，彼此是可以直接localhost通信的，类似于同一台机器上进行通信，所以这里面是无所谓隔离和安全一说，对外而言就是一个环境，所以pod就是这个环境的业务实体。如下图：



创建一个多容器的pod，在容器内，你可以查看到它们的ip地址是同一个。



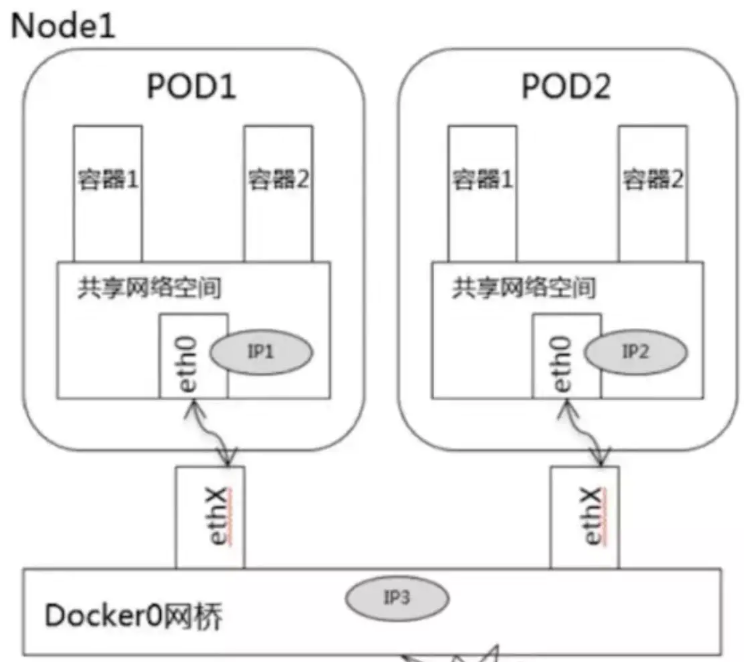
查看每个容器的ip地址：



这也告诉我们，同一个pod内共享网络，容器间的port是互斥的，你无法让pod内的两个容器使用同一个port

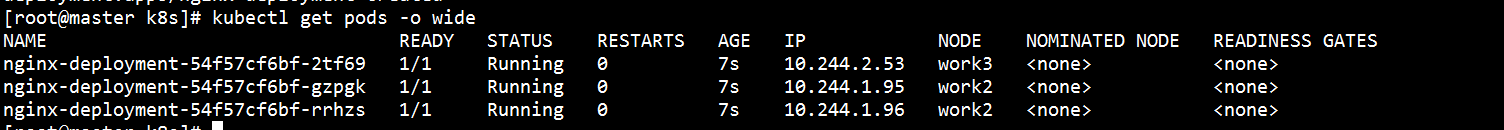
## Pod间的容器通讯

两个pod间的网络通讯，可以直接理解为两台主机的ip通讯



只要能寻址到对方pod的ip，就轻松通讯了

在K8s的集群虚拟的ip中，ip网段是根据node节点来分配的，每个node节点得一个独立的ip子网段，如下例的3个pod集群：



可以看到，节点work2上的所有pod处理同一个网段，work3的pod处于另一个网段。很明显，work2上的两台pod处于同一个子网，网络天然是通的。但是K8s还为我们多做了一步，即work2与work3上的网络，相互之间也是畅通无阻的。这里就用到了k8s的网络插件的功能，常用的就是flannel

## flannel原理

flannel组建一个大二层扁平网络，pod的ip分配由flannel统一分配，通讯过程也是走flannel的网桥。  
每个node上面都会创建一个flannel0虚拟网卡，用于跨node之间通讯。所以容器直接可以直接使用pod id进行通讯。  
 跨节点通讯时，发送端数据会从docker0路由到flannel0虚拟网卡，接收端数据会从flannel0路由到docker0。

