k8s介绍

Kubernetes是谷歌严格保密十几年的秘密武器——Borg的一个开源版本，是Docker分布式系统解决方案。 Borg是谷歌内部使用的大规模集群管理系统，基于容器技术，目的是实现资源管理的自动化，以及跨多个数据中心的资源利用率的最大化

容器编排引擎三足鼎立：

```
    Mesos
    Docker Swarm
    Kubernetes
```

早在 2015 年 5 月，Kubernetes 在 Google 上的的搜索热度就已经超过了 Mesos 和 Docker Swarm，从那儿之后更是一路飙升，将对手甩开了十几条街,容器编排引擎领域的三足鼎立时代结束。

目前，AWS、Azure、Google、阿里云、腾讯云等主流公有云提供的是基于 Kubernetes 的容器服务；Rancher、CoreOS、IBM、Mirantis、Oracle、Red Hat、VMWare 等无数厂商也在大力研发和推广基于 Kubernetes 的容器 CaaS 或 PaaS 产品。可以说，Kubernetes 是当前容器行业最炙手可热的明星。

Google 的数据中心里运行着超过 20 亿个容器，而且 Google 十年前就开始使用容器技术。 最初，Google 开发了一个叫 Borg 的系统（现在命令为 Omega）来调度如此庞大数量的容器和工作负载。在积累了这么多年的经验后，Google 决定重写这个容器管理系统，并将其贡献到开源社区，让全世界都能受益。这个项目就是 Kubernetes。简单的讲，Kubernetes 是 Google Omega 的开源版本。

k8s核心概念

```
1.Pod
  Pod直译是豆荚，可以把容器想像成豆荚里的豆子，把一个或多个关系紧密的豆子包在一起就是豆荚（一个Pod）。在k8s中我们不会直接操作容器，而是把容器包装成Pod再进行管理
  运行于Node节点上，若干相关容器的组合。Pod内包含的容器运行在同一宿主机上，使用相同的网络命名空间、IP地址和端口，能够通过localhost进行通信。Pod是Kurbernetes进行创建、调度和管理的最小单位，它提供了比容器更高层次的抽象，使得部署和管理更加灵活。一个Pod可以包含一个容器或者多个相关容器。

  • 每个Pod都有一个特殊的被称为“"容器”"Pause容器，还包含一个或多个紧密相关的用户业务容器；
  • 一个Pod里的容器与另外主机上的Pod容器能够直接通信；
  • 如果Pod所在的Node宕机，会将这个Node上的所有Pod重新调度到其他节点上；
  • 普通Pod及静态Pod，前者存放在etcd中，后者存放在具体Node上的一个具体文件中，并且只能在此Node上启动运行；
  • Docker Volume对应Kubernetes中的Pod Volume；
  • 每个Pod可以设置限额的计算机资源有CPU和Memory；
  • Requests，资源的最小申请量；
  • Limits，资源最大允许使用的量；

Event
```

是一个事件记录，记录了事件最早产生的时间、最后重复时间、重复次数、发起者、类型，以及导致此事件的原因等信息。Event通常关联到具体资源对象上，式排查故障的重要参考信息；

Pod IP

Pod的IP地址，是Docker Engine根据docker0网桥的IP地址段进行分配的，通常是一个虚拟的二层网络，位于不同Node上的Pod能够彼此通信，需要通过Pod IP所在的虚拟二层网络进行通信，而真实的TCP流量则是通过Node IP所在的物理网卡流出的；

Cluster IP

Service的IP地址。特性如下：

- 仅仅作用于Kubernetes Servcie这个对象，并由Kubernetes管理和分配IP地址；
- 无法被Ping，因为没有一个"实体网络对象"来响应；
- 只能结合Service Port组成一个具体的通信端口；
- Node IP网、Pod IP网域Cluster IP网之间的通信，采用的是Kubernetes自己设计的一种编程方式的特殊的路由规则，与IP路由有很大的不同；

Node IP

Node节点的IP地址，是Kubernetes集群中每个节点的物理网卡的IP地址，是真是存在的物理网络，所有属于这个网络的服务器之间都能通过这个网络直接通信；

2.Replication Controller

Replication Controller用来管理Pod的副本，保证集群中存在指定数量的Pod副本。集群中副本的数量大于指定数量，则会停止指定数量之外的多余容器数量，反之，则会启动少于指定数量个数的容器，保证数量不变。Replication Controller是实现弹性伸缩、动态扩容和滚动升级的核心。

部署和升级Pod，声明某种Pod的副本数量在任意时刻都符合某个预期值；

- Pod期待的副本数；
- 用于筛选目标Pod的Label Selector；
- 当Pod副本数量小于预期数量的时候，用于创建新Pod的Pod模板（template）；

Replica Set

下一代的Replication Controlle，Replication Controlle只支持基于等式的selector（env=dev或environment!=qa）但Replica Set还支持新的、基于集合的selector（version in (v1.0, v2.0)或env notin (dev, qa)），这对复杂的运维管理带来很大方便。

3.Service

Service定义了Pod的逻辑集合和访问该集合的策略，是真实服务的抽象。Service提供了一个统一的服务访问入口以及服务代理和发现机制，用户不需要了解后台Pod是如何运行。

一个service定义了访问pod的方式，就像单个固定的IP地址和与其相对应的DNS名之间的关系。

Service其实就是我们经常提起的微服务架构中的一个"微服务"，通过分析、识别并建模系统中的所有服务为微服务—Kubernetes Service，最终我们的系统由多个提供不同业务能力而又彼此独立的微服务单元所组成，服务之间通过TCP/IP进行通信，从而形成了我们强大而又灵活的弹性网络，拥有了强大的分布式能力、弹性扩展能力、容错能力；

每个Pod都提供了一个独立的Endpoint（Pod IP+ContainerPort）以被客户端访问，多个Pod副本组成了一个集群来提供服务，一般的做法是部署一个负载均衡器来访问它们，为这组Pod开启一个对外的服务端口如8000，并且将这些Pod的Endpoint列表加入8000端口的转发列表中，客户端可以通过负载均衡器的对外IP地址+服务端口来访问此服务。运行在Node上的kube-proxy其实就是一个智能的软件负载均衡器，它负责把对Service的请求转发到后端的某个Pod实例上，并且在内部实现服务的负载均衡与会话保持机制。Service不是共用一个负载均衡器的IP地址，而是每个Servcie分配一个全局唯一的虚拟IP地址，这个虚拟IP被称为Cluster IP。

4.Label

Kubernetes中的任意API对象都是通过Label进行标识，Label的实质是一系列的K/V键值对。Label是Replication Controller和Service运行的基础，二者通过Label来进行关联Node上运行的Pod。

一个label是一个被附加到资源上的键/值对，譬如附加到一个Pod上，为它传递一个用户自定的并且可识别的属性.Label还可以被应用来组织和选择子网中的资源

selector是一个通过匹配labels来定义资源之间关系得表达式，例如为一个负载均衡的service指定所目标Pod

Label可以附加到各种资源对象上，一个资源对象可以定义任意数量的Label。给某个资源定义一个Label，相当于给他打一个标签，随后可以通过Label Selector（标签选择器）查询和筛选拥有某些Label的资源对象。我们可以通过给指定的资源对象捆绑一个或多个Label来实现多维度的资源分组管理功能，以便于灵活、方便的进行资源分配、调度、配置、部署等管理工作；

5.Node

Node是Kubernetes集群架构中运行Pod的服务节点（亦叫agent或minion）。Node是Kubernetes集群操作的单元，用来承载被分配Pod的运行，是Pod运行的宿主机。

6.Endpoint（IP+Port）

标识服务进程的访问点；

k8s核心组件

Kubernetes Master：
集群控制节点，负责整个集群的管理和控制，基本上Kubernetes所有的控制命令都是发给它，它来负责具体的执行过程，我们后面所有执行的命令基本都是在Master节点上运行的；

包含如下组件：

1.Kubernetes API Server

作为Kubernetes系统的入口，其封装了核心对象的增删改查操作，以RESTful API接口方式提供给外部客户和内部组件调用。维护的REST对象持久化到Etcd中存储。

2.Kubernetes Scheduler

为新建立的Pod进行节点(node)选择(即分配机器)，负责集群的资源调度。组件抽离，可以方便替换成其他调度器。

3.Kubernetes Controller

负责执行各种控制器，目前已经提供了很多控制器来保证Kubernetes的正常运行。

- Replication Controller

管理维护Replication Controller，关联Replication Controller和Pod，保证Replication Controller定义的副本数量与实际运行Pod数量一致。

- Node Controller

管理维护Node，定期检查Node的健康状态，标识出(失效|未失效)的Node节点。

- Namespace Controller

管理维护Namespace，定期清理无效的Namespace，包括Namesapce下的API对象，比如Pod、Service等。

- Service Controller

管理维护Service，提供负载以及服务代理。

- EndPoints Controller

管理维护Endpoints，关联Service和Pod，创建Endpoints为Service的后端，当Pod发生变化时，实时更新Endpoints。

- Service Account Controller

管理维护Service Account，为每个Namespace创建默认的Service Account，同时为Service Account创建Service Account Secret。

- Persistent Volume Controller

管理维护Persistent Volume和Persistent Volume Claim，为新的Persistent Volume Claim分配Persistent Volume进行绑定，为释放的Persistent Volume执行清理回收。

- Daemon Set Controller

管理维护Daemon Set，负责创建Daemon Pod，保证指定的Node上正常的运行Daemon Pod。

- Deployment Controller

管理维护Deployment，关联Deployment和Replication Controller，保证运行指定数量的Pod。当Deployment更新时，控制实现Replication Controller和 Pod的更新。

- Job Controller

管理维护Job，为Jod创建一次性任务Pod，保证完成Job指定完成的任务数目

- Pod Autoscaler Controller

实现Pod的自动伸缩，定时获取监控数据，进行策略匹配，当满足条件时执行Pod的伸缩动作。

Kubernetes Node：
除了Master，Kubernetes集群中的其他机器被称为Node节点，Node节点才是Kubernetes集群中的工作负载节点，每个Node都会被Master分配一些工作负载（Docker容器），当某个Node宕机，其上的工作负载会被Master自动转移到其他节点上去；
包含如下组件：

1.Kubelet

负责管控容器，Kubelet会从Kubernetes API Server接收Pod的创建请求，启动和停止容器，监控容器运行状态并汇报给Kubernetes API Server。

2.Kubernetes Proxy

负责为Pod创建代理服务，Kubernetes Proxy会从Kubernetes API Server获取所有的Service信息，并根据Service的信息创建代理服务，实现Service到Pod的请求路由和转发，从而实现Kubernetes层级的虚拟转发网络。

3.Docker Engine（docker），Docker引擎，负责本机的容器创建和管理工作；

k8s集群部署

集群环境： 系统:centos7u4

本次实验使用三台机器用于部署k8s的运行环境，1台master,2台node：细节如下

节点及功能	主机名	IP
Master、etcd、registry	K8s-master	192.168.245.250
Node1	K8s-node-1	192.168.245.251
Node2	K8s-node-2	192.168.245.252

设置三台机器的主机名：

Master上执行：

```
#  hostnamectl --static set-hostname  k8s-master
```

Node1上执行：

```
# hostnamectl --static set-hostname  k8s-node-1
```

Node2上执行：

```
# hostnamectl --static set-hostname  k8s-node-2
```

修改每台机器的hosts文件互解：

```
192.168.245.250 k8s-master
192.168.245.250 etcd
192.168.245.250 registry
192.168.245.251 k8s-node-1
192.168.245.252 k8s-node-2
```

所有机器关闭防火墙和selinux：

```
# systemctl stop firewalld && systemctl disable firewalld && setenforce 0
```

所有机器安装epel-release源

```
# yum -y install epel-release
```

部署master:

1.使用yum安装etcd

etcd服务作为Kubernetes集群的主数据库,在安装Kubernetes各服务之前需要首先安装和启动。

```
# yum -y install etcd
```

2.编辑/etc/etcd/etcd.conf文件(修改加粗部分3行)

```
#[Member]
#ETCD_CORS=""
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
#ETCD_WAL_DIR=""
#ETCD_LISTEN_PEER_URLS="http://localhost:2380"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379,http://0.0.0.0:4001"
#ETCD_MAX_SNAPSHOTS="5"
```

```
#ETCD_MAX_WALS="5"
ETCD_NAME="master"
#ETCD_SNAPSHOT_COUNT="100000"
#ETCD_HEARTBEAT_INTERVAL="100"
#ETCD_ELECTION_TIMEOUT="1000"
#ETCD_QUOTA_BACKEND_BYTES="0"
#ETCD_MAX_REQUEST_BYTES="1572864"
#ETCD_GRPC_KEEPALIVE_MIN_TIME="5s"
#ETCD_GRPC_KEEPALIVE_INTERVAL="2h0m0s"
#ETCD_GRPC_KEEPALIVE_TIMEOUT="20s"
#
#[Clustering]
#ETCD_INITIAL_ADVERTISE_PEER_URLS="http://localhost:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://etcd:2379,http://etcd:4001"
```

启动并验证状态:

```
# systemctl start etcd
#  etcdctl set testdir/testkey0 0
0
#  etcdctl get testdir/testkey0
0
# etcdctl -C http://etcd:4001 cluster-health
member 8e9e05c52164694d is healthy: got healthy result from http://0.0.0.0:2379
cluster is healthy

# etcdctl -C http://etcd:2379 cluster-health
member 8e9e05c52164694d is healthy: got healthy result from http://0.0.0.0:2379
cluster is healthy
```

安装Docker

```
[root@k8s-master ~]# yum install docker -y
```

配置Docker配置文件，使其允许从registry中拉取镜像

```
[root@k8s-master ~]# vim /etc/sysconfig/docker
# /etc/sysconfig/docker
# Modify these options if you want to change the way the docker daemon runs
OPTIONS='--selinux-enabled --log-driver=journald --signature-verification=false'
if [ -z "${DOCKER_CERT_PATH}" ]; then
    DOCKER_CERT_PATH=/etc/docker
fi
OPTIONS='--insecure-registry registry:5000'
```

设置开机自启动并开启服务

```
[root@k8s-master ~]# chkconfig docker on
[root@k8s-master ~]# service docker start
```

安装kubernets

```
[root@k8s-master ~]# yum install kubernetes -y
```

配置并启动kubernetes
在kubernetes master上需要运行以下组件：
        Kubernets API Server
        Kubernets Controller Manager
        Kubernets Scheduler

相应的要更改以下几个配置中带颜色部分信息：
[root@k8s-master ~]# vim /etc/kubernetes/apiserver
###
kubernetes system config
#
The following values are used to configure the kube-apiserver
#
# The address on the local server to listen to.
KUBE_API_ADDRESS="--insecure-bind-address=0.0.0.0"

# The port on the local server to listen on.
KUBE_API_PORT="--port=8080"

Port minions listen on
KUBELET_PORT="--kubelet-port=10250"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=http://etcd:2379"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

default admission control policies
#KUBE_ADMISSION_CONTROL="--admission-
control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityContextDeny,Service
Account,ResourceQuota"
KUBE_ADMISSION_CONTROL="--admission-
control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityContextDeny,Resourc
eQuota"

# Add your own!
KUBE_API_ARGS=""

[root@k8s-master ~]# vim /etc/kubernetes/config
###
kubernetes system config
#
The following values are used to configure various aspects of all
kubernetes services, including
#
kube-apiserver.service
kube-controller-manager.service

```
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow-privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER="--master=http://k8s-master:8080"
```

启动服务并设置开机自启动

```
[root@k8s-master ~]# systemctl enable kube-apiserver.service
[root@k8s-master ~]# systemctl start kube-apiserver.service
[root@k8s-master ~]# systemctl enable kube-controller-manager.service
[root@k8s-master ~]# systemctl start kube-controller-manager.service
[root@k8s-master ~]# systemctl enable kube-scheduler.service
[root@k8s-master ~]# systemctl start kube-scheduler.service
```

部署node：

安装配置启动docker(两台node一样)

```
# yum install docker -y
```

配置Docker配置文件，使其允许从registry中拉取镜像

```
# vim /etc/sysconfig/docker
# /etc/sysconfig/docker

# Modify these options if you want to change the way the docker daemon runs
OPTIONS='--selinux-enabled --log-driver=journald --signature-verification=false'
if [ -z "${DOCKER_CERT_PATH}" ]; then
    DOCKER_CERT_PATH=/etc/docker
fi
OPTIONS='--insecure-registry registry:5000'
```

设置开机自启动并开启服务

```
[root@k8s-master ~]# chkconfig docker on
[root@k8s-master ~]# service docker start
```

安装配置启动kubernets(两台node一样)

```
# yum install kubernetes -y
```

配置并启动kubernetes

在kubernetes node上需要运行以下组件：

- Kubelet
- Kubernets Proxy

相应的要更改以下几个配置文中带颜色部分信息：

```
[root@K8s-node-1 ~]# vim /etc/kubernetes/config
###
# kubernetes system config
#
# The following values are used to configure various aspects of all
# kubernetes services, including
#
#   kube-apiserver.service
#   kube-controller-manager.service
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow-privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER="--master=http://k8s-master:8080"

[root@K8s-node-1 ~]# vim /etc/kubernetes/kubelet
###
# kubernetes kubelet (minion) config
# The address for the info server to serve on (set to 0.0.0.0 or "" for all
interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname-override=k8s-node-1"

# location of the api-server
KUBELET_API_SERVER="--api-servers=http://k8s-master:8080"

# pod infrastructure container
KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-
image=registry.access.redhat.com/rhel7/pod-infrastructure:latest"

# Add your own!
KUBELET_ARGS=""
```

启动服务并设置开机自启动

```
[root@k8s-master ~]# systemctl enable kubelet.service
[root@k8s-master ~]# systemctl start kubelet.service
[root@k8s-master ~]# systemctl enable kube-proxy.service
[root@k8s-master ~]# systemctl start kube-proxy.service
```

查看状态：
在master上查看集群中节点及节点状态

```
[root@k8s-master ~]#  kubectl -s http://k8s-master:8080 get node
NAME         STATUS    AGE
k8s-node-1   Ready     3m
k8s-node-2   Ready     16s
[root@k8s-master ~]# kubectl get nodes
NAME         STATUS    AGE
k8s-node-1   Ready     3m
k8s-node-2   Ready     43s
```

创建覆盖网络—Flannel
安装Flannel

在master、node上均执行如下命令，进行安装

```
[root@k8s-master ~]# yum install flannel -y
```

配置Flannel

master、node上均编辑/etc/sysconfig/flanneld，修改红色部分

```
[root@k8s-master ~]# vi /etc/sysconfig/flanneld
# Flanneld configuration options
# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD_ENDPOINTS="http://etcd:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_PREFIX="/atomic.io/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

配置etcd中关于flannel的key
Flannel使用Etcd进行配置，来保证多个Flannel实例之间的配置一致性，所以需要在etcd上进行如下配置：（‘/atomic.io/network/config’这个key与上文/etc/sysconfig/flannel中的配置项FLANNEL_ETCD_PREFIX是相对应的，错误的话启动就会出错）

```
[root@k8s-master ~]# etcdctl mk /atomic.io/network/config '{ "Network": "10.0.0.0/16" }'
{ "Network": "10.0.0.0/16" }
```

启动

启动Flannel之后，需要依次重启docker、kubernete。

在master执行：

```
systemctl enable flanneld.service
systemctl start flanneld.service
```

```
service docker restart
systemctl restart kube-apiserver.service
systemctl restart kube-controller-manager.service
systemctl restart kube-scheduler.service

在node上执行：
systemctl enable flanneld.service
systemctl start flanneld.service
service docker restart
systemctl restart kubelet.service
systemctl restart kube-proxy.service
```

部署dashboard

部署dashboard需要两个费劲的镜像
注：正常来讲只需要配置好yaml文件，不需要事先下载好镜像到节点当中，也就是如果yaml配置文件内所使用的镜像在node中不存在就会自动从docker源下载，但是我们现在使用的镜像是没有国内源的，所以先下载下来导入到每个node内

准备镜像：国外下载，国内导入
核心操作：从海外的服务器上pull下来对应的镜像，之后通过docker save保存成tar包，将tar包传回国内，在每个node上执行docker load将镜像导入(我在master上也倒入了)。

镜像1：在dashboard.yaml中定义了dashboard所用的镜像：
    daocloud.io/daocloud/google_containers_kubernetes-dashboard-amd64:v1.6.1
    也可以选择其他的版本

注:这个镜像我是从国内源下载的，每个节点都下载
```
# docker pull daocloud.io/daocloud/google_containers_kubernetes-dashboard-amd64:v1.6.1
```

镜像2：启动k8s的pod还需要一个额外的镜像：
    registry.access.redhat.com/rhel7/pod-infrastructure:latest（node中，/etc/kubernetes/kubelet的配置），由于一些众所周知的原因，这个镜像在国内是下载不下来的

    我先申请了香港服务器安装doker，pull镜像下来打tar包,但是香港服务器scp到我本地的时候速度每次降到0，换服务器，我又申请了美国的服务器50K的速度下了1个多小时，具体方法如下：

美国服务器：
```
    # yum install docker -y
    # systemctl start docker
    # docker search  pod-infrastructure  //从找到的结果中下载了一个其他版本的镜像,原装的不能下载
    # docker pull docker.io/tianyebj/pod-infrastructure
    # docker save -o podinfrastructure.tar  docker.io/tianyebj/pod-infrastructure
```

本地机器node1：

```
    # scp   美国服务器IP:/podinfrastructure.tar   /
   下载下来之后拷贝给所有node节点，然后导入：
    # docker load < podinfrastructure.tar

    查看导入之后的镜像
    # docker images
    REPOSITORY                                                  TAG          IMAGE ID
           CREATED             SIZE
    daocloud.io/daocloud/google_containers_kubernetes-dashboard-amd64   v1.6.1
         71dfe833ce74        10 months ago       134 MB
    docker.io/tianyebj/pod-infrastructure           latest       34d3450d733b
14 months ago   205 MB

修改配置文件（每个节点都修改）：
# vim /etc/kubernetes/kubelet   //修改下行内的镜像名称
KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=docker.io/tianyebj/pod-
infrastructure:latest"

重启服务（每个节点都重启）：
# systemctl  restart kubelet

master上编辑dashboard.yaml：
任何目录下都可以，注意或更改以下红色部分
[root@k8s-master /]# vim dashboard.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
# Keep the name in sync with image version and
# gce/coreos/kube-manifests/addons/dashboard counterparts
  name: kubernetes-dashboard-latest
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        k8s-app: kubernetes-dashboard
        version: latest
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: kubernetes-dashboard
        image: daocloud.io/daocloud/google_containers_kubernetes-dashboard-
amd64:v1.6.1

        resources:
          # keep request = limit to keep this container in guaranteed class
          limits:
            cpu: 100m
```

```
            memory: 50Mi
          requests:
            cpu: 100m
            memory: 50Mi
        ports:
        - containerPort: 9090
        args:
         -  --apiserver-host=http://192.168.245.250:8080
        livenessProbe:
          httpGet:
            path: /
            port: 9090
          initialDelaySeconds: 30
          timeoutSeconds: 30
```

master上编辑dashboardsvc.yaml文件
任何目录下都可以：

```
[root@k8s-master /]# vim dashboardsvc.yaml
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-dashboard
  namespace: kube-system
  labels:
    k8s-app: kubernetes-dashboard
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    k8s-app: kubernetes-dashboard
  ports:
  - port: 80
    targetPort: 9090
```

启动
在master执行如下命令：

```
# kubectl create -f dashboard.yaml
# kubectl create -f dashboardsvc.yaml
```

dashboard搭建完成。

验证
命令验证，master上执行如下命令：

```
[root@k8s-master /]# kubectl get deployment --all-namespaces
NAMESPACE     NAME                          DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
kube-system   kubernetes-dashboard-latest   1         1         1            1
      19s

[root@k8s-master /]# kubectl get svc  --all-namespaces
```

```
NAMESPACE     NAME                   CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
default       kubernetes             10.254.0.1      <none>        443/TCP   4d
kube-system   kubernetes-dashboard   10.254.231.52   <none>        80/TCP    32s

[root@k8s-master /]# kubectl get pod  -o wide  --all-namespaces
NAMESPACE     NAME                                          READY     STATUS
RESTARTS   AGE       IP           NODE
kube-system   kubernetes-dashboard-latest-1231782504-t79t7   1/1       Running   0
          1m        10.0.27.2    k8s-node-2
```

界面验证，浏览器访问：http://192.168.245.250:8080/ui

部署集群本地仓库

生产环境下，势必不能够每个机器都导入一遍从海外下载回来的镜像，这方法都不是可以长期使用的。可以通过搭建本地的私有镜像仓库（docker registry，这个镜像可以在国内直接下载）来解决这个问题。

注：除了第3、4步，其他部署本地仓库的方法和讲docker的时候方法是一样一样的

1、部署docker registry
在master上搭建registry。
1.1 拉取registry镜像

```
# docker pull docker.io/registry
# docker images
REPOSITORY           TAG                 IMAGE ID            CREATED
SIZE
docker.io/registry   latest              d1e32b95d8e8        4 weeks ago
33.17 MB
```

1.2 启动registry

```
# docker run -d -p 5000:5000 --name=registry --restart=always --privileged=true  --log-driver=none -v /home/data/registrydata:/tmp/registry registry
```

其中，/home/data/registrydata是一个比较大的系统分区，今后镜像仓库中的全部数据都会保存在这个外挂目录下。
里边的 /tmp/registry 在我下载的镜像中对应的是：/var/lib/registry，如果不确定是那个目录，大家最好使用 docker inspect registry查看一下Image相关的信息！

2、更改名称并推送

```
[root@K8s-node-2 ~]# docker images
REPOSITORY                                             TAG                 IMAGE ID
           CREATED             SIZE
registry.access.redhat.com/rhel7/pod-infrastructure   latest
34d3450d733b        2 weeks ago         205 MB
gcr.io/google_containers/kubernetes-dashboard-amd64   v1.5.1
1180413103fd        5 weeks ago         103.6 MB
```

```
[root@K8s-node-2 ~]# docker tag registry.access.redhat.com/rhel7/pod-
infrastructure:latest registry:5000/pod-infrastructure:latest

[root@K8s-node-2 ~]# docker tag gcr.io/google_containers/kubernetes-dashboard-
amd64:v1.5.1 registry:5000/kubernetes-dashboard-amd64:v1.5.1

[root@K8s-node-2 ~]# docker push registry:5000/pod-infrastructure:latest

[root@K8s-node-2 ~]# docker push registry:5000/kubernetes-dashboard-amd64:v1.5.1

[root@K8s-node-2 ~]# docker images
REPOSITORY                                          TAG                 IMAGE ID
           CREATED             SIZE
registry.access.redhat.com/rhel7/pod-infrastructure   latest
34d3450d733b        2 weeks ago         205 MB
registry:5000/pod-infrastructure                    latest
34d3450d733b        2 weeks ago         205 MB
gcr.io/google_containers/kubernetes-dashboard-amd64   v1.5.1
1180413103fd        5 weeks ago         103.6 MB
registry:5000/kubernetes-dashboard-amd64            v1.5.1
1180413103fd        5 weeks ago         103.6 MB
gcr.io/google_containers/kubedns-amd64              1.7
bec33bc01f03        5 months ago        55.06 MB
```

3、更改所使用的镜像名称

Dashboard是在yaml中定义的，要更改dashboard.yaml中对应的"image: gcr.io/google_containers/kubernetes-dashboard-amd64:v1.5.1"为"image: registry:5000/kubernetes-dashboard-amd64:v1.5.1"

pod-infrastructure是在node的kubelet配置文件中定义的：
要更改每个node中/etc/kubernetes/kubelet中对应的：
"KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=registry.access.redhat.com/rhel7/pod-infrastructure:latest"
为"KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image= registry:5000/pod-infrastructure:latest ""。
更改之后需要重启kubelet服务。

4、重建dashboard应用

执行完上一节的"销毁应用"之后，再次执行"启动"，即可完成dashboard的重建。

部署集群dns服务

1.说明
与1.2版本相比，1.4中的DNS增加了解析Pod（HostName）对应的域名的新功能，且在部署上也有了一些变化。1.2中，需要部署etcd（或使用master上的Etcd）、kube2sky、skydns三个组件；1.4中，DNS组件由kubedns（监控Kubernetes中service变化）、dnsmasq（DNS服务）和一个健康检查容器——healthz组成。

在搭建PetSet（宠物应用）时，系统首先要为PetSet设置一个HeadLess service，即service的ClusterIP显示的设置成none，而对每一个有特定名称的Pet（Named Pod），可以通过其HostName进行寻址访问。这就用到了1.4中的新功能。以下给出具体的搭建过程。

2、修改配置
2.1修改各个node上的kubelet
修改以下红色部分，完成后重启kubelet服务。

```
[root@k8s-node-1 /]# cat /etc/kubernetes/kubelet
###
# kubernetes kubelet (minion) config

# The address for the info server to serve on (set to 0.0.0.0 or "" for all interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname-override=k8s-node-1"

# location of the api-server
KUBELET_API_SERVER="--api-servers=http://k8s-master:8080"

# pod infrastructure container
KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=docker.io/tianyebj/pod-infrastructure:latest"

# Add your own!
KUBELET_ARGS="--cluster-dns=10.254.10.2 --cluster-domain=cluster.local. --allow-privileged=true"

[root@k8s-node-1 ~]# systemctl restart kubelet.service
```

2.2修改APIserver
修改以下红色部分：

```
[root@k8s-master /]# cat /etc/kubernetes/apiserver
###
# kubernetes system config
#
# The following values are used to configure the kube-apiserver
#

# The address on the local server to listen to.
KUBE_API_ADDRESS="--insecure-bind-address=0.0.0.0"

# The port on the local server to listen on.
# KUBE_API_PORT="--port=8080"
KUBE_API_PORT="--port=8080"
```

```
# Port minions listen on
# KUBELET_PORT="--kubelet-port=10250"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=http://etcd:2379"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

# default admission control policies
KUBE_ADMISSION_CONTROL="--admission-
control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityContextDeny,Resourc
eQuota"

# Add your own!
KUBE_API_ARGS=""
```

2.3 修改yaml文件

注意修改以下红色部分：

```
[root@k8s-master /]# cat kube-dns-rc_14.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v20
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    version: v20
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v20
  template:
    metadata:
      labels:
        k8s-app: kube-dns
        version: v20
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ''
        scheduler.alpha.kubernetes.io/tolerations: '[{"key":"CriticalAddonsOnly",
"operator":"Exists"}]'
    spec:
      containers:
      - name: kubedns
        image: docker.io/ist0ne/kubedns-amd64:latest
        imagePullPolicy: IfNotPresent
```

```
        resources:
          # TODO: Set memory limits when we've profiled the container for large
          # clusters, then set request = limit to keep this container in
          # guaranteed class. Currently, this container falls into the
          # "burstable" category so the kubelet doesn't backoff from restarting
it.
          limits:
            memory: 170Mi
          requests:
            cpu: 100m
            memory: 70Mi
        livenessProbe:
          httpGet:
            path: /healthz-kubedns
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 60
          timeoutSeconds: 5
          successThreshold: 1
          failureThreshold: 5
        readinessProbe:
          httpGet:
            path: /readiness
            port: 8081
            scheme: HTTP
          # we poll on pod startup for the Kubernetes master service and
          # only setup the /readiness HTTP server once that's available.
          initialDelaySeconds: 3
          timeoutSeconds: 5
        args:
        # command = "/kube-dns"
        - --domain=cluster.local.
        - --dns-port=10053
        - --kube-master-url=http://192.168.245.250:8080
        ports:
        - containerPort: 10053
          name: dns-local
          protocol: UDP
        - containerPort: 10053
          name: dns-tcp-local
          protocol: TCP
      - name: dnsmasq
        image: docker.io/mritd/kube-dnsmasq-amd64:latest
        imagePullPolicy: IfNotPresent
        livenessProbe:
          httpGet:
            path: /healthz-dnsmasq
            port: 8080
            scheme: HTTP
```

```
          initialDelaySeconds: 60
          timeoutSeconds: 5
          successThreshold: 1
          failureThreshold: 5
        args:
        - --cache-size=1000
        - --no-resolv
        - --server=127.0.0.1#10053
        - --log-facility=-
        ports:
        - containerPort: 53
          name: dns
          protocol: UDP
        - containerPort: 53
          name: dns-tcp
          protocol: TCP
      - name: healthz
        image: docker.io/ist0ne/exechealthz-amd64:latest
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            memory: 50Mi
          requests:
            cpu: 10m
            # Note that this container shouldn't really need 50Mi of memory. The
            # limits are set higher than expected pending investigation on
#29688.
            # The extra memory was stolen from the kubedns container to keep the
            # net memory requested by the pod constant.
            memory: 50Mi
        args:
        - --cmd=nslookup kubernetes.default.svc.cluster.local. 127.0.0.1
>/dev/null
        - --url=/healthz-dnsmasq
        - --cmd=nslookup kubernetes.default.svc.cluster.local. 127.0.0.1:10053
>/dev/null
        - --url=/healthz-kubedns
        - --port=8080
        - --quiet
        ports:
        - containerPort: 8080
          protocol: TCP
      dnsPolicy: Default  # Don't use cluster DNS.

[root@k8s-master dns14]# cat kube-dns-svc_14.yaml
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
```

```
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 10.254.10.2
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

2.4 下载下面3个镜像
docker.io/ist0ne/kubedns-amd64:latest
docker.io/mritd/kube-dnsmasq-amd64:latest
docker.io/ist0ne/exechealthz-amd64:latest
还是没有国内源，需要先在国外服务器上下载然后打包拷贝到本地，再导入

3、启动

```
[root@k8s-master dns14]# kubectl create -f kube-dns-rc_14.yaml
replicationcontroller "kube-dns-v20" created
[root@k8s-master dns14]# kubectl create -f kube-dns-svc_14.yaml
service "kube-dns" created
```

4、查看

```
[root@k8s-master /]# kubectl get pod  -o wide  --all-namespaces
NAMESPACE     NAME                                          READY     STATUS
RESTARTS   AGE       IP          NODE
kube-system   kube-dns-v20-gbd1m                            3/3       Running   0
         19m       10.0.27.3   k8s-node-2
kube-system   kubernetes-dashboard-latest-1231782504-t79t7   1/1       Running   0
         6h        10.0.27.2   k8s-node-2
```

使用：例子在使用yam創建rc->java web應用那一小節
創建web容器-->通過kubectl exec -it Podip /bin/bash進入容器-->cat /etc/resolve.conf-->發現已經指定好DNS服務器IP，接下來可以直接使用其他pod的name了，比如：

```
root@myweb-76h6w:/usr/local/tomcat# curl myweb:8080
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
        <link href="favicon.ico" rel="icon" type="image/x-icon" />
        <link href="favicon.ico" rel="shortcut icon" type="image/x-icon" />
```

```
        <link href="tomcat.css" rel="stylesheet" type="text/css" />
    </head>
```

查看集群信息

```
在master 查看 node状态:
[root@vm1 etc]# kubectl get nodes
NAME                    STATUS    AGE
192.168.245.251   Ready     17h
192.168.245.252   Ready     17h

删除无效节点:
[root@k8s-master ~]# kubectl delete node 192.168.245.252
node "192.168.245.252" deleted

[root@vm1 etc]# kubectl get node 192.168.245.251
NAME                    STATUS    AGE
192.168.245.251   Ready      17h

[root@vm1 etc]# kubectl describe node 192.168.245.251
Name:           192.168.245.251
Role:
Labels:         beta.kubernetes.io/arch=amd64
            beta.kubernetes.io/os=linux
            kubernetes.io/hostname=192.168.245.251
Taints:         <none>
CreationTimestamp:  Tue, 20 Mar 2018 05:25:09 -0400
....
....

查看集群信息:
[root@vm1 etc]# kubectl cluster-info
Kubernetes master is running at http://localhost:8080

查看各组件信息:
[root@vm1 etc]#  kubectl -s http://localhost:8080 get componentstatuses
NAME                        STATUS    MESSAGE              ERROR
controller-manager   Healthy     ok
etcd-0                      Healthy     {"health": "true"}
scheduler                 Healthy     ok

查看service的信息
[root@vm1 etc]# kubectl get service
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.254.0.1   <none>        443/TCP   22h
```

使用命令创建资源

1、启动一个简单的容器。
一旦在container中打包好应用并将其commit为image之后，就可以将其部署在k8s集群上。

一个简单的nginx服务器例子：
先决条件：你需要拥有的是一个部署完毕并可以正常运行的k8s集群。

在Master节点上使用kubectl命令来启动一个运行着nginx服务器的容器：
```
# kubectl run my-nginx --image=nginx --replicas=2 --port=80
```
注意：更换镜像
以上命令会让节点上的Docker从nginx这个image上启动一个容器监听80端口，此为一个pod。而replicas=2则表示会起两个一模一样的pod。

使用以下命令来查看创建的pod：
```
# kubectl get po
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-l8n3i   1/1       Running   0          29m
my-nginx-q7jo3   1/1       Running   0          29m
```

k8s会确保你的应用是一直运行的，当容器运行失败时，k8s会自动重启容器，当整个节点失败时，会在另外一个健康的节点启动这个容器。

2、通过端口将应用连接到Internet上。
先决条件：拥有公网IP，或者在云服务器上，如：阿里云，亚马逊云等。
以下命令将上一步骤中的nginx容器连接到公网中：
```
$ kubectl expose rc my-nginx --port=80 --type=LoadBalancer
NAME       LABELS         SELECTOR       IP(S)     PORT(S)my-nginx   run=my-nginx
  run=my-nginx             80/TCP
```
rc即Replication Controller，上一步骤中的命令其实会自动创建一个名为my-nginx的rc来确保pod的数量维持在2个。可以使用以下命令来查看rc：

```
# kubectl get rc
CONTROLLER   CONTAINER(S)  IMAGE(S)   SELECTOR      REPLICAS my-nginx     my-nginx
     nginx      run=my-nginx     2
```
expose命令将会创建一个service，将本地（某个节点上）的一个随机端口关联到容器中的80端口。
可以使用以下命令来查service：

```
# kubectl get svc my-nginx
NAME         LABELS          SELECTOR     IP(S)              PORT(S)         my-
nginx     run=my-nginx    run=nginx    10.254.110.117     80/TCP
```
type指明这个svc将会起到一个负载均衡的作用，会将流量导入两个pod中。
svc会分配一个虚拟IP用来访问容器，如上步骤中分配的IP为10.254.110.117，则可以在任意节点上通过curl 10.254.110.117得到nginx的欢迎界面。在分配虚拟IP的过程中，你可能需要等待一些时间。
在任一节点上使用netstat -tunpl命令可以看到，kube-proxy监听的端口多了一个，端口号是随机的，可以在浏览器中输入该节点的公网IP:端口访问nginx的欢迎界面。

3、删除容器。
删除rc，即删除该rc控制的所有容器。删除svc，即删除分配的虚拟IP。
```
# kubectl delete rc my-nginx
```

```
replicationcontrollers/my-nginx
# kubectl delete svc my-nginx
services/my-nginx
```

注意，如果使用delete pod ${podName}来删除是没有效果的，因为rc会马上启动另外一个pod来维持总数量为2。

提示：注意这里现在新版本创建的是deployment,不是rc，切记

同时删除多个deployment:

```
[root@k8s-master ~]# kubectl delete deployment  nginx nginx1 nginx2
deployment "nginx" deleted
deployment "nginx1" deleted
deployment "nginx2" deleted
```

使用yaml创建资源

YAML
除了某些强制性的命令，如：kubectl run或者expose等，会隐式创建rc或者svc，k8s还允许通过配置文件的方式来创建这些操作对象。
通常，使用配置文件的方式会比直接使用命令行更可取，因为这些文件可以进行版本控制，而且文件的变化和内容也可以进行审核，当使用及其复杂的配置来提供一个稳健、可靠和易维护的系统时，这些点就显得非常重要。
在声明定义配置文件的时候，所有的配置文件都存储在YAML或者JSON格式的文件中并且遵循k8s的资源配置方式。
kubectl可以创建、更新、删除和获得API操作对象，当前apiVersion、kind和name会组成一个API Path以供kubectl来调用。

YAML是专门用来写配置文件的语言，非常简洁和强大，使用比json更方便。它实质上是一种通用的数据串行化格式。

kubernetes中用来定义YAML文件创建Pod和创建Deployment。

YAML语法规则：
- 大小写敏感
- 使用缩进表示层级关系
- 缩进时不允许使用Tab键，只允许使用空格
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- ”"“ 表示注释，从这个字符一直到行尾，都会被解析器忽略

在Kubernetes中，只需要知道两种结构类型即可：
- Lists
- Maps

字典{key:value,key1:{key2:value2},key3:{key4:[1,{key5:value5},3,4,5]},,}

```
key:value
key1:
  key2:value2
```

```
key3:
  key4:
  - 1
  - key5:
    - value5
  - 3
  - 4
  - 5
spec:
    container:
```

使用YAML用于K8s的定义带来的好处：

- 便捷性：不必添加大量的参数到命令行中执行命令
- 可维护性：YAML文件可以通过源头控制，跟踪每次操作
- 灵活性：YAML可以创建比命令行更加复杂的结构

YAML Maps
Map指的是字典，即一个Key:Value 的键值对信息。
例如：

```
---
apiVersion: v1
kind: Pod
```

注：--- 为可选的分隔符 ，当需要在一个文件中定义多个结构的时候需要使用。上述内容表示有两个键apiVersion和kind，分别对应的值为v1和Pod。

Maps的value既能够对应字符串也能够对应一个Maps。
例如：

```
---
apiVersion: v1
kind: Pod
metadata:
  name: kube100-site
  labels:
    app: web
```

注：上述的YAML文件中，metadata这个KEY对应的值为一个Maps，而嵌套的labels这个KEY的值又是一个Map。实际使用中可视情况进行多层嵌套。

YAML处理器根据行缩进来知道内容之间的关联。上述例子中，使用两个空格作为缩进，但空格的数据量并不重要，只是至少要求一个空格并且所有缩进保持一致的空格数 。例如，name和labels是相同缩进级别，因此YAML处理器知道他们属于同一map；它知道app是lables的值因为app的缩进更大。
注意：在YAML文件中绝对不要使用tab键

YAML   Lists
List即列表，就是数组
例如：

```
args:
 -beijing
 -shanghai
 -shenzhen
```

```
 -guangzhou
```
可以指定任何数量的项在列表中，每个项的定义以破折号（-）开头，并且与父元素之间存在缩进。

在JSON格式中，表示如下：

```
{
  "args": ["beijing", "shanghai", "shenzhen", "guangzhou"]
}
```

当然Lists的子项也可以是Maps，Maps的子项也可以是List，例如：

```
---
apiVersion: v1
kind: Pod
metadata:
  name: kube100-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: flaskapp-demo
      image: jcdemo/flaskapp
      ports: 8080
```

如上述文件所示，定义一个containers的List对象，每个子项都由name、image、ports组成，每个ports都有一个KEY为containerPort的Map组成，转成JSON格式文件：

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
        "name": "kube100-site",
        "labels": {
            "app": "web"
        },

  },
  "spec": {
        "containers": [{
            "name": "front-end",
            "image": "nginx",
            "ports": [{
                "containerPort": "80"
            }]
        }, {
            "name": "flaskapp-demo",
            "image": "jcdemo/flaskapp",
            "ports": [{
```

```
                "containerPort": "5000"
            }]
        }]
  }
}
```

k8s的pod中运行容器，一个包含简单的Hello World容器的pod可以通过YAML文件这样来定义：

```
apiVersion: v1
kind: Pod
metadata:
    name: hello-world
spec: # 当前pod内容的声明
    restartPolicy: Never
    containers:
    - name: hello
      image: "ubuntu:14.04"
      command: ["/bin/echo","hello","world"]
```

创建的pod名为metadata.name的值：hello-world，该名称必须是唯一的。
spec的内容为该pod中，各个容器的声明：
restartPolicy：Never表示启动后运行一次就终止这个pod。
containers[0].name为容器1的名字。
containers[0].image为该启动该容器的镜像。
containers[0].command相当于Dockerfile中定义的Entrypoint，可以通过下面的方式来声明cmd的参数：

```
command: ["/bin/echo"]
args: ["hello","world"]
```

使用YAML创建Pod

```
创建Pod
---
apiVersion: v1
kind: Pod
metadata:
  name: kube100-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: flaskapp-demo
      image: jcdemo/flaskapp
      ports:
```

```
        - containerPort: 5000
```

上面定义了一个普通的Pod文件，简单分析下文件内容：
• apiVersion：此处值是v1，这个版本号需要根据安装的Kubernetes版本和资源类型进行变化，记住不是写死的。
• kind：此处创建的是Pod，根据实际情况，此处资源类型可以是Deployment、Job、Ingress、Service等。
• metadata：包含Pod的一些meta信息，比如名称、namespace、标签等信息。
• spec：包括一些container，storage，volume以及其他Kubernetes需要的参数，以及诸如是否在容器失败时重新启动容器的属性。可在特定Kubernetes API找到完整的Kubernetes Pod的属性。

典型的容器的定义：
一个简单的最小定义：一个名字（front-end）、基于nginx的镜像，以及容器将会监听的指定端口号（80）。

```
…
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
…
```

容器可选的设置属性：
除了上述的基本属性外，还能够指定复杂的属性，包括容器启动运行的命令、使用的参数、工作目录以及每次实例化是否拉取新的副本。 还可以指定更深入的信息，例如容器的退出日志的位置。
容器可选的设置属性包括：
name、image、command、args、workingDir、ports、env、resource、volumeMounts、livenessProbe、readinessProbe、livecycle、terminationMessagePath、imagePullPolicy、securityContext、stdin、stdinOnce、tty

创建Pod：
"了解"了Pod的定义后，将上面创建Pod的YAML文件保存成pod.yaml，然后使用Kubectl创建Pod：

```
# kubectl create -f pod.yaml
pod "kube100-site" created
```

验证语法：
当你不确定声明的配置文件是否书写正确时，使用以下命令要验证：

```
# kubectl create -f ./hello-world.yaml --validate
```

注：使用--validate只是会告诉你它发现的问题，仍然会按照配置文件的声明来创建资源，除非有严重的错误使创建过程无法继续，如必要的字段缺失或者字段值不合法，不在规定列表内的字段会被忽略。

查看pod状态
通过get命令来查看被创建的pod。
如果执行完创建pod的命令之后，你的速度足够快，那么使用get命令你将会看到以下的状态：

```
# kubectl get pods
```

```
NAME          READY     STATUS    RESTARTS   AGE
hello-world   0/1       Pending   0          0s

# kubectl get pods
NAME           READY     STATUS    RESTARTS   AGE
kube100-site   2/2       Running   0          1m
```

注： Pod创建过程中如果出现错误，可以使用kubectl describe 进行排查。

各字段含义：

NAME：Pod的名称

READY：Pod的准备状况，右边的数字表示Pod包含的容器总数目，左边的数字表示准备就绪的容器数目。

STATUS：Pod的状态。

RESTARTS：Pod的重启次数

AGE：Pod的运行时间。

pod的准备状况指的是Pod是否准备就绪以接收请求，Pod的准备状况取决于容器，即所有容器都准备就绪了，Pod才准备就绪。这时候kubernetes的代理服务才会添加Pod作为分发后端，而一旦Pod的准备状况变为false(至少一个容器的准备状况为false),kubernetes会将Pod从代理服务的分发后端移除，即不会分发请求给该Pod。

一个pod刚被创建的时候是不会被调度的，因为没有任何节点被选择用来运行这个pod。调度的过程发生在创建完成之后，但是这个过程一般很快，所以你通常看不到pod是处于unscheduler状态的除非创建的过程遇到了问题。

pod被调度之后，分配到指定的节点上运行，这时候，如果该节点没有所需要的image，那么将会自动从默认的Docker Hub上pull指定的image，一切就绪之后，你就可以看到pod是处于running状态了：

```
[root@k8s-master ~]# kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
my-nginx-379829228-2zjv3      1/1       Running   0          1h
my-nginx-379829228-mm8f8      1/1       Running   0          1h
```

查看pods所在的运行节点：

```
# kubectl get pods -o wide
```

查看pods定义的详细信息：

```
# kubectl get pods -o yaml
# kubectl get pod nginx-8v3cg --output yaml
```

kubectl get支持以Go Template方式过滤指定的信息，比如查询Pod的运行状态

```
# kubectl get pods busybox --output=go-template --template={{.status.phase}}
Running
```

查看pod输出：

你可能会有想了解在pod中执行命令的输出是什么，和Docker logs命令一样，kubectl logs将会显示这些输出：

```
# kubectl logs hello-world
hello world
```

查看kubectl describe 支持查询Pod的状态和生命周期事件：

```
[root@k8s-master ~]# kubectl describe   pod  busybox
Name:       busybox
Namespace:  default
Node:       k8s-node-1/116.196.105.133
Start Time: Thu, 22 Mar 2018 09:51:35 +0800
Labels:     name=busybox
        role=master
Status:     Pending
IP:
Controllers:    <none>
Containers:
  busybox:
    Container ID:
    Image:      docker.io/busybox
    Image ID:
    Port:
    Command:
      sleep
      360000
    State:          Waiting
      Reason:           ContainerCreating
    Ready:          False
    Restart Count:      0
    Volume Mounts:      <none>
    Environment Variables:  <none>
Conditions:
  Type      Status
  Initialized   True
  Ready     False
  PodScheduled  True
No volumes.
QoS Class:  BestEffort
Tolerations:    <none>
Events:
  FirstSeen LastSeen    Count   From            SubObjectPath   Type        Reason
     Message
  --------- --------    -----   ----            -------------   --------    ------
     -------
  7m        7m      1   {default-scheduler }            Normal      Scheduled
Successfully assigne
d busybox to k8s-node-1  7m     1m      6   {kubelet k8s-node-1}
Warning     FailedSync  Error syncing pod, s
kipping: failed to "StartContainer" for "POD" with ErrImagePull: "image pull
failed for registry.access.redhat.com/rhel7/pod-infrastructure:latest, this may be
because there are no credentials on this request.  details: (open
/etc/docker/certs.d/registry.access.redhat.com/redhat-ca.crt: no such file or
directory)"
```

```
  6m    13s 27  {kubelet k8s-node-1}        Warning FailedSync  Error syncing pod,
skipping: failed to "StartContain
er" for "POD" with ImagePullBackOff: "Back-off pulling image
\"registry.access.redhat.com/rhel7/pod-infrastructure:latest\""
```

各字段含义：

Name: Pod的名称

Namespace: Pod的Namespace。

Image(s): Pod使用的镜像

Node: Pod所在的Node。

Start Time: Pod的起始时间

Labels: Pod的Label。

Status: Pod的状态。

Reason: Pod处于当前状态的原因。

Message: Pod处于当前状态的信息。

IP: Pod的PodIP

Replication Controllers: Pod对应的Replication Controller。

Containers:Pod中容器的信息

Container ID: 容器的ID。

Image: 容器的镜像

Image ID:镜像的ID。

State: 容器的状态。

Ready: 容器的准备状况(true表示准备就绪)。

Restart Count: 容器的重启次数统计。

Environment Variables: 容器的环境变量。

Conditions: Pod的条件，包含Pod准备状况(true表示准备就绪)

Volumes: Pod的数据卷。

Events: 与Pod相关的事件列表。

进入Pod对应的容器内部

```
[root@k8s-master /]# kubectl exec -it myweb-76h6w /bin/bash
```

删除pod：

```
# kubectl delete pod pod名   //单个删除
# kubectl delete pod --all   //批量删除
```

例：

```
[root@k8s-master /]# kubectl  delete pod hello-world
pod "hello-world" deleted
```

重新启动基于yaml文件的应用

```
# kubectl delete -f XXX.yaml
# kubectl create -f XXX.yaml
```

给容器定义环境变量

当你建立了一个Pod,你可以给你运行在Pod中的容器使用env设置环境变量。

在本例中，建了一个运行了一个container的Pod。这个配置文件给这个Pod定义了一个名为DEMO_GREETING值为"Hello from the environment"的环境变量。

下面是这个Pod的配置文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envars
spec:
  containers:
  - name: envar-demo-container
    image: gcr.io/google-samples/node-hello:1.0
    env:
    - name: DEMO_GREETING
      value: "Hello from the environment"
```

1.新建一个Pod基于YAML配置文件：

```
kubectl create -f http://k8s.io/docs/tasks/configure-pod-container/envars.yaml
```

2.运行Pod的列表：

```
kubectl get pods -l purpose=demonstrate-envars
```

3.获取一个shell到Pod运行的容器里：

```
kubectl exec -it envar-demo -- /bin/bash
```

4.在shell里，运行printenv命令列出环境变量

```
root@envar-demo:/# printenv
```

输出类似于下面：

```
NODE_VERSION=4.4.2
EXAMPLE_SERVICE_PORT_8080_TCP_ADDR=10.3.245.237
HOSTNAME=envar-demo
...
DEMO_GREETING=Hello from the environment
```

5.退出shell,输入exit。

查看运行的pod的环境变量

```
# kubectl exec pod名 env
```

使用yaml创建rc

1.使用yaml创建并启动replicas集合
k8s通过Replication Controller来创建和管理各个不同的重复容器集合（实际上是重复的pods）。

Replication Controller会确保pod的数量在运行的时候会一直保持在一个特殊的数字，即replicas的设置。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

和定义一个pod的YAML文件相比，不同的只是kind的值为ReplicationController，replicas的值需要指定，pod的相关定义在template中，pod的名字不需要显式地指定，因为它们会在rc中创建并赋予名字

创建rc：
rc可以通过create命令像创建pod一样来创建：

```
# kubectl create -f ./nginx-rc.yaml
replicationcontrollers/my-nginx
```

和直接创建pod不一样，rc将会替换因为任何原因而被删除或者停止运行的Pod，比如说pod依赖的节点挂了。所以我们推荐使用rc来创建和管理复杂应用，即使你的应用只要使用到一个pod，在配置文件中忽略replicas字段的设置即可

2、查看Replication Controller的状态
可以通过get命令来查看你创建的rc：

```
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS
my-nginx     nginx          nginx      app=nginx   2
```

这个状态表示，你创建的rc将会确保你一直有两个nginx的副本。

也可以和直接创建Pod一样查看创建的Pod状态信息：

```
# kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-065jq   1/1       Running   0          51s
my-nginx-buaiq   1/1       Running   0          51s
```

3、删除Replication Controller
当你想停止你的应用，删除你的rc，可以使用：

```
# kubectl delete rc my-nginx
replicationcontrollers/my-nginx
```

默认的，这将会删除所有被这个rc管理的pod，如果pod的数量很大，将会花一些时间来完成整个删除动作，如果你想使这些pod停止运行，请指定--cascade=false。
如果你在删除rc之前尝试删除pod，rc将会立即启动新的pod来替换被删除的pod，就像它承诺要做的一样。

4、Labels
k8s使用用户自定义的key-value键值对来区分和标识资源集合（就像rc、pod等资源），这种键值对称为label。
在上面的例子中，定义pod的template字段包含了一个简单定义的label，key的值为app，value的值为nginx。所有被创建的pod都会卸载这个label，可以通过-L参数来查看：

```
# kubectl get rc my-nginx -L app
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   APP
my-nginx11     nginx          nginx      app=nginx111   2          nginx
```

默认情况下，pod的label会复制到rc的label，同样地，k8s中的所有资源都支持携带label。

更重要的是，pod的label会被用来创建一个selector，用来匹配过滤携带这些label的pods。
你可以通过kubectl get请求这样一个字段来查看template的格式化输出：

```
# kubectl get rc my-nginx -o template --template="{{.spec.selector}}"
map[app:nginx111]
```

5.使用labels定位pods

```
[root@k8s-master ~]# kubectl get pods -l app=nginx11 -o wide
NAME               READY     STATUS    RESTARTS   AGE       IP           NODE
my-nginx11-1r2p4   1/1       Running   0          11m       10.0.6.3     k8s-node-
1
my-nginx11-pc4ds   1/1       Running   0          11m       10.0.33.6    k8s-node-
2
```

检查你的Pod的IPs：

```
# kubectl get pods -l app=nginx -o json | grep podIP
                "podIP": "10.245.0.15",
                "podIP": "10.245.0.14",
```

6.k8s中连接容器的模型
现在，你已经有了组合的、多份副本的应用，你可以将它连接到一个网络上。在讨论k8s联网的方式之前，有必要和Docker中连接网络的普通方式进行一下比较。

默认情况下，Docker使用主机私有网络，所以容器之间可以互相交互，只要它们在同一台机器上。
为了让Docker容器可以进行跨节点的交流，必须在主机的IP地址上为容器分配端口号，之后通过主机IP和端口将信息转发到容器中。

这样一来，很明显地，容器之间必须谨慎地使用和协调端口号的分配，或者动态分配端口号。

在众多开发者之间协调端口号的分配是十分困难的，会将集群级别之外的复杂问题暴露给用户来处理。

在k8s中，假设Pod之间可以互相交流，无论它们是在哪个宿主机上。
我们赋予每个Pod自己的集群私有IP，如此一来你就不需要明确地在Pod之间创建连接，或者将容器的端口映射到主机的端口中。
这意味着，Pod中的容器可以在主机上使用任意彼此的端口，而且集群中的Pods可以在不使用NAT的方式下连接到其他Pod。

这时，你应该能够通过使用curl来连接任一IP（如果节点之间没有处于同一个子网段，无法使用私有IP进行连接的话，就只能在对应节点上使用对应的IP进行连接测试）。
注意，容器并不是真的在节点上使用80端口，也没有任何的NAT规则来路由流量到Pod中。
这意味着你可以在同样的节点上使用同样的containerPort来运行多个nginx Pod，并且可以在集群上的任何Pod或者节点通过这个IP来连接它们。
和Docker一样，端口仍然可以发布到宿主机的接口上，但是因为这个网络连接模型，这个需求就变得很少了。

7.创建Service:cluster ip
现在，在集群上我们有了一个运行着niginx并且有分配IP地址空间的的Pod。
理论上，你可以直接和这些Pod进行交互，但是当节点挂掉之后会发生什么?
这些Pod会跟着节点挂掉，然后RC会在另外一个健康的节点上重新创建新的Pod来代替，而这些Pod分配的IP地址都会发生变化，对于Service类型的服务来说这是一个难题。

k8s上的Service是抽象的，其定义了一组运行在集群之上的Pod的逻辑集合，这些Pod是重复的，复制出来的，所以提供相同的功能。
当Service被创建，会被分配一个唯一的IP地址（也称为集群IP）。这个地址和Service的生命周期相关联，并且当Service是运行的时候，这个IP不会发生改变。
Pods进行配置来和这个Service进行交互，之后Service将会自动做负载均衡到Service中的Pod。

你可以通过以下的YAML文件来为你的两个nginx容器副本创建一个Service:

```
$ cat nginxsvc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginxsvc
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

这个YAML定义创建创建一个Service，带有Label为app=nginx的Pod都将会开放80端口，并将其关联到一个抽象的Service端口。

（targetPort字段是指容器内开放的端口Service通过这个端口连接Pod，port字段是指抽象Service的端口，nodePort为节点上暴露的端口号，不指定的话为随机。）

现在查看你创建的Service：

```
[root@k8s-master ~]# kubectl get svc
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.254.0.1      <none>        443/TCP   6d
nginxsvc     10.254.15.142   <none>        80/TCP    17m
```

和之前提到的一样，Service是以一组Pod为基础的。
这些Pod通过endpoints字段开放出来。
Service的selector将会不断地进行Pod的Label匹配，结果将会通知一个Endpoints Object，这里创建的也叫做nginxsvc。
当Pod挂掉之后，将会自动从Endpoints中移除，当有新的Pod被Service的selector匹配到之后将会自动加入这个Endpoints。
你可以查看这个Endpoint，注意，这些IP和第一步中创建Pods的时候是一样的：

```
[root@k8s-master ~]# kubectl describe svc nginxsvc
Name:           nginxsvc
Namespace:      default
Labels:         app=nginx11
Selector:       app=nginx111
Type:           ClusterIP
IP:         10.254.15.142
Port:           <unset> 80/TCP
Endpoints:      10.0.33.6:80,10.0.6.3:80
Session Affinity:   None
No events.

[root@k8s-master ~]# kubectl get ep
NAME         ENDPOINTS                   AGE
kubernetes   192.168.245.250:6443        6d
nginxsvc     10.0.33.6:80,10.0.6.3:80    19m
```

你现在应该可以通过10.254.15.142:80这个IP从集群上的任何一个节点（node）使用curl命令来连接到nginx的Service。

删除service:

```
#kubectl delete service service名
```

滚动升级

动态修改副本数量

```
# kubectl scale rc XXX --replicas=3

autoscale:
```

scale虽然能够很方便的对副本数进行扩展或缩小，但是仍然需要人工介入，不能实时自动的根据系统负载对副本数进行扩、缩。autoscale命令提供了自动根据pod负载对其副本进行扩缩的功能。

autoscale命令会给一个rc指定一个副本数的范围，在实际运行中根据pod中运行的程序的负载自动在指定的范围内对pod进行扩容或缩容。如前面创建的nginx，可以用如下命令指定副本范围在1~4

```
[root@k8s-master /]# kubectl autoscale rc myweb --min=1 --max=4
replicationcontroller "myweb" autoscaled
```

查看RC的详细信息

```
# kubectl describe rc 标签名或者选择器名
```

Pod在创建以后希望更新Pod，可以在修改Pod的定义文件后执行：

```
#kubectl replace /path/to/busybox.yaml
```

因为Pod的很多属性是没办法修改的，比如容器镜像，这时候可以用kubectl replace命令设置--force参数，等效于重新建Pod。

通过RC修改Pod副本数量

```
# kubectl replace -f rc.yaml
或
#kubect edit replicationcontroller replicationcontroller名
```

对RC使用滚动升级，来发布新功能或修复BUG

```
#kubectl rolling-update replicationcontroller名 --image=镜像名
```

滚动升级

```
#kubectl rolling-update replicationcontroller名 -f XXX.yaml
```

如果在升级过程中，发现有问题还可以中途停止update，并回滚到前面版本

```
#kubectl rolling-update rc-nginx-2 --rollback
```

使用yaml创建Deployment

Deployment是新一代用于Pod管理的对象，与Replication Controller相比，它提供了更加完善的功能，使用起来更加简单方便。

如果Pod出现故障，对应的服务也会挂掉，所以Kubernetes提供了一个Deployment的概念 ，目的是让Kubernetes去管理一组Pod的副本，也就是副本集 ，这样就能够保证一定数量的副本一直可用，不会因为某一个Pod挂掉导致整个服务挂掉。

一个完整的Deployment的YAML文件：

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kube100-site
spec:
  replicas: 2
  template:
```

```
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: front-end
          image: nginx
          ports:
            - containerPort: 80
        - name: flaskapp-demo
          image: jcdemo/flaskapp
          ports:
            - containerPort: 5000

apiVersion:
    注意这里apiVersion对应的值是extensions/v1beta1

kind的类型:
    指定为Deployment。

metadata指定一些meta信息，包括名字或标签之类的。
spec 选项定义需要两个副本，此处可以设置很多属性，主要是受此Deployment影响的Pod的选择器
spec 选项的template其实就是对Pod对象的定义
可以在Kubernetes v1beta1 API 参考中找到完整的Deployment可指定的参数列表

创建Deployment:
将上述的YAML文件保存为deployment.yaml，然后创建Deployment:
# kubectl create -f deployment.yaml
deployment "kube100-site" created

检查Deployment的列表:
# kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube100-site   2         2         2            2           2m

删除Deployment:
[root@k8s-master ~]# kubectl delete deployments my-nginx
deployment "my-nginx" deleted
```

取消自动拉取镜像

```
镜像拉取策略
yaml配置文件参数
    参数选项：imagePullPolicy: Always  ，镜像的拉取策略，总是拉取
    官方说明：https://kubernetes.io/docs/concepts/containers/images/
```

```
    By default, the kubelet will try to pull each image from the  specified
registry. However, if the imagePullPolicy property of the  container is set to
IfNotPresent or Never, then a local image is used  (preferentially or exclusively,
respectively).

    #默认情况是会根据配置文件中的镜像地址去拉取镜像，如果设置为IfNotPresent 和Never就会
使用本地镜像。
    IfNotPresent ：如果本地存在镜像就优先使用本地镜像。
    Never：直接不再去拉取镜像了，使用本地的；如果本地不存在就报异常了。

参数的作用范围：
spec:
  containers:
    - name: nginx
      image: image: reg.docker.lc/share/nginx:latest
      imagePullPolicy: IfNotPresent   #或者使用Never
参数默认为：imagePullPolicy: Always ，如果你yaml配置文件中没有定义那就是使用默认的。
```

k8s的卷操作

```
[root@k8s-master /]# cat test-hostpath.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: test-hostpath
  name: test-hostpath
spec:
  containers:
    - name: test-hostpath
      image: daocloud.io/library/nginx
      volumeMounts:
       - name: testpath  #注意这里的名字和volumes里面定义的卷的名字要一致
         mountPath: /testpath  #事先不用存在
  volumes:
  - name: testpath
    hostPath:
     path: /testpath   #事先不用存在

创建pod后进入pod测试：
[root@k8s-master /]# kubectl exec -it test-hostpath /bin/bash
root@test-hostpath:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  testpath  usr
boot  etc  lib   media  opt  root  sbin  sys  tmp   var


NFS（网络数据卷）
```

```
NFS类型的volume。允许一块现有的网络硬盘在同一个pod内的容器间共享
volumes:
- name: nfs-storage
  nfs:
   server: 192.168.20.47
   path: "/data/disk1"
```

简化 Kubernetes Yaml 文件创建

Kubernetes 提供了丰富的 kubectl 命令，可以较为方便地处理常见任务。如果需要自动化处理复杂的Kubernetes任务，常常需要编写Yaml配置文件。由于Yaml文件格式比较复杂，即使是老司机有时也不免会犯错或需要查询文档，也有人开玩笑这是使用 Yaml 编程。

方式1：模拟命令执行
kubectl中很多命令支持 --dry-run 和 -o yaml 参数，可以方便地模拟命令执行，并输出yaml格式的命令请求，这样我们就可以将执行结果 Copy & Paste到自己的编辑器中，修改完成自己的配置文件。

```
# kubectl run myapp --image=nginx --dry-run -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: myapp
  name: myapp
spec:
  replicas: 1
  selector:
    matchLabels:
      run: myapp
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: myapp
    spec:
      containers:
      - image: nginx
        name: myapp
        resources: {}
status: {}

# kubectl create secret generic mysecret --from-literal=quiet-phrase="Shh! Dont'
tell" -o yaml --dry-run
```

```
apiVersion: v1
data:
  quiet-phrase: U2hoISBEb250JyB0ZWxs
kind: Secret
metadata:
  creationTimestamp: null
  name: mysecret
```

方式2：导出资源描述

```
#kubectl get <resource-type> <resource> --export -o yaml
```

上面命令会以Yaml格式导出系统中已有资源描述

比如，可以将系统中 nginx 部署的描述导成 Yaml 文件：

先查看：

```
[root@k8s-master ~]# kubectl get deployment
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
my-nginx    2         2         2            1           42m
my-nginx1   2         2         2            2           35m
```

再导出：

```
# kubectl get deployment my-nginx1 --export -o yaml > nginx.yaml
```

k8s的pod调度

这里介绍两种调度方式

方法1、使用NodeName进行pod调度：

Pod.spec.nodeName用于强制约束将Pod调度到指定的Node节点上，这里说是"调度"，但其实指定了nodeName的Pod会直接跳过Scheduler的调度逻辑，直接写入PodList列表，该匹配规则是强制匹配。

```
[root@k8s-master ~]# cat nginxrc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeName: k8s-node-1  #指定调度节点为k8s-node-1
      containers:
      - name: nginx
        image: daocloud.io/library/nginx:latest
        ports:
```

```
        - containerPort: 80
```

创建rc:

略

查看pod运行于哪个节点:

```
[root@k8s-master ~]# kubectl get pods -o wide
NAME             READY     STATUS    RESTARTS   AGE       IP          NODE
my-nginx-dwp13   1/1       Running   0          21m       10.0.6.2   k8s-node-1
```

方法2:使用NodeSelector进行pod调度

Pod.spec.nodeSelector是通过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。启用节点选择器的步骤为:

1.Node添加label标记

标记规则:kubectl label nodes <node-name> <label-key>=<label-value>

```
#kubectl label nodes k8s.node1 cloudnil.com/role=dev
```

确认标记

```
root@k8s.master1:~# kubectl get nodes k8s.node1 --show-labels
NAME        STATUS    AGE       LABELS
k8s.node1   Ready     29d
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,cloudnil.com/role=dev,kubernetes.io/hostname=k8s.node1
```

2.Pod定义中添加nodeSelector

```
[root@k8s-master ~]# cat nginxrc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        cloudnil.com/role: dev#指定调度节点为带有label标记为:cloudnil.com/role=dev的node节点
      containers:
      - name: nginx
        image: daocloud.io/library/nginx:latest
        ports:
        - containerPort: 80
```

创建rc:

略

查看pod运行于哪个节点：

```
[root@k8s-master ~]# kubectl get pods -o wide
NAME             READY     STATUS    RESTARTS   AGE       IP          NODE
my-nginx-zttrj   1/1       Running   0          5s        10.0.6.2   k8s-node-1
```

下面这句话忽略先：
关键在于spec.selector与spec.template.metadata.labels，这两个字段必须相同，否则下一步创建RC会失败。（也可以不写spec.selector，这样默认与spec.template.metadata.labels相同）

暴露IP给外网

外部网络（即非K8S集群内的网络）访问cluster IP的办法

1. 修改master的/etc/kubernetes/proxy
把KUBE_PROXY_ARGS=""改为KUBE_PROXY_ARGS="--proxy-mode=userspace"

2.重启kube-proxy服务

3.在核心路由设备或者源主机上添加一条路由，访问cluster IP段的路由指向到master上。
我们实验的时候是在客戶端上添加如下路由条目：

```
[root@vm20 yum.repos.d]# route add -net 10.254.244.0/24 gw 192.168.245.250
```

注：10.254.244.0/24是创建service之后的cluster ip