kubernetes

https://my.oschina.net/tantexian/blog/785963 漫画

**[root@localhost ~]# yum install epel-*
[root@localhost ~]# yum install cherrytree**

k8s介绍

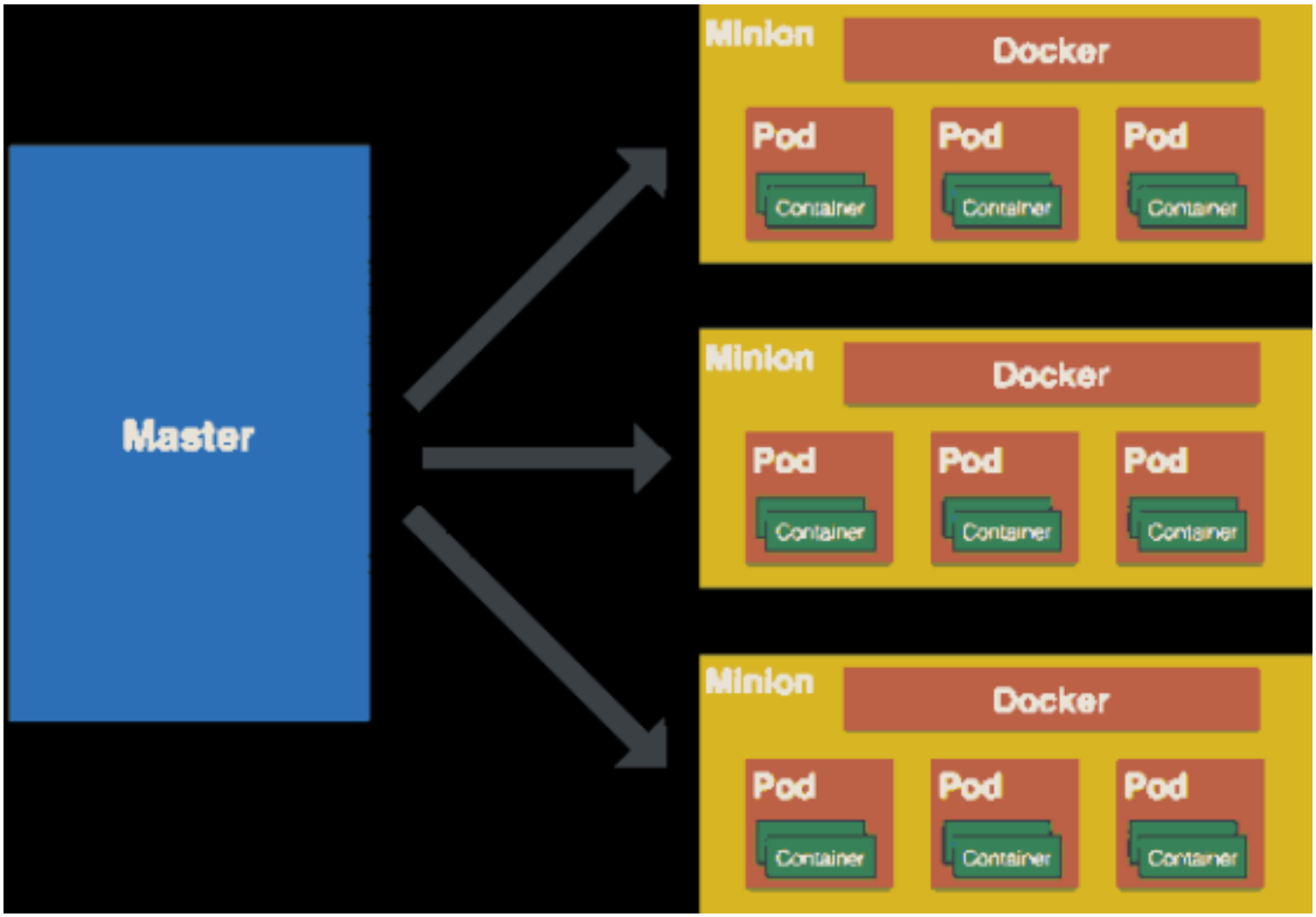
Kubernetes原理

Kubernetes是一个基于Docker容器的开源编制系统，它能在跨多个主机上管理Docker应用，并提供应用程序部署 维护和扩展的基本机制。

它透明地为用户提供原生态系统，如“需要5个 WildFly服务器和1个 MySQL服务器运行". Kubernetes具有自我修复机制，如重新启动 重新启动定时计划 复制容器以确保恢复状态，用户只需要定义状态，那么 Kubernetes就会确保状态总是在集群中。

Docker定义了运行代码时的容器，有命令用来启动 停止 重启 链接容器，Kubernetes使用Docker打包以及实例化应用程序。

一个典型的应用程序必须跨多个主机。 例如,您的web层(Apache)可能运行在一个容器。 同样地,应用程序层将会运行在另外一组不同的容器中。 web层需要将请求委托给应用程序层。 当然，在某些情况下，你可能将web服务器和应用服务器打包在一起放在相同的容器。 但是数据库层通常运行在一个单独层中。 这些容器之间需要相互交互。 使用上面的任何解决方案都需要编制脚本启动容器，以及监控容器，因防止出现问题。 而Kubernetes在应用程序状态被定义后将为用户实现所有这些工作。



主要组件说明

Kubernetes 集群中主要存在两种类型的节点，分别是 master 节点，以及 minion 节点。
Minion 节点是实际运行 Docker 容器的节点，负责和节点上运行的 Docker 进行交互，并且提供了代理功能。
Master 节点负责对外提供一系列管理集群的 API 接口，并且通过和 Minion 节点交互来实现对集群的操作管理。

组件说明

apiserver：用户和 kubernetes 集群交互的入口，封装了核心对象的增删改查操作，提供了 RESTFul 风格的 API 接口，通过 etcd 来实现持久化并维护对象的一致性。
scheduler：负责集群资源的调度和管理，例如当有 pod 异常退出需要重新分配机器时，scheduler 通过一定的调度算法从而找到最合适的节点。
controller-manager：主要是用于保证 replicationController 定义的复制数量和实际运行的 pod 数量一致，另外还保证了从 service 到 pod 的映射关系总是最新的。
kubelet：运行在 minion 节点，负责和节点上的 Docker 交互，例如启停容器，监控运行状态等。
proxy：运行在 minion 节点，负责为 pod 提供代理功能，会定期从 etcd 获取 service 信息，并根据 service 信息通过修改 iptables 来实现流量转发（最初的版本是直接通过程序提供转发功能，效率较低。），将流量转发到要访问的 pod 所在的节点上去。

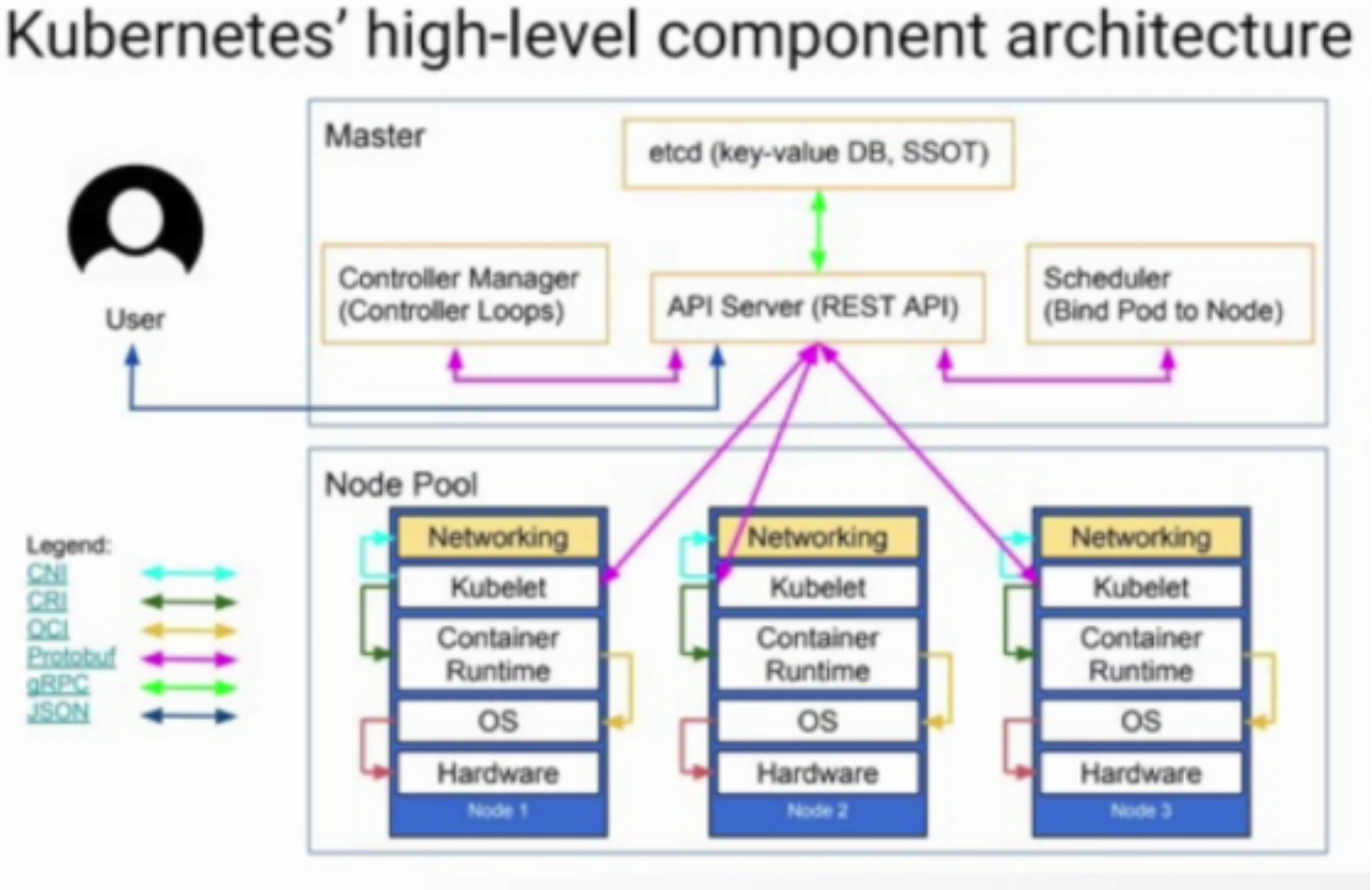
etcd：key-value键值存储数据库，用来存储kubernetes的信息的。
flannel：Flannel 是 CoreOS 团队针对 Kubernetes 设计的一个覆盖网络（Overlay Network）工具，需要另外下载部署。我们知道当我们启动 Docker 后会有一个用于和容器进行交互的 IP 地址，如果不去管理的话可能这个 IP 地址在各个机器上是一样的，并且仅限于在本机上进行通信，无法访问到其他机器上的 Docker 容器。Flannel 的目的就是为集群中的所有节点重新规划 IP 地址的使用规则，从而使得不同节点上的容器能够获得同属一个内网且不重复的 IP 地址，并让属于不同节点上的容器能够直接通过内网 IP 通信。

RC:Replication Controller

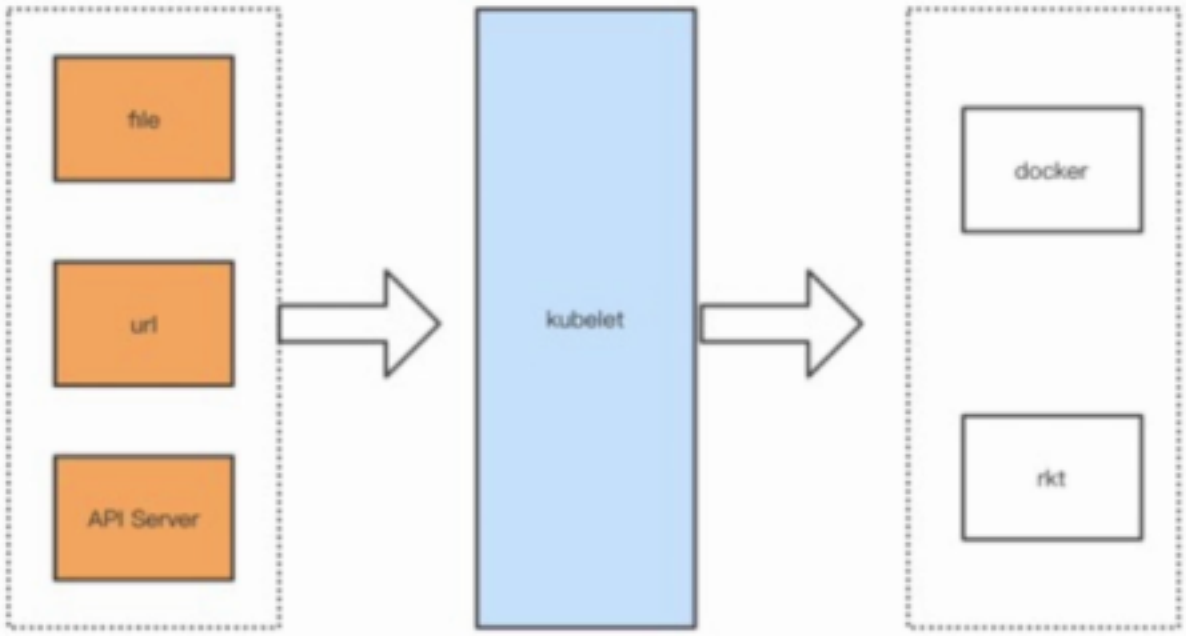
确保pod数量：RC用来管理正常运行Pod数量，一个RC可以由一个或多个Pod组成，在RC被创建后，系统会根据定义好的副本数来创建Pod数量。在运行过程中，如果Pod数量小于定义的，就会重启停止的或重新分配Pod，反之则杀死多余的。

service :service 可以为一组相同功能的pod应用提供统一的入口地址，并将请求负载均衡分发到各个容器应用上。service的负载均衡功能由node节点上的kube-proxy提供

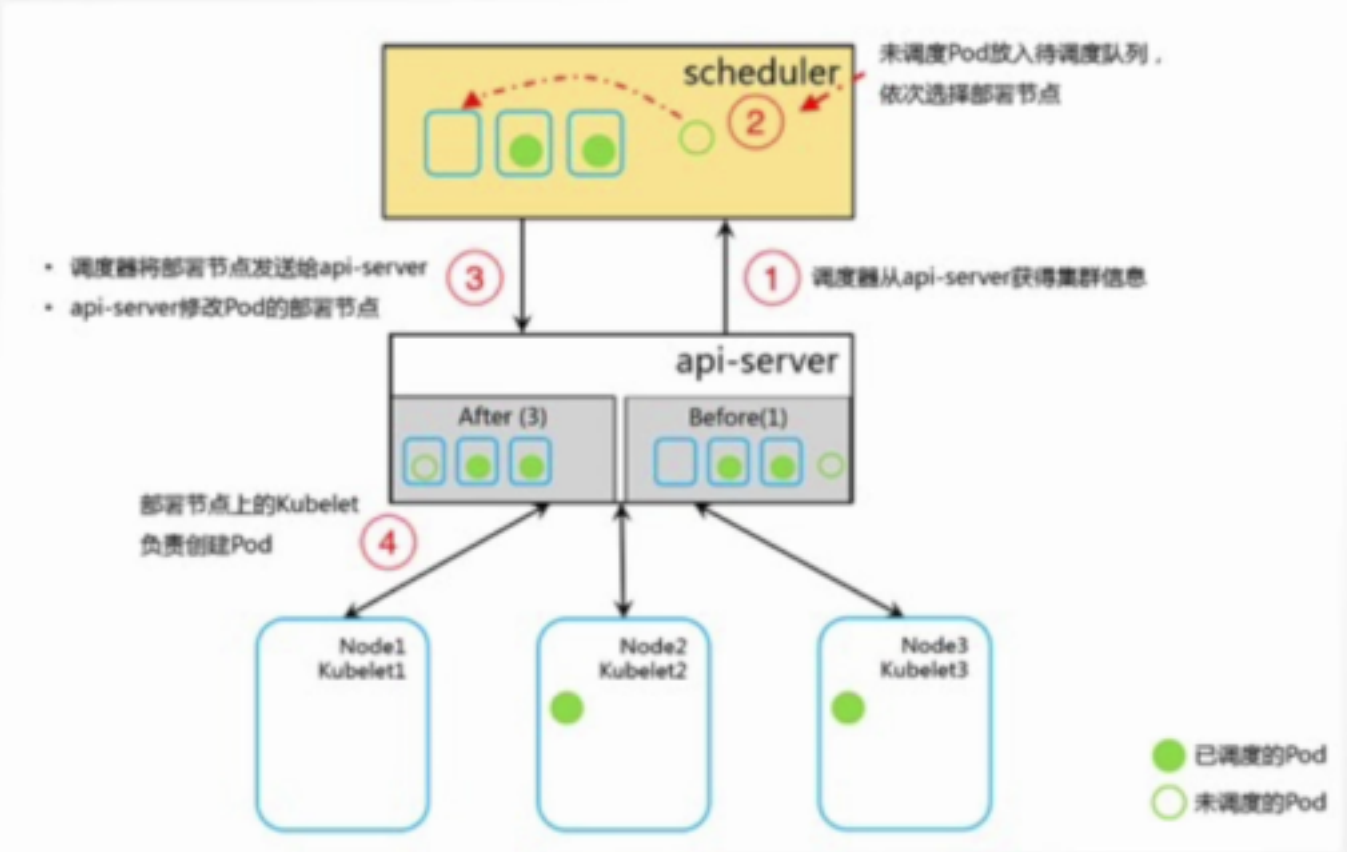
二、K8S 组件及其作用


Kubernetes' high-level component architecture
Master
etcd (key-value DB, SSOT)
Controller Manager
(Controller Loops)
API Server (REST API)
Scheduler
(Bind Pod to Node)
User
Node Pool
Networking
Kubelet
Container
Runtime
OS
Hardware
Node 1
Node 2
Node 3
Legend:
CNI
CRI
OCI
Protobuf
gRPC
JSON

2.1 kubelet : 节点上的管家



2.2 scheduler 给你的 pod 找个家



master节点(v1.10.0)

一.环境准备

主机准备
192.168.0.200 master.com
192.168.0.201 node1.com
192.168.0.202 node2.com

1.关闭防火墙
systemctl stop firewalld
systemctl disable firewalld

2.关闭selinux
setenforce 0
cat /etc/selinux/config
SELINUX=disabled

3.创建/etc/sysctl.d/k8s.conf,添加如下内容
cat > /etc/sysctl.d/k8s.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF

4.执行如下命令,是修改生效
modprobe br_netfilter
sysctl -p /etc/sysctl.d/k8s.conf

5.安装系统工具
yum install -y yum-utils device-mapper-persistent-data lvm2

6.添加软件源信息
yum-config-manager --add-repo http://*mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo*

7.查看docker软件版本,安装指定版本
yum list docker-ce.x86_64 --showduplicates | sort -r
yum -y install docker-ce-[VERSION]
比如:
yum -y install docker-ce-17.12.1.ce-1.el7.centos

8.安装完毕,启动docker
systemctl start docker
systemctl enable docker

二.安装k8s

1.拉取镜像,master节点,执行(建议写成脚本执行)
docker pull cnych/kube-apiserver-amd64:v1.10.0
docker pull cnych/kube-scheduler-amd64:v1.10.0
docker pull cnych/kube-controller-manager-amd64:v1.10.0
docker pull cnych/kube-proxy-amd64:v1.10.0
docker pull cnych/k8s-dns-kube-dns-amd64:1.14.8
docker pull cnych/k8s-dns-dnsmasq-nanny-amd64:1.14.8
docker pull cnych/k8s-dns-sidecar-amd64:1.14.8
docker pull cnych/etcd-amd64:3.1.12
docker pull cnych/flannel:v0.10.0-amd64
docker pull cnych/pause-amd64:3.1

修改镜像名字
docker tag cnych/kube-apiserver-amd64:v1.10.0 k8s.gcr.io/kube-apiserver-amd64:v1.10.0
docker tag cnych/kube-scheduler-amd64:v1.10.0 k8s.gcr.io/kube-scheduler-amd64:v1.10.0
docker tag cnych/kube-controller-manager-amd64:v1.10.0 k8s.gcr.io/kube-controller-manager-amd64:v1.10.0
docker tag cnych/kube-proxy-amd64:v1.10.0 k8s.gcr.io/kube-proxy-amd64:v1.10.0
docker tag cnych/k8s-dns-kube-dns-amd64:1.14.8 k8s.gcr.io/k8s-dns-kube-dns-amd64:1.14.8
docker tag cnych/k8s-dns-dnsmasq-nanny-amd64:1.14.8 k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64:1.14.8
docker tag cnych/k8s-dns-sidecar-amd64:1.14.8 k8s.gcr.io/k8s-dns-sidecar-amd64:1.14.8
docker tag cnych/etcd-amd64:3.1.12 k8s.gcr.io/etcd-amd64:3.1.12
docker tag cnych/flannel:v0.10.0-amd64 quay.io/coreos/flannel:v0.10.0-amd64
docker tag cnych/pause-amd64:3.1 k8s.gcr.io/pause-amd64:3.1

```
docker rmi cnych/kube-apiserver-amd64:v1.10.0
docker rmi cnych/kube-scheduler-amd64:v1.10.0
docker rmi cnych/kube-controller-manager-amd64:v1.10.0
docker rmi cnych/kube-proxy-amd64:v1.10.0
docker rmi cnych/k8s-dns-kube-dns-amd64:1.14.8
docker rmi cnych/k8s-dns-dnsmasq-nanny-amd64:1.14.8
docker rmi cnych/k8s-dns-sidecar-amd64:1.14.8
docker rmi cnych/etcd-amd64:3.1.12
docker rmi cnych/flannel:v0.10.0-amd64
docker rmi cnych/pause-amd64:3.1
```

其他node节点,拉取images,并修改名字

```
docker pull cnych/kube-proxy-amd64:v1.10.0
docker pull cnych/flannel:v0.10.0-amd64
docker pull cnych/pause-amd64:3.1
docker pull cnych/kubernetes-dashboard-amd64:v1.8.3
docker pull cnych/heapster-influxdb-amd64:v1.3.3
docker pull cnych/heapster-grafana-amd64:v4.4.3
docker pull cnych/heapster-amd64:v1.4.2

docker rmi cnych/kube-proxy-amd64:v1.10.0
docker rmi cnych/flannel:v0.10.0-amd64
docker rmi cnych/pause-amd64:3.1
docker rmi cnych/kubernetes-dashboard-amd64:v1.8.3
docker rmi cnych/heapster-influxdb-amd64:v1.3.3
docker rmi cnych/heapster-grafana-amd64:v4.4.3
docker rmi cnych/heapster-amd64:v1.4.2
```

修改名字

```
docker tag cnych/flannel:v0.10.0-amd64 quay.io/coreos/
flannel:v0.10.0-amd64
docker tag cnych/pause-amd64:3.1 k8s.gcr.io/pause-amd64:3.1
docker tag cnych/kube-proxy-amd64:v1.10.0 k8s.gcr.io/kube-proxy-
amd64:v1.10.0
docker tag cnych/kubernetes-dashboard-amd64:v1.8.3 k8s.gcr.io/
kubernetes-dashboard-amd64:v1.8.3
docker tag cnych/heapster-influxdb-amd64:v1.3.3 k8s.gcr.io/heapster-
influxdb-amd64:v1.3.3
docker tag cnych/heapster-grafana-amd64:v4.4.3 k8s.gcr.io/heapster-
grafana-amd64:v4.4.3
docker tag cnych/heapster-amd64:v1.4.2 k8s.gcr.io/heapster-
amd64:v1.4.2
```

2.在确保docker安装完成后，上面的相关环境配置也完成了，对应所需要的镜像(如果可以科学上网可以跳过这一步)也下载完成了，现在我们就可以来安装kubeadm了，我们这里是通过指定yum源的方式来进行安装的：

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes] name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg        https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

当然了，上面的yum源也是需要科学上网的，如果不能科学上网的话，我们可以使用阿里云的源进行安装：

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=http://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg    http://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF
```

由于我之前安装的时候最新版本是1.10版本，所以我上面对应的镜像都是1.10版本对应的镜像，现在阿里云对应的版本最新是1.10.3了，所以需要安装指定的版本，不然镜像会对应不上的

```
yum install -y kubelet-1.10.0-0 kubeadm-1.10.0-0 kubectl-1.10.0-0
```

配置kubelet

安装完成后，我们还需要对kubelet进行配置，因为用yum源的方式安装的kubelet生成的配置文件将参数--cgroup-driver改成了systemd，而docker的cgroup-driver是cgroupfs，这二者必须一致才行，我们可以通过docker info命令

```
[root@master ~]# docker info |grep Cgroup
Cgroup Driver: cgroupfs
```

修改文件kubelet的配置文件/etc/systemd/system/kubelet.service.d/10-

kubeadm.conf，将其中的KUBELET_CGROUP_ARGS参数更改成cgroupfs：
Environment="KUBELET_CGROUP_ARGS=--cgroup-driver=cgroupfs"

Kubernetes从1.8开始要求关闭系统的 Swap ，如果不关闭，默认配置的kubelet将无法启动，我们可以通过 kubelet 的启动参数--fail-swap-on=false更改这个限制，所以我们需要在上面的配置文件中增加一项配置(在ExecStart之前)：
Environment="KUBELET_EXTRA_ARGS=--fail-swap-on=false"
当然最好的还是将swap给关掉，这样能提高kubelet的性能。修改完成后，重新加载我们的配置文件即可：

systemctl daemon-reload

集群安装
1.初始化
到这里我们的准备工作就完成了，接下来我们就可以在master节点上用kubeadm命令来初始化我们的集群了：
[root@master ~]# kubeadm init --kubernetes-version=v1.10.0 --pod-network-cidr=10.244.0.0/16
Your Kubernetes master has initialized successfully!

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

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

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

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

命令非常简单，就是kubeadm init，后面的参数是需要安装的集群版本，因为我们这里选择flannel作为 Pod 的网络插件，所以需要指定–pod-network-cidr=10.244.0.0/16，然后是apiserver的通信地址，这里就是我们master节点的 IP 地址。执行上面的命令，如果出现 running with swap on is not supported. Please disable swap之类的错误，则我们还需要增加一个参数--ignore-preflight-errors=Swap来忽略swap的错误提示信息：

添加节点使用的token

kubeadm join 192.168.0.200:6443 --token at7tvr.x5vt3mraifbxoni0 --discovery-token-ca-cert-hash sha256:726ab76c651e13686be7f9c18d4c1c44f07057e7cef136eb4ef5328

token创建及使用
[root@master ~]# kubeadm token list ### token用于机器加入kubernetes集群时用到，默认token 24小时就会过期，后续的机器要加入集群需要重新生成token
TOKEN TTL EXPIRES USAGES DESCRIPTION EXTRA GROUPS
f2b012.1416e09e3d1fff4d 23h 2018-06-24T21:26:17+08:00 authentication,signing The default bootstrap token generated by 'kubeadm init'. system:bootstrappers:kubeadm:default-node-token

24小时候失效重新创建需要如下命令：
[root@master ~]# kubeadm token create --print-join-command
kubeadm join --token 48a5ec.6297ad9983652bc6 192.168.191.175:6443 --discovery-token-ca-cert-hash sha256:25e52920789d850d1b04b032e0da4a814fa0efd9ee85542500769b

上面的信息记录了kubeadm初始化整个集群的过程，生成相关的各种证书、kubeconfig文件、bootstraptoken等等，后边是使用kubeadm join往集群中添加节点时用到的命令，下面的命令是配置如何使用kubectl访问集群的方式： mkdir -p $HOME/.kube sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config sudo chown $(id -u):$(id -g) $HOME/.kube/config 最后给出了将节点加入集群的命令：
对于非root用户
[root@master ~]# mkdir -p $HOME/.kube
[root@master ~]# cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[root@master ~]# chown $(id -u):$(id -g) $HOME/.kube/config

对于root用户
export KUBECONFIG=/etc/kubernetes/admin.conf
也可以直接放到~/.bash_profile
echo "export KUBECONFIG=/etc/kubernetes/admin.conf" >> ~/.bash_profile
source一下环境变量
source ~/.bash_profile

kubectl版本测试
[root@master ~]# kubectl version
Client Version: version.Info{Major:"1", Minor:"10",

GitVersion:"v1.10.0",
GitCommit:"fc32d2f3698e36b93322a3465f63a14e9f0eaead",
GitTreeState:"clean", BuildDate:"2018-03-26T16:55:54Z",
GoVersion:"go1.9.3", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"10",
GitVersion:"v1.10.0",
GitCommit:"fc32d2f3698e36b93322a3465f63a14e9f0eaead",
GitTreeState:"clean", BuildDate:"2018-03-26T16:44:10Z",
GoVersion:"go1.9.3", Compiler:"gc", Platform:"linux/amd64"}

安装网络，可以使用flannel、calico、weave、macvlan这里我们用flannel。
下载此文件
wget https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Documentation/kube-flannel.yml

若要修改网段，需要kubeadm –pod-network-cidr=和这里同步
vim kube-flannel.yml
修改network项
"Network": "10.244.0.0/16",

执行
kubectl create -f kube-flannel.yml

排错
问题一：

[root@k8s-master ~]# kubeadm init --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=172.17.1.52 --skip-preflight-checks --kubernetes-version=stable-1.7.5
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production clusters.
unable to fetch release information. URL: "https://storage.googleapis.com/kubernetes-release/release/stable-1.7.5.txt" Status: 404 Not Found
#解决：
添加版本信息“--kubernetes-version=v1.7.5”，kubeadm reset，再次执行init问题二：

Dec 05 18:49:21 k8s-master kubelet[106548]: W1205 18:49:21.323220 106548 cni.go:189] Unable to update cni config: No

networks found in /etc/cni/net.d
Dec 05 18:49:21 k8s-master kubelet[106548]: E1205 18:49:21.323313 106548 kubelet.go:2136] Container runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni config uninitialized
#解决：
修改文件内容： /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
Environment="KUBELET_NETWORK_ARGS=--network-plugin=cni --cni-conf-dir=/etc/cni/ --cni-bin-dir=/opt/cni/bin""问题三：

Dec 05 18:48:49 k8s-master kubelet[106548]: E1205 18:48:49.825136 106548 reflector.go:190] k8s.io/kubernetes/pkg/kubelet/config/apiserver.go:46: Failed to list *v1.Pod: Get https://172.17.1.52:6443/api/v1/pods?fieldSelector=spec.nodeName%3Dk8s-master&resourceVersion=0: dial tcp 172.17.1.52:6443: getsockopt: connection refused
Dec 05 18:48:49 k8s-master kubelet[106548]: E1205 18:48:49.827944 106548 reflector.go:190] k8s.io/kubernetes/pkg/kubelet/kubelet.go:400: Failed to list *v1.Service: Get https://172.17.1.52:6443/api/v1/services?resourceVersion=0: dial tcp 172.17.1.52:6443: getsockopt: connection refused
Dec 05 18:48:49 k8s-master kubelet[106548]: E1205 18:48:49.839976 106548 reflector.go:190] k8s.io/kubernetes/pkg/kubelet/kubelet.go:408: Failed to list *v1.Node: Get https://172.17.1.52:6443/api/v1/nodes?fieldSelector=metadata.name%3Dk8s-master&resourceVersion=0: dial tcp 172.17.1.52:6443: getsockopt: connection refused
Dec 05 18:48:50 k8s-master kubelet[106548]: E1205 18:48:50.293031 106548 event.go:209] Unable to write event: 'Post https://172.17.1.52:6443/api/v1/namespaces/kube-system/events: dial tcp 172.17.1.52:6443: getsockopt: connection refused' (may retry after sleeping)
Dec 05 18:48:50 k8s-master kubelet[106548]: W1205 18:48:50.677732 106548 status_manager.go:431] Failed to get status for pod "etcd-k8s-master_kube-system (5802ae0664772d031dee332b3c63498e)": Get https://172.17.1.52:6443/api/v1/namespaces/kube-system/pods/etcd-k8s-master: dial tcp 172.17.1.52:6443: getsockopt: connection refused
#解决：
打开防火墙：
systemctl start firewalld
添加火墙规则：
firewall-cmd --zone=public --add-port=80/tcp --permanent
firewall-cmd --zone=public --add-port=6443/tcp --permanent
firewall-cmd --zone=public --add-port=2379-2380/tcp --permanent
firewall-cmd --zone=public --add-port=10250-10255/tcp --permanent

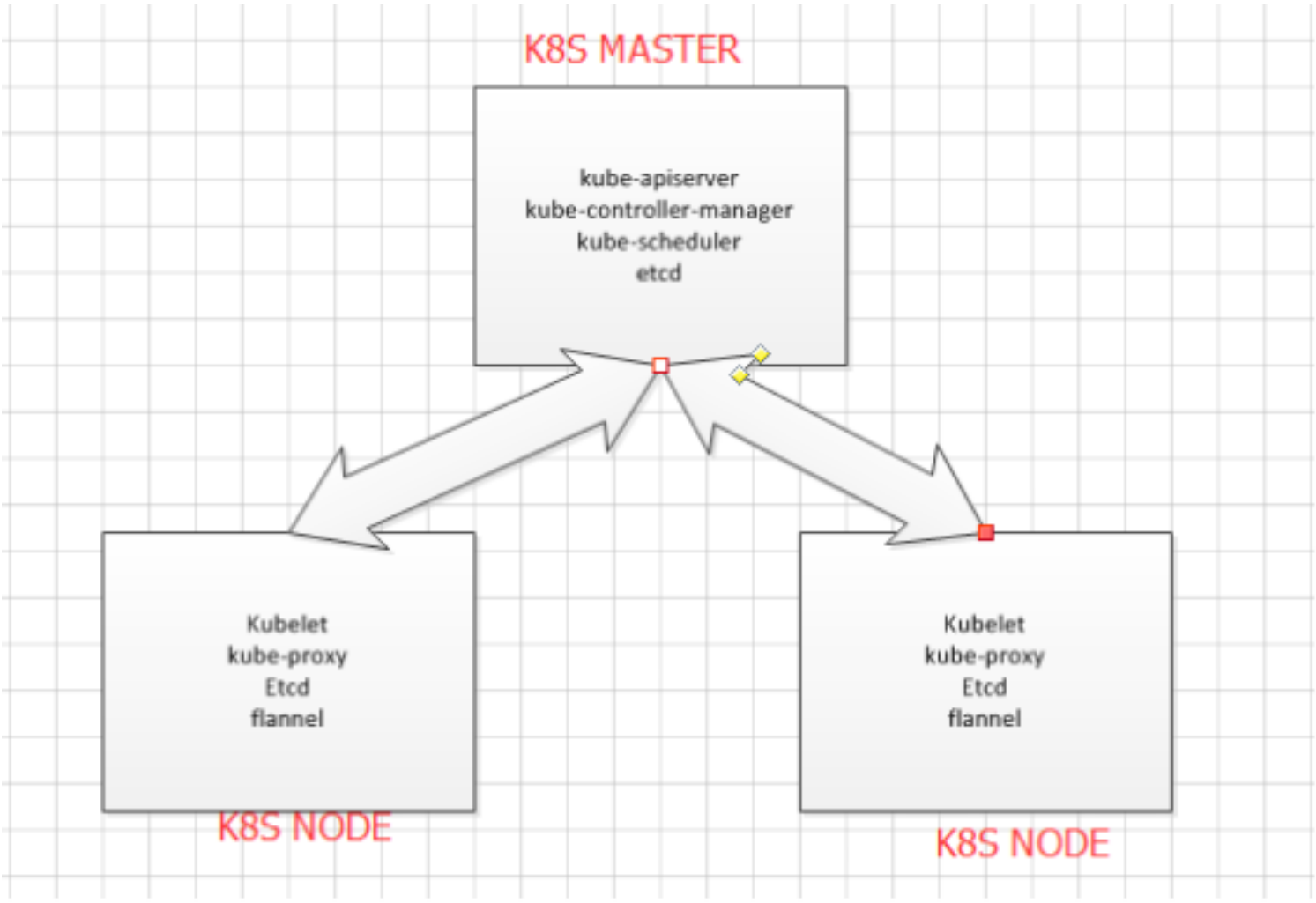
```
firewall-cmd --zone=public --add-port=30000-32767/tcp --permanent
firewall-cmd --reload
firewall-cmd --zone=public --list-ports问题四：

[root@localhost kubernetes-1.9.1]# kubectl get node
Unable to connect to the server: x509: certificate signed by unknown authority (possibly because of "crypto/rsa: verification error" while trying to verify candidate authority certificate "kubernetes")

#解决：
[root@localhost kubernetes-1.9.1]# mv  $HOME/.kube $HOME/.kube.bak
[root@localhost kubernetes-1.9.1]# mkdir -p $HOME/.kube
[root@localhost kubernetes-1.9.1]# sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[root@localhost kubernetes-1.9.1]# sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

master节点v1.11.1

一.环境准备



主机准备
192.168.0.200 master.com
192.168.0.201 node1.com
192.168.0.202 node2.com

1.关闭防火墙
systemctl stop firewalld
systemctl disable firewalld

2.关闭selinux
setenforce 0
cat /etc/selinux/config
SELINUX=disabled

3.创建/etc/sysctl.d/k8s.conf,添加如下内容
cat > /etc/sysctl.d/k8s.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF

4.执行如下命令,是修改生效
modprobe br_netfilter
sysctl -p /etc/sysctl.d/k8s.conf

5.安装系统工具
yum install -y yum-utils device-mapper-persistent-data lvm2

6.添加软件源信息
yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

7.查看docker软件版本,安装指定版本
yum list docker-ce.x86_64 --showduplicates | sort -r
yum -y install docker-ce-[VERSION]
比如:
yum -y install docker-ce-17.12.1.ce-1.el7.centos

[root@master ~]# yum remove docker-ce-cli-1:18.09.0-3.el7.x86_64
[root@master ~]# yum remove docker-ce

[root@master ~]# yum -y install docker-ce-18.06.1.ce-3.el7

8.安装完毕,启动docker
systemctl start docker
systemctl enable docker

二.安装k8s

docker pull cloudnil/etcd-amd64:3.2.18
docker pull cloudnil/pause-amd64:3.1
docker pull cloudnil/kube-proxy-amd64:v1.11.1
docker pull cloudnil/kube-scheduler-amd64:v1.11.1
docker pull cloudnil/kube-controller-manager-amd64:v1.11.1
docker pull cloudnil/kube-apiserver-amd64:v1.11.1
docker pull cloudnil/k8s-dns-sidecar-amd64:1.14.4
docker pull cloudnil/k8s-dns-kube-dns-amd64:1.14.4
docker pull cloudnil/k8s-dns-dnsmasq-nanny-amd64:1.14.4
docker pull cloudnil/kube-discovery-amd64:1.0
docker pull cloudnil/dnsmasq-metrics-amd64:1.0
docker pull cloudnil/exechealthz-amd64:1.2
docker pull cloudnil/coredns:1.1.3
#对镜像重命名
docker tag cloudnil/etcd-amd64:3.2.18 k8s.gcr.io/etcd-amd64:3.2.18
docker tag cloudnil/pause-amd64:3.1 k8s.gcr.io/pause:3.1
docker tag cloudnil/kube-proxy-amd64:v1.11.1 k8s.gcr.io/kube-proxy-amd64:v1.11.1
docker tag cloudnil/kube-scheduler-amd64:v1.11.1 k8s.gcr.io/kube-scheduler-amd64:v1.11.1

```
docker tag cloudnil/kube-controller-manager-amd64:v1.11.1 k8s.gcr.io/kube-controller-manager-amd64:v1.11.1
docker tag cloudnil/kube-apiserver-amd64:v1.11.1 k8s.gcr.io/kube-apiserver-amd64:v1.11.1
docker tag cloudnil/kube-discovery-amd64:1.0 k8s.gcr.io/kube-discovery-amd64:1.0
docker tag cloudnil/k8s-dns-sidecar-amd64:1.14.4 k8s.gcr.io/k8s-dns-sidecar-amd64:1.14.4
docker tag cloudnil/k8s-dns-kube-dns-amd64:1.14.4 k8s.gcr.io/k8s-dns-kube-dns-amd64:1.14.4
docker tag cloudnil/k8s-dns-dnsmasq-nanny-amd64:1.14.4 k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64:1.14.4
docker tag cloudnil/dnsmasq-metrics-amd64:1.0 k8s.gcr.io/dnsmasq-metrics-amd64:1.0
docker tag cloudnil/exechealthz-amd64:1.2 k8s.gcr.io/exechealthz-amd64:1.2
docker tag cloudnil/coredns:1.1.3   k8s.gcr.io/coredns:1.1.3
#删除镜像
docker rmi cloudnil/etcd-amd64:3.2.18
docker rmi cloudnil/pause-amd64:3.1
docker rmi cloudnil/kube-proxy-amd64:v1.11.1
docker rmi cloudnil/kube-scheduler-amd64:v1.11.1
docker rmi cloudnil/kube-controller-manager-amd64:v1.11.1
docker rmi cloudnil/kube-apiserver-amd64:v1.11.1
docker rmi cloudnil/k8s-dns-sidecar-amd64:1.14.4
docker rmi cloudnil/k8s-dns-kube-dns-amd64:1.14.4
docker rmi cloudnil/k8s-dns-dnsmasq-nanny-amd64:1.14.4
docker rmi cloudnil/kube-discovery-amd64:1.0
docker rmi cloudnil/dnsmasq-metrics-amd64:1.0
docker rmi cloudnil/exechealthz-amd64:1.2
docker rmi cloudnil/coredns:1.1.3
```

2.在确保docker安装完成后，上面的相关环境配置也完成了，对应所需要的镜像(如果可以科学上网可以跳过这一步)也下载完成了，现在我们就可以来安装kubeadm了，我们这里是通过指定yum源的方式来进行安装的：

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes] name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg        https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

当然了，上面的yum源也是需要科学上网的，如果不能科学上网的话，我们可以使用阿里

云的源进行安装：
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=http://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg http://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF

由于我之前安装的时候最新版本是1.10版本，所以我上面对应的镜像都是1.10版本对应的镜像，现在阿里云对应的版本最新是1.10.3了，所以需要安装指定的版本，不然镜像会对应不上的
yum install -y kubelet-1.11.1 kubeadm-1.11.1 kubectl-1.11.1

systemctl enable docker.service && systemctl start docker.service
systemctl enable kubelet.service && systemctl start kubelet.service

集群安装
关闭交换分区
[root@master ~]# swapoff -a
1.初始化
到这里我们的准备工作就完成了，接下来我们就可以在master节点上用kubeadm命令来初始化我们的集群了：
kubeadm init --kubernetes-version=v1.11.1 --pod-network-cidr=10.244.0.0/16

成功完成
Your Kubernetes master has initialized successfully!

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

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

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

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

命令非常简单，就是kubeadm init，后面的参数是需要安装的集群版本，因为我们这里选择flannel作为 Pod 的网络插件，所以需要指定–pod-network-cidr=10.244.0.0/16，然后是apiserver的通信地址，这里就是我们master节点的 IP 地址。执行上面的命令，如果出现 running with swap on is not supported. Please disable swap之类的错误，则我们还需要增加一个参数--ignore-preflight-errors=Swap来忽略swap的错误提示信息：

添加节点使用的token
kubeadm join 192.168.0.200:6443 --token at7tvr.x5vt3mraifbxoni0 --discovery-token-ca-cert-hash sha256:726ab76c651e13686be7f9c18d4c1c44f07057e7cef136eb4ef5328

token创建及使用
[root@master ~]# kubeadm token list ### token用于机器加入kubernetes集群时用到，默认token 24小时就会过期，后续的机器要加入集群需要重新生成token
TOKEN TTL EXPIRES USAGES DESCRIPTION EXTRA GROUPS
f2b012.1416e09e3d1fff4d 23h 2018-06-24T21:26:17+08:00 authentication,signing The default bootstrap token generated by 'kubeadm init'. system:bootstrappers:kubeadm:default-node-token

24小时候失效重新创建需要如下命令：
[root@master ~]# kubeadm token create --print-join-command
kubeadm join --token 48a5ec.6297ad9983652bc6 192.168.191.175:6443 --discovery-token-ca-cert-hash sha256:25e52920789d850d1b04b032e0da4a814fa0efd9ee85542500769b

上面的信息记录了kubeadm初始化整个集群的过程，生成相关的各种证书、kubeconfig文件、bootstraptoken等等，后边是使用kubeadm join往集群中添加节点时用到的命令，下面的命令是配置如何使用kubectl访问集群的方式： mkdir -p $HOME/.kube sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config sudo chown $(id -u):$(id -g) $HOME/.kube/config 最后给出了将节点加入集群的命令：
对于非root用户
[root@master ~]# mkdir -p $HOME/.kube
[root@master ~]# cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[root@master ~]# chown $(id -u):$(id -g) $HOME/.kube/config

对于root用户
export KUBECONFIG=/etc/kubernetes/admin.conf
也可以直接放到~/.bash_profile
echo "export KUBECONFIG=/etc/kubernetes/admin.conf" >>
~/.bash_profile
source一下环境变量
source ~/.bash_profile

kubectl版本测试
[root@master ~]# kubectl version
Client Version: version.Info{Major:"1", Minor:"10",
GitVersion:"v1.10.0",
GitCommit:"fc32d2f3698e36b93322a3465f63a14e9f0eaead",
GitTreeState:"clean", BuildDate:"2018-03-26T16:55:54Z",
GoVersion:"go1.9.3", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"10",
GitVersion:"v1.10.0",
GitCommit:"fc32d2f3698e36b93322a3465f63a14e9f0eaead",
GitTreeState:"clean", BuildDate:"2018-03-26T16:44:10Z",
GoVersion:"go1.9.3", Compiler:"gc", Platform:"linux/amd64"}

安装网络，可以使用flannel、calico、weave、macvlan这里我们用flannel。
下载此文件
wget https://raw.githubusercontent.com/coreos/flannel/v0.9.1/
Documentation/kube-flannel.yml

若要修改网段，需要kubeadm –pod-network-cidr=和这里同步
vim kube-flannel.yml
修改network项
"Network": "10.244.0.0/16",

执行
kubectl create -f kube-flannel.yml

kubeadm join 192.168.122.202:6443 --token euwr0i.xffjh5hifm7qlrqm
--discovery-token-ca-cert-hash
sha256:b17d61dce6e134019c716df17ff4a70b275888b464fa660951e41e3

node节点

master节点所做操作都要做,除了初始化
docker pull cloudnil/kube-proxy-amd64:v1.11.1
docker pull cnych/flannel:v0.10.0-amd64
docker pull cloudnil/pause-amd64:3.1
docker pull cnych/kubernetes-dashboard-amd64:v1.8.3
docker pull cnych/heapster-influxdb-amd64:v1.3.3
docker pull cnych/heapster-grafana-amd64:v4.4.3
docker pull cnych/heapster-amd64:v1.4.2

docker tag cnych/flannel:v0.10.0-amd64 quay.io/coreos/flannel:v0.10.0-amd64
docker tag cloudnil/pause-amd64:3.1 k8s.gcr.io/pause:3.1
docker tag cloudnil/kube-proxy-amd64:v1.11.1 k8s.gcr.io/kube-proxy-amd64:v1.11.1
docker tag cnych/kubernetes-dashboard-amd64:v1.8.3 k8s.gcr.io/kubernetes-dashboard-amd64:v1.8.3
docker tag cnych/heapster-influxdb-amd64:v1.3.3 k8s.gcr.io/heapster-influxdb-amd64:v1.3.3
docker tag cnych/heapster-grafana-amd64:v4.4.3 k8s.gcr.io/heapster-grafana-amd64:v4.4.3
docker tag cnych/heapster-amd64:v1.4.2 k8s.gcr.io/heapster-amd64:v1.4.2

docker rmi cloudnil/kube-proxy-amd64:v1.11.1
docker rmi cnych/flannel:v0.10.0-amd64
docker rmi cloudnil/pause-amd64:3.1
docker rmi cnych/kubernetes-dashboard-amd64:v1.8.3
docker rmi cnych/heapster-influxdb-amd64:v1.3.3
docker rmi cnych/heapster-grafana-amd64:v4.4.3
docker rmi cnych/heapster-amd64:v1.4.2

加入集群
kubeadm join --token 10f625.cfecd30f2d65712b 192.168.122.201:6443 --discovery-token-ca-cert-hash sha256:dacb139347071e4c56d4f27647f8ff5ea9f7d19973306ae48db0543

master察看
[root@master tmp]# kubectl get nodes
NAME STATUS ROLES AGE VERSION
master Ready master 50m v1.9.0
node1 Ready <none> 16m v1.9.0

kubernetes会在每个node节点创建flannel和kube-proxy的pod

```
[root@master tmp]# kubectl get pods --all-namespaces
NAMESPACE     NAME                              READY     STATUS    RESTARTS   AGE
kube-system   etcd-master                       1/1       Running   0          36m
kube-system   kube-apiserver-master             1/1       Running   0          36m
kube-system   kube-controller-manager-master    1/1       Running   0          36m
kube-system   kube-dns-6f4fd4bdf-s59kh          3/3       Running   0          52m
kube-system   kube-flannel-ds-pstr7             1/1       Running   0          18m
kube-system   kube-flannel-ds-xwc4j             1/1       Running   0          36m
kube-system   kube-proxy-5mnxr                  1/1       Running   0          52m
kube-system   kube-proxy-5xtwr                  1/1       Running   0          18m
kube-system   kube-scheduler-master             1/1       Running   0          36m
```

查看集群信息

```
[root@master tmp]# kubectl cluster-info
Kubernetes master is running at https://192.168.122.201:6443
KubeDNS is running at https://192.168.122.201:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

集群状态

```
[root@master tmp]# kubectl get cs
NAME                 STATUS    MESSAGE             ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

测试集群

在master节点上发起个创建应用请求
这里我们创建个名为httpd-app的应用，镜像为httpd，有两个副本pod

```
[root@master tmp]# kubectl run httpd-app --image=httpd --replicas=2
```

```
[root@master tmp]# kubectl get deployment
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
httpd-app   2         2         2            2           1m
```

检查pod
可以看见pod分布在node-1和node-2上

```
[root@master tmp]# kubectl get pods
NAME                     READY    STATUS    RESTARTS   AGE
httpd-app-5fbccd7c6c-2fq46   1/1      Running   0          2m
httpd-app-5fbccd7c6c-szzn7   1/1      Running   0          2m
[root@master tmp]# kubectl get pods -o wide
NAME                     READY    STATUS    RESTARTS   AGE      IP
NODE
httpd-app-5fbccd7c6c-2fq46   1/1      Running   0          2m
10.244.1.3   node1
httpd-app-5fbccd7c6c-szzn7   1/1      Running   0          2m
10.244.1.2   node1
```

访问测试页

```
[root@master tmp]# curl 10.244.1.2
<html><body><h1>It works!</h1></body></html>
[root@master tmp]# curl 10.244.1.3
<html><body><h1>It works!</h1></body></html>
```

kubectl delete deployment httpd-app

dashboard

docker pull cnych/kubernetes-dashboard-amd64:v1.10.0
docker tag cnych/kubernetes-dashboard-amd64:v1.10.0 k8s.gcr.io/kubernetes-dashboard-amd64:v1.10.0

部署kubernetes-dashboard

kubernetes-dashboard是可选组件，因为，实在不好用，功能太弱了。
建议在部署master时一起把kubernetes-dashboard一起部署了,不然在node节点加入集群后，kubernetes-dashboard会被kube-scheduler调度node节点上，这样根kube-apiserver通信需要额外配置。
下载kubernetes-dashboard的配置文件或直接使用离线包里面的kubernetes-dashboard.yaml

wget https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml

修改kubernetes-dashboard.yaml
如果需要让外面访问需要修改这个yaml文件端口类型为NodePort默认为clusterport外部访问不了，
修改service段:

```
kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  type: NodePort
  ports:
    - port: 443
      targetPort: 8443
      nodePort: 32666
  selector:
    k8s-app: kubernetes-dashboard
```

service 注意 service提供的ip是virtual ip，无法ping通。但是可以通过virutal ip访问端口（iptables转发）
三种端口：重点！！！
targetPort targetPort是pod上的端口，从port和nodePort上到来的数据最终经过kube-proxy流入到后端pod的targetPort上进入容器。
port service暴露在cluster ip上的端口，<cluster ip>:port 是提供给集群内部客户访问service的入口。
nodePort 是kubernetes提供给集群外部客户访问service入口的一种方式（另一种方式是LoadBalancer），所以，<nodeIP>:nodePort 是提供给集群外部客户访问service的入口。

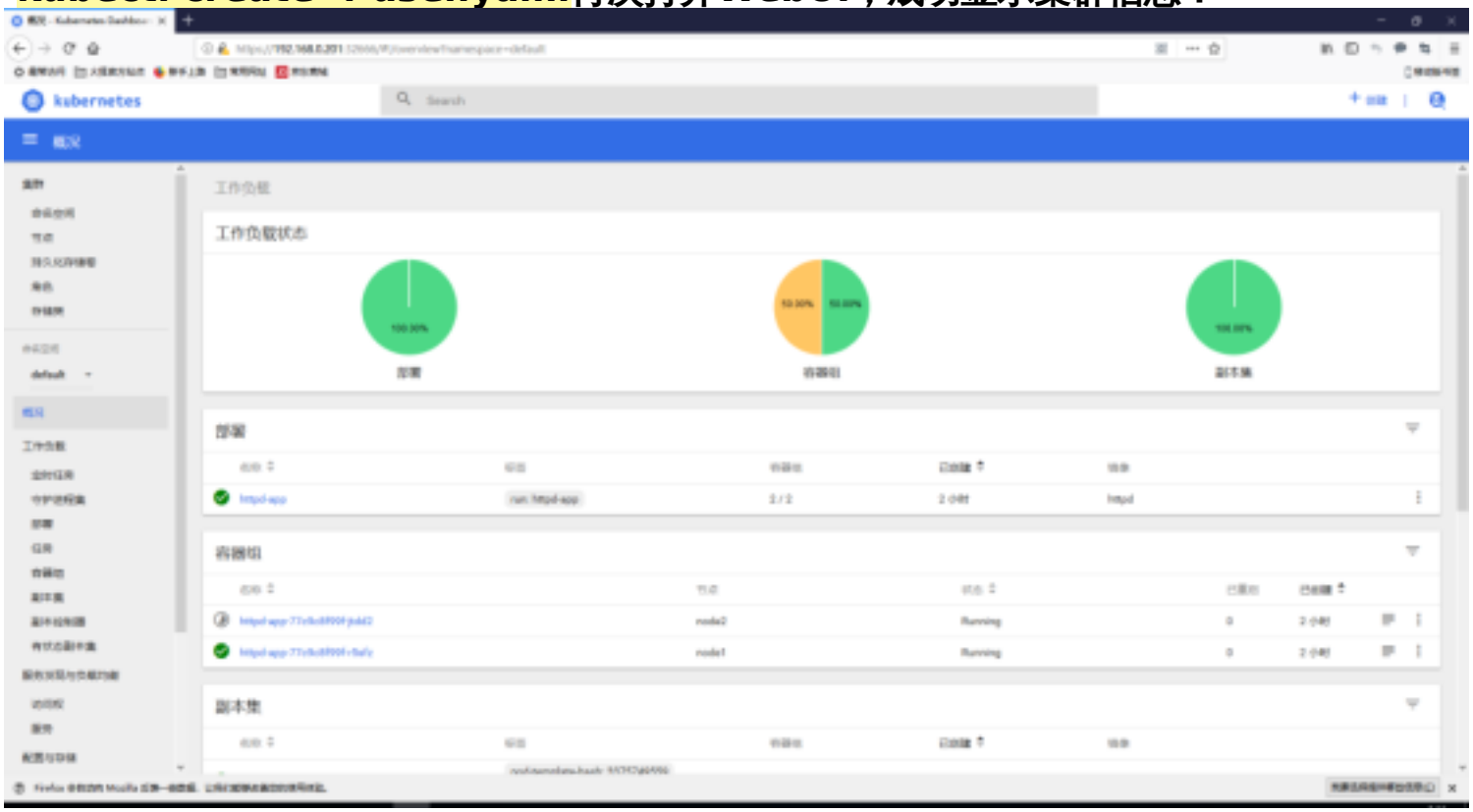
开启容器:
kubectl create -f kubernetes-dashboard.yaml

查询
[root@master ~]# kubectl get deployment --all-namespaces
[root@master ~]# kubectl get svc --all-namespaces
访问:
https://192.168.0.200:32666

这是需要用一个可用的ClusterRole进行登录，该账户需要有相关的集群操作权限，如果跳过，则是用默认的系统角色kubernetes-dashboard（该角色在创建该容器时生成），初始状态下该角色没有任何权限，需要在系统中进行配置，角色绑定：在主节点上任意位置创建一个文件xxx.yaml，名字随意： vim ClusterRoleBinding.yaml
编辑文件：

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: kubernetes-dashboard
subjects:
 - kind: ServiceAccount
   name: kubernetes-dashboard
   namespace: kube-system
roleRef:
   kind: ClusterRole
   name: cluster-admin
   apiGroup: rbac.authorization.k8s.io
```

保存，退出，执行该文件：
kubectl create -f user.yaml再次打开WebUI，成功显示集群信息：



注意⚠：给kubernetes-dashboard角色赋予cluster-admin权限仅供测试使用，本身这种方式并不安全，建议新建一个系统角色，分配有限的集群操作权限，方法如下：新建一个yaml文件，写入：

```
kind: ClusterRole #创建集群角色
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dashboard  #角色名称
rules:
- apiGroups: ["*"]
  resources: ["*”] #所有资源
  verbs: ["get", "watch", "list", "create","proxy","update”] #赋予获取>，监听，列表，创建，代理，更新的权限
- apiGroups: ["*"]
```

```
  resources: ["pods”] #容器资源单独配置（在所有资源配置的基础上）
  verbs: ["delete”] #提供删除权限
---
apiVersion: v1
kind: ServiceAccount  #创建ServiceAccount
metadata:
  name: dashboard
  namespace: kube-system
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dashboard-extended
subjects:
  -kind: ServiceAccount
    name: dashboard
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name:dashboard   #填写cluster-admin代表开放全部权限
  apiGroup: rbac.authorization.k8s.io
```

执行该文件，查看角色是否生成：

```
kubectl get serviceaccount --all-namespaces
```

查询该账户的密钥名：

```
kubectl get secret  -n kube-system
```

根据密钥名找到token：

```
kubectl discribe secret dashboard-token-wd9rz -n kube-system
```

输出一段信息：

将此token用于登陆WebUI即可。

```
Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJ
uZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY29
1bnQvc2VjcmV0Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZC10b2tlbi13ZDlyeiIsImt1YmVybmV0ZXMuaW8vc2VydmljZWF
jY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWF
jY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6ImMwNTc1ZTgyLTNkNGUtMTFlOC1hOWQ2LTAwMWM0MjhhNzM3NiIsInN1YiI6InN
5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlLXN5c3RlbTprdWJlcm5ldGVzLWRhc2hib2FyZCJ9.kIenTTaDDC7SiX3tCA7fbtRxA0
LAspQ0_uAbD8n71QrhtcfA-JnWSrNCfnQgIYjD9PDXhvOhjUJhxoXcSXPGUdyDVap4iCzgz5-sdrv1xGUGDiaN9f-tAcWjikCLSi
DDsKDSo1EreYHkOGmKnnnpvN8E8sb16VeKoctc3unQwkC00guyG6K_smIXgy9ZrTqFmLfarBfvk_DRPfCJ77A8pCkyPpa9yYOkKG
XpzMr8yiE17HlmHMBFgd0tF9TT-NfTSjwbAbDzSMR1nEsWQudJ35vi4TNfFdsgfNNeKvqDzZR4W45PPH8DsXiWixXmk0Jp1ffGd3
TSJT7fnUVT9jKhLw
```

kubectl的使用

工具
yum install epel-*
yum install python34
yum install python34 python34-pip
pip3.4 install kube-shell

查看集群有多少节点
[root@master ~]# kubectl get nodes
NAME STATUS ROLES AGE VERSION
master Ready master 2h v1.10.0
node1 Ready <none> 1h v1.10.0
node2 Ready <none> 1h v1.10.0

查看版本
[root@master ~]# kubectl api-versions

kubectl get 查看某个类型的资源
kubectl describe 查看特定的资源显示详细信息
kubectl logs 查看某个pod的日志
kubectl exec 在容器内执行命令
kubectl scale 实现水平扩展或伸缩
kubectl rollout status 部署状态变更检查
kubectl rollout history 部署历史
Kubectl rollout undo 回滚部署到最近的某个版本
kubectl set images 升级某个deployment的image到新的版本
kubectl delete 删除某个资源

[root@master ~]# kubectl get pods --all-namespaces
NAMESPACE NAME READY STATUS RESTARTS AGE
default httpd-app-77c9c8f99f-jtdd2 1/1 Running 0 1h
default httpd-app-77c9c8f99f-r5sfz 1/1 Running 0 1h
kube-system etcd-master 1/1 Running 1 2h
kube-system kube-apiserver-master 1/1 Running 0 2h
kube-system kube-controller-manager-master 1/1 Running 3 2h
kube-system kube-dns-86f4d74b45-2rhdr 3/3 Running 35 2h
kube-system kube-flannel-ds-h8sdj 1/1 Running

```
2         1h
kube-system   kube-flannel-ds-x5mnm                1/1       Running
1         2h
kube-system   kube-flannel-ds-xrzgz                1/1       Running
0         1h
kube-system   kube-proxy-4kclc                     1/1       Running
1         2h
kube-system   kube-proxy-bc772                     1/1       Running
0         1h
kube-system   kube-proxy-gzt4c                     1/1       Running
0         1h
kube-system   kube-scheduler-master                1/1       Running
3         2h
kube-system   kubernetes-dashboard-7d5dcdb6d9-pz6gl   1/1
Running
```

查看集群信息

```
[root@master ~]# kubectl cluster-info
Kubernetes master is running at https://192.168.0.200:6443
KubeDNS is running at https://192.168.0.200:6443/api/v1/
namespaces/kube-system/services/kube-dns:dns/proxy
```

查看组件的信息

```
[root@master ~]# kubectl get cs
NAME                 STATUS    MESSAGE             ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

创建pod ，deployment为2个

```
[root@master ~]# kubectl run httpd-app --image=httpd --replicas=2
```

查看httpd-app的deployment是否为2个

```
[root@master ~]# kubectl get deployment
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
httpd-app   2         2         2            2           1h
```

查看启动的pod信息

```
[root@master ~]# kubectl get pod
NAME                         READY     STATUS    RESTARTS   AGE
httpd-app-77c9c8f99f-jtdd2   1/1       Running   0          1h
httpd-app-77c9c8f99f-r5sfz   1/1       Running   0          1h
```

查看更详细的信息，包含运行的node和ip地址

```
[root@master ~]# kubectl get pod -o wide
NAME                         READY     STATUS    RESTARTS   AGE       IP
NODE
httpd-app-77c9c8f99f-jtdd2   1/1       Running   0          1h
```

```
10.244.2.2   node2
httpd-app-77c9c8f99f-r5sfz   1/1      Running   0        1h
10.244.1.2   node1

[root@master ~]# kubectl get namespace
NAME          STATUS    AGE
default       Active    2h
kube-public   Active    2h
kube-system   Active    2h
```

创建名称空间

```
[root@master ~]# kubectl create namespace demotest
namespace "demotest" created
[root@master ~]# kubectl get namespace
NAME          STATUS    AGE
default       Active    2h
demotest      Active    3s
kube-public   Active    2h
kube-system   Active    2h
```

在该空间内创建pod

```
[root@master ~]# kubectl run httpd-app --image=httpd --replicas=1 --
namespace=demotest
deployment.apps "httpd-app" created
```

查看

```
[root@master ~]# kubectl get pod --namespace=demotest
NAME                         READY     STATUS              RESTARTS   AGE
httpd-app-77c9c8f99f-fq6ks   0/1       ContainerCreating   0          53s
```

负载均衡

```
[root@master ~]# kubectl run httpd-app --image=httpd --replicas=2 --
port=80

[root@master ~]# kubectl get pod
NAME                         READY     STATUS    RESTARTS   AGE
httpd-app-586f898c8-4j2np    1/1       Running   0          5m
httpd-app-586f898c8-psvbj    1/1       Running   0          5m


[root@master ~]# kubectl get pod --namespace=default
NAME                         READY     STATUS    RESTARTS   AGE
NAME                         READY     STATUS    RESTARTS   AGE
httpd-app-586f898c8-4j2np    1/1       Running   0          5m
httpd-app-586f898c8-psvbj    1/1       Running   0          5m
```

```
[root@master ~]# kubectl expose deployment/httpd-app --
type="ClusterIP" --port=80
service "httpd-app" exposed

[root@master ~]# kubectl get service
NAME        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
httpd-app   ClusterIP   10.111.37.22   <none>        80/TCP    27s

[root@master ~]# kubectl get endpoints
NAME        ENDPOINTS                     AGE
httpd-app   10.244.1.2:80,10.244.2.2:80   1m

[root@master ~]# kubectl exec -it httpd-app-586f898c8-4j2np /bin/
bash
root@httpd-app-586f898c8-4j2np:/usr/local/apache2# ls
bin  build  cgi-bin  conf  error  htdocs  icons  include  logs  modules
root@httpd-app-586f898c8-4j2np:/usr/local/apache2# cd htdocs/
root@httpd-app-586f898c8-4j2np:/usr/local/apache2/htdocs# cat
index.html
<html><body><h1>It works!</h1></body></html>
root@httpd-app-586f898c8-4j2np:/usr/local/apache2/htdocs# echo
"test1" > index.html
root@httpd-app-586f898c8-4j2np:/usr/local/apache2/htdocs# cat
index.html
test1


[root@master ~]# kubectl exec -it httpd-app-586f898c8-psvbj /bin/
bash
root@httpd-app-586f898c8-psvbj:/usr/local/apache2# cd htdocs/
root@httpd-app-586f898c8-psvbj:/usr/local/apache2/htdocs# ls
index.html
root@httpd-app-586f898c8-psvbj:/usr/local/apache2/htdocs# cat
index.html
<html><body><h1>It works!</h1></body></html>
root@httpd-app-586f898c8-psvbj:/usr/local/apache2/htdocs# echo
"test2" > index.html
root@httpd-app-586f898c8-psvbj:/usr/local/apache2/htdocs# cat
index.html
test2


[root@master ~]# curl 10.98.236.206
test1
[root@master ~]# curl 10.98.236.206
test1
```

```
[root@master ~]# curl 10.98.236.206
test2
[root@master ~]# curl 10.98.236.206
test2
```

NodePort　nodePort是kubernetes提供给集群外部客户访问service入口的一种方式（另一种方式是LoadBalancer）

```
[root@master ~]# kubectl delete service httpd-app
service "httpd-app" deleted

[root@master ~]# kubectl expose deployment/httpd-app --
type="NodePort" --port=80
service "httpd-app" exposed

[root@master ~]# kubectl get service
NAME        TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)       AGE
httpd-app   NodePort   10.105.176.97   <none>        80:30232/TCP
16s

[root@master ~]# curl 10.105.176.97
test1
[root@master ~]# curl 10.105.176.97
test1
[root@master ~]# curl 10.105.176.97
test2
[root@master ~]# curl 10.105.176.97
test2



[root@master ~]# kubectl get pods -o wide
NAME                    READY    STATUS    RESTARTS   AGE      IP
NODE
httpd-app-586f898c8-4j2np   1/1      Running   0          12m
10.244.2.10   node01
httpd-app-586f898c8-psvbj   1/1      Running   0          12m
10.244.1.7    node02

[root@master ~]# curl 10.244.2.10
test1
[root@master ~]# curl 10.244.1.7
test2

[root@master ~]# kubectl get service
NAME        TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
httpd-app   NodePort   10.101.193.241   <none>        80:30367/
TCP     3h

[root@master ~]# curl node01:30367
```

<html><body><h1>It works!</h1></body></html>

kubenetes的弹性伸缩
[root@master ~]# kubectl scale deployment/httpd-app --replicas=3
deployment "httpd-app" scaled

[root@master ~]# kubectl get pods
NAME READY STATUS RESTARTS AGE
httpd-app-586f898c8-4j2np 1/1 Running 0 15m
httpd-app-586f898c8-mhvlk 1/1 Running 0 12s
httpd-app-586f898c8-psvbj 1/1 Running 0 15m

[root@master ~]# kubectl scale deployment/httpd-app --replicas=5
[root@master ~]# kubectl get pods
NAME READY STATUS RESTARTS AGE
httpd-app-586f898c8-4j2np 1/1 Running 0 16m
httpd-app-586f898c8-gn2jg 1/1 Running 0 10s
httpd-app-586f898c8-mhvlk 1/1 Running 0 1m
httpd-app-586f898c8-psvbj 1/1 Running 0 16m
httpd-app-586f898c8-zm8zx 1/1 Running 0 10s

[root@master ~]# kubectl scale deployment/httpd-app --replicas=2
[root@master ~]# kubectl get pods
NAME READY STATUS RESTARTS AGE
httpd-app-586f898c8-4j2np 1/1 Running 0 17m
httpd-app-586f898c8-psvbj 1/1 Running 0 17m

更具deployment的伸缩，service访问也会跟随deployment变化，kubernetes有一个hpa，自动伸缩

lable
[root@master ~]# kubectl get node --show-labels
NAME STATUS ROLES AGE VERSION LABELS
master Ready master 1d v1.9.0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master,node-role.kubernetes.io/master=
node01 Ready <none> 1d v1.9.0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=node01
node02 NotReady <none> 1d v1.9.0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=node02

```
[root@master ~]# kubectl label node node1 disktype=ssd
node "node01" labeled
[root@master ~]# kubectl get node node1 --show-labels
NAME     STATUS    ROLES     AGE       VERSION   LABELS
node01   Ready     <none>    1d        v1.9.0    beta.kubernetes.io/
arch=amd64,beta.kubernetes.io/
os=linux,disktype=ssd,kubernetes.io/hostname=node01

[root@master ~]# kubectl label node node1 disktype-
node "node1" labeled
[root@master ~]# kubectl get node node1 --show-labels
NAME     STATUS    ROLES     AGE       VERSION   LABELS
node1    Ready     <none>    2h        v1.10.0   beta.kubernetes.io/
arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/
hostname=node1
```

更新及回滚

```
[root@master ~]# kubectl run nginx --image=nginx:1.7.5 --replicas=4

[root@master ~]# kubectl get pod -o wide
NAME                    READY    STATUS    RESTARTS   AGE      IP
NODE
nginx-7d4bfb4d9d-lbg2s   1/1      Running   0          53s
10.244.1.46   node02
nginx-7d4bfb4d9d-rt6b6   1/1      Running   0          53s
10.244.2.42   node01
nginx-7d4bfb4d9d-tqfgz   1/1      Running   0          53s
10.244.2.43   node01
nginx-7d4bfb4d9d-vxlmh   1/1      Running   0          53s
10.244.1.45   node02

[root@master ~]# kubectl exec -it nginx-7d4bfb4d9d-tqfgz /bin/bash

root@nginx-7d4bfb4d9d-tqfgz:/# nginx -v
nginx version: nginx/1.7.5




[root@master ~]# kubectl set image deployment/nginx
nginx=nginx:1.7.9
deployment "nginx" image updated

[root@master ~]# kubectl get pod -o wide
NAME                    READY    STATUS    RESTARTS   AGE      IP
NODE
nginx-6f8cf9fbc4-4w5cd   1/1      Running   0          7s
10.244.1.48   node02
nginx-6f8cf9fbc4-k9gz6   1/1      Running   0          32s
```

```
10.244.2.44   node01
nginx-6f8cf9fbc4-ncf6l   1/1     Running   0       32s
10.244.1.47   node02
nginx-6f8cf9fbc4-v5txl   1/1     Running   0       6s      10.244.2.45
node01

[root@master ~]# kubectl exec -it nginx-6f8cf9fbc4-4w5cd /bin/bash
root@nginx-6f8cf9fbc4-4w5cd:/# nginx -v
nginx version: nginx/1.7.9

[root@master ~]# kubectl rollout history deployment/nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>


[root@master ~]# kubectl rollout undo deployment/nginx
deployment "nginx"

[root@master ~]# kubectl get pod -o wide
NAME                READY    STATUS    RESTARTS   AGE     IP
NODE
nginx-7d4bfb4d9d-9hd57   1/1     Running   0       17s
10.244.2.46   node01
nginx-7d4bfb4d9d-d9xjj   1/1     Running   0       15s
10.244.2.47   node01
nginx-7d4bfb4d9d-vkczl   1/1     Running   0       17s
10.244.1.49   node02
nginx-7d4bfb4d9d-z67gp   1/1     Running   0       15s
10.244.1.50   node02


[root@master ~]# kubectl exec -it nginx-7d4bfb4d9d-9hd57 /bin/bash
root@nginx-7d4bfb4d9d-9hd57:/# nginx -v
nginx version: nginx/1.7.5
```

使用yaml文件

```
[root@master ~]# kubectl get deployment bootcamp -o
yaml                    ### 可以将你用命令行创建的资源以yaml的文件格式导出
[root@master ~]# kubectl get service httpd-app -o yaml >
service.yaml

[root@master ~]# kubectl delete service httpd-app
```

[root@master ~]# kubectl create -f service.yaml
service "httpd-app" created

pod自动迁移
[root@master ~]# kubectl run httpd-app --image=httpd --replicas=4
deployment "httpd-app" created

[root@master ~]# kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE
httpd-app-5fbccd7c6c-5vzqm 1/1 Running 0 37s 10.244.2.54 node01
httpd-app-5fbccd7c6c-jwlp5 1/1 Running 0 37s 10.244.2.55 node01
httpd-app-5fbccd7c6c-lkbq5 1/1 Running 0 37s 10.244.1.29 node02
httpd-app-5fbccd7c6c-xs2nv 1/1 Running 0 37s 10.244.1.30 node02

在node节点测试，关闭一台node节点
[root@node02 ~]# poweroff

[root@master ~]# watch -n1 kubectl get pod -o wide #### 虚拟环境比较慢，需要等一段时间会实现自动迁移

yaml管理集群

https://v1-11.docs.kubernetes.io/docs/tutorials/kubernetes-basics/ 部署文档
https://v1-11.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.11/ api文档

关于labels
Label 是 Kubernetes 中的核心概念。一个 Label 是一个 key=value 的键值对，其中 key 与 valume 由用户自己制定。 Label 可以附加到各种资源对象，例如 Node、Pod Service、RC 等，一个资源对象可以定义任意数量的 Label，同一个 Label 也可以被添加到任意数量的资源对象上去，Label 通常在资源对象定义时 确定，也可以在对象创建后动态添加或删除。
可以通过给指定的资源对象捆绑一个或多个不同的 Label 来实现多维 度的资源分组管理功能，以便于灵活、方便地进行资源分配、调度、配 置、部署等管理工作。例如：部署不同版本的应用到不同的环境中；或 者监控和分析应用（日志记录、监控、告警）等。一些常用的 Label 示例如下。
版本标签："release": "stable" , "release" : "canary"...
环境标签："environment" : "dev" , "environment" : "qa" , "environment" : "oriduc
架构标签："tier" : "frontend" , "tier" : "backend" , "tier" : "middleware"
分区标 签："partition" : "customerA" , "partition" : "custome rB"...
质量管控标签："track" : "daily" , "track" : "weekly"

Label 相当于我们熟悉的 “标签”，给某个资源对象定义一个 Label，就相当于给它打了一个标签，随后可以通过 Label Selector（标签选择器器）查询和筛选拥有某些 Label 的资源对象， Kubernetes 通过这种方式实现了类似 SQL 的简单又通用的对象查 询机制。
Label Selector 可以被类比为 SQL 语句中的 where 查询条件， 例如， name=redis-slave 这个 Label Selector 作用于 Pod 时，可以被类比为 SELECT * FROM pod WHERE pods name = 'redis-slave' 这样的语句。当前有两种 Label Selector 的表 达式：基于等式的（Equality-based）和基于集合的（Setbased） ，
前者采用 “等式类” 的表达式匹配标签下面是一些具体的 例子。
name = redis-slave：匹配所有具有标签 name = redisslave 的资源对象。
env != production：匹配所有不具有标签 env=production 的资源对象，比如 env=dev 就是满足此条件 的标签之一。

而后者则使用集合操作的表达式匹配标签，下面是一些具体的例子。
name in（redis-master，redis-slave）：匹配所有具有 标签 name=redis-master 或name=redis-slave 的资源 对象。
name not in（php-frontend）：匹配所哟不具有标签 name=php-frontend 的资源对象。

可以通过多个 Label Selector 表达式的组合实现复杂的条件选 择，多个表达式之间用“,” 进行分割即可，几个条件之间是 “AND” 的关系，即同时满足多个条件，比如下面的例
name=redis-slave,env!=production
name not in (php-frontend),env!=production

1.创建pod

```
[root@master k8s]# cat testpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
    name: myweb
spec:
  containers:
  - name: myweb
    image: httpd
    ports:
    - containerPort: 8080
    env:
    - name: MYSQL_SERVICE_HOST
      value: 'mysql'
    - name: MYSQL_SERVICE_PORT
      value: '3306'
```

```
[root@master k8s]# kubectl create -f testpod.yaml
[root@master k8s]# kubectl get pod -o wide
NAME     READY    STATUS    RESTARTS   AGE      IP          NODE
myweb    1/1      Running   0          18m      10.244.1.4   node1
```

kind为pod表明这是个pod的定义
metadata里的name属性为pod的 名字，
metadata里还能定义资源对象的 label，这里声明myweb拥有一个name=myweb的 label。
pod里所包含的容器器组的定义则在spec 中声明，这里定义了一个名字为myweb、对应镜像为 kubeguide/tomcat-app:v1 的容器器， 该容器器注入了名为MYSQL_SERVICE_HOST='mysql'和 MYSQL_SERVICE_PORT='3306'的环境变量量（env），并且在8080 containerPort 上启动容器器进程

资源限制

每个pod都可以对其能使用的服务器器上的计算资源设置限额，当前可以 设置限额的计算资源有CPU与memory两种，其中cpu资源单位为 cpu core的数量，是一个绝对值而非相对值.

一个cpu的配额对于绝大多数容器器来说是相当大的一个资源配额，
所 以，在kubernetes里，通常以千分之一的cpu配额为最小单位，用m来 表示。通常一个容器器的cpu配额被定义为100～300m，即占用0.1～ 0.3个cpu。由于cpu配额是一个绝对值，所以无论在拥有一个core的机器器上，还是拥有48个core的机器器上，100m这个配额所代表的cpu的 使用量都是一样的。与cpu配额类似，memory配额也是一个绝对值，它 的单位是内存字节数。
在kubernetes里，一个计算资源进行配额限定需要设定以下两个参数

requests：该资源的最小申请量，系统必须满足要求，

limits：该资源最大允许使用的量，不能被突破，当容器器试图使 用超过这个量的资源时，可能会被kubernetes kill并重启

通常requests会设置为一个较小的数值，符合容器器平时的工作负载情 况下的资源需求，而把limits设置为峰值负载情况下资源占用的最大 量。比如下面定义，mysql容器器申请最少0.25个cpu及64MiB内存，在 运行过程中mysql容器器所能使用的资源配额为0.5 个cpu 以及 128MiB内存。

```
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Namespace

Namespace（命名空间）是 Kubernetes 系统中的另一个非常重要 的概念，Namespace 在很多情况下用于实现多租户的资源隔离。 Namespace 通过将集群内部的资源对象 “分配” 到不同的 Namespace 中，形成逻辑上分组的不同项目、小组或用户组，便于不 同的分组在共享使用整个集群的资源的同时还能被分别管理。
Kubernetes 集群在启动后，会创建一个名为 “default” 的 Namespace，通过 kubectl 可以查看到：

```
[root@master k8s]# kubectl get namespace
NAME          STATUS    AGE
default       Active    14d
kube-public   Active    14d
kube-system   Active    14d
```

接下来，如果不特别指明 Namespace，则用户创建的 Pod、RC、 Service 都将被系

统创建到这个默认的名为 default 的 Namespace 中。
Namespace 的定义很简单。如果所示的 yaml 定义名为development 的 Namespace。

```
[root@master k8s]# cat ns.ymal
apiVersion: v1
kind: Namespace
metadata:
  name: development

[root@master k8s]# kubectl create -f ns.ymal
namespace "development" created

[root@master k8s]# kubectl get namespace
NAME          STATUS    AGE
default       Active    14d
development   Active    44s
kube-public   Active    14d
kube-system   Active    14d
```

指定namespace创建pod

```
[root@master k8s]# cat testpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  namespace: development
  labels:
    name: myweb
spec:
  containers:
  - name: myweb
    image: httpd
    ports:
    - containerPort: 8080
    env:
    - name: MYSQL_SERVICE_HOST
      value: 'mysql'
    - name: MYSQL_SERVICE_PORT
      value: '3306'

[root@master k8s]# kubectl create -f testpod.yaml
pod "myweb" created

[root@master k8s]# kubectl get pod
No resources found.
[root@master k8s]# kubectl get pod --namespace=development
NAME      READY     STATUS              RESTARTS   AGE
```

myweb 0/1 ContainerCreating 0 19s

Replication Controller

Replication Controller （简称 RC）是 Kubernetes 系统中 的核心概念之一，简单来说，它其实是定义了一个期望的场景，即声明 某种 Pod 的副本数在任意时刻都符合某个预期值，所以 RC 的定义包 括如下几个部分。
Pod 的期待的副本数（replicas）。
用于筛选目标 Pod 的 Label Selector。
当 Pod 的副本数量小于预期数量时，用于创建新 Pod 的模板 （template）。
下面是一个完整的 RC 定义的例子，即确保拥有 app=nginx 标签的 这个 Pod（运行 tomcat ）在整个 Kuberntes 集群中始终只有三 个副本。

```
[root@master k8s]# cat nginx.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: nginx-demo
        image: nginx
        imagePullPolicy: IfNotPresent
        env:
        - name: GET_HOSTS_FROM
          value: dns
# If your cluster config does not include a dns service, then to     #
instead access environment variables to find service host
# info, comment out the ‘value: dns’ line above, and uncomment the
# line below.
# value: env
        ports:
        - containerPort: 80

[root@master k8s]# kubectl create -f nginx.yaml
replicationcontroller "frontend" created
[root@master k8s]# kubectl get rc
```

```
NAME      DESIRED   CURRENT   READY    AGE
frontend   2        2         0        10s
[root@master k8s]# kubectl get pods
NAME          READY    STATUS              RESTARTS   AGE
frontend-24khx   0/1      ContainerCreating   0         13s
frontend-sx92k   0/1      ContainerCreating   0         13s
[root@master k8s]# kubectl get pods -o wide
NAME          READY    STATUS              RESTARTS   AGE      IP
NODE
frontend-24khx   0/1      ContainerCreating   0         19s
<none>      node2
frontend-sx92k   1/1      Running             0         19s      10.244.1.5
node1
```

Replica Set

Replica Set，官方解释为“下一代的 RC”，它与 RC 当前存在的唯一区别是：Replica Sets 支持基于集合的 Label selector（Set-based selector），而 RC 只支持基于等式的 Label Selector（equality-based selector），这使得 Replica Set 的功能更强，下面是等价于之前 RC 例子的 Replica Set 的定义：

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
     matchLabels:
       tier: frontend
     matchExpressions:
       - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: nginx-demo
        image: nginx
        imagePullPolicy: IfNotPresent
        env:
        - name: GET_HOSTS_FROM
          value: dns
# If your cluster config does not include a dns service, then to     #
instead access environment variables to find service host
```

```
# info, comment out the 'value: dns' line above, and uncomment the
# line below.
# value: env
        ports:
        - containerPort: 80
```

kubectl 命令行工具适用于 RC 的绝大部分命令都同样适用于Replica Set。此外，当前我们很少单独使用 Replica Set，它主 要被 Deployment 这个更高级的资源对象所使用，从而形成一整套 Pod 创建、删除、更新的编排机制。当使用 Deplyment 时，无需关心它是如何创建和维护 Replica Set 的，这一切都是自动发生的。 Replica Set 与 Deployment 这两个重要资源对象逐步替换了之前 的 RC 的作用，是 Kubernetes v1.3 里Pod 自动扩容（伸缩） 这个告警功能实现的基础，也将继续在 Kubernetes 未来的版本中发 挥重要的作用。

Deployment

Deployment 是 Kubernetes v1.2 引入的新概念，引入的目的是 为了更好的解决 Pod 的编排问题。为此，Deployment 在内部使用 了Replica Set 来实现目的，无论从 Deployment 的作用与目 的、它的 YAML 定义，还是从它的具体命令操作来看，都可以把它看 做 RC 的一次升级两者的相似度超过 90%。
Deployment 相对于 RC 的大升级是可以随时知道当前 Pod "部 署" 的进度。实际上由于一个 Pod 的创建、调度、绑定节点以及在目标 Node 上启动对应的容器器这一完整过程需要一定的时间，所以期待 系统启动 N 个 Pod 副本的目标状态，实际上是一个连续变化的 "部 署过程" 导致的终状态。
Deployment 的典型使用场景有以下几个。
• 创建一个 Deployment 对象来生成对应的 Replica Set 并 完成 Pod 副本的创建过程。
• 检查 Deployment 的状态来部署动作是否完成（Pod 副本的数 量是否达到预期的值）。
• 更新 Deployment 以创建新的 Pod（比如镜像升级）。
• 如果当前 Deployment 不稳定，则回滚到一个早先的 Deployment 版本。
• 暂停 Deployment 以便于下一次性修改多个 PodTemplateSpec 的配置项，之后再回复 Deployment，进行新的发布。
• 扩展 Deployment 以应对高负载。
• 查看 Deployment 的状态，以此作为发布是否完成的指标。
• 清理不在需要的旧版本 ReplicaSets。
• Deployment 的定义与 Replica Set 的定义很类似，除了API 声 明与 Kind 类型等有所区别：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment

apiVersion: v1
kind: ReplicaSet
```

```
metadata:
  name: nginx-repset
```

创建：

```
[root@master k8s]# cat devel.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
    - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: nginx-dome
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080

[root@master k8s]# kubectl create -f devel.yaml
deployment.extensions "nginx-deployment" created



[root@master k8s]# kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   1         1         1            1           55s
```

对上述输出中涉及的数量解释如下： **DESIRED**：Pod 副本数量的期望值，即 Deployment 里定义的 Replica。

CURRENT：当前 Replica 的值，实际上是 Deployment 所创 建的 Replica Set 里的 Replica 值，这个值不断增加，直到 达到 **DESIRED** 为止，表名整个部署过程完成。

UP-TO-DATE：新版本的 Pod 的副本数量，用于只是在滚动升级的过程中，有多少个 Pod 副本已经成功升级。

AVAILABLE：当集群中可用的 Pod 副本数量，即集群中当前存 活的 Pod 数量。

运行下述命令查看对应的 Replica Set，看到它的命名与 Deployment 的 名字有关

系：

```
[root@master k8s]# kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-b5c8bd9bf   1         1         1         3m
```

运行下述命令查看创建的 Pod，发现 Pod 的命名与 Deployment 对应的 Replica Set 的名字为前缀，这种命名很清晰的表明了一个 Replica Set 创建了那些 Pod，对于 Pod 滚动升级这种复杂的过程来说，很容易排查 错误：

```
[root@master k8s]# kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-b5c8bd9bf-7b4gc   1/1       Running   0          10s
```

扩容

```
[root@master k8s]# kubectl scale deployment nginx-deployment --
replicas 2
deployment.extensions "nginx-deployment" scaled
[root@master k8s]# kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-b5c8bd9bf   2         2         1         6m
```

回退Deployment

```
[root@vlnx251101 test]# kubectl set image deployment/nginx-
deployment nginx=nginx:1.9.1
deployment.extensions/nginx-deployment image updated

[root@vlnx251101 test]# kubectl rollout status deployments
nginxdeployment Waiting for deployment "nginx-deployment" rollout
to finish: 1 old replicas are pending termination... ^C

[root@vlnx251101 test]kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-67594d6bf6   0         0         0         1m
nginx-deployment-6fdbb596db   1         1         1         36s

[root@vlnx251101 test]# kubectl get pods
NAME                                READY     STATUS              RESTARTS   AGE
nginx-deployment-67594d6bf6-xs8tq   1/1       Running             0
33s
nginx-deployment-6fdbb596db-vmpks   0/1       ContainerCreating
0          9s

[root@vlnx251101 test]# kubectl rollout history deployment/
nginxdeployment deployments "nginx-deployment"
```

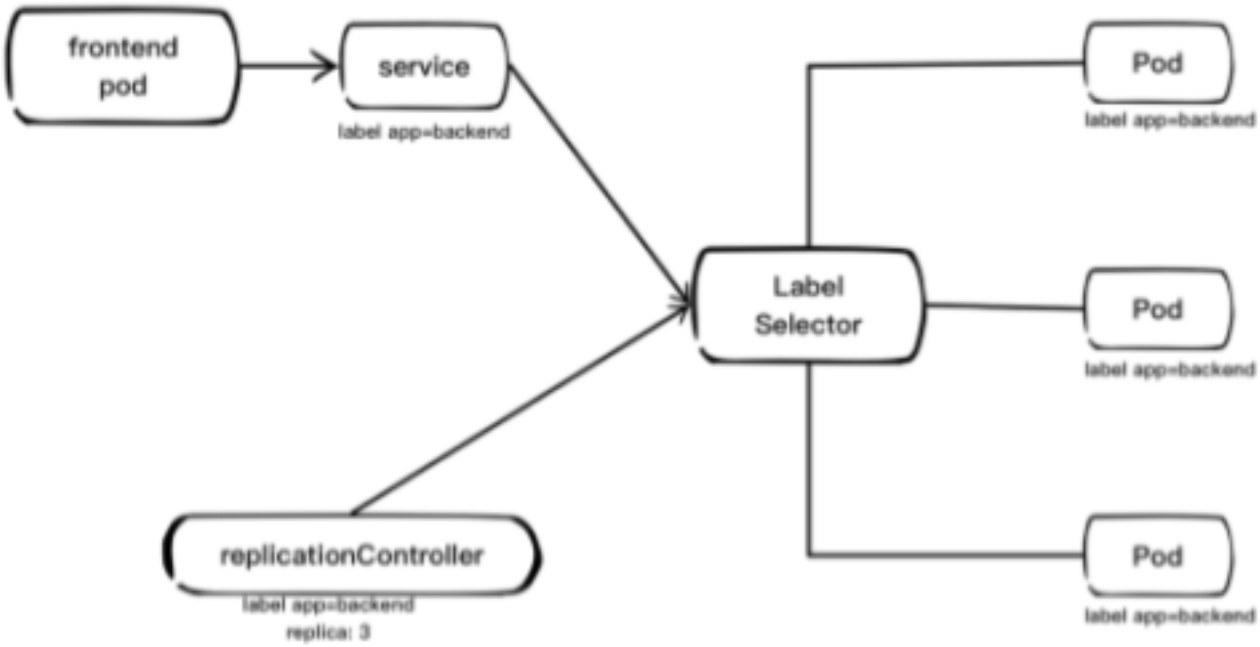
REVISION CHANGE-CAUSE
1   <none>
2   <none>

[root@vlnx251101 test]# kubectl rollout history deployment/
nginxdeployment --revision=1
deployments "nginx-deployment" with revision #1
Pod Template:
** Labels: app=nginx**
**   pod-template-hash=2315082692**
** Containers: nginx:**
** Image: nginx:1.7.9**
** Port: 80/TCP**
** Host Port: 0/TCP**
** Environment: <none>**
** Mounts: <none>**
** Volumes: <none>**

Service

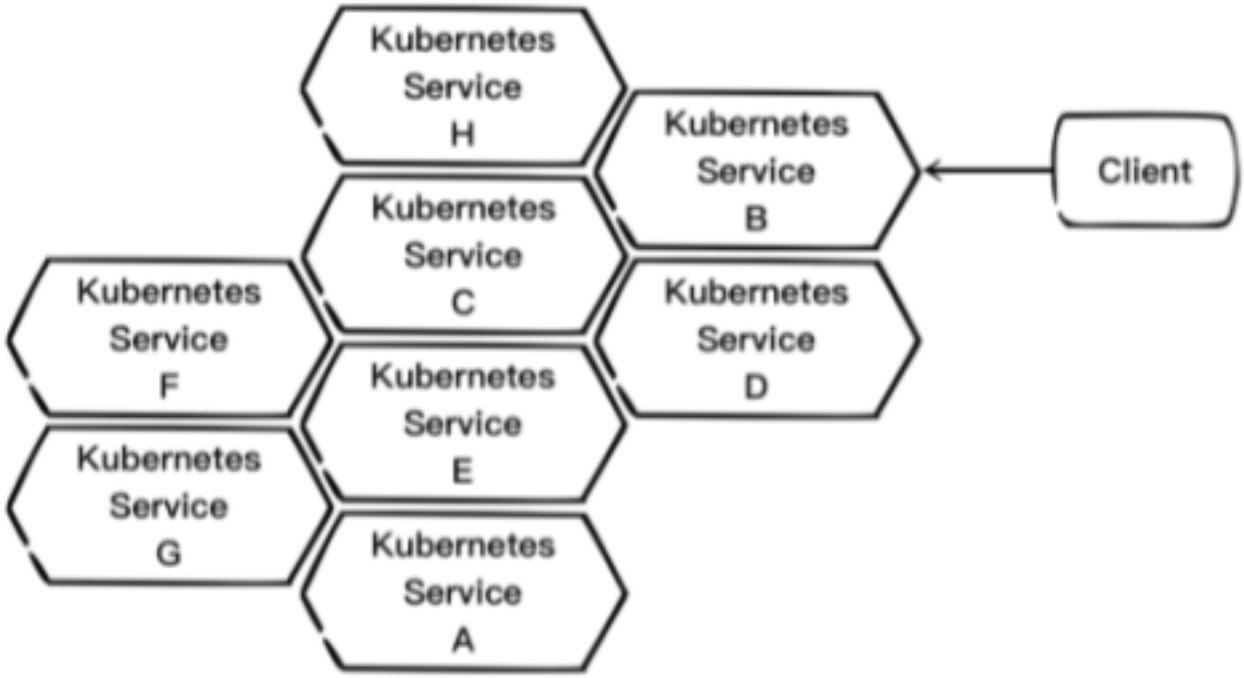
Kubernetes Pod 是有生命周期的，它们可以被创建，也可以被销毁，然 而一旦被销毁生命就永远结束。 通过 ReplicationController 能够动态地创建 和销毁 Pod（例如，需要进行扩缩容，或者执行 滚动升级）。 每个 Pod 都会获取它自己的 IP 地址，即使这些 IP 地址不总是稳定可依赖的。 这 会导致一个问题：在 Kubernetes 集群中，如果一组 Pod（称为 backend）为其它 Pod （称为 frontend）提供服务，那么那些 frontend 该如何发现，并连接到这组 Pod 中的哪些 backend 呢？

Service 也是 Kubernetes 里的就核心的资源对象之一， Kubernertes 里的每个 Service 其实就是我们经常提起的微服务 架构中的一个 “微服务”。 下图显示了pod、RC与Service的逻辑关系



Pod、RC 与 Service 的关系

从上图中可以看到，Kubernetes 的 Service 定义了一个服务的访问入口地址，前端的应用（Pod）通过这个入口地址访问其背后的一组 由 Pod 副本组成的集群实例，Service 与其后端 Pod 副本集群之 间则是通过 Label Selector 来实现”无缝对接“ 的。而 RC 的作 用实际上是保证 Service 的服务能力和服务质量始终处于预期的标 准。 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，与此同时，程序架构也变得简单和直观许多：



Kubernetes 所提供的微服务网络架构

既然每个 **Pod** 都会被分配一个单独的 **IP** 地址，而且每个 **Pod** 都提 供了一个独立的 **Endpoint**（**Pod IP + ContainerPort**）以被客 户端访问，现在多个 **Pod** 副本组成了一个集群来提供服务，那么客户 端如何来访问它们呢？一般的做法是部署一个负载均衡器器（软件或硬 件），为这组 **Pod** 开启一个对外的服务端口如 8000 端口，并且将 这些 **Pod** 的 **Endpoint** 列表加入 8000 端口的转发列表中，客户 端就可以通过负载均衡器器的对外 **IP** 地址 + 服务端口来访问此服务， 而客户端的请求最后会被转发到哪个 **Pod**，则由负载均衡器器的算法所决
定。
Kubernetes 也遵循了上述常规做法，运行在每个 **Node** 上的 **kube-proxy** 进程其实就是一个智能的软件负载均衡器器，它负责把对 **Service** 的请求转发到后端的某个 **Pod** 实际上，并在内部实现服务 的负载均衡与会话保持机制。但 **Kubernetes** 发明了一种很巧妙又影 响深远的设计：**Service** 不是共用一个负载均衡器器的 **IP** 地址，而是 每个 **Service** 分配一个全局唯一的虚拟 **IP** 地址，这个虚拟 **IP** 被 称为 **Cluster IP**。这样一来，每个服务就变成了具备唯一 **IP** 地址 的 ”通信节点“，服务调用就变成了最基础的 **TCP** 网络通信问题。 **Pod** 的 **Endpoint** 地址会随着 **Pod** 的销毁和重新创建而发生改变， 因为新 **Pod** 的 **IP** 地址与之前旧 **Pod** 的不同。而 **Service** 一旦 被创建，**Kubernetes** 就会自动为它分配一个可用的 **Cluster IP**， 而且在 **Service** 的整个生命周期内，它的 **Cluster IP** 不会发生 改变。于是，服务发现这个棘手的问题在 **Kubernetes** 的架构里也得 以轻松解决：只要用 **Service** 的 **Name** 与 **Service** 的 **Cluster IP** 地址做一个 **DNS** 域名映射即可完美解决问题。

下面创建一个 **Service**，来加深对它的理解。内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
```

```
  - port: 8080
  selector:
    tier: frontend

[root@master k8s]# kubectl create -f ser.yaml
service "nginx-service" created

[root@master k8s]# kubectl get endpoints
NAME            ENDPOINTS                       AGE
kubernetes      192.168.0.200:6443              14d
nginx-service   10.244.1.6:8080,10.244.2.6:8080   27s
```

这里查看的实际是 Pod 的真是 IP 地址，并不是 Cluster IP，运行下面的命令查看 nginx service 信息以及 Cluster IP 及更多的信息：

```
[root@master k8s]# kubectl get svc nginx-service -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2018-08-31T18:07:34Z
  name: nginx-service
  namespace: default
  resourceVersion: "32094"
  selfLink: /api/v1/namespaces/default/services/nginx-service
  uid: bc965f8f-ad48-11e8-b46b-000c29211aed
spec:
  clusterIP: 10.111.193.216
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    tier: frontend
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

管理volume

Volume

Volume 是 Pod 中能够被多个容器器访问的共享目录。Kubernetes 的 Volume 概念、用途和目的与 Docker 的 Volume 比较类似，但 两者不能等价。首先，Kubernetes 中的 Volume 定义在 Pod 上， 然后被一个 Pod 里的多个容器器挂载到具体的文件目录下；其次， Kubernetes 中的 Volume 与 Pod 的生命周期相同，但与容器器的声 明周期不相干关，当容器器中止或者启动时，Volume 中的数据也不会丢失。后，Kubernetes 支持多种类型的 Volume，例如 GlusterFS、Ceph 等先进的分布式文件存储。 Volume 的使用也比较简单，在大多数情况下，先在 Pod 上声明一个 Volume，然后在容器器里引用该 Volume 并 Mount 到容器器里的某个 目录上。举例来说，要给之前的 Tomcat Pod 增加一个名字为 datavol 的 Volume，并且 Mount 到容器器的 /mydata-data 目 录上，则只要对 Pod 的定义文件做如下修改即可：

创建

```
[root@master k8s]# cat test-volume.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: httpd-demo
        image: httpd
        volumeMounts:
          - mountPath: /mydata-data
            name: datavol
        imagePullPolicy: IfNotPresent
      volumes:
      - name: datavol
        emptyDir: {}

 [root@master k8s]# kubectl create -f test-volume.yaml
replicationcontroller "frontend" created
```

EmptyDir

一个 EmptyDir Volume 是在 Pod 分配到 Node 时创建的。从它 的名称就可以看出，它的初始化内容为空，并且无须指定宿主机上对应 的目录文件，因为这是 Kubernetes 自动分配的一个目录，当 Pod 从 Node 上移除时，emptyDir 中的数据也会被永久删除。
emptyDir 的一些用途如下。
临时空间，例如用于某些应用程序运行时所需的临时目录，且无 须永久保留。
长时间任务的中间过程 CheckPoint 的临时保存目录。
一个容器器需要从另一个容器器中获取数据的目录（多容器器共享目录）。
目前，用户无法控制 emptyDir 使用的介质种类。如果 kubelet 的 配置是使用硬盘，那么所有 emptyDir 都将创建在该硬盘上。Pod 在 将来可以设置 emptyDir 是位于硬盘、固态硬盘上还是基于内存的 tmpfs 上，上面的例子采用了 emptyDir 类的 Volume。

HostPath
hostPath 为在 Pod 上挂载宿主机上的文件或目录，它通常可以用于 以下几方面。 容器器应用程序生成的日志文件需要永久保存时，可以使用宿主机 的高速文件系统进行存储。
需要访问宿主机上 Docker 引擎内部数据结构的容器器应用时，可 以通过定义 hostPath 为宿主机 /var/lib/docker 目录，使 容器器内部应用可以直接访问 Docker 的文件系统。

在使用这种类型的 Volume，需要注意以下几点。

在不同的 Node 上具有相同配置的 Pod 可能会因为宿主机上的 目录和文件不同而导致对 Volume 上目录和文件的访问结果不一 致。
如果使用了资源配额管理，则 Kubernetes 无法将 hostPath 在宿主机上使用的资源纳入管理。

在上面的例子中使用宿主机的 /opt/volume 目录定义了一个 hostPath 类型的 Volume：

```
volumes:
- name: datavol
  hostPath:
      path: “/opt/volume/"
```

NFS 使用 NFS 网络文件系统提供的共享存储数据时，需要在系统中部署一 个NFS Server。定义 NFS 类型的 Volume 示例如下：

```
 …
 volumes:
 - name: nfs
   nfs:
   server: nfs-server.localhost # 自己 NFS 服务的地址
   path: "/"
```

Persistent Volume

Volume 是定义在 Pod 上的，属于 “计算资源” 的一部分，而实际 上，“网络存储” 是相对独立于 “计算资源” 而存在的一种实体资源。 比如在使用虚拟机的情况下，通常会先定义一个网络存储，然后从中划 出一个 “网盘” 并挂载到虚拟机上的。Persistent Volume（简称 PV）和与之相关联的 Persistent Volume Claim（简称 PVC）也 起到了类似的作用。
PV 可以理解成 Kubernetes 集群中的某个网络存储中相应的一块存储，它与 Volume 很类似，但有以下区别。
PV 只能是网络存储，不属于任何 Node，但可以在每个 Node 上访问。
PV 并不是定义在 Pod 上的，而是独立于 Pod 之外定义。
PV 目前支持的类型包括：gcePersistenDisk、 AWSElasticBlockStore、AzureFile、AzoreDisk、 FC(Fibre Channel)、Flocker、NFS、iSCSI、RBD（Rados Block Device）、CephFS、Chnder、GlusterFS、VsphereVolume、Quobyte Volumes、VMware Photon、 Portworx Volumes、ScaleIO Volumes 和 HostPath（仅供 单机模式）。
下 面给出了NFS 类型 PV 的一个 yaml 定义文件，声明了需要 5Gi 的存储空间：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
     storage: 5Gi
  accessModes:
     - ReadWriteOnce
  nfs:
    path: /somepath
    server: 172.17.0.2
```

比较重要的是 PV 的 accessModes 属性，目前有以下类型。
ReadWriteOnce：读写权限，并且只能被单个 Node 挂载。
ReadOnlyMany：只读权限，允许被多个 Node 挂载。
ReadWriteMany：读写权限，允许被多个 Node 挂载。

如果某个 Pod 想申请某种类型的 PV，则首先需要定义一个 PersistentVolumeClaim（PVC）对象

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: myclaim
spec:
```

```
  accessModes:
    - ReadWriteOnce
  resources:
      requests:
          storage: 5Gi
```

然后，在 Pod 的 Volume 定义中引用上述 PVC 即可：

```
...
volumes:
  - name: mypd
    persistentVolumeClaim:
        claimName: myclaim
```

然后，PV 是有状态的对象，它有以下几种状态

Available：空闲状态。

Bound：已经绑定到某个 PVC 上。

Released：对应的 PVC 已经删除，但资源还没有被集群收回。

Failed：PV 自动回收失败。

举例:

```
[root@master k8s]# vim test-volume.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv001
  labels:
     pv: pv001
spec:
  capacity:
      storage: 1Gi
  accessModes:
      - ReadWriteOnce
  nfs:
      path: /opt/nfsshare
      server: 192.168.251.103


kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
      - ReadWriteOnce
  resources:
    requests:
        storage: 1Gi
```

```
  selector:
    matchLabels:
      pv: pv001


apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: tomcat-demo
        image: tomcat
        volumeMounts:
          - mountPath: /mydata-data
            name: mypv
        imagePullPolicy: IfNotPresent
      volumes:
      - name: mypv
        persistentVolumeClaim:
          claimName: myclaim

[root@vlnx251101 volume]# kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM           STORAGECLASS   REASON    AGE pv001     1Gi
RWO           Retain          Bound     default/myclaim
1m


[root@vlnx251101 volume]# kubectl get pvc
NAME      STATUS    VOLUME    CAPACITY   ACCESS MODES
STORAGECLASS   AGE myclaim   Bound     pv001     1Gi
RWO                       1m


[root@vlnx251101 volume]# kubectl get pod
NAME           READY     STATUS    RESTARTS   AGE frontend-dsdrb
1/1       Running   0          2m
```

[root@vlnx251101 volume]# kubectl exec -it frontenddsdrb -- /bin/bash

root@frontend-dsdrb:/usr/local/tomcat# ls /mydata-data/
a.txt
root@frontend-dsdrb:/usr/local/tomcat# cat /mydatadata/a.txt
103