K8S 概览

1.1 K8S 是什么?

K8S官网文档：**https://kubernetes.io/zh/docs/home/**

K8S 是Kubernetes的全称，源于希腊语，意为“舵手”或“飞行员”，官方称其是：用于自动部署、扩展和管理“容器化（containerized）应用程序”的开源系统。翻译成大白话就是：“**K8S 是负责自动化运维管理多个 Docker 程序的集群**”。

1.2 K8S核心特性

- 服务发现与负载均衡：无需修改你的应用程序即可使用陌生的服务发现机制。
- 存储编排：自动挂载所选存储系统，包括本地存储。
- Secret和配置管理：部署更新Secrets和应用程序的配置时不必重新构建容器镜像，且不必将软件堆栈配置中的秘密信息暴露出来。
- 批量执行：除了服务之外，Kubernetes还可以管理你的批处理和CI工作负载，在期望时替换掉失效的容器。
- 水平扩缩：使用一个简单的命令、一个UI或基于CPU使用情况自动对应用程序进行扩缩。
- 自动化上线和回滚：Kubernetes会分步骤地将针对应用或其配置的更改上线，同时监视应用程序运行状况以确保你不会同时终止所有实例。
- 自动装箱：根据资源需求和其他约束自动放置容器，同时避免影响可用性。
- 自我修复：重新启动失败的容器，在节点死亡时替换并重新调度容器，杀死不响应用户定义的健康检查的容器。

1.3 K8S集群安装

搭建K8S集群，准备三台2核4G的虚拟机(内存至少2G以上)，操作系统选择用centos 7以上版本，先在三台机器上装好docker(安装参考docker课程，docker版本最好跟docker课上用的版本一致，防止与K8S的兼容性问题)：

在三台机器上都执行如下命令操作：

```
1、关闭防火墙
systemctl stop firewalld
systemctl disable firewalld

2、关闭 selinux
sed -i 's/enforcing/disabled/' /etc/selinux/config # 永久关闭
setenforce 0 # 临时关闭

3、关闭 swap
 swapoff -a # 临时关闭
 vim /etc/fstab # 永久关闭
 #注释掉swap这行
```

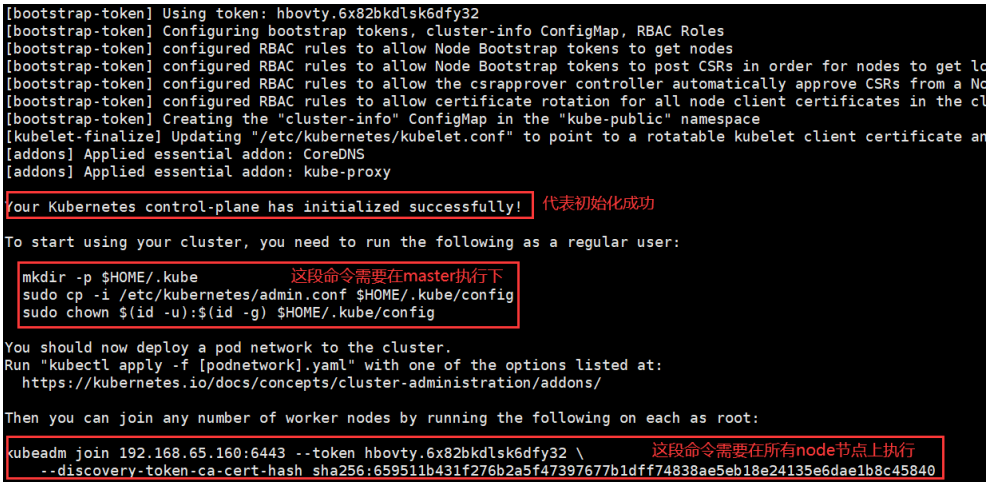
```
# /dev/mapper/centos-swap swap swap defaults 0 0

systemctl reboot #重启生效
free -m #查看下swap交换区是否都为0，如果都为0则swap关闭成功

4、给三台机器分别设置主机名
hostnamectl set-hostname <hostname>
第一台：k8s-master
第二台：k8s-node1
第三台：k8s-node2

5、在 k8s-master机器添加hosts，执行如下命令，ip需要修改成你自己机器的ip
cat >> /etc/hosts << EOF
192.168.65.160 k8s-master
192.168.65.203 k8s-node1
192.168.65.210 k8s-node2
EOF

6、将桥接的IPv4流量传递到iptables
cat > /etc/sysctl.d/k8s.conf << EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

sysctl --system # 生效

7、设置时间同步
yum install ntpdate -y
ntpdate time.windows.com

8、添加k8s yum源
cat > /etc/yum.repos.d/kubernetes.repo << EOF
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF

9、如果之前安装过k8s，先卸载旧版本
yum remove -y kubelet kubeadm kubectl

10、查看可以安装的版本
yum list kubelet --showduplicates | sort -r

11、安装kubelet、kubeadm、kubectl 指定版本，我们使用kubeadm方式安装k8s集群
yum install -y kubelet-1.18.0 kubeadm-1.18.0 kubectl-1.18.0

12、开机启动kubelet
```

```
systemctl enable kubelet
systemctl start kubelet
```

在k8s-master机器上执行初始化操作(里面的第一个ip地址就是k8s-master机器的ip，改成你自己机器的，后面两个ip网段不用动)

```
kubeadm init --apiserver-advertise-address=192.168.65.160 --image-repository registry.aliyuncs.com/google_containers --kubernetes-version v1.18.0 --service-cidr=10.96.0.0/12 --pod-network-cidr=10.244.0.0/16
```

执行完后结果如下图：



在k8s-master机器上执行如下命令：

```
#配置使用 kubectl 命令工具(类似docker这个命令)，执行上图第二个红框里的命令
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

#查看kubectl是否能正常使用
kubectl get nodes

#安装 Pod 网络插件
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
# 如果上面这个calico网络插件安装不成功可以试下下面这个
# kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kubeflannel.yml
```

在所有k8s node机器执行上图第三个红框里的命令

```
# 将node节点加入进master节点的集群里，复制上图第三个红框里的命令执行
kubeadm join 192.168.65.160:6443 --token hbovty.6x82bkdlsk6dfy32 \
 --discovery-token-ca-cert-hash sha256:659511b431f276b2a5f47397677b1dff74838ae5eb18e24135e6dae1b8c45840
```

在k8s-master机器执行查看节点命令

```
kubectl get nodes
```

```
[root@k8s-master local]# kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
k8s-master   Ready    master   90d   v1.18.0
k8s-node1    Ready    <none>   90d   v1.18.0
k8s-node2    Ready    <none>   90d   v1.18.0
```

刚刚安装的三个k8s节点都已经准备就绪，**大功告成**！

用K8S部署Nginx

在k8s-master机器上执行

```
# 创建一次deployment部署
kubectl create deployment nginx --image=nginx
kubectl expose deployment nginx --port=80 --type=NodePort
# 查看Nginx的pod和service信息
kubectl get pod,svc -o wide
```

```
[root@k8s-master ~]# kubectl get pod,svc -o wide
NAME                         READY   STATUS    RESTARTS   AGE    IP               NODE        NOMINATED NODE   READINESS GATES
pod/nginx-f89759699-ngqjl    1/1     Running   0          106s   10.244.169.130   k8s-node2   <none>           <none>

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE   SELECTOR
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        6d    <none>
service/nginx        NodePort    10.109.128.56   <none>        80:30433/TCP   22s   app=nginx
```

访问Nginx地址： http://任意节点的ip:图中Nginx的对外映射端口，http://192.168.65.203:30433

补充：如果node节点添加进集群失败，可以删除节点重新添加

要删除 k8s-node1 这个节点，首先在 master 节点上依次执行以下两个命令

```
kubectl drain k8s-node1 --delete-local-data --force --ignore-daemonsets
kubectl delete node k8s-node1
```

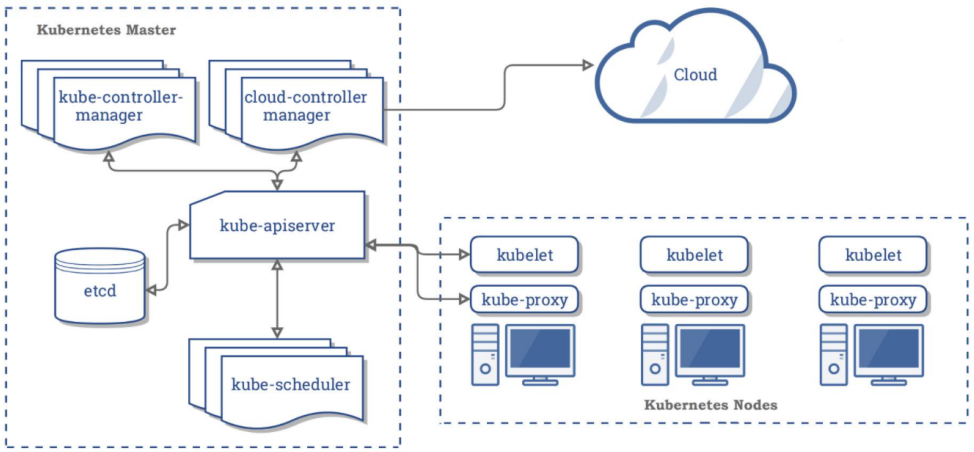
执行后通过 kubectl get node 命令可以看到 k8s-node1 已被成功删除

接着在 k8s-node1 这个 Node 节点上执行如下命令，这样该节点即完全从 k8s 集群中脱离开来，之后就可以重新执行命令添加到集群

```
kubeadm reset
```

1.4 K8S 核心架构原理

我们已经知道了 K8S 的核心功能：自动化运维管理多个容器化程序。那么 K8S 怎么做到的呢？这里，我们从宏观架构上来学习 K8S 的设计思想。首先看下图：



K8S 是属于**主从设备模型（Master-Slave 架构）**，即有 Master 节点负责核心的调度、管理和运维，Slave 节点则执行用户的程序。但是在 K8S 中，主节点一般被称为**Master Node 或者 Head Node**，而从节点则被称为

Worker Node 或者 Node。

注意：Master Node 和 Worker Node 是分别安装了 K8S 的 Master 和 Woker 组件的实体服务器，每个 Node 都对应了一台实体服务器（虽然 Master Node 可以和其中一个 Worker Node 安装在同一台服务器，但是建议 Master Node 单独部署），**所有 Master Node 和 Worker Node 组成了 K8S 集群**，同一个集群可能存在多个 Master Node 和 Worker Node。

首先来看**Master Node**都有哪些组件：

- **API Server。K8S 的请求入口服务。**API Server 负责接收 K8S 所有请求（来自 UI 界面或者 CLI 命令行工具），然后，API Server 根据用户的具体请求，去通知其他组件干活。
- **Scheduler。K8S 所有 Worker Node 的调度器。**当用户要部署服务时，Scheduler 会选择最合适的 Worker Node（服务器）来部署。
- **Controller Manager。K8S 所有 Worker Node 的监控器。**Controller Manager 有很多具体的 Controller， Node Controller、Service Controller、Volume Controller 等。Controller 负责监控和调整在 Worker Node 上部署的服务的状态，比如用户要求 A 服务部署 2 个副本，那么当其中一个服务挂了的时候，Controller 会马上调整，让 Scheduler 再选择一个 Worker Node 重新部署服务。
- **etcd。K8S 的存储服务。**etcd 存储了 K8S 的关键配置和用户配置，K8S 中仅 API Server 才具备读写权限，其他组件必须通过 API Server 的接口才能读写数据。

接着来看**Worker Node**的组件：

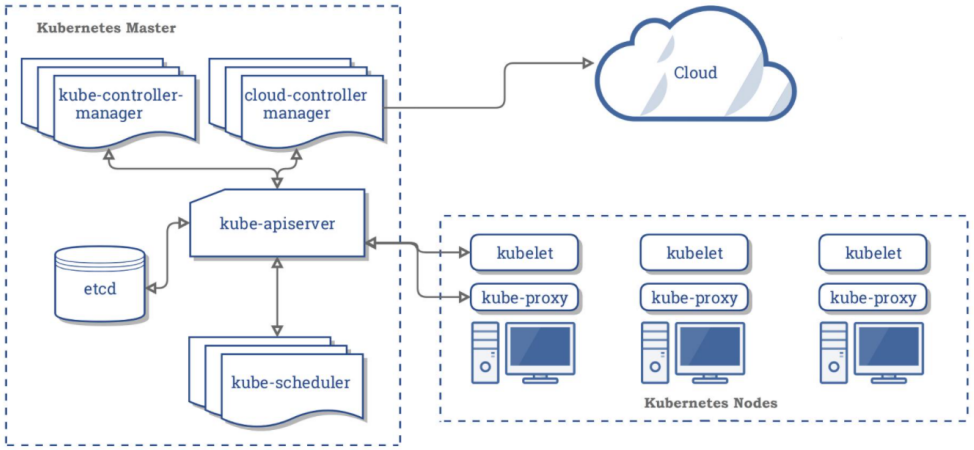
- **Kubelet。Worker Node 的监视器，以及与 Master Node 的通讯器。**Kubelet 是 Master Node 安插在 Worker Node 上的“眼线”，它会定期向 Master Node 汇报自己 Node 上运行的服务的状态，并接受来自 Master Node 的指示采取调整措施。负责控制所有容器的启动停止，保证节点工作正常。
- **Kube-Proxy。K8S 的网络代理。**Kube-Proxy 负责 Node 在 K8S 的网络通讯、以及对外部网络流量的负载均衡。
- **Container Runtime。Worker Node 的运行环境。**即安装了容器化所需的软件环境确保容器化程序能够跑起来，比如 Docker Engine运行环境。

在大概理解了上面几个组件的意思后，我们来看下上面用K8S部署Nginx的过程中，K8S内部各组件是如何协同工作的：

我们在master节点执行一条命令要master部署一个nginx应用(kubectl create deployment nginx --image=nginx)

- 这条命令首先发到master节点的网关api server，这是matser的唯一入口
- api server将命令请求交给controller mannager进行控制
- controller mannager 进行应用部署解析
- controller mannager 会生成一次部署信息，并通过api server将信息存入etcd存储中
- scheduler调度器通过api server从etcd存储中，拿到要部署的应用，开始调度看哪个节点有资源适合部署
- scheduler把计算出来的调度信息通过api server再放到etcd中
- 每一个node节点的监控组件kubelet，随时和master保持联系（给api-server发送请求不断获取最新数据），拿到master节点存储在etcd中的部署信息
- 假设node2的kubelet拿到部署信息，显示他自己节点要部署某某应用
- kubelet就自己run一个应用在当前机器上，并随时给master汇报当前应用的状态信息
- node和master也是通过master的api-server组件联系的

- 每一个机器上的kube-proxy能知道集群的所有网络，只要node访问别人或者别人访问node，node上的kube-proxy网络代理自动计算进行流量转发



K8S 快速实战

1、kubectl命令使用

kubectl是apiserver的客户端工具，工作在命令行下，能够连接apiserver实现各种增删改查等操作

kubectl官方使用文档：https://kubernetes.io/zh/docs/reference/kubectl/overview/

```
[root@k8s-master ~]# kubectl  -h
kubectl controls the Kubernetes cluster manager.

 Find more information at: https://kubernetes.io/docs/reference/kubectl/overview/

Basic Commands (Beginner):
  create        Create a resource from a file or from stdin.
  expose        使用 replication controller, service, deployment 或者 pod 并暴露它作为一个 新的 Kubernetes
Service
  run           在集群中运行一个指定的镜像
  set           为 objects 设置一个指定的特征

Basic Commands (Intermediate):
  explain       查看资源的文档
  get           显示一个或更多 resources
  edit          在服务器上编辑一个资源
  delete        Delete resources by filenames, stdin, resources and names, or by resources and label selector

Deploy Commands:
  rollout       Manage the rollout of a resource
  scale         Set a new size for a Deployment, ReplicaSet or Replication Controller
  autoscale     自动调整一个 Deployment, ReplicaSet, 或者 ReplicationController 的副本数量

Cluster Management Commands:
  certificate   修改 certificate 资源.
  cluster-info  显示集群信息
  top           Display Resource (CPU/Memory/Storage) usage.
  cordon        标记 node 为 unschedulable
  uncordon      标记 node 为 schedulable
  drain         Drain node in preparation for maintenance
```

K8S的各种命令帮助文档做得非常不错，遇到问题可以多查help帮助

2、创建一个Tomcat应用程序

使用 kubectl create deployment 命令可以创建一个应用部署**deployment**与**Pod**

```
#my-tomcat表示pod的名称 --image表示镜像的地址
kubectl create deployment my-tomcat --image=tomcat:7.0.75-alpine
```

```
[root@k8s-master ~]# kubectl create deployment my-tomcat --image=tomcat:7.0.75-alpine
deployment.apps/my-tomcat created
```

查看一下deployment的信息

```
kubectl get deployment
```

```
[root@k8s-master ~]# kubectl get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
my-tomcat   1/1     1            1           111s
nginx       1/1     1            1           102m
```

获取pod的信息，-o wide 表示更详细的显示信息

```
kubectl get pod -o wide
```

```
[root@k8s-master ~]# kubectl get pod -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP              NODE        NOMINATED NODE   READINESS GATES
my-tomcat-685b8fd9c9-rw42d  1/1     Running   0          2m42s   10.244.36.69    k8s-node1   <none>           <none>
nginx-f89759699-ngqjl       1/1     Running   0          102m    10.244.169.130  k8s-node2   <none>           <none>
```

查看Pod打印的日志

```
kubectl logs my-tomcat-685b8fd9c9-rw42d(pod名称)
```

使用 exec 可以在Pod的容器中执行命令，这里使用 env 命令查看环境变量

```
kubectl exec my-tomcat-685b8fd9c9-rw42d -- env
kubectl exec my-tomcat-685b8fd9c9-rw42d -- ls / # 查看容器的根目录下面内容
```

```
[root@k8s-master ~]# kubectl exec my-tomcat-685b8fd9c9-q6xzh -- env
PATH=/usr/local/tomcat/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
HOSTNAME=my-tomcat-685b8fd9c9-q6xzh
TOMCAT_SERVICE_PORT=8080
TOMCAT_PORT_8080_TCP_PORT=8080
TOMCAT_PORT_8080_TCP_ADDR=10.101.176.202
KUBERNETES_PORT_443_TCP_PORT=443
NGINX_SERVICE_PORT=80
NGINX_PORT_80_TCP_PROTO=tcp
NGINX_PORT=tcp://10.109.128.56:80
TOMCAT_SERVICE_HOST=10.101.176.202
TOMCAT_PORT=tcp://10.101.176.202:8080
TOMCAT_PORT_8080_TCP=tcp://10.101.176.202:8080
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
NGINX_SERVICE_HOST=10.109.128.56
NGINX_PORT_80_TCP=tcp://10.109.128.56:80
```

进入Pod容器内部并执行bash命令，如果想退出容器可以使用exit命令

```
kubectl exec -it my-tomcat-685b8fd9c9-rw42d -- sh
```

访问一下这个tomcat pod

集群内访问（在集群里任一worker节点都可以访问）

```
curl 10.244.36.69:8080
```

```
[root@k8s-master ~]# curl 10.244.36.69:8080

<!DOCTYPE html>


<html lang="en">
    <head>
        <title>Apache Tomcat/7.0.75</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>

    <body>
        <div id="wrapper">
            <div id="navigation" class="curved container">
                <span id="nav-home"><a href="http://tomcat.apache.org/">Home</a></span>
                <span id="nav-hosts"><a href="/docs/">Documentation</a></span>
                <span id="nav-config"><a href="/docs/config/">Configuration</a></span>
                <span id="nav-examples"><a href="/examples/">Examples</a></span>
                <span id="nav-wiki"><a href="http://wiki.apache.org/tomcat/FrontPage">Wiki</a></s
                <span id="nav-lists"><a href="http://tomcat.apache.org/lists.html">Mailing Lists<
                <span id="nav-help"><a href="http://tomcat.apache.org/findhelp.html">Find Help</a
                <br class="separator" />
            </div>
            <div id="asf-box">
                <h1>Apache Tomcat/7.0.75</h1>
            </div>
            <div id="upper" class="curved container">
                <div id="congrats" class="curved container">
                    <h2>If you're seeing this, you've successfully installed Tomcat. Congratulati
                </div>
```

集群外部访问

当我们在集群之外访问是发现无法访问，那么集群之外的客户端如何才能访问呢？这就需要我们的service服务了，下面我们就创建一个service，使外部客户端可以访问我们的pod

3、创建一个service

```
kubectl expose deployment my-tomcat --name=tomcat --port=8080 --type=NodePort
```

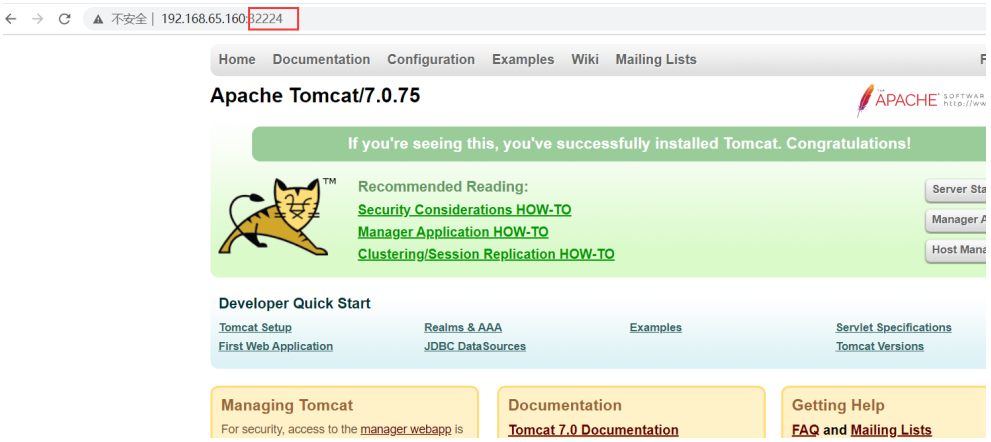
```
[root@k8s-master ~]# kubectl expose deployment my-tomcat  --name=tomcat   --port=8080  --type=NodePort
service/tomcat exposed
```

```
#查看service信息，port信息里冒号后面的端口号就是对集群外暴露的访问接口
kubectl get svc -o wide
```

```
[root@k8s-master ~]# kubectl get svc -o wide
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE    SELECTOR
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP          6d2h   <none>
nginx        NodePort    10.109.128.56    <none>        80:30433/TCP     115m   app=nginx
tomcat       NodePort    10.101.176.202   <none>        8080:32224/TCP   57s    app=my-tomcat
```

集群外部访问

使用集群worker节点的ip加上暴露的端口就可以访问



service服务有个特点，如果端口暴露类型为NodePort，那么可以通过集群内所有worker主机加暴露的端口进行访问

现在我们来删除刚刚添加的pod，看看会发生什么

```
#查看pod信息，-w意思是一直等待观察pod信息的变动
kubectl get pod -w
```

```
[root@k8s-master ~]# kubectl get pod -w
NAME                         READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-rw42d   1/1     Running   0          28m
nginx-f89759699-ngqjl        1/1     Running   0          129m
```

开另外一个命令窗口执行如下命令，同时观察之前命令窗口的变化情况

```
kubectl delete pod my-tomcat-685b8fd9c9-rw42d
```

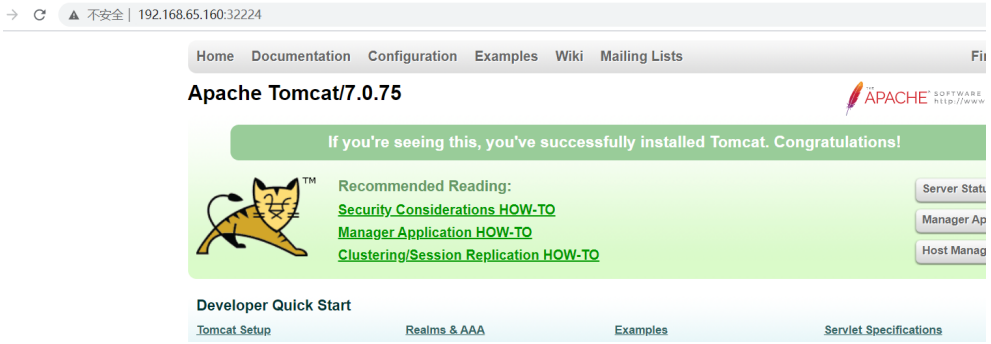
```
[root@k8s-master ~]# kubectl get pod -w
NAME                        READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-rw42d  1/1     Running   0          28m
nginx-f89759699-ngqjl       1/1     Running   0          129m

my-tomcat-685b8fd9c9-rw42d  1/1     Terminating   0          46m
my-tomcat-685b8fd9c9-6992h  0/1     Pending       0          0s
my-tomcat-685b8fd9c9-6992h  0/1     Pending       0          0s
my-tomcat-685b8fd9c9-6992h  0/1     ContainerCreating   0          0s
my-tomcat-685b8fd9c9-rw42d  1/1     Terminating         0          46m
my-tomcat-685b8fd9c9-6992h  0/1     ContainerCreating   0          1s
my-tomcat-685b8fd9c9-rw42d  0/1     Terminating         0          46m
my-tomcat-685b8fd9c9-rw42d  0/1     Terminating         0          46m
my-tomcat-685b8fd9c9-rw42d  0/1     Terminating         0          46m
my-tomcat-685b8fd9c9-6992h  1/1     Running             0          3s
```

我们可以看到之前那个tomcat的pod被销毁，但是又重新启动了一个新的tomcat pod，这是k8s的服务**自愈功能**，不需要运维人员干预

查看下deployment和service的状态

```
[root@k8s-master ~]# kubectl get deploy,svc
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/my-tomcat   1/1     1            1           49m
deployment.apps/nginx       1/1     1            1           149m

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP          6d2h
service/nginx        NodePort    10.109.128.56    <none>        80:30433/TCP     147m
service/tomcat       NodePort    10.101.176.202   <none>        8080:32224/TCP   33m
```

再一次访问service地址，依然可以访问成功



4、对my-tomcat这个deployment进行扩缩容

```
# 扩容到5个pod
kubectl scale --replicas=5 deployment my-tomcat
```

查看pod信息，发现已经有5个tomcat的pod

```
kubectl get pod
```

```
[root@k8s-master ~]# kubectl get pod
NAME                        READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-6992h  1/1     Running   0          10m
my-tomcat-685b8fd9c9-6fxf4  1/1     Running   0          60s
my-tomcat-685b8fd9c9-cv2zx  1/1     Running   0          60s
my-tomcat-685b8fd9c9-mnkh2  1/1     Running   0          60s
my-tomcat-685b8fd9c9-vjcwz  1/1     Running   0          60s
nginx-f89759699-ngqjl       1/1     Running   0          157m
```

缩容

```
# 扩容到3个pod
kubectl scale --replicas=3 deployment my-tomcat
```

5、滚动升级与回滚

对my-tomcat这个deployment进行滚动升级和回滚，将tomcat版本由tomcat:7.0.75-alpine升级到tomcat:8.0.41-jre8-alpine，再回滚到tomcat:7.0.75-alpine

滚动升级：

```
kubectl set image deployment my-tomcat tomcat=tomcat:8.0.41-jre8-alpine
```

可以执行 kubectl get pod -w 观察pod的变动情况，可以看到有的pod在销毁，有的pod在创建

```
[root@k8s-master ~]# kubectl get pod -w
NAME                        READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-6992h   1/1     Running   0          17m
my-tomcat-685b8fd9c9-mnkh2   1/1     Running   0          8m7s
my-tomcat-685b8fd9c9-vjcwz   1/1     Running   0          8m7s
nginx-f89759699-ngqjl        1/1     Running   0          164m
my-tomcat-547db86547-rtdrz   0/1     Pending   0          0s
my-tomcat-547db86547-rtdrz   0/1     Pending   0          0s
my-tomcat-547db86547-rtdrz   0/1     ContainerCreating   0          0s
my-tomcat-547db86547-rtdrz   0/1     ContainerCreating   0          1s
my-tomcat-547db86547-rtdrz   1/1     Running             0          37s
my-tomcat-685b8fd9c9-mnkh2   1/1     Terminating         0          12m
my-tomcat-547db86547-4btmd   0/1     Pending             0          0s
my-tomcat-547db86547-4btmd   0/1     Pending             0          0s
my-tomcat-547db86547-4btmd   0/1     ContainerCreating   0          0s
my-tomcat-685b8fd9c9-mnkh2   1/1     Terminating         0          12m
my-tomcat-547db86547-4btmd   0/1     ContainerCreating   0          1s
my-tomcat-685b8fd9c9-mnkh2   0/1     Terminating         0          12m
my-tomcat-685b8fd9c9-mnkh2   0/1     Terminating         0          12m
my-tomcat-685b8fd9c9-mnkh2   0/1     Terminating         0          12m
```

查看pod信息

```
kubectl get pod
```

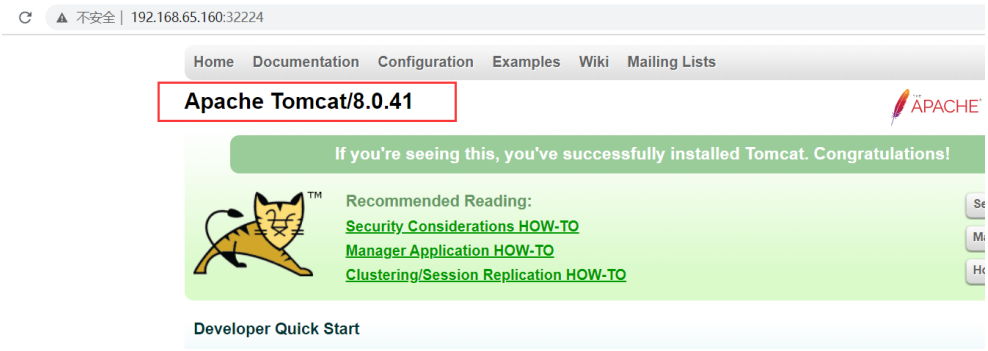
```
[root@k8s-master ~]# kubectl get pod
NAME                        READY   STATUS    RESTARTS   AGE
my-tomcat-547db86547-4btmd   1/1     Running   0          109s
my-tomcat-547db86547-fcfcv   1/1     Running   0          71s
my-tomcat-547db86547-rtdrz   1/1     Running   0          2m26s
nginx-f89759699-ngqjl        1/1     Running   0          170m
```

查看某个pod的详细信息，发现pod里的镜像版本已经升级了

```
kubectl describe pod my-tomcat-547db86547-4btmd
```

```
[root@k8s-master ~]# kubectl describe pod my-tomcat-547db86547-4btmd
Name:         my-tomcat-547db86547-4btmd
Namespace:    default
Priority:     0
Node:         k8s-node2/192.168.65.210
Start Time:   Wed, 26 May 2021 17:59:09 +0800
Labels:       app=my-tomcat
              pod-template-hash=547db86547
Annotations:  cni.projectcalico.org/podIP: 10.244.169.133/32
              cni.projectcalico.org/podIPs: 10.244.169.133/32
Status:       Running
IP:           10.244.169.133
IPs:
  IP:           10.244.169.133
Controlled By:  ReplicaSet/my-tomcat-547db86547
Containers:
  tomcat:
    Container ID:   docker://9b44a210d73c6ed27e32427c15cca9d2b3f15d8e7de511c8bea2849551a1c1c4
    Image:          tomcat:8.0.41-jre8-alpine
    Image ID:       docker-pullable://tomcat@sha256:17b2137b86c64013a03047e4c90b7dc63aebb7d1bd28641539d38ff00281ab9e
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 26 May 2021 17:59:47 +0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
```

访问下tomcat，看到版本也已经升级



版本回滚：

查看历史版本

```
kubectl rollout history deploy my-tomcat
```

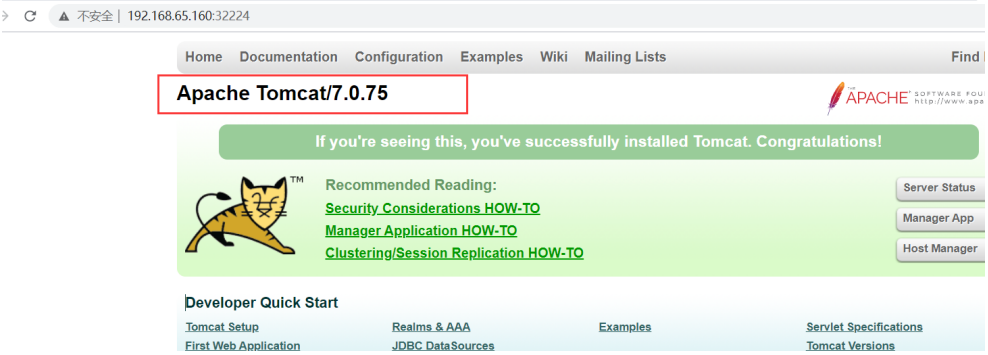
```
[root@k8s-master ~]# kubectl rollout history deploy my-tomcat
deployment.apps/my-tomcat
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

回滚到上一个版本

```
kubectl rollout undo deployment my-tomcat #--to-revision 参数可以指定回退的版本
```

```
[root@k8s-master ~]# kubectl rollout undo deployment my-tomcat
deployment.apps/my-tomcat rolled back
```

再次访问tomcat，发现版本已经回退



6、标签的使用

通过给资源添加Label，可以方便地管理资源（如Deployment、Pod、Service等）。

查看Deployment中所包含的Label

```
kubectl describe deployment my-tomcat
```

```
[root@k8s-master ~]# kubectl describe deployment my-tomcat
Name:                   my-tomcat
Namespace:              default
CreationTimestamp:      Wed, 26 May 2021 16:50:26 +0800
Labels:                 app=my-tomcat
Annotations:            deployment.kubernetes.io/revision: 3
Selector:               app=my-tomcat
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=my-tomcat
  Containers:
   tomcat:
    Image:        tomcat:7.0.75-alpine
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
```

通过Label查询Pod

```
kubectl get pods -l app=my-tomcat
```

```
[root@k8s-master ~]# kubectl get pods -l app=my-tomcat
NAME                              READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-4ngsb        1/1     Running   0          8d
my-tomcat-685b8fd9c9-lrwst        1/1     Running   0          8d
my-tomcat-685b8fd9c9-q6xzh        1/1     Running   0          8d
my-tomcat-yaml-685b8fd9c9-8glxt   1/1     Running   0          5d21h
my-tomcat-yaml-685b8fd9c9-ltbbf   1/1     Running   0          5d21h
```

通过Label查询Service

```
kubectl get services -l app=my-tomcat
```

```
[root@k8s-master ~]# kubectl get services -l app=my-tomcat
NAME               TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
tomcat             NodePort   10.101.176.202   <none>        8080:32224/TCP   8d
tomcat-service-yaml   NodePort   10.104.55.220    <none>        8080:31524/TCP   5d21h
```

给Pod添加Label

```
kubectl label pod my-tomcat-685b8fd9c9-lrwst version=v1
```

查看Pod的详细信息，可以查看Label信息：

```
kubectl describe pods my-tomcat-685b8fd9c9-lrwst
```

```
[root@k8s-master ~]# kubectl describe pods my-tomcat-685b8fd9c9-lrwst
Name:         my-tomcat-685b8fd9c9-lrwst
Namespace:    default
Priority:     0
Node:         k8s-node2/192.168.65.210
Start Time:   Wed, 26 May 2021 18:06:21 +0800
Labels:       app=my-tomcat
              pod-template-hash=685b8fd9c9
              version=v1
Annotations:  cni.projectcalico.org/podIP: 10.244.169.136/32
              cni.projectcalico.org/podIPs: 10.244.169.136/32
Status:       Running
IP:           10.244.169.136
IPs:
  IP:           10.244.169.136
Controlled By:  ReplicaSet/my-tomcat-685b8fd9c9
Containers:
```

通过Label查询Pod

```
kubectl get pods -l version=v1
```

```
[root@k8s-master ~]# kubectl get pods -l version=v1
NAME                              READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-lrwst        1/1     Running   0          8d
```

通过Label删除服务

```
kubectl delete service -l app=test-service
```

小总结：

```
kubectl create deployment #创建一个deployment来管理创建的容器
kubectl get #显示一个或多个资源，可以使用标签过滤，默认查看当前名称空间的资源
kubectl expose #将一个资源暴露为一个新的kubernetes的service资源，资源包括pod（po），service（svc），replicationcontroller（rc），deployment(deploy)，replicaset（rs）
kubectl describe #显示特定资源或资源组的详细信息
kubectl scale #可以对Deployment，ReplicaSet，Replication Controller，或者StatefulSet设置新的值，可以指定一个或多个先决条件
kubectl set #更改现有的应用程序资源
kubectl rollout #资源回滚管理
```

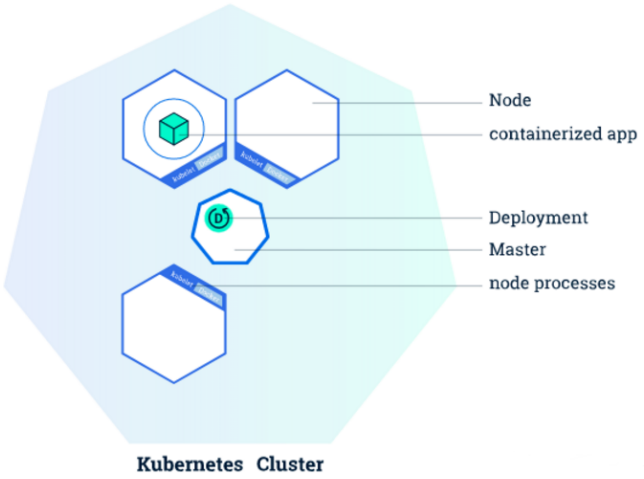
以上就是kubectl命令行下一些简单的操作，主要是让我们对kubernetes有一个快速的认识。

K8S 核心概念

Kubernetes有很多核心概念，我们先看下几个核心的概念。

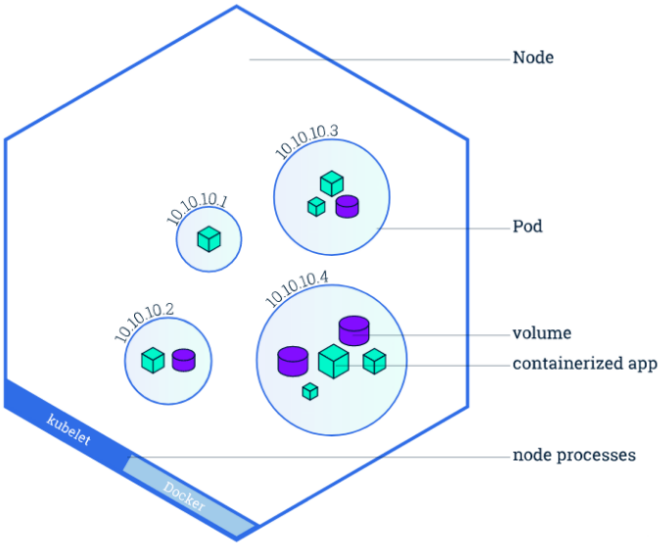
Deployment

Deployment负责创建和更新应用程序的实例。创建Deployment后，Kubernetes Master 将应用程序实例调度到集群中的各个节点上。如果托管实例的节点关闭或被删除，Deployment控制器会将该实例替换为群集中另一个节点上的实例。这提供了一种自我修复机制来解决机器故障维护问题。



Pod

Pod相当于**逻辑主机**的概念，负责托管应用实例。包括一个或多个应用程序容器（如 Docker），以及这些容器的一些共享资源（共享存储、网络、运行信息等）。



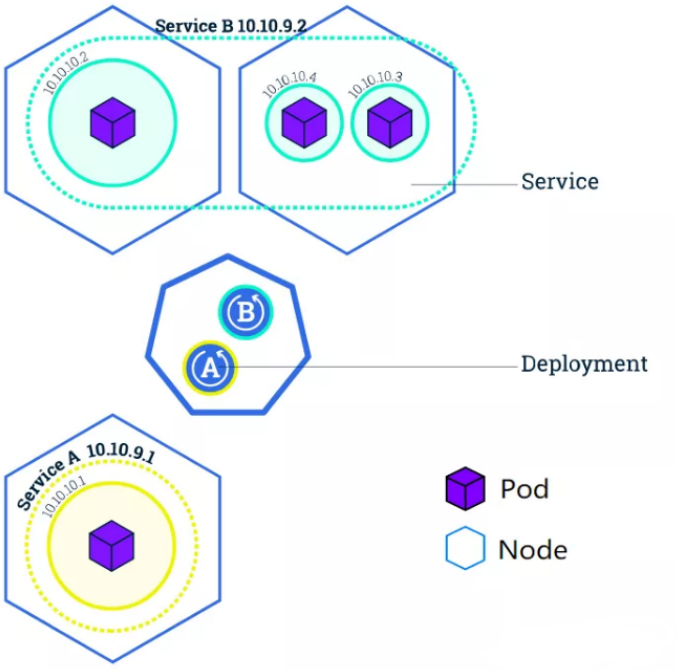
Service

Service是一个抽象层，它定义了一组Pod的逻辑集，并为这些Pod支持外部流量暴露、负载均衡和服务发现。

尽管每个Pod 都有一个唯一的IP地址，但是如果没有Service，这些IP不会暴露在群集外部。Service允许您的应用程序接收流量。Service也可以用在ServiceSpec标记type的方式暴露，type类型如下：

- ClusterIP（默认）：在集群的内部IP上公开Service。这种类型使得Service只能从集群内访问。
- NodePort：使用NAT在集群中每个选定Node的相同端口上公开Service。使用 **<NodeIP>:<NodePort>** 从集群外部访问Service。是ClusterIP的超集。

- LoadBalancer：在当前云中创建一个外部负载均衡器(如果支持的话)，并为Service分配一个固定的外部IP。是NodePort的超集。
- ExternalName：通过返回带有该名称的CNAME记录，使用任意名称（由spec中的externalName指定）公开Service。不使用代理。



k8s中的资源

k8s中所有的内容都抽象为资源， 资源实例化之后，叫做对象，上面说的那些核心概念都是k8s中的资源。

k8s中有哪些资源

- 工作负载型资源(workload)： Pod，ReplicaSet，Deployment，StatefulSet，DaemonSet等等
- 服务发现及负载均衡型资源(ServiceDiscovery LoadBalance)：Service，Ingress等等
- 配置与存储型资源： Volume(存储卷)，CSI(容器存储接口,可以扩展各种各样的第三方存储卷)
- 特殊类型的存储卷：ConfigMap(当配置中心来使用的资源类型)，Secret(保存敏感数据)，DownwardAPI(把外部环境中的信息输出给容器)

以上这些资源都是配置在名称空间级别

- 集群级资源：Namespace，Node，Role，ClusterRole，RoleBinding(角色绑定)，ClusterRoleBinding(集群角色绑定)
- 元数据型资源：HPA(Pod水平扩展)，PodTemplate(Pod模板,用于让控制器创建Pod时使用的模板)，LimitRange(用来定义硬件资源限制的)

资源清单

之前我们直接用命令创建deployment，pod，service这些资源，其实在k8s中，**我们一般都会使用yaml格式的文件来创建符合我们预期期望的资源**，这样的yaml文件我们一般称为资源清单

资源清单yaml的格式

```
apiVersion: group/apiversion # 如果没有给定group名称，那么默认为croe，可以使用kubectl api-versions获取当前k8s版本上所有的apiVersion版本信息(每个版本可能不同)
kind: #资源类别
metadata:  #资源元数据
```

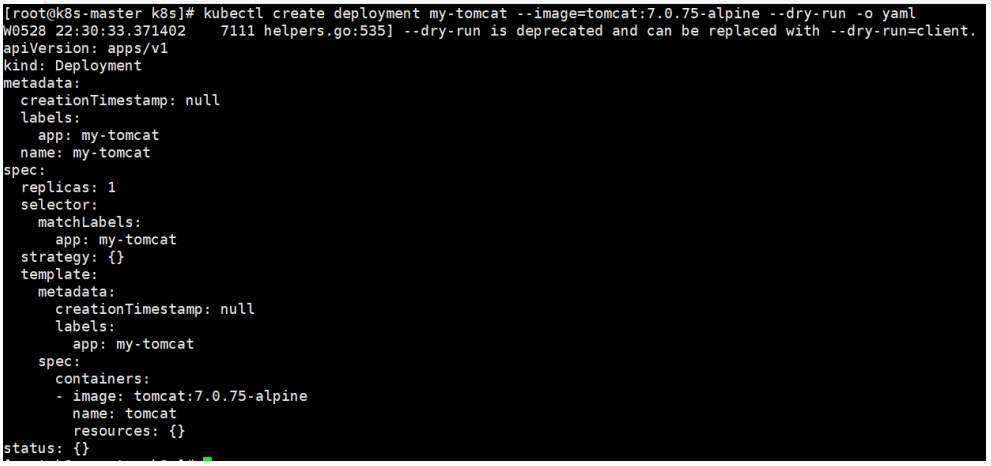
```
 name
 namespace #k8s自身的namespace
 lables
 annotations #主要目的是方便用户阅读查找
spec:期望的状态（disired state）
status：当前状态，本字段由kubernetes自身维护，用户不能去定义
#配置清单主要有五个一级字段，其中status字段用户不能定义，由k8s自身维护
```

使用资源清单yaml来创建k8s的资源对象

用yaml创建deployment资源的对象

我们可以用创建deployment的命令加上参数 --dry-run -o yaml 就可以输出这次部署的资源清单yaml

```
kubectl create deployment my-tomcat --image=tomcat:7.0.75-alpine --dry-run -o yaml
```



我们可以对上面的yaml适当的修改下保存为文件deployment-demo.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
 creationTimestamp: null
 labels:
 app: my-tomcat-yaml
 name: my-tomcat-yaml #修改deployment的名称
spec:
 replicas: 2 #修改pod副本为两个
  selector:
  matchLabels:
  app: my-tomcat-yaml
  strategy: {}
  template:
  metadata:
  creationTimestamp: null
  labels:
  app: my-tomcat-yaml
  spec:
  containers:
  - image: tomcat:7.0.75-alpine
  name: tomcat
  resources: {}
 status: {}
```

然后执行如下命令就可以用yaml文件来创建这次部署

```
kubectl apply -f deployment-demo.yaml
```

```
[root@k8s-master k8s]# kubectl get all
NAME                                    READY   STATUS    RESTARTS   AGE
pod/my-tomcat-685b8fd9c9-4ngsb          1/1     Running   0          2d4h
pod/my-tomcat-685b8fd9c9-lrwst          1/1     Running   0          2d4h
pod/my-tomcat-685b8fd9c9-q6xzh          1/1     Running   0          2d4h
pod/my-tomcat-yaml-685b8fd9c9-8glxt     1/1     Running   0          3s
pod/my-tomcat-yaml-685b8fd9c9-ltbbf     1/1     Running   0          3s
pod/nginx-f89759699-ngqjl               1/1     Running   0          2d7h

NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
service/kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP                      8d
service/nginx          NodePort    10.109.128.56    <none>        80:30433/TCP                 2d7h
service/test-service   NodePort    10.104.189.121   <none>        80:32080/TCP,8080:32088/TCP  85m
service/tomcat         NodePort    10.101.176.202   <none>        8080:32224/TCP               2d5h

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/my-tomcat         3/3     3            3           2d5h
deployment.apps/my-tomcat-yaml    2/2     2            2           3s
deployment.apps/nginx             1/1     1            1           2d7h

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/my-tomcat-547db86547         0         0         0       2d4h
replicaset.apps/my-tomcat-685b8fd9c9         3         3         3       2d5h
replicaset.apps/my-tomcat-yaml-685b8fd9c9    2         2         2       3s
replicaset.apps/nginx-f89759699              1         1         1       2d7h
```

从上图看出我们用yaml生成的部署已经成功。

用yaml创建service资源的对象

```
kubectl expose deployment my-tomcat --name=tomcat --port=8080 --type=NodePort --dry-run -o yaml
```

```
[root@k8s-master k8s]# kubectl expose deployment my-tomcat --name=tomcat --port=8080 --type=NodePort --dry-run -o yaml
W0528 22:54:38.067663    4726 helpers.go:535] --dry-run is deprecated and can be replaced with --dry-run=client.
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: my-tomcat
  name: tomcat
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: my-tomcat
  type: NodePort
status:
  loadBalancer: {}
```

修改下上面yaml内容，保存为文件：service-demo.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
 creationTimestamp: null
 labels:
 app: my-tomcat-yaml
 name: tomcat-service-yaml #修改service名称
spec:
 ports:
 - port: 80 # service的虚拟ip对应的端口，在集群内网机器可以访问用service的虚拟ip加该端口号访问服务
 protocol: TCP
 targetPort: 8080 # pod暴露的端口，一般与pod内部容器暴露的端口一致
 selector:
 app: my-tomcat-yaml
 type: NodePort
status:
 loadBalancer: {}
```

然后执行命令如下命令就可以用yaml文件来创建service

```
kubectl apply -f service-demo.yaml
```

```
[root@k8s-master k8s]# kubectl get svc
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
eureka-app-service    NodePort    10.106.87.91     <none>        8761:30558/TCP               26d
kubernetes            ClusterIP   10.96.0.1        <none>        443/TCP                      43d
nginx                 NodePort    10.109.128.56    <none>        80:30433/TCP                 37d
test-service          NodePort    10.104.189.121   <none>        80:32080/TCP,8080:32088/TCP  35d
tomcat                NodePort    10.101.176.202   <none>        8080:32224/TCP               37d
tomcat-service-yaml   NodePort    10.104.55.220    <none>        80:31368/TCP                 34d
```

从上图看出我们用yaml创建的service已经成功。

针对已有资源输出资源清单yaml

查看pod资源列表

```
[root@k8s-master k8s]# kubectl get pod
NAME                        READY   STATUS    RESTARTS   AGE
my-tomcat-685b8fd9c9-4ngsb  1/1     Running   0          2d3h
my-tomcat-685b8fd9c9-lrwst  1/1     Running   0          2d3h
my-tomcat-685b8fd9c9-q6xzh  1/1     Running   0          2d3h
nginx-f89759699-ngqjl       1/1     Running   0          2d6h
pod-demo                    2/2     Running   0          6m36s
```

将资源的配置以yaml的格式输出出来

```
#使用 -o 参数加yaml，可以将资源的配置以yaml的格式输出出来，也可以使用json，输出为json格式
kubectl get pod nginx-deploy-7db697dfbd-2qh7v -o yaml
```

```
[root@k8s-master k8s]# kubectl get pod nginx-f89759699-ngqjl -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cni.projectcalico.org/podIP: 10.244.169.130/32
    cni.projectcalico.org/podIPs: 10.244.169.130/32
  creationTimestamp: "2021-05-26T07:10:13Z"
  generateName: nginx-f89759699-
  labels:
    app: nginx
    pod-template-hash: f89759699
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:generateName: {}
        f:labels:
          .: {}
          f:app: {}
          f:pod-template-hash: {}
        f:ownerReferences:
          .: {}
          k:{"uid":"75b30eb6-fb58-40ef-a637-3062d6057768"}:
            .: {}
            f:apiVersion: {}
            f:blockOwnerDeletion: {}
            f:controller: {}
```

K8S 高级特性

K8S中还有一些高级特性有必要学习下，比如**弹性扩缩应用(见上文)、滚动更新(见上文)、配置管理、存储卷、网关路由**等。

在学习这些高级特性之前有必要再看几个K8S的核心概念：

ReplicaSet

ReplicaSet确保任何时间都有指定数量的Pod副本在运行。通常用来保证给定数量的、完全相同的Pod的可用性。建议使用Deployment来管理ReplicaSet，而不是直接使用ReplicaSet。

ConfigMap

ConfigMap是一种API对象，用来将非机密性的数据保存到键值对中。使用时，Pod可以将其用作环境变量、命令行参数或者存储卷中的配置文件。使用ConfigMap可以将你的配置数据和应用程序代码分开。

Volume

Volume指的是存储卷，包含可被Pod中容器访问的数据目录。容器中的文件在磁盘上是临时存放的，当容器崩溃时文件会丢失，同时无法在多个Pod中共享文件，通过使用存储卷可以解决这两个问题。

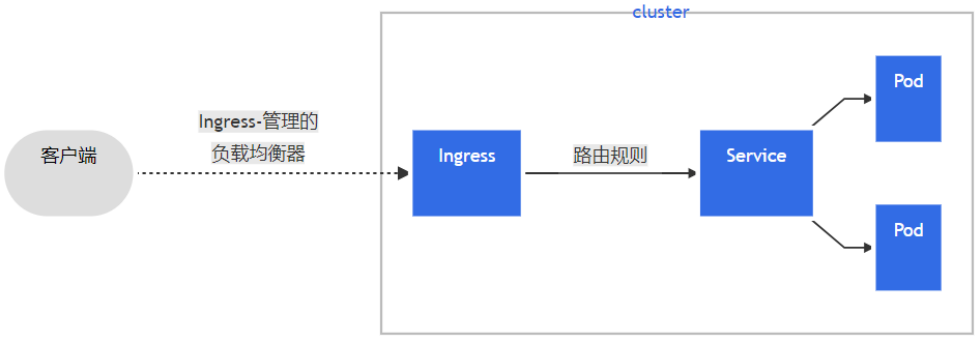
常用的存储卷有如下几种：

- configMap：configMap卷提供了向Pod注入配置数据的方法。ConfigMap对象中存储的数据可以被configMap类型的卷引用，然后被Pod中运行的容器化应用使用。

- emptyDir：emptyDir卷可用于存储缓存数据。当Pod分派到某个Node上时，emptyDir卷会被创建，并且Pod在该节点上运行期间，卷一直存在。当Pod被从节点上删除时emptyDir卷中的数据也会被永久删除。
- hostPath：hostPath卷能将主机节点文件系统上的文件或目录挂载到你的Pod中。在Minikube中的主机指的是Minikube所在虚拟机。
- local：local卷所代表的是某个被挂载的本地存储设备，例如磁盘、分区或者目录。local卷只能用作静态创建的持久卷，尚不支持动态配置。
- nfs：nfs卷能将NFS（网络文件系统）挂载到你的Pod中。
- persistentVolumeClaim：persistentVolumeClaim卷用来将持久卷（PersistentVolume）挂载到Pod中。持久卷（PV）是集群中的一块存储，可以由管理员事先供应，或者使用存储类（Storage Class）来动态供应，持久卷是集群资源类似于节点。

Ingress

通过K8S的Ingress资源可以实现类似Nginx的基于域名访问，从而实现Pod的负载均衡访问。



安装Ingress

进入页面https://github.com/kubernetes/ingress-nginx/blob/nginx-0.20.0/deploy/mandatory.yaml，将里面内容复制，保存到k8s master机器上的一个文件ingress-controller.yaml里，里面的镜像地址需要修改下，大家直接用我下面这个yaml的内容

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx

---

kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
  namespace: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx

---
kind: ConfigMap
apiVersion: v1
metadata:
```

```yaml
  name: tcp-services
  namespace: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx

---
kind: ConfigMap
apiVersion: v1
metadata:
  name: udp-services
  namespace: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress-serviceaccount
  namespace: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: nginx-ingress-clusterrole
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
rules:
  - apiGroups:
  - ""
  resources:
  - configmaps
  - endpoints
  - nodes
  - pods
  - secrets
  verbs:
  - list
  - watch
  - apiGroups:
  - ""
  resources:
  - nodes
  verbs:
```

```
  - get
  - apiGroups:
  - ""
  resources:
  - services
  verbs:
  - get
  - list
  - watch
  - apiGroups:
  - "extensions"
  resources:
  - ingresses
  verbs:
  - get
  - list
  - watch
  - apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
  - patch
  - apiGroups:
   - "extensions"
   resources:
   - ingresses/status
   verbs:
   - update

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
 name: nginx-ingress-role
 namespace: ingress-nginx
 labels:
 app.kubernetes.io/name: ingress-nginx
 app.kubernetes.io/part-of: ingress-nginx
rules:
 - apiGroups:
 - ""
 resources:
 - configmaps
 - pods
 - secrets
 - namespaces
 verbs:
 - get
 - apiGroups:
```

```
  - ""
  resources:
  - configmaps
  resourceNames:
  # Defaults to "<election-id>-<ingress-class>"
  # Here: "<ingress-controller-leader>-<nginx>"
  # This has to be adapted if you change either parameter
  # when launching the nginx-ingress-controller.
  - "ingress-controller-leader-nginx"
  verbs:
  - get
  - update
  - apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - create
  - apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - get

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: nginx-ingress-role-nisa-binding
  namespace: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: nginx-ingress-role
subjects:
  - kind: ServiceAccount
  name: nginx-ingress-serviceaccount
  namespace: ingress-nginx

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: nginx-ingress-clusterrole-nisa-binding
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
```

```yaml
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: nginx-ingress-clusterrole
subjects:
  - kind: ServiceAccount
  name: nginx-ingress-serviceaccount
  namespace: ingress-nginx

---

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
spec:
  selector:
  matchLabels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
  template:
  metadata:
  labels:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
  annotations:
  prometheus.io/port: "10254"
  prometheus.io/scrape: "true"
  spec:
  hostNetwork: true
  serviceAccountName: nginx-ingress-serviceaccount
  containers:
  - name: nginx-ingress-controller
  image: siriuszg/nginx-ingress-controller:0.20.0
  args:
  - /nginx-ingress-controller
  - --configmap=$(POD_NAMESPACE)/nginx-configuration
  - --tcp-services-configmap=$(POD_NAMESPACE)/tcp-services
  - --udp-services-configmap=$(POD_NAMESPACE)/udp-services
  - --publish-service=$(POD_NAMESPACE)/ingress-nginx
  - --annotations-prefix=nginx.ingress.kubernetes.io
  securityContext:
  allowPrivilegeEscalation: true
  capabilities:
  drop:
  - ALL
  add:
```

```
  - NET_BIND_SERVICE
  # www-data -> 33
  runAsUser: 33
  env:
  - name: POD_NAME
  valueFrom:
  fieldRef:
  fieldPath: metadata.name
  - name: POD_NAMESPACE
  valueFrom:
  fieldRef:
  fieldPath: metadata.namespace
  ports:
  - name: http
  containerPort: 80
  - name: https
  containerPort: 443
  livenessProbe:
  failureThreshold: 3
  httpGet:
  path: /healthz
  port: 10254
  scheme: HTTP
  initialDelaySeconds: 10
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 10
  readinessProbe:
  failureThreshold: 3
  httpGet:
  path: /healthz
  port: 10254
  scheme: HTTP
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 10

---
apiVersion: v1
kind: Service
metadata:
 name: ingress-nginx
 namespace: ingress-nginx
spec:
 #type: NodePort
 ports:
 - name: http
 port: 80
 targetPort: 80
 protocol: TCP
 - name: https
```

```
  port: 443
  targetPort: 443
  protocol: TCP
  selector:
  app.kubernetes.io/name: ingress-nginx
  app.kubernetes.io/part-of: ingress-nginx
```

安装ingress，执行如下命令

```
kubectl apply -f ingress-controller.yaml
```

查看是否安装成功

```
kubectl get pods -n ingress-nginx -o wide
```

```
[root@k8s-master k8s]# kubectl get pods -n ingress-nginx -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP              NODE        NOMINATED NODE   READINESS GATES
nginx-ingress-controller-jrkjp   1/1     Running   5          3h12m   192.168.65.203   k8s-node1   <none>           <none>
nginx-ingress-controller-v8th6   1/1     Running   2          3h12m   192.168.65.210   k8s-node2   <none>           <none>
```

配置ingress访问规则（就是类似配置nginx的代理转发配置），让ingress将域名tomcat.tuling.com转发给后端的tomcat-service-yaml 服务，新建一个文件ingress-tomcat.yaml，内容如下：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
 name: web-ingress
spec:
 rules:
 - host: tomcat.tuling.com #转发域名
 http:
 paths:
 - path: /
 backend:
 serviceName: tomcat-service-yaml
 servicePort: 80 #service的端口
```

执行如下命令生效规则：

```
kubectl apply -f ingress-tomcat.yaml
```

查看生效的ingress规则：

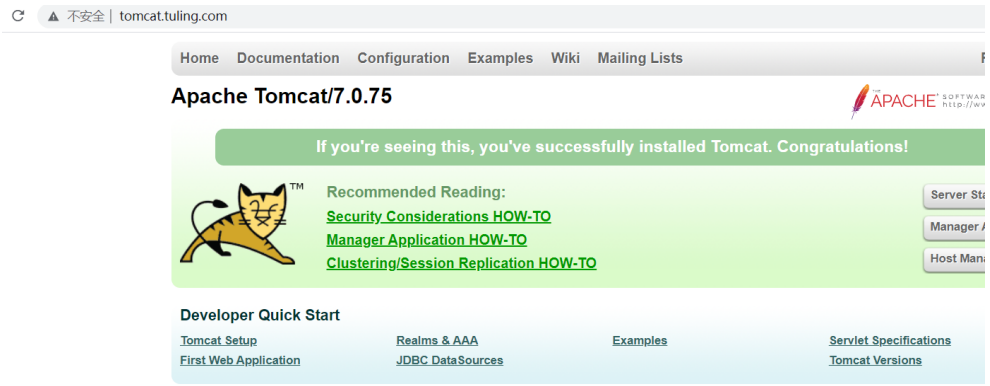
```
kubectl get ing
```

```
[root@k8s-master k8s]# kubectl get ing
NAME          CLASS    HOSTS              ADDRESS   PORTS   AGE
web-ingress   <none>   tomcat.tuling.com             80      3h8m
```

在访问机器配置host，win10客户机在目录：C:\Windows\System32\drivers\etc，在host里增加如下host(ingress部署的机器ip对应访问的域名)

```
192.168.65.203 tomcat.tuling.com
或者
192.168.65.210 tomcat.tuling.com
```

配置完后直接在客户机浏览器访问http://tomcat.tuling.com/ ，能正常访问tomcat。



配置管理

ConfigMap允许你将配置文件与镜像文件分离，以使容器化的应用程序具有可移植性。接下来我们演示下如何将ConfigMap的的属性注入到Pod的环境变量中去。

- 添加配置文件nginx-config.yaml用于创建ConfigMap，ConfigMap名称为nginx-config，配置信息存放在data节点下：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  namespace: default
data:
  nginx-env: test
```

- 应用 nginx-config.yaml 文件创建ConfigMap：

```
kubectl create -f nginx-config.yaml
```

- 获取所有ConfigMap：

```
kubectl get configmap
NAME DATA AGE
nginx-config 1 7s
```

- 通过yaml格式查看ConfigMap中的内容：

```
kubectl get configmaps nginx-config -o yaml
```

```
[root@k8s-master k8s]# kubectl get configmaps nginx-config -o yaml
apiVersion: v1
data:
  nginx-env: test
kind: ConfigMap
metadata:
  creationTimestamp: "2021-06-03T14:13:58Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:nginx-env: {}
    manager: kubectl
    operation: Update
    time: "2021-06-03T14:13:58Z"
  name: nginx-config
  namespace: default
  resourceVersion: "2965728"
  selfLink: /api/v1/namespaces/default/configmaps/nginx-config
  uid: 49042f56-38c6-4d2c-ac5e-e7fba34ec9a4
```

- 添加配置文件 nginx-deployment.yaml 用于创建Deployment，部署一个Nginx服务，在Nginx的环境变量中引用ConfigMap中的属性：

```yaml
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - containerPort: 80
          env:
            - name: NGINX_ENV # 在Nginx中设置环境变量
              valueFrom:
                configMapKeyRef:
                  name: nginx-config # 设置ConfigMap的名称
                  key: nginx-env # 需要取值的键
```

- 应用配置文件文件创建Deployment：

```
kubectl apply -f nginx-deployment.yaml
```

- 创建成功后查看Pod中的环境变量，发现NGINX_ENV变量已经被注入了；

```
kubectl exec deployments/nginx-deployment -- env
......
NGINX_ENV=test
```

存储卷使用

通过存储卷，我们可以把外部数据挂载到容器中去，供容器中的应用访问，这样就算容器崩溃了，数据依然可以存在。

- 之前我们使用Docker部署软件时是可以挂载文件的

```
docker run -p 80:80 --name nginx \
-v /mydata/nginx/html:/usr/share/nginx/html \
-v /mydata/nginx/logs:/var/log/nginx  \
-v /mydata/nginx/conf:/etc/nginx \
-d nginx:1.10
```

- K8S也可以挂载文件，添加配置文件nginx-volume-deployment.yaml用于创建Deployment：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-volume-deployment
  labels:
    app: nginx
```

```yaml
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - containerPort: 80
          volumeMounts:
            - mountPath: /usr/share/nginx/html
              name: html-volume
            - mountPath: /var/log/nginx
              name: logs-volume
            - mountPath: /etc/nginx
              name: conf-volume
      volumes:
        - name: html-volume
          hostPath:
            path: /home/docker/mydata/nginx/html
            type: Directory
        - name: logs-volume
          hostPath:
            path: /home/docker/mydata/nginx/logs
            type: Directory
        - name: conf-volume
          hostPath:
            path: /home/docker/mydata/nginx/conf
            type: Directory
```

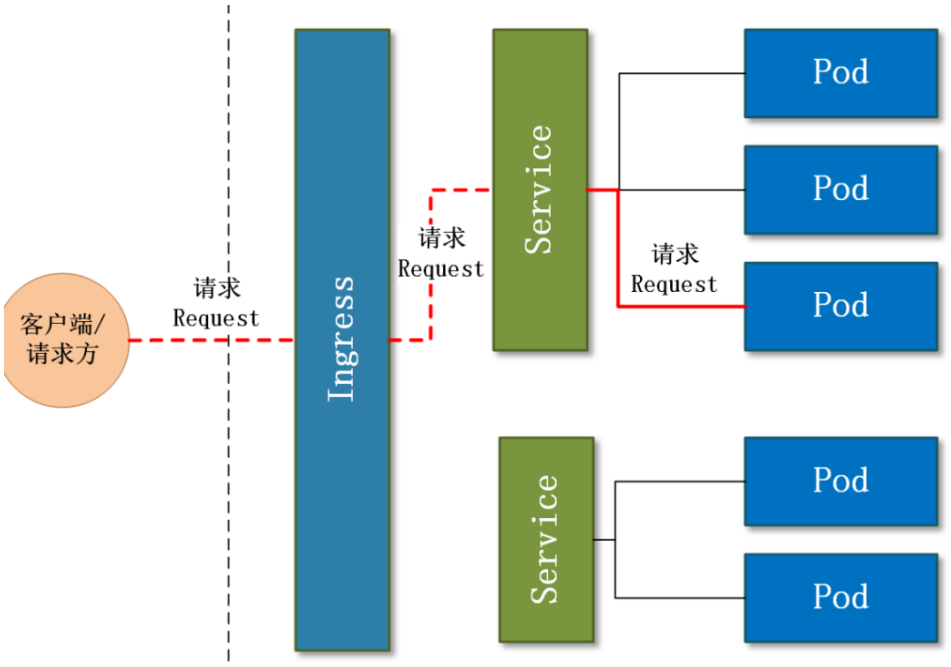
总结

Service 是 K8S 服务的核心，屏蔽了服务细节，统一对外暴露服务接口，真正做到了“微服务”。举个例子，我们的一个服务 A，部署了 3 个备份，也就是 3 个 Pod；对于用户来说，只需要关注一个 Service 的入口就可以，而不需要操心究竟应该请求哪一个 Pod。优势非常明显：**一方面外部用户不需要感知因为 Pod 上服务的意外崩溃、K8S 重新拉起 Pod 而造成的 IP 变更，外部用户也不需要感知因升级、变更服务带来的 Pod 替换而造成的 IP 变化，另一方面，Service 还可以做流量负载均衡。**

但是，Service 主要负责 K8S 集群**内部的网络拓扑。**集群外部需要用 Ingress 。

Ingress 是整个 K8S 集群的接入层，复杂集群内外通讯。

Ingress 和 Service 的网络拓扑关系图如下：



kubectl 排查服务问题

K8S 上部署服务失败了怎么排查?

用这个命令:

```
kubectl describe ${RESOURCE} ${NAME}
```

拉到最后看到Events部分，会显示出 K8S 在部署这个服务过程的关键日志。

一般来说，通过kubectl describe pod ${POD_NAME}已经能定位绝大部分部署失败的问题了，当然，具体问题还是得具体分析。

K8S 上部署的服务不正常怎么排查?

如果服务部署成功了，且状态为running，那么就需要进入 Pod 内部的容器去查看自己的服务日志了:

- 查看 Pod 内部容器打印的日志:

```
kubectl logs ${POD_NAME}
```

- 进入 Pod 内部某个 container:

```
kubectl exec -it [options] ${POD_NAME} -c ${CONTAINER_NAME} [args]
```

这个命令的作用是通过 kubectl 执行了docker exec xxx进入到容器实例内部。之后，就是用户检查自己服务的日志来定位问题。

K8S真的放弃Docker了吗?

Docker作为非常流行的容器技术，之前经常有文章说它被K8S弃用了，取而代之的是另一种容器技术containerd!其实containerd只是从Docker中分离出来的底层容器运行时，使用起来和Docker并没有啥区别，从Docker转型containerd非常简单，基本没有什么门槛。只要把之前Docker命令中的docker改为crictl基本就可以了，都是同一个公司出品的东西，用法都一样。所以不管K8S到底弃用不弃用Docker，对我们开发者使用来说，基本没啥影响!

K8S CRI

K8S发布CRI（Container Runtime Interface），统一了容器运行时接口，凡是支持CRI的容器运行时，皆可作为K8S的底层容器运行时。
K8S为什么要放弃使用Docker作为容器运行时，而使用containerd呢？
如果你使用Docker作为K8S容器运行时的话，kubelet需要先要通过dockershim去调用Docker，再通过Docker去调用containerd。
如果你使用containerd作为K8S容器运行时的话，由于containerd内置了CRI插件，kubelet可以直接调用containerd。
使用containerd不仅性能提高了（调用链变短了），而且资源占用也会变小（Docker不是一个纯粹的容器运行时，具有大量其他功能）。
当然，未来Docker有可能自己直接实现K8S的CRI接口来兼容K8S的底层使用。

```
文档：04-VIP-Kubernetes快速实战与核心原理
链接：http://note.youdao.com/noteshare?id=bc7bee305611b52d6900ba209a92bd4d&sub=BCF5561C9FB34AD895B95CDD4095DF36
```