

# 将应用制作成镜像发布到服务器k8s上作为容器微服务运行

首先我们需要在本地docker中调试运行一遍,再发布到k8s上去。

如果需要在本地部署k8s环境，可以使用mimnikube，参考：https://blog.csdn.net/luanpeng825485697/article/details/80862581

拉取公用镜像

如果在docker拉取公用镜像,例如拉取redis和mysql

拉取mysql和redis镜像

docker pull mysql

docker pull redis

将本地应用制作成镜像

将应用制作成docker镜像,需要我们编写dockerfile文件.

可以参考https://blog.csdn.net/luanpeng825485697/article/details/80921390

编写好dockerfile以后,我们将应用制作成镜像

docker build -t images\_name -f dockerfilepath .

1

上面这句话images\_name是自己指定的镜像的名称 dockerfilepath是dockerfile文件的相对路径,最后面的.是当前目录.在dacokerfile文件中就是以最后的这个.代表的路径作为上下文路径的

运行公用镜像

还是以redis和mysql为例

docker run -p 3306:3306 --name mysql\_container -v $PWD/conf:/etc/mysql/conf.d -v $PWD/logs:/logs -v $PWD/data:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql

docker run -p 6379:6379 --name redis\_container -v $PWD/data:/data -d redis redis-server --appendonly yes

上面的语句直接运行镜像,通过命令设置了端口号,容器名称,设置了挂载和部分参数设置.

其中我们也可以看出,对于数据库镜像,我们要将容器的数据文件挂载成本地的.这样容器重启后数据也还在本地.

运行自定义镜像

根据构建的镜像启动容器

docker run --name container\_name -p 8881:8881 -d image\_name # 容器名 镜像名 映射端口

1

这样就能按照镜像来启动容器了. 命令中端口映射前面是主机端口,后面是容器环境的端口. 只有绑定了才能将主机端口的信息送达到容器的端口,容器才能从自己的端口处得到信息.

当然也可以启动,停止,删除容器

docker ps -a 查看容器

docker stop container\_id 停止容器,但是不删除

docker start contain\_id 启动停止的容器

docker rm contain\_id 删除容器

这个需要注意的是在docker中,容器启动后,再关闭容器中的内容是会保存的,所以启动一个已经关闭的容器,会在上一次运行以后的结果下再运行.但是这些结果不会保存到镜像中,如果我们将容器删了,再重新根据镜像启动容器,那会是一个原始的容器.

而在k8s中,容器一但关闭,就彻底删除,不可能保留上一次的结果.

进入容器进行调试

上面我们已经学习了一种启动模型

docker run --name container\_name -p 8881:8881 -d image\_name # 容器名 镜像名 映射端口

1

启动-d表示后台运行的意思.

当然也可以以交互的方式启动容器

docker run --name container\_name -p 8881:8881 -it image\_name /bin/bash

1

这种启动方式,启动后就自动进入容器内部命令行,不过这种启动方式,在退出容器后会,容器会自动关闭.

所以建议还是以-d后台启动的形式启动容器,然后我们再进入正在运行的容器中

docker ps -a 查看所有的容器

docker exec -it container\_id /bin/bash 　　根据容器id进入容器

1

2

这样就可以进入一个正在运行的容器,做相关修改

容器内部修改

容器内部是一个非常简单的环境，所以不会像我们的电脑包含很多工具。

容器应用配置文件问题

比如最简单的vim都没有，所以你会发现很多命令没法用了。我们想编辑文件，可能就需要安装一下vim

apt-get update

apt-get install vim

1

2

但是只要安装了就会增大容器的大小，而且安装的内容并不经常运行，所以我们也可以经本地的文件修改，复制到容器中

docker cp xxx.txt container\_id:/xxx/xxx/xxxx

1

不过这种方法也过于繁琐，所以最好的办法还是通过主机和容器建共享目录。在本机上和容器中使用同一个共享文件。

应用初始化问题

有些应用第一次运行是需要初始化的，比如mysql，我们要创建数据库，这些数据库和数据表创建以后会保存到共享文件中。

比如下面是一个初始化mysql的命令

docker exec进入正在运行的mysql容器

配置mysql

初始化设置mysql数据库

mysql -u root -p

密码123456

创建数据库 create database database\_name

然后运行本地初始化脚本

或者直接

use database\_name;

create table table\_name(face\_id varchar(64),age int,gender tinyint, handle\_time int, vip tinyint,face\_group char(64),face\_info char(64),save\_path char(64) not null, reg\_path char(64));

下面就可以在本机模拟客户端进行调试运行，查看每个容器的运行，查看配置文件该怎么写。以便放在k8s上能正常运行。

当我们成功在docker中成功跑通了我们的所有应用，并修正了所有的问题后，build了最近的镜像。下面我们来在k8s上运行应用

将本地docker镜像上传到服务器docker镜像中

我们需要先将本地docker镜像save到本地压缩文件

docker save -o image\_name.tar image\_name

1

再通过其他方式上传到服务器，比如ssh的scp方式

scp image\_name.tar xxx@192.168.1.11:/home/xxx/

1

然后登录服务器，在服务器的docker中加载镜像压缩文件

docker load -i image\_name.tar

1

这样本地的docker中的镜像就到了服务器的docker镜像中，下面需要将docker中的镜像推送到服务器私有仓库的镜像中

在服务器的docker中创建私有仓库

这一步操作是管理员做的事情。我们需要先在docker中创建私有仓库，再将k8s指定到docker的私有仓库。这样k8s才能使用私有仓库

1.master主机的docker先从公有仓库下载registry 镜像

docker pull registry

1

2.master主机的docker启动镜像

docker run -d -p 5000:5000 --restart=always --privileged --name registry-container -v /root/docker/my\_registry:/var/lib/registry registry

1

添加挂载使得本地/root/docker/my\_registry用来存放私有仓库的镜像，这样容器重启镜像也不会删除。

3.master主机上修改每个结点的docker配置文件,用来将docker绑定到私有仓库

网络上有些教程，修改/etc/sysconfig/docker这个配置文件。但是这些教程对应的docker版本比较老，所以不要使用这些教程。新版本的docker使用下面的命令

vi /lib/systemd/system/docker.service

添加或修改

ExecStart=/usr/bin/dockerd -H fd:// --insecure-registry=192.168.42.58:5000

1

2

3

如果不配置上面的情况会出现https错误

重启docker

systemctl daemon-reload

systemctl restart docker

1

2

service docker stop 关闭docker服务

service docker start 启动docker服务

1

2

上面的教程是使docker能正常拉取私有仓库镜像

我们平时基本把镜像部署在k8s中，所以k8s绑定docker私有仓库镜像也需要对k8s进行设置，以避免https错误，当然如果你的版本正确，是不会出现https错误，那样也就不需要修改配置文件。

根据k8s文件版本不同，也需要修改k8s的配置文件kube-apiserver文件，也是要添加

--insecure-registry=192.168.42.58:5000

1

不过每种格式的书写格式可能不同。

如果是minikube也可以设置insecure-registry。根据版本不同包含向量两种写法

minikube start --insecure-registry=192.168.42.58:5000

但是如果你已经‘minikube start’过，那必须先：

minikube delete

或者

minikube start --insecure-registry "192.168.42.58:5000" 我的版本比较新，这种成功的

如果你觉的在服务器上搭建私有仓库麻烦，你可以在docker-hub上申请私有仓库，那样就不需要修改配置文件了。

为私有仓库配置登录账号密码

如果想要控制registry的使用权限，使其只有在登录用户名和密码之后才能使用的话，还需要做额外的设置。如果只是调试那就不需要太多的控制，可以忽略这一步。

先创建存储账号密码的地方

mkdir /data/docker-registry-auth/

1

2

创建账号密码

docker run --entrypoint htpasswd registry:latest -Bbn username password > /data/docker-registry-auth/htpasswd

例如

docker run --entrypoint htpasswd registry:latest -Bbn luanpeng 123456 > /data/docker-registry-auth/htpasswd

1

2

3

上面这条命令是为luanpeng用户名生成密码为123456的一条用户信息，存在/data/docker-registry-auth/htpasswd文件里面，文件中存的密码是被加密过的。

此容器启动后会自动停止。

检查下htpasswd是否已经被创建：

$ cat /data/docker-registry-auth/htpasswd

luanpeng:$2y$05$WcUjExtIbG4w1.viDLYxQeo65dUzVeN80YWHwuIpWHTwDIjXRnJIS

1

2

使用带用户权限的registry时候，容器的启动命令就跟上面不一样了，将之前的容器停掉并删除，然后执行下面的命令：

docker run -d -p 5000:5000 --restart=always --name registry-container -v /data/docker-registry-auth/:/auth/ -e "REGISTRY\_AUTH=htpasswd" -e "REGISTRY\_AUTH\_HTPASSWD\_REALM=Registry Realm" -e REGISTRY\_AUTH\_HTPASSWD\_PATH=/auth/htpasswd -v /root/docker/my\_registry:/var/lib/registry registry:latest

1

上面的命令添加了配置文件的映射，又添加了镜像存储地方的映射

带有用户权限的registry启动以后，如果直接想查看仓库信息、pull或push都会出现权限报错，必须先登录

sudo docker login --username=luanpeng 192.168.42.58:5000 # 登录

docker pull 192.168.42.58:5000/images\_name:latest

1

2

将镜像推送到带有访问权限的私有仓库也是一样

sudo docker login --username=yourusername 192.168.42.58:5000

$ sudo docker tag [ImageId] 192.168.42.58:5000/images\_name:[镜像版本号]

$ sudo docker push 192.168.42.58:5000/images\_name:[镜像版本号]

根据提示，输入用户名和密码即可。如果登录成功，会在/root/.docker/config.json文件中保存账户信息，这样就可以继续使用了。

上传以后可以查看

curl http://192.168.42.58:5000/v2/\_catalog 查看私有仓库里面的镜像

curl http://192.168.42.58:5000/v2/imagename/tags/list 查看私有仓库里面的镜像的版本

1

2

（下面是阿里云私有镜像）

sudo docker login --username=yourusername registry.cn-shenzhen.aliyuncs.com

$ sudo docker tag [ImageId] registry.cn-shenzhen.aliyuncs.com/haibincoder/centos:[镜像版本号]

$ sudo docker push registry.cn-shenzhen.aliyuncs.com/haibincoder/centos:[镜像版本号]

1

2

3

如果是minikube启动的单机效果可以不用设置账号密码。

直接使用

docker login localhost:5000 就可以登录了，账号密码就是电脑账号密码

1

为k8s集群创建Secret

启动私有仓库账号密码权限以后，k8s链接私有仓库就需要账号密码了。

kubectl delete secret your-registr-name 删除旧的或错的秘钥，当然也可以不删除

kubectl create secret docker-registry lpsecret --docker-server=192.168.42.58:5000 --docker-username=luanpeng --docker-password=123456 --docker-email=825485697@qq.com -n kube-system

1

2

3

使用命令方式创建邮箱为必选参数，老版本可能不是必填项

注意: -n your-namespaces 为指定命名空间，需要自己先创建命名空间，（后面的教程包含创建命名空间）一般搭建k8s集群时，建议新建一个命名空间来隔离资源。

your-registr-name为你为保密字典定义的名称。为后面的yaml中就要填写这个秘钥的形成

spec:

imagePullSecrets:

- name: your\_registr\_name

containers:

- name: your-container # 容器名称

image: 192.168.42.201:5000/your\_images:1.0 # 基于的镜像名, 根据镜像创建容器

创建以后你就可以在k8s的dashboard中保密字典类目看到你创建的秘钥

如果没有dashboard可以通过下面的命令查看

kubectl get secret -n your-namespaces

1

在k8s中拉取私有仓库的镜像有时也会报错https错误，这个时候就需要修改k8s的配置文件

将本地docker镜像推送到私有仓库中

要想在k8s中拉取镜像，必须按照上面的教程，为k8s绑定私有仓库服务器，并配置私有仓库提供的账号密码。

对于有权限限制的docker私有仓库，我们要先使用docker login进行登录

sudo docker login --username=yourusername 192.168.42.58:5000

docker tag image\_name:latest 192.168.42.58:5000/image\_name:latest #在docker镜像标记为私有仓库的镜像

docker push 192.168.42.58:5000/image\_name:latest # 将本地镜像push到私有仓库

如果出现下面类似的错误，说明使用的是https,需要按上面的教程修改docker配置文件

v1 ping attempt failed with error: Get https://192.168.174.128:5000/v1/\_ping: http: server gave HTTP response to HTTPS client

1

如果tag错了，想要删除的话需要

关闭容器后

docker rmi 192.168.42.58:5000/image\_name 先根据名称删除，删除tag

docker rmi 1c8ea16890a8 再根据镜像id删除，删除镜像

1

2

3

查看仓库里面情况

curl http://192.168.42.58:5000/v2/\_catalog

查看私有仓库里面的镜像

curl http://192.168.42.58:5000/v2/imagename/tags/list

查看私有仓库里面的镜像的版本

curl http://192.168.42.58:5000/v2/imagename/manifests/tag (镜像每一层的哈希值（digest））

推送到私有仓库以后，自定义镜像便有了镜像地址。在后面编写yaml文件时，需要使用私有仓库镜像的地址

若在将镜像推送到私有仓库时，有错误类似Get https://192.168.42.58:5000/v1/\_ping: http: server gave HTTP response to HTTPS client

修改/etc/docker/daemon.json文件，写入：

{

"registry-mirrors": ["http://hub-mirror.c.163.com"]

}

{

"insecure-registries":["192.168.42.58:5000"]

}

或者修改

vi /lib/systemd/system/docker.service

添加或修改

ExecStart=/usr/bin/dockerd -H fd:// --insecure-registry=192.168.42.58:5000

1

2

3

或者修改

/etc/init/docker.conf，在其中增加--insecure-registry 192.168.112.136:5000

由于各个版本不一样，所以修改的地方就不一样

//重启docker

service docker restart

本地控制服务器k8s

k8s的控制是通过~/.kube/config文件来实现的。

所以我们只需要将服务器上/.kube/config的文件内容拷贝到本地pc机器上的/.kube/config就可以通过kubectl来控制服务器上的k8s的资源了。

如果本地使用monikube调试，/.kube/config文件必须删除以后，monikube才能重新创建config文件，不会自动修改config。所以如果我们当前想通过minikube调试，就将/.kube/config文件删除，再启动minikube start，如果想控制远程服务器的k8s，就修改/.kube/config为远程服务器的/.kube/config内容。

学习yaml文件语法

yaml文件的语法格式可以参考：https://blog.csdn.net/luanpeng825485697/article/details/79478338

当然，如果你已经学过了或者你有现成的yaml文件demo可以直接跳过

在k8s上创建空间

创建文件create\_namespace.yaml

apiVersion: v1

kind: Namespace

metadata:

name: luanpeng

labels:

name: luanpeng

在当前目录下运行

kubectl create -f create\_namespace.yaml

1

就可以在kubectl链接的k8s上创建命名空间luanpeng这个空间了。以后设置资源都可以在这个空间中设置

创建服务

这里我们以创建service服务为例，分别以私有镜像和共有镜像为例。

使用公有镜像创建容器

使用公有镜像创建容器，比如我们使用官方最新版的mysql来创建服务。

如果我们创建的是共有镜像，我们就不需要前面介绍的那些build镜像，上传，加载等一系列操作。只需要编写yaml文件。系统会自动从官方拉取镜像。不过要保障官方有此版本的镜像文件。

创建mysql.yaml文件

---

---

apiVersion: v1

kind: Service

metadata:

name: mysql-service

namespace: luanpeng

labels:

app: mysql-service

spec:

type: NodePort

ports:

- port: 3306

targetPort: 3306

protocol: TCP

name: http

selector:

app: mysql-deployment

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: mysql-deployment

namespace: luanpeng

spec:

selector:

matchLabels:

app: mysql-pod

replicas: 1

template:

metadata:

labels:

app: mysql-pod

spec:

nodeName: node1 # 使用nodeName可以指定运行节点

volumes:

- name: mysql-data

hostPath:

path: /mysql/data

- name: mysql-log

hostPath:

path: /mysql/log

- name: mysql-conf

hostPath:

path: /mysql/conf

containers:

- name: mysql-container

image: mysql:5.6 # 最新版本的不能进行划分数据卷

ports:

- containerPort: 3306

env:

- name: MYSQL\_ROOT\_PASSWORD

value: introcks

volumeMounts:

- name: mysql-data

mountPath: /var/lib/mysql

- name: mysql-log

mountPath: /logs

- name: mysql-conf

mountPath: /etc/mysql/conf.d

1

上面的yaml文件中，我们创建了一个mysql服务，服务下包含一个deployment，deployment下包含一个pod，一个pod下面启动一个container

使用kubectl创建镜像

kubectl create -f mysql.yaml

1

启动以后我们可以在k8s的dashboard中查看运行情况，如果报错，我们可以先关闭服务，然后根据提示解决问题

kubectl delete -f mysql.yaml

1

启动mysql的容器后，

我们进入容器

kubectl get pods -n luanpeng # 查看指定空间下的pod

kubectl exec mysql-deployment-db9bc9f9c-5cdpq -it -n luanpeng /bin/bash # 进入指定容器

1

2

对数据库进行初始化

初始化设置mysql数据库

mysql -u root -p

密码xxxxx

创建数据库

create database yourdatabase;

然后运行本地初始化脚本

或者直接

use yourdatabase;

create table yourtable(face\_id varchar(64),age int,gender tinyint, handle\_time int, vip tinyint,face\_group char(64),face\_info char(64),save\_path char(64) not null, reg\_path char(64));

desc yourtable; 查看表结构

这样就完成了创建，由于使用了共享卷，所以pod重启以后数据库结构信息仍然存在

使用私有仓库拉取镜像创建容器

对于公司内部的项目, 我们不可能使用公有开放的镜像仓库, 一般情况可能会花钱买 docker私仓服务, 或者说自己在服务器上搭建自己的私仓, 但不管怎样, 我们如何让k8s能够拉取私有仓库的镜像

自己的服务器创建私有仓库

如果在自己服务器上创建私有仓库。则直接在yaml中输入192.168.xx.xx:5000/imagesname:1.0这种镜像地址，然后输入拉取秘钥就可以拉取成功。

....

spec:

imagePullSecrets:

- name: registry-key

containers:

- name: your-container # 容器名称

image: 192.168.42.201:5000/your-images:1.0 # 基于的镜像名, 根据镜像创建容器

.....

使用阿里云提供的私有仓库

1、登录docker镜像仓库

这里以阿里云docker镜像仓库为例

docker login --username=yin32167@aliyun.com registry.cn-hangzhou.aliyuncs.com

1

之后输入密码就可以了, 这个时候我们可以在配置文件中查看登录情况

cat ~/.docker/config.json

1

这个时候我们虽然可以通过docker pull命令拉取镜像, 但无法通过k8s创建pod方式拉取

2、生成密钥secret

kubectl create secret docker-registry regsecret --docker-server=registry.cn-hangzhou.aliyuncs.com --docker-username=yin32167@aliyun.com --docker-password=xxxxxx --docker-email=yin32167@aliyun.com

其中:

regsecret: 指定密钥的键名称, 可自行定义

--docker-server: 指定docker仓库地址

--docker-username: 指定docker仓库账号

--docker-password: 指定docker仓库密码

--docker-email: 指定邮件地址(选填)

可以看到当前除了默认的密钥, 还有我们刚才生成的. 另外要注意的是, 该密钥只能在对应namespace使用, 也就是这里的default, 如果需要用到其他namespace, 比如说test, 就需要在生成的时候指定参数 -n test

3、 yml文件加入密钥参数

containers:

- name: channel

image: registry-internal.cn-hangzhou.aliyuncs.com/yin32167/channel:dev-1.0

ports:

- containerPort: 8114

imagePullSecrets:

- name: regsecret

其中imagePullSecrets是声明拉取镜像时需要指定密钥, regsecret 必须和上面生成密钥的键名一致, 另外检查一下pod和密钥是否在同一个namespace, 之后k8s便可以拉取镜像

配置文件

配置文件一般我们包含在应用中，而入股打包成镜像以后，配置文件就不能再修改了，所以我们有集中方式来实现配置文件。

1、设置挂载，我们可以将配置文件的目录挂载成主机地址或网络地址。

在第一次创建pod时，不要让pod包含任何启动命令，然后进入pod，里面的配置文件是空的，因为容器中配置文件的目录，其实是主机的目录，而这个目录可能都不存在。在容器中，进入配置文件目录，添加配置文件，或修改配置文件。当然也可以通过kubectl cp直接复制到容器指定目录下。这样容器的配置文件目录下就有了配置文件，其实这个配置文件是存储在服务器主机或者网络上的（因为这个目录是通过挂载添加到容器的）。然后删除这个服务（或者pod），重新创建一个服务（pod），这个pod是带有启动命令的。并且也添加了挂载。这个pod启动后就能正常读取配置文件了。

2、通过k8s的configmap实现。

configmap是k8s环境下的配置文件。跟secret一样。通过命令创建存储在k8s中。pod可以绑定configmap来读取配置。

通过命令行创建configmap

kubectl create configmap lykops-config --from-file=db\_config\_file=database.conf --from-file=ver.conf --from-literal=username=test --from-literal=hostname=localhost

1

–from-file表示来自文件，直接把文件内容写入configmap中，可以为目录也可以为文件，如果是文件的话，可以使用db\_config\_file=database.conf来修改key值

–from-literal表示使用键值对配置

通过yaml文件创建configmap

kubectl delete -f lykops-config.yaml

cat << EOF > lykops-config.yaml

kind: ConfigMap

apiVersion: v1

metadata:

name: lykops-config

namespace: default

labels:

software: apache

project: lykops

app: configmap

version: v1

data:

PWD: /

user: lykops

mysql.config : |-

username: lykops

host: localhost

port: 3306

EOF

kubectl create -f lykops-config.yaml

ata就是配置变量：

PWD和user两行就是两个环境变量属性

mysql.config : |-就是配置文件，下面的内容是配置文件mysql.config内容

使用ConfigMap

两种方式让pod使用，第一种是环境变量或参数，第二种是文件挂载。

cat << EOF > lykops-cm-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: lykops-cm-pod

labels:

project: lykops

app: lykops-cm

version: v1

spec:

containers:

- name: lykops-cm-pod

image: web:apache

command: ['sh',/etc/run.sh]

env:

- name: SPECIAL\_USER

valueFrom:

configMapKeyRef:

name: lykops-config

key: username

resources:

requests:

cpu: 0.01

memory: 8Mi

limits:

cpu: 0.1

memory: 16Mi

volumeMounts:

- name: config-volume

mountPath: /data/

volumes:

- name: config-volume

configMap:

name: lykops-config

EOF

kubectl create -f lykops-cm-pod.yaml

1

当ConfigMap以数据卷的形式挂载进Pod时，更新ConfigMap（或删掉重建ConfigMap），Pod内挂载的配置信息会热更新，但使用环境变量方式加载到pod，则不会自动更新。

点赞 1

————————————————

版权声明：本文为CSDN博主「数据架构师」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/luanpeng825485697/article/details/81256680

# K8S api接口详解

一、创建管理员用户，授权，获取token值

1. 创建用户

编写CreateServiceAccount.yaml文件

apiVersion: v1

kind: ServiceAccount

metadata:

name: admin-user

namespace: kube-system

创建用户命令：

kubectl create -f CreateServiceAccount.yaml

查看创建用户的命令

kubectl -n kube-system get sa

2. 用户授权

编写RoleBinding.yaml文件，给于cluster-admin角色

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRoleBinding

metadata:

name: admin-user

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-admin

subjects:

- kind: ServiceAccount

name: admin-user

namespace: kube-system

给用户授权，执行命令

kubectl create -f RoleBinding.yaml

3. 获取token

kubectl get secret -n kube-system | grep admin

kubectl describe secret admin-user-token-7lsrn -n kube-system |grep token

4. 然后可以拿取到的token，去访问api

可以通过postman来调试，访问api

在postaman中关闭SSL认证（setting）

二、管理Pod

1. 创建pod

编写yaml文件，namespace可以默认default，也可以提前自己创建

apiVersion: v1

kind: Pod

metadata:

name: test-pod2

namespace: testzhang

spec:

nodeSelector:

type: node1

containers:

- name: nginx-container

image: nginx:1.12.2

ports:

- containerPort: 80

在nodeSelector中可以指定node节点

将pod调度到指定节点有2种方式：

方式一：强制约束Pod调度到指定Node节点上

方式二：通过label-selector机制选择节点

以下使用第二种方式实现pod调度到指定节点。

这个方式是通过匹配节点的标签来将pod部署在指定的node上的，所以节点必须先打好标签

①为Node添加label

kubectl label node k8s-node1<节点名称> type=node1

kubectl get nodes --show-labels

创建Pod对象

kubectl create -f pod.yaml

1

查看pod对象

kubectl get pods

1

查看pod详细信息

kubectl get pods -o wide

查看指定的namespace的pod信息：

kubectl get pods -o wide -n testzhang（namespace名称）

查看全部namespace的pod信息：

kubectl get pods -o wide -A

查看指定的pod信息，不指定namespace 则会找默认的default命名空间

kubectl get pod test-pod1 -o wide -n testzhang

查看pod 详细描述

kubectl describe pod test-pod1 -n testzhang

删除pod

kubectl delete pod [podname]

还可以指定创建pod时的yaml文件进行删除指定的pod:

kubectl delete -f pod.yaml

更新pod

更新pod必须先删除pod然后创建新的。

可以将其中镜像升级后更行yaml文件。然后创建pod

还有一个方法是修改yaml文件后执行命令kubectl replace -f pod.yaml --force来用新的yaml生成的pod替换原有的pod。这里使用–force参数是因为在替换的时候可能会失败，所以使用这个参数表示强制替换。

这个命令的原理也是先删除原有pod，再创建新的pod

关于pod的其他操作

// 查看pod具体信息

kubectl describe pod [podname]

// 查看pod日志

kubectl log [podname]

// 连接pod

kubectl exec -it [podname] bash

————————————————

版权声明：本文为CSDN博主「Shallow22」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/xili2532/article/details/104543205>

# K8S API接口汇总

一、资源对象

首先附上官方API参考文档

官方api文档

1. namespace

增（创建）POST请求：

创建namespace： /api/v1/namespaces

删（删除） DELETE请求：

删除namespace： /api/v1/namespaces/{name}

改（修改）PUT请求：

替换指定的命名空间： /api/v1/namespaces/{name}

替换指定名称空间的状态： /api/v1/namespaces/{name}/status

如果部分更新可以用 PATCH

查（查询） GET请求：

查询全部： /api/v1/namespaces

查询指定namespace： /api/v1/namespaces/{name}

2. Pod

增（创建）POST请求：

创建pod： /api/v1/namespaces/{namespace}/pods

删（删除） DELETE请求：

删除pod： /api/v1/namespaces/{namespace}/pods/{name}

改（修改）PUT请求：

替换指定的pod： /api/v1/namespaces/{namespace}/pods/{name}

如果部分更新可以用 PATCH

查（查询） GET请求：

查询全部： /api/v1/namespaces/{namespace}/pods

查询指定pod： /api/v1/namespaces/{namespace}/pods/{name}

3. Node

增（创建）POST请求：

创建node： /api/v1/nodes

删（删除） DELETE请求：

删除node： /api/v1/nodes/{name}

改（修改）PUT请求：

替换指定的node： /api/v1/nodes/{name}

替换指定node的状态： /api/v1/nodes/{name}/status

如果部分更新可以用 PATCH

查（查询） GET请求：

查询全部： /api/v1/nodes

查询指定node： /api/v1/nodes/{name}

查询指定节点内所有Pod的信息： /api/v1/nodes/{name}/pods/

查询指定节点内物理资源的统计信息: /api/v1/nodes/{name}/stats/

查询指定节点的概要信息： /api/v1/nodes/{name}/spec/

4. Service

增（创建）POST请求：

创建service： /api/v1/namespaces/{namespace}/services

删（删除） DELETE请求：

删除service： /api/v1/namespaces/{namespace}/services/{name}

改（修改）PUT请求：

替换指定的service： /api/v1/namespaces/{namespace}/services/{name}

如果部分更新可以用 PATCH

查（查询） GET请求：

查询全部： /api/v1/namespaces/{namespace}/services

查询指定service： /api/v1/namespaces/{namespace}/services/{name}

其实大多都差不多相似，照着规律就行

————————————————

版权声明：本文为CSDN博主「Shallow22」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/xili2532/article/details/104562184>

# k8s service的yaml定义

Service：

Service有四种type: ClusterIP(默认）、NodePort、LoadBalancer、ExternalName. 其中NodePort和LoadBalancer两类型的Services可以对外提供服务。

engine.yaml：

apiVersion: v1 #API的版本号，版本号可以用 kubectl api-versions 查询到

kind: Service #表明资源对象，例如Pod、RC、Service、Namespace及Node等

metadata: #资源对象的元数据定义

name: engine #service名称

spec: #资源对象的详细定义，持久化到etcd中保存

type: ClusterIP #Service类型，ClusterIP供kubernates集群内部pod访问

ports: #暴露的端口列表

- port: 8080 #Service监听的端口，对应ClusterIP，即ClusterIP+ServicePort供集群内部pod访问的

targetPort: 8080 #对应pod中容器的端口

protocol: TCP #协议，支持TCP、UDP，默认TCP

name: http #端口名称

selector: #label选择器，管理label对应的pod

name: enginehttpmanage #pod的label

kubectl create -f engine.yaml

kubectl get services | grep engine

 kubectl delete service engine

图片参考：kubernetes pod、service文件yaml模版格式

# Kubernetes之yaml文件详解(汇总-详细）

## **一、YAML基础**

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

YAML语法规则：

大小写敏感

使用缩进表示层级关系

缩进时不允许使用Tal键，只允许使用空格

缩进的空格数目不重要，只要相同层级的元素左侧对齐即可

”#” 表示注释，从这个字符一直到行尾，都会被解析器忽略

在Kubernetes中，只需要知道两种结构类型即可：

Lists

Maps

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

#### **1.1 YAML Maps**

Map顾名思义指的是字典，即一个Key:Value 的键值对信息。例如：

apiVersion: v1

kind: Pod

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

* 1
* 2
* 3

Maps的value既能够对应字符串也能够对应一个Maps。例如：

apiVersion: v1

kind: Pod

metadata:

name: kube100-site

labels:

app: web

* 1
* 2
* 3
* 4
* 5
* 6

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

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

注意：在YAML文件中绝对不要使用tab键

#### **1.2 YAML Lists**

List即列表，说白了就是数组，例如：

args

-beijing

-shanghai

-shenzhen

-guangzhou

* 1
* 2
* 3
* 4
* 5

可以指定任何数量的项在列表中，每个项的定义以破折号（-）开头，并且与父元素之间存在缩进。在JSON格式中，表示如下：

{

"args": ["beijing", "shanghai", "shenzhen", "guangzhou"]

}

* 1
* 2
* 3

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

apiVersion: v1

kind: Pod

metadata:

name: kube100-site

labels:

app: web

spec:

containers:

- name: front-end

image: nginx

ports:

- containerPort: 80

- name: flaskapp-demo

image: jcdemo/flaskapp

ports: 8080

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

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

{

"apiVersion": "v1",

"kind": "Pod",

"metadata": {

"name": "kube100-site",

"labels": {

"app": "web"

},

},

"spec": {

"containers": [{

"name": "front-end",

"image": "nginx",

"ports": [{

"containerPort": "80"

}]

}, {

"name": "flaskapp-demo",

"image": "jcdemo/flaskapp",

"ports": [{

"containerPort": "5000"

}]

}]

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

## **二、说明**

定义配置时，指定最新稳定版API  
配置文件应该存储在集群之外的版本控制仓库中。如果需要，可以快速回滚配置、重新创建和恢复  
应该使用YAML格式编写配置文件，而不是json。YAML对用户更加友好  
可以将相关对象组合成单个文件，通常会更容易管理  
不要没必要指定默认值，简单和最小配置减小错误  
在注释中说明一个对象描述更好维护

## **三、使用YAML创建Pod**

apiVersion: v1

kind: Pod

metadata:

name: kube100-site

labels:

app: web

spec:

containers:

- name: front-end

image: nginx

ports:

- containerPort: 80

- name: flaskapp-demo

image: jcdemo/flaskapp

ports:

- containerPort: 5000

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

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

# kubectl api-versions

admissionregistration.k8s.io/v1beta1

apiextensions.k8s.io/v1beta1

apiregistration.k8s.io/v1

apiregistration.k8s.io/v1beta1

apps/v1

apps/v1beta1

apps/v1beta2

authentication.k8s.io/v1

authentication.k8s.io/v1beta1

authorization.k8s.io/v1

authorization.k8s.io/v1beta1

autoscaling/v1

autoscaling/v2beta1

batch/v1

batch/v1beta1

certificates.k8s.io/v1beta1

events.k8s.io/v1beta1

extensions/v1beta1

networking.k8s.io/v1

policy/v1beta1

rbac.authorization.k8s.io/v1

rbac.authorization.k8s.io/v1beta1

storage.k8s.io/v1

storage.k8s.io/v1beta1

v1

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

（2）下面是一个典型的容器定义：

spec:

containers:

- name: front-end

image: nginx

ports:

- containerPort: 80

* 1
* 2
* 3
* 4
* 5
* 6

上述例子只是一个简单的最小定义：一个名字（front-end）、基于nginx的镜像，以及容器将会监听的指定端口号（80）。

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

name、image、command、args、workingDir、ports、env、resource、volumeMounts、livenessProbe、readinessProbe、livecycle、terminationMessagePath、imagePullPolicy、securityContext、stdin、stdinOnce、tty

（3）kubectl创建Pod

# kubectl create -f test\_pod.yaml

pod "kube100-site" created

* 1
* 2

（4）查看Pod状态

# kubectl get pod

NAME READY STATUS RESTARTS AGE

kube100-site 2/2 Running 0 2m

* 1
* 2
* 3
* 4
* 5

## **四、创建Deployment**

名词解释

#test-pod

apiVersion: v1 #指定api版本，此值必须在kubectl apiversion中

kind: Pod #指定创建资源的角色/类型

metadata: #资源的元数据/属性

name: test-pod #资源的名字，在同一个namespace中必须唯一

labels: #设定资源的标签

k8s-app: apache

version: v1

kubernetes.io/cluster-service: "true"

annotations: #自定义注解列表

- name: String #自定义注解名字

spec: #specification of the resource content 指定该资源的内容

restartPolicy: Always #表明该容器一直运行，默认k8s的策略，在此容器退出后，会立即创建一个相同的容器

nodeSelector: #节点选择，先给主机打标签kubectl label nodes kube-node1 zone=node1

zone: node1

containers:

- name: test-pod #容器的名字

image: 10.192.21.18:5000/test/chat:latest #容器使用的镜像地址

imagePullPolicy: Never #三个选择Always、Never、IfNotPresent，每次启动时检查和更新（从registery）images的策略，

# Always，每次都检查

# Never，每次都不检查（不管本地是否有）

# IfNotPresent，如果本地有就不检查，如果没有就拉取

command: ['sh'] #启动容器的运行命令，将覆盖容器中的Entrypoint,对应Dockefile中的ENTRYPOINT

args: ["$(str)"] #启动容器的命令参数，对应Dockerfile中CMD参数

env: #指定容器中的环境变量

- name: str #变量的名字

value: "/etc/run.sh" #变量的值

resources: #资源管理

requests: #容器运行时，最低资源需求，也就是说最少需要多少资源容器才能正常运行

cpu: 0.1 #CPU资源（核数），两种方式，浮点数或者是整数+m，0.1=100m，最少值为0.001核（1m）

memory: 32Mi #内存使用量

limits: #资源限制

cpu: 0.5

memory: 1000Mi

ports:

- containerPort: 80 #容器开发对外的端口

name: httpd #名称

protocol: TCP

livenessProbe: #pod内容器健康检查的设置

httpGet: #通过httpget检查健康，返回200-399之间，则认为容器正常

path: / #URI地址

port: 80

#host: 127.0.0.1 #主机地址

scheme: HTTP

initialDelaySeconds: 180 #表明第一次检测在容器启动后多长时间后开始

timeoutSeconds: 5 #检测的超时时间

periodSeconds: 15 #检查间隔时间

#也可以用这种方法

#exec: 执行命令的方法进行监测，如果其退出码不为0，则认为容器正常

# command:

# - cat

# - /tmp/health

#也可以用这种方法

#tcpSocket: //通过tcpSocket检查健康

# port: number

lifecycle: #生命周期管理

postStart: #容器运行之前运行的任务

exec:

command:

- 'sh'

- 'yum upgrade -y'

preStop:#容器关闭之前运行的任务

exec:

command: ['service httpd stop']

volumeMounts: #挂载持久存储卷

- name: volume #挂载设备的名字，与volumes[\*].name 需要对应

mountPath: /data #挂载到容器的某个路径下

readOnly: True

volumes: #定义一组挂载设备

- name: volume #定义一个挂载设备的名字

#meptyDir: {}

hostPath:

path: /opt #挂载设备类型为hostPath，路径为宿主机下的/opt,这里设备类型支持很多种

#nfs

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74

（1）创建一个yaml文件

apiVersion: apps/v1beta2

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.10

ports:

- containerPort: 80

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

（2）创建deployment

[root@master-01 YAML\_k8s]# kubectl create -f nginx-deployment.yaml

deployment.apps "nginx-deployment" created

[root@master-01 YAML\_k8s]# kubectl get pod -o wide

NAME READY STATUS RESTARTS AGE IP NODE

nginx-deployment-6b7b4d57b4-26wzj 1/1 Running 0 2m 10.20.184.83 master-01

nginx-deployment-6b7b4d57b4-9w7tm 1/1 Running 0 2m 10.20.190.60 node-01

nginx-deployment-6b7b4d57b4-mhh8t 1/1 Running 0 2m 10.20.254.108 node-03

[root@master-01 YAML\_k8s]# kubectl get deployment

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 2m

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

（3）查看标签

[root@master-01 YAML\_k8s]# kubectl get pod --show-labels

NAME READY STATUS RESTARTS AGE LABELS

nginx-deployment-6b7b4d57b4-26wzj 1/1 Running 0 3m app=nginx,pod-template-hash=2636081360

nginx-deployment-6b7b4d57b4-9w7tm 1/1 Running 0 3m app=nginx,pod-template-hash=2636081360

nginx-deployment-6b7b4d57b4-mhh8t 1/1 Running 0 3m app=nginx,pod-template-hash=2636081360

* 1
* 2
* 3
* 4
* 5

（4）通过标签查找Pod

[root@master-01 YAML\_k8s]# kubectl get pod -l app=nginx

NAME READY STATUS RESTARTS AGE

nginx-deployment-6b7b4d57b4-26wzj 1/1 Running 0 6m

nginx-deployment-6b7b4d57b4-9w7tm 1/1 Running 0 6m

nginx-deployment-6b7b4d57b4-mhh8t 1/1 Running 0 6m

* 1
* 2
* 3
* 4
* 5

（5）deployment创建过程

Deployment 管理的是replicaset-controller，RC会创建Pod。Pod自身会下载镜像并启动镜像

[root@master-01 YAML\_k8s]# kubectl describe rs nginx-deployment

...

...

...

Events:

Type Reason Age From Message

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

Normal SuccessfulCreate 33m replicaset-controller Created pod: nginx-deployment-6b7b4d57b4-9w7tm

Normal SuccessfulCreate 33m replicaset-controller Created pod: nginx-deployment-6b7b4d57b4-26wzj

Normal SuccessfulCreate 33m replicaset-controller Created pod: nginx-deployment-6b7b4d57b4-mhh8t

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

[root@master-01 YAML\_k8s]# kubectl describe pod nginx-deployment-6b7b4d57b4-26wzj

...

...

...

Events:

Type Reason Age From Message

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

Normal Scheduled 36m default-scheduler Successfully assigned nginx-deployment-6b7b4d57b4-26wzj to master-01

Normal SuccessfulMountVolume 36m kubelet, master-01 MountVolume.SetUp succeeded for volume "default-token-v5vw9"

Normal Pulled 36m kubelet, master-01 Container image "nginx:1.10" already present on machine

Normal Created 36m kubelet, master-01 Created container

Normal Started 36m kubelet, master-01 Started container

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

（6）升级镜像（nginx1.10–>nginx1.11）

[root@master-01 YAML\_k8s]# kubectl set image deploy/nginx-deployment nginx=nginx:1.11

deployment.apps "nginx-deployment" image updated

[root@master-01 YAML\_k8s]# kubectl exec -it nginx-deployment-b96c97dc-2pxjf bash

root@nginx-deployment-b96c97dc-2pxjf:/# nginx -V

nginx version: nginx/1.11.13

* 1
* 2
* 3
* 4
* 5
* 6

升级镜像的过程是逐步进行的，pod不会一下子全部关闭，而是一个一个升级

（7）查看发布状态

[root@master-01 ~]# kubectl rollout status deploy/nginx-deployment

deployment "nginx-deployment" successfully rolled out

* 1
* 2

（8）查看deployment历史修订版本

[root@master-01 ~]# kubectl rollout history deploy/nginx-deployment

deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 <none>

2 <none>

# 显示历史有两个版本

[root@master-01 ~]# kubectl rollout history deploy/nginx-deployment --revision=1

deployments "nginx-deployment" with revision #1

Pod Template:

Labels: app=nginx

pod-template-hash=2636081360

Containers:

nginx:

Image: nginx:1.10

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

[root@master-01 ~]# kubectl rollout history deploy/nginx-deployment --revision=2

deployments "nginx-deployment" with revision #2

Pod Template:

Labels: app=nginx

pod-template-hash=65275387

Containers:

nginx:

Image: nginx:1.11

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35

（8）编辑deployment

[root@master-01 ~]# kubectl edit deploy/nginx-deployment

# 将nginx版本改为1.12

...

...

...

spec:

containers:

- image: nginx:1.12

imagePullPolicy: IfNotPresent

name: nginx

ports:

- containerPort: 80

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

升级过程：

[root@master-01 ~]# kubectl rollout status deploy/nginx-deployment

Waiting for rollout to finish: 1 out of 3 new replicas have been updated...

Waiting for rollout to finish: 1 out of 3 new replicas have been updated...

Waiting for rollout to finish: 1 out of 3 new replicas have been updated...

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

Waiting for rollout to finish: 1 old replicas are pending termination...

Waiting for rollout to finish: 1 old replicas are pending termination...

deployment "nginx-deployment" successfully rolled out

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

（10）扩容/缩容（指定–replicas的数量）

[root@master-01 ~]# kubectl get pod

NAME READY STATUS RESTARTS AGE

nginx-deployment-6b47cf4878-8mjkr 1/1 Running 0 1m

nginx-deployment-6b47cf4878-kr978 1/1 Running 0 1m

nginx-deployment-6b47cf4878-tvhvl 1/1 Running 0 1m

[root@master-01 ~]# kubectl scale deploy/nginx-deployment --replicas=5

deployment.extensions "nginx-deployment" scaled

[root@master-01 ~]# kubectl get pod

NAME READY STATUS RESTARTS AGE

nginx-deployment-6b47cf4878-6r5dz 0/1 ContainerCreating 0 4s

nginx-deployment-6b47cf4878-7sjtt 0/1 ContainerCreating 0 4s

nginx-deployment-6b47cf4878-8mjkr 1/1 Running 0 2m

nginx-deployment-6b47cf4878-kr978 1/1 Running 0 2m

nginx-deployment-6b47cf4878-tvhvl 1/1 Running 0 2m

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

（11）创建Service提供对外访问的接口

apiVersion: v1

kind: Service

metadata:

name: nginx-service

labels:

app: nginx

spec:

ports:

- port: 88

targetPort: 80

selector:

app: nginx

####

apiVersion: 指定版本

kind: 类型

name: 指定服务名称

labels: 标签

port: Service 服务暴露的端口

targetPort: 容器暴露的端口

seletor: 关联的Pod的标签

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

创建service

# kubectl create -f nginx-service.yaml

* 1

查看service（访问Pod是有负载均衡的）

[root@master-01 YAML\_k8s]# kubectl get svc/nginx-service

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

nginx-service ClusterIP 10.254.131.176 <none> 88/TCP 1m

* 1
* 2
* 3

# curl 10.254.131.176:88

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

<style>

body {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</style>

</head>

<body>

<h1>Welcome to nginx!</h1>

<p>If you see this page, the nginx web server is successfully installed and

working. Further configuration is required.</p>

<p>For online documentation and support please refer to

<a href="http://nginx.org/">nginx.org</a>.<br/>

Commercial support is available at

<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>

</body>

</html>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

对service的描述

# kubectl describe svc/nginx-service

Name: nginx-service

Namespace: default

Labels: app=nginx

Annotations: <none>

Selector: app=nginx

Type: ClusterIP

IP: 10.254.131.176

Port: <unset> 88/TCP

TargetPort: 80/TCP

Endpoints: 10.20.184.19:80,10.20.184.84:80,10.20.190.62:80 + 2 more...

Session Affinity: None

Events: <none>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

（12）回滚到以前的版本

# kubectl rollout history deploy/nginx-deployment

deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 <none>

2 <none>

3 <none>

* 1
* 2
* 3
* 4
* 5
* 6

# kubectl rollout history deploy/nginx-deployment --revision=3

deployments "nginx-deployment" with revision #3

Pod Template:

Labels: app=nginx

pod-template-hash=2603790434

Containers:

nginx:

Image: nginx:1.12

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

# 回滚到上一个版本

# kubectl rollout undo deploy/nginx-deployment

deployment.apps "nginx-deployment"

# 查看版本

# kubectl describe deploy/nginx-deployment

...

...

Labels: app=nginx

Containers:

nginx:

Image: nginx:1.11

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

（13）回滚到指定版本

# kubectl rollout history deploy/nginx-deployment

deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 <none>

3 <none>

4 <none>

# 指定版本

# kubectl rollout undo deploy/nginx-deployment --to-revision=1

deployment.apps "nginx-deployment"

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

### **附上一个具体的yaml解释文件：**

# yaml格式的pod定义文件完整内容：

apiVersion: v1 #必选，版本号，例如v1

kind: Pod #必选，Pod

metadata: #必选，元数据

name: string #必选，Pod名称

namespace: string #必选，Pod所属的命名空间

labels: #自定义标签

- name: string #自定义标签名字

annotations: #自定义注释列表

- name: string

spec: #必选，Pod中容器的详细定义

containers: #必选，Pod中容器列表

- name: string #必选，容器名称

image: string #必选，容器的镜像名称

imagePullPolicy: [Always | Never | IfNotPresent] #获取镜像的策略 Alawys表示下载镜像 IfnotPresent表示优先使用本地镜像，否则下载镜像，Nerver表示仅使用本地镜像

command: [string] #容器的启动命令列表，如不指定，使用打包时使用的启动命令

args: [string] #容器的启动命令参数列表

workingDir: string #容器的工作目录

volumeMounts: #挂载到容器内部的存储卷配置

- name: string #引用pod定义的共享存储卷的名称，需用volumes[]部分定义的的卷名

mountPath: string #存储卷在容器内mount的绝对路径，应少于512字符

readOnly: boolean #是否为只读模式

ports: #需要暴露的端口库号列表

- name: string #端口号名称

containerPort: int #容器需要监听的端口号

hostPort: int #容器所在主机需要监听的端口号，默认与Container相同

protocol: string #端口协议，支持TCP和UDP，默认TCP

env: #容器运行前需设置的环境变量列表

- name: string #环境变量名称

value: string #环境变量的值

resources: #资源限制和请求的设置

limits: #资源限制的设置

cpu: string #Cpu的限制，单位为core数，将用于docker run --cpu-shares参数

memory: string #内存限制，单位可以为Mib/Gib，将用于docker run --memory参数

requests: #资源请求的设置

cpu: string #Cpu请求，容器启动的初始可用数量

memory: string #内存清楚，容器启动的初始可用数量

livenessProbe: #对Pod内个容器健康检查的设置，当探测无响应几次后将自动重启该容器，检查方法有exec、httpGet和tcpSocket，对一个容器只需设置其中一种方法即可

exec: #对Pod容器内检查方式设置为exec方式

command: [string] #exec方式需要制定的命令或脚本

httpGet: #对Pod内个容器健康检查方法设置为HttpGet，需要制定Path、port

path: string

port: number

host: string

scheme: string

HttpHeaders:

- name: string

value: string

tcpSocket: #对Pod内个容器健康检查方式设置为tcpSocket方式

port: number

initialDelaySeconds: 0 #容器启动完成后首次探测的时间，单位为秒

timeoutSeconds: 0 #对容器健康检查探测等待响应的超时时间，单位秒，默认1秒

periodSeconds: 0 #对容器监控检查的定期探测时间设置，单位秒，默认10秒一次

successThreshold: 0

failureThreshold: 0

securityContext:

privileged:false

restartPolicy: [Always | Never | OnFailure]#Pod的重启策略，Always表示一旦不管以何种方式终止运行，kubelet都将重启，OnFailure表示只有Pod以非0退出码退出才重启，Nerver表示不再重启该Pod

nodeSelector: obeject #设置NodeSelector表示将该Pod调度到包含这个label的node上，以key：value的格式指定

imagePullSecrets: #Pull镜像时使用的secret名称，以key：secretkey格式指定

- name: string

hostNetwork:false #是否使用主机网络模式，默认为false，如果设置为true，表示使用宿主机网络

volumes: #在该pod上定义共享存储卷列表

- name: string #共享存储卷名称 （volumes类型有很多种）

emptyDir: {} #类型为emtyDir的存储卷，与Pod同生命周期的一个临时目录。为空值

hostPath: string #类型为hostPath的存储卷，表示挂载Pod所在宿主机的目录

path: string #Pod所在宿主机的目录，将被用于同期中mount的目录

secret: #类型为secret的存储卷，挂载集群与定义的secre对象到容器内部

scretname: string

items:

- key: string

path: string

configMap: #类型为configMap的存储卷，挂载预定义的configMap对象到容器内部

name: string

items:

- key: string



 备用：

apiVersion: v1

kind: Service

metadata:

name: servicename

labels:

name: servicename

spec:

type: NodePort

ports:

- port: 8080

targetPort: 8080

nodePort: 8080 #port和nodePort都是service的端口，前者暴露给k8s集群内部服务访问，后者暴露给

#k8s集群外部流量访问。从上两个端口过来的数据都需要经过反向代理kube-proxy，流

#入后端pod的targetPort上，最后到达pod内的容器。

name: http

protocol: tcp

selector:

name: servicename

补充：NodePort类型的service可供外部集群访问是因为service监听了宿主机上的端口，即监听了（所有节点）nodePort，该端口的请求会发送给service，service再经由负载均衡转发给Endpoints的节点。

————————————————

版权声明：本文为CSDN博主「yzh\_1346983557」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/yzh_1346983557/article/details/86717093>

      Service资源用于为pod对象提供一个固定、统一的访问接口及负载均衡的能力，并借助新一代DNS系统的服务发现功能，解决客户端发现并访问容器化应用的问题。

1、service资源实现模型

（1）service资源

       Service资源基于标签选择器将一组pod定义成一个逻辑组合，并通过自己的IP地址和端口调度代理请求至组内的对象上。并对客户端隐藏了真实的处理用户请求的pod资源。Service资源会通过API Service持续监视着标签选择器匹配到的后端pod对象，并实时跟踪个对象的变动；但是service并不直接连接至pod对象，它们直接还有一个中间层（Endpoint）资源对象，创建Service资源对象时，其关联的Endpoint对象会自动创建。

（2）service实现方式

       一个service对象就是工作节点上的一些iptables或ipvs规则，用于将到达service对象的IP地址的流量调度转发至相应的Endpoint对象制定的IP地址和端口上。Service ip就是用于生成iptables或ipvs规则时使用的IP地址。Kube-proxy将请求代理至相应端点的方式主要有三种：

       1）userspace代理模型

       这种代理模型为kubernetes1.1版本之前默认的代理模型；这种代理模型中，请求在内核空间和用户空间来回转发导致效率不高。

       2）iptables代理模型

       这种调度模型为kubernetes1.2版本至1.11版本之间的默认调度模型；默认算法是随机调度；iptables代理模型不会再背挑选中的后端pod资源无响应时自动进行重定向。

       3）ipvs调度模型

       Ipvs调度模型从1.11版本开始成为默认的调度模型；这种模型中，ipvs构建于netfilter的钩子函数之上，使用hash表作为底层数据结构并工作与内核空间，具有流量转发速度快，规则同步性能好等特性。IPvs支持支持众多调度算法，如rr、lc、dh、sh、sed、nq等。

2、service资源的使用

       Service资源的定义与其他资源的定义方式基本相同，spec字段经常嵌套的的字段有selector（定义标签选择器）ports（定义要暴露的端口）。

# 为前面创建的deployment控制器创建定义service资源

[root@master01 test03]# cat test01.yaml

apiVersion: v1

kind: Service

metadata:

  name: test-svc

spec:

  selector:

    app: test-deploy

  ports:

  - protocol: TCP

    port: 80

    targetPort: 80

# 创建service资源

[root@master01 test03]# kubectl apply -f test01.yaml

service/test-svc created

# 查看创建的service

[root@master01 test03]# kubectl get svc test-svc

NAME       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE

test-svc   ClusterIP   10.100.249.83   <none>        80/TCP    3m6s

# 查看endpoint资源端点列表

[root@master01 test03]# kubectl get endpoints test-svc

NAME       ENDPOINTS                                                  AGE

test-svc   10.244.1.38:80,10.244.1.39:80,10.244.2.37:80 + 2 more...   4m24s

3、服务发现

       在kubernetes集群中，service为pod中的服务类应用提供了一个稳定的访问入口，pod中的应用是通过服务发现机制来得知某个特定service资源的IP和端口的。

（1）服务发现

       根据服务发现的实现方式，可将服务发现分为客户端发现（由客户端到服务注册中心发现其依赖到的服务的相关信息）和服务端发现（通过中央路由器或者服务均衡器组件实现）两类。

       在kubernetes中，1.3版本开始服务发现是由kubeDNS实现的，从1.11版本后，服务发现功能是由CoreDNS组件实现的。

（2）服务发现的方式

       1）环境变量方式

       在创建pod资源时，kubelet会将其所属名称空间内的每个活动的Service对象以一系列环境变量的方式注入其中，支持使用kubernetes service环境变量及与docker的links兼容的变量。

       2）DNS的方式

       ClusterDNS是kubernetes系统中用于服务解析和名称发现的服务，集群中创建的每个service对象，都会由其生成相关的资源记录；默认集群内的各pod资源会自动配置其作为名称解析服务器，并在其DNS搜索列表中包含它所属名称空间的域名后缀。

       在创建service资源对象时，ClusterDNS会为它自动创建资源记录用于名称解析和服务注册，pod资源可直接使用标准的DNS名称来访问这些“servie”资源。

4、服务暴露

       Service的IP地址仅在集群内可达，而有些服务需要暴露到外部网络中接受各类客户端的访问，此时就需要在集群的边缘为其添加一层转发机制，以实现将外部请求接入到集群service资源之上，及将集群中的服务发布到外部网络中。Kubernetes中service共有四种类型，分别如下：

（1）ClusterIP

通过集群内部的IP地址暴露服务，地址仅在集群内部可达，无法被集群外部的客户端访问。

（2）NodePort

       NodePort类型建立在ClusterIP类型之上，用其所在的每个节点的IP地址的某静态端口暴露服务，它依然会为service分配集群IP地址，并将此作为NodePort的路由目标。

       Kubernetes集群在安装部署时通常会预留一个端口范围用于NodePort，默认为30000-322767之间的端口。而在使用NodePort类型的service时，需要通过” service.spec.type”字段指定service类型名称。

# 定义NodePort类型的service资源，并指定端口（一般不建议指定端口）

[root@master01 test03]# cat test02.yaml

apiVersion: v1

kind: Service

metadata:

  name: test-nodportsvc

spec:

  type: NodePort

  selector:

    app: test-deploy

  ports:

  - protocol: TCP

    port: 80

    targetPort: 80

    nodePort: 30033

# 查看创建好的service，依然创建了clusterIP，并将容器内的端口映射到了node节点上

[root@master01 test03]# kubectl get svc test-nodportsvc

NAME              TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE

test-nodportsvc   NodePort   10.111.192.243   <none>        80:30033/TCP   6m20s

（3）LoadBalancer

LoadBalancer类型建立在NodePort类型之上，其通过cloud provider提供的负载均衡器将服务暴露到集群外部。LoadBalancer类型的service与NodePort类型的service的使用方法基本类似。

（4）ExternalName

       ExternalNamel类型的service资源用于将集群外部的服务发布到集群中以供Pod中的应用程序访问，因此，不需要使用标签选择器关联任何的pod对象，但必须要用spec.externalName属性定义一个CNAME记录用于返回外部真正提供服务的主机的别名，而后通过CHAME记录获取到相关主机的IP地址。

5、Headless类型的Service资源

如果客户端需要直接访问Service资源后端的所有Pod资源，这时就应该向客户端暴露每个Pod资源的IP地址，而不再是中间层Service对象的ClusterIP，这种类型的service称为headless service。这种类型的service也没有使用负载均衡代理它的需要。

如何为Headless类型的Service资源配置IP地址取决于它的标签选择器的定义；如果具有标签选择器，则端点控制器会在API中为其创建Endpoint记录，并将ClusterDns服务中的A记录直接解析到此Service后端的各Pod对象的IP地址对象上；如果没有标签选择器，则端点控制器不会在API中为其创建Endpoints记录。

（1）创建Headless Service资源

       定义Headless Service类型的资源时，只需将ClusterIP字段的至设置为“None”即可。

# 定义Headless Service类型

[root@master01 test03]# cat test03.yaml

kind: Service

apiVersion: v1

metadata:

  name: test-headless-svc

spec:

  clusterIP: None

  selector:

    app: test-deploy

  ports:

  - name: httpport

    port: 80

    targetPort: 80

# 查看创建的headless service资源

[root@master01 test03]# kubectl get svc test-headless-svc

NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE

test-headless-svc   ClusterIP   None         <none>        80/TCP    62s

[root@master01 test03]# kubectl describe test-headless-svc

error: the server doesn't have a resource type "test-headless-svc"

[root@master01 test03]# kubectl describe svc test-headless-svc

. . . . . .

Endpoints:         10.244.1.38:80,10.244.1.39:80,10.244.2.37:80 + 2 more...

（2）pod资源发现

       Headless Service通过标签选择器关联到所有pod资源的IP地址之上，客户端向此service对象发起的请求将通过dns查询时返回的IP地址（多个IP地址则以轮询方式返回）直接接入到Pod资源中的应用之上，而不再由Service资源进行代理转发。

# 查看dns对此service名称的解析

[root@master01 test03]# kubectl exec -it cirros-58cc4cdc59-2f4s6 -- sh

/ # nslookup test-headless-svc

Server:    10.96.0.10

Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      test-headless-svc

Address 1: 10.244.2.39 10-244-2-39.test-headless-svc.default.svc.cluster.local

Address 2: 10.244.2.38 10-244-2-38.test-headless-svc.default.svc.cluster.local

Address 3: 10.244.2.37 10-244-2-37.test-headless-svc.default.svc.cluster.local

Address 4: 10.244.1.38 10-244-1-38.test-headless-svc.default.svc.cluster.local

Address 5: 10.244.1.39 10-244-1-39.test-headless-svc.default.svc.cluster.local

6、Ingress资源

       Kubernetes提供了两种负载均衡机制，一种是工作与传输层的Service资源，实现“TCP负载均衡器”，另一种是Ingress资源，实现“HTTP(S)负载均衡器”。

（1）Ingress和Ingress控制器

       Ingress是kubernetes API的标准类型资源之一，它其实是一组基于DNS名称或URL路劲把请求转发至指定的Service资源的规则，用于将集群外部的请求转发至集群内部完成服务的发布。

       Ingre资源并不能进行“能量穿透”，它仅是一组路由规则的集合，而能够为Ingress资源监听套接字并转发流量的组件称为Ingress控制器（Ingress Controller）。

       Ingress控制器可以由任何具有反向代理功能的服务程序实现，Ingress也是运行于集群中的Pod资源，应与被代理的pod运行与同一网络中。

（2）Ingress资源的定义

       Ingress资源是基于HTTP虚拟主机或URL的转发规则，Ingress.spec资源是定义Ingress资源的核心组成部分，它主要由以下嵌套字段组成：

           rules：用于定义当前Ingress资源的转发规则列表，未定义的rule规则或者匹配不到  任何规则时，所有的流量都会转发到backend定义的默认后端。

           backend：默认的后端用于服务那些没有匹配到任何规则的请求。Backend对象的   定义由两个必选的内嵌字段serviceName和servicePort组成。

           tls：目前仅支持通过默认端口443提供服务，tls也有以下两个内嵌字段组成，仅     在定义TLS的转发规则时才需要定义此类对象：

              hosts：包含于使用的TLS证书之内的主机名称字符串列表

             secretName：用于引用SSL回话的secret对象名称，在基于SNI实现多主机路由的场景中，此字段为可选。

# 定义ingress资源

[root@master01 test03]# cat test04.yaml

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

  name: test-ingress

  annotations:

    kubernetes.io/ingress: “nginx test”

spec:

  rules:

  - host: www.dayi123.com

    http:

      paths:

      - backend:

          serviceName: test-headless-svc

          servicePort: 80

（3）Ingress资源的类型

       1）单service资源型ingress

        暴露单个服务的方法可以使用service的NodePort、LoadBalancer类型；也可使用Ingress,使用Ingress时，只需指定ingress的default backend即可。

# 定义一个单Service资源型的ingress

[root@master01 test03]# cat test05.yaml

apiVersion: extensions/v1beat1

kind: Ingress

metadata:

  name: test-ingress02

spec:

  backend:

    serviceName: nginx-svc

    serviceName: 80

       2）基于URL路劲进行流量分发的ingress

        基于URL路径转发是根据客户端请求不同的URL路径转发到不同的后端服务中。

# 定义一个多路径的ingress资源

[root@master01 test03]# cat test06.yaml

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

  name: test-ingress05

  annotations:

    kubernetes.io/ingress: “nginx test path”

spec:

  rules:

  - host: www.dayi123.com

    http:

      paths:

      - path: /nginx

        backend:

          serviceName: nginx-svc

          servicePort: 80

      - path: /tomcat

        backend:

          serviceName: test-svc

          servicePort: 80

       3）基于主机名称的ingress虚拟主机

         基于主机名的ingress虚拟主机是将每个应用分别以独立的FQDN主机名进行输出。

# 定义一个基于主机名的service

[root@master01 test03]# cat test07.yaml

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

  name: test-ingress06

spec:

  rules:

  - host: blog.dayi123.com

    http:

      paths:

      - backend:

          serviceName: nginx-svc

          servicePort: 80

  - host: doc.dayi123.com

    http:

      paths:

      - backend:

          serviceName: test-svc

          servicePort: 80

       4）TLS类型的Ingress资源

         TLS类型的Ingress资源用于以HTTPS发布Service资源，基于一个含有私钥和证书的Secret对象即可配置TLS协议的Ingress资源。Ingress资源目前仅支持单TLS端口，并且还会卸载TLS回话。

（4）部署基于nginx的Ingress控制器

         Ingress是自身运行于Pod中的容器应用，一般是nginx或Envoy一类的具有代理及负载均衡功能的守护进程，它监视着来自于API Service的Ingress对象状态，并以其规则生

       相应的应用程序专有格式的配置文件并通过重载或重启守护进程而使新配置生效。运行Pod资源的Ingress控制器接入外部请求有以下两种方法：

       1）以Deployment控制器管理Ingress控制器的Pod资源，并通过NodePort或LoadBalancer类型的Service对象为其接入集群外部的请求流量

       2）借助于DaemonSet控制器，将Ingress控制器的Pod资源各自以单一实力的方式运行于集群的所有或部分工作节点，并配置该类pod以hostPort或hostNetwork的方式在当前节点接入外部流量。

# 在线创建nginx ingress控制器

[root@master01 ~]# kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/mandatory.yaml

# 查看创建的nginx ingress控制器是否运行正常

[root@master01 ~]# kubectl get pods -n ingress-nginx

NAME                                       READY   STATUS    RESTARTS   AGE

nginx-ingress-controller-c595c6896-6xcd5   1/1     Running   0          92m

       在线的nginx ingress配置清单中采用了基于deployment控制器部署方式，因此接入外部流量之前需要通过NodePort或LoadBalancer类型的service资源对象。

# 为nginx ingress定义service资源配置清单

[root@master01 test03]# cat test-nginx-ingress.yaml

apiVersion: v1

kind: Service

metadata:

  name: nginx-ingress-controller

  namespace: ingress-nginx

spec:

  type: NodePort

  ports:

    - name: http

      port: 80

    - name: https

      port: 443

  selector:

     app.kubernetes.io/name: ingress-nginx

# 为nginx ingress控制器创建service

[root@master01 test03]# kubectl apply -f test-nginx-ingress.yaml

service/nginx-ingress-controller created

# 查看创建的service

[root@master01 test03]# kubectl get svc -n ingress-nginx

NAME                       TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE

nginx-ingress-controller   NodePort   10.105.134.97   <none>        80:30409/TCP,443:31235/TCP   16s

7、使用ingress发布Tomcat

       实验的拓扑如下

（1）创建名称空间

# 定义名称空间的资源配置清单

[root@master01 ingress-tomcat]# cat ingress-tomcat.yaml

apiVersion: v1

kind: Namespace

metadata:

  name: test-ingress

  labels:

    env: test-ingress

# 创建名称空间

[root@master01 ingress-tomcat]# kubectl apply -f ingress-tomcat.yaml

namespace/test-ingress created

（2）部署tomcat实例

# 定义基于deployment的tomcat资源配置清单

[root@master01 ingress-tomcat]# cat tomcat-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: tomcat-deployment

  namespace: test-ingress

spec:

  replicas: 2

  selector:

    matchLabels:

      app: ingress-tomcat

  template:

    metadata:

      labels:

        app: ingress-tomcat

    spec:

      containers:

      - name: tomcat

        image: tomcat

        ports:

        - name: httport

          containerPort: 8080

        - name: ajpport

          containerPort: 8009

# 创建tomcat的pod控制器

[root@master01 ingress-tomcat]# kubectl apply -f tomcat-deployment.yaml

deployment.apps/tomcat-deployment created

# 查看创建的tomcat pod

[root@master01 ingress-tomcat]# kubectl get pods -n test-ingress

NAME                                 READY   STATUS    RESTARTS   AGE

tomcat-deployment-67f5dcbdb4-7mmhz   1/1     Running   0          96s

tomcat-deployment-67f5dcbdb4-km5q8   1/1     Running   0          96s

（3）创建Service资源

        Ingress资源仅能通过Service资源识别相应的Pod资源，获取其IP地址和端口，然后Ingress控制器即可直接使用各Pod对象的IP地址与Pod内的服务直接进行通信，不经过Service的代理和调度，因此Service资源的ClusterIP对Ingress控制器来说存不存在无所谓。

# 定义service的资源配置清单文件

[root@master01 ingress-tomcat]# cat tomcat-service.yaml

apiVersion: v1

kind: Service

metadata:

  name: tomcat-svc

  namespace: test-ingress

  labels:

    app: tomcat-svc

spec:

  selector:

    app: ingress-tomcat

  ports:

  - name: http

    port: 80

    targetPort: 8080

    protocol: TCP

# 创建并查看service

[root@master01 ingress-tomcat]# kubectl apply -f tomcat-service.yaml

service/tomcat-svc created

[root@master01 ingress-tomcat]# kubectl get svc -n test-ingress

NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE

tomcat-svc   ClusterIP   10.106.153.248   <none>        80/TCP    15s

（4）创建Ingress资源

# 定义ingress资源配置清单

[root@master01 ingress-tomcat]# cat tomcat-ingress.yaml

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

  name: tomcat

  namespace: test-ingress

  annotations:

    kubernetes.io/ingress: "nginx"

spec:

  rules:

  - host: tomcat.dayi123.com

    http:

      paths:

      - path:

        backend:

          serviceName: tomcat-svc

          servicePort: 80

# 创建并查看Ingress

[root@master01 ingress-tomcat]# kubectl apply -f tomcat-ingress.yaml

ingress.extensions/tomcat created

[root@master01 ingress-tomcat]# kubectl get ingress -n test-ingress

NAME     HOSTS                ADDRESS   PORTS   AGE

tomcat   tomcat.dayi123.com             80      14s

        上面的配置完成后，在本地主机的hosts中添加tomcat.dayi123.com对应node节点的解析后，就可在本地通过http://tomcat.dayi123.com:30409去访问tomcat服务。而各node节点的30409端口是在前面为nginx类型的ingress控制器创建的service中定义的。

（5）配置TLS Ingress资源

         互联网中的服务基本都是以https的方式提供服务的，如果希望ingress控制器接受客户端的请求时又希望它能够提供https服务，就应该配置tls类型的ingress资源。

# 生成用于测试的私钥和自签证书

[root@master01 ingress-tomcat]# openssl genrsa -out tls.key 2048

[root@master01 ingress-tomcat]# openssl req -new -x509 -key tomcat.key -out tomcat.crt -subj /C=CN/ST=Shanghai/L=Shanghai/O/dev/CN=tomcat.dayi123.com -days 736

        在ingress控制器上配置HTTPS主机时，是不能直接使用私钥和证书文件的，而是要使用Secret资源对象来传递相关的数据。

# 创建一个TLS类型名为tomcat-ingress-secret的secret资源

[root@master01 ingress-tomcat]# kubectl create secret tls tomcat-ingress-secret --cert=tomcat.crt --key=tomcat.key -n test-ingress

secret/tomcat-ingress-secret created

# 查看创建的secret资源

[root@master01 ingress-tomcat]# kubectl get secrets tomcat-ingress-secret -n test-ingress

NAME                    TYPE                DATA   AGE

tomcat-ingress-secret   kubernetes.io/tls   2      109s

        secret资源创建完成后，就可以将创建的secret应用到ingress资源的配置清单中。

# 定义TLS类型的Ingress资源的配置清单

[root@master01 ingress-tomcat]# cat tomcat-ingress-tls.yaml

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

  name: tomcat-ingress-tls

  namespace: test-ingress

  annotations:

    kubernetes.io/ingress: "nginx"

spec:

  tls:

  - hosts:

    - tomcat.dayi123.com

    secretName: tomcat-ingress-secret

  rules:

  - host: tomcat.dayi123.com

    http:

      paths:

      - path:

        backend:

          serviceName: tomcat-svc

          servicePort: 80

# 创建并查看tls类型的ingress资源

[root@master01 ingress-tomcat]# kubectl apply -f tomcat-ingress-tls.yaml

ingress.extensions/tomcat-ingress-tls created

[root@master01 ingress-tomcat]# kubectl get ingress tomcat-ingress-tls -n test-ingress

NAME                 HOSTS                ADDRESS   PORTS     AGE

tomcat-ingress-tls   tomcat.dayi123.com             80, 443   3m35s

（6）测试

        在前面为基于nginx的ingress控制器创建的service中已将443端口映射到了node节点的31235端口，在本地对域名tomcat.dayi123.com做了解析就可以通过https://tomcat.dayi123.com: 31235进行测试。

# 查看ingress控制器中映射到本地的端口

[root@master01 ingress-tomcat]# kubectl get svc -n ingress-nginx

NAME                       TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE

nginx-ingress-controller   NodePort   10.105.134.97   <none>        80:30409/TCP,443:31235/TCP   167m

# 在linux客户端进行测试（测试前要做hosts解析）

[root@master01 ingress-tomcat]# curl -k -v https://tomcat.dayi123.com:31235

————————————————

版权声明：本文为CSDN博主「dayi\_123」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/dayi_123/article/details/89703961>

1、kubernetes存储卷

（1）存储卷

        Kubernetes提供的存储卷属于pod资源级别，共享于Pod内的所有容器，可用于在容器的文件系统之外存储应用存储的相关数据，也可独立pod生命周期之外实现数据的持久化。

（2）kubernetes支持的存储卷类型

        Kubernetes支持非常丰富的出处卷类型，包括本地储存及各种网络存储，同时还支持Secret和ConfigMap这样的特殊存储资源。

        由于网络存储配置使用复杂，对此，kubernetes设计了一种集群级别的资源PersistentVolume(PV)并由管理员配置存储系统，然后通过persistentVolumeClain（PVC）存储卷直接申请使用的机制简化了存储的使用配置过程。

（3）存储卷的使用方式

        在pod中定义配置存储卷由两部分组成，一是通过pod.spec.volumes字段定义在Pod之上的存储卷列表；然后通过pod.spec.containers.volumeMount字段在容器上定义存储卷挂载列表，在容器中只能挂载在当前Pod资源中定义的具体存储卷。

        在容器中定义挂载卷时的主要字段：

            name：必选字段，指定要挂载的存储的名称，

            mountPath：必选字段，挂载路径，即挂载容器的那个路径下

            readOnly：是否挂载为只读卷

           subPath：挂载存储卷时时使用的子路径

2、临时存储卷

（1）emptyDir

         emptyDir是pod生命周期中的一个临时目录，生命周期跟随pod的生命周期；emptyDir存储卷通过pod.spec.volumes.emptyDir字段定义，主要的字段如下：

            medium：存储介质的类型，值有”default”和“memory”,默认为default。

            SizeLimit：当前存储卷的空间限额，默认值为nil,不限制。

# 定义一个自助式pod，定义一个emptyDir存储，定义两个容器并共享存储

[root@master01 volumes]# cat test-emptydir.yaml

apiVersion: v1

kind: Pod

metadata:

  name: test-emptydir-dir

spec:

  volumes:

  - name: html

    emptyDir: {}

  containers:

  - name: nginx

    image: nginx:1.12

    volumeMounts:

    - name: html

      mountPath: /usr/share/nginx/html

  - name: busybox

    image: busybox

    volumeMounts:

    - name: html

      mountPath: /html

    command: ["/bin/sh", "-c"]

    args:

    - echo $(date) >> /html/index.html

（2）gitRepo存储卷

       gitRepo存储卷可以看作是emptyDir存储卷的一种实际应用，但是该存储卷从kubernetes1.12开始已经被废弃。在创建使用gitRepo存储卷的pod时，首先会创建一个空目录（emptyDir）并克隆一份指定的git仓库中的数据至该目录下，然后再创建容器并挂载该存储卷。

       创建gitRepo类型的存储卷时，主要的字段如下：

              repository：必选字段，为Git仓库的URL

              directory：目标目录的名称

              revision：特定revision的提交哈希码

3、节点存储卷

       hostPath类型的存储卷是指将工作节点上的某文件系统的目录或文件挂载于Pod中的一种存储卷，它可独立于Pod资源的生命周期，因而具有持久性，但是它是工作在本地节点的存储空间，仅适用于特定情况下的存储卷使用要求。

      定义配置hostPath类型的存储卷时，主要是通过pod.spec.volumes.hostPath字段嵌套的path（用于指定工作节点上的目录路径）字段及type（用于指定存储卷的类型）字段来定义的，type值有：

           DirectoryOrCreate：指定的路径不存在时自动创建

           Directory：必须存在的目录路径

           FileOrCreate：指定的路径不存在时自动将其创建为空文件

           File：必须存在的文件路径

           Socket：必须存在的Socket文件路径

           CharDevice：必须存在的字符设备文件路径

           BlockDevice：必须存在的块设备文件路径

# 定义一个使用hostpath存储卷的自助式pod

[root@master01 volumes]# cat test-hostpath.yaml

apiVersion: v1

kind: Pod

metadata:

  name: test-hostpath-vol

spec:

  volumes:

  - name: html

    hostPath:

      path: /data/test/html

      type: DirectoryOrCreate

  containers:

  - name: nginx

    image: nginx:1.12

    volumeMounts:

    - name: html

      mountPath: /usr/share/nginx/html

# 创建并查看该pod的详细信息

[root@master01 volumes]# kubectl apply -f test-hostpath.yaml

pod/test-hostpath-vol created

[root@master01 volumes]# kubectl get pods -o wide test-hostpath-vol

NAME                READY   STATUS    RESTARTS   AGE     IP            NODE                 NOMINATED NODE   READINESS GATES

test-hostpath-vol   1/1     Running   0          2m49s   10.244.2.46   node02.dayi123.com   <none>           <none>

# 在该pod所在的节点上创建测试文件并测试

[root@node02 ~]# echo "$(date +%F-%T) test-volumes-hostpath" > /data/test/html/index.html

[root@node02 ~]# curl 10.244.2.46

2019-01-24-00:51:00 test-volumes-hostpath

4、网络存储器

       Kubernetes支持众多类型的网络存储卷，包括传统的NAS或SAN设备（如NFS、Iscsi、fc）、分布式存储（如RDB、GlusterFS）、云端存储（如cinder、awsElasticBlockStore、azureDisk）及构建在各类存储系统之上的抽象管理层。

（1）NFS存储卷

       Kubernetes的NFS存储卷用于将某实现存放的NFS服务器上的存储空间挂载到Pod中以供容器使用，NFS存储卷在pod对象终止后会被卸载并不会被删除。

       NFS存储卷的定义是通过pod.spec.volumes.nfs地段定义，主要的嵌套字段有：

              server：NFS服务器的IP地址或主机名

              path：NFS服务器导出的文件系统路径

              readOnly：是否以制度方式挂载

       在使用nfs存储卷之前首先得有一个nfs服务器；先搭建一个nfs服务器，IP地址为192.168.17.195，主机名为nfs01.dayi123.com,并在各节点主机hosts文件中做解析。

# 安装nfs服务

[root@nfs01 ~]# yum install nfs-utils

# 配置nfs服务

[root@nfs01 ~]# cat /etc/exports

/data/volums/data01 \*(async,insecure,no\_root\_squash,no\_subtree\_check,rw)

# 创建nfs挂载目录

[root@nfs01 ~]# mkdir /data/volums/data01/ -p

# 启动服务

[root@nfs01 ~]# systemctl start rpcbind

[root@nfs01 ~]# systemctl enable rpcbind

[root@nfs01 ~]# systemctl start nfs-server

[root@nfs01 ~]# systemctl enable nfs-server

# 查看nfs状态

[root@nfs01 ~]# showmount -e 192.168.17.195

Export list for 192.168.17.195:

/data/volums/data01 192.168.17.0/24

       存在nfs网络文件系统后，就可以在各pod中使用nfs存储卷，nfs是文件系统级的共享服务，它支持同时存在的多路径挂载请求，但在使用nfs时需要在各节点安装nfs客户端工具（只需安装nfs-utils，不需要启动）

# 创建一个使用nfs存储卷的自助式运行nginx的pod

[root@master01 volumes]# cat test-nginx-nfsvol.yaml

apiVersion: v1

kind: Pod

metadata:

  name: test-nginx-nfsvol

spec:

  volumes:

  - name: html

    nfs:

      server: nfs01.dayi123.com

      path: /data/volums/data01

      readOnly: false

  containers:

  - name: nginx

    image: nginx:1.12

    volumeMounts:

    - name: html

      mountPath: /usr/share/nginx/html

# 创建并查看pod的详细信息

[root@master01 volumes]# kubectl apply -f test-nfs-vol.yaml

[root@master01 volumes]# kubectl get pods test-nginx-nfsvol -o wide

NAME                READY   STATUS    RESTARTS   AGE   IP            NODE                 NOMINATED NODE   READINESS GATES

test-nginx-nfsvol   1/1     Running   0          14m   10.244.2.52   node02.dayi123.com   <none>           <none>

# 在nfs服务挂载的目录上写入数据通过nginx测试

[root@master01 volumes]# kubectl exec -it test-nginx-nfsvol -- /bin/sh

# echo "$(date +%F-%T) $(hostname) - TEST-NGINX-NFSVOLMS" >/usr/share/nginx/html/index.html

[root@master01 volumes]# curl 10.244.2.52

2019-01-24-07:45:46 test-nginx-nfsvol - TEST-NGINX-NFSVOLMS

       由于nfs存储卷保存的数据是永久性的，所以删除使用nfs存储卷的pod重新创建后数据不会丢失。

# 删除并重新创建该pod并进行测试

[root@master01 volumes]# kubectl delete -f test-nginx-nfsvol.yaml

[root@master01 volumes]# kubectl apply -f test-nginx-nfsvol.yaml

[root@master01 volumes]# kubectl get pods test-nginx-nfsvol -o wide

NAME                READY   STATUS    RESTARTS   AGE   IP            NODE                 NOMINATED NODE   READINESS GATES

test-nginx-nfsvol   1/1     Running   0          16s   10.244.2.53   node02.dayi123.com   <none>           <none>

[root@master01 volumes]# curl 10.244.2.53

2019-01-24-07:45:46 test-nginx-nfsvol - TEST-NGINX-NFSVOLMS

（2）RBD存储卷的使用

       Kubernetes支持通过RBD卷类型使用ceph存储系统为Pod提供存储卷，要配置Pod资源使用RBD存储卷，需要满足以下的条件：

       1）存在可用的ceph集群

       2）需要在ceph rbd集群中创建一个能满足Pod资源需求数据存储需要的存储映像

       3）在kubernetes集群内的各节点上安装ceph客户端程序包

       RBD类型的存储卷是通过explain pod.spec.volumes.rbd字段定义的，主要的内嵌字段有：

              monitors：必选字段，为ceph存储监视器，使用逗号分隔的字符串列表

              image：必选字段，为rados image的名称

              pool：rados存储池的名称，默认为RBD

              user：rados用户名，默认为admin

              keyring：RBD用户认证时使用的keyring文件路径，默认为/etc/ceph/keyring

              secretRef：RBD用户认证时使用的保存有相应认证信息的Secret对象，会覆盖由keyring字段提供的秘钥信息

              readOnly：是否以只读方式进行访问

              fsType：要挂载的存储卷的文件系统类型

5、持久存储卷

       直接使用自助式pod直接连接存储的方式与kubernetes向用户和开发隐藏底层架构的目标相背离；对此，kubernetes使用PersistentVolume（简称PV）子系统在系统和管理员之间添加一个自系统，解决了这种问题。

       PersistentVolume是由集群管理员配置提供的某存储系统上的一段存储空间，是对底层共享存储的抽象，将共享存储作为一种可由用户申请使用的资源。用户对pv资源的使用是通过PersistentVolumeClaim（简称PVC）提出使用申请来完成绑定，它向PV申请特定大小的空间及访问模式，从而创建出PVC存储卷，再由pod资源通过PVC存储卷关联使用。

（1）PV的创建

      PV对存储系统的创建可通过插件来实现，kubernetes可支持的插件可通过” kubectl explain PersistentVolume.spec”命令查看。Pv的资源定义清单中PersistentVolume.spec字段除了定义存储插件外，还可以定义的通用字段有：

       1）Capacity：当前pv的容量

       2）accessModes：定义访问模式，常用的值有：ReadWriteOnce（仅可被单个节点读写挂载，简写为RWO）、       ReadOnlyMany（可被多个节点制度挂载，简写为ROX）、ReadWriteMany（可悲多个节点只读挂载，简写为RWX）

       3）persistentVolumeReclaimPolicy：PV空间被释放时的处理机制，选项有Retain（默认值,由管理员手动回收）、Recycle（空间回收，删除存储卷下所有文件，目前仅NFS和hostPath支持）、Delete（删除从存储卷）

       4）volumeMode：卷模型，用于此卷可被用作文件系统还是块设备，默认为filesystem

       5）storageClassName：当前pv所属的StorageClass的名称，默认空值。

       6）mountOptions：挂载选项组成的列表，值有ro、soft、hard等

# 在创建爱你pv资源之前，查看可挂载的nfs存储目录

[root@master01 volumes]# showmount -e nfs01.dayi123.com

Export list for nfs01.dayi123.com:

/data/volums/vol03  192.168.17.0/24

/data/volums/vol02  192.168.17.0/24

/data/volums/vol01  192.168.17.0/24

/data/volums/data01 192.168.17.0/24

# 定义pv资源配置清单

[root@master01 volumes]# cat test-pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

  name: test-pv01

  labels:

    name: pv01

spec:

  nfs:

    server: nfs01.dayi123.com

    path: /data/volums/vol01

  accessModes: ["ReadWriteMany"]

  capacity:

    storage: 2Gi

  volumeMode: Filesystem

---

apiVersion: v1

kind: PersistentVolume

metadata:

  name: test-pv02

  labels:

    name: pv02

spec:

  nfs:

    server: nfs01.dayi123.com

    path: /data/volums/vol02

  accessModes: ["ReadWriteOnce"]

  capacity:

    storage: 5Gi

  volumeMode: Filesystem

---

apiVersion: v1

kind: PersistentVolume

metadata:

  name: test-pv03

  labels:

    name: pv03

spec:

  nfs:

    server: nfs01.dayi123.com

    path: /data/volums/vol03

  accessModes: ["ReadWriteMany","ReadWriteMany"]

  capacity:

    storage: 10Gi

  volumeMode: Filesystem

# 创建并查看pv

[root@master01 volumes]# kubectl apply -f test-pv.yaml

[root@master01 volumes]# kubectl get pv

NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE

test-pv01   2Gi        RWX            Retain           Available                                   5s

test-pv02   5Gi        RWO            Retain           Available                                   5s

test-pv03   10Gi       RWX            Retain           Available                                   5s

（2）创建PVC

       PVC是存储卷类型的资源，它通过申请占用某个pv而创建，它与pv是一对一的关系，创建pvc申请pv时用户需要指定目标空间大小，访问模式，pv标签选择器和StoreClass等信息，pvc的spec字段嵌套的字段有：

       accessMode：当前pvc的访问模式，可用模式与PV相同

       resources：当前PVC存储卷需要占用的资源量的最小值

       selector：绑定时对PV应用的标签选择器或匹配条件表达式，用于挑选要绑定的pv

       storageClassVolume：所依赖的存储类的名称

       volumeMode：卷模型，默认为Filesystem

       volumeName：用于直接指定要绑定的PV的卷名

# 定义一个pvc资源，让它自动找合适的pv绑定

[root@master01 volumes]# cat test-pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: test-pvc01

spec:

  accessModes: ["ReadWriteMany"]

  resources:

    requests:

      storage: 5Gi

# 创建并查看创建的PVC

[root@master01 volumes]# kubectl apply -f test-pvc.yaml

[root@master01 volumes]# kubectl get pvc

NAME         STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE

test-pvc01   Bound    test-pv03   10Gi       RWX                           85s

# 查看pv的绑定状态

[root@master01 volumes]# kubectl get pv

NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM                STORAGECLASS   REASON   AGE

test-pv01   2Gi        RWX            Retain           Available                                                38m

test-pv02   5Gi        RWO            Retain           Available                                                38m

test-pv03   10Gi       RWX            Retain           Bound       default/test-pvc01                           38m

（3）在pod中使用PVC

       创建好PVC资源后，可在Pod中通过” pod.spec.volumes.persistentVolumeClaim”字段去调用PVC,调用pvc时只要有以下的嵌套字段：

       claimName：要调用的PVC的存储卷的名称，pvc要与pod在同一名称空间中

       readOnly：是否将存储卷强制挂载为制度模式，默认为false。

# 定义一个pod资源调用pvc存储

[root@master01 volumes]# cat test-pod-pvc.yaml

apiVersion: v1

kind: Pod

metadata:

  name: test-pod-pvc

spec:

  containers:

  - name: pvc-nginx

    image: nginx:1.12

    volumeMounts:

    - name: html

      mountPath: /usr/share/nginx/html

  volumes:

  - name: html

    persistentVolumeClaim:

      claimName: test-pvc01

（4）存储类

       Storage class（存储类）是kubernetes资源类型的一种，是由管理员为管理PV之便而按需创建的类别（逻辑组）。存储类支持PV的动态创建，当用户用到持久性存储时，需要创建pvc来绑定匹配的pv，当管理员手动创建的pv无法满足pvc的所有需求时，系统会按pvc的需求标准动态创建适配的pv。

1）存储类的创建

       存储类资源的定义除了名称外，还需要定义的以下的三个关键字：

           provisioner（供给方）：提供给存储资源的存储系统

parameters：存储类使用参数描述要关联到的存储卷，不同的provisioner可用的参数各不相同

           reclaimPolicy：为当前存储类活动动态创建的PV指定回收策略，可用值为Delete（默认值）和Retain

           volumeBindingMode：定义如何为PVC完成供给和绑定

           mountOptions：由当前类动态创建的PV的挂载选项列表

2）动态pv的供给

      存储类创建完成后，在PVC的定义中指定使用的存储类资源的方式有两种，一是通过spec.storageClassName字段定义，另一种是使用volume.beta.kubernetes.io/storage-class注解信息。

————————————————

版权声明：本文为CSDN博主「dayi\_123」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/dayi\_123/article/details/89839544