目录

[前言 2](#_Toc3588)

[一、Docker制作镜像并推到远程仓库 2](#_Toc32701)

[1、 制作jar包 2](#_Toc2874)

[2、 编写Dockerfile文件 2](#_Toc3579)

[3、 docker本地验证根据镜像创建容器 3](#_Toc127)

[4、 给镜像打tag 3](#_Toc14665)

[5、 镜像推到远程 3](#_Toc7127)

[二、 利用k8s根据镜像创建容器 5](#_Toc21401)

[2.1 根据镜像创建pod 5](#_Toc15231)

[2.2 根据deployment创建 5](#_Toc3583)

[2.2.1、根据deployment文件创建 5](#_Toc4312)

[2.2.2 暴露服务service 6](#_Toc3691)

[三、 附录 7](#_Toc25546)

[1、Dockerfile源码 7](#_Toc6165)

[2、 myjavaapp-deployment.yaml源码 7](#_Toc16876)

[3、 myjavaapp-service.yaml源码 8](#_Toc1137)

# 前言

本文主要基于SpringBootDemo项目，利用docker制作镜像、推到远程仓库，利用docker

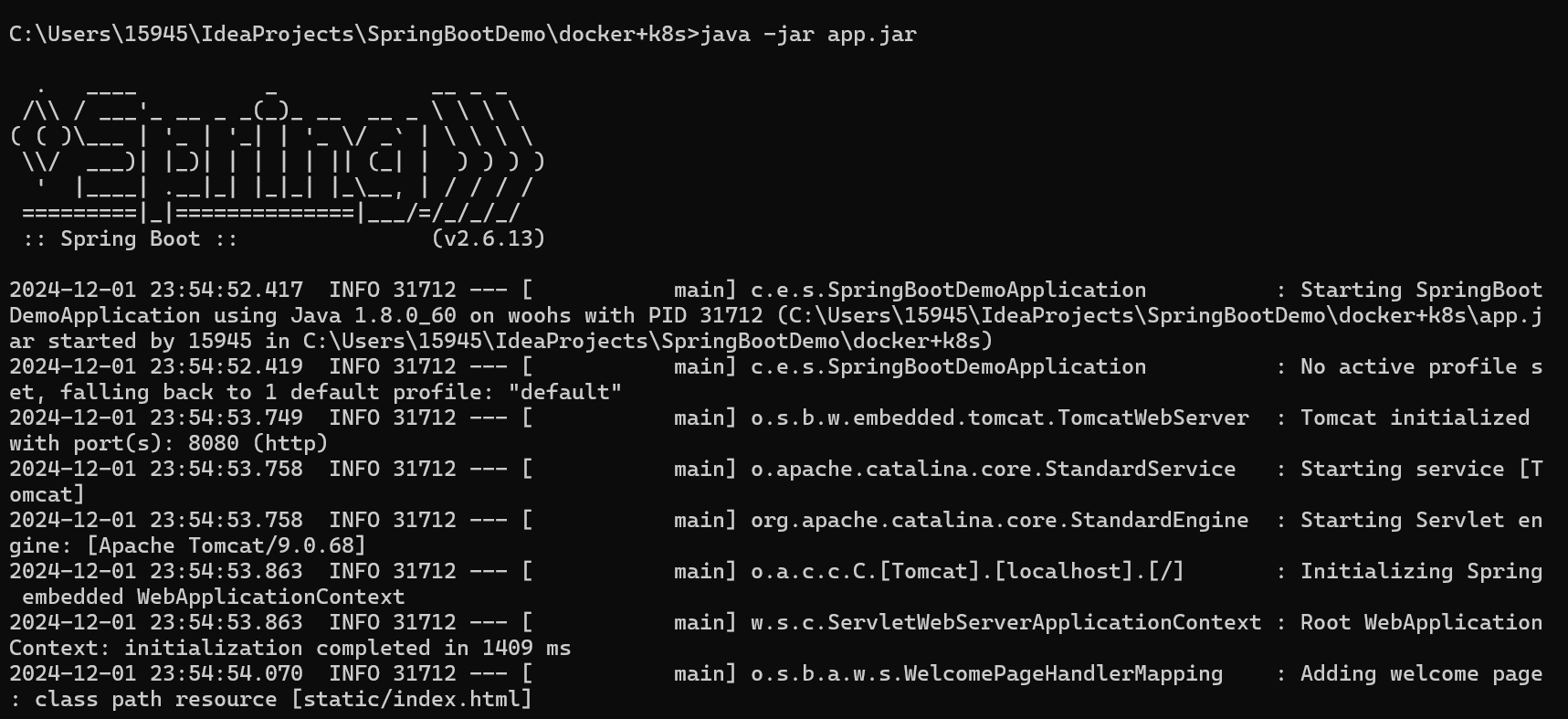
基于创建的镜像启动容器，以及利用k8s基于创建的镜像启动容器。

环境：docker+k8s+jdk8

# 一、Docker制作镜像并推到远程仓库

## 制作jar包

将SpringBootDemo生成对应的jar包，并更名为 app.jar（已有）。这一步可以先验证下jar包是否可用，在 app.jar 的目录，启动cmd命令窗口，在命令窗口输入 java -jar app.jar



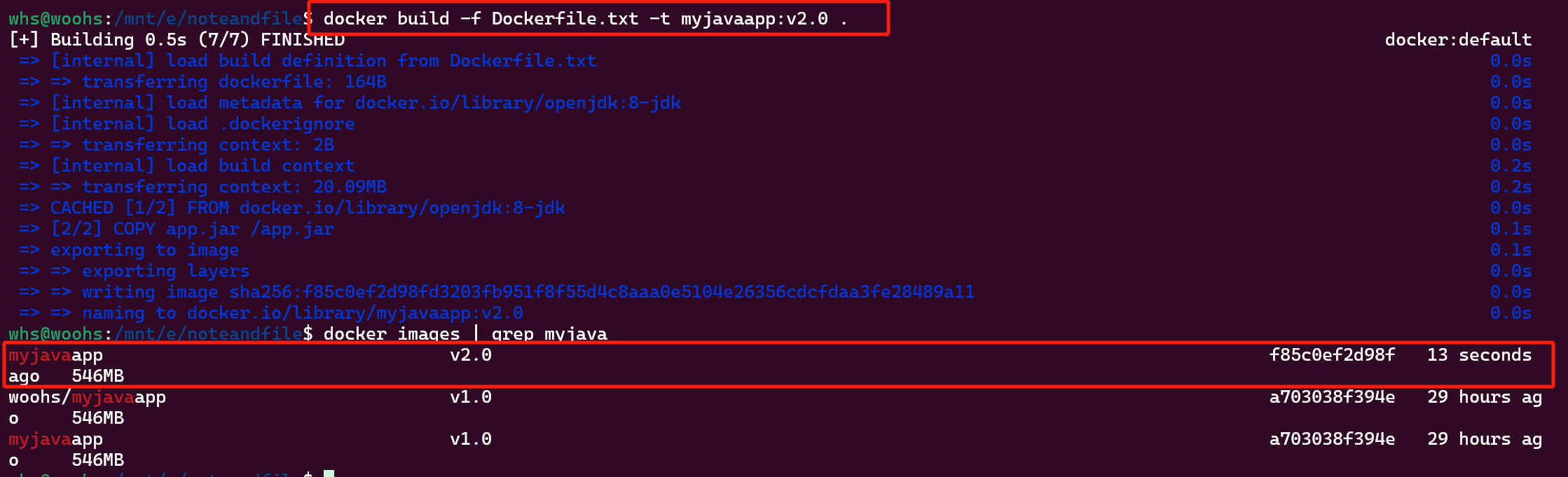
其中，可通过浏览器中输入localhost:8080/html 或者 localhost:8080/hello?name=xxx来访问验证服务启动以及看效果

## 编写Dockerfile文件

将Dockerfile文件与app.jar 放在同一个文件夹下（Dockerfile文件在附录1），输入命令：

“docker build -f Dockerfile.txt -t myjavaapp:v2.0 .”后回车，等待镜像构建。

查看镜像输入命令：“docker images | grep myjava”

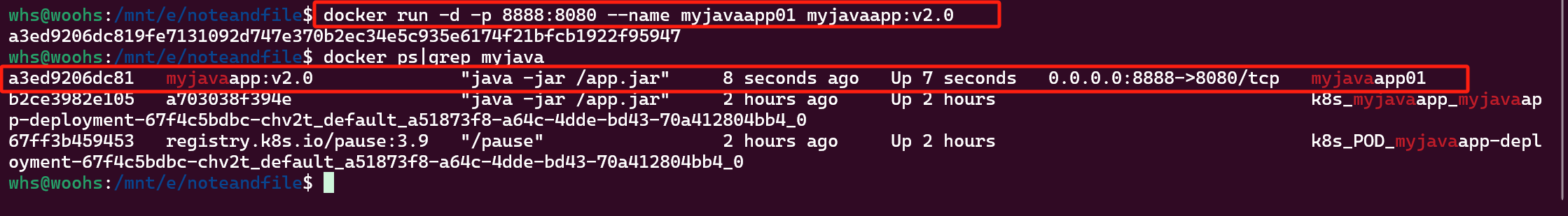


## docker本地验证根据镜像创建容器

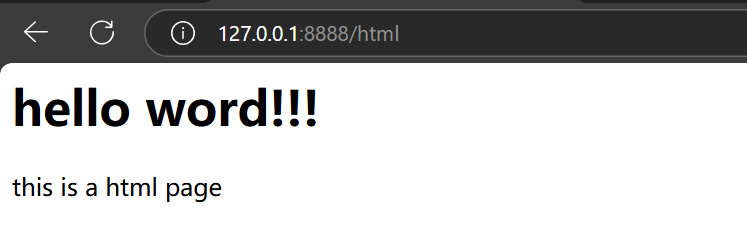
利用镜像myjavaapp:v2.0创建容器，容器名称为myjavaapp01,主机端口为8888，与容器内部8080映射。

输入命令： docker run -d -p 8888:8080 --name myjavaapp01 myjavaapp:v2.0

查看启动的容器：docker ps|grep myjava



浏览器中输入：[127.0.0.1:8888/html](http://127.0.0.1:8888/html)



## 给镜像打tag

命令：docker tag [要推到远程的进行名称]:版本 [docker-hub的用户名]/[远程镜像名称]:版本

把这个镜像：myjavaapp/v2.0 推到远程仓库，并且远程仓库名为woohs/myjavaapp:v2.0

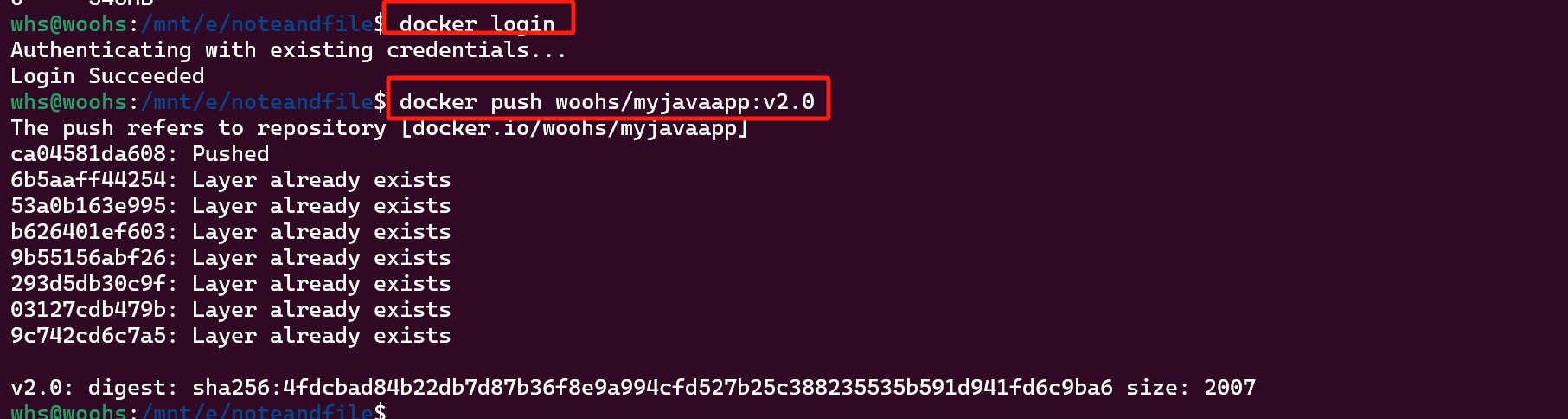
输入命令docker tag myjavaapp:v1.0 woohs/myjavaapp:v1.0



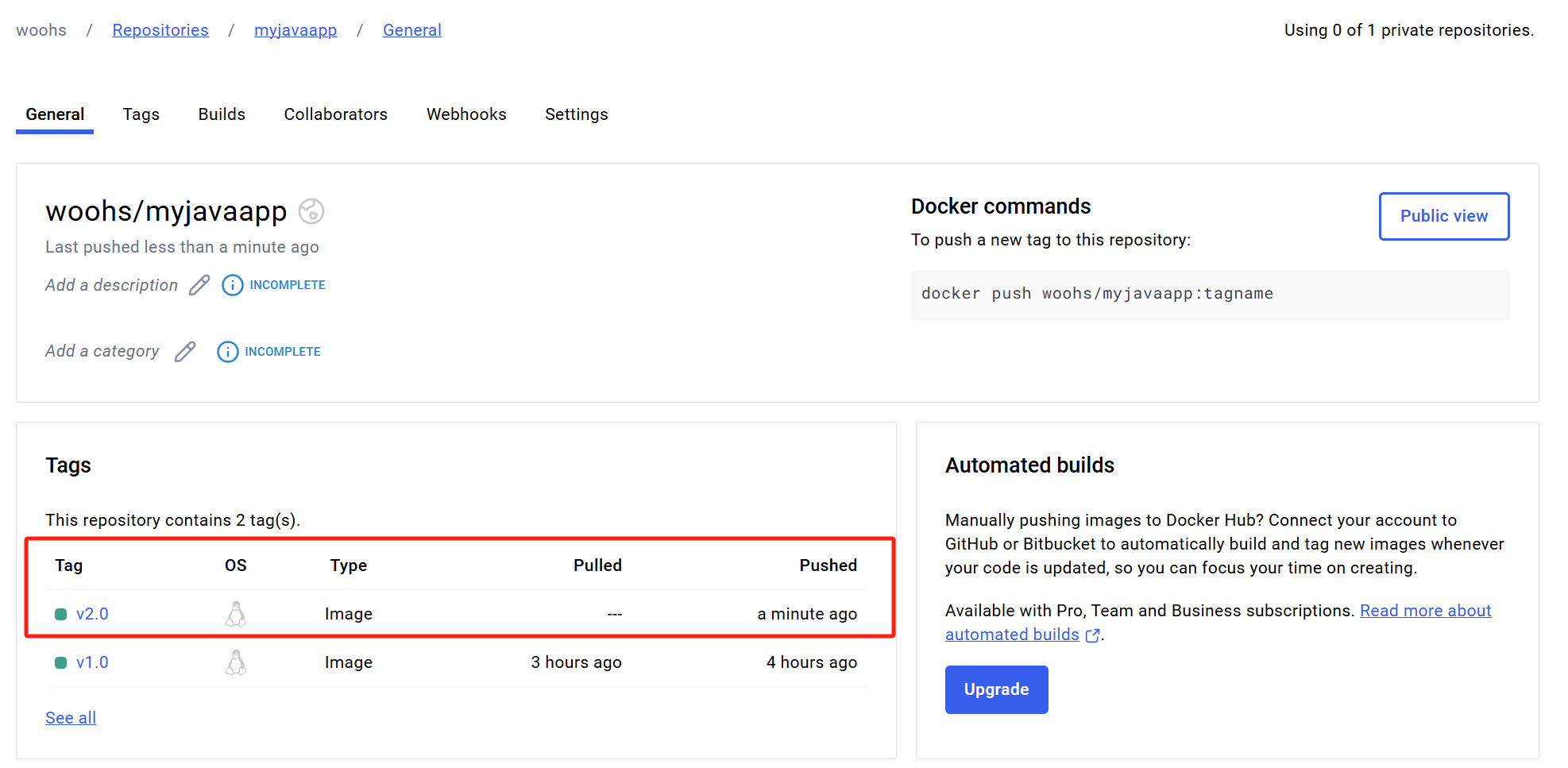
## 镜像推到远程

要将打好tag 的镜像推到远程仓库docker hub上

首先要docker login 登录自己的账号，然后执行命令：docker push [镜像名]。本文例子推镜像的命令：docker push woohs/myjavaapp:v2.0



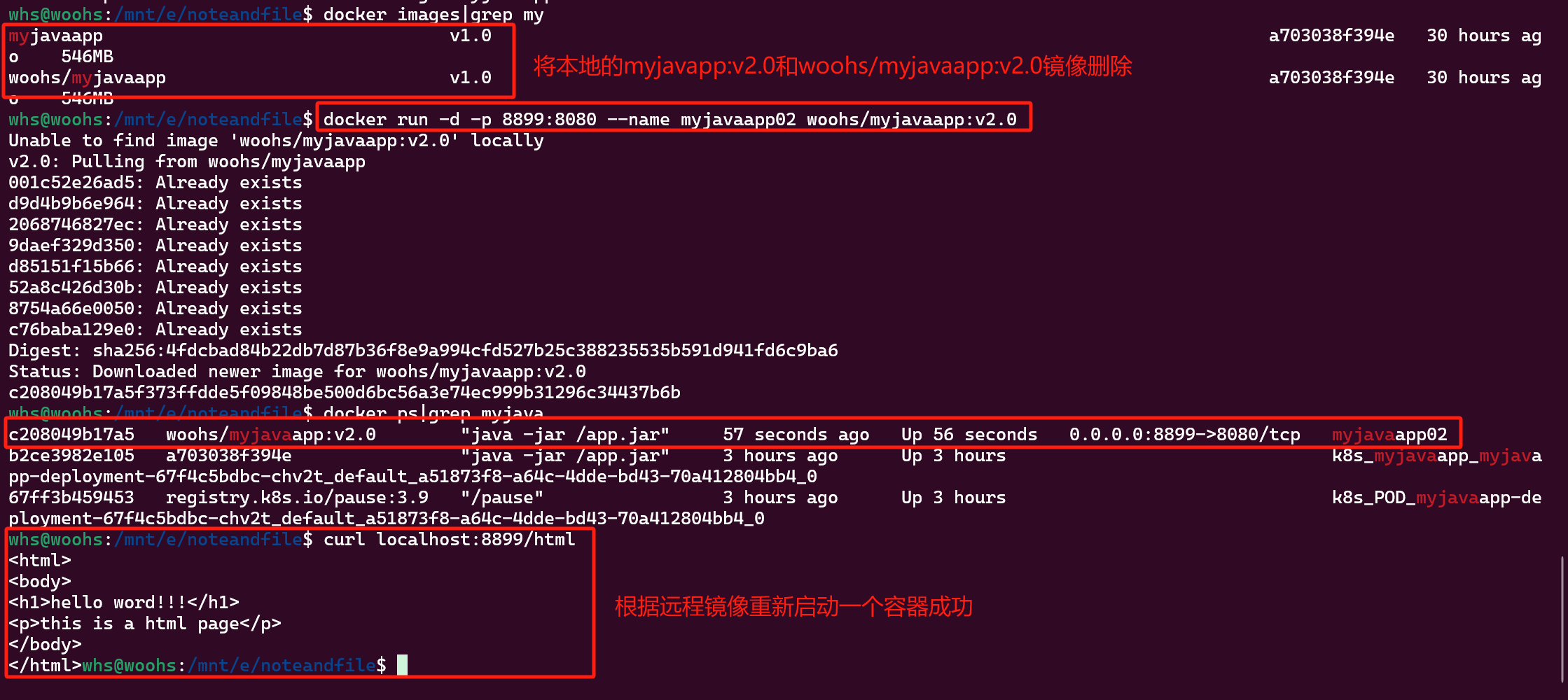
可以到docker hub上看到对应的镜像信息

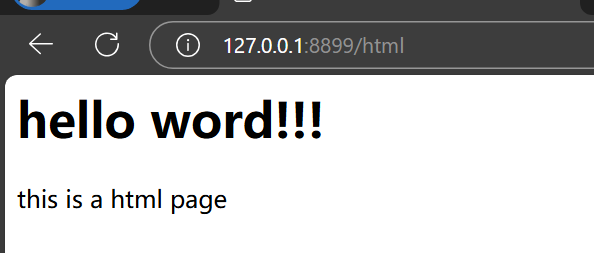


接下来可以再验证一下，根据远程仓库中的镜像直接再本地再启动一个容器试试。先将本地的myjavaapp:v2.0的镜像都删除。

然后执行命令：

docker run -d -p 8899:8080 --name myjavaapp02 woohs/myjavaapp:v2.0



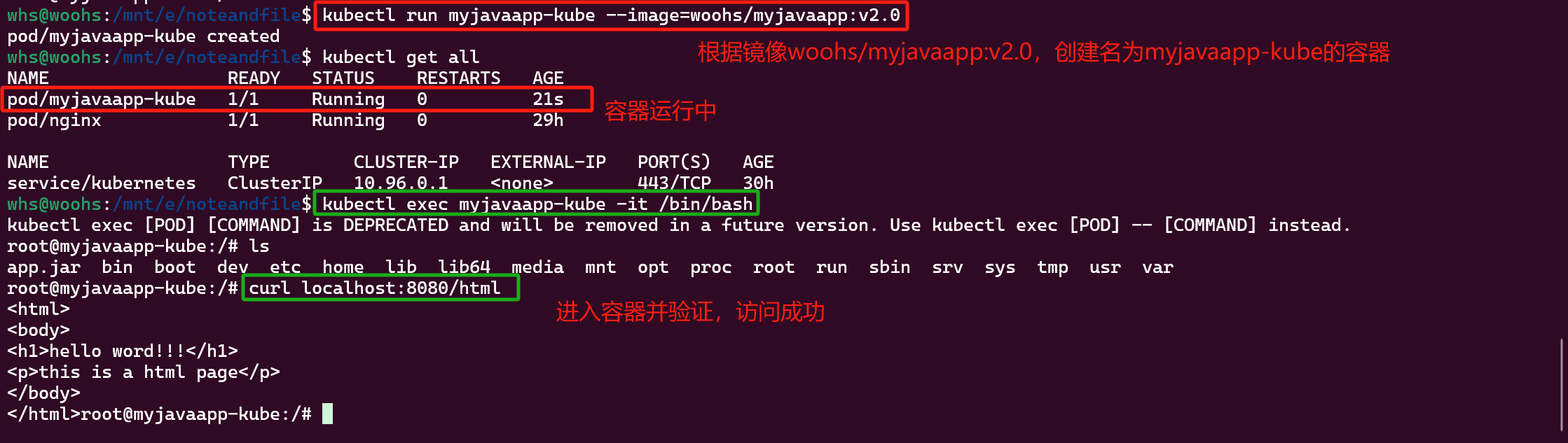


# 利用k8s根据镜像创建容器

## 2.1 根据镜像创建pod

创建容器命令：kubectl run 容器名称 --image=指定需要的镜像

执行命令：kubectl run myjavaapp-kube --image=woohs/myjavaapp:v2.0



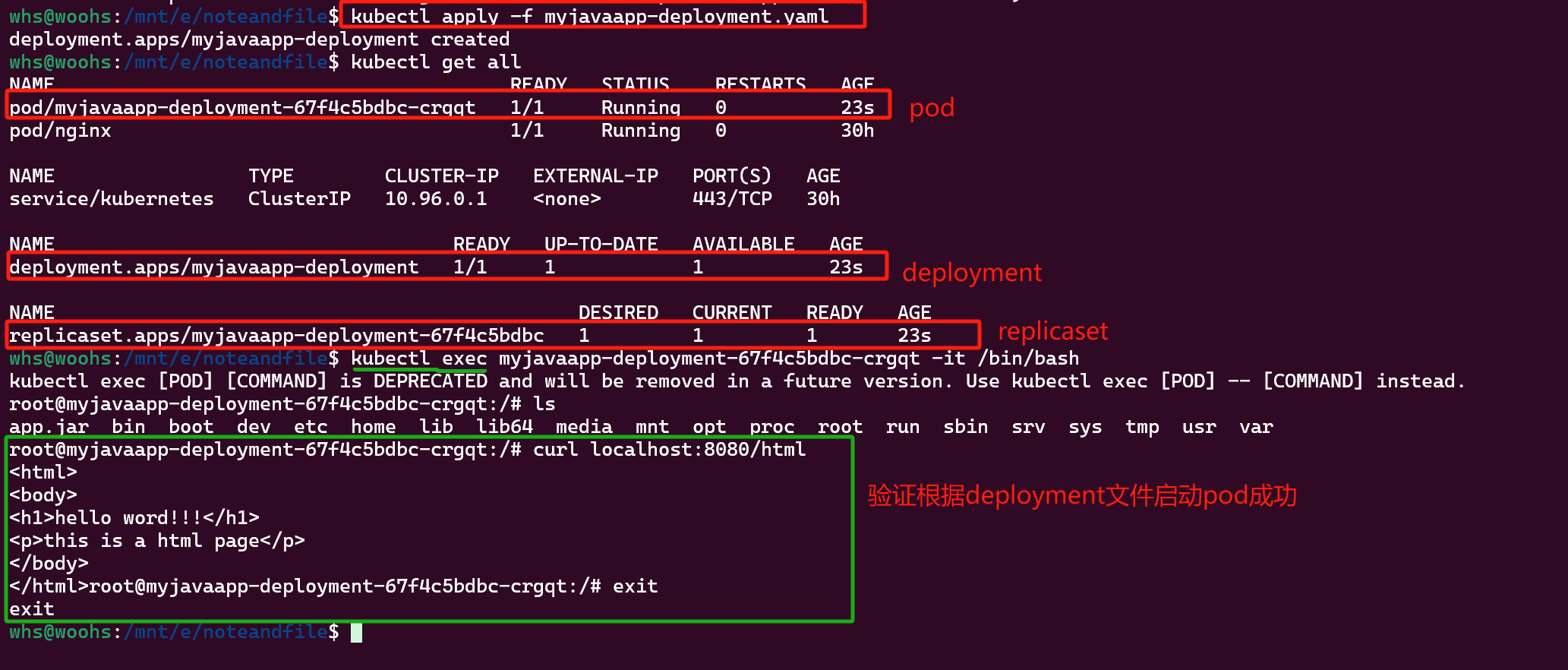
## 2.2 根据deployment创建

### 2.2.1、根据deployment文件创建

编写deployment.yaml文件（附录2-myjavaapp-deployment.yaml），执行kubectl apply命令。appply命令根据指定deployment文件创建/更新deployment。

apply命令：kubectl apply -f deployment文件名

先删除2.1创建的pod，然后执行命令：kubectl apply -f myjavaapp-deployment.yaml



可见，根据deployment创建容器成功，并且根据deployment文件会创建3中资源，分别是deployment、replicaset和pod，这三种资源的关系可以自行了解。

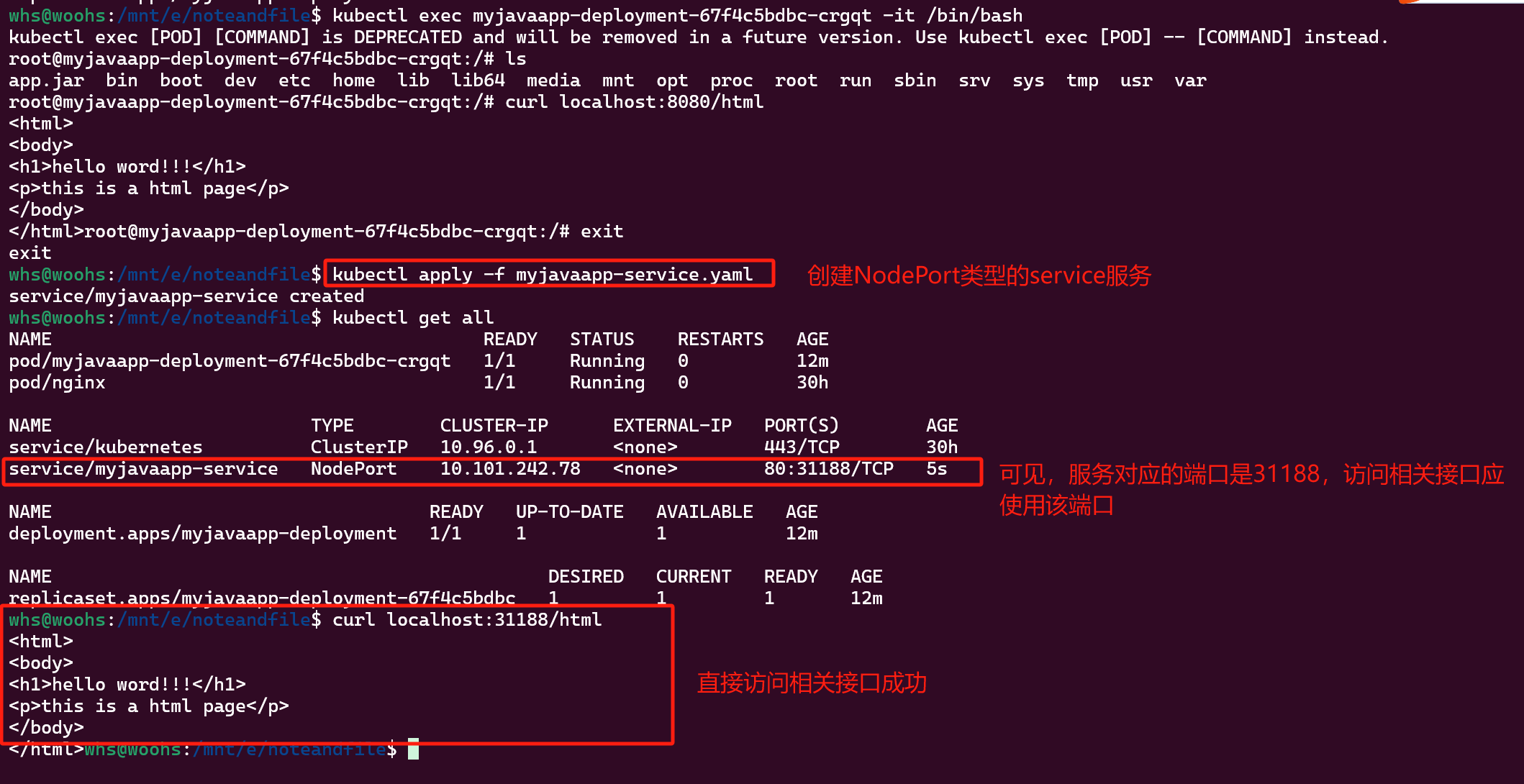
但是这样创建的服务，不能直接在主机访问。需要将这个服务暴露出去，这就需要创建service资源且类型是NodePort。接下来会演示如何将服务通过service暴露出去，让主机可以直接访问。

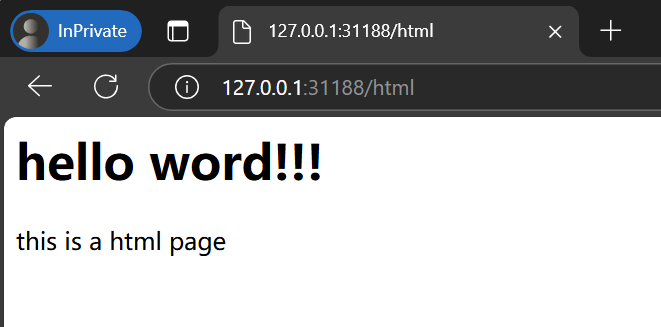
### 2.2.2 暴露服务service

通过闯将NodePort类型的service，可以将创建的服务让主机直接访问。

编写service.yaml文件（附录3-myjavaapp-service.yaml），执行kubectl apply命令。保留2.2.1产生的deployment、replicaset以及pod资源。

执行命令：kubectl apply -f myjavaapp-service.yaml





# 附录

## 1、Dockerfile源码

FROM openjdk:8-jdk

LABEL author=woohs

COPY app.jar /app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app.jar"]

## myjavaapp-deployment.yaml源码

apiVersion: apps/v1

kind: Deployment

metadata:

name: myjavaapp-deployment # 资源对象名称

spec:

selector:

matchLabels:

app: myjavaapp

replicas: 1

template:

metadata:

labels:

app: myjavaapp

spec:

containers:

- name: myjavaapp

image: myjavaapp:v1.0

ports:

- containerPort: 8080 # 对外暴露的端口是80

## myjavaapp-service.yaml源码

apiVersion: v1

kind: Service

metadata:

name: myjavaapp-service # 资源对象名称

spec:

# 没指定type时，默认是ClusterIP类型的服务，即集群内部的服务；

# NodePort表示节点端口类型的服务，该服务可对外提供服务，公网可以访问，需要配置 nodePort

type: NodePort

selector:

app: myjavaapp # 对应deployment下的 spec.selector.matchLabels.app值，通过标签选择对应的字段

ports:

- protocol: TCP

port: 80 # 该service对外公开的端口

targetPort: 8080 # pod的端口

nodePort: 31188 # 节点端口必须在30000~32767之间