

docker

by Yulei Bai

FILE	25246_YULEI_BAI_DOCKER_343871_726594106.DOCX (2.07M)		
TIME SUBMITTED	08-MAY-2021 03:10PM (UTC+0100)	WORD COUNT	4961
SUBMISSION ID	152281825	CHARACTER COUNT	25839

DOCKER

In the traditional application deployment mode, developers have their own development environment, such as operating systems and dependent services. After the development is complete, the developer packages the code to the operation and maintenance personnel for deployment in production. The operation and maintenance personnel need to build a production environment that is the same as the development environment and install the operating system. The process of configuring parameters is very cumbersome, and after the completion of the setup, the application deployment may fail due to subtle differences in the environment. The emergence of Docker has subverted this traditional model. The whole process has become packaging, traditional, and just running. The Docker image contains all the environment dependencies, ensuring that the development and production environments are consistent. In addition, containers can be run repeatedly anywhere, and the emergence of Docker has opened the prelude to the way of releasing products based on cloud computing platforms.

Cloud platform is a popular server architecture in recent years, and large data centers are usually equipped with thousands of servers. The cloud platform is a management system specifically oriented to a large number of servers. There are two types of KVM and container, and Docker belongs to the latter. The meaning of a container means that the developer packages the program and dependent files into an image. The user transfers the image to the server and runs it as a container on the Docker engine. Each container is actually an instance, and they have their own independence. The file system is isolated from each other, and there are strict restrictions on access between them.

Before the cloud platform, servers used a combination of operating system-basic software-application programs. For example, three software projects required three servers. Such deployment caused a serious waste of resources. KVM is a method to improve efficiency for this deployment method. The purpose of the virtual machine is to install multiple different operating systems on a physical machine, each operating system installs different versions of software, and these systems can run at the same time. Although the virtual machine method solves the problem of system fragmentation, the problem of slow startup has always plagued everyone, so container technology was born.

The essence of a container is to run only one operating system on a server, so that only the startup time of the software itself is required. Compared with a virtual machine, a container is just a file system (application + library file), and all containers use the same linux kernel. There are two types of Docker containers: one is a stateful container and the other is a stateless container. Stateful containers can keep data in a specific location, but data will be lost if the container is dropped. Stateless containers are easy to replace and suitable for orchestration services. Compared with kvm, containers have technical advantages in many aspects. Shown by the following figure:

Number of virtual machines started on each physical server
Docker's server utilization rate far exceeds KVM

KVM	Dozens
Docker	several hundred

Virtual machine startup time
Docker has a shorter startup time than KVM and is easier to maintain

KVM	minute
Docker	second

The performance of virtual machines compared to physical machines

KVM	Less than 90%
Docker	Close to 100%

Memory usage
Docker saves more memory than KVM

KVM	GB level
Docker	MB level

Hard disk usage
Docker saves hard disk more than KVM

KVM	GB level
Docker	MB level

Virtual machine migration time
Docker is faster than KVM

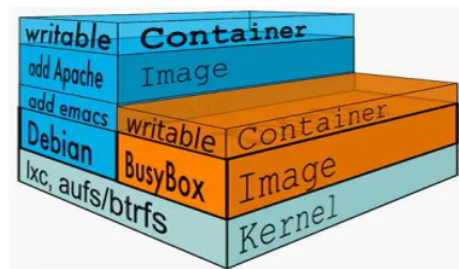
KVM	Minute level
Docker	Second level

It can be seen that in terms of memory, hard disk usage, virtual machine startup time or the number of deployments on the server, Docker is far ahead of KVM. Next, specific code examples will be used to demonstrate the benefits of Docker.

Portability

Docker image is the core of docker, and mirroring is the prerequisite for running containers. A Docker image can be regarded as a special file system, which can provide the required programs, libraries, resources, configuration files, and some configuration parameters (such as environment variables, users, etc.) for the operation of the container. The image does not contain dynamic data, and its content will not change. Inside the mirror is a layered file system, also called Union FS. The joint file system mounts several levels of directories together to form a virtual file system. Each layer of the file system becomes a layer, and each layer of files has three permissions under the system.

readonly, readwrite, and whiteout-able. Each layer of files in the Docker image is read-only. Starting from the operating system, each construction operation adds a layer of files on the original basis, and its structure is similar to the figure below.



If you want to create a docker image, you must create a corresponding Dockerfile. Dockerfile is a text file created by developers to control the image construction process. All operation commands for a container are recorded in this file. It is like A script, then use the docker build command to make an image based on this file and submit it to the local warehouse, so that the construction of the image will become transparent and maintenance will become simpler. The composition of the Dockerfile is similar to the following picture:

```
FROM some-existing-image-name
RUN some unix commands
EXPOSE some-portnumber
ENTRYPOINT [some unix command and flags]
```

There are some basic images in FROM, usually extracted from Docker Hub. RUN: Inside is the command to be executed by the generated container. EXPOSE: The open port of the server. Let's use an example to illustrate the dockerfile

FROM ubuntu:14.04

RUN apt-get update

RUN apt-get install -y nginx

COPY nginx.conf /etc/nginx/nginx.conf

COPY ./www-data /www-data

EXPOSE 80

CMD ["nginx"]

In the code shown above, ubuntu:14.04 is the benchmark operating system, RUN apt-get update is to update the software, RUN apt-get install is to install nginx, and COPY nginx.conf /etc/nginx/nginx.conf is to add from the host nginx.conf, The COPY ./www-data /www-data command is to add website files. The following is to create a mirror named my-nginx to demonstrate how to build dockerfile and my-nginx image.

First create a my-nginx folder. This is the name of the created image selection. Go to the folder and add a file named "Dockerfile" to it.

```
Docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

Start interactive shell

Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
C:\> mkdir my-nginx

Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
C:\> cd ./my-nginx

Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox/my-nginx
C:\> touch Dockerfile
```

Next, use the vim Dockerfile command to enter the file content editing interface, and enter the following command

```
MINGW64:/h/Program Files/Docker Toolbox/my-nginx
FROM ubuntu
RUN apt-get update && apt-get install -y nginx
EXPOSE 80
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

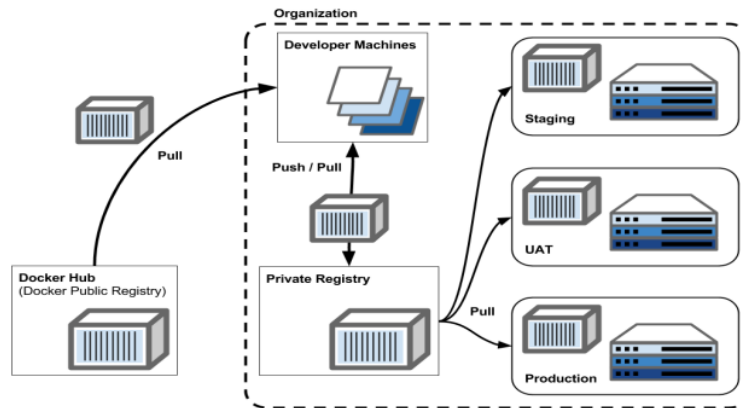
After exiting the editing directory, use the docker image build -t my-nginx command to build the my-nginx image

As shown in the figure above, the image construction process is carried out step by step according to the commands in the Dockerfile. It can be seen that we can easily

control the image construction just by writing the dockerfile file.

3 post images

We often send Docker images through the Docker registry, and the public registry is called Docker Hub. If we want to send the image to the registry, we use the command `docker image push`. If we want to download the image from the registry, we use the command `docker image pull`. The process is like the following diagram:



The developer pulls the image from the docker Hub public registry, and then pushes it to the private registry, and then other servers in the organization can download the image from the private registry, which can greatly improve the download speed, work efficiency and reduce Difficulty of debugging. Next, we will demonstrate the publishing function by publishing the local `my-nginx` to the Docker Hub public registry.

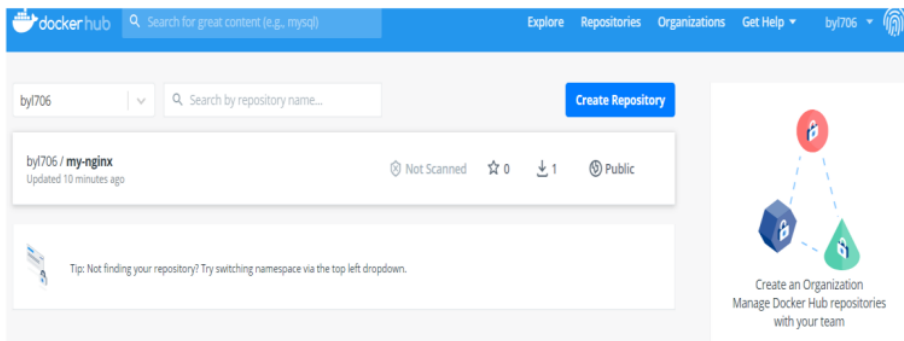
First, you need to register a dockerhub account, enter the command `docker login` in docker, and use the command `docker tag local-image:tag-name username/new-repo:tagname` to change the image name to the format specified by Docker Hub after the login is successful. This is not a direct change The original image is to create a tag reference of the original image (note that the tag is clearly written each time you upload, that is, which version), use the command `docker push by1706/my-nginx` to push the image to Docker Hub, and finally in dockerhub Check the `my-nginx` image just pushed over on the official website.

```
Administrator@PLDRI06UDXZ32 MINGW64 /h/Program Files/Docker Toolbox/my-nginx
$ docker login
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in C:\Users\Administrator\.docker\config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

Administrator@PLDRI06UDXZ32 MINGW64 /h/Program Files/Docker Toolbox/my-nginx
$ docker image tag my-nginx by1706/my-nginx

Administrator@PLDRI06UDXZ32 MINGW64 /h/Program Files/Docker Toolbox/my-nginx
$ docker push by1706/my-nginx
The push refers to repository [docker.io/by1706/my-nginx]
1c9bec35527b: Pushing [==>] 4.395MB/86.44MB
c20d459170d8: Mounted from library/ubuntu
db978cae6a05: Mounted from library/ubuntu
aeb3f02e9374: Preparing
```



4 container volume

The Docker image is made up of multiple file systems. When we start the container, Docker will load the read-only image layer and add a read-write layer on it. If the running container modifies the file, the file will be read-only. The layer is copied to the read-write layer, and the read-only layer will be hidden. When the Docker container is deleted and started via the mirror, the previous changes will be lost.

In order to save persistent data and share data between containers, Docker uses Volume to achieve this. Docker Volume can separate the container from the data generated by the container. Just like a mount point on a Linux server, a container can mount multiple different directories. Volume and the life of the container are independent. Even if the container is deleted, the volume will not be deleted, and the volume will not be recycled because it is not used by the container.

Volume does not use a hierarchical file system, which is very beneficial for data that is used frequently. In the actual construction process, the code directory can be mounted in the container to realize real-time acceptance of the returned data after code changes. The mount point of the container must be an absolute path. The address on the host machine can be either an absolute path or the name of a data volume.

Actual operation: You can create a volume named `my_volumeB` through the command `docker volume create my_volumeB`, as shown in the figure below

```
Q NAMES
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker volume create my_volumeB
my_volumeB
```

But at this time the volume has not been attached to the container. We use the command `docker run -it -v my_volumeB:/etc/app:ro --name container1 my-nginx` to attach the created volume to the container (ro in the command represents only Read, the front is the path where the volume is installed in the container), and then use the `inspect` command to view the details of the volume. The code and results are shown in the figure below.

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker run -it -v my_volumeB:/etc/app:ro --name container1 my-nginx
C

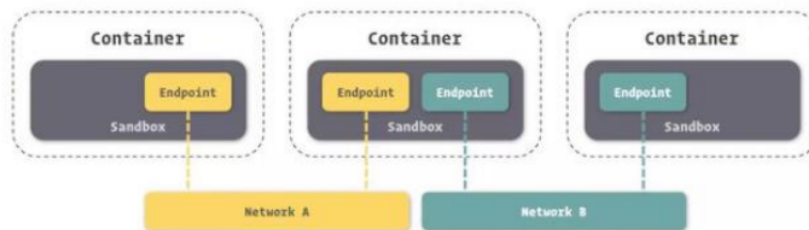
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker volume inspect my_volumeB
[
  {
    "CreatedAt": "2021-03-23T15:20:38Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/mnt/sdal/var/lib/docker/volumes/my_volumeB/_data",
    "Name": "my_volumeB",
    "Options": {},
    "Scope": "local"
  }
]
```

The data of the data volume is persistent, even if the container is deleted, the data of the data volume will not be affected.

Container networking

In the great age of the Internet. The network is an essential infrastructure. Docker is very complete in terms of network functions, and it is easy to develop different network solutions for containers through commands and parameters.

The container network is essentially a part of the virtual environment created by Docker for the application. It allows the application to be independent from the network environment of the host operating system to form a set of its own network equipment, IP protocol stack, and IP routing. Tables, firewalls and other network modules.



There are three core concepts in Docker network, they are Sandbox, Network, Endpoint.

Sandbox provides a virtual network stack of containers. Isolate the container network and the host network to form a completely independent container network environment. Network is a virtual subnet inside Docker, and members of the network can communicate with each other. Endpoint is the hole in the wall between the container and the network. It can control the access to the network environment. These three forms the core model of the Docker network.

Docker's network implementation

Docker officially provides five Docker network drivers, namely Bridge Driver, Host Driver, Overlay Driver, MacLan Driver, and None Driver. Bridge and Overlay are used frequently in development.

Bridge, the default network driver of Docker containers, realizes network communication through the bridge.

Overlay, a cross-Docker Daemon network built with the help of the Docker cluster module Docker Swarm. It can be used to realize network communication between different physical machines, so that containers in different physical machines can not feel the existence of multiple physical machines.

Container interconnection

Docker advocates the concept of lightweight containers in which containers and applications coexist, so there is only one application in a general container. But now there are more and more demands, and it is difficult to meet the demands of single containerization, and many systems are multi-node type. For example, web applications need to be composed of business applications, database applications, and cache applications. In order to achieve this function, docker uses multiple containers to implement this system, so the communication between containers becomes very important. Connecting one container to another container can be configured through link when the container is created through docker create or docker run. In the past development, every time you change the environment, you need to reconfigure the connection address and other parameters in the program, but in Docker, you don't need to care about this, you only need to configure the alias of the connected container in the program to map the IP. Docker can help us with it.

When Docker is installed, it will automatically create three networks, bridge (the created container is connected to this network by default), none, host. In Host mode, the container does not virtualize its own network card and other configurations, but uses the host's IP and port. In Bridge mode, each container will be assigned, set the IP, and the container will be connected to a docker0 virtual bridge, and communicate with the host through the docker0 bridge and the table nat configuration. In None mode, the network function of the container will be turned off. The initial three network modes as shown below

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
76d02d96cd19        bridge              bridge              local
e9e7d5b2c952        host                host                local
95413fe67018        none                null                local
```

We can customize the network bridge to control which containers can communicate with each other, and can also automatically resolve the container name to the IP address by DNS. Docker provides a network driver to help create a network. The default is to use the bridge type to create a network.

We will demonstrate the role of the Docker network through the mutual

communication between the two containers.

First, we build a bridge network called some-network through the command docker network create some-network

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker network create some-network
d0590380210de6d46f75e92206b24566b9156bc419213f7522bfeff582fc6491
```

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
76d02d96cd19        bridge              bridge              local
e9e7d5b2c952        host                host                local
95413fe67018        none                null                local
d0590380210d        some-network        bridge              local
```

Then run two debian containers, as shown in the following figure, --network some-network is to add the container to the some-network network, and then use the command docker container run --network some-network --name unix2 debian ping unix1 to ping unix2 to unix1 Transmit data and check whether the network is unblocked.

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker container run --network some-network -itd --name unix1 debian
34e0cf4efa40b1bbb6094420b06f1e08149ed07990c230fd8c48c28ec1f5a7b9
```

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker container run --network some-network --name unix2 debian ping unix1
PING unix1 (172.20.0.2) 56(84) bytes of data:
64 bytes from unix1.some-network (172.20.0.2): icmp_seq=1 ttl=64 time=0.081 ms
64 bytes from unix1.some-network (172.20.0.2): icmp_seq=2 ttl=64 time=0.089 ms
64 bytes from unix1.some-network (172.20.0.2): icmp_seq=3 ttl=64 time=0.062 ms
64 bytes from unix1.some-network (172.20.0.2): icmp_seq=4 ttl=64 time=0.068 ms
64 bytes from unix1.some-network (172.20.0.2): icmp_seq=5 ttl=64 time=0.076 ms
64 bytes from unix1.some-network (172.20.0.2): icmp_seq=6 ttl=64 time=0.062 ms
```

It can be seen that we can easily use the self-created network to help transfer data between containers. At the same time, you can view the IP address of the bridge and the information of the container that is using this network through the command docker network inspect some-network.

```
Administrator@LPLDRI06UDXZZ3Z MINGW64 /h/Program Files/Docker Toolbox
$ docker network inspect some-network
[
  {
    "Name": "some-network",
    "Id": "d0590380210de6d46f75e92206b24566b9156bc419213f7522bfeff582fc6491",
    "Created": "2021-03-24T11:31:57.86471252Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.20.0.0/16",
          "Gateway": "172.20.0.1"
        }
      ]
    }
  }
]
```

```

},
"ConfigOnly": false,
"Containers": {
  "34e0cf4efa40b1bbb6094420b06f1e08149ed07990c230fd8c48c28ec1f5a7b9": {
    "Name": "unix1",
    "EndpointID": "85d36f5f990fba10d45e40bea0c124b8231476324890b5558c367da711837cdc",
    "MacAddress": "02:42:ac:14:00:02",
    "IPv4Address": "172.20.0.2/16",
    "IPv6Address": ""
  },
  "5a85a9c2695424436b8f3e27f8c021bbdf031d4a4307ec21099f5e31226eaac8": {
    "Name": "unix2",
    "EndpointID": "6d3c4e74fcce8297eb47c27c209c948b66e2332851deb5440682e31e8c1aa0cb",
    "MacAddress": "02:42:ac:14:00:03",
    "IPv4Address": "172.20.0.3/16",
    "IPv6Address": ""
  }
}
}

```

Create my own application

We deploy the Vue project through Docker. First, create the Vue project through the Vue CLI. This step is relatively simple and can be created directly through the Vue create command. Vue-cli is an officially released vue.js project scaffold, a scaffold designed to quickly build complex single-page applications. It can easily create new applications and can be used to automatically generate vue and webpack project templates. It is a project template used to automatically generate vue.js+webpack, and it provides batteries-included build settings for modern front-end workflows. It greatly reduces the difficulty of using webpack and supports hot updates, so we only need to focus on development.

1 The first step to build vueApp is to create a project called test-docker, then enter the root directory of the test-docker project and use the npm run build command to build a vue project

```

Administrator@LPLDR106UDXZZ3Z MINGW64 /h/Program Fi
$ vue create test-docker

Vue CLI v4.5.12

New version available 4.5.12 -> 4.5.13
Run npm i -g @vue/cli to update!

? Please pick a preset: Default ([Vue 2] babel, esl
Vue CLI v4.5.12
  Creating project in H:\Program Files\Docker Too
  Initializing git repository.
  Installing CLI plugins. This might take a whi
> yorkie@2.0.0 install H:\Program Files\Docker Tool
> node bin/install.js

```

```

> core-js@3.12.0 postinstall H:\Program Files\Docker Toolbox\my-node-app\test-docker\node_modules\core-js
> node -e "try(require('./postinstall'))catch(e){}"

> ejs@2.7.4 postinstall H:\Program Files\Docker Toolbox\my-node-app\test-docker\node_modules\ejs
> node ./postinstall.js

added 1248 packages from 660 contributors in 63.704s

70 packages are looking for funding
  run `npm fund` for details

❖ Invoking generators...
❖ Installing additional dependencies...

added 53 packages from 36 contributors in 13.153s

75 packages are looking for funding
  run `npm fund` for details

❑ Running completion hooks...

❖ Generating README.md...

❖ Successfully created project test-docker.
❖ Get started with the following commands:

```

```

You will need to perform the initial commit yourself.

Administrator@LPLDR106UDXZ232 MINGW64 /h/Program Files/Docker Toolbox/my-node-app
$ cd test-docker

Administrator@LPLDR106UDXZ232 MINGW64 /h/Program Files/Docker Toolbox/my-node-app/test-docker (master)
$ npm run build

> test-docker@0.1.0 build H:\Program Files\Docker Toolbox\my-node-app\test-docker
> vue-cli-service build

- Building for production...

DONE Compiled successfully in 8443ms

File                               Size                               Gzipped
dist\js\chunk-vendors.96ba5d7f.js  89.90 KiB                         32.25 KiB
dist\js\app.640778e9.js           4.58 KiB                          1.63 KiB
dist\css\app.fb0c6elc.css          0.33 KiB                          0.23 KiB

Images and other types of assets omitted.

DONE Build complete. The dist directory is ready to be deployed.
INFO Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html

```

After executing `npm run build`, a `dist` directory will be created in the root directory of the project, which stores the packaged result of the vue project for production release. Then as long as the content in the `dist` directory is deployed to the static resource site, the project can be directly accessed. We create a `nginx` container in `docker` through the `nginx` server to run the server, and put the contents in the `dist` directory under the static resource site of the `nginx` server.

2 Create an `nginx` mirror containing the contents of the `dist` directory

We build a new image, which is based on `nginx`, and copy the contents of the `dist` directory under it. After the image is built, use this image to create the container.

2.1 Obtain the official `nginx` mirror

Download the latest `nginx` image from the official via `docker pull nginx`.

```

Administrator@LPLDR106UDXZ32 MINGW64 /h/Program Files/Docker Toolbox/my-node-app/test-docker (master)
$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
f7ec5a41d630: Pull complete
aa1efaf4b3bf: Pull complete
b78b95af9b17: Pull complete
c7d6bca2b8dc: Pull complete
cf16cd8e71e0: Pull complete
0241c68333ef: Pull complete
Digest: sha256:75a55d33ecc73c2a242450a9f1cc858499d468f077ea942867e662c247b5e412
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

```

2.2 Write dockerfile file

Create a new file named dockerfile in the root directory of the vue project with the following content

MINGW64:/h/Program Files/Docker Toolbox/my-node-app/test-docker

```

FROM nginx
COPY dist/ /usr/share/nginx/html/

```

Command: COPY dist/ /usr/share/nginx/html/ is to copy the contents of the dist directory in the directory where the current dockerfile is located to /usr/share/nginx/html/ of the container. Immediately use the command: docker build -t vuenginx to build a mirror named vuenginx.

```

Administrator@LPLDR106UDXZ32 MINGW64 /h/Program Files/Docker Toolbox/my-node-app/test-docker (master)
$ docker build -t vuenginx .
Sending build context to Docker daemon 127.1MB
Step 1/2 : FROM nginx
--> 62d49f9bab67
Step 2/2 : COPY dist/ /usr/share/nginx/html/
--> 3b905888e1ea
Successfully built 3b905888e1ea
Successfully tagged vuenginx:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.

```

Open another terminal and build by command docker images | grep "vuenginx"

```

Administrator@LPLDR106UDXZ32 MINGW64 /h/Program Files/Docker Toolbox
$ docker images | grep "vuenginx"
vuenginx          latest          3b905888e1ea    About a minute ago  134MB

```

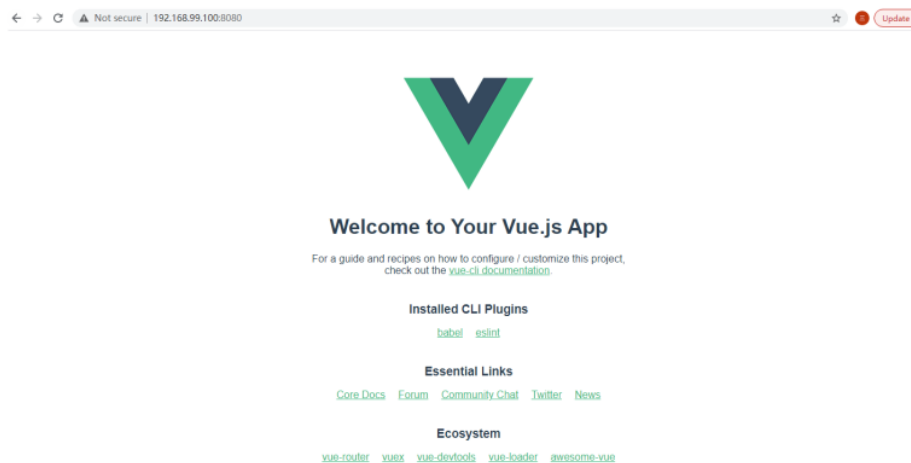
2.3 Create and run the vueApp container

After the image is created, we can create and run a container based on the image.

```
Administrator@PLDR106UDXZ32 MINGW64 /h/Program Files/Docker Toolbox
$ docker run -itd --name vueApp -p 8080:80 vuenginx
41e2fe0115a099b4d8133a8f08a3e4c0b3alcc5caF017154af02d5446f66d08a

Administrator@PLDR106UDXZ32 MINGW64 /h/Program Files/Docker Toolbox
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
41e2fe0115a0        vuenginx           "/docker-entrypoint..." 6 seconds ago       Up 5 seconds        0.0.0.0:8080->80/tcp
48136ca5e964        hello-world        "/docker-entrypoint.s..." 34 minutes ago      Up 34 minutes        0.0.0.0:8081->8081/tcp
04b177c617f        fb881d4fc473       "/bin/sh -c 'echo 'h...' 4 hours ago         Exited (2) 4 hours ago
nostalgic_golick    fb881d4fc473       "/bin/sh -c 'echo 'h...' 4 hours ago         Exited (2) 4 hours ago
88517a7e6b67        awesome_swirles    fb881d4fc473       "/bin/sh -c 'echo 'h...' 4 hours ago         Exited (2) 4 hours ago
bc26a91cf6ca        busy_gagarin       fb881d4fc473       "/bin/sh -c 'echo 'h...' 4 hours ago         Exited (2) 4 hours ago
64394c78aed3        3fd58903abb8       "/bin/sh -c 'pat-get..." 4 hours ago         Exited (127) 4 hours ago
condescending_lumiere 3fd58903abb8       "/bin/sh -c 'pat-get..." 4 hours ago         Exited (127) 4 hours ago
2aca54eca87e        zen_wilson         my-nginx           "nginx -g 'daemon of..." 2 weeks ago         Exited (255) 4 hours ago
49537a8e3eec        my-nginx           "nginx -g 'daemon of..." 2 weeks ago         Exited (255) 4 hours ago
```

We can run the container based on vuenginx through the command `docker run -itd --name vueApp -p 8080:80 vuenginx`, and view the container information through `docker ps -a`. We run the container and bind the port 8080 of the local host to the port 80 of the container. When we access port 8080 on the local host, the request will be forwarded to port 80 of the container, which will be processed by the nginx server in the container. At this time, visit `192.168.99.100:8080` on the local host browser, and you can see our vueApp homepage, indicating that we have successfully deployed a static resource server in the docker container.



What is Kubernetes?

Kubernetes is an open source Linux container automated operation and maintenance platform. If we combine multiple hosts into clusters to run Linux containers, Kubernetes can help users manage those clusters simply and efficiently. It can help us manage containerized workloads and services, and achieve declarative configuration

and automation. The Kubernetes ecosystem is huge and growing rapidly. Kubernetes was open sourced by Google in 2014. Kubernetes has Google's rich experience in running production workloads on a large scale.

The emergence of Kubernetes is because in the traditional deployment era, various organizations cannot define resource boundaries for physical server applications, which leads to resource allocation problems, and often an application occupies most of the resources. In order to solve this problem, Various organizations have introduced virtualization. Through virtualization, we can run many virtual machines on a single server. Under virtualization, applications are isolated between virtual machines, which can ensure application security. Later in the container era, restrictions were relatively relaxed. Applications can share an operating system. Containers are similar to VMs and have their own file system, CPU, memory, process space, etc.

What can Kubernetes do?

In actual production, we need to ensure that the container running the application will not be shut down, but it often fails, and another container needs to be started at this time. Kubernetes can provide a system framework to solve these problems and batch these behaviors in the form of a system.

Kubernetes can provide:

1 Service discovery and load balancing

Kubernets can use the DNS name or its own IP address to expose the container. If the traffic is large, it can evenly distribute the network traffic.

2 Storage arrangement:

Kubernets allows you to automatically mount the storage system of your choice.

3 Automatic deployment and rollback

I can use Kubernetes to create new containers for my deployment, delete existing containers and use all their resources for new containers.

4 Self-complete packing calculation

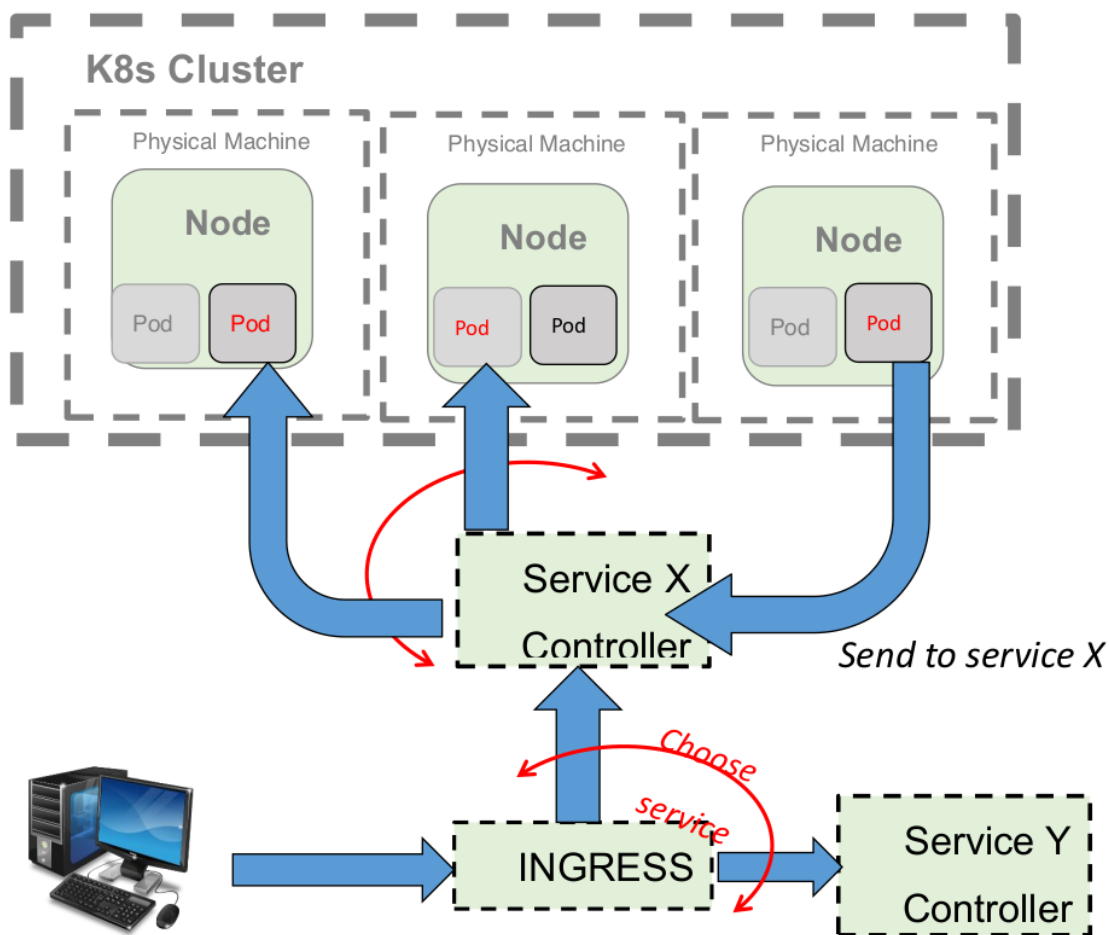
Kubernets can help us specify the CPU and memory required for each container.

5 self-repair

Kubernetes restarts the failed container, replaces the container, kills the container that does not respond to user-defined health checks, and does not notify the client until it is ready for service.

6 Key and configuration management

In Kubernetes, we can store and manage private information, such as passwords, ssh keys, etc. We can update the key and application configuration without rebuilding the container image



The structure of Kubernetes is shown in the figure above. K8s.Cluster is a cluster responsible for coordinating the physical computers connected together. The cluster contains two resources, one is Master and the other is Node. Master represents the computer that controls the Kubernetes node. Node is responsible for the computer that executes requests and tasks, and Master is responsible for node control. Pod is a container group that is deployed on a node and contains one or more containers. Containers in the same Pod share an IP address, IPC, host name and other resources. The Service Controller in the figure can provide the same network service for a group of Pods and manage the communication between nodes in the cluster. Ingress accesses multiple HTTP requests by providing the same IP address to the outside, so as to handle the external traffic entering the K8s cluster.

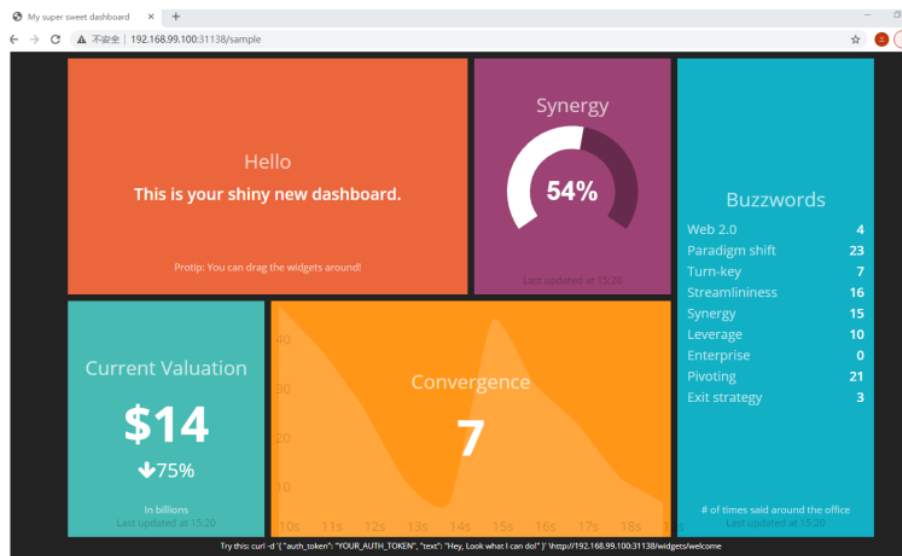
In the Kubernetes system, Kubernetes entities exist in the form of Kubernetes Objects. These entities can represent the state of the cluster. Objects are objects that the cluster uses to save the required state list. These objects can be created and deleted through the Kubernetes API. Let's demonstrate its deployment by building a cluster.

```
PS C:\WINDOWS\system32> kubectl run smashing --image=visibilityspots/smashing --port=3030 --restart=Never
pod/smashing created
PS C:\WINDOWS\system32> kubectl get pods --watch
NAME                                READY   STATUS    RESTARTS   AGE
k8s-pod-deploy-6f7459f794-79vrk    1/1     Running   1           16h
k8s-pod-deploy-6f7459f794-8b8kl    1/1     Running   1           16h
k8s-pod-deploy-6f7459f794-r9rhk    1/1     Running   1           16h
myapp-deployment-676d695f94-6hmng  0/1     InvalidImageName 0           16h
myapp-deployment-676d695f94-f66jc  0/1     InvalidImageName 0           16h
myapp-deployment-676d695f94-k45tb  0/1     InvalidImageName 0           16h
myapp-deployment-676d695f94-ljf7r  0/1     InvalidImageName 0           16h
myapp-deployment-676d695f94-s7c4s  0/1     InvalidImageName 0           16h
nginx-pod                           0/1     Error     0           42h
smashing                            1/1     Running   0           7s
PS C:\WINDOWS\system32> kubectl expose pods smashing --type=NodePort
service/smashing exposed
PS C:\WINDOWS\system32> minikube service smashing
```

NAMESPACE	NAME	TARGET PORT	URL
default	smashing	3030	http://192.168.99.100:31138

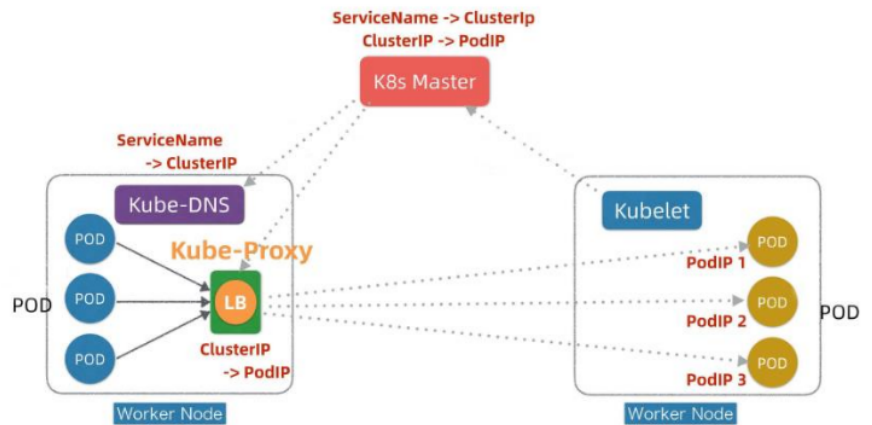
```
* 正通过默认浏览器打开服务 default/smashing...
PS C:\WINDOWS\system32>
```

Start a smashing instance through the kubectl run command and open port 3030 of the container, then use the kubectl get pods --watch command to monitor the creation and deletion of POD events, and then use the command Kubectl expose pods smashing --type=NodePort to create a service To create a service in the factory, we specify the service type as NodePort. NodePort is based on the inside of the cluster. This service is opened on each node of the cluster. This service can be accessed through any <NodeIP>:NodePort address, and finally through minikube Service smashing to get the access url, get the following WEB page.

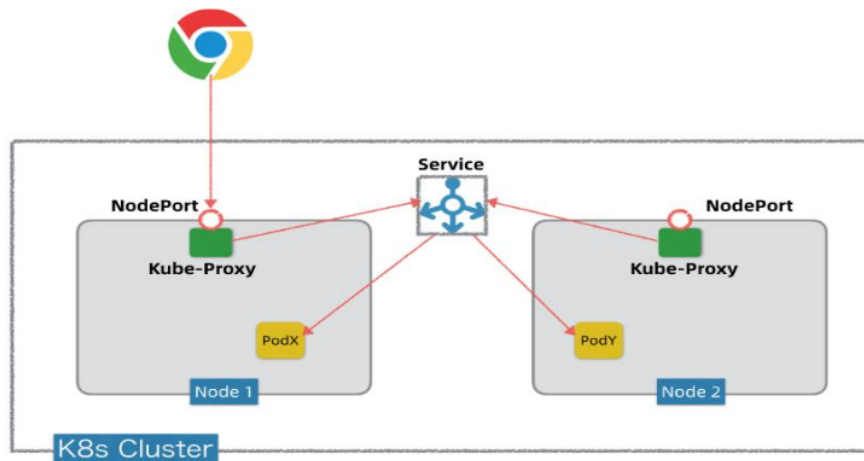


Use Kubernetes controllers and services to manage containers to generate network applications

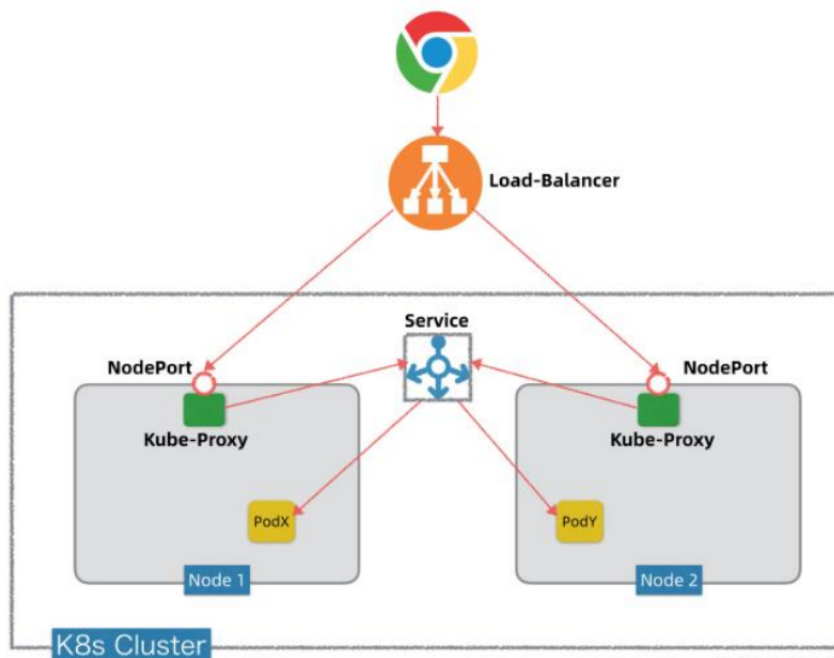
In order to build a network application, we first need to create a worker node. In order to become a node in a Kubernetes cluster, the physical machine or VM must run the Kubelet process. Each worker node in the cluster has one or more IP addresses for it. These The IP address is the physical IP through which the worker node communicates. The worker node has been assigned a private IP initially, but if the cluster is exposed to the external environment, it may be assigned both a private IP and a public IP address. There are two modules in Kubernetes: controller and service. The controller is responsible for creating and managing Pods, such as ensuring that the correct number of Pods are running. The Service manages the communication between Pods. The service requires a controller to ensure that the Pod of the service is up and running. At the same time, both the service and the controller use tags and tag selectors to select which Pods to operate on. When building an application, you need to use the same tag to select the same Pod. Intervene in the network outside the cluster:



The left part of the figure shows the service discovery part, in which Kube-Proxy is responsible for the associated communication between the Service network and the Pod network. It can expose the internal services of K8s. Kube-Proxy is deployed on all Worker nodes in the K8s cluster. One, it masters all the information of the Services network. If you want to connect Kube-Proxy to the node network, you only need to expose a listening port, the term is NodePort, the following figure is a simplified model of exposing services through NodePort.

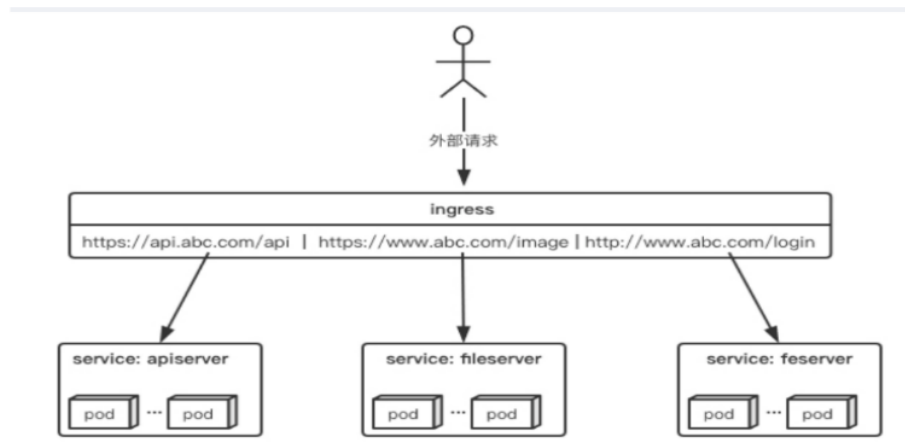


The left part of the figure shows the service discovery part, in which Kube-Proxy is responsible for the associated communication between the Service network and the Pod network. It can expose the internal services of K8s. Kube-Proxy is deployed on all Worker nodes in the K8s cluster. One, it masters all the information of the Services network. If you want to connect Kube-Proxy to the node network, you only need to expose a listening port, the term is NodePort, the following figure is a simplified model of exposing services through NodePort.

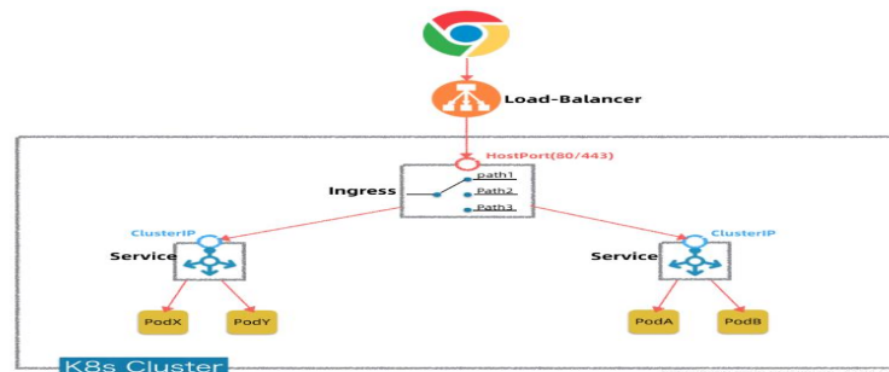


For the service to be exposed, we set the type of service publication to LoadBalancer. After the service is released, K8s will automatically create the service's NodePort port

forwarding, and will apply for an SLB with an independent public IP, and K8s will automatically map the SLB to the NodePort corresponding to the background K8s cluster, so that the public IP of the SLB can be used To access K8s internal services, Load-Balancer will do load balancing behind the scenes. The three methods (ClusterIP, NodePort and LoadBalance) of K8s exposing services are not very suitable in actual production, so there is another way, that is, ingress. The simple understanding of Ingress is the Service of Service. It uses independent ingress objects to formulate request forwarding rules. Instead of considering each Service separately, it can be considered from a business perspective. For example, there are APIs, file storage, and front-end Service in the cluster. Object to realize request forwarding, as shown in the figure below.



Ingress actually appeared because the cost of LB's exposure service method was not cost-effective. It was found that exposing a service requires the purchase of an independent LB+IP, which is obviously not cost-effective. In order to expose more services with only one LB+IP, we deploy an independent reverse proxy service inside K8s, which acts as a proxy for forwarding, as shown in the figure below.



The following example demonstrates the use of Kubernetes controllers and services to manage containers to generate network applications. First build the deployment controller.

```
PS C:\WINDOWS\system32> kubectl create deployment k8s-pod-deploy --image=mikeduwtsd/k8s-pod
deployment.apps/k8s-pod-deploy created
PS C:\WINDOWS\system32> kubectl scale --replicas=3 deploy/k8s-pod-deploy
deployment.apps/k8s-pod-deploy scaled
PS C:\WINDOWS\system32> kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
k8s-pod-deploy 2/3     3            2           17s
myapp-deployment 0/5     5            0           28h
PS C:\WINDOWS\system32> kubectl get pods --watch
NAME          READY   STATUS    RESTARTS   AGE
k8s-pod-deploy-6f7459f794-gkwx5 1/1     Running   0          22s
k8s-pod-deploy-6f7459f794-xflz4 1/1     Running   0          30s
k8s-pod-deploy-6f7459f794-xgczx 1/1     Running   0          22s
myapp-deployment-676d695f94-6hmnq 0/1     InvalidImageName 0          28h
myapp-deployment-676d695f94-f66jc 0/1     InvalidImageName 0          28h
myapp-deployment-676d695f94-k45tb 0/1     InvalidImageName 0          28h
myapp-deployment-676d695f94-ljf7r 0/1     InvalidImageName 0          28h
myapp-deployment-676d695f94-s7c4s 0/1     InvalidImageName 0          28h
nginx-pod      0/1     Error     0          2d6h
smashing       1/1     Running   0          12h
PS C:\WINDOWS\system32> kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
k8s-pod-deploy 3/3     3            3           79s
myapp-deployment 0/5     5            0           28h
PS C:\WINDOWS\system32>
```

Through the command `kubectl create deployment k8s-pod-deploy --image=mikeduwtsd/k8s-pod` to build a deployment controller named K8s-pod-deploy, when we set a set of expected number of Pods, the Controller will maintain the number of Pods consistent with expectations. And the controller will update the Pod according to the given strategy. The specific principle is that the Deployment Controller obtains all the Pods with the required tags from Etcd, and then counts the number of them. This is the actual state, and the value of the replicas of the Deployment object is the expectation Status, and then the controller compares the two statuses, and based on the comparison result, determines whether to create a Pod or delete an existing Pod. After completing the controller configuration, set the number of pod replicas named K8s-pod-deploy to 3 through the `kubectl scale --replicas=3 deploy/K8s-pod-deploy` command, and then list all deploys through `Kubectl get deploy`. To view For the status information of all pods, we use the command `kubectl get pods --watch` to watch stream of pod status, and we can see that all three containers are in a running state. Next, create the ClusterIP service by deploying the controller.

```
PS C:\WINDOWS\system32> kubectl expose deployment k8s-pod-deploy --name=k8s-pod-svc --port=778 --target-port=1339 --type=ClusterIP
service/k8s-pod-svc exposed
PS C:\WINDOWS\system32> kubectl get svc
```

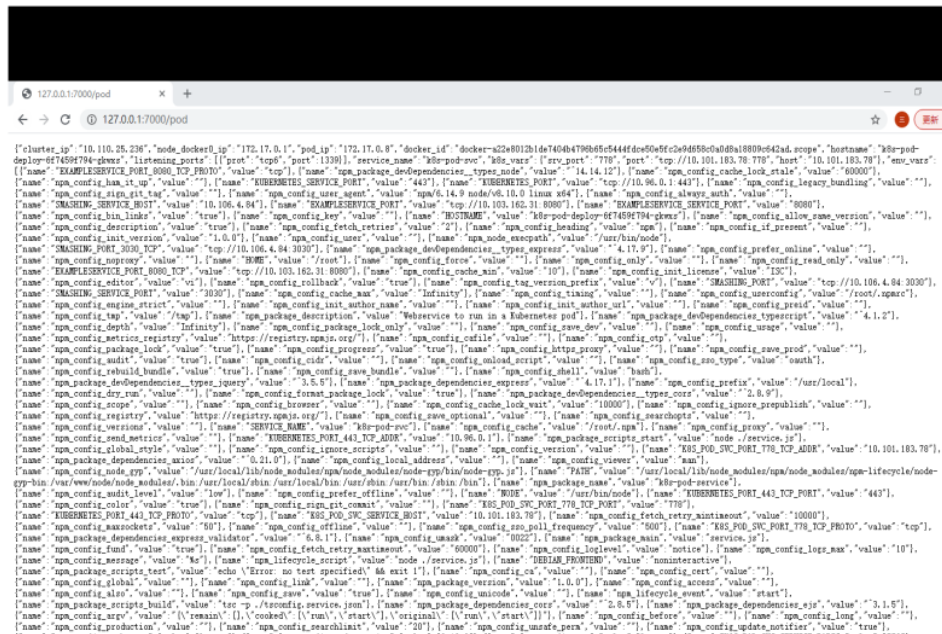
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
exampleservice	LoadBalancer	10.103.162.31	<pending>	8080:30001/TCP	34h
k8s-pod-svc	ClusterIP	10.110.25.236	<none>	778/TCP	9s
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	2d10h
smashing	NodePort	10.106.4.84	<none>	3030:31138/TCP	12h

Use the command `Kubectl expose deployment K8s-pod-deploy --name=K8s-pod-svc --port=778--target=Cluster`

Create a ClusterIP service named "k8s-pod-svc" for the deployment controller named "k8s-pod-deploy", the service port is set to 778, and the target port is 1339. And the ClusterIP address is 10.110.25.236 obtained through the get svc command

```
PS C:\WINDOWS\system32> kubectl port-forward pod/k8s-pod-deploy-6f7459f794-gkwx5 7000:1339
Forwarding from 127.0.0.1:7000 -> 1339
Forwarding from [::]:7000 -> 1339
Handling connection for 7000
Handling connection for 7000
```

Finally, by adding port forwarding to allow the host to use the target port 1339 on the Pod and port 7000 on the computer to forward to the Pod in the cluster, use the URL on the web side: <http://127.0.0.1:7000/pod>
Visit to view the results returned by the Pod, as shown in the figure below.



Similar to docker, in K8s we can also use yaml files to build the controllers and services we need to configure. Take the name running the "tutum-hello" image to create a YAML file for the ClusterIP service as an example. First, you need to build a joint yaml configuration file for the service and control.

As shown in the figure above, first build the yaml file we need, and select the API version of the service and control. Compared with the previous case, yaml configures

some lengthy and complicated configuration commands in data serialization format in advance, such as `kubectl scale --replicas=3 deploy/K8s-pod-deploy` This instruction is filled in the configuration after the Spec field in the yaml file, corresponding to the yaml

```
Kind: Service
apiVersion: v1
metadata:
  name: clusterip-service
Spec:
  type: ClusterIP
  Selector:
    application: clusterip-app
  ports:
    - protocol: "TCP"
      port: 8888
      targetPort: 80
---
Kind: Deployment
apiVersion: apps/v1
metadata:
  name: clusterip-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      application: clusterip-app
  template:
    metadata:
      labels:
        application: clusterip-app
    spec:
      containers:
        - name: clusterip-container
          image: tutum/hello-world
          ports:
            - containerPort: 80
```

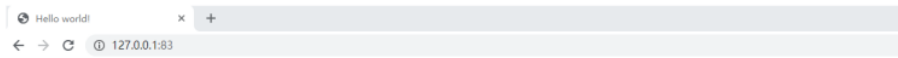
```
spec:
  replicas: 3
  selector:
    matchLabels:
```

This can greatly improve the speed and efficiency of service creation and control. All this thanks to a series of advantages of yaml files

- • YAML files are easy for humans to read, expressive and extensible.
- • YAML files are easy to implement and use.
- • Can be easily transplanted between programming languages.
- • Matches with the native data structure of agile language.
- • Yaml is easy to use, so there is no need to add all the parameters to the command line.
- • Easy to maintain-YAML files can be added to source control to track changes.
- • Flexible and convenient-YAML can be used to create more complex structures (compared to the structures that can be created using the command line)

```
PS C:\WINDOWS\SysWOW64> kubectl apply -f clusterip-service.yaml
service/clusterip-service created
deployment.apps/clusterip-deployment created
PS C:\WINDOWS\SysWOW64> kubectl get deploy
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
clusterip-deployment 3/3      3            3           86s
myapp-deployment     0/5      5            0           46h
PS C:\WINDOWS\SysWOW64> kubectl get pods --watch
NAME                READY   STATUS              RESTARTS   AGE
clusterip-deployment-655d8f7c-2fnfx 1/1     Running            0          2m23s
clusterip-deployment-655d8f7c-jzm6n 1/1     Running            0          2m23s
clusterip-deployment-655d8f7c-q9d9t 1/1     Running            0          2m23s
myapp-deployment-676d695f94-6hmgg 0/1     InvalidImageName   0          46h
myapp-deployment-676d695f94-f66jc 0/1     InvalidImageName   0          46h
myapp-deployment-676d695f94-k45tb 0/1     InvalidImageName   0          46h
myapp-deployment-676d695f94-ljf7r 0/1     InvalidImageName   0          46h
myapp-deployment-676d695f94-s7c4s 0/1     InvalidImageName   0          46h
nginx-pod           0/1     Error              0          3d
smashing            0/1     Error              0          30h
PS C:\WINDOWS\SysWOW64> kubectl get svc
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
clusterip-service   ClusterIP    10.111.216.246   <none>         8888/TCP         2m57s
exampleservice      LoadBalancer 10.103.162.31    <pending>      8080:30001/TCP   2d4h
kubernetes           ClusterIP    10.96.0.1        <none>         443/TCP          3d4h
smashing            NodePort     10.106.4.84      <none>         3030:31138/TCP   30h
PS C:\WINDOWS\SysWOW64> kubectl port-forward svc/clusterip-service 83:8888
Forwarding from 127.0.0.1:83 -> 80
Forwarding from [::1]:83 -> 80
Handling connection for 83
Handling connection for 83
```

As shown in the figure above, first use the command `kubectl apply -f clusterip-service.yaml`. The function of this command is to upgrade the existing resource object according to the content listed in the `clusterip-service.yaml` file, and then create the name The service of `clusterip-service` and the deployment named `clusterip-deployment`. As in the previous example, we use the commands `kubectl get deploy` and `kubectl get pods --watch` to view all deploy and pods, and finally forward the local port 83 to the operation through the command `kubectl port-forward svc/clusterip-service 83:8888` In the `Clusterip-service` service in the pod, and enter the URL `127.0.0.1:83` in the browser to access the address to get the result as shown in the figure below.



Hello world!

My hostname is clusterip-deployment-655d8f7c-q9d9t

Links found

SMASHING listening in 3030 available at tcp://10.106.4.84:3030
EXAMPLESERVICE listening in 8080 available at tcp://10.103.162.31:8080
CLUSTERIP_SERVICE listening in 8888 available at tcp://10.111.216.246:8888
KUBERNETES listening in 443 available at tcp://10.96.0.1:443

Conclusion:

Docker's lightweight virtualization technology helps us to conveniently apply containers anytime and anywhere. When Docker is applied to specific business implementations, docker has difficulties in scheduling, management, and scheduling. K8s helps solve this problem with a cluster system. It helps to monitor the Pod assigned to the Node where it is located, and realizes the automatic deployment, automatic scaling and maintenance of the container cluster. Simply put, kubernetes is used to manage the Docker cluster, that is, Docker can be regarded as a low-level internal use of Kubernetes Components.

FINAL GRADE

68/100

GENERAL COMMENTS

Instructor

This is a very well-written and nicely structured report, with a good implementation. Unfortunately, however, there are no sources to indicate where your information was derived from. It's imperative that your academic work contains a source list along with inline citations to indicate the origins of your ideas.

If you'd like to discuss this, please don't hesitate to ask - this is the only thing which excluded you from a very high mark in this work.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24
