k8s之docker基础

1.docker 基础命令

docker-info 之 LiveRestoreEnable:

配置方式: deamond.json
{
    "live-restore":true
}
生产环境下面需要设置这个为true，防止docker重新启动之后不拉取对应的容器。

docker cp命令使用
dcoker cp [本地文件] [docker 容器ID]:[docoker容器路径] , 反过来亦然。即把容器的文件拷贝到本地
例子:
将本地目录下的index.html文件拷贝到docker的NGINX容器中的 /usr/share/nginx/html , 替换原有的index.html
docker cp /Users/yurisa/Desktop/index.html 51e721a20403:/usr/share/nginx/html

docker history 查看一个镜像的修改记录

docker commit 将容器的内部变更保存至镜像
例子:
比如更改了nginx的index.html想保存这个镜像，则使用如下命令
-- docker commit -a '作者' -m '提交日志' 容器ID 容器名称:TAG
docker commit -a 'yzh' -m 'modify nginx file' 51e721a20403 nginx:commit

docker run 启动镜像的时候，加上--rm命令方便容器在退出的时候自动清理容器内部的数据。

Dockerfile 内容相关:

FROM: 继承的基础镜像

MAINTAINER:镜像制作者信息 新版建议使用LABEL maintainer="xxx"

RUN:用来执行的shell命令

EXPOSE:暴露的端口号

CMD:启动容器默认执行的命令

ENTRYPOINT:启动容器真正执行的命令

VOLUME:创建挂在卷

ENV:配置环境变量

ADD:复制文件到容器

COPY:复制文件到容器

WORKDIR:设置容器的工作目录

USER:容器使用的用户

CMD和ENTRYPOINT 必须要有一个, 如果有ENTRYPOINT,CMD就是ENTRYPOINT的参数

ADD和COPY异同，如果是ADD，则复制一个TAR的包的时候会自动解压，COPY则不会。

Dockerfile例子:

chat01:
Dockerfile内容

```
FROM alpine:latest
LABEL maintainer="yurisa@9527"
LABEL test_env="developer"

RUN mkdir -p /home/yurisa
RUN adduser -D yurisa
```

在该文件的目录下面执行docker build

```
docker build -t alpine:chat01 .
```

运行该镜像

```
docker run -ti --rm alpine:chat01 sh
```

一般不建议RUN 运行多次，每一个RUN,对应一个docker 的镜像层。

```
... 前面的内容省略
RUN adduser -D yurisa && mkdir -p /home/yurisa # 将多条命令合并成一条
```

chat02:
CMD的使用

```
FROM alpine:latest
LABEL maintaienr="yurisa@9527"
LABEL environment="chat02"

RUN adduser -D yurisa2 && mkdir -p /home/yurisa2

CMD ["sh","-c","echo maintaienr=yurisa@9527"]
```

build镜像:
Dockerfile文件使用执行方式

```
 docker build -t alpine:chat02 -f chat02/Dockerfile .
 # 注意build命令最后一个`.` ，指代上下文的意思
```

运行镜像:

```
docker run -it --rm alpine:chat02
# 运行结果:即cmd的内容:
echo maintaienr=yurisa@9527
```

```
# 如果在运行的镜像后面加上参数，则参数会覆盖cmd的命令
docker run -it --rm alpine:latest sh
# sh会覆盖cmd的内容
```

chat03 ENTRYPONIT使用：

```
FROM alpine:latest
LABEL auth="chat03"

ENTRYPOINT ["echo"]
CMD ["hello world"]
```

构建镜像命令同上,运行镜像结果如下：

```
➜  dockerfilers docker run -ti --rm alpine:chat03
hello world
➜  dockerfilers docker run -ti --rm alpine:chat03 'yurisa love'
yurisa love
# 如果运行过程中添加参数，则参数会把CMD覆盖掉
```

chat04 ENV使用:

ENV 书写形式如下，一般构建镜像的时候 ， ENV建议从外部传入,因为每一个ENV指令都是一个镜像层，会增加镜像的大小。

```
FROM alpine:latest
ENV VERSION=1.0 DEBUG=on \
    NAME="Happy Feet"
ENV AUTH "yurisa"
ENV GOVERSION=1.20.4

CMD echo "$VERSION,$NAME,$DEBUG,$AUTH,$GOVERSION"

```

镜像build同上，镜像输出为:

```
➜  dockerfilers docker run -ti --rm alpine:chat04
1.0,Happy Feet,on,yurisa,1.20.4
```

chat05 ADD与COPY的使用:

```
FROM alpine:latest
LABEL auth="yurisa"

WORKDIR /yurisa
ADD ./composess.tar /yurisa/add/
COPY ./composess.tar /yurisa/copy/

```

由于ADD以及COPY的本地路径都是以"."开始的路径，所以需要切换到当前的路径下面打包

```
# 打包镜像
➜  chat05 docker build -t alpine:chat05 .
# 运行镜像
➜  chat05 docker run -ti --rm alpine:chat05 sh
# 查看 ADD 与 COPY 的结果
/ # cd yurisa/
/yurisa # ls -l
total 8
drwxr-xr-x    2 root     root          4096 May 30 16:30 add
drwxr-xr-x    2 root     root          4096 May 30 16:30 copy
/yurisa # tree .
.
├── add
|   ├── 1.txt
|   ├── 2.txt
|   ├── Dockerfile
|   └── composess.tzr
```

```
└── copy
    └── composess.tar

2 directories, 5 files
```

可以看到通过add进来的tar压缩包，ADD进文件夹的过程中自动解包了。
而通过COPY进来的压缩包，则是原封不动的放在目录下面。

ADD与COPY命令异同，ADD可以添加当前路径下的目录以及目录内容到docker文件系统中，而COPY命令只能添加当前路下的目录里面的文件到docker文件系统中，不会添加路径下的目录进docker的文件系统重。

apline中安装tree命令:

```
apk add --no-cache tree
```

docker 清除 <none>镜像:

```
docker image prune
```

chat06 WORKDIR使用:
WORKDIR类型linux的chroot指令，用来更该特定的root路径。

```
FROM alpine:latest
LABEL Auth="yurisa"

RUN apk add --no-cache tree;
WORKDIR /yurisa
ADD ./1.txt /yurisa/a/
ADD ./2.txt /yurisa/b/

CMD pwd;tree;

```

镜像build同上。镜像的输出为:

```
➜  chat06 docker run -ti --rm alpine:chat06
/yurisa
.
├── a
|   └── 1.txt
└── b
    └── 2.txt

2 directories, 2 files
```

镜像打印出了当前的运行路径为workdir的目录。

caht07 USER的使用:
镜像默认的用户一般都是root，有时候为了安全考虑，会限制镜像的默认用户不是root.

```
FROM alpine:latest
LABEL Auth="yurisa"

RUN adduser -D yurisa
USER yurisa
CMD whoami;
```

镜像build同上，镜像输出为:

```
➜  chat07 docker run -ti --rm alpine:chat07
yurisa
```

可以看到当前使用的用户已经发生改变。

chat08 VOLUME 挂在卷使用:

```
FROM alpine:latest
```

```
LABEL Auth="yursia"
WORKDIR yurisa
VOLUME ["/yurisa/aaa","/yurisa/bbb"]
CMD ls -l;
```

镜像build同上，镜像运行的结果如下:

```
➜  chat08 docker run -ti --rm alpine:chat08
total 8
drwxr-xr-x    2 root     root          4096 May 30 17:28 aaa
drwxr-xr-x    2 root     root          4096 May 30 17:28 bbb
```

该镜像在yurisa目录下面创建了两个挂载点:

1. 如果镜像不适用 –v 参数挂载外部卷的话，则docker会在指定的本机路径放这两个卷,则在容器运行结束的时候该卷的 内容清 理.

```
docker inspect 镜像ID
```

2. 使用–v参数挂载容器外部卷

```
docker run -ti --rm -v /Users/yurisa/Desktop/dockerfilers:/yurisa/aaa -v
/Users/yurisa/Desktop/yzh:/yurisa/bbb alpine:chat08 sh
```

进入容器可以看到 aaa 文件夹已经映射到了 /Users/yurisa/Desktop/dockerfilers,ls 看到当前目录下的文件:

```
/yurisa # pwd
/yurisa
/yurisa # ls -l
total 0
drwxr-xr-x   11 root     root           352 May 30 17:21 aaa
drwxr-xr-x    2 root     root            64 May 30 17:36 bbb
/yurisa # cd aaa
```

```
/yurisa/aaa # ls -l
total 0
drwxr-xr-x    3 root     root            96 May 29 16:04 chat01
drwxr-xr-x    3 root     root            96 May 29 16:05 chat02
drwxr-xr-x    3 root     root            96 May 29 16:40 chat03
drwxr-xr-x    3 root     root            96 May 29 16:45 chat04
drwxr-xr-x    7 root     root           224 May 30 16:24 chat05
drwxr-xr-x    5 root     root           160 May 30 17:02 chat06
drwxr-xr-x    3 root     root            96 May 30 17:13 chat07
drwxr-xr-x    3 root     root            96 May 30 17:21 chat08
```

在本机文件系统"/Users/yurisa/Desktop/yzh"里面增加新文件 final.docx,在容器的bbb目录里面ls查看当前的文件系统:

```
/yurisa # cd bbb
/yurisa/bbb # ls -l
total 1808
-rw-r--r--    1 root     root       1850716 Jun 10  2022 final.docx
/yurisa/bbb #
```

发现文件已经存在了容器的bbb目录里面。

k8s-docker，多阶段制作镜像:

1.制作小镜像，不应该使用centos、ununtu这种体积特别大的镜像。
推荐使用Alpine，Busybox镜像， 以及Scratch (空镜像)

2.1.下载Golang镜像，基于Linux/Arm64平台生成Golang二进制文件。

```
FROM golang:latest
LABEL ImageName="golang:hello"

WORKDIR /yurisa
COPY ./main.go /yurisa
RUN go build -o hello /yurisa/main.go
```

```
CMD ./hello

```

build该镜像，查看镜像大小

```
golang                          hello               40e18faca859    3 minutes ago    773MB
```

运行该镜像:

```
➜  chat09 docker run -ti --rm golang:hello
[linux,arm64] golang.<hello world> [2023-06-01T16:47:12Z]
```

2.2 分多阶段构建镜像，缩小镜像的体积:

```
FROM golang:latest
LABEL IMAGE="GOLANG:Hello"
WORKDIR /yurisa
COPY ./main.go /yurisa
RUN go build -o hello ./main.go
CMD "./hello"

FROM alpine:latest
LABEL IMAGE="ALPINE:Hello"
WORKDIR /yurisa
# --from=0 指的是第一个FROM
COPY --from=0 /yurisa/hello /yurisa/
RUN mv /yurisa/hello /yurisa/alpine
CMD "./alpine"

```

build该镜像，并比较该镜像与golang:hello大小对比:

```
REPOSITORY                      TAG                 IMAGE ID       CREATED          SIZE
```

```
alpine                          hello               a76f62d764b3    6 seconds ago
9.35MB
golang                          hello               40e18faca859    11 minutes ago   773MB
```

运行该镜像查看运行结果

```
➜  chat10 docker run -ti --rm alpine:hello
[linux,arm64] golang.<hello world> [2023-06-01T16:57:26Z]
```

2.3 多阶段构建镜像使用别名:

```
FROM golang:latest AS golang
LABEL IMAGE="GOLANG:Hello"
WORKDIR /yurisa
COPY ./main.go /yurisa
RUN go build -o hello ./main.go
CMD "./hello"

FROM alpine:latest
LABEL IMAGE="ALPINE:Hello"
WORKDIR /yurisa
# --from=0 指的是第一个FROM
# --from=golang 构建过程使用别名，更好区分
COPY --from=golang /yurisa/hello /yurisa/
RUN mv /yurisa/hello /yurisa/alpine
CMD "./alpine"

```

运行结果与上面无异。

3.scratch镜像(空镜像)使用

```
FROM golang:latest AS golang
LABEL IMAGE="GOLANG:Hello"
WORKDIR /yurisa
```

```
COPY ./main.go /yurisa
RUN go build -o hello ./main.go
CMD "./hello"

FROM scratch
LABEL IMAGE="ALPINE:Hello"
COPY --from=golang /yurisa/hello /
CMD "./hello"
```

build镜像同上, 对比镜像大小:

```
REPOSITORY                    TAG                 IMAGE ID       CREATED
SIZE
scratch                       hello               5418370d4008   About a minute ago
2.01MB
alpine                        hello               a76f62d764b3   19 minutes ago
9.35MB
golang                        hello               40e18faca859   30 minutes ago
773MB
```

运行scratch:hello镜像， （scratch既没有bash也没有sh。）

```
➜  chat10 docker run -ti --rm scratch:hello ./hello
[linux,arm64] golang.<hello world> [2023-06-01T17:16:34Z]
```

注：scratch只适合没有任何依赖的二进制文件运行。