

Docker入门

原创

Zerg_Wang

已于 2022-02-21 17:18:30 修改

81

收藏

编辑

版权

分类专栏：


Management

 文章标签：

docker

容器

运维



Management 专栏收录该内容

0 订阅

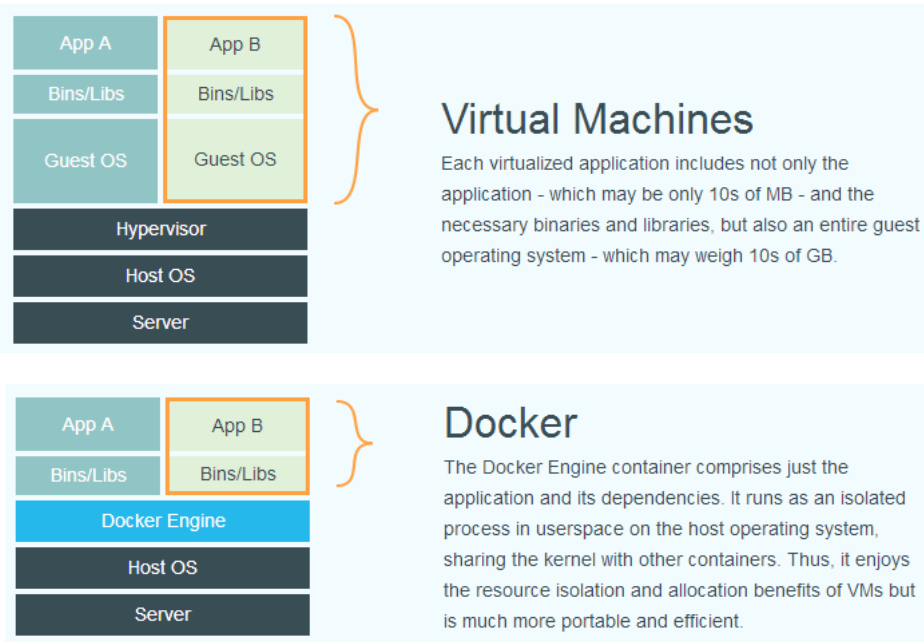
9 篇文章

Docker 简介

Docker是Linux 容器的一种封装，它将用户应用程序与该程序的依赖项打包成一个文件，用户要运行其程序时，Docker就会生成一个与外界隔离的虚拟容器，将打包的文件放置其中运行，从而解决程序运行环境不兼容的问题。

这里所谓的“容器”与虚拟机类似，但比虚拟机效率更高、使用更为方便。一台虚拟机包括应用，必要的二进制和库，以及一个完整的用户操作系统。而容器与宿主机共享硬件资源及操作系统，实现对资源的动态分配。此外，容器对进程进行隔离，相当于是正常进程的外面套了一个保护层。里面的进程接触到的各种资源都是虚拟的，从而实现与底层系统的隔离，不受环境影响。

得益于容器的这些特性，我们可以通过Docker方便地创建和使用容器，部署与管理自己的代码、配置和依赖关系，免除运行环境兼容的烦恼，提高开发效率。



App A

Bins/Libs

App B

Bins/Libs

Docker Engine

Host OS

Server

Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

直接比对		
特性	Docker	VM
启动速度	秒级	分钟级
硬盘使用	MB	GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

Docker为CS（Client-Server）架构，通过Docker Daemon以及Docker Client与用户交互。Docker Daemon为Docker守护进程，运行在宿主机上，用户通过Docker Client的命令行工具与Docker Daemon通信，Daemon接受用户命令，对宿主机执行相应操作并返回结果。

这里有几个名词需要解释一下：

镜像（Image）

镜像中包括了要运行的程序以及其依赖的库，镜像只读，用于创建容器。镜像相当于模板，与容器的关系类似于类与对象。镜像可存储于宿主机中，也可存储于仓库（Registry）中。镜像的创建主要有以下三种形式：通过拉取他人现成镜像、从无到有创建或基于现有镜像创建。

容器（Container）

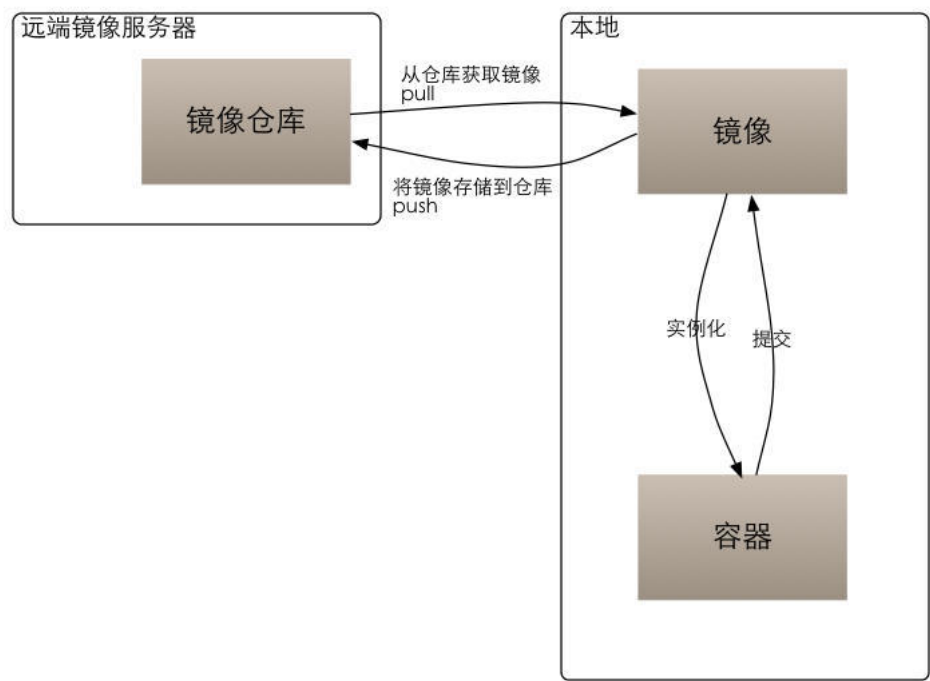
用户通过Docker运行的就是容器。容器基于某个镜像生成，可通过Docker Client使容器执行或停止等。

仓库（Registry）

存放镜像的地方。用户可从官方维护的仓库Docker Hub中获取镜像（需要注册Docker账号），也可以搭建自己的私人Registry（相当于一个存镜像的网盘）并从中获得镜像。

Docker File

Docker File为一种特殊的文本文件，其内容由多条指令构成，用于用户DIY自己的镜像。



开始

Linux下安装Docker：

```
wget -qO- https://get.docker.com/ | sh
```

启动Docker服务：

```
sudo service docker start
```

从Docker Hub拉取一个hello-world镜像：

```
docker pull hello-world
```

每个镜像都会有一个标签（tag）来表示其版本，在镜像名后接冒号表示，例如hello-world:v1，若在命令中不指定版本，则默认为latest（最新版）

运行hello-world：

```
docker run hello-world
```

常用命令

以下命令均要以docker开头，命令后接Container ID或容器名来指定容器

更多具体的命令可在[Docker 命令大全 | 菜鸟教程](#)中获取。

镜像管理

查看镜像详细信息：history（容器也可用该命令查看）

查看本机镜像：images

删除镜像：rmi（若存在以此生成的容器，则镜像无法删除）

容器管理

查看本机活动中的容器：ps

查看本机所有容器：ps -a

创建并开始运行容器：run

仅创建容器：create

开始运行某容器：start

重启容器：restart

暂停容器：pause（解除暂停：unpause）

停止一个容器：stop（强行停止一个容器：kill，两者的区别是，前者在停止容器时会提前发出信号，让容器做一些保护性措施）

删除容器：rm（不可删除运行或暂停状态下的容器）

批量操作（例如批量停止容器）：docker stop \$(docker ps -a -q)

其他

登录到Docker Hub：login -u 用户名 -p 密码

Dockerfile详解

Docker File用于定制用户自己的镜像，其指令主要由四部分组成：基础镜像信息、维护者信息、镜像操作指令、容器启动执行指令。此外，Dockerfile支持“#”开头的注释信息（与Python类似）。

基础镜像信息必须是Dockerfile中第一条指令，用于指定该镜像是基于哪个镜像模板创建的，格式为（不指定版本则默认为latest。）：

```
FROM 镜像名:版本
```

该镜像模板可不必事先pull，执行Dockerfile指令时会自动pull（如果本地没有的话）。

若想白手起家，不基于其他镜像创建一个Base镜像，可以FROM scratch

维护者信息可以没有，用来声明镜像作者及其联系方式。（可以无联系方式）

```
MAINTAINER 作者名 作者联系方式
```

接下来重点介绍镜像操作指令和容器启动执行指令。

WORKDIR

```
WORKDIR path
```

在容器内设置一个工作目录，之后Dockerfile中的RUN、CMD等命令会在该目录下执行。若不指定（或指定为当前目录，即WORKDIR .），则在docker看来是/，即根目录，而在宿主机中，其实际工作目录在/var/lib/docker/下某个目录中。如果指定为某个目录，实际上这个目录也会在/var/lib/docker/下面的某个目录中，例如：

```
1 COPY sum.py .
2 WORKDIR .
```

更改后的Dockerfile：

```
1 COPY sum.py /home/Wang/python-test/container/  
2 WORKDIR /home/Wang/python-test/container/
```

最后在宿主机中查找sum.py，发现更改前后的镜像分别对应上下两行：

```
/var/lib/docker/overlay2/d2f6895827f57fbc8684888162556bb0ad16e1846fa3c923d41d9ceb51bf127f/diff/sum.py  
/var/lib/docker/overlay2/41d32127c58c7ac337cf0d84008f3ac91f081f6c2d052fb773dc385c724df13f/diff/home/Wang/python-test/container/sum.py
```

虽然这个路径搞得挺麻烦的，但不用担心空间占用，镜像删除的话，这些文件也会一并删除。

COPY

`COPY file path`

path可以是容器内的绝对路径，也可以是相对于工作目录的相对路径，若路径不存在不必提前创建，执行时会自动创建。

前一个参数file若只填文件名不填路径，默认的路径与Dockerfile处于同目录下。后一个path若为.则为docker默认目录（详见上文WORKDIR命令）

ADD

`ADD file path`

更为高级的一种复制命令，复制的如果是压缩包，文件会自动解压到path中。此外，file可填入网络链接，执行时会自动下载。

ENV

设置环境变量，之后运行的指令及容器运行时都可用。

```
1 ENV key value  
2 ENV key1=value1 key2=value2...
```

RUN

在镜像的构建过程中执行指定命令。一般使用RUN安装应用和软件包（apt-get install），该命令一般会与软件包升级命令一起执行（apt-get update），这是因为若单独执行RUN apt-get update，若之前有执行过同样的命令，会留下缓存，导致此次镜像构建的时候直接使用之前的缓存，未能实现升级，因此常将其与install同写。

RUN的指令格式有两种，一种为Shell格式（与Linux命令一致），另一种为Exec格式，后面的CMD与ENTRYPOINT也都支持这两种格式的命令，区别：

Shell格式：instruction command

```
1 RUN apt-get install python3  
2  
3 CMD echo "Hello world"  
4  
5 ENTRYPOINT echo "Hello world"
```

Exec 格式：instruction ["executable", "param1", "param2", ...]

```
1 RUN ["apt-get", "install", "python3"]  
2  
3 CMD ["/bin/echo", "Hello world"]  
4  
5 ENTRYPOINT ["/bin/echo", "Hello world"]
```

CMD

指定容器启动时所执行的命令。这条命令不会在build镜像时执行，只在容器运行时执行，因此该命令不产生单独一层的文件系统。

CMD指令可存在多个，但只有最后一个生效。

若docker run指定了命令，则CMD命令会被忽略。

ENTRYPOINT

与CMD类似，存在多个的话，只会执行最后一个。不同的是，ENTRYPOINT命令不会被docker run顶替。

ARG

制定变量并初始化。（镜像建立后失效）

```
ARG 变量名=默认值
```

镜像架构

文件系统分层

在用户看来，镜像似乎就是一个按Dockerfile构建后的一个包，实际上，镜像并不是“一个文件”，它是由多个文件系统构成的。在基于Dockerfile构建镜像时，我们可以看到：

```
Sending build context to Docker daemon   2.56kB
Step 1/3 : FROM nginx
--> bb776ce48575
Step 2/3 : MAINTAINER ZERG
--> Running in b861b4699f18
Removing intermediate container b861b4699f18
--> d3c21d398eed
Step 3/3 : ADD 123.txt .
--> 51627b180311
Successfully built 51627b180311
Successfully tagged empt3:latest           https://blog.csdn.net/Zerg_Wang
```

每执行Dockerfile中的一步，都会生成一个ID，这个就是镜像的文件系统。镜像通过FROM得到其基础的文件系统，之后的指令就如同“叠叠乐”，在该镜像原有的文件系统上一层层叠加文件系统，直到Dockerfile指令结束。

通过以下命令可以查看镜像的构成。

```
docker history 镜像名:版本
```

文件系统缓存

因为镜像分层构建的体系，一份镜像可供多个容器使用。此外，若在构建新镜像时Dockerfile中有与之前相同的命令，则在构建当前镜像时，会直接使用之前的文件系统（缓存），如图：

```
Step 2/3 : MAINTAINER ZERG
--> Using cache
--> 3776d111fc60
```

然而，这些缓存也是要基于其下层文件系统而存在的，不可能真的和叠叠乐中的积木一样，可以直接抽出来使用。因此在构建镜像的某一步若找不到缓存，则在执行之后的构建命令时，即使指令一模一样都不会再使用缓存。此外，不仅本地构建镜像时会复用缓存，从Docker Hub上pull时也会复用。

文件系统只读

当容器运行的时候，镜像会在其文件系统上再加载一个读写层。然而，容器运行时可能有对自身的读写需求，但文件系统只读，这时读写层会记录这些命令，某些被更改的文件系统会被其标记，但文件系统本身不变。

例如，我以原来的一个输出“Hello I am Zerg”的镜像生成了容器，但我感觉这句输出太蠢，通过docker exec命令直接对容器修改，改成了“Hello Docker”，现在想以该容器生成镜像（使用docker commit），生成之后查看新镜像的构成：

```
Wang@VM-0-11-ubuntu:~$ docker history nginx:zerg-v3
IMAGE          CREATED          CREATED BY                                      SIZE
10e3127fbd22   17 seconds ago  nginx -g daemon off;                          150B
6e01b5addfe4   2 days ago     /bin/sh -c echo 'Hello, I am Zerg.' > /usr/s... 18B
70798f9346bd   2 days ago     /bin/sh -c #(nop) MAINTAINER ZergWang          0B
```

可以发现，刚刚的更改完全没有改写之前只读的文件系统，而是在最上面附加了一层。

PS：不推荐使用commit命令创建镜像，因为无法获知具体创建命令（如上图，用history也看不到），而且会使得镜像所叠层数过多。

实战：基于Nginx镜像的Hello-World

制作一个以Nginx为模板的hello-world镜像作为例子来解释Docker File指令。

首先在空白目录中新建文件Dockerfile，编辑其内容：

```
1 FROM nginx
2 RUN echo 'Hello, I am Zerg.' > /usr/share/nginx/html/index.html
```

保存退出后在该目录下运行以下命令来根据我们刚刚编辑的Dockerfile生成镜像：

```
docker build -t 镜像名:镜像版本 .
```

名字和版本可自定义，但要求小写英文字母或数字。注意命令最后有个点，表示当前目录。

通过该镜像生成并运行容器：

```
docker run --name 容器名 -d -p 宿主机端口:Docker端口 镜像名:镜像版本
```

我的是：

```
docker run --name my-container -d -p 8080:80 my-image:v1
```

这里的端口是指将宿主机的某个端口映射为docker的某个端口，实现服务的接入。

打开宿主机的地址：



Hello, I am Zerg.

实战：Python程序部署

编写Dockerfile

搞个最简单的：（本来服务器中装的是python3.5，为了区别出是docker中的python，指定安装了3.6版本的）

```
1 FROM python:3.6
2 MAINTAINER ZergWang
3 COPY sum.py .
4 WORKDIR .
5 CMD python sum.py
```

路径设成默认就好了，sum.py与Dockerfile在同一目录下，事先编好：

```
1 a = 3
2 b = 4
3 print(a+b)
```

在Dockerfile的目录中生成镜像：

```
docker build -t python-test:v1 .
```

运行镜像：

```
docker run python-test:v1
```

结果：

```
root@python-test:~/python-test$ docker run python-test:v1
7
root@python-test:~/python-test$
```

其他

免sudo使用docker

```
sudo ls -l /var/run/docker.sock
```

查看docker权限发现，只有docker对应的用户组才有充足权限，因此将当前用户添加到docker所属用户组即可。

```
sudo gpasswd -a 用户名 docker
```

然后重新登录并重启docker服务：

```
sudo service docker restart
```

最后将会话切换到docker组中即可。

```
newgrp - docker
```

参考资料

这可能是最为详细的Docker入门吐血总结_编程语言_技术博文_js代码

[如何免 sudo 使用 docker - 泰晓科技](#)

[Docker\(二\)：Dockerfile 使用介绍 - 纯洁的微笑 - 博客园](#)

[Docker 教程 | 菜鸟教程](#)

[前言 - Docker —— 从入门到实践](#)

[Docker\(三\)：Dockerfile 命令详解 - 纯洁的微笑 - 博客园](#)

[RUN vs CMD vs ENTRYPOINT - 每天5分钟玩转 Docker 容器技术 \(17 \) - CloudMan - 博客园](#)