Docker练习题目

1. Hello World & 基本命令

要求：

- 运行官方 `hello-world` 镜像
- 依次执行以下命令并理解作用：

```
docker pull hello-world
docker run hello-world
docker ps -a  # 查看容器状态
docker logs <容器ID>  # 查看日志
docker stop/rm <容器ID>  # 停止/删除容器
```

验收标准：

- 能解释每条命令的作用
- 成功看到 `Hello from Docker!` 输出

---

2. Nginx 网站容器

要求：

- 运行 Nginx 容器并映射主机 `8080` 端口到容器 `80` 端口
- **两种方式修改默认页面**：
  1. 进入容器直接修改 `/usr/share/nginx/html/index.html`
  2. 使用 `bind mount` 将本地 HTML 文件挂载到容器
- 验证浏览器访问 `http://localhost:8080` 显示自定义内容

验收标准：

- 能通过两种方式修改页面内容
- 理解 `-v` 和 `--mount` 挂载的区别

---

3. 持久化 MySQL 数据库

要求：

1. 启动 MySQL 容器：

```
docker run --name mysql-db -e MYSQL_ROOT_PASSWORD=123456 -v
mysql_data:/var/lib/mysql -d mysql:8.0
```

2. 进入容器创建数据库和表：

```
docker exec -it mysql-db mysql -uroot -p123456
# 执行 SQL 创建数据库和表
```

3. 删除容器后重新挂载相同卷，验证数据是否保留

验收标准：

- 数据卷（`mysql_data`）能持久化数据库
- 重新创建的容器能读取之前的数据

---

4. 自定义 Python Flask 镜像

要求：

- 编写 `Dockerfile` 构建镜像，要求：
  - 基于 `python:3.9-slim`
  - 安装 `flask` 依赖（通过 `requirements.txt`）
  - 暴露 `5000` 端口
  - 启动命令运行 `app.py`
- 构建镜像并运行容器，测试 API 访问

验收标准：

- 镜像大小不超过 150MB（使用 `docker images` 查看）
- 能通过 `curl http://localhost:5000` 访问 Flask 应用

---

5. Docker Compose 编排 WordPress

要求：

- 编写 `docker-compose.yml` 包含：
  - `wordpress` 服务（依赖 `mysql`）
  - `mysql` 服务（配置密码和数据卷）
  - 自定义网络
- 通过 `docker-compose up` 启动，验证 WordPress 安装页面

验收标准：

- 能通过 `http://localhost:8000` 访问 WordPress
- 数据卷确保数据库持久化

---

6. 多阶段构建优化

要求：

- 以 Go 或 Java 应用为例：
  1. 第一阶段：使用完整 SDK 编译应用
  2. 第二阶段：仅复制二进制文件到轻量级镜像（如 `alpine`）

- 对比单阶段和多阶段构建的镜像大小

验收标准：

- 多阶段构建的镜像比单阶段缩小至少 50%

---

7. 容器网络实践

要求：

1. 创建自定义网络：

   ```
   docker network create my-net
   ```

2. 部署后端（如 Flask）和前端（如 Nginx）服务：
   - 使用 `--network my-net` 启动
   - 测试前端能否通过容器名访问后端（如 `http://backend:5000`）

验收标准：

- 理解容器间通过服务名通信的原理

---

8. CI/CD 自动化构建

要求：

- 创建 GitHub Actions 工作流：
  1. 代码推送时自动构建 Docker 镜像
  2. 打上 `git commit ID` 作为标签
  3. 推送镜像到 Docker Hub
- 使用 `docker-compose` 自动拉取最新镜像部署

验收标准：

- 代码提交后能在 Docker Hub 看到新镜像

---

9. 容器安全加固

要求：

- 运行一个加固的 Nginx 容器：
  - 使用 `--user 1000` 非 root 用户
  - 设置 `--read-only` 只读文件系统
  - 限制内存（`--memory 512m`）和 CPU（`--cpus 1`）
- 测试容器功能是否正常

验收标准：

- 能通过 `docker inspect` 验证安全配置生效

---

10. 实际项目容器化

要求:

- 选择一个已有项目（如 Node.js/Spring Boot）：
    1. 编写 `Dockerfile` 和 `.dockerignore`
    2. 处理配置文件、日志输出到主机
    3. 使用 `docker-compose` 编排依赖服务（如 Redis）

验收标准:

- 项目能通过 `docker-compose up` 一键启动

每个练习均包含 **"动手操作"** 和 **"原理理解"** 两部分，建议配合 Docker 官方文档实践。

Docker练习题目解答

练习1 Hello World & 基本命令

1. 运行`docker pull hello-world`，输出结果如下：

```
Using default tag: latest
latest: Pulling from library/hello-world
e6590344b1a5: Pull complete
Digest: sha256:fc08e727181e2668370f47db6319815c279ed887e2f01be96b94106bc2781430
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

2. 运行`docker run hello-world`，输出结果如下：

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

3. 运行命令`docker ps -a`，查看容器状态。输出如下：

```
CONTAINER ID   IMAGE         COMMAND    CREATED         STATUS
  PORTS     NAMES
aa80097f567f   hello-world   "/hello"   3 minutes ago   Exited (0) 3 minutes ago
            busy_jackson
```

4. 运行命令`docker logs aa80097f567f`查看日志。输出如下：

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
```

```
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

5. 运行命令`docker stop/rm aa80097f567f`停止/删除容器。输出如下：

```
aa80097f567f/aa80097f567f
```

任务一完成！

练习2. Nginx网站容器

1. **运行Nginx容器：**

```
docker run -d -p 8080:80 --name mynginx nginx
```

- -d : 表示后台运行
- -p 8080:80 ： 表示将主机的8080端口映射到容器的80端口
- 访问 localhost:8080 显示默认界面

问题：访问http://123.60.84.41:8080/ 失败。

原因：

- 检查华为云安全组规则，放行8080端口

此时顺利访问成功！

2. **进入容器直接修改默认界面**
    1. 进入容器终端`docker exec -it mynginx /bin/bash`
    2. 修改默认界面：`echo "Hello from container!" > /usr/share/nginx/html/index.html`
    3. 验证修改：浏览器刷新，显示自定义内容。
    4. 修改成功！

3. **方式2 使用Bind Mount挂载本地文件**

    1. 准备本地的HTML文件

    ```
    mkdir ~/nginx-custom-html
    echo "Hello from host!" > ~/nginx-custom-html/index.html
    ```

    2. 启动容器，并使用`--mount`挂载本地目录

    ```
    docker run -d -p 8081:80 --name mynginx-mount \
      --mount type=bind,source=$HOME/nginx-custom-
    html,target=/usr/share/nginx/html nginx
    ```

    - `type=bind`：使用绑定挂载Bind Mount，将主机上的目录挂载到容器中。
    - `source=$HOME/nginx-custom-html`:主机上的目录路径作为源目录
    - `target=/usr/share/nginx/html nginx` 容器内的目标路径，默认的静态文件放置路径

    3. 访问http://localhost:8081 验证结果
    4. 成功挂载本地文件！

任务二完成！

练习3.持久化 MySQL 数据库

1. **启动MySQL容器并挂载数据卷**

```
docker run --name mysql-db -e MYSQL_ROOT_PASSWORD=123456 -v
mysql_data:/var/lib/mysql -d mysql:8.0
```

参数说明：

- -e : 表示设置容器内的环境变量（MySQL官方要求强制设置MYSQL_ROOT_PASSWORD）
- -v ： 将Docker卷中的`mysql_data`挂载到容器内的`/var/lib/mysql`目录。
    - MySQL 默认将数据写入`var/lib/mysql`目录，存储在主机的`mysql_data`中
    - `mysql_data`的默认地址：`/var/lib/docker/volumes/`
    - 可以通过`cd`命令查看
    - 成功创建！

2. **进入容器，并创建数据库及表**

    1. 执行命令进入MySQL命令行`docker exec -it mysql-db mysql -uroot -p123456`
        - 说明：`mysql -uroot -p123456`是在容器内执行的命令。
    2. 在MySQL中执行SQL语句

```
CREATE DATABASE testdb;
USE testdb;
CREATE TABLE example (id INT PRIMARY KEY, data VARCHAR(50));
INSERT INTO example VALUES (1, 'Persisted Data')
```

命令解释：

- `CREATE DATABASE testdb;`：创建一个名为`testdb`的新数据库
- `USE testdb;`：切换到数据库目录
- `CREATE TABLE example (id INT PRIMARY KEY, data VARCHAR(50));`： 创建example 表，包含两列。一列为id，int类型，设为主键；另一列为data，可变字符最多50个
- 向example表中插入一行数据。id为1，data字段为'Persisted Data'

3. **删除容器并重新挂载数据卷**

    1. 停止并删除原容器 `docker stop mysql-db` `docker rm mysql-db`
    2. 重新启动原容器并挂载数据卷 `docker run --name mysql-db-new -e MYSQL_ROOT_PASSWORD=123456 -v mysql_data:/var/lib/mysql -d mysql:8.0`
        - 说明：新的镜像名称 `mysql-db-new`

4. **验证数据持久化**

    1. 进入到数据库： `docker exec -it mysql-db-new mysql -uroot -p123456 "`
    2. 查看数据库信息： `SHOW DATABASES;` 输出如下：

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| testdb             |
+--------------------+
5 rows in set (0.01 sec)
```

    3. 查看`testdb`中的所有数据： `USE testdb; SELECT * FROM example;` 。输出如下：

```
+----+----------------+
| id | data           |
+----+----------------+
|  1 | Persisted Data |
+----+----------------+
1 row in set (0.00 sec)
```

任务3完成！

练习4. 自定义 Python Flask 镜像

1. 了解概念
   - Flask： 轻量级的Python Web框架，用于快速构建Web应用和API。
2. 项目文件结构示意图

```
flask-app/
├── Dockerfile
├── requirements.txt
└── app.py
```

3. 构建文件目录

```
mkdir flask-docker && cd flask-docker
touch Dockerfile app.py requirements.txt
```

   参数解释：
   - `touch`： 常用于创建空文件
4. 编辑app.py的内容

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_docker():
    return '<h1>Hello from Docker container!</h1>'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

   代码解释：
   - `app = Flask(__name__)`：创建一个Flask应用实例，`__name__` 表示当前模块的名称
   - `@app.route('/')`：定义URL路径。装饰器 `@app.route()` 将URL与函数绑定。
   - `if __name__ == '__main__':` 确保脚本直接运行时才启动服务器。
   - `port=5000`： 是默认端口
5. 编辑 requirements.txt 的内容

```
Flask==2.3.2
```

   说明：
   - pip安装时不区分大小写
6. 编写 `Dockfile` 文件

```
# 使用轻量级 Python 基础镜像
FROM python:3.9-alpine

# 设置工作目录
```

```
WORKDIR /app

# 复制依赖文件
COPY requirements.txt .

# 安装依赖（使用 --no-cache-dir 减少镜像体积）
RUN pip install --no-cache-dir -r requirements.txt

# 复制应用代码
COPY . .

# 暴露端口
EXPOSE 5000

# 启动命令（确保监听所有网络接口）
CMD ["python", "app.py"]
```

说明：

- `FROM`： 第一个指令，表示基于哪个镜像构建
- `WORKDIR`：设置工作目录
- `COPY`： 复制文件到容器中。第一个参数主机，第二个参数容器
- `RUN`：在容器中执行的命令
- `EXPOSE`：容器运行时监听的端口
- `CMD`：启动时运行的默认命令

7. 构建 Docker 镜像

```
docker build -t flask-app .
```

8. 运行容器

```
docker run -d -p 5000:5000 --name myflask flask-app
```

9. 验证容器运行

```
docker ps
```

10. 测试API访问

```
curl http://localhost:5000
```

11. 验证镜像大小

```
docker images | grep flask-app
```

说明：

- `grep`：文本搜索工具，按行匹配关键字
- 管道符 `|`：前一个命令的输出作为下一个命令的输入

任务4完成！

练习5 Docker Compose 编排WordPress

1. 零基础说明
   - Docker Compose： 容器管理工具，利用简单的YAML文件，来定义和运行多个容器。
   - WordPress： 流行的博客/网站系统
   - MySQL：数据库，用于存储WordPress的所有内容
2. 安装Docker Compose

   ```
   sudo apt install docker-compose-plugin
   ```

   说明：`docker-compose-plugin`：表示插件形式的docker compose
3. 创建`docker-compose.yml`文件，编排WordPress和MySQL服务，内容如下：

   ```
   version: '3.8'

   services:
     wordpress:
       image: wordpress:latest
       depends_on:
         - mysql
       ports:
         - "8000:80"
       environment:
         WORDPRESS_DB_HOST: mysql
         WORDPRESS_DB_USER: wordpress
         WORDPRESS_DB_PASSWORD: wordpress
         WORDPRESS_DB_NAME: wordpress
       networks:
         - wp-network
       restart: unless-stopped

     mysql:
       image: mysql:5.7
       environment:
         MYSQL_ROOT_PASSWORD: rootpassword
         MYSQL_DATABASE: wordpress
         MYSQL_USER: wordpress
         MYSQL_PASSWORD: wordpress
       volumes:
         - mysql_data:/var/lib/mysql
       networks:
         - wp-network
       restart: unless-stopped
   ```

```
volumes:
  mysql_data:

networks:
  wp-network:
    driver: bridge
```

- 说明：
  - 三大部分：Services定义服务 、 volumes 数据存储位置 、 networks 网络配置
  - WordPress服务
    - `image: wordpress:latest`：使用最新版的WordPress 官方镜像
    - `depends_on`：需要MySQL数据库先启动
    - `port: 8000:80`：端口映射
    - `environment`：设置连接数据库信息
    - `networks`：连接网络，表示服务器可以相互通信
    - `restart`：自动重启，除非手动停止
  - MySQL数据库服务
    - `image` ： 使用的镜像
    - `environment`：设置数据库的配置，root密码，数据库名称，用户名，密码
    - `volumes`：数据库保存位置
    -
  - `volumes:mysql_data`：定义存储位置，永久存储
  - `networks:wp-network`：创建虚拟网络，两者可以相互通信

4. 启动服务

```
docker compose up -d
```

说明：

- `docker compose` ： 命令
- `up`：表示启动
- `-d`：表示后台自己运行

5. 在浏览器中访问。

任务五完成！

练习六 多阶段构建优化

1. 准备一个简单的Java应用，保存为HelloWorld.java文件。

```java
// HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

2. 单阶段构建，创建Dockerfile.single文件。

```
# 使用完整 JDK 作为基础镜像
FROM openjdk:17-jdk-slim

# 设置工作目录
WORKDIR /app

# 将源代码复制到容器中
COPY HelloWorld.java .

# 编译 Java 源代码
RUN javac HelloWorld.java

# 运行应用
CMD ["java", "HelloWorld"]
```

说明：

- `run`：一般表示安装或者编译。

3. 多阶段创建，创建Dockerfile.multi文件。

```
# 第一阶段：使用完整 JDK 编译应用
FROM openjdk:17-jdk-slim AS builder

# 设置工作目录
WORKDIR /app

# 将源代码复制到容器中
COPY HelloWorld.java .

# 编译 Java 源代码
RUN javac HelloWorld.java

# 第二阶段：使用轻量级镜像运行应用
FROM alpine:latest

# 安装 Java 运行时环境
RUN apk add --no-cache openjdk17-jre

# 设置工作目录
WORKDIR /app

# 从第一阶段复制编译后的二进制文件
COPY --from=builder /app/HelloWorld.class .
```

```
# 运行应用
CMD ["java", "HelloWorld"]
```

说明：

- 第一阶段：使用完整的JDK来编译应用
- 第二阶段： 将编译的二进制（`.jar`文件）复制到轻量级镜像，减少最终镜像的大小

4. 构建镜像

```
docker build -t java-app-multi -f Dockerfile.multi
docker build -t java-app-single -f Dockerfile.single
```

说明：

- `-t`：表示为镜像指定名称和标签。
- `-f`：表示用于构建的Dockerfile文件名（因为有两个文件）

5. 查看镜像大小 `docker images`
6. 对比结果。

任务六完成！
