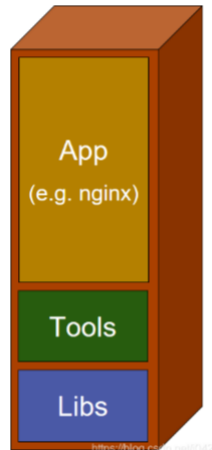


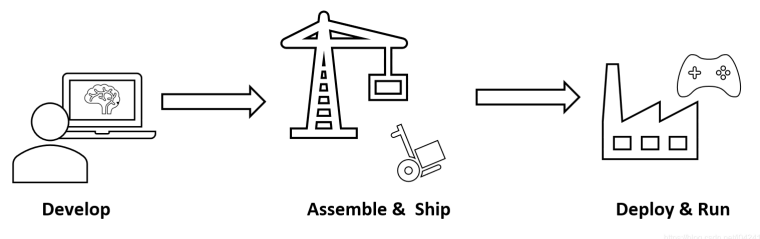
什么是容器？

- 定义：容器是一种轻量级的虚拟化技术，它在操作系统层面上实现了资源的隔离和限制，使得多个容器可以在同一台主机上共享操作系统内核，同时又能独立运行，互不干扰。简单地说，一个容器包含了完整的运行时环境：除了应用程序本身之外，这个应用所需的全部依赖、类库、其他二进制文件、配置文件等，都统一被打入了一个称为容器镜像的包中。通过将应用程序本身，和其依赖容器化，操作系统发行版本和其他基础环境造成的差异，都被抽象掉了。



- 为什么要用：
 - 场景1：作为程序员，让我们回忆我们每天从事的熟悉得不能再熟悉的软件开发工作：在本地搭好开发环境，进行开发工作，完了进行单元测试，把开发好的代码部署到测试系统，重复测试，最后部署到生产系统。

Traditional Software development and delivery process



- 场景2：我们不可避免地会遇到这种情况：同样的代码，运行环境发生变化之后无法正常运行。这种运行环境的变化可以分成不同的维度：比如代码从程序员的笔记本电脑切换到测试服务器，或者从一台物理服务器切换到公有云/私有云上；代码依赖的运行库版本发生变化，比如开发时用的python2.7，但生产机上用的python3；也可能是代码运行的操作系统发生了变化，比如开发及用的ubuntu，生产机用的redhat。程序员除了投入时间在应用程序本身开发上之外，还需要花费额外的精力去处理这种环境或者说infrastructure问题，有的时候很头痛。作为一个应用程序开发人员，我对底层的这些环境问题不感兴趣，有没有一种办法能使我不要去考虑它们呢？有，就是使用容器技术。
- 好处：
 - 使得镜像从一个环境移植到另外一个环境更加灵活。因为容器封装了所有运行应用程序所必需的相关的细节。
 - 标准化：大多数容器实现技术基于开放标准，可以运行在所有主流的Linux发行版、Microsoft等操作系统上。
 - 容器隔离带来的安全性：一台宿主主机上可以运行多个容器，但这些容器内的进程是相互隔离的，且无法相互感知。其中一个容器的升级或者出现故障，不会影响其他容器。

- 轻量级：虚拟机和容器的目的类似，都致力于对应用程序及其关联性进行隔离，从而构建起一套能够不依赖于具体环境而运行的应用单元。

Docker 快速安装软件

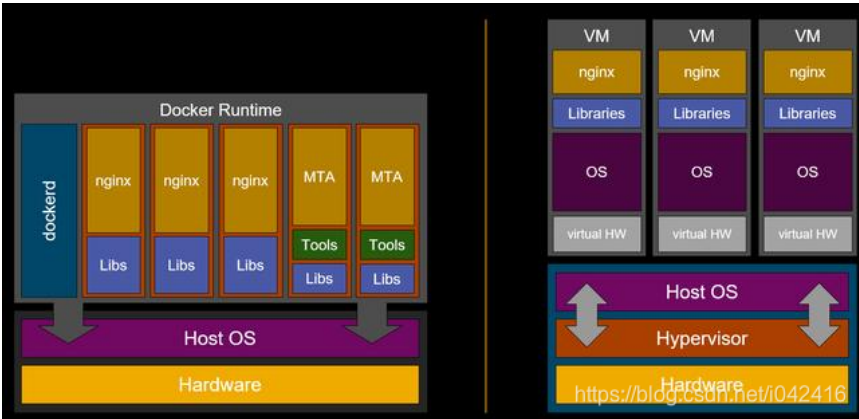
直接安装的缺点

- 安装麻烦，可能有各种依赖，运行报错。例如：WordPress，ElasticSearch，Redis，ELK
- 可能对 Windows 并不友好，运行有各种兼容问题，软件只支持 Linux 上跑
- 不方便安装多版本软件，不能共存。
- 电脑安装了一堆软件，拖慢电脑速度。
- 不同系统和硬件，安装方式不一样

本文档课件配套 [视频教程](#)

Docker 安装的优点

- 一个命令就可以安装好，快速方便
 - 有大量的镜像，可直接使用
 - 没有系统兼容问题，Linux 专享软件也照样跑
 - 支持软件多版本共存
 - 用完就丢，不拖慢电脑速度
 - 不同系统和硬件，只要安装好 Docker 其他都一样了，一个命令搞定所有
- 对比虚拟机：



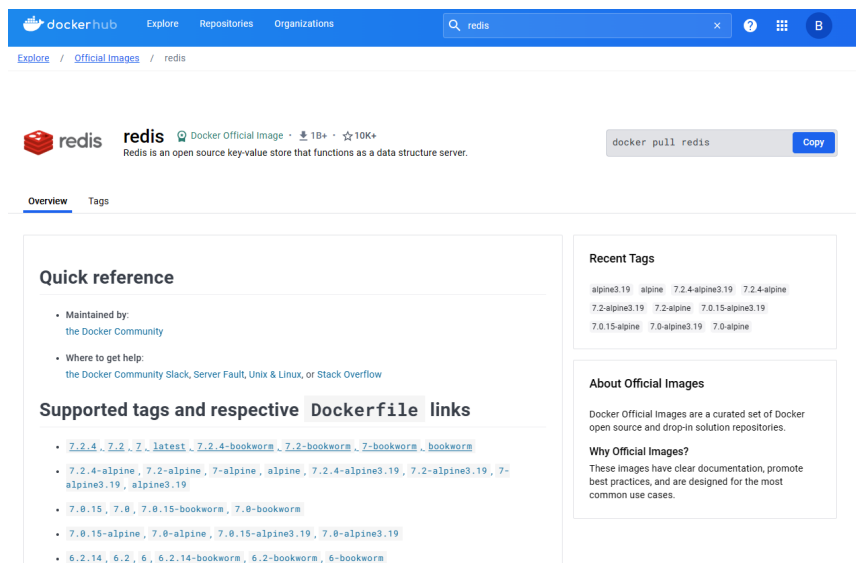
特性	普通虚拟机	Docker
跨平台	通常只能在桌面级系统运行，例如 Windows/Mac，无法在不带图形界面的服务器上运行	支持的系统非常多，各类 windows 和 Linux 都支持
性能	性能损耗大，内存占用高，因为是把整个完整系统都虚拟出来了	性能好，只虚拟软件所需运行环境，最大化减少没用的配置
自动化	需要手动安装所有东西	一个命令就可以自动部署好所需环境
稳定性	稳定性不高，不同系统差异大	稳定性好，不同系统都一样部署方式

解决方案（如何使用Docker）

- 使用流程：（与app icon寓意一致）
 - **打包**：就是把你软件运行所需的依赖、第三方库、软件打包到一起，变成一个安装包（镜像）
 - **分发**：你可以把你打包好的“安装包”上传到一个镜像仓库，其他人可以非常方便的获取和安装
 - **部署**：拿着“安装包”就可以一个命令运行起来你的应用，自动模拟出一模一样的运行环境，不管是在 Windows/Mac/Linux。



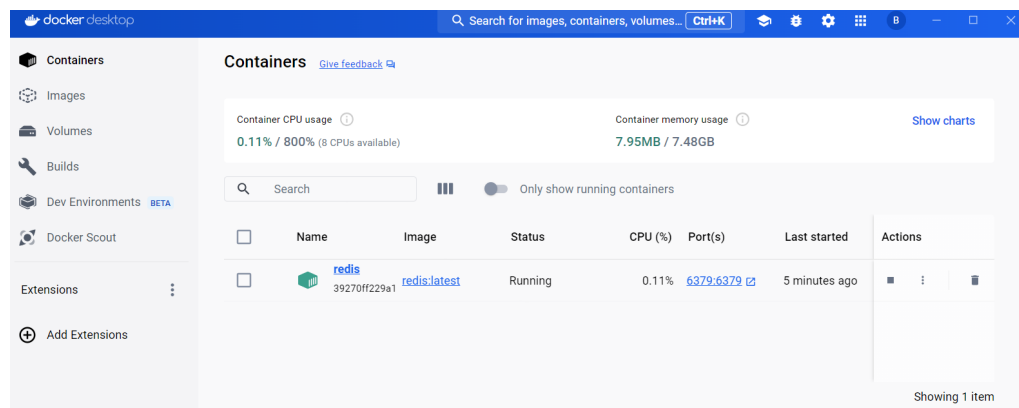
- 安装软件：



- `docker run -d -p 6379:6379 --name redis redis:latest`

- -d: 后台运行
- -p: 端口暴露，容器里的6379（后）暴露到宿主机的6379
- --name redis: 命名为redis
- redis:latest: 镜像源是最新版本的redis

```
PS C:\Users\CurryMars> docker run -d -p 6379:6379 --name redis redis:latest
Unable to find image 'redis:latest' locally
latest: Pulling from library/redis
8a1e25ce7c4f: Pull complete
8ab039a68e51: Pull complete
2b12a49dcfb9: Pull complete
cdf9868f47ac: Pull complete
e73ea5d3136b: Pull complete
890ad32c613f: Pull complete
4f4fb700ef54: Pull complete
ba517b76f92b: Pull complete
Digest: sha256:3134997edb04277814aa51a4175a588d45eb4299272f8eff2307bbf8b39e4d43
Status: Downloaded newer image for redis:latest
39270ff229a15322a7861abb9a269165e6cd777dbfaf253c94c5f433ff030643
PS C:\Users\CurryMars> |
```



- 制作自己的镜像：（举个web项目栗子）
 - 编写一个Dockerfile

```
# 通过FROM指定的镜像名称必须是一个已经存在的镜像，这个镜像称之为基础镜像，必须位于第一条非注释指令
FROM node:11

# 指定镜像的作者信息，包含镜像的所有者和联系人信息
MAINTAINER easydoc.net

# 将文件或目录复制到Dockerfile构建的镜像中
ADD . /app

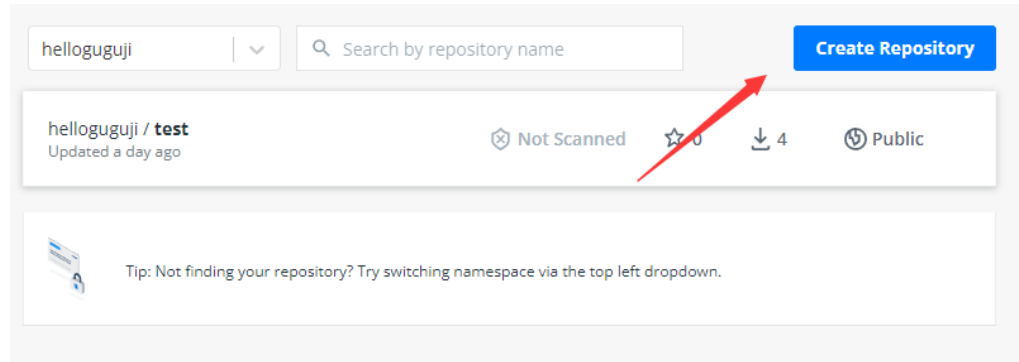
# 设置容器启动后的默认运行目录（在容器内部设置工作目录）
WORKDIR /app

# 运行命令，安装依赖
# RUN 命令可以有多个，但是可以用 && 连接多个命令来减少层级。
# 例如 RUN npm install && cd /app && mkdir logs
RUN npm install --registry=https://registry.npm.taobao.org

# CMD 指令只能一个，是容器启动后执行的命令，算是程序的入口。
# 如果还需要运行其他命令可以用 && 连接，也可以写成一个shell脚本去执行。
# 例如 CMD cd /app && ./start.sh
CMD node app.js
```

- Build 为镜像（安装包） & 运行
 - 编译 `docker build -t test:v1 .`
 - `-t` 设置镜像名字和版本号
 - 运行 `docker run -p 8080:8080 --name test-hello test:v1`
 - `-p` 映射容器内端口到宿主机
 - `--name` 容器名字
 - `-d` 后台运行
- 发布
 - 镜像仓库用来存储我们 build 出来的“安装包”，我们可以把自己 build 出来的镜像上传到 docker 提供的镜像库中，方便传播。

■ 创建一个镜像库



■ 命令行登录账号:

```
docker login -u username
```

■ 新建一个tag, 名字必须跟你注册账号一样

```
docker tag test:v1 username/test:v1
```

■ 推上去

```
docker push username/test:v1
```

■ 部署试下

```
docker run -dp 8080:8080 username/test:v1
```

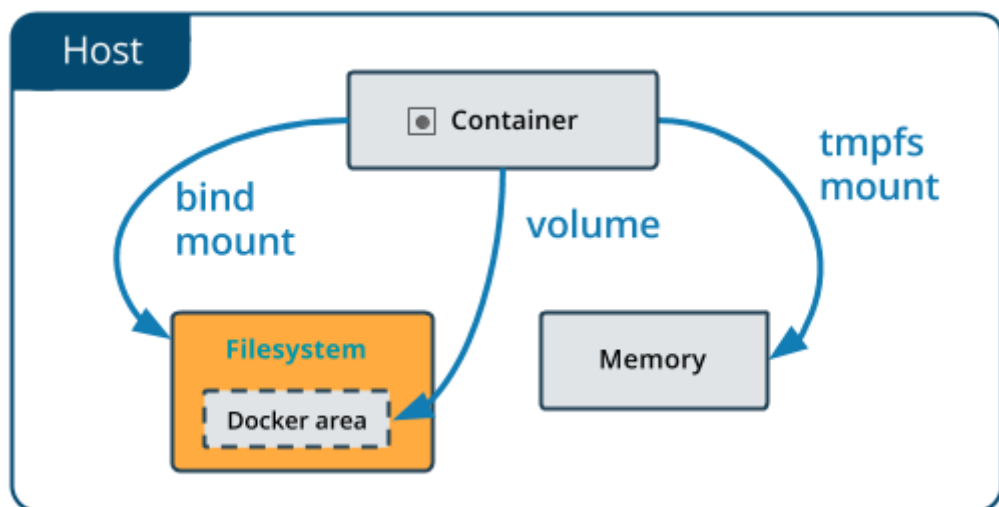
• 目录挂载:

◦ 问题:

- 使用 Docker 运行后, 我们改了项目代码不会立刻生效, 需要重新 `build` 和 `run`, 很是麻烦。
- 容器里面产生的数据, 例如 log 文件, 数据库备份文件, 容器删除后就丢失了。

◦ 常用方法:

- `bind mount` 直接把宿主机目录映射到容器内, 适合挂代码目录和配置文件。可挂到多个容器上
- `volume` 由容器创建和管理, 创建在宿主机, 所以删除容器不会丢失, 官方推荐, 更高效, Linux 文件系统, 适合存储数据库数据。可挂到多个容器上



• 多容器通信:

- 项目往往都不是独立运行的, 需要数据库、缓存这些东西配合运作。要想多容器之间互通, 从 Web 容器访问 Redis 容器, 我们只需要把他们放到同个网络中就可以了。

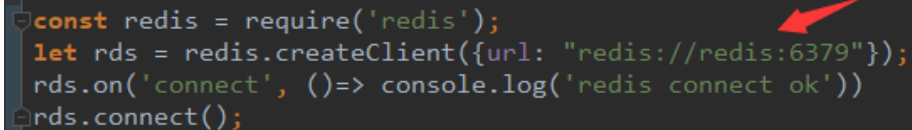
- 创建一个名为 `test-net` 的网络：

```
docker network create test-net
```

运行 Redis 在 `test-net` 网络中，别名 `redis`

```
docker run -d --name redis --network test-net --network-alias redis
redis:latest
```

修改代码中访问 `redis` 的地址为网络别名



```
const redis = require('redis');
let rds = redis.createClient({url: 'redis://redis:6379'});
rds.on('connect', ()=> console.log('redis connect ok'))
rds.connect();
```

运行 Web 项目，使用同个网络

```
docker run -p 8080:8080 --name test -v D:/test:/app --network test-net -
d test:v1
```

查看数据

`http://localhost:8080/redis`

容器终端查看数据是否一致

- Docker-Compose:
 - 如果项目依赖更多的第三方软件，我们需要管理的容器就更加多，每个都要单独配置运行，指定网络。故我们使用 `docker-compose` 把项目的多个服务集合到一起，一键运行。
 - 要把项目依赖的多个服务集合到一起，我们需要编写一个 `docker-compose.yml` 文件，描述依赖哪些服务

```
version: "3.7"

services:
  app:
    build: ./
    ports:
      - 80:8080
    volumes:
      - ./:/app
    environment:
      - TZ=Asia/Shanghai
  redis:
    image: redis:5.0.13
    volumes:
      - redis:/data
    environment:
      - TZ=Asia/Shanghai

volumes:
  redis:
```

在 `docker-compose.yml` 文件所在目录，执行：`docker-compose up` 就可以跑起来了。

安装docker（无需多言）

安装两个容器（redis【见上】、wordpress【docker-compose方式】）

wordpress需要依赖数据库，所以使用docker-compose方式进行安装。

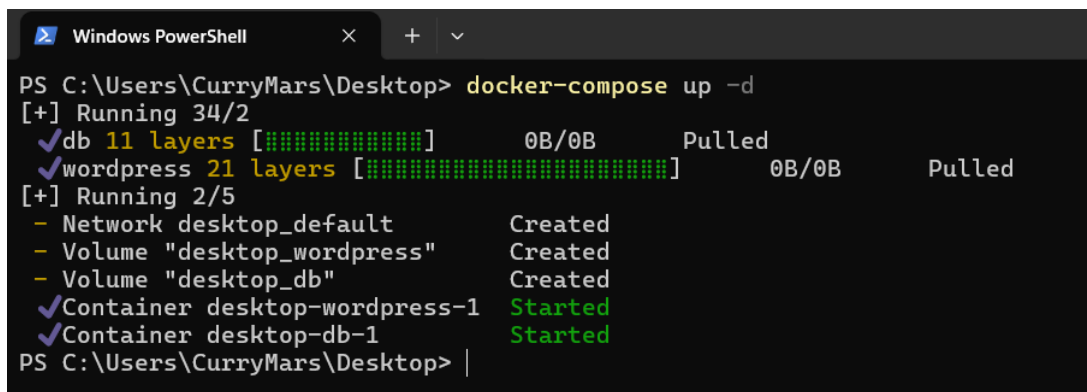
```
version: '3.1'

services:

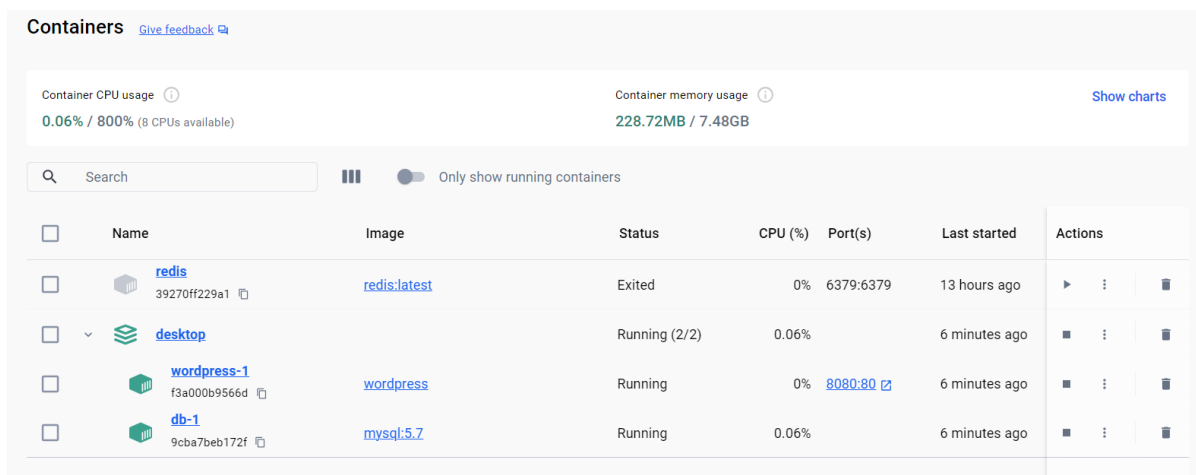
  wordpress:
    image: wordpress
    restart: always
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: exampleuser
      WORDPRESS_DB_PASSWORD: examplepass
      WORDPRESS_DB_NAME: exampledb
    volumes:
      - wordpress:/var/www/html

  db:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_DATABASE: exampledb
      MYSQL_USER: exampleuser
      MYSQL_PASSWORD: examplepass
      MYSQL_RANDOM_ROOT_PASSWORD: '1'
    volumes:
      - db:/var/lib/mysql

volumes:
  wordpress:
  db:
```



```
PS C:\Users\CurryMars\Desktop> docker-compose up -d
[+] Running 34/2
✔db 11 layers [#####] 0B/0B Pulled
✔wordpress 21 layers [#####] 0B/0B Pulled
[+] Running 2/5
- Network desktop_default Created
- Volume "desktop_wordpress" Created
- Volume "desktop_db" Created
✔Container desktop-wordpress-1 Started
✔Container desktop-db-1 Started
PS C:\Users\CurryMars\Desktop> |
```

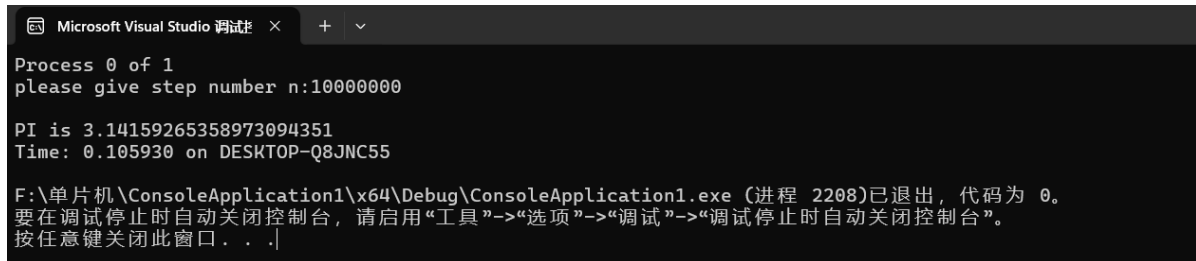



```

        printf("Time: %f on %s\n", stop - start, processor_name);
        fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}

```

不用容器得到的结果:



```

Microsoft Visual Studio 调试
Process 0 of 1
please give step number n:10000000

PI is 3.14159265358973094351
Time: 0.105930 on DESKTOP-Q8JNC55

F:\单片机\ConsoleApplication1\Debug\ConsoleApplication1.exe (进程 2208)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...

```

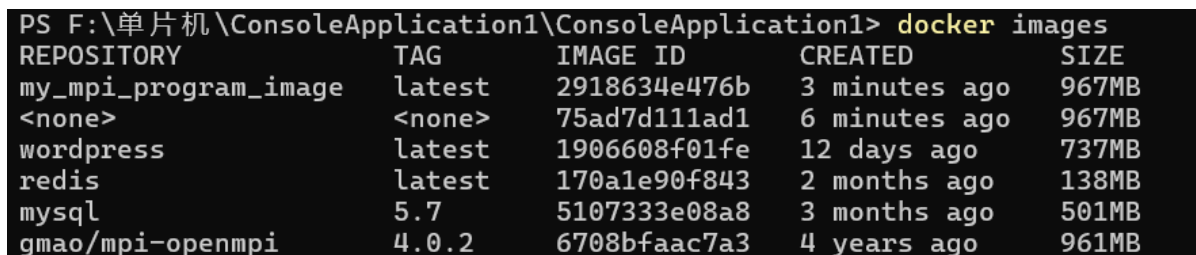
使用容器得到的结果:

- 先拉取mpi需要的镜像:

```

docker pull gmao/mpi-openmpi:4.0.2
docker images

```



```

PS F:\单片机\ConsoleApplication1\ConsoleApplication1> docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
my_mpi_program_image latest       2918634e476b     3 minutes ago   967MB
<none>              <none>      75ad7d111ad1     6 minutes ago   967MB
wordpress            latest       1906608f01fe     12 days ago     737MB
redis                latest       170a1e90f843     2 months ago    138MB
mysql                5.7         5107333e08a8     3 months ago    501MB
gmao/mpi-openmpi     4.0.2       6708bfaac7a3     4 years ago     961MB

```

- 在编写dockerfile文件

```

# 基于 gmao/mpi-openmpi:4.0.2 镜像构建
FROM gmao/mpi-openmpi:4.0.2

# 创建一个非 root 用户 mpiuser
RUN useradd -ms /bin/bash mpiuser

# 设置工作目录
WORKDIR /home/mpiuser

# 将当前目录下的所有文件复制到容器的 /home/mpiuser 目录中
COPY . /home/mpiuser

# 切换到 mpiuser 用户
USER mpiuser

# 编译 MPI 程序
RUN mpicc -o my_mpi_program ConsoleApplication1.cpp

# 设置执行命令
CMD ["mpiexec", "-n", "4", "./my_mpi_program"]

```

- 解释

- **基于现有镜像构建：**使用 `FROM` 指令基于 `gmoo/mpi-openmpi:4.0.2` 镜像构建你自己的镜像。这个基础镜像已经包含了 OpenMPI 4.0.2，因此你无需从零开始构建 MPI 环境。
- **创建非 root 用户：**使用 `RUN` 指令创建一个名为 `mpiuser` 的非 root 用户，并设置其登录 shell 为 `/bin/bash`。这样做是为了提高容器的安全性，避免以 root 用户的身份运行 MPI 程序带来的潜在风险。
- **设置工作目录：**使用 `WORKDIR` 指令设置容器中的工作目录为 `/home/mpiuser`。在这个目录下进行后续操作，保持整个过程的一致性。
- **复制 MPI 程序源代码：**使用 `COPY` 指令将当前目录下的所有文件复制到容器的 `/home/mpiuser` 目录中。这样就将 MPI 程序源代码复制到了容器中，以便后续编译和执行。
- **切换到非 root 用户：**使用 `USER` 指令切换到 `mpiuser` 用户。后续的编译和执行操作将以这个非 root 用户的身份进行，提高了容器的安全性。
- **编译 MPI 程序：**使用 `RUN` 指令在容器中使用 `mpicc` 编译 `ConsoleApplication1.cpp`，生成一个名为 `my_mpi_program` 的可执行文件。这个文件将用于后续的 MPI 执行。
- **设置执行命令：**使用 `CMD` 指令设置容器启动时要执行的默认命令。在这里，我们使用 `mpiexec` 在 4 个进程中运行 `my_mpi_program`，这样在容器启动时会自动执行 MPI 程序。

- 运行以下命令来构建 Docker 镜像：

```
docker build -t my_mpi_program_image .
```

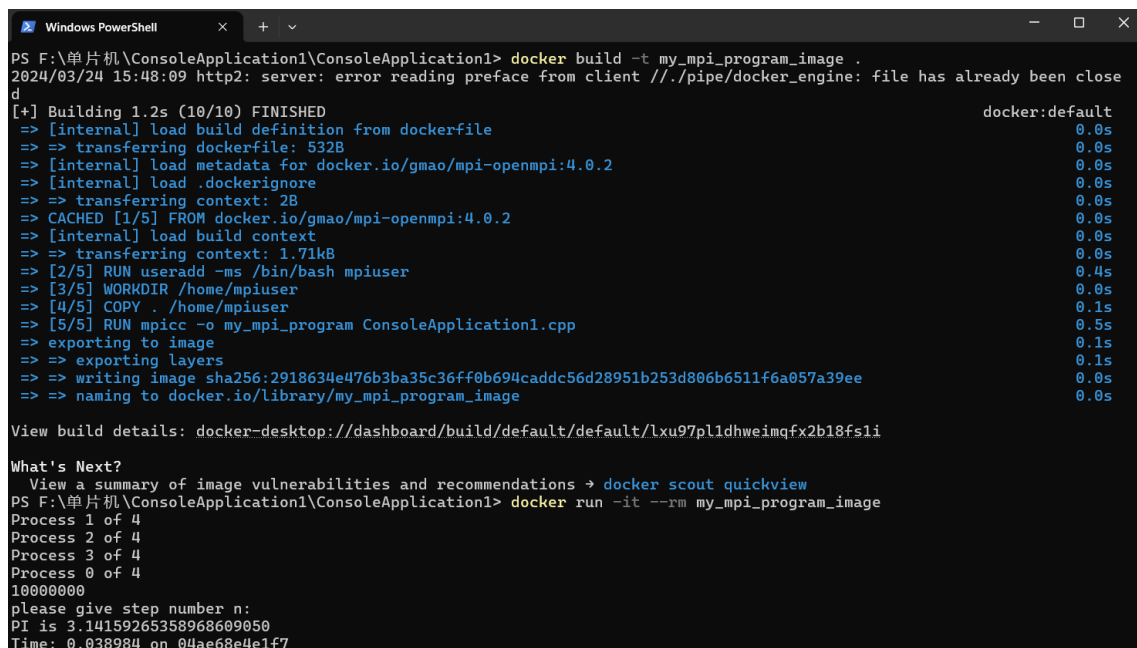
这将根据 Dockerfile 中的指令构建一个名为 `my_mpi_program_image` 的 Docker 镜像。

- 构建完成后，使用以下命令在 Docker 容器中运行 MPI 程序：

```
docker run -it --rm my_mpi_program_image
```

这将在 Docker 容器中以 4 个进程运行 MPI 程序，并输出计算出的 PI 值和执行时间。

- 运行结果：



```
PS F:\单片机\ConsoleApplication1\ConsoleApplication1> docker build -t my_mpi_program_image .
2024/03/24 15:48:09 http2: server: error reading preface from client //./pipe/docker_engine: file has already been close
d
[+] Building 1.2s (10/10) FINISHED                                                                                                     docker:default
=> [internal] load build definition from dockerfile                                         0.0s
=> => transferring dockerfile: 532B                                                         0.0s
=> [internal] load metadata for docker.io/gmoo/mpi-openmpi:4.0.2                         0.0s
=> [internal] load .dockerignore                                                            0.0s
=> => transferring context: 2B                                                              0.0s
=> CACHED [1/5] FROM docker.io/gmoo/mpi-openmpi:4.0.2                                    0.0s
=> [internal] load build context                                                            0.0s
=> => transferring context: 1.71kB                                                           0.0s
=> [2/5] RUN useradd -ms /bin/bash mpiuser                                                  0.4s
=> [3/5] WORKDIR /home/mpiuser                                                            0.0s
=> [4/5] COPY . /home/mpiuser                                                             0.1s
=> [5/5] RUN mpicc -o my_mpi_program ConsoleApplication1.cpp                             0.5s
=> exporting to image                                                                      0.1s
=> => exporting layers                                                                      0.1s
=> => writing image sha256:2918634e476b3ba35c36ff0b694caddc56d28951b253d806b6511f6a057a39ee 0.0s
=> => naming to docker.io/library/my_mpi_program_image                                     0.0s

View build details: docker-desktop://dashboard/build/default/default/1xu97pl1dhweimqfx2b18fs1i

What's Next?
View a summary of image vulnerabilities and recommendations -> docker scout quickview
PS F:\单片机\ConsoleApplication1\ConsoleApplication1> docker run -it --rm my_mpi_program_image
Process 1 of 4
Process 2 of 4
Process 3 of 4
Process 0 of 4
10000000
please give step number n:
PI is 3.14159265358968609050
Time: 0.038984 on 04ae68e4e1f7
```

ps：少使用几个进程

```
PS F:\单片机\ConsoleApplication1\ConsoleApplication1> docker build -t my_mpi_program_image .
[+] Building 0.6s (10/10) FINISHED
=> [internal] load build definition from dockerfile                                docker:default 0.0s
=> => transferring dockerfile: 532B                                              0.0s
=> [internal] load metadata for docker.io/gmiao/mpi-openmpi:4.0.2                0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 2B                                                  0.0s
=> [1/5] FROM docker.io/gmiao/mpi-openmpi:4.0.2                                0.0s
=> [internal] load build context                                                0.0s
=> => transferring context: 1.71kB                                              0.0s
=> CACHED [2/5] RUN useradd -ms /bin/bash mpiuser                               0.0s
=> CACHED [3/5] WORKDIR /home/mpiuser                                           0.0s
=> [4/5] COPY . /home/mpiuser                                                  0.1s
=> [5/5] RUN mpicc -o my_mpi_program ConsoleApplication1.cpp                   0.3s
=> exporting to image                                                            0.1s
=> => exporting layers                                                            0.1s
=> => writing image sha256:7d08a035dcb5a3f614f22846bd19ffff46d1c9e7062d1094d85167f55cc72584f 0.0s
=> => naming to docker.io/library/my_mpi_program_image                         0.0s

View build details: docker-desktop://dashboard/build/default/default/en4x9vyh01v2yff2hc0mxwrh

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
PS F:\单片机\ConsoleApplication1\ConsoleApplication1> docker run -it --rm my_mpi_program_image
Process 0 of 1
10000000
please give step number n:
PI is 3.14159265358973094351
Time: 0.060785 on ff148cec36ea
```

实验总结：

1. 容器与非容器相比：使用了容器技术后，mpi程序运行的速度变快了；经过思考，其中原因可能是：
 1. **资源隔离**：容器提供了资源隔离的环境，它们可以独立地管理 CPU、内存、网络等资源。因此，如果在本地环境中有其他正在运行的程序占用了系统资源，可能会影响到程序的性能。而在容器中，可以更好地控制和管理程序所使用的资源，从而提高程序的性能。
 2. **优化的镜像**：在构建容器镜像时，可以对程序及其依赖项进行优化，例如去除不必要的组件、减小镜像大小等。这样可以减少资源消耗，并提高程序的运行效率。
2. 进程数的多少决定了程序执行的效率。