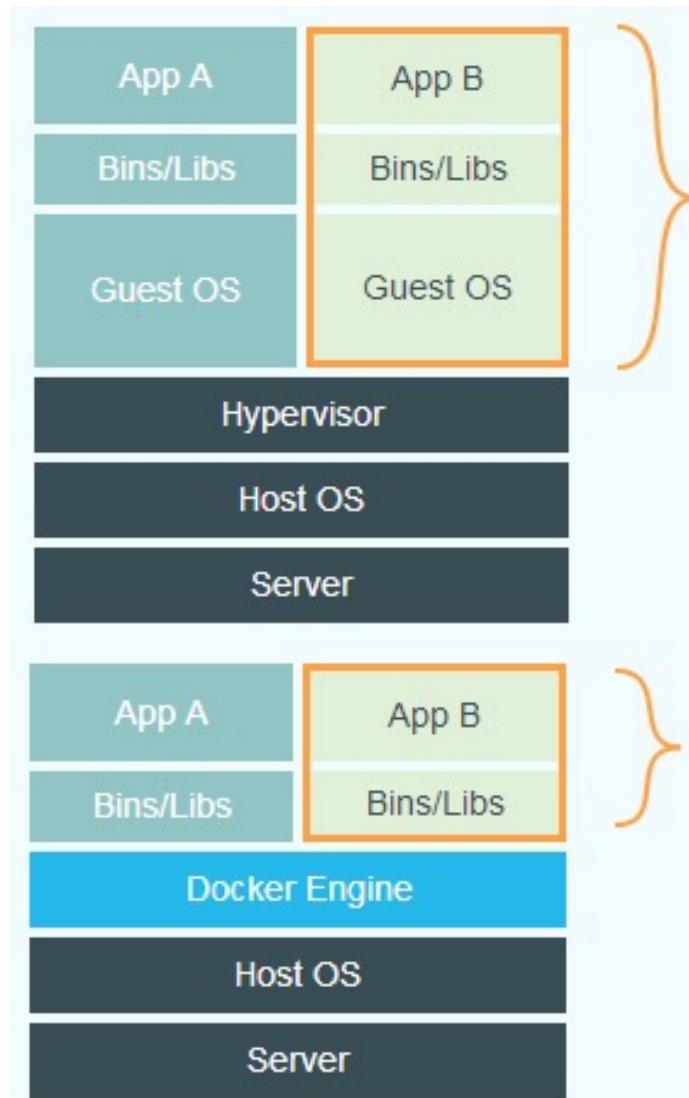


Docker

什么是Docker

- Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从 Apache2.0 协议开源。
- Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。
- 容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app ）,更重要的是容器性能开销极低。

Docker 架构



Virtual Machines

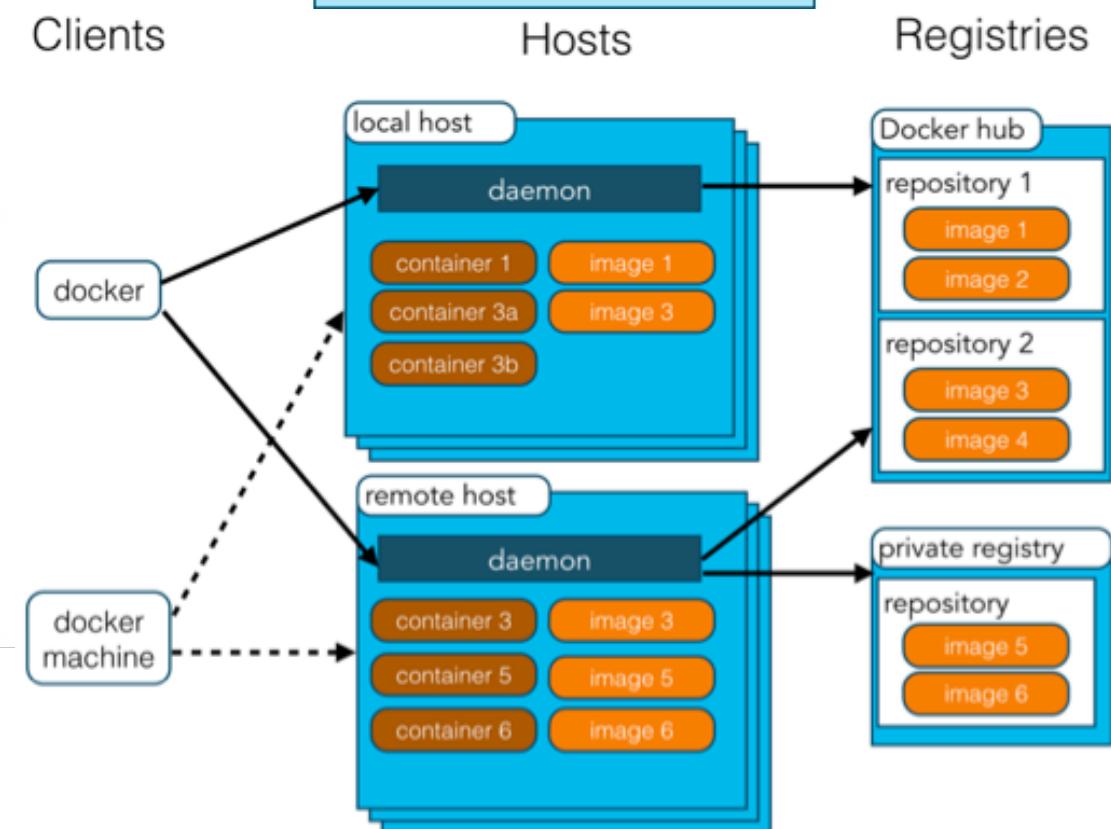
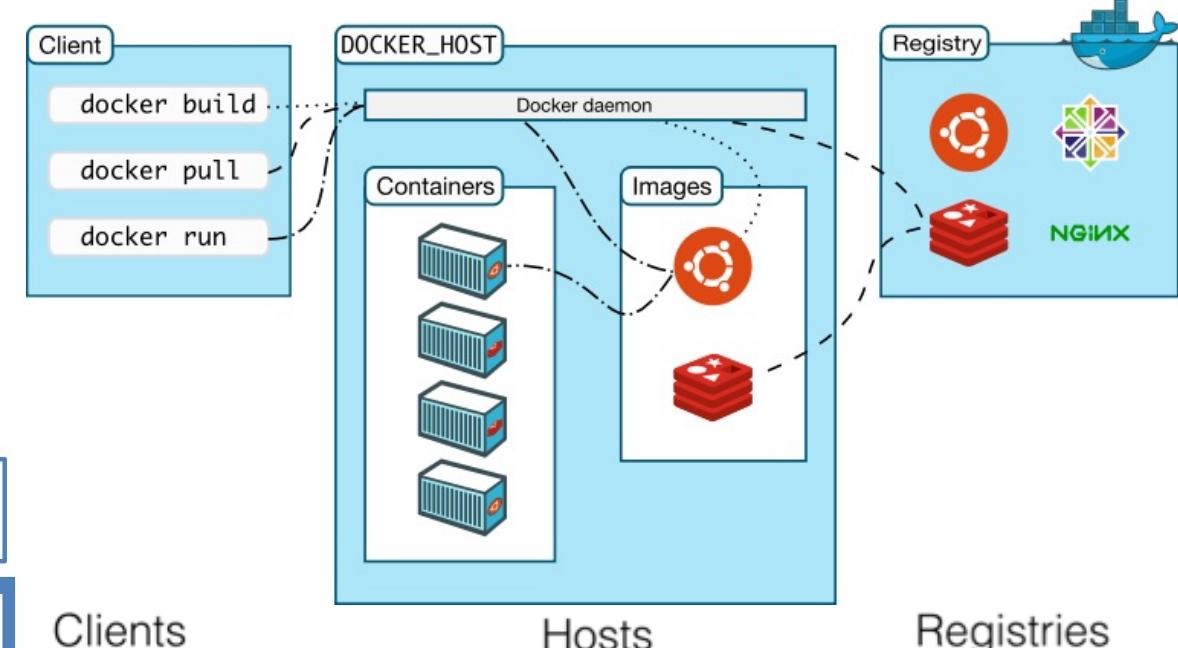
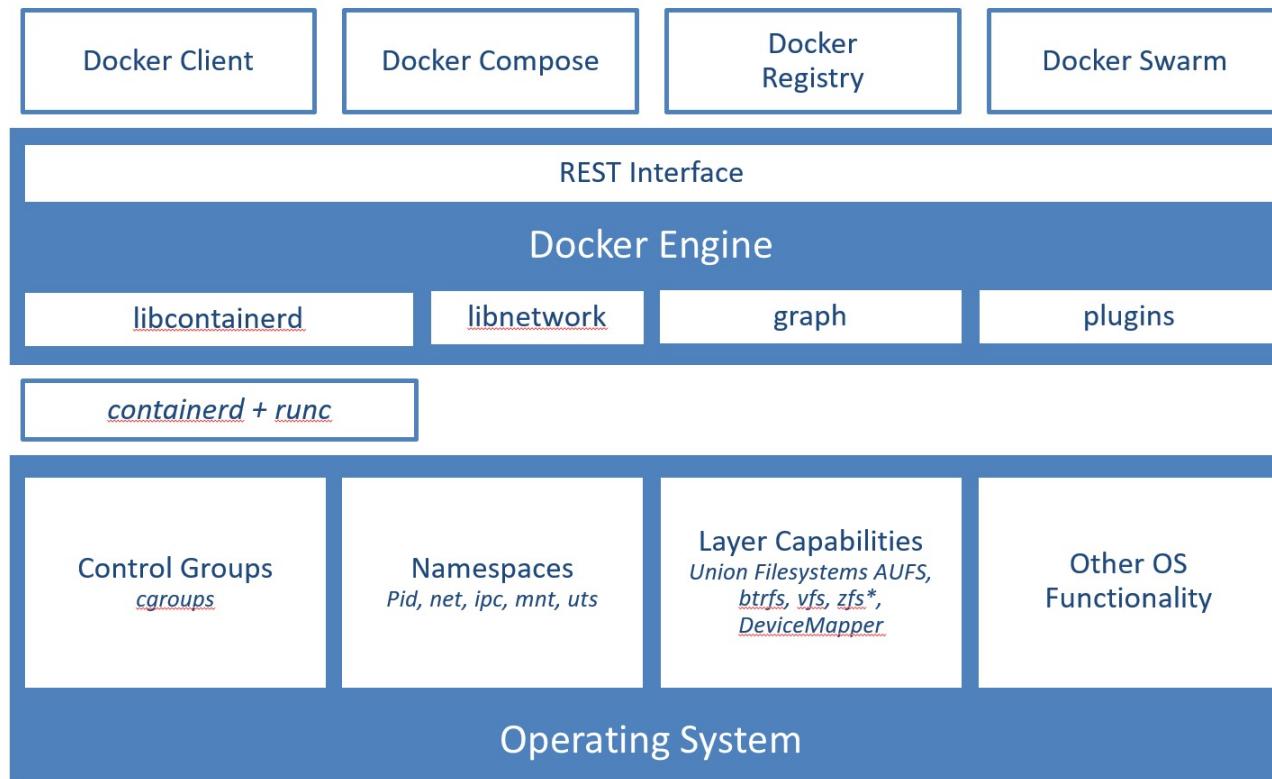
Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.

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.

Docker 架构

Architecture In Linux



Docker优势

- Docker 是一个用于开发，交付和运行应用程序的开放平台。Docker 使您能够将应用程序与基础架构分开，从而可以快速交付软件。借助 Docker，您可以与管理应用程序相同的方式来管理基础架构。通过利用 Docker 的方法来快速交付，测试和部署代码，您可以大大减少编写代码和在生产环境中运行代码之间的延迟。
- 更高效的利用系统资源
- 更快速的启动时间
- 一致的运行环境
- 持续交付和部署
- 更轻松的迁移
- 更轻松的维护和扩展

Docker 三个基本概念

- 镜像(Image)
 - 就相当于是个 root 文件系统。比如官方镜像 `ubuntu:16.04` 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统。
- 容器(Container)
 - 镜像 (Image) 和容器 (Container) 的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。
- 仓库(Repository)
 - 仓库可看成一个代码控制中心，用来保存镜像。Docker 使用客户端-服务器 (C/S) 架构模式，使用远程API来管理和创建Docker容器。

Docker command

- docker run ubuntu:15.10 /bin/echo "Hello world" -i -t -d --name -P -p
- docker pull ubuntu
- docker ps -a
- docker logs
- docker stop/start/restart
- docker exec -it 243c32535da7 /bin/bash (\Rightarrow virtual fs in container)
- docker export/import
- docker kill
- docker rm/rmi
- docker tag

构建镜像

- Dockerfile 是一个用来构建镜像的文本文件，文本内容包含了一条条构建镜像所需的指令和说明。
- Dockerfile 的指令每执行一次都会在 docker 上新建一层。所以过多无意义的层，会造成镜像膨胀过大。
- `docker build -t my-app:1.0 .`
- 上下文路径: 是指 docker 在构建镜像，有时候想要使用到本机的文件（比如复制），`docker build` 命令得知这个路径后，会将路径下的所有内容打包。
- 由于 docker 的运行模式是 C/S。我们本机是 C，docker 引擎是 S。实际的构建过程是在 docker 引擎下完成的，所以这个时候无法用到我们本机的文件。这就需要把我们本机的指定目录下的文件一起打包提供给 docker 引擎使用。
- 如果未说明最后一个参数，那么默认上下文路径就是 Dockerfile 所在的位置。

Dockerfile

Dockerfile

INSTRUCTION

ARGUMENT

Dockerfile

FROM Ubuntu

RUN apt-get update

RUN apt-get install python

RUN pip install flask

RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run

Start from a base OS or another image

Install all dependencies

Copy source code

Specify Entrypoint



SUBSCRIBE

Dockerfile

- FROM : 定制的镜像都是基于 FROM 的镜像，后续的操作都是基于此镜像。
- RUN : 用于执行后面跟着的命令行命令。 (docker build时运行)
- COPY: 复制指令，从上下文目录中复制文件或者目录到容器里指定路径。
- CMD: 为启动的容器指定默认要运行的程序，程序运行结束，容器也就结束。可被 docker run 命令行参数中指定要运行的程序所覆盖。 (docker run 时运行)
- ENTRYPOINT: 类似CMD，不会被 docker run 的命令行参数指定的指令所覆盖，而且这些命令行参数会被当作参数送给 ENTRYPOINT 指令指定的程序。
- (docker run 时使用了 --entrypoint 选项，将覆盖 CMD 指令指定的程序。在执行 docker run 的时候可以指定 ENTRYPOINT 运行所需的参数)

Dockerfile

- ENV: 设置环境变量，定义了环境变量，那么在后续的指令中，就可以使用这个环境变量。
- VOLUME: 定义匿名数据卷。在启动容器时忘记挂载数据卷，会自动挂载到匿名卷。
- EXPOSE: 声明端口。
- WORKDIR: 指定工作目录。用 WORKDIR 指定的工作目录（必须是提前创建好的），会在构建镜像的每一层中都存在。docker build 构建镜像过程中的，每一个 RUN 命令都是新建的一层。只有通过 WORKDIR 创建的目录才会一直存在。

Dockerfile

Dockerfile

INSTRUCTION

ARGUMENT

Dockerfile

FROM Ubuntu

RUN apt-get update

RUN apt-get install python

RUN pip install flask

RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run

Start from a base OS or another image

Install all dependencies

Copy source code

Specify Entrypoint



SUBSCRIBE

Layered architecture

Dockerfile

```
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask
run
```

```
docker build Dockerfile -t mmumshad/my-custom-app
```

Dockerfile2

```
FROM Ubuntu

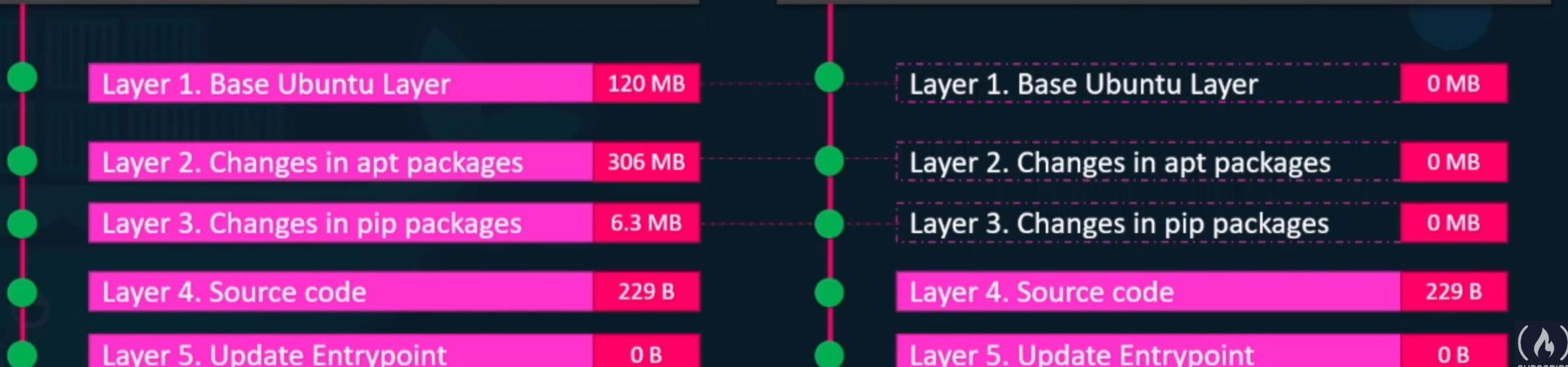
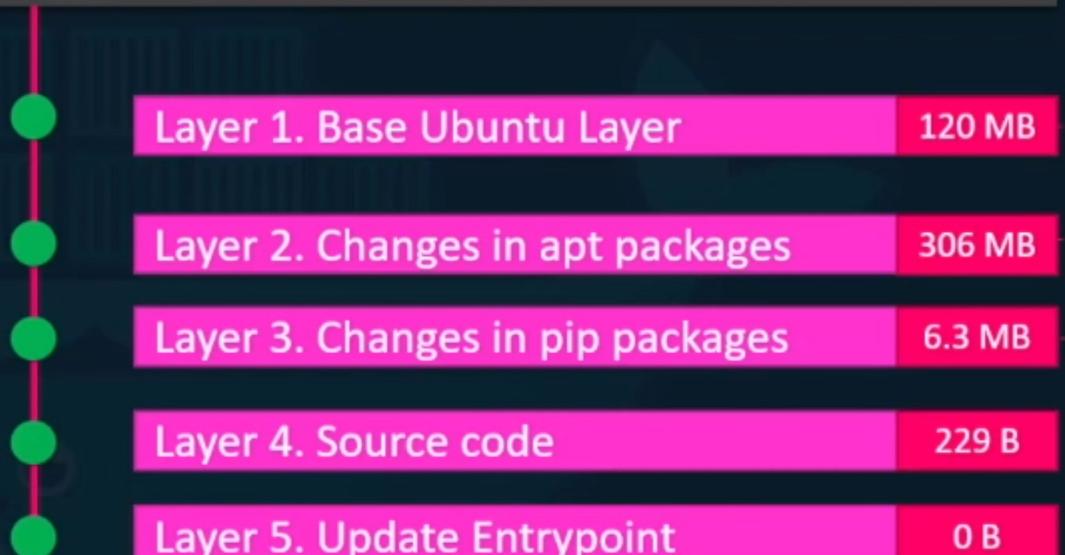
RUN apt-get update && apt-get -y install python

RUN pip install flask flask-mysql

COPY app2.py /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app2.py flask
run
```

```
docker build Dockerfile2 -t mmumshad/my-custom-app-2
```



Layered architecture

Container Layer

Read Write

Layer 6. Container Layer

```
docker run mmumshad/my-custom-app
```

Image Layers

Read Only

Layer 5. Update Entrypoint with “flask” command

Layer 4. Source code

Layer 3. Changes in pip packages

Layer 2. Changes in apt packages

Layer 1. Base Ubuntu Layer

```
docker build Dockerfile -t mmumshad/my-custom-app
```

app.py — demo

EXPLORER ... requirements.txt app.py Dockerfile

DEMOP app.py > ...

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def hello_world():
6     return 'Hello, Docker!'
7
8 if __name__ == '__main__':
9     app.run(debug=True, host='0.0.0.0')
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

duanruxiao@duanruxiaodeMacBook-Pro demo %

zsh + ^ x

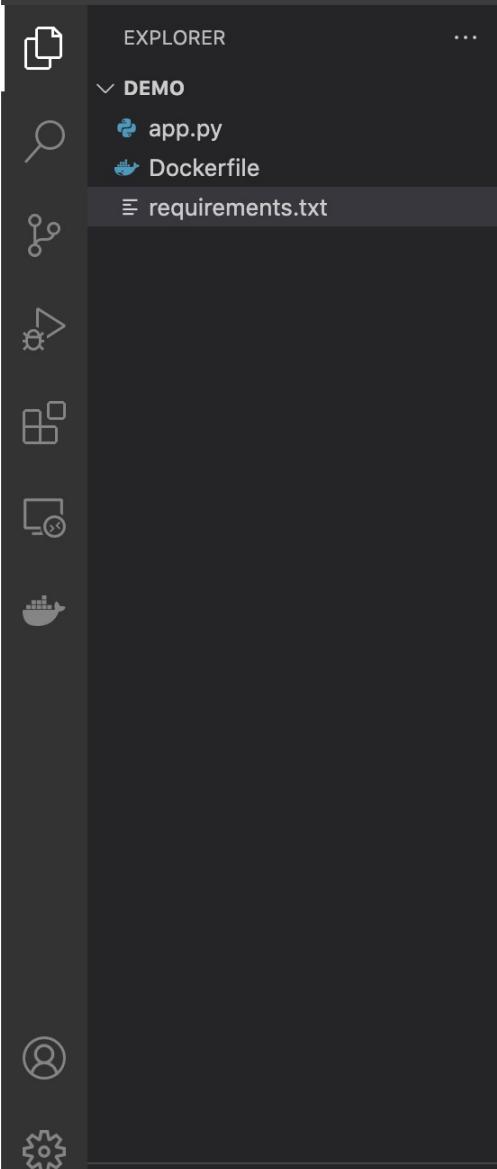
> OUTLINE

Python 3.8.2 64-bit ⊗ 0 △ 0

Ln 1, Col 1 Spaces: 4 UTF-8 LF Python ⚙ 🔍



requirements.txt — demo



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

duanruxiao@duanruxiaodeMacBook-Pro demo %

zsh + ^ x

> Python 3.8.2 64-bit

Ln 1, Col 1 Spaces: 4 UTF-8 LF pip requirements



Dockerfile — demo

EXPLORER ...

DEMO

- app.py
- Dockerfile

requirements.txt

```
FROM python:3.8
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
ENTRYPOINT ["python3"]
CMD ["app.py"]
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

duanrxiao@duanrxiaodeMacBook-Pro demo % docker build -t my-app .

```
[+] Building 11.8s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 156B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/python:3.8
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load build context
=> => transferring context: 303B
=> [1/4] FROM docker.io/library/python:3.8@sha256:739a62ba65b7dd500f3290422826231d37a8a9a95c15d38ca3ae28633d9cf13c
=> CACHED [2/4] WORKDIR /app
=> [3/4] COPY .
=> [4/4] RUN pip install -r requirements.txt
```

zsh + ^ ×

> OUTLINE

Python 3.8.2 64-bit 0 ▲ 0

Ln 8, Col 15 Spaces: 4 UTF-8 LF Dockerfile

The screenshot shows a dark-themed version of the Visual Studio Code interface. On the left is the Explorer sidebar with a tree view of a 'DEMO' folder containing 'app.py', 'Dockerfile', and 'requirements.txt'. The main editor area displays a Dockerfile with the following content:

```
FROM python:3.8
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
ENTRYPOINT ["python3"]
CMD ["app.py"]
```

Below the editor is a terminal window showing the output of a 'docker build' command:

```
duanrxiao@duanrxiaodeMacBook-Pro demo % docker build -t my-app .
[+] Building 11.8s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 156B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/python:3.8
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load build context
=> => transferring context: 303B
=> [1/4] FROM docker.io/library/python:3.8@sha256:739a62ba65b7dd500f3290422826231d37a8a9a95c15d38ca3ae28633d9cf13c
=> CACHED [2/4] WORKDIR /app
=> [3/4] COPY .
=> [4/4] RUN pip install -r requirements.txt
```

The status bar at the bottom indicates the environment is 'Python 3.8.2 64-bit' and shows file statistics: 0 ▲ 0. The bottom right corner shows the current language is 'Dockerfile'.

Dockerfile — demo

EXPLORER ... requirements.txt app.py Dockerfile

Dockerfile > ...

```
1 FROM python:3.8
2
3 WORKDIR /app
4 COPY .
5
6 RUN pip install -r requirements.txt
7
8 ENTRYPOINT ["python3"]
9 CMD ["app.py"]
```

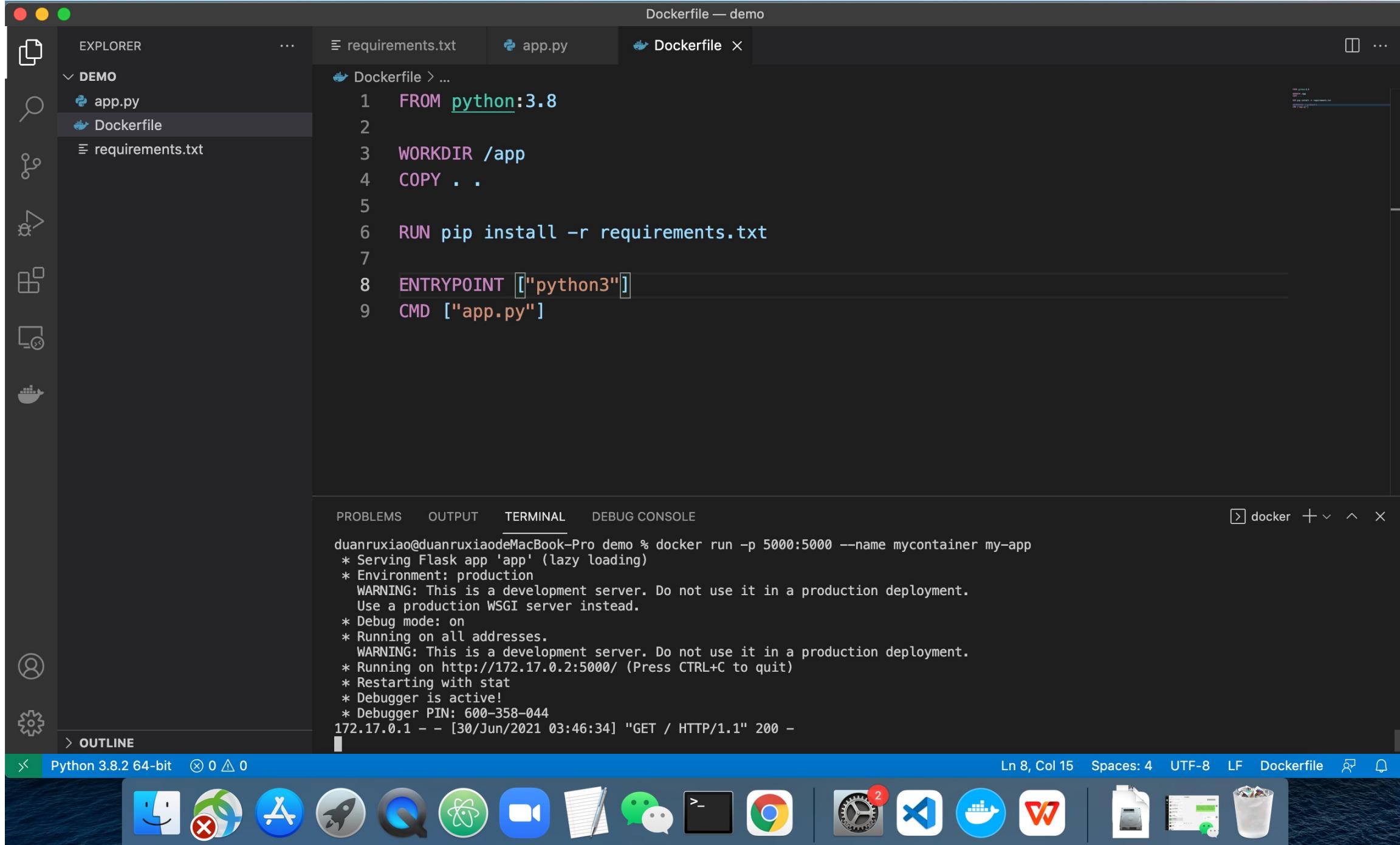
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

duanruxiao@duanruxiaodeMacBook-Pro demo % docker images

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|---------------|--------|--------------|--------------------|--------|
| my-app | latest | 809c9cfbde19 | About a minute ago | 894MB |
| mongo | latest | 0e120e3fce9a | 5 days ago | 449MB |
| mongo-express | latest | 30d9cf086bf8 | 11 days ago | 136MB |
| ubuntu | latest | 9873176a8ff5 | 12 days ago | 72.7MB |
| ubuntu | <none> | f63181f19b2f | 5 months ago | 72.9MB |

duanruxiao@duanruxiaodeMacBook-Pro demo %





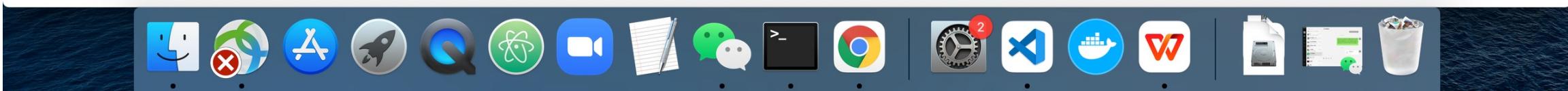
Chrome 文件 编辑 视图 历史记录 书签 用户 标签页 窗口 帮助

88% 周三上午11:46

localhost:5000

HKU Portal 中国移动香港 Gmail jpg2pdf Reimbursement Cl... fdpd hk minormath Online Syllabuses... Internship port mapping

Hello, Docker!



Docker File System

- `/var/lib/docker`
 - `/aufs`
 - `/containers`
 - `/image`
 - `/volumes`

AUFS

- 联合文件系统（ Union File System , Unionfs ）是一种分层的轻量级文件系统，它可以把多个目录内容联合挂载到同一目录下，从而形成一个单一的文件系统，这种特性可以让使用者像是使用一个目录一样使用联合文件系统。
- 对于Docker来说，联合文件系统可以说是其镜像和容器的基础。联文件系统可以使得Docker把镜像做成分层结构，从而使得镜像的每一层都可以被共享。从而节省大量的存储空间。

AUFS

- AUFS 是联合文件系统，意味着它在主机上使用多层目录存储，每一个目录在 AUFS 中都叫作分支，而在 Docker 中则称之为层（layer），但最终呈现给用户的是一个普通单层的文件系统，我们把多层以单一层的方式呈现出来的过程叫作联合挂载。
- 每一个镜像层和容器层都是 /var/lib/docker 下的一个子目录，镜像层和容器层都在 aufs/diff 目录下，每一层的目录名称是镜像或容器的 ID 值，联合挂载点在 aufs/mnt 目录下，mnt 目录是真正的容器工作目录。

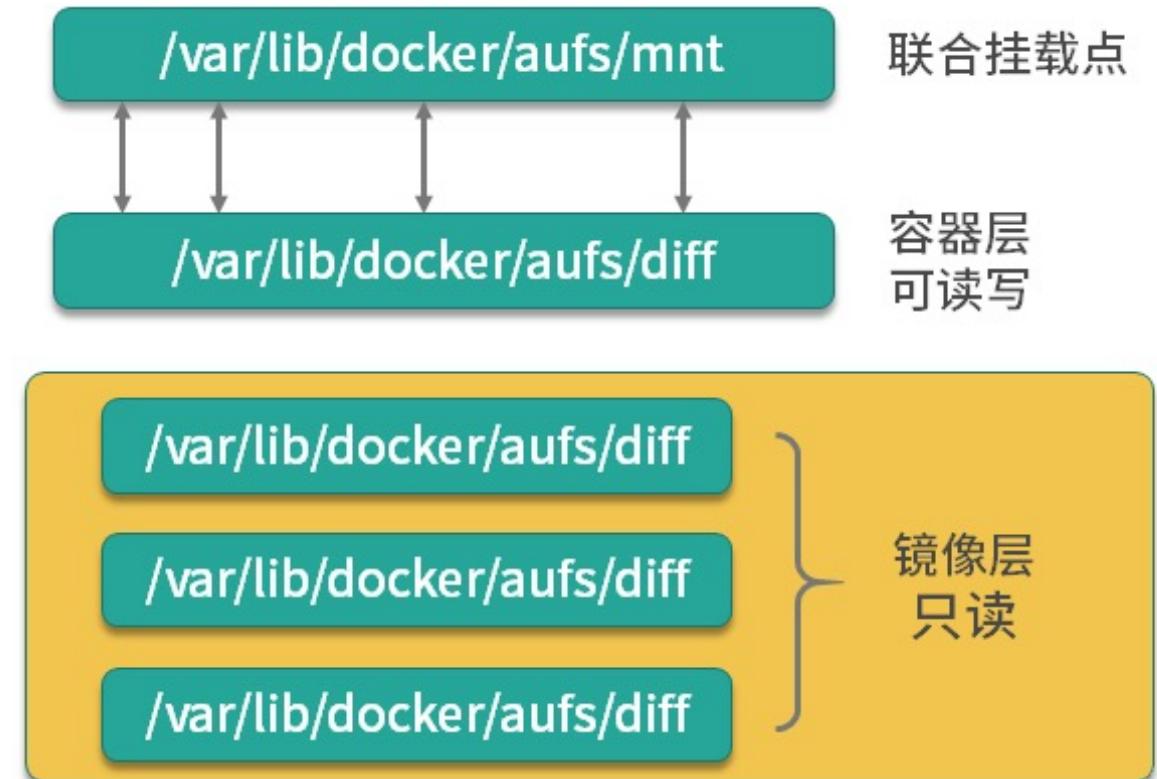


图 1 AUFS 工作原理示意图

创建容器过程中 aufs 文件夹的变化

- 当一个镜像未生成容器时：
 - diff文件夹：存储镜像内容，每一层都存储在镜像层ID命名的子文件夹中。
 - layers文件夹：存储镜像层关系的元数据，在diff文件夹下的每一个镜像层在这里都会有一个文件，文件的内容为该层镜像的父级镜像的ID
 - mnt文件夹：联合挂载点目录，未生成容器时，该目录为空
- 当一个镜像生成容器后，AUFS存储结构会发生如下变化：
 - diff文件夹：当容器运行时会在diff文件夹下生成容器层
 - layers文件夹：增加容器相关的元数据
 - mnt文件夹：容器的联合挂载点，这和容器中看到的文件内容一致

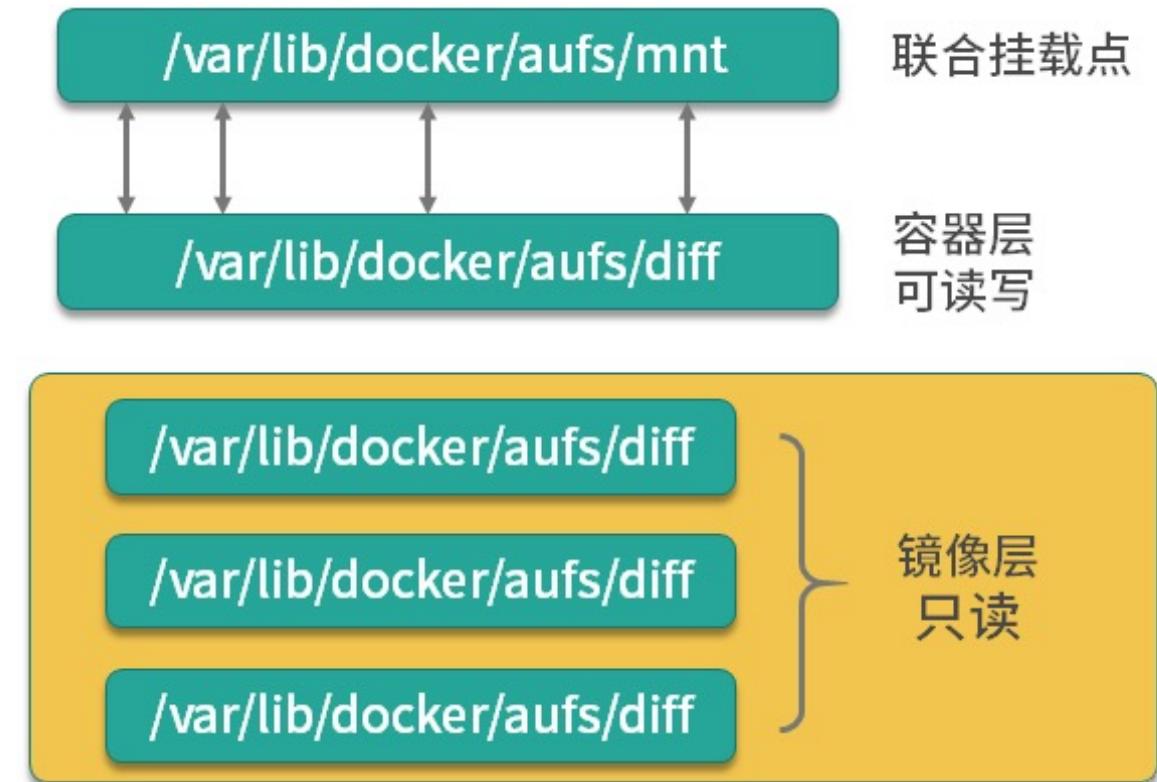


图 1 AUFS 工作原理示意图

AUFS 如何工作

- 读取文件：
 - 文件在容器层中存在时：直接从容器层读取。
 - 当文件在容器层中不存在时：从镜像层查找该文件，然后读取文件内容。
 - 文件既存在于镜像层，又存在于容器层：从容器层读取该文件。
- 修改文件或者目录：
 - 第一次修改文件：AUFS 会触发写时复制操作，AUFS 首先从镜像层复制文件到容器层，然后再执行对应的修改操作。
 - 删除文件或目录：AUFS 并不会真正从镜像中删除它，因为镜像层是只读的，AUFS 会创建一个特殊的文件或文件夹，这种特殊的文件或文件夹会阻止容器的访问

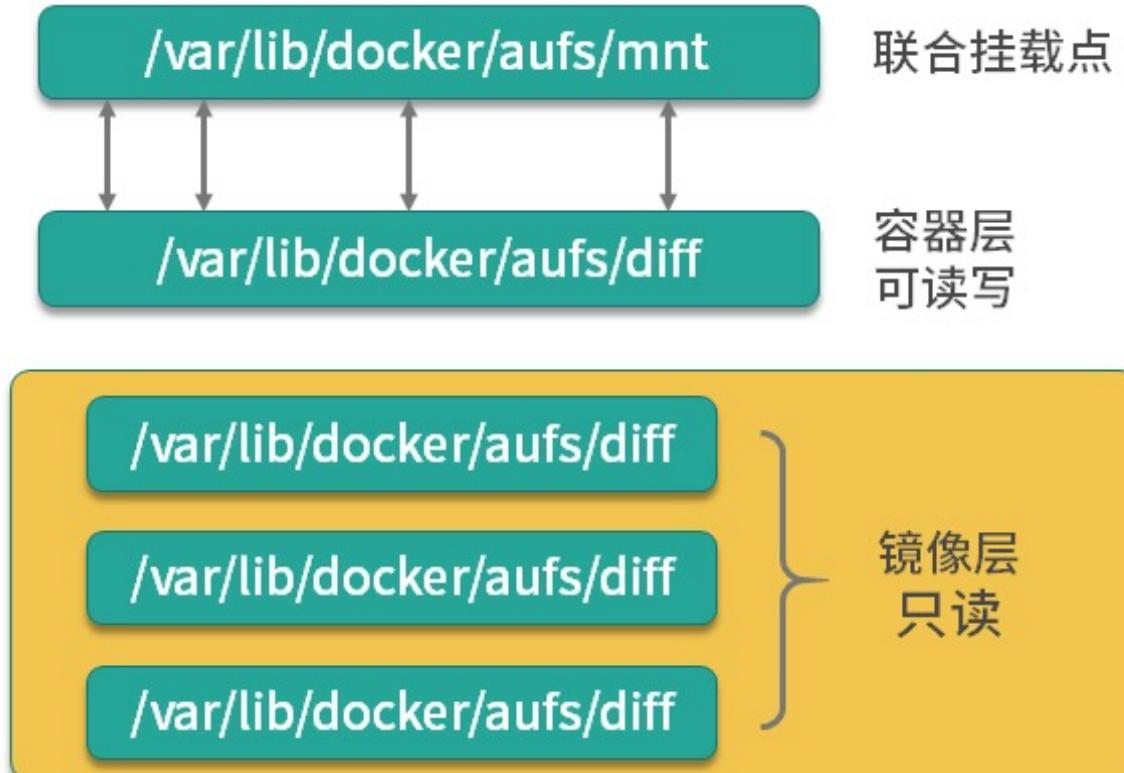


图 1 AUFS 工作原理示意图

Layered architecture

Container Layer

Read Write

Layer 6. Container Layer

```
docker run mmumshad/my-custom-app
```

Image Layers

Read Only

Layer 5. Update Entrypoint with “flask” command

Layer 4. Source code

Layer 3. Changes in pip packages

Layer 2. Changes in apt packages

Layer 1. Base Ubuntu Layer

```
docker build Dockerfile -t mmumshad/my-custom-app
```

容器数据管理-----数据卷（ Volumes ）

- 数据卷 是一个可供一个或多个容器使用的特殊目录，它可以提供很多有用的特性：
- 数据卷 可以在容器之间共享和重用
- 对 数据卷 的修改会立马生效
- 对 数据卷 的更新，不会影响镜像
- 数据卷 默认会一直存在，即使容器被删除

容器数据管理-----数据卷 (Volumes)

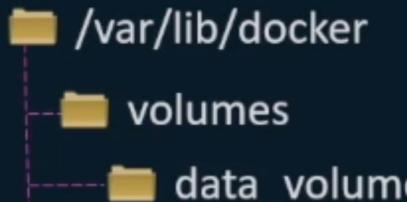
- docker volume create my-vol
- docker volume ls
- docker run -d -P \
• --name web \
• # -v my-vol:/usr/share/nginx/html \
• --mount source=my-vol, target=/usr/share/nginx/html \
• nginx:alpine
• docker volume rm my-vol

容器数据管理-----挂载主机目录 (Bind mounts)

- docker run -d -P \
- --name web \
- # -v /src/webapp:/usr/share/nginx/html \
- --mount **type=bind**, source=/src/webapp, target=/usr/share/nginx/html \
- nginx:alpine

volumes

```
docker volume create data_volume
```

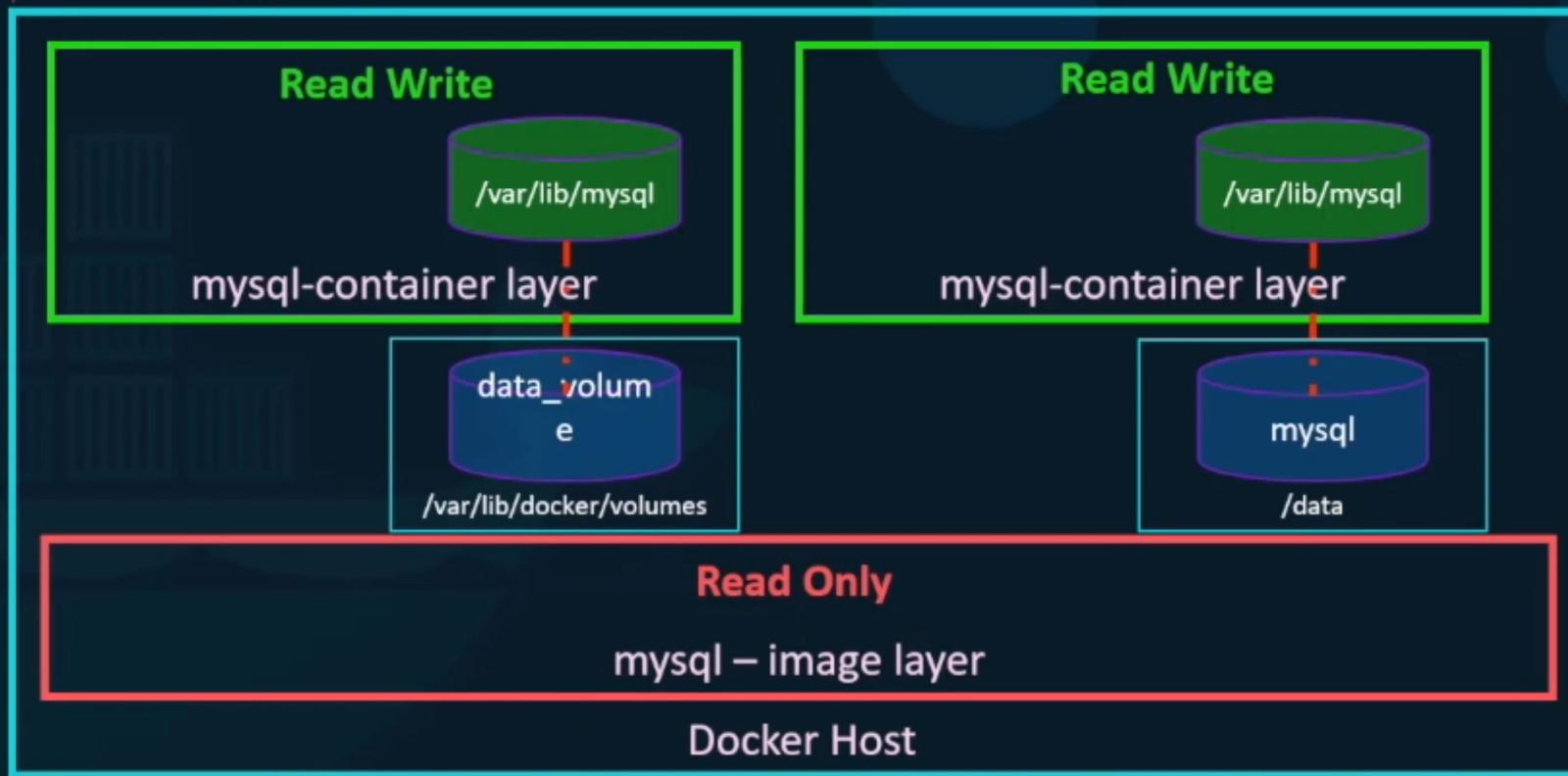


```
docker run -v data_volume:/var/lib/mysql mysql
```

```
docker run -v data_volume2:/var/lib/mysql mysql
```

```
docker run -v /data/mysql:/var/lib/mysql mysql
```

```
docker run \
--mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql
```



Docker 容器连接

- 网络端口映射
- docker run -d -P webapp python app.py 随机映射一个端口到内部容器开放的网络端口
- docker run -d -p 5000:5000 webapp python app.py 指定要映射的端口 (在一个指定端口上只可以绑定一个容器)
 - 格式 ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort
 - -p 127.0.0.1:80:80 指定映射使用一个特定地址，比如 localhost 地址 127.0.0.1
 - -p 127.0.0.1::80 绑定 localhost 的任意端口到容器的 80 端口，本地主机会自动分配一个端口
 - -p 80:80 本地的 80 端口映射到容器的 80 端口
 - -p 标记可以多次使用来绑定多个端口
- docker port 查看当前映射的端口配置
- docker inspect 查看容器自己的内部网络和 ip 地址

Run – PORT mapping

```
docker run kodekloud/webapp
```

```
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

http://172.17.0.2:5000

Internal IP

```
docker run -p 80:5000 kodekloud/simple-webapp
```

```
docker run -p 8000:5000 kodekloud/simple-webapp
```

```
docker run -p 8001:5000 kodekloud/simple-webapp
```

```
docker run -p 3306:3306 mysql
```

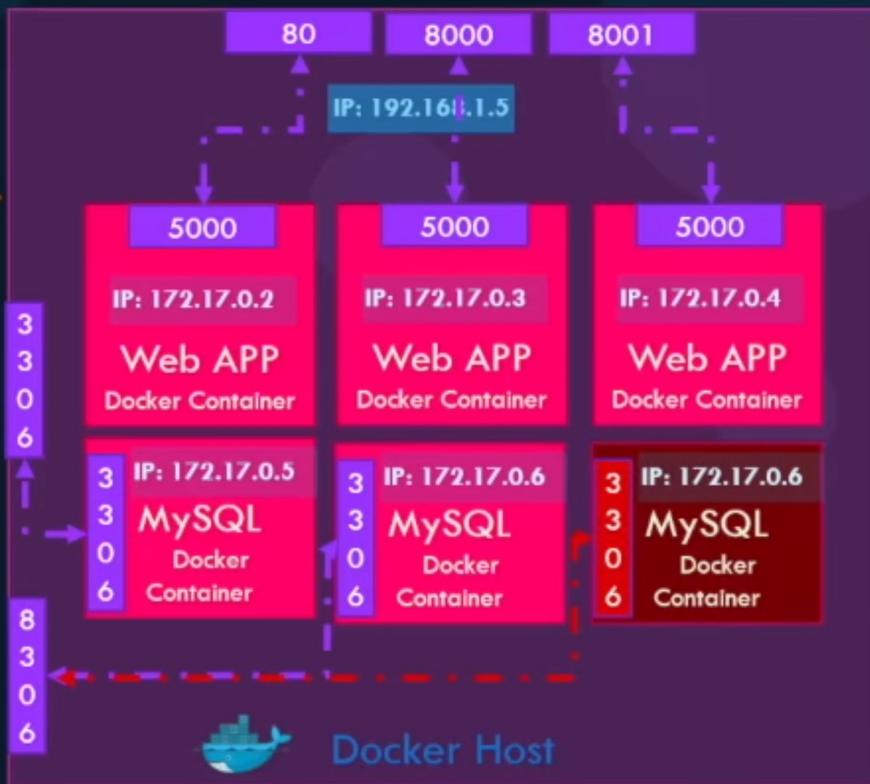
```
docker run -p 8306:3306 mysql
```

```
docker run -p 8306:3306 mysql
```

```
root@osboxes:/root # docker run -p 8306:3306 -e MYSQL_ROOT_PASSWORD=pass mysql
docker: Error response from daemon: driver failed programming external connectivity on endpoint boring_bhabha (5079d342b7e8ee11c71d46): Bind for 0.0.0.0:8306 failed: port is already allocated.
```



http://192.168.1.5:80



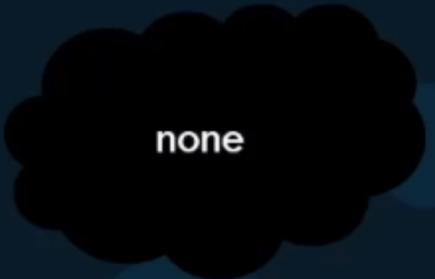
Docker 容器互联

- 新建网络
 - docker network create -d bridge test-net
- 连接容器
 - docker run -itd --name test1 **--network test-net** ubuntu /bin/bash
 - docker run -itd --name test2 **--network test-net** ubuntu /bin/bash
 - test1 容器和 test2 容器建立了互联关系。
 - 如果你有多个容器之间需要互相连接，推荐使用 Docker Compose

Default networks



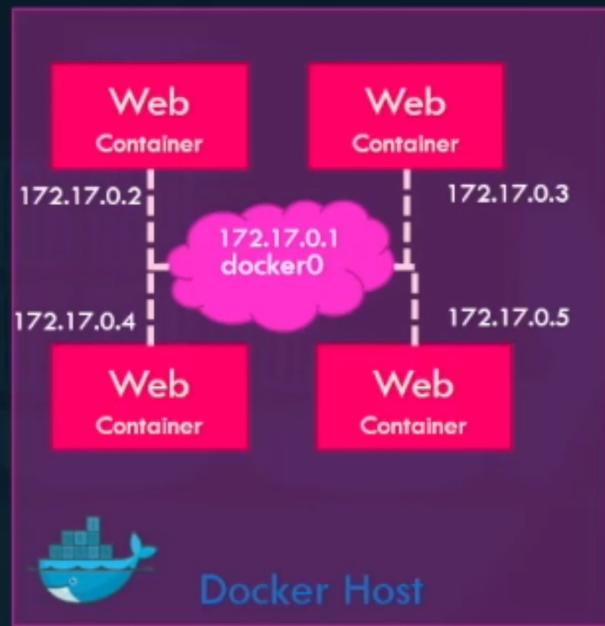
`docker run ubuntu`



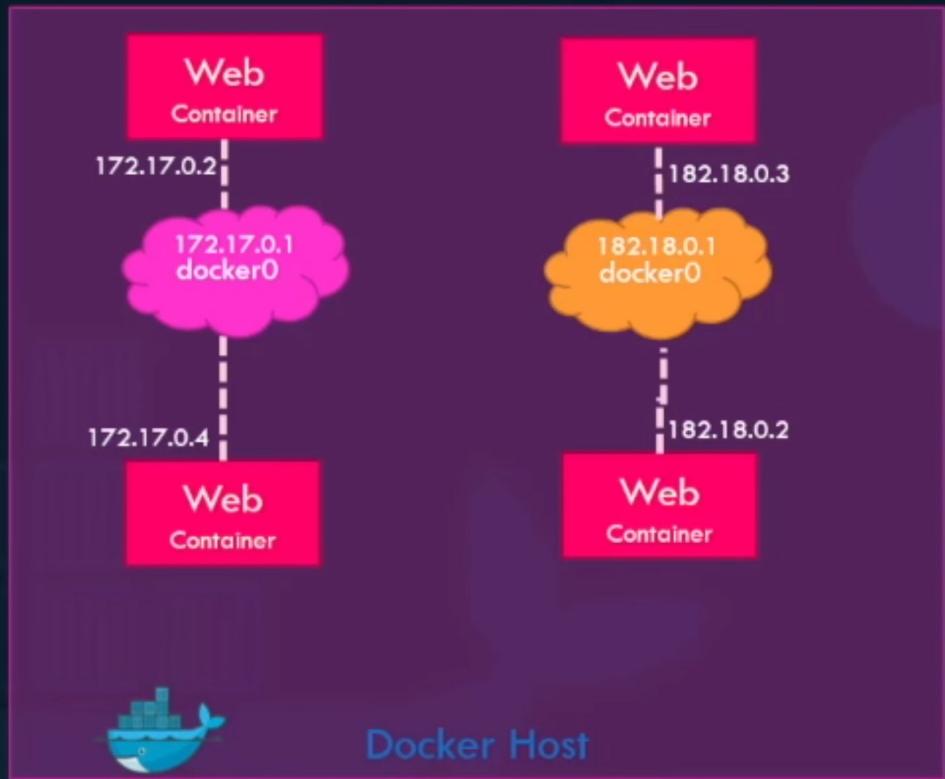
`docker run Ubuntu --network=none`



`docker run Ubuntu --network=host`



User-defined networks



```
docker network create \
--driver bridge \
--subnet 182.18.0.0/16
custom-isolated-network
```

Docker Compose

- Compose 是用于定义和运行多容器 Docker 应用程序的工具。通过 Compose，您可以使用 YML 文件来配置应用程序需要的所有服务。然后，使用一个命令，就可以从 YML 文件配置中创建并启动所有服务。
- 1. 使用 Dockerfile 定义应用程序的环境。
- 2. 使用 docker-compose.yml 定义构成应用程序的服务，这样它们可以在隔离环境中一起运行。
- 3. 最后执行 docker-compose up 命令来启动并运行整个应用程序。

Docker compose

```
docker run mmumshad/simple-webapp
```

```
docker run mongodb
```

```
docker run redis:alpine
```

```
docker run ansible
```

docker-compose.yml

```
services:  
  web:  
    image: "mmumshad/simple-webapp"  
  database:  
    image: "mongodb"  
  messaging:  
    image: "redis:alpine"  
  orchestration:  
    image: "ansible"
```

```
docker-compose up
```



Public Docker registry - dockerhub



docker run --links

```
docker run -d --name=redis redis
```

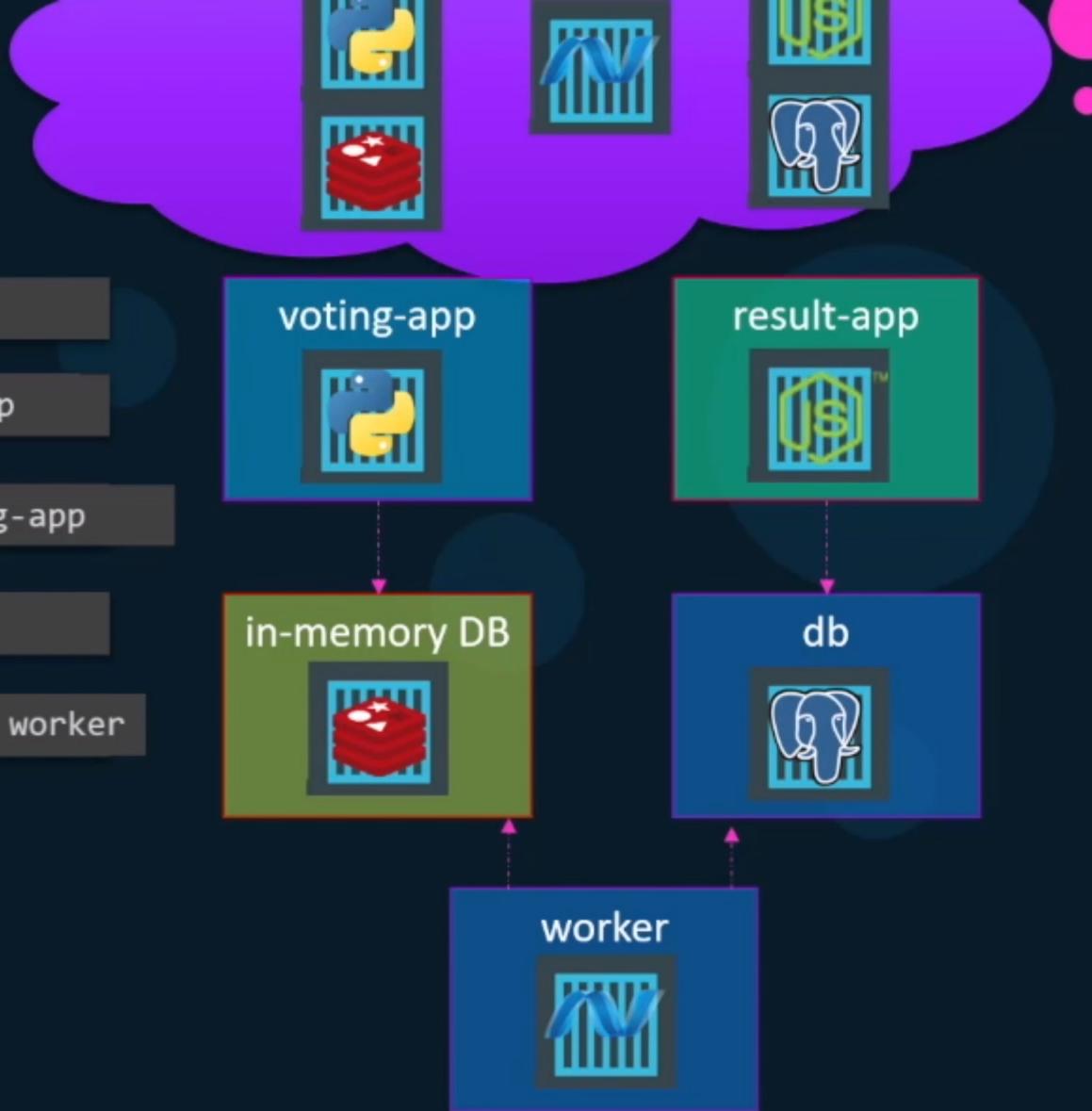
```
docker run -d --name=db postgres:9.4 --link db:db result-app
```

```
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
```

```
docker run -d --name=result -p 5001:80
```

```
docker run -d --name=worker --link db:db --link redis:redis worker
```

```
try {  
    Jedis redis = connectToRedis("redis");  
    Connection dbConn = connectToDB("db");  
  
    System.err.println("Watching vote queue");
```



Deprecation Warning



SUBSCRIBE

Docker compose

```
docker run -d --name=redis redis
```

```
docker run -d --name=db postgres:9.4
```

```
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
```

```
docker run -d --name=result -p 5001:80 --link db:db result-app
```

```
docker run -d --name=worker --link db:db --link redis:redis worker
```

db:db = db

docker-compose.yml

```
redis:  
  image: redis  
db:  
  image: postgres:9.4  
vote:  
  image: voting-app  
  ports:  
    - 5000:80  
  links:  
    - redis  
result:  
  image: result-app  
  ports:  
    - 5001:80  
  links:  
    - db  
worker:  
  image: worker  
  links:  
    - redis  
    - db
```

Docker compose - versions

docker-compose.yml

```
redis:  
    image: redis  
  
db:  
    image: postgres:9.4  
  
vote:  
    image: voting-app  
    ports:  
        - 5000:80  
    links:  
        - redis
```

version: 1

docker-compose.yml

```
version: 2  
services:  
    redis:  
        image: redis  
    db:  
        image: postgres:9.4  
    vote:  
        image: voting-app  
        ports:  
            - 5000:80  
        depends_on:  
            - redis
```

version: 2

docker-compose.yml

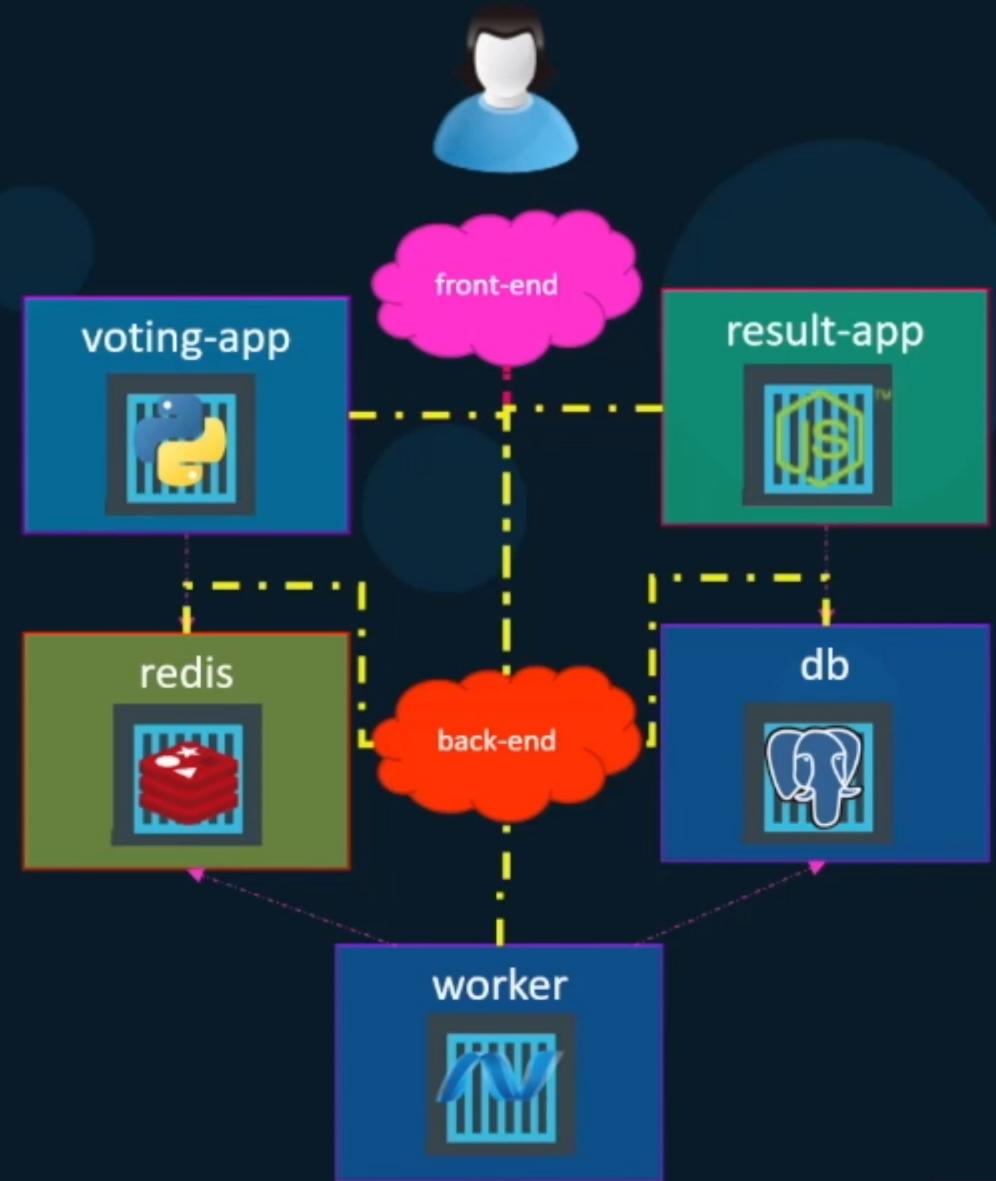
```
version: 3  
services:  
    redis:  
        image: redis  
    db:  
        image: postgres:9.4  
    vote:  
        image: voting-app  
        ports:  
            - 5000:80
```

version: 3

Docker compose

docker-compose.yml

```
version: 2
services:
    redis:
        image: redis
        networks:
            - back-end
    db:
        image: postgres:9.4
        networks:
            - back-end
    vote:
        image: voting-app
        networks:
            - front-end
            - back-end
    result:
        image: result
        networks:
            - front-end
            - back-end
networks:
    front-end:
    back-end:
```



Docker 仓库管理

- 仓库是集中存放镜像的地方。不止 docker hub，只是远程的服务商不一样，操作都是一样的。
- 目前 Docker 官方维护了一个公共仓库 Docker Hub。大部分需求都可以通过在 Docker Hub 中直接下载镜像来实现。
- docker login/logout
- docker search ubuntu
- docker pull ubuntu
- docker tag ubuntu:18.04 username/ubuntu:18.04
- docker push username/ubuntu:18.04

References

- <https://baike.baidu.com/item/Docker/13344470?fr=aladdin>
- [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software))
- <https://www.cnblogs.com/goWithHappy/p/thress-file-system-for-docker.html>
- <https://www.runoob.com/docker/docker-tutorial.html>
- <https://www.youtube.com/watch?v=3c-iBn73dDE&t=4668s>
- <https://www.youtube.com/watch?v=fqMOX6JJhGo&t=4628s>
- https://yeasy.gitbook.io/docker_practice/