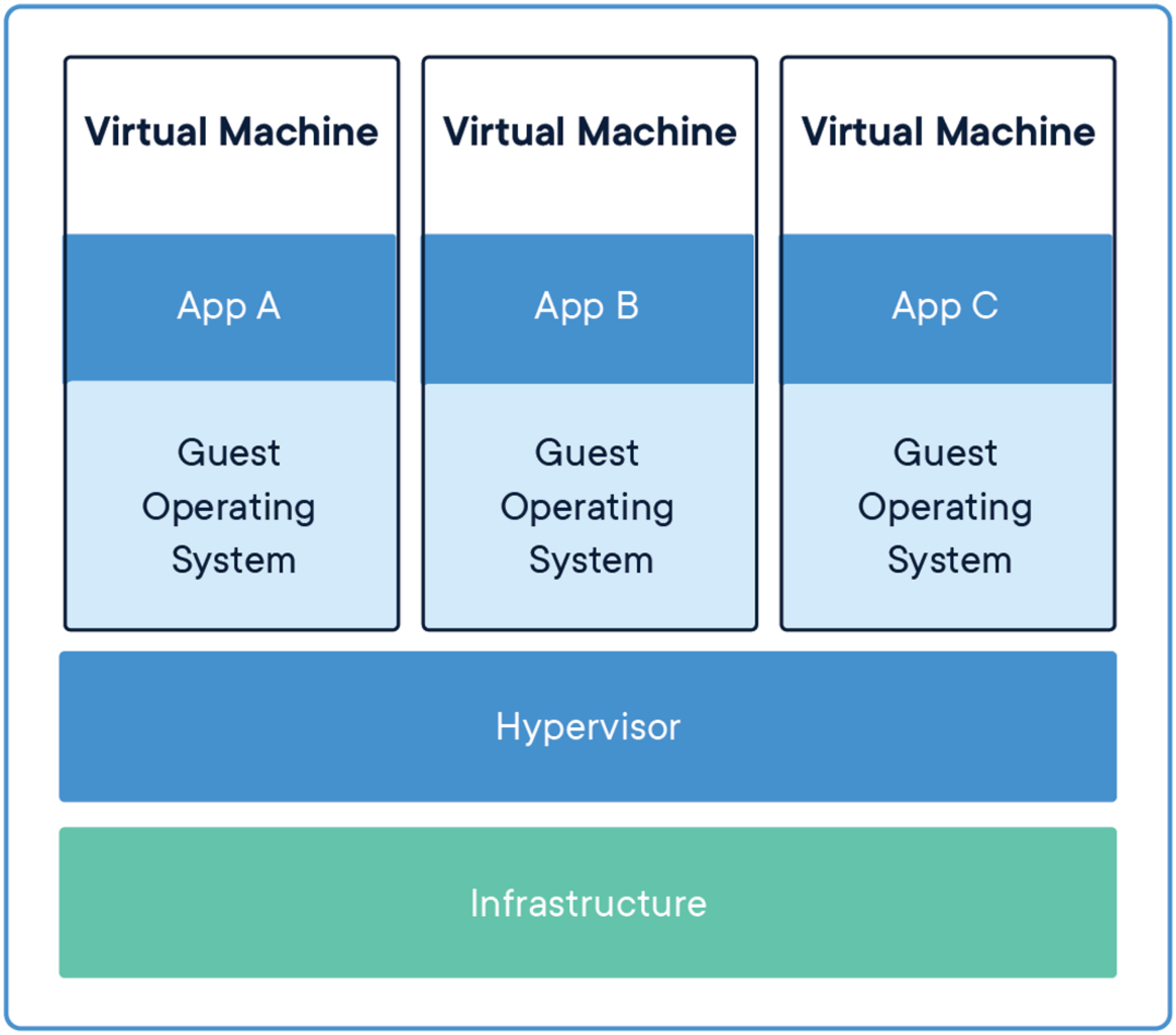
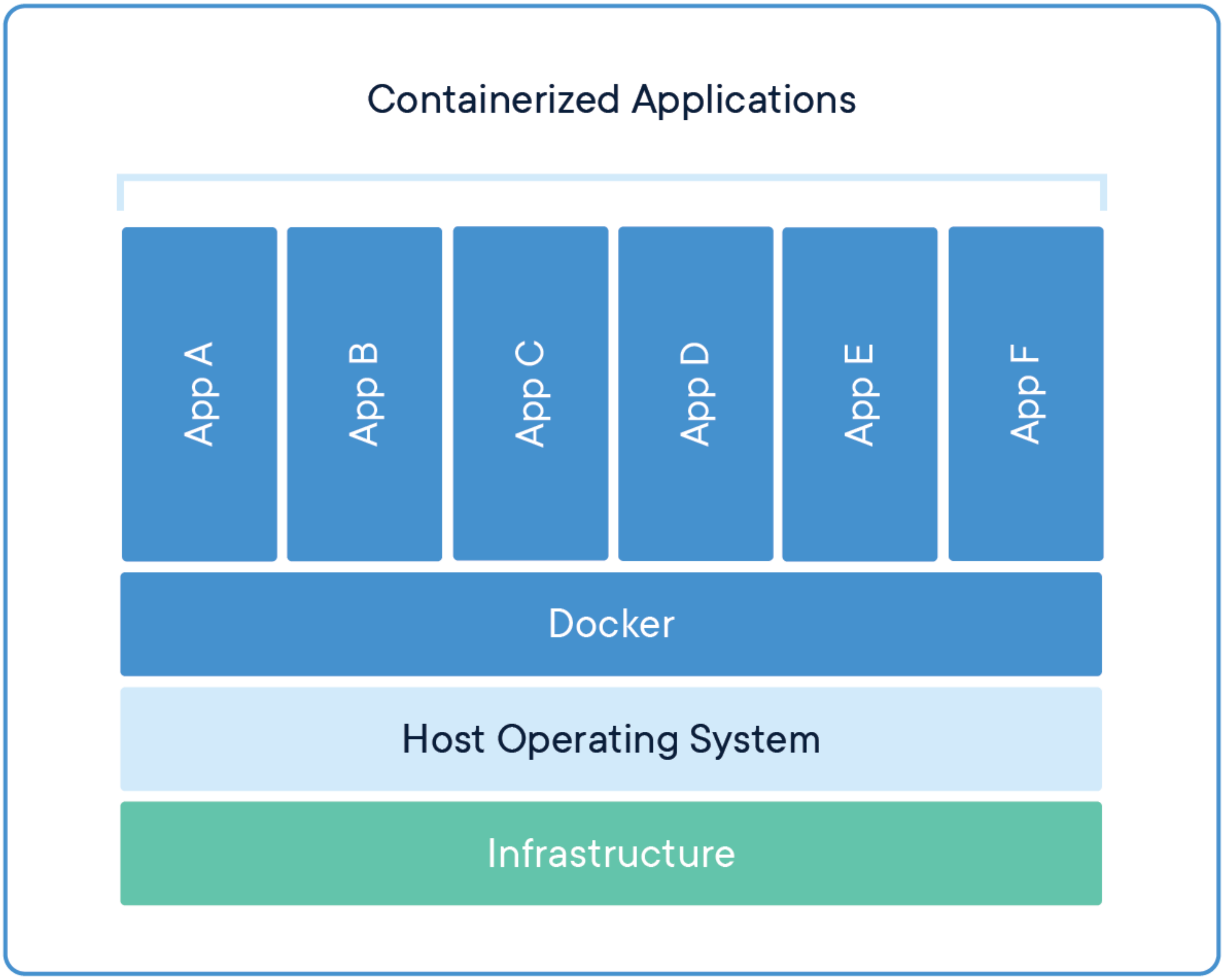
Docker

1. 容器与虚拟机的区别



VIRTUAL MACHINES
Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.



CONTAINERS
Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

2. 服务器部署

Older versions of Docker were called docker, docker.io , or docker-engine. If these are installed, uninstall them:

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

Update the apt package index:

```
sudo apt-get update
```

Install packages to allow apt to use a repository over HTTPS:

```
 sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

Add Docker’s official GPG key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, by searching for the last 8 characters of the fingerprint.

```
$ sudo apt-key fingerprint 0EBFCD88

pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

Use the following command to set up the stable repository. To add the nightly or test repository, add the word nightly or test (or both) after the word stable in the commands below. Learn about nightly and test channels.

```
 sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

Update the apt package index

```
sudo apt-get update
```

Install the latest version of Docker CE and containerd:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Verify that Docker CE is installed correctly by running the hello-world image..

```
sudo docker run hello-world
```

Uninstall Docker CE

```
$ sudo apt-get purge docker-ce
```

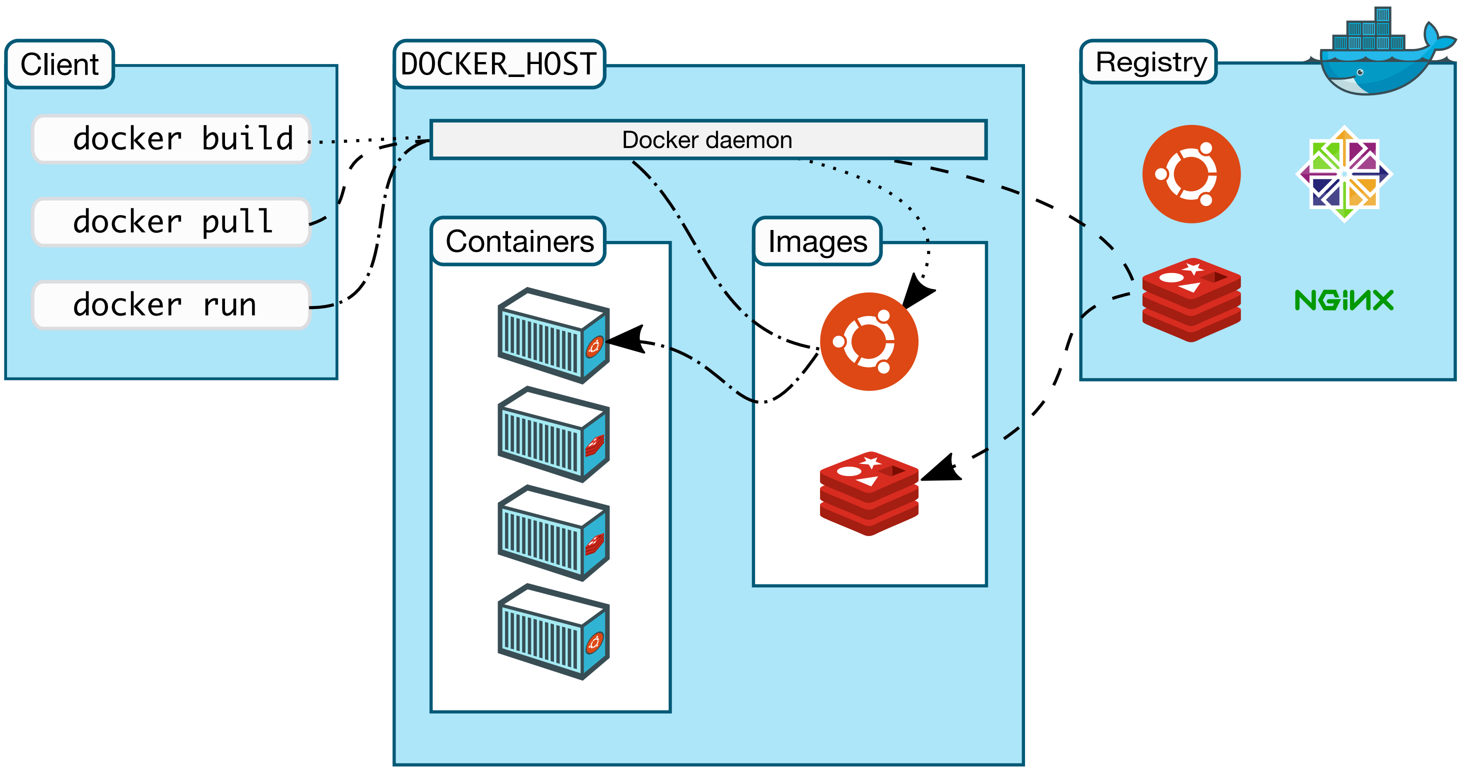
Images, containers, volumes, or customized configuration files on your host are not automatically removed. To delete all images, containers, and volumes:

```
$ sudo rm -rf /var/lib/docker
```

Nvidia docker安装：

```
# If you have nvidia-docker 1.0 installed: we need to remove it and all existing GPU containers
docker volume ls -q -f driver=nvidia-docker | xargs -r -I{} -n1 docker ps -q -a -f volume={} | xargs -r
sudo apt-get purge -y nvidia-docker

# Add the package repositories
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \
  sudo apt-key add -
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | \
  sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update

# Install nvidia-docker2 and reload the Docker daemon configuration
sudo apt-get install -y nvidia-docker2
sudo pkill -SIGHUP dockerd

# Test nvidia-smi with the latest official CUDA image
docker run --runtime=nvidia --rm nvidia/cuda:9.0-base nvidia-smi
```

3. 基本概念



The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

IMAGES

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

CONTAINERS

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Example docker run command

The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs /bin/bash.

```
$ docker run -i -t ubuntu /bin/bash
```

When you run this command, the following happens (assuming you are using the default registry configuration):

If you do not have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.

Docker creates a new container, as though you had run a docker container create command manually.

Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.

Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine’s network connection.

Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.

When you type exit to terminate the /bin/bash command, the container stops but is not removed. You can start it again or remove it.

SERVICES

Services allow you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API. A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time. By default, the service is load-balanced across all worker nodes. To the consumer, the Docker service appears to be a single application. Docker Engine supports swarm mode in Docker 1.12 and higher.

4. 操作

4.1 SSH 登录

ssh服务器安装：

```
$ sudo apt-get install openssh-server
```

检查ssh服务器是否运行：

```
$ ps -e | grep ssh

ubuntu@ubun:~/Documents/liuyifan/detectron$ ps -e | grep ssh
 1520 ?        00:00:00 sshd
 2837 ?        00:00:00 sshd
 2897 ?        00:00:00 sshd
```

如果看到sshd那说明ssh-server已经启动了。

SSH登录服务器

登录：

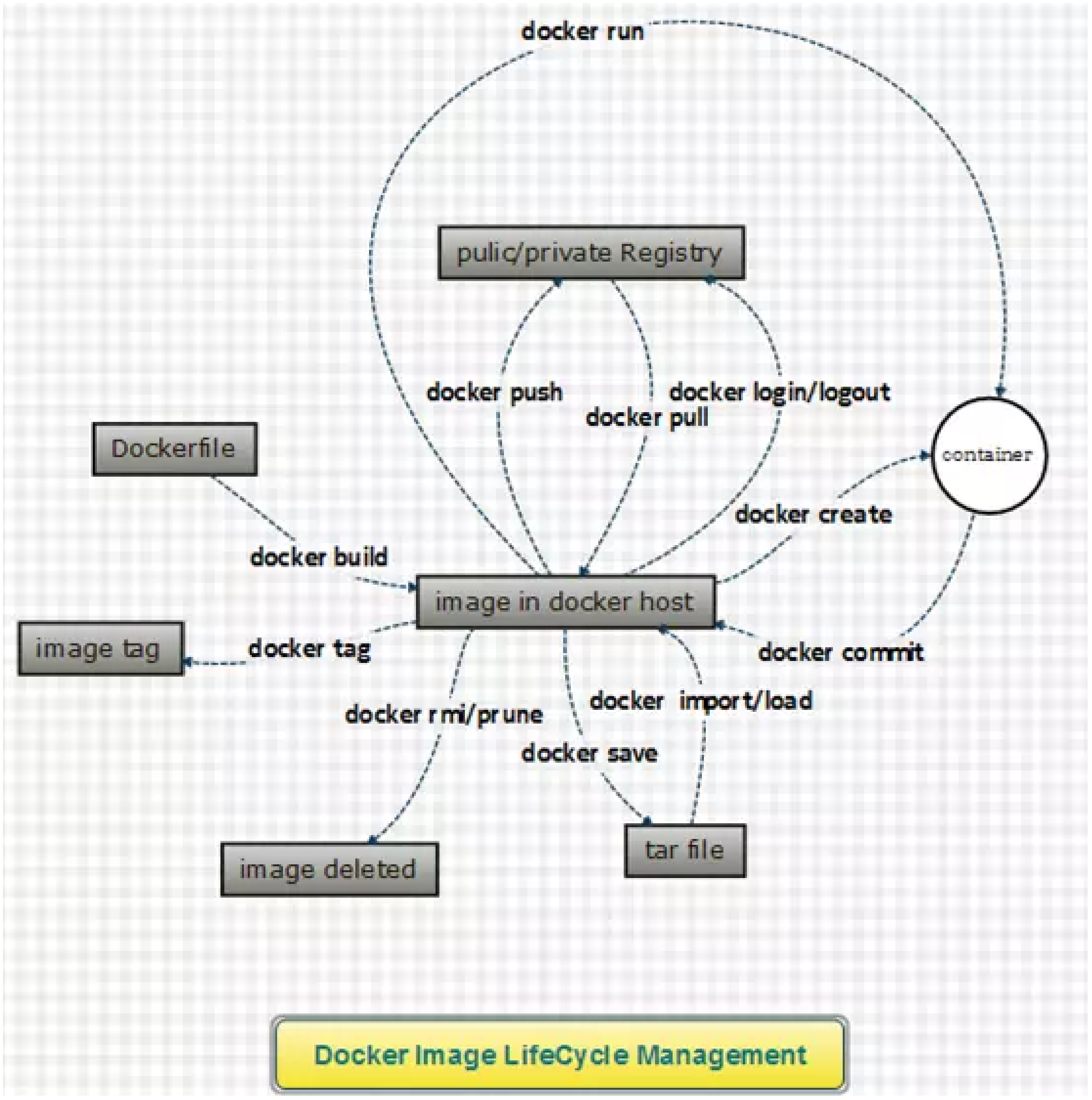
```
ivan@ivan:~$ ssh ubuntu@192.168.12.191

Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.15.0-29-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

49 packages can be updated.
26 updates are security updates.

Last login: Wed May 29 08:47:33 2019 from 192.168.12.139

ubuntu@ubun:~$ ^C
```

退出：

```
$ exit

logout
Connection to 192.168.12.191 closed.
```

4.2 Docker基础操作

4.2.1 镜像基本操作



镜像创建容器：docker create
镜像启动容器：docker run
拉取镜像：docker pull
推送镜像：docker push
给镜像打标签：docker tag
列出本地镜像：docker image ls/docker images
删除本地镜像：docker rmi/docker image rm
构建镜像：docker commit, docker build
查询镜像构建过程：docker history
查询镜像的详细信息：docker inspect
将镜像打成tar包：docker save
登录登出仓库：docker login/docker logout

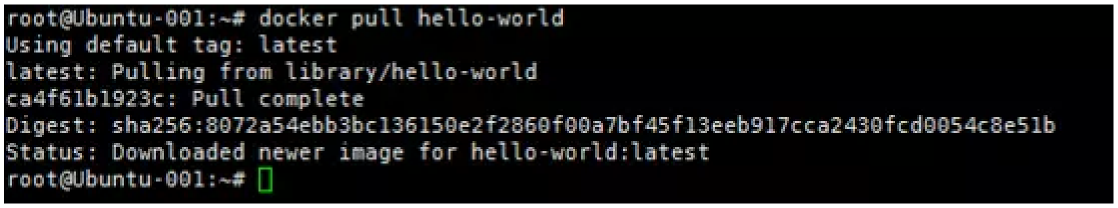
示例：

在远程image仓库里搜索hello-world 镜像

```
$ docker search hello-world

NAME                                        DESCRIPTION                                     STARS
hello-world                                 Hello World! (an example of minimal Dockeriz…   935
kitematic/hello-world-nginx                 A light-weight nginx container that demonstr…   124
tutum/hello-world                           Image to test docker deployments. Has Apache…   60
dockercloud/hello-world                     Hello World!                                    15
crccheck/hello-world                        Hello World web server in under 2.5 MB          6
ppc64le/hello-world                         Hello World! (an example of minimal Dockeriz…   2
souravpatnaik/hello-world-go                hello-world in Golang                           1
carinamarina/hello-world-app                This is a sample Python web application, run…   1
markmnei/hello-world-java-docker            Hello-World-Java-docker                         1
ansibleplaybookbundle/hello-world-db-apb    An APB which deploys a sample Hello World! a…   0
koudaiii/hello-world                                                                        0
kevindockercompany/hello-world                                                              0
ansibleplaybookbundle/hello-world-apb       An APB which deploys a sample Hello World! a…   0
burdz/hello-world-k8s                       To provide a simple webserver that can have …   0
uniplaces/hello-world                                                                       0
ansibleplaybookbundle/hello-world           Simple containerized application that tests …   0
infrastructureascode/hello-world            A tiny "Hello World" web server with a healt…   0
nirmata/hello-world                                                                         0
stumacsolutions/hello-world-container                                                       0
mbrainar/hello-world                        Python-based hello-world web service            0
sharor/hello-world                                                                          0
s390x/hello-world                           Hello World! (an example of minimal Dockeriz…   0
kousik93/hello-world                                                                        0
ebenjaminv9/hello-world                     Hello-world                                     0
jensendw/hello-world                                                                        0
```

从远程仓库拉去hello-world镜像

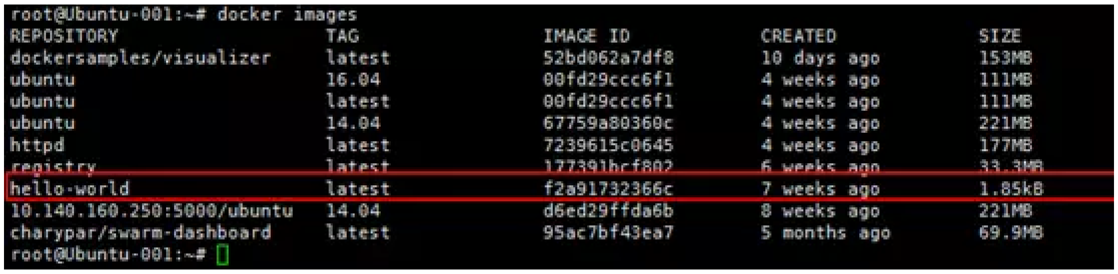
```
$ docker pull hello-world
```



```
root@Ubuntu-001:~# docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:8072a54ebb3bc136150e2f2860f00a7bf45f13eeb917cca2430fcd0054c8e51b
Status: Downloaded newer image for hello-world:latest
root@Ubuntu-001:~#
```

查看本地镜像：

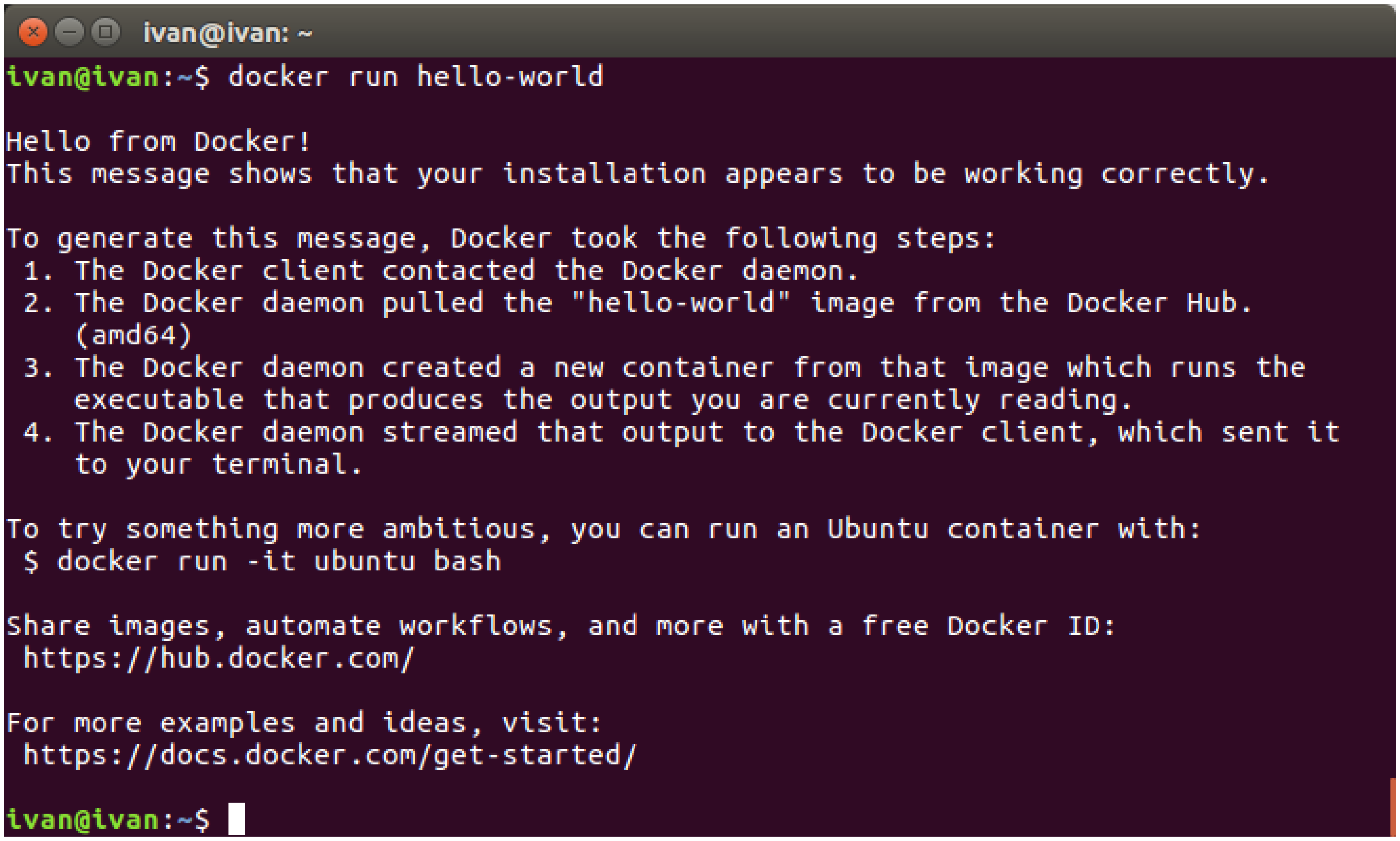
```
$ docker images
```



```
root@Ubuntu-001:~# docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
dockersamples/visualizer      latest              52bd062a7df8        10 days ago         153MB
ubuntu                        16.04               00fd29ccc6f1        4 weeks ago         111MB
ubuntu                        latest              00fd29ccc6f1        4 weeks ago         111MB
ubuntu                        14.04               67759a80360c        4 weeks ago         221MB
httpd                         latest              7239615c0645        4 weeks ago         177MB
registry                      latest              177391bcf802        6 weeks ago         33.3MB
hello-world                   latest              f2a91732366c        7 weeks ago         1.85kB
10.140.160.250:5000/ubuntu    14.04               d6ed29ffda6b        8 weeks ago         221MB
charypar/swarm-dashboard      latest              95ac7bf43ea7        5 months ago        69.9MB
root@Ubuntu-001:~#
```

运行容器：

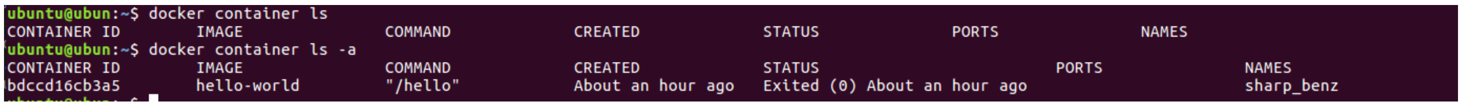
```
$ docker run hello-world
```



```
ivan@ivan: ~
ivan@ivan:~$ docker run hello-world

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/

ivan@ivan:~$
```

每次执行docker run，就创建一个Docker容器进程，拥有独立的文件系统、网络和进程树。使用docker ps或者docker container ls可以查询运行的容器。

那么有人会问，我怎么查询不到呢？ps也查不到相应的进程呢？这是因为容器启动后又退出了。使用docker ps -a或者docker container ls -a查询



```
ubuntu@ubun:~$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
ubuntu@ubun:~$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
bdccd16cb3a5        hello-world         "/hello"            About an hour ago   Exited (0) About an hour ago                        sharp_benz
```

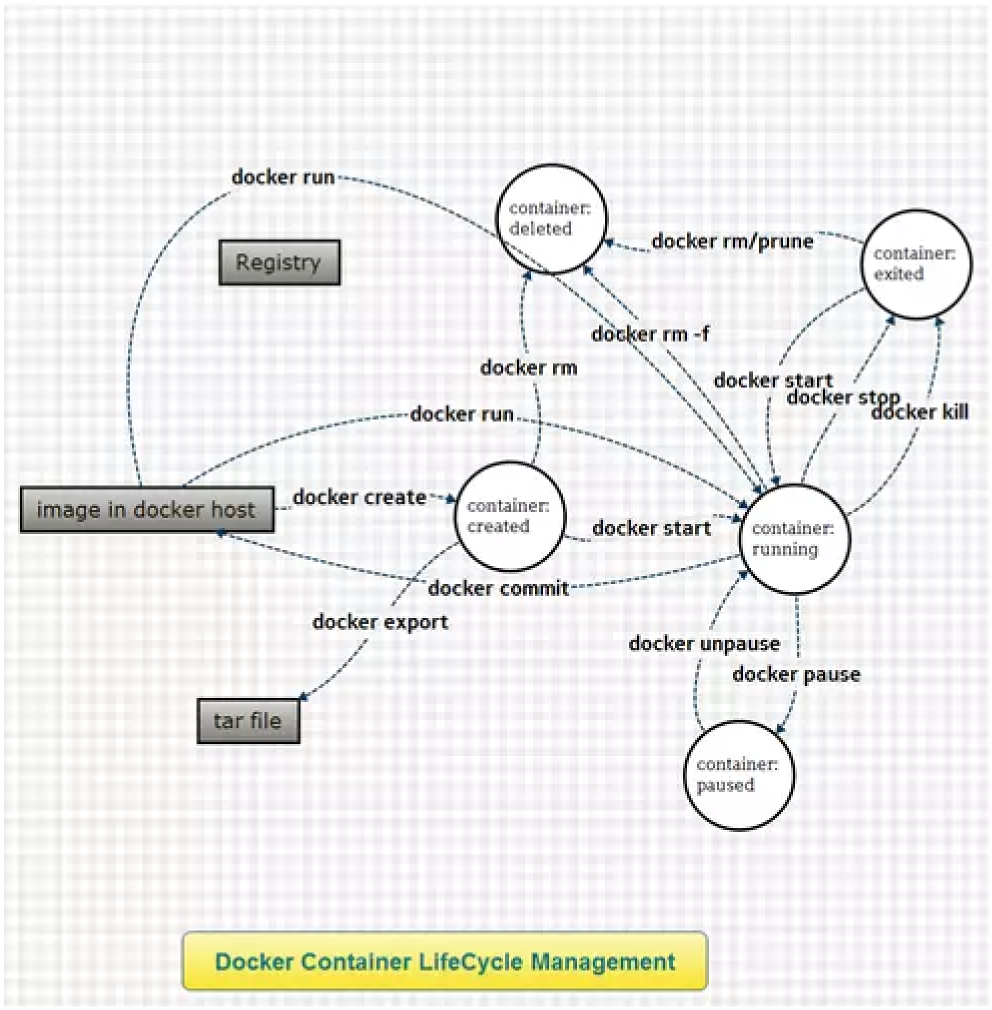
当利用 docker run 来启动容器时，Docker在后台运行的标准操作包括：

1、检查本地是否存在指定的镜像，不存在就从公有仓库下载

2、利用镜像创建一个容器

3、分配一个文件系统，并在只读的镜像层外面挂载一层可读写的容器层
4、从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去，从地址池配置一个ip地址给容器
5、启动容器并执行用户指定的应用程序
6、执行完毕后容器被终止

4.2.2 容器基本操作



新建并启动一个容器：docker run（等价于docker create + docker start）
注：docker run 包含许多可选项，如对内存，CPU等限制，容器的重启策略，指定容器名，导入环境变量等等

新建容器：docker create

启动已停止止或创建好的容器：docker start

停止容器：docker stop/docker kill

重启运行的容器：docker restart（等价于docker stop + docker start）

暂停/恢复容器：docker pause/docker unpause

删除容器：docker rm（运行中的容器无法删除，需要先stop或kill）

查询容器：docker ps（查询运行中的容器）/docker ps -a（查询所有容器）

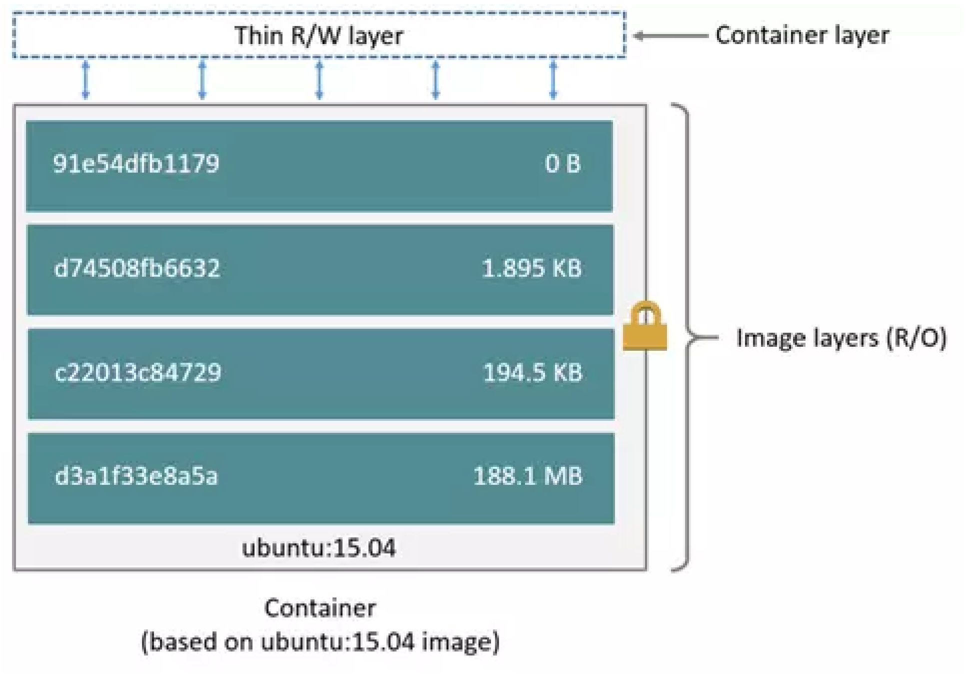
查询容器的详细信息：docker inspect

进入容器： docker exec -it NAME /bin/bash

退出并关闭容器： exit (ctrl+d)

4.3 数据管理

4.3.1 容器内数据区

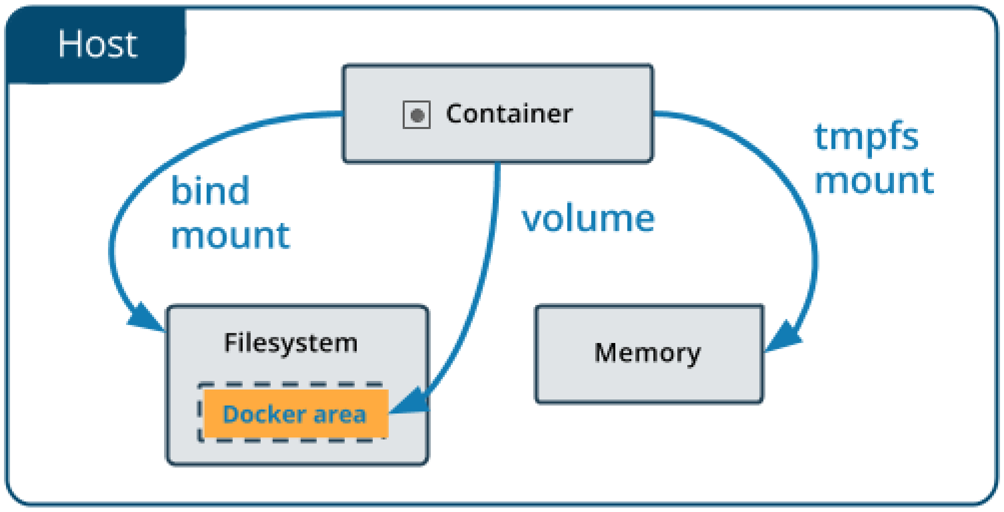


Docker中的数据包含两部分：镜像数据和容器数据。我们已经知道Docker镜像是由多个文件系统（只读层）叠加而成。当我们启动一个容器的时候，Docker会加载只读的镜像层并在其上添加一个可读写的容器层。这个设计使得Docker可以提高镜像构建、存储和分发的效率，节省了时间和存储空间。如果运行中的容器修改了现有的一个已经存在的文件，那该文件将会从下面的只读层复制到读写层，该文件的只读版本仍然存在，只是已经被读写层中该文件的副本所隐藏。当删除Docker容器，并通过该镜像重新启动时，之前的更改将会丢失。在Docker中，镜像层及在顶部的容器层的组合被称为Union File System（联合文件系统）。

容器的读写层可以用来保存数据，但是存在的最大不足就是：容器中的数据会随着容器的消亡而消亡。然而有状态的容器都有持久化数据的需求，为了能够保存（持久化）数据以及共享容器间的数据，Docker提出了Volume的概念。简单来说，Volume就是目录或者文件，它可以绕过默认的联合文件系统，而以正常的文件或者目录的形式存在于宿主机上。Docker对容器数据的管理有两种方式：

1、数据卷（data volume）
2、数据卷容器（data volume container）

4.3.2 数据卷



数据卷是一个或多个容器专门指定绕过Union File System，为持续性或共享数据提供一些有用的功能：

（1）数据卷可以在容器间共享和重用。

（2）数据卷数据的改变是直接修改的。

（3）数据卷数据的改变不会被包括在容器中。

（4）数据卷是持续性的，直到没有容器使用它们。

数据卷操作Docker启动的时候可以通过-v选项添加数据卷，实现将主机上的目录或者文件挂载到容器中。完整的参数为：

```
（1）-v volume-name:container-dir:[rw|wo]

（2）-v host-dir:container-dir:[rw|wo]

（3）-v container-dir:[rw|wo]
```

其中，

volume-name：表示卷名，如果该卷不存在，docker将自动创建。

container-dir：表示容器内部对应的目录，如果该目录不存在，Docker 也会在容器内部创建该目录。

rw|ro：用于控制volume的读写权限。

示例：

（1）docker run -d -v my_volume:/test --name myhttp httpd

查看myhttp容器中是否有my_volume的数据卷：

```
$ docker inspect myhttp
```

找到Mounts部分：

```
        "Mounts": [
            {
                "Type": "volume",
                "Name": "my_volume",
                "Source": "/var/lib/docker/volumes/my_volume/_data",
                "Destination": "/test",
                "Driver": "local",
                "Mode": "z",
                "RW": true,
                "Propagation": ""
            }
        ],
```

查看该数据卷的信息：

```
$ docker inspect my_volume

[
    {
        "CreatedAt": "2019-05-24T17:17:59+08:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/my_volume/_data",
        "Name": "my_volume",
        "Options": null,
        "Scope": "local"
    }
]
```

在服务器硬盘创建文件：

```
$ sudo chmod 777 /var/lib/docker/volumes

$ cd /var/lib/docker/volumes/my_volume/_data

sudo vim test

ls
```

在容器中查看该文件：

```
docker exec -it myhttp /bin/bash

cd /test

ls

cat test
```

(1) 删除容器是如果使用docker rm container将不会删除对应的Volume。如果想要删除可以使用docker rm -v container。另外也可以单独使用docker volume rm volume_name删除volume。
(2） 对于已运行的数据卷容器，不能动态的调整其卷的挂载。Docker官方提供的方法是先删除容器，然后启动时重新挂载。

4.4 如何创建自己的镜像

4.4.1 寻找属于自己的基础镜像

(1)在docker镜像服务器上搜索相关镜像

```
docker search detectron
```

(2)去算法官网找相应的docker 镜像

4.4.2 Dockerfile 修改重修编译镜像

4.4.2.1 指令集：

FROM

FROM命令可能是最重要的Dockerfile命令。改命令定义了使用哪个基础镜像启动构建流程。基础镜像可以为任意镜 像。如果基础镜像没有被发现，Docker将试图从Docker image index来查找该镜像。

FROM命令必须是Dockerfile的首个命令。

```
# Usage: FROM [image name]
FROM ubuntu
```

ADD

ADD命令有两个参数，源和目标。它的基本作用是从源系统的文件系统上复制文件到目标容器的文件系统。如果源是一个URL，那该URL的内容将被下载并复制到容器中。

```
# Usage: ADD [source directory or URL] [destination directory]
ADD /my_app_folder /my_app_folder
```

RUN

RUN命令是Dockerfile执行命令的核心部分。它接受命令作为参数并用于创建镜像。不像CMD命令，RUN命令用于创建镜像（在之前commit的层之上形成新的层）。

```
# Usage: RUN [command]
RUN aptitude install -y riak
```

CMD

和RUN命令相似，CMD可以用于执行特定的命令。和RUN不同的是，这些命令不是在镜像构建的过程中执行的，而是在用镜像构建容器后被调用。

```
# Usage 1: CMD application "argument", "argument", ..
CMD "echo" "Hello docker!"
```

ENTRYPOINT

配置容器启动后执行的命令，并且不可被 docker run 提供的参数覆盖。

每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个起效。

ENTRYPOINT 帮助你配置一个容器使之可执行化，如果你结合CMD命令和ENTRYPOINT命令，你可以从CMD命令中移除“application”而仅仅保留参数，参数将传递给ENTRYPOINT命令。

```
# Usage: ENTRYPOINT application "argument", "argument", ..
# Remember: arguments are optional. They can be provided by CMD
# or during the creation of a container.
ENTRYPOINT echo
# Usage example with CMD:
# Arguments set with CMD can be overridden during *run*
CMD "Hello docker!"
ENTRYPOINT echo
```

ENV

ENV命令用于设置环境变量。这些变量以”key=value”的形式存在，并可以在容器内被脚本或者程序调用。这个机制给在容器中运行应用带来了极大的便利。

```
# Usage: ENV key value
ENV SERVER_WORKS 4
```

EXPOSE

EXPOSE用来指定端口，使容器内的应用可以通过端口和外界交互。

```
# Usage: EXPOSE [port]
EXPOSE 8080
```

MAINTAINER

我建议这个命令放在Dockerfile的起始部分，虽然理论上它可以放置于Dockerfile的任意位置。这个命令用于声明作者，并应该放在FROM的后面。

```
# Usage: MAINTAINER [name]
MAINTAINER authors_name
```

USER

USER命令用于设置运行容器的UID。

```
# Usage: USER [UID]
USER 751
```

VOLUME

VOLUME命令用于让你的容器访问宿主机上的目录。

```
# Usage: VOLUME ["/dir_1", "/dir_2" ..]
VOLUME ["/my_files"]
```

WORKDIR

WORKDIR命令用于设置CMD指明的命令的运行目录。

```
# Usage: WORKDIR /path
WORKDIR ~/
```

4.4.2.2 示例：

```
# Use Caffe2 image as parent image
FROM caffe2/caffe2:snapshot-py2-cuda9.0-cudnn7-ubuntu16.04

RUN mv /usr/local/caffe2 /usr/local/caffe2_build
ENV Caffe2_DIR /usr/local/caffe2_build

ENV PYTHONPATH /usr/local/caffe2_build:${PYTHONPATH}
ENV LD_LIBRARY_PATH /usr/local/caffe2_build/lib:${LD_LIBRARY_PATH}

# Clone the Detectron repository
RUN git clone https://github.com/facebookresearch/detectron /detectron

RUN cd /detectron && git checkout d56e267

# Install Python dependencies
RUN pip install -r /detectron/requirements.txt

# Install the COCO API
RUN git clone https://github.com/cocodataset/cocoapi.git /cocoapi
WORKDIR /cocoapi/PythonAPI
RUN make install

# Go to Detectron root
WORKDIR /detectron

# Set up Python modules
RUN make

# [Optional] Build custom ops
RUN make ops

# change ubuntu sources to CN

ADD sources.list /etc/apt/

RUN mv /etc/apt/sources.list.d /etc/apt/sources.list.d.odd

RUN apt-get clean
```

4.4.2.3 构建命令

进入Dockerfile所在文件夹

```
cd $DETECTRON/docker
```

构建命令

```
docker build -t detectron:c2-cuda9-cudnn7-lyf .
```

(-t:指定镜像 name:tag)

查看已构建镜像

```
$ docker images

REPOSITORY          TAG                                     IMAGE ID            CREATED             S
detectron           c2-cuda9-cudnn7-lyf                     2c30f8100411        15 hours ago        4
detectron           c2-cuda9-cudnn7                         aa8b7e02e115        24 hours ago        4
nvidia/cuda         latest                                  ea87ef28cacb        7 days ago          2
httpd               latest                                  b7cc370ac278        2 weeks ago         1
ylashin/detectron   latest                                  3f2b6bfd94d7        4 months ago        6
caffe2/caffe2       snapshot-py2-cuda9.0-cudnn7-ubuntu16.04 9ae3e8ea7508        16 months ago       3
```