博学谷
www.boxuegu.com

# 第十二章 微服务容器化

一样的在线教育，不一样的教学品质

博学谷
www.boxuegu.com

# 目录 Contents

# Docker 简介



## 小节导学

Docker 是一种虚拟化技术，那么和我们熟悉的虚拟机有什么区别呢?

学习一项新技术，了解它的核心概念是非常重要的，有助于我们更好的掌握技术。

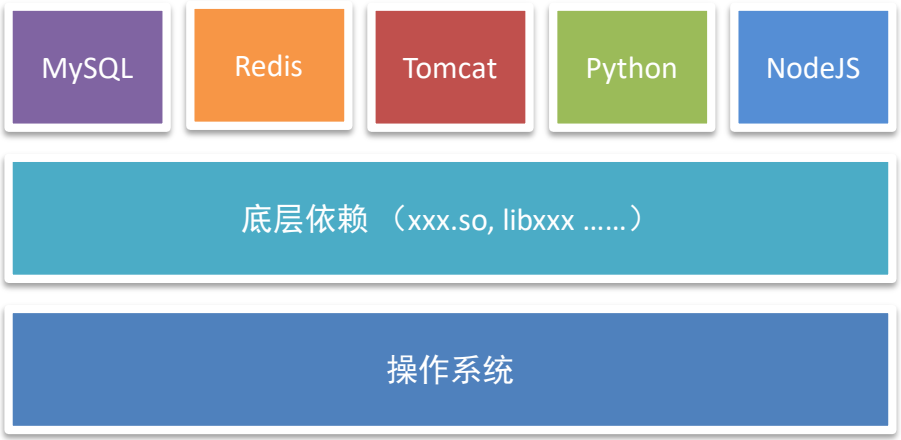
本节我们就从宏观上认识一下 Docker。

- 容器与虚机的区别
- Docker 的主要特点
- Docker 的关键概念
- Docker 架构

# 1. 容器与虚机的区别



## 单机




问题

- 混乱，什么都安装在一起
- 资源利用率低

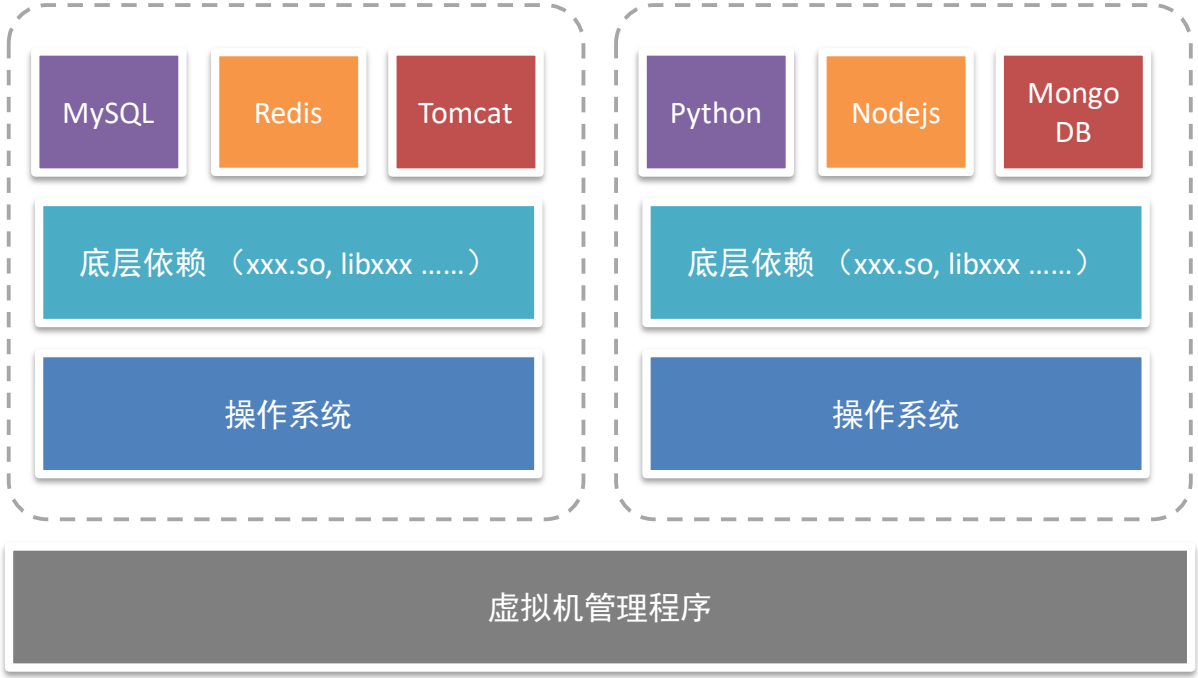
解决方法

- 隔离

# 1. 容器与虚机的区别



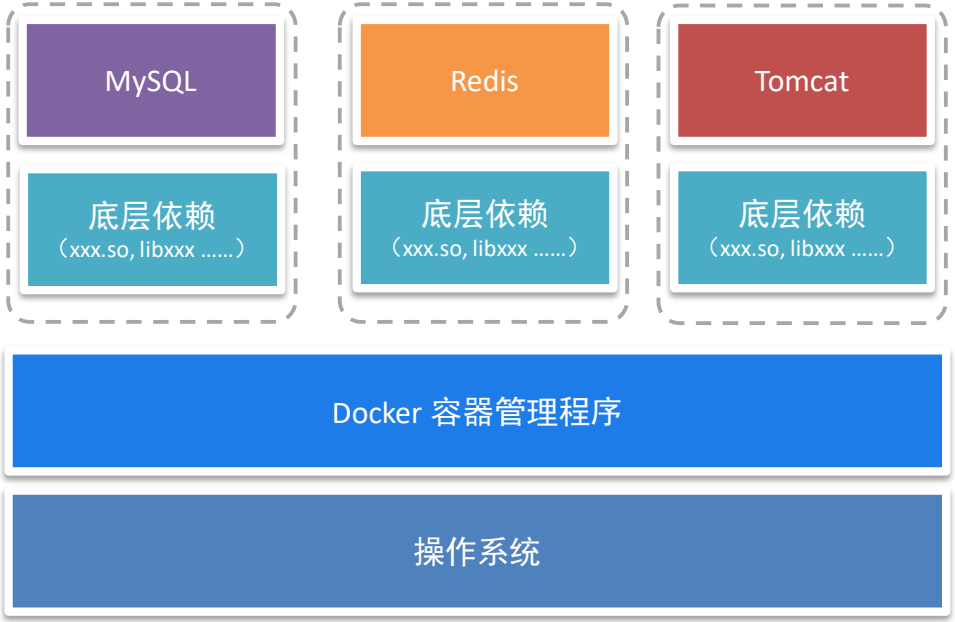
## 虚拟机



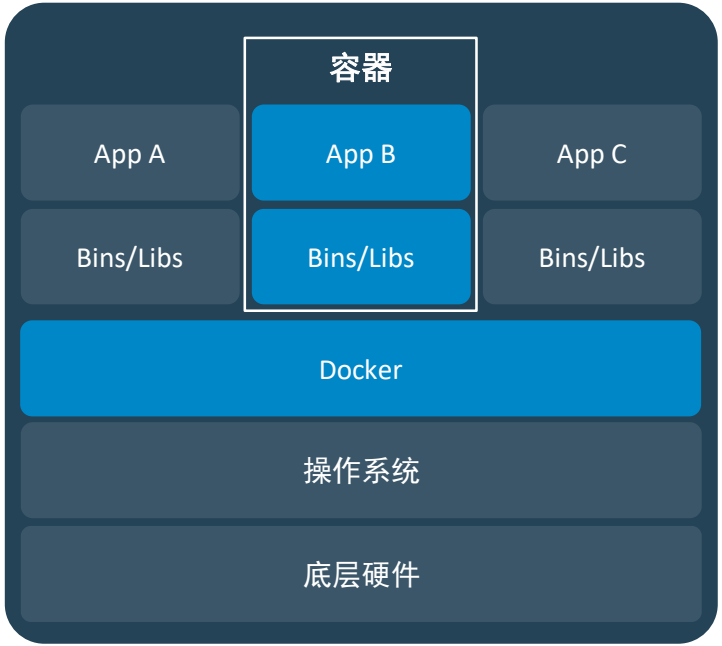
# 1. 容器与虚机的区别

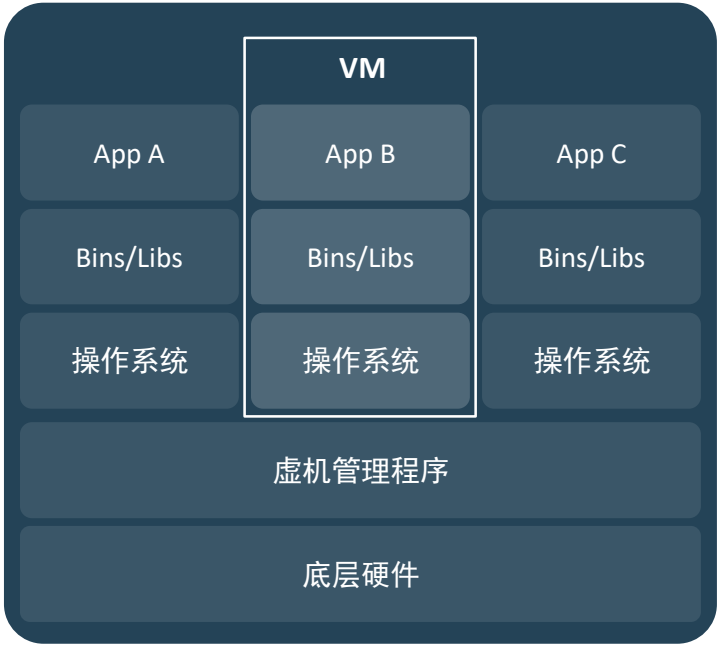


## 容器

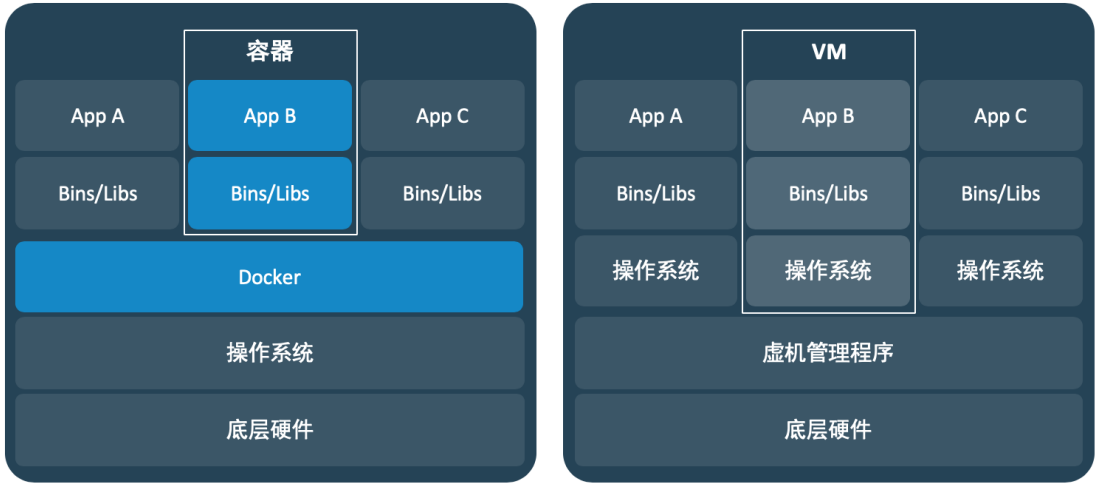


# 1. 容器与虚机的区别


容器
App A
App B
App C
Bins/Libs
Bins/Libs
Bins/Libs
Docker
操作系统
底层硬件


VM
App A
App B
App C
Bins/Libs
Bins/Libs
Bins/Libs
操作系统
操作系统
操作系统
虚机管理程序
底层硬件

# 2. Docker 的主要特点



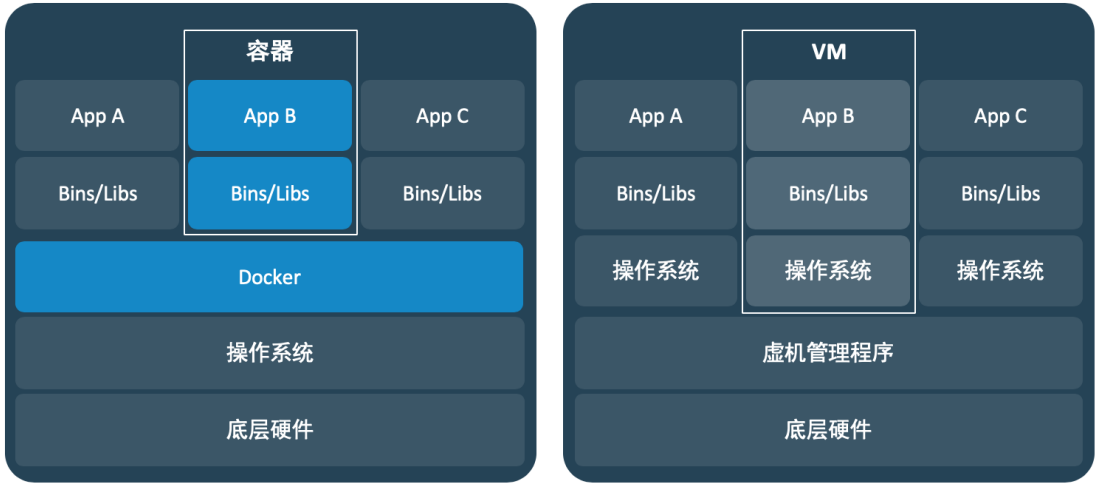

■ 高效利用系统资源

容器不需要进行硬件虚拟

不需要运行完整操作系统

容器对系统资源的利用率更高

# 2. Docker 的主要特点



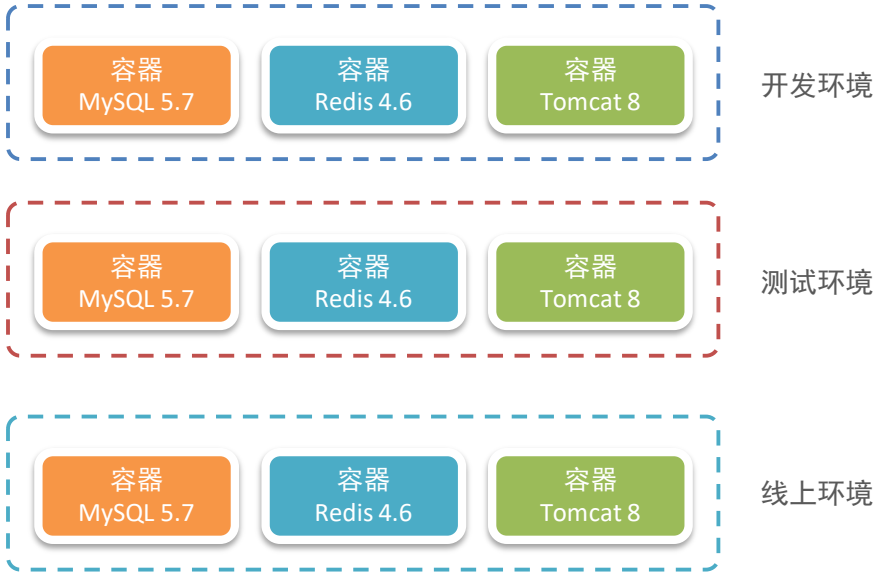

- 高效利用系统资源
- 快速启动

Docker 容器应用直接运行于宿主内核

无需启动完整的操作系统

可以秒级、甚至毫秒级启动

# 2. Docker 的主要特点




- 高效利用系统资源
- 快速启动
- 多环境一致

杜绝出现

*“在我机器上没问题啊”*

这类问题

# 2. Docker 的主要特点


一次编写 到处运行

一次构建 到处运行

- 高效利用系统资源
- 快速启动
- 多环境一致
- 快速迁移

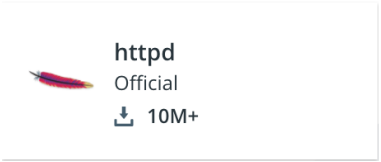
Docker 确保了执行环境的一致性

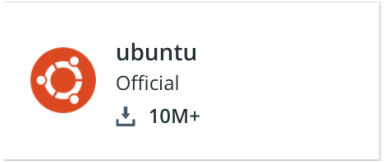
不用担心环境的变化

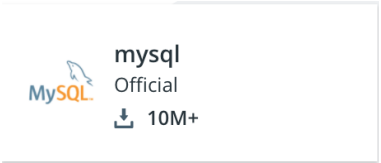
导致无法正常运行

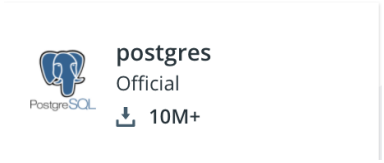
# 2. Docker 的主要特点






- 高效利用系统资源
- 快速启动
- 多环境一致
- 快速迁移
- 轻松扩展

海量官方镜像，直接可用

也可作为自定制的基础

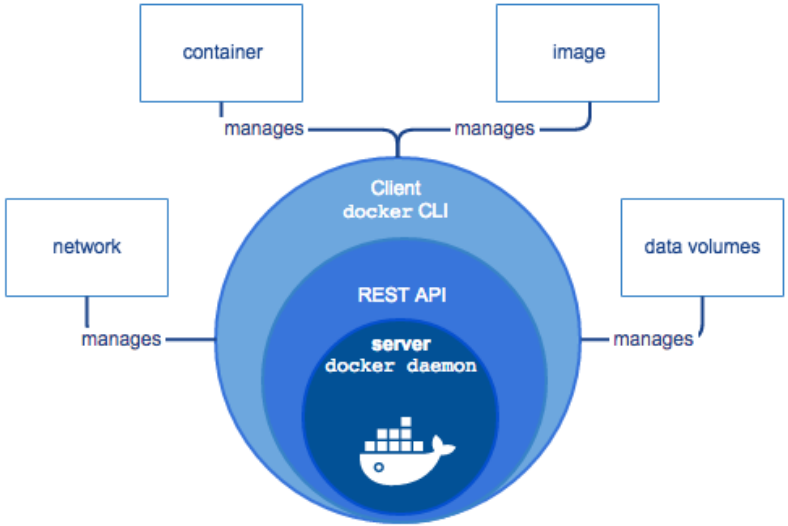
# 3. Docker 的关键概念



## Docker 引擎

核心组件:

- Server
- REST API
- CLI



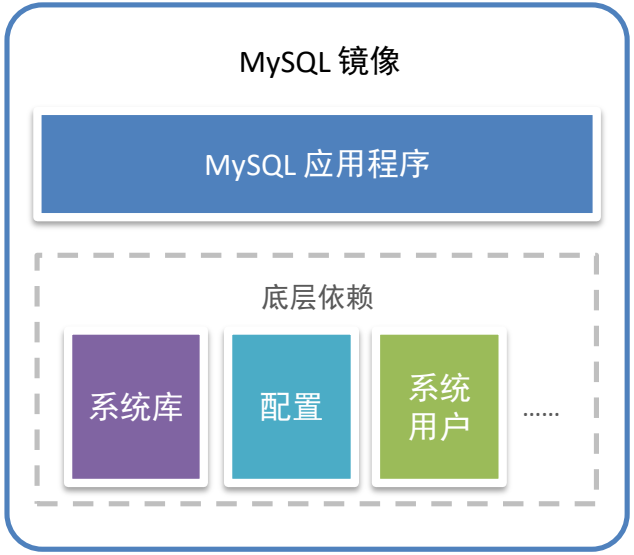
# 3. Docker 的关键概念



## Docker 镜像

Docker 镜像是一个特殊的文件系统，提供：

- 应用程序
- 依赖库
- 资源文件
- 配置信息



# 3. Docker 的关键概念

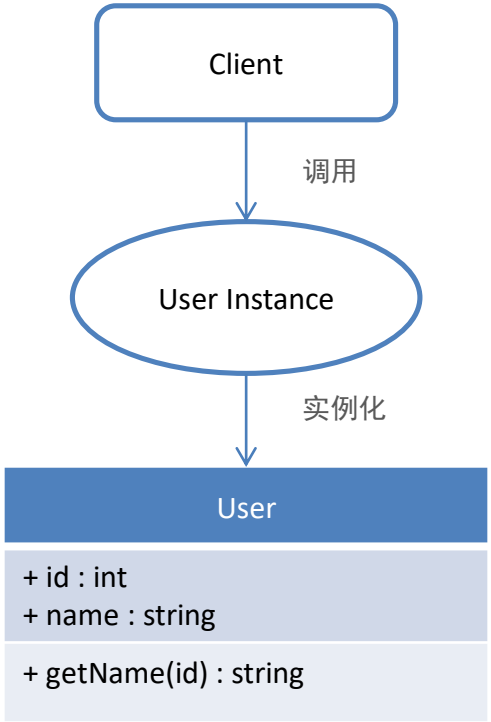


## Docker 容器

镜像是静态的定义，容器是镜像运行时的实例。

类比：

- 镜像 -> 类
- 容器 -> 实例



# 3. Docker 的关键概念

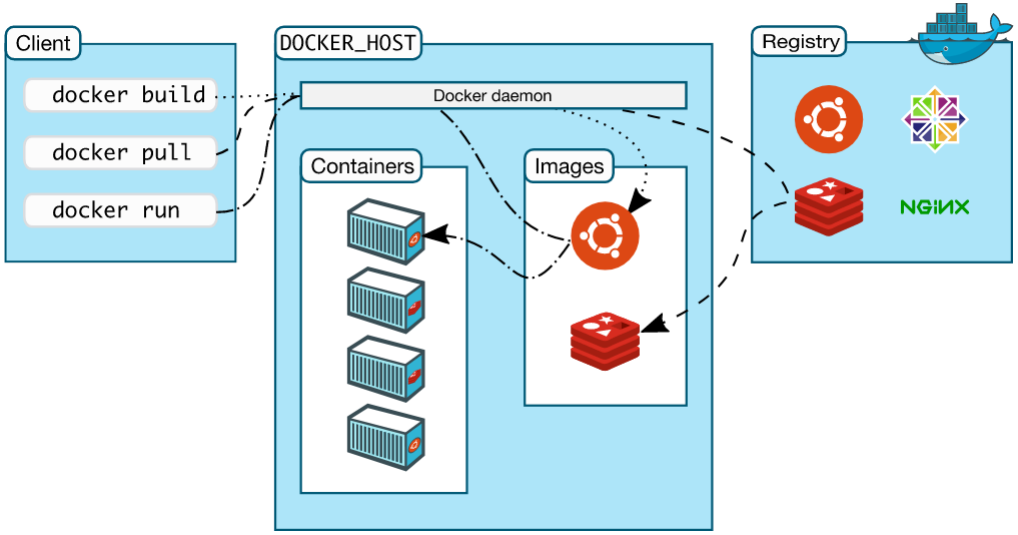


## Docker 仓库

仓库（Registry）是存放 Docker 镜像的地方，例如代码有代码仓库来存放管理。

仓库分为 **公有、私有。**
例如代码仓库也公有的 GitHub，也有私有的 Gitlab。

# 4. Docker 架构


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

# Docker 简介-总结




## 重难点

1. Docker 的作用
2. Docker 的关键概念
3. Docker 整体架构

# Docker 简介-总结




## 重难点

1. Docker 的作用

（ Docker使用一种非常轻量化，低成本的方式实现了应用进程之间的彻底隔离）

2. Docker 的关键概念

3. Docker 整体架构

# Docker 简介-总结



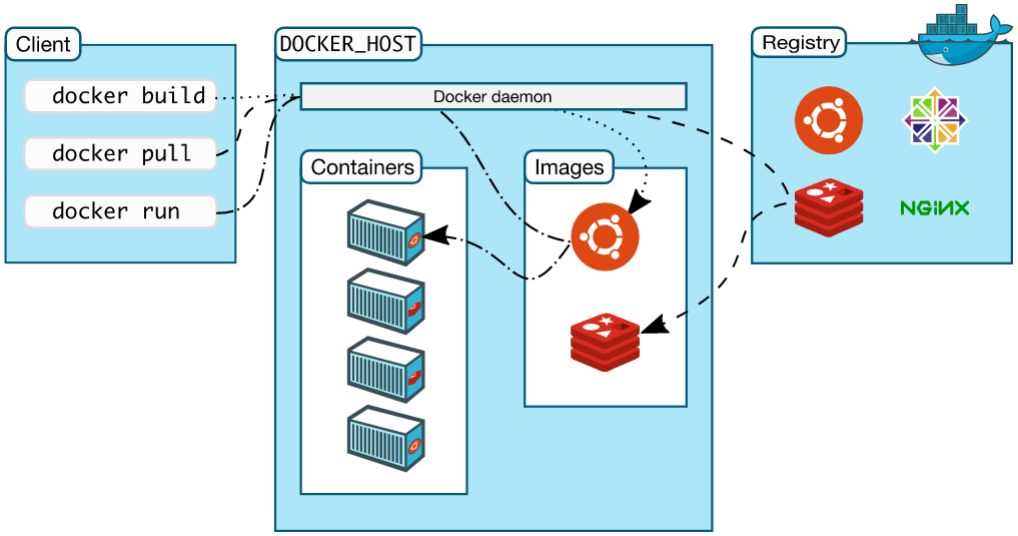

## 重难点

1. Docker 的作用
2. Docker 的关键概念

（引擎、镜像、容器、仓库）

3. Docker 整体架构

# Docker 简介-总结


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

# Docker 简介-总结




## 重难点

1. Docker 的作用
2. Docker 的关键概念
3. Docker 整体架构

## 下节

Docker 安装，运行起来

博学谷
www.boxuegu.com

# 目录 Contents

# Docker 安装启动



## 小节导学

本节学习 Docker 的安装方法，还有一个重点，由于国内网络原因，从 Docker 官方镜像仓库下载速度极慢，几乎无法使用，所以需要配置国内 Docker 仓库的镜像。

- Linux 中 Docker 安装
- Windows  中 Docker 安装
- Mac 中 Docker 安装
- Docker 容器启动
- Docker 仓库镜像配置

# 1. Linux 中 Docker 安装



## 以 CentOS7 为例：

```
> sudo yum remove docker  docker-client docker-client-latest \
                  docker-common docker-latest docker-latest-logrotate \
                  docker-logrotate docker-selinux \
                  docker-engine-selinux docker-engine
> sudo yum install -y yum-utils

> sudo yum-config-manager \
    --add-repo \
    http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

> sudo yum makecache fast
> sudo yum install docker-ce
> sudo service docker start
```

# 2. Windows 中 Docker 安装



## 步骤

1. 下载安装

```
https://hub.docker.com/editions/community/docker-ce-desktop-windows/
```

2. 测试

```
> docker --version
Docker version 19.03.5, build 633a0ea
```

注意 系统环境要求：

- 64位系统
- 4GB 内存
- BIOS 必须开启硬件虚拟化的支持

# 3. Mac 中 Docker 安装



## 步骤

1. 下载安装

```
https://hub.docker.com/editions/community/docker-ce-desktop-mac/
```

2. 测试

```
> docker --version
Docker version 19.03.5, build 633a0ea
```

注意 系统环境要求：

- 硬件需 2010 年之后的
- 系统版本需 10.13+
- 至少 4GB 内存
- 不可以安装 VirtualBox 4.3.30 之前的版本

# 4. Docker 容器启动



## 启动测试容器 hello

```
> docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
Status: Downloaded newer image for hello-world:latest

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

# 5. Docker 仓库镜像配置



## 镜像加速器列表

| 镜像加速器 | 加速器地址 |
| --- | --- |
| Docker 中国官方镜像 | https://registry.docker-cn.com |
| DaoCloud | http://f1361db2.m.daocloud.io（登录，系统分配） |
| 阿里云 | https://<your_code>.mirror.aliyuncs.com（登录，系统分配） |
| 七牛云 | https://reg-mirror.qiniu.com |
| 网易云 | https://hub-mirror.c.163.com |
| 腾讯云 | https://mirror.ccs.tencentyun.com |

# 4. Docker 容器启动



## 加速器配置（Linux）

以 Linux 环境下配置 Docker 官方加速器为例：

```
sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'
{
    "registry-mirrors": [
        "https://registry.docker-cn.com"
    ]
}
EOF

sudo systemctl daemon-reload
sudo systemctl restart docker
```

# Docker 安装启动-总结




## 重难点

1. 自己系统环境下的 Docker 安装
2. Docker 镜像加速器的配置

# Docker 安装启动-总结




## 重难点

1. 自己系统环境下的 Docker 安装
2. Docker 镜像加速器的配置

## 下节

Docker 镜像操作

# 目录 Contents

# Docker 镜像操作



## 小节导学

本节学习 Docker 镜像的主要操作方法。

### 拉取镜像

- 命令格式
- 命令示例
- 镜像多层存储

### 列出镜像

- 命令格式
- 命令示例
- 悬挂镜像
- 特定格式显示

### 删除镜像

- 命令格式
- 命令示例
- 配合列出命令

# 1. 拉取镜像



命令格式：

```
docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]
```

命令示例：

```
> docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

# 1. 拉取镜像

## 镜像多层存储

从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储构成。

下载也是一层层的去下载，并非单一文件。

```
docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
5bed26d33875: Pull complete
f11b29a9c730: Pull complete
930bda195c84: Pull complete
78bf9a5ad49e: Pull complete
Digest: sha256:bec5a2727be7fff3d308193cf
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

# 2. 列出镜像



命令格式：

```
docker images
```

命令示例：

```
> docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
ubuntu                    latest              4e5021d210f6        8 days ago          64.2MB
prom/prometheus           latest              e935122ab143        6 weeks ago         134MB
grafana/grafana           latest              e8b174eeb4d4        7 weeks ago         233MB
rabbitmq                  3.8.2-management    a64a4ae7bc1f        7 weeks ago         181MB
```

列表包含：仓库名、标签、镜像 ID、创建时间、所占用的空间。

# 2. 列出镜像



## 特定格式显示

例如以表格等距显示，有标题行，自定义列：

```
> docker image ls --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"
IMAGE ID            REPOSITORY                TAG
4e5021d210f6        ubuntu                    latest
e935122ab143        prom/prometheus           latest
e8b174eeb4d4        grafana/grafana           latest
a64a4ae7bc1f        rabbitmq                  3.8.2-management
2ddef5390d3a        nacos/nacos-server        latest
```

# 2. 列出镜像



## 特定格式显示

只显示镜像ID：

```
> docker images -q
4e5021d210f6
e935122ab143
e8b174eeb4d4
a64a4ae7bc1f
2ddef5390d3a
8bf17b01b097
fce289e99eb9
f6a778d59b4a
```

# 2. 列出镜像



## 悬挂镜像

名称为“<none>”的镜像称为“悬挂镜像”。

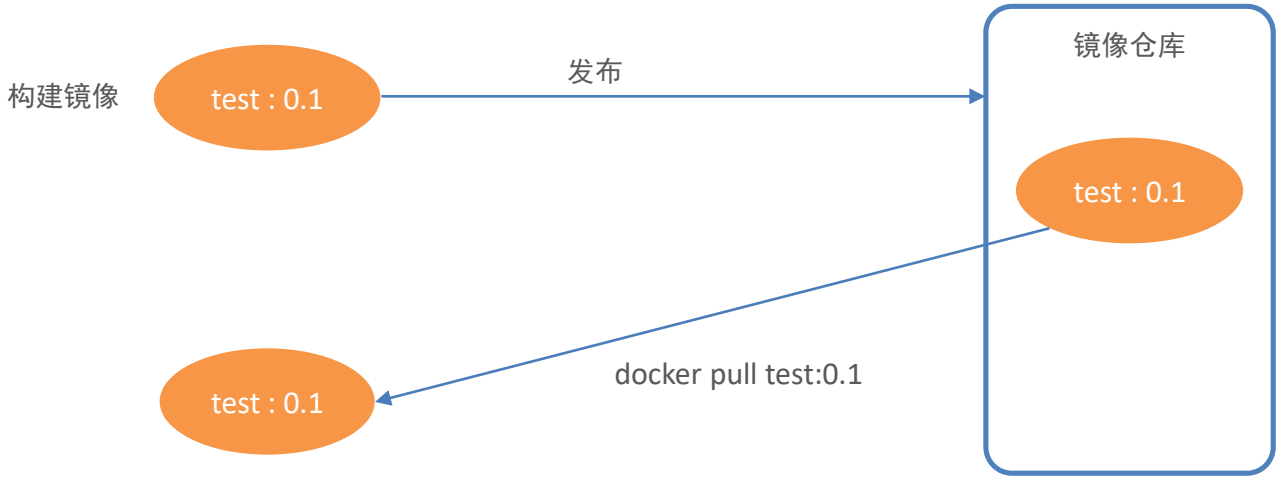
例如某个版本有了新镜像，本地旧镜像名称就被撤销了，产生此类镜像。

查询命令：

```
> docker images -f dangling=true
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              00285df0df87        1 days ago          34 MB
```

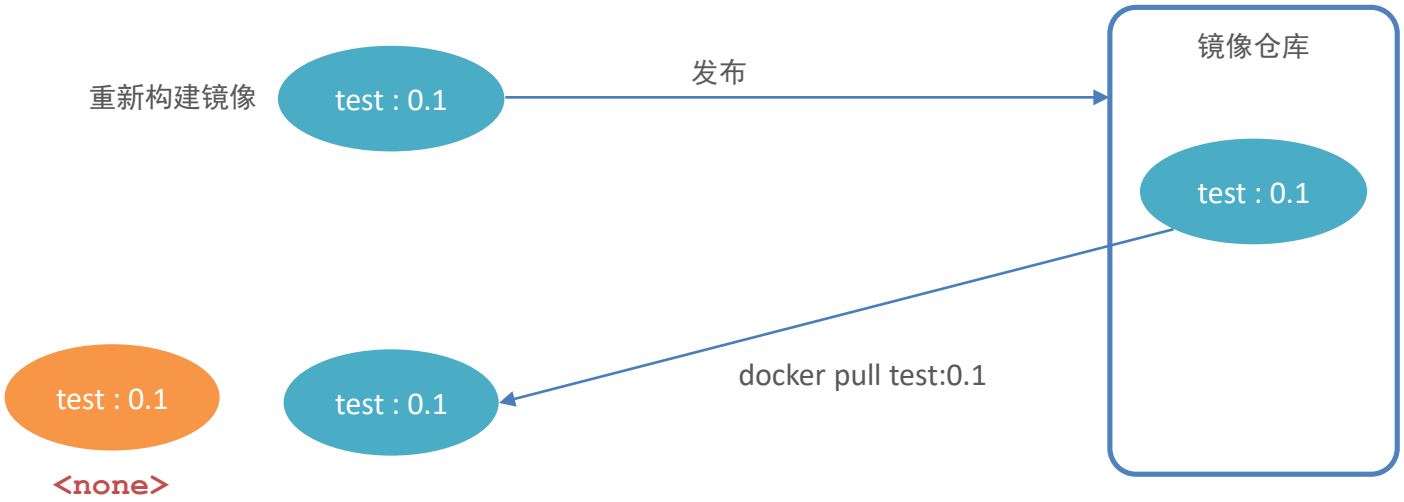
# 2. 列出镜像

## 悬挂镜像



# 2. 列出镜像

## 悬挂镜像


重新构建镜像
test : 0.1
发布
镜像仓库
test : 0.1
test : 0.1
test : 0.1
<none>
docker pull test:0.1

# 2. 列出镜像



## 悬挂镜像

名称为“<none>”的镜像称为“悬挂镜像”。

例如某个版本有了新镜像，本地旧镜像名称就被撤销了，产生此类镜像。

查询命令：

```
> docker images -f dangling=true
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              00285df0df87        1 days ago          34 MB
```

# 3. 删除镜像



命令格式：

```
docker image rm [选项] <镜像1> [<镜像2> ...]
```

命令示例：

```
# 列出镜像
> docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
ubuntu                      latest              4e5021d210f6        8 days ago          64.2MB
prom/prometheus             latest              e935122ab143        6 weeks ago         134MB
# 删除镜像
> docker image rm 4e5021d210f6
```

镜像ID简化：可以使用 ID 的前 3 位来代替，方便输入。

```
> docker image rm 4e5
```

# 3. 删除镜像



## 与列出镜像命令配合

列出镜像的同时，直接删除镜像，例如：

```
# 删除 ubuntu 镜像
> docker image rm $(docker image ls -q ubuntu)

# 删除悬挂镜像
> docker image rm $(docker images -f dangling=true)
```

## 清理所有悬挂镜像

```
> docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
```

# Docker 镜像操作-总结




## 重难点

1. 拉取镜像操作
2. 列出镜像操作
3. 删除镜像操作

# Docker 镜像操作-总结


## 重难点

1. 拉取镜像操作（docker pull image:tag、多层存储结构）
2. 列出镜像操作
3. 删除镜像操作

# Docker 镜像操作-总结




## 重难点

1. 拉取镜像操作
2. 列出镜像操作（docker images、悬挂镜像）
3. 删除镜像操作

# Docker 镜像操作-总结




## 重难点

1. 拉取镜像操作
2. 列出镜像操作
3. 删除镜像操作（docker image rm [id]/[image:tag]）

# Docker 镜像操作-总结




## 重难点

1. 拉取镜像操作
2. 列出镜像操作
3. 删除镜像操作

## 下节

自己动手**构建镜像**

# 目录 Contents

# Dockerfile 构建镜像

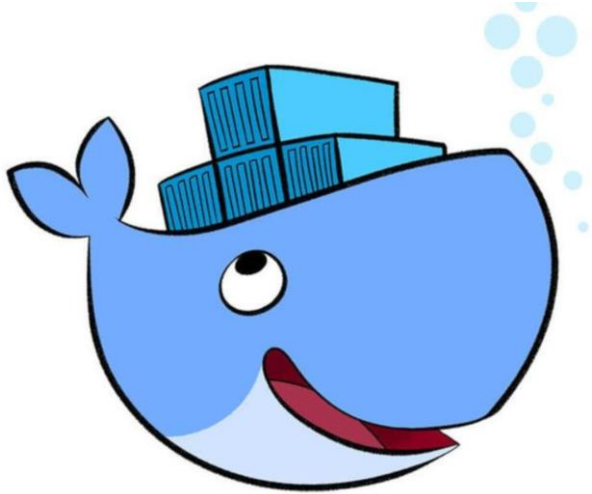


## 小节导学

Docker 的口号：*一次构建，到处运行*

使用 dockerfile 来定制化我们自己的镜像

- Dockerfile 基础结构
- Dockerfile 核心指令



# 1. Dockerfile 基础结构

```
#基于centos镜像
FROM centos
```

基础镜像

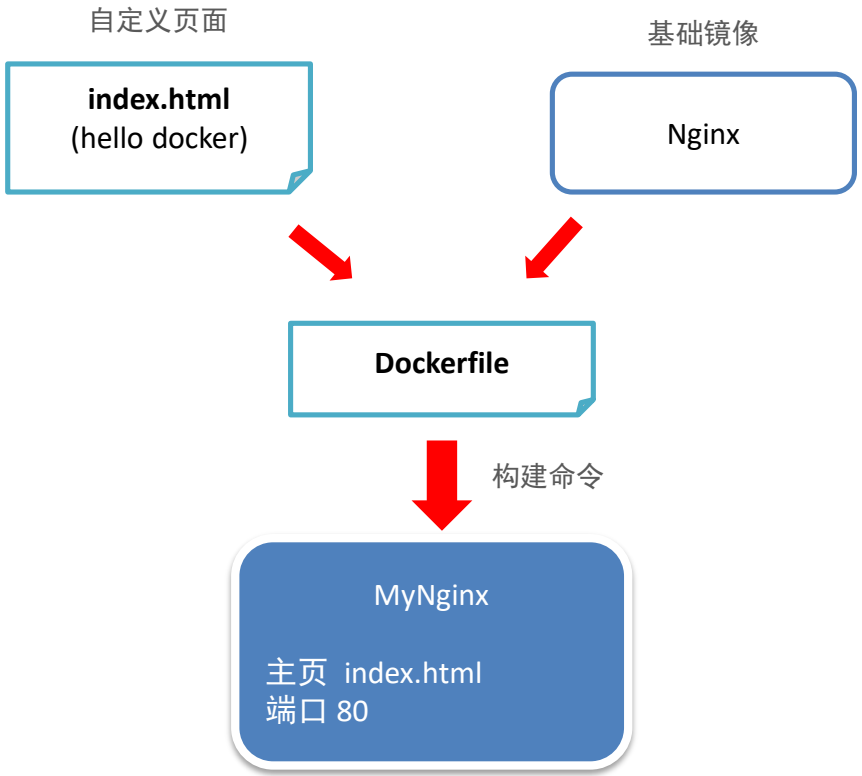
```
#安装httpd软件包
RUN yum -y install httpd

#开启80端口
EXPOSE 80

#复制网站文件包拷贝并解压到web站点下
ADD webapp.zip /var/www

#复制该脚本至镜像中
COPY run.sh /run.sh
```

镜像操作指令

```
#当启动容器时执行的脚本文件
CMD ["/run.sh"]
```

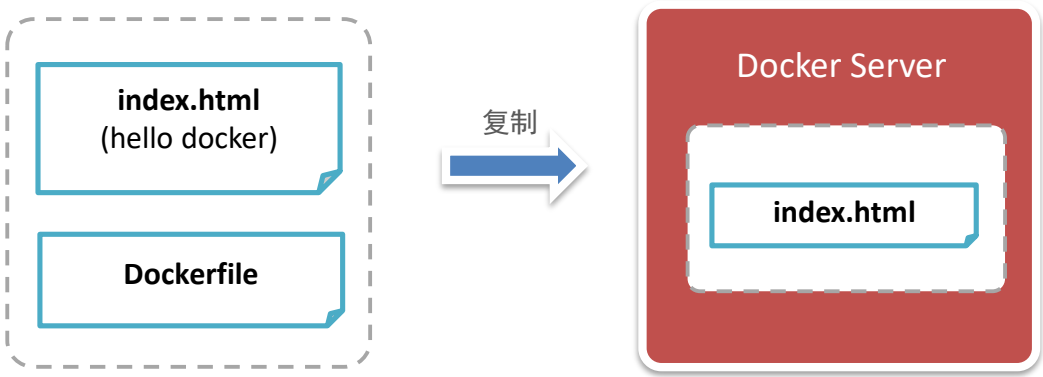
容器启动时执行的指令

# 1. Dockerfile 基础结构



# 1. Dockerfile 基础结构

## 构建上下文




docker build -t mynginx .

# 2. Dockerfile 核心指令



## FROM 基础镜像

从镜像仓库拿一个镜像，作为此次构建镜像的基石。

```
FROM centos:6
```

## RUN 执行命令

构建镜像过程中运行的 Shell 命令

```
RUN ["yum", "install", "httpd"]
RUN yum install httpd
RUN buildDeps='gcc libc6-dev make' \
         && apt-get update \
         && apt-get install -y
```

# 2. Dockerfile 核心指令



## CMD、 ENTRYPOINT 启动容器执行命令

**相同点**

- 容器启动时执行
- 各自都只能有一个生效，有多个时，只有最后一个生效

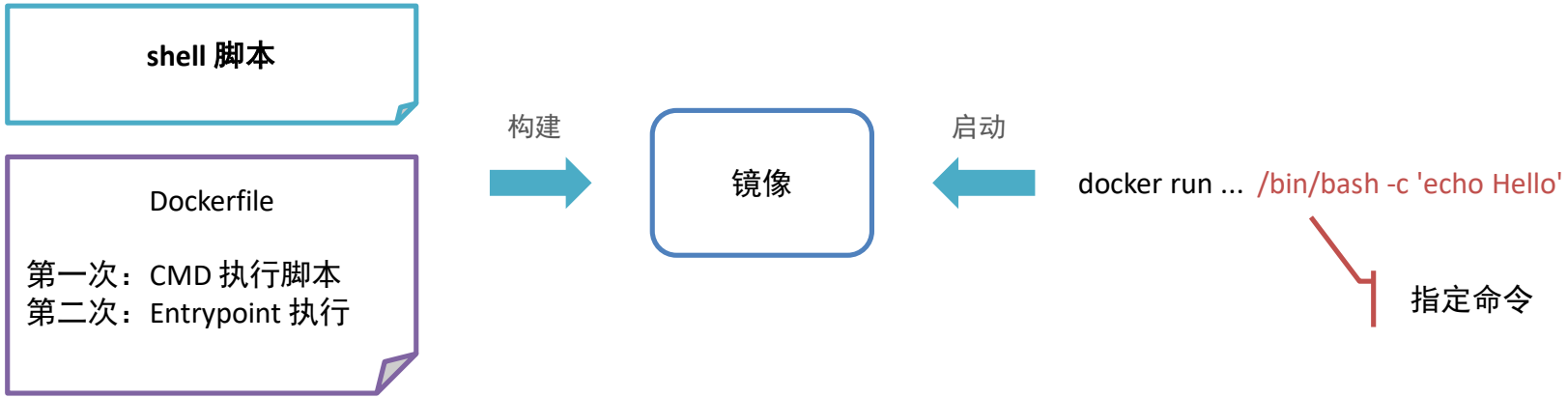
**不同点**

- CMD 指定的命令可以被 docker run 中指定的命令覆盖，而 ENTRYPOINT 不会，会作为自己命令的参数
- CMD 可以不指定命令，只有参数，这时可以作为 ENTRYPOINT 的默认参数，而且可以在 docker run 时替换

# 2. Dockerfile 核心指令

## CMD、 ENTRYPOINT 启动容器执行命令

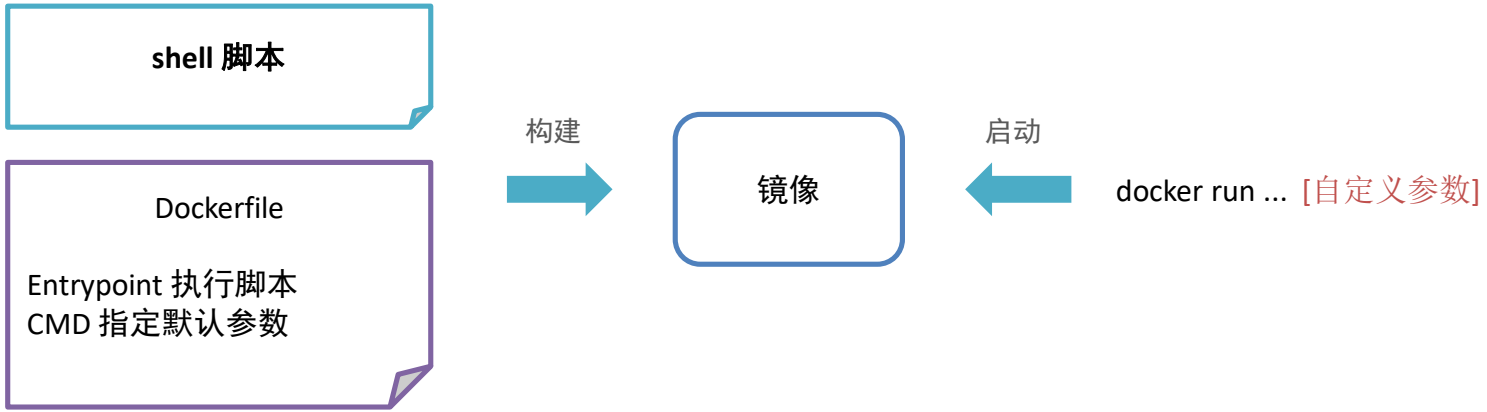
**实验1 –** 是否可以被启动容器时指定的命令覆盖



# 2. Dockerfile 核心指令

## CMD、 ENTRYPOINT 启动容器执行命令

**实验2 –** CMD 为 ENTRYPOINT 提供默认参数



# 2. Dockerfile 核心指令



## COPY/ADD 拷贝文件或目录到镜像

ADD 含义同 COPY，但有增强功能：自动解压。

```
COPY index.jsp /var/www
ADD webapp.zip /var/www
```

## WORKDIR 指定工作目录

指定当前的工作路径，类似 shell 中的“cd”，RUN、CMD、ENTRYPOINT、COPY、AND 都基于此路径
也是用户进入容器后的默认路径。

```
WORKDIR /home
```

# 2. Dockerfile 核心指令



## EXPOSE 暴露端口

**声明**容器打算使用什么端口

```
EXPOSE 80 8080
```

## ENV 环境变量

设置容器内的环境变量，其他指令可以直接引用

```
ENV VERSION=3.0 DEBUG=true
ENV MYSQL_PASSWORD 123456
```

# Dockerfile 构建镜像-总结

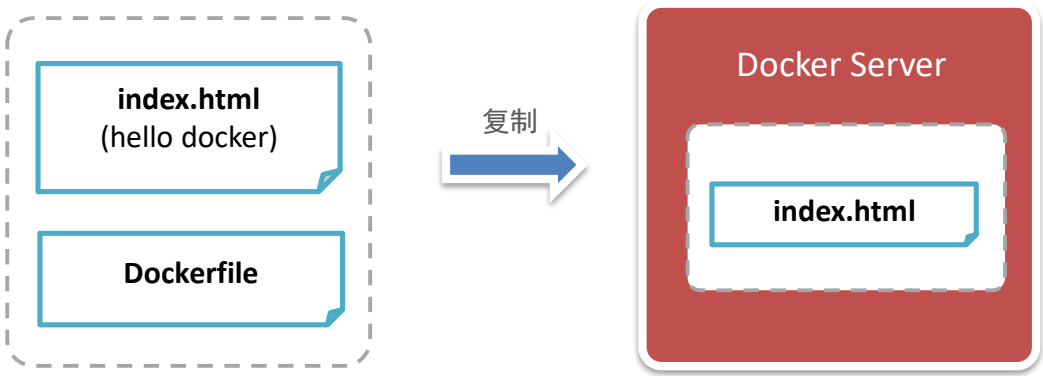



## 重难点

1. 镜像构建的流程
2. Dockerfile 的核心指令

# Dockerfile 构建镜像-总结

## 构建上下文




docker build -t mynginx .

# Dockerfile 构建镜像-总结




## 重难点

1. 镜像构建的流程
2. Dockerfile 的核心指令

# Dockerfile 构建镜像-总结



## 核心指令

- FROM - 指定基础镜像，当前构建的镜像的基石是什么
- COPY/ADD - 拷贝文件或目录到镜像里，添加材料
- RUN - 执行 shell 指令，表明想怎么加工
- WORKDIR - 指定工作位置
- ENV - 设置环境变量
- EXPOSE - 对外开放什么
- CMD/ENTRYPOINT - 容器启动时执行的命令，指明怎么跑起来

# Dockerfile 构建镜像-总结




## 重难点

1. 镜像构建的流程
2. Dockerfile 的核心指令

## 下节

实践 Docker **容器操作**

# 目录 Contents

# Docker 容器操作



## 小节导学

### 实际应用场景

**启动**

命令格式
命令示例
关键选项

**列出**

运行中容器
所有容器

**进入**

登录后台容器

**停止**

命令格式
命令示例
启动停止容器

**删除**

删除某个容器
删除所有容器
删除所有停止的容器

**导出**

容器导出
容器导入

# 1. 启动容器



命令格式:

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

命令示例:

```
> docker run ubuntu:16.04 /bin/echo 'Hello world'
Hello world

> docker run -t -i ubuntu:16.04 /bin/bash
root@b6bb71fb6ef4:/#
```

# 1. 启动容器



## 关键选项

| 选项名称 | 含义 |
| --- | --- |
| -d | 后台运行 |
| -p | 映射宿主机与容器端口 |
| -v | 挂载数据卷，映射宿主机与容器目录 |
| -i | 让容器的标准输入保持打开 |
| -t | 让Docker分配一个伪终端，并绑定到容器的标准输入上 |
| -e | 设置环境变量 |
| --name | 设置容器名 |

# 1. 启动容器



## 命令示例

```
> docker run --name some-redis -p 6379:6379 -d redis

> docker run -p 3306:3306 --name mysql5.7 \
    -v /data/mysqldata:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=123456 \
    -d mysql:5.7

> docker run -t -i ubuntu:16.04 /bin/bash
root@b6bb71fb6ef4:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root
```

# 2. 列出容器



列出当前正在运行的容器：

```
> docker ps
```

显示列：

```
CONTAINER ID   IMAGE     COMMAND     CREATED     STATUS      PORTS       NAMES
```

列出正在运行与停止的容器：

```
> docker ps -a
```

# 3. 停止容器



停止容器的命令：

```
> docker stop CONTAINER_ID
```

命令示例：

```
# 容器列表
> docker ps

# 停止某个容器
> docker stop 23a
```

启动某个停止状态的容器：

```
> docker start 23a
```

# 4. 进入容器



容器通常在后台运行，有时我们希望进入容器做一些操作。

例如后台运行了MySQL 容器，现在想进入容器，并登陆MySQL客户端：

```
# 启动 MySQL 容器
> docker run -p 3306:3306 --name mysql5.7 \
    -v /data/mysqldata:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=123456 \
    -d mysql:5.7

# 进入 MySQL 容器
> docker exec -it mysql5.7 bash

# 登录 MySQL
> mysql -u root -p
```

# 5. 删除容器



删除一个处于终止状态的容器:

```
> docker rm 3ab
```

删除所有终止状态的容器:

```
> docker container prune
```

删除所有容器:

```
# 先停止所有容器
$ docker stop $(docker ps -a -q)
# 删除
$ docker rm $(docker ps -a -q)
```

批量删除部分容器:

```
> docker stop $(docker ps | grep rock | awk '{print $1}')
> docker rm $(docker ps -a| grep rock | awk '{print $1}')
```

# 6. 导入导出容器



例如想把容器发给别人，就需要先把容器导出来：

```
> docker export 7691a814370e > ubuntu.tar
```

别人收到后，导入容器：

```
> docker load < ubuntu.tar
```

# Docker 容器操作-总结




## 重难点

1. 启动容器命令用法
2. 列出容器命令用法
3. 停止容器命令用法
4. 进入容器命令用法
5. 删除容器命令用法
6. 导入导出容器命令用法

# Docker 容器操作-总结



## 命令示例

```
> docker run -p 3306:3306 --name mysql5.7 \
    -v /data/mysqldata:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=123456 \
    -d mysql:5.7
```

# Docker 容器操作-总结




## 重难点

1. 启动容器命令用法
2. 列出容器命令用法
3. 停止容器命令用法
4. 进入容器命令用法
5. 删除容器命令用法
6. 导入导出容器命令用法

## 下节

如果你有**一组容器**需要启动，

如何**方便的管理**?

**Docker Compose**

# 目录 Contents

# Docker Compose 容器编排



## 小节导学

支撑一套服务应用环境通常需要启动多个容器，例如 Tomcat 容器、MySQL 容器、Redis 容器 ……

挨个启动、停止，繁琐、易出错，

如果可以一键启动、停止应用相关的整套容器，是不是很方便。

Docker Compose 就是实现此类需求的容器编排脚本。

Docker Compose 适用于开发环境，和测试环境


- Docker Compose 应用
- Docker Compose 核心指令

# 1. Docker Compose 应用



## 安装

docker compose 下载地址：

```
https://github.com/docker/compose/releases
```

安装命令：

```
# 下载
> curl -L https://github.com/docker/compose/releases/download/1.25.4/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

# 授权
> chmod +x /usr/local/bin/docker-compose
```

# 1. Docker Compose 应用



## 应用实例

docker-compose.yaml：

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: redis
```

启动：

```
docker-compose up -d
```

查看启动日志：

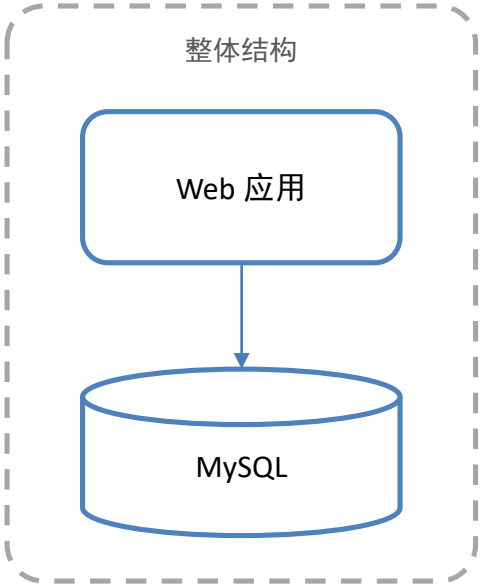
```
docker-compose logs
```

停止：

```
docker-compose down
```

# 1. Docker Compose 应用

## 实践流程




编排流程

**MySQL 容器**

- 指定MySQL镜像
- 设置环境变量
- 执行数据库初始化脚本

**Web应用容器**

- 构建Web应用镜像并启动
- 开放端口
- 需要提前启动 MySQL 容器
- 连接MySQL容器

# 2. Docker Compose 核心指令



## Services

此标签中包含了所有需要启动的容器，是容器的父标签。

```
services:
  proxy:
    build: ./proxy
  app:
    build: ./app
  db:
    image: postgres
```

# 2. Docker Compose 核心指令



## image

指定容器使用的镜像，可以是镜像仓库中的基础镜像，也可以是基于 dockerfile 构建。

```
version: '3.3'

services:
    alpine:
        image: alpine:latest # 使用基础镜像
        stdin_open: true
        tty: true
        command: sh
```

```
version: '3.3'
services:
    app:
        container_name: website
        restart: always
        build: . # 自己构建
        ports:
            - '3000:3000'
    command:
        - 'npm run start'
```

# 2. Docker Compose 核心指令



## ports

暴露端口。

方式1：暴露给关联的其他容器，不暴露给主机

```
expose:
  - "3000"
  - "8000"
```

方式2：暴露给主机

```
ports:
   - "8000:80"  # host:container
```

# 2. Docker Compose 核心指令



## command

用于容器启动后执行命令，可以作为 dockerfile 中 CMD 命令的替换。

```
app:
        container_name: website
        restart: always
        build: ./
        ports:
            - '3000:3000'
        command:
            - 'npm run start'
```

# 2. Docker Compose 核心指令



## Volumes

volume 用于容器持久化数据，数据存放在主机上，由容器来管理，可以作为主机与容器间的共享目录。

容器的生命周期结束后，存放在主机上的数据还在，下次启动容器时，只要再次挂载这个 volume，就可以继续使用。

```
volumes:
  - /opt/data:/var/lib/mysql
```

# 2. Docker Compose 核心指令



## Dependencies

用于定义服务的启动顺序，例如一个内容管理系统 CMS，不能没有数据库 db，所以，可以指明 CMS 依赖 db，就会先启动 db。

```
ghost:
        image: ghost
        ports:
            - 2368:2368
        depends_on: [db]
        ...
    db:
        image: mysql
        ...
```

# 2. Docker Compose 核心指令



## Environment

用于为服务配置数据。

方式1：key-value 设置

```
web:
  environment:
    - NODE_ENV=production
```

方式2：引用 compose 文件中设置的变量

```
web:
  environment:
    - NODE_ENV
```

方式3：使用 .env 文件

```
web:
  env_file:
    - variables.env
```

# 2. Docker Compose 核心指令



## network

自定义网络，同一个网络中的容器之间可以通过主机名沟通

```
  backend:
    networks:
      - react-spring
      - spring-mysql
  db:
    image: mysql:8.0.19
    networks:
      - spring-mysql
  frontend:
    networks:
      - react-spring
networks:
  react-spring: {}
  spring-mysql: {}
```

# 2. Docker Compose 核心指令



## links

同一网络中的容器已经可以互通，通过`link`可以设置一个别名，这样就不依赖与其他容器的名字。

```
version: "3"
services:
    web:
        build: .
        links:
            - "db:database"
    db:
        image: mongo
```

web 容器可以通过 “db” 或者 “database” 都可以访问数据库。

# Docker Compose 容器编排-总结




## 重难点

1. docker compose 基本用法
2. docker compose 的核心指令

# Docker Compose 容器编排-总结


## 重难点

1. docker compose 基本用法
2. docker compose 的核心指令

## 下节

本章总结


一样的在线教育，不一样的教学品质