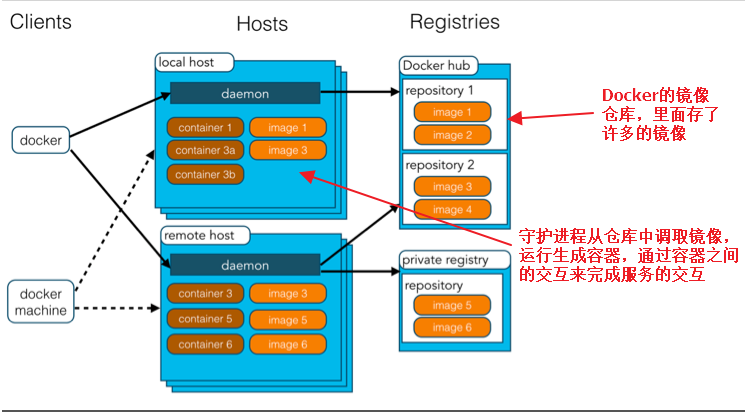
# Docker入门文档

## Docker是什么

* Docker是一个轻量级，可移植的容器，可以用于Web应用的自动打包和发布，自动化测试和集成，在服务型环境中部署调整数据库或后台其他应用，从头编译拓展现有的Open Shift或Cloud Foundry来搭建自己的PaaS环境。
* 说到容器，我们最熟悉的是Spring的容器。Spring是我们项目开发时的代码容器，封装了bean，以IOC和AOP的形式把传统的J2EE进行标准化，简单化。Docker也是一个容器，不过它主要不是为开发代码服务，而是类似于一个虚拟机，在里面可以部署软件系统

## Docker架构

* **镜像（Image）**：Docker 镜像（Image），就相当于是一个 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的安装

* Docker 的旧版本被称为 docker，docker.io 或 docker-engine 。如果已安装，请卸载它们：

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

* 使用Docker仓库进行安装（方式一）
  + 更新 apt 包索引

$ sudo apt-get update

* + 安装 apt 依赖包，用于通过HTTPS来获取仓库:

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

* + 添加 Docker 的官方 GPG 密钥：

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

* + 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88 通过搜索指纹的后8个字符，验证您现在是否拥有带有指纹的密钥

$ **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**]**

* + 使用以下指令设置稳定版仓库

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

* + 安装 Docker Engine-Community

$ sudo apt-get update

* + 安装最新版本的 Docker Engine-Community 和 containerd ，或者转到下一步安装特定版本：

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

* + 要安装特定版本的 Docker Engine-Community，请在仓库中列出可用版本，然后选择一种安装。列出您的仓库中可用的版本：

$ **apt-cache** madison docker-ce

* + 使用第二列中的版本字符串安装特定版本，例如 5:18.09.1~3-0~ubuntu-xenial

$ sudo apt-get install docker-ce=<VERSION\_STRING> docker-ce-cli=<VERSION\_STRING> containerd.io

* + 测试 Docker 是否安装成功

$ **sudo** docker run hello-world

## 容器的使用

* + 查询仓库内的所有容器信息：

~$ docker ps

//查询最后一次创建的容器

~$ docker ps –l

//查询所有容器（包括未启动的容器）

~$ docker ps -a

* + 停止容器

runoob@runoob:~$ docker stop docker容器的CONTAINER ID

或

runoob@runoob: ~$ docker stop docker容器的NAME

* + 进入容器（在后台启动时，默认不会进入容器）
    - Attach(使用这个命令进入，如果推出容器，会导致容器运行停止)

docker attach 容器ID

* + - Exec(使用这个命令进入，退出容器不会导致容器的停止)

docker exec -it 243c32535da7 /bin/bash

* 查询docker的所有命令以及其支持的选项

~# docker

//下面是了解命令的详情

~# docker命令--help

* 通过镜像启动一个容器

$ docker run -it ubuntu /bin/bash

* 导入与导出容器
  + 导出容器快照

docker export 容器ID > ubuntu.tar

* + 导入容器快照（将快照文件 ubuntu.tar 导入到镜像 test/ubuntu:v1）

cat docker/ubuntu.tar | docker import - test/ubuntu:v1

* + 通过UTL或某个目录来导入

docker import http://example.com/exampleimage.tgz example/imagerepo

//我们就可以通过访问 127.0.0.1:5001 来访问容器的 5000 端口

docker run -d -p 127.0.0.1:5001:5000 training/webapp python app.py

* 删除容器

docker rm -f 1e560fca3906

//下命令可以删除docker中所有终止状态的容器

docker container prune

## 构建一个Web应用的容器

这里以一个Python Flask应用为例

* 运行一个web镜像

docker pull training/webapp # 载入镜像

docker run -d -P training/webapp python app.py # 启动容器

**-P:**将容器内部使用的网络端口映射到我们使用的主机上

* 查看WEB应用容器

docker ps

* 查看容器和实体机的端口映射

docker port 容器ID

* 查看Web应用程序的访问日志

docker logs -f bf08b7f2cd89

**-f:** 让 **docker logs** 像使用 **tail -f** 一样来输出容器内部的标准输出

* 查看容器内部的运行进程信息

docker top 容器ID

* 检查Web应用容器

docker inspect 容器ID

* 停止Web应用容器

docker stop 容器ID

* 移除Web应用容器（删除容器时，容器必须是停止状态）

docker rm 容器名

## 镜像使用

* 列出镜像列表

docker images

* 拉取镜像

docker pull ubuntu:13.10

* 通过镜像运行容器命令：docker run 容器 启动的容器中执行的命令

~$ docker run ubuntu:15.10 /bin/echo "Hello world"

以上命令完整的意思可以解释为：Docker 以 ubuntu15.10 镜像创建一个新容器，然后在容器里执行 bin/echo "Hello world"，然后输出结果。

* 运行容器（交互模式）：运行后进入我们所创建的新容器内，我们可以使用exit或者CTRL + D来退出容器

runoob@runoob:~$ docker run -i -t ubuntu:15.10 /bin/bash

* 运行容器（后台模式）：

~$ docker run -d ubuntu:15.10 /bin/sh -c "while true; do echo hello world; sleep 1; done"

* 查找镜像

docker search 镜像名

* 删除镜像

docker rmi hello-world

* 创建镜像
  + 从已经创建的容器中更新镜像，并且提交这个镜像
  + 使用 Dockerfile 指令来创建一个新的镜像：docker build –t 目标镜像名 Dockerfile的所在的目录
* 更新镜像
  + 使用该镜像运行一个容器
  + 在容器内部使用**apt-get update**
  + **退出容器**
  + 提交我们刚刚更新的容器 docker commit –m=”提交信息描述” –a=”镜像作者” 容器ID 要创建的奖项名（例如：runoob/ubuntu:v2）

## Docker远程仓库

* 登录Docker Hub

docker login

* 查询Doker Hub的镜像仓库

docker search 仓库名

* 从Doker Hub上拉取仓库

docker pull 仓库名

* 将本地的仓库推送到Docker Hub上

$ docker tag ubuntu:18.04 username/ubuntu:18.04

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED ...

ubuntu 18.04 275d79972a86 6 days ago ...

username/ubuntu 18.04 275d79972a86 6 days ago ...

$ docker push username/ubuntu:18.04

$ docker search username/ubuntu

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

username/ubuntu

## Dockerfile

* 使用Dockerfile来构建Docker的流程
  + 在一个空目录下面新建一个名为Dockerfile的文件
    - FROM：定制的镜像都是基于FROM的镜像，这里是基础镜像，后续镜像都是基于此镜像
    - RUN：用于执行后面跟着的命令行命令，可以使用shell格式或exec格式（Dockerfile每执行一层，就会建立一个新的层，所以应该尽量让RUN命令少，多个命令可以用&&符号连接例如：命令1 \ && 命令2 \ && 命令3）
  + 构建镜像：docker build –t 镜像名:TAG 上下文路径

上下文路径：因为Docker是c/s结构，我们主机是c，Docker是s，所以在进行构建时，是无法使用我们本机的文件的，所以需要把一些要提供给Docker引擎的文件打包发送。如果不指定上下文路径，默认使用Dockerfile文件所在的目录

* 指令详解
  + COPY：从上下文目录中的文件或者目录复制到容器的指定路径中（上下文路径是本机路径，具体参考上面，目标路径是容器里面的路径）

COPY [--chown=<user>:<group>] <源路径1>... <目标路径>

COPY [--chown=<user>:<group>] ["<源路径1>",... "<目标路径>"]

* + ADD：用法与COPY几乎一致，与COPY的区别是如果上下文路径是压缩文件，用了ADD命令到容器里面会自动解压
  + CMD：类似于RUN命令，不过两者运行的时间点不同，RUN命令运行在build时，CMD命令运行在run时。CMD运行时的命令会被run时指定的参数覆盖，如果Dockerfile里面有多个CMD命令，只有最后一个CMD命令会生效
  + ENTRYPOINT：类似于CMD命令，但是不会被run时运行的参数覆盖。不过run命令后面指定了--entrypoint，这后面的参数会覆盖ENTRYPOINT命令，多个ENTRYPOINT命令运行，只有最后一个生效

这里是CMD和ENTRYPOINT的一个小例子：

通过下面Dockerfile生成了nginx:test的镜像

FROM nginx

ENTRYPOINT ["nginx", "-c"] # 定参

CMD ["/etc/nginx/nginx.conf"] # 变参

不传参运行

docker run nginx:test

容器会默认使用CMD参数，执行下面命令

nginx -c /etc/nginx/nginx.conf

传参运行

docker run nginx:test -c /etc/nginx/new.conf

容器会执行下面命令

nginx -c /etc/nginx/new.conf

* + ENV：设置环境变量（如果设置了a=1，后续的指令可以直接使用$a）

ENV <key> <value>

ENV <key1>=<value1> <key2>=<value2>...

* + ARG：类似ENV，但是二者的作用域不一样，ARG只对Dockerfile文件内有效，构建好的镜像启动之后此变量将无效build命令中的-arg会覆盖ARG的值
  + VOLUME：数据卷。这里的数据卷是解决容器重启时，数据全部消失的问题。这里会在宿主机开辟一块区域用来存贮数据，这样即使是重启容器，数据依然不会消失

VOLUME ["<路径1>", "<路径2>"...]

VOLUME <路径>

* + EXPOSE：声明端口，在使用-P随机映射时，会匹配EXPOSE里面声明的端口

EXPOSE <端口1> [<端口2>...]

* + USER：指定后续命令执行的用户和用户组

USER <用户名>[:<用户组>]

* + ONBUILD：延迟构建，相当于这个命令下面的指令本次构建不会执行，而其他构建的基本构建使用了本次构建（FROM），那么ONBUILD命令的内容才会执行

ONBUILD <其它指令>

## Compose

* compose是用于定义运行于多个Docker应用程序的工具，用yml文件配置应用程序所需要的服务，使用一条命令就可以创建并启动所有的服务
* compose的使用步骤
  + 使用Dockerfile定义应用程序的环境
  + 使用docker-compose.yml定义构成应用程序的服务，这样它们可以隔离环境中一起运行
  + 执行docker-compose up命令启动起来整个应用程序
* compose的安装

sudo curl -L "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

将其文件设置成可执行文件

sudo chmod +x /usr/local/bin/docker-compose

创建软连接

sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

测试是否安装成功

docker-compose --version

* compose的使用
  + 创建一个测试目录

mkdir composetest

cd composetest

* + 创建一个名为app.py文件

在此示例中，redis 是应用程序网络上的 redis 容器的主机名，该主机使用的端口为 6379

在 composetest 目录中创建另一个名为 requirements.txt 的文件，内容如下：

flask

redis