Docker 基础与实践

# 背景

容器技术的兴起源于 PaaS 技术的普及。在以往的 PaaS 项目中，用户会把**应用的可执行文件和启动脚本打进一个压缩包**内，然后大规模部署应用到集群里。但是这个打包的工作却一波三折，费尽心机。用户需要为每种语言、每种框架、每个版本的应用维护一个打好的包，并且要做很多修改和配置工作才能在 PaaS 里运行起来。

而 Docker 镜像功能解决了打包这个根本性问题。**所谓 Docker 镜像（不是Docker容器），其实就是一个压缩包。**但是这个压缩包里的内容，却不止应用的可执行文件和启动脚本。大多数 Docker 镜像是直接由一个完整操作系统的所有文件和目录构成的，所以这个压缩包里的内容跟你本地开发和测试环境用的操作系统是完全一样的。

**这种打包机制直接打包了应用运行所需要的整个操作系统，所需要的所有依赖，从而保证了本地环境和云端环境的高度一致**，避免了用户通过“试错”来匹配两种不同运行环境之间差异的痛苦过程。

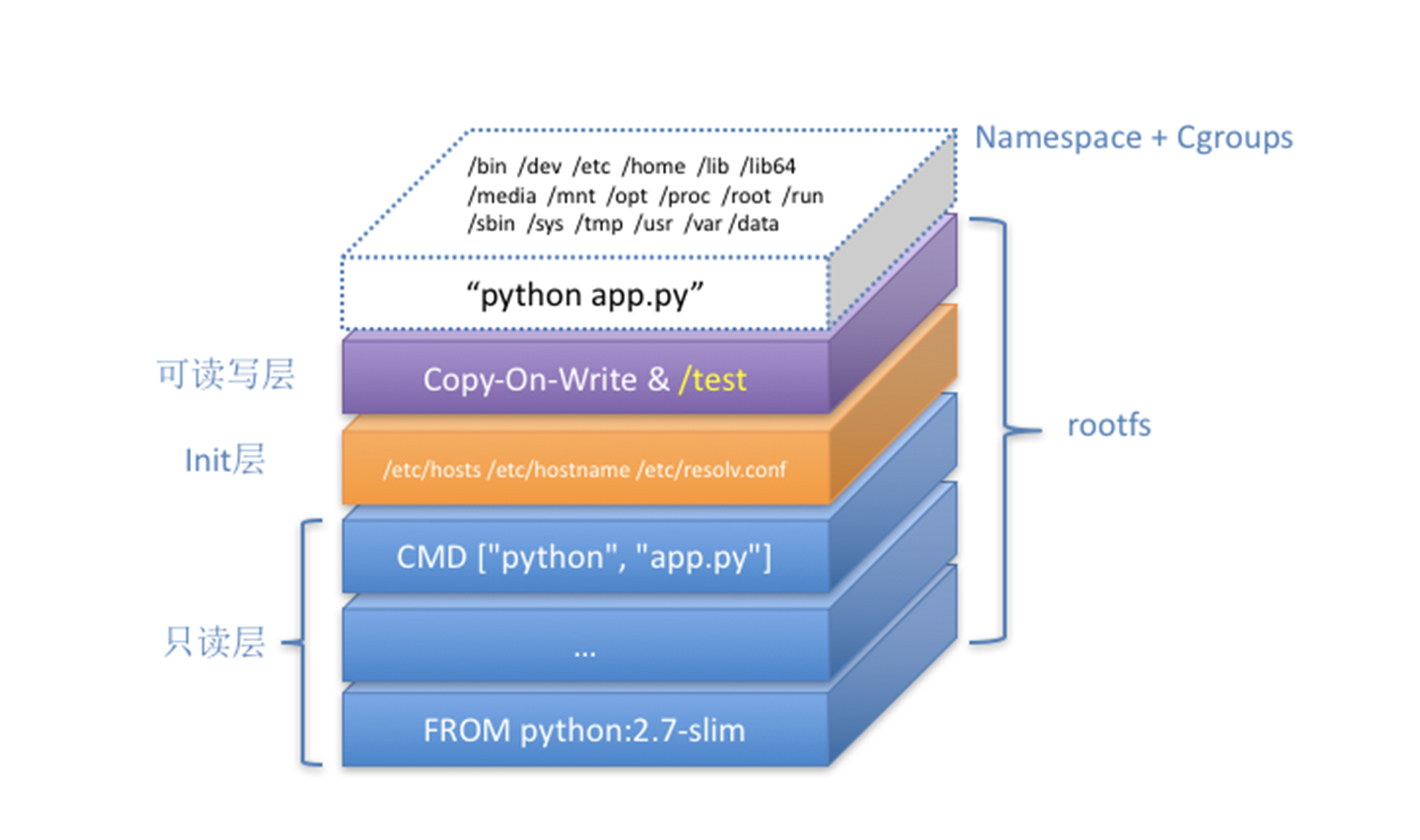
# 基本概念

## Docker 容器是什么？

容器其实是一种沙盒技术。顾名思义，沙盒就是能够像一个集装箱一样，把你的应用“装”起来的技术。这样，应用自己运行在一个独立空间中；应用与应用之间，就因为有了边界而不至于相互干扰；而被装进集装箱的应用，也可以被方便地搬来搬去。

**Docker 容器的本质是一种特殊的进程**。这个进程是应用所有进程的父进程。容器技术通过**环境隔离**和**资源限制**，从而为进程创造出一个沙盒。 **Namespace** 技术用来实现环境隔离，**Cgroups** 技术用来实现资源限制， **rootfs** 机制（也经常被称为容器镜像）为进程提供了文件系统。

形象地说，**我们可以把创造沙盒比作造房子，把进程作为生活在屋子里的人**。**Cgroups** 规定了这个房子的面积，这个人能使用多少资源（CPU、内存、网络带宽等）。**Namespace** 建造了这个房子的四周墙壁和屋顶，把人与外界环境隔离开来。**rootfs** 机制建造了这个房子的地下室， 存放着生活必需用品（应用所需文件，应用运行所需要的依赖和整个操作系统）。



## Docker 容器 与 Docker 镜像的区别

Docker容器是进程，Docker镜像是压缩包。

Docker容器通过Docker镜像来创建。

Docker镜像是Docker生命周期的构建和打包阶段。

Docker容器是Docker生命周期的启动和执行阶段。

## Namespace

Linux提供了六种 namespace，用于不同环境的隔离，分别是：**PID， Mount，UTS，IPC，Network，User。**

Namespace 实际上是一种“障眼法”。比如，在某一个PID Namespace中的应用进程，看不到宿主机里真正的进程空间，也看不到其他 PID Namespace 里的具体情况，所以他会认为自己是当前容器里的第1号进程。

Mount Namespace，用于让被隔离进程只看到当前 Namespace 里的挂载点信息；Network Namespace，用于让被隔离进程看到当前 Namespace 里的网络设备和配置。

## Cgroups

虽然进程被隔离了起来，但是它与所有其他的进程之间依然是平等的竞争关系。它自己能使用的资源（CPU，内存）随时会被其他进程（或者其他容器）占用，他自己也可能吃光资源，所以需要进行资源限制。

**Linux Cgroups** 的全称是 Linux Control Group。**它最主要的作用，就是限制一个进程组能够使用的资源上限，包括 CPU、内存、磁盘、网络带宽等等。**

## rootfs

rootfs 是用来辅助 Mount Namespace 进行文件系统的环境隔离的。rootfs 就是挂载在容器根目录上、用来为容器进程提供隔离后执行环境的文件系统。常见的 rootfs会包含 /bin， /dev, /etc, /home, /lib等等。

因为 Mount Namespace 修改的，是容器进程对文件系统“挂载点”的认知。但是，这也就意味着，**只有在“挂载”这个操作发生之后**（如果不进行挂载，容器进程看见的文件系统和宿主机是完全一样的），进程的视图才会被改变。所以，Mount Namespace 需要与 rootfs 协同运作才能为进程构建出一个完善的文件系统隔离环境。

为了便于维护和能够复用已有的基础容器镜像，Docker 公司做了个小创新，将 rootfs 变成了**增量 rootfs**（这样，用户只需要维护相对于基础镜像修改的增量内容），所以一层的地下室变成了多层地下室，每一层都包含着应用所需要的一部分文件。（原来是一个大地下室，如果出问题了，需要维修整个地下室。但是现在把大地下室拆分成了多层小地下室，现在只需要维修特定的一层小地下室就可以了。）在用户制作镜像时，每一步操作，都会生成一个层，也就是一个增量 rootfs。

## Docker 组成架构

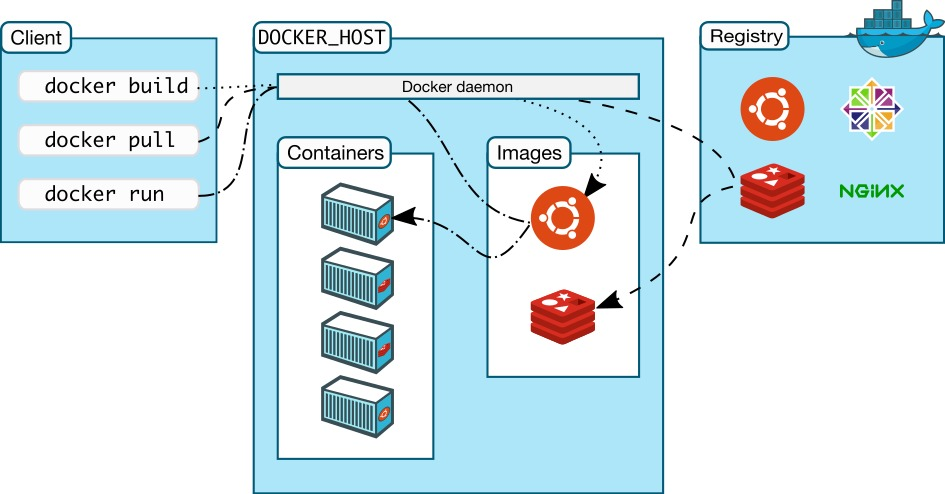
Docker 使用 C/S 架构，Client 通过接口与 Server 进程通信实现容器的构建，运行和发布。client和server可以运行在同一台主机，也可以通过跨主机实现远程通信。

Docker由五个部分组成：**Docker Client 客户端、Docker Daemon守护进程、Docker Registry仓库、Docker Container容器**和 **Docker Iamge镜像**。

Docker Client 负责向服务端（Docker Daemon守护进程）发起请求。

Docker Registry 负责存储 Docker镜像。

Docker Daemon 负责 从 Docker Registry 下载 Docker镜像 和 通过 Docker镜像 启动 Docker 容器。



一个 Docker 容器的启动过程（即 Clinet 运行 docker run 命令后）：

1. Docker daemon 接收到 Client请求；
2. 优先查找本地镜像，如果本地没有，Docker daemon 会从远端镜像仓库拉取所需镜像；
3. 启动 Linux Namespace 配置，设置指定的 Cgroups 参数，挂载 rootfs（容器镜像），切换进程的根目录；
4. 容器运行

# 简单实践

接下来将通过一个简单的例子，让大家熟悉 docker 的基本命令和常用功能。

## 实践环境

可以通过 docker info 查看相关环境信息，摘要如下：

Kernel Version: 3.10.0-957.5.1.el7.x86\_64

Docker Version: 17.12.1-ce

Operating System: CentOS Linux 7 (Core)

## 实践过程

### 准备 Demo 文件

编写两个简单文件 hello.cpp 和 test.sh 如下，编译得到可执行文件 hello

// hello.cpp

#include <iostream>

using namespace std;

int main(){

cout << "hello docker" << endl;

return 0;

}

# test.sh

#! /bin/sh

echo "hello docker sh"

$ g++ -o hello hello.cpp

### 编写 Dockerfile

回忆上文对于 Docker 镜像层次结构和镜像生成方式的介绍，同时准备如下 Dockerfile

# Dockerfile

From ubuntu # 以ubuntu为基础镜像

RUN mkdir -p /root/test # 执行递归创建目录命令

COPY hello test.sh /root/test/ # 将本地二进制和脚本拷贝到镜像中

ENV NAME world # 设置必要的环境变量

注意：将本地Dockerfile所在目录下的 hello 和 test.sh 拷贝到镜像的 /root/test下，Dockerfile中进行**多文件 COPY 时，目的地址如果是目录，必须加上 ' / '**。

### 构建镜像 docker build

查看一下准备好的原料

[root@tdc-tester01 dockertest]# tree

.

├── Dockerfile

├── hello # 编译出的可执行文件

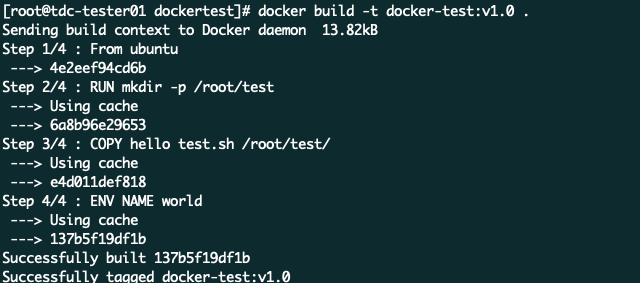
├── hello.cpp

└── test.sh

docker build 会默认查找当前目录下 **Dockerfile**，按其中定义的规则进行打包，运行如下命令

$ docker build -t docker-test:v1.0 .

**-t** 代表以“ **name:tag** ”的形式指定生成镜像名，最后的 **' . '** 代表指定当前目录。此外，常用的参数还有 **-f** 来指定不同路径的 Dockerfile，以达到同目录下构建多份不同镜像的目的。



**docker build 运行状态**

上图可以清晰看出 docker build 镜像时的步骤，每执行一句原语都会生成一个对应的镜像层。

docker build 操作完成后，可以通过 docker images 命令查看结果：

$ docker images | grep docker-test

docker-test v1.0 137b5f19df1b 2 hours ago 73.9MB

### 运行容器 docker run

$ docker run -it docker-test:v1.0 /bin/bash

root@c721e1346bbc:/#

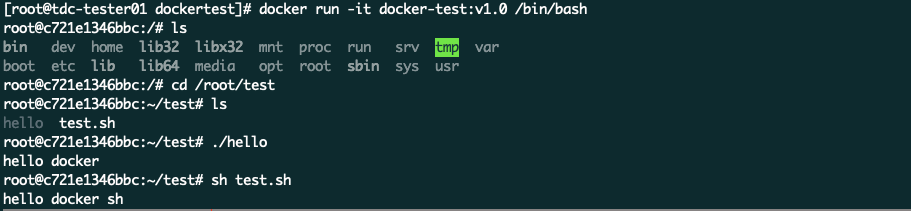
root@c721e1346bbc:/# ls

bin dev home lib32 libx32 mnt proc run srv tmp var

boot etc lib lib64 media opt root sbin sys usr

运行如下命令，其中 **-it** 代表以交互式运行该镜像，指定镜像 **name: tag** 。

进入容器内部后，可以观察到他在根目录下有一套新的文件系统，上文在 Dockerfile 中我们新建了 **/root/test/** 这里我们进入后可以观察到里面有 **COPY** 进来的二进制 hello 和 test.sh，运行符合预期。



**docker run 后查看容器内部**

此外，同时可以另开一个窗口运行 **docker ps** 可以查看到正在运行的 docker 进程

$ docker ps | grep docker-test

c721e1346bbc docker-test:v1.0 "/bin/bash" 3 minutes ago Up 3 minutes xenodochial\_nightingale

### 上传镜像 docker push

从上文的 docker 原理中可知，docker run 的过程中会先拉取本地镜像，如果没有就会去注册的 registry 找同名 name: tag ，因此需要先配置好公司内部的镜像 registry，便于后期对镜像的管理。

公司内部的 Harbor 镜像仓库配置方法如下链接，申请权限完成本地配置后 docker login 即可往该仓库内 push image

**http://172.16.1.168:8090/display/QA/Inner+Registry+Of+Harbor?focusedCommentId=24598468#comment-24598468**

$ docker tag docker-test:v1.0 172.16.1.99/tmp/docker-test:v1.0

$ docker push 172.16.1.99/tmp/docker-test:v1.0

这里先重新打 tag 为 172.16.1.99/tmp/docker-test:v1.0 ，因为是测试的例子，所以 push 到registry 的 tmp 目录下，其他专有目录权限需要根据账号而定。



**docker tag 和 docker push**

注意这里的 **sha256: fa163db8 ...** ，是该镜像的 hash 值，可以根据这个来判断同名镜像是不是我们修改的那个版本。

### 验证镜像 docker pull

制作的镜像已经上传到 registry，下面我们测试一下在本地没有该镜像的情况下能不能顺利拉取到该镜像，并启动容器。

$ docker images | grep docker-test

172.16.1.99/tmp/docker-test v1.0 137b5f19df1b .....

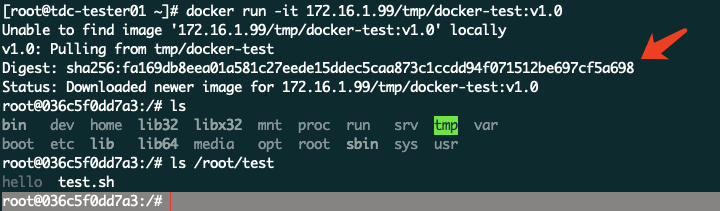
docker-test v1.0 137b5f19df1b .....

$ docker rmi 172.16.1.99/tmp/docker-test:v1.0

把本地的镜像先删除，然后运行docker

$ docker run -it 172.16.1.99/tmp/docker-test:v1.0 /bin/bash

docker run 命令会先检查本地是否有该镜像，没有则执行 docker pull 从 registry 拉取镜像。



**pull image from registry**

核对 hash 值是否与上文 push 时一致，这里也是 **sha256: fa163db8 ...** , 验证镜像符合预期**。**

# 深入探索

基于上面的介绍，我们对于 Docker 有了直观的认识，但想必同学们对其背后的实现还是很感兴趣的。提几个问题大家可以思考一下。

1. **同一主机内的容器间是怎么通信的？不同主机间的容器呢？**
2. **为什么有时候把旧镜像里的东西删了，镜像也没有变小呢？**

**......**

接下来我们将以问答、实验的形式帮助大家深入理解 Docker，感兴趣可以继续学习。

# 参考资料

<https://docs.docker.com/engine/>

<https://kubernetes.io/zh/docs/home/>

[10张图带你深入理解Docker容器和镜像](http://merrigrove.blogspot.com/2015/10/visualizing-docker-containers-and-images.html)

《深入剖析 Kubernetes 》张磊（好评！）