# Docker 学习总结

## Windows10 docker 安装：

1.1 右键点击windows图标,选择应用和功能；

1.2 进入页面后，找到程序和功能并点击；

1.3 进入复选框后，选择 Hyper-V,打开后将所有的内容进行勾选；

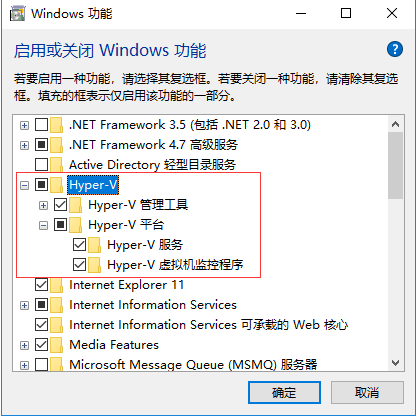


图1.1

1.4 关闭复选框；

1.5 下载 docker， 下载地址：[https://www.docker.com/get-docker](https://www.docker.com/get-docker" \t "https://www.runoob.com/docker/_blank)

1.6 选择 Download Desktop and Take a Tutorial, 并下载Windows

版本（如果没有登录，需要注册登录）；

1.7 下载完成后，会有docker for windows install.exe(类似形式)的安

装包出现，点击安装即可；

1.8 安装完成后，docker 会自动运行，有鲸鱼图标。 可在cmd终端

运行docker version 查看docker版本号，

1.9 在终端运行docker run hello-word 来载入测试镜像测试（如果本地没有hello-world会自动下载一个hello-world）。

详情链接:<https://www.runoob.com/docker/windows-docker-install.html>

## 1.a Centos 安装docker

Docker 支持以下的CentOS版本：

CentOS 7 (64-BIT)

CentOS 6.5 (64-bit) 或更高的版本

1.a.1 前提条件

目前，CentOS仅发行版本中的内核支持Docker

Docker 运行在CentOS 7 上，要求系统为64位，系统内核版本要求在3.10 以上

Docker 运行在 CentOS-6.5 或更高版本的CentOS上，要求系统为64位，系统内核 版本为 2.6.32-431 或者更高版本。

1. a.2 使用yum 安装（CentOS7）下

通过 uname -r 命令查看你当前的内核版本（window10 自带的linux）



1.a.3 安装Docker

1.a.3.1 通过下面命令移除旧版本：

$ 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

1. a.3.2

安装一些必要的系统工具：



yum-utils: 功能： 管理repository及扩展包工具

device-mapper-persistent-data 设备映射持久化数据的存储

1. a.3.3

安转docker

yum install -y docker (加上-y参数便不会进行提示)

1. a.3.4

启动Docker后台服务

sudo systemctl start docker

1. a.3.5

测试运行hello-world

docker run hello-world

## Docker Hello-World

2.1 Docker Hello World

Docker 允许你在容器内运行程序，使用docker run 命令来在容器内运行

第一个应用程序。

在cmd 终端中运行命令：

Docker run ubuntu:15.10 /bin/echo “Hello world”

参数解析：

docker : Docker 的二进制执行文件

run : 与前面的docker组合起来运行一个容器

ubuntu:15.10 : 指定要运行的镜像，Docker首先从本地主机查找镜像

是否存在，如果不存在就从docker hub 下载公共镜像。

/bin/echo”Hello world”: 在启动的容器里面执行命令

以上命令完成的意思可以解释为：Docker 以 ubuntu 15.10 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。

2.2 运行交互式的容器

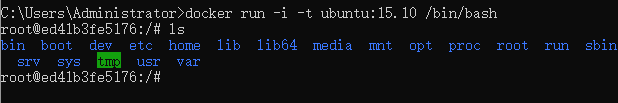
在cmd 终端中（windows系统下），运行命令：

docker run -i -t ubunt:15.10 /bin/bash

参数解析：

-t : 在新容器内指定一个伪终端或终端。

-i : 允许你对容器内的标准输入（stdin）进行交互。



2.3 启动容器（后台模式）

在cmd终端中运行一下命令：



在输出中，我没有看到期望的”hello world”,而是一长串字符（简称str）,这个长字 符串叫做容器的ID,对每个容器来说都是唯一的，我们可以通过容器ID来查看对应 容器内发生了什么。

首先：在cmd 终端中运行 docker ps (与linux ps相似查看进程)



CONTAINER ID: 容器ID

NAMES: 自动分配的容器名称

在容器内使用docker logs 命令，查看容器内的标准输出

使用 容器id查看容器内标准输出



使用 容器name 查看容器内标准输出



2.4 停止容器

在cmd 终端使用docker stop +容器id/容器name 来停止容器

docker stop 4eb7b18744ae/ blissful\_pasteur

命令执行完毕后，可以使用docker ps 来查看容器情况。

## docker 容器使用

3.1 Docker 客户端

我们可以在cmd终端直接输入docker命令来查看Docker客户端的所有命令选项。

可以通过命令 docker command --help 更深入的了解指定的Docker 命令使用方 法。

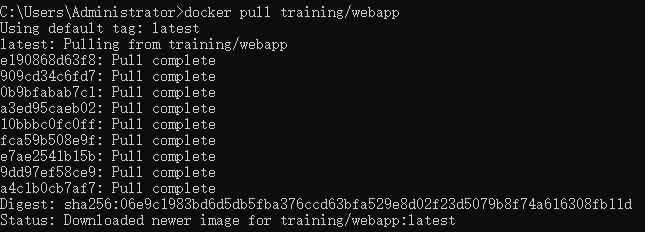
3.2 运行一个web应用

尝试使用docker构建一个web应用程序。

我们将在docker容器中运行一个python flask应用来运行一个web应用。

3.2.1 在cmd 终端执行:

docker pull training/webapp # 载入镜像



3.2.2 在cmd终端执行

Docker run -d -P training/webapp python app.py

参数说明：

-d : 让容器在后台运行。

-P(大写) : 将容器内部使用的网络端口映射到我们使用的主机上。

3.3 查看WEB应用容器

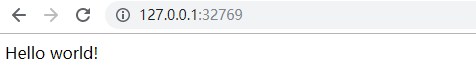
使用 docker ps 来查看我们正在运行的容器



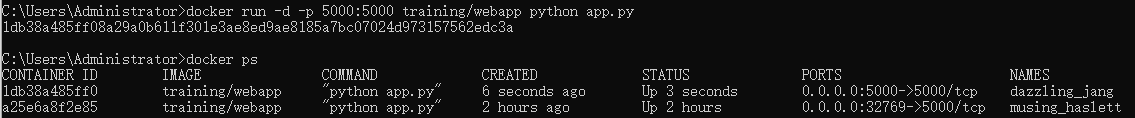
这里多了端口信息：

Docker 开放了5000端口（默认Python Flask 端口）映射到主机端口32769上。

我们可以通过 127.0.0.1:32769来访问flask webapp



我们也可以通过-p 参数来设置不一样的端口



容器内部的5000端口映射到我们本地主机的5000端口上。

3.4 网络端口的快捷方式

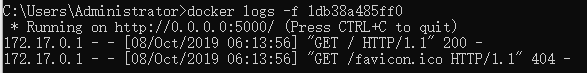
通过docker ps 命令可以查看到容器的端口映射，docker还提供了另一个快捷 方式docker post，使用docker post 可以查看指定（ID或者名字）容器的某个 确定端口映射到宿主机的端口号。

上面我们创建的web应用容器ID为1db38a485ff0名字为 dazzling\_jang

我们可以使用docker port dazzling\_jang 或 docker port 1db38a485ff0 来查看 容器端口映射的情况。

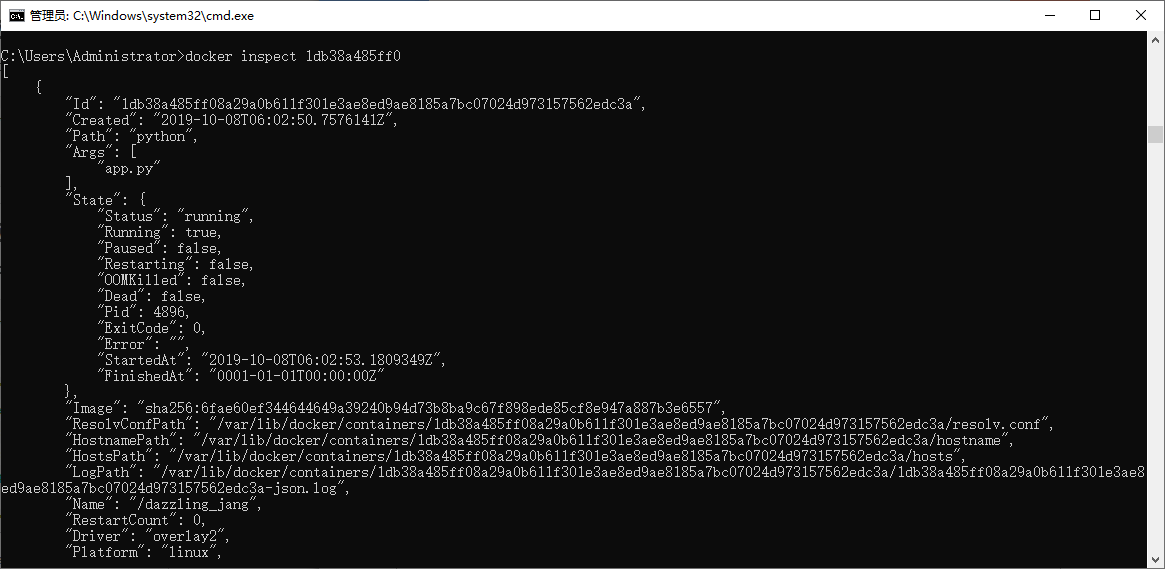
3.5 查看WEB应用程序日志

docker logs [ID或者名字]可以查看容器内部的标准输出。



3.6 检查WEB应用程序

使用docker inspect 来查看Docker 的底层信息。他会返回一个json文件记录 着Docker容器的配置和状态信息。



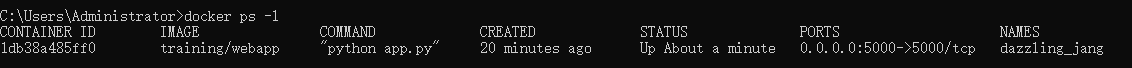
3.7 停止WEB应用容器



3.8 重启WEB应用容器



使用docker ps -l 查询最后一次创建的容器：

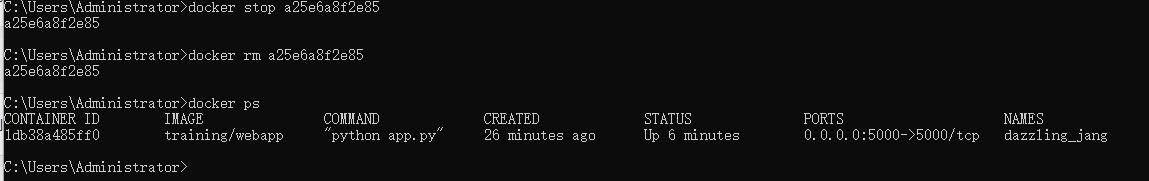


3.9 移除WEB应用容器

我们可以使用docker rm 命令来爱删除不需要的容器

注: 删除容器时，容器必须时停止状态，否则会报以下错误。





## Docker 镜像使用

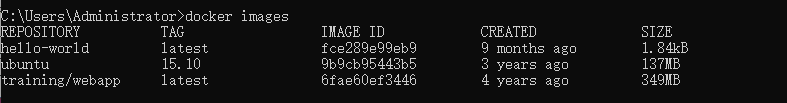
当运行容器时， 使用的镜像如果在本地中不存在，docker就会自动从docker镜像仓库中下载，默认是从Docker Hub 公共镜像源下载。

下面学习：

1. 管理和使用本地Docker 主机镜像
2. 创建镜像

4.1 列出镜像列表

我们可以使用docker images 来列出本地主机上的镜像。



各个选项说明：

REPOSITORY : 表示镜像的仓库源

TAG : 镜像的标签

IMAGE ID : 镜像ID

CREATED : 镜像创建时间

SIZE : 镜像大小

注： 同一个仓库源可以有多个TAG,代表这个仓库源的多个不同的版本，如ubuntu 仓库源里，有15.10 代表了15.10 版本，我们可以使用REPOSITORY:TAG来定义不 同的镜像。

例：当我们使用版本为15.10的ubuntu系统镜像来运行容器时，命令如下:



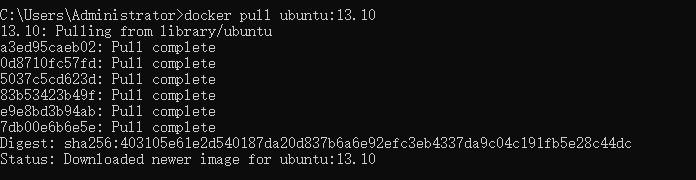
当我们使用版本为14.04 的ubuntu系统镜像来运行容器时，命令如下:



注: 如果你不指定一个镜像的版本标签，列入你只使用ubuntu，docker将默认使用 ubuntu:latest镜像。

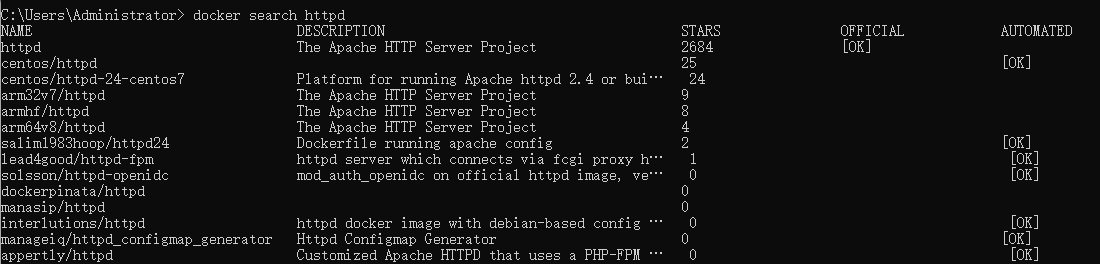
4.2 获取一个新的镜像

当我们在本地主机上使用一个不存在的镜像时Docker就会自动下载这个镜像。如果我们想预先下载这个镜像，以使用docker pull 命令来下载它。



4.3 查找镜像

我们可以从Docker Hub 网站来搜索镜像，DockerHub 网址为：<https://hub.docker.com/;> 我们也可以使用docker search 命令来搜索镜像。比如我们需要一个httpd的镜像来作为我们的web服务。我们可以通过docker search 命令搜索 httpd 来寻找适合我们的镜像。



NAME： 镜像仓库源的名称

DESCRIPTION: 镜像的描述

OFFICIAL: 是否docker官方发布

4.3 拖取镜像

使用上图中httpd官方版本的镜像，使用docker pull 来现在镜像



下载完成后，就可以使用这个镜像了。



4.4 创建镜像

当我们从docker 镜像仓库中下载的镜像不能满足我们的需求时，我们可以通过以下两种方式对镜像进行更改。

1. 从已经创建的容器中更新镜像，并提交这个镜像
2. 使用Dockerfile 指令来创建一个新的镜像

4.5 更新镜像

更新镜像之前，我们需要使用镜像创建一个容器



首先运行容器，然后在运行的容器内使用apt-get update命令进行更新。

在完成操作之后，输入exit命令退出这个容器。

此时ID为d6d54c2e6479的容器，是按照我们需求更改的容器，我们可以通过命令docker commit来提交容器副本。



注： -a与等号之间不能有空格， 双引号不能被替换成单引号。

各个参数说明：

-m : 提交的描述信息

-a : 指定镜像作者

d6d54c2e6479 ： 容器ID

Guangcui/ubuntu:v1 : 指定要船舰的目标镜像名

使用docker images 命令来查看我们的新镜像 guangcui/ubuntu:v2 :



4.6 构建镜像

我们使用命令 docker build， 从零开始来创建一个新的镜像。为此，我们需要创建一个Dockerfile文件，其中包含一组指令来告诉Docker如何构建我们的镜像。

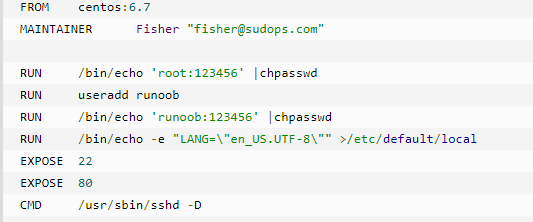
1. Ubuntu 创建 Dockerfile 命令

cat Dockerfile

2. Windows 10 创建 Dockerfile 命令



在本地文件中打开Dockerfile,并输入



每一条指定都会在镜像上创建一个新的层，每一个指令的前缀都必须是大写的。

1. FROM,指定使用哪个镜像源

RUN 指令高速docker在镜像内执行命令，安装了什么。

然后，我们使用Dockerfile文件，通过dockerbuild命令来构建一个镜像。

参数说明：

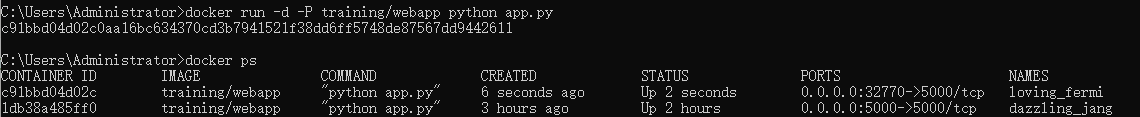
-t : 指定要创建的目标镜像名。

. : Dockerfile 文件所在目录，可以指定Dockerfile 的绝对路径。

## Docker 容器连接

前面我们实现了通过网络端扣来访问运行在docker容器内的服务。下面我们实现通过 一个端口链接到另一个容器

创建一个python 应用的容器。



另外，我们可以指定容器绑定的网络地址，比如绑定127.0.0.1.

我们使用-P 参数创建一个容器。使用docker ps 可以看到容器端口5000绑定主机端口 32770。

我们也可以使用-p表示来指定容器端口绑定主机端口。

两种方式的区别是：

-P: 是容器内部端口随机映射到主机的高端口。

-p: 是容器内部端口绑定到指定的主机端口。



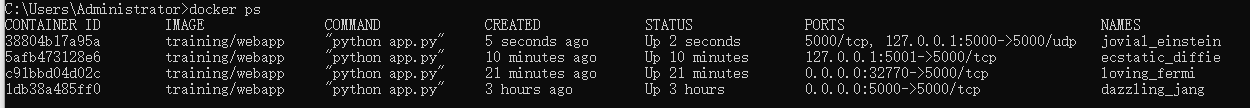
另外，我们可以给指定容器绑定网络地址，比如绑定127.0.0.1。



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

上面的例子中。默认都是绑定tcp端口，如果要绑定UDP端口，可以在端口后面加上/udp。





使用 docker port 命令可以让我们快捷的查看端口的绑定情况。

5.1 Docker 容器连接

端口映射并不是唯一把docker 连接到另一个容器的方法。

docker 有一个连接系统允许将多个容器连接在一起，共享链接信息。

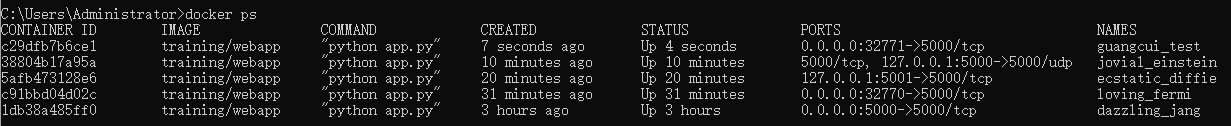
docker 链接会创建一个父子关系，其中父容器可以看懂子容器的信息。

5.2 容器命名

当我们创建一个容器的时候，docker会自动对它进行命名。另外我们也可以使用--name 标识来命名容器。



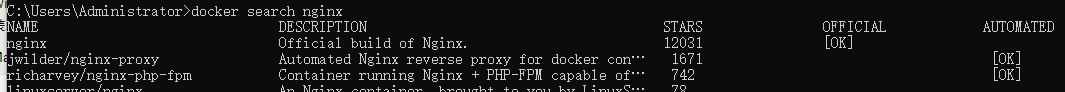
使用docker ps 查看结果



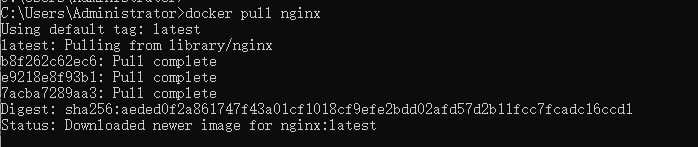
## Docker 安装Nginx

6.1 docker pull nginx 命令安装

查找Docker Hub 上的nginx镜像



拉取官方的镜像



等待下载完成后，我们就可以在本地镜像列表里查到REPOSITORY 为 ngnix的镜像。

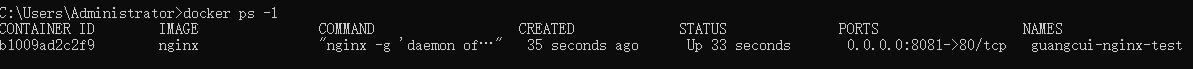
以下命令使用NGINX默认的配置来启动一个Nginx容器实例:



guangcui-ngnix-test : 容器名称。

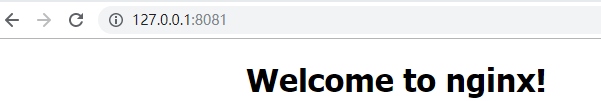
the -d 设置容器在后台一直运行。

the -p 端口进行映射，将本地8081端口映射到容器内部的80端口。



浏览器访问 127.0.0.1:8081

PORTS 部分表示端口映射，本地的8081端口映射到容器内部的80 端口。



<https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-x64.tar.xz>

<https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-x64.tar.xz>

## Test: Docker centos7 安装 Nodejs

7.1 首先 获取centos7 镜像：

docker search centos

7.2 选择版本（官方），拉取镜像

docker pull centos:7

7.3 进入到centos7 容器中（启动centos7）

docker run -t -i -d centos:7 /bin/bash (如果不是用-d参数，会随着前台exit退出，线程也会结束，安装的环境也会随之销毁。)

7.4 为centos 安装 wget

yum install wget

7.5 获取 Nodejs

wget <https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-x64.tar.xz>

7.6 解压安装包

xz -d node-v10.16.3-linux-x64.tar.xz

tar -xf node-v10.16.3-linux-x64.tar

7.7 为nodejs创建软连接

ln -s /node-v10.16.3-linux-x64/bin/node /usr/bin/node

ln -s /node-v10.16.3-linux-x64/bin/npm /usr/bin/npm

7.8 测试安装是否成功：

node -v

7.9 安装 crypto-js 模块

npm install crypto-js

8.0 将后台运行容器提到前台命令：

docker exec -it 容器ID/容器名称 +命令（/bin/bash）

8.1 退出centos命令

Ctrl + p +q

注： 如果容器意外停止，配置的环境会随之消失。

解决办法：

docker inspect 容器ID

docker start 容器ID

## Docker 引擎的升级

升级Docker引擎（Engine）是一项重要的任务，尤其是在生产环境。

下面介绍升级Docker 引擎的关键步骤，以及一些相关的小建议和升级示例。

升级Docker引擎的关键步骤如下：

8.1 需要重视升级操作的每个前置条件，包括确保容器配置了正确的重启策略；在Swarm Mode 模式下使用服务时，需要确保配置了draining node.

当完成了上述前置条件的检查之后，可以通过如下步骤完成升级操作。

8.1.1停止Docker 守护程序。

8.1.2 移除旧版本Docker。

8.1.3 安装新版本Docker。

8.1.4 配置新版本的Docker为开机自启动。

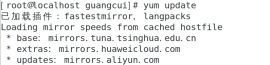
8.1.5 确保容器启动成功。

8.2 在Centos7上升级Docker CE

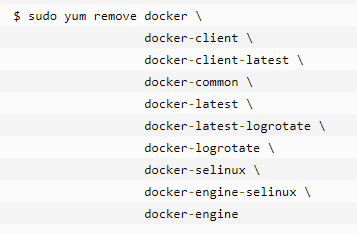
假设已经完成了全部的升级前置步骤并且Docker处于可以升级的状态，同时还可以用root用户身份运行升级命令。

以root用户运行升级命令是不推荐的，但是可以简化示例，如果不采用root用户运行升级命令，那最好不过了。 那么需要通过sudo来执行下列命令

8.2.1更新yum



8.2.2移除旧版本docker



8.2.3安装新版本的Docker

yum install -y docker

8.2.4 将Docker配置为开机自启动。

Systemctl enable docker



此时重启测试自己的节点，确保刚安装的Docker不会对系统开机有任何的影响。

8.2.5 检查并确保每一个容器和服务都已经重启成功。

docker container ls （-all）

列出正在运行的容器,

与docker ps -a 不同的地方在与 docker 会将已经停止的容器也一并列出来

docker service ls (Docker swarm 中有相关介绍)

列出服务，此命令在运行定向管理器时，正在集群中的列表服务。

## Docker驱动

每个Docker容器都有一个本地存储空间，用于保存层叠的镜像（Image layer）以及挂咋的容器文件系统。

默认情况下，容器的所有读写操作都发生在其镜像层上或挂在的文件系统中，所以存储是每个容器的性能和稳定性不可或缺的一个环节。

以往，本地存储是通过存储驱动（Storage Driver）进行管理的，有时候也被称为Graph Driver 或者Graph Driver.

虽然存储驱动在上层抽象设计中都采用了栈式镜像层存储和写实复制（Copy-on-Write）的设计思想方法，但是Docker 在 linux 底层支持几种不同的存储驱动的具体实现，每一种实现方式都采用不同的方法实现了镜像层和写时复制。

虽然底层实现的差异不影响用户与docker之间的交互，但是对docker的性能和稳定性至关重要。

在linux上，Docker可选择的一些存储驱动包括AUFS(最原始的也是最老的)，Overlay2(可能是未来的最佳选择），Device Mapper, Btrfs 和 ZFS。

Docker 在 Windows 操作系统上只支持一种驱动，即Windows Filter.

存储区东的选择是节点级别的，这意味着每个Docker主机只能选择一种存储驱动，而不是为每个容器选择不同的存储驱动。

在 linux 上，读者可以通过修改/etc/docker/daemon.json 文件来修改存储引擎配置，修改完成之后需要重启docker才能够生效。

下面代码片段展示了如何将存储驱动设置为overlay2

{“storage-driver”:”overlay2”} # centos7 docker19 表示没有找到

注： 如果配置所在行不是文件的最后一行，则需要在行尾处增加逗号。

9.1. 如果读者修改了正在运行的Docker 主机的存储引擎类型，则现有的镜像和容器在重启以后将不可用，这是因为每种存储驱动在主机上存储镜像层的内置是不同的（通常在/var/lib/docker/<storage-driver>/... 目录下）。

修改了存储驱动的类型，Docker就无法找到原有的镜像和容器 ，切换到原来的存储驱动，之前的镜像和容器就可以继续使用了。

如果希望在切换存储引擎之后还能够继续使用之前的镜像和容器保存为Docker格式，上传到某个镜像仓库，修改本地Docker存储引擎并重启，之后从镜像仓库将镜像拉取到本地，最后重启容器。

可以通过下面命令，查看Docker当前的存储驱动类型

docker system info

选择存储驱动并正确的配置在Docker环境中是一件重要的事情，特别是在生产环境中。

docker container run -it ubuntu:latest /bin/bash

-it 参数 是告诉Docker开启容器的交互模式并将当前的shell链接到容器终端

docker ps -a

显示所有容器包括未运行的。

docker container exec 命令格式如下：

docker container exec <options> <container-name or container-id> <command/app>

docker container exec -it 容器ID/容器名称 命令（/bin/bash）

执行docker container exec 命令可以将Shell 连接到运行容器的终端

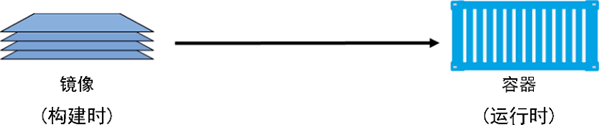
docker container stop 容器ID/name:

用来停止容器

docker container rm 容器ID/name:

用来删除容器

## Docker 镜像



10.1 镜像和容器

上图从顶层设计层面展示了镜像和容器间的关系。通常使用docker container run 和 docker service create 命令从一个镜像启动一个或多个容器。

一旦容器从镜像启动后，二者之间就变成了互相依赖的关系，并且在镜像上启动的容器全部停止之前，镜像是无法被删除的。尝试删除镜像而不停止或销毁使用它的容器，会导致出错。

10.2 镜像通常比较小

容器目的就是运行应用或者服务，这意味着容器的镜像中必须包含应用/服务运行所必须的操作系统和应用文件。

但是容器有追求快速和小乔，这意味着构建镜像的时候通常需要剪裁掉不必要的部分，保持较小的体积。

注： Hyper-V容器运行在专用的轻量级VM上，同时利用VM内部的操作系统内核。

10.3 拉取镜像

Docker 主机安装之后，本地并没有镜像。

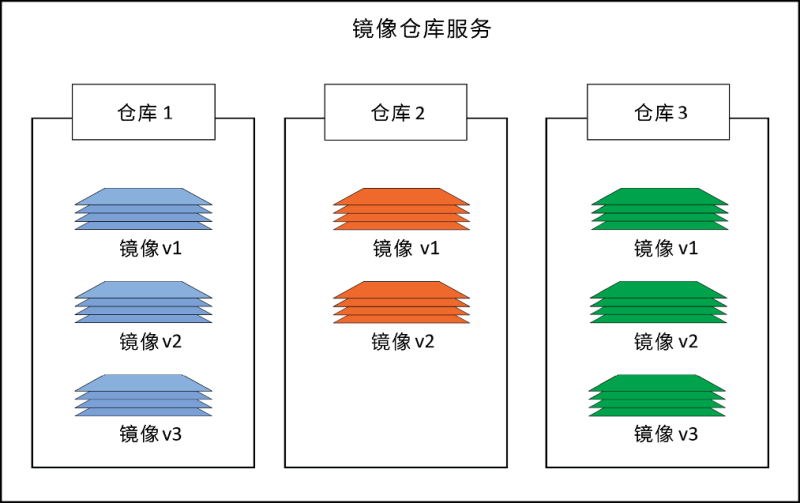
docker imager pull 是下载镜像的命令。镜像从远程镜像仓库的仓库中下载。

通过 docker image ls 可以查看本地仓库包含镜像的情况。

10.4 镜像仓库服务

Docker 镜像存储在镜像仓库服务（Image Registry）当中。

Docker 镜像客户端的镜像仓库服务是可配置的，默认使用Docker Hub

镜像仓库服务包含多个镜像仓库。同样，一个镜像仓库可以包含多个镜像。 

官方和非官方镜像仓库

Dcoker Hub 也分为官方仓库（Official Repository）和 非官方仓库（Unofficial Repository）

10.5 镜像名米和标签

只需要给出镜像的名字和标签,就能在官方仓库中定位一个镜像（采用”:”分割），从官方仓库拉取镜像时：

docker image pull 命令的格式如下。

docker image pull <repository>:<tag>

例：

docker image pull ubuntu:latest

注：如果没有在仓库名称后制定具体的镜像标签，则Docker 会假设用户希望拉取标签为latest的镜像。

标签为latest的镜像没有什么特殊魔力！标有latest标签的镜像不保证时仓库最新的镜像。

从非官方仓库拉取镜像也是类似的，只需要在仓库名称前面加上Docker Hub 的用户名或者组织名称。

下面示例来展示如何从tu-demo 仓库拉取v2 这个镜像，其中镜像的拥有者是Docker Hub 账户nigelpolton,一个不应该被信任的账户。

docker image pull nigeloulton/tu-demo:v2

docker image pull microsoft/dotnet:latest

假设上面的示例中的镜像位于Google容器镜像仓库（GCR）中，则需要在仓库名称前加上gcr.io。 如：

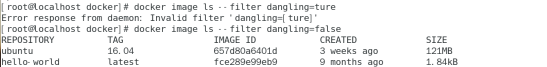
Docker pull gcr.io/nigelpouton/tu-demo:v2(并不存在)

可能需要拥有第三方镜像仓库服务的账户，并在拉取镜像前完成登录。

10.6 一个镜像的多个标签

Docker 提供了 --filter 参数来过滤docker image ls 命令返回的镜像列表内容.

通过 docker image ls --filter dangling=ture 命令 返回悬虚镜像



悬虚镜像： 悬虚镜像是指那些没有被标签的镜像 列表展示为<none>:<none>

当构建一个新图像，然后为该镜像打了一个已经存在的标签。

当此情况出现，Docker会构建新的图像，然后发现已有镜像包含相同的标签，接着Docker会移除旧镜像上面的标签，将该标签标在新的镜像上。

可以通过docker image prune 命令移除全部的虚空镜像

docker image prune -a 命令会额外移除没有被使用的镜像（那些没有被任何容器使用的镜像） 。

Docker 目前支持如下的过滤器。

dangling : 可以指定true 或者 false ，仅返回虚空镜像（true），或者非悬虚镜像（false）

before: 需要镜像名称或者ID作为参数，返回在之前被创建的全部镜像。

since: 与before 类似，不过返回的是指定镜像之后创建的全部镜像。

label: 根据标注（label）的名称或者值，对镜像进行过滤。 docker image ls 命令输出中不现实标注内容。

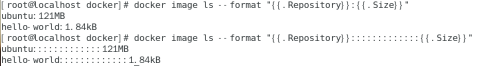
其他过滤方式可以使用reference.

docker image ls --filter=reference=’\*:latest’



使用 --format参数来通过Go模板对输出内容进行格式化。例如，下面的指令将只返回Docker主机上镜像的大小属性。

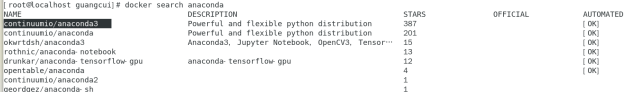




10.7 通过CLI 方式搜索Docker Hub

docker search 命令允许通过CLI的方式搜索Docker Hub。 可以通过”NAME ”字段的内容进行匹配，并且给予返回内容中任意列的值进行过滤。

例如:

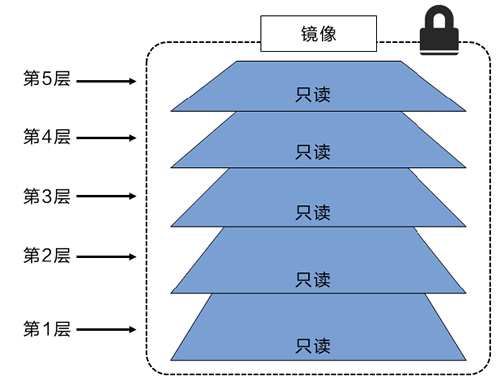


docker search anaconda --filter “is-official=true” 用来筛选官方镜像

docker search anaconda --filter “is-atuomated=true” 用来筛选自动创建的仓库

10.8 镜像和分层

Docker 镜像由一些松耦合的只读镜像层组成。



Docker 负责堆叠这些镜像层，并且将他们表示为单个的统一的对象。

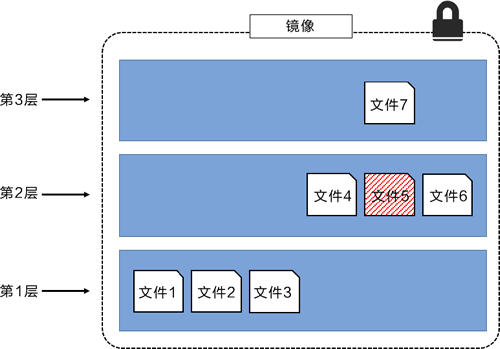
查看镜像分层的方式可以通过docker image inspect 命令。

例如：



在参数 Layers中。

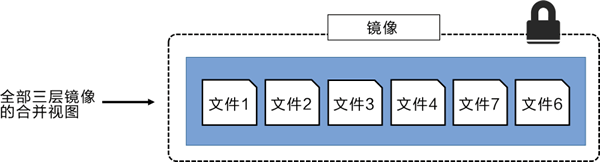
下图中展示了一个稍微复杂的三层镜像，在外部看来整个镜像直有6个文件，这是因为最上层中的文件7是文件5的一个更新版本。



当遇到这种情况时，上层镜像层中的文件覆盖二零底层中的文件。这样就使得文件的更新版本作为一个新镜像层添加到镜像层当中。

Docker 通过存储引擎（新版本采用快照机制）的方式实现镜像层对战，并保证多镜像层对外展示为统一的文件系统。

下图展示了与系统显示相同的三层镜像。所有镜像层堆叠并合并，对外提供统一的视图。



10.9 共享镜像层

多个镜像之间可以并且确实会共享镜像层。这样可以有效节省空间并提升性能。

当拉取多个镜像时，拉取的镜像包含已经拉取的镜像，Docker 则会跳过。

10.10 根据摘要拉去镜像

前面学习了通过标签来拉取镜像，这也是常见的方式。



10. 11 镜像散列值（摘要）

从Docker 1.10 版本开始，镜像就是一系列松耦合的独立层的集合。

镜像本身就是一个配置对象，其中包含了镜像层的列表以及一些元数据信息。

镜像层才是实际数据存储的地方（比如文件等，镜像曾之间是完全独立的，并没有从属于某个镜像集合的概念）。

镜像的唯一标识 是一个加密ID，即配置对象本身的散列值。每个镜像层也由一个加密ID区分，其值为镜像层本身内容的散列值。

这意味着修改镜像的内容或其中任意的镜像层，都会导致加密散列值的变化。所以，镜像和其镜像层都是不可变的，任何改动都能很轻松的被辨别。

这就是所谓的内容散列（Content Hash）.

在推送和拉取镜像的时候，都会对镜像层进行压缩来节省网络带宽以及仓库二进制存储空间。

但是压缩会改变镜像内容，这意味着镜像的内容散列值在推送或者拉取操作之后，会与镜像内容不相符！这显然是个问题。

例如，在推动镜像层到DockerHub的时候,Docker Hub 会尝试确认接收到的镜像没有在传输过程中被篡改。

为了完成校验，Docker Hub 会根据镜像层重新计算散列值，并与原散列值京姓比较。

因为镜像在传输过程中被压缩（发生了改变），所以散列值的校验也会失败。

为避免该问题，每个镜像层同时会包含一个分发散列值（Distribution Hush）。这是一个要锁版镜像的散列值，当从镜像仓库服务拉取或者推送镜像的时候，其中就包含了分发散列值，该散列值会用于校验拉取的镜像是否被篡改过。

这个内容寻址存储模型极大地提升了镜像的安全性，因为在拉取和推送操作后提供了一种方式来确保镜像和镜像层的数据是一致的。

该模型也解决了随机生成镜像和镜像层ID这种方式可能导致的ID冲突问题。

10.12 多层架构的镜像

多层架构镜像解决了镜像在不同架构间是否匹配的问题。

简单的说，就是一个镜像标签下可以支持多个平台和架构。

为了实现这个特性，镜像仓库服务API支持两种重要结构：Manifest列表（新）和Manifest。

Manifest 列表是指某个镜像标签支持的结构列表。 其支持的每种架构，都有自己的Mainfest定义，其中列举了该镜像的构成。

所有官方镜像都支持Manifest列表，但是全面支持各种架构的工作仍在推进当中。

创建支持多架构的镜像需要镜像的发布者作贡多的工作。同时，某些软件也并非跨平台的。在这个前提下,Manifest li列表是可选的----在没有Manifest 列表的情况下，镜像仓库服务会返回普通法的Manifest。

10.13 删除镜像

可以通过命令 docker image rm 容器ID/name

删除操作会在当前主机上删除该镜像以及相关的镜像层。这意味着无法通过docker image ls 命令看到删除后的镜像，并且对应的包含镜像层数的目录也会版删除。

但是，如果某个镜像曾被多个镜像共享，那直有当全部以来该镜像曾的镜像都被删除后，该镜像才会被删除。

docker image ls -q 命令值返回了系统中被恩迪拉取全部镜像的ID列表。

## Docker 容器（container）详解

容器是镜像的运行时实例。正如从虚拟机模板启动VM一样，用户也同样可以从单个镜像启动一个或多个容器。

虚拟机和容器的最大区别是容器更快并且更轻量级----与虚拟机运行在完整的操作系统之上相比，容器会共享其所在主机的操作系统/内核。

启动容器最简便方式是使用docker container run 命令。

该命令可以携带很多参数，在其基础的格式

docker container run <image> <app>中，指定了启动所需的镜像以及要运行的应用。

Docker container run -it ubuntu /bin/bash 则会启动某个Ubuntu linux容器，并运行 Bash Shell 作为其应用。

在linux主机（或者在linux 容器某时下的Windows主机上）运行：

Docker container run ubuntu:latest sleep 10 命令， shell 会连接到容器Shell 10s 的时间，然后退出。

可以使用docker container stop 命令手动停止容器运行，并且使用

docker container start 再次启用该容器。如果再也不需要该容器，则使用docker container rm 命令来删除容器。

11.1 虚拟机的额外开销

虚拟机模型将底层硬件资源划分到虚拟机当中，每个虚拟机都包含了虚拟CPU,虚拟RAM,虚拟磁盘等资源的一种软件结构。

因此，每个虚拟机都需要有自己的操作系统来声明，初始化管理这些虚拟资源。

每个操作系统都需要独立认证，并且都需要打补丁升级，每个操作系统都面临被攻击的风险。

通常将这种现象乘坐OS Tax 或者VM Tax，每个操作系统都占用一定的资源。

而容器内部并不需要内核，也就没有定位，解压以及初始化的过程----更不用再内核启动过程中对硬件的遍历和初始化了。

11.2 检查Docker daemon

通常登录Docker 主机后的第一件事情是检查Docker 是否正在运行。

通过docker version 命令来确定Docker 是否运行；当命令输出中包含Client 和

Server 的内容时，（docker 运行），可以继续下面的操作。如果在Server 部分中包含了错误码，表示Docker daemon 很可能没有运行，或者当前用户没有权限。

如果在linux 中遇到无权限访问的问题，需要确认当前用户是否属于本地Docker unix 组，如果不是可以通过usermod -aG docker <user>来添加，然后退出重新登录shell, 即可生效。

在不同操作系统可以使用下面的命令来检查Docker Daemon的装状态。



在 Windows Server 2016的PowerShell窗口执行该命令：

Get-Service docker

11.3 启动一个简单容器

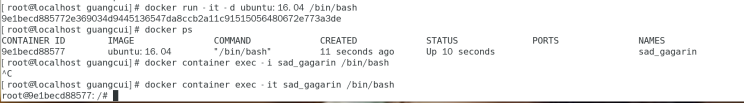
启动容器命令的基础格式：

docker container run <options> <im- age>:<tag> <app>

11.4 容器进程

运用docker container ls 命令来查看当前系统正在运行的容器列表

后台运行的容器 可以通过 docker container exec 命令将终端重新连接到Docker



11.5 容器生命周期

容器的生命周期包含以下几个过程： 创建，运行，休眠，销毁等过程。

容器创建：

docker container run --name percy -it ubuntu:latest /bin/bash

新建名为percy 的容器。

使用Ctrl + PQ 组合件退出当前容器。

使用docker container stop percy 命令，来停止容器运行，切换到暂停（vacation）状态.

docker container start 将容器重新启动

docker container rm （-f 参数，可以用来一次性删除运行中的容器）：删除容器

注： 优雅的停止容器：

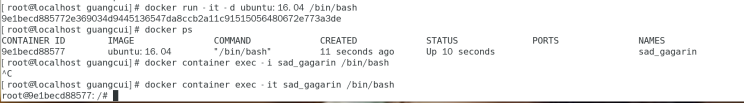
使用 docker container stop 可以容器暂停，然后使用docker container rm 命令删除.

11.6 利用重启策略进行容器的自我修复

通常建立在运行容器时配置好重启策略。这时容器的一种自我修复能力，可以在指定时间或者错误后重启来完成自我修复。

重启策略应用于每个容器，可以作为参数被强制传入docker-container run 命令中，或者在Compose 文件中声明（在使用Docker Compose 以及Docker Stacks 的情况下）。

always 策略是一种简单的方式。除非容器被明确停止，比如通过docker container stop命令，否则该策略会一直尝试重启处于停止状态的容器。



注意： 容器于18分钟前创建，但却在1s前才启动。这是因为在容器中输入退出命令的时候，容器被杀死，然后Docker 又重启启动了该容器。

--restart always 策略有一个很有意思的特性，当daemon 重启的时候，停止的容器也会被重启。

Always 和unless-stopped的最大区别，就是那些指定了 --restart unless-stopped 并处于Stopped(Eexited)状态的容器，不会在Docker daemon重启的时候被重启。

on-failure 策略会在容器退出容器并且返回值不是0的时候，重启容器。

就算容器处于stopped 状态，在Docker daemon 重启的时候，容器叶辉被重启。

## Docker 应用容器化（将应用部署到容器中）

Docker 的核心思想就是如何将应用整合到容器中，并且能在容器中实际运行。

将应用整合到容器中并且运行起来的过程，成为容器“容器化”（Containerizing）有时也叫“Docker化”（Dockerizing）。

容器是为应用而生的，具体来说，容器能能够简化应用的构建，部署和运行过程。

完整的医用容器化过程主要分为以下几个步骤：

编写应用代码。

创建一个Dockerfile,其中包括当前应用描述，依赖以及该如何运行的应用。

对该Dockerfile 执行 docker image build 命令。

等待Docker 将应用程序构建到Docker 镜像中。、

一旦应用容器化完成（即应用被打包为一个Docker 镜像，就可以镜像的形式交付并以容器的方式运行了。）

12.1 单体应用容器化

应用容器化的过程大致分为如下步骤：

获取应用代码。

分析Dockerfile。

构建应用镜像。

运行该应用。

测试应用。

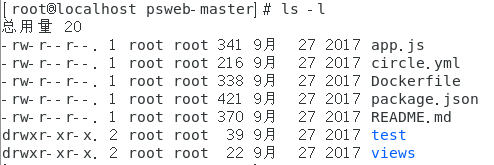
容器应用化细节。

生产环境中的多阶段构建。

最佳实践。

12.2 获取应用带码：

应用代码可以从网（[https://pan.baidu.com/s/150UgIJPvuQUf0yO3KBLegg](https://pan.baidu.com/s/150UgIJPvuQUf0yO3KBLegg" \t "http://c.biancheng.net/view/_blank) 提取码：pkx4）

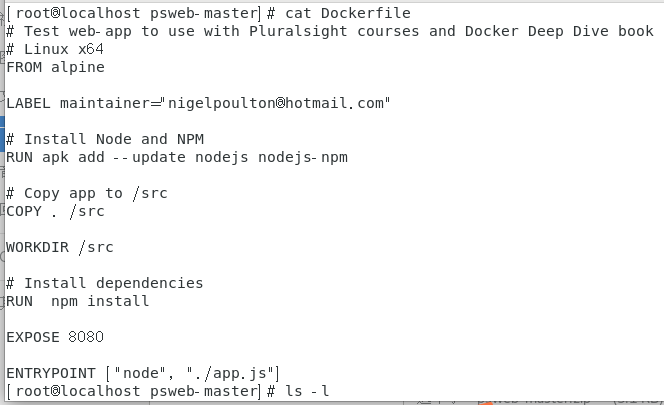


12.3 分析Dockerfile

在代码目录当中，有个名称为Dockerfile的文件，这个文件包含了对当前应用的描述，并且能够知道Docker镜像的构建。

在Docker 当中，包含应用文件的目录通常被称为构建上下文（Build Context）。 通常将Dockerfile方法构建上下文的根目录下。

另外很总要的一点是，文件开头字母是大写D，这里是一个单词。想“dockerfile”“Docker file”这种写法都是不被允许的。



Dockerfile 主要包含两个用途：

对当前应用的描述

指导Docker完成应用的容器化（创建一个包含当前应用的镜像）。

不要因为Dockerfile 就是一个描述文件而对其有所轻视。Dockerfile 能实现开发和部署两个过程的无缝切换。

同时Dockerfile还能帮助新手快速舒心项目。Dockerfile对当前的因公及期以来有一个清晰准确的描述，而且非常容易阅读和理解。

因此，要像重视你的代码一样重视这个文件，并将它纳入源控制系统当中

下面是这个文件中的一些个关键步骤概述： 以 alpine镜像作为当前镜像基础，制定维护着（maintainer）为 [“nigelpoultion@hotmail.com”,安装Node.js和NPM,](mailto:\“nigelpoultion@hotmail.com\”,安装Node.js和NPM,)将应用的代码复制到镜像当中，设置新的工作目录，安装依赖包，记录应用端口，最后将app.js设置为默认运行的应用。

DOckerfile 文件中：

每个标签其实是一个键值对（key-value）,在一个镜像当中可以通过增加标签方式来为镜像添加定义元数据。

FROM指令指定的镜像，回座位当前镜像的一个基础镜像，当前应用的剩余内容会作为新增镜像层添加到基础镜像层之上。

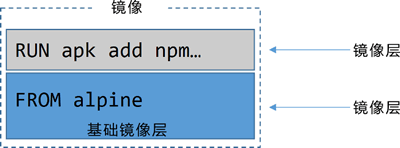
[LABEL指定了当前镜像的维护者为“nigelpoulton@hotmail.com”](mailto:LABEL指定了当前镜像的维护者为\“nigelpoulton@hotmail.com\”)

备注维护者信息有助于为该镜像的潜在使用提供沟通途径，这是一种值得提倡的做法。

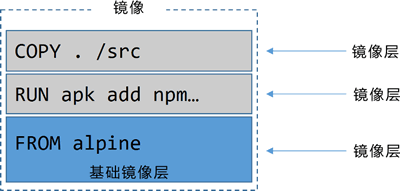
RUN apk add --update nodejs nodejs-npm 指令使用alpine 的apk 包管理器将nodejs 和 nodejs-npm 安装到当前镜像之中。

RUN指令会在FROM指定的alpine 基础镜像之上，新建一个镜像层来存储这些安装内容。

当前镜像的结构如下图所示。



COPY./src 指令将应用相关文件从构建上下文复制到了当前镜像中，并且新建一个镜像层来存储。COPY执行结束之后，当前镜像共包含3层，如下图所示。



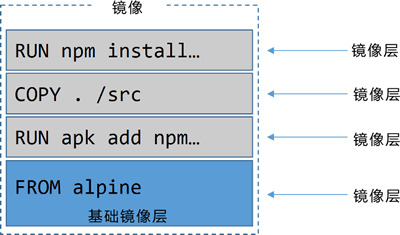
下一步，Dockerfile 通过WORKDIR 指令，为Dockerfile 中尚未制定的指令设置工作目录。

该目录与镜像相关，并且会作为元数据记录当镜像配置中，当不会创建新的镜像层。

然后,RUN npm install 指令会根据package.json中的配置信息，使用npm来安装当前应用的相关依赖包。

npm 命令会在前文设置的一个工作目录中执行，并且在镜像中新建镜像层来保存相应的依赖文件。

目前镜像一共包含4层，如下图所示。



因为当前应用需要配置TCP端口 8080 对外提供一个web服务，所以Dockfile中通过 EXPOSE 8080指令来完成响应端口的设置。

这个配置信息回座位镜像的元数据被保存下来，并不会产生新的镜像层。

最终，通过ENTRYPOINT指令来指定当前镜像的入口程序。ENTRYPOINT指定的配置信息也是通过镜像元数据的形式保存下来，而不是新增镜像层。

12.4 容器化当前应用/构建具体的镜像

下面的命令会构建恒生成一个名为web：latest的镜像。命令最后的点（.）表示Docker 在进行构建的时候，使用当前目录作为构建上下文。

一定要在命令最后包含这个点，并且在执行命令前，要确认当前目录是psweb（包含Dockerfile和应用代码）。

命令： docker build -t psweb:latest .

参数解析：

-t 为镜像添加标签

-f 指定Dockerfile 文件所在位置。

12.5 推送镜像到仓库

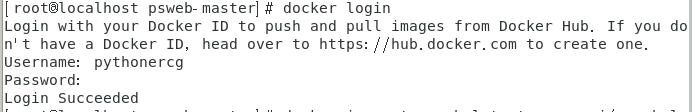
在创建镜像之后，将其保存在一个镜像服仓库是一个不错的方式。这样存储镜像会比较安全，并且可以被其他人访问使用。

Docker Hub 就是这样的一个开放的公共镜像仓库服务，并且这也是docker image push 命令默认推送的地方。

在推送镜像之前，需要先试用Docker ID 登录 Docker Hub .除此之外，还需要为待推送的镜像打上合适的标签。

接下来介绍一下如何登录Docker Hub ，并将镜像推送到其中。

使用命令 docker login 然后输入用户名和密码



推送 Docker 镜像之前，还需要为镜像打标签。这是因为Docker 在镜像推送的过程中需要如下信息。

Registry (镜像仓库服务)

Repository (镜像仓库)

Tag (镜像标签)

无需为Registry 和 Tag 指定值。当没有上述信息指定具体指的时候，Docker 会

默认Registry=docker.io 、 Tag=latest 。

但是Docker 并没有给Repository 提供默认值，而是被推送镜像中的REPOSITORY属性值获取。这一点可能并不好李杰，下面会通过一个完整的例子来介绍如何向Docker Hub 中推送一个镜像。

在前面的例子中执行了docker image ls 命令，该命令对应输出内容可以看到，镜像仓库的名称是web。

这意味着执行 docker image push 命令，会尝试将镜像推送到docker.io/psweb:latest 中。

但是其实 pythonercg 这个用户并没有psweb 这个镜像仓库的访问权限，所以只能尝试推送到 pythonercg 这个二级命名空间（Namespace）之下。

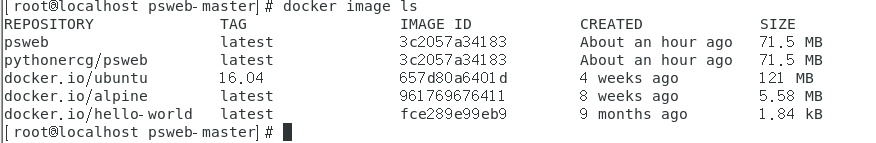
因此需要使用pythonercg(Docker Hub账号) 这个ID,为当前镜像重新打一个标签。

命令：

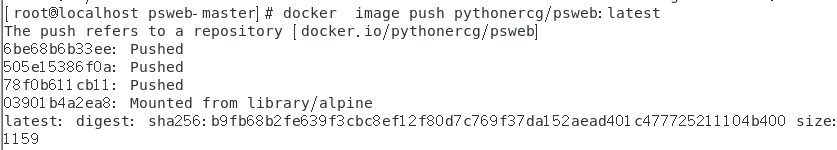
docker image tag psweb:latest pythonercg/psweb:latest

为镜像打标签命令的格式是docker image tag <current-tag> <new-tag>，其作用是未指定镜像添加一个额外标签，并且不需要已经存在的标签。

使用docker image ls 命令查看



使用 docker image push 命令进行推送



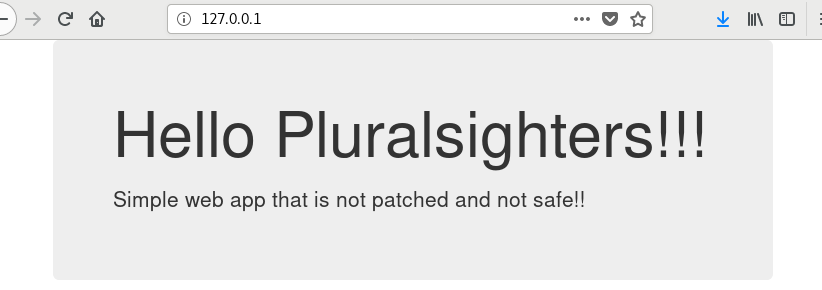
12.6 运行应用程序

前文中容器化的这个应用程序其实很简单，从app.js 这个文件内容中可以看出，这其实就是一个在8080端口提供web服务的应用程序。

下面的命令会基于web:latest 这个镜像，启动一个名为C1 的 容器，该容器将内部的8080端口 与Docker所在主机的80端口进行映射。

这就意味着，打开浏览器输入127.0.0.1:80 就可以访问应用程序了。

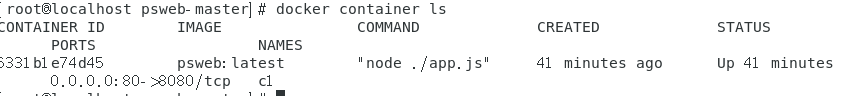




-d 参数的作用是让应用程序以守护线程的方式在后台运行。

-p 80:8080 参数的作用是将主机的80端口与容器内的8080端口进行映射。

使用docker container ls 命令验证是否启动成功



12.7 详述

到现在为止，应当成功完成一个示例应用程序的容器化。下面是其中一些谢姐部分的回顾和总结。

Dockerfile 中的注释行，都是以# 开头的。

除注释之外，每一行都一条指令（Instruction）。指令的格式是指令 参数 如下。

INSTRUCTION argument

指令是不区分大小写的，但是通常都采用大写的方式。这样Dockerfile的可读性会高一些。

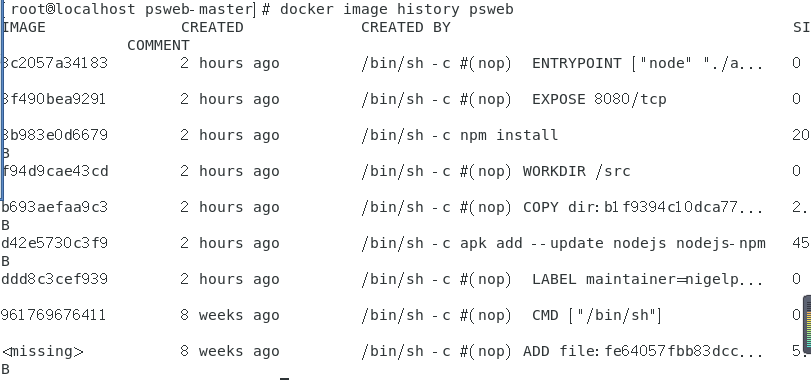
Docker image build 命令会按行来解析Dockerfile 中的指令并顺序执行。

部分指令会在镜像中创建新的镜像层，其他指令只会增加回修改镜像的元数据信息。

在上面的例子当中，新增镜像层的指令包括FROM, RUN Y以及COPY ,而新增元数据的指令包括EXPOSE,WORKDIR, ENV 以及 ENTRTPOINT.

关于如何区分命令是否会新建镜像层，一个基本的原则则是，如果指令的作用是想镜像中增添新的文件或者应用程序，那么这条指令就会新建镜像层；如果只告诉Docker 如何完成构建或者如何运行应用程序，那么只会增加镜像的元数据。

可以通过 docker image history 来查看在构建镜像的过程中都执行了哪些指令。



在上面的输出内容当中，有两点是需要注意的。

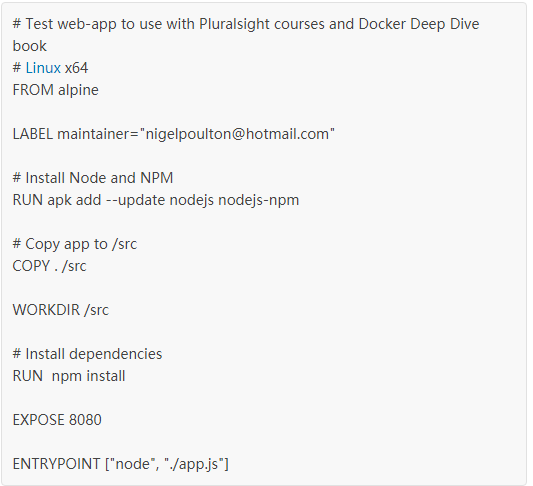
首先，每行内容都对应了Dockerfile 中的一条指令（顺序是自下而上）。CREATE BY 这一列中还展示了当前行具体对应Dockerfile中的那条指令。

其次，从这个输出内容中，可以观察到只有4条指令会新创建镜像层（就是那些SIZE 列对应的数值不为0的指令），分别是Dockerfile 中FROM. RUN 以及COPY.

DockerFile

使用Docker 中的 docker image build 命令会读取 Dockerfile, 并将应用程序容器化。

Dockerfile 由一行行命令语句组成，并支持以#开头的注释行。例如：



13.1 使用-t 参数为镜像打标签，使用-f参数指定Dockerfile的路径和名称，使用-f 参数可以指定位于任意路径下的任意名称的Dockerfile。

13.2 构建上下文是指应用文件存放的位置，可能是本地Docker主机上的一个目录或者远程Git库

13.3 Dockerfile 中的from 指令用于指定要构建的镜像的基础镜像。它通常是Dockerfile的第一条指令。

13.4 Dockerfile中的Run指令用于镜像中执行命令，这会创建新的镜像层。每个RUN指令创建一个新的镜像层。

13.5 Dockerfile中的COPY指令用于将文件作为一个新的层添加到镜像中。通常使用COPY指令将应用代码复制到镜像中。

13.6 Dockerfile 中的EXPOSE指令用于记录应用所用的网络端口。

13.7 Dockerfile 中的ENTRYPOINT指令用于指定镜像以容器方式启动后默认运行的程序。

13.8 其他的Dockerfile 指令还有 LABEL, ENV. ONBUILD, HEALTHCHECK,CMD等。

## 14. Docker Swarm 简介

Swarm 是Docker 官方提供的一款集群管理工具，其主要作用是把若干台Docker主机抽象为一个整体，并且通过一个入口统一管理这些Docker 主机上的各种Docker资源。

Docker Swarm 包含两方面： 一个企业级的Docker 安全集群，以及一个微服务应用编排引擎。

从集群角度来说，一个Swarm 有一个或多个Docker 节点组成。这些节点可以是物理服务器，虚拟机，树莓派或者云实例。位移的前提就是要求所有节点通过可靠的网络相连。

节点会被配置为挂不给力节点（Manager）或工作节点（Worker)。广利节点负责集群控制面（COntrol Plane）, 进行诸如监控集群状态，分发任务至工作节点等操作。工作节点接受来自管理节点的任务并执行。

Swarn 的配置和状态信息保存在一套位于所有管理节点上的分布式etcd数据库中。改数据库运行于内存中，并保持数据的最新状态。关于改数据库最棒的是，他既不户需要任何配置，作为Swarm的一部分被安装，无需管理。

关于集群管理，最大的挑战在与保证其安全性，搭建Swarm集群时将不可避免是TLS,因为他被Swarm紧密集成。

## 15. Docker swarm 创建集群

15.1需要环境：

1. VMware 2. Centos 7 3. Docker

首先创建一个虚拟机，系统为Centos 7

15.2 Centos 7 安装docker 和 swarm

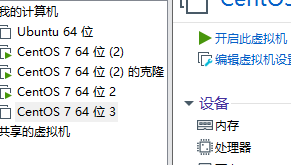
yum install docker 安装docker 命令

docker pull swarm 安装swarm

15.3 克隆镜像

将虚拟机镜像克隆，步骤如下：

打开虚拟机列表：



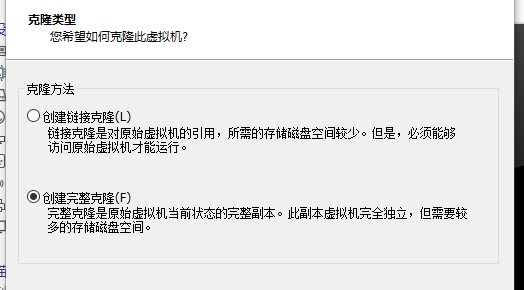
鼠标右键---》管理---》克隆

第一步： 点击下一步

然后出现：

选择默认选项即可。

第二部： 点击下一步



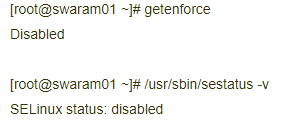
选择创建完整克隆

然后点击下一步--》完成即可。

15.4 虚拟机克隆完成后，测试网络连接

如果无网络链接/重启网络服务失败：<https://blog.csdn.net/qq_24137739/article/details/102621011>

15.5 关闭 SElinux



如果disable 不是disabled

vi /etc/selinux/config

将 SELINUX=enforcing 改为 SELINUX=disabled

注意： 设置后需要重新启动才能生效。

15.6 关闭防火墙

systemctl stop firewalld.service # 停止firewall

systemctl disable firewalld.service # 禁止firewall开机启动

15.7 修改docker 监听端口

Swarm 是通过监听2375端口进行通信的，所以在使用Swarm进行集群管理之前，需要设置一下2375端口的监听。所有主机节点docker开启2375,2377（swarm集群），docker 版本不同，配置方式不一样

vi /lib/systemd/system/docker.service

在 ExecStart加入

-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

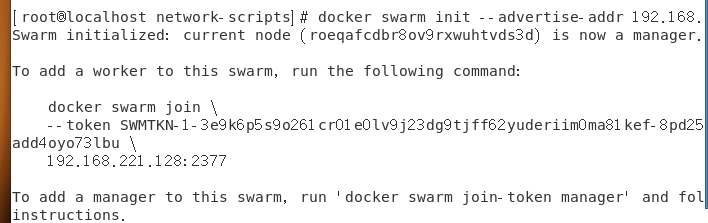
随后重启docker服务

systemctl daemon-reload ##使配置文件生效

systemctl restart docker

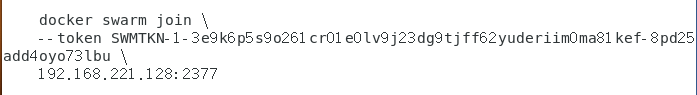
15.8 初始化swarm ,在mamager(或leade)机器上执行swarm init 命令

docker swarm init --advertise-add 192.168.x.x

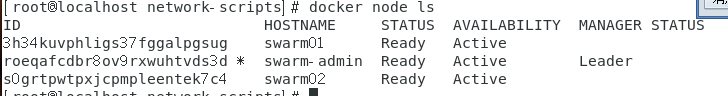


将其他机器加入到swarm 集群

在其他机器上运行：



然后使用docker node ls命令可以查看节点



注：

1. 在docker swarm init 初始化过程中，如果报错。



可以使用 docker swarm leave (如果有错误提示 请加上--force参数)