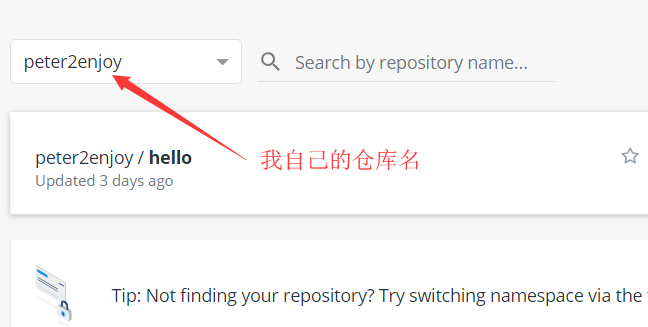
# 仓库使用

## docker官方仓库

### 注册

https://hub.docker.com

自由注册，邮件激活即可使用



### 命令使用

Docker pull/search/login/push/tag

tag [镜像名：版本] [仓库]/[镜像名：版本]：标记本地镜像，将其归入某一仓库

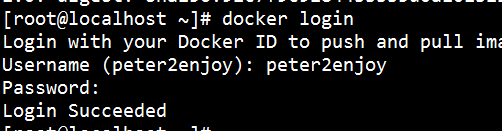
Push [仓库]/[镜像名：版本]: 推送镜像到仓库 --需要登陆

Search [镜像名]：在仓库中查询镜像 – 无法查询到tag版本

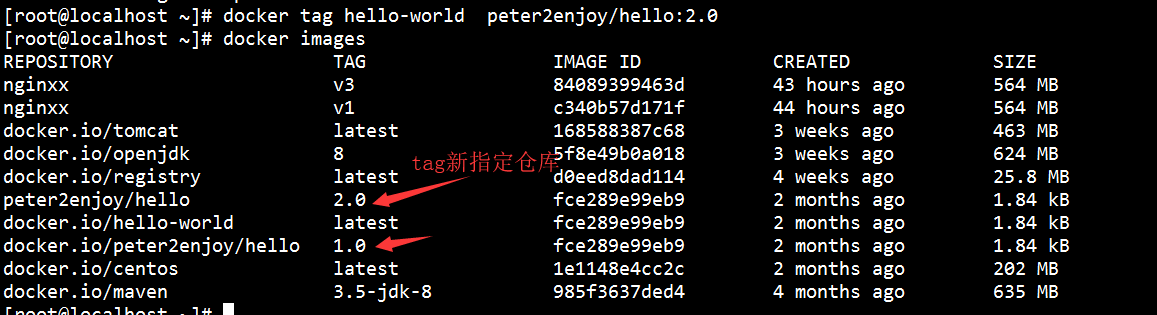
Pull [镜像名：版本]： 下载镜像到本地

Login：登陆仓库

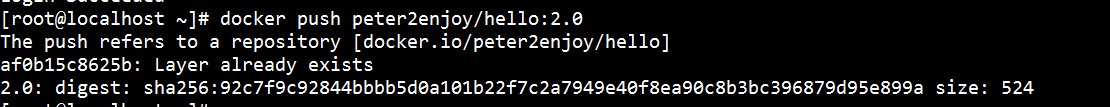
1、命令登陆dockerhub



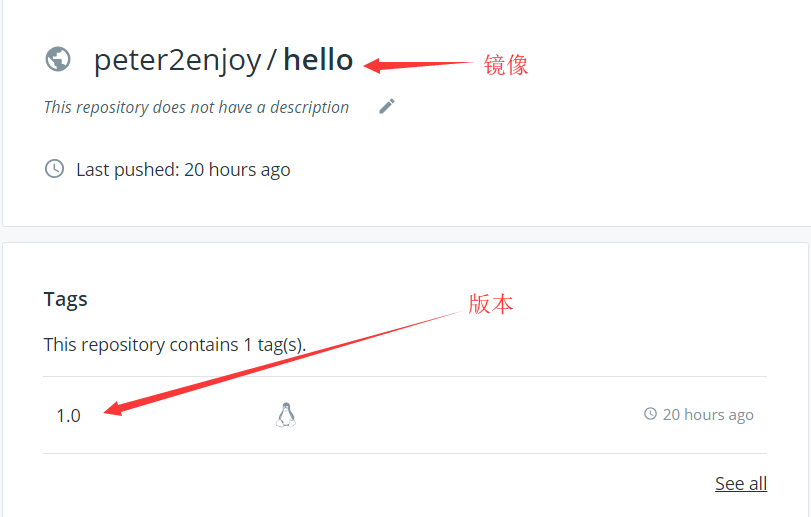
2、再使用tag命令标记一个镜像，指定自己的仓库



3、使用push命令推送此镜像到仓库里



4、打开查询自己仓库的镜像

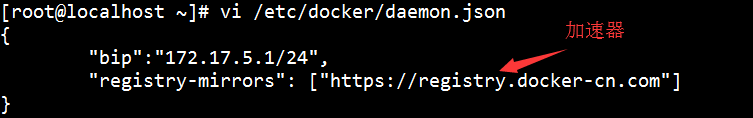


## 私有仓库

### 搭建

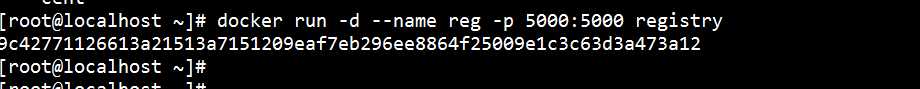
下载registry镜像：docker pull registry

-----可配置加速器加速下载

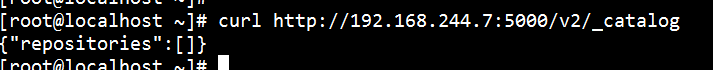


### 启动

docker run -d --name reg -p 5000:5000 registry

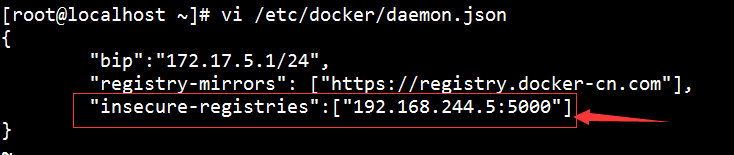


然后可以通过restful接口查看仓库中的镜像（当前仓库是空的）



### 配置http传输

私服默认只能使用https，需要配置开放http



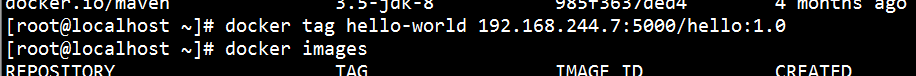
配置完毕重启下docker服务

systemctl daemon-reload

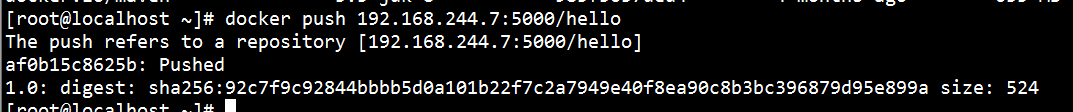
systemctl restart docker

### 私服仓库推送镜像

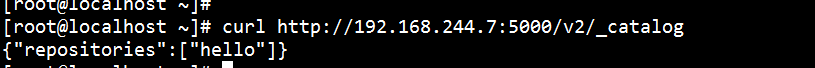
docker tag hello-world 192.168.244.7:5000/hello-world



docker push 192.168.244.7:5000/hello-world



查询镜像：http://192.168.244.5:5000/v2/\_catalog



查询hello版本： http://192.168.244.5:5000/v2/hello/tags/list



## commit镜像并上传仓库

### 创建一个centos容器：

启动后自动进入此容器



### 容器内安装nginx服务：

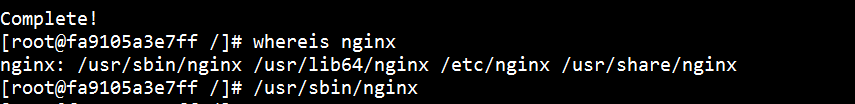
添加一下nginx源：

rpm -ivh http://nginx.org/packages/centos/7/noarch/RPMS/nginx-release-centos-7-0.el7.ngx.noarch.rpm

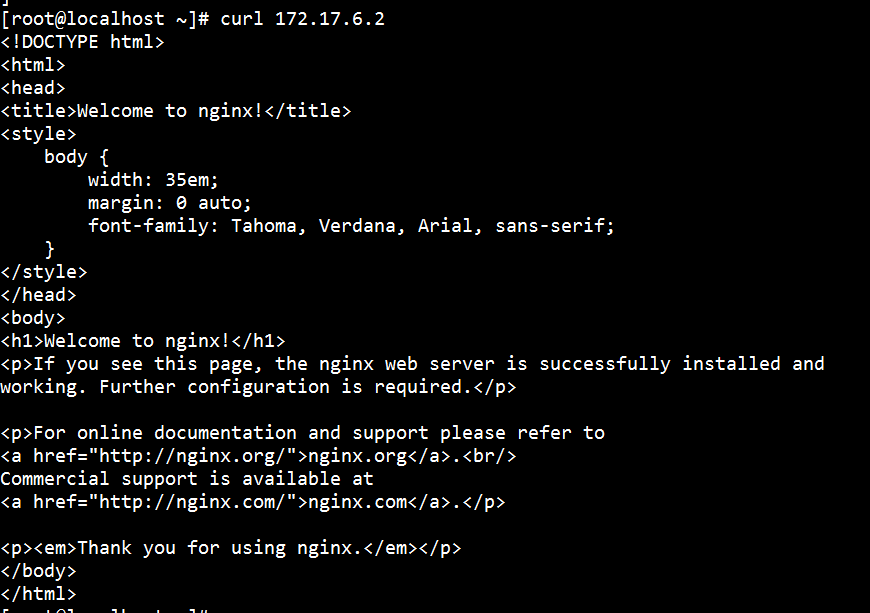
yum search nginx ##搜索一下看看

yum install nginx -y ## 安装

启动nginx服务



ctrl +P+Q退出容器，在主机环境内校验nginx请求，正常得到欢迎页

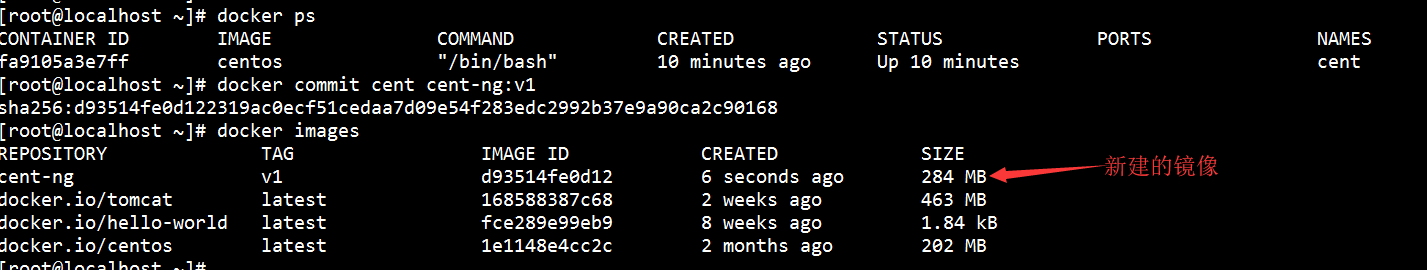


### commit服务为一个nginx镜像

现在要将cent容器提交成为一个镜像，命令如下：

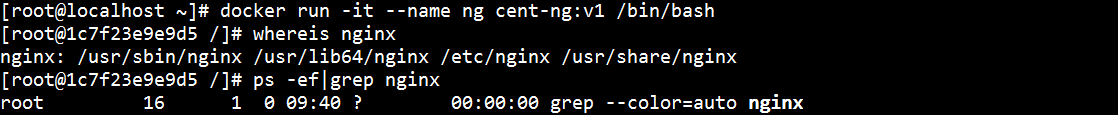
docker commit cent cent-ng:v1

可看到得到了新的镜像cent-ng:v1

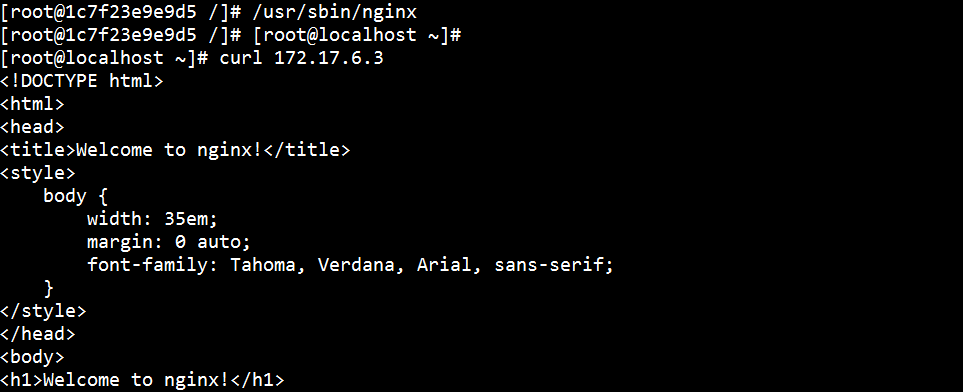


### 启动此nginx镜像

1、使用新建的镜像创建容器，并进入查看，发现已安装有nginx，但nginx并未启动

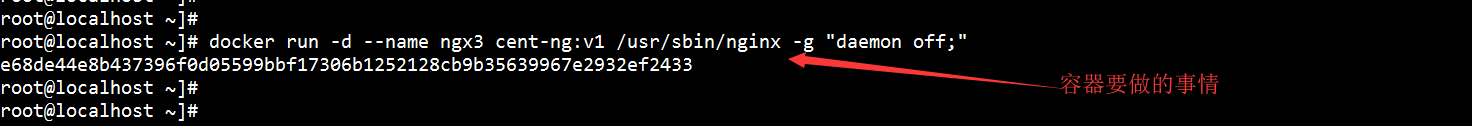


容器内启动nginx服务，并退出容器。在主机方校验，nginx欢迎页面出现



2、现在我们希望启动容器时，直接启动nginx服务，怎么做？

docker run -d --name ngx3 cent-ng:v1 /usr/sbin/nginx -g "daemon off;"



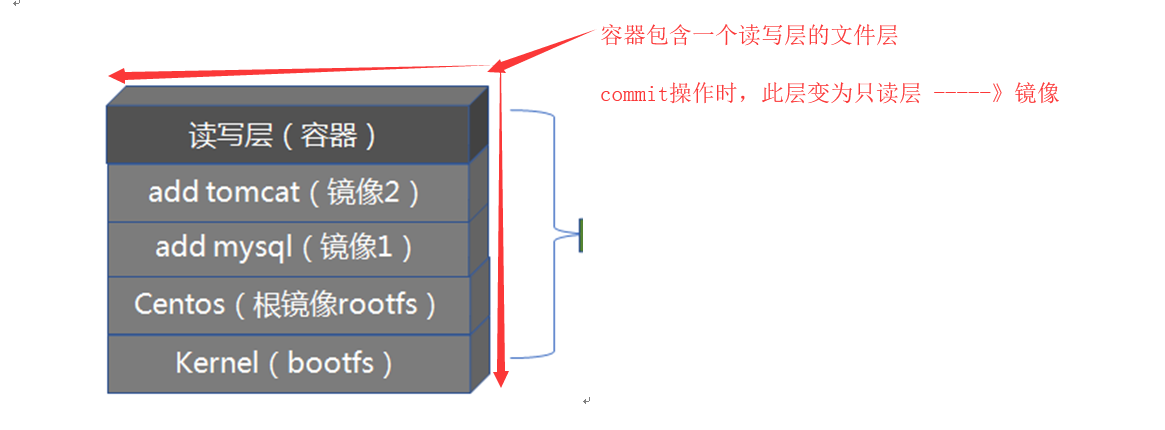
可看到，容器内nginx服务也已正常运行

ps:后面运行的命令都是容器命令，由于nginx命令没有设置到path中，所以全路径启动，

而nginx -g这个参数是指可以在外面添加指令到nginx的配置文件中，

daemon off是指nginx服务不运行在后端，而是在前台运行（container中的服务必须运行在前台）

## commit创建镜像方式的本质

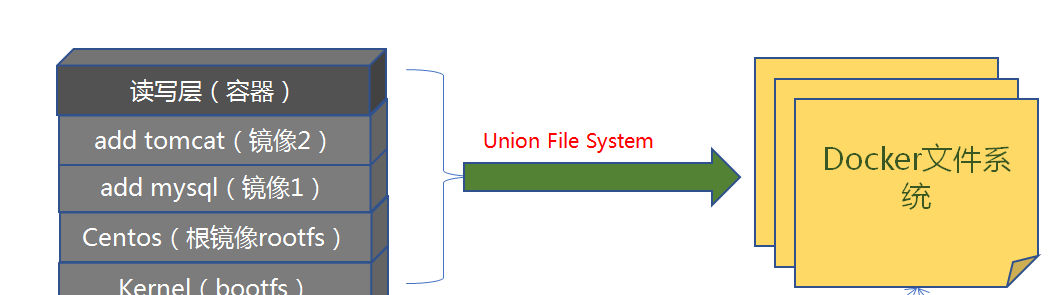


原容器与commit后的镜像，在文件系统上并无区别。只是把容器层原来的可写属性，置成了只读。于是变成了一个不可改的镜像

# 数据管理

docker容器运行，产生一些数据/文件/等等持久化的东西，不应该放在容器内部。应当以挂载的形式存在主机文件系统中。

## docker的文件系统



1. 镜像与容器读写层，通过联合文件系统，组成系统文件视角
2. 容器服务运行中，一定会生成数据
3. 容器只是运行态的服务器，是瞬时的，不承载数据的持久功能

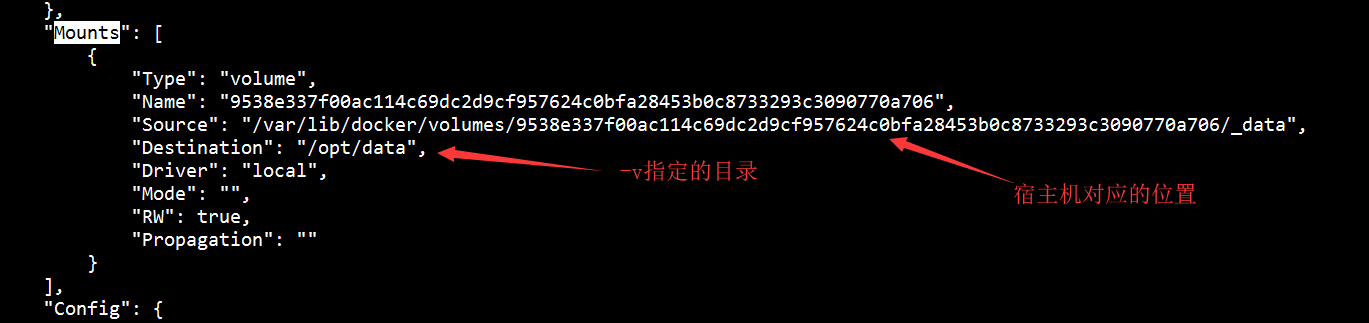
## volume文件挂载的探究

1、volume参数创建容器数据卷

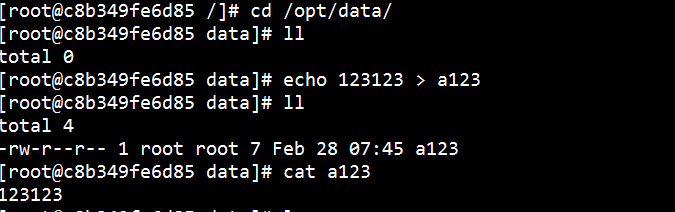


2、我们通过docker inspect data查看容器元数据，可看到挂载信息

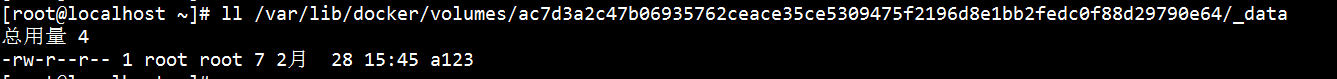




3、在容器端添加一个文件



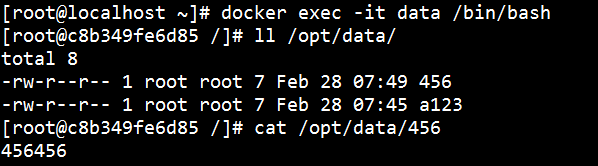
回主机目录查看，果然存在此文件：



4、在主机方添加一个文件



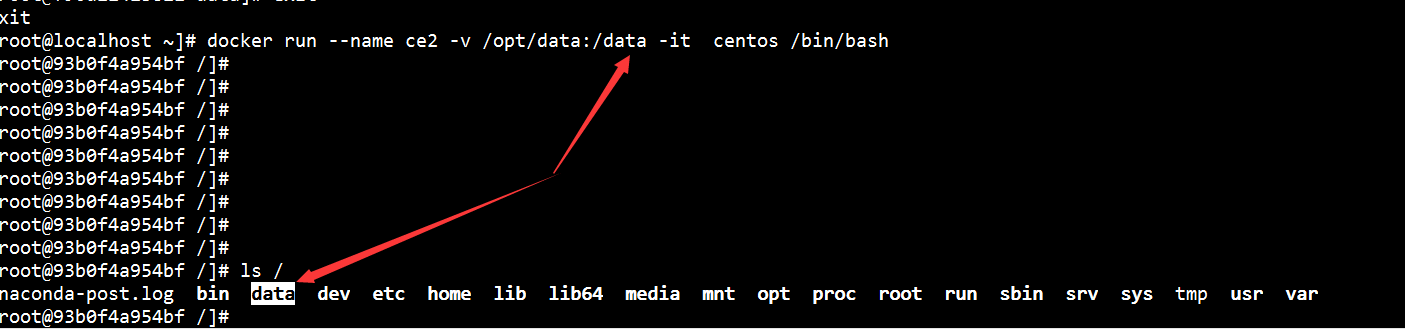
回容器里查看，果然也同步增加了此文件



5、指定主机目录方式挂载文件

格式：-v path1：path2

如下命令，容器方会自动增加一个data目录



宿主机方，同样自动增加一个/opt/data目录

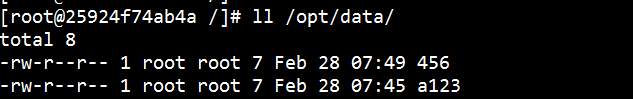


## volumes-from引用数据卷

新启一容器，引入上一步的data容器目录



自动得到同一个目录，内容与data容器里挂载一样



## 备份/恢复数据卷

备份：docker run --rm --volumes-from data -v $(pwd):/backup centos tar cvf /backup/data.tar /opt/data

恢复：docker run --rm --volumes-from data -v $(pwd):/backup centos tar xvf /backup/data.tar -C /

释义：

docker run --rm ----- 启动一个新的容器，执行完毕删除

--volumes-from data ------- data容器中挂载卷

-v $(pwd):/backup --------挂载当前目录到容器中为backup

cvf /backup/data.tar /opt/data --------- 备份/opt/data目录（即卷中所有的数据）为data.tar

xvf /backup/data.tar -C / ---------- 解压data.tar 到根目录/ ，因tar归档中已包含了/opt/data路径

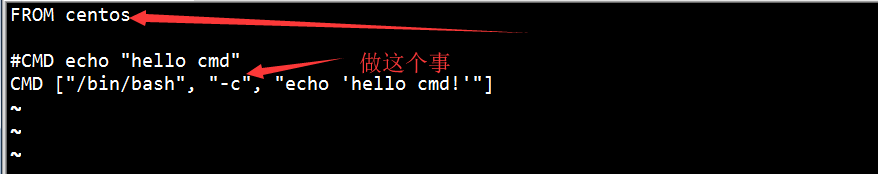
## 删除数据卷：

docker rm -v data

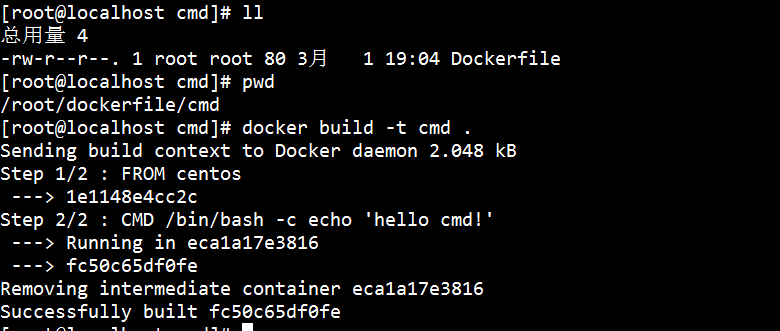
# Dockerfile使用

## dockerfile方式创建容器

最简单的dockerfile



创建镜像



使用此镜像运行一个容器



### dockerfile基本要素



### dockerfile指令

#### FROM：

　　FROM {base镜像}

　　必须放在DOckerfile的第一行，表示从哪个baseimage开始构建

#### MAINTAINER：

可选的，用来标识image作者的地方

#### RUN

RUN都是启动一个容器、执行命令、然后提交存储层文件变更。

第一层 RUN command1 的执行仅仅是当前进程，一个内存上的变化而已，其结果不会造成任何文件。

而到第二层的时候，启动的是一个全新的容器，跟第一层的容器更完全没关系，自然不可能继承前一层构建过程中的内存变化。

而如果需要将两条命令或者多条命令联合起来执行需要加上&&。

如：cd /usr/local/src && wget xxxxxxx

#### CMD：

　　CMD的作用是作为执行container时候的默认行为（容器默认的启动命令）

　　当运行container的时候声明了command，则不再用image中的CMD默认所定义的命令

一个Dockerfile中只能有一个有效的CMD，当定义多个CMD的时候，只有最后一个才会起作用

#### EXPOSE

EXPOSE 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 docker run -P 时，会自动随机映射 EXPOSE 的端口。

#### entrypoint：

entrypoint的作用是，把整个container变成可执行的文件，且不能够通过替换CMD的方法来改变创建container的方式。但是可以通过参数传递的方法影响到container内部

每个Dockerfile只能够包含一个entrypoint，多个entrypoint只有最后一个有效

当定义了entrypoint以后，CMD只能够作为参数进行传递

#### ADD & COPY：

　　把host上的文件或者目录复制到image中（能够进行自动解压压缩包）

#### ENV：

　　用来设置环境变量，后续的RUN可以使用它所创建的环境变量

#### WORKDIR：

　　用来指定当前工作目录（或者称为当前目录）

#### USER：

　　运行RUN指令的用户

#### VOLUME：

　　用来创建一个在image之外的mount point

### nginx镜像制作实战

#### 编译/安装nginx

mkdir一个目录，在此目录内下载nginx源码包

wget http://nginx.org/download/nginx-1.13.2.tar.gz

并创建一个Dockerfile文件，文件内制作一系列nginx的编译安装流程，内容如文件：

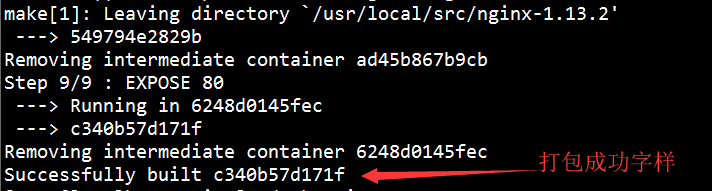




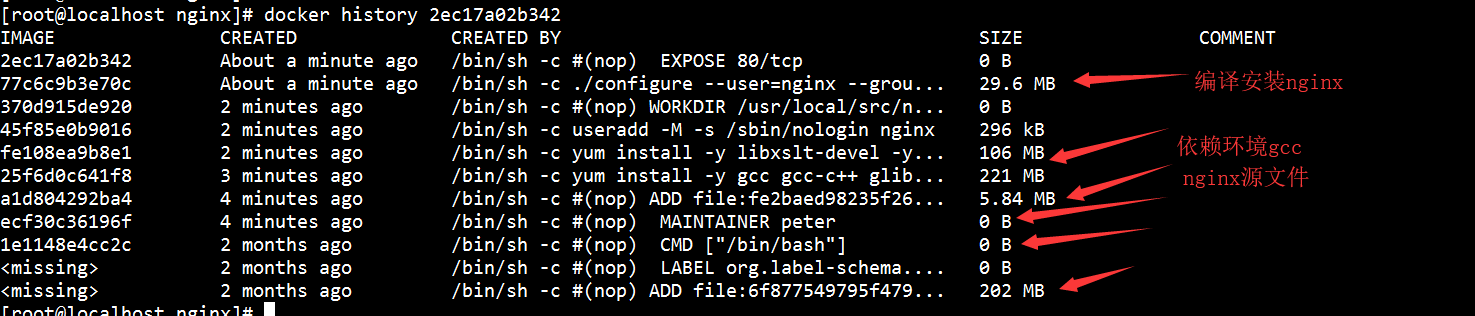
其中，每一个RUN就是增加一个镜像层文件，一层层的RUN命令最终形成一系列镜像层

运行build指令（注意最后的.代表当前路径），制作镜像

docker build -t cent-ngx2 .



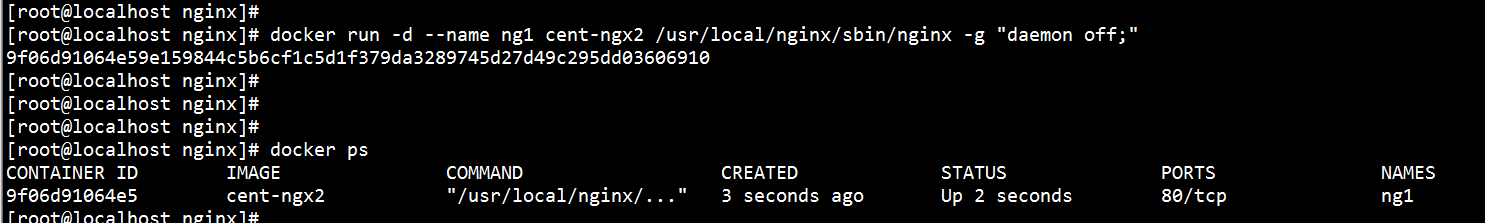
我们查看一下这个镜像的层次历史



可看到，此镜像层基本与dockerfile文件的RUN是一一对应的

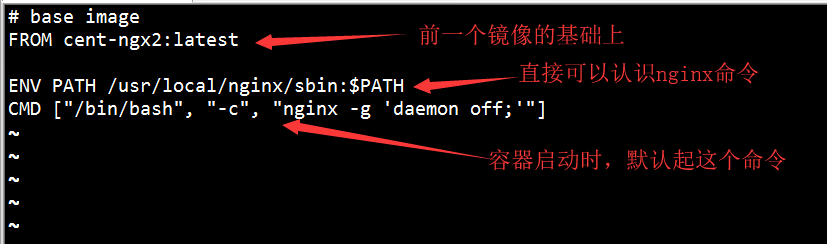
使用制作的nginx镜像，创建一个容器。

因此镜像无前台命令，因为必须指定启动命令 ：/usr/local/nginx/sbin/nginx -g "daemon off;"

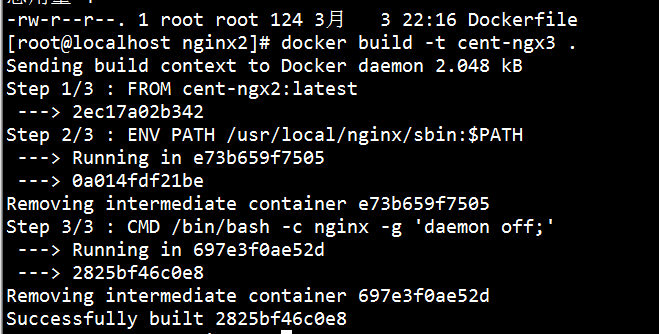


#### 为镜像指定环境变量，挂载目录，默认启动命令

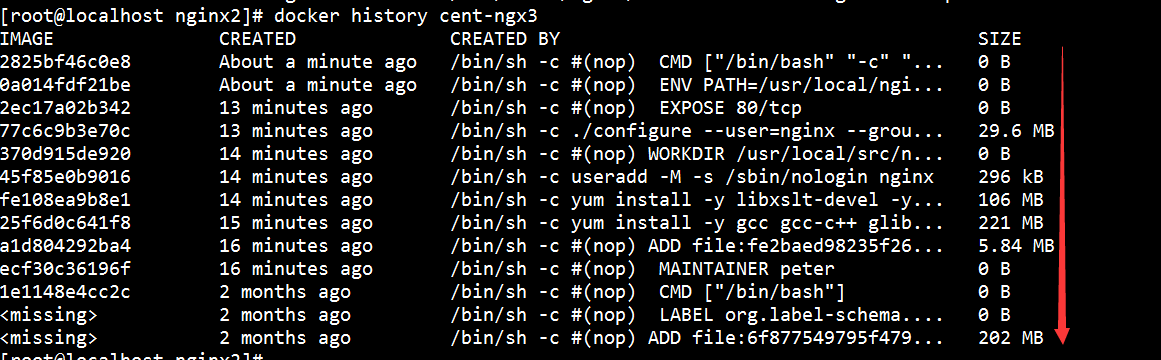
在上一版镜像的基础上，我们新加配置



执行：docker build -t cent-ngx3 .



查看镜像的历史，可看到比ngx2的镜像多了几个层



ngx3的镜像创建容器，已经不需要再指定cmd命令了

可执行命令自行校验：docker run -d --name ng2 cent-ngx3