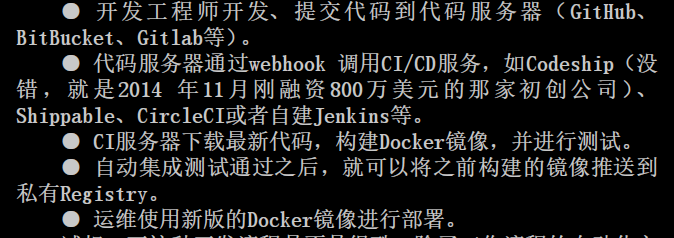
说实话，Docker算不上是什么全新的技术，它基于LXC（Linux Containers），使用AUFS，而这些都是已经存在很长世间并被广泛应用了得技术。但是运营PaaS服务的dotCloud公司将这些技术整合到一起，提供了简单易用的跨平台、可移植的容器解决方案。Docker最初由dotCloud公司在2013年发布。自发布以来，其发展速度之快超乎了很多人的想象，一路高歌猛进，2014年6月终于发布了1.0稳定版，而dotCloud在2013年10月干脆连公司名字也改为了Docker.Inc.。

Docker也可以被称为轻量级虚拟化技术。与传统的VM相比，它更轻量，启动速度更快，单台硬件上可以同时跑成百上千个容器，所以非常适合在业务高峰期通过启动大量容器进行横向扩展。

Docker是可移植的（或者说是跨平台的），可以在各种主流Linux发布版或者OS X以及Windows上使用。Java可以做到“一次编译，到处运行”，而Docker则可以称为“构建一次，在各平台上运行”（Build once, run anywhere）。

从这一点毫不夸张地说，Docker是革命性的，它重新定义了软件开发、测试、交付和部署的流程。我们交付的东西不再只是代码、配置文件、数据库定义等，而是整个应用程序运行环境：“OS+各种中间件、类库+应用程序代码”。

我们只需运行几条docker run就可以配置好开发环境，通过dockerfile或者docker hub与他人分享我们的镜像，与其他服务集成，进行开发流程的自动

Docker是为了Infrastructure as code而生的，通过dockerfile，镜像创建过程变得自动且可重复，还能进行版本管理。

Docker是为不可变基础设施而生的，对无状态服务的升级、部署将会更轻便更简单：我们无需再对他们的配置进行修改，只需要销毁这个服务并重建一个就好了。

# 第一章 入门篇

对Docker来说，他得益于现代Linux内核特性，如控件组、命名空间技术，容器和宿主机之间的隔离更加彻底，容器有独立的网络和存储栈，它还拥有自己的资源管理能力，使得一台宿主机中的多个容器可以友好的共存。

## 1 docker的目标

1、提供一个简单、轻量的建模方式；

Docker依赖于“写时复制”模型，使修改应用程序也非常迅速，可以说达到了“随心所至，代码即改”的境界。

2、职责的逻辑分离；

使用docker，开发人员只需要关心容器中运行的应用程序，而运维人员只需要关心如何管理容器。Docker设计的目的就是要加强开发人员写代码的开发环境与应用程序要部署的生产环境的一致性，从而降低那种“开发时一切都正常，肯定是运维的问题”的风险。

3、快速、高效的开发声明周期；

4、鼓励使用面向服务的架构

docker推荐单个容器只运行一个应用程序或进程，这样就形成了一个分布式的应用程序模型；当然，亦可在一个容器内运行多个进程的应用程序。

## 2 Docker组件

Docker客户端和服务器

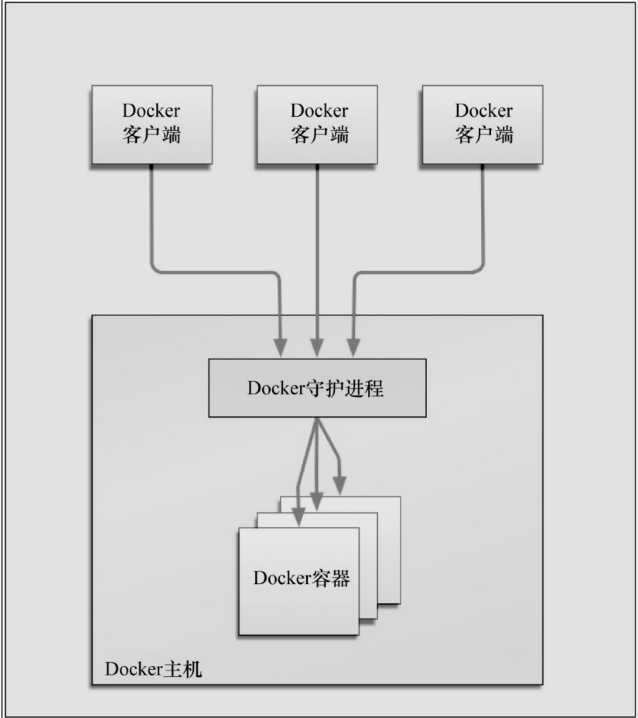
Docker镜像

Registry

Docker容器

### 2.1 Docker客户端和服务器

Docker是一个客户-服务器（C/S）架构的程序。Docker客户端只需向Docker服务器或守护进程发出请求，服务器或守护进程将完成所有工作并返回结果。Docker提供了一个命令行工具docker以及一整套RESTful API。你可以在同一台宿主机上运行Docker守护进程和客户端，也可以从本地的Docker客户端连接到运行在另一台宿主机上的远程Docker守护进程。下图描绘了Docker的架构。



### 2.2 Docker镜像

镜像是构建Docker世界的基石。用户基于镜像来运行自己的容器。镜像也是Docker声明周期中的“构建”部分。镜像是基于联合文件系统的一种层次的结构，由一系列指令一步一步构建出来/例如：

1、添加一个文件

2、执行一个命令

3、打开一个端口

也可以把镜像当作容器的“源代码”。镜像体积很小，非常“便携”，易于分享、存储和更新。

### 2.3 Registry

Docker用Registry来保存用户构建的镜像。Registry分为公共和私有两种。Docker公司运营的公共Registry叫做Docker Hub。用户可以在Docker Hub注册账号，分享并保存自己的镜像。

### 2.4 容器

Docker可以帮你构建和部署容器，你只需要把自己的应用程序或服务打包放进容器即可。我们刚刚提到，容器是基于镜像启动起来的，容器中可以运行一个或多个进程。我们可以认为，镜像是Docker生命周期中的构建或打包的阶段，而容器则是启动或执行阶段。

总结起来，Docker容器就是：

1、一个镜像格式

2、一系列标准的操作

3、一个执行环境

Docker借鉴了标准集装箱的概念。标准集装箱将货物运往世界各地，Docker将这个模型运用到自己的设计哲学中，唯一不同的是：集装箱运输货物，而Docker运输软件。

每个容器都包含一个软件镜像，也就是容器的“货物”，而且与真正的货物一样，容器里的软件镜像可以进行一些操作。例如，镜像可以被创建、启动、关闭、重启以及销毁。

和集装箱一样，Docker在执行上述操作时，并不关心容器中到底塞进了什么，他不管里满是Web服务器，还是数据库，或者是应用程序服务器什么的。所有容器都按照相同的方式将内容“装载”进去。

Docker也不关心你要把容器运到何方:你可以在自己的笔记本中构建容器，上传到Registry，然后下载到一个物理的或者虚拟的服务器来测试，再把容器部署到Amazon EC2主机的集群中去。像标准集装箱一样，Docker容器方便替换，可以叠加，抑郁分发，并尽量通用。

使用Docker，我们可以快速构建一个应用程序服务器、一个消息总线、一套试用工具、一个持续集成测试环境或者任意一种应用程序、服务或工具。我们可以在本地构建一个完整的测试环境，也可以为生产或开发快速复制一套复杂的应用程序栈。可以说，Docker的应用场景相当广泛。

## 3 安装、配置、启动Docker

王国维在《人间词话》谈到了治学经验，他说，“古今之成大事业、大学问者，必经过三种之境界。‘昨夜西风凋碧树，独上高楼，望尽天涯路’，此第一境也；‘衣带渐宽终不悔，为伊消得人憔悴’，此第二境也；‘众里寻他千百度，回头蓦见，那人正在灯火阑珊处’，此第三境也。”

此处用于Docker的入门，难免有些大题小做，但要领会精神。以吾师赵班长语，学习软件/工具/系统的步骤与此三句诗句有异曲同工之妙。

第一步为“安装/搭建”，此为入门阶段：安装（源码、包管理工具）；配置（网络、存储、性能、安全）；启动（手动，&放后台，nohup/screen；脚本）

第二步为“用好”，此为进阶阶段:掌握原理（便于选型）；方案；性能优化；

第三步为“关联”，此为架构师阶段。

以上三步，亦为笔者努力的方向；第三步亦为笔者的目标。

**昨夜西风凋碧树。独上高楼，望尽天涯路**

这词句出自晏殊的《蝶恋花》。在王国维此句中解成，做学问成大事业者，首先要有执着地追求，登高望远，瞰查路径，明确目标与方向，了解事物的概貌。

### 检查先决条件

运行64位CPU的计算机

1、内核版本

运行Linux 3.8或更高版本的内核

uname -a检查，本机使用CentOS 7。

[root@linux-node1 ~]# uname -a

Linux linux-node1.example.com **3.10.0-327.22.2.el7.x86\_64** #1 SMP Thu Jun 23 17:05:11 UTC 2016 x86\_64 x86\_64 x86\_64 GNU/Linux

2、检查Device Mapper

内核必须支持一种适合的存储驱动（storage driver），例如：

Device Manager；

AUFS；

vfs；

btrfs；

默认存储驱动通常是Device Mapper。自2.6.9版本的Linux内核开始已经集成了Device Mapper，并且提供了一个将块设备映射到高级虚拟设备的方法。Device Mapper支持‘自动精简配置“（thin-provisioning）概念，可以在一种文件系统存储多台虚拟设备（Docker镜像中的层）。因此，用Device Mapper作为Docker的存储驱动是再合适不过的了。

可以通过以下所示命令来确认是否已经安装。

[root@linux-node1 ~]# grep device-mapper /proc/devices

253 device-mapper

[root@linux-node1 ~]#

或

[root@linux-node1 ~]# ls -l /sys/class/misc/device-mapper/

total 0

-r--r--r-- 1 root root 4096 Aug 3 08:07 dev

drwxr-xr-x 2 root root 0 Aug 3 08:07 power

lrwxrwxrwx 1 root root 0 Aug 3 08:07 subsystem -> ../../../../class/misc

-rw-r--r-- 1 root root 4096 Aug 3 07:06 uevent

[root@linux-node1 ~]#

如果没有监测到Device Mapper，我们也可以试着安装device-mapper软件包：

[root@linux-node1 ~]# yum list |grep device-mapper

device-mapper.x86\_64 7:1.02.107-5.el7\_2.5 @updates

device-mapper-event.x86\_64 7:1.02.107-5.el7\_2.5 @updates

device-mapper-event-libs.x86\_64 7:1.02.107-5.el7\_2.5 @updates

device-mapper-libs.x86\_64 7:1.02.107-5.el7\_2.5 @updates

device-mapper-persistent-data.x86\_64 0.5.5-1.el7 @anaconda

device-mapper-devel.i686 7:1.02.107-5.el7\_2.5 updates

device-mapper-devel.x86\_64 7:1.02.107-5.el7\_2.5 updates

device-mapper-event-devel.i686 7:1.02.107-5.el7\_2.5 updates

device-mapper-event-devel.x86\_64 7:1.02.107-5.el7\_2.5 updates

device-mapper-event-libs.i686 7:1.02.107-5.el7\_2.5 updates

device-mapper-libs.i686 7:1.02.107-5.el7\_2.5 updates

device-mapper-multipath.x86\_64 0.4.9-85.el7\_2.5 updates

device-mapper-multipath-libs.i686 0.4.9-85.el7\_2.5 updates

device-mapper-multipath-libs.x86\_64 0.4.9-85.el7\_2.5 updates

device-mapper-multipath-sysvinit.x86\_64 0.4.9-85.el7\_2.5 updates

[root@linux-node1 ~]#

3、内核必须支持并开启cgroup、namespace功能

cgroup和namespace自2.6版本开始已经集成在Linux内核中了。2.6.38以后的内核对cgroup和namespace都提供了良好的支持，基本没什么bug。

### 1、安装Docker，检查

[root@linux-node2 ~]# yum install -y docker

[root@linux-node1 ~]# rpm -qa docker

docker-1.10.3-44.el7.centos.x86\_64

### 2、配置Docker 守护进程

安装完Docker后，我们需要确认Docker的守护进程是否运行。Docker以root权限运行它的守护进程，来处理普通用户无法完成的操作（如挂载文件系统）。Docker程序是Docker守护进程的客户端程序，同样也需要以root身份运行。

当Docker软件包安装完毕后，默认会立即启动Docker守护进程。守护进程监听/var/run/docker.sock这个UNIX套接字文件，来获取来自客户端的Docker请求。如果系统中存在名为docker的用户组的话，docker则会将该套接字文件的所有者设置为该用户组。这样，docker用户组的所有用户都可以直接运行Docker，而无须再使用sudo命令了。

[root@linux-node2 ~]# ll /var/run/docker.sock

srw-rw---- 1 root root 0 Aug 3 07:07 /var/run/docker.sock

[root@linux-node2 ~]# ll /var/run/docker.pid

-rw-r--r-- 1 root root 4 Aug 3 07:07 /var/run/docker.pid

[root@linux-node2 ~]#

WARNING：前面已经提到，尽管Docker用户组方便了Docker的使用，但它毕竟是一个安全隐患。因为docker用户组对Docker具有与root用户相同的权限，所以docker用户组中应该只能添加哪些确实需要使用Docker的用户和程序。

### 3、启动与操作

1、启动Docker、加入开机自启动项，检查

[root@linux-node1 ~]# systemctl status docker

[root@linux-node1 ~]# systemctl start docker

[root@linux-node1 ~]# systemctl status docker

[root@linux-node1 ~]# systemctl enable docker

Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to /usr/lib/systemd/system/docker.service.

2、镜像操作

前面提到过Docker的四大组件，镜像是Docker容器运行的“源代码“，用户基于镜像来运行自己的容器。

[root@linux-node1 ~]# docker search centos

INDEX NAME DESCRIPTION STARS OFFICIAL AUTOMATED

docker.io docker.io/centos The official build of CentOS. 2495 [OK]

[root@linux-node1 ~]# docker pull centos

[root@linux-node1 ~]# docker save centos >/opt/centos.tar.gz

[root@linux-node1 ~]# ls /opt/ -l

total 199556

-rw-r--r-- 1 root root 204341760 Aug 3 09:14 centos.tar.gz

[root@linux-node1 ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/centos latest 970633036444 4 days ago 196.7 MB

[root@linux-node1 ~]#

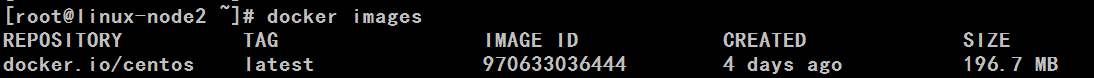
[root@linux-node2 ~]# docker load < /opt/centos.tar.gz

The image docker.io/centos:latest already exists, renaming the old one with ID sha256:05188b417f30f797c354961df144dd5516c7f730172903d5fdd01a25a5a05fa5 to empty string

[root@linux-node2 ~]# ll /opt/centos.tar.gz

-rw-r--r-- 1 root root 204341760 Aug 3 09:14 /opt/centos.tar.gz

[root@linux-node2 ~]#



3、容器操作

容器是基于镜像启动起来的，可以认为容器在Docker的生命周期中是启动或执行阶段。

[root@linux-node1 ~]# docker run docker.io/centos /bin/echo "sb"#创建一个镜像名为centos的容器并在里面运行

sb

[root@linux-node1 ~]# docker run centos /bin/echo "sb"

sb

#<--执行时您会心生疑问：

1、此种运行方式，效率太差，每次运行都要创建一个容器，执行完后自动退出

2、容器的名字没指定？

[root@linux-node1 ~]# docker run --name mydocker -t -i centos /bin/bash

**#解释一下这个命令，首先我们告诉Docker执行docker run命令，并指定了-i和-t两个命令行参数。-i标志标志保证容器中STDIN是开启的，尽管我们并没有附着到容器中。持久的标准输入是交互式shell的半边天，-t标志则是另外半边天，他告诉Docker为要创建的容器分配一个伪tty终端。这样新创建的容器才能提供一个交互式shell。若要在命令行下创建一个我们能与之进行交互的容器，而不是一个运行后台服务的容器，则这两个参数已经是最基本的参数了。**

[root@ee1377cfa732 /]# whoami #<--注意从此处开始主机名变了！

root

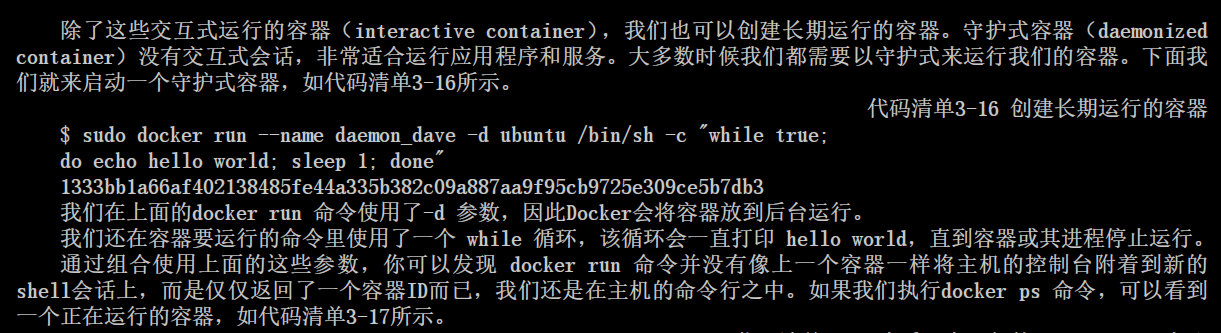
[root@ee1377cfa732 /]# echo nihao

nihao

[root@ee1377cfa732 /]# echo Bejing

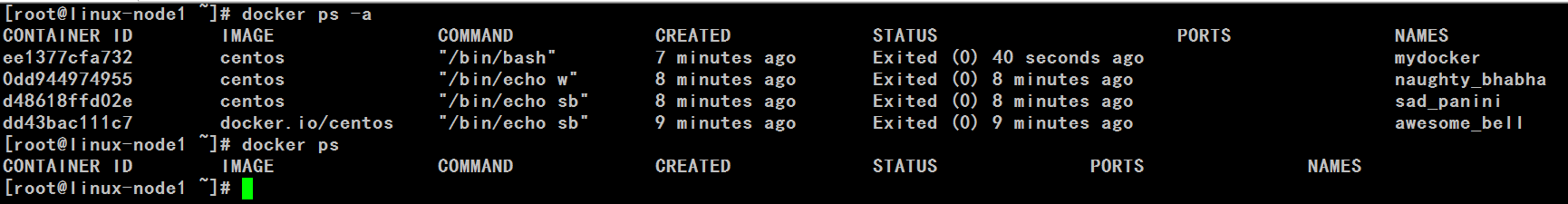
Bejing

[root@ee1377cfa732 /]#



[root@linux-node1 ~]# docker ps -a#查看所有docker容器，这里的-a去掉后只能查看活动的容器了。

[root@linux-node1 ~]# docker ps



这里我们发现，每执行一条命令都启动一个新的容器，看出第二种方式启动容器比较合理！

[root@linux-node1 ~]# docker ps -a#看看创建过的容器，明显后面三个是非交互式的方法创建的容器

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ee1377cfa732 centos "/bin/bash" 11 minutes ago Exited (0) 4 minutes ago mydocker

0dd944974955 centos "/bin/echo w" 12 minutes ago Exited (0) 12 minutes ago naughty\_bhabha

d48618ffd02e centos "/bin/echo sb" 12 minutes ago Exited (0) 12 minutes ago sad\_panini

dd43bac111c7 docker.io/centos "/bin/echo sb" 13 minutes ago Exited (0) 13 minutes ago awesome\_bell

[root@linux-node1 ~]# docker start mydocker#启动容器

mydocker

[root@linux-node1 ~]# docker inspect --format "{{.State.Pid}}" mydocker#获取正在运行分容器进程的PID

6657

[root@linux-node1 ~]# docker attach mydocker

**#进入正在运行的容器，此方法存在小bug，其他人通过attach进入相同的容器，你所有的操作对他人可见。并且在使用exit退出后容器就自动关闭了，坑爹啊！**

[root@ee1377cfa732 /]#

**#那么启动容器后，可以使用nsenter命令指定创建容器的pid登入容器操作，使用exit命令也不会关闭容器。**

[root@linux-node1 ~]# docker start mydocker

mydocker

[root@linux-node1 ~]# docker inspect --format "{{.State.Pid}}" mydocker

6816

[root@linux-node1 ~]# nsenter -t 6816 -m -u -i -p

[root@ee1377cfa732 /]# exit

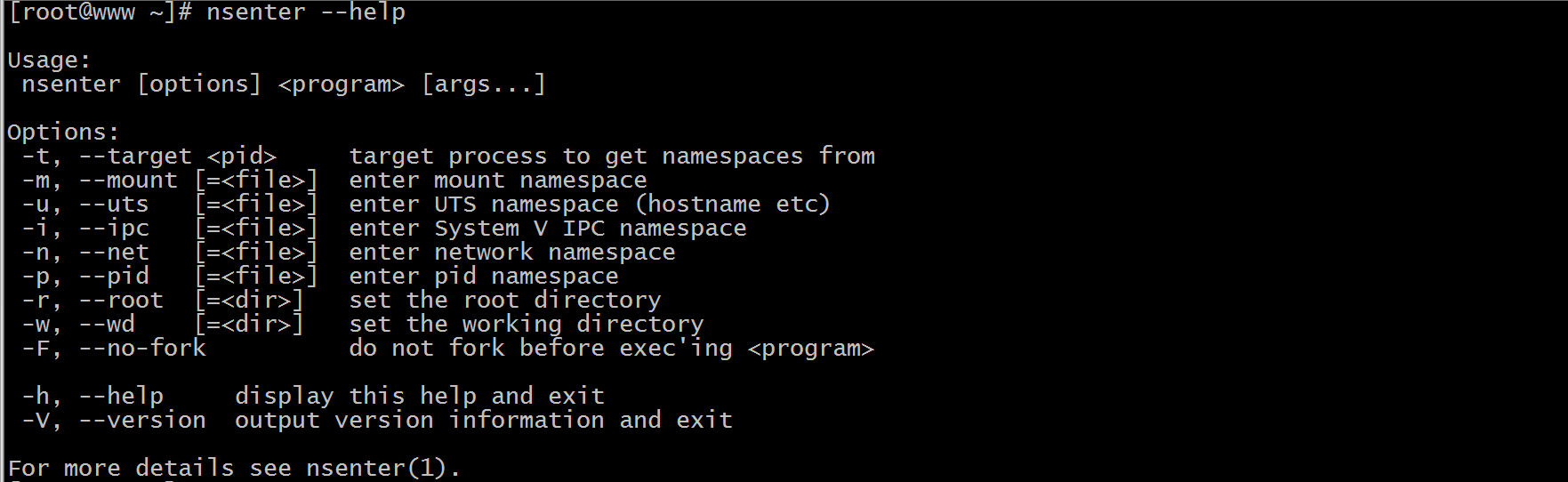
logout

[root@linux-node1 ~]# nsenter -t 6816 -m -u -i -p

[root@ee1377cfa732 /]# exit

logout

[root@linux-node1 ~]#



为了方便起见，可将上述内容写成脚本，内容如下：

#!/bin/bash

pid=`docker inspect --format "{{.State.Pid}}" $1`

nsenter -t $pid -m -u -i -n -p

[root@linux-node1 ~]# docker rm dd43bac111c7#删除容器（ID），-f可以强制删除正在运行的容器

dd43bac111c7

[root@linux-node1 ~]# docker rm d48618ffd02e

d48618ffd02e

[root@linux-node1 ~]# docker rm 0dd944974955

0dd944974955

既然如此，我们在做实验的时候创建了一堆容器，还需要删除多么麻烦！能不能退出就直接删除容器？当然可以！如下命令可以实现：

[root@linux-node1 ~]# docker run --rm --name testhehe -i -t centos

[root@37da2c9a3899 /]# exit

exit

[root@linux-node1 ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ee1377cfa732 centos "/bin/bash" 36 minutes ago Up 6 minutes mydocker

那么，既然可以删除容器，万一某一奇葩需求要求杀死所有正在运行的容器。考虑到Docker是一个客户-服务器（C/S）架构的程序，只要server一停就完事，想想会不会造成大量错误日志产生？还是温柔点，用kill！！！

[root@linux-node1 ~]# docker kill $(docker ps -a -q)#其中docker ps -a -q表示获取正在运行的所有容器的id

ee1377cfa732

[root@linux-node1 ~]#

## 4 Docker网络

安装Docker之后，Docker会默认创建docker0网卡，用于NAT转发。下面介绍 Docker的网络服务。

首先通过docker镜像仓库下载官方docker镜像：

[root@linux-node1 ~]# docker search nginx

INDEX NAME DESCRIPTION STARS OFFICIAL AUTOMATED

docker.io docker.io/nginx Official build of Nginx. 3667 [OK]

[root@linux-node1 ~]# docker pull nginx

[root@linux-node1 ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/centos latest 970633036444 4 days ago 196.7 MB

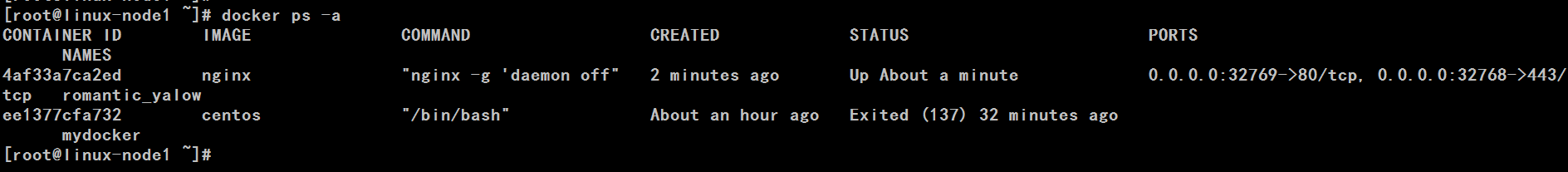
docker.io/nginx latest 0d409d33b27e 8 weeks ago 182.7 MB

### 4.1 随机映射

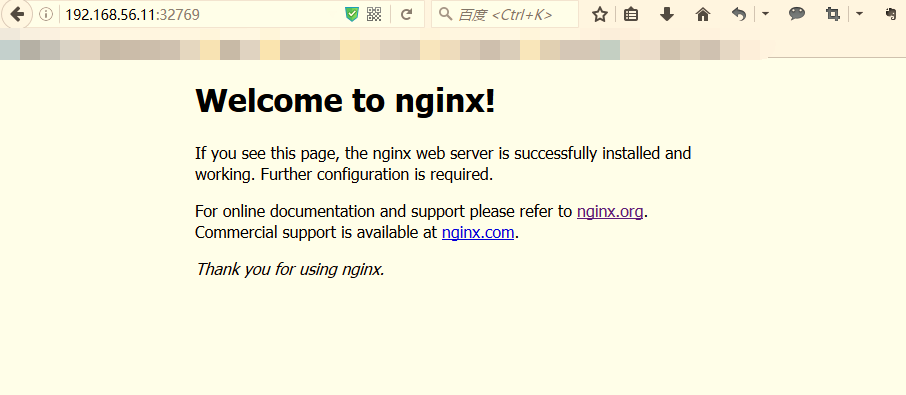
[root@linux-node1 ~]# docker run --rm -it -P nginx

#相比以前只多了一个新的参数-P（大写），P 就代表着随机映射。这条命令的意思就是启动一个镜像为nginx的容器，分配一个伪终端。并在物理机随机映射端口到容器的80端口，且在容器停止时删除。

通过下图，我们可以看出映射关系：



访问此随机端口，观察：



访问后一大波访问日志显示出来了：

192.168.56.1 - - [03/Aug/2016:14:40:10 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0" "-"

2016/08/03 14:40:10 [error] 6#6: \*1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 192.168.56.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.56.11:32769"

192.168.56.1 - - [03/Aug/2016:14:40:10 +0000] "GET /favicon.ico HTTP/1.1" 404 169 "-" "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0" "-"

2016/08/03 14:40:10 [error] 6#6: \*1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 192.168.56.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.56.11:32769"

192.168.56.1 - - [03/Aug/2016:14:40:10 +0000] "GET /favicon.ico HTTP/1.1" 404 169 "-" "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0" "-"

多么蛋疼？又没什么办法将访问日志保存到文件，而不是标准输出？

D:\Documents\Tencent Files\476782158\Image\Group\%`0GJCP895NI_Q2I0VH7KWD.jpgbingo！创建容器时再添加一个参数，执行后台模式！即创建守护式容器。

[root@linux-node1 ~]# docker run -d -P nginx

2087a8508dab813066fdff0071315e50f0b4eee850e39424773e1a0ebaeca301

[root@linux-node1 ~]# docker ps -a

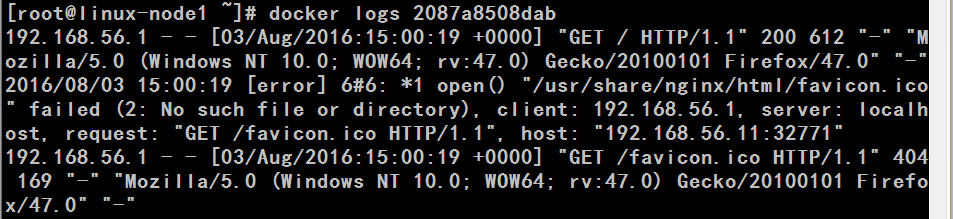
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2087a8508dab nginx "nginx -g 'daemon off" 7 seconds ago Up 6 seconds 0.0.0.0:32771->80/tcp, 0.0.0.0:32770->443/tcp compassionate\_euclid

使用下面的命令可以查看到相应的访问日志：

[root@linux-node1 ~]# docker logs 2087a8508dab

[root@linux-node1 ~]# docker logs compassionate\_euclid



### 4.2 指定映射

注意：指定映射选项-p为小写！随机端口映射选项-P为大写！

方法一、

docker rum --rm -it -p 81:80 nginx

#把物理机的81端口映射到容器里的80端口

方法二、

docker run --rm -it -p 192.168.56.20:81:80 nginx

#将192.168.56.20的81端口映射到容器的80端口，这个方法需要宿主机多IP

方法三、

docker run --rm -it -p 192.168.56.20::80 nginx

#将192.168.56.20的随机端口映射到容器的80端口（注意，这里的：为两个），至于这个需求主要针对宿主机多IP且需要随即映射端口

方法四、

docker run -rrm -it -p 81:80/tcp nginx

#将宿主机的81端口的tcp请求转发到容器的80端口

方法五、

docker run --rm -it -p 192.168.56.20:81:80/tcp nginx

#将192.168.56.20的81端口的tcp请求转发到容器的80端口。

除了TCP还有UDP请求一样可以转发，在此不一一列举。

## 5 Docker数据管理

Docker对数据的管理主要分为两种：数据卷和数据卷容器。用户在使用Docker的过程中，往往需要能**查看**容器内应用产生的数据，或者需要把容器内的数据进行**备份**，甚至多个容器之间进行**数据的共享**，这必然涉及容器的数据管理操作。

### 5.1 数据卷Data Volumes

数据卷是一个供容器使用的特殊目录，~~它绕过文件系统~~，它提供了很多有用的特性：

1、数据卷可以在容器间共享和重用

2、对数据卷的修改会立马生效

3、对数据卷的更新，不会影响镜像

4、卷会一直存在，直到没有容器使用。

数据卷的使用，类似于Linux下对目录或文件进行**mount操作。**

#### 1、容器内创建一个数据卷

[root@linux-node1 ~]# docker run --rm -it --name volume -v /data centos

**#创建一个镜像为centos名为volume的容器，并挂载到容器内部的/data目录，然后停止容器时自动销毁。选项-v可以在容器内创建一个数据卷，多次使用-v可以创建多个数据卷。**

**#注意：若目录不存在，Docker会自动创建**

[root@03f41dd3f078 /]# df -h

Filesystem Size Used Avail Use% Mounted on

/dev/mapper/docker-253:0-514387-75168316a1aba89a5db1533e2774fe85bb5162c51272da9797e8313fe78b5c5a 10G 240M 9.8G 3% /

tmpfs 489M 0 489M 0% /dev

tmpfs 489M 0 489M 0% /sys/fs/cgroup

**/dev/mapper/centos-root 18G 1.9G 16G 11% /data**

shm 64M 0 64M 0% /dev/shm

[root@03f41dd3f078 /]#

Linux系统中一切皆文件，那么大家不妨思考一下，这个挂载点的数据到底存在宿主机的什么位置?好！我们开始测试一下！

**#首先，在刚刚创建的volume容器中的/data目录创建一些测试文件：**

[root@03f41dd3f078 data]# touch test1.file test2.file

[root@03f41dd3f078 data]# echo 'Welcome to Dockers World' > test1.file

[root@03f41dd3f078 data]# cat test1.file

Welcome to Dockers World

[root@03f41dd3f078 data]# cat test2.file

**#宿主机上查询刚才创建的测试文件**

[root@linux-node1 mapper]# docker inspect volume|grep Source|awk -F "[:|,]" '{print $2}'

**#查看容器的详细信息，由于内容包含项过多，故过滤出需要的信息**

"/var/lib/docker/volumes/cc8139a4a2b2a1017b783cee505eb46ed66ba50a2b390d19d1a18c482d711938/\_data"

[root@linux-node1 mapper]# ll /var/lib/docker/volumes/cc8139a4a2b2a1017b783cee505eb46ed66ba50a2b390d19d1a18c482d711938/\_data -d

drwxr-xr-x 2 root root 40 Aug 3 13:45 /var/lib/docker/volumes/cc8139a4a2b2a1017b783cee505eb46ed66ba50a2b390d19d1a18c482d711938/\_data

[root@linux-node1 mapper]# cat /var/lib/docker/volumes/cc8139a4a2b2a1017b783cee505eb46ed66ba50a2b390d19d1a18c482d711938/\_data/test1.file

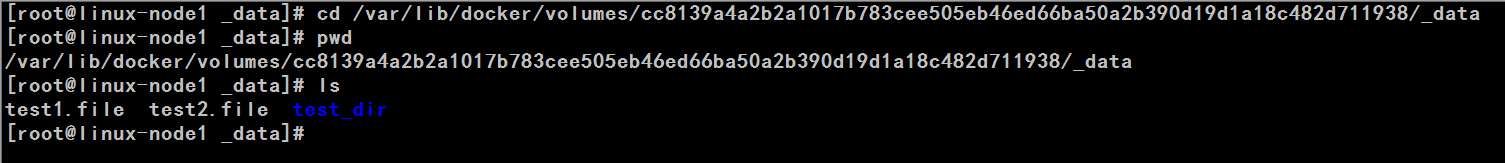
Welcome to Dockers World

[root@linux-node1 mapper]# cat /var/lib/docker/volumes/cc8139a4a2b2a1017b783cee505eb46ed66ba50a2b390d19d1a18c482d711938/\_data/test2.file

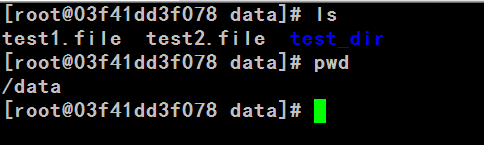
[root@linux-node1 mapper]#

[root@linux-node1 mapper]#

宿主机：



容器内：



#### 2、挂载一个主机上的目录作为数据卷

上一个标题我们发现，挂载到容器内的目录在物理机系统的实际存储位置；我们知道Docker可以实现多个容器之间的数据共享，那么如何实现容器与物理机之间共享数据呢？

Docker是人造的，办法还是有的！是的，聪明的你应该可以想到，“挂载物理机上的目录到容器内”。

[root@linux-node1 /]# mkdir test\_dir

[root@linux-node1 /]# docker run --rm --name volume -v /test\_dir/:/test\_dir\_docker:rw -it centos

#Docker挂载数据卷的默认权限是读写（rw）

[root@52beecc5e6b6 /]# ll

drwxr-xr-x 2 root root 6 Aug 3 18:11 test\_dir\_docker

指定Docker挂载数据卷的权限为ro：**加了：ro之后，容器内挂载的数据卷的数据就无法修改了**

[root@linux-node1 /]# docker run --rm --name volume -v /test\_dir/:/test\_dir\_docker2:ro -it centos

**#Docker挂载数据卷的默认权限是读写（rw），用户也可以通过，ro指定为只读：**

[root@850b8620fe73 /]# cd test\_dir\_docker2/

[root@850b8620fe73 test\_dir\_docker2]# mkdir 123

**mkdir: cannot create directory '123': Read-only file system**

[root@850b8620fe73 test\_dir\_docker2]#

注意：如果直接挂载一个文件到容器，使用文件编辑工具，包括vi或者sed --iniplace的时候，可能会造成文件inode的改变，从Docker1.1.0起，这会导致报错误信息。所以推荐的方式是直接挂载文件所在的目录。话说回来，我从没遇到过挂载文件的场景！！！

### 5.2 数据卷容器Data Volumes Containers

当用户需要在容器之间共享一些持续更新的数据，最简单的方式就是使用这个了——数据卷容器。简单的说，就是让一个容器访问另一个容器的卷，可以理解为一种特殊的容器。

专门用它提供数据卷供其它容器挂载使用的方法如下：

#首先创建一个数据卷容器volume，并在其中创建一个数据卷/vol挂载到此容器内

[root@linux-node1 /]# docker run --rm --name volume1 -v /test\_dir/:/vol -it centos

**#创建volume1，并挂在主机目录到容器内**

[root@dd2e44fc5a2f /]# ls

anaconda-post.log etc lib64 mnt root srv usr

bin home lost+found opt run sys var

dev lib media proc sbin tmp vol

**#创建容器volume2并挂载volume1中的挂载目录**

[root@linux-node1 /]# docker run --rm --name volume2 --volumes-from volume1 -it centos

#测试，在物理机上的目录创建测试文件，观察volume1与volume2挂载目录内的变化

**#物理机创建测试文件**

[root@linux-node1 ~]# cd /test\_dir/

[root@linux-node1 test\_dir]# touch hehe

[root@linux-node1 test\_dir]# touch haha

[root@linux-node1 test\_dir]#

**#观察volume1**

[root@dd2e44fc5a2f vol]# pwd

/vol

[root@dd2e44fc5a2f vol]#

[root@dd2e44fc5a2f vol]# ls

[root@dd2e44fc5a2f vol]#

[root@dd2e44fc5a2f vol]# ls

haha hehe

[root@dd2e44fc5a2f vol]#

**#观察volume2**

[root@922a948d1ffd vol]# pwd

/vol

[root@922a948d1ffd vol]# ls

[root@922a948d1ffd vol]#

[root@922a948d1ffd vol]# ls

haha hehe

[root@922a948d1ffd vol]#

注意：使用--volumes-from 参数所挂在数据卷的容器本身并不需要保持在运行状态。

正如前文所讲，如果删除了挂载的容器，数据卷并不会被自动删除。如果要删除一个数据卷，必须删除最后一个还挂在者它的容器时显式使用docker rm -v 命令来指定他那个是删除关联的容器。

### 5.3 利用数据卷容器迁移数据

可以利用数据卷容器对其中的数据卷进行备份、恢复，以实现数据的迁移。

#### 5.3.1 备份

#创建volume1容器，并在容器内挂载/data

[root@linux-node1 /]# docker run --rm --name volume1 -v /data -it centos

[root@ff027835077e /]# ls

anaconda-post.log dev lib media proc sbin tmp

bin etc lib64 mnt root srv usr

data home lost+found opt run sys var

#创建测试文件

[root@ff027835077e /]# cd data/

[root@ff027835077e data]# ls

[root@ff027835077e data]# touch {1..5}

[root@ff027835077e data]# ls

1 2 3 4 5

[root@ff027835077e data]#

#备份，使用如下命令来备份volume1数据卷容器内的数据卷

[root@linux-node1 /]# docker run --rm --name volume2 --volumes-from volume1 -v $(pwd):/backup -it centos

[root@1f2d7e552b68 /]# ls

anaconda-post.log data home lost+found opt run sys var

backup dev lib media proc sbin tmp

bin etc lib64 mnt root srv usr

[root@1f2d7e552b68 /]# ls data/

1 2 3 4 5

[root@1f2d7e552b68 /]# tar zcf /backup/backup.tar data

[root@linux-node1 /]# ls

**backup.tar** boot etc lib media opt root sbin sys tmp var

bin dev home lib64 mnt proc run srv test\_dir usr

[root@linux-node1 /]#

“docker run --rm --name volume2 --volumes-from volume1 -v $(pwd):/backup -it centos”，分析这个命令：首先利用centos镜像创建了一个容器volume2。使用--volumes-from volume1参数让volume2容器挂载volume1容器的数据卷；使用-v $(pwd):/backup参数来挂载本地的当前目录到volume2容器的/backup目录。

volume2容器启动后，使用了tar zcvf /backup/backup.tar /backup命令来将/data目录内容备份为容器内的/backup/backup.tar，及宿主机当前目录下的backup.tar。。

#### 5.3.2 恢复

如果要恢复数据到一个容器，可以按下面的思路进行操作。首先创建一个带有数据卷的容器volumeA:

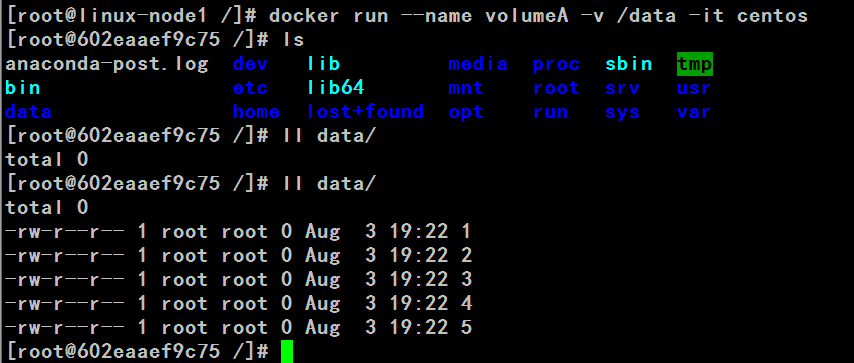
docker run --name volumeA -v /data -it centos

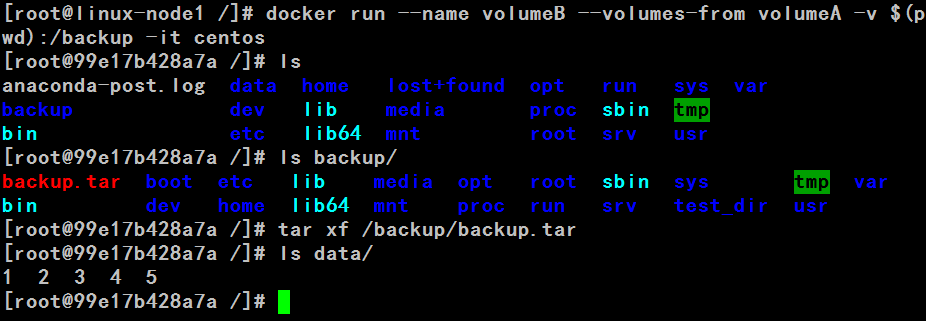
然后创建另一个新的容器，挂载volumeA的容器，并使用tar zxvf解压备份文件到所挂载的数据卷中即可：

docker run --name volumeB --volumes-from volumeA -v $(pwd):/backup -it centos

tar zxf /backup/backup.tar

展示效果如下所示：





### 5.4 小结

本章节介绍了通过数据卷和数据卷容器对容器内数据进行共享、备份和恢复等操作，通过这些机制，即使容器在运行过程中出现故障，用户也不必担心数据发生丢失，只需要快速地重新创建容器即可。

在生产环境中，笔者推荐再使用数据卷或数据卷容器之外，定期将主机的本地数据进行备份，或者使用支持容错的存储系统，包括RAID或分布式文件系统，如Ceph、GPFS、HDFS等.

现在笔者抛出一个问题，如何容器挂载多个数据卷，如何实现数据卷容器指定挂载数据卷？

## 6 构建镜像

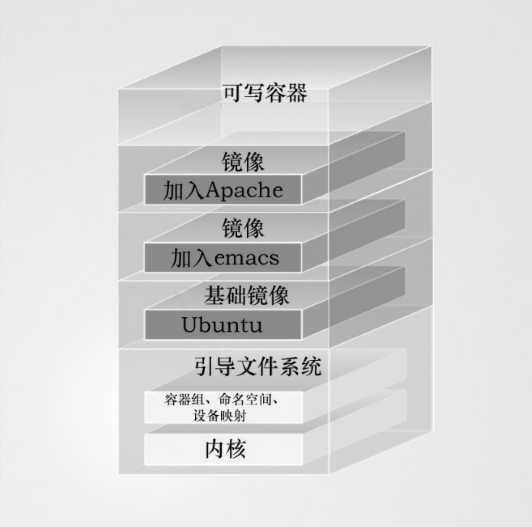
通过前面的学习，我们已经看到了如何拉取已经构建好的带有定制内容的Docker镜像，那么我们如何修改自己的镜像，并且更新和管理这些镜像呢？构建镜像有以下两种方法：

1、使用docker commit命令；

2、使用docker build命令和Dockerfile文件。

这里笔者不推荐使用docker commit命令，而应该使用更灵活、更强大的Dockerfile来构建Docker镜像。不过为了对Docker有一个更全面的了解，我们还是会先介绍如何使用docker commit构建Docker镜像。之后，我们将重点介绍Docker所推荐的镜像构建方法：编写Dockerfile之后使用docker build命令。

注意：一般来说，我们不是真正“创建”新景象，而是基于一个已有的基础镜像，如ubuntu或fedora等，构建新镜像而已。如果真的想从零构建一个全新的镜像，也可以参考<https://docs.docker.com/engine/userguide/eng-image/baseimages/>



### 6.1 commit命令创建镜像

创建Docker镜像的第一种方法是使用docker commit命令。可以将此想象为我们是在以往版本控制系统里提交变更。我们先创建一个容器，并在容器里做出修改，就像修改代码一样，最后再将修改提交为一个新镜像。

1、创建一个容器

[root@linux-node1 /]# docker run --name myserver2 -it centos

2、定制该镜像

[root@539b0c372e67 /]# rpm -ivh <http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm>

[root@539b0c372e67 /]# yum install -y vim

[root@539b0c372e67 /]# yum install -y nginx

[root@539b0c372e67 /]# vim /etc/nginx/nginx.conf

daemon off;

3、提交镜像

[root@linux-node1 /]# docker commit -m "new images for nginx" -a=”blackplum” myserver blackplum/nginx:v2

[root@linux-node1 /]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

blackplum/nginx v2 bbbd75a1c0b6 8 seconds ago 386.3 MB

我们来解释以下提交镜像的命令：docker commit -m "new images for nginx" --author=”blackplum” myserver blackplum/nginx:v2。在这条命令里，我们指定了更多的信息选项。首先-m选项用来添加描述镜像的备注信息，-a选项用来添加作者信息。接着指定了想要提交的容器ID。最后的“blackplum/nginx:v2”指定了镜像的用户名和仓库名，并为该镜像增加了一个v2标签。

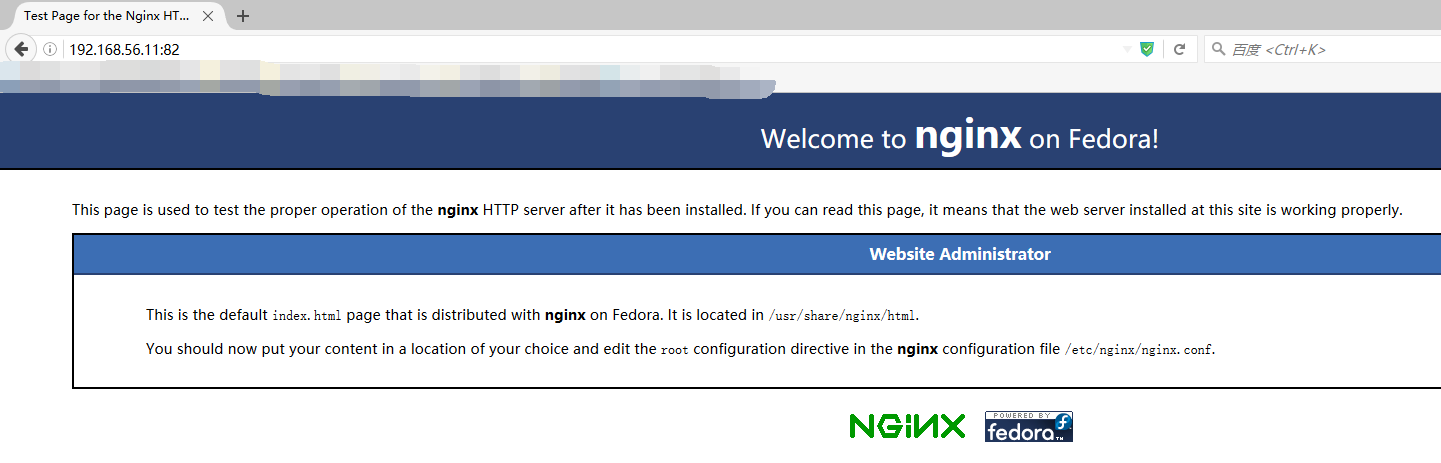
使用docker inspect命令可以查看新创建的镜像的相信信息。

4、运行镜像

[root@linux-node1 /]# docker run --name mynginx2 -dit -p 82:80 blackplum/nginx:v2 nginx

我们来解释一下这条指令。首先，docker run创建并运行容器mynginx2，-d选项将容器放到后台运行，-i选项采用交互模式运行容器，保持stin标准输入文件为打开状态，-t选项分配一个伪终端标识符，-i、-t这两个选项能够让我们通过终端进入容器中进行交互式命令行操作，如果不带这两个选项，那么我们将无法通过attach命令进入容器中，容器内的进程处于不可观测的状态；-p选项用来对外暴露指定端口82，可以理解为映射到容器内的80端口。

可以看出，我们用了完整标签blackplum/mynginx2:v2来指定这个镜像，访问效果如下图：



### 6.2 基于Dockerfile构建新镜像

本节介绍Dockerfile典型的基本结构及其支持的众多指令，并具体讲解通过这些指令来编写定制镜像的Dockerfile。

最后，会介绍使用Dockerfile创建镜像的过程。

#### 6.2.1 commit VS Dockerfile

我们并不推荐使用docker commit的方法构建镜像。相反，推荐使用被称为Dockerfile的定义文件和docker build命令来构建镜像。它与commit命令的区别在于其全部的客户化过程都是默认安装的，**无需人工交互**，这对专业的系统管理员、运维人员制作标准的中间件镜像、数据库镜像等非常有帮助。

Dockerfile使用基本的基于DSL语法的指令来构建一个Docker镜像，之后使用docker build命令基于该Dockerfile中的指令构建一个新的镜像。

#### 6.2.2 Dockerfile基本结构

一般而言，Dockerfile分为四部分：

1、基础镜像信息；

2、维护者信息；

3、镜像操作指令

4、容器启动时执行指令

下面开始我们的第一个Dockerfile，让我们从一个最简单的实例开始，在本地创建一个Dockerfile文件，内容如下：

FROM centos #指定基础镜像信息

MAINTAINER blackplum [www.webscaledb.cn](http://www.webscaledb.cn) #维护者信息

RUN rpm -ivh <http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm> #以下为镜像操作命令

RUN yum install -y nginx

ADD index.html /usr/share/nginx/html/index.html

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

EXPOSE 80

CMD ["nginx"] #容器启动时执行命令



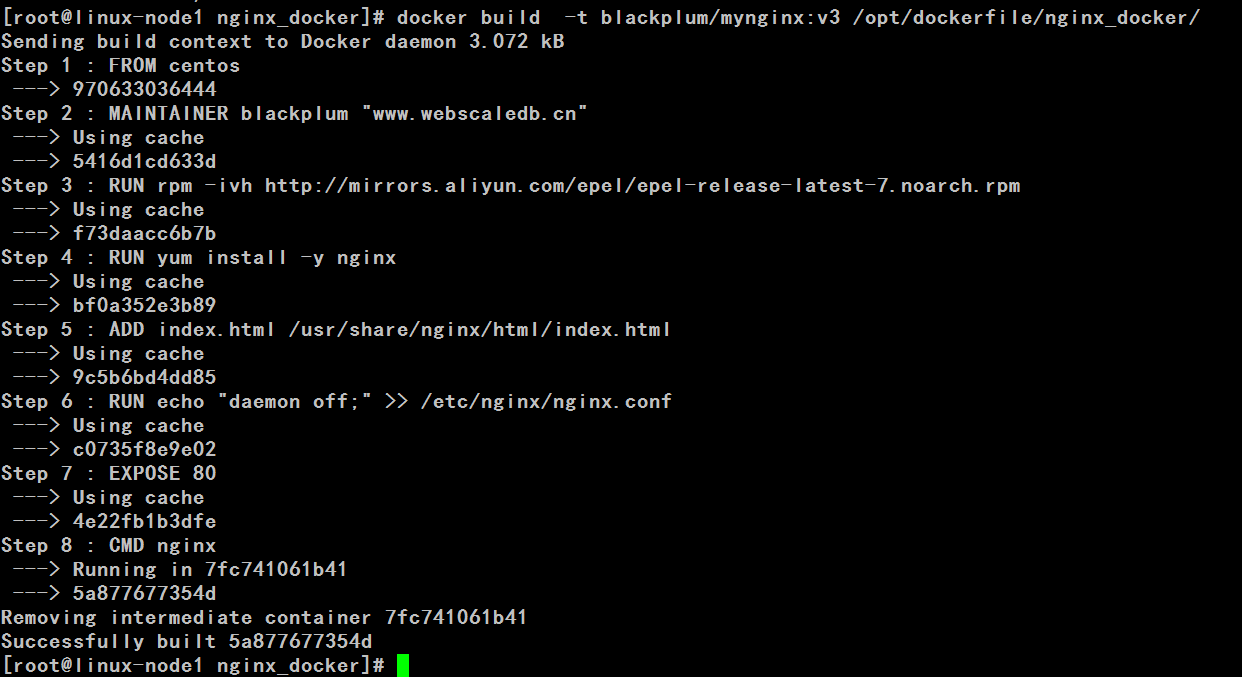
初次写Dockerfile时，你需要注意:

1、做好目录规划，以本文构建mynginx3镜像为例，指定目录为“/opt/dockerfile/nginx\_docker/”

2、在指定目录“/opt/dockerfile/nginx\_docker/”下创建名为“Dockerfile”的文件！不可以为其他名称。

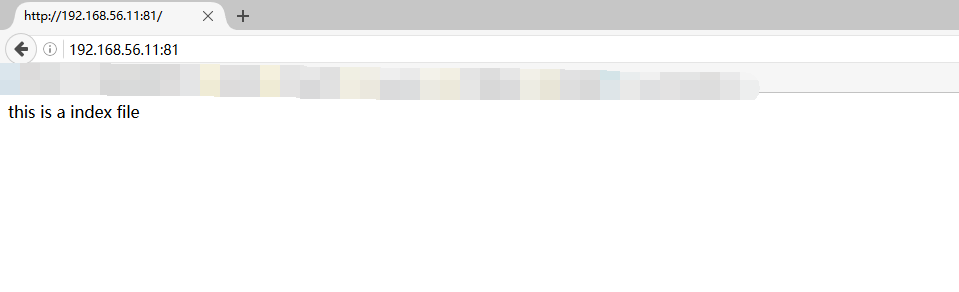
3、Dockerfile的指令集可以大写，接受小写，但还是建议统一大写，规范！

随后我们通过build命令从本地读取Dockerfile文件，生成一个镜像“blackplum/mynginx:v3”:



完成后启动一个容器：

docker run --name hehe -dit -p 192.168.56.11:81:80 blackplum/mynginx:v3



检查镜像是否已自动提交：



#### 6.2.3 Dockerfile 命令

以上面的Dockerfile为例：

FROM centos #指定基础镜像信息

MAINTAINER blackplum [www.webscaledb.cn](http://www.webscaledb.cn) #维护者信息

RUN rpm -ivh <http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm> #以下为镜像操作命令

RUN yum install -y nginx

ADD index.html /usr/share/nginx/html/index.html

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

EXPOSE 80

CMD ["nginx"] #容器启动时执行命令

**1、基本指令集**

a、FROM指令

*FROM <image>*

*FROM <image>:<tag>*

该指令用于指定基础镜像；

b、MAINTAINER指令

MAINTAINER <name>

该指令用于设定镜像制作者的信息

c、RUN指令

*RUN <command> #在shell终端中运行命令*

*RUN ["executable","param1","param2"] #使用exec执行。*

RUN指令与CMD指令很容易让人误解，RUN指令运行后所有的内容都将被持久化，例如安装package、修改文件等，它是为了修改镜像数据而存在的，进程执行完毕后就会退出，不会保留在启动的容器中。

d、CMD指令

*CMD ["executable","param1","param2"] #使用exec执行，推荐方式*

*CMD command param1 param2 #在/bin/sh中执行，提供给需要交互 的应用*

*CMD ["param1","param2"]#提供给EVTRYPOINT的默认参数*

CMD指令时镜像设置的默认指令，它与运行的进程相关。

e、EXPOSE指令

*EXPOSE <port> [<port>….] #例如：EXPOSE 22 80 8443*

EXPOSE指令告诉容器在启动时将哪些端口暴露给外部。这里需要读者朋友们注意的是，即便在Dockerfile中设置了该选项，在run命令执行时亦然要设置-p选项的映射关系。自启动容器是指定-P Docker主机会自动分配一个端口转发到指定的端口。

以上为最基本的5条指令，在这个过程中我们使用了build指令进行镜像的生成。

build命令的使用方式如下：

*docker build [OPTIONS] PATH | URL*

|  |  |
| --- | --- |
| 选项 | 说明 |
| -t，--tag=”” | 在镜像生成成功后，应用到镜像上的名称及标签 |
| -q，--quite | 构建过程不显示过程信息，默认是显示的 |
| --rm=true | 成功构建后删除中间所有的临时容器 |
| --force-rm | 强制删除所有中间容器的数据，即便没有成功 |
| --no-cache | 在构建过程中使用缓存 |

PATH、URL指定了Dockerfile所在位置，它会将这个目录下的所有文件传送到Docker daemon中。

**2、环境指令集**

a、ENV指令

ENV指令用来设置环境变量：

*ENV <key> <value>*

该指令将<value>值设置到<key>中，随后的Dockerfile的所有RUN命令可以读取环境变量值。

b、WORKDIR指令

该指令设置当前环境的工作目录，主要用来为RUN/ CMD命令查找可执行文件的目录。指令可以在Dockerfile文件中多次使用，也支持使用相对路径。

c、USER指令

USER指令用来设置在Dockerfile脚本中运行RUN进程的用户，可以是用户名，也可以是UID

https://segmentfault.com/a/1190000004527476

**3、数据指令集**

a、VOLUME指令

*VOLUME ["/data"]*

VOLUME指令和RUN命令中的-v选项类似，在容器中创建一个mount点，从外部挂载一个卷，可以是外部存储或其他容器

b、ADD指令

ADD指令用来将本地文件复制到镜像中：

*ADD <src> <dest>*

ADD指令中的<src>所指的文件或者目录必须是相对于build命令的PATH或者URL路径的，<dest>是镜像中的绝对路径，文件将复制到镜像中的这个位置。

c、COPY指令

COPY指令和ADD基本一样，最大的区别在于COPY只支持本地文件中复制，也就是说如果你需要从URL外部或者stdin标准输入中复制文件，那么将不受COPY指令支持。

4、ENTRYPOINT指令

RUN、CMD和ENTRYPOINT这三条指令比较容器混淆，如上问所述，RUN指令命令用于在基础镜像上修改文件数据，将内容持久化而构建新镜像。在Dockerfile中有多条RUN，在执行完成任务后便退出，而CMD和ENTRYPOINT只有一条。

CMD是镜像生成后默认启动的命令。ENTRYPOINT与CMD类似，也是容器默认执行的命令，但ENTRYPOINT让这个镜像更像一个独立的可执行文件，而在RUN命令之后输入的内容仅作为参数传入。

*ENTRYPOINT ["executable","param1","param2"]*

*ENTRYPOINT command param1 param2 #shell中执行*

每个Dockerfile中只能有一个ENTRYPONIT，当指定多个ENTRYPOINT时，只有最后一个生效。