Docker技术资料

[Docker技术资料 1](#_Toc451524822)

[第1章. docker简介 1](#_Toc451524823)

[1.1 docker的特点 1](#_Toc451524824)

[1.2 docker的组件 1](#_Toc451524825)

[1.3 docker使用场景 3](#_Toc451524826)

[1.4 docker技术原理 4](#_Toc451524827)

[第2章. docker安装 5](#_Toc451524828)

[2.1 docker安装准备条件 5](#_Toc451524829)

[2.1.1 内核准备 5](#_Toc451524830)

[2.1.2 软件准备 10](#_Toc451524831)

[2.2 安装docker 10](#_Toc451524832)

[2.2.1 centos6.5下安装docker 10](#_Toc451524833)

[2.2.2 centos7.0下安装docker 11](#_Toc451524834)

[2.2.3 docker脚本和二进制安装 11](#_Toc451524835)

[2.2.4 docker升级安装 12](#_Toc451524836)

[2.2.5 docker守护进程配置 13](#_Toc451524837)

[第3章. docker镜像和仓库 13](#_Toc451524838)

[3.1 什么是docker镜像 13](#_Toc451524839)

[3.2 docker镜像管理及构建 14](#_Toc451524840)

[3.2.1 列出镜像 14](#_Toc451524841)

[3.2.3 拉取镜像 14](#_Toc451524842)

[3.2.4 查找镜像 15](#_Toc451524843)

[3.2.5 删除镜像 15](#_Toc451524844)

[3.2.6 构建镜像 16](#_Toc451524845)

[3.2.6.1 基于Dockerfile构建镜像 16](#_Toc451524846)

[3.2.6.2 基于新镜像启动容器 18](#_Toc451524847)

[3.2.6.3 基于commit制作镜像 19](#_Toc451524848)

[3.3 docker镜像存储位置 19](#_Toc451524849)

[3.4 docker私有仓库管理 20](#_Toc451524850)

[3.4.1 docker registry 1.X系列搭建 20](#_Toc451524851)

[3.4.2 docker registry 2.X系列搭建(普通模式) 22](#_Toc451524852)

[3.4.3 docker registry 2.X系列搭建(基于用户验证) 23](#_Toc451524853)

[3.5 使用harbor来管理仓库镜像 24](#_Toc451524854)

[3.5.2 harbor简介 24](#_Toc451524855)

[3.5.2 harbor部署 24](#_Toc451524856)

[3.5.3 上传镜像到harbor 26](#_Toc451524857)

[第4章. docker容器管理 27](#_Toc451524858)

[4.1 创建第一个容器 27](#_Toc451524859)

[4.2 命名容器 28](#_Toc451524860)

[4.3 启动容器 29](#_Toc451524861)

[4.4 进入容器 29](#_Toc451524862)

[4.5 删除容器 29](#_Toc451524863)

[4.6 创建守护式容器 30](#_Toc451524864)

[4.7 查看容器内的进程 30](#_Toc451524865)

[4.8 交互式管理容器 31](#_Toc451524866)

[4.9 查看容器详细信息 31](#_Toc451524867)

[4.10 数据卷容器管理 32](#_Toc451524868)

[4.10.1 数据卷 32](#_Toc451524869)

[4.10.2 数据卷容器 32](#_Toc451524870)

[4.10.3 数据卷容器维护 33](#_Toc451524871)

[4.11 容器访问 34](#_Toc451524872)

[4.11.1 外部访问容器 34](#_Toc451524873)

[4.11.2 容器互联 34](#_Toc451524874)

[第5章. 使用Dockerfile构建镜像 36](#_Toc451524875)

[5.1 Dockerfile 36](#_Toc451524876)

[5.1.1 CMD 36](#_Toc451524877)

[5.1.2 ENTRYPOINT 36](#_Toc451524878)

[5.1.3 WORKDIR 37](#_Toc451524879)

[5.1.4 ENV 37](#_Toc451524880)

[5.1.5 USER 38](#_Toc451524881)

[5.1.6 VOLUME 38](#_Toc451524882)

[5.1.7 ADD 39](#_Toc451524883)

[5.1.8 COPY 39](#_Toc451524884)

[5.1.9 ONBUILD 39](#_Toc451524885)

[5.1.10 EXPOSE 41](#_Toc451524886)

[5.2 Dockerfile构建镜像 41](#_Toc451524887)

[5.2.1 非守护式方式构建容器 41](#_Toc451524888)

[5.2.2 守护式方式构建容器 43](#_Toc451524889)

[5.2.2.1 构建tomcat镜像 43](#_Toc451524890)

[5.2.2.2 构建MySQL镜像 44](#_Toc451524891)

[第6章. docker网络介绍 46](#_Toc451524892)

[6.1 docker网络模式 46](#_Toc451524893)

[6.1.1 host模式 46](#_Toc451524894)

[6.1.2 container模式 47](#_Toc451524895)

[6.1.3 none模式 47](#_Toc451524896)

[6.1.4 bridge模式 48](#_Toc451524897)

[6.2 docker第三方网络工具 49](#_Toc451524898)

[6.2.1 docker网络之pipework 49](#_Toc451524899)

[6.2.1.1 docker容器配置本地网络环境 49](#_Toc451524900)

[6.2.1.2 docker单主机容器VLAN划分 51](#_Toc451524901)

[6.1.1.3 docker多主机容器VLAN划分 52](#_Toc451524902)

[6.2.2 docker网络之weave 54](#_Toc451524903)

[6.2.2.1 weave使用场景 54](#_Toc451524904)

[6.2.2.2 weave部署 55](#_Toc451524905)

[6.3 docker网络之DNS配置 57](#_Toc451524906)

[第7章. docker compose 58](#_Toc451524907)

[7.1 compose介绍 58](#_Toc451524908)

[7.2 compose案例实战 58](#_Toc451524909)

[附录一 资料参考 61](#_Toc451524910)

[附录二 docker命令 62](#_Toc451524911)

第1章. docker简介

docker是一个能够把开发的应用程序自动部署到容器的开源引擎。由docker公司团队编写，基于apache2.0开源授权协议发行。那么docker有什么特别之处呢？docker在虚拟化的容器执行环境中增加了一个应用程序部署引擎。该引擎的目标就提供一个轻量、快速的环境，能够运行开发者的程序，并方便高效地将程序从开发者的笔记本部署到测试环境，然后再部署到生产环境。docker极其简洁，它所需的全部环境只是一台仅仅安装了兼容版本的linux内核。

1.1 docker的特点

**（1）提供一个简单、轻量的建模方式**

docker能够快速构建一个程序运行的环境，用户只需要几分钟就可以把自己的程序“docker化”。docker依赖于“写时复制”（copry-on-write）模型，使修改应用程序也非常迅速。由于去除了管理程序的开销，docker容器拥有很高的性能，同时同一台宿主机中也可以运行更多的容器，使用户可以尽可能充分利用系统资源。

**（2）职责的逻辑分离**

使用docker，开发人员只需要关系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一致性。

**（3）快速、高效的开发生命周期**

docker的目标之一就是缩短代码从开发、测试到部署、上线运行的周期，让你的应用程序具备可移植性，易于构建，并易于协作。

**（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过程序或进程，这样就形成了一个分布式的应用程序模型，在这种模型下，应用程序或服务都可以表示一系列内部互联的容器，从而使分布式部署的应用程序，扩展或调试应用程序都变得非常简单，同时也提高了程序运行的性能。当然一个容器也可以跑多个进程的应用。

1.2 docker的组件

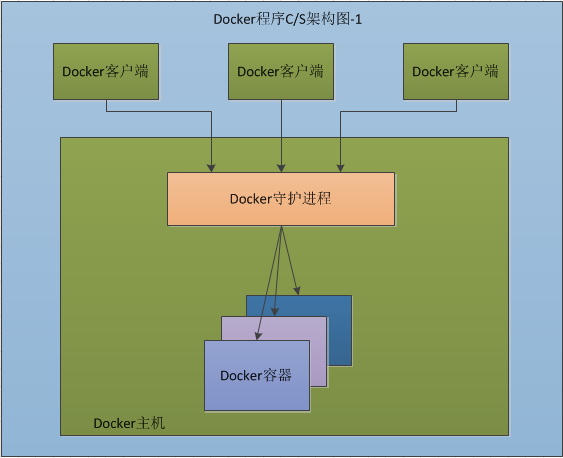
首先来了解docker由哪些组件构成，它主要有以下组件：

* docker客户端和服务器；
* docker镜像；
* registry；
* docker容器

下面来介绍这些组件的功能。

**（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RESTfulAPI。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。下图为docker架构图。



**（2）docker镜像**

镜像是构建docker世界的的基石。用户基于镜像来运行自己的容器。镜像也是docker生命周期中的“构建”部分。镜像是基于联合（union）文件系统的一种层式的结构，由一系列指令一步步构建出来。例如：

添加一个文件，执行一个命令，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。

**（3）Registry**

docker用registry来保存用户构建的镜像。registry分为公有和私有两种。docker公司运营的公共registry叫做registry hub。用户可以在docker hub注册账号，分享并保存自己的镜像。docker hub分享的镜像很多，需要nginx web服务器的docker镜像，或者mysql数据库的镜像。这些上面都有。

也可以假设私有自己的私有registry，私有registry可以受到防火墙的保护，将镜像保存在防火墙后面。以满足一些组织的特殊需求。

**（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。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生命周期的构建或打包阶段，而容器则是启动或执行阶段。

总结起来讲，docker容器就是：

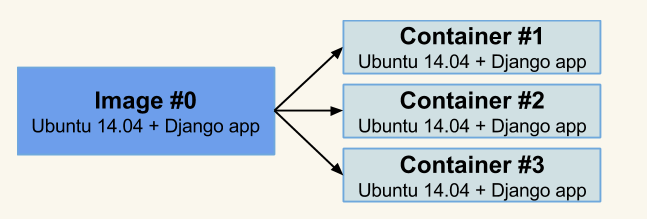
* 一个镜像格式；
* 一系列标准的操作；
* 一个执行环节。

docker借鉴了标准集装箱的概念。标准集装箱件货物运往世界各地，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。

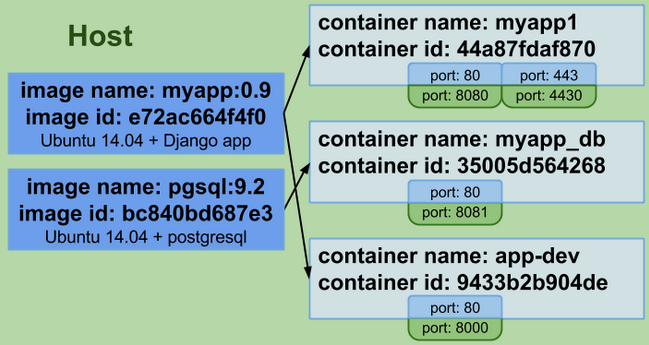
每个容器都包含一个软件镜像，也就是容器的“货物”，而且与真正的货物一样，容器的软件镜像可以进行一些操作。例如：镜像可以被创建、启动、关闭、重启以及销毁。

docker容器方便替换，可以叠加，易于分发，并且尽量通用。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、一套实用工具、一个持续集成（continuous integration CI）测试环境或任意一种应用程序、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。容器与虚拟机一样，是隔离的（有一点要注意，我稍后会讨论到）。它们也拥有一个唯一ID和唯一的供人阅读的名字。容器对外公开服务是必要的，因此Docker允许公开容器的特定端口。

下面是镜像和容器之间的关系



与虚拟机相比，容器有一个很大的差异，它们被设计用来运行单进程，无法很好地模拟一个完整的环境（如果那是你需要的，请看看LXC）。你可能会尝试运行runit或supervisord实例来启动多个进程，这是不推荐做法。



单进程与多进程之争非常精彩。你应该知道的是，Docker设计者极力推崇“一个容器一个进程的方式”，如果你要选择在一个容器中运行多个进程，那唯一情况是：出于调试目的，运行类似ssh的东西来访问运行中的容器，不过docker exec命令解决了这个问题。

1.3 docker使用场景

docker可以为各种测试提供很好的沙盒环境，并且容器本身就具有“标准”的特征，非常适合为服务创建构建块。docker主要应用场景如下：

* 加速本地开发和构建流程，使其更加高效、更加轻量化。本地开发人员可以构建、运行并分享docker容器。容器可以在开发环境种构建，然后轻松地提交到测试环境，并最终进入生产环境。
* 能够让独立服务或应用程序在不同的环境中，得到相同的运行结果。这一点在面向服务的架构和重度依赖微型服务的部署中尤其实用。
* 用docker创建隔离的环境来进行测试。例如，用jenkins CI这样的持续集成工具启动一个用于测试的容器。
* docker可以让开发者先在本机上构建一个复杂的程序或架构来进行测试，而不是一开始就在生产环境部署、测试。
* 构建一个多用户的平台即服务（PaaS）基础设施。
* 为开发、测试提供一个轻量级的独立沙盒环境，或者用于技术教学。
* 提供软件即服务（SaaS）应用程序，如Memcached即服务。
* 高性能、超大规模的宿主机部署。

docker生态环境构建项目表：<http://blog.docker.com/2013/07/docker-projects-from-the-docker-community/>

1.4 docker技术原理

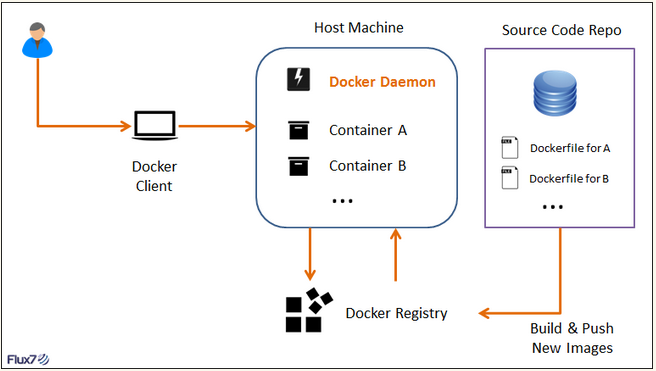
docker可以运行于任何安装了现代linux内核的x64主机上，推荐使用的内核版本是3.8以上。docker开销比较低，可以用于服务器、台式机或笔记本。

一个原生的linux容器格式，docker中成为libcontainer，或者很流行的容器平台lxc。libcontainer格式现在就是docker容器的默认格式。

linux内核的命名空间(namespace)，用于隔离文件系统、进程和网络。docker的技术组件有以下几种。

* 文件系统隔离：每个容器都有自己的root文件系统；
* 进程隔离：每个容器都运行在自己的进程环境中；
* 网络隔离：容器间的虚拟网络接口和IP地址都是分开的；
* 资源隔离和分组：使用cgroups（control group，linux的内核特性之一）将CPU和内存之类的资源独立分配给每个docker容器。
* 写时复制：文件系统都是通过写时复制创建的，这就意味着文件系统是分层的、快速的，而且占用的磁盘空间更小。
* 日志：容器产生的STDOUT、STDERR和STDIN这些IO流都会被收集并计入日志，用来进行日志分析和故障排错。
* 交互式shell：用户可以创建一个伪tty终端，将其连接到STDIN，为容器提供一个交互式的shell。

docker是C/S架构的程序，工作架构如下图：



第2章. docker安装

Docker安装快速而且简单，目前docker支持平台很广泛，包括Ubuntu、RHEL、Debian、Centos、Fedora、Oracle Linux等衍生系统和相关的发行版。Docker团队推荐在Ubuntu或RHEL宿主机中部署Docker，这两个发行版中直接提供了可安装的软件包。

2.1 docker安装准备条件

Docker要求的条件具体如下：

* 运行64位CPU架构的计算机，docker目前不支持32位CPU。
* 运行Linux3.8或更高版本的内核。
* 内核必须支持一种适合的存储驱动（storage driver）：Device Manager、AUFS、vfs、btrfs、默认存储驱动通常是Device Manager。
* 内核必须支持开启cgroup和命名空间(namespace)功能。

2.1.1 内核准备

1）查看本机内核为2.6.32-279，这里将升级到3.10.80版本

[root@node2 tools]# uname -a

Linux node2.mfs.com 2.6.32-279.el6.x86\_64 #1 SMP Wed Jun 13 18:24:36 EDT 2012 x86\_64 x86\_64 x86\_64 GNU/Linux

2）安装内核升级所需软件

yum grouplist -y

yum groupinstall "Development Tools" -y

yum install ncurses-devel -y

yum install hmaccalc zlib-devel binutils-devel elfutils-libelf-devel -y

3）安装docker需要的内核模块

wget <https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.10.80.tar.xz>

tar -xf linux-3.10.80.tar.xz

cd linux-3.10.80

cp /boot/config-2.6.32-431.el6.x86\_64 .config #🡸复制原内核文件一份

使用check-config.sh脚本来检查是否包含docker所需的模块。如果有missing分，则需要再次make menuconfig添加缺失部分模块。下面提取了缺失部分模块，红色部分为必须安装，橙色部分可选。

[root@docker01 linux-3.10.80]# sh check-config.sh .config |grep missing

warning: .config does not exist, searching other paths for kernel config...

- CONFIG\_NF\_NAT\_IPV4: missing #🡸必须安装模块

- CONFIG\_NETFILTER\_XT\_MATCH\_ADDRTYPE: missing #🡸必须安装模块

- CONFIG\_MEMCG\_SWAP: missing

- CONFIG\_MEMCG\_SWAP\_ENABLED: missing

- CONFIG\_AUFS\_FS: missing

- CONFIG\_OVERLAY\_FS: missing

- /dev/zfs: missing

- zfs command: missing

- zpool command: missing

解决CONFIG\_NF\_NAT\_IPV4和CONFIG\_NETFILTER\_XT\_MATCH\_ADDRTYPE这两个必装模块问题，执行 make menuconfig

cd linux-3.10.80

make menuconfig

Networking support ---> Networking options ---> Network packet filtering framework --->

Core Netfilter Configuration --->[M]“addrtype” address type match support

IP: Netfilter Configuration --->[M] IP tables support (required for filtering/masq/NAT)

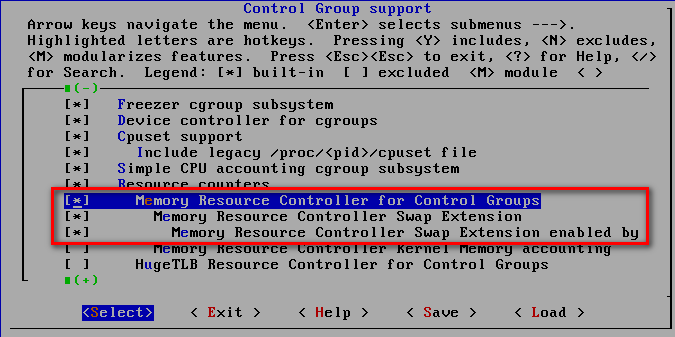
--->[M] Ipv4 NAT

--->[M] MASQUERADE target support

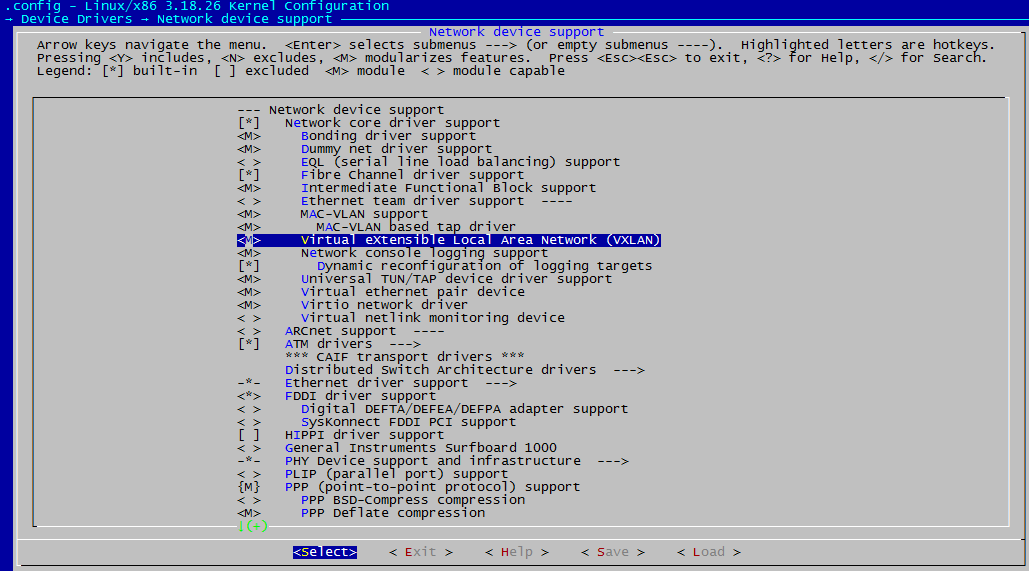
解决CONFIG\_MEMCG\_SWAP和CONFIG\_MEMCG\_SWAP\_ENABLED，不安装这两个模块docker也能使用，只是启动的时候会出现“cannot mount memory to /cgroup/memory: No such file or directory”报错问题。

make menuconfig

General Setup ---> Control Group Support ---> 选中下面三个,然后保存



加入VxLAN支持，OpenvSwitch需要此模块支持。



简单方式（推荐）

最简单的方式是通过check-config.sh来检查.config，如果发现有missing部分，可以直接编辑.config文件将出现missing的部分用下面的方式直接加入进去：“CONFIG\_NF\_NAT\_IPV4=y”表示启用；“=m”表示模块启用，包含相关套件。下面提取了需要加入的部分。

#复制原内核文件到编译目录

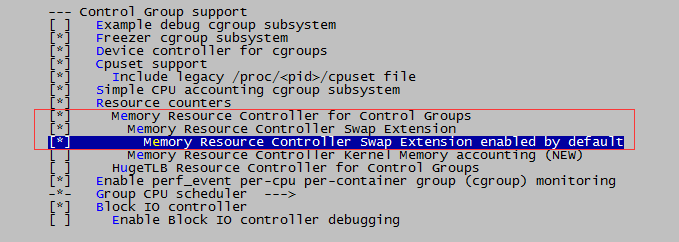
cd linux-3.10.80

cp /boot/config-2.6.32-431.el6.x86\_64 .config

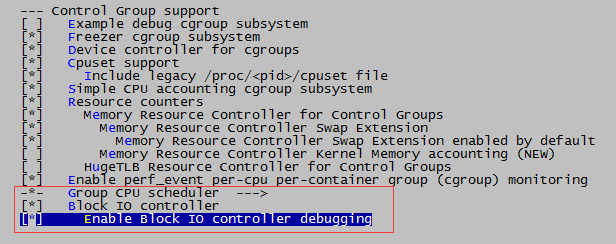
#自动整理.config

sh -c 'yes "" | make oldconfig'

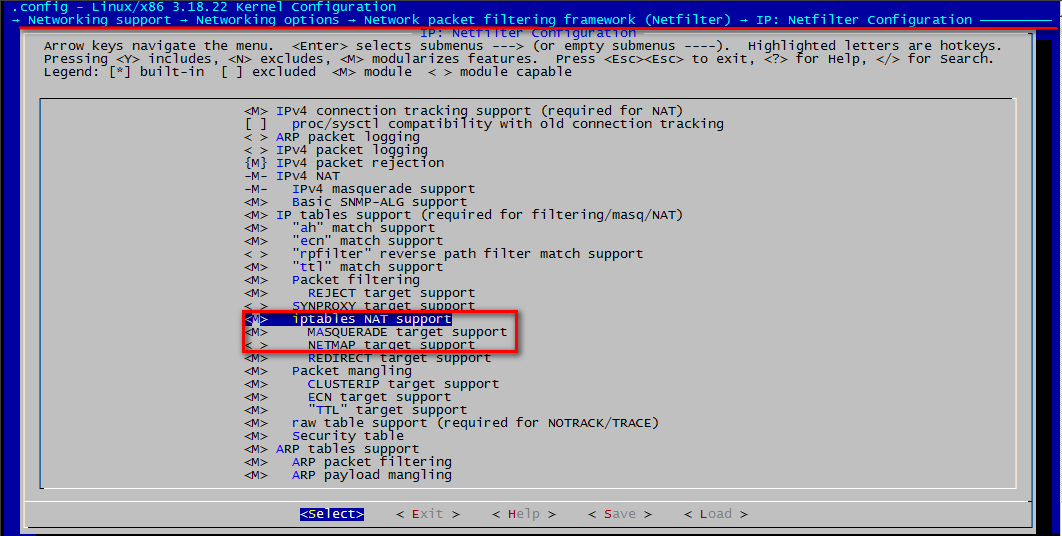
#运行make menuconfig，选中下面红框勾选部分,加入内存资源控制模块。



#加入cgroup相关的支持，否则启动docker会出错。

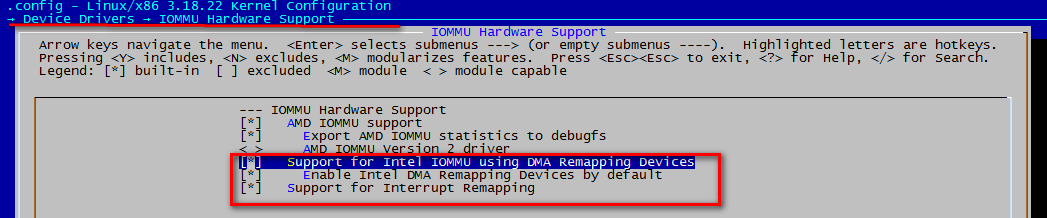


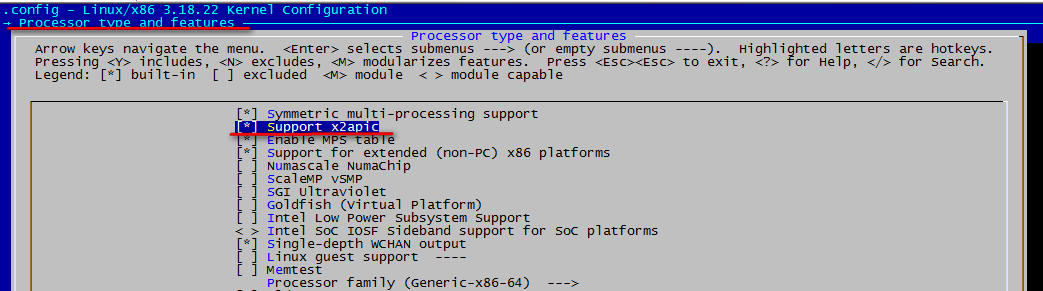
#加入iptables NAT support支持



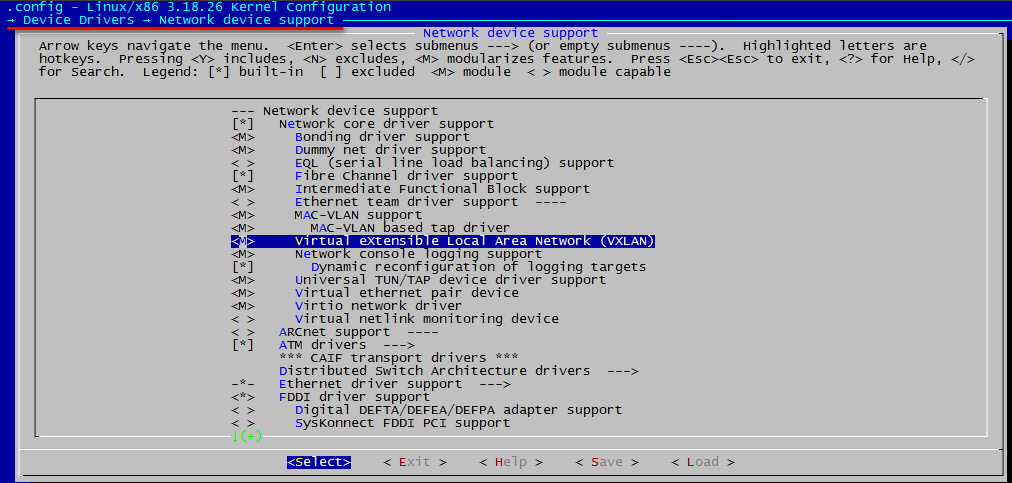
#解决升级新内核启动后主机的cpu变成单核问题，需要加入以下支持。

<https://forums.gentoo.org/viewtopic-t-1017154-view-next.html?sid=e77923129a11adf4e9efda10302fc2b7>





加入VxLAN支持，默认开启



4）开始编译

cd linux-3.10.80

make -j8 bzImage #生成内核文件

make -j8 modules #编译模块

make -j8 modules\_install #编译安装模块

make install

#可以一条命里全部执行：

make -j8 bzImage && make -j8 modules && make -j8 modules\_install && make install

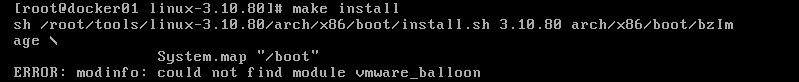
# -j [N], --jobs[=N] Allow N jobs at once; infinite jobs with no arg. 多任务处理，加快编译速度。一般根据cpu的逻辑核多少设置。

#如果要重新编译，需要清理上次编译的现场

make mrproper

make clean

虚拟机会可能会提示下面错误，这个错误可以忽略，不影响docker安装及使用。



5）调整内核启动顺序

[root@node2 ~]# vim /boot/grub/grub.conf

default=0

timeout=5

splashimage=(hd0,0)/grub/splash.xpm.gz

hiddenmenu

title CentOS (3.10.80)

root (hd0,0)

kernel /vmlinuz-3.10.80 ro root=UUID=e709ba54-2ceb-4ab1-a2f3-87eda97277c1 rd\_NO\_LUKS rd\_NO\_LVM LANG=e

n\_US.UTF-8 rd\_NO\_MD SYSFONT=latarcyrheb-sun16 crashkernel=auto KEYBOARDTYPE=pc KEYTABLE=us rd\_NO\_DM rhgb qui

et

initrd /initramfs-3.10.80.img

title CentOS (2.6.32-431.el6.x86\_64)

root (hd0,0)

kernel /vmlinuz-2.6.32-431.el6.x86\_64 ro root=UUID=e709ba54-2ceb-4ab1-a2f3-87eda97277c1 rd\_NO\_LUKS rd

\_NO\_LVM LANG=en\_US.UTF-8 rd\_NO\_MD SYSFONT=latarcyrheb-sun16 crashkernel=auto KEYBOARDTYPE=pc KEYTABLE=us rd\_

NO\_DM rhgb quiet

initrd /initramfs-2.6.32-431.el6.x86\_64.img

重启检查结果，升级完成！

[root@node2 ~]# reboot

[root@node2 ~]# uname -a

Linux node2.mfs.com 3.10.80 #1 SMP Tue Jun 9 16:04:37 CST 2015 x86\_64 x86\_64 x86\_64 GNU/Linux

[root@node2 ~]# cat /etc/redhat-release

Red Hat Enterprise Linux Server release 6.3 (Santiago)

2.1.2 软件准备

1）检查Device Mapper驱动，最好使用yum更新一次，否则后面启动docker时会可能会出错。

# yum update device-mapper

这里将使用Device Mapper作为存储驱动，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）的概念，可以在一种文件系统中存储多台虚拟设备（docker镜像中的层）。可以使用下面命令来确认是否安装。

[root@docker01 ~]# ls -l /sys/class/misc/device-mapper

lrwxrwxrwx. 1 root root 0 Jun 7 09:34 /sys/class/misc/device-mapper -> ../../devices/virtual/misc/device-mapper

也可以检查下面配置文件

[root@docker01 ~]# grep device-mapper /proc/devices

253 device-mapper

如果未出现，可以使用下面命令加载

[root@docker01 ~]# modprobe dm\_mod

2.2 安装docker

2.2.1 centos6.5下安装docker

1）配置软件仓库

rpm -Uvh <http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm>

[root@docker01 ~]# ll /etc/yum.repos.d/epel\*

-rw-r--r--. 1 root root 1056 Nov 5 2012 /etc/yum.repos.d/epel-testing.repo

-rw-r--r--. 1 root root 957 Nov 5 2012 /etc/yum.repos.d/epel.repo

2）安装docker

下面是在centos 6发行版中安装docker，注意：安装包是docker-io，而不是docker。

$ yum install docker-io -y

$ rpm -qa docker-io

docker-io-1.5.0-1.el6.x86\_64

3）启动docker

[root@docker01 ~]# /etc/init.d/docker start

[root@docker01 ~]# ps -ef|grep docker

root 1219 1 0 03:07 ? 00:00:00 /usr/bin/docker -d

[root@docker01 ~]# docker info

[root@docker01 ~]# docker version

2.2.2 centos7.0下安装docker

[root@docker02 ~]# uname -a

Linux docker02 3.10.0-123.el7.x86\_64 #1 SMP Mon Jun 30 12:09:22 UTC 2014 x86\_64 x86\_64 x86\_64 GNU/Linux

[root@docker02 ~]# cat /etc/redhat-release

CentOS Linux release 7.0.1406 (Core)

1）配置软件仓库，这里配置了阿里云的镜像来安装docker

wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.repo

yum makecache

2）安装docker

yum install docker-io -y

3）启动docker

[root@docker02 ~]# systemctl start docker

[root@docker02 ~]# ps -ef|grep docker

root 7064 1 0 14:01 ? 00:00:00 /usr/bin/docker -d --selinux-enabled

4）测试docker是否正确安装

[root@docker02 ~]# docker info

Containers: 0

Images: 0

……

Execution Driver: native-0.2

Kernel Version: 3.10.0-123.el7.x86\_64

Operating System: CentOS Linux 7 (Core)

CPUs: 1

Total Memory: 987.2 MiB

Name: docker02

ID: TOPC:7VR6:J5GW:7OYH:2VHE:3LZF:SLHW:YQ2G:JUUN:T64S:N334:ANOF

2.2.3 docker脚本和二进制安装

1）使用脚本的方式安装docker

get.docker.io网站提供了docker的安装脚本，这个脚本会自动安装docker所需的依赖，并且检查当期系统的内核版本是否满足要求，以及是否支持所需的存储驱动，最后会安装docker并启动docker守护进程

curl <https://get.docker.io/> |sudo sh （这种方式目前支持Ubuntu\Fedroa\Debian\Gentoo）

2）使用二进制的方式安装docker

二进制方式安装满足离线安装要求，适用服务器无法直接连接外部网络的场景。

# To install, run the following command as root:

curl -sSL -O https://get.docker.com/builds/Linux/x86\_64/docker-1.6.2 && chmod +x docker-1.6.2 && sudo mv docker-1.6.2 /usr/local/bin/docker

# Then start docker in daemon mode:

sudo /usr/local/bin/docker -d

2.2.4 docker升级安装

1）当前版本

[root@host103 ~]# docker version

Client:

Version: 1.8.3

API version: 1.20

Go version: go1.4.2

Git commit: f4bf5c7

Built: Mon Oct 12 18:01:15 UTC 2015

OS/Arch: linux/amd64

Server:

Version: 1.8.3

API version: 1.20

Go version: go1.4.2

Git commit: f4bf5c7

Built: Mon Oct 12 18:01:15 UTC 2015

OS/Arch: linux/amd64

2）下载最新版本docker

wget https://get.docker.com/builds/Linux/x86\_64/docker-1.9.1.tgz

3）新版本替换旧版本

[ -f /usr/local/bin/docker ] && cp /usr/local/bin/docker{,-`docker --version|awk -F'[ ,]+' '{print $3}'`}

/etc/init.d/docker stop

rm -f /usr/local/bin/docker

tar -zxf docker-1.9.1.tgz

cp -rf usr/local/bin/docker /usr/local/bin/

[ -f /usr/bin/docker ] && rm -f /usr/bin/docker && ln -s /usr/local/bin/docker /usr/bin/docker

/etc/init.d/docker start

4）centos7上升级docker

wget https://get.docker.com/builds/Linux/x86\_64/docker-1.10.3.tar.gz

[ -f /usr/bin/docker ] && cp /usr/bin/docker{,-`docker --version|awk -F'[ ,]+' '{print $3}'`}

systemctl stop docker

rm -f /usr/bin/docker

tar zxf docker-1.10.3.tgz

cp -rf usr/local/bin/docker /usr/bin/docker

#修改/usr/lib/systemd/system/docker.service

[root@localhost tools]# vim /usr/lib/systemd/system/docker.service

[Unit]

Description=Docker Application Container Engine

Documentation=http://docs.docker.com

After=network.target

Wants=docker-storage-setup.service

[Service]

Type=notify

EnvironmentFile=-/etc/sysconfig/docker

EnvironmentFile=-/etc/sysconfig/docker-storage

EnvironmentFile=-/etc/sysconfig/docker-network

Environment=GOTRACEBACK=crash

ExecStart=/usr/bin/docker daemon $OPTIONS \

$DOCKER\_STORAGE\_OPTIONS \

$DOCKER\_NETWORK\_OPTIONS \

$ADD\_REGISTRY \

$BLOCK\_REGISTRY \

$INSECURE\_REGISTRY

LimitNOFILE=1048576

LimitNPROC=1048576

LimitCORE=infinity

MountFlags=slave

[Install]

WantedBy=multi-user.target

#启动docker

systemctl start docker

2.2.5 docker守护进程配置

默认情况我们可以不加任何参数，在centos7下直接使用systemctl start docker运行docker，也可以使用-H参数调整守护进程绑定监听接口的方式。-H指定不同的网络接口和端口配置。

[root@docker02 ~]# /usr/bin/docker -d -H tcp://192.168.3.15:9000

错误记录1

[root@docker02 ~]# /usr/bin/docker -d -H tcp://192.168.3.15:9000

No --storage-opt dm.thinpooldev specified, using loopback; this configuration is strongly discouraged for production use

第3章. docker镜像和仓库

3.1 什么是docker镜像

Docker镜像是由文件系统叠加而成，最低端是一个引导文件系统，即bootfs。Docker看起来像一个典型的Linux虚拟化栈。实际上，docker镜像的第二层是root文件系统rootfs，它位于引导文件系统之上。Rootfs可以是一种或多种操作系统（如debian、ubuntu文件系统）。

在传统的Linux引导过程中，root文件系统会最先以只读方式加载，当引导结束并完成了完整性检查之后，它才会被切换为读写模式。但是docker里面，root文件系统永远都是只读状态，并且docker利用联合加载(union mount)技术又会在root文件系统层上加载更多的只读文件系统。联合加载指的是一次同时加载多个文件系统，但是在外面看起来只能看到一个文件系统。联合加载将会各层文件系统叠加到一起，这样最终的文件系统会包含所有底层的文件和目录。

Docker将这样的文件系统称为镜像。一个镜像可以放到另一个镜像的顶部，位于下面的镜像称为父镜像（parent image）,可以依次类推，知道镜像栈的最底部，最底部的镜像称为基础镜像（base image）。最后，当一个镜像启动容器时，docker会在该镜像的最顶层加载一个读写文件系统，docker中运行的程序就是在这个读写层中执行的。

第一次启动容器，初始读写层是空的。只有当文件系统发生变化时，这些变化都会应用到这一层上。比如想修改一个文件，这个文件首先会从该读写层下面的只读层复制到该读写层，该文件的只读版本依然存在，但是已经被读写层中的该文件副本所隐藏了。这种机制被称为写时复制（copy on write），是docker最具有特点技术之一。

每个只读镜像层都是只读的，并且以后永远不会变化。当创建一个新容器时，docker会构建出一个镜像栈，并在栈的最顶端添加一个读写层。这个读写层在再加上其下面的镜像层以及一些配置数据，就构成了一个容器。容器是可以修改的，可以启动和停止。容器的这种特点加上镜像分层框架（image-layering framework）,使我们可以快速构建镜像并运行包含我们自己的应用程序和服务的容器。

3.2 docker镜像管理及构建

3.2.1 列出镜像

镜像默认保存在/var/lib/docker目录下，一般是在存储驱动器下，devicemapper或aufs目录下面。

镜像从仓库下载下来，镜像保存在仓库中，而仓库存在于registry中。默认的registry是由docker公司运营的公共registry服务，即docker hub。官方站点：<https://registry.hub.docker.com/>

列出当前下载的镜像

[root@localhost ~]# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

docker.io/centos 7 fd44297e2ddb 6 weeks ago 215.7 MB

docker.io/centos latest fd44297e2ddb 6 weeks ago 215.7 MB

下面这个例子表示从镜像centos:7启动一个容器，容器命名为container\_002，而这个镜像的操作系统则是centos7。在构建docker容器时，指定详细的镜像名是比较好的做法，后面查询时候就能清楚的知道这个容器下的镜像是什么版本。

[root@localhost ~]# docker run --name container\_002 -t -i centos:7 /bin/bash

[root@d476745ff6d8 /]#

Docker hub网站中有两中类型的仓库：用户仓库（user repository）和顶层仓库（top-level repository）。用户仓库的镜像是由docker用户创建的，而顶层仓库是由docker hub内部的人员管理的。

用户仓库的命名由用户名和仓库名两部分组成，如jamtur01/puppet。

* 用户名：jamtur01
* 仓库名：puppet

3.2.3 拉取镜像

使用docker pull命令将镜像拉取到本地，可以节省从一个新镜像启动一个容器所需的时间。

[root@docker01 ~]# docker pull centos:6 #🡸拉取centos6镜像到本地，镜像是从下面的站点拉取到<https://registry.hub.docker.com/>

Pulling repository centos

b9aeeaeb5e17: Download complete

f1b10cd84249: Download complete

Status: Downloaded newer image for centos:6

[root@docker01 ~]# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

centos centos6 b9aeeaeb5e17 6 weeks ago 202.6 MB

centos 6 b9aeeaeb5e17 6 weeks ago 202.6 MB

关于拉取镜像慢的问题，目前由于国内访问国外的站点速度比较缓慢，甚至有时候会被墙掉，所以配置国内镜像站点是必不可少的。推荐国内比较好的镜像站点：<https://www.daocloud.io/>

#1 配置Docker加速器

sudo sed -i 's|other\_args=|other\_args=--registry-mirror=http://7bb93a3c.m.daocloud.io |g' /etc/sysconfig/docker

sudo sed -i "s|OPTIONS='|OPTIONS='--registry-mirror=http://7bb93a3c.m.daocloud.io |g" /etc/sysconfig/docker

/etc/init.d/docker restart

3.2.4 查找镜像

[root@docker02 ~]# docker search puppet

INDEX NAME DESCRIPTION STARS OFFICIAL AUTOMATED

docker.io docker.io/devopsil/puppet Dockerfile for a container with puppet ins... 9 [OK]

docker.io docker.io/macadmins/munki-puppet Basic munki nginx image with Puppet used f... 6 [OK]

docker.io docker.io/devopsil/puppet-yum 5 [OK]

docker.io docker.io/layerworx/puppetmaster Standard dockerized Puppetmaster server bu... 5 [OK]

docker.io docker.io/macadmins/puppetmaster Simple puppetmaster based on CentOS 6 5 [OK]

docker.io docker.io/layerworx/puppet A Puppet enabled container running the 3.7... 4 [OK]

docker.io docker.io/rfkrocktk/puppet 2 [OK]

docker.io docker.io/devopsil/puppetmaster 1 [OK]

docker.io docker.io/lukasz/docker-puppet-openssl-nginx This image is derived from https://index.d... 1 [OK]

#NAME #🡸表示仓库名

#DESCRIPTION #🡸镜像描述

#STARS #🡸用户评价

#OFFICAL #🡸是否官方

#AUTOMATED #🡸表示这个镜像是由docker hub自动构建流程创建的

从上面的镜像中拉取一个镜像，会将puppetmaster镜像下载到本地，这个镜像预装了puppetmaster。可以通过拉取的镜像创建容器，这个容器包含了puppetmaster服务。

[root@docker02 ~]# docker pull docker.io/macadmins/puppetmaster

3.2.5 删除镜像

如果一个镜像不需要再使用它，可以使用docker rmi命令来删除一个镜像。

[root@docker01 apache\_web]# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

apache\_web/web\_0.0.1 latest 77fbdc966138 2 hours ago 271.3 MB

centos 6 b9aeeaeb5e17 6 weeks ago 202.6 MB

centos centos6 b9aeeaeb5e17 6 weeks ago 202.6 MB

[root@docker01 apache\_web]# docker rmi apache\_web/web\_0.0.1

批量删除none镜像方法

docker images|grep none|awk '{print $3}'|xargs docker rmi

3.2.6 构建镜像

构建镜像有两种方式，一种是基于commit命令创建镜像，一种是基于dockerfile构建镜像。

3.2.6.1 基于Dockerfile构建镜像

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

1）下面使用dockerfile构建一个简单的apache服务器 httpd服务。

mkdir /images/apache\_web/ -p #🡸创建构建环境目录

cd /images/apache\_web/ && touch Dockerfile #🡸创建构建文件

2）编辑构建文件，在dockerfile文件里面加入以下内容:

[root@docker02 apache\_web]# cat Dockerfile

#version 0.0.1

FROM centos:6

MAINTAINER yang "yang.hong@abc.com"

RUN yum install -y httpd

RUN echo 'Hi,I am in your container' > /var/www/html/index.html

EXPOSE 80

**# FROM：** 表示指定一个已存在的镜像，后续指令都将基于该镜像进行；

**# MAINTAINER：** 指定该docker镜像作者及联系方式；

**# RUN：**表示会在当前镜像中运行指定的命令，默认情况下RUN指令会在shell里使用命令/bin/sh –c来执行，如果是在一个不支持的shell的平台上运行或不希望在shell执行，可以使用exec格式的RUN指令,格式如下：

RUN [ “yum”,”install”,”-y”,”nginx” ]

**# EXPOSE 80：**用来向外部公开的端口，可以使用多个EXPOSE来指定多个外部端口。

提示：Dockerfile由一系列指令和参数组成。每条指令，FROM必须为大写字母，且后面要跟一个参数。Dockerfile中的指令会按顺序从上到下执行，所以应该根据需要合理安排指令的顺序。“#”号开头表示注释。

3）开始构建新镜像

[root@docker01 apache\_web]# cd /images/httpd\_web/

[root@docker01 apache\_web]# docker build -t="httpd/apache\_web" .

Sending build context to Docker daemon 2.048 kB

Sending build context to Docker daemon

Step 0 : FROM centos:6

---> b9aeeaeb5e17

Step 1 : MAINTAINER yang "yang.hong@abc.com"

---> Using cache

---> a6a86e3cc7e1

Step 2 : RUN yum install -y httpd

---> Running in 9c47aeaf3f34

Loaded plugins: fastestmirror

Setting up Install Process

Resolving Dependencies

……

Step 3 : RUN echo 'Hi,I am in your container' > /var/www/html/index.html

---> Running in 4675ae9a326c

---> e6bd50932684

Removing intermediate container 4675ae9a326c

Step 4 : EXPOSE 80

---> Running in 773025c3348e

---> 71a65ac2ff29

Removing intermediate container 773025c3348e

Successfully built 71a65ac2ff29 #🡸构建成功，新镜像ID为71a65ac2ff29

4）查看构建的新镜像

[root@docker01 apache\_web]# docker images httpd/apache\_web

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

httpd/apache\_web latest 71a65ac2ff29 About a minute ago 271.2 MB

5）构建过程错误记录

错误记录1

[root@docker01 httpd\_web]# docker build -t="web/httpd\_web" .

FATA[0000] Cannot locate Dockerfile: Dockerfile

#解决方法：mv dockerfile Dockerfile

错误记录2

[root@docker01 httpd\_web]# docker build -t="web/httpd \_web" .

FATA[0000] Invalid namespace name (web). Cannot be fewer than 4 or more than 30 characters.

#解决方法：这里设置的仓库名为web，太短了，需要至少4个，最多为30个字符的名称。加长仓库名称

错误记录3：

[root@docker01 nginx\_web]# docker build -t="system/nginx\_web" .

Sending build context to Docker daemon 2.048 kB

Sending build context to Docker daemon

Step 0 : FROM centos:6

---> b9aeeaeb5e17

Step 1 : MAINTAINER yang "yang.hong@abc.com"

---> Using cache

---> a6a86e3cc7e1

Step 2 : RUN yum install -y nginx

---> Running in 7a76468eadbe

Loaded plugins: fastestmirror

Setting up Install Process

No package nginx available.

Error: Nothing to do

INFO[0020] The command [/bin/sh -c yum install -y nginx] returned a non-zero code: 1

思考? 这个问题是由于基础镜像里面yum源里没有nginx程序包，那么如何操作才能安装nginx服务呢？

方法1：（这种方法不行，再次构建容器时镜像就恢复为基础镜像，docker是基于写时复制技术）

这里创建的容器yum里面没有nginx程序包，可以进入最后一步成功创建的Step1的容器（a6a86e3cc7e1），对该容器进行yum配置，然后再次执行构建。

3.2.6.2 基于新镜像启动容器

上面已经成功构建了新的镜像httpd/apache\_web，现在将在这个镜像的基础上启动一个新的容器。容器名设定为apache\_web01。

1）基于httpd/apache\_web镜像创建新容器

a. 🡸以守护式进程方式创建容器。Docker可以在宿主机上随机选择一个位于49000-49900的一个比较大的端口号来映射到容器中的80端口上,也可以指定宿主机某一个端口映射到容器的80端口上，下面是随机选择了宿主主机的端口49177映射到容器里面的80端口。

#1 -p 8080:80 表示指定宿主主机的8080端口映射到容器内部80端口

#2 -p 192.168.100.20:8080:80 表示指定宿主主机的IP地址及8080端口映射容器内部，默认是宿主主机的eth0

[root@docker01 ~]# docker run -d -i -t -p 80 --name apache\_web01 httpd/apache\_web /bin/bash

5382df0c806f1e792767160b4d2208c2f08019b827d0f1bc8c7235bd6d7f6852

[root@docker01 ~]# docker ps -l

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

5382df0c806f httpd/apache\_web:latest "/bin/bash" 8 seconds ago Up 7 seconds 0.0.0.0:49179->80/tcp apache\_web01

[root@docker01 ~]# docker port apache\_web01 80

0.0.0.0:49179

b.🡸启动容器内的apache服务进程

[root@docker01 ~]# docker exec -it apache\_web01 /etc/init.d/httpd start

Starting httpd: [ OK ]

[root@docker01 ~]# docker exec -it apache\_web01 ps -ef|grep httpd

root 104 1 0 22:34 ? 00:00:00 /usr/sbin/httpd

apache 106 104 0 22:34 ? 00:00:00 /usr/sbin/httpd

apache 107 104 0 22:34 ? 00:00:00 /usr/sbin/httpd

……

c．🡸主页在刚才构建的时间已经加入，这里也可以修改容器的首页，使用exec方式进入修改完后，再退出容器时不会停止容器

[root@docker01 ~]# docker exec -it apache\_web01 /bin/bash

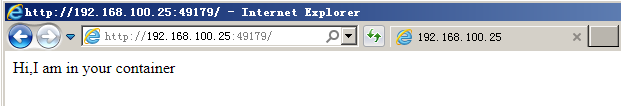
d．🡸访问测试，这里测试宿主主机使用curl命令来测试，以及在宿主主机之外的机器访问，。

#本地宿主主机访问测试

[root@docker01 ~]# curl http://127.0.0.1:49179

Hi,I am in your container

#外部主机访问测试，外部主机访问是通过宿主主机IP地址+端口方式来访问。



3.2.6.3 基于commit制作镜像

docker commit containerID imagesname

#🡸containerID是要打包的容器ID号

#🡸imagesname打包后的指定镜像名

3.3 docker镜像存储位置

在/var/lib/docker/目录存储这镜像相关的内容，如下：

[root@host63 docker]# cat /var/lib/docker/repositories-devicemapper |python -mjson.tool

{

"Repositories": {

"centos": {

"6": "b9aeeaeb5e17b5414e5caa9a6b2f99e9ccef50561bdfe137cd05956961f1cec6",

"centos6": "b9aeeaeb5e17b5414e5caa9a6b2f99e9ccef50561bdfe137cd05956961f1cec6"

},

"repo/sandcard": {

"v1": "cfbba6629b516bbcbaef9e3000dd2b85ba2c8d03f88ff089a4e537ece6b3d6bf"

},

"repo/scsp": {

"v1": "3b00bff0f69ded7eb9d3bebabe6be4264656c7877be0b1ef287f2dd465b1b2fa"

}

}

}

可以观察到上面有相同部分的镜像ID。

# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

repo/sandcard v1 cfbba6629b51 About an hour ago 609.1 MB

repo/scsp v1 3b00bff0f69d 30 hours ago 564.3 MB

centos 6 b9aeeaeb5e17 10 weeks ago 202.6 MB

centos centos6 b9aeeaeb5e17 10 weeks ago 202.6 MB

在/var/lib/docker目录下存储如下内容。

# ll /var/lib/docker/graph/|grep 3b00bff0f69d

drwx------ 2 root root 4096 Jul 1 09:08 3b00bff0f69ded7eb9d3bebabe6be4264656c7877be0b1ef287f2dd465b1b2fa

# ll /var/lib/docker/graph/|grep b9aeeaeb5e17

drwx------ 2 root root 4096 Jun 18 11:47 b9aeeaeb5e17b5414e5caa9a6b2f99e9ccef50561bdfe137cd05956961f1cec6

目录内部内容如下：

json #🡸保存着关于这个镜像的元数据

layersize #🡸一个整数，表示layer的大小。

layer/ #🡸子文件夹，保存着rootfs该容器的镜像

$ cat json |python -mjson.tool

{

"Size": 0,

"architecture": "amd64",

"author": "\"jame.yang\"",

"config": {

"AttachStderr": false,

"AttachStdin": false,

"AttachStdout": false,

"Cmd": null,

"CpuShares": 0,

"Cpuset": "",

"Domainname": "",

"Entrypoint": [

"/bin/sh",

"-c",

"sh /home/scspuat/scsp.sh && sudo /etc/init.d/sshd start && tail -f /var/log/lastlog"

],

3.4 docker私有仓库管理

Docker官方提供了公共的registry，但有时构建和存储包含不想被公开的信息或数据的镜像。这个时候我们一般有两种选择。

* 利用docker hub上的私有仓库。
* 制作私人的registry。

3.4.1 docker registry 1.X系列搭建

1）新建仓库

docker pull registry

docker run -d -p 5000:5000 -e STORAGE\_PATH=/registry -v /u01/registry:/registry --name local\_registry registry

#🡸默认仓库镜像存储位置为/tmp/registry，为了防止容器被删除导致镜像丢失，可以将此目录映射到宿主主机指定目录。-e STORAGE\_PATH=/tmp/registry指定存储路径。

2）上传镜像到仓库

这里将本地已构建好的镜像apache/web:v0.0.1上传到新制作的仓库里面。首先查看要上传的镜像ID。

[root@docker02 ~]# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

apache-2.2.27 latest 29093a455f50 24 minutes ago 713.3 MB

tomcat-7.0.54 latest 9b449a21dbce 46 minutes ago 522.4 MB

registry latest ae27e8792444 5 days ago 423.2 MB

centos 6 b9aeeaeb5e17 5 months ago 202.6 MB

centos centos6 b9aeeaeb5e17 5 months ago 202.6 MB

atcol/docker-registry-ui latest d838355ad903 6 months ago 890.5 MB

使用容器ID为镜像打标签，设置镜像名称及版本号

[root@docker02 ~]# docker tag 9b449a21dbce 192.168.3.17:5000/tomcat-7.0.54

[root@docker02 ~]# docker tag 29093a455f50 192.168.3.17:5000/apache-2.2.27

将镜像推送到local\_registry里面

[root@docker02 ~]# docker push 192.168.3.17:5000/apache-2.2.27

[root@docker02 ~]# docker push 192.168.3.17:5000/tomcat-7.0.54

新的镜像推送成功后，其他主机可以使用docker run来直接基于该镜像创建启动使用。

docker run -d -p 8000:80 --name apache01 192.168.3.17:5000/tomcat-7.0.54

docker run -d -p 8001:8080 --name tomcat01 192.168.3.17:5000/apache-2.2.27

3）查看和搜索远程仓库镜像(适用于与registry v1.x)

[root@docker02 ~]# curl -s http://192.168.3.17:5000/v1/search|python -mjson.tool

{

"num\_results": 2,

"query": "",

"results": [

{

"description": "",

"name": "library/tomcat-7.0.54"

},

{

"description": "",

"name": "library/apache-2.2.27"

}

]

}

[root@docker02 ~]# docker search 192.168.3.17:5000/tomcat

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

library/tomcat-7.0.54 0

4）拉取远程仓库镜像

docker run -d -p 9000:8080 --name tomcat1 192.168.3.17:5000/tomcat-7.0.54

5）删除镜像(适用于与registry v1.x)

[root@docker02 ~]# curl -X DELETE http://192.168.3.17:5000/v1/repositories/tomcat-7.0.54/

[root@docker02 ~]# docker search 192.168.3.17:5000/tomcat-7.0

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

[root@docker02 ~]# curl -s http://192.168.3.17:5000/v1/search|python -mjson.tool

{

"num\_results": 0,

"query": "",

"results": []

}

6）将registry和registry-ui整合

参考文档：

Docker介绍以及Registry的安装：<http://dockone.io/article/108>

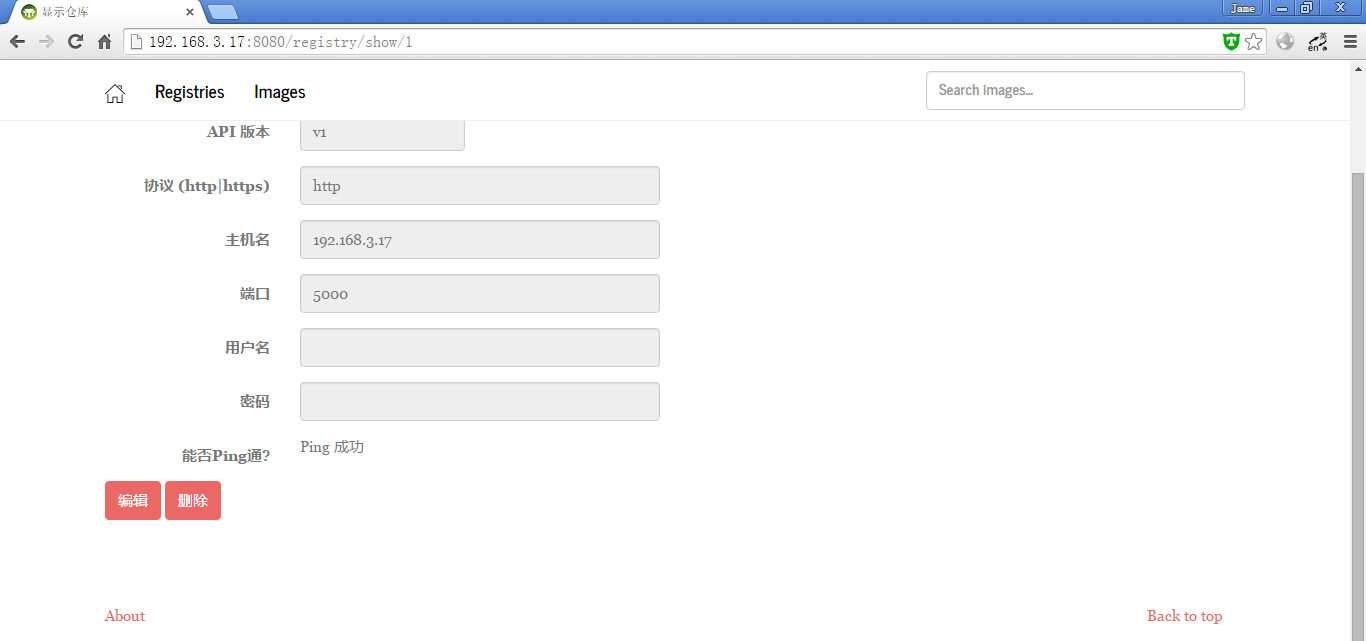
docker-registry-ui github站点：<https://github.com/atc-/docker-registry-ui>

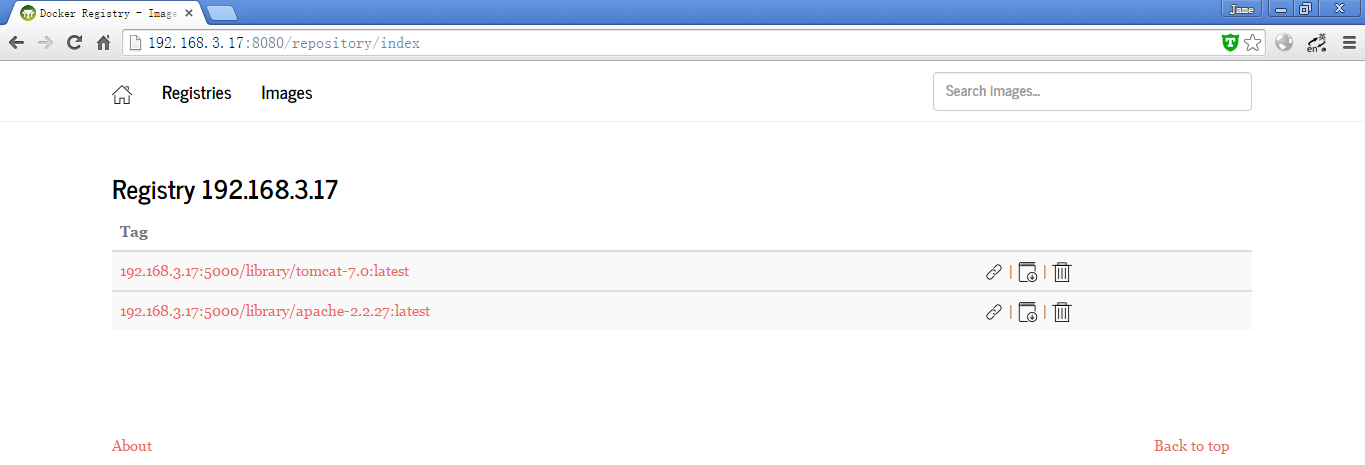
Docker私有仓库用户认证代理配置：<http://robbiefeng.iteye.com/blog/2210454>

# docker pull atcol/docker-registry-ui

# docker run -d -p 8080:8080 -v /u01/registry-ui/h2:/var/lib/h2 --name registry-ui atcol/docker-registry-ui

通过浏览器来访问镜像仓库站点：<http://192.168.3.17:8080/registry/index>





3.4.2 docker registry 2.X系列搭建(普通模式)

<http://www.dockerdocs.cn/Registry/README.html>

1）创建仓库

cd /dockerdata/registrydata

docker run -d -p 5000:5000 --restart=always --name registry2 \

-v `pwd`/data2:/var/lib/registry \

registry:2

2）搜索远程仓库镜像

# curl http://172.28.250.122:5000/v2/\_catalog

{"repositories":["jdk"]}

# curl http://172.28.250.122:5000/v2/jdk/tags/list

{"name":"jdk","tags":["1.7.79"]}

上传镜像到仓库参考上面的步骤。

3）删除镜像

# curl -v -X DELETE http://172.28.250.122:5000/v2/openssh-server/manifests/v1

3.4.3 docker registry 2.X系列搭建(基于用户验证)

#创建Docker仓库数据和配置目录

mkdir /dockerdata/registrydata/{data,auth,cert} -p

#生成密码

docker run --entrypoint htpasswd registry:2 -Bbn admin 123456 > auth/htpasswd

###########暂时忽略证书这一步

#生成签名证书

openssl req -x509 -nodes -newkey rsa:2048 -keyout /dockerdata/registrydata/cert/docker-registry.key -out /dockerdata/registrydata/cert/docker-registry.crt

###########

#创建仓库

docker run -d -p 5000:5000 --restart=always --name registry \

-v /dockerdata/registrydata/auth:/auth \

-e "REGISTRY\_AUTH=htpasswd" \

-e "REGISTRY\_AUTH\_HTPASSWD\_REALM=Registry Realm" \

-e REGISTRY\_AUTH\_HTPASSWD\_PATH=/auth/htpasswd \

-v /dockerdata/registrydata/data:/var/lib/registry \

registry:2

#客户端登录

#注意在/etc/sysconfig/docker中加入--insecure-registry参数

/usr/bin/docker -d --insecure-registry 172.28.250.122:5000 --registry-mirror=http://f07735fb.m.daocloud.io

[root@localhost ~]# docker login 172.28.250.122:5000

Username: admin

Password:

Email:

WARNING: login credentials saved in /root/.docker/config.json

Login Succeeded

#推送镜像

docker run -id --name test:v1 centos:6 /bin/bash/

docker commit test\_v1 test\_v1\_0.1

docker tag 96272e94c5ce 172.28.250.122:5000/test\_v1\_0.1

docker push 172.28.250.122:5000/test\_v1\_0.1

#访问仓库镜像

<http://172.28.250.122:5000/v2/_catalog>

3.5 使用harbor来管理仓库镜像

3.5.2 harbor简介

VMware公司最近开源了企业级Registry项目Harbor，其的目标是帮助用户迅速搭建一个企业级的Docker registry 服务。它以Docker公司开源的registry

为基础，提供了管理UI, 基于角色的访问控制(Role Based Access Control)，AD/LDAP集成、以及审计日志(Audit logging) 等企业用户需求的功能，同时还原生支持中文，对广大中国用户是一个好消息。本文将介绍Harbor 项目的主要组件，并阐述Harbor的工作原理。

Harbor的每个组件都是以Docker容器的形式构建的，使用Docker Compose来对它进行部署。用于部署Harbor的Docker Compose模板位于 /Deployer/docker-compose.yml，由5个容器组成：

* Proxy: 由Nginx 服务器构成的反向代理。
* Registry:由Docker官方的开源registry 镜像构成的容器实例。
* UI: 即架构中的core services, 构成此容器的代码是Harbor项目的主体。
* Mysql: 由官方MySql镜像构成的数据库容器。
* Log: 运行着rsyslogd的容器，通过log-driver的形式收集其他容器的日志。

这几个容器通过Docker link的形式连接在一起，在容器之间通过容器名字互相访问。对终端用户而言，只需要暴露proxy （即Nginx）的服务端口。

参考网址：<https://github.com/vmware/harbor>

3.5.2 harbor部署

（1）harbor安装

#创建目录，clone harbor到指定目录

mkdir /data/{database,registry,harbor} -p

cd /data/harbor/

git clone <https://github.com/vmware/harbor>

cd harbor-master/Deploy/

#修改首先修改harbor.cfg文件

#The IP address or hostname to access admin UI and registry service.

#DO NOT use localhost or 127.0.0.1, because Harbor needs to be accessed by external clients.

hostname = 172.28.250.122 #对外公开的主机名或域名

##The password of Harbor admin, change this before any production use.

harbor\_admin\_password = Harbor12345 #harbor登录的密码

##By default the auth mode is db\_auth, i.e. the credentials are stored in a local database.

#Set it to ldap\_auth if you want to verify a user's credentials against an LDAP server.

auth\_mode = db\_auth #harbor密码存储方式，默认是存储mysql数据库中，也可以通过LDAP来认证

#执行prepare

[root@host122 Deploy]# ./prepare

#运行./prepare脚本更新配置，完成配置后，就可以使用docker-compose快速部署harbor：

[root@host122 Deploy]# docker-compose up -d

……

Creating deploy\_log\_1

Creating deploy\_registry\_1

Creating deploy\_ui\_1

Creating deploy\_mysql\_1

Creating deploy\_proxy\_1

[root@host122 Deploy]# docker-compose ps

Name Command State Ports

-----------------------------------------------------------------------------------------------------

deploy\_log\_1 /bin/sh -c cron && rsyslogd -n Up 0.0.0.0:1514->514/tcp

deploy\_mysql\_1 docker-entrypoint.sh mysqld Up 3306/tcp

deploy\_proxy\_1 nginx -g daemon off; Up 0.0.0.0:443->443/tcp, 0.0.0.0:80->80/tcp

deploy\_registry\_1 /bin/registry serve /etc/r ... Up 5000/tcp, 0.0.0.0:5001->5001/tcp

deploy\_ui\_1 /go/bin/harbor\_ui Up 80/tcp

问题记录：

a. 在构建ui的时候，docker pull golang:1.6.2会可能出现超时，可以先手动pull下来，或者通过其他方式直接下载导入。

b. 在apt-get的时候也会超时，访问国外仓库一般都会出现这个问题。可以通过配置网易163的镜像源解决这个问题。http://mirrors.163.com/.help/debian.html。修改容器的源文件，需要先修改Dcokerfile.ui这个文件，添加如下代码：

RUN rm -f /etc/apt/sources.list

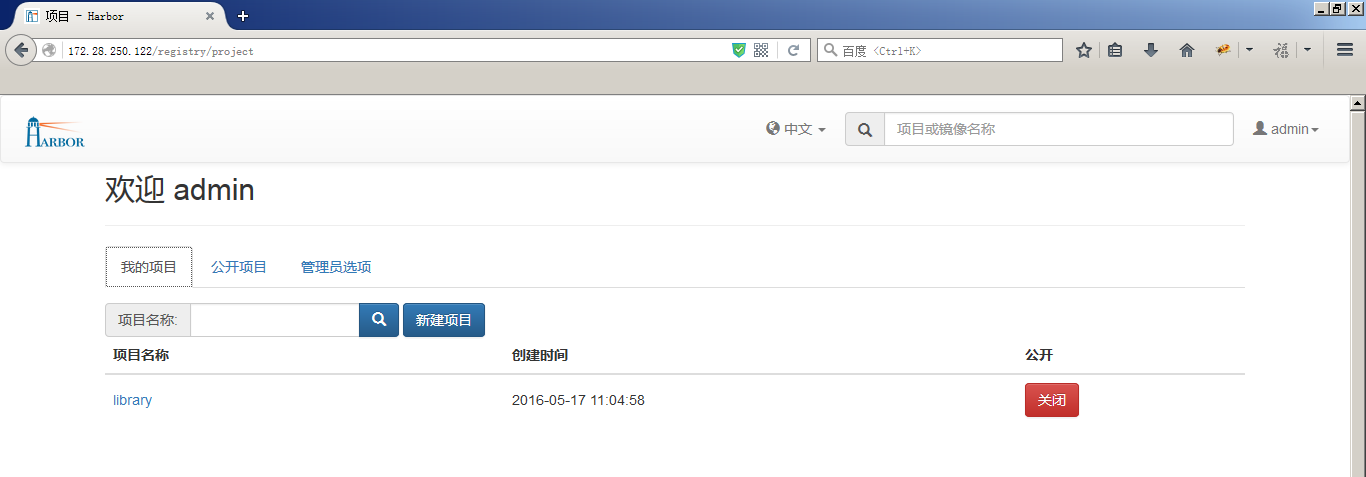
COPY ./sources.list /etc/apt/sources.list

（2）harbor Web UI访问

安装完成后，访问Web UI，地址：<http://172.28.250.122>，端口为80, 默认账户admin/Harbor12345。如图：







3.5.3 上传镜像到harbor

#1 配置DOCKER\_OPTS，配置完成重启一次docker

[root@host122 Deploy]# ps -ef|grep docker|grep 172.28.250.122

root 20514 1 12 13:20 pts/1 00:00:00 /usr/bin/docker daemon --insecure-registry 172.28.250.122 --registry-mirror=http://f07735fb.m.daocloud.io -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

#2 使用docker client登录到harbor，docker-client版本必须>=1.6.0。

[root@host122 Deploy]# docker login -u admin -p Harbor12345 -e test@test.com 172.28.250.122

WARNING: login credentials saved in /root/.docker/config.json

Login Succeeded

#harbor标签格式为：Harbor地址/项目名/镜像名称:镜像标签

#推送镜像到仓库

[root@host122 Deploy]# docker tag nginx:1.9 172.28.250.122/webserver/nginx:1.9.0

[root@host122 Deploy]# docker push 172.28.250.122/webserver/nginx:1.9.0

The push refers to a repository [172.28.250.122/webserver/nginx]

5f70bf18a086: Pushed

b91b7f6cd891: Pushed

7ec8eb54c64e: Pushed

6eb35183d3b8: Pushed

1.9.0: digest: sha256:efad78a72d136edbf57ebadd7739239dc6c77b3d6a1a477ff6eecea2854b2250 size: 1956

#登录到web ui界面查看

第4章. docker容器管理

docker容器文件在/var/lib/docker/containers目录下。首先启动启动docker，确保docker正常运行

[root@docker02 ~]# /etc/init.d/docker start

[root@docker02 ~]# ps -ef|grep docker |grep -v grep

root 20895 1 1 15:12 ? 00:00:00 /usr/bin/docker -d --selinux-enabled

[root@docker02 ~]# docker info

Containers: 0

Images: 0

Storage Driver: devicemapper

Pool Name: docker-253:1-971077-pool

Pool Blocksize: 65.54 kB

Backing Filesystem: xfs

Data file: /dev/loop0

Metadata file: /dev/loop1

Data Space Used: 307.2 MB

Data Space Total: 107.4 GB

Data Space Available: 17.26 GB

Metadata Space Used: 729.1 kB

Metadata Space Total: 2.147 GB

Metadata Space Available: 2.147 GB

……

以下介绍的命令都可以通过help来查询使用

docker --help

docker ps --help

docker images --help

docker rmi --help

docker inspect --help

……

4.1 创建第一个容器

使用docker run命令创建容器，docker run命令提供了docker容器的创建到启动的功能。启动容器前提是能有镜像，否则启动会失败，docker也会自动连接到get.docker.in网站下载镜像（网速问题一般下载可能会出现中断情况）。

镜像仓库地址：<https://registry.hub.docker.com/> ，下面是拉取centos:7的镜像。

[root@docker02 ~]# docker pull centos:7

7: Pulling from docker.io/centos

6941bfcbbfca: Pull complete

41459f052977: Pull complete

fd44297e2ddb: Already exists

docker.io/centos:7: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.

Digest: sha256:7717ea9c410eeef2b38911c19725039976644ce7b3e585d9373bf069d3bd6a3c

Status: Downloaded newer image for docker.io/centos:7

创建第一个容器

[root@docker02 docker]# docker run -i -t centos:7 /bin/bash

[root@a2e342149420 /]# uname -r

3.10.0-123.el7.x86\_64

[root@db6f52ec39f3 /]# ps -aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.0 0.1 11744 1900 ? Ss 19:52 0:00 /bin/bash

root 35 0.0 0.1 19764 1248 ? R+ 19:53 0:00 ps -aux

-i #🡸保证容器中的STDIN是开启的，表示可以通过标准输入支持交互式输入。

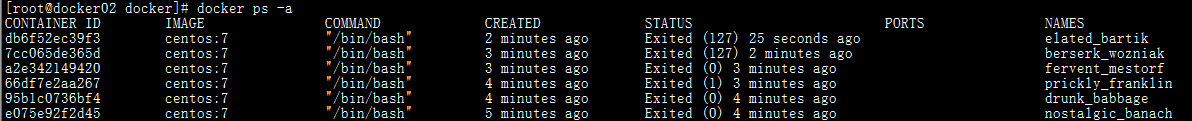
-t #🡸打开一个伪tty终端。

centos #🡸表示运行的镜像,docker会检查本地是否存在centos镜像，如果本地没有则会连接到官方维护的Docker Hub Registry，查看Docker Hub中是否有该镜像，一旦找到，则将下载该镜像到本地主机。

/bin/bash #🡸表示使用该容器启动了一个shell。

退出输入exit，该容器就会停止运行。dokcer ps -a查看正在运行和退出的容器

[root@docker02 docker]# docker ps -a



有三种方式可以指定唯一容器，容器ID（CONTAINER ID）、长UUID、容器名（NAMES）

以下命令是创建一个特权容器，特权容器可以对容器内sysctl.conf等核心文件调整。

[root@docker01 ~]# docker run -it --privileged --name oracle01 -v /u01/app/oracle:/u01/app/oracle -v /u01/tmp/database:/u01/tmp/database jame01/oracle:v1 /bin/bash

4.2 命名容器

使用--name参数实现命名容器，一个合法的容器名称只能包含这些字符：小写字母a-z、大写字母A-Z、数字0-9、下划线、圆点、横线。一般来讲容器名称使用该容器的功能命名比较容器管理。

[root@docker02 docker]# docker run --name apache\_container -i -t centos:7 /bin/bash

[root@7ef06a777001 /]# exit

[root@docker02 docker]# docker ps -a|grep apache

7ef06a777001 centos:7 "/bin/bash" 11 seconds ago Exited (0) 9 seconds ago apache\_container

提示：容器的命名必须是唯一的，否则会创建失败，可以通过docker rm/rename删除或重命名已创建的容器名。

[root@docker02 docker]# docker rm apache\_container #🡸移除容器

apache\_container

使用--hostname参数来固定容器的主机名。

[root@localhost ~]# docker run -itd --hostname=docker\_test4 --name test4 centos:6 /bin/bash

[root@localhost ~]# docker exec -it test4 /bin/bash

[root@docker\_test4 /]# #🡸进入容器主机名为设定名称

4.3 启动容器

第一次创建完容器后，退出时容器会自动停止，那么如何再次启动该容器呢？使用如下命令：

[root@docker02 docker]# docker start apache\_container

apache\_container

或者

[root@docker02 docker]# docker start c1e36cdf136f #🡸使用端UUID方式指定容器

[root@docker02 docker]# docker ps -a|grep apache

c1e36cdf136f centos:7 "/bin/bash" 2 minutes ago Up 12 seconds apache\_container

#docker restart apache\_container #🡸表示重启容器

#docker stop apache\_container #🡸表示停止容器

提示：man docker查看更详细命令手册

如果由于某种原因导致容器停止运行，也可以通过--restart参数，让docker自动重新启动容器。--restart参数会自动检查容器的退出代码，以此来决定是否要重启容器。默认情况不会自动重启容器。

[root@docker02 ~]# docker run --restart=always --name container1 -d centos:7 /bin/sh -c "while true;do echo hello world;sleep 1;done"

#--restart=always #🡸always表示无论容器的退出代码是什么，都会自动重启该容器。

#--restart=on-failure #🡸当容器的退出代码非0时会自动重启容器，on-failure:5表示授权最多重启5次。

4.4 进入容器

进入当前正在运行的容器，使用attach命令

[root@docker02 docker]# docker attach apache\_container

[root@c1e36cdf136f /]#

在容器外执行交互命令

[root@docker02 docker]# docker exec -it test1 ifconfig

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

inet6 addr: ::1/128 Scope:Host

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

4.5 删除容器

如果容器不再使用，可以通过rm命令删除。注意，删除之前需要停止容器，否则无法删除。

[root@docker02 ~]# docker rm test #🡸容器名称

[root@docker02 ~]# docker rm 6938e08ff369 #🡸容器ID号

如果删除全部容器，可以通过下面命令。

[root@docker02 ~]# docker rm `docker ps -a -q` #🡸删除全部容器

通过镜像ID删除带有none镜像相关联的容器方法

docker ps -a |grep "`docker images|grep none|awk '{print $3}'`"|awk '{print $1}'|xargs docker rm

批量删除none镜像方法

docker images|grep none|awk '{print $3}'|xargs docker rmi

4.6 创建守护式容器

守护式容器没有交互式会话，可以长期运行，适合运行应用程序和服务。这里创建一个容器名为test的容器，在这个容器里面执行了while循环操作, “-d”表示在后台运行。

[root@docker02 docker]# docker run --name test -d centos:7 /bin/sh -c "while true;do echo hello world;sleep 1;done"

e6eac629faeb85c97a3ecd67de56753f7d3a986199d2d4869e01d8d692a812dd

[root@docker02 docker]# docker ps -a|grep test

e6eac629faeb centos:7 "/bin/sh -c 'while t 3 minutes ago Up 3 minutes test

使用日志功能查看容器执行的程序

[root@docker02 docker]# docker logs test #🡸全部显示出来

hello world

hello world

hello world

[root@docker02 docker]# docker logs -f test #🡸持续输出，和tail -f功能相似

hello world

hello world

……

[root@docker02 docker]# docker logs --tail 3 test #🡸显示最后三条

hello world

hello world

hello world

[root@docker02 docker]# docker logs --tail 0 -f test #🡸从最新的地方开始持续输出

[root@docker02 docker]# docker logs --tail 0 -ft test #🡸加上时间戳来持续输出

2015-06-07T20:39:12.930007817Z hello world

2015-06-07T20:39:13.932572105Z hello world

2015-06-07T20:39:14.934507601Z hello world

2015-06-07T20:39:15.936063568Z hello world

4.7 查看容器内的进程

[root@docker02 docker]# docker top test

UID PID PPID C STIME TTY TIME CMD

root 22642 20895 0 16:28 ? 00:00:00 /bin/sh -c while true;do echo hello world;sleep 1;done

root 23529 22642 0 16:40 ? 00:00:00

4.8 交互式管理容器

可以通过docker exec命令在容器内部额外启动新进程。可以在容器内运行的进程有两种类型：后台任务和交互式任务。后台任务在容器内运行且没有交互需求，而交互式任务则保持在前台运行。

[root@docker02 ~]# docker exec -d apache\_container touch /etc/test.txt

#🡸这句表示通过内部执行命令，在apache\_container容器内创建了一个test.txt文件

在apache\_container容器中启动一个shell交互式任务，使用docker -exec方式进入容器进行管理，退出时不会停止当前管理运行的容器。

[root@docker02 ~]# docker exec -i -t apache\_container /bin/bash

[root@c1e36cdf136f /]# ls /etc/test.txt

/etc/test.txt

[root@c1e36cdf136f /]# exit

[root@docker02 ~]# docker ps -l

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

c1e36cdf136f centos:6 "/bin/bash" About a minute ago Up 14 seconds apache\_container

4.9 查看容器详细信息

除了通过docker ps命令获取容器的信息，还可以使用docker inspect来获取更多的容器信息。docker inspect会对容器进行的详细的检查，返回的配置信息包括：名称、命令、网络配置等。

[root@docker02 ~]# docker inspect apache\_container

[{

"AppArmorProfile": "",

"Args": [],

"Config": {

"AttachStderr": true,

"AttachStdin": true,

"AttachStdout": true,

"Cmd": [

"/bin/bash"

……

可以通过--format参数来指定输出的值。注意大括号里面两边有空格，多个显示输入多个匹配的值。

[root@docker02 ~]# docker inspect --format '{{ .Name }} {{ .NetworkSettings.IPAddress }}' apache\_container

/apache\_container 172.17.0.16

可以使用docker inpsec -f来查看容器ID，然后拷贝文件到容器内部或拷贝出来，docker cp命令只能将容器文件拷贝到宿主主机。

[root@docker02 ~]# docker inspect -f "{{.Id}}" web01 e7105af461f07785ec79832e363a9df3c56f9a839999bf4afe6d2183d443ad5a

# ll /var/lib/docker/devicemapper/mnt/e7105af461f07785ec79832e363a9df3c56f9a839999bf4afe6d2183d443ad5a

total 24

-rw-------. 1 root root 64 Oct 23 15:13 id

drwx------. 2 root root 16384 Oct 23 15:02 lost+found

drwxr-xr-x. 23 root root 4096 Oct 27 15:53 rootfs

4.10 数据卷容器管理

4.10.1 数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

* 数据卷可以在容器之间共享和重用
* 对数据卷的修改会立马生效
* 对数据卷的更新，不会影响镜像
* 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。

1）创建一个数据卷

在用 docker run 命令的时候，使用 -v 标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用可以挂载多个数据卷。下面创建一个 web 容器，并加载一个数据卷到容器的 /webapp 目录。

[root@host122 \_data]# docker run -d -P --name web03 -v /webapp apache-2.2.27

# -P表示自动将容器EXPOSE的端口映射到宿主机的随机端口(大于32768以上的端口)。

[root@host122 \_data]# docker inspect web03|grep webapp

"/webapp": "/var/lib/docker/volumes/65c3e6b82cc1d7bd4ed16c27a86af636655b3aa519851d63628486a18c245626/\_data"

"/webapp": true

"/webapp": {}

#查看-v /webapp在宿主机的实际位置。这里的添加方式和在Dockerfile里面使用VOLUME指令效果一样。

#docker inspect -f {{.Volumes}} container/images

2）挂载一个主机目录作为数据卷

使用 -v 标记也可以指定挂载一个本地主机的目录到容器中去。

docker run -d --name web05 -v /dockerdata:/tmp/dockerdata apache-2.2.27

#表示将宿主机目录的/dockerdata目录挂载到容器中的/tmp/dockerdata目录。

#本地目录的路径(/dockerdata)必须是绝对路径，如果目录不存在 Docker 会自动为你创建它。

# Docker 挂载数据卷的默认权限是读写，用户也可以通过 /dockerdata:/tmp/dockerdata:ro 指定为只读。

\*注意：Dockerfile 中不支持这种用法，这是因为 Dockerfile 是为了移植和分享用的。然而，不同操作系统的路径格式不一样，所以目前还不能支持。

3）挂载一个本地主机文件作为数据卷

-v 标记也可以从主机挂载单个文件到容器中。

docker run -d --name web06 -v ~/.bash\_history:/.bash\_history apache-2.2.27

\*注意：如果直接挂载一个文件，很多文件编辑工具，包括 vi 或者 sed --in-place，可能会造成文件 inode 的改变，从 Docker 1.1.0起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

4.10.2 数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。数据卷容器也是一个正常的容器，专门用来提供一个数据卷供其它挂载使用。

1. 创建一个数据卷容器dbdata01

docker run -d -v /dbdata01 --name dbdata01 centos:6 echo Data-only container for postgres data

#该容器不需要处于运行状态

1. 创建其他容器使用 --vloumes-from来挂载/dbdata01容器中的数据卷

docker run -d --volumes-from dbdata01 --name web07 apache-2.2.27

docker run -d --volumes-from dbdata01 --name web08 apache-2.2.27

#可以使用多个 --volumes-from 参数来从多个容器挂载多个数据卷。 也可以从其他已经挂载了数据卷的容器来挂载数据卷。

docker run -d --volumes-from web08 --name web09 apache-2.2.27

\*注意：如果删除了挂载的容器（包括 dbdata、db1 和 db2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 docker rm -v 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。

4.10.3 数据卷容器维护

可以利用数据卷对其中的数据进行进行备份、恢复和迁移。

1）备份

#可以直接备份数据卷所对应的宿主机物理目录

# docker inspect -f {{.Volumes}} dbdata01

map[/dbdata01:/var/lib/docker/volumes/d3c75ee0e9e35764e780536089b9be21708559b07df5f6014a054988a0a34146/\_data]

#备份数据卷脚本

#!/bin/bash

# author: Jame Yang

set -e

if [ -z $1 ];then

echo "Usage: $0 {Containername|ContainerID}"

exit 0

fi

containerName=$1

dbDatapath=`docker inspect -f {{.Volumes}} $containerName|awk -F'[]:]++' {'print $2'}`

dbBackup=/root/backup

if [ ! -d $dbBackup ];then

mkdir $dbBackup

fi

if [ -n $containrName ];then

cd $dbDatapath && \

tar -czf $dbBackup/${containerName}-$(date +%F).tar.gz ./\* >/dev/null 2>&1

fi

2）恢复

如果要恢复数据到一个容器，首先创建一个带有数据卷的容器 dbdata2。然后恢复数据到数据卷容器。

docker run -d -v /dbdata02 --name dbdata02 centos:6

docker inspect -f {{.Volumes}} dbdata02

4.11 容器访问

4.11.1 外部访问容器

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 -P 或 -p 参数来指定端口映射。当使用 -P 标记时，Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。

4.11.2 容器互联

容器的连接（linking）系统是除了端口映射外，另一种跟容器中应用交互的方式。该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

自定义容器命名连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器命名。虽然当创建容器的时候，系统默认会分配一个名字。自定义命名容器有2个好处：

* 自定义的命名，比较好记，比如一个web应用容器我们可以给它起名叫web
* 当要连接其他容器时候，可以作为一个有用的参考点，比如连接web容器到db容器。

使用 --name 标记可以为容器自定义命名。

$ sudo docker run -d -P --name web training/webapp python app.py

使用 docker ps 来验证设定的命名。

$ sudo docker ps -l

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

aed84ee21bde training/webapp:latest python app.py 12 hours ago Up 2 seconds 0.0.0.0:49154->5000/tcp web

也可以使用 docker inspect 来查看容器详细信息，下面是查看容器名。

$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde

/web

注意：容器的名称是唯一的。如果已经命名了一个叫 web 的容器，当你要再次使用 web 这个名称的时候，需要先用docker rm 来删除之前创建的同名容器。在执行 docker run 的时候如果添加 --rm 标记，则容器在终止后会立刻删除。注意，--rm 和 -d 参数不能同时使用。

* **使用 --link 参数可以让容器之间安全的进行交互。**

下面先创建一个新的数据库容器。

$ sudo docker run -d --name db training/postgres

然后创建一个新的 web 容器，并将它连接到 db 容器

$ sudo docker run -d -P --name web --link db:db training/webapp python app.py

此时，db 容器和 web 容器建立互联关系。--link 参数的格式为： --link name:alias，其中 name 是要链接的容器的名称，alias 是这个连接的别名。

使用 docker ps 来查看容器的连接：

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

349169744e49 training/postgres:latest su postgres -c '/usr About a minute ago Up About a minute 5432/tcp db, web/db

aed84ee21bde training/webapp:latest python app.py 16 hours ago Up 2 minutes 0.0.0.0:49154->5000/tcp web

可以看到自定义命名的容器，db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主主机上。在启动 db 容器的时候并没有使用 -p 和 -P 标记，从而避免了暴露数据库端口到外部网络上。

* **Docker 通过 2 种方式为容器公开连接信息：**
* 环境变量
* 更新 /etc/hosts 文件

使用 env 命令来查看 web 容器的环境变量，其中 DB\_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。

$ sudo docker run --rm --name web2 --link db:db training/webapp env

. . .

DB\_NAME=/web2/db

DB\_PORT=tcp://172.17.0.5:5432

DB\_PORT\_5000\_TCP=tcp://172.17.0.5:5432

DB\_PORT\_5000\_TCP\_PROTO=tcp

DB\_PORT\_5000\_TCP\_PORT=5432

DB\_PORT\_5000\_TCP\_ADDR=172.17.0.5

除了环境变量，Docker 还添加 host 信息到父容器的 /etc/hosts 的文件。下面是父容器 web 的 hosts 文件。

sudo docker run -t -i --rm --link db:db training/webapp /bin/bash

root@aed84ee21bde:/opt/webapp# cat /etc/hosts

172.17.0.7 aed84ee21bde

. . .

172.17.0.5 db

用户可以链接多个父容器到子容器，比如可以链接多个 web 到 db 容器上。

第5章. 使用Dockerfile构建镜像

前面介绍了docker基础知识，了解什么是镜像，基本的启动流程，以及如何运作容器。了解这些基础知识后，下面将试着在实际开发和测试过程中使用docker。首先来看看docker如何使开发和测试更加流程化，效率更高。

本章将会演示下面三个应用场景：

* 使用docker测试一个静态网站。
* 使用docker创建并测试一个web应用。
* 将docker用于持续集成。

5.1 Dockerfile

上面配置的Dockerfile文件简单的写入几个指令，Dockerfile还可以写入更多的指令，包括：CMD、ENTRYPOINT、ADD、COPY、VOLUME、WORKDIR、USER、ONBUILD和ENV等。更多请查看官方地址： <http://docs.docker.com/reference/builder/>

docker build语法：

docker build [options] PATH | URL

#docker还提供格外选项，如：

--rm=true 表示构建成功后，移除所有中间容器

--no-cache=false 表示在构建过程中不使用缓存

5.1.1 CMD

CMD命令用于指定一个容器启动时要运行的命令，类似于RUN命令，只是RUN命令是指定镜像被构建时要运行的命令，而CMD是指定容器被启动时要运行的命令。支持三种格式。

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 ENTRYPOINT 的默认参数；

如果用户启动容器时候指定了运行的命令，则会覆盖掉 CMD 指定的命令。

[root@docker01 testimg]# cat Dockerfile

#version 0.0.1

FROM centos:6

MAINTAINER yang "yang.hong@abc.com"

CMD ["/bin/bash"]

[root@docker01 testimg]# docker build -t="testimg/test\_01" .

[root@docker01 testimg]# docker run -i -t --name test\_container01 testimg/test\_01

[root@3c4df87f52eb /]# hostname

3c4df87f52eb

#🡸上面未指定/bin/bash就进入容器了，是因为执行了CMD指令。注意CMD指令不能有多个，如果有多个也是最后一条指令生效。docker run指定的指令优先级高于CMD指令

5.1.2 ENTRYPOINT

ENTRYPOINT指令和CMD类似，也是用于容器启动时的命令，它不容易被docker run指定命令覆盖。实际上是在docker run命令行中指定的任何参数都会被当做参数再次传递给ENTRYPOINT指令中指定的命令。

[root@docker01 apache\_web]# cat Dockerfile

#version 0.0.1

FROM centos:6

MAINTAINER yang "yang.hong@abc.com"

RUN yum install -y httpd

RUN echo 'Hi,I am in your container' > /var/www/html/index.html

EXPOSE 80

ENTRYPOINT ["/etc/init.d/httpd"]

[root@docker01 apache\_web]# docker build -t="httpd/apache\_web01" .

[root@docker01 apache\_web]# docker run -i -t httpd/apache\_web01 start

Starting httpd: [ OK ]

#🡸这里docker run后面的start参数传递给ENTRYPOINT定义的/etc/init.d/httpd指令

#🡸如果想覆盖定义好的ENTRYPOINT，可以在docker run后面指定—entrypoint

也可以使用数组形式来定义启动脚本，例如这里配置了脚本为test.sh，放在/root下面，那么构建容器后就启动该脚步。

ENTRYPOINT [“sh”,”/root/test.sh”]

5.1.3 WORKDIR

WORKDIR指令用来从镜像创建一个新容器时，在容器内部设置一个工作目录，ENTRYPOINT和CMD指定的程序会在这个目录下执行。

[root@docker01 testimg]# cat Dockerfile

#version 0.0.1

FROM centos:6

MAINTAINER yang "yang.hong@abc.com"

WORKDIR /opt/webapp/db

RUN touch file1

WORKDIR /opt/webapp

RUN touch file2

[root@docker01 testimg]# docker run -it --name test001 test/test:v1 /bin/bash

[root@c21d40246109 webapp]# pwd

/opt/webapp

[root@c21d40246109 webapp]# ll

total 4

drwxr-xr-x 2 root root 4096 Jun 9 01:12 db

-rw-r--r-- 1 root root 0 Jun 9 01:12 file2

#🡸将工作目录设定为/opt/webapp/db，后运行了touch file1，之后又将工作目录切换为/opt/webapp，运行了touch file2

5.1.4 ENV

ENV指令用来在镜像构建过程中设置环境变量。该环境变量可以在后续的任何RUN、WORKDIR等指令中使用

ENV RVM\_PATH /home/rvm

ENV TARGET\_DIR /opt/app

RUN touch $RVM\_PATH/test.txt

WORKDIR $TARGET\_DIR

可以使用docker run -ti -e临时指定ENV环境变量，只在本次运行时生效

[root@docker01 ~]# docker run -it -e "WEB\_PORT=8080" centos:6 env

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=c9bbeef958e7

TERM=xterm

WEB\_PORT=8080

HOME=/root

5.1.5 USER

USER指令用来指定该镜像会以什么样的用户去运行。

USER user

USER user:group

USER uid

USER uid:gid

提示：也可以使用docker run命令中通过-u来覆盖该指令指定的值。不指定默认是root来运行。

5.1.6 VOLUME

VOLUME指令用来向基于镜像创建的容器添加卷。一个卷是可以存在于一个或着多个容器内的特定的目录，这个目录可以绕过联合文件系统，并提供如下共享数据或这对数据进行持久化的功能。

* 卷可以在容器之间共享和重用；
* 一个容器可以不是必须和其他容器共享；
* 对卷的修改是立刻生效的；
* 对卷的修改不会对更新镜像产生影响；
* 卷会一直存在知道没有任何容器再使用它。

卷功能可以让我们将数据、数据库或这其他内容添加到镜像中而不是将这些内容提交到镜像中。并且允

许在多个容器间共享这些内容。可以利用此功能来测试容器和内部的应用程序代码，管理日志，或者处理容器内部的数据库。

* 通过在Dockerfile里面来指定挂载点，是将容器内指定某个目录挂载到宿主主机。

#🡸将会为基于此镜像创建的任何容器创建一个名为/opt/project的挂载点。

VOLUME [“/opt/project”]

#🡸指定多个挂载点。

VOLUME [“/opt/project”,”/data”]

#🡸查看volume挂载位置，表示将容器内/data目录挂载到宿主主机的/var/lib/docker/vfs……位置。

[root@docker01 ]# docker inspect -f {{.Volumes}} container/images

map[/data:/var/lib/docker/vfs/dir/cde167197ccc3e138a14f1a4f...b32cec92e79059437a9]

#🡸删除容器且删除没有其他容器连接的卷

docker rm -v container

* 也可以在容器创建时加入指定VOLUME，例如下面将宿主主机的/tools目录映射到了容器中的/root/tools目录。这个和在Dockerfile里不同，创建容器时将宿主机指定目录挂载进来。

[root@node2 tools]# docker run -it --name web\_01 -v /tools:/root/tools apache/web:v0.0.1 /bin/bash

# Docker 挂载数据卷的默认权限是读写，用户也可以通过 :ro 指定为只读。docker run -it --name web\_01 -v /tools:/root/tools apache/web:v0.0.1:ro /bin/bash 加入ro表示将该目录设定为只读状态。

# 映射的本地目录必须为绝对路径，如果本地目录不存在 Docker 会自动为你创建它。

5.1.7 ADD

ADD指令用来将构建环境下的文件和目录复制到镜像中。比如，在安装一个应用程序时，ADD指令需要源文件位置和目的文件位置两个参数。

ADD software.lic /opt/application/software.lic

ADD http://wordpress.org/latest.zip /root/wordpress.zip

#🡸这里ADD指令将会将构建目录下的software.lic文件复制到镜像中的/opt/application/software.lic。

#🡸指向源文件的位置参数可以是一个URL，或者构建上下文或环境中文件名或者目录，ADD的URL合法归档文件只能是gzip、bzip2、xz，URL方式不支持.tar.gz，如果以文件方式则支持.tar.gz，会自动解压到指定目录。

#🡸在ADD文件时，docker通过目的地址参数末尾的字符来判断文件源是目录还是文件。如果目标地址以/结尾，那么docker就会认为源位置指向的是一个目录。如果目的地址不是以/结尾，那么docker就认为源位置指向的是文件。

#🡸注意,ADD指令不能对构建目录或上下文之外的文件进行ADD操作。

ADD latest.tar.gz /root/www/wordpress/

#🡸这条命令会将归档文件latest.tar.gz解压到/root/www/wordpress/目录下。

提示：

* 如果目的位置不存在，docker将会为我们创建这个全路径，新创建的文件和目录的权限为0755,并且UID和GID都为0。
* ADD指令会使得构建的缓存变得无效，如果通过ADD 指令向镜像添加一个文件或者目录，那么将使得Dokcerfile中的后续指令都不能继续使用之前的构建缓存。

5.1.8 COPY

COPY指令类似于ADD，只是它只关心在构建上下文中复制本地文件，而不会去做文件提取（extraction）和解压（decompression）的工作。

COPY conf.d/ /etc/apache2/

#🡸这条指令将会把本地conf.d目录中的文件复制到/etc/apache2/目录中。

#🡸文件源路径必须与当前构建环境相对的文件或目录，本地文件都放到和Dockerfile同一个目录下。它不能复制该目录之外的任何文件。因为构建环境将会上传到Docker守护进程，而复制是在docker守护进程中进行的。任何位于构建环境之外的东西都是不可用的。

#🡸COPY指令的目的位置必须是容器内部的一个绝对路径，如果目的位置不存在，docker会自动创建目录结构。

#🡸任何由该指令创建的文件或目录的UID和GID都会设置为0。

5.1.9 ONBUILD

ONBUILD指令能为镜像添加触发器(trigger)，当一个镜像被用做其他镜像的基础镜像时，该镜像中的触发器将会被执行，注意，它只能被继承到子镜像，不能继承到孙子镜像中。触发器会在构建过程中插入新的指令，我们可以认为这些指令是紧跟在FROM之后指定的。

ONBULID ADD . /app/src

ONBUILD RUN cd /app/src && make

#🡸这两行代码表示会在创建的镜像中加入ONBUILD触发器，可以通过docker inspect 命令查看

如下实例：

父镜像中的Dockerfile

#version 0.0.1

FROM centos:6

MAINTAINER yang.hong

RUN yum install -y httpd

ENV APACHE\_RUN\_USER www

ENV APACHE\_RUN\_GROUP WWW

ENV APACHE\_LOG\_DIR /var/log/apache2

ONBUILD ADD www/index.html /var/www/html/

EXPOSE 80

父镜像构建完成，摘取关于ONBUILD部分

[root@node2 web\_01]# docker inspect d6f30d242e2d|egrep -i 'OnBuild|ADD'

"MacAddress": "",

"OnBuild": [

"ADD www/index.html /var/www/html/"

基于父镜像来构建子镜像，观察过程，先定义子镜像Dockerfile。

#version 0.0.1

FROM apache/web:v0.0.1 #🡸这里拉取的基础镜像是父镜像

MAINTAINER yang.hong

ENV APPLICATION\_NAME webapp

ENV ENVIRONMENT development

观察构建过程是否触发了父镜像设定的ONBUILD。

[root@node2 web\_01]# docker build -t="web\_01/web\_01:v0.0.1" .

Sending build context to Docker daemon 3.584 kB

Sending build context to Docker daemon

Step 0 : FROM apache/web:v0.0.1

# Executing 1 build triggers #🡸子镜像构建过程中触发了父镜像设定的ONBUILD

Trigger 0, ADD www/index.html /var/www/html/

Step 0 : ADD www/index.html /var/www/html/

---> 0d3bc5abab34

Removing intermediate container 63aba155647d

Step 1 : MAINTAINER yang.hong

---> Running in 555f882d48f6

---> 692807403eea

Removing intermediate container 555f882d48f6

Step 2 : ENV APPLICATION\_NAME webapp

---> Running in cc3bb3525730

---> 0bb95b19970f

Removing intermediate container cc3bb3525730

Step 3 : ENV ENVIRONMENT development

---> Running in 363a07eb4376

---> a90a8763443f

Removing intermediate container 363a07eb4376

Successfully built a90a8763443f

创建一个容器，查看触发器生成的结果

[root@node2 web\_01]# docker run -it --name webapp web\_01/web\_01:v0.0.1 /bin/bash

[root@90968cf56c47 ~]# cat /var/www/html/index.html

This test container for apache.

5.1.10 EXPOSE

因为EXPOSE通常只是作为记录机制，也就是告诉用户哪些端口会提供服务，Docker可以很容易地把Dockerfile里的EXPOSE指令转换成特定的端口绑定规则。只需要在运行时加上-P参数，Docker会自动为用户创建端口映射规则，并且帮助避免端口映射的冲突。

添加如下行到上文使用的Web应用Dockerfile里：

EXPOSE 1000

EXPOSE 2000

5.2 Dockerfile构建镜像

将docker作为本地web开发环境是使用docker的一个最简单的场景。这个环境可以完全重现生成环境，保证开发环境和部署环境一致。下面将从apache安装到容器来架构一个简单的网站。

这里通过两种方式来创建容器，第一种是非守护式方式来创建的；第二种是守护式方式构建容器，这也是在应用服务上面最常用的方式，达到容器即服务，容器启动服务启动，容器停止服务停止。

5.2.1 非守护式方式构建容器

1. 创建构建环境目录

mkdir /images/apache\_web && touch /images/apache\_web/Dockerfile

1. 编写基本Dockerfile

#version 0.1

FROM centos:6

MAINTAINER yang [yang.hong@abc.com](mailto:yang.hong@abc.com)

RUN useradd mockbuild -g mockbuild

WORKDIR /root/tools/

ADD wget-1.12-5.el6.x86\_64.rpm /root/tools

RUN rpm -ivh wget-1.12-5.el6.x86\_64.rpm

WORKDIR /etc/yum.repos.d/

RUN for n in `ls /etc/yum.repos.d/`;do mv $n $n.bak;done

RUN wget -O /etc/yum.repos.d/CentOS-Base.repo <http://mirrors.aliyun.com/repo/Centos-6.repo>

RUN yum makecache

RUN yum install -y httpd

RUN mkdir -p /var/www/html

ADD www/index.html /var/www/html/

WORKDIR /root/

EXPOSE 80

1. 开始构建apache镜像

[root@docker01 apache\_web]# docker build -t="apache/web:v0.1" .

[root@docker01 apache\_web]# docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

apache/web v0.1 b138429650ce 2 minutes ago 563.8 MB

centos centos6 b9aeeaeb5e17 6 weeks ago 202.6 MB

centos 6 b9aeeaeb5e17 6 weeks ago 202.6 MB

1. 构建一个基于apache/web:v0.1镜像的web容器，并且将web容器的80映射到宿主主机的8080端口

[root@docker01 apache\_web]# docker run -it -p 8080:80 --name web\_01 apache/web:v0.1 /bin/bash [root@7b5dbadce72d ~]# cat /var/www/html/index.html

This is container.

[root@docker01 apache\_web]# docker start web\_01

web\_01

[root@docker01 apache\_web]# docker exec -it web\_01 /etc/init.d/httpd start

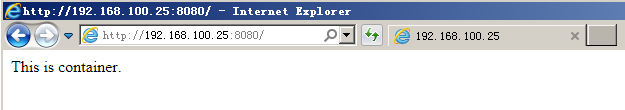
172.17.0.26 for ServerName [ OK ]

[root@docker01 apache\_web]# docker ps -l

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7b5dbadce72d apache/web:v0.1 "/bin/bash" About a minute ago Up 15 seconds 0.0.0.0:8080->80/tcp web\_0

1. 访问测试构建的web容器，通过宿主主机地址来访问：<http://192.168.100.25:8080>



6）网站的站点目录可以挂载宿主主机指定的目录，通过修改宿主主机的指定目录相关文件，就能更新网站站点文件。这个比较常用于多容器之间共享数据的场景。下面是将宿主主机的/var/website/挂在到容器的/var/www/html目录。

[root@docker01 apache\_web]# mkdir /var/website

[root@docker01 apache\_web]# echo 'web\_02 container' >/var/website/index.html

[root@docker01 apache\_web]# docker run -it --name web\_02 -p 8081:80 -v /var/website:/var/www/html apache/web:v0.1 /bin/bash

[root@8b09926f72b2 ~]# cat /var/www/html/index.html

web\_02 container

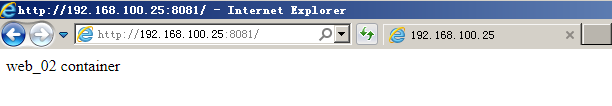
[root@docker01 apache\_web]# docker start web\_02

web\_02

[root@docker01 apache\_web]# docker exec -it web\_02 /etc/init.d/httpd start

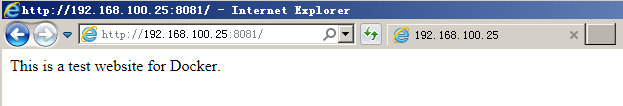
172.17.0.31 for ServerName [ OK ]

测试访问该站点，<http://192.168.100.25:8081>



更新宿主主机/var/website/index.html文件，再次测试

[root@docker01 ~]# echo 'This is a test website for Docker.' >>/var/website/index.html



**结论：**

通过上面构建apache静态网站，共享目录方式对生产场景应用很有帮助，可以给PHP，MySQL各配置一个容器，通过共享目录方式将各个容器连接到一起。

5.2.2 守护式方式构建容器

守护式容器与非守护式不同的是，守护式容器在容器启动时，该容器在Dockerfile里面指定构建的服务就会跟随启动，容器停止时，服务亦停止。

5.2.2.1 构建tomcat镜像

1）创建构建环境目录，准备添加文件

[root@host63 ~]# mkdir /images/tomcat -p

[root@host63 tomcat]# ll

total 158540

-rw-r--r-- 1 root root 388 Oct 10 14:53 Dockerfile

-rw-r--r-- 1 root root 8824528 Jun 1 13:33 apache-tomcat-7.0.62.tar.gz

-rw-r--r-- 1 root root 153512879 Jun 1 11:17 jdk-7u79-linux-x64.gz

2）编写构建文件Dockerfile

[root@host63 tomcat]# cat Dockerfile

#version 0.0.1

FROM centos:6

MAINTAINER jame.yang

ADD jdk-7u79-linux-x64.gz /usr/

ENV JAVA\_HOME /usr/jdk1.7.0\_79

ENV PATH $PATH:$HOME/bin:$JAVA\_HOME/bin

ADD apache-tomcat-7.0.62.tar.gz /application/

RUN ln -s /application/apache-tomcat-7.0.62/ /application/tomcat

ENTRYPOINT sh /application/tomcat/bin/startup.sh && tail -f /application/tomcat/logs/catalina.out

EXPOSE 8080

3）开始构建

[root@host63 tomcat]# docker build -t="tomcat-7.0.62" .

省略部分……

Step 8 : EXPOSE 8080

---> Running in 6b2dd188fd58

---> 0014dc0c2f6e

Removing intermediate container 6b2dd188fd58

Successfully built 0014dc0c2f6e

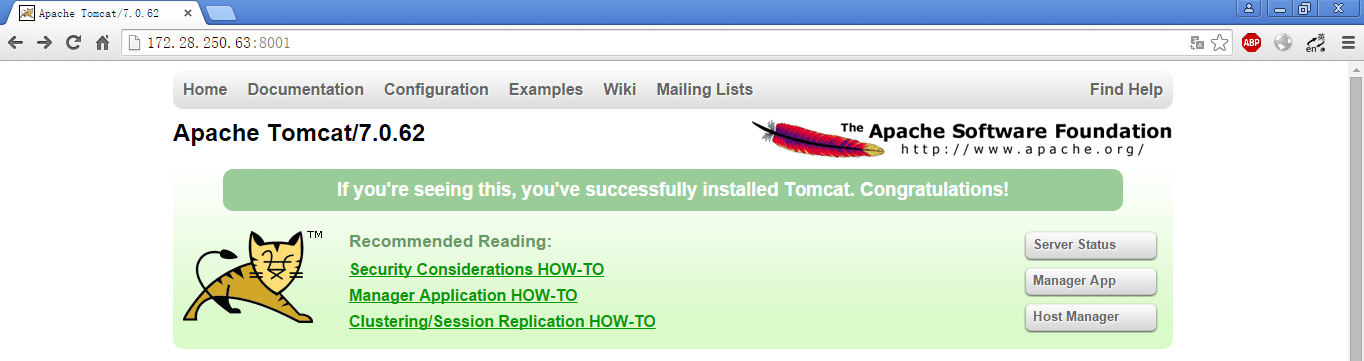
4）创建容器

[root@host63 tomcat]# docker run -d -p 8001:8080 --name tomcat01 tomcat-7

[root@host63 tomcat]# docker ps -a|grep tomcat

e038bb467611 tomcat-7.0.62 "/bin/sh -c 'sh /app 9 minutes ago Up 9 minutes 0.0.0.0:8001->8080/tcp tomcat01

#测试访问站点



5.2.2.2 构建MySQL镜像

（1）构建准备

[root@host103 mysql]# ll

total 29588

-rw-rw-rwT 1 root root 5691656 Jun 16 2015 cmake-2.8.8.tar.gz

-rw-r--r-- 1 root root 1877 Jan 14 18:33 Dockerfile

-rw-r--r-- 1 root root 3225 Jan 14 17:14 my.cnf

-rw-rw-rwT 1 root root 24596474 Jun 16 2015 mysql-5.5.32.tar.gz

（2）dockerfile

[root@host103 mysql]# cat Dockerfile

#version 0.1

FROM centos:6

MAINTAINER yang.hong@sand.com.cn

#yum install or update packages

RUN yum install ncurses-devel tar gcc gcc-c++ ncurses-devel -y

#add packages

ADD cmake-2.8.8.tar.gz /root/tools/

ADD mysql-5.5.32.tar.gz /root/tools/

ADD my.cnf /root/tools/

#create mysql user

RUN groupadd mysql && \

useradd mysql -s /sbin/nologin -M -g mysql

#compile install cmake and mysql

WORKDIR /root/tools

RUN cd cmake-2.8.8 && \

./configure && \

gmake && \

gmake install && \

cd ..

RUN cd mysql-5.5.32 && \

cmake . -DCMAKE\_INSTALL\_PREFIX=/application/mysql-5.5.32 \

-DMYSQL\_DATADIR=/application/mysql-5.5.32/data \

-DMYSQL\_UNIX\_ADDR=/application/mysql-5.5.32/tmp/mysql.sock \

-DEXTRA\_CHARSETS=gbk,gb2312,utf8,ascii \

-DENABLED\_LOCAL\_INFILE=ON \

-DWITH\_INNOBASE\_STORAGE\_ENGINE=1 \

-DWITH\_FEDERATED\_STORAGE\_ENGINE=1 \

-DWITH\_BLACKHOLE\_STORAGE\_ENGINE=1 \

-DWITHOUT\_EXAMPLE\_STORAGE\_ENGINE=1 \

-DWITHOUT\_PARTITION\_STORAGE\_ENGINE=1 \

-DWITH\_FAST\_MUTEXES=1 \

-DWITH\_ZLIB=bundled \

-DENABLED\_LOCAL\_INFILE=1 \

-DWITH\_READLINE=1 \

-DWITH\_EMBEDDED\_SERVER=1 \

-DWITH\_DEBUG=0 && \

make && make install && \

ln -s /application/mysql-5.5.32/ /application/mysql

#configure mysql environment variable

RUN echo 'export PATH=/application/mysql/bin:$PATH' >>/etc/profile && \

source /etc/profile && \

cp /root/tools/my.cnf /etc/

#init mysql

RUN mkdir -p /application/mysql/{data,logs} && \

chown -R mysql:mysql /application/mysql && \

chmod -R 1777 /tmp && \

cd /application/mysql/scripts && \

./mysql\_install\_db \

--basedir=/application/mysql/ \

--datadir=/application/mysql/data/ \

--user=mysql

#configure mysql start scripts

RUN cd /root/tools/mysql-5.5.32 && \

cp support-files/mysql.server /etc/init.d/mysqld && \

chmod +x /etc/init.d/mysqld

#mysql start

EXPOSE 3306

ENTRYPOINT /etc/init.d/mysqld start && tail -f /application/mysql/logs/mysql.error

第6章. docker网络介绍

网络方面是目前docker比较薄弱的地方，因此，我们有必要深入了解Docker的网络知识，以满足更高的网络需求。本章先介绍了Docker自身的4种网络工作方式，然后后面介绍关于常见网络配置，以及介绍借助于第三方工具来配置docker网络。

docker网络弱点主要包含这几部分：

•多主机网络问题

•外部只能通过port mapping访问容器

•网络隔离

•网络QoS

•流量监控

•数据包分析

6.1 docker网络模式

我们在使用docker run创建Docker容器时，可以用--net选项指定容器的网络模式，Docker有以下4种网络模式：

host模式 --net=host指定。

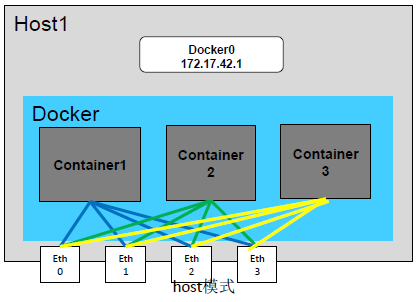
container模式 --net=container:NAME\_or\_ID指定。

none模式 --net=none指定。

bridge模式 --net=bridge指定，默认设置。

6.1.1 host模式

Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。但如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。



docker run -itd --net=host --name=web03 jame01/tomcat:v1

docker exec -it web03 ifconfig

通过ifconfig看到的都是宿主机上的网卡信息。而外界访问容器中的应用，则直接使用“宿主机IP:80”即可，不用任何NAT转换，就如直接跑在宿主机中一样。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

6.1.2 container模式

这个模式指定新创建的容器和已经存在的一个容器共享一个Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过lo网卡设备通信。

# docker run -itd --net=container:9fcf7d4e7079 -p 8088:80 --name=web02 jame01/apache:v1

# docker exec -it web01 ifconfig|awk -F'[: ]+' 'NR==2{print $4}'

172.17.0.4

# docker exec -it web02 ifconfig|awk -F'[: ]+' 'NR==2{print $4}'

172.17.0.4

#可以观察到两个容器IP是一样的。

6.1.3 none模式

这个模式和前两个不同。在这种模式下，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等。

# docker run -itd --net=none -p 8080:8080 --name=web04 jame01/tomcat:v1

# docker exec -it web04 ifconfig

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

inet6 addr: ::1/128 Scope:Host

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

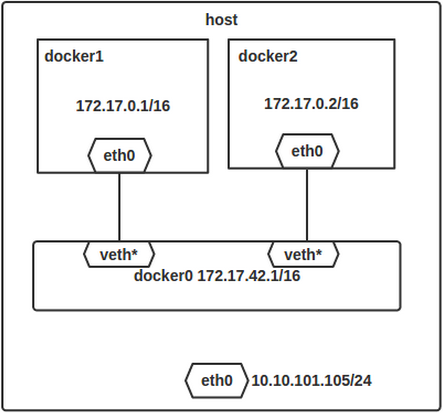
RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

6.1.4 bridge模式

该模式是docker默认的网路配置模式，此模式会为每一个容器分配Network Namespace、设置IP等，并将一个主机上的Docker容器连接到一个虚拟网桥上。

在bridge模式下，连在同一网桥上的容器可以相互通信（若出于安全考虑，也可以禁止它们之间通信，方法是在DOCKER\_OPTS变量中设置--icc=false，这样只有使用--link才能使两个容器通信）。

当Docker server启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的Docker容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。接下来就要为容器分配IP了，Docker会从RFC1918所定义的私有IP网段中，选择一个和宿主机不同的IP地址和子网分配给docker0，连接到docker0的容器就从这个子网中选择一个未占用的IP使用。如一般Docker会使用172.17.0.0/16这个网段，并将172.17.42.1/16分配给docker0网桥（在主机上使用ifconfig命令是可以看到docker0的，可以认为它是网桥的管理接口，在宿主机上作为一块虚拟网卡使用）。



容器也可以与外部通信，我们看一下主机上的iptable规则，可以看到这么一条

docker run -itd -p 8088:8080 --name web05 jame01/tomcat:v1

[root@docker01 ~]# iptables -t nat -L -n

Chain PREROUTING (policy ACCEPT)

target prot opt source destination

DOCKER all -- 0.0.0.0/0 0.0.0.0/0 ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)

target prot opt source destination

Chain OUTPUT (policy ACCEPT)

target prot opt source destination

DOCKER all -- 0.0.0.0/0 !127.0.0.0/8 ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)

target prot opt source destination

MASQUERADE all -- 172.17.0.0/16 0.0.0.0/0

MASQUERADE tcp -- 172.17.0.2 172.17.0.2 tcp dpt:8080

Chain DOCKER (2 references)

target prot opt source destination

DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 tcp dpt:8088 to:172.17.0.2:8080

Docker完成以上网络配置的过程大致是这样的：

1）在主机上创建一对虚拟网卡veth pair设备。veth设备总是成对出现的，它们组成了一个数据的通道，数据从一个设备进入，就会从另一个设备出来。因此，veth设备常用来连接两个网络设备。

2）Docker将veth pair设备的一端放在新创建的容器中，并命名为eth0。另一端放在主机中，以veth65f9这样类似的名字命名，并将这个网络设备加入到docker0网桥中，可以通过brctl show命令查看。

[root@docker01 ~]# brctl show

bridge name bridge id STP enabled interfaces

docker0 8000.56847afe9799 no vethbcf2ff8

vethcb834f8

3）从docker0子网中分配一个IP给容器使用，并设置docker0的IP地址为容器的默认网关。

6.2 docker第三方网络工具

Docker自身的网络功能比较简单，不能满足很多复杂的应用场景。因此，有很多开源项目用来改善Docker的网络功能，如pipework、weave、flannel等。

6.2.1 docker网络之pipework

pipework是由Docker的工程师Jérôme Petazzoni开发的一个Docker网络配置工具，由200多行shell实现，方便易用。pipework具有以下两个特点

* 一个集成了网络配置命令的的脚本，操作容器的net namespace。
* 用于配置容器网络：
* 添加容器的多网卡支持，实现更为复杂的网络拓扑结构
* 不能增强网络性能，只是一个配置网络的插件

pipework参考网址：<http://www.infoq.com/cn/articles/docker-network-and-pipework-open-source-explanation-practice/>

下面介绍pipework几种工作场景。

6.2.1.1 docker容器配置本地网络环境

为了使本地网络中的机器和Docker容器更方便的通信，我们经常会有将Docker容器配置到和主机同一网段的需求。这个需求其实很容易实现，我们只要将Docker容器和主机的网卡桥接起来，再给Docker容器配上IP就可以了。

下面操作一下，主机地址为：192.168.100.25/24,网关：192.168.100.1，需要给docker容器的地址为192.168.100.200/24。

1）安装pipework

git clone https://github.com/jpetazzo/pipework

cp ./pipework/pipework /usr/local/bin/

2）pipework命令用法

语法：

 pipework <hostinterface> [-i containerinterface] <guest> <ipaddr>/<subnet>[@default\_gateway] [macaddr][@vlan]

 pipework <hostinterface> [-i containerinterface] <guest> dhcp [macaddr][@vlan]

 pipework --wait [-i containerinterface]

# hostinterface: 主机上的网卡，可以为物理网卡、Linux bridge、Open vSwitch bridge

# containerinterface: 容器内的网卡名字如eth0、eth1

# guest: 容器名称或者容器的ID

# <ipaddr>/<subnet> : 配置容器内网卡的IP和subnet

# [@default\_gateway] :配置默认网关

# [macaddr][@vlan]:配置网卡的mac地址和vlan

# dhcp: 使用dhcp配置网卡

# 功能：给容器添加一块额外的网卡

3）使用pipework配置容器网络

#🡸启动Docker容器。

docker run -itd --name web05 jame01/tomcat:v1

#🡸配置容器网络，并连到网桥br0上。网关在IP地址后面加@指定。

pipework br0 web05 192.168.100.200/24@192.168.100.10

#🡸若主机环境中存在dhcp服务器，也可以通过dhcp的方式获取IP

#pipework br0 web05 dhcp

#🡸将主机eth0桥接到br0上，并把eth0的IP配置在br0上。这里由于是远程操作，中间网络会断掉，所以放在一条命令中执行。

#🡸将宿主主机地址添加到br0

#🡸将宿主主机地址从eth0上面删除掉

#🡸将br0和eth0桥接

ip addr add 192.168.100.25/24 dev br0; \

ip addr del 192.168.100.25/24 dev eth0; \

brctl addif br0 eth0

#🡸可以将下面命令加入到/etc/rc.local，后面新运行的容器只要执行上面pipework命令即可。可以按照下面的效果添加。注意，如果没有br0不存在，那么需要执行brctl addbr创建，具体参见brctl命令。

#🡸提示：执行下面的操作错误时会将网络中断掉，所以执行前最好写一个清除脚本加入任务计划里面，以防连不上的时候可以自动清除还原回来，否则，只能去机房或者通过管理控制台进去调整了。

cat >>/etc/rc.local<<EOF

brctl addbr br0

ip addr add 192.168.100.25/24 dev br0

ip addr del 192.168.100.25/24 dev eth0

ifconfig br0 up

brctl addif br0 eth0

route del default

route add default gw 192.168.100.10 dev br0

EOF

4）在宿主主机docker02上面测试

[root@docker02 ~]# curl -I http://192.168.100.200:8080/

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: text/html;charset=ISO-8859-1

Transfer-Encoding: chunked

Date: Tue, 14 Jul 2015 09:32:43 GMT

从上面测试来看，容器可以和宿主主机处于同一网络，那么这个方式即解决了容器互联问题，也解决了多宿主主机容器之间的互联问题，当然，对于之前出现的单宿主主机不能复用端口的情况也解决了。对于单宿主主机通过pipework工具为容器配置网络后，容器内的服务对外的端口可以和其他容器相同，因为此时IP不同。

错误提示1

**问题描述：**

在centos6.5用 pipework, ,报如下错：Object "nets" is unknown, try "ip help".把这个报错解决了，就可以用了。

**解决方法：**

#1 Centos6.5的内核已经支持network namespace, 但安装的iproute版本过低，不支持ip nets 命令。

升级iproute 至 iproute-2.6.32-130.el6ost.netns.2.x86\_64

方法如下：

yum install -y <http://rdo.fedorapeople.org/rdo-release.rpm>

#2 编辑这个文件地址，默认软件包地址已经失效

cat /etc/yum.repos.d/rdo-release.repo

[openstack-juno]

name=OpenStack Juno Repository

baseurl= https://repos.fedorapeople.org/repos/openstack/EOL/openstack-havana/epel-6/

enabled=1

skip\_if\_unavailable=0

gpgcheck=0

gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-RDO-Juno

#3 升级iproute

yum upgrade iproute -y

6.2.1.2 docker单主机容器VLAN划分

pipework不仅可以使用Linux bridge连接Docker容器，还可以与OpenVswitch结合，实现Docker容器的VLAN划分。下面，就来简单演示一下，在单机环境下，如何实现Docker容器间的二层隔离。

为了演示隔离效果，我们将4个容器放在了同一个IP网段中。但实际他们是二层隔离的两个网络，有不同的广播域。

1. 在主机A上创建4个docker容器，分别为test1、test2、test3、test4

docker run -itd --name test1 jame01/tomcat:v1

docker run -itd --name test2 jame01/tomcat:v1

docker run -itd --name test3 jame01/tomcat:v1

docker run -itd --name test4 jame01/tomcat:v1

2）将4个容器分别划分到两个vlan100和vlan200中去。

#将test1，test2划分到vlan100中，vlan在mac地址后加@指定，此处mac地址省略。

pipework ovs0 test1 192.168.100.50/24 @100

pipework ovs0 test2 192.168.100.51/24 @100

#将test3,test4划分到vlan200中。

pipework ovs0 test3 192.168.100.52/24 @200

pipework ovs0 test4 192.168.100.53/24 @200

#经过上面配置后，同一vlan可以互相访问，不同vlan之间不能互访。

错误记录1

**问题描述：**

使用pipework ovs0时出现如下错误提示:

Need OVS installed on the system to create an ovs bridge

**解决方法：**

yum install openvswitch -y

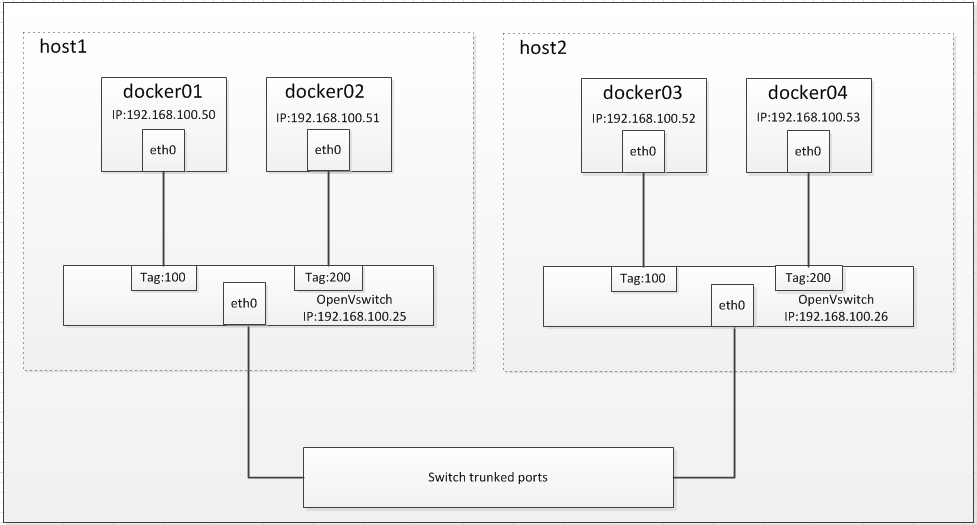
/etc/init.d/openvswitch start

chkconfig openvswitch on

6.1.1.3 docker多主机容器VLAN划分

上面介绍完了单主机上VLAN的隔离，下面我们将情况延伸到多主机的情况。有了前面两个例子做铺垫，这个也就不难了。为了实现这个目的，我们把宿主机上的网卡桥接到各自的OVS网桥上，然后再为容器配置IP和VLAN就可以了。

我们实验环境：主机A和B各有一块网卡eth0，IP地址分别为192.168.100.25/24、192.168.100.28/24。在主机A上创建两个容器test1、test2，分别在VLAN 100和VLAN 200上。在主机B上创建test3、test4，分别在VLAN 100和VLAN 200 上。最终，test1可以和test3通信，test2可以和test4通信,test1和test2不通，test3和test4不通。



1. 在docker01上配置

#🡸创建容器

docker run -itd --name test1 jame01/tomcat:v1

docker run -itd --name test2 jame01/tomcat:v1

#🡸划分vlan100和vlan200

pipework ovs0 test1 192.168.100.50/24 @100

pipework ovs0 test2 192.168.100.51/24 @200

#🡸将eth0桥接到ovs0上

ip addr add 192.168.100.25/24 dev ovs0; \

ip addr del 192.168.100.25/24 dev eth0; \

ovs-vsctl add-port ovs0 eth0; \

route add default gw 192.168.100.10 dev ovs0

# route del default; \

#🡸为防止配置错误导致远程连不上主机,配置前可以写好清理脚本加入crontab，配置无问题后再取消该脚本。

ip addr del 192.168.100.25/24 dev ovs0; \

ip addr add 192.168.100.25/24 dev eth0; \

ovs-vsctl del-port ovs0 eth0

route del default; \

route add default gw 192.168.100.10 dev ovs0

2）在docker02上配置

#🡸创建容器

docker run -itd --name test3 centos:6 /bin/bash

docker run -itd --name test4 centos:6 /bin/bash

#🡸划分VLAN

pipework ovs0 test3 192.168.100.52/24 @100

pipework ovs0 test4 192.168.100.53/24 @200

#🡸将eth0桥接到ovs0上

ip addr add 192.168.100.26/24 dev ovs0; \

ip addr del 192.168.100.26/24 dev eth0; \

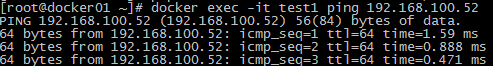
ovs-vsctl add-port ovs0 eth0; \

route add default gw 192.168.100.10 dev ovs0

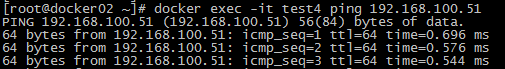
# route del default; \

3）联通测试

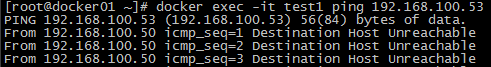
#🡸主机A容器test1上测试联通主机B容器test3



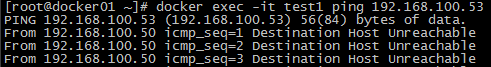
#🡸主机B容器test4上测试联通主机A容器test2



#🡸主机A容器test1测试联通主机B容器test4



#🡸主机B容器test3测试联通主机A容器test2



除此之外，pipework还支持使用macvlan设备、设置网卡MAC地址等功能。不过，pipework有一个缺陷，就是配置的容器在关掉重启后，之前的设置会丢失。

**总结：**

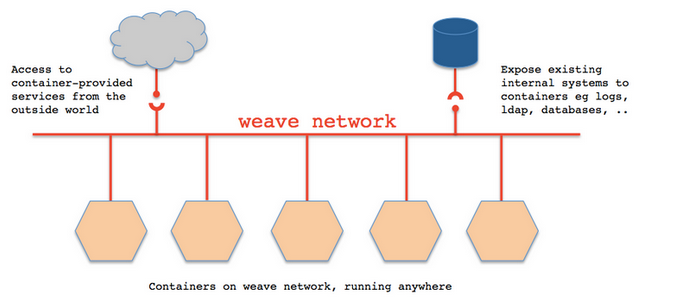
通过上面的介绍，对于一个基本应用而言，Docker的网络模型已经很不错了。然而，随着云计算和微服务的兴起，我们不能永远停留在使用基本应用的级别上，我们需要性能更好且更灵活的网络功能。pipework正好满足了我们这样的需求，从上面的样例中，我们可以看到pipework的方便之处。但是，同时也应注意到，pipework并不是一套解决方案，它只是一个网络配置工具，我们可以利用它提供的强大功能，帮助我们构建自己的解决方案。

6.2.2 docker网络之weave

weave是什么呢？简单来讲，weave创建了一个虚拟网络，用来连接部署在多台机器上的docker容器。

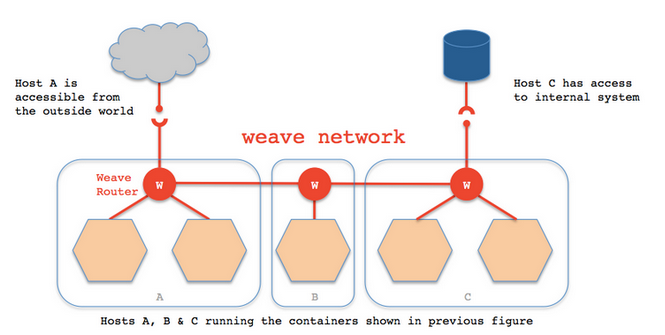
6.2.2.1 weave使用场景

**场景一：**



应用在使用该网络的时候就像所有的容器都在同一个交换机网络下一样，不需要配置端口映射、连接等等，容器中的应用提供的服务在weaver网络中可以被外部世界访问，不论你的容器运行在哪里。同样的，已经存在的系统应用也可以暴露给容器中的应用来调用，而不用担心内部应用运行的位置。

**场景二：**



weave可以穿透防火墙，流量是被加密的，允许主机连接通过一个不被信任的网络，使用weave你可以方便的部署多个容器在不同的地方运行

6.2.2.2 weave部署

1）安装weave

wget -O /usr/local/bin/weave https://github.com/zettio/weave/releases/download/latest\_release/weave

chmod a+x /usr/local/bin/weave

[root@docker01 ~]# weave version

weave script 1.0.1

weave router git-e50b1f1fe70e

Unable to find weaveworks/weavedns:1.0.1 image.

weave exec 1.0.1

[root@docker01 ~]# docker images|grep weave

weaveworks/weaveexec git-e50b1f1fe70e db0e5e7de40e 7 days ago 47.93 MB

weaveworks/weaveexec latest db0e5e7de40e 7 days ago 47.93 MB

weaveworks/weave git-e50b1f1fe70e becc20b0d76d 7 days ago 14.57 MB

weaveworks/weave latest becc20b0d76d 7 days ago 14.57 MB

weaveworks/weaveexec 1.0.1 8916b0188c16 3 weeks ago 47.22 MB

#提示：需要在每台主机上都安装weave,跨宿主机之间容器通过weave联通通讯。

1. 启动weave
2. 在主机docker01上运行

#启动 weave router，需要在每台主机上都启用这个服务。该服务所需的docker镜像weavetools会自动下载。可能出现被墙的问题导致不能自动下载，那么使用docker pull拉取镜像，之前docker配置了国内镜像站点。

weave launch

#运行一个容器

C1=$(weave run 192.168.30.10/24 -it --name web01 jame01/tomcat:v1)

1. 在主机docker02上运行

#启动weaverouter，192.168.100.25是docker01主机IP地址

weave launch 192.168.100.25

#运行一个容器

C2=$(weave run 192.168.30.11/24 -it --name web01 jame01/tomcat:v1)

6.3 docker网络之DNS配置

docker没有为每个容器专门定制镜像，那么怎么自定义配置容器的主机名和 DNS 配置呢？ 秘诀就是它利用虚拟文件来挂载到来容器的 3 个相关配置文件。

容器中使用 mount 命令可以看到挂载信息：

$ mount

...

/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...

/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...

tmpfs on /etc/resolv.conf type tmpfs ...

...

# 这种机制可以让宿主主机 DNS 信息发生更新后，所有 Docker容器的dns配置通过 /etc/resolv.conf 文件立刻得到更新。

如果用户想要手动指定容器的配置，可以利用下面的选项。

-h HOSTNAME or --hostname=HOSTNAME

#🡸设定容器的主机名，它会被写到容器内的 /etc/hostname 和 /etc/hosts。但它在容器外部看不到，既不会在 docker ps 中显示，也不会在其他的容器的 /etc/hosts 看到。

--link=CONTAINER\_NAME:ALIAS

#🡸选项会在创建容器的时候，添加一个其他容器的主机名到 /etc/hosts 文件中，让新容器的进程可以使用主机名 ALIAS 就可以连接它。

--dns=IP\_ADDRESS

#🡸添加 DNS 服务器到容器的 /etc/resolv.conf 中，让容器用这个服务器来解析所有不在 /etc/hosts 中的主机名。

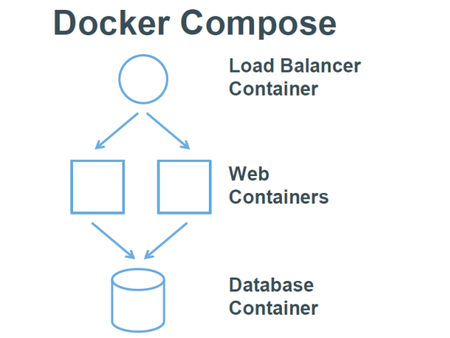
--dns-search=DOMAIN

#🡸设定容器的搜索域，当设定搜索域为 .example.com 时，在搜索一个名为 host 的主机时，DNS 不仅搜索host，还会搜索 host.example.com。 注意：如果没有上述最后 2 个选项，Docker 会默认用主机上的 /etc/resolv.conf 来配置容器。

第7章. docker compose

7.1 compose介绍

Docker Compose属于一个“应用层”的服务，用户可以定义哪个容器组运行哪些应用，它支持动态改变应用，并在需要时扩展。Docker Compose 是Docker官方编排（Orchestration）项目之一，负责快速在集群中部署分布式应用。使用Compose，你只需定义一个多容器应用的yml文件，然后使用一条命令即可部署运行所有容器。



docker compose部署：

wget https://github.com/docker/compose/releases/download/1.7.0/docker-compose-`uname -s`-`uname -m`

mv docker-compose-Linux-x86\_64 /usr/local/bin/docker-compose

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

# docker-compose --version

docker-compose version 1.7.0, build 0d7bf73

7.2 compose案例实战

下面通过docker-compose来组建一个Flask应用服务，即一个基于python的轻web应用。

（1）创建工作目录及准备文件

#创建目录

mkdir /dockerdata/composetest -p

cd /dockerdata/composetest

#编写app应用，通过下面代码可以看到应用里面有.route(‘/’),即访问根目录，然后返回一个从Redis里读取出来的值，该值通过自加来统计访问次数。最后web容器在0.0.0.0上监听默认端口5000并启动。

[root@host122 composetest]# cat app.py

#!/usr/bin/python

from flask import Flask

from redis import Redis

app = Flask(\_\_name\_\_)

redis = Redis(host='redis', port=6379)

@app.route('/')

def hello():

redis.incr('hits')

return 'Hello World! I have been seen %s times.' % redis.get('hits')

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", debug=True)

#准备文件

[root@host122 composetest]# cat requirements.txt

flask

redis

#编写Dockerfile，一个python容器，安装了依赖，然后将当前目录添加到容器工作目录/code里面。启动容器时会执行python app.py。

[root@host122 composetest]# cat Dockerfile

FROM python

ADD . /code

WORKDIR /code

RUN pip install -r requirements.txt

CMD python app.py

（2）通过yaml文件定义服务

[root@host122 composetest]# cat docker-compose.yaml

version: '2'

services:

web:

build: .

ports:

- "5000:5000"

volumes:

- .:/code

depends\_on:

- redis

redis:

image: redis

#说明：这个yaml文件是编排部署核心所在，在这个文件里面可以看到两个最高级别的key: web和redis，表示Compose定义了由两个“服务”组成的docker集群。

- web服务：该容器从当前文件夹的Dockerfile创建，并运行python app.py命令；将web容器内部的5000端口映射到host的5000端口；挂载当前文件夹到容器内部的/code文件夹，并将web容器与redis容器连接；

- redis服务：该容器直接由官方的redis镜像创建，Dockerfile不必另外重写。

#使用docker-compose up表示在创建容器后在前端启动

[root@host122 composetest]# docker-compose up

Building web

Step 1 : FROM python

---> 67b5207e5d3b

Step 2 : ADD . /code

---> f21cbd5f1e03

Removing intermediate container 306b2c5a20e7

Step 3 : WORKDIR /code

……

Creating composetest\_redis\_1

Creating composetest\_web\_1

Attaching to composetest\_redis\_1, composetest\_web\_1

……

web\_1 | \* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

web\_1 | \* Restarting with stat

web\_1 | \* Debugger is active!

web\_1 | \* Debugger pin code: 326-801-679

#使用docker-compose up -d表示创建容器后在后台启动

[root@host122 composetest]# docker-compose up -d

Starting composetest\_redis\_1

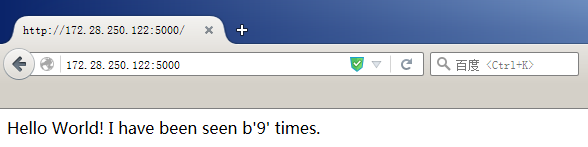
Starting composetest\_web\_1

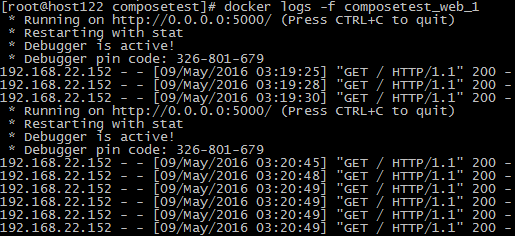
#访问测试

[root@host122 composetest]# docker ps -a|grep composetest

6ea68fdba9b0 composetest\_web "python app.py" 3 minutes ago Up 23 seconds 0.0.0.0:5000->5000/tcp composetest\_web\_1

fd65a55a34d8 redis "docker-entrypoint.sh" 3 minutes ago Up 23 seconds 6379/tcp composetest\_redis\_1





附录一 资料参考

1. docker官方手册：

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

1. dokcer中国镜像站点：

<https://dashboard.daocloud.io/>

1. fig测试参考：<http://www.infoq.com/cn/articles/docker-build-development-environment-based-on-fig>
2. Docker编排工具Fig介绍：

<http://dockone.io/article/119>

1. 《第一本docker书》的docker仓库：

<https://github.com/jamtur01/>

1. 利用Docker构建开发环境：

<http://tech.uc.cn/?p=2726>

1. 持续交付系列（一）使用Docker、Mesos实现持续交付：

<http://dockone.io/article/239>

1. 构建基于 Docker + Jenkins + Sahi 的 Web UI 自动化测试环境：<http://www.ibm.com/developerworks/cn/opensource/os-cn-JenkinsDockerSahi/index.html>
2. Docker网络原则入门

<http://dockone.io/article/455>

1. Docker DNS：

<http://dockerpool.com/static/books/docker_practice/advanced_network/dns.html>

1. Docker网络详解及Libnetwork前瞻：

<http://dockone.io/article/402>

1. 多台物理主机之间的容器互联:

<http://dockerpool.com/static/books/docker_practice/cases/container_connect.html>

1. Docker网络详解及pipework源码解读与实践：

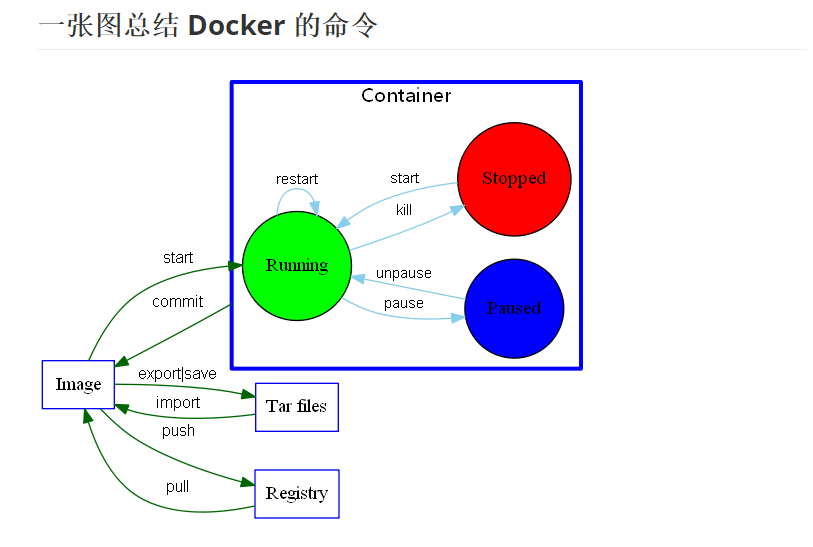
<http://www.infoq.com/cn/articles/docker-network-and-pipework-open-source-explanation-practice>

1. Mesos的体系结构和工作流：

<http://www.infoq.com/cn/articles/analyse-mesos-part-02>

1. Mesos官方文档：<http://mesos.apache.org/documentation/latest/index.html>

附录二 docker命令



* **docker save**

用于持久化镜像（不是容器）。

#将镜像centos:6、tomcat-7.0.54保存到指定目录下。

docker save centos:6 > ~/centos6.tar

docker save tomcat-7.0.54 > ~/tomcat-7.0.54.tar

#在其他主机导入指定镜像

docker load < ./centos6.tar

docker load < ./tomcat-7.0.54.tar

* **docker export**

用于持久化容器（不是镜像）。

#获取容器ID，并导出容器

# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6aba0ec4054c tomcat-7.0.54 "/bin/sh -c 'sh /app About an hour ago Up About an hour 0.0.0.0:9091->8080/tcp web02

e7105af461f0 apache-2.2.27 "/bin/sh -c '/applic About an hour ago Up About an hour 22/tcp, 0.0.0.0:9090->80/tcp web01

# docker export e7105af461f0 > ~/web01.tar

#在其他主机导入指定镜像

# cat web01.tar | docker import - apache-2.2.27

导出后再导入(exported-imported)的镜像会丢失所有的历史，而保存后再加载（saveed-loaded）的镜像没有丢失历史和层(layer)。这意味着使用导出后再导入的方式，你将无法回滚到之前的层(layer)，同时，使用保存后再加载的方式持久化整个镜像，就可以做到层回滚（可以执行docker tag <LAYER ID> <IMAGE NAME>来回滚之前的层）。

* **docker stats**

docker stats可以直接显示容器磁盘的I/O情况。

[root@host122 etc]# docker stats nginx

CONTAINER CPU % MEM USAGE/LIMIT MEM % NET I/O

nginx 0.00% 265.6 MB/33.61 GB 0.79% 49.62 MB/1.365 MB

CONTAINER CPU % MEM USAGE/LIMIT MEM % NET I/O

nginx 0.00% 265.6 MB/33.61 GB 0.79% 49.62 MB/1.365 MB

研发中心--杨鸿

2016-05-20