Contents

[tutorial 1](#_Toc454548652)

[Install Docker 1](#_Toc454548653)

[Image -> container 2](#_Toc454548654)

[Build your own image 2](#_Toc454548655)

[Tag & push your image to DockerHub 2](#_Toc454548656)

[自动构建 3](#_Toc454548657)

[<第一本Docker书> 3](#_Toc454548658)

[Docker命令 3](#_Toc454548659)

[使用Docker镜像和仓库 5](#_Toc454548660)

[构建镜像 = Dockerfile + docker build 5](#_Toc454548661)

[Dockerfile指令 7](#_Toc454548662)

[范例1：使用Docker测试静态网站 8](#_Toc454548663)

[范例2：构建Redis镜像和容器 9](#_Toc454548664)

[Docker容器互连 10](#_Toc454548665)

[使用Docker构建服务 11](#_Toc454548666)

[<第一本Docker书>修订版 12](#_Toc454548667)

[范例1: 使用Docker构建并测试Web应用程序 12](#_Toc454548668)

[范例2：使用Docker构建一个Java应用服务 15](#_Toc454548669)

[范例3：多容器的应用栈 16](#_Toc454548670)

[Docker Compose --容器编排 18](#_Toc454548671)

[实战 20](#_Toc454548672)

[镜像Jdk8 20](#_Toc454548673)

[Spring Boot with Docker 20](#_Toc454548674)

Docker

A container is a running instance of an image

# tutorial

## Install Docker

install virtualBox and Ubuntu

refer to: <https://docs.docker.com/linux/step_one/>

ctrl + alt + t打开终端

$ curl -fsSL https://get.docker.com/ | sh

若出现is Docker Daemon running? 将qzlin加入docker用户组 and Require reboot Ubuntu

## Image -> container

$docker run hello-world

若无法connection

$sudo gedit /etc/default/docker

export http\_proxy=”<http://HOST:PORT/>”

或者命令行

export http\_proxy=http://165.225.96.34:10015/

export https\_proxy=https://165.225.96.34:10015/

$sudo service docker restart //restart the docker service

docker: 告诉操作系统使用docker程序

run：docker的子命令创建与运行Docker容器

hello-world: 告诉Docker将hello-world 镜像装入到容器里

## Build your own image

1. Write a Dockerfile

FROM docker/whalesay:latest

RUN apt-get -y update && apt-get install -y fortunes

CMD /usr/games/fortune -a | cowsay

1. Build an image from your Dockerfile

$docker build – t docker-whale .

（Note: 若无法connection，改为

$docker build - -build-arg http\_proxy= <http://165.225.96.34:10015/> -t docker-whale .

1. Run your image

$docker run docker-whale

## Tag & push your image to DockerHub

$ docker tag imageID qizhonglin/docker-whale:latest

$ docker login - - username=qizhonglin - - [email=qizhong.lin@philips.com](mailto:email=qizhong.lin@philips.com)

$ docker push qizhonglin/docker-whale

## 自动构建

只需要将GitHub or BitBucket中含有Dockerfile文件的仓库连接到Docker Hub即可。向这个代码仓库推送代码时，将会触发一次镜像构建活动并创建一个新镜像。

Docker Hub -> Add Repository -> Automated Build -> GitHub -> Public and Private -> select projects

不能通过docker push命令推送一个自动构建，只能通过更新你的GitHub or BigBucket仓库来更新你的自动构建。

# <第一本Docker书>

## Docker命令

$docker info

该命令返回所有容器和镜像的数量，Docker使用的执行驱动和存储驱动以及Docker的基本配置

$docker run –it ubuntu /bin/bash

该命令提供Docker容器的创建到启动的功能。

- i 标志保证容器中STDIN是开启的

- t 标志告诉Docker为要创建的容器分配一个伪tty终端

ubuntu镜像告诉Docker基于ubuntu镜像来创建容器，该容器拥有自己的网络、IP地址、以及一个用来和宿主机进行通信的桥接网络接口

/bin/bash告诉Docker在新容器中运行的命令，在这里指启动一个bash shell

[root@de28...:/#](mailto:root@de28...:/) apt-get updata && apt-get install vim

若无法连接网络，需要设置

export http\_proxy=http://165.225.96.34:10015/

export https\_proxy=https://165.225.96.34:10015/

$docker ps –a

查看当前系统中容器的列表

$docker run - - name bob\_the\_container - i - t ubuntu /bin/bash

--name: 指定容器名称，可以用容器的名称来替代容器ID,容器的命名必须是唯一的。

$docker start bob\_the\_container

启动容器

$docker attach bob\_the\_container

重新附着到该容器的会话上

$docker rmi - f imageID

or $docker rmi - f bob\_the\_container

除了交互式运行的容器(interactive container),我们也可以创建长期运行的容器（守护式容器daemonized container）没有交互式会话，非常适合运行应用程序和服务

$docker run - - name daemon\_dave -d ubuntu /bin/sh –c “while true; do echo hello world; sleep 1; done”

-d标识告诉Docker将容器入到后台运行

$docker ps

查看正在运行的容器

$docker logs –ft daemon\_dave

获取容器的日志

-f监控Docker的日志，这与tail –f 命令非常相信

$docker top daemon\_dave

查看容器内部运行的进程

$docker exec –d daemon\_dave touch /etc/new\_config\_file

-d: 标志表明运行一个后台进程

daemon\_dave: 指定的容器

touch /etc/new\_config\_file: 执行的命令

$docker exec -it daemon\_dave /bin/bash

$docker run - -restart=always –name daemon\_dave –d Ubuntu /bin/sh –c “while true;…”

restart=always标志表明无论容器的退出代码是什么，Docker都会自动重启该容器

restart=on-failure标志表明只有当容器的退出代码为非0值的时候，才会自动重启

$docker inspect daemon\_dave

该命令对容器进行详细的检查，然后返回其配置信息，包括名称、命令、网络配置以及很多有用的数据

$docker rm daemon\_dave

删除容器

$docker rm `docker ps –a –q`

删除所有容器

docker ps命令会列出现有的全部容器

-a标志代表列出所有容器包括运行和停止

-q标志则表示只需要返回容器的ID而不会返回容器的其他信息

## 使用Docker镜像和仓库

Docker镜像是由文件系统叠加而成，最底端是一个引导文件系统bootfs,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 or Ubuntu。

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，可以依次类推，直至镜像栈的最底部，最底部的镜像称为基础镜像。最后，当从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。我们想在Docker中运行的程序就是在这个读写层中执行（见page: 50）

当Docker第一次启动一个容器时，初始的读写层是空的。当文件系统发生变化时，这些变化都会应用到这一层上。比如如果想修改一个文件，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。

这种机制称为写时复制copy on write.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。这个读写层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。

$docker images

列出镜像列表，本地镜像都保存在Docker宿主机的/var/lib/docker目录下。可以在/var/lib/docker/container目录下面看到所有的容器

远程镜像，保存在仓库Registry，即Docker Hub

$docker search puppet

查找所有Docker Hub上公共的可用镜像

$docker pull 用户名/镜像名:tag

拉取顶层仓库或用户仓库中的所有内容，用户仓库镜像是由Docker用户创建，顶层仓库是由Docker公司和选定的厂商创建

$docker run –it --name next\_container Ubuntu /bin/bash

docker run命令从镜像启动一个容器时，如果该镜像不在本地，Docker会先从Docker Hub下载该镜像。如果没有指定具体的镜像标签，那么Docker会自动下载latest标签的镜像。

## 构建镜像 = Dockerfile + docker build

一般基于一个已有的基础镜像，如ubuntu,构建新镜像

Dockerfile使用基本的基于DSL语法的指令来构建一个Docker镜像

目录结构

static\_web

Dockerfile

Static\_web目录就是我们的构建环镜build environment, Docker则称此环境为上下文context or 构建上下文.Docker会在构建镜像时将构建上下文和该上下文中的文件和目录上传到Docker守护进程。这样Docker守护进程就能直接访问你想在镜像中存储的任何代码、文件或者其他数据

Dockerfile

# Version: 0.0.1

FROM ubuntu:14.04

MAINTAINER QiZhong Lin [qizhong.lin@philips.com](mailto:qizhong.lin@philips.com)

RUN apt-get update

RUN apt-get install –y nginx

RUN echo ‘Hi, I am in your container’ > /usr/share/nginx/html/index.html

EXPOSE 80

#开头的行都会被认为是注释

Docker中的指令会按顺序从上而下执行，每条指令都会创建一个新的镜像层并对镜像进行提交。Docker大体上按照如下流程执行Dockerfile中的指令

* Docker从基础镜像运行一个容器
* 执行一条指令，对容器做出修改
* 执行类似docker commit的操作，提交一个新的镜像层
* Docker再基于刚提交的镜像运行一个新容器
* 执行Dockerfile中的下一条指令，直至所有指令都执行完毕

如果你的Dockerfile由于某些原因如某条指令失败了没有正常结束，那么你将得到一个可以使用的镜像。这对调试非常有帮助

FROM指令：指定一个已经存在的镜像，后续指令都将基于该镜像进行

MAINTAINER指令：告诉Docker该镜像的作者是谁，以及作者的电子邮件地址。

RUN指令：在当前镜像中运行指定的命令。默认情况下，RUN指令会在shell里使用命令包装器/bin/sh –c来执行。

EXPOSE指令：告诉Docker该容器内的应用程序将会使用容器的指定端口。这并不意味着可以自动访问任意容器运行中服务的端口。出于安全考虑，Docker并不会自动打开该端口，而是需要你在使用docker run运行容器时来指定需要打开哪些端口。

$docker build –t=”qizhonglin/static\_web:v0” .

Dockerfile中的所有指令都会被执行并且提交，并且在该命令成功结束后返回一个新镜像

-t选项为新镜像设置了仓库和名称

最后的.告诉Docker到本地目录中去找Dockerfile文件。也可以指定一个Git仓库的源地址来指定Dockerfile的位置。

如果在构建上下文的根目录下存在以.dockerignore命名的文件的话，那么该文件内容会被按行进行分割，每一行都是一条文件过滤匹配模式。非常像.gitignore文件。该文件用来设置哪些文件不会被上传到构建上下文中去。

Docker会将之前构建时创建的镜像当做缓存并作为新的开始点。

$docker history qizhonglin/static\_web:v0

查看镜像是如何构建出来

$docker run –d –p 80 --name static\_web qizhonglin/static\_web:v0 nginx –g “daemon off;”

从新镜像启动容器

-d选项告诉Docker以分离detached的方式在后台运行。这种方式非常适合运行类似nginx守护进程这样的需要长时间运行的过程。

nginx –g “daemon off;” 容器里运行的命令，表明将以前台运行的方式启动nginx作来web服务器

-p标志用来控制Docker在运行时应该公开哪些网络端口给外部（宿主机）

$docker ps -l

查看容器中的80端口被映射到宿主机的随机端口号

$docker run –d –p 8080:80 --name static\_web qizhonglin/static\_web:v0 nginx –g “daemon off;”

将容器80端口映射到宿主机的指定端口8080

$curl localhost:映射端口号

### Dockerfile指令

RUN指令：指定镜像被构建时要运行的命令

CMD指令：指定容器被启动时要运行的命令

举例：

$docker run –it qizhonglin/static\_web:v0 /bin/true

等价于

CMD [“/bin/true”]

docker run命令可以覆盖CMD指令。如果我们在Dockerfile里指定了CMD指令，而同时在docker run命令行中也指定了要运行的命令，命令行中指定的命令会覆盖Dockerfile中的CMD指令。

在Dockerfile中只能指定一条CMD指令。如果指定了多条CMD指令，也只最后一条CMD指令会被使用。

ENTRYPOINT指令：与CMD指令类似，但不容易在启动容器时被覆盖。实际上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。

WORKDIR指令：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

ENV指令：用来在镜像构建过程中设置环境变量

USER指令：用来指定该镜像会以什么样的用户去运行

VOLUMN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者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者对数据进行持久化的功能。

* 卷可以在容器间共享和重用
* 一个容器可以不是必须和其他容器共享卷
* 对卷的修改是立时生效的
* 对卷的修改不会对更新镜像产生影响
* 卷会一直存在直至没有任何容器再使用它

卷功能让我们可以将数据（源代码）、数据库或者其他内容添加到镜像中而不是将这些内容提交到镜像中，并且允许我们在多个容器间共享这些内容。我们可以利用此功能来测试容器和内容的应用程序代码，管理日志，或者处理容器内部的数据库

ADD指令：将构建环境下的文件和目录 复制 到镜像中

COPY指令：构建环境下的文件和目录 复制 到镜像中，但不会做文件提取和解压工作

注意：复制时，文件源路径必须是一个与当前构建环境相对的文件或者目录，本地文件都放到和Dockerfile同一个目录下，不能复制该目录之外的任何文件。

ONBUILD指令：为镜像添加触发器。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 。触发器会在构建过程中插入新指令，我们可以认为这些指令是紧跟在FROM之后指定的。触发器可以是任何构建指令

$docker rmi qizhonglin/static\_web:v0

删除一个镜像

该操作只会删除本地的镜像，如果想删除Docker Hub上的镜像仓库，DockerHub -> Delete repository

$docker rmi `docker images –a –q`

运行自己的Docker Registry，见page90

### 范例1：使用Docker测试静态网站

将Docker作为本地Web开发环境，这个环境可以完全重现生产环境，保证开发环境和部署环境一致

sample

Dockerfile

nginx

global.conf

nginx.conf

website

index.html

$docker build –t qizhonglin/nginx .

$docker run –dp 80 –name website –v $PWD/website:/var/www/html/website qizhonglin/nginx nginx

-v这个选项允许我们将宿主机的目录$PWD/website作卷，挂载到容器里/var/www/html/website,而容器会将该目录/var/www/html/website作为nginx服务器的根目录root，从而挂载了正在开发的本地网站

卷是在一个或者多个容器内被选定的目录，可以绕过分层的联合文件系统，为Docker提供持久数据或者共享数据。这意味着对卷的修改会直接生效，并绕过镜像。当提交或者创建镜像时，卷不被包含在镜像里

$docker ps –l

查看正在运行的容器，名为website的容器正处于活跃状态，其80端口被映射到本地…端口port

$curl localhost:port

访问容器里运行的服务

修改本地网站内容

$curl localhost:port

刷新镜像容器里挂载该网站，自动更新

### 范例2：构建Redis镜像和容器

Dockerfile

FROM ubuntu

RUN apt-get update

RUN apt-get -y install redis-server redis-tools

EXPOSE 6379

ENTRYPOINT ["/usr/bin/redis-server"]

CMD []

$docker build –t qizhonglin/redis .

$docker run –d –p 6379 –name redis qizhonglin/redis

$docker port redis 6379

查看容器redis的端口6379映射到host机的端口是多少(假定是49161)？

Host机安装Redis客户端，连接容器redis,验证Redis服务器是否工作

$apt-get –y install redis-tools

$redis-cli –h 127.0.0.1 –p 49161

Redis 127.0.0.1:49161>

Docker容器公开端口 -> 本地网络

Docker内部网络

在安装Docker时，会创建一个新的网络接口，名字是docker0.每个容器都会在这个接口上分配一个IP地址，如

$ip a show docker0

接口docker0是一个虚拟的以太网桥，用于连接容器和本地宿主网络

Docker每创建一个容器就会创建一组互联的网络接口。这组接口就像管道的两端。这组接口其中一 端作为容器的eth0接口，而另一端统一命令为类似vethec6a这种名字，作为宿主机的一个端口。

容器默认是无法访问的，从宿主网络与容器通信时，必须明确指定打开的端口

$docker inspect redis

查看网关地址是docker0, 容器的IP是”IPAddress”, 端口是”Ports”, 具体见page109

$redis-cli –h 172.17.0.18 IPAddress

$redis-cli –h localhost –p 49161

### Docker容器互连

$docker run –d --name redis qizhonglin/redis 启动redis容器

$docker run –p 4567 --name webapp --link redis:db –it –v $PWD/webapp:/opt/webapp qizhonglin/Sinatra /bin/bash

启动webapp容器，对外暴露4567端口，内连redis容器

--link标志创建了两个容器间的父子连接，连接的容器名字:连接后容器的别名

通过把容器连接在一起，可以让父容器直接访问任意子容器的公开端口（比如，父容器webapp可以连接到子容器redis的6379端口）。容器化的应用程序限制了可被攻击的界面，减少了公开暴露的网络

被连接的容器必须运行在同一个Docker宿主机上，不同Docker宿主机上运行的容器无法连接

Docker在父容器中的以下两个地方写入了连接信息

/etc/hosts文件中

包含连接信息的环境变量中

$docker run –p 4567 --name webapp ---link redis:db …

[root@...:/#](mailto:root@...:/) env

HOSTNAME=…

DB\_NAME=…

DB\_PORT=…

Docker在连接webapp and redis容器时，自动创建了这些以DB开头的环境变量，这些连接信息可以让容器内的应用程序使用相同的方法与别的容器进行连接，而不用关心被连接的容器的具体细节

可以给Sinatra应用程序加入一些连接信息，以便与Redis通信。

require ‘uri’

…

uri = URI.parse(ENV[‘DB\_PORT’])

redis = Redis.new(:host => uri.host, :port -> uri.port)

…

## 使用Docker构建服务

Jekyll镜像：容器存放通过卷挂载的网站源代码

Apache镜像：容器利用包含编译后的网站的卷，并为其服务

Jekyll

Dockerfile

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install ruby ruby-dev make nodejs

RUN gem install --no-rdoc --no-ri jekyll

VOLUMN /data/

VOLUMN /var/www/html

WORKDIR /data

ENTRYPOINT ["jekyll", "build", "--destination=/var/www/html"]

/data/：用来存放网站的源代码

/var/www/html: 用来存放编译后的jekyll网站码

ENTRYPOINT: 自动构建，将工作目录/data/中的所有jekyll网站代码构建到/var/www/html/目录中

Apache镜像

apache

Dockerfile

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install apache2

VOLUMN ["/var/www/html"]

WORKDIR /var/www/html

EXPOSE 80

ENTRYPOINT ["/usr/sbin/apache2"]

CMD ["-D", "FOREGRAOUND"]

使用VOLUMN指令创建一个卷/var/www/html，用来存放编译后的jekyll网站，然后将/var/www/html设为工作目录

使用ENTRYPOINT and CMD指令组合在容器启动时默认运行Apache

$docker build –t qizhonglin/Jekyll .

$docker build –t qizhonglin/apache .

$docker run –v /home/qizhonglin/blog:/data/ --name blog qizhonglin/Jekyll

把本地的/home/qizhonglin/blog目录作为/data/卷挂载到容器里。容器已经拿到网站的源代码，并将其构建到已编译的网站，存放到/var/www/html/目录

卷是在一个或多个容器中特殊指定的目录，卷会绕过联合文件系统，为持久化数据和数据提供几个有用的特性。

* 卷可以在容器间共享和重用
* 共享卷时不一定要运行相应的容器
* 对卷的修改会直接在卷上反映出来
* 更新镜像时不会包含对卷的修改
* 卷会一直存在，直至没有容器使用它们

利用卷，可以在不用提交镜像修改的情况下，向镜像里加入数据（如源代码、数据或者其他内容），并且可以在容器间共享这些数据

$docker run –d –P –volumns-from blog qizhonglin/apache

--volumns-from标志把指定容器里的所有卷都加入新创建的容器里。这意味着Apache容器可以访问之前创建的blog容器里/var/www/html卷中存放的编译后的jekyll网站。即使blog容器没有运行，apache容器也可以访问这个卷。

$docker port containerID 80

假定为49160

localhost:49160

# <第一本Docker书>修订版

## 范例1: 使用Docker构建并测试Web应用程序

sinatra

Dockerfile

webapp

bin

webapp

lib

app.rb

webapp\_redis

bin

webapp

lib

app.rb

redis

Dockerfile

Dockerfile

FROM Ubuntu:14.04

RUN apt-get update –yqq && apt-get –yqq iinstall ruby ruby-dev build-essential redis-tools

RUN gem install --no-rdoc --no-ri Sinatra json redis

RUN mkdir –p /opt/webapp

EXPOSE 4567

CMD [“/opt/webapp/bin/webapp” ]

$docker build –t qizhonglin/sinatra .

启动服务

$docker run –d –p 4567 --name webapp –v $PWD/webapp:/opt/webapp qizhonglin/sinatra

将本地的webapp目录挂载到容器里创建的目录/opt/webapp

容器启动时，执行/opt/webapp/bin/webapp这个文件

$docker port webapp 4567 (假定映射端口是49160)

测试

$curl -i –H ‘Accept:application/json’ –d ‘name=Foo&status=Bar’ <http://localhost:49160/json>

若web应用程序使用Redis后端数据库，则在应用程序文件app.rb里创建到Redis连接redis=Redis.new(:host=>’db’, :port=>’6379’)

构建启动Redis容器

Redis

Dockerfile

FROM ubuntu:14.04

RUN apt-get –yqq update

RUN apt-get –yqq install redis-server redis-tools

EXPOSE 6379

ENTRYPOINT [ “/usr/bin/redis-server” ]

CMD []

$docker build –t qizhonglin/redis .

$docker run –d –p 6379 --name redis qizhonglin/redis

$docker port redis 6379 （假定映射端口是49161）

在本地安装redis客户端，访问redis容器这个服务端

$apt-get –y install redis-tools

$redis-cli –h 127.0.0.1 –p 49161

绑定sinatra容器与redis容器

从Docker1.9版本开始，可以使用Docker Networking以及docker network命令

Docker Networking允许用户创建自己的网络，容器可以通过这个网络互相通信。

* Docker Networking可以将容器连接到不同宿主机上的容器
* 通过Docker Networking连接的容器可以在无需更新连接的情况下，对停止、启动或者重启容器。
* 使用Docker Networking，不必事先创建容器再去连接它。同样，也不必关心容器的运行顺序

创建网络

$docker network create app

ddocker network命令创建了一个桥接网络，命名为app.

在Docker网络app中启动Redis容器

$docker run -d --net=app --name db qizhonglin/redis

--net标志指定了新容器db将会在网络app中运行

在Docker网络app中启动sinatra容器

$docker run –p 4567 --net=app --name webapp\_redis –it –v $PWD/webapp\_redis:/opt/webapp qizhonglin/sinatra

在同一个Docker网络app启动的容器，会将它们的IP地址信息自动更新在/etc/hosts文件中。通过主机名和端口可以互相访问。比如webapp访问数据库redis,可以通过db:6379

redis = Redis.new(:host=> ‘db’, :port=> ‘6379’)

$docker port webapp 4567 （假定映射端口为49161）

测试sinatra应用程序

测试redis容器数据

见page124

$docker network ls

命令列出当前系统中的所有网络

$docker network rm app

删除一个Docker网络

$docker network inspect app

查看网络更详细的信息

$docker net connect app db2

将正在运行的容器db2加入docker网络app中

$docker network disconnect app db2

容器db2断开docker网络app

## 范例2：使用Docker构建一个Java应用服务

fetcher

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install wget

VOLUME [ “/var/lib/tomcat7/webapps/” ]

WORKDIR /var/lib/tomcat7/webapps/

ENTRYPOINT [ “wget”]

CMD [ “-?” ]

使用wget从指定的URL获取文件，并把文件保存在/var/lib/tomcat7/webapps/目录。（之前把这个卷共享给Tomcat服务器并且运行里面的内容

若没有指定URL, 指令ENTRYPOINT and CMD会返回wget帮助

$docker build –t qizhonglin/fetcher .

$docker run –I –t --name sample\_war qizhonglin/fetcher <https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war>

下载sample.war使存于共享卷中

tomcat7

Dockerfile

FROM ubuntu:14.04

RUN apt-get -yqq update

RUN apt-get -yqq install tomcat7 default-jdk

ENV CATALINA\_HOME /usr/share/tomcat7

ENV CATALINA\_BASE /var/lib/tomcat7

ENV CATALINA\_PID /var/run/tomcat7.pid

ENV CATALINA\_SH /usr/share/tomcat7/bin/catalina.sh

ENV CATALINA\_TMPDIR /tmp/tomcat7-tomcat7-tmp

RUN mkdir –p $CATALINA\_TMPDIR

VOLUME [ “/var/lib/tomcat7/webapps/” ]

EXPOSE 8080

ENTRYPOINT [ “/usr/share/tomcat7/bin/catalina.sh”, “run” ]

安装jdk and tomcat

创建应用程序的共享卷

启动tomcat

$docker build –t qizhonglin/tomcat7 .

$docker run --name sample\_app --volumes-from sample\_war –d –P qizhonglin/tomcat7

使容器sample\_app与sample\_war共享卷，（即存储在/var/lib/tomcat7/webapps/卷里的WAR文件会从sample容器挂载到sample\_app容器，最终被Tomcat加载并执行。

$docker port sample\_app 8080 (假定映射端口是49154)

localhost:49154/sample

## 范例3：多容器的应用栈

Express + redis

Node容器 + 主Redis/2从Redis + 日志容器

nodejs

nodeapp

package.json

server.js

Dockerfile

redis\_base

Dockerfile

redis\_primary

Dockerfile

redis\_replica

Dockerfile

logstash

Dockerfile

主从Redis服务

redis\_base

FROM ubuntu:14.04

RUN apt-get -yqq update && apt-get -yqq install redis-server redis-tools

VOLUME [ "/var/lib/redis", "/var/log/redis/" ]

EXPOSE 6379

CMD []

$docker build -t qizhonglin/redis\_base .

redis\_primary

FROM qizhonglin/redis\_base

ENTRYPOINT [ "redis-server", "--logfile /var/log/redis/redis-server.log" ]

$docker build –t qizhonglin/redis\_primary

redis\_replica

FROM qizhonglin/redis\_base

ENTRYPOINT [ "redis-server", "--logfile /var/log/redis/redis-replica.log", "--slaveof redisprimary 6379" ]

$docker build –t qizhonglin/redis\_replica

启动Redis主从服务器

由镜像qizhonglin/redis\_primary and qizhonglin/redis\_replica运行主从redis服务

$docker network create express

$docker run -d -h redisprimary --net express --name redisprimary qizhonglin/redis\_primary

$docker run -d -h redispreplica1 --net express --name redisreplica1 qizhonglin/redis\_replica

$docker run -d -h redispreplica2 --net express --name redisreplica2 qizhonglin/redis\_replica

查看redis容器的日志

$docker run -it --rm --volumes-from redisprimary ubuntu cat /var/log/redis/redis-server.log

启动应用程序

$docker build -t qizhonglin/nodejs .

$docker run -d --net express -p 3000:3000 --name nodeapp qizhonglin/nodejs

host机测试

localhost:3000

同一网络express内的虚机可以互通

$docker inspect redisprimary （假定IP地址: 172.18.0.2）

$docker exec -it nodeapp bash

[root@...:/opt/nodeapp#](mailto:root@...:/opt/nodeapp) ping 172.18.0.2

虚拟机open tcp 3000,从而可以在window甚至外网连接

捕获应用日志

在生产环境里需要确保可以捕获日志并将日志保存到日志服务器。一般使用Logstash来完成

Dockerfile

FROM Ubuntu:14.04

RUN apt-get –yqq install logstash

ADD logstash.conf /etc/

WORKDIR /opt/logstash

ENTRYPOINT [ “bin/logstash” ]

CMD [ “--config=/etc/logstash.conf” ]

Logstash配置文件logstash.conf

input { file { type => "syslog" path => ["/var/log/nodeapp/nodeapp.log", "/var/log/redis/redis-server.log"] }}output { stdout { codec => rubydebug }}

input: Logstash会监视这两个文件，将其中新的内容发送给Logstash.

output: 接受所有Logstash输入的内容将其输出到标准输出上。现实中，一般会将Logstash配置到Elasticsearch集群或者其他的目的。

$docker build –t qizhonglin/logstash .

$docker run -d --name logstash --volumes-from redisprimary --volumes-from nodeapp qizhonglin/logstash

分别挂载了redisprimary and nodeapp容器的卷，这样就可以访问Redis and Node的日志文件了，任何加到这些日志文件里的内容都会反映在logstash容器的卷里，即input.file.path, logstash容器启动时由命令bin/logstash --config=/etc/logstash.conf 启动Logstash并加载/etc/logstash.conf配置文件

$docker logs –f logstash

查看logstash容器的日志（因此Logstash配置文件将输出写至stdout）

## Docker Compose --容器编排

使用YAML文件定义一组要启动的容器以及容器运行时的属性

Install

$sudo apt-get –y install python-pip

$sudo pip --proxy <http://165.225.96.34:10015> install docker-compose

多容器应用程序栈

范例1：应用容器(python) + Redis容器

domposeapp

Dockerfile

app.py

requirement.txt

docker-compose.yml

requirement.txt

flask

redis

Dockerfile

FROM python:2.7

ADD . /composeapp

WORKDIR /composeapp

RUN pip --proxy http://165.225.96.34:10015 install –r requirements.txt

app.py

见page189, python web服务，连上redis数据库

docker-compose.yml

web:

image: qizhonglin/composeapp

command: python app.py

ports:

- "5000:5000"

volumes:

- .:/composeapp

links:

- redis

redis:

image: redis

web：服务

image: 镜像 (or build: Dockerfile)

ports: 主机->容器端口映射

volumes: 主机->容器卷映射

links: 要连接到该服务的其他服务

以上服务等效于docker run命令

$docker run -d -p 5000:5000 -v .:/composeapp --link redis:redis qizhonglin/composeapp python app.py

$docker run -d -p 6379:6379 redis

$docker-compose up -d

Compose会启动这些容器，使用指定的参数来执行，并将所有的日志输出合并到一起

Compose创建了composeapp\_redis\_1 and composeapp\_web\_1这两个新的服务。服务取名规则：为了保证服务是唯一的，compose将docker-compose.yml文件中指定的服务名字加上了目录名作为前缀，并分别使用数字作为后缀。

Compose接管了每个服务输出的日志，输出的日志每一行都使用缩短的服务名字作为前缀，并交替输出在一起

$docker-compose ps

列出本地docker-compose.yml文件里定义的正在运行的所有服务

$docker-compose logs

显示Docker Compose服务的日志

$docker-compose stop

$docker-compose start

$docker-compose kill

$docker-compose rm

# 实战

## 镜像Jdk8

jdk8-image-ex

Dockerfile

docker-compose.yml

Main.java

**Dockerfile**

FROM frolvlad/alpine-oraclejdk8:slim

WORKDIR /mnt

ADD Main.java .

ENTRYPOINT [ "sh" ]

CMD [ "-c", "javac Main.java && java Main" ]

**docker-compose.yml**

jdk8-ex:

build: .

jdk8-image-ex$ docker-compose up

## Spring Boot with Docker

Refer:

<https://spring.io/guides/gs/spring-boot-docker/>

<https://hub.docker.com/r/frolvlad/alpine-oraclejdk8/>

Docker is a Linux container management toolkit with a "social" aspect, allowing users to publish container images and consume those published by others. A Docker image is a recipe for running a containerized process

gs-spring-boot-docker

src

main

java

com.philips.bootdocker

Application.java

docker

build.gradle

build.gradle

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.5.RELEASE")

**classpath("se.transmode.gradle:gradle-docker:1.2")**

}

}

**group = "qizhonglin" //docker hub 帐号**

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'idea'

apply plugin: 'spring-boot'

**apply plugin: 'docker'**

jar {

baseName = 'gs-spring-boot-docker'

version = '0.1.0'

}

repositories {

mavenCentral()

}

sourceCompatibility = 1.8

targetCompatibility = 1.8

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

task wrapper(type: Wrapper) {

gradleVersion = '2.3'

}

// docker任务

task buildDocker(type: Docker, dependsOn: build) {

push = true

applicationName = jar.baseName

dockerfile = file('src/main/docker/Dockerfile')

doFirst {

copy {

from jar

into stageDir

}

}

}

Application.java

@SpringBootApplication

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello Docker World";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Dockerfile

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD **gs-spring-boot-docker-0.1.0.jar** app.jar

RUN sh -c "touch /app.jar"

ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar" ]

run application, the result **gs-spring-boot-docker-0.1.0.jar will be put to /tmp**

gs-spring-boot-docker$gradle build buildDocker

$docker run –p 8080:8080 –t qizhonglin/gs-spring-boot-docker

$curl localhost:8080

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

Docker Engine v1.12

* Network containers

Networks are natural ways to isolate containers from other containers or other networks.

default network

$ docker network ls

By default, Docker provides two network drivers for you, the bridge and the overlay drivers. A bridge network is limited to a single host running Docker Engine. An overlay network can include multiple hosts

$ docker run -itd --name=networktest ubuntu

$ docker network inspect bridge

Unless you tell it otherwise, Docker always launches your containers in network bridge

Create your own bridge network

$ docker network create -d bridge my-bridge-network

You could have left this flag "-d bridge" off as bridge is the default value for this flag

$ docker network inspect my-bridge-network

$ docker run -d --net=my-bridge-network --name db training/postgres

Add containers to a network

$ docker inspect --format='{{json .NetworkSettings.Networks}}' db

$ docker run -d --net=my-bridge-network --name web training/webapp python app.py

$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web

get the IP address of your web, assume ip is 172.17.0.2

$ docker exec -it db bash

root@a205f0dd33b2:/# ping 172.17.0.2

* Manage data in containers

Manage data inside and between your Docker containers

A data volume is a specially-designated directory within one or more containers that bypasses the Union File System.Data volumes are designed to persist data, independent of the container’s life cycle.

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

create a new volume inside a container at /webapp.

$ docker inspect web

locate the volume on the host

Mount a host directory as a data volume

$ docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py

If the path /opt/webapp already exists inside the container’s image, the /src/webapp mount overlays but does not remove the pre-existing content. Once the mount is removed, the content is accessible again.

Docker volumes default to mount in read-write mode, but you can also set it to be mounted read-only.

$ docker run -d -P --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py

Note: The host directory is, by its nature, host-dependent. For this reason, you can’t mount a host directory from Dockerfile because built images should be portable

Mount a shared-storage volume as a data volume

$ docker create -v /dbdata --name dbstore training/postgres /bin/true

$ docker run -d --volumes-from dbstore --name db1 training/postgres

$ docker run -d --volumes-from db1 --name db3 training/postgres

Backup, restore, or migrate data volumes

$ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

uses tar to backup the contents of the dbdata volume to a backup.tar file inside our /backup directory, and then mounted a local host directory as /backup

Manage a Swarm [1.12 RC]

Docker1.12通过swarm实现了容器集群，在集群之上可以service命令创建一个服务，实现服务的多实例。在服务之上，通过stack可以描述一组服务，最终实现一个大系统的部署

Docker Engine 1.12 includes swarm mode for natively managing a cluster of Docker Engines called a Swarm. Use the Docker CLI to create a swarm, deploy application services to a swarm, and manage swarm behavior.

* Service discovery
* Load balancing
* Multi-host networking
* Scaling

orchestration and cluster management

The cluster management and orchestration features embedded in the Docker Engine

You enable swarm mode for the Engine by either initializing a swarm or joining an existing swarm.

To deploy your application to a swarm, you submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default manager nodes are also worker nodes, but you can configure managers to be manager-only nodes. The agent notifies the manager node of the current state of its assigned tasks so the manager can maintain the desired state.

replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes

global services, the swarm runs one task for the service on every available node

The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a PublishedPort;The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service

操作过程：

1. initializing a cluster of Docker Engines in swarm mode

$docker swarm init --listen-addr <MANAGER-IP>:<PORT> //ssh终端连上manager node

1. adding nodes to the swarm

$docker swarm join <MANAGER-IP>:<PORT> //ssh终端连上worker node

1. deploying application services to the swarm

//ssh终端连上manager node

$ docker service create --replicas 1 --name helloworld alpine ping docker.com

1. managing the swarm once you have everything running

//ssh终端连上manager node

see which nodes are running the service:

$ docker service tasks helloworld 可能是worker1, worker2,甚至是manager node

Scale the service in the swarm

$ docker service scale helloworld=5

see the updated task list

$ docker service tasks helloworld

drain a node that had a task assigned to it, 从而worker1 node的任务会被别的worker node接手

$docker node update --availability drain worker1

操作命令：

（1）docker swarm：集群管理，子命令有init, join, leave, update

（2）docker service：服务创建，子命令有create, inspect, update, remove, tasks

（3）docker node：节点管理，子命令有accept, promote, demote, inspect, update, tasks, ls, rm

（4）docker stack/deploy：试验特性，用于多应用部署 下面在一个三节点环境中，针对各个命令进行试用，详细展开说明

Bundle

Docker 1.12引入了一种新的文件格式，名为Distributed Application Bundle（即分布式应用包，目前尚处于实验阶段）。Bundle是一种立足于服务之上的新型抽象机制，主要面向全堆栈应用。

一个Docker Bundle文件属于一组服务的声明性规范，负责说明：

运行哪套具体镜像版本

创建怎样的网络

各服务中的容器如何联网并运行

Bundle文件具备全面的可移植性，且可通过软件交付通道实现部署，这是因为其允许大家对多容器Docker应用进行规范指定与版本控制。

容器编排

将容器从只能部署在单一主机之上，升级为能够将多种复杂的容器应用广泛部署于大量设备当中.要实现编排方案，我们需要一套独立于基础设施之外的分布式平台，贯穿整个应用生命周期始终，且能够在硬件发生故障或软件更新时继续正常起效。

$docker swarm init

这条命令会创建一个Raft节点协作组。作为首个节点，其负责实现管理功能，意味着它可以接收命令并调度任务。随着向Swarm中添加更多节点，后续节点将默认作为工作节点存在，分别执行由管理节点分派的任务。管理节点属于Raft协作组的组成部分。我们利用一套经过优化的Raft存储体系，其直接读取内存内容以提升调度工作的性能表现。

$docker service create –name frontend –replicas 5 -p 80:80/tcp nginx:latest

由5套Nginx容器构成的Swarm作为单一内部负载均衡型服务，且于Swarm内任意节点的端口80上进行交付

在创建服务时，我们可以选择创建复制型或者全局服务。复制型服务意味着我们定义的任意数量的容器都可扩散至全部可用主机。相比之下，全局服务则代表调度Swarm当中每台主机上同一容器的一个实例。

Docker如何实现弹性?

warm模式下的引擎拥有自组织与自修复特性，意味着它们能够识别我们定义的应用，并在出现差错时持续检查并修复环境。举例来说，如果大家关闭某台运行有Nginx实例的设备，则另一节点上会自动启动一套新的容器。

安全性

Docker 1.12提供经过严格认证的TLS、身份验证、授权与加密机制来保护Swarm中的各相关节点，且全部采取开箱即用的实现方式。



在建立首个管理节点时，Docker Engine会生成新的认证中心（简称CA）以及一组初始证书。在这一初始步骤完成后，加入该Swarm的每个节点都会自动被分配予一份新证书、随机生成的ID以及Swarm中的当前角色（管理节点或者工作节点）。这些证书将被作为其加密安全节点的身份且贯穿于整个生命周期，而管理节点则利用证书确保任务与更新以安全方式进行。

以自动化方式解决了TLS证书中最难搞定的部分：证书轮换。

从深层角度看，参与至Swarm内的各个节点都会持续不断地刷新自身证书，确保可能存在的泄露或者违规证书不会长久有效。用户可以对各证书的轮换频率进行设定，且最高可设置为每30分钟更换一次