Docker基础

1 单机

1.1 安装Docker

> 注：https://mirrors.aliyun.com 有Docker Engine Linux版和docker toolbox的下载链接。

Mac

1. 访问 https://download.docker.com/mac/stable/Docker.dmg 获得安装包
2. 双击Docker.dmg，把Docker.app拖到Applications文件夹
3. 运行Docker.app
4. 打开命令行窗口运行如下命令：

```
$ docker info
$ docker version
```

Windows

Windows 10 专业版或以上，或Windows 2016：

1. 访问 https://download.docker.com/win/stable/InstallDocker.msi 获得安装包
2. 运行InstallDocker.msi
3. 打开命令行窗口运行如下命令

```
PS C:\Users\Docker> docker --version
Docker version 17.03.0-ce, build 60ccb22
```

Windows 10 家庭版，Windows 8请安装Docker Toolbox

1. 访问https://download.docker.com/win/stable/DockerToolbox.exe 获得安装程序
2. 运行安装程序
3. 成功后启动 Docker Toolbox terminal，执行如下命令

```
$ docker version
```

> 注意：在Windows平台上，Docker for Windows支持LInux和Windows两种容器类型，需要把Docker改成支持Linux容器，在Docker图标点击右键，选择"switch to linux container".

1.2 镜像加速

在国内访问Docker Hub速度比较慢，可以在Docker引擎设置镜像加速器加速对DockerHub的访问。

1. Mac 和 Windows

在Docker的配置“Demon”中，在“Registry mirrors”中增加如下内容：

```
https://registry.docker-cn.com
```

2. Linux

更新 */etc/docker/daemon.json*，添加如下参数，并重启Docker引擎。

```
{
  "registry-mirrors": ["https://registry.docker-cn.com"]
}
```

验证加速器是否生效的方法是比对`docker pull nginx`速度是否比没有加速器的时候有明显提升。

3. Docker toolbox

利用Windows docker toolbox的用户，可以用如下方式查看并进入linux虚拟机：

```
> docker-machine ssh default
$ vi /etc/docker/daemon.json   # 输入上面的JSON内容
$ exit  # 退出虚拟机
> docker-machine ls
```

让后重启虚拟机即可

```
docker-machine stop default
docker-machine start default
```

1.3 Docker Help

了解Docker命令可以执行Docker命令查看

```
$ docker
$ docker run --help
```

1.4 运行容器

在命令行中启动容器是一项基本操作，在本练习中我们运行一个容器。

1. 运行hello-world镜像，查看输出

```
~ $ docker run hello-world
```

2. 在centos容器中执行命令

`-it`表示可以利用本命令行窗口和容器进行交互

```
~ $ docker run -it centos
```

在容器中执行ls和env可以看到如下输出

```
[root@76c23ff56d3d /]# ls
anaconda-post.log  dev  home  lib64       media  opt   root  sbin  sys  usr
bin                etc  lib   lost+found  mnt    proc  run   srv   tmp  var
[root@76c23ff56d3d /]# env
HOSTNAME=76c23ff56d3d
TERM=xterm
LS_COLORS=rs=0:di=01;...36:*.xspf=01;36:
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
_=/usr/bin/env
```

3. 退出容器

```
[root@76c23ff56d3d /]# exit
```

退出后容器也终止了。用docker ps命令会发现容器已经不在系统里运行了。

4. 停止的容器还在系统里，可以用`docker ps -a`查看到，删除容器用`dockcer rm`

```
$ docker ps -a
$ docker rm <容器id>
```

1.5 容器的启动和停止

在后台启动nginx，并把nginx 80端口映射到主机8080端口

```
~ $ docker run -d -p 8000:80 nginx
```

访问本机8000端口，会显示nginx欢迎界面。

尝试执行如下命令：

```
# 先把前面占用8000端口的容器停掉
~ $ docker stop <容器id>

# 容器命名 mynginx
~ $ docker run -d -p 8000:80 --name mynginx nginx

# 显示系统内所有运行中的容器
~$ docker ps

# 通过filter只显示mynginx的信息
~ $ docker ps --filter name=mynginx

# 动态显示容器的运行状况，ctrl-c结束
~ $ docker stats

# 如果只想看最新数据
~ $ docker stats --no-stream
CONTAINER     CPU %     MEM USAGE / LIMIT     MEM %        NET I/O      BLOCK I/O
   PIDS
2d3cb564ad94  0.00%     1.941MiB / 1.952GiB   0.10%        1.38kB / 0B  0B / 0B
   2

# 停止容器
~ $ docker stop mynginx

# 启动容器
~ $ docker start mynginx

# 删除容器
~ $ docker stop mynginx
~ $ docker rm mynginx
```

1.6 查看日志信息

1. 运行nginx容器，并命名为nginx1

```
~ $ docker run --name nginx1 -d -p "8080:80" nginx
```

2. 用浏览器访问http://localhost:8080 可以看到nginx的欢迎页面
3. 在命令行窗口看到nginx的日志输出

```
~ $ docker logs -f nginx1
```

1.7 数据卷（data volume）

1. 在当前目录创建index.html，并输入如下内容

```
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
    <title>Nginx Demo</title>
  </head>
  <body>
    <div class="container">
      <h1>欢迎</h1>
      <p>进入容器世界</p>
    </div>
  </body>
</html>
```

2. 在命令行中运行nginx命令，把当前目录作为数据卷挂载进入nginx容器

```
docker run -v $(pwd):/usr/share/nginx/html -d -p 8080:80 nginx

# 在Windows中用目录名做为参数传入，一不能有中文，二不能有空格，
docker run -v <目录名>:/usr/share/nginx/html -d -p 8080:80 nginx
```

NOTE: Windows没有(pwd)，所以

1. 访问http://localhost:8080 可以看到如下输出

```
欢迎
进入容器世界
```

如果页面显示为乱码，请把页面编码改为如下：

```
<meta charset="GB18030">
```

1.8 容器网络

1. 显示本机上的网络

```
$ ~ $ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
17f8c2eb7ec6        bridge              bridge              local
6dfa696dfdc6        host                host                local
85e8f287f892        none                null                local
```

2. 运行容器并查看网络信息

```
$ docker run -itd --name=mycentos centos

# 查看容器的网络信息
$ docker inspect mycentos

# 查看bridge网络上都有哪些容器
$ docker network inspect bridge

# 从bridge网络上把容器摘除
$ docker network disconnect bridge mycentos
```

3. 创建新的bridge网络

```
$ docker network create -d bridge mybridge

# 还没有容器挂到网络上
$ docker inspect mybridge

# 创建一个容器
$ docker run -itd --net=mybridge --name anothercentos centos
$ docker inspect anothercentos

# 查看网络
$ docker network inspect mybridge
```

1.9 创建镜像

我们可以尝试把上面的应用打包成为一个新的镜像。注意，在当前目录中要有前面练习里的index.html文件。

1. 在当前目录创建文件，名字为Dockerfile，输入如下内容

```
FROM nginx:latest
COPY index.html /usr/share/nginx/html
EXPOSE 80
CMD ["nginx","-g","daemon off;"]
```

2. 运行docker build命令生成镜像

```
$ docker build -t mynginx .
```

3. 查看新生成的镜像

```
basic $ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED
    SIZE
mynginx                     latest              3d2a916cdafc        4 minutes ago
    109MB
```

4. 启动容器

```
docker run -p -d 8080:80 mynginx
```

5. 用浏览器访问 http://localhost:8080 可以看到如下输出

```
欢迎
进入容器世界
```

1.10 上传镜像仓库

本实验中我们把镜像上传到阿里云镜像服务，操作步骤和原理与和其他镜像仓库类似。

1. 在阿里云 https://www.aliyun.com/ 上注册一个账号
2. 开通容器服务
3. 进入容器镜像服务管理控制台 https://cr.console.aliyun.com
4. 点击"创建镜像仓库"，请选择所在地域，Namespace，仓库名“mynginx"，代码源部分请选择“本地仓库”
5. 点击“创建镜像仓库“
6. 进入新创建的镜像仓库详情页面的操作指南部分详细描述了如何从命令行登录，如何推送和拉取镜像。在本例中假定镜像仓库的地址为 registry.xxx.com/name/mynginx
7. 在本地的命令行窗口中运行如下命令

```
docker login --username=name registry.cn-hangzhou.aliyuncs.com
```

8. 查看本机上镜像列表

```
REPOSITORY                                TAG      IMAGE ID      CREATED       SIZE
mynginx                                   latest   3d2a916cdafc  17 hours ago  109MB
registry.xxx.aliyuncs.com/name/mynginx    latest   3d2a916cdafc  17 hours ago  109MB
```

9. 上传到镜像仓库

```
docker tag <镜像ID> registry.xxx.aliyuncs.com/name/mynginx
docker push registry.xxx.aliyuncs.com/name/mynginx
```

10. 把本地的mynginx镜像删除，或者登录到另外一台计算机上执行如下命令

```
docker run -v $(pwd):/usr/share/nginx/html -d -p 8080:80
registry.xxx.aliyuncs.com/name/mynginx
```

1.11 Docker Compose

在前面的练习中都是用*docker run*命令启动容器，在运行多个容器，并且每个容器都有很多参数需要配置的时候，用*docker run*就会很繁琐。本练习中我们利用*docker-compose*命令来启动多个容器组成的应用。

1. 创建新目录作为工作目录，名为*wordpress*
2. 进入工作目录创建*docker-compose.yml*文件，内容如下：

```
version: '3'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
    db_data:
```

1. 执行docker-compose命令启动

```
$ docker-compose up -d
```

2. 查看已经运行中的容器信息

```
basic $ docker-compose ps
      Name                     Command               State          Ports
-------------------------------------------------------------------------------------
basic_db_1          docker-entrypoint.sh mysqld      Up      3306/tcp
basic_wordpress_1   docker-entrypoint.sh apach ...   Up      0.0.0.0:8000->80/tcp
```

3. 进入浏览器访问http://localhost:8000，可以看到wordpress应用界面。
4. 停掉容器

```
basic $ docker-compose stop
Stopping basic_wordpress_1 ... done
Stopping basic_db_1 ... done
```

5. 删除容器

```
basic $ docker-compose rm
Going to remove basic_wordpress_1, basic_db_1
Are you sure? [yN] y
Removing basic_wordpress_1 ... done
Removing basic_db_1 ... done
```

1.12 清理空间

运行Docker时间长了会有不用的文件占用空间，可以利用*docker system prune* 来清理空间。

1. 查看空间占用情况

```
$ docker system df
```

2. 清理空间

```
$ docker system prune
WARNING! This will remove:
    - all stopped containers
    - all volumes not used by at least one container
    - all networks not used by at least one container
    - all dangling images
Are you sure you want to continue? [y/N] y
```

1.13 Docker multi-stage-builds 构建镜像

用Docker来编译程序很方便，但是编译的中间结果也会进入最终生成的镜像里，这些编译结果会大大增加最终生成的镜像的大小。Docker 17.05以后引入了multi stage builds的功能可以解决这个问题。

1. 创建工作目录
2. 在当前目录下创建HelloWorldExample.java文件

```
public class HelloWorldExample{
  public static void main(String args[]){
    /*
    Use System.out.println() to print on console.
    */
    System.out.println("Hello World !");

  }
}
```

3. 在当前目录下创建Dockerfile

```
FROM java:8 as builder
WORKDIR /app
ADD HelloWorldExample.java /app
RUN javac *.java

FROM java:8-jre-alpine
WORKDIR /app
COPY --from=builder /app/*.class /app
ENTRYPOINT ["java","-cp",".","HelloWorldExample"]
```

4. 编译并打包镜像

```
$ docker build -t javahello .
$ docker run javahello
Hello World !

# 查看生成的镜像
$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED
     SIZE
javahello                   latest              1d5f43537642        4 seconds ago
     108MB
<none>                      <none>              cdcc3cbfe753        54 seconds ago
      643MB
<none>                      <none>              706511eaf86f        About a minute
ago   643MB
```

可以看到编译的镜像为600M+，最终运行的镜像为100M+。

注：如果你的Docker版本还没有到17.05，可以用2个Dockerfile加上一点脚本可以打到同样效果。

2 Swarm Mode集群

2.1 创建集群

1. 利用*docker-machine*创建3节点

```
$ docker-machine create --engine-registry-mirror=https://registry.docker-cn.com -d
virtualbox manager1
$ docker-machine create --engine-registry-mirror=https://registry.docker-cn.com -d
virtualbox worker1
$ docker-machine create --engine-registry-mirror=https://registry.docker-cn.com -d
virtualbox worker2
```

Windows 10 Docker利用了hyperv虚拟机机制，用*Administrator*运行，命令如下：

```
Administrator> docker-machine.exe create -d hyperv manager1
Adminstrator> docker-machine.exe create -d hyperv worker1
Adminstrator> docker-machine.exe create -d hyperv worker2
```

创建成功后查看新生成的节点

```
$ docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                         SWARM
DOCKER        ERRORS
manager1   -        virtualbox   Running   tcp://192.168.99.100:2376
v17.05.0-ce
worker1    -        virtualbox   Running   tcp://192.168.99.101:2376
v17.05.0-ce
worker2    -        virtualbox   Running   tcp://192.168.99.102:2376
v17.05.0-ce
```

2. *ssh*登录进入*manager1*

```
$ docker-machine ssh manager1
```

3. 在*manager1*中执行命令，创建swarm mode集群，并把该节点作为管理节点

假定*manager1*的IP地址为*192.168.99.100*

```
docker@manager1:~$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (nad0af3rxn7a1n78nfndoph8v) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-17f...wp60g9 \
    192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

可以看到把一个worker节点加入集群的命令。

另外，如果你也可以在命令中指定网卡来创建集群，类似如下操作：

```
$ docker swarm init --advertsie-addr eth0
```

4. 登录进入woker1 和 worker2，执行如下命令

```
docker swarm join \
--token SWMTKN-1-17f...wp60g9 \
192.168.99.100:2377
```

5. 查看集群节点状态

```
$ docker-machine ssh manager1 docker node ls
ID                            HOSTNAME        STATUS      AVAILABILITY
MANAGER STATUS
am85tf4suub4wr1pk7g9ek4q1     worker2         Ready       Active
nad0af3rxn7a1n78nfndoph8v *   manager1        Ready       Active
Leader
r4dpqnvzjidkmuc0t8ud5kkfb     worker1         Ready       Active
```

从这里看出来，这个集群有3个节点，*manager1*为管理节点，*worker1*和*worker2*为工作节点。三个节点都处于正常状态。

6. 查看节点加入命令

执行如下命令查看manager节点加入的命令：

```
docker swarm join-token manager
```

如下命令显示如何将一个节点以worker身份加入：

```
docker swarm join-token worker
```

对比两个命令的输出，可以看到manager和worker加入命令的区别在于token是不同的。

2.2 连接远程Docker daemon

在上面的例子里，我们都是ssh进入节点完成Docker命令操作的。现在我们尝试一下不登陆进入节点，用本机的Docker管理远程节点的Docker daemon。

1. 获取*manager1*的环境信息

```
$ docker-machine env manager1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/libin/.docker/machine/machines/manager1"
export DOCKER_MACHINE_NAME="manager1"
# Run this command to configure your shell:
# eval $(docker-machine env manager1)
```

2. 连接manager1所在docker daemon

```
$ eval $(docker-machine env manager1)
```

执行完这个命令后，docker client连接的就是manager1的daemon，你可以运行*docker info*查看。

```
xamples $ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
...
Swarm: active
 NodeID: nad0af3rxn7a1n78nfndoph8v
 Is Manager: true
 ClusterID: x1a6g51ayl8wzsrq5g8iubwdq
 Managers: 1
 Nodes: 3
 Orchestration:
  Task History Retention Limit: 5
...
Live Restore Enabled: false
```

对比不在swarm mode集群里的节点info输出，可以看到*Swarm*的值为*active*。

3. 查看集群节点状态

```
$ docker-machine ssh manager1 docker node ls
ID                           HOSTNAME        STATUS      AVAILABILITY
MANAGER STATUS
am85tf4suub4wr1pk7g9ek4q1    worker2         Ready       Active
nad0af3rxn7a1n78nfndoph8v *   manager1        Ready       Active
Leader
r4dpqnvzjidkmuc0t8ud5kkfb    worker1         Ready       Active
```

可以看到和登陆进入*manager1*里执行的输出一样。

2.2 集群管理命令

执行*docker*命令不加参数，可以看到其中集群管理命令集合：

```
Management Commands:
  container   Manage containers
  image       Manage images
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  volume      Manage volumes
```

我们随后的练习都是围绕着这些命令来进行的。另外，除非特殊说明，下面的docker命令都是访问*manager1*的docker engine来完成。

2.3 部署应用

在swarm mode集群中部署应用可以通过compose v3版本的描述文件完成。

1. 在工作目录创建*wordpress.yml*文件，内容如下：

```
version: '3'
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:4.7.5
    deploy:
      replicas: 2
      update_config:
        parallelism: 1
        delay: 30s
      restart_policy:
        condition: on-failure
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
    db_data:
```

> 注意，在生产环境中还要配置wordpress的共享存储。

把wordpress.yml拷贝到*manager1*上。

```
$ docker-machine scp wordpress.yml manager1:/home/docker
```

2. 部署应用

docker-machine ssh进入manager1

```
$ docker stack deploy -c wordpress.yml wp
```

3. 查看stack状态

```
docker@manager1:~$ docker stack ls
NAME    SERVICES
wp      2
```

可以看到*wpapp*有2个服务

4. 查看服务状态

```
docker@manager1:~$ docker service ls
$ docker service ls
ID            NAME          MODE        REPLICAS  IMAGE             PORTS
ts07mnyjktf7 wp_db         replicated  1/1       mysql:5.7
z6emm0y5gtqe wp_wordpress  replicated  2/2       wordpress:4.7.5   *:8000->80/tcp
```

这个过程会拉取*wordpress*和*mysql*镜像，等待两个服务的实例全部启动。

5. 访问应用

用浏览器访问*manager1*的*8000*端口，*manager1*的IP地址可以用如下命令获得。

```
$ docker-machine ip manager1
```

在浏览器中可以看到wordpress应用已经正常启动了。

6. 查看wordpress的2个实例分别运行在哪个节点上

```
$ docker service ps wp_wordpress
ID             NAME             IMAGE            NODE     DESIRED STATE  CURRENT
STATE  ...
6xxh2gzmxurh   wp_wordpress.1   wordpress:4.7.5  worker1  Running        Running
...
i5zex9nz3ak0   wp_wordpress.2   wordpress:4.7.5  worker2  Running        Running
...
```

从这个输出看，wordpress的两个实例分别运行在*worker1*和*worker2*上。

2.3 更新应用

这个练习把wordpress服务的镜像从4.7.5更新为4.8.0。

```
$ docker service update --image wordpress:4.8.0 wp_wordpress
```

在更新期间访问wordpress应用，看更新是否造成服务中断。

在wordpress服务的部署描述中有如下内容：

```
    deploy:
      replicas: 2
      update_config:
        parallelism: 1
        delay: 30s
```

wordpress服务部署为2个实例，更新时一次一个。

2.4 弹性伸缩

1. 把wordpress服务的实例数目增加为4个

```
$ docker service scale wp_wordpress=4
```

2. 查看服务状态

```
$ docker service ls
$ docker service ps wp_wordpress
```

2.5 Routing Mesh机制

前面的练习都是访问*manager1*的8000端口，尝试访问一下*worker1*和*worker2*的8000端口。

2.6 Secrets

1. 创建secret

```
$ echo "Password4DB" | docker secret create db_password -
$ docker service create --name checksecret --secret db_password redis:alpine
```

2. 登陆进入*checksecret*容器所在节点，执行如下命令

```
$ docker exec -ti $(docker ps --filter name=checksecret -q) cat
/run/secrets/db_password
Password4DB
```

2.7 编排模版secrets

1. 用编排模版部署nginx应用，使用secrets，部署文件*nginxsecrets.yml*内容

```
version: "3.1"
services:
  nginx:
    image: 'nginx:latest'
    ports:
      - '80'
    secrets:
      - mysecrets
secrets:
  mysecrets:
    external: true
```

2. 创建secretes并部署

```
$ echo "credentials" | docker secret create mysecrets -
$ docker stack deploy -c nginxsecrets.yml nginxsecrets
```

3. 进入nginxsecrets容器所在主机执行如下命令获得容器id

```
$ docker exec -ti $(docker ps --filter name="nginxsecrets" -q) cat
/run/secrets/mysecrets
credentials
```

可以看到secrets内容动态加载到容器内存中，并可以通过*/run/secrets/mysecrets*来得到内容。

2.7 创建Overlay网络

1. 创建overlay网络，命名为*testoverlay*

```
$ docker network create -d overlay testoverlay
```

2. 查看网络，确保*scope*为*swarm*

```
docker network ls  | grep testoverlay
ckq6bszc2dbu        testoverlay         overlay             swarm
```

3. 创建两个服务连接到*testoverlay*

```
$ docker service create --name app1 --network testoverlay nginx
$ docker service create --name app2 --network testoverlay nginx
```

4. 创建一个服务不在*testoverlay*网络上

```
$ docker service create --name app-not-in-testoverlay nginx
```

4. 登陆到*app1*所在主机，检查*app1*是否可以访问*app2*

```
$ docker exec -ti $(docker ps --filter name="app1" -q) ping app2
PING app2 (10.0.2.4): 56 data bytes
64 bytes from 10.0.2.4: icmp_seq=0 ttl=64 time=0.031 ms
```

5. 登陆到*app-not-in-testoverlay*所在主机，检查容器*app-not-in-testoverlay*是否可以访问*app2*

```
$ $ docker exec -ti $(docker ps --filter name="app-not-in-testoverlay" -q) ping
app2
ping: unknown host
```

说明不在同一网络的容器之间是不互通的

2.8 删除应用或者服务

```
$ docker stack rm wp
$ docker service rm app1
```