### Linux容器

Linux容器是对进程进行隔离，对于容器里面的进程来说，它接触到的各种资源都是虚拟的，从而实现与底层系统的隔离。

优点：

1. 启动快。容器里面的应用，直接就是底层系统的一个进程
2. 资源占用少
3. 体积小

虚拟机是包含一个完成的操作系统的，资源占用多，冗余步骤多（要先启动系统），启动慢等缺点。

### Docker

docker 是服务器----客户端架构, 命令行运行docker命令的时候，需要本机有 Docker 服务

|  |
| --- |
| # service 命令的用法  $ sudo service docker start  # systemctl 命令的用法  $ sudo systemctl start docker |

### image文件

Docker 把应用程序及其依赖，打包在 image 文件里面。

只有通过这个文件，才能生成 Docker 容器。

image 文件可以看作是容器的模板。

Docker 根据 image 文件生成容器的实例。

同一个 image 文件，可以生成多个同时运行的容器实例。

|  |
| --- |
| docker image ls //列出本机上的镜像  docker image rm [imageName] //删除 image 文件  docker image pull [imageName] //将 image 文件从仓库抓取到本地 |

### container

|  |
| --- |
| // 从 image 文件，生成一个正在运行的容器实例。  docker container run [imageName] |

docker container run 会自动抓取image文件，若发现本地没有image文件，会从仓库自动抓取，因此前面的docker image pull 并不是必需的。

|  |
| --- |
| // 手动终止容器  docker container kill [containID] |

容器文件：image 文件生成的容器实例，本身就是一个文件；

一旦容器生成，就会同时存在两个文件，image文件和容器文件，且关闭容器并不会删除容器文件，只是容器停止运行了而已。

|  |
| --- |
| # 列出本机正在运行的容器  $ docker container ls  # 列出本机所有容器，包括终止运行的容器  $ docker container ls --all  # 删除容器文件（kill之后，容器文件还在）  docker container rm [containerID] |

### Dockerfile

|  |
| --- |
| # 该 image 文件继承官方的 node image，冒号表示标签，这里标签是8.4，即8.4版本的 node。  FROM node:8.4  # 将当前目录下的所有文件（除了.dockerignore排除的路径），都拷贝进入 image 文件的/app目录。  COPY . /app  # 指定接下来的工作路径为/app。  WORKDIR /app  # 在/app目录下，运行npm install命令安装依赖。注意，安装后所有的依赖，都将打包进入 image 文件。  RUN npm install --registry=https://registry.npm.taobao.org  # 将容器 3000 端口暴露出来， 允许外部连接这个端口。  EXPOSE 3000 |

有了dockerfile文件后，构建image文件：

|  |
| --- |
| docker image build -t [imageName[:标签]] [dockerfile文件路径] |

-t：指定image文件名

例如：docker image build -t koa-demo:0.0.1 .

然后docker image ls查看新生成的image

### 生成新容器

docker container run 从image文件生成容器，每运行都会生成一个新容器文件

|  |
| --- |
| docker container run -p 8000:3000 -it koa-demo /bin/bash  -p参数：容器的 3000 端口映射到本机的 8000 端口。  -it参数：容器的 Shell 映射到当前的 Shell，然后你在本机窗口输入的命令，就会传入容器。  koa-demo:0.0.1：image 文件的名字（如果有标签，还需要提供标签，默认是 latest 标签）。  /bin/bash：容器启动以后，内部第一个执行的命令。这里是启动 Bash，保证用户可以使用 Shell。 |

因为容器中进程接触到的文件系统和网络接口都是虚拟的，与本机的文件系统和网络接口是隔离的，因此需要定义容器与物理机的端口映射（map）。

使用docker container kill [containerId] 终止容器运行

docker container rm [containerId] 删除容器

亦可：docker container run **--rm** -p 8000:3000 -it koa-demo /bin/bash 在启动时候，传参--rm，在程序终止后自动删除容器文件。

### CMD命令

在Dockerfile里添加CMD命令，可以在容器启动后自动执行一些命令

|  |
| --- |
| FROM node:8.4  COPY . /app  WORKDIR /app  RUN npm install --registry=https://registry.npm.taobao.org  EXPOSE 3000  CMD node demos/02.js |

RUN命令在 image 文件的构建阶段执行，执行结果都会打包进入 image 文件；

CMD命令则是在容器启动后执行;

一个 Dockerfile 可以包含**多个**RUN命令，但是只能有**一个**CMD命令。

**TIP**: 若添加了CMD命令，docker container run命令就不能附加命令了（比如前面的/bin/bash），否则它会**覆盖CMD命令**；此时只能这样启动：

|  |
| --- |
| docker container run -p 8000:3000 -it koa-demo |

### 其他命令

**docker container start [containerId]**：启动已经存在的容器（run 每次执行会生成新的容器）

**docker container stop [containerId]:** 向容器里面的主进程发出 SIGTERM 信号，然后过一段时间再发出 SIGKILL 信号。

不同于docker container kill , kill是直接向容器发送SIGKILL信号，强行终止；而stop是先发送SIGTERM信号，容器可以自行进行收尾清理工作，之后再发送SIGKILL信号终止。

**docker container logs [containerId]:** 查看 docker 容器的输出，即容器里面 Shell 的标准输出（docker run命令运行容器的时候，没有使用-it参数，就要用这个命令查看输出）

**docker container exec -it [containerID] /bin/bash：** 进入一个正在运行的 docker 容器。如果docker run命令运行容器的时候，没有使用-it参数，就要用这个命令进入容器。一旦进入了容器，就可以在容器的 Shell 执行命令了

**docker container cp [containerID]:[filePath] [toPath]** : 从正在运行的 Docker 容器里面，将文件拷贝到本机；

filePath是容器中文件的绝对路径，toPath是要复制到的本机文件路径

### 实战A ：自建wordpress容器

#### apache-php

|  |
| --- |
| docker container run \  -p 80:80  --rm \  --name wordpress \  --volume "$PWD/":/var/www/html \  php:5.6-apache |

基于php的image文件新建容器，并且运行该容器；php标签是5.6-apache，代表php版本是5.6，且自带apache服务器；

|  |
| --- |
| -p ：本机端口和容器端口映射  --rm：停止运行后，自动删除容器文件。  --name wordpress：容器的名字叫做wordpress。  --**volume** "$PWD/":/var/www/html：将当前目录（$PWD）映射到容器的/var/www/html（Apache 对外访问的默认目录）。因此，当前目录的任何修改，都会反映到容器里面，进而被外部访问到。 |

此时其实容器内服务器已经启动，只是/var/www/html里没有东西，所有访问localhost时没有反应；

可以在本地当前目录下创建index.php 里写入

<?php

phpinfo();

?>

即可在localhost时看到页面。

#### wordpress安装包

下载wordpress-4.9.4安装包，解压放到docker-demo/wordpress下，会映射到容器的/var/www/html里，此时访问localhost/wordpress会到wordpress提示安装页面；

#### mysql

|  |
| --- |
| docker container run \  -d \  --rm \  --name wordpressdb \  --env MYSQL\_ROOT\_PASSWORD=123456 \  --env MYSQL\_DATABASE=wordpress \  mysql:5.7 |

基于mysql 5.7新建一个容器

|  |
| --- |
| -d：容器启动后，在**后台**运行。  --rm：容器终止运行后，自动删除容器文件。  --name wordpressdb：容器的名字叫做wordpressdb  --env MYSQL\_ROOT\_PASSWORD=123456：向容器进程传入一个环境变量MYSQL\_ROOT\_PASSWORD，该变量会被用作 MySQL 的根密码。  --env MYSQL\_DATABASE=wordpress：向容器进程传入一个环境变量MYSQL\_DATABASE，容器里面的 MySQL 会根据该变量创建一个同名数据库（本例是wordPress）。 |

此时已经有两个容器运行成功了：wordpress和wordpressdb；

wordpressdb是后台运行的，前台看不到输出，使用查看输出：

|  |
| --- |
| docker container logs wordpressdb |

#### 定制php-apache容器

由于官方的php容器没有mysqli扩展，我们必须自己新建image文件

新建一个Dockerfile文件：

|  |
| --- |
| FROM php:5.6-apache  RUN docker-php-ext-install mysqli  CMD apache2-foreground |

在原来 PHP 的 image 基础上，安装mysqli的扩展。然后，启动 Apache；

|  |
| --- |
| docker image build -t phpwithmysql **.** |

#### 启动新容器并连接mysql

现在基于phpwithmysql 重新新建个容器

|  |
| --- |
| docker container run \  -p 80:80  --rm \  --name wordpress \  --volume "$PWD/":/var/www/html \  **--link wordpressdb:mysql \**  phpwithmysql |

--link wordpressdb:mysql : 表示将容器连接到wordpressdb容器，冒号表示**别名**

修改wordpress权限为777

此时访问localhost/wordpress就可以开始安装了！！！

### 实战B ：Wordpress 官方镜像

#### 新建并启动mysql容器

|  |
| --- |
| docker container run \  -d \  --rm \  --name wordpressdb \  --env MYSQL\_ROOT\_PASSWORD=123456 \  --env MYSQL\_DATABASE=wordpress \  mysql:5.7 |

#### 基于官方wordpress image，新建并启动容器

|  |
| --- |
| docker container run \  -d \  -p 80:80 \  --rm \  --name wordpress \  --env WORDPRESS\_DB\_PASSWORD=123456 \  --link wordpressdb:mysql \  --volume "$PWD/wordpress":/var/www/html \  wordpress |

|  |
| --- |
| -p 80:80：将容器的 80 端口映射到本地80端口  --volume "$PWD/wordpress":/var/www/html：将容器的/var/www/html目录映射到当前目录的wordpress子目录，可以在本地修改容器里的文件  WORDPRESS\_DB\_PASSWORD：MySQL 容器的根密码 |

此时访问localhost/wordpress即可正常访问！！ 亦可在本地当前目录下wordpress下更改容器内的文件。

### 实战C: Docker compose 工具

实战B已经够简单了，但还是要启动两个容器，且要在启动时，在命令行提供两个容器的连接信息

#### docker-compose

管理多个 Docker 容器组成一个应用。需要定义一个 YAML 格式的配置文件docker-compose.yml，写好多个容器之间的调用关系；

|  |
| --- |
| # 启动所有服务  $ docker-compose up  # 关闭所有服务  $ docker-compose stop |

以上两个命令必须在docker-compose.yml 的当前目录执行；

#### wordpress实例

|  |
| --- |
| mysql:  image: mysql:5.7  environment:  - MYSQL\_ROOT\_PASSWORD=123456  - MYSQL\_DATABASE=wordpress  web:  image: wordpress  links:  - mysql  environment:  - WORDPRESS\_DB\_PASSWORD=123456  ports:  - "80:80"  working\_dir: /var/www/html  volumes:  - wordpress:/var/www/html |

两个顶层标签表示有两个容器mysql和web；

启动两个容器：

|  |
| --- |
| docker-compose up |

访问localhost即可进入wordpress安装页面

关闭两个容器：

|  |
| --- |
| docker-compose stop |

容器文件还在，下次启动还可复用，要删除容器：

|  |
| --- |
| docker-compose rm |

### 启动

* 1. 阿里云镜像加速

容器镜像服务->镜像加速器

* 1. 运行交互式容器

docker run -d -p 80:80 httpd

* 1. 启动容器

创建一个以进程方式运行的容器

docker run -d ubuntu:15.10 /bin/sh -c "while true; do echo hello world; sleep 1; done"

-d:让容器在后台运行

返回容器id

docker ps 查看正在运行的容器（会自动分配容器名称NAMES）

参数： -a 查看所有容器

-q : 静默模式，只显示容器编号

停用所有运行中的容器：docker stop $(docker ps -q)

删除所有容器：docker rm $(docker ps -aq)

1. 命令停用并删除所有容器：

docker stop $(docker ps -q) & docker rm $(docker ps -aq)

或：

**docker rm -f `docker ps -aq`** // -f参数，可以强制删除正在运行的docker容器

docker logs -f 容器id/容器名称NAMES 查看容器内的标准输出(类似tail -f 输出)

docker stop 容器id/容器名称NAMES 停止容器

docker start 容器id/容器名称NAMES 启动容器

docker restart 容器id/容器名称NAMES 重启容器

docker port 容器id/容器名称NAMES 查看端口映射情况

docker top 容器id/容器名称NAMES 查看容器内部运行的进程

docker inspect 容器id/容器名称NAMES 查看容器的配置和状态信息

docker rm 容器id/容器名称NAMES 移除web应用容器(移除前必须先stop)

* 1. 镜像

docker pull ubuntu:13.10 下载一个镜像

docker images 列出本机上的镜像

docker search 镜像名 搜索镜像

* 1. 容器连接

docker run -d -P training/webapp python app.py

-d :容器在后台运行

-P: 容器内部端口随机映射到主机的高端口。

docker run -d -p 5000:5000 training/webapp python app.py

-p: 容器内部端口绑定到指定的主机端口

docker run -d -p 127.0.0.1:5001:5000 training/webapp python app.py

指定容器绑定的网络地址127.0.0.1

容器命名：

创建重启的时候，docker会自动对它进行命名，也可使用--name来指定名称

### 常用命令

在运行的容器内执行命令：

docker exec [OPTIONS] [容器ID或容器名称] [具体命令]

-d :分离模式: 在后台运行

-i :即使没有附加也保持STDIN 打开

-t :分配一个伪终端

查看所有容器或某个容器资源使用实时统计：

docker stats [容器ID或容器名称]，容器参数可选

### 镜像

#### base镜像

base镜像底层直接用 Host 的 kernel，自己只需要提供 rootfs 就行了

注意：1、base 镜像只是在用户空间与发行版一致，kernel 版本与发型版是不同的。

例如 CentOS 7 使用 3.x.x 的 kernel，如果 Docker Host 是 Ubuntu 16.04（比如我们的实验环境），那么在 CentOS 容器中使用的实际是是 Host 4.x.x 的 kernel。

ps : uname -r 查看操作系统发行版本

2、容器只能使用 Host 的 kernel，并且不能修改

#### 镜像分层结构

当容器启动时，一个新的可写层被加载到镜像的顶部。

这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”。

所有对容器的改动 - 无论添加、删除、还是修改文件都只会发生在容器层中。

镜像层数量可能会很多，所有镜像层会联合在一起组成一个统一的文件系统。如果不同层中有一个相同路径的文件，比如 /a，上层的 /a 会覆盖下层的 /a，也就是说用户只能访问到上层中的文件 /a。在容器层中，用户看到的是一个叠加之后的文件系统。

添加文件

在容器中创建文件时，新文件被添加到容器层中。

读取文件

在容器中读取某个文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，打开并读入内存。

修改文件

在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后修改之。

删除文件

在容器中删除文件时，Docker 也是从上往下依次在镜像层中查找此文件。找到后，会在容器层中记录下此删除操作。

**容器层记录对镜像的修改，所有镜像层都是只读的，不会被容器修改，所以镜像可以被多个容器共享**

#### 构建镜像

##### docker commit

运行容器->修改容器->将容器保存为新**镜像**

docker run -it ubuntu

安装软件

docker ps 查看当前运行的容器

docker commit oldName newName

##### Dockerfile构建镜像

1. docker build -t 'continer name' .

-t: 容器命名

. : 指明build context（build 环境） 为当前目录，Docker会从build context中查找Dockerfile文件；或用-f 指明Dockerfile文件位置

1. docker将bulid context中的所有文件发送给docker deamon(docker服务器)
2. 执行Dockerfile中的命令：FROM 为base镜像，RUN会进行一系列操作（会在临时创建的镜像上进行），然后将容器保存为镜像，并删除临时容器。

#### 镜像缓存特性

Docker 会缓存已有镜像的镜像层，构建新镜像时，如果某镜像层已经存在，就直接使用，无需重新创建

默认使用缓存，若不想使用缓存，使用 docker build –no-cache

注意：Dockerfile 中每一个指令都会创建一个镜像层，上层是依赖于下层的。无论什么时候，只要某一层发生变化，其上面所有层的缓存都会失效。这时不会使用缓存，会重新创建。

#### debug Dockerfile

通过 Dockerfile 构建镜像的过程:

从 base 镜像运行一个容器。

执行一条指令，对容器做修改。

执行类似 docker commit 的操作，生成一个新的镜像层。

Docker 再基于刚刚提交的镜像运行一个新容器。

重复 2-4 步，直到 Dockerfile 中的所有指令执行完毕。

如果 Dockerfile 由于某种原因执行到某个指令失败了，我们也将能够得到前一个指令成功执行构建出的镜像，此时可以执行:

docker run -it '最后一个成功的镜像id' ; 然后可以手动执行后续操作来调试

#### RUN vs CMD vs ENTRYPOINT

Shell 和 Exec 格式：

两种方式来指定RUN CMD ENTRYPOINT要运行的命令。

Shell格式：<instruction> <command>

shell 格式底层会调用 /bin/sh -c <command>

ENV name Cloud Man

ENTRYPOINT ["/bin/echo", "Hello, $name"]

输出：Hello, Cloud Man

Exec格式：<instruction> ["executable", "param1", "param2", ...]

当指令执行时，会直接调用 <command>，不会被 shell 解析

ENV name Cloud Man

ENTRYPOINT ["/bin/echo", "Hello, $name"]

输出：Hello, $name 此时$name并没被替换

**CMD 和 ENTRYPOINT 推荐使用 Exec 格式，因为指令可读性更强，更容易理解。RUN 则两种格式都可以。**

* RUN 指令通常用于安装应用和软件包；在当前镜像顶部执行命令，创建新的镜像层
* CMD 指令允许用户指定容器的默认执行的命令。

**此命令会在容器启动且 docker run 没有指定其他命令时运行。**

**注意：**

*如果 docker run 指定了其他命令，CMD 指定的默认命令将被忽略*。

如果 Dockerfile 中有多个 CMD 指令，只有最后一个 CMD 有效。

CMD 有三种格式：

Exec 格式：CMD ["executable","param1","param2"]

这是 CMD 的推荐格式。

CMD ["param1","param2"] 与Exec 格式 的 ENTRYPOINT 指令配合使用，是为 ENTRYPOINT 设置默认的参数，此时 ENTRYPOINT 必须使用 Exec 格式。

例如：

Dockerfile: CMD echo "Hello world"

运行容器 docker run -it [image] 将输出：Hello world

若这样运行：docker run -it [image] /bin/bash，CMD 会被忽略掉

Shell 格式：CMD command param1 param2

* ENTRYPOINT

让容器以应用程序或者服务的形式运行。

Exec 格式：

ENTRYPOINT *不会被忽略，一定会被执行*，即使运行 docker run 时指定了其他命令。

ENTRYPOINT 中的参数始终会被使用，而 CMD 的额外参数可以在容器启动时动态替换掉

例如： ENTRYPOINT ["/bin/echo", "Hello"]

CMD ["world"]

当docker run -it [image] 启动时，输出Hello world；

当docker run -it [image] CloudMan启动时，输出Hello CloudMan；

Shell格式：ENTRYPOINT 的 Shell 格式会忽略任何 CMD 或 docker run 提供的参数。

***最佳实践***

使用 RUN 指令安装应用和软件包，构建镜像。

如果 Docker 镜像的用途是运行应用程序或服务，比如运行一个 MySQL，应该优先使用 Exec 格式的 ENTRYPOINT 指令。CMD 可为 ENTRYPOINT 提供额外的默认参数，同时可利用 docker run 命令行替换默认参数。

如果想为容器设置默认的启动命令，可使用 CMD 指令。用户可在 docker run 命令行中替换此默认命令。

#### 镜像命名

实际上一个特定镜像的名字由两部分组成：repository 和 tag



若执行docker build没有指定tag，会默认使用latest，相当于：

docker build -t ubuntu-with-vi:latest

tag一般用于描述版本信息：

每个 repository 可以有多个 tag，而多个 tag 可能对应的是同一个镜像。

假设我们现在发布了一个镜像 myimage，版本为 v1.9.1。那么我们可以给镜像打上四个 tag：1.9.1、1.9、1 和 latest。



docker tag myimage-v1.9.1 myimage:1

docker tag myimage-v1.9.1 myimage:1.9

docker tag myimage-v1.9.1 myimage:1.9.1

docker tag myimage-v1.9.1 myimage:latest

过了一段时间，我们发布了 v1.9.2。这时可以打上 1.9.2 的 tag，并将 1.9、1 和 latest 从 v1.9.1 移到 v1.9.2。



之后，v2.0.0 发布了。这时可以打上 2.0.0、2.0 和 2 的 tag，并将 latest 移到 v2.0.0。



最终：

myimage:1 始终指向 1 这个分支中最新的镜像。

myimage:1.9 始终指向 1.9.x 中最新的镜像。

myimage:latest 始终指向所有版本中最新的镜像。

如果想使用特定版本，可以选择 myimage:1.9.1、myimage:1.9.2 或 myimage:2.0.0。