## Docker概览

## 课程概要

* Docker 简介
* Docker 整体结构了解
* Docker 底层技术了解
* 总结

## Docker简介

### 什么是Docker

Docker是开发，运行和部署应用程序的开放管理平台。

* 开发人员能利用docker 开发和运行应用程序
* 运维人员能利用docker 部署和管理应用程序



### Docker平台介绍 (The Docker platform)

* Docker提供了在一个完全隔离的环境中打包和运行应用程序的能力，这个隔离的环境被称为容器。
* 由于容器的隔离性和安全性，因此可以在一个主机(宿主机)上同时运行多个相互隔离的容器，互不干预。
* Docker已经提供工具和组件(Docker Client、Docker Daemon等)来管理容器的生命周期：
  + 使用容器来开发应用程序及其支持组件。
  + 容器成为分发和测试你的应用程序的单元。
  + 准备好后，将您的应用程序部署到生产环境中，作为容器或协调服务。无论您的生产环境是本地数据中心，云提供商还是两者的混合，这都是一样的。

### 为什么使用Docker？

* Docker使您能够将应用程序与基础架构分开，以便您可以快速交付软件。
* 借助Docker，您可以像管理应用程序一样管理基础架构。
* 通过利用Docker的方法快速进行运输，测试和部署代码，您可以显着缩短编写代码和在生产环境中运行代码之间的延迟。
* 如：
  + 开发人员在本地编写代码，可以使用Docker同事进行共享，实现协同工作。
  + 使用Docker开发完成程序，可以直接对应用程序执行自动和手动测试。
  + 当开发人员发现错误或BUG时，可以直接在开发环境中修复后，并迅速将它们重新部署到测试环境进行测试和验证。
  + 利用Docker开发完成后，交付时，直接交付Docker，也就意味着交付完成。后续如果有提供修补程序或更新，需要推送到生成环境运行起来，也是一样的简单。
* Docker主要解决的问题：
  + **保证程序运行环境的一致性；**
  + **降低配置开发环境、生产环境的复杂度和成本；**
  + **实现程序的快速部署和分发。**

## Docker整体结构了解

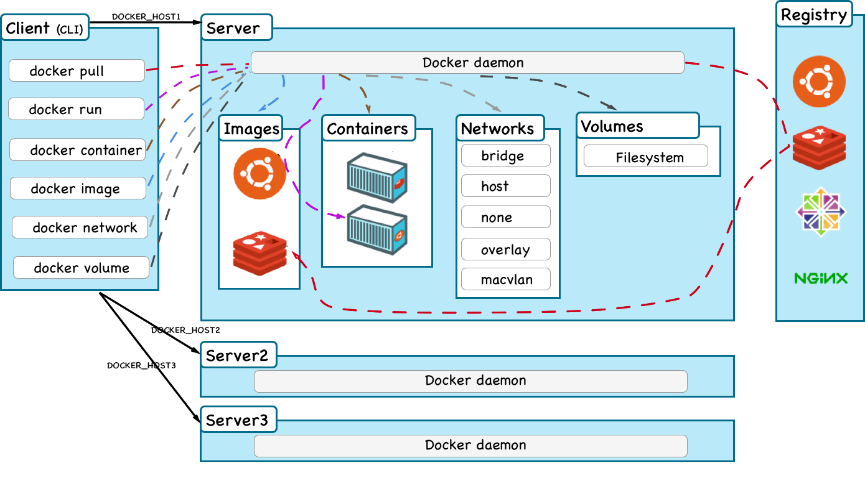
### Docker引擎介绍 (Docker Engine)

* Docker Engine是一个包含以下组件的客户端-服务端(C/S)应用程序
  + 服务端 --- 一个长时间运行的守护进程(Docker Daemon)
  + REST API --- 一套用于与Docker Daemon通信并指示其执行操作的接口
  + 客户端 --- 命令行接口CLI( Command Line Interface)



* CLI利用Docker命令通过REST API直接操控Docker Daemon执行操作
* Docker Daemon负责创建并管理Docker的对象(镜像、容器、网络、数据卷)

### Docker结构概览图



### Docker结构简介

* Docker客户端(Docker Client)
  + Docker客户端(Docker Client)是用户与Docker进行交互的最主要方式。当在终端输入docker命令时，对应的就会在服务端产生对应的作用，并把结果返回给客户端。Docker Client除了连接本地服务端，通过更改或指定DOCKER\_HOST连接远程服务端。
  + Docker服务端(Docker Server)
    - Docker Daemon其实就是Docker 的服务端。它负责监听Docker API请求(如Docker Client)并管理Docker对象(Docker Objects)，如镜像、容器、网络、数据卷等
  + Docker Registries
    - 俗称Docker仓库，专门用于存储镜像的云服务环境.
    - Docker Hub就是一个公有的存放镜像的地方，类似Github存储代码文件。同样的也可以类似Github那样搭建私有的仓库。
  + Docker 对象(Docker Objects)
    - 镜像：一个Docker的可执行文件，其中包括运行应用程序所需的所有代码内容、依赖库、环境变量和配置文件等。
    - 容器：镜像被运行起来后的实例。
    - 网络：外部或者容器间如何互相访问的网络方式，如host模式、bridge模式。
    - 数据卷：容器与宿主机之间、容器与容器之间共享存储方式，类似虚拟机与主机之间的共享文件目录。

## Docker 底层技术了解

Docker底层使用的技术介绍

* Docker使用Go语言实现。
* Docker利用linux内核的几个特性来实现功能:
  + 利用linux的命名空间(Namespaces)
  + 利用linux控制组(Control Groups)
  + 利用linux的联合文件系统(Union File Systems)

这也就意味着Docker只能在linux上运行。

在windows、MacOS上运行Docker，其实本质上是借助了虚拟化技术，然后在linux虚拟机上运行的Docker程序。

* 容器格式（ Container Format ）:
  + Docker Engine将namespace、cgroups、UnionFS进行组合后的一个package，就是一个容器格式(Container Format)。Docker通过对这个package中的namespace、cgroups、UnionFS进行管理控制实现容器的创建和生命周期管理。
  + 容器格式(Container Format)有多种，其中Docker目前使用的容器格式被称为libcontainer
* Namespaces（命名空间）：为Docker容器提供操作系统层面的隔离
  + 进程号隔离：每一个容器内运行的第一个进程，进程号总是从1开始起算
  + 网络隔离：容器的网络与宿主机或其他容器的网络是隔离的、分开的，也就是相当于两个网络
  + 进程间通隔离：容器中的进程与宿主机或其他容器中的进程是互相不可见的，通信需要借助网络
  + 文件系统挂载隔离: 容器拥有自己单独的工作目录
  + 内核以及系统版本号隔离：容器查看内核版本号或者系统版本号时，查看的是容器的，而非宿主机的
* Control Groups（控制组-cgroups）：为Docker容器提供硬件层面的隔离
  + 控制组能控制应用程序所使用的硬件资源。
  + 基于该性质，控制组帮助docker引擎将硬件资源共享给容器使用，并且加以约束和限制。如控制容器所使用的内存大小。
* Union File Systems（联合文件系统--UnionFS）：利用分层(layer)思想管理镜像和容器

## Docker概览-总结

重点掌握Docker以下组件或对象直接的关系

* Docker Client CLI
* Docker Daemon（Docker server）
* Docker Objects
* Registry

了解以下列出的Docker使用的底层技术

* Namespaces
* Control Groups
* Union File Systems
* Container Format

# Docker版本与安装介绍

## 课程概要

* Docker-CE 和 Docker-EE
* Centos 上安装 Docker-CE
* Ubuntu 上安装 Docker-CE
* Windows /MacOS 上安装 Docker-CE

## Docker版本与安装介绍

### Docker-CE和Docker-EE

* Docker-CE指Docker社区版，由社区维护和提供技术支持，为免费版本，适合个人开发人员和小团队使用。
* Docker-EE指Docker企业版，为收费版本，由售后团队和技术团队提供技术支持，专为企业开发和IT团队而设计。
* 相比Docker-EE，增加一些额外功能，更重要的是提供了更安全的保障。
* 此外，Docker的发布版本分为Stable版和Edge版，区别在于前者是按季度发布的稳定版(发布慢)，后者是按月发布的边缘版(发布快)。
* 通常情况下，Docker-CE足以满足我们的需求。后面学习主要针对Docker-CE进行学习。

### CentOS安装Docker-CE

系统要求：CentOS7

查看官方源安装教程、阿里源安装教程(建议)

* 系统要求：CentOS7
* 查看[官方源安装教程](https://docs.docker.com/install/linux/docker-ce/centos/)、[阿里源安装教程](https://help.aliyun.com/document_detail/60742.html)(建议) <https://help.aliyun.com/document_detail/60742.html>

CentOS 7 (使用yum进行安装)

# step 1: 安装必要的一些系统工具

sudo yum install -y yum-utils device-mapper-persistent-data lvm2

# Step 2: 添加软件源信息

sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

# Step 3: 更新并安装Docker-CE

sudo yum makecache fast

sudo yum -y install docker-ce

# Step 4: 开启Docker服务

sudo service docker start

# 注意：

# 官方软件源默认启用了最新的软件，您可以通过编辑软件源的方式获取各个版本的软件包。例如官方并没有将测试版本的软件源置为可用，您可以通过以下方式开启。同理可以开启各种测试版本等。

# vim /etc/yum.repos.d/docker-ee.repo

# 将[docker-ce-test]下方的enabled=0修改为enabled=1

#

# 安装指定版本的Docker-CE:

# Step 1: 查找Docker-CE的版本:

# yum list docker-ce.x86\_64 --showduplicates | sort -r

# Loading mirror speeds from cached hostfile

# Loaded plugins: branch, fastestmirror, langpacks

# docker-ce.x86\_64 17.03.1.ce-1.el7.centos docker-ce-stable

# docker-ce.x86\_64 17.03.1.ce-1.el7.centos @docker-ce-stable

# docker-ce.x86\_64 17.03.0.ce-1.el7.centos docker-ce-stable

# Available Packages

# Step2: 安装指定版本的Docker-CE: (VERSION例如上面的17.03.0.ce.1-1.el7.centos)

# sudo yum -y install docker-ce-[VERSION]

### Ubuntu上安装Docker-CE

#### 官方源安装教程

<https://docs.docker.com/install/linux/docker-ce/ubuntu/>

##### Get Docker CE for Ubuntu

###### Uninstall old versions

Older versions of Docker were called docker or docker-engine. If these are installed, uninstall them:

sudo apt-get remove docker docker-engine docker.io

It’s OK if apt-get reports that none of these packages are installed.

The contents of /var/lib/docker/, including images, containers, volumes, and networks, are preserved. The Docker CE package is now called docker-ce.

###### Install using the repository

Before you install Docker CE for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

SET UP THE REPOSITORY

Update the apt package index:

sudo apt-get update

Install packages to allow apt to use a repository over HTTPS:

首先安装依赖：

sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

gnupg-agent \

software-properties-common

sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-properties-common

Add Docker’s official GPG key:

信任Docker的GPG公钥：

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, by searching for the last 8 characters of the fingerprint.

sudo apt-key fingerprint 0EBFCD88

# output

pub rsa4096 2017-02-22 [SCEA]

9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

uid [ unknown] Docker Release (CE deb) <docker@docker.com>

sub rsa4096 2017-02-22 [S]

Use the following command to set up the stable repository. You always need the stable repository, even if you want to install builds from the edge or test repositories as well. To add the edge or test repository, add the word edge or test (or both) after the word stable in the commands below.

对于amd64架构的计算机，添加软件仓库：

sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

INSTALL DOCKER CE

Update the apt package index.

sudo apt-get update

Install the latest version of Docker CE, or go to the next step to install a specific version:

sudo apt-get install docker-ce

ubuntu 16.04 server 安装docker时出错

(Reading database ... 103698 files and directories currently installed.)

Preparing to unpack .../docker-ce\_18.06.1~ce~3-0~ubuntu\_amd64.deb ...

Unpacking docker-ce (18.06.1~ce~3-0~ubuntu) ...

Processing triggers for man-db (2.7.5-1) ...

Processing triggers for systemd (229-4ubuntu21.4) ...

Processing triggers for ureadahead (0.100.0-19) ...

Setting up docker-ce (18.06.1~ce~3-0~ubuntu) ...

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

insserv: Script redis\_17839 is broken: incomplete LSB comment.

insserv: missing `Required-Start:' entry: please add even if empty.

insserv: missing `Required-Stop:' entry: please add even if empty.

root@aliyun:~# vim /etc/init.d/redis\_17839

root@aliyun:~# vim /etc/init.d/redis\_17839

root@aliyun:~# vim /etc/init.d/redis\_17839

root@aliyun:~# apt remove docker-ce

Reading package lists... Done

参考: <https://gist.github.com/lsbardel/257298>

解决方法: 修改redis启动文件, 在Begin init info中增加required-start和required-stop字段.

vim /etc/init.d/redis\_17839

### BEGIN INIT INFO

# Provides: redis\_17839

# Required-Start: $syslog

# Required-Stop: $syslog

# Should-Start: $local\_fs

# Should-Stop: $local\_fs

# Default-Start: 2 3 4 5

# Default-Stop: 0 1 6

# Short-Description: Redis data structure server

# Description: Redis data structure server. See https://redis.io

### END INIT INFO

卸载docer-ce, 再次安装

apt remove docker-ce

apt install docker-ce

###### Uninstall Docker CE

Uninstall the Docker CE package:

$ sudo apt-get purge docker-ce

Images, containers, volumes, or customized configuration files on your host are not automatically removed. To delete all images, containers, and volumes:

$ sudo rm -rf /var/lib/docker

You must delete any edited configuration files manually.

#### 阿里源安装教程(建议)

只适用于ubuntu 14, 16, 18安装会出错

<https://opsx.alibaba.com/mirror>

<https://help.aliyun.com/document_detail/60742.html>

<https://yq.aliyun.com/articles/110806>

删除源

sudo add-apt-repository -r "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb\_release -cs) stable"

**# step 1: 安装必要的一些系统工具**

sudo apt-get update

sudo apt-get -y install apt-transport-https ca-certificates curl software-properties-common

**# step 2: 安装GPG证书**

curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -

**# Step 3: 写入软件源信息**

sudo add-apt-repository "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb\_release -cs) stable"

**# Step 4: 更新并安装Docker-CE**

sudo apt-get -y update

sudo apt-get -y install docker-ce

**# 安装指定版本的Docker-CE:**

# Step 1: 查找Docker-CE的版本:

# apt-cache madison docker-ce

# docker-ce | 17.03.1~ce-0~ubuntu-xenial | http://mirrors.aliyun.com/docker-ce/linux/ubuntu xenial/stable amd64 Packages

# docker-ce | 17.03.0~ce-0~ubuntu-xenial | http://mirrors.aliyun.com/docker-ce/linux/ubuntu xenial/stable amd64 Packages

# Step 2: 安装指定版本的Docker-CE: (VERSION例如上面的17.03.1~ce-0~ubuntu-xenial)

# sudo apt-get -y install docker-ce=[VERSION]

开启Docker服务

sudo service docker start

安装成功后，使用 docker version查看

如果出现权限限制, 需要切换到root用户执行命令

#### 添加到用户组（可选项）

添加到用户组（so easy）

sudo groupadd docker

sudo usermod -aG docker david

注销系统重新进入系统，就可以直接使用docker开头了。

如果不添加到用户组会发生什么呢？

如果直接运行:

docker run hello-world

你会发现下面的错误:

Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.30/containers/json: dial unix /var/run/docker.sock: connect: permission denied

这是因为：

docker守护程序绑定到Unix套接字而不是TCP端口。默认情况下，Unix套接字由用户root拥有，其他用户只能使用sudo访问它。 docker守护程序始终以root用户身份运行。 如果您不想在使用docker命令时使用sudo，请创建名为docker的Unix组，并将用户添加到该组。当docker守护进程启动时，它会使Docker组的Unix套接字的所有权读/写。

linux下安装一条命令即可

curl -sSL https://get.daocloud.io/docker | sh

这条命令在ubuntu 14.04和ubuntu 16.04都可以成功安装docker。

安装完成功后，可能会提示你这样的信息:

If you would like to use Docker as a non-root user, you should now consider

adding your user to the "docker" group with something like:

sudo usermod -aG docker vagrant

Remember that you will have to log out and back in for this to take effect!

vagrant是你的用户名，可能你的用户名跟我的不一样。

意思就是说，你可以把当前用户加入到docker组，以后要管理docker就方便多了，不然你以后有可能要使用docker命令前，要在前面加sudo。

如果没加sudo就是类似这样的提示:

Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.26/containers/json: dial unix /var/run/docker.sock: connect: permission denied

不过执行了sudo usermod -aG docker vagrant之后，你再重新登录(ssh)，就可以免去加sudo。

安装成功，需要把docker这个服务启动起来：

如果是ubuntu 14.04的系统，它会自动启动，你也可以使用下面的命令来启动。

$ sudo /etc/init.d/docker start

如果是ubuntu 16.04的系统，就用下面的命令：

$ sudo systemctl status docker.service

### 开启Docker服务

sudo service docker start

安装成功后，使用 docker version 命令查看

如果出现权限禁止, 需要切换到root用户执行命令.

### Windows/MacOS上安装Docker-CE



* [安装软件下载地址](https://download.daocloud.io/Docker_Mirror/Docker)

### Docker 加速器配置

* 为什么使用加速器：
  + 配置Docker加速器，将会提升在国内获取Docker官方镜像的速度，否则后面下载镜像的过程会很慢，甚至有可能无法下载镜像
* 配置阿里云加速器

<https://help.aliyun.com/document_detail/60750.html?spm=a2c4g.11186623.6.545.OY7haW>

官方镜像加速

我们使用Docker的第一步，应该是获取一个官方的镜像，例如mysql、wordpress。由于网络原因，我们下载一个Docker官方的镜像需要很长的时间，甚至下载失败。为此，阿里云容器镜像服务提供了官方的镜像站点，从而加速官方镜像的下载速度。

使用镜像加速器

在不同的系统下，配置加速器的方式略有不同，所以我们介绍主要的几个操作系统的配置方法。

关于加速器的地址，您登录容器镜像服务控制台 后左侧的加速器帮助页面就会显示为您独立分配的加速地址。

<https://cr.console.aliyun.com/?spm=a2c4g.11186623.2.13.2c3111beKdD48s>

例如：

公网Mirror：[系统分配前缀].mirror.aliyuncs.com

当您的Docker版本较新时

当您下载安装的Docker Version不低于1.10时，建议直接通过daemon config进行配置。使用配置文件/etc/docker/daemon.json（没有时新建该文件）。

https://dlfa9xic.mirror.aliyuncs.com

**Ubuntu**

**1. 安装／升级Docker客户端**

推荐安装1.10.0以上版本的Docker客户端，参考文档 docker-ce

**2. 配置镜像加速器**

针对Docker客户端版本大于 1.10.0 的用户

您可以通过修改daemon配置文件/etc/docker/daemon.json来使用加速器

sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://dlfa9xic.mirror.aliyuncs.com"]

}

EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

**CentOS**

**1. 安装／升级Docker客户端**

推荐安装1.10.0以上版本的Docker客户端，参考文档 docker-ce

**2. 配置镜像加速器**

针对Docker客户端版本大于 1.10.0 的用户

您可以通过修改daemon配置文件/etc/docker/daemon.json来使用加速器

sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://dlfa9xic.mirror.aliyuncs.com"]

}

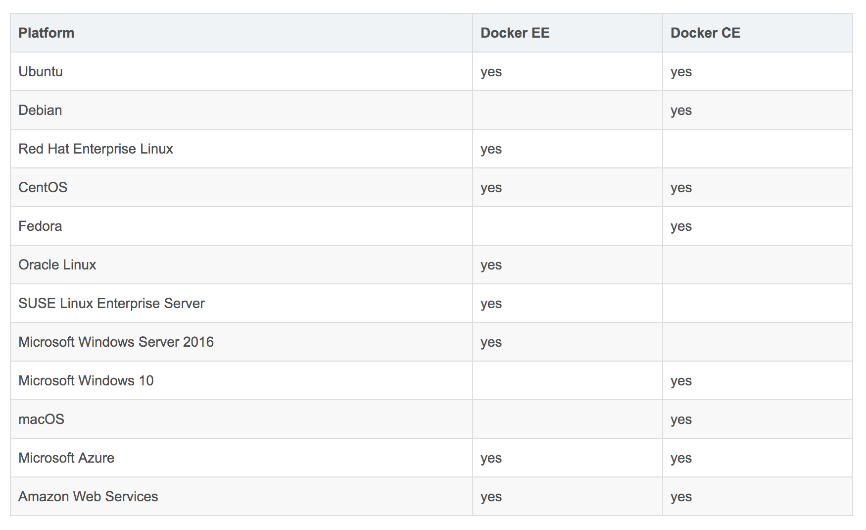
EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

### 本章总结





## Docker核心技术之镜像

### 镜像简介

* 镜像是一个Docker的可执行文件，其中包括运行应用程序所需的所有代码内容、依赖库、环境变量和配置文件等。
* 通过镜像可以创建一个或多个容器。

如果是系统镜像, 需要指定启动的命令, 如bash命令. bash命令的作用: 打开一个新的终端. exit退出当前终端

### 镜像搜索 - docker search

* 作用：

**搜索Docker Hub(镜像仓库)上的镜像**

* 命令格式：

**docker search [OPTIONS] TERM**

* 命令参数(OPTIONS)：

**-f, --filter filter 根据提供的格式筛选结果**

**--format string 利用Go语言的format格式化输出结果**

**--limit int 展示最大的结果数，默认25个**

**--no-trunc 内容全部显示, 显示完整的描述信息**

# 搜索镜像

docker search centos

docker search cent

# 搜索官方镜像

docker search -f is-official=true centos

docker search -f is-official=true redis

--limit 限制条件执行的优先级比较高, 如果无其它搜索限制条件时会限制搜索结果的数量. 有其它搜索限制条件时会先执行limit, 再执行其它的限制条件, 可能会无法搜索到想要的结果.

AUTOMATED, [OK] 表示通过docker file构建的镜像, 没有[OK]的表示用户上传的就是构建好的镜像.

### 镜像下载 - docker pull

* 作用：

**下载远程仓库（如Docker Hub）中的镜像**

* 命令格式：

**docker pull [OPTIONS] NAME[:TAG|@DIGEST]**

* 命令参数(OPTIONS)：

**-a, --all-tags 下载所有符合给定tag的镜像**

docker pull centos:7

docker pull centos:latest

docker pull ubuntu:16.04

### 镜像查看 - docker images/docker image ls

* 作用：

**列出本地镜像**

* 命令格式：

**docker images [OPTIONS] [REPOSITORY[:TAG]]**

或者 **docker image ls [OPTIONS] [REPOSITORY[:TAG]]**

* 命令参数(OPTIONS)：

**-a, --all 展示所有镜像 (默认隐藏底层的镜像)**

**--no-trunc 不缩略显示**

**-q, --quiet 只显示镜像ID**

Usage: docker image COMMAND

Manage images

Commands:

build Build an image from a Dockerfile

history Show the history of an image

import Import the contents from a tarball to create a filesystem image

inspect Display detailed information on one or more images

load Load an image from a tar archive or STDIN

ls List images

prune Remove unused images

pull Pull an image or a repository from a registry

push Push an image or a repository to a registry

rm Remove one or more images

save Save one or more images to a tar archive (streamed to STDOUT by default)

tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

Run 'docker image COMMAND --help' for more information on a command.

docker images

docker images ubu\*

docker images ubuntu

docker images ubuntu

docker images ubuntu:latest

docker images cen\*

docker images ubu\*

# 如果本地镜像中有多个name相同, 但是tag不相同的镜像, 使用ubu\*的方式就无法显示出来了, 必须要指定镜像的版本.

docker images ubu\*:latest

docker pull centos:latest

# 最新版本的是7, 版本同latest, 如果本地已经存在相同版本的, 就不会再启动下载过程了, 而是从本地复制, 把centos:latest复制为centos:7. 只是又帖了个标签.

docker pull centos:7

### 镜像删除 - docker rmi/docker image rm

* 作用：

**将本地的一个或多个镜像删除**

* 命令格式：

**docker rmi [OPTIONS] IMAGE [IMAGE...]**

或者 **docker image rm [OPTIONS] IMAGE [IMAGE...]**

* 命令参数(OPTIONS)：

**-f, --force 强制删除, 已经创建出容器并运行时, 无法正常删除镜像, 必须使用 -f 命令强制删除**

docker rmi -h

docker rmi --help

# 使用image id删除, 指定的image id的前几位必须要能够唯一的指定某个镜像. 否则会报错.

docker rmi e934 452a

# 使用name:tag删除

docker rmi ubuntu, 会删除latest版本.

docker rmi ubuntu:18.04

-f 强制删除. 已经通过某个镜像生成了容器, 就必须使用-f才能强制删除.

### 镜像保存备份–docker save

Usage: docker save [OPTIONS] IMAGE [IMAGE...]

Save one or more images to a tar archive (streamed to STDOUT by default)

Options:

-o, --output string Write to a file, instead of STDOUT

* 作用：

**将本地的一个或多个镜像打包保存成本地tar文件(输出到STDOUT)**

* 命令格式：

**docker save [OPTIONS] IMAGE [IMAGE...]**

* 命令参数(OPTIONS)：

**-o, --output string 指定写入的文件名和路径**

# 使用重定向符

docker save centos ubuntu ubuntu14.04 centos:7 > linux.tar

# 使用-o

docker save centos ubuntu ubuntu14.04 centos:7 -o linux2.tar

如果是以指定image id的方式导出的, 就会丢失name和tag, 再次导入时这两个都会变成none.

# docker save -o linux\_images.tar centos ubuntu

### 镜像备份导入 - docker load

Usage: docker load [OPTIONS]

Load an image from a tar archive or STDIN

Options:

-i, --input string Read from tar archive file, instead of STDIN

-q, --quiet Suppress the load output

* 作用：

**将save命令打包的镜像导入本地镜像库中**

* 命令格式：

**docker load [OPTIONS]**

* 命令参数(OPTIONS)：

**-i, --input string 指定要打入的文件，如没有指定，默认是STDIN**

**-q, --quiet 不打印导入过程信息**

# 从stdin中导入

docker load

# 从文件中导入

docker load -i linux.tar

docker images

docker load -i linux\_images.tar

docker load -i -q linux\_images.tar

### 镜像重命名–docker tag

Usage: docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]

Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

* 作用：

**对本地镜像的NAME、TAG进行重命名，并新产生一个命名后镜像**

* 命令格式：

**docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]**

* 命令参数(OPTIONS)：

**无**

docker tag e934 centos-newname:newtag

# 重命名并不是把原来的镜像进行重命名, 而是增加了一个引用, 这个引用的name和tag就是这里指定的新的名称.

docker tag centos centos:7.4

# 如果name和tag都为none, 重命名时就是在原镜像上进行的重命名. 当name和tag都为空时, 相当于镜像在本地是存在的, 但是没有标签指向这个镜像, 重命名时就相当于给它贴上了一个标签.

docker tag 452a ubuntu:latest

### 镜像详细信息–docker image inspect/docker inspect

docker image inspect -h

Usage: docker image inspect [OPTIONS] IMAGE [IMAGE...]

Display detailed information on one or more images

Options:

-f, --format string Format the output using the given Go template

docker inspect -h

Usage: docker inspect [OPTIONS] NAME|ID [NAME|ID...]

Return low-level information on Docker objects

Options:

-f, --format string Format the output using the given Go template

-s, --size Display total file sizes if the type is container

--type string Return JSON for specified type

* 作用：

**查看本地一个或多个镜像的详细信息**

* 命令格式：

**docker image inspect [OPTIONS] IMAGE [IMAGE...]**

或者 **docker inspect [OPTIONS] IMAGE [IMAGE...]**

* 命令参数(OPTIONS)：

**-f, --format string 利用特定Go语言的format格式输出结果**

docker image inspect -h

# 返回json字符串

docker image inspect centos

# 查看某个字段的值, 只查看 "Created" 字段的值

docker image inspect centos | grep Created

# 但grep无法查询多行的信息, 只能查询一行的信息.

docker image inspect centos | grep Data

# 使用-f参数. .Id 中的点代表的是第一层

docker image inspect -f "{{json .Id}}" centos

docker image inspect -f "{{json .GraphDriver.Data}}" centos

docker image inspiect -f "{{json .Created}}" centos

docker inspect -f "{{json .Data}}"

docker image inspect centos

# docker images inspect只能查看镜像.

docker image inspect -h

# docker inspect 不只能查看镜像, 还可以查看容器

docker inspect -h

### 镜像历史信息–docker history

Usage: docker history [OPTIONS] IMAGE

Show the history of an image

Options:

--format string Pretty-print images using a Go template

-H, --human Print sizes and dates in human readable format

(default true)

--no-trunc Don't truncate output

-q, --quiet Only show numeric IDs

* 作用：

**查看本地一个镜像的历史(历史分层)信息**

* 命令格式：

**docker history [OPTIONS] IMAGE**

* 命令参数(OPTIONS)：

**-H, --human 将创建时间、大小进行优化打印(默认为true)**

**-q, --quiet 只显示镜像ID**

**--no-trunc 不缩略显示**

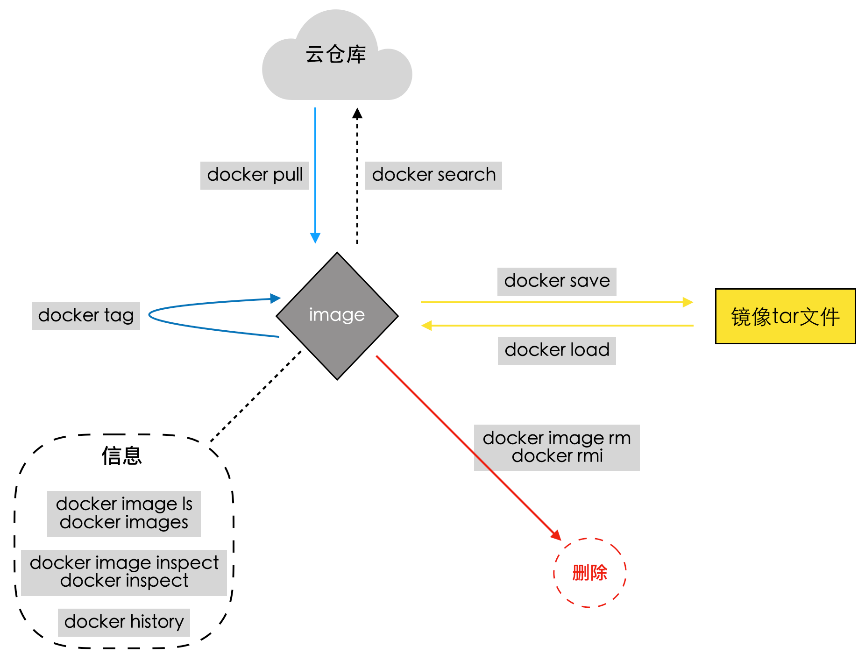
# 只能查看一个镜像的信息

docker histroy ubuntu

# -H将创建时间、大小进行优化打印

docker history ubuntu -H=false

### 镜像总结



## Docker核心技术之容器

### 课程概要

* 容器简介
* 容器与虚拟机
* **容器生命周期**
* **容器生命周期管理**
* **容器信息查看**
* **容器运行时操作**
* 容器总结

### 容器简介

#### 什么是容器

**容器（Container）：容器是一种轻量级、可移植、并将应用程序进行的打包的技术，使应用程序可以在几乎任何地方以相同的方式运行**

* **Docker将镜像文件运行起来后，产生的对象就是容器。容器相当于是镜像运行起来的一个实例。相当于类和对象之间的关系.**
* **容器具备一定的生命周期。**
* **另外，可以借助docker ps命令查看运行的容器，如同在linux上利用ps命令查看运行着的进程那样。**

**容器可以看成是一堆进程的集合.**

### 容器与虚拟机

#### Docker容器与虚拟机相同点

* 容器和虚拟机一样，都会对物理硬件资源进行共享使用。
* 容器和虚拟机的生命周期比较相似（创建、运行、暂停、关闭等等）。
* 容器中或虚拟机中都可以安装各种应用，如redis、mysql、nginx等。也就是说，在容器中的操作，如同在一个虚拟机(操作系统)中操作一样。
* 同虚拟机一样，容器创建后，会存储在宿主机上：linux上位于/var/lib/docker/containers下

cd /var/lib/docker/containersdocker ps

docker ps

ls

docker run --rm -dti centos

docker ps

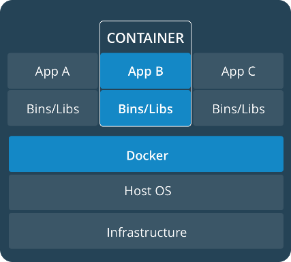
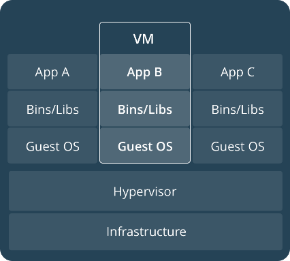
ls

cd 12891safa90afsdafa

#### Docker容器与虚拟机不同点

注意：容器并不是虚拟机，但它们有很多相似的地方

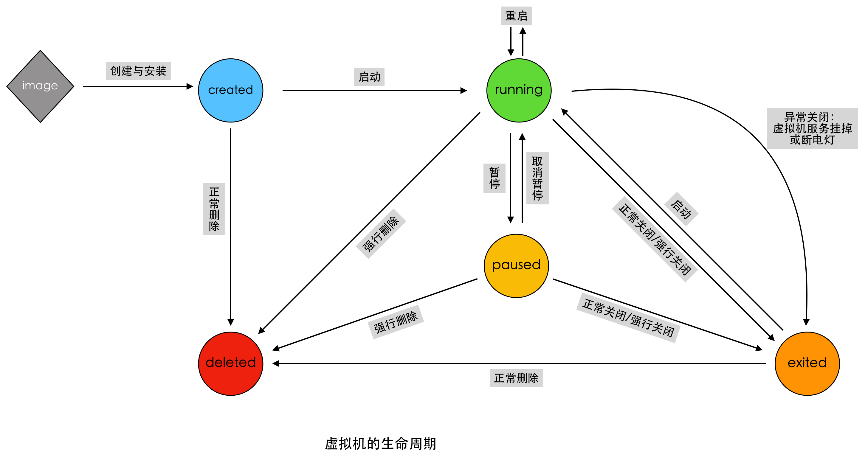
* 虚拟机的创建、启动和关闭都是基于一个完整的操作系统。一个虚拟机就是一个完整的操作系统。而容器直接运行在宿主机的内核上，其本质上以一系列进程的结合。
* 容器是轻量级的，虚拟机是重量级的。首先容器不需要额外的资源来管理(不需要Hypervisor、Guest OS)，虚拟机额外更多的性能消耗；其次创建、启动或关闭容器，如同创建、启动或者关闭进程那么轻松，而创建、启动、关闭一个操作系统就没那么方便了。
* 也因此，意味着在给定的硬件上能运行更多数量的容器，甚至可以直接把Docker运行在虚拟机上。

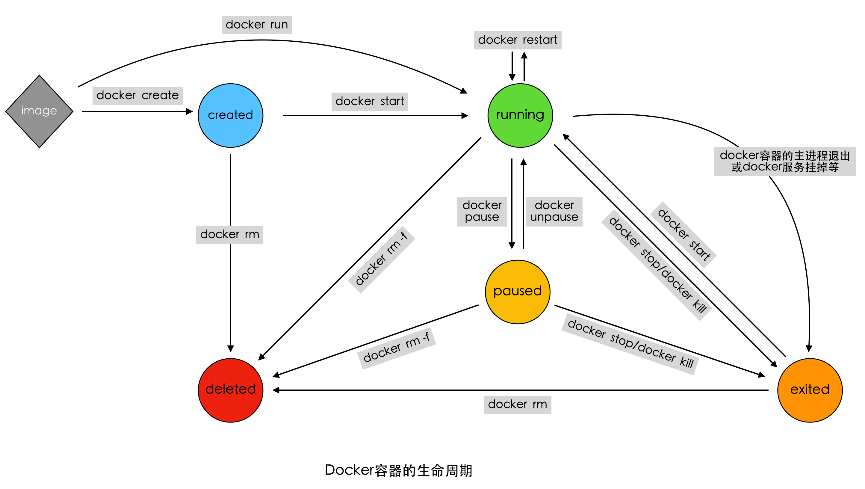
容器本质上是一些进程的集合, 创建容器就相当于创建一些进程.

### 虚拟机的生命周期

#### 虚拟机的生命周期



#### 容器的生命周期



#### 容器的生命周期管理

#### 容器创建–docker create

Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]

Create a new container

* 作用：

**利用镜像创建出一个Created 状态的待启动容器**

* 命令格式：

**docker create [OPTIONS] IMAGE [COMMAND] [ARG...]**

* 命令参数(OPTIONS)：[查看更多](https://docs.docker.com/engine/reference/commandline/create/)

**-t, --tty 分配一个伪TTY，也就是分配虚拟终端**

**-i, --interactive 即使没有连接，也要保持STDIN打开**

**--name 为容器起名，如果没有指定将会随机产生一个名称**

* 命令参数（COMMAND\ARG）:

**COMMAND 表示容器启动后，需要在容器中执行的命令，如ps、ls 等命令**

**ARG 表示执行 COMMAND 时需要提供的一些参数，如ps 命令的 aux、ls命令的-a等等**

# 不能使用-h查看帮助文档, docker create -h --hostname 用来设置docker 的hostname

# docker create -h

# 查看帮助文档

docker create --help

# 关于command参数, 每一个镜像都有一些自己特殊的命令, 如ubuntu apt, centos yum, linux通用的ls, ps命令, redis的redis-cli, redis-server命令, 代表镜像的命令行/终端命令. command可以不传.

# 关于 ARG 参数. 很多命令 command 都有自己的参数, 如ls -a, ps -a 参数. ARG参数是可选的参数, 可以省略, 省略时会使用默认的参数, 如查看centos镜像的信息

docker inspect centos

# cmd: /bin/bash # 表示在启动容器时会执行的默认命令, 相当于使用docker create centos /bin/bash 命令来创建的容器

# command命令可以不传, 对于centos, 默认的是inspect centos中的cmd命令, 即 "/bin/bash"

# 新建ls -a 命令的容器, ls -a就是最后的COMMAND和ARG, 此时就会覆盖掉原来默认的 "/bin/bash" 命令, 运行容器时执行的就是 "ls -a" 命令,

docker create centos ls -a

# 再次使用 docker inspect centos 命令查看centos容器的参数, 在command中就会看到 "ls -a"

# 查看所有的容器. docker ps 默认查看处于运行状态的容器, 加上 -a 参数可以查看所有的容器.

docker ps -a

# --name 创建时指定容器的名称. docker ps -a 中的NAMES 参数, 如果不指定, 会分配一个随机的名称

docker create --name test-container centos ps -A

# -t 虚拟终端. -i 标准输入.

-i, --interactive Keep STDIN open even if not attached

-t, --tty Allocate a pseudo-TTY

# 以python命令创建基于centos的容器, 想要进入python交互环境, 必须有一个终端, -t 即虚拟终端. 想要在python交互环境中进入输入, 还要提供标准输入, 输入的内容才能被python交互环境接收并执行. -i 就提供了标准输入.

docker create -ti centos python

docker create -ti --name test-container2 centos /bin/bash

#### 容器启动–docker start

* 作用：

**将一个或多个处于创建状态或关闭状态的容器启动起来**

* 命令格式：

**docker start [OPTIONS] CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**-a, --attach 将当前shell的 STDOUT/STDERR 连接到容器上**

**-i, --interactive 将当前shell的 STDIN连接到容器上**

docker start -h

Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]

Start one or more stopped containers

Options:

-a, --attach Attach STDOUT/STDERR and forward signals

--detach-keys string Override the key sequence for detaching a container

-i, --interactive Attach container's STDIN

# 创建带有命令的容器. 因为默认的 /bin/bash命令是没有输出内容的, 所以为了演示, 这里创建带有ls -a命令的容器. 命令是在容器启动的时候就会被执行的.

docker create --name centos-test centos ls -a

docker ps -a

# 启动容器时只输出了一个容器的id值. 并没有看到执行ls -a命令的过程, 也没有看到 ls -a 命令的执行结果.

docker start centos-test

# 364f

# 对于问题1, 没有看到 ls -a 命令的执行过程. docker ps -a查看容器的状态, centos-test 容器是处于Exited状态的, 并不是我们认为的Running状态, 这是因为容器实际上是针对一个命令去执行的进程, ls -a命令执行结束之后, 进程就结束了, 容器的运行也就结束了. 因为ls -a命令的执行过程非常短, 就会出现一启动容器就结束的现象, 不会看到中间的running状态.

# 对于问题2, 并没有看到 ls -a 命令的执行结果.

# -a --attach Attach STDOUT/STDERR and forward signals, 把当前终端的标准输出和标准错误绑定到容器中

# docker start的 -a参数把容器的标准输出或标准错误绑定到当前物理机的标准输出中. 使用docker start -a 执行centos-test, 就会在物理机的终端中执行ls -a的命令.

docker start -a centos-test

# 创建一个执行python命令进入到python终端中的容器

docker create --name centos-python centos python

docker ps -a

# 使用 -a参数启动容器, 本应输出进入到python交互环境的提示, 但这里什么输出都没有出现, 这是因为python这样的进入交互环境的命令想要执行成功, 要给它提供一个交互环境的终端.

docker start -a centos-python

# -i 参数

# -i, --interactive Keep STDIN open even if not attached

# -i, --interactive 将当前shell的 STDIN连接到容器上

# -t, --tty Allocate a pseudo-TTY

# -t, --tty 分配一个伪TTY，也就是分配虚拟终端

# 想要能够在当前物理机的环境中进入到容器中的python命令的交互环境中, 必须要在创建容器时使用 -t命令给容器添加一个虚拟终端, 并且要在创建容器时使用 -i命令把当前shell的标准输入连接到容器上. 这样才能在当前物理机的shell中对容器中的python交互环境进行交互操作.

# -ti, 分配一个虚拟终端, 并且提供一个标准输入

docker create -ti --name python-centos2 centos python

# 使用 -a 参数再次运行容器, 把容器的标准输出和标准错误绑定到当前物理机中. 出现了进入python交互环境的提示信息, 并且进入到了python交互环境, 但是无法进行输入和交互. 但在上面创建容器时已经提供了 -i 参数, 为什么还是无法进行交互输入呢, 这是因为 docker create 命令中的 -i 是创建容器时的参数, 运行容器时也要提供 -i 参数才能进入交互环境并提供标准输入.

docker start -a python-centos2

# 还必须要使用docker start -i参数把容器的标准输入绑定到物理机中, 这样才能在物理机中对容器中的python交互界面进行交互.

docker start -ai python-centos2

# 实际上运行容器时只需要提供一个 -i 参数就能启动容器进入到python交互环境, 不用提供 -a 参数

docker start -i python-centos2

**/bin/bash 作用详解**

# 在终端中输入bash命令会新打开一个终端. 每执行一次就会打开一个终端, 并把当前终端绑定到新打开的终端上.

ps -a

# PID TTY TIME CMD

# 25605 pts/0 00:00:00 ps

bash

ps -a

# PID TTY TIME CMD

# 25636 pts/0 00:00:00 bash

# 25664 pts/0 00:00:00 ps

bash

ps -a

# PID TTY TIME CMD

# 25636 pts/0 00:00:00 bash

# 25701 pts/0 00:00:00 bash

# 25723 pts/0 00:00:00 ps

# bash命令与python命令类似, /bin/bash是centos创建时的默认命令, 想要在物理机的shell中对容器的shell进行交互操作, 必须也要在创建是指定 -ti 参数, 在运行时添加 -ai 参数.

#### 容器创建并启动–docker run

* 作用：

**利用镜像创建并启动一个容器**

* 命令格式：

**docker run [OPTIONS] IMAGE [COMMAND] [ARG...]**

* 命令参数(OPTIONS)：[查看更多](https://docs.docker.com/engine/reference/commandline/run/)

**-t, --tty 分配一个伪TTY，也就是分配虚拟终端**

**-i, --interactive 即使没有连接，也要保持STDIN打开**

**--name 为容器起名，如果没有指定将会随机产生一个名称**

**-d, --detach 在后台运行容器并打印出容器ID**

**--rm 当容器退出运行后，自动删除容器**

* 命令参数（COMMAND\ARG）:

**COMMAND 表示容器启动后，需要在容器中执行的命令，如ps、ls 等命令**

**ARG 表示执行 COMMAND 时需要提供的一些参数，如ps 命令的 aux、ls命令的-a等等**

**# docker run 相当于 docker create + docker start –a 前台模式**

# docker run == docker create + docker start -a

# 创建一个命令为 ls -a 的容器, 会直接执行 ls -a命令, 并在物理机上显示执行结果. 这里就不需要使用 -a attach 参数了.

docker run centos ls -a

# 对于需要交互界面的命令, 如python, 直接使用run命令无任何结果.

docker run centos python

# 必须还要使用 -ti 参数来指定虚拟终端并绑定到物理机. 才能进入到容器中python命令的交互界面中.

docker run -ti centos python

# -d参数. 在后台执行

# -d, --detach Run container in background and print container ID

# 使用 -d参数, 就不会进入到python的交互界面中去. 而是只打印出创建的容器的id.

docker run -d -ti centos python

# 等价于

docker create -ti centos python

docker start 5832

**# docker run -d == docker create + docker start 后台模式**

docker start -a (--attach) -a 把当前物理机终端的标准输出和标准错误绑定到容器中.

docer start -i (--interactive) 把当前物理机的标准输入绑定到容器上.

docker run -d (--dettach) 如果create和start时 -a -i参数都不指定, 相当于 docker run -d, 什么都不绑定, 后台模式.

docker run centos ps -A

docker run -d centos ps -a

--rm参数, 当容器退出时, 自动删除容器.

# 不使用--rm参数, 在运行结束时可以再次启动.

docker run -d centos ps -A

# 使用--rm参数, 运行结束时就自动删除.

docker run --rm -d centos ps -A

docker run --rm -ti centos bash

#### 容器暂停–docker pause

* 作用：

**暂停一个或多个处于运行状态的容器**

* 命令格式：

**docker pause CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**无**

Usage: docker pause CONTAINER [CONTAINER...]

Pause all processes within one or more containers

docker pause test-container2

容器可以看成是多个容器的集合, 一个容器中可能存在多个进程, docker pause是暂停容器中的所有进程

# 暂停状态是运行状态的一种特殊状态. up 2 minutes (paused)

docker ps -a

#### 容器取消暂停–docker unpause

* 作用：

**取消一个或多个处于暂停状态的容器，恢复运行**

* 命令格式：

**docker unpause CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**无**

docker uppause test-container2

#### 容器关闭–docker stop

* 作用：

**关闭一个或多个处于暂停状态或者运行状态的容器**

* 命令格式：

**docker stop [OPTIONS] CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**-t, --time int 关闭前，等待的时间，单位秒(默认 10s)**

# 创建并运行带有python命令的容器, 会在当前终端中直接进入容器的python的交互环境中, 并且处于阻塞模式下. 但只要使用quit() 或 Ctrl+D 退出了python交互环境, python命令的执行就结束了, 容器的运行也就结束了.

docker run -ti centos python

# 必须要加上-d参数, 让容器在后台运行, 这样才不会随着python交互终端的退出而导致容器运行的结束.

docker run -dti centos python

# 使用stop命令关闭在后台运行的centos容器.

# 直接使用stop命令退出时, 就会阻塞10s再关闭容器.

docker stop 8515

# 使用 -t 参数指定退出前阻塞的时间.

docker start 8515

docker stop -t 2 8515

# 再次运行8515, 再次退出. 但一执行docker stop 8515命令, 就马上按下Ctrl+C中止程序的运行. 如果在10s内执行 docker ps -a, 就会看到8515的容器依然在运行. 但如果等到10s之后再运行docker ps -a, 就会发现8515的容器已经处于exit的状态了. 也就是说Ctrl+C命令并不会中断docker stop命令的执行.

# 可以多次结束一个程序的运行, 不管它之前是否处在结束状态, 都不会报错.

docker stop 8515

#### 容器终止–docker kill

* 作用：

**强制并立即关闭一个或多个处于暂停状态或者运行状态的容器**

* 命令格式：

**docker kill [OPTIONS] CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**-s, --signal string 指定发送给容器的关闭信号 (默认"KILL"信号)**

Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill one or more running containers

Options:

-s, --signal string Signal to send to the container (default "KILL")

docker start 8515

# docker kill 命令立即结束容器进程, 不会堵塞和等待.

docker kill 8515

# kill与stop -t 0的区别:

**# 理解liunx的kill信号**

# 在一个终端中进入python2.7, 在另一个终端中查找python2.7的进程号.

ps -aux | grep python2.7

# 使用kill来结束进程

kill 28922

# 此时在运行python2.7的终端中会显示出 "Terminated", 并退出终端.

# 在一个终端中再次执行python2.7, 在另一个终端中查找 python2.7的进程号

ps -aux | grep python2.7

# 使用kill -9 来强制结束进程

kill -9 28923

# 此时在运行python2.7的终端中会显示出 "Killed", 并退出终端.

# 查看linux kill命令的所有信号值.

kill -l

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1) | SIGHUP | 2) | SIGINT | 3) | SIGQUIT |
| 4) | SIGILL | 5) | SIGTRAP | 6) | SIGABRT |
| 7) | SIGBUS | 8) | SIGFPE | 9) | SIGKILL |
| 10) | SIGUSR1 | 11) | SIGSEGV | 12) | SIGUSR2 |
| 13) | SIGPIPE | 14) | SIGALRM | 15) | SIGTERM |
| 16) | SIGSTKFLT | 17) | SIGCHLD | 18) | SIGCONT |
| 19) | SIGSTOP | 20) | SIGTSTP | 21) | SIGTTIN |
| 22) | SIGTTOU | 23) | SIGURG | 24) | SIGXCPU |
| 25) | SIGXFSZ | 26) | SIGVTALRM | 27) | SIGPROF |
| 28) | SIGWINCH | 29) | SIGIO | 30) | SIGPWR |
| 31) | SIGSYS | 34) | SIGRTMIN | 35) | SIGRTMIN+1 |
| 36) | SIGRTMIN+2 | 37) | SIGRTMIN+3 | 38) | SIGRTMIN+4 |
| 39) | SIGRTMIN+5 | 40) | SIGRTMIN+6 | 41) | SIGRTMIN+7 |
| 42) | SIGRTMIN+8 | 43) | SIGRTMIN+9 | 44) | SIGRTMIN+10 |
| 45) | SIGRTMIN+11 | 46) | SIGRTMIN+12 | 47) | SIGRTMIN+13 |
| 48) | SIGRTMIN+14 | 49) | SIGRTMIN+15 | 50) | SIGRTMAX-14 |
| 51) | SIGRTMAX-13 | 52) | SIGRTMAX-12 | 53) | SIGRTMAX-11 |
| 54) | SIGRTMAX-10 | 55) | SIGRTMAX-9 | 56) | SIGRTMAX-8 |
| 57) | SIGRTMAX-7 | 58) | SIGRTMAX-6 | 59) | SIGRTMAX-5 |
| 60) | SIGRTMAX-4 | 61) | SIGRTMAX-3 | 62) | SIGRTMAX-2 |
| 63) | SIGRTMAX-1 | 64) | SIGRTMAX | |  |

# 在linux中通过 kill 2892 命令来结束进程的运行时默认发送的是15号信号来结束进程的, 即SIGTERM信号.

# 而kill -9 28923则是发送的SIGKILL信号kill命令.

# docker stop xxxx 来结束docker的运行时, 默认是发送的SIGTERM信号. SIGTERM信号默认会给进程 10s 的时间来进行后续的处理.

# 当docker stop命令使用 -t 5 指定等待时间为5s后, 会再次发送 SIGKILL信号来结束进程.

# docker kill xxxx 则是直接发送SIGKILL信号来结束容器的运行. 可以通过指定 -s 参数来修改发送的中止信号.

docker kill -s 'SIGSYS' xxxx

#### docker stop和docker kill的区别

* 前提知识点：
  + Linux其中两种终止进程的信号是：SIGTERM和SIGKILL
  + **SIGKILL**信号：无条件终止进程信号。进程接收到该信号会立即终止，不进行清理和暂存工作。该信号不能被忽略、处理和阻塞，它向系统管理员提供了可以杀死任何进程的方法。
  + **SIGTERM**信号：程序终结信号，可以由kill命令产生。与SIGKILL不同的是，SIGTERM信号**可以被阻塞和终止**，以便程序在退出前可以保存工作或清理临时文件等。有些保护性非常强的进程, 可以忽略或中止SIGTERM信号, 而SIGKILL信号则会无理由关闭进程, 不能被忽略, 中止或阻塞.
  + docker stop 会先发出SIGTERM信号给进程，告诉进程即将会被关闭。在-t指定的等待时间过了之后，将会立即发出SIGKILL信号，直接关闭容器。
  + docker kill 直接发出SIGKILL信号关闭容器。但也可以通过-s参数修改发出的信号。
  + 因此会发现在docker stop的等过过程中，如果终止docker stop的执行，容器最终没有被关闭。而docker kill几乎是立刻发生，无法撤销。
  + 此外还有些异常原因也会导致容器被关闭，比如docker daemon重启、容器内部进程运行发生错误等等"异常原因"。

#### 容器重启–docker restart

* 作用：

**重启一个或多个处于运行状态、暂停状态、关闭状态或者新建状态的容器**

**该命令相当于stop和start命令的结合**

* 命令格式：

**docker restart [OPTIONS] CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**-t, --time int 重启前，等待的时间，单位秒(默认 10s)**

**实则是关闭前等待的时间**

Usage: docker restart [OPTIONS] CONTAINER [CONTAINER...]

Restart one or more containers

Options:

-t, --time int Seconds to wait for stop before killing the container

(default 10)

#### 容器删除–docker container rm

* 作用：

**删除一个或多个容器**

* 命令格式：

**docker container rm [OPTIONS] CONTAINER [CONTAINER...]**

**或者 docker rm [OPTIONS] CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**-f, --force 强行删除容器(会使用 SIGKILL信号)**

**-v, --volumes 同时删除绑定在容器上的数据卷**

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Options:

-f, --force Force the removal of a running container (uses SIGKILL)

-l, --link Remove the specified link

-v, --volumes Remove the volumes associated with the container

docker rm -h

# 查看所有的容器

docker ps -a

# 同时删除多个容器, 可以指定名称或id进行删除

docker rm centos-test b513

# 可以在容器运行退出的时候自动删除容器

docker run --help

# --rm Automatically remove the container when it exits

docker run -dit --rm centos bash

### 容器信息查看

#### 容器详细信息–docker container inspect

* 作用：

**查看本地一个或多个容器的详细信息**

* 命令格式：

**docker container inspect [OPTIONS] CONTAINER [CONTAINER...]**

或者 **docker inspect [OPTIONS] CONTAINER [CONTAINER...]**

* 命令参数(OPTIONS)：

**-f, --format string 利用特定Go语言的format格式输出结果**

**-s, --size 显示总大小**

docker container inspect 5ab2

docker containder inspect -f "{{json .State.Status}}" 5ab2

docker run -dti centos bash

docker ps -a

docker inspect a35b

# 查看在一行中的信息

docker inspect a35b | grep xxx

# 查看某一个具体的信息

docker inspect a35b -f "{{json .NetworkSettings.Networks.bridge}}"

docker inspect a35b -f "{{json .State}}"

#### 容器日志信息–docker logs

* 作用：

**查看容器的日志信息**

* 命令格式：

**docker logs [OPTIONS] CONTAINER**

* 命令参数(OPTIONS)：

**--details 显示日志的额外信息**

**-f, --follow 动态跟踪显示日志信息. 如果容器一直在运行, 会阻塞在这里, 一直动态显示日志信息**

**--since string 只显示某事时间节点之后的**

**--tail string 显示倒数的行数(默认全部)**

**-t, --timestamps 显示timestamps时间**

**--until string 只显示某事时间节点之前的**

* 注意：

**容器日志中记录的是容器主进程的输出STDOUT\STDERR**

docker logs -h

Usage: docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

Options:

--details Show extra details provided to logs

-f, --follow Follow log output

--since string Show logs since timestamp (e.g.

2013-01-02T13:23:37) or relative (e.g. 42m for 42

minutes)

--tail string Number of lines to show from the end of the logs

(default "all")

-t, --timestamps Show timestamps

--until string Show logs before a timestamp (e.g.

2013-01-02T13:23:37) or relative (e.g. 42m for 42

minutes)

# bash命令执行后没有任何的输出

docker run -dti centos bash

docker ps -a

# 在logs中无任何信息

docker logs 69cb

# python命令

docker run -dti centos python

# 此时logs中就出现了进入python交互环境时的提示信息

docker logs 046d

# ls -a命令

docker run -dti centos ls -a

docker logs 978d

创建容器时指定的命令是容器的主进程, docker logs中看到的信息实际上是容器的COMMAND命令即容器的主进程的输出结果.

由于exec命令是创建子进程并执行, 所以它的输出不会记录到log日志中.

#### 容器重命名–docker rename

* 作用：

**修改容器的名称**

* 命令格式：

**docker rename CONTAINER NEW\_NAME**

* 命令参数(OPTIONS)：

**无**

# 修改docker ps -a 中查看到的NAMES 中显示的信息

docker rename 69cb centos-test3

### 容器运行时操作

例如对于在后台运行的容器, 想要进入到容器中查看容器中正在运行的程序, 或者在容器中再执行其它的命令, 就是容器运行时的操作.

#### 容器连接–docker attach

* **作用：**

Attach local standard input, output, and error streams to a running container

将当前终端的STDIN、STDOUT、STDERR绑定到正在运行的容器的主进程上实现连接

* **命令格式：**

docker attach [OPTIONS] CONTAINER

* **命令参数(OPTIONS)：**

--no-stdin 不绑定STDIN , Do not attach STDIN

docker attach -h

Usage: docker attach [OPTIONS] CONTAINER

Attach local standard input, output, and error streams to a running container

Options:

--detach-keys string Override the key sequence for detaching a

container

--no-stdin Do not attach STDIN

--sig-proxy Proxy all received signals to the process

(default true)

docker run -ti centos

docker start 5332

docker ps -a

# docker attach进入到程序的终端中执行. 因为在使用docker run创建容器时没有指定命令, 使用默认的/bin/bash命令, bash命令会打开一个新的终端, 所以使用attach时就进入到了容器的终端中.

docker attach 5332

# ps -A 查看容器中所有的进程, 1号进程就是bash命令. 13号进程是这里执行的ps命令

ps -A

# 再次查看容器中的所有进程. ps命令变成为14号进程.

ps -A

# 退出容器时, 相当于bash命令的执行结束了, 容器也就会退出了.

exit

# 查看容器, 已经处于退出状态

docker ps -a

# 此时想要再次连接到容器中, 由于容器已经退出, 就会报错, 此时要先启动容器再attach

docer attach 5332

docker start 5332

docker ps -a

docker attach 5332

# docker attach命令是连接到 ps -a 中看到的 COMMAND 命令所在的进程中. 即把当前物理机的终端中的标准输入输出和错误流绑定到容器的pid为1的主进程上.

# 创建执行python命令的容器.

docker run -dti centos python

docker ps -a

# 进入到容器的python交互环境中

docker attach 2b34

# 由于docker attach是连接到command命令所在的进程中, 使用quit, exit或ctrl+D 退出容器的python交互环境, 就会中止容器的运行并退出容器.

quit()

--no-stdin 不绑定stdin, 只绑定stdout和stderr. 此时容器的所有输出都会显示到物理机中, 而物理机的所有输入都不会被容器接收到.

#### 容器中执行新命令–docker exec

* **作用：**

在容器中运行一个新的命令

* **命令格式：**

docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

* **命令参数(OPTIONS)：**

-d, --detach 后台运行命令

-i, --interactive 即使没连接容器，也将当前的STDIN绑定上

-t, --tty 分配一个虚拟终端

-w, --workdir string 指定在容器中的工作目录

-e, --env list 设置容器中运行时的环境变量

docker exec -h

Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Run a command in a running container

Options:

-d, --detach Detached mode: run command in the background

--detach-keys string Override the key sequence for detaching a

container

-e, --env list Set environment variables

-i, --interactive Keep STDIN open even if not attached

--privileged Give extended privileges to the command

-t, --tty Allocate a pseudo-TTY

-u, --user string Username or UID (format:

<name|uid>[:<group|gid>])

-w, --workdir string Working directory inside the container

docker run --dit centos python

docker ps -a

# 执行新的命令, 查看容器中运行的所有进程, python命令是第1号PID进程, 即容器的主进程. 多次执行 ps -A 时ps进程的id号会增大.

docker exec 9309 ps -A

docker exec 9309 ps -A

# 使用exec命令执行一个新的python命令, 但没有出现任何结果, 因为这里的命令相当于在使用docker start命令运行容器时没有指定 -ti命令, 要想进入到交互环境中, 必须要提供终端和标准输出输入

docker exec 9309 python

# 指定-ti参数, 运行新的python命令, 会进入到python的交互环境中

docker exce -ti 9309 python

# 退出python的交互环境, 这时退出的是新执行的python子进程, 不会影响原来1号PID的python进程, 所以不会结束容器的运行. 而attach 命令连接的是1号主进程, 如果使用attach命令连接容器并退出后, 就结束了1号主进程, 整个容器的运行就会停止.

quit()

# 执行新的bash命令, 会进入到容器的bash环境中, 相当于直接在容器中进行操作.

docker exec -ti 9309 bash

# 查看容器中所有的进程, 1号进程依然是python

ps -A

# 多次执行bash, 再查看容器中所有的进程.

bash

bash

ps -A

# 多次使用exit退出bash环境. 就会返回到运行容器的物理机中, 由于不影响1号python主进程, 所以容器的运行并不会结束.

exit

exit

exit

当想要在容器中执行一个新的命令, 但是又不影响容器主进程的运行, 就使用exec来运行新的命令,

当想要查看主进程的运行状态, 并希望我们的任何操作都会影响到主进程的运行, 就应该使用attach命令.

-d 参数 在后台运行新命令.

# 在后台打开一个新的终端

docer exec -dti 9309 bash

# 查看容器中所有的进程, 发现bash命令始终在运行, 不会退出

docker exec 9309 ps -A

# 使用kill结束bash的执行.

docker exec 9309 kill -9 54

-w参数, 指定命令的工作路径.

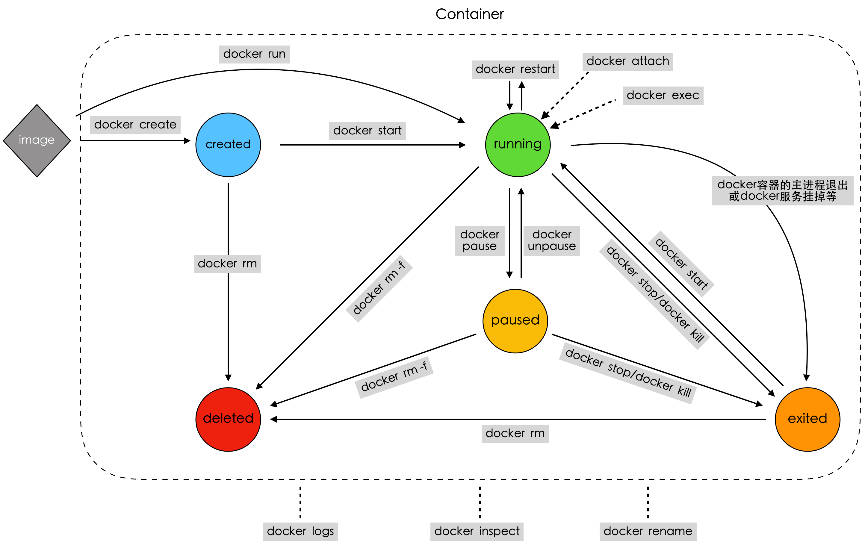
# ls命令默认在根目录下执行

docker exec 9309 ls

# 使用-w参数指定ls命令的执行目录.

docker exec 9309 -w /root/ ls

### 容器总结



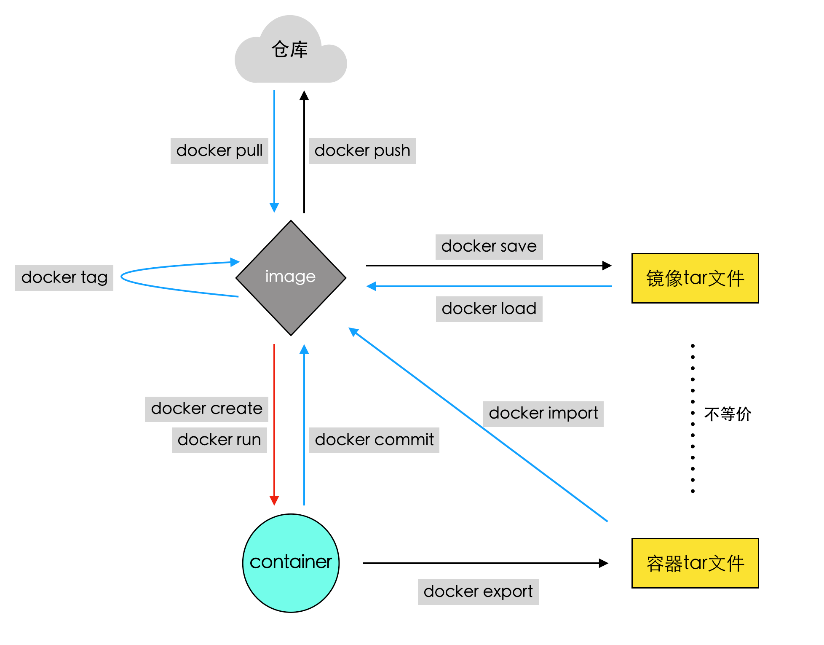
## Docker核心技术之容器与镜像

### 课程概要

* Docker的容器与镜像
* 深入理解Docker的容器与镜像
* 总结

### Docker 容器与镜像

#### Docker容器与镜像的关系



重点关注docker commit命令, 因为后面基于Dockerfile创建镜像时, 使用的就是docker commit命令及容器与镜像之间关系的原理.

#### 容器提交 – docker commit

* **作用：**

根据容器生成一个新的镜像

* **命令格式：**

docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

* **命令参数(OPTIONS)：**

-a, --author string 作者

-c, --change list 为创建的镜像加入Dockerfile命令

-m, --message string 提交信息，类似git commit -m

-p, --pause 提交时暂停容器 (default true)

docker commit --help

Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

Options:

-a, --author string Author (e.g., "John Hannibal Smith

<hannibal@a-team.com>")

-c, --change list Apply Dockerfile instruction to the created image

-m, --message string Commit message

-p, --pause Pause container during commit (default true)

# 生成新的容器

docker run -dti centos bash

# 安装net-tools工具, centos 中使用ifconfig查看网络命令

docker exec 1b15 yum install -y net-tools

# 查看网络

docker exec 1b15 ifconifg

# 把容器打包为镜像.

docker commit -m "install net-tools" 1b15 centos-net:v1.0

# 使用生成的镜像创建容器

docker run -dti 001a bash

# 可以直接运行ifconfig命令. 把容器保存为镜像会保存之前的容器中所有的内容.

dockker exec f92b ifconfig

#### 容器导出 – docker export

* **作用：**

将容器当前的文件系统导出成一个tar文件. image save的tar文件与container export 的tar文件不同. image save load是一对命令, container export import 是一对命令.

* **命令格式：**

docker export [OPTIONS] CONTAINER

* **命令参数(OPTIONS)：**

-o, --output string 指定写入的文件，默认是STDOUT

docker export --help

Usage: docker export [OPTIONS] CONTAINER

Export a container's filesystem as a tar archive

Options:

-o, --output string Write to a file, instead of STDOUT

docker run -dti centos bash

docker exec 36c8 yum -y install net-tools

docker exec 36c8 ifconfig

# 会把容器打包为一个tar文件, 打包过程中容器依然处于运行过程中.

docker export -o net-tools-image.tar 35c8

#### 容器打包的导入 – docker import

* **作用：**

从一个tar文件中导入内容创建一个镜像

* **命令格式：**

docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]

* **命令参数(OPTIONS)：**

-c, --change list 为创建的镜像加入Dockerfile命令

-m, --message string 导入时，添加提交信息

docker import --help

Usage: docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]

Import the contents from a tarball to create a filesystem image

Options:

-c, --change list Apply Dockerfile instruction to the created image

-m, --message string Set commit message for imported image

docker import -m '[import] install net-tools' net-tools-image.tar centos-net2:v1.0

### 深入理解Docker容器与镜像

docker commit 与 docker import 创建的镜像的区别

docker images

容器 commit > 镜像

容器 export > tar > import > 镜像

# 查看使用commit保存的镜像的原始镜像centos的历史记录.

docker history centos

# 查看使用commit保存的镜像的历史记录, 会继承cenots镜像的所有记录, 并且记录了在原来镜像基础上的变更. 能够知道是基于哪个镜像创建的, 且知道在原有的镜像的基础上进行了哪些变更.

docker histroy centos-net:v1.0

# 查看使用export - import保存的镜像的历史记录. 把以前的记录都合并, 只保存最后一层镜像. 相当于重新生成了一个全新的镜像, 看不出原来的镜像历史了.

docker history centos-net2:v1.0

# 查看centos镜像的详细信息

docker inspect centos

# 查看centos镜像的config信息.

docker inspect -f "{{json .Config}}" centos

# 查看centos-net:v1.0的详细信息.

docker inspect centos-net:v1.0

# 查看centos-net:v1.0镜像的config信息. 其中多了一个Hostname的字段, 字段值为上一个生成它的容器的id值. 而Config.Cmd也由原来的 "/bin/bash" 变成了 "bash", 这是在创建容器是指定的命令. 即commit保存的镜像会把原来镜像的命令修改为创建容器时指定的命令.

docker inspect -f "{{json .Config}}" centos-net:v1.0

# 查看centos-net2:v1.0的详细信息

docker inspect centos-net2:v1.0

# 查看centos-net2:v1.0镜像的Config信息, 发现其中所有的信息都丢失了. 即使用import export方式创建的镜像会丢失原来父镜像的所有信息即原数据. 而使用commit方法创建的镜像则会保存父镜像的所有数据, 并且可以在原数据的基础上重写一部分信息.

#### 镜像的Layer

# 查看原centos 镜像RootFS.Layers中的信息. 其中只包含了一层.

docker inspect -f "{{json .RootFS.Layers}}" centos

# 查看使用commit方法生成的镜像中RootFS.Layers的信息. 其中包含了2层. 会把对所进行的更改写入到 Layers 层中, 对应于 history 中大小变更的信息.

docker inspect -f "{{json .RootFS.Layers}}" centos-net:v1.0

# 再次查看centos镜像和centos-net:v1.0的详细信息, 在RootFs.Layers中可以看到一层层的信息. centos镜像的信息只有一层, centos-net:v1.0的信息有2层.

# 查看centos镜像的历史信息, 其中有3层信息, 但Layers只有1层, 在3层历史信息中有2层的Size都为0B, 只有1层的Size有实际的大小. 所以只有在对原来的镜像创建出来的容器进行更改后, 再commit保存的镜像才会生成新的一层.

# 基于centos-net:v1.0创建一个新的容器, 什么也不做更改就把它保存为一个新的镜像, 再查看它的历史信息和详细信息, layer中并没有新增一层.

docker history centos

docker run -dti centos-net:v1.0 bash

docker commit -m 'noting to do' a8fa centos-net:v2.0

docker history centos-net:v2.0

docker inspect centos-net:v2.0

# 基于centos-net:v1.0再创建一个容器, 在其中进行更改, 再保存为镜像, 查看它的历史信息和详情信息.

docker run -dti centos-net:v1.0 touch a.txt

docker commit -m 'touch empty file' bd68 centos-net:v3.0

# 查看centos-net:v3.0的历史信息, 其中touch empty file的Size依旧为0B.

docker history centos-net:v3.0

# 查看centos-net:v3.0的详细信息, 在layer中新增加了一层. 也就是说, 只要对容器的文件系统进行了修改, 哪怕是创建了一个空白文件, 再基于容器创建镜像时layer中都会新增加一层. 而仅仅是执行了如ps, ls这些不会对系统进行修改的操作, 就不会增加新的一层.

docker inspect centos-net:v3.0

一个系统中所有的文件和目录加一起就是一个文件系统. 镜像的每一层都是一个文件系统, 每一层文件系统都是只读的. 每一层的文件系统都是通过一定的方式结合在一起的, 所有层的文件系统统称为联合文件系统.

镜像的每一层都通过一个指针与它底层的镜像进行着关联, 如果某一层没有了指针, 它就是最底层的镜像层了.



#### 镜像的视角

对于用户来说, 只能查看到镜像的最上层的一层, 也就是只能看到最新层的信息, 只能看到各层文件系统结合在一起成为一个整体的只读的文件系统的信息, 这就是联合文件系统的含义. 这里的信息也就是使用docker images查看镜像信息时看到的那一层信息. 而使用docker inspect查看镜像的详情信息, 就会看到镜像底层的每一层文件系统.



#### 容器的Layer

每一层镜像都是一个只读层, 想要进行修改, 就会创建一个新的层, 在新的层中保存用户的修改.

docker create 命令的作用就是在原来的只读层上创建出了一个可读可写层的文件系统. 如果是使用ls, ps这样的命令, 不会对文件系统进行修改, 就不会创建新层. 而如果是使用touch, rm等命令新建或删除了一个文件, 就会对文件系统进行修改, 就会创建新的一层.

而docker commit 命令的作用就是把docker create命令创建的可读可写层的文件系统变为只读层. 这就是用户在使用commit命令把容器保存为镜像时添加一个新层的原因. 而新添加的一层与原来已经存在的层结合在一起就成为一个新的镜像, 这也是 "联合文件系统" 的含义.





#### 容器的视角

使用docker ps -A 只能看到容器最上层的可读可写层的信息, 无法看到容器底层不能被修改的文件层的信息, 这是因为容器的修改和运行都是基于docker create命令创建的新的可读可写层进行的.

而对于镜像, 则可以使用docker inspect 查看镜像的详细信息.



#### 容器与镜像的底层关系



第4层的容器是在前3层叠加的基础上创建出来的, 是前面3层文件系统叠加的结果.

#### 容器的运行

通过docker create命令只是创建出了一个可读可写的文件系统.

使用docker start命令则会基于原来镜像的文件系统创建出一个进程空间, 在这个进程空间中就可以运行命令. 运行命令时消耗硬件资源, 并可能会对文件系统进行更改. 一旦对文件系统进行了修改, 使用docker commit命令由容器生成镜像时就会生成新的一层只读层, 成为镜像新的一层.





### 总结

#### Docker核心技术之容器与镜像- 总结



## Docker核心技术之网络管理

### 课程概要

* Docker 网络管理简介
* Docker 网络管理命令
* Docker 网络模式简介
* 总结

### Docker 网络管理简介

#### 为什么需要Docker网络管理

docker的命名空间, 它的一个功能就是网络隔离的功能, 使容器的网络默认与宿主机、与其他容器之间相互隔离。

* 容器中可以运行一些网络应用(如nginx、web应用、数据库等)，如果要让外部也可以访问这些容器内运行的网络应用，那么就需要配置网络来实现。
* 有可能有的需求下，容器不想让它的网络与宿主机、与其他容器隔离。
* 有可能有的需求下，容器根本不需要网络。计算, 加密型容器.
* 有可能有的需求下，容器需要更高的定制化网络（如定制特殊的集群网络、定制容器间的局域网）。多台nginx提供相同的功能.
* 有可能有的需求下， 容器数量特别多，体量很大的一系列容器的网络管理如何
* ……

因此容器的网络管理是非常重要的

#### Docker中有哪些网络驱动模式

Docker有五种网络驱动模式

* bridge network 模式（网桥）：默认的网络模式。类似虚拟机的nat模式
* host network 模式（主机）：容器与宿主机之间的网络无隔离，即容器直接使用宿主机网络
* None network 模式：容器禁用所有网络。
* Overlay network 模式（覆盖网络）： 利用VXLAN实现的bridge模式
* Macvlan network 模式：每个容器具备单独的Mac地址，使其显示为网络上的物理设备

### Docker网络管理命令

docker network --help

Usage: docker network COMMAND

Manage networks

Commands:

connect Connect a container to a network

create Create a network

disconnect Disconnect a container from a network

inspect Display detailed information on one or more networks

ls List networks

prune Remove all unused networks

rm Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.

#### 查看网络 – docker network ls

* 作用：

**查看已经建立的网络对象**

* 命令格式：

**docker network ls [OPTIONS]**

* 命令参数(OPTIONS)：

**-f, --filter filter 过滤条件(如 'driver=bridge')**

**--format string 格式化打印结果**

**--no-trunc 不缩略显示**

**-q, --quiet 只显示网络对象的ID**

* 注意：

**默认情况下，docker安装完成后，会自动创建bridge、host、none三种网络驱动**

docker network ls --help

Usage: docker network ls [OPTIONS]

List networks

Aliases:

ls, list

Options:

-f, --filter filter Provide filter values (e.g. 'driver=bridge')

--format string Pretty-print networks using a Go template

--no-trunc Do not truncate the output

-q, --quiet Only display network IDs

# 默认有3个网络对象, driver表示的是网络的类型, 默认是bridge, 即桥接网络.

# SCOPE表示网络的使用范围, 默认是locol, 即在宿主机上使用的. overlay 模式的网络驱动对应的SCOPE是swarm, 这种模式可以实现跨主机的集群服务

docker network ls

NETWORK ID NAME DRIVER SCOPE

485862f7a662 bridge bridge local

8803a05bf8a3 host host local

f9f8192753e1 none null local

# 筛选某种特定模式的网络

docker network ls -f "driver=brideg"

#### 创建网络 – docker network create

* 作用：

**创建新的网络对象**

* 命令格式：

**docker network create [OPTIONS] NETWORK**

* 命令参数(OPTIONS)：

**-d, --driver string 指定网络的驱动, 即网络模式. (默认"bridge")**

**--subnet strings 指定子网网段(如192.168.0.0/16、172.88.0.0/24)**

**--ip-range strings 执行容器的IP范围，格式同subnet参数**

**--gateway strings 子网的IPv4 or IPv6网关，如(192.168.0.1)**

* 注意：

**host和none模式网络只能存在一个**

**docker自带的overlay 网络创建依赖于docker swarm(集群负载均衡)服务**

**subnet子网网段.** 在使用网络时, 由路由器进行ip地址的分配, 路由器有一个 192.168.0.1或192.168.1.1的网段, 有两种范围的网络: /16, /24, 例如

**192.168.0.0/16 等于 192.168.0.0~192.168.255.255**

**172.88.0.0/24 等于 172.88.0.0~172.88.0.255**

**ip-range 容器的IP范围.** 创建的网络是给容器使用的, 容器使用某种网络时, 对应的容器的ip地址必须是该网络的subnet的子集. 也就是ip地址必须要在子网网段中. 例如, 如果子网网段为 192.168.0.0/16, ip-range 必须要取在 192.168.0.0~192.168.255.255 之间

**网关**, 相当于路由器自身的ip地址. 连接路由器的设备则通过网关的ip地址进行通信.

docker network create --help

Usage: docker network create [OPTIONS] NETWORK

Create a network

Options:

--attachable Enable manual container attachment

--aux-address map Auxiliary IPv4 or IPv6 addresses used by

Network driver (default map[])

--config-from string The network from which copying the configuration

--config-only Create a configuration only network

-d, --driver string Driver to manage the Network (default "bridge")

--gateway strings IPv4 or IPv6 Gateway for the master subnet

--ingress Create swarm routing-mesh network

--internal Restrict external access to the network

--ip-range strings Allocate container ip from a sub-range

--ipam-driver string IP Address Management Driver (default "default")

--ipam-opt map Set IPAM driver specific options (default map[])

--ipv6 Enable IPv6 networking

--label list Set metadata on a network

-o, --opt map Set driver specific options (default map[])

--scope string Control the network's scope

--subnet strings Subnet in CIDR format that represents a

network segment

# 使用-d指定网络模式, 一共有5种网络模式. 如果是要创建bridge模式的网络, 可以省略 -d bridge参数. NETWORK参数即为网络的名称, network ls中的NAME字段. 名称不能重复.

docker network create -d bridge my-bridge

# host和null网络模式只能存在一个. 因为在docker安装后已经自动创建了一个host和null的网络模式, 所以再创建这两种网络时就会出错.

docker network create -d host my-host

docker network create -d null my-host

# overlay模式需要启动集群**swarm**支持. 如果没有开启, 就会创建失败. swarm是docker中集群的管理服务.

docker network create -d overlay my-overlay

# Error response from daemon: This node is not a swarm manager. Use "docker swarm init" or "docker swarm join" to connect this node to swarm and try again.

# macvlan网络模式, 默认不会创建, 与bridge模式类似, 可以创建多个.

docker create -d macvlan my-macvlan

#### 网络删除 – docker network rm

* 作用：

**删除一个或多个网络**

* 命令格式：

**docker network rm NETWORK [NETWORK...]**

* 命令参数(OPTIONS)：

**无**

docker network rm --help

Usage: docker network rm NETWORK [NETWORK...]

Remove one or more networks

Aliases:

rm, remove

#### 查看网络详细信息 – docker network inspect

* 作用：

**查看一个或多个网络的详细信息**

* 命令格式：

**docker network inspect [OPTIONS] NETWORK [NETWORK...]**

或者 **docker inspect [OPTIONS] NETWORK [NETWORK...]**

* 命令参数(OPTIONS)：

**-f, --format string 根据format输出结果, 对结果进行筛选**

docker network inspect --help

Usage: docker network inspect [OPTIONS] NETWORK [NETWORK...]

Display detailed information on one or more networks

Options:

-f, --format string Format the output using the given Go template

-v, --verbose Verbose output for diagnostics

docker inspect --help

Usage: docker inspect [OPTIONS] NAME|ID [NAME|ID...]

Return low-level information on Docker objects

Options:

-f, --format string Format the output using the given Go template

-s, --size Display total file sizes if the type is container

--type string Return JSON for specified type

docker network inspect bridge

docker inspect bridge

docker inspect -f "{{json. IPAM.Config}}" bridge

docker inspect image

docker inspect container

docker inspect network

#### 使用网络 – docker run --network

* 作用：

**为启动的容器指定网络模式**

* 命令格式：

**docker run/create --network NETWORK**

* 命令参数(OPTIONS)：

**无**

* 注意：

**默认情况下，docker创建或启动容器时，会默认使用名为bridge的网络**

# 网络必须要使用在容器上, 所以需要在 create 或 run 命令时加上 --network的参数.

docker run --network my-bridge -dti centos bash

# 默认使用名称 NAME 为bridge的网络, 以下两种写法效果相同.

docker run --network bridge --dit centos bash

docker run -dit centos bash

# 创建使用host网络模式的容器

docker run --network host -dti centos bash

# a231

# host和bridge网络模式默认可以连接外网

docker exec a231 ping [www.baidu.com](http://www.baidu.com)

# 创建none网络模式的容器

docker run --network none --dti centos bash

# a370

# none模式是没有网络的, 所以无法ping通.

docker exec a370 ping [www.baidu.com](http://www.baidu.com)

#### 网络连接与断开 – docker network connect/disconnect

* 作用：

**将指定容器与指定网络进行连接或者断开连接**

* 命令格式：

**docker network connect [OPTIONS] NETWORK CONTAINER**

**docker network disconnect [OPTIONS] NETWORK CONTAINER**

* 命令参数(OPTIONS)：

**-f, --force 强制断开连接(用于disconnect)**

docker network connect --help

Usage: docker network connect [OPTIONS] NETWORK CONTAINER

Connect a container to a network

Options:

--alias strings Add network-scoped alias for the container

--ip string IPv4 address (e.g., 172.30.100.104)

--ip6 string IPv6 address (e.g., 2001:db8::33)

--link list Add link to another container

--link-local-ip strings Add a link-local address for the container

docker network disconnect --help

Usage: docker network disconnect [OPTIONS] NETWORK CONTAINER

Disconnect a container from a network

Options:

-f, --force Force the container to disconnect from a network

# 使用默认的bridge网络模式来创建容器

docker run -dti centos

# 查看创建的容器

docker inspect 1728

# 查看创建的容器的网络信息, 使用的是bridge的网络

docker inspect -f "{{json .NetworkSettings.Networks}}" 1728

# 断开网络连接

docker network disconnect bridge 1728

# 再次查看创建的容器的网络信息, Networks中的信息就为空了

docker inspect -f "{{json .NetworkSettings.Networks}}" 1728

# 把容器与网络进行连接

docker network connect bridge 1728

注意 **不能** 把容器连接到host网络模式的网络上, 只能在创建容器时指定使用host网络模式. 可以把容器连接到none网络上.

# none网络不能与其它类型的网络同时连接到同一个容器上. 例如none网络和bridge网络不能同时连接到同一个容器上, 必须先断开一种网络, 才能再连接另一种网络.

# host网络不能连接或断开连接. 同一个容器可以连接多个bridge网络. 一个容器可以同时连接macvlan和bridge网络上, 并且可以同时连接到多个macvlan网络上. overlay不能进行connect与disconnect, 并且不能与其它网络同时连接.

### Docker网络模式简介

#### bridge 网络模式（一）



特点：

* 宿主机上需要单独的bridge网卡，如默认docker默认创建的docker0。
* 容器之间、容器与主机之间的网络通信，是借助为每一个容器生成的一对veth pair虚拟网络设备对，进行通信的。一个在容器上，另一个在宿主机上。
* 每创建一个基于bridge网络的容器，都会自动在宿主机上创建一个veth\*\*虚拟网络设备。
* 外部无法直接访问容器。需要建立**端口映射**才能访问。
* 容器借由veth虚拟设备通过如docker0这种bridge网络设备进行通信。
* 每一容器具有单独的IP

在宿主机上执行ifconfig, 查看docker0网卡的ip地址. docker0是安装好docker之后在宿主机上默认安装的docker虚拟网卡. 与docker inspect bridge中的Config.Gateway即网关中的地址是相同的. 即docker0与bridge网络是存在着关联的.

ifconfig

docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255

ether 02:42:e7:7e:eb:f9 txqueuelen 0 (Ethernet)

RX packets 0 bytes 0 (0.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

docker inspect bridge -f "{{json .IPAM.Config}}"

[{"Subnet":"172.17.0.0/16","Gateway":"172.17.0.1"}]

# 创建一个新的bridage网卡, 省略 -d bridge 参数就默认创建bridge网卡.

docker network create my-bridge

# 再次使用ifconfig查看宿主机中的网络信息. 会新增加一个网卡, ip地址与my-bridge网卡的Gateway网关中的地址是相同的.

docker inspect my-bridge -f "{{json .IPAM.Config}}"

[{"Subnet":"172.18.0.0/16","Gateway":"172.18.0.1"}]

ifconfig

br-a110a12d4296: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500

inet 172.18.0.1 netmask 255.255.0.0 broadcast 172.18.255.255

ether 02:42:1a:00:50:ee txqueuelen 0 (Ethernet)

RX packets 0 bytes 0 (0.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

规则: 每次手动创建一个新的bridge网络, 就会在宿主机中创建一个新的以 "br-" 开头的随机名称的docker虚拟网卡, 作为桥梁来完成宿主机与容器, 容器与容器之间的通信.

# 基于my-bridge创建一个新的容器.

docker run -dti --network my-bridge centos

# 使用ifconfig查看宿主机的网络信息, 会多出来一个veth\*\*\* 的网卡. 每创建并运行一个容器就会在宿主机中创建一个虚拟的网卡.

docker中基于命名空间来完成容器与容器之间的隔离. 容器与容器之间的通信要通过veth pair虚拟网络设备对来完成. 每一对veth pair对都会分别在容器中和宿主机中创建对应的虚拟网卡并一一对应来完成通信.

每创建出一个bridge网络, 在宿主机中就会创建出来一个veth\*\*\* 的虚拟网络设备. 而容器中则会创建出一个eth\* 的虚拟网卡, 在通信时, 容器先把数据传递给容器中的虚拟网络设备, 默认为eth0, 再由容器的虚拟网络设备传递给宿主机中对应的虚拟网络设备如veth1, veth1再把数据传递给宿主机中的docker0, 如果容器是与宿主机进行通信, docker0就把数据转发给宿主机的物理网卡eth0, 如果容器是与其它容器进行通信, docker0就会把数据经过veth\* 转发给其它容器的eth0虚拟网络设备, 从而完成通信.

# 查看容器中的虚拟网络设备. 在创建的容器中运行新的命令来安装net-tools工具

docker exec acd0 yum install -y net-tools

# 在容器中运行ifconfig 命令来查看容器的网络信息

docker exec acd0 ifconfig

每创建出一个容器并指定它的桥接网络时, 会自动创建 veth\*-eth\* veth对, 来完成容器与宿主机, 容器与容器之间的通信. 创建veth\*-eth\*对的工作是自动完成的, 我们只需要指定使用的是哪个桥接网络 (docker0) 就可以了. docker0是安装完docker之后自动创建的bridge网络, 还可以使用docker network create -d bridge my-bridge来手动创建新的桥接网络, 并且手动指定 subnet子网网段, ip-range ip范围, gateway 网关.

# 查看bridge网络的网络参数.

docker inspect -f "{{json .IPAM.Config}}" bridge

# [{"Subnet":"172.17.0.0/16","Gateway":"172.17.0.1"}]

# Subnet子网网段为172.17.0.0/16, 即在 172.17.0.0 - 172.17.255.255之间, 如果在手动创建bridge网络时没有指定ip-range, 它的范围就等同于子网网段的范围. 如果要手动设置ip-range, 它的范围必须要包含在子网网段中, 是子网网段的子集. 网关gateway的ip地址也必须在子网网段中.

# 自定义参数创建网络, 参数不能与现有的网络冲突.

docker network create -d bridge --subnet 192.168.0.0/16 --ip-range 192.168.8.0/24 --gateway 192.168.8.250 custom-bridge

# 查看创建的网络, IPAM.Config中的内容就是上面手动设置的.

docker inspect custom-bridge

# 使用创建的自定义bridge网络来创建容器

docker run -dti --network custom-bridge centos

# 查看容器的信息, IPAddress为 "192.168.8.0", Gateway为 "192.168.8.254"

docker inspect 152b

#### bridge 网络模式（二） – 端口映射

* 作用：

**启动的容器时，为容器进行端口映射**

* 命令格式：

**docker run/create -P …**

或者 **docker run/create –p …**

* 命令参数(OPTIONS)：

**-P, --publish-all 将容器内部所有暴露端口进行随机映射, 不用指定端口.**

**容器内暴露端口的概念:** 对于数据库的镜像, 如redis, mysql, 数据库镜像创建的容器要对外提供服务, 就会默认开启并暴露一定的端口上. 如redis, 在运行基于它创建的容器时会自动运行服务并把redis服务绑定到3306端口上, 通过ip:port就能够访问到该服务. 而对于如ubuntu, centos的系统镜像, 默认情况下不会开启并暴露端口. 随机映射, 由系统自动把宿主机中的未被占用的端口绑定到容器中暴露的端口上.

**-p, --publish list 手动指定端口映射**

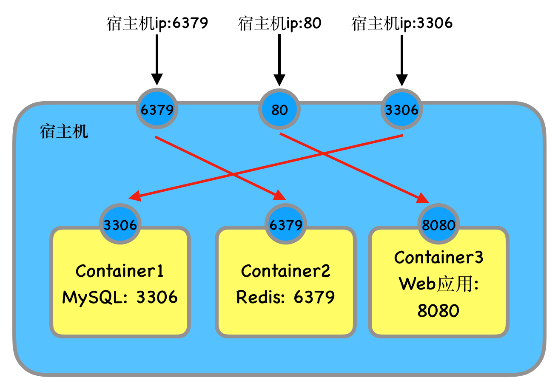
* 注意：

**-p [HOST\_IP]:[HOST\_PORT]:CONTAINER\_PORT**

**如：-p ::80 将容器的80端口随机(端口)映射到宿主机任意IP**

**-p :8000:6379 将容器的6379端口映射到宿主机任意IP的8000端口**

**-p 192.168.5.1::3306 将容器的3306端口随机(端口)映射到宿主机的192.168.5.1IP上**



连接到同一个bridge网络上的容器的ip地址是不重复的, 不能冲突. 在一个容器如container1上开通的服务, 如web服务或数据库服务, 是在一定的端口上运行的, 其它容器中的服务通过ip:port的方式就能访问到这个容器上的web服务或数据库服务. 同样, 宿主机也可以通过ip:port访问到对应的服务.

但是 127.17.0.2 是一个局域网ip地址, 只能在宿主机内部访问到, 如果想要在宿主机外部访问到容器中启动的服务, 就要进行端口映射.

端口映射: 把宿主机上的某个端口映射到容器中的某个端口上, 如把宿主机的6379端口映射到容器2的6379端口上, 在外部通过 "宿主机ip:6379" 访问时, 宿主机的6379端口就会接收到外部发来的数据, 然后把数据完整转发到容器2的6379端口上. 数据处理之后就会把处理后的数据原路返回给宿主机, 进而返回给外部访问者.

宿主机未被占用的端口都可以用来进行端口映射. 但一般情况下还是使用与某个功能相同的端口号进行映射.

# 基于redis创建容器并自动进行端口映射, 将容器内部所有暴露端口进行随机映射.

docker run -dti -P redis

# 查看所有的容器, 在PORTS列中就会看到映射的ip地址端口号. 0.0.0.0表示任意ip地址都可以访问, 32768->6379 表示把宿主机的32768端口映射到了容器中的6379端口.

docker ps -a --no-trunc

0331c0d16b0cc64acb171485926cd174dc289e46ff5c8a2bb7e121e4529f038c redis "docker-entrypoint.sh redis-server" About a minute ago Up About a minute 0.0.0.0:32768->6379/tcp vibrant\_hodgkin

# 再次基于redis创建容器, 自动进行端口映射并查看所有容器

docker run -dti -P redis

docker ps -a --no-trunc

2c7b9b9456a3e906a9f252c7d0383a9fcf6b62205786e5d9e533717e74821fc1 redis "docker-entrypoint.sh redis-server" 5 seconds ago Up 4 seconds 0.0.0.0:32769->6379/tcp practical\_lederberg

# 基于centos 创建容器, 自动进行端口映射

docker run -dti -P centos

# 查看所有的容器, 因为centos默认没有暴露端口, 所以这里也不会进行端口的绑定.

docker ps -a --no-trunc

5feeee3aad9c28dbf0fc45e910186ca8c47d0e144f24ef998e9f0fdd774d1882 centos "/bin/bash" 29 seconds ago Up 28 seconds friendly\_bell

**-p [HOST\_IP]:[HOST\_PORT]:CONTAINER\_PORT**

# 创建新的容器, 只指定宿主机的端口

docker -dti -p :6379:6379 redis

# 查看新创建的容器的端口映射情况. 省略宿主机的ip地址表示可以通过任意ip地址进行访问.

docker ps -a

# 创建容器并手动指定端口映射, 宿主机的ip和端口可以省略, 省略时宿主机的ip时是所有的ip地址都可以进行映射, 省略宿主机的端口时表示随机分配端口进行映射. 指定宿主机的ip地址为127.0.0.1表示只能通过宿主机上的127.0.0.1:6379来访问容器中的服务.

docker -dti -p 127.0.0.1:6378:6379 redis

# 查看新创建的容器的端口映射情况.

docker ps -a

# 可以指定某个网卡对应的ip地址, 如eth0物理网卡的地址是 10.211.55.7, 可以指定通过这个ip地址来进行端口映射.

docker -dti -p 10.211.55.7:6377:6379 redis

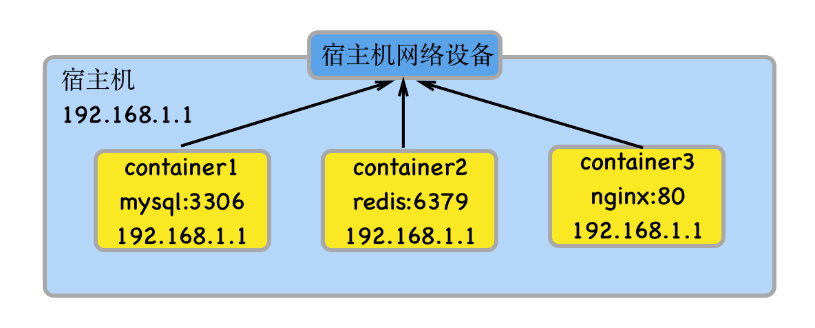
# 现在在宿主机外部就可以通过 10.211.55.7:6377来访问容器中的redis服务.

#### host 网络模式

bridge网络模式下, 容器和宿主机之间的网络是进行隔离的, 容器和宿主机之间的网络是不同的网络, 容器和宿主机都有自己的网卡, 它们之间的通信是对应的网卡之间的通信.

host网络模式的特点：

* 容器完全共享宿主机的网络。网络没有隔离。每个宿主机的网络就是容器的网络, 容器的网络与宿主机的网络完全相同。
* 容器、主机上的应用所使用的端口不能重复。外部访问时既是访问宿主机又是访问容器, 此时就要通过端口来对访问目标进行区分. 例如：如果宿主机已经占用了8090端口，那么任何一个host模式的容器都不可以使用8090端口了；反之同理。
* 外部可以直接访问容器，不需要端口映射。对比bridge网络模式, host省略了不同网卡之间进行数据交换的过程, 它的网络性能是最优的. 在容器数量比较少或者容器和宿主机之间的端口不存在冲突的情况下, 就可以使用host网络模式.
* 容器的IP就是宿主机的IP
* 可能的弊端: 外部可以直接访问容器中的应用, 可能会丧失一部分的安全性.



# 基于host网络模式创建容器

docker run --network host -dti centos

docker run --network host -dti redis

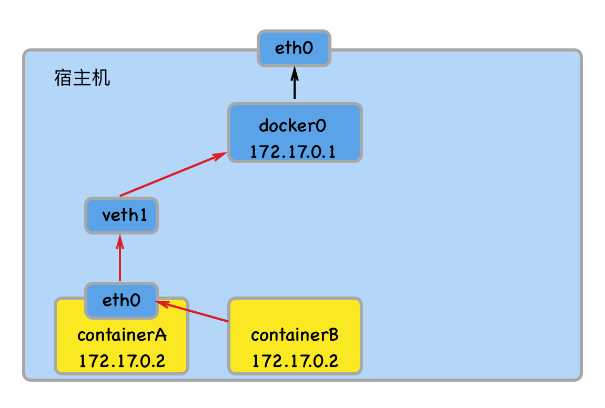
#### 特殊 host 网络模式（Container网络模式）

* Container网络模式，其实就是容器共享其他容器的网络。
* 相当于该容器,，在网络层面上，将其他容器作为"主机"。它们之间的网络没有隔离。容器与容器之间的网络没有隔离.
* 这些容器之间的特性同host模式。

使用方法：

**Docker run/create --network container:CONTAINER …**

**Docker run/create --network container:容器名或容器id …**



# 可以基于host或bridge网络模式创建容器

docker run --network host -dti centos

# 662b0d98f213 centos "/bin/bash" 23 seconds ago Up 22 seconds agitated\_lehmann

docker run --network host -dti redis

# f33732f33746 redis "docker-entrypoint.s…" 2 seconds ago Up 1 second nifty\_taussig

# 基于已有的centos容器创建container模式的容器

docker run -dti --network container:662b centos

# 基于已有的redis容器创建container模式的容器

docker run -dti --network container:662b redis

# 查看容器的运行状态. 因为在创建基于host模式的redis容器时, redis的6379端口已经被占用了, 此时基于4565容器的网络创建出来的redis容器的端口就冲突了, 就不能正常启动新创建的容器了.

docker ps -a

# a29d79e6def3 redis "docker-entrypoint.s…" 1 second ag Exited (1) 1 second ago frosty\_mclean

#### none 网络模式

特点：

* 容器上没有网络，也无任何网络设备。
* 如果需要使用网络，需要用户自行安装与配置。

应用场景

* 该模式适合需要高度定制网络的用户使用。如定制mac地址, 就需要先创建为none网络模式, 再进行个性化的定制和修改.
* 容器中只执行一些不需要网络的计算型的任务, 如加密型的运算.

#### overlay 网络模式（一）

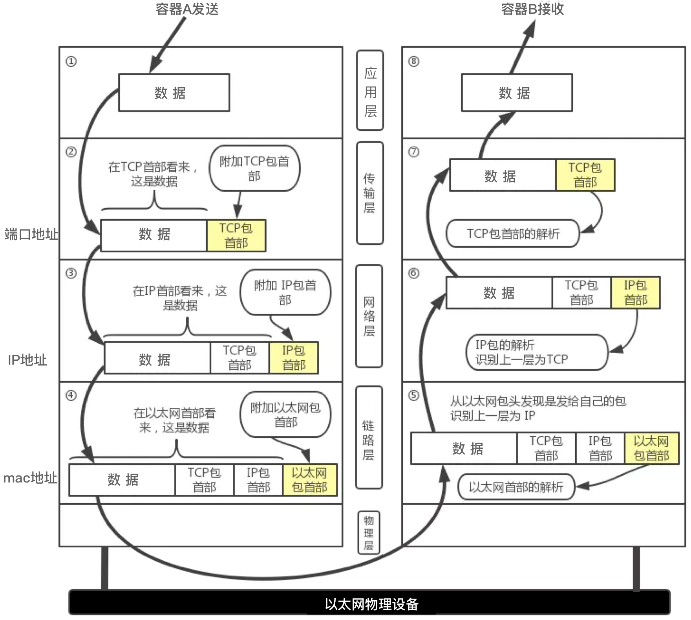
特点：

* Overlay 网络，也称为覆盖网络。
* Overlay 网络的实现方式和方案有多种。Docker自身集成了一种，基于VXLAN隧道技术实现。
* Overlay 网络主要用于实现跨主机容器之间的通信。

应用场景：需要管理成百上千个跨主机的容器集群的网络时。

跨主机的容器之间的通信, 通过bridge的端口映射和host网络模式都能实现. 但成百上千个容器时, bridge网络模式需要进行无数的端口映射, 后期管理起来也非常麻烦, host网络模式会占用很多宿主机的端口.

#### overlay 网络模式（二）- 了解TCP/IP协议栈



#### overlay 网络模式（三）- 实现原理

##### IP隧道网络原理





##### macvlan 网络模式

* macvlan网络模式，最主要的特征就是他们的通信会直接基于mac地址进行转发。
* 这时宿主机其实充当一个二层交换机。Docker会维护着一个MAC地址表，当宿主机网络收到一个数据包后，直接根据mac地址找到对应的容器，再把数据交给对应的容器。
* 容器之间可以直接通过IP互通，通过宿主机上内建的虚拟网络设备（创建macvlan网络时自动创建），但与主机无法直接利用IP互通。

应用场景：由于每个外来的数据包的目的mac地址就是容器的mac地址，这时每个容器对于外面网络来说就相当于一个真实的物理网络设备。因此当需要让容器来的网络看起来是一个真实的物理机时，使用macvlan模式



### Docker核心技术之网络管理 - 总结

重点掌握：

* bridge网络、host网络、Container网络模式的原理和使用（应用得较多，且host网络性能最优）。
* docker network命令的使用

了解：

* none网络的效果
* overlay网络、macvlan网络的原理。（使用起来难度较大）

## Docker核心技术之数据卷管理

### 课程概要

* Docker 数据卷简介
* Docker 数据卷管理
* Docker 数据卷注意事项
* 总结

### Docker 数据卷简介

为什么用数据卷

* 宿主机无法直接访问容器中的文件
* 容器中的文件没有持久化，导致容器删除后，文件数据也随之消失
* 容器之间也无法直接访问互相的文件. 多个数据库使用同一个文件. 不同的应用使用同一份配置文件.

为解决这些问题，docker加入了数据卷(volumes)机制，能很好解决上面问题，以实现：

* 容器与主机之间、容器与容器之间共享文件
* 容器中数据的持久化
* 将容器中的数据备份、迁移、恢复等

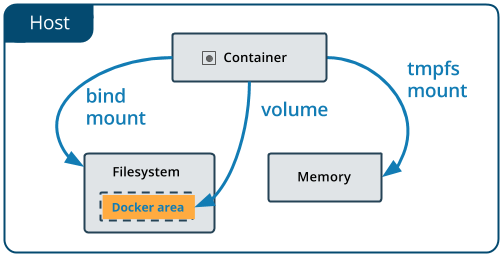
### 数据卷的特点

* 数据卷存在于宿主机的文件系统中，独立于容器，和容器的生命周期是分离的。
* 数据卷可以目录也可以是文件，容器可以利用数据卷与宿主机进行数据共享，实现了容器间的数据共享和交换。
* 容器启动初始化时，如果容器使用的镜像包含了数据，这些数据会拷贝到数据卷中。
* 容器对数据卷的修改是实时进行的。
* 数据卷的变化不会影响镜像的更新。数据卷是独立于联合文件系统，镜像是基于联合文件系统。镜像与数据卷之间不会有相互影响。

### Docker 数据卷管理

#### Docker挂载容器数据卷的三种方式

* **bind mounts**：将宿主机上的一个文件或目录挂载到容器上。相当于对文件或目录进行映射, 把文件或目录从宿主机映射到容器中的某个目录中.
* **volumes**：由Docker创建和管理。使用docker volume命令管理, 相当于是在bind mounts的基础上进行了进一步的封装.
* **tmpfs mounts**：tmpfs 是一种基于内存的临时文件系统。tmpfs mounts 数据不会存储在磁盘上。



#### bind mounts方式挂载数据卷

* 利用docker run/create的参数为容器挂载数据卷
* 用法：

方式一： -v, --volume参数

**-v 宿主机文件或文件夹路径:容器中的文件或者文件夹路径**

# 创建容器, 并把宿主机上的/root/volume\_dir映射到容器中的/root/c\_dir目录中

docker run -dti -v /root/volume\_dir:/root/c\_dir centos

# 会在当前/root目录中会自动创建一个volume\_dir的目录.

ls

# 此时volume\_dir下没有文件

cd volume\_dir

ls

# 在容器中的/root/c\_dir中创建新文件

docker exec 8bb8 touch /root/c\_dir/test.txt

# 此时在宿主机的volume\_dir中就出现了test.txt, 即文件是实时保存在宿主机中的.

ls

方式二：--mount参数

**--mount type=bind, src=宿主机文件或文件夹路径, dst=容器中的文件或者文件夹路径**

注意：src指定的文件和路径必须提前创建或存在, type, src, dst之间没有空格. 必须使用绝对路径.

cd

mkdir mount\_dir

ls

cd mount\_dir

ls

docker run -dti --mount type=bind,src=/root/mount\_dir,dst=/root/c\_dir2 centos

docker exec d5c4 touch /root/c\_dir2/test2.txt

ls mount\_dir/

课件演示:

docker run --rm -dti -v /root/host\_dir:/root/c\_dir centos bash

mkdir /root/host\_dir2

docker run --rm -dti --mount type=bind,src=/root/host\_dir2,dst=/root/c\_dir2 centos bash

#### volumes方式挂载数据卷

* 利用docker run/create为容器挂载数据卷
* 用法：

方式一： -v, --volume参数

**-v VOLUME-NAME:容器中的文件或者文件夹路径**

方式二：--mount 参数

**--mount type=volume, src=VOLUME-NAME, dst=容器中的文件或者文件夹路径**

* volume对象管理：

docker volume 命令管理volume数据卷对象

docker volume **create 创建数据卷对象**

docker volume **inspect 查看数据卷详细信息**

docker volume **ls 查看已创建的数据卷对象**

docker volume **prune 删除未被使用的数据卷对象**

docker volume **rm 删除一个或多个数据卷对象**

# 使用-v方式指定数据卷的名称挂载数据卷, 如果指定的数据卷不存在, 会自动创建.

docker run -dti -v volume\_test:/root/c\_dir centos

# 查看所有的数据卷

docker volume ls

# 再次自动创建数据卷

docker run -dti -v volume\_test2:/root/c\_dir centos

# 查看所有的数据卷

docker volume ls

# 使用已经存在的数据卷

docker run -dti -v volume\_test:/root/c\_dir centos

# 可以省略数据卷的名称, 这样docker 会生成随机的名称. 在只关注数据卷的内容而不手动进行文件的管理时就可以使用这种方式.

docker run -dti -v /root/c\_dir centos

# 查看所有的数据卷

docker volume ls

# --mount 的方式挂载不存在的数据卷, 会自动创建数据卷

docker run -dti --mount type=volume,src=volume\_test3,dst=/root/c\_dir centos

# 查看所有的数据库

docker volume ls

# --mount 方式挂载已经存在的数据卷

docker run -dti --mount type=volume,src=volume\_test,dst=/root/c\_dir centos

手动创建数据卷docker volume create

docker volume create --help

Usage: docker volume create [OPTIONS] [VOLUME]

Create a volume

Options:

-d, --driver string Specify volume driver name (default "local")

--label list Set metadata for a volume

-o, --opt map Set driver specific options (default map[])

docker volume create volume\_test4

docker volume inspect

docker volume inspect --help

Usage: docker volume inspect [OPTIONS] VOLUME [VOLUME...]

Display detailed information on one or more volumes

Options:

-f, --format string Format the output using the given Go template

docker inspect volume\_test

[

{

"CreatedAt": "2018-12-03T15:00:34+08:00",

"Driver": "local",

"Labels": null,

"Mountpoint": "/var/lib/docker/volumes/volume\_test/\_data",

"Name": "volume\_test",

"Options": null,

"Scope": "local"

}

]

# 在宿主机中切换到root账户, 进入Mountpoint中的路径. 相当于把此路径挂载到了docker run -dti -v volume\_test:/root/c\_dir centos 创建的容器中的/root/c\_dir路径. volume方式挂载数据卷相当于是对bind方式的封装, 数据卷保存到指定的 "/var/lib/docker/volumes/" 位置.

cd /var/lib/docker/volumes/volume\_test/\_data

docker volume prune

docker volume prune --help

Usage: docker volume prune [OPTIONS]

Remove all unused local volumes

Options:

--filter filter Provide filter values (e.g. 'label=<label>')

-f, --force Do not prompt for confirmation

# 删除未使用的数据卷

docker volume prune

# 查看所有的容器

docker ps -a

# 删除所有的容器

docker rm -f fc c7 8b a6

# 再次删除所有未使用的数据卷

docker volume prune

docker volume rm

docker volume rm --help

Usage: docker volume rm [OPTIONS] VOLUME [VOLUME...]

Remove one or more volumes

Aliases:

rm, remove

Examples:

$ docker volume rm hello

hello

Options:

-f, --force Force the removal of one or more volumes

#### tmpfs mount方式挂载数据卷

* 利用docker run/create为容器挂载数据卷
* 用法：

方式一：--tmpfs list参数

方式二：--mount 参数, 推荐

**--mount type=tmpfs,dst=PATH**

# 数据保存在内存中

docker run -dti --mount type=tmpfs,dst=/root/c\_dir centos

#### 共享其他容器的数据卷-数据卷容器

* 利用docker run/create 的--volumes-from参数指定数据卷容器. 与container网络模式的思想是一致的. 使用其它容器的数据卷, 这样就能达到容器使用同一个数据卷的目的.
* 用法：

docker run/create --volumes-from CONTAINER

# 一般方式使用数据卷

docker run -dti --mount type=volume,src=volume\_test,dst=/root/c\_dir centos

docker inspect volume\_test

cd /var/lib/docker /volumes/volume\_test/\_data

ls

# 使用某容器的数据卷

docker run -dti --volumes-from ffb4 cenots

# 在新创建的容器中创建文件

docker exec 502f touch /root/c\_dir/test.txt

# 查看宿主机中的文件

cd /var/lib/docker/volumes/volume\_test/\_data

ls

### volume方式挂载数据卷注意事项

**Docker的数据卷更多会是使用volumes方式来进行使用。原因如下:**

1. 使用volumes方式创建数据卷不需要指定路径, 并且不需要事先创建路径. 如果数据卷不存在, 会自动创建.

2. 如果把容器复制到其它宿主机中使用, 使用 --mount type=bind 方式挂载的数据卷因为路径不存在会报错.

3. 可以使用docker volumes命令对数据卷进行统一管理.

**使用时需注意：**

* 如果挂载一个空的数据卷到容器中的一个非空目录中，那么容器中这个目录下的文件会被复制到数据卷中。
* 如果挂载一个非空的数据卷到容器中的一个目录中，不管容器中本目录中有没有文件, 那么容器中的目录中都会显示数据卷中的数据。如果原来容器中的目录中有数据，那么这些原始数据会被隐藏掉。

这两个规则都非常重要，灵活利用第一个规则可以帮助我们初始化数据卷中的内容。可以把一个容器或镜像中的数据全部初始化到宿主机的数据卷中, 掌握第二个规则可以保证挂载数据卷后的数据总是你期望的结果。

规则1演示:

docker run -dti centos

docker exec b2d0 ls /run

# 创建一个容器, 把数据卷挂载到容器中的/run 目录下, src参数可以省略, 省略后会创建随机名称的数据卷

docker run -dti --mount type=volume,src=volume\_test,dst=/run centos

# docker run -dti --mount type=volume,dst=/run centos

# 查看所有的数据卷

docker volume ls

# 查看数据卷

docker volume inspect volume\_test

# 查看数据卷中的文件

cd /var/lib/docker/volumes/volume\_test/\_data

ls

# 查看容器中/run目录下的所有文件, 与数据卷中的文件是相同的. 即容器中/run目录中的所有文件都被拷贝到了宿主机中新建的数据卷中, 相当于对新建的数据卷进行了一个初始化.

docker exec 677d ls /run

规则二演示

# 挂载已有的volume到容器中的空目录下

docker run -dti --mount type=volume,src=volume\_test,dst=/root/test\_dir centos

# 查看新创建的容器中的挂载目录下的文件, 原来volume中的内容就会映射到容器中的挂载目录下

docker exec 626f ls /root/test\_dir

# 挂载已有的volume到容器中的非空目录下

docker run -dti --mount type=volume,src=volume\_test,dst=/root/

# 查看容器挂载目录下的文件, 此时显示的依然是数据卷中的内容

docker exec 0dfb ls /root/

### 数据卷总结

重点掌握

* 数据卷特征和简介
* mount方式绑定数据卷
* 数据卷挂载的三种形式
* **数据卷使用的注意事项**

## Docker核心技术之仓库

### 课程概要

* Docker 仓库简介
* Docker 私有仓库搭建
* 总结

### Docker 仓库简介

什么是Docker仓库

* Docker仓库就是存放docker镜像并有docker pull方法下载的云环境
* Docker仓库分为公有仓库和私有仓库。

公有仓库指Docker Hub(官方)等开放给用户使用、允许用户管理镜像。

私有仓库指由用户自行搭建的存放镜像的云环境。

学习过程中更重要的不是学会每一步的操作, 而是思考每一步的操作的目的和作用.

### Docker 私有仓库搭建

#### 搭建无认证私有仓库

* 第一步：在需要搭建仓库的服务器上安装docker。
* 第二步：在服务器上，从docker hub下载registry仓库. registry仓库是一个特殊的镜像/容器, 可以通过registry创建一个自己的私有仓库.

**docker pull registry**

* 第三步：在服务器上，启动仓库

**docker run -d -ti --restart always\**

**--name my-registry\**

**-p 8000:5000\**

**-v /my-registry/registry:/var/lib/registry\**

**registry**

--restart always 如果使用service docker restart重启了docker服务, 或者宿主机重启后, 容器都会停止运行, 使用--restart always参数, 在重启了docker服务后会自动重启此容器.

-p 8000:5000 registry是一个封装好的web服务镜像, 创建的容器默认情况下会暴露5000端口, 把宿主机的8000端口映射到registry仓库的5000端口上, 这样通过宿主机的8000端口就可以访问到仓库的服务了.

-v 挂载数据卷. 私有仓库是存储镜像的. 默认情况下镜像是保存在容器中的/var/lib/registry目录中. 默认情况下镜像是保存在registry创建的容器中的, 而使用数据卷就可以把镜像保存到宿主机中. 宿主机中的路径是可以自定义的, 容器中的路径是固定的.

**注意：registry内部对外开放端口是5000。默认情况下，会镜像存放于容器内的/var/lib/registry (官网Dockerfile中查看) 目录下，这样如果容器被删除，则存放于容器中的镜像也会丢失。**

本地利用curl 服务器IP:8000/v2/\_catalog 查看当前仓库中的存放的镜像列表。（注意打开8000端口访问）

# 创建仓库

docker run -dti --restart always --name my-registry -p 8000:5000 -v /my\_registry/registry:/var/lib/registry registry

# 查看容器

docker ps -a

# 在本机访问宿主机的8000端口, 查看仓库中所有的镜像, 如果出现类似于 {"repositories":[]} 的字符串, 就说明仓库创建成功了.

curl 127.0.0.1:8000/v2/\_catalog

# 因为仓库使用的是http的协议, 所以也可以通过浏览器来访问. 在docker ps -a中看到端口映射中的ip地址是0.0.0.0, 也就是可以被所有的ip地址访问,

# 查看运行registry容器的宿主机的ip地址, 例如为47.94.153.230, 就可以在任意外部主机上通过浏览器访问 47.94.153.230:8000/v2/\_catalog, 显示的是仓库中所有的镜像文件.

#### 私有仓库--上传、下载镜像

* 第一步：利用docker tag重命名需要上传的镜像

docker tag IMAGE 服务器IP:端口/IMAGE\_NAME

* 第二步：利用docker push上传刚刚重命名的镜像

docker push 服务器IP:端口/centos

**注意：**

必须重命名为服务器IP:端口/IMAGE\_NAME

如果push出现了类似https的错误那么需要往配置文件/etc/docker/daemon.json里添加：**"insecure-registries":["服务器IP:端口"]**

然后重启docker。

# 查看所有的镜像

docker images

# 对要上传的进行更名. 服务器的ip地址为47.94.153.230:8000

docker tag centos 47.94.153.230:8000/centos\_latest

# 查看所有的镜像

docker images

# 上传镜像, 会出现错误. 在创建仓库时使用的是http的协议, 要配置https协议, 需要手动进行CA证书的配置等一系列工作

docker push 47.94.153.230:8000/centos\_latest

# 修改docker配置文件, 把https请求修改为http的请求, 表示 ["47.94.153.230:8000"] 这个仓库是不安全的仓库, 即走的是http的协议.

{

"registry-mirrors": ["https://dlfa9xic.mirror.aliyuncs.com"],

"insecure-registries": ["47.94.153.230:8000"]

}

# 重启docker服务

service docker restart

# 向仓库中推送镜像, 上传完成后, 使用curl命令或在浏览器中访问47.94.153.230:8000/v2/\_catalog, 就能看到仓库中中的镜像

docker push 47.94.153.230:8000/centos\_latest

# 使用curl检测镜像

curl 47.94.153.230:8000/v2/\_catalog

# 删除本地原来的46.94.153.230:8000/centos\_latest镜像

docker rmi 46.94.153.230:8000/centos\_latest

# 从私有仓库中下载镜像

docker pull 46.94.153.230:8000/centos\_latest

#### 搭建带认证的私有仓库（一）

在服务器上：

第一步：删除先前创建的无认证的仓库容器

**docker rm -f my-registry**

第二步：创建存放认证用户名和密码的文件：

**mkdir /my-registry/auth -p**

第三步：创建密码验证文件。注意将将**USERNAME**和**PASSWORD**替换为设置的用户名和密码

**docker run --entrypoint htpasswd registry -Bbn USERNAME PASSWORD > /my-registry/auth/htpasswd**

第四步：重新启动仓库镜像

**docker run -d -p 8000:5000 --restart=always --name docker-registry \**

**-v /my-registry/registry:/var/lib/registry \**

**-v /my-registry/auth:/auth \**

**-e "REGISTRY\_AUTH=htpasswd" \ # 指定使用的认证文件**

**-e "REGISTRY\_AUTH\_HTPASSWD\_REALM=Registry Realm" \ # 指定使用的认证方式**

**-e "REGISTRY\_AUTH\_HTPASSWD\_PATH=/auth/htpasswd" \ # 指定使用的认证文件在容器中的路径**

**registry**

-v /my-registry/auth:/auth 仓库容器中默认进行认证的文件是/auth, 使用数据卷把宿主机中的认证文件映射到容器中.

**视频演示**

# 删除无认证的仓库

docker rm -f my-registry

# 删除原有仓库的本地目录

rm -rf my-registry

# 创建保存认证用户名和密码的目录

mkdir /my-registry/auth -p

# 创建密码验证文件. 使用docker run文件启动registry仓库, 指定一系列的参数, 就会创建密码验证的密钥文件. --entrypoint htpasswd参数会把registry默认执行的命令修改为htpasswd, 启动容器时执行htpasswd命令, 使用提供的用户名和密码生成密码验证文件, 同时把生成的密码验证文件重定向到保存用户名和密码信息的目录中的htpasswd文件中.

docker run --entrypoint htpasswd registry -Bbn test\_username test\_password > /my-registry/auth/htpasswd

# 查看创建的密钥文件

ls /my-registry/auth/

# 查看密钥文件的内容

cat /my-registry/auth/htpasswd

# 创建带认证的私有仓库, 仓库中进行认证时默认使用/auth文件, 使用数据卷 -v /my-registry/auth:/auth 来使用宿主机中的/my-registry/auth/htpasswd这个文件来进行验证. 而在容器中, 相当于使用/auth/htpasswd这个文件来进行验证.

docker run -d -p 8000:5000 --restart=always --name docker-registry \

-v /my-registry/registry:/var/lib/registry \

-v /my-registry/auth:/auth \

-e "REGISTRY\_AUTH=htpasswd" \ # 设置环境变量

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

-e "REGISTRY\_AUTH\_HTPASSWD\_PATH=/auth/htpasswd" \ # 指定容器中认证的路径

registry

# 查看创建的私有仓库, 使用浏览器访问 47.94.153.230:8000/v2/\_catalog, 就会出现登录的对话框.

#### 带认证的私有仓库 -上传、下载镜像

在本地机器上：

第一步：首先登录到服务器, 如果出现了https请求的错误, 需要修改为http的协议

**docker login -u username -p password 47.94.153.230:8000**

第二步：然后执行pull或者push命令  
 第三步：操作完毕后，可以退出登录

**docker logout 47.94.153.230:8000**

这时如果想查看仓库中已有的镜像，那么需要进行http验证才可以。可以直接借助浏览器访问47.94.153.230:8000/v2/\_catalog就可以访问了

注意这里：47.94.153.230指服务器IP

#### Docker核心技术之仓库 - 总结

重点掌握：

* 私有仓库搭建（安全）

可以使用Dockerfile直接在远程仓库中更加方便的构建镜像..

## Docker核心技术之Dockerfile

### 课程概要

* Dockerfile 简介
* Dockerfile 示例与使用
* Dockerfile 特征
* Dockerfile 命令概述
* 总结

### Dockerfile简介

什么是Dockerfile

* Dockerfile其实就是根据特定的语法格式撰写出来的一个普通的文本文件
* 利用docker build命令依次执行在Dockerfile中定义的一系列命令，最终生成一个新的镜像（定制镜像）

### Dockerfile 示例与使用

Dockerfile参考示例

# 课件演示

mkdir df\_dir

cd df\_dir

vim Dockerfile

# Test

# VERSION 0.0.1

FROM ubuntu

LABEL Description="This image is used to test Dockerfile" Author="Itcast" Version="1.0"

RUN echo 'hello itcast'

CMD ["echo", "this is an image created by itcast"]

ls

# Dockerfile

cat Dockerfile

# 视频演示

mkdir dockerfile\_dir

cd dockerfile\_dir

vim Dockerfile

# 注释

# 测试DockerFile

# 关键字使用全大写, 变量使用小写. 以centos镜像为基础构建镜像

FROM centos

# 打印一句提示

RUN echo '这是一个测试的dockerfile'

### Dockerfile使用命令 – docker build

作用：

**根据dockerfile创建镜像**

命令格式：

**docker build [OPTIONS] PATH | URL | -**

命令参数：

**PATH Dockerfile所在路径(文件夹所在的路径), 文件名必须是Dockerfile**

**URL Dockerfile所在URL地址, 在使用时会先把Dockerfile下载下来, 再依据Dockerfile构建镜像.**

OPTIONS:

**-t, --tag list 为镜像设置名称和tag**

**-f, --file string 指定Dockerfile的路径(这是可以使用其他名称命名Dockerfile)**

docker build --help

Usage: docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

Options:

--add-host list Add a custom host-to-IP mapping (host:ip)

--build-arg list Set build-time variables

--cache-from strings Images to consider as cache sources

--cgroup-parent string Optional parent cgroup for the container

--compress Compress the build context using gzip

--cpu-period int Limit the CPU CFS (Completely Fair

Scheduler) period

--cpu-quota int Limit the CPU CFS (Completely Fair

Scheduler) quota

-c, --cpu-shares int CPU shares (relative weight)

--cpuset-cpus string CPUs in which to allow execution (0-3, 0,1)

--cpuset-mems string MEMs in which to allow execution (0-3, 0,1)

--disable-content-trust Skip image verification (default true)

-f, --file string Name of the Dockerfile (Default is

'PATH/Dockerfile')

--force-rm Always remove intermediate containers

--iidfile string Write the image ID to the file

--isolation string Container isolation technology

--label list Set metadata for an image

-m, --memory bytes Memory limit

--memory-swap bytes Swap limit equal to memory plus swap:

'-1' to enable unlimited swap

--network string Set the networking mode for the RUN

instructions during build (default "default")

--no-cache Do not use cache when building the image

--pull Always attempt to pull a newer version of

the image

-q, --quiet Suppress the build output and print image

ID on success

--rm Remove intermediate containers after a

successful build (default true)

--security-opt strings Security options

--shm-size bytes Size of /dev/shm

-t, --tag list Name and optionally a tag in the

'name:tag' format

--target string Set the target build stage to build.

--ulimit ulimit Ulimit options (default [])

#### 课件演示

cd df\_dir

ls

docker images

docker build . -t test-df

docker images

#### 视频演示

# 使用dockerfile构建一个新的镜像

cd dockerfile\_dir

ls

pwd

# 在生成新的镜像时Dockerfile中写的每一行语句都会以一个步骤来执行

docker build /root/dockerfile\_dir

# 构建的镜像的CREATED生成时间为运行build命令时的时间

docker images

# 构建镜像时添加TAG标签信息, 如果省略TAG时会使用 "latest"

docker build /root/dockerfile\_dir -t test\_image:v1.0

# 查看生成的镜像. 第二次构建镜像使用相同的Dockerfile, 没有进行任何更改, 所以不会重新生成一个镜像, 只是给第一次的镜像添加标签, 镜像id也不发生变化, CREATED时间依旧是第一次构建时的时间.

docker images

# 删除生成的镜像再次构建. 镜像id发生了变化, 所以镜像id实际上是随机生成的字符串.

docker rmi 7782

docker build /root/dockerfile\_dir -t test\_image:v1.0

# 进入到上一级目录中, 再次构建镜像. 只要PATH中提供的Dockerfile的路径正确, 当前的工作路径可以是任意的.

cd ..

docker build /root/dockerfile\_dir -t test\_image:v1.0

# 修改Dockerfile的文件名

mv dockerfile\_dir/Dockerfile dockerfile\_dir/test\_df

# 再次构建镜像, 就会出错. 找不到Dockerfile文件.

docker build /root/dockerfile\_dir -t test\_image:v1.0

# 使用 -f 参数指定构建镜像时使用的Dockerfile文件

docker build . -f dockerfile\_dir/test\_df -t test\_image:v1.0

docker build -f /root/dockerfile\_dir/test\_df -t test\_image:v1.0

### Dockerfile 特征

#### Dockerfile 构建特征（一）

**Dockerfile构建过程中的特征**

# 修改dockerfile文件为默认的Dockerfile文件名

cd dockerfile\_dir

mv test\_df Dockerfile

# 删除原来的镜像并重新构建镜像, 因为Dockerfile是在当前目录下的, 所以可以只写一个点表示当前的目录, 这是相当路径的写法, 也可以使用绝对路径来指定Dockerfile所在的目录.

docker build . -t test\_image:v1.0

# 再次构建镜像, 比较与第一次构建时的区别. 因为Dockerfile文件没有发生任何变化, 所以不会生成镜像. 但在第二次构建镜像时, 会对已存在的镜像进行检测, 发现第二次的Dockerfile文件没有发生变化, 之前没有发生变化的部分就不再执行, 而是使用之前生成的镜像文件.

docker build . -t test\_image:v1.0

# 修改Dockerfile文件, 添加一句新的语句.

vim Dockerfile

# 注释

# 测试DockerFile

# 关键字使用全大写, 变量使用小写. 以centos镜像为基础构建镜像

FROM centos

# 打印一句提示

RUN echo '这是一个测试的dockerfile'

RUN echo '在后面新增一句话'

# 再次执行镜像的构建, 因为第2条语句已经执行过, 在step2中依旧不会重新执行, 而是使用原来的文件, 在step3中会执行新的语句.

docker build . -t test\_image:v1.0

# 再次修改Dockerfile文件, 在前面添加一条语句

# 注释

# 测试DockerFile

# 关键字使用全大写, 变量使用小写. 以centos镜像为基础构建镜像

FROM centos

# 打印一句提示

RUN echo '在前面新增一句话'

RUN echo '这是一个测试的dockerfile'

RUN echo '在后面新增一句话'

# 再次执行镜像的构建, 因为 "RUN echo '这是一个测试的dockerfile'" 这句话前面还有对应的语句, 相当于在新的镜像的基础上执行的语句, 就不会再使用原来的缓存文件, 而是完全重新开始生成镜像, 每一条语句都会重新执行.

# 再次修改第2条语句, 也不会使用缓存文件, 而是完全重新构建镜像. **只要Dockerfile中某一条语句发生了变化, 从这一条语句开始, 都会重新执行. 而前面未发生变化的语句和操作, 则会使用缓存文件.**

docker build . -t test\_image:v1.0

**Dockerfile生成的镜像的特征**

# 删除之前使用Dockerfile生成的所有的镜像

# 修改Dockerfile文件, 只保留 " RUN echo '这是一个测试的dockerfile'" 这条语句

# 注释

# 测试DockerFile

# 关键字使用全大写, 变量使用小写. 以centos镜像为基础构建镜像

FROM centos

# 打印一句提示

RUN echo '这是一个测试的dockerfile'

# 重新构建镜像.

docker build . -t test\_image

docker images

# 查看生成的镜像的历史信息

docker history 9afd

# 查看并对比centos的历史信息. 新生成的镜像的历史信息在centos的基础上又添加了一层. 类似于docker commit 容器 --> 镜像 的操作. docker build命令底层实现的过程就类似于docker commit的过程.

docker history centos

**# docker build命令分析**

docker build . -t test\_image

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM centos # 使用centos镜像, 以此为母版进行操作. 如果centos已经下载, 就会使用本地的镜像, 否则就会尝试下载此镜像. 只是引用了centos镜像, 不会生成新的一层.

---> 75835a67d134

Step 2/2 : RUN echo '这是一个测试的dockerfile'

---> Running in 3bf2517d7a2c # linux中执行echo '这是一个测试的dockerfile', 会打印出来这句话, 此时linux会分配进程或线程来执行命令, 而在docker中, 想要执行命令, 必须要基于一个容器, 所以在执行这句话的时候, 会先基础centos的镜像75835a67d134生成一个容器, 在生成的容器中执行命令. 这里显示的Running in 3bf2 中的 3bf2 就是容器的id.

这是一个测试的dockerfile

Removing intermediate container 3bf2517d7a2c

---> 26f31311a68b # 这两句的执行过程是: 使用docker commit命令根据中间的临时容器生成新的镜像, 然后删除中间的临时容器, 得到了一个id为 26f3 的新的镜像.

Successfully built 26f31311a68b

Successfully tagged test\_image:latest

# 修改Dockerfile, 增加一层内容.

# 注释

# 测试DockerFile

# 关键字使用全大写, 变量使用小写. 以centos镜像为基础构建镜像

FROM centos

# 打印一句提示

RUN echo '这是一个测试的dockerfile'

RUN echo '在后面新增一句话'

# 再次构建镜像. 第2条语句不会重新执行, 而是使用test\_image1的镜像.

docker build . -t test\_image2

# 查看构建的镜像, 发现原来的镜像已经被新生成的镜像替代了.

docker images

# 查看新构建的镜像的历史信息. 在原来镜像test\_image的基础上又添加了一层

docker history 1107

# 修改Dockerfile, 在前面增加一条语句.

# 注释

# 测试DockerFile

# 关键字使用全大写, 变量使用小写. 以centos镜像为基础构建镜像

FROM centos

# 打印一句提示

RUN echo '在前面新增一句话'

RUN echo '这是一个测试的dockerfile'

RUN echo '在后面新增一句话'

# 再次构建镜像. 这时会重新执行每一条语句.

docker build . -t test\_image3

# 查看构建的镜像, 由于在构建新的镜像时没有继承原来的镜像, 而是重新生成了一个新的镜像, 所以原来的镜像会被遗弃, name和tag字段都会成为<none>

docker images

# 查看新构建的镜像的历史信息. 就会在centos历史信息的基础上增加3层的信息, 而不会使用原来构建的test\_image2镜像.

docker history 1107

在构建镜像时, 如已经进行了一部分的操作, 如安装了程序包.

FROM centos

RUN yum -y install net-tools

RUN yum -y install gcc

**想要再进行其它的操作, 一般都是在原来Dockerfile语句的基础上继续向下添加语句, 不要在原有的语句的上面添加操作, 这样就能使用原来已经存在的镜像.**

FROM centos

RUN yum -y install net-tools

RUN yum -y install gcc

RUN yum -y install cmake

* 查看官方的Dockerfile：<https://github.com/docker-library/docs>

查看并对比redis:latest的Dockerfile与本地的redis镜像的历史信息.



#### Dockerfile 构建特征（二）

* Dockerfile必须具备一个FROM命令来进行构建
* 每一个Dockerfile命令都会构建一层镜像（本质上是每一层都会启动一个容器，执行完命令后，将容器进行提交后，产生新的镜像层）
* 通过查看下载下来的镜像，发现历史层信息的层ID是missing，其实是因为原本的层id只存在于构建镜像的宿主机上，一旦转移镜像后，历史层消息中将只保留最新一层的ID

### Dockerfile 命令概述

#### Dockerfile 命令概述（一）

查看完整介绍

<https://docs.docker.com/engine/reference/builder/>

对比官方的Dockerfile和下载的对应镜像的历史信息, 来理解Dockerfile每一句的含义.

* FROM: 指定基础镜像
* RUN： 构建镜像过程中需要执行的命令。可以有多条。如centos的yum命令, ubuntu的apt install 命令, shell命令等. 执行docker build构建镜像时写在RUN后面的所有命令才会被依次执行.

RUN命令有两种方式, shell方式执行, 会以子进程的方式执行命令. RUN yum -y install net-tools

exec 方式执行, 会在当前进程下执行命令. RUN ["yum", "-y", "install", "net-tools"]

在linux中执行shell命令时, 一般调用的是/bin/sh来执行命令. /bin/sh -c echo 'hello', 因为是在后台以子进程的方式执行的, 所以直接在linux命令行中执行时没有任何输出. 当前shell中有一些环境变量, 使用/bin/sh来执行时可以使用当前shell中的环境变量. 很多情况下都是使用shell方式执行命令的.

在linux中执行exec命令, exec echo 'hello', 程序会把当前的shell终端kill掉, 然后执行exec的命令, exec命令执行结束时, 当前的shell终端就会退出. 由于exec方式执行时会先kill当前的shell终端, 所以无法使用shell中的环境变量.

* CMD：添加启动容器时需要执行的命令。多条命令时只有最后一条生效。可以在启动容器时被覆盖和修改。

docker inspect centos查看centos镜像的详细信息. 在Config.Cmd的信息为 "/bin/bash", centos镜像中的Cmd命令是在执行docker run IMAGE [COMMAND] 且忽略[COMMAND]命令时执行的命令, 例如, 在docker run -dti centos时, 会默认去执行 "/bin/bash" 命令. 如果在docker run时提供了COMMAND命令, 就会执行提供的COMMAND命令, 而忽略 Config.Cmd中的命令, 如docker run -dti centos python, 此时执行的就是python命令, 而忽略centos默认的 "bin/bash" 命令.

CMD命令的三种方式

1. CMD ["executable","param1","param2"] (exec form, this is the preferred form)

以主进程的方式执行. 推荐的方式, 因为一般都是要在程序的主进程中执行某一个命令. 另外, 容器的生命周期是与进程号为1的主进程相关的, 当进程号为1的主进程结束时, 容器的运行就结束并退出. 如果以子进程的方式执行, 当主进程退出时, 子进程的执行可能还没有结束, CMD中的命令可能不会完整执行. 所以这里要以主进程的方式执行.

2. CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

在ENTRYPOINT中指定了要运行的命令, 可以在CMD中给ENTRYPOINT中要执行的命令添加参数. 也就是说, 如果在exec 模式的CMD 命令中指定的不是命令而是参数, 那么这些参数都是传递给ENTRYPOINT命令使用的.

3. CMD command param1 param2 (shell form)

* ENTRYPOINT：指定镜像生成容器后要执行的脚本. 功能同CMD，但这个命令一定会被执行，不会被覆盖修改。

ENTRYPOINT命令的两种方式

1. ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)

2. ENTRYPOINT command param1 param2 (shell form)

* LABEL：为镜像添加对应的数据。

使用docker inspect centos查看详情, Config.Labels中的信息.

查看centos:latest的label信息.

LABEL org.label-schema.schema-version="1.0" \ # 版本

org.label-schema.name="CentOS Base Image" \ # 名称

org.label-schema.vendor="CentOS" \

org.label-schema.license="GPLv2" \ # 协议

org.label-schema.build-date="20181006" # 构建日期

* MAINTAINER：表明镜像的作者, docker inspect Author中的信息。将被遗弃，被LABEL代替。
* EXPOSE：设置对外暴露的端口。

在docker中提供对外的服务, web, 数据库的服务, 需要对外暴露端口才提供服务. 可以在inspect中的Config.ExposedPorts中查看暴露出来的端口.

如对于redis, 需要暴露出6379端口.

EXPOSE 6379

在使用docker run -P redis时, 会把宿主机中的某一个随机端口映射到EXPOSE中设置的暴露端口上来.

可以暴露出多个端口, 如.

EXPOSE 80/tcp

EXPOSE 80/udp

在使用docker run -P 时, 会把宿主机上的多个随机端口映射到所有暴露的端口上来.

* ENV：设置执行命令时的环境变量，并且在构建完成后，仍然生效

# 在linux中设置环境变量

export KEY="value"

# 使用环境变量

echo $KEY

在RUN, CMD和ENTRYPOINT中, 对于shell方式执行的命令, 是以子进程的方式执行的, 可以使用shell的环境变量, 而对于exec方式执行的命令, 会先把终端kill掉, 不是以shell方式执行的命令, 所以不能使用shell中的环境变量, 但exec sh -c表示以exec的方式执行一个shell, 再在这个shell中执行一些命令, 这样就可以使用shell中的环境变量了.

ENV <key> <value>

ENV <key>=<value> ...

For example:

ENV myName="John Doe" myDog=Rex\ The\ Dog \

myCat=fluffy

and

ENV myName John Doe

ENV myDog Rex The Dog

ENV myCat fluffy

* ARG：设置只在构建过程中使用的环境变量，构建完成后，将消失

ARG 不只可以设置CMD和ENTRYPOINT中可以使用的环境变量, 还可以给EXPOSE, ADD, COPY等其它命令设置环境变量.

ARG命令只在构建的过程中才生效. 使用ENV设置的环境变量, 在构建成镜像后, 使用docker run/create命令时依然可以使用, 使用ARG定义的环境变量在构建成镜像后就不能再使用了.

使用方法:

ARG <name>[=<default value>]

FROM busybox

ARG user1

ARG buildno

...

An ARG instruction can optionally include a default value:

FROM busybox

ARG user1=someuser

ARG buildno=1

...

在Dockerfile中使用ARG参数定义了环境变量, 可以在构建镜像的过程中使用docker build --build-arg 这个参数把一些如用户名的变量动态的传递给Dockerfile. 如果ARG指定了一个默认值并且在构建期间没有传递值过去，那么就使用默认值。

ARG变量定义从在Dockerfile定义的行生效，而不是从在命令行参数的使用或其它地方。

例如下面的Dockerfile:

1 FROM busybox

2 USER ${user:-some\_user}

3 ARG user

4 USER $user

...

使用如下命令构建：

$ docker build --build-arg user=what\_user Dockerfile

第2行的USER的值为some\_user，因为user变量在第3行才定义。第4行的USER值为what\_user，因为user变量在它之前定义了并且在命令行给user变量赋值为what\_user。在ARG指令定义变量之前引用这个变量的得，都会得到空值。

警告：不推荐在构建期间的命令行传递密码如github密钥，用户凭证等数据。构建期间设置的变量可以通过docker history命令来查看。

* ADD：将本地文件或目录拷贝到镜像的文件系统中。能解压特定格式文件，能将URL作为要拷贝的文件. tar, gz格式的打包文件或压缩文件可以解包和解压, 但ADD方式会对所有的tar和gz尝试解包及解压, 且有可能会解压失败, 所以实际中一般使用COPY复制文件, 然后使用RUN执行tar文件来解包或解压. 对于url文件的复制, 可以使用RUN执行curl来下载文件.
* COPY：将本地文件或目录拷贝到镜像的文件系统中。
* VOLUME：添加数据卷.

这里的VOLUME是指定的镜像或容器中的路径, 不是宿主机中的路径.

查看redis:latest的Dockerfile文件中的VOLUME.

VOLUME /data

如果在Dockerfile中设置了VOLUME, 那么在使用docker run或docker create命令时, 即使不使用-v来挂载数据卷, 也会自动生成数据卷.

docker volume ls

# VOLUME /data相当于docker run -dti -v /data redis的操作.

docer run -dti redis

docer volume ls

* USER：指定以哪个用户的名义执行RUN, CMD 和ENTRYPOINT等命令. 有些应用需要特定的用户才能使用. 如mysql
* WORKDIR：设置工作目录, 类似于linux中的cd命令, 可以先进入一个路径, 再执行某一种命令.

#### Dockerfile 命令概述（二）

* ONBUILD：如果制作的镜像被另一个Dockerfile使用，将在那里执行Dockerfile命令
* STOPSIGNAL：设置容器退出时发出的关闭信号。这里的信号来源于 kill -l中的信号
* HEALTHCHECK：设置容器状态检查。
* SHELL：更改执行shell命令的程序。Linux的默认shell是["/bin/sh", "-c"]，Windows的是["cmd", "/S", "/C"]。

bash命令相当于是sh命令的增强版.

bash命令与sh命令的区别

### 总结

重点掌握：

* Docker 容器与镜像之间的关系（尤其commit命令的作用和效果）
* Dockerfile 的书写规则和使用规则. 通过Docker官方镜像Dockerfile进行学习.

## Docker核心技术之Docker Compose

### 课程概要

* Docker Compose 简介
* 了解 Docker Compose File
* Docker Compose 应用
* 总结

### Docker Compose 简介

Docker Compose是什么？

* Docker Compose是一个能一次性定义和管理多个Docker容器的工具。
* 详细地说：

Compose中定义和启动的每一个容器都相当于一个服务(service)

Compose中能定义和启动多个服务，且它们之间通常具有协同关系

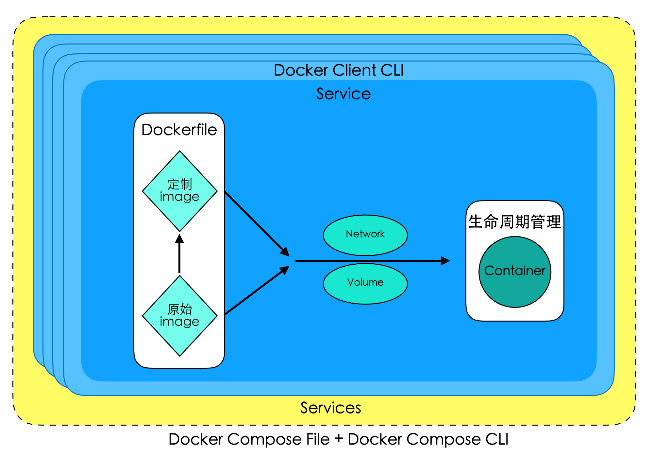
* 管理方式：

使用YAML文件来配置我们应用程序的服务。

使用单个命令（docker-compose up），就可以创建并启动配置文件中配置的所有服务。

Dockerfile解决的是镜像的构建问题, 而Docker Compose解决的是镜像的使用, 即容器的批量化管理的问题.

### Docker Compose 工作原理



Docker Client CLi, Docker命令行工具.

一个容器提供的功能在运行时可以看成是运行在宿主机上的一个服务, 如对于redis服务, 需要有原始的镜像, 在其中安装并配置好redis, 然后通过docker run从镜像生成容器, 由于redis的数据一般要保存到宿主机中, 所以还要给容器挂载数据卷. 为了使其它的容器/主机/用户能够使用redis的服务, 还需要配置网络和端口. 生成容器之后就可以对它进行生命周期的管理, 即容器的启动, 暂停, 重启等操作.

以上这个服务可以看成是通过Docker Client CLI来进行管理的.

一个完整的项目往往需要有多个不同的功能来配合实现复杂的功能, 如一个web项目, 需要有web应用服务, 数据库存储服务, 数据缓存服务等. 每一个服务都对应着上面的一个完整的流程. docker compose就是以服务为基本单位来对容器来进行管理的功能.

Docker Compose主要有2部分组成, 一部分是Docker compose file, 即Docker compose的配置文件, 包含每/多个服务/容器使用的原始镜像, 构建镜像时的Dockerfile, 数据卷和网络的配置, 生命周期管理如自动重启规则等内容. 第二部分是Docker Compose CLI, 类似于Docker Clinet CLI, 即Docker Compose的命令行管理工具, 根据Docker Compose File来启动一系列的服务.

### Docker Compose安装

* Docker for Mac与Docker for Windows自带docker-compose
* Linux下需要单独安装：

<https://docs.docker.com/compose/install/#install-compose>

第1步: 下载Docker Compose

sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

第2步: 更改权限

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

第3步: 终端中查看安装的版本

docker-compose --version

这里示例安装版本是1.21.2，很可能您看到这里时，已经出现更新的版本，因此建议换成最新版本。<https://github.com/docker/compose/releases>

* 其他安装方法查看

https://docs.docker.com/compose/install/

1. Run this command to download the latest version of Docker Compose:

sudo curl -L "https://github.com/docker/compose/releases/download/1.23.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# Use the latest Compose release number in the download command.

The above command is an example, and it may become out-of-date. To ensure you have the latest version, check the Compose repository release page on GitHub.

<https://github.com/docker/compose/releases>

If you have problems installing with curl, see Alternative Install Options tab above. <https://docs.docker.com/compose/install/#alternative-install-options>

2. Apply executable permissions to the binary:

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

3. Optionally, install command completion for the bash and zsh shell.

<https://docs.docker.com/compose/completion/>

4. Test the installation.

docker-compose --version

# docker-compose version 1.23.1, build 1719ceb

### Docker Compose CLI

* 利用**docker-compose --help**查看或者查看官方文档.

<https://docs.docker.com/compose/reference/overview/>

* 对比后会发现：Docker Compose CLI的很多命令的功能和Docker Client CLI是相似的。最主要的区别就是前者能一次性运行管理多个容器，后者只能一次管理一个。

docker-compose --help

Define and run multi-container applications with Docker.

Usage:

docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]

docker-compose -h|--help

Options:

-f, --file FILE Specify an alternate compose file

(default: docker-compose.yml)

-p, --project-name NAME Specify an alternate project name

(default: directory name)

--verbose Show more output

--log-level LEVEL Set log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)

--no-ansi Do not print ANSI control characters

-v, --version Print version and exit

-H, --host HOST Daemon socket to connect to

--tls Use TLS; implied by --tlsverify

--tlscacert CA\_PATH Trust certs signed only by this CA

--tlscert CLIENT\_CERT\_PATH Path to TLS certificate file

--tlskey TLS\_KEY\_PATH Path to TLS key file

--tlsverify Use TLS and verify the remote

--skip-hostname-check Don't check the daemon's hostname against the

name specified in the client certificate

--project-directory PATH Specify an alternate working directory

(default: the path of the Compose file)

--compatibility If set, Compose will attempt to convert deploy

keys in v3 files to their non-Swarm equivalent

Commands:

build Build or rebuild services

bundle Generate a Docker bundle from the Compose file

config Validate and view the Compose file

create Create services

down Stop and remove containers, networks, images, and volumes

events Receive real time events from containers

exec Execute a command in a running container

help Get help on a command

images List images

kill Kill containers

logs View output from containers

pause Pause services

port Print the public port for a port binding

ps List containers

pull Pull service images

push Push service images

restart Restart services

rm Remove stopped containers

run Run a one-off command

scale Set number of containers for a service

start Start services

stop Stop services

top Display the running processes

unpause Unpause services

up Create and start containers

version Show the Docker-Compose version information

docker --help

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:

--config string Location of client config files (default "/home/david/.docker")

-D, --debug Enable debug mode

-H, --host list Daemon socket(s) to connect to

-l, --log-level string Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")

--tls Use TLS; implied by --tlsverify

--tlscacert string Trust certs signed only by this CA (default "/home/david/.docker/ca.pem")

--tlscert string Path to TLS certificate file (default "/home/david/.docker/cert.pem")

--tlskey string Path to TLS key file (default "/home/david/.docker/key.pem")

--tlsverify Use TLS and verify the remote

-v, --version Print version information and quit

Management Commands:

builder Manage builds

config Manage Docker configs

container Manage containers

engine Manage the docker engine

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

trust Manage trust on Docker images

volume Manage volumes

Commands:

attach Attach local standard input, output, and error streams to a running container

build Build an image from a Dockerfile

commit Create a new image from a container's changes

cp Copy files/folders between a container and the local filesystem

create Create a new container

diff Inspect changes to files or directories on a container's filesystem

events Get real time events from the server

exec Run a command in a running container

export Export a container's filesystem as a tar archive

history Show the history of an image

images List images

import Import the contents from a tarball to create a filesystem image

info Display system-wide information

inspect Return low-level information on Docker objects

kill Kill one or more running containers

load Load an image from a tar archive or STDIN

login Log in to a Docker registry

logout Log out from a Docker registry

logs Fetch the logs of a container

pause Pause all processes within one or more containers

port List port mappings or a specific mapping for the container

ps List containers

pull Pull an image or a repository from a registry

push Push an image or a repository to a registry

rename Rename a container

restart Restart one or more containers

rm Remove one or more containers

rmi Remove one or more images

run Run a command in a new container

save Save one or more images to a tar archive (streamed to STDOUT by default)

search Search the Docker Hub for images

start Start one or more stopped containers

stats Display a live stream of container(s) resource usage statistics

stop Stop one or more running containers

tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

top Display the running processes of a container

unpause Unpause all processes within one or more containers

update Update configuration of one or more containers

version Show the Docker version information

wait Block until one or more containers stop, then print their exit codes

Run 'docker COMMAND --help' for more information on a command.

### 了解 Docker Compose File

Docker Compose File版本

* [Docker Compose File](https://docs.docker.com/compose/compose-file/) 有多个版本，基本是向后兼容的，但也有极个别配置项高版本中没有。
* 在docker-compose.yml一开始就需要利用version关键词标明当前file使用的版本

docker-compose --version

docker --version

|  |  |
| --- | --- |
| **Compose file format** | **Docker Engine release** |
| 3.7 | 18.06.0+ |
| 3.6 | 18.02.0+ |
| 3.5 | 17.12.0+ |
| 3.4 | 17.09.0+ |
| 3.3 | 17.06.0+ |
| 3.2 | 17.04.0+ |
| 3.1 | 1.13.1+ |
| 3.0 | 1.13.0+ |
| 2.4 | 17.12.0+ |
| 2.3 | 17.06.0+ |
| 2.2 | 1.13.0+ |
| 2.1 | 1.12.0+ |
| 2.0 | 1.10.0+ |
| 1.0 | 1.9.1.+ |

### Docker Compose File TOP配置参数概览

Docker compose file中顶格写的是TOP配置项. 使用多个空格的缩进来区分不同等级的配置项, 且缩进必须要一致.

每一个冒号和它在同一行中的内容之间都有一个空格

Docker Compose File 顶级配置项：

* version：指定Docker Compose File版本号. 在第1行指定版本号.
* services：定义多个服务并配置启动参数.
* volumes：声明或创建在多个服务中共同使用的数据卷对象
* networks：定义在多个服务中共同使用的网络对象
* configs：声明将在本服务中要使用的一些配置文件. Docker Compose可以配置并启动多个容器, 可以构建一个全局的配置文件, 供多个服务/容器使用, 这样, 只需要维护这一个单独的配置文件, 就能够达到对多个服务/容器进行管理的目的..
* secrets：声明将在本服务中要使用的一些秘钥、密码文件. 用户名和密码一般不写在docker compose file中, 而是要写到宿主机中的一个文件中, 再通过命令拷贝到容器中使用.
* x-\*\*\*：自定义配置。主要用于复用相同的配置。每个服务/容器中都有相同的配置, 就可以在top层中进行配置, 然后再在各个服务/容器中进行引用即可. 而修改时只需要修改top层中的配置即可. 类似于全局变量的作用.

更多详细配置

https://docs.docker.com/compose/compose-file/

### services顶级配置项的常用参数

#### build

用于指定创建某个服务/容器时使用的Dockerfile

<https://docs.docker.com/compose/compose-file/#build>

version: '3'

services:

webapp:

build:

context: ./dir # Dockerfile文件所在的路径

dockerfile: Dockerfile-alternate # Dockerfile文件, 如果不指定, 就默认使用 "Dockerfile"

args: # 设置build过程中的参数

buildno: 1

#### command

重写镜像Cmd中的命令

Override the default command.

command: bundle exec thin -p 3000

The command can also be a list, in a manner similar to [dockerfile](https://docs.docker.com/engine/reference/builder/#cmd):

command: ["bundle", "exec", "thin", "-p", "3000"]

#### container\_name

指定容器的名称

Specify a custom container name, rather than a generated default name.

container\_name: my-web-container

Because Docker container names must be unique, you cannot scale a service beyond 1 container if you have specified a custom name. Attempting to do so results in an error.

Note: This option is ignored when [deploying a stack in swarm mode](https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

#### entrypoint

重写镜像中 entrypoint中的参数设置

Override the default entrypoint.

entrypoint: /code/entrypoint.sh

The entrypoint can also be a list, in a manner similar to dockerfile:

entrypoint:

- php

- -d

- zend\_extension=/usr/local/lib/php/extensions/no-debug-non-zts-20100525/xdebug.so

- -d

- memory\_limit=-1

- vendor/bin/phpunit

Note: Setting entrypoint both overrides any default entrypoint set on the service’s image with the ENTRYPOINT Dockerfile instruction, and clears out any default command on the image - meaning that if there’s a CMD instruction in the Dockerfile, it is ignored.

#### deploy

Docker Compose可以一次性配置并启动多个服务/容器, 但默认情况下只能在单机上启动多个服务, 想要跨主机配置和启动, 就要使用deploy

只要使用了deploy配置项, docker compose的版本必须在3以上.

[**Version 3**](https://docs.docker.com/compose/compose-file/compose-versioning/#version-3)**only.**

Specify configuration related to the deployment and running of services. This only takes effect when deploying to a [swarm](https://docs.docker.com/engine/swarm/) with [docker stack deploy](https://docs.docker.com/engine/reference/commandline/stack_deploy/), and is ignored by docker-compose up and docker-compose run.

version: '3'

services:

redis:

image: redis:alpine

deploy:

replicas: 6

update\_config:

parallelism: 2

delay: 10s

restart\_policy:

condition: on-failure

#### depends\_on

有些容器/服务是其它其它的容器/服务才能启动和运行的, 要使用depends\_on设置容器/服务启动的顺序.

如果没有设置depends\_on, 则会使用随机的启动顺序.

Express dependency between services, Service dependencies cause the following behaviors:

* docker-compose up starts services in dependency order. In the following example, db and redis are started before web.
* docker-compose up SERVICE automatically includes SERVICE’s dependencies. In the following example, docker-compose up web also creates and starts db and redis.

Simple example:

version: '3'

services:

web:

build: .

depends\_on: # web服务/容器只有在db和redis服务/容器启动之后才启动

- db

- redis

redis:

image: redis

db:

image: postgres

There are several things to be aware of when using depends\_on:

* depends\_on does not wait for db and redis to be “ready” before starting web - only until they have been started. If you need to wait for a service to be ready, see [Controlling startup order](https://docs.docker.com/compose/startup-order/) for more on this problem and strategies for solving it.
* Version 3 no longer supports the condition form of depends\_on.
* The depends\_on option is ignored when [deploying a stack in swarm mode](https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a version 3 Compose file.

#### env\_file

把环境变量写入到文件中, 使用env\_file配置项来指定该文件, 这样才能在Docker Compose中使用. 这种方法一般适用于环境变量特别多, 需要进行分类的情况下使用. 在环境变量比较少的情况下, 一般使用Environment来进行配置.

Add environment variables from a file. Can be a single value or a list.

If you have specified a Compose file with docker-compose -f FILE, paths in env\_file are relative to the directory that file is in.

Environment variables declared in the [environment](https://docs.docker.com/compose/compose-file/#environment) section override these values – this holds true even if those values are empty or undefined.

env\_file: .env

env\_file:

- ./common.env

- ./apps/web.env

- /opt/secrets.env

Compose expects each line in an env file to be in VAR=VAL format. Lines beginning with # are treated as comments and are ignored. Blank lines are also ignored.

# Set Rails/Rack environment

RACK\_ENV=development

Note: If your service specifies a build option, variables defined in environment files are not automatically visible during the build. Use the args sub-option of build to define build-time environment variables.

The value of VAL is used as is and not modified at all. For example if the value is surrounded by quotes (as is often the case of shell variables), the quotes are included in the value passed to Compose.

Keep in mind that the order of files in the list is significant in determining the value assigned to a variable that shows up more than once. The files in the list are processed from the top down. For the same variable specified in file a.env and assigned a different value in file b.env, if b.env is listed below (after), then the value from b.env stands. For example, given the following declaration in docker\_compose.yml:

services:

some-service:

env\_file:

- a.env

- b.env

And the following files:

# a.env

VAR=1

and

# b.env

VAR=hello

$VAR is hello.

#### environment

适用于环境变量比较少的情况. 使用key: value的形式进行配置.

Add environment variables. You can use either an array or a dictionary. Any boolean values; true, false, yes no, need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser.

Environment variables with only a key are resolved to their values on the machine Compose is running on, which can be helpful for secret or host-specific values.

environment:

RACK\_ENV: development

SHOW: 'true'

SESSION\_SECRET:

environment:

- RACK\_ENV=development

- SHOW=true

- SESSION\_SECRET

Note: If your service specifies a build option, variables defined in environment are not automatically visible during the build. Use the args sub-option of build to define build-time environment variables.

#### expose

暴露端口, 只是暴露, 并不作映射

Expose ports without publishing them to the host machine - they’ll only be accessible to linked services. Only the internal port can be specified.

expose:

- "3000"

- "8000"

#### ports

Expose ports.

暴露端口, 并进行映射

Note: Port mapping is incompatible with network\_mode: host

##### SHORT SYNTAX

Either specify both ports (HOST:CONTAINER), or just the container port (an ephemeral host port is chosen).

Note: When mapping ports in the HOST:CONTAINER format, you may experience erroneous results when using a container port lower than 60, because YAML parses numbers in the format xx:yy as a base-60 value. For this reason, we recommend always explicitly specifying your port mappings as strings.

ports:

- "3000" # 暴露3000并进行随机映射

- "3000-3005" # 暴露某一个范围并进行随机映射

- "8000:8000" # 暴露8000并指定宿主机端口进行映射

- "9090-9091:8080-8081"

- "49100:22"

- "127.0.0.1:8001:8001"

- "127.0.0.1:5000-5010:5000-5010"

- "6060:6060/udp"

##### LONG SYNTAX

The long form syntax allows the configuration of additional fields that can’t be expressed in the short form.

* target: the port inside the container
* published: the publicly exposed port
* protocol: the port protocol (tcp or udp)
* mode: host for publishing a host port on each node, or ingress for a swarm mode port to be load balanced.

ports:

- target: 80

published: 8080

protocol: tcp

mode: host

Note: The long syntax is new in v3.2

#### image

Specify the image to start the container from. Can either be a repository/tag or a partial image ID.

image如果作为build命令的二级命令, 表示给构建出来的镜像添加一个名称/版本号

image命令如果单独使用, 则表示使用某个镜像. 如下, redis服务使用的是redis:latest的镜像来启动, db服务使用的是postgres镜像来启动.

version: '3'

services:

web:

build: .

depends\_on:

- db

- redis

redis:

image: redis

db:

image: postgres

image: redis

image: ubuntu:14.04

image: tutum/influxdb

image: example-registry.com:4000/postgresql

image: a4bc65fd

If the image does not exist, Compose attempts to pull it, unless you have also specified build, in which case it builds it using the specified options and tags it with the specified tag.

#### labels

Add metadata to containers using Docker labels. You can use either an array or a dictionary.

It’s recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

labels:

com.example.description: "Accounting webapp"

com.example.department: "Finance"

com.example.label-with-empty-value: ""

labels:

- "com.example.description=Accounting webapp"

- "com.example.department=Finance"

- "com.example.label-with-empty-value"

#### networks

Networks to join, referencing entries under the top-level networks key.

services:

some-service:

networks:

- some-network

- other-network

#### volumes

Mount host paths or named volumes, specified as sub-options to a service.

给启动的服务绑定数据卷, 可以同时绑定多个数据卷.

You can mount a host path as part of a definition for a single service, and there is no need to define it in the top level volumes key.

But, if you want to reuse a volume across multiple services, then define a named volume in the top-level volumes key. Use named volumes with services, swarms, and stack files.

Note: The top-level volumes key defines a named volume and references it from each service’s volumes list. This replaces volumes\_from in earlier versions of the Compose file format. See Use volumes and Volume Plugins for general information on volumes.

This example shows a named volume (mydata) being used by the web service, and a bind mount defined for a single service (first path under db service volumes). The db service also uses a named volume called dbdata (second path under db service volumes), but defines it using the old string format for mounting a named volume. Named volumes must be listed under the top-level volumes key, as shown.

version: "3.2"

services:

web:

image: nginx:alpine

volumes:

- type: volume

source: mydata

target: /data

volume:

nocopy: true

- type: bind

source: ./static

target: /opt/app/static

db:

image: postgres:latest

volumes:

- "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.sock"

- "dbdata:/var/lib/postgresql/data"

volumes:

mydata:

dbdata:

Note: See Use volumes and Volume Plugins for general information on volumes.

##### SHORT SYNTAX

Optionally specify a path on the host machine (HOST:CONTAINER), or an access mode (HOST:CONTAINER:ro).

You can mount a relative path on the host, that expands relative to the directory of the Compose configuration file being used. Relative paths should always begin with . or ...

volumes:

# Just specify a path and let the Engine create a volume

- /var/lib/mysql

# Specify an absolute path mapping

- /opt/data:/var/lib/mysql

# Path on the host, relative to the Compose file

- ./cache:/tmp/cache

# User-relative path

- ~/configs:/etc/configs/:ro

# Named volume

- datavolume:/var/lib/mysql

##### LONG SYNTAX

The long form syntax allows the configuration of additional fields that can’t be expressed in the short form.

* type: the mount type volume, bind or tmpfs
* source: the source of the mount, a path on the host for a bind mount, or the name of a volume defined in the [top-levelvolumes key](https://docs.docker.com/compose/compose-file/#volume-configuration-reference). Not applicable for a tmpfs mount.
* target: the path in the container where the volume is mounted
* read\_only: flag to set the volume as read-only
* bind: configure additional bind options
  + propagation: the propagation mode used for the bind
* volume: configure additional volume options
  + nocopy: flag to disable copying of data from a container when a volume is created
* tmpfs: configure additional tmpfs options
  + size: the size for the tmpfs mount in bytes

version: "3.2"

services:

web:

image: nginx:alpine

ports:

- "80:80"

volumes:

- type: volume

source: mydata

target: /data

volume:

nocopy: true

- type: bind

source: ./static

target: /opt/app/static

networks:

webnet:

volumes:

mydata:

Note: The long syntax is new in v3.2

#### restart

指定容器/服务的自动重启规则, 也就是说在什么情况下自动重启服务.

no is the default restart policy, and it does not restart a container under any circumstance. When always is specified, the container always restarts. The on-failure policy restarts a container if the exit code indicates an on-failure error.

restart: "no"

restart: always

restart: on-failure

restart: unless-stopped

Note: This option is ignored when deploying a stack in swarm mode with a (version 3) Compose file. Use restart\_policy instead.

### Docker Compose File 参考示例

version: '3.6' # 使用的Docker Compose的版本

services: # 在services中定义要启动的服务/容器, 这里包含多个服务/容器

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:6.2.4

environment:

- cluster.name=docker-cluster

- bootstrap.memory\_lock=false

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

volumes:

- esdata1:/usr/share/elasticsearch/data

ports:

- 9200:9200

networks:

- esnet

deploy:

placement:

constraints:

- node.role == manager

elasticsearch2:

image: docker.elastic.co/elasticsearch/elasticsearch:6.2.4

environment:

- cluster.name=docker-cluster

- bootstrap.memory\_lock=false

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

- "discovery.zen.ping.unicast.hosts=elasticsearch"

volumes:

- esdata2:/usr/share/elasticsearch/data

networks:

- esnet

deploy:

placement:

constraints:

- node.role == worker

logstash:

image: docker.elastic.co/logstash/logstash:6.2.4

environment:

- "LS\_JAVA\_OPTS=-Xms256m -Xmx256m"

networks:

- esnet

deploy:

replicas: 2

logstash2:

image: docker.elastic.co/logstash/logstash:6.2.4

environment:

- "LS\_JAVA\_OPTS=-Xms256m -Xmx256m"

networks:

- esnet

deploy:

replicas: 2

kibana:

image: docker.elastic.co/kibana/kibana:6.2.4

ports:

- "5601:5601"

networks:

- esnet

deploy:

placement:

constraints:

- node.role == manager

volumes:

esdata1:

driver: local

esdata2:

driver: local

networks:

esnet:

driver: "overlay"

### Docker Compose 应用

#### 案例一 小型web服务项目搭建

步骤：

* 搭建一个flask的小型web项目
* 根据项目环境，利用Dockerfile构建镜像
* 撰写docker-compose.yaml配置文件，启动项目

步骤1: 搭建一个flask的小型web项目

项目所需文件:

app.py为简单服务器. 统计并在浏览器中显示网站的访问量, 访问量是保存在redis服务器中的.

*# encoding=utf-8*

**import** **time**

**import** **redis**

**from** **flask** **import** Flask

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

*# 此处host是docker-compose.yaml配置文件中 redis服务的名称*

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

**def** get\_hit\_count():

*'''利用redis统计访问次数'''*

retries = 5

*# 由于当redis重启时，可能会有短暂时间无法访问 redis*

*# 因此循环的作用就是在这个期间重试，默认重试5次*

**while** True:

**try**:

*# redis的incr方法，如果hits值存在则自动+1，否则新增该键，值为1*

**return** cache.incr("hits")

**except** redis.execeptions.ConnectError **as** **exec**:

**if** retries == 0:

**raise** **exec**

retries -= 1

time.sleep(0.5)

@app.route("/")

**def** main():

count = get\_hit\_count()

**return** "欢迎访问！网站已经累计访问{}次**\n**".format(count)

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

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

requirements.txt为项目所需要的第3方python包.

redis

flask

mkdir -p /root/workspace/case1-flask-web/flask-web-code

把2个项目文件复制到 flask-web-code 文件夹中

步骤2: 根据项目环境，利用Dockerfile构建镜像

cd /root/workspace/case1-flask-web

vim Dockerfile

# flask web app v1.0

# 搭建一个基于flask的web项目: 实现了简单的访问量统计

# 第1步: 获取 一个镜像: python3.6

# 从<https://github.com/docker-library/docs>中查找相应的python官方镜像. 推荐使用alpine相当的镜像, alpine是一个轻量级的linux镜像, 只保留了最核心的功能. alpine中安装包的方法是 apk add xxx. 如3.6.8-alpine3.9, 3.6-alpine3.9, 3.6.8-alpine, 3.6-alpine (3.6/alpine3.9/Dockerfile), 这里显示的名称就是镜像的TAG信息

FROM python:3.6-alpine

# 可以添加自定义的标签

# LABEL Description="本镜像用于启动建议的基于flask的web程序" Author="Itcast" Version="1.0"

# 第2步: 拷贝项目代码到容器中 ADD COPY

COPY ./flask-web-code /code

# 第3步: 安装项目的依赖环境, 第3方模块: flask redis

# 因为项目的文件都已经复制到容器的/code目录下了, 所以需要先切换工作目录

WORKDIR /code

RUN pip install -r requirements.txt

# 第4步: 配置项目的启动 CMD 参数 python app.py

# 使用exec方式来执行启动命令

CMD ["python", "app.py"]

步骤2: 通过Dockerfile构建镜像

# 本步骤只是用来测试Dockerfile文件, 在执行Docker Compose文件时, 会自动执行build命令, 从Dockerfile来生成镜像.

# redis镜像不需要进行配置, 只要基于redis镜像生成容器就可以了

docker build . -t my-flask-image

步骤3: 创建Docker Compose File

# 如果使用默认的文件名 docker-compose.yaml, 在后面运行时就不用指定了, 如果使用自定义的文件名, 运行时就要指定docker compose file文件.

vim docker-compose.yaml

version: "3.6" *# 查看 docker 的版本, Docker version 18.09.0, 支持的 docker compose 版本为 3.6 和 3.7. 注意 冒号之后要有一个空格, 否则就会出错. 错误的真实位置在错误信息中指定的行的上一行.*

services: *# 运行的容器/服务, 一是 flask-web 服务, 二是 redis 服务*

flask-web: *# 指定服务的名称, 可以自己定义*

build: . *# 运行容器/服务需要镜像, 这里直接使用当前路径下即 docker-compose.yaml 文件所在目录下的 Dockerfile 来构建镜像.*

ports: *# web 服务要对外暴露端口, 以供用户访问. flask 默认是5000端口, 所以要暴露并进行端口映射. 不用手动创建网络, 会自动生成一个网络来完成两个容器/服务之间的通信.*

- "5000:5000"

container\_name: flask\_web *# 自定义容器名称*

redis: *# redis服务只提供给 flask-web 来使用, 宿主机之外的主机不直接访问 redis 数据库, 所以不需要使用 ports 对外暴露端口. 又因为 redis 本身已经暴露了 6379 端口, 所以也不需要使用 expose 来暴露端口.*

image: redis *# redis直接使用镜像生成容器就可以了, 而上面的flask, 则需要指定Dockerfile, 通过Dockerfile构建的镜像来生成容器*

container\_name: redis

# 检测 docker-compose 文件的格式是否正确

docker-compose config

# 使用 docker-compose 来启动容器, 如果之前没有使用Dockerfile来构建镜像, 就会在启动时自动创建镜像. 如果镜像已经存在, 就不会再次创建. 所以在Docker Compose File中对于某一个服务/容器, 如果使用了build方式, 就不需要事先构建好镜像.

docker-compose up

# 在本机和远程机器上访问服务器

# 本机多次访问

curl 127.0.0.1:5000

# 使用Ctrl+C停止容器的运行, 使用docker-compose up命令来运行时是阻塞的方式执行, 加上 -d 参数以非阻塞的方式运行.

docker-compose up -d

# 查看容器是否运行

docker ps -a

# docker-compose down命令会停止容器的运行, 删除容器, 删除网络 network case1-flask-web\_default

docker-compose down

# 查看网络

docker network ls

# 再次运行docker compose

docker-compose up -d

# 再次查看网络, 会多出一个Name为case1-flask-web\_default, DRIVER为bridge的网络. 也就是说, 如果在 docker compose file 中不指定网络, 会自动创建一个bridge的网络来完成flask-web和redis两个服务之间的通信.

docker network ls

# 修改docker compose file, 指定网络, 需要在top级定义使用的网络, 并且在每个服务中指定top中使用的网络.

vim docker-compose.yaml

version: "3.6" *# 查看 docker 的版本, Docker version 18.09.0, 支持的 docker compose 版本为 3.6 和 3.7. 注意 冒号之后要有一个空格, 否则就会出错. 错误的真实位置在错误信息中指定的行的上一行.*

services: *# 运行的容器/服务, 一是 flask-web 服务, 二是 redis 服务*

flask-web:

build: . *# 运行容器/服务需要镜像, 这里直接使用当前路径下的 Dockerfile 来构建镜像.*

ports: *# web 服务要对外暴露端口, 以供用户访问. flask 默认是5000端口, 所以要暴露并进行端口映射. 网络不进行设置, 使用docker0网络??*

- "5000:5000"

container\_name: flask\_web *# 自定义容器名称*

networks: *# 指定 flask-web 容器与 redis 容器通信时使用的网络*

- flask-web-network

redis: *# redis服务只提供给 flask-web 来使用, 宿主机之外的主机不直接访问 redis 数据库, 所以不需要使用 ports 对外暴露端口. 又因为 redis 本身已经暴露了 6379 端口, 所以也不需要使用 expose 来暴露端口.*

image: redis

container\_name: redis

networks: *# 指定 flask-web 容器与 redis 容器通信时使用的网络*

- flask-web-network

networks:

flask-web-network: *# 定义 flask-web 容器与 redis 容器通信时使用的网络*

driver: "bridge"

# 对docker compose file进行检测

docker-compose config

# 运行docker compose

docker-compose up -d

# 查看网络, 会多出来一个名为 case1-flask-web\_flask-web-network的bridge网络

docker network ls

# 停止docker compose, 会删除所有的容器和网络

docker-compose down

# 再次运行docker compose, 再次访问127.0.0.1:5000, 因为是重新生成的redis容器, 所以访问量还是从1开始计算的.

docker-compose up -d

# 查看生成的容器

docker ps -a

# 修改docker-compose.yaml文件, 把redis的container\_name修改为redis2, 再次运行docker-compose, 此时docker compose就会在原有的基础上对docker compose file中修改过的内容进行重新运行, 其它未修改的内容则不会重新运行. 这里会重新生成一个redis的容器.

docker-compose up -d

# 查看生成的容器, redis的容器被删除, 重新创建了一个redis2的容器.

docker ps -a

# 因为在docker compose file中并没有手动设置redis的数据卷, redis的数据是保存到redis容器中的, 再次访问127.0.0.1:5000, 此时访问量的统计量应该被重置, 但事实上统计量却没有重置.

# 查看redis的镜像的详细信息, Config.Volumes中的值并不是空的, 而是 "/data": {}, 也就是说在基于redis镜像创建容器时, 会在宿主机中自动创建一个数据卷, 并映射到redis容器中.

docker inspect redis

# 查看所有的数据卷, 其中有一个随机命名的数据卷.

docker volume ls

# 再次修改docker-compose.yaml文件, 把redis的container\_name修改为redis3, 再次通过redis-compose up -d 来运行, 查看数据卷. 并没有创建出数据卷, 也就是说还是使用的宿主机上的同一个数据卷. 这就是在修改redis的容器时访问次数不会被重置的原因.

docker volume ls

# 使用docker-compose down 删除容器, 此时会删除网络, 但是不会删除数据卷. 再次使用docker-compose up -d 来运行容器, 在本机使用curl 127.0.0.1:5000访问服务器, 访问次数就被重置了, 意味着此时使用的已经不是原来存在的数据卷了, 而是会重新生成新的数据卷并使用.

docker-compose down

docker-compose up -d

docker volume ls

# 所以一般不使用docker-compose down命令来删除容器, 而是使用docker-compose stop/kill/pause来中止或暂停容器的运行. 而使用docker-compose start/unpause来重启停止运行或暂停运行的容器.

# 如果想要在使用docker-compose down删除容器相关的内容之后, 再次使用docker-compose up -d 来生成并运行容器时还能使用原来的数据卷, 就要手动指定docker-compose file中的数据卷.

# 先在top配置项中创建数据卷, 然后在redis服务中指定创建的数据卷.

vim docker-compose.yaml

version: "3.6" *# 查看 docker 的版本, Docker version 18.09.0, 支持的 docker compose 版本为 3.6 和 3.7. 注意 冒号之后要有一个空格, 否则就会出错. 错误的真实位置在错误信息中指定的行的上一行.*

services: *# 运行的容器/服务, 一是 flask-web 服务, 二是 redis 服务*

flask-web:

build: . *# 运行容器/服务需要镜像, 这里直接使用当前路径下的 Dockerfile 来构建镜像.*

ports: *# web 服务要对外暴露端口, 以供用户访问. flask 默认是5000端口, 所以要暴露并进行端口映射. 网络不进行设置, 使用docker0网络??*

- "5000:5000"

container\_name: flask\_web *# 自定义容器名称*

networks: *# 指定 flask-web 容器与 redis 容器通信时使用的网络*

- flask-web-network

redis: *# redis服务只提供给 flask-web 来使用, 宿主机之外的主机不直接访问 redis 数据库, 所以不需要使用 ports 对外暴露端口. 又因为 redis 本身已经暴露了 6379 端口, 所以也不需要使用 expose 来暴露端口.*

image: redis

container\_name: redis

networks: *# 指定 flask-web 容器与 redis 容器通信时使用的网络*

- flask-web-network

volumes: # 指定redis使用的数据卷为top配置项中生成的数据卷, 并指定数据卷在redis容器中的挂载位置. 挂载位置可以在docker inspect redis中查看.

- redis-data:/data

networks:

flask-web-network: *# 定义 flask-web 容器与 redis 容器通信时使用的网络*

driver: "bridge"

volumes: # 创建数据卷

redis-data:

driver: "local"

# 基于新的docker-compose运行更新创建的容器/服务, 检测代码是否正确.

docker-compose up -d

# 停止并删除原来的容器

docker-compose down

# 删除所有的数据卷

docker volume prune

# 重新运行容器, 停止并删除容器, 再次运行容器, 每次运行容器之后都访问服务器, 测试访问次数是否被重置了.

docker-compose up -d

curl 127.0.0.1:5000

docker-compose down

docker volume ls

docker-compose up -d

curl 127.0.0.1:5000

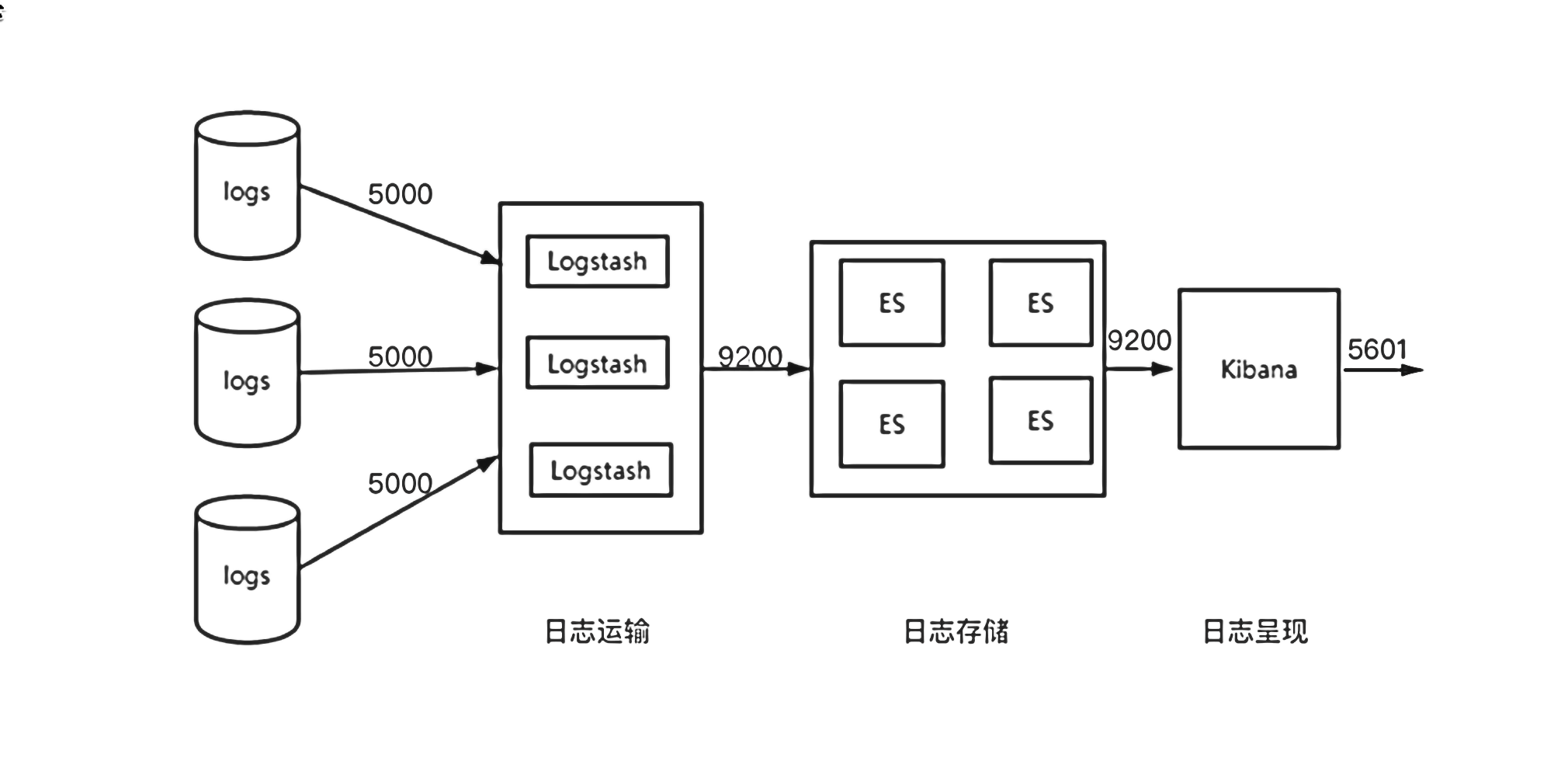
docker-compose down

如果想要持久化的保存访问记录, 就要把redis的数据保存到宿主机中.

#### 案例二 单机环境ELK系统搭建（一）

* [ELK](https://www.elastic.co/guide/index.html)工作原理介绍

<https://www.elastic.co/guide/index.html>



#### 案例二 单机环境ELK系统搭建（二）

步骤：

* 配置单机版的docker-compose.yaml文件（[ELK镜像地址](https://www.docker.elastic.co/)）

<https://www.docker.elastic.co/>

* 利用docker-compose up启动环境

1. 下载镜像

docker pull docker.elastic.co/elasticsearch/elasticsearch:6.7.0

docker pull docker.elastic.co/kibana/kibana:6.7.0

docker pull docker.elastic.co/logstash/logstash:6.7.0

2. 使用官方docker-compose.yaml尝试运行生产环境下的es服务

mkdir case2-single-node-elk

cd case2-single-node-elk

vim docker-compose.yaml

version: '2.2'

services: *# 配置了两个elasticsearch服务, es的集群服务, 这个es的集群中有两个节点.*

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:6.7.0 *# 使用的镜像*

container\_name: elasticsearch *# 启动后的容器/服务名称*

environment: *# 配置环境变量*

- cluster.name=docker-cluster *# 集群名称, 两个es服务的cluster.name相同, 表示这两个服务是处于同一个集群的, 在从外部来看, 二者是相同的*

- bootstrap.memory\_lock=true *# es内存锁, 如果不设置, es会无限制的增加内容*

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m" *# 设置内存的最小, 最大使用量. Xms512m最小512m, Xmx512m最大512m*

ulimits:

memlock:

soft: -1

hard: -1

volumes:

- esdata1:/usr/share/elasticsearch/data

ports:

- 9200:9200 *# 因为两个es是属于同一个集群的, 所以只需要对一个服务进行端口映射即可. es默认会打开9200和9300端口, es2只是没有进行端口映射而已, 端口其实也已经打开了.*

networks:

- esnet

elasticsearch2:

image: docker.elastic.co/elasticsearch/elasticsearch:6.7.0

container\_name: elasticsearch2

environment:

- cluster.name=docker-cluster

- bootstrap.memory\_lock=true

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

- "discovery.zen.ping.unicast.hosts=elasticsearch" *# web项目中的flask如果想要使用redis, 根据redis的服务名redis:端口来进行调用. 即redis:6379进行访问. es2对外提供的服务是通过第一个es服务的服务名elasticsearch:9200来进行的, 外部在向es1中写入或从es2中读取数据的时候, 是通过es1来进行的, 也就是外部能同时访问到es1和es2, 有了这个设置, es1和es2才能真正算的上是属于同一个集群的服务.*

ulimits:

memlock: *# 打开内存锁的权限. 只有这里打开了权限, 上面环境变量中设置的memory\_lock才能真正的生效.*

soft: -1

hard: -1

volumes:

- esdata2:/usr/share/elasticsearch/data

networks: *# 让两个es服务处于同一个网络中*

- esnet

volumes:

esdata1:

driver: local

esdata2:

driver: local

networks:

esnet:

# 启动es服务

docker-compose up -d

# 查看服务是否启动成功

docker ps -a

# 在docker-compose.yaml所在路径下查看日志信息, 以确定es服务启动过程中是否有错误. 服务的启动需要一段时间, 需要等待一段时间后再次查看确认.

docker-compose logs

# 发现启动过程中的错误, 最大虚拟内存设置的过低.

elasticsearch | ERROR: [1] bootstrap checks failed

elasticsearch | [1]: max virtual memory areas vm.max\_map\_count [65530] is too low, increase to at least [262144]

elasticsearch | [2019-04-01T00:26:23,912][INFO ][o.e.n.Node ] [V8ECwlo] stopping ...

elasticsearch | [2019-04-01T00:26:23,942][INFO ][o.e.n.Node ] [V8ECwlo] stopped

elasticsearch | [2019-04-01T00:26:23,943][INFO ][o.e.n.Node ] [V8ECwlo] closing ...

elasticsearch | [2019-04-01T00:26:23,968][INFO ][o.e.n.Node ] [V8ECwlo] closed

# 再次查看容器运行状态, 发现es服务已经退出了.

docker ps -a

# 过一段时间再次查看日志和容器运行状态, 发现日志会一直更新, 容器的运行状态也会发生变量, 这是因为es在运行出错后会在一段时间内多次尝试重启服务.

查看官方说明, 需要把宿主机最大的虚拟内存设置的大一些.

The vm.max\_map\_count kernel setting needs to be set to at least 262144 for production use. Depending on your platform:

**Linux**

The vm.max\_map\_count setting should be set permanently in /etc/sysctl.conf:

$ grep vm.max\_map\_count /etc/sysctl.conf

vm.max\_map\_count=262144

To apply the setting on a live system type:

sysctl -w vm.max\_map\_count=262144

每次启动都要运行这一句话, 可以把这句话放到开机运行的脚本中.

# 可以先用docker-compose down删除所有的容器, 再重启启动容器, 然后查看容器的运行状态和日志信息. 如果没有出现错误, 使用curl来访问es服务

curl 127.0.0.1:9200

{

"name" : "V8ECwlo",

"cluster\_name" : "docker-cluster",

"cluster\_uuid" : "sAOOMEWhQNSURnDK1U-cig",

"version" : {

"number" : "6.7.0",

"build\_flavor" : "default",

"build\_type" : "docker",

"build\_hash" : "8453f77",

"build\_date" : "2019-03-21T15:32:29.844721Z",

"build\_snapshot" : false,

"lucene\_version" : "7.7.0",

"minimum\_wire\_compatibility\_version" : "5.6.0",

"minimum\_index\_compatibility\_version" : "5.0.0"

},

"tagline" : "You Know, for Search"

}

# esk之logstash设置

# 下载镜像

docker pull docker.elastic.co/logstash/logstash:6.7.0

# 配置logstash

<https://www.elastic.co/guide/en/logstash/6.7/docker-config.html>

Configuring Logstash for Docker

pipeline\_configuration

logstash的作用是收集日志并把日志保存到es中, pipeline中设置的就是从哪里收集日志并把日志保存到什么地方的问题. 如果不进行设置, logstash默认会把本身运行过程中的日志收集起来推送到es中.

settings

logstash运行过程中的一些配置项

Table 1. Example Docker Environment Variables

|  |  |
| --- | --- |
| **Environment Variable** | **Logstash Setting** |
| PIPELINE\_WORKERS | pipeline.workers |
| LOG\_LEVEL | log.level |
| XPACK\_MONITORING\_ENABLED | xpack.monitoring.enabled |

Docker defaults

The following settings have different default values when using the Docker images:

|  |  |  |
| --- | --- | --- |
| http.host | 0.0.0.0 | 绑定地址为4个0, 表示无论容器运行的ip是什么, 都能在外部通过这个ip来访问到logstash. |
| xpack.monitoring.elasticsearch.hosts | http://elasticsearch:9200 |  |

6.2.4设置

|  |  |  |
| --- | --- | --- |
| http.host | 0.0.0.0 |  |
| path.config | /usr/share/logstash/pipeline | pipeline文件默认的路径 |

In the x-pack image, the following additional defaults are also set:

|  |  |  |
| --- | --- | --- |
| xpack.monitoring.elasticsearch.url | http://elasticsearch:9200 | 服务之间相互访问, 处在同一个网络中, 使用服务名:端口来进行访问 |
| xpack.monitoring.elasticsearch.username | logstash\_system |  |
| xpack.monitoring.elasticsearch.password | changeme |  |

修改docker-compose.yaml文件, 添加logstash的配置信息, 再次使用docker-compose up -d 来运行服务. 多次查看容器运行状态和服务信息, 确定服务已经正常启动.

在logstash没有配置日志来源的情况下, 会把自身运行过程中产生的服务收集并发送给es, 为了能够查看es中是否已经存在了日志信息, 需要安装kibana.

version: '2'

services:

kibana:

image: docker.elastic.co/kibana/kibana:6.7.0

volumes:

- ./kibana.yml:/usr/share/kibana/config/kibana.yml # 如果自定义了kibana的yml文件, 需要使用数据卷的方式替换到默认的yml文件.

查看kibana的默认配置信息

Docker defaults[edit](https://github.com/elastic/kibana/edit/6.7/docs/setup/docker.asciidoc)

The following settings have different default values when using the Docker images:

|  |  |
| --- | --- |
| server.name | kibana |
| server.host | "0" |
| elasticsearch.hosts | http://elasticsearch:9200 |
| xpack.monitoring.ui.container.elasticsearch.enabled | true |

xpack需要es授权的用户名和密码才能使用

添加kibana的yaml信息

version: '2.2'

services:

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:6.2.4

container\_name: elasticsearch

environment:

- cluster.name=docker-cluster

- bootstrap.memory\_lock=true

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

ulimits:

memlock:

soft: -1

hard: -1

volumes:

- esdata1:/usr/share/elasticsearch/data

ports:

- 9200:9200

networks:

- esnet

elasticsearch2:

image: docker.elastic.co/elasticsearch/elasticsearch:6.2.4

container\_name: elasticsearch2

environment:

- cluster.name=docker-cluster

- bootstrap.memory\_lock=true

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

- "discovery.zen.ping.unicast.hosts=elasticsearch"

ulimits:

memlock:

soft: -1

hard: -1

volumes:

- esdata2:/usr/share/elasticsearch/data

networks:

- esnet

logstash:

image: docker.elastic.co/logstash/logstash:6.2.4

environment:

- "LS\_JAVA\_OPTS=-Xms256m -Xmx256m" *# 限制logstash的内存使用*

container\_name: logstash

networks:

- esnet *# 与es在同一个网络中, 才能正常访问es*

depends\_on: *# logstash依赖于es服务, 在es服务启动之后才启动*

- elasticsearch

- elasticsearch2

logstash2:

image: docker.elastic.co/logstash/logstash:6.2.4

environment:

- "LS\_JAVA\_OPTS=-Xms256m -Xmx256m"

container\_name: logstash2

networks:

- esnet

depends\_on:

- elasticsearch

- elasticsearch2

kibana:

image: docker.elastic.co/kibana/kibana:6.2.4

container\_name: kibana

ports: *# 端口映射, 才能在外部使用kibana服务*

- "5601:5601"

networks:

- esnet

depends\_on:

- elasticsearch

- elasticsearch2

volumes:

esdata1:

driver: local

esdata2:

driver: local

networks:

esnet:

查看配置信息的语法是否正确

docker-compose config

docker-compose up -d

# 查看日志

docker-compose logs -f

# 查看容器运行状态

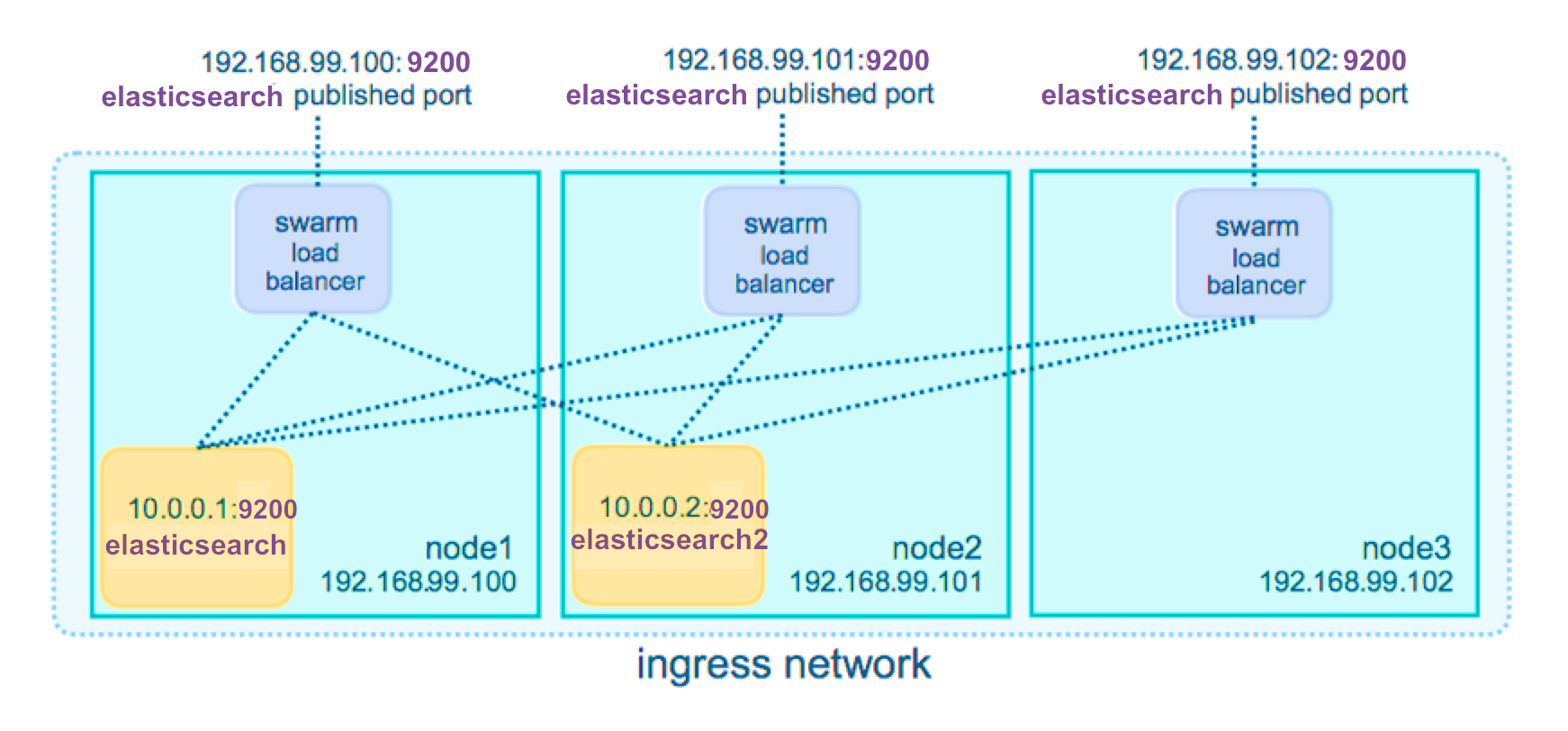
docker ps -a

使用127.0.0.1:9200访问es

使用127.0.0.1:5601访问kibana

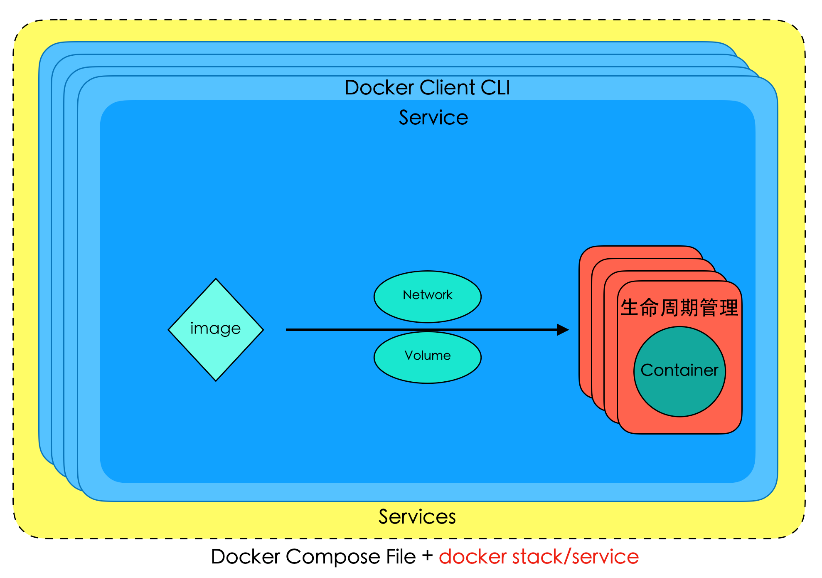
#### 案例三 多主机环境ELK系统搭建（一）

* Swarm 介绍



#### 案例三 多主机环境ELK系统搭建（二）

* 集群版Docker Compose工作原理



#### 案例三 多主机环境ELK系统搭建（三）

步骤

* 使用[docker swarm](https://docs.docker.com/engine/reference/commandline/swarm/)配置多个docker node集群节点
* 配置集群版ELK的docker-compose.yaml文件
* 利用docker stack deploy部署集群版ELK环境

**总结**