作业三

学号：1300013022

姓名：武守北

一、安装配置Docker：

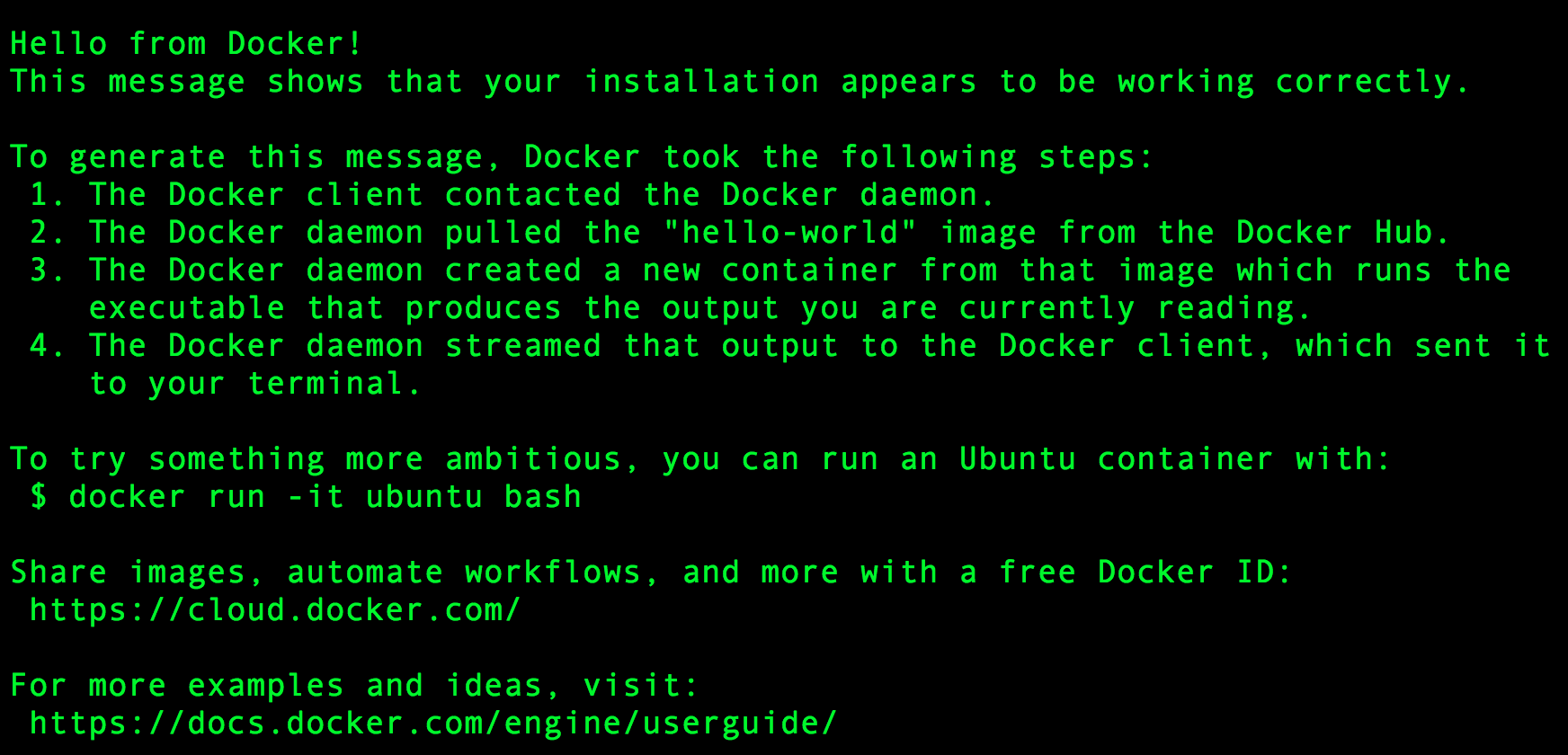
直接运行：

curl -fsSL https://get.docker.com/gpg | sudo apt-key add -

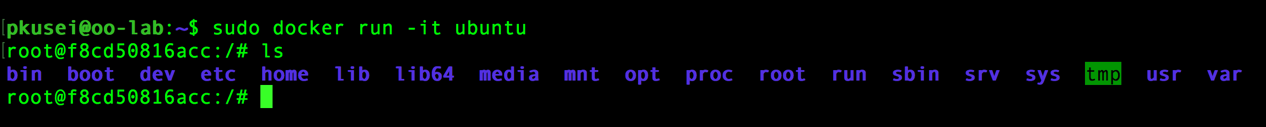
sudo docker run hello-world

sudo docker run -it ubuntu

运行hello-world结果如下：



运行ubantu：



二、介绍Docker五个命令：

1、Docker run：创建一个新的容器并且运行一个命令：

docker用于运行镜像的命令，会新建一个容器并在该容器上运行镜像，对于没有分配名字的container，run会自动分配一个名字（不能创建与运行中container名字相同的container），如上面的helloworld和下面的多个ubuntu。使用-itd为了让其生成一个后台运行的bash，使得不是建立后直接消亡。

语法：

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

OPTIONS:

-a stdin: 指定标准输入输出内容类型，可选 STDIN/STDOUT/STDERR 三项；

-d: 后台运行容器，并返回容器ID；

-i: 以交互模式运行容器，通常与 -t 同时使用；

-t: 为容器重新分配一个伪输入终端，通常与 -i 同时使用；

--name="nginx-lb": 为容器指定一个名称；

--dns 8.8.8.8: 指定容器使用的DNS服务器，默认和宿主一致；

--dns-search example.com: 指定容器DNS搜索域名，默认和宿主一致；

-h "mars": 指定容器的hostname；

-e username="ritchie": 设置环境变量；

--env-file=[]: 从指定文件读入环境变量；

--cpuset="0-2" or --cpuset="0,1,2": 绑定容器到指定CPU运行；

-m :设置容器使用内存最大值；

--net="bridge": 指定容器的网络连接类型，支持 bridge/host/none/container: 四种类型；

--link=[]: 添加链接到另一个容器；

--expose=[]: 开放一个端口或一组端口；

2、docker images: 列出本地镜像

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

option 说明：

-a :列出本地所有的镜像（含中间映像层，默认情况下，过滤掉中间映像层）；

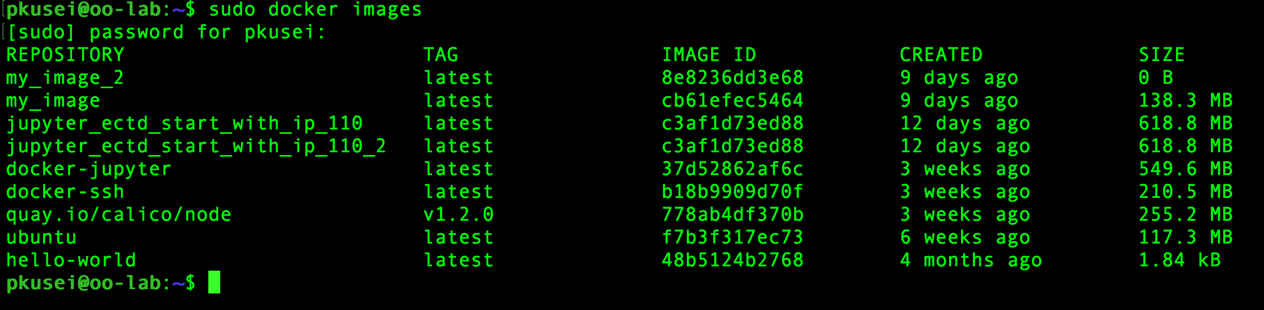
--digests :显示镜像的摘要信息；

-f :显示满足条件的镜像；

--format :指定返回值的模板文件；

--no-trunc :显示完整的镜像信息；

-q :只显示镜像ID。



3、docker network creat: 创建网络：

语法：docker network creat [OPTIONS] NETWORK

option 说明：

--attachable Enable manual container attachment

--aux-address Auxiliary IPv4 or IPv6 addresses used by Network driver

--driver, -d Driver to manage the Network

--gateway IPv4 or IPv6 Gateway for the master subnet

--internal Restrict external access to the network

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

--ipam-driver IP Address Management Driver

--ipam-opt Set IPAM driver specific options

--ipv6 Enable IPv6 networking

--label Set metadata on a network

--opt, -o Set driver specific options

--subnet Subnet in CIDR format that represents a network segment

4、docker build: 从指定路径下的Dockerfile 中建立一个镜像：

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

option 说明：

--build-arg Set build-time variables

--cache-from Images to consider as cache sources

--cgroup-parent Optional parent cgroup for the container

--compress false Compress the build context using gzip

--cpu-period Limit the CPU CFS (Completely Fair Scheduler) period

--cpu-quota Limit the CPU CFS (Completely Fair Scheduler) quota

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

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

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

--disable-content-trust Skip image verification

--file, -f Name of the Dockerfile (Default is ‘PATH/Dockerfile’)

--force-rm Always remove intermediate containers

--isolation Container isolation technology

--label Set metadata for an image

--memory, -m Memory limit

--memory-swap Swap limit equal to memory plus swap: ‘-1’ to enable unlimited swap

--network Set the networking mode for the RUN instructions during build

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

--pull false Always attempt to pull a newer version of the image

--quiet, -q false Suppress the build output and print image ID on success

--rm true Remove intermediate containers after a successful build

--security-opt Security options

--shm-size Size of /dev/shm, default value is 64MB

--squash Squash newly built layers into a single new layer

--tag, -t Name and optionally a tag in the ‘name:tag’ format

--ulimit Ulimit options

5、Docker exec & attach：

docker进入容器进行操作的重要操作，类似ssh但是不需要开启sshd，更安全。一般使用 –it；

类似的有attach命令，但是attach使用Ctrl-c推出后container也停止运行

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

OPTIONS:

-d :分离模式: 在后台运行

-i :即使没有附加也保持STDIN 打开

-t :分配一个伪终端

6、docker pull：从 Docker Hub（或第三方源）拉取镜像。

命令用法：docker pull [OPTIONS] NAME[:TAG|@DIGEST]

TAG 是要拉取镜像的版本号或版本名。DIGEST 是镜像的哈希，用于防止篡改或下载到错误的镜像。NAME 是要拉取镜像的名称。

OPTION 可以设一些选项。-a 或 --all-tags 下载对应镜像的所有版本。--disable-content-trust 不校验哈希（这个选项默认是开启的）。

使用例子：docker pull debian:jessie，拉取 Debian 8。

三、创建一个基础镜像为ubuntu的docker镜像，随后再其中加入nginx服务器，之后启动nginx服务器并利用tail命令将访问日志输出到标准输出流。要求该镜像中的web服务器主页显示自己编辑的内容，编辑的内容包含学号和姓名。之后创建一个自己定义的network，模式为bridge，并让自己配的web服务器容器连到这一网络中。要求容器所在宿主机可以访问这个web服务器搭的网站。请在报告中详细阐述搭建的过程和结果。

1、创建一个基础镜像为 ubuntu 的 docker 镜像：

$ sudo docker run -i -t --name assignment -p 9999:80 ubuntu /bin/bash

设置容器名称为assignment，将容器的 80 端口映射到 host 9999 端口。执行命令后系统会自动 pull 缺少的ubuntu:latest。

2、加入 nginx 服务器：

$ apt update

$ apt install nginx -y

$ apt install vim

$ nginx

3、编辑 web 服务器主页：

$ cd /var/www/html/

$ vim index.nginx-debian.html

修改服务器主页，显示姓名和学号。

修改完成后，从主机访问 127.0.0.1:9999, 显示如下页面：



4、创建一个自己定义的 network：

$ exit

$ sudo docker commit assignment netimage

$ sudo docker run -d --name netserver -p 9999:80 netimage nginx -g 'daemon off;'

5、退出容器，保存镜像并创建新容器以运行新镜像：

$ sudo docker network create anetwork

$ sudo docker network connect anetwork netserver

$ sudo docker network inspect anetwork

创建自己定义的 network。

四、尝试让docker容器分别加入四个不同的网络模式:null,bridge,host,overlay。请查阅相关资料和docker文档，阐述这些网络模式的区别。

1、host模式 Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。但如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。 例如，我们在10.10.101.105/24的机器上用host模式启动一个含有web应用的Docker容器，监听tcp80端口。当我们在容器中执行任何类似ifconfig命令查看网络环境时，看到的都是宿主机上的信息。而外界访问容器中的应用，则直接使用10.10.101.105:80即可，不用任何NAT转换，就如直接跑在宿主机中一样。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

host 模式的优点在于，容器可以直接使用宿主机的 IP 地址与外界通信，同时容器内服务的端口也可以使用宿主机的端口，无需进行额外的 NAT 转换。 但是容器不再拥有隔离、独立的网络栈也带来了一些问题。 另外，host 模式的容器虽然可以令其内部的服务和传统情况无差别、无改造的使用，但是由于网络隔离性的弱化，该容器会与宿主机共享竞争网络栈的使用；另外，容器内部将不再拥有所有的端口资源，因为部分端口资源可能已经被宿主机本身的服务占用。

通过以下命令启动 host 模式的 container:

$ docker run -it --network host ubuntu /bin/bash

2、bridge模式 bridge模式是Docker默认的网络设置，此模式会为每一个容器分配Network Namespace、设置IP等，并将一个主机上的Docker容器连接到一个虚拟网桥上。 当Docker server启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的Docker容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。接下来就要为容器分配IP了，Docker会从RFC1918所定义的私有IP网段中，选择一个和宿主机不同的IP地址和子网分配给docker0，连接到docker0的容器就从这个子网中选择一个未占用的IP使用。如一般Docker会使用172.17.0.0/16这个网段，并将172.17.42.1/16分配给docker0网桥。

通过以下命令启动 bridge 模式的 container:

$ docker run -i -t mysql:latest /bin/bash

$ docker run -i -t --net="bridge" mysql:latest /bin/bash

3、null：

这一模式将容器放置在它自己的网络栈中，但是不进行任何配置。实际上，null 模式关闭了容器的网络功能，在以下情况下可以应用这一模式：1、容器并不需要 network 时（例如只需要写磁盘卷的批处理任务）；2、用户希望自定义网络时。 可以通过以下命令启动 null 模式的 container：

$ docker run -it --network none ubuntu /bin/bash

4、overlay：

这一模式使用 docker 内置的 swarm 来管理结点，首先在第一台主机上输入docker swarm init，便会创建一个 swarm 的管理结点。 之后在其他主机上输入docker swarm join，即可将它们加入 worker 之中，在第一台主机可以通过docker node ls查看这些 worker.

overlay 模式主要用于 docker 服务和集群的创建。相比于只能在本地网络中访问的模式，overlay网络通过一个新的网段来管理一个集群，通过注册的方式来发现新结点，避免了普通模式下跨主机通讯的繁琐过程。

五、阅读mesos中负责与docker交互的代码，谈谈mesos是怎样与docker进行交互的，并介绍docker类中run函数大致做了什么。

代码位于mesos-1.1.0/src/docker文件夹中：

1、docker.cpp

docker.hpp 头文件中定义了 Docker 类，该类内部又定义了 Container 和 Image 两个类。docker.cpp 封装了一些 docker 使用的 API：

create创建容器或者创建镜像

run运行docker

stop停止运行 docker

kill杀死docker

rm删除docker

2、executor.cpp：

实现了 mesos framework 的一个 executor, 调用docker.cpp中的一系列 API 管理 docker.

3、spec.cpp：

负责解析 JSON 格式的 INFO 信息。

4、run 函数：

检查docker info 是否存在并获取 docker info；

添加命令行参数；

分配 CPU 资源，并添加环境变量；

检查磁盘挂载信息，设置命令行参数；

--net网络配置，将sandbox目录映射到mapped；

检查和重写 entrypoint；

添加容器名和指定镜像名；

添加运行容器后的命令和参数；

运行容器。

六、写一个framework，以容器的方式运行task，运行前面保存的nginx服务器镜像，网络为HOST，运行后，外部主机可以通过访问宿主ip+80端口来访问这个服务器搭建的网站，网站内容包含学号和姓名。报告中对源码进行说明，并附上源码和运行的相关截图。

建立framework发送docker命令：

from \_\_future\_\_ import print\_function

import sys

import uuid

import time

import socket

import signal

import getpass

from threading import Thread

from os.path import abspath, join, dirname

from pymesos import MesosSchedulerDriver, Scheduler, encode\_data

from addict import Dict

TASK\_CPU = 1

TASK\_MEM = 256

TASK\_NUM = 1

class sche(Scheduler):

def resourceOffers(self, driver, offers):

filters = {'refuse\_seconds': 5}

for offer in offers:

cpu = self.getResource(offer.resources, 'cpus')

mem = self.getResource(offer.resources, 'mem')

if cpu < TASK\_CPU or mem < TASK\_MEM:

continue

#information of container

ContainerInfo = Dict()

ContainerInfo.type = 'DOCKER'

ContainerInfo.docker = DockerInfo

#information of docker

DockerInfo = Dict()

DockerInfo.image = 'netimage'

DockerInfo.network = 'HOST'

#information of nginx command

CommandInfo = Dict()

CommandInfo.shell = False

CommandInfo.value = 'nginx'

CommandInfo.arguments = ['-g', "'daemon off;'"]

#set the task

task = Dict()

task\_id = str(uuid.uuid4())

task.task\_id.value = task\_id

task.agent\_id.value = offer.agent\_id.value

task.name = 'nginx'

task.container = ContainerInfo

task.command = CommandInfo

task.resources = [

dict(name='cpu', type='SCALAR', scalar={'value': TASK\_CPU}),

dict(name='mem', type='SCALAR', scalar={'value': TASK\_MEM}),

]

#launch the task

driver.launchTasks(offer.id, [task], filters)

break

def getResource(self, res, name):

for r in res:

if r.name == name:

return r.scalar.value

return 0.0

def statusUpdate(self, driver, update):

logging.debug('Status update TID %s %s', update.task\_id.value, update.state)

def main(master):

framework = Dict()

framework.user = getpass.getuser()

framework.name = "sche"

framework.hostname = socket.gethostname()

driver = MesosSchedulerDriver(

sche(),

framework,

master,

use\_addict=True,

)

def signal\_handler(signal, frame):

driver.stop()

def run\_driver\_thread():

driver.run()

driver\_thread = Thread(target=run\_driver\_thread, args=())

driver\_thread.start()

print('Scheduler running, Ctrl+C to quit.')

signal.signal(signal.SIGINT, signal\_handler)

while driver\_thread.is\_alive():

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

import logging

logging.basicConfig(level=logging.DEBUG)

if len(sys.argv) != 2:

print("Usage: {} <mesos\_master>".format(sys.argv[0]))

sys.exit(1)

else:

main(sys.argv[1])

代码主要完成了docker, container, task 等的基本属性设定和启动工作；

运行framework，访问。