

Docker 预研报告

本文记述了应用 Docker 容器技术构建一个简单地分布式应用的过程，以及在构建应用过程中对 Docker 容器技术的理解。

基本概念

Docker 引擎

Docker 引擎（Docker Engine¹）是一个客户端/服务器结构的应用程序，包含两个组件：

- 1) 服务守护进程（Docker Daemon）
- 2) 命令行客户端（Command Line Interface）

服务守护进程是操作系统中的一个守护进程，默认开机启动。命令行客户端则是以 docker 命令的形式提供。如图 1，命令行客户端通过一套 REST API 访问服务守护进程，管理镜像、容器、网络、数据卷等资源。

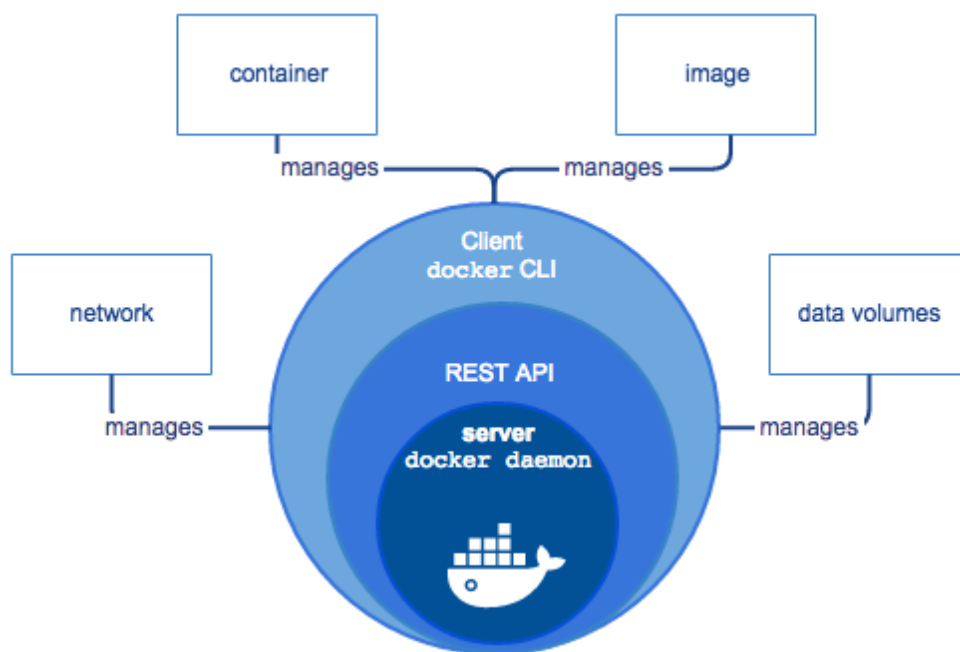


图 1 Docker 引擎

Docker 仓库

Docker 仓库（Docker Registry²）是一个用来存储以及分发镜像的服务侧应用程序。Docker 引擎和仓库之间的关系如图 2 所示。命令行客户端提供 build 命令构建镜像，提供 push、

pull 命令向仓库推送、拉取镜像，提供 run 命令根据镜像生成容器。命令行客户端只是所有这些服务的请求者，而服务守护进程才是这些服务的执行者。若有需要，可以根据 REST API 文档构建自己的客户端程序。Portainer³ 就是一款开源的 Docker Web 客户端程序。本文使用 Portainer 辅助展示应用构建过程。

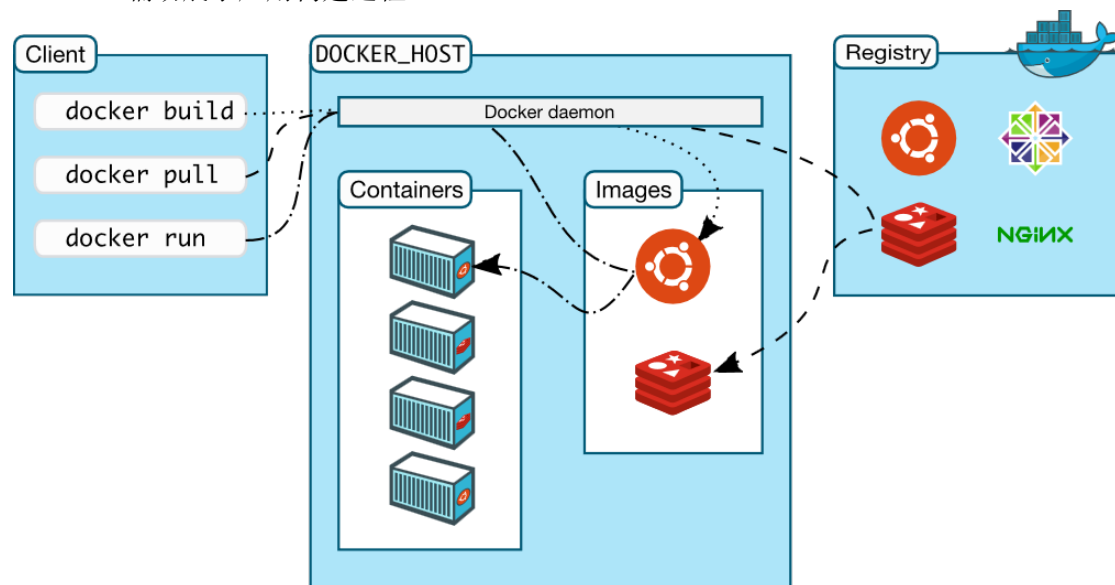


图 2 Docker Architecture

Docker Swarm

Docker Swarm³ 是 Docker 内生支持的集群模式。一组运行了 Docker 引擎的主机通过内在的通信机制可以构成一个计算集群。集群中的每台主机被称为节点（Swarm Node）。节点有两种类型：管理员（Swarm Manager）和工人（Swarm Worker）。管理员类型的节点可以管理集群中的节点、服务和容器等计算资源，可以将工人类型的节点提升为管理员，也可以将其他管理员节点下降为工人。

管理员类型的节点通过命令 `docker swarm` 实现对集群的管理。而工人节点只是单纯地为集群提供计算能力，除了管理自己（比如加入、退出集群等）之外，不能访问集群以及其他节点的任何信息。集群相关的命令有 `docker swarm`、`docker service`、`docker node`。

此外，Docker 还推出了 Docker Compose、Docker Machine 两款工具。下面简单介绍应用构建过程中对这些工具使用的理解。

工具介绍

在下文构建基于 Docker 容器技术的分布式应用的过程中，使用了 Docker 命令行、Docker Compose 和 Docker Machine 三款工具。Docker Compose 和 Docker Machine 需要独立安装，并不与 Docker 引擎一起安装。Portainer 虽然功能强大，只是为方便展示构建过程。

Docker 命令行

Docker 命令行⁴ 也就是 `docker` 命令，主要功能有创建和管理镜像等（图 1）、创建和管

理 **swarm** 集群。常用的命令有（未列出选项参数）：

- 1) `docker build`，创建镜像
- 2) `docker push`，将镜像推送到仓库中
- 3) `docker pull`，从仓库中拉取镜像
- 4) `docker search`，在仓库中搜索镜像
- 5) `docker image`，管理本节点镜像
- 6) `docker container`，管理本节点容器
- 7) `docker run`，根据镜像创建容器
- 8) `docker swarm init`，节点以管理员的角色开启 **swarm** 模式
- 9) `docker swarm join`，节点以管理员或者工人的角色加入 **swarm** 集群
- 10) `docker swarm leave`，节点脱离 **swarm** 集群
- 11) `docker service`，管理 **swarm** 集群中运行的多容器多主机服务
- 12) `docker node`，管理 **swarm** 集群中的节点

Docker Compose

Docker Compose⁵ 是 **Docker** 推出的用于定义、运行和管理多容器服务的工具，以命令行工具 `docker-compose` 的方式提供使用。多容器服务是指一个服务由若干容器子功能组成，这些容器子功能互相协作完成服务的整体功能。

创建多容器服务的第一步是定义 **YAML** 服务编排文件（**Compose File**⁹）。服务编排文件定义了各容器的镜像、容器使用的计算资源（**cpu**、内存等）、容器使用的存储以及网络资源。在定义好服务编排文件之后，在命令行模式下进入到文件保存目录，执行 `docker-compose up` 命令启动服务。`docker-compose` 还提供了其他的子命令可以方便地管理服务。

简单理解，**Docker Compose** 以 `docker-compse` 命令和 **YAML** 服务编排文件的方式简化了多容器服务的定义和管理，提供了比 `docker` 命令原语更高一级的容器服务操作接口。

Docker Machine

Docker Machine⁶ 是 **Docker** 推出的方便于创建、管理安装了 **Docker** 引擎的虚拟机以及操作虚拟机的工具，以命令行工具 `docker-machine` 的方式提供使用。常用的命令有（未列出选项参数）：

- 1) `docker-machine create --driver`，以指定的虚拟机技术创建虚拟主机
- 2) `docker-machine ssh`，使用 **SSH** 协议登陆虚拟机或者在虚拟机环境执行命令

Portainer

Portainer⁷ 是一个开源的 **Docker** 环境管理 **Web** 客户端程序。**Portainer** 可以管理安装了 **Docker** 引擎的主机、添加 **Docker** 仓库（包括 **Docker Hub** 和私有仓库）、上传下载镜像到 **Docker** 仓库、管理 **Swarm** 集群以及管理用户。本文主要使用 **Portaniner** 展示构建过程中 **Docker** 主机以及 **Swarm** 集群的变化。

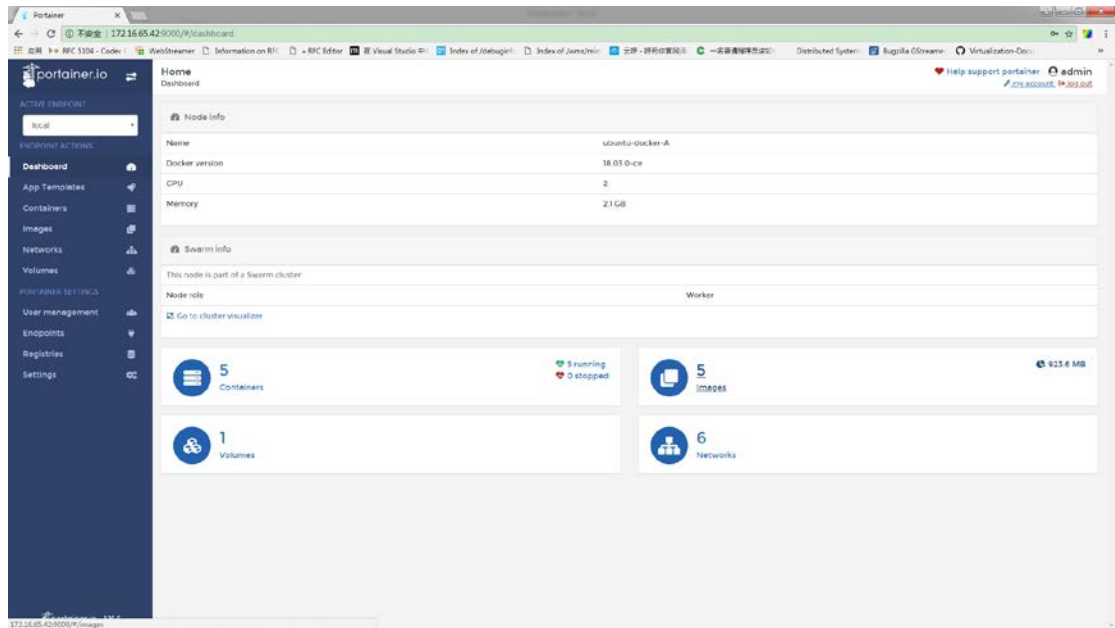


图 3 Portainer Web UI

镜像与服务定义

基于 Docker 容器技术的分布式应用构建涉及两种定义文件：镜像构建定义文件 **Dockfile**、YAML 服务编排文件（.yml）。Docker 对这两种定义文件的格式提供了详细的参考文档，这里作简单介绍。

Dockfile

Dockfile⁸ 是 Docker 提供的用于定义镜像构建过程的文本文件。每次创建新的 Docker 镜像，用户首先需要编写符合规格的 Dockerfile 文件。该文件是由若干 Dockerfile 命令组成的，并且必须以 FROM 命令开始。FROM 命令指定了目标镜像构建的基础镜像（Base Image）。基础镜像为后续 Dockerfile 命令提供了上下文环境。任何有效的镜像都可以作为基础镜像。Dockerfile 定义好之后，用户使用 docker build 创建镜像。

YAML Compose File

见 Docker Compose 中的介绍（见上方）。

分布式应用构建

下文通过一个简单的分布式应用的构建展示如何使用 Docker 开发分布式应用。该分布式应用由两个服务组成：Web 服务、数据库服务。Web 服务基于 FLASK 框架（一个 Python 编写的 Web 微框架）实现，数据库服务采用 Redis。Redis 已经推出官方 Docker 镜像，需要创建 Web 服务镜像（见下方）。

该应用已部署在小组长拷环境中，可通过 Portainer (<http://172.16.65.42:9000>) 查看。

环境搭建

分布式应用运行在两台计算机组成的 Docker Swarm 集群中。配置如下，

- 1) 主机：ubuntu-docker-A (172.16.65.42)、ubuntu-docker-C (172.16.65.44)
- 2) 操作系统：Ubuntu 16.04.2 x86_64 (xenial)
- 3) Docker：18.03.0~ce-0~ubuntu_amd64

私有仓库搭建

通过 Docker 发布的仓库镜像 registry，搭建私有镜像仓库。

- 1) 通过 PuTTY¹¹ SSH 登陆主机 ubuntu-docker-A
- 2) 运行 registry 镜像启动仓库容器

```
docker run -d -p 5000:5000 --restart=always --name private-hub \
-e REGISTRY_STORAGE_DELETE_ENABLED=true \
-v /var/docker/registry:/var/lib/registry registry
```
- 3) 访问 http://172.16.65.42:5000/v2/_catalog，验证仓库容器创建成功（如图 4）。



图 4 私有镜像仓库

- 4) 运行 hyper/docker-registry-web 镜像启动仓库内容浏览容器

```
docker run -d -p 8080:8080 --restart=always --name private-hub-webui \
--link private-hub -e REGISTRY_NAME=172.16.65.42:5000 \
-e REGISTRY_URL=http://172.16.65.42:5000/v2 hyper/docker-registry-web
```
- 5) 访问 <http://172.16.65.42:8080/>，验证仓库内容浏览容器创建成功（如图 5）。

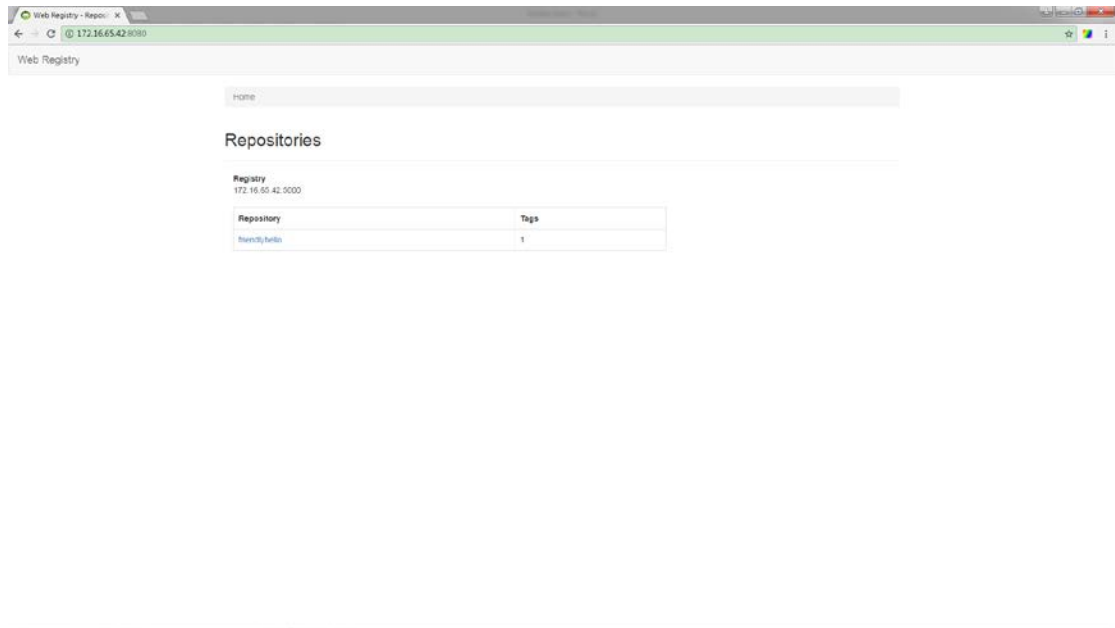


图 5 私有仓库浏览 Web UI

Portainer 管理前端

- 1) 通过 PuTTY SSH 登陆主机 ubuntu-docker-A
- 2) 运行 portainer/portainer 镜像启动 Portainer 容器
docker run -d -p 9000:9000 --restart always --name docker-webmanager \\\n-v /var/run/docker.sock:/var/run/docker.sock \\\n-v /var/docker/portainer:/data portainer/portainer
- 3) 访问 <http://172.16.65.42:9000>, 进入 Portainer 登陆主页(用户名 admin 密码 admin123)

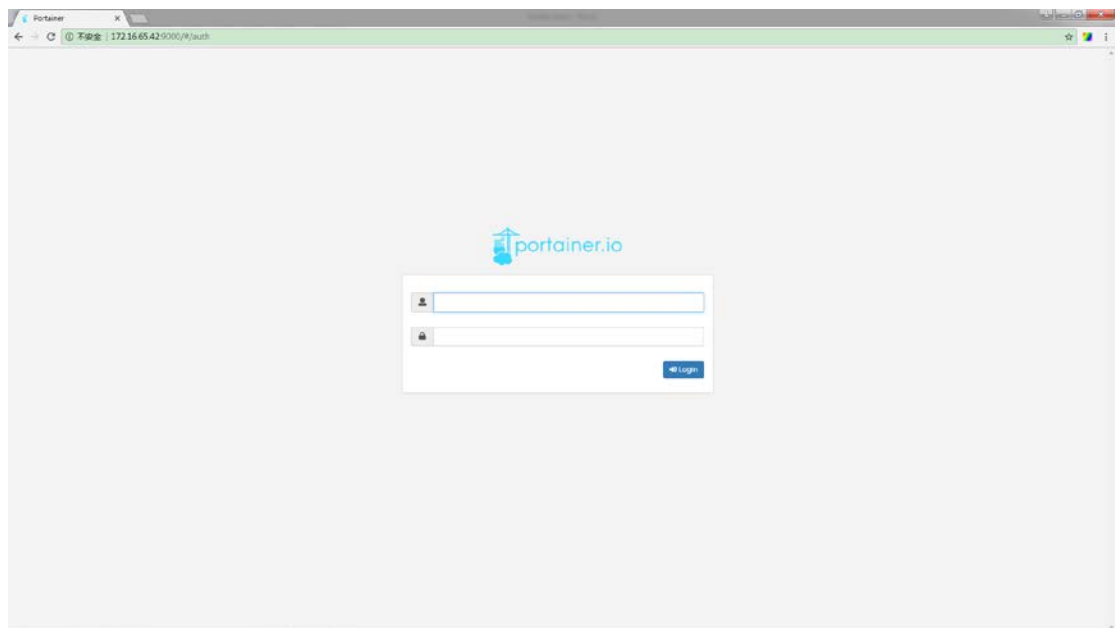


图 6 Portainer 登陆主页

- 4) 开启 Docker Daemon 端口 2375 TCP 监听¹²

5) 向 Portainer 中添加 Docker 引擎主机（Portainer 称为 Endpoint），如图 7。

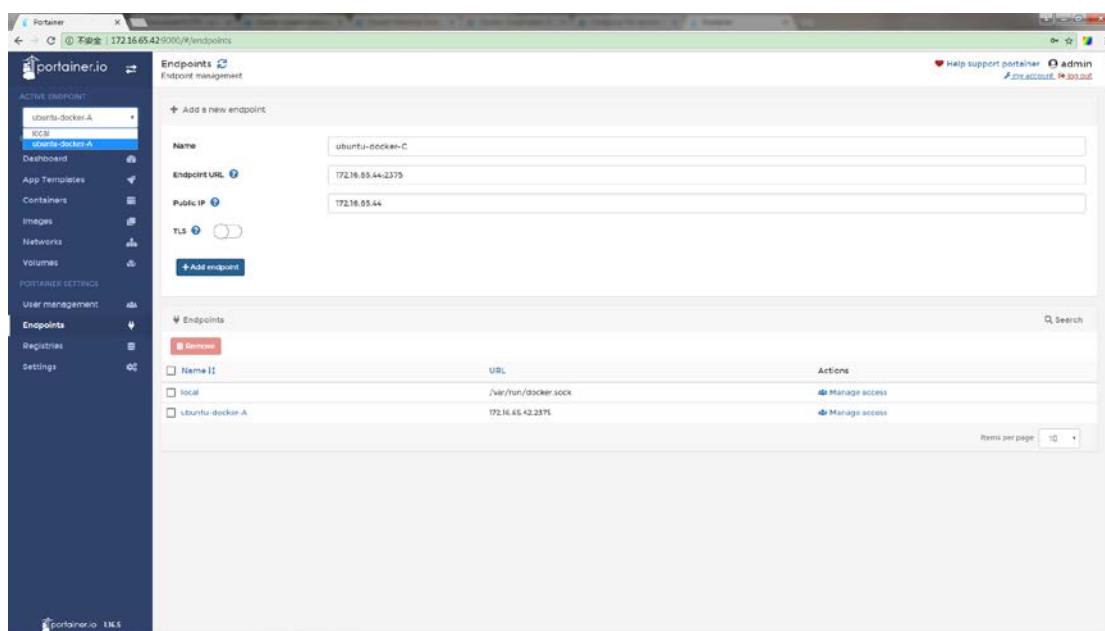


图 7 Portainer Docker 主机管理

6) 向 Portainer 添加私有仓库（如图 8）

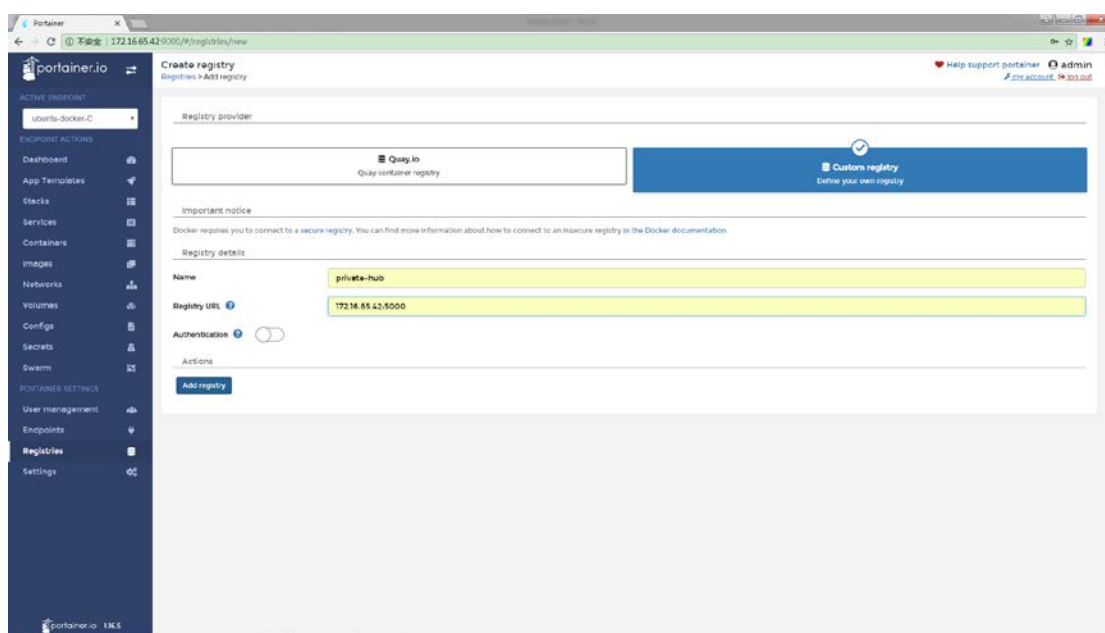


图 8 Portainer 私有仓库管理

7) 浏览 Portainer

通过 Portainer 页面左侧的快速访问标签，可以查看指定主机的容器、镜像、网络、数据卷等资源信息，以及所有已添加主机、私有仓库信息。除此之外，还可以通过 Portainer 创建镜像、向仓库推送拉取镜像、创建容器、创建网络、创建数据卷。这里不一一演示，可访问长拷机试验（<http://172.16.65.42:9000>）。

创建 Web 服务镜像

1) Dockerfile 定义

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

图 9 Web 服务镜像 Dockerfile

命令说明：

- i. FROM，指定 Web 服务镜像的基础镜像为 python，版本为 2.7-slim。
- ii. WORKDIR，为其后的 ADD、RUN、CMD 命令（还包括 ENTRYPOINT、COPY 命令，这里未使用）设置工作目录/app。
- iii. ADD，将 Docker 主机中目录“.”（也即当前目录）的内容添加到镜像文件系统目录/app 中。

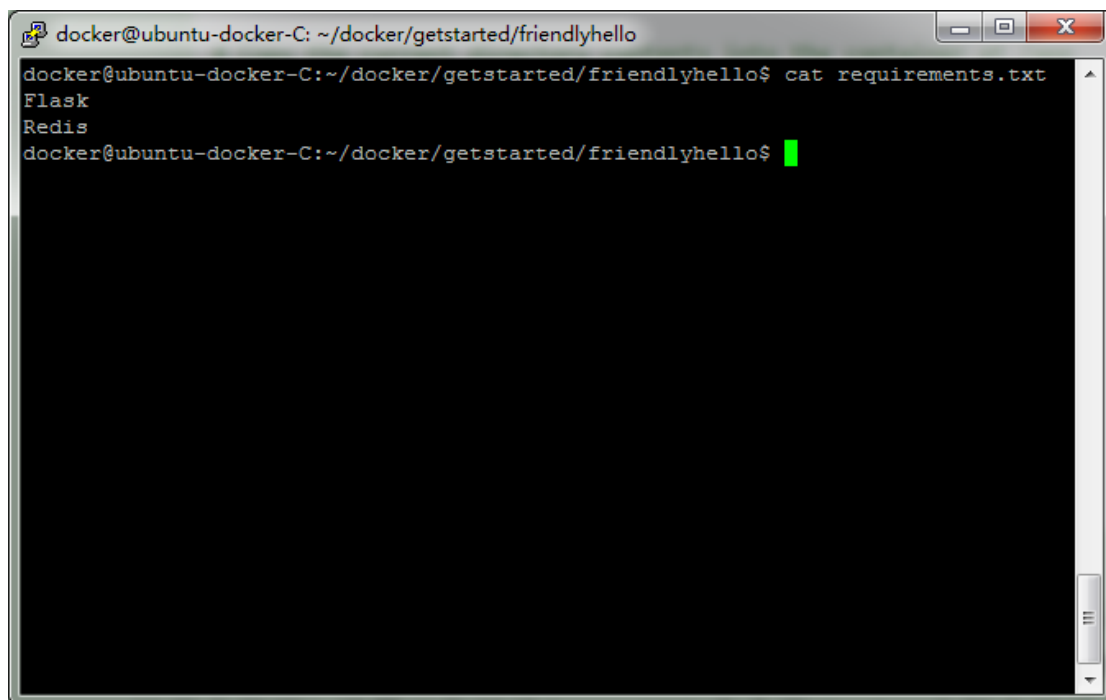
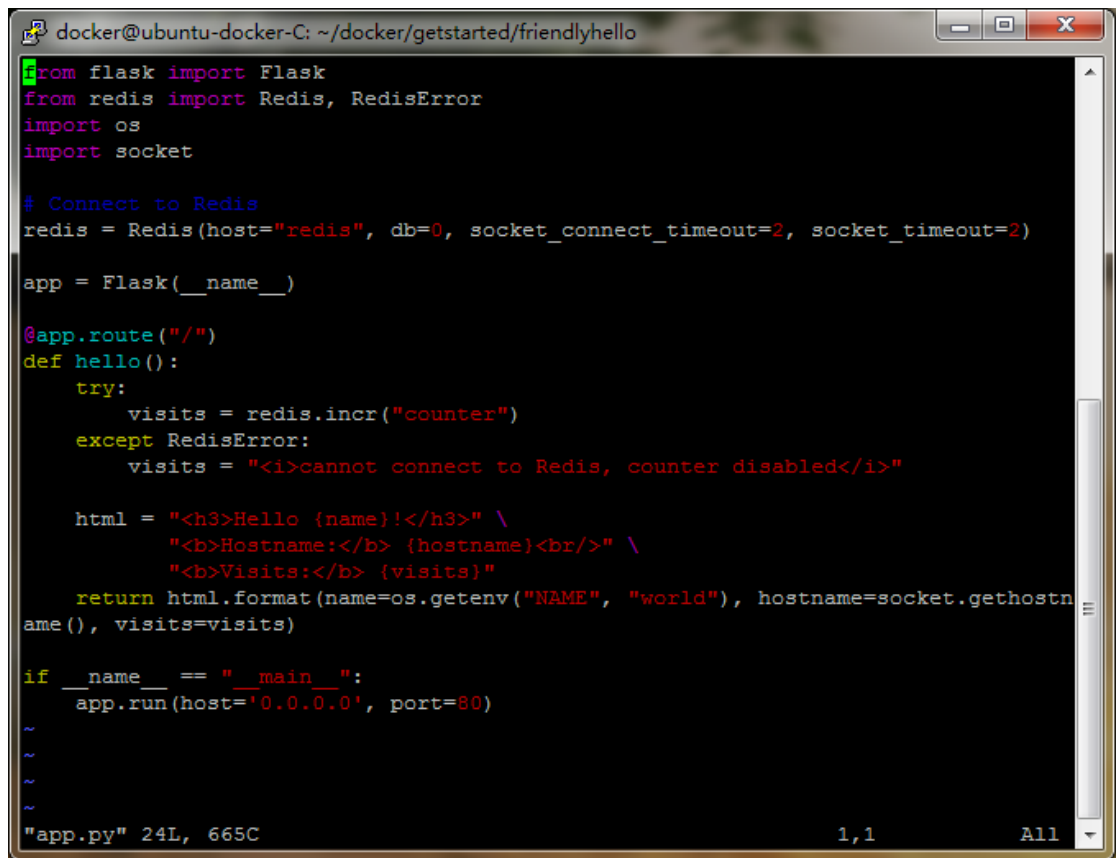


图 10 requirements.txt

- iv. RUN，在当前镜像的顶层执行指定命令。在这里是安装

requirements.txt (如图 10) 指定的 python 模块: FLASK、Redis。FLASK 是轻量级 Web 框架, Redis 是 python 语言 redis 客户端。

- v. EXPOSE, 指定容器的监听端口, 默认 TCP 协议。因此 Web 服务容器监听端口 80。
- vi. ENV, 设置环境变量 NAME 值为 “World”, 对后续所有指令有效。
- vii. CMD, 执行 shell 命令。这里是运行 python 程序 app.py (如图 11)。程序 app.py 实现了一个简单的 Web 服务。该服务工作在 80 端口, 返回给客户端一段文本字符串。在该字符串中, 包含了环境变量、服务主机名以及页面访问次数。环境变量展示了 Dockerfile 指令 ENV 是有效的。页面访问次数是保存在 Redis 数据库中的, 因此要使用 Redis 服务。



```
docker@ubuntu-docker-C: ~/docker/getstarted/friendlyhello
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

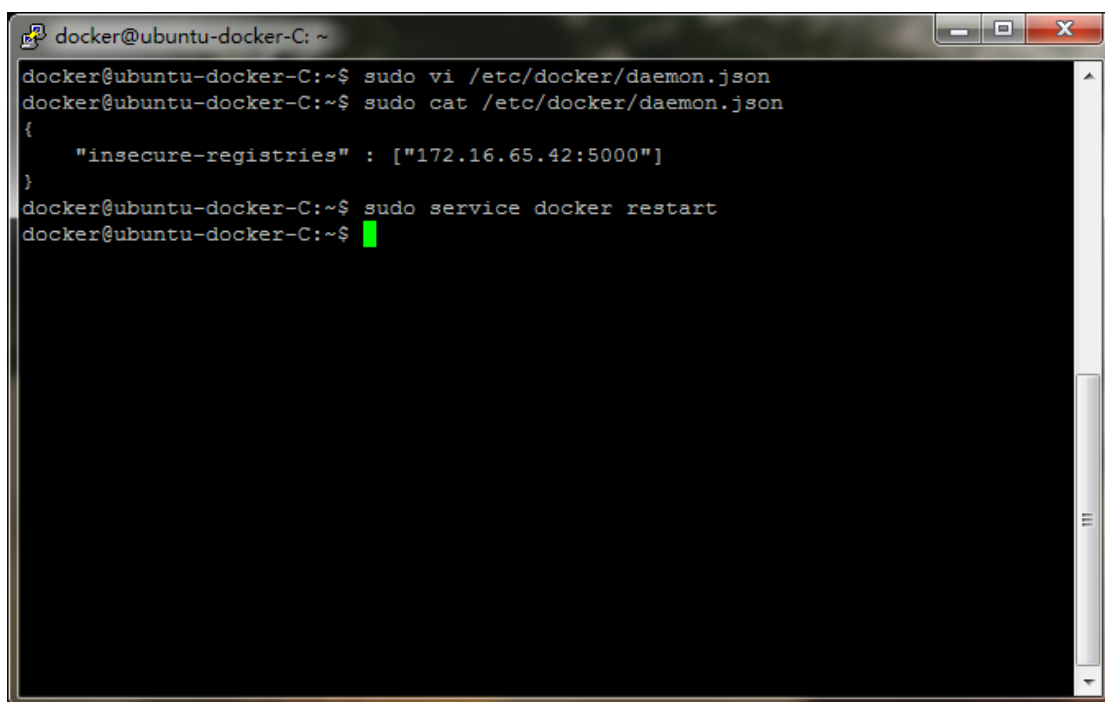
    html = "<h3>Hello {name}!</h3> \\"
        "<b>Hostname:</b> {hostname}<br/> \\"
        "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)

~
~
~
~
"app.py" 24L, 665C 1,1 All
```

图 11 app.py

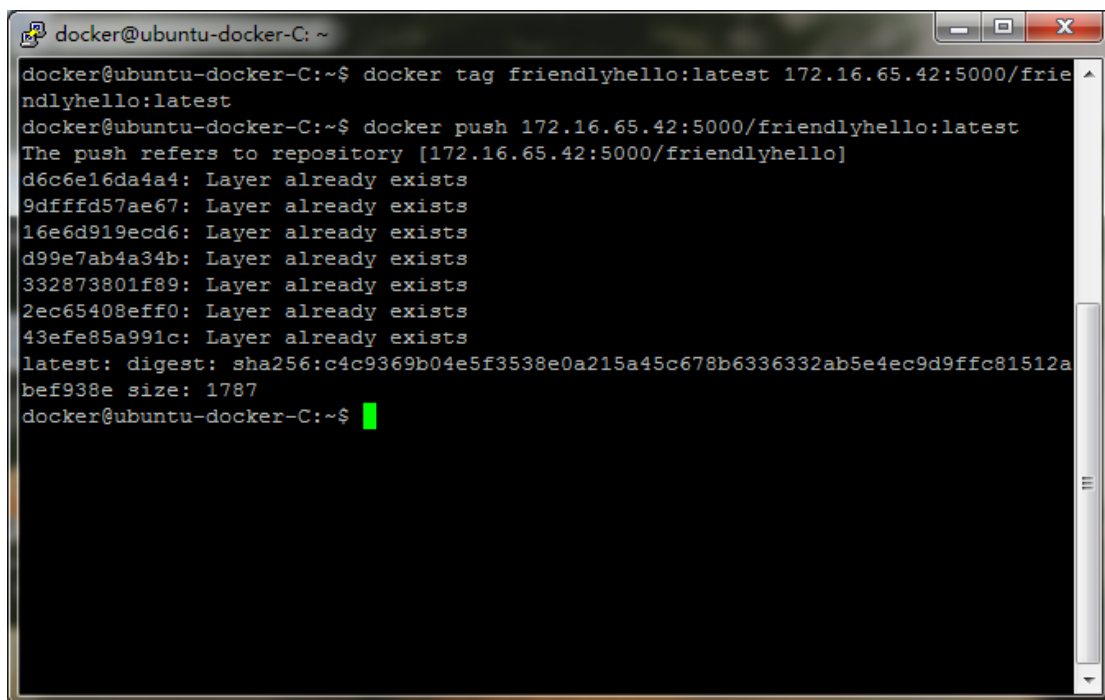
- 2) 创建镜像
docker build -t friendlyhello
- 3) 配置 Docker Daemon (如图 12) 以向私有仓库上传镜像

A terminal window titled 'docker@ubuntu-docker-C: ~' with standard window controls. It shows the following commands and output:

```
docker@ubuntu-docker-C:~$ sudo vi /etc/docker/daemon.json
docker@ubuntu-docker-C:~$ sudo cat /etc/docker/daemon.json
{
    "insecure-registries" : ["172.16.65.42:5000"]
}
docker@ubuntu-docker-C:~$ sudo service docker restart
docker@ubuntu-docker-C:~$
```

图 12 Docker Deamon 配置向仓库上传镜像

4) 上传镜像（如图 13）

A terminal window titled 'docker@ubuntu-docker-C: ~' with standard window controls. It shows the following commands and output:

```
docker@ubuntu-docker-C:~$ docker tag friendlyhello:latest 172.16.65.42:5000/friendlyhello:latest
docker@ubuntu-docker-C:~$ docker push 172.16.65.42:5000/friendlyhello:latest
The push refers to repository [172.16.65.42:5000/friendlyhello]
d6c6e16da4a4: Layer already exists
9dffd57ae67: Layer already exists
16e6d919ecd6: Layer already exists
d99e7ab4a34b: Layer already exists
332873801f89: Layer already exists
2ec65408eff0: Layer already exists
43efe85a991c: Layer already exists
latest: digest: sha256:c4c9369b04e5f3538e0a215a45c678b6336332ab5e4ec9d9ffc81512abef938e size: 1787
docker@ubuntu-docker-C:~$
```

图 13 使用 docker 命令上传镜像到仓库

访问 <http://172.16.65.42:8080/> 查看镜像仓库内容, 图 14 显示镜像已经上传成功。

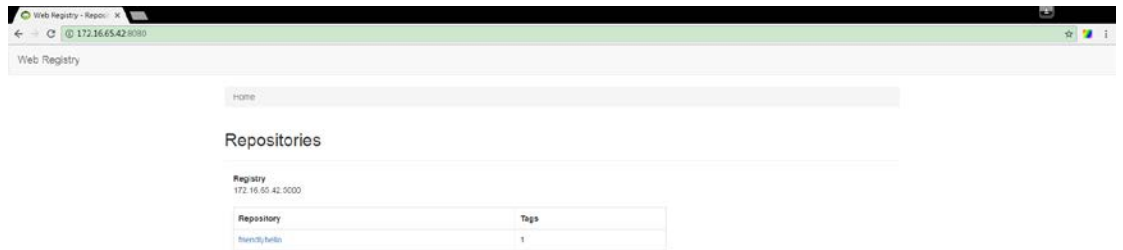


图 14 私有仓库镜像列表

5) 运行镜像启动 Web 服务容器

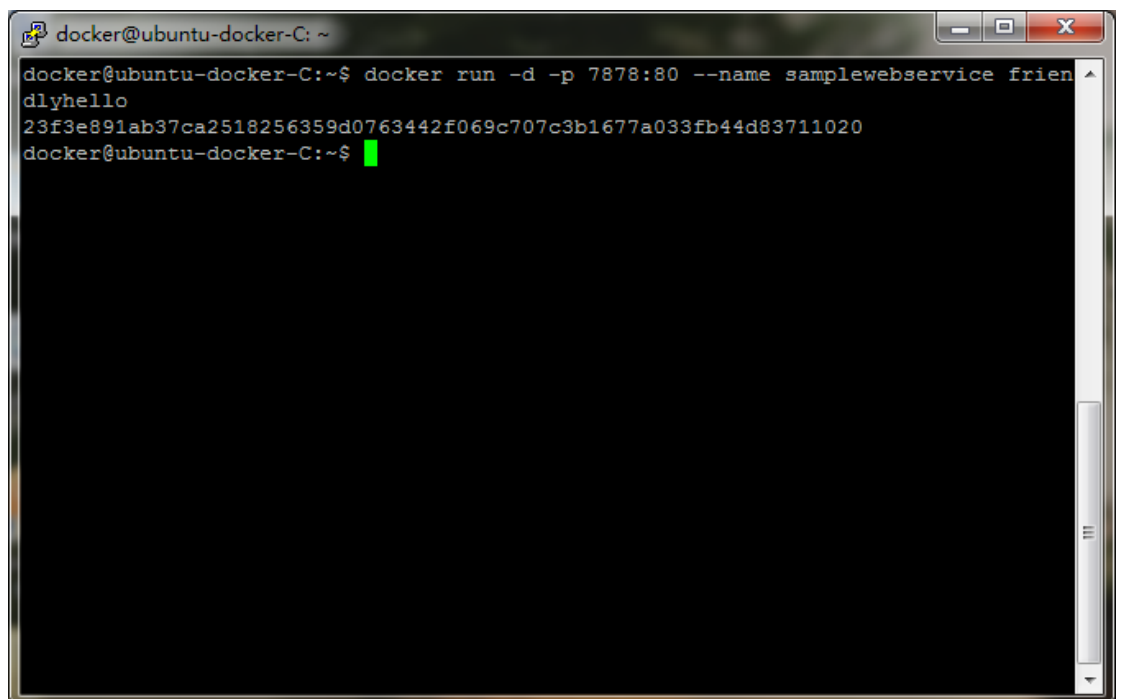


图 15 运行 Web 服务容器

在浏览器地址栏输入 <http://172.16.65.44:7878/> 访问本节定义的 Web 服务。返回页面中 “cannot connect to Redis counter disabled”，指示 Web 服务不能访问 Redis 服务，页面访问次数值无效。下面小节中同时启动 Web 服务和 Redis 服务时，页面访问次数数字段会显示正确的结果。



图 16 浏览 Web 服务

服务编排

1) YAML 服务编排文件定义

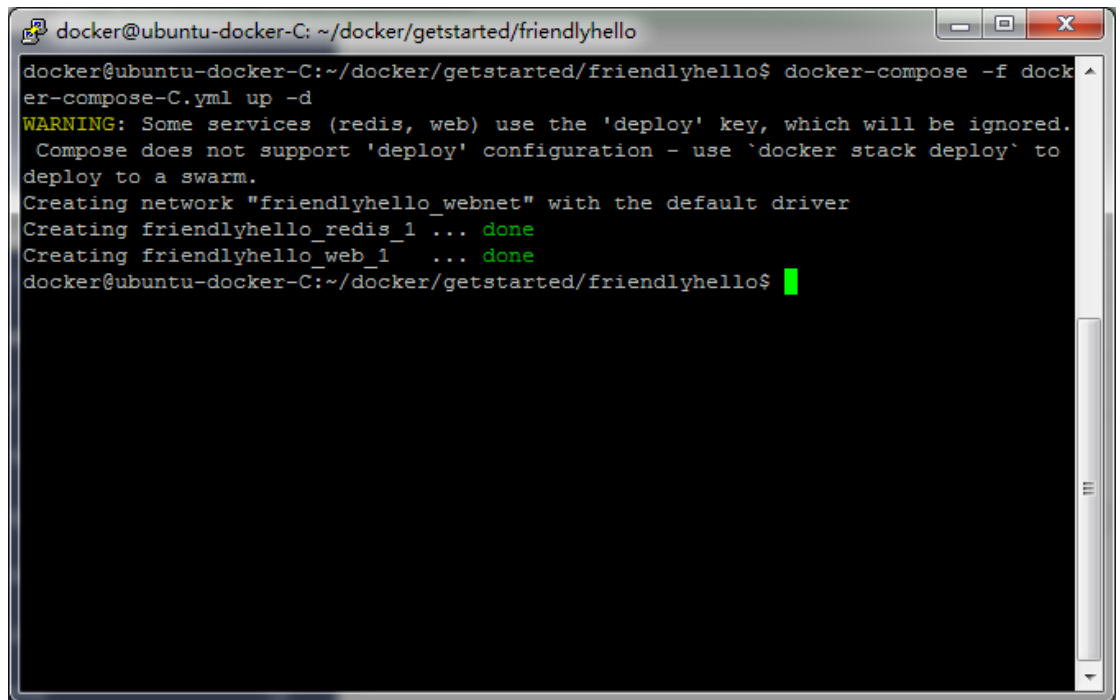


图 17 YAML 服务编排文件

图 17 的服务编排文件定义了两个服务：web、redis。

- i. 字段 `version` 指示编排文件遵守版本 3。
- ii. **web 服务**： `image` 字段指定容器镜像为上节创建并已上传到私有仓库的镜像； `deploy` 字段指定了容器复制个数（2 个实例）、计算资源约束（cpu 使用 0.2，内存 100M）和重启策略（运行失败时）； `ports` 字段指定容器端口 80 映射到主机端口 6767； `networks` 指定容器使用默认网络配置。
- iii. **redis 服务**： `command` 字段指示 redis 服务开启数据持久化； `volumes` 字段指示容器目录 `/data` 映射到主机目录 `/var/lib/docker/data` 上（持久化数据保存在该主机目录中，即使容器重启持久化数据也不会丢失）； 其他与 **web** 服务类似。

2) 启动服务



```
docker@ubuntu-docker-C: ~/docker/getstarted/friendlyhello
docker@ubuntu-docker-C:~/docker/getstarted/friendlyhello$ docker-compose -f dock
er-compose-C.yml up -d
WARNING: Some services (redis, web) use the 'deploy' key, which will be ignored.
Compose does not support 'deploy' configuration - use 'docker stack deploy' to
deploy to a swarm.
Creating network "friendlyhello_webnet" with the default driver
Creating friendlyhello_redis_1 ... done
Creating friendlyhello_web_1    ... done
docker@ubuntu-docker-C:~/docker/getstarted/friendlyhello$
```

图 18 docker-compose 启动服务

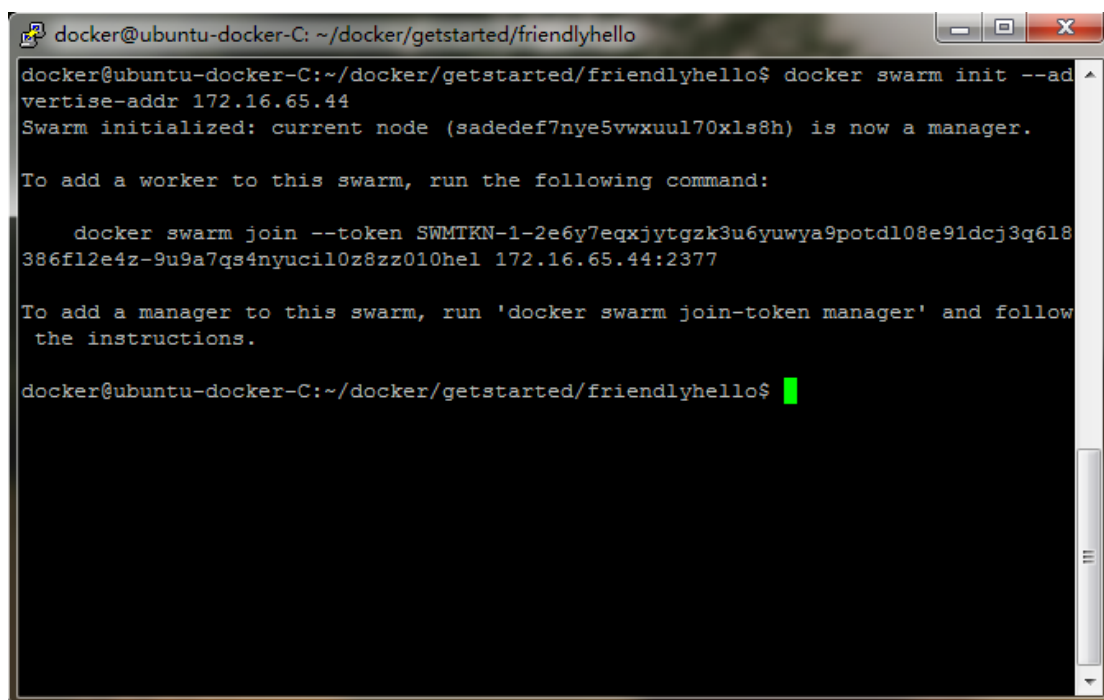


图 19 访问 docker-compose 所启动服务

图 18 中警告 deploy 字段无效。这是因为 deploy 字段需要在 Swarm 模式下调用 docker stack deploy 命令部署服务时才会生效。访问 <http://172.16.65.44:6767/>，显示 web 服务、redis 服务都已经启动。

Swarm 服务栈

1) 启动 swarm 模式



```
docker@ubuntu-docker-C: ~/docker/getstarted/friendlyhello
docker@ubuntu-docker-C:~/docker/getstarted/friendlyhello$ docker swarm init --advertise-addr 172.16.65.44
Swarm initialized: current node (sadedef7nye5vwxuul70xls8h) is now a manager.

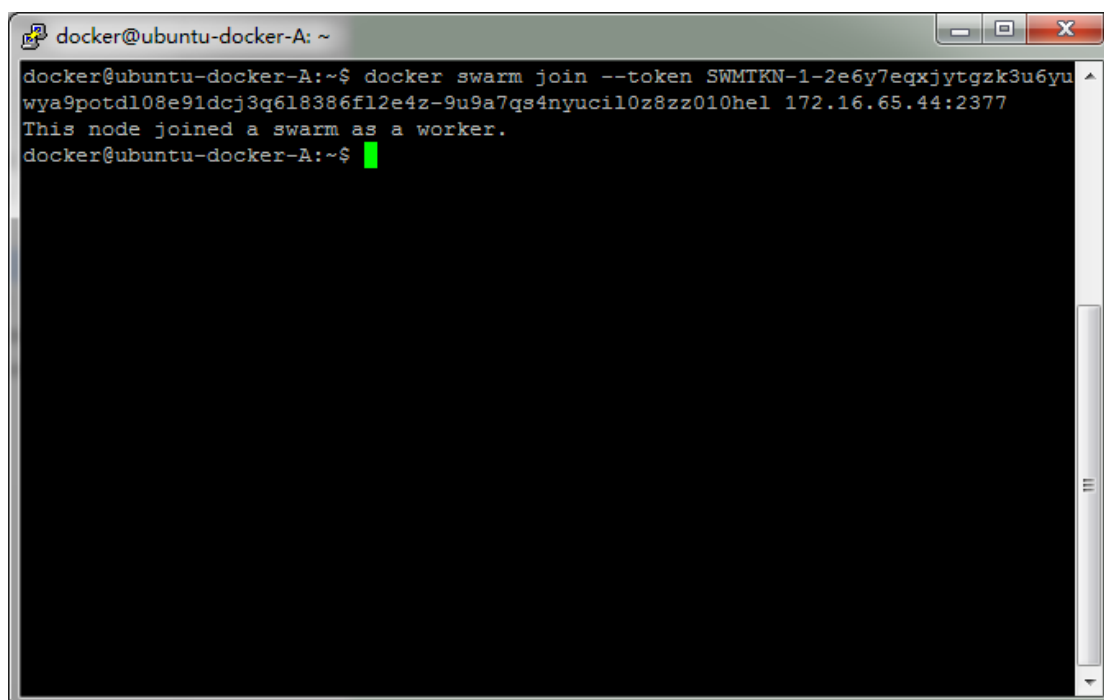
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-2e6y7eqxjytgzk3u6yuwya9potdl08e91dcj3q6l8386fl2e4z-9u9a7qs4nyucil0z8zz010hel 172.16.65.44:2377

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

docker@ubuntu-docker-C:~/docker/getstarted/friendlyhello$
```

图 20 Swarm Manager (ubuntu-docker-C)



```
docker@ubuntu-docker-A: ~
docker@ubuntu-docker-A:~$ docker swarm join --token SWMTKN-1-2e6y7eqxjytgzk3u6yuwya9potdl08e91dcj3q6l8386fl2e4z-9u9a7qs4nyucil0z8zz010hel 172.16.65.44:2377
This node joined a swarm as a worker.
docker@ubuntu-docker-A:~$
```

图 21 Swarm Worker (ubuntu-docker-A)

2) 访问 Portainer 查看 Swarm 模式状态

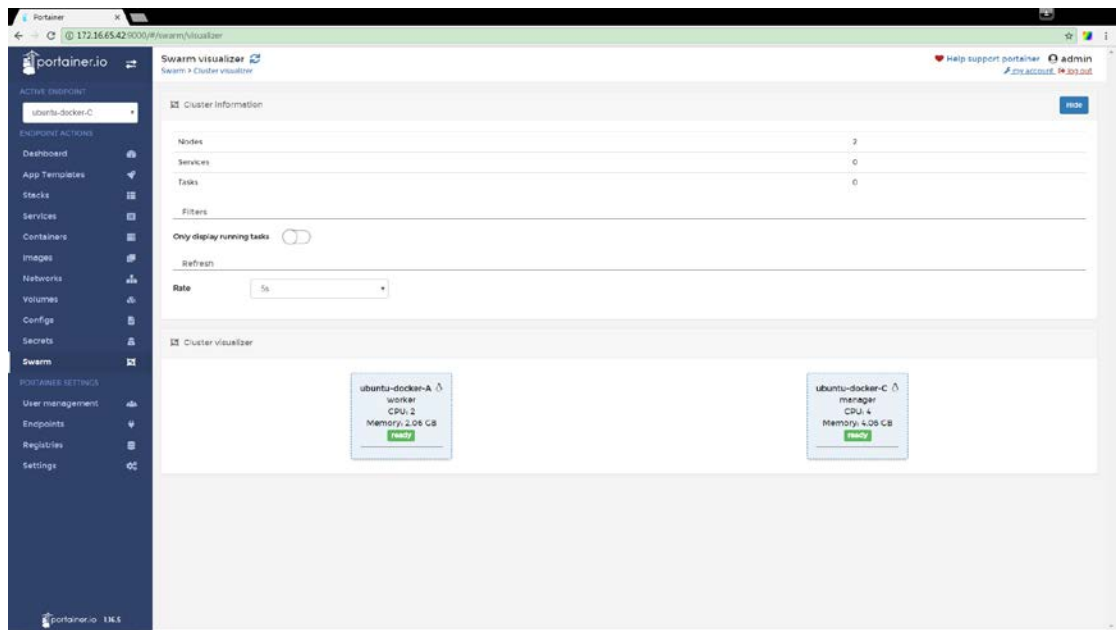


图 22 Portainer 展示 Swarm 模式

3) 部署服务栈

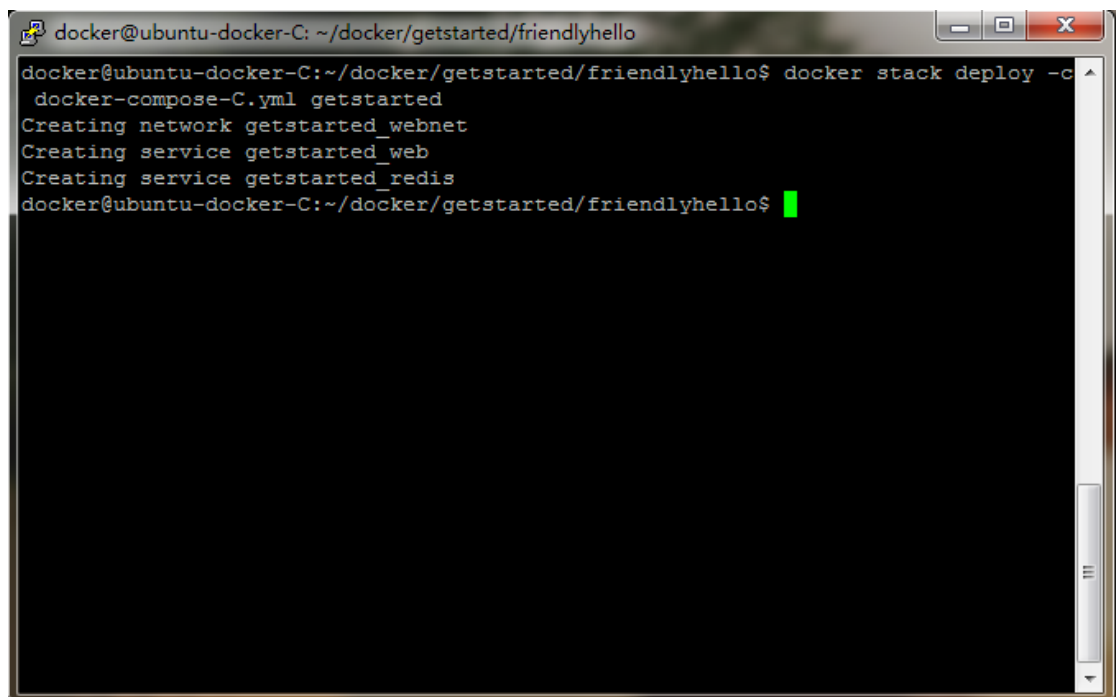


图 23 部署 web、redis 服务栈

4) 访问 Portainer 查看服务栈部署状态

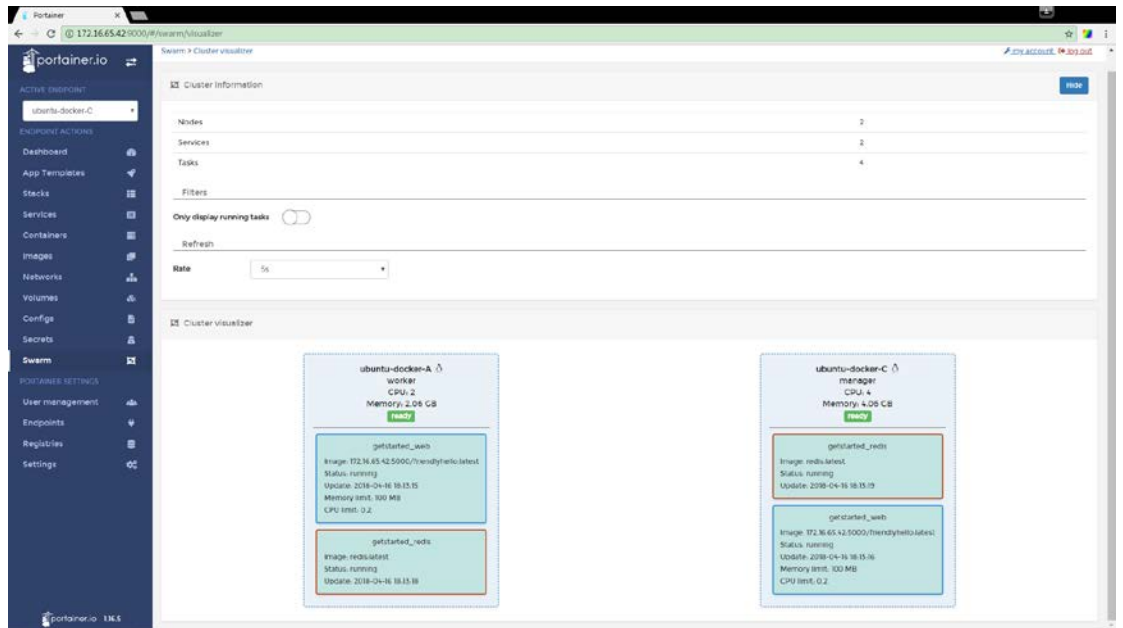


图 24 web、redis 服务栈部署状态

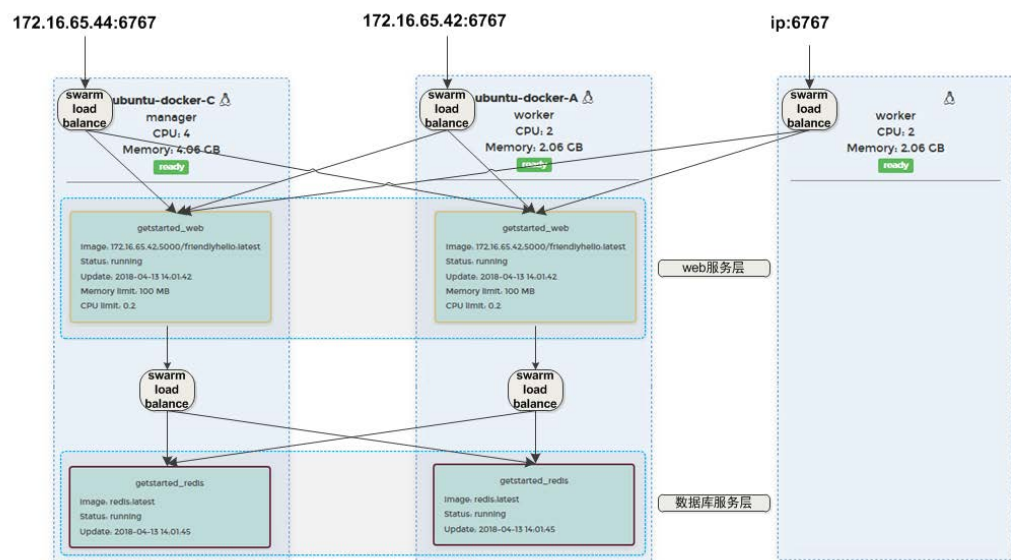


图 25 Swarm 进入路由网络

图 24 显示 web 服务、redis 服务各启动了两个容器，按照默认均衡机制分别部署在 swarm 集群的两个节点上。不光是容器部署，对于服务的访问 Swarm 也提供了负载均衡机制。这种机制 Docker 术语称为进入路由网络（ingress routing mesh），如图 25。路由网络机制保证了可以从 swarm 集群的任意主机访问 web 服务。集群中的 web 服务、redis 服务逻辑分层形成所谓的 service 栈（service stack）。

5) 服务栈验证

访问 <http://172.16.65.44:6767/>、<http://172.16.65.42:6767/> 多次得到图 26 所示结果。在这个结果里，两个 redis 数据库保持了数据一致性。

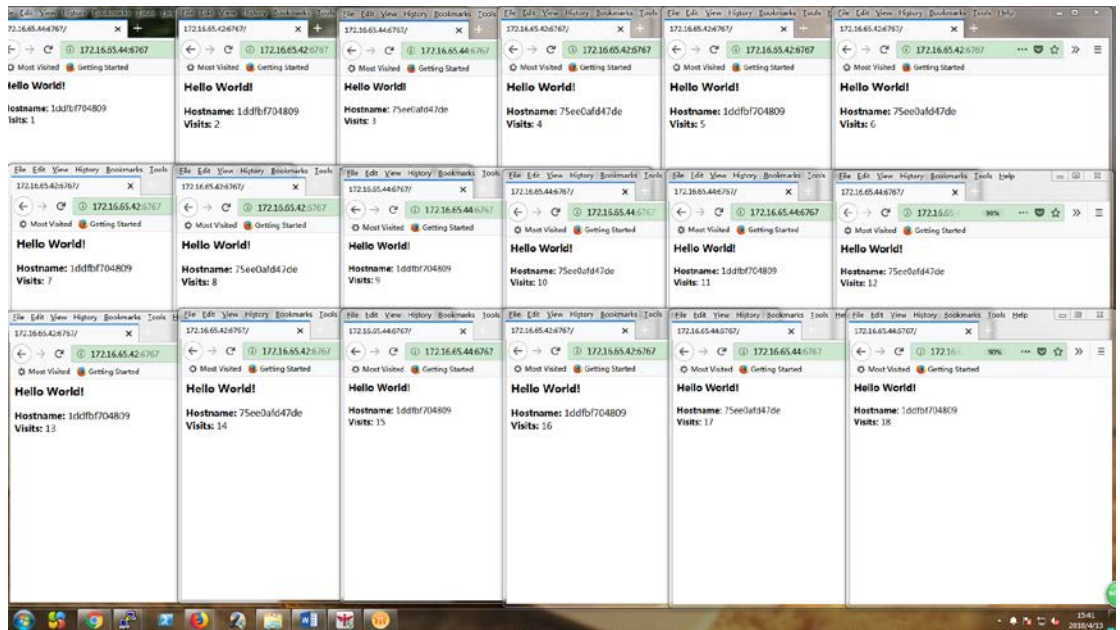


图 26 web、redis 服务栈负载均衡

Docker WebStreamr

后续工作。

WebStreamr-Compose.yml

后续工作。

参考文献

1. <https://docs.docker.com/engine/docker-overview/>
2. <https://docs.docker.com/registry/>
3. <https://docs.docker.com/swarm/overview/>
4. <https://docs.docker.com/engine/reference/commandline/cli/>
5. <https://docs.docker.com/compose/overview/>
6. <https://docs.docker.com/machine/overview/>
7. <https://github.com/portainer/portainer>
8. <https://docs.docker.com/engine/reference/builder/>
9. <https://docs.docker.com/compose/compose-file/>
10. <http://flask.pocoo.org/>
11. <https://www.putty.org/>
12. <https://docs.docker.com/install/linux/linux-postinstall/#configure-where-the-docker-daemon-listens-for-connections>
13. <https://docs.docker.com/engine/swarm/ingress/>