docker-compose

介绍

`Docker Compose`是`Docker`官方编排（Orchestration）项目之一，负责快速的部署分布式应用。其代码目前在https://github.com/docker/compose上开源。Compose 定位是 「定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）」，其前身是开源项目`Fig`。

前面我们已经学习过使用一个`Dockerfile`模板文件，可以很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器或者缓存服务容器，甚至还包括负载均衡容器等。Compose 恰好满足了这样的需求。它允许用户通过一个单独的`docker-compose.yml`模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

Compose 中有两个重要的概念：

- 服务 (service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
- 项目 (project)：由一组关联的应用容器组成的一个完整业务单元，在 docker-compose.yml 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。
Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。所以只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

安装与卸载

`Compose`支持 Linux、macOS、Windows 10 三大平台。Compose 可以通过 Python 的包管理工具`pip`进行安装，也可以直接下载编译好的二进制文件使用，甚至能够直接在 Docker 容器中运行。前两种方式是传统方式，适合本地环境下安装使用；最后一种方式则不破坏系统环境，更适合云计算场景。
Docker for Mac 、Docker for Windows 自带 docker-compose 二进制文件，安装 Docker 之后可以直接使用。

```
# pip安装
pip3 install docker-compose
# 卸载
pip3 uninstall docker-compose
```

使用

先来一个简单的，将nginx启动的命令写到compose文件中

```
version: "3"  # 表示compose版本，有1，2，3 最新是3 必须用字符串表示
services:
  #定义服务-容器启动相关参数
  frontend:   #服务的名称
    image: nginx:latest       # 指定容器启动所需的镜像
    container_name: mynginx   # 指定容器名称
    ports:                    # 端口映射   可以映射多个端口
    - 80:80                   # 端口映射不要加空格
    - 8080:8080
    volumes:                  # 目录挂载   可以挂载多个
    - /root/conf:/etc/nginx
    - /root/html:/usr/share/nginx/html
```

启动compose文件，docker会自动创建一个bridge网络，命名规则是当前目录名_default

```
[root@localhost ~]# docker-compose up -d
Creating network "root_default" with the default driver
Creating mynginx ... done
```

查看网络信息

```
[root@localhost ~]# docker network ls
NETWORK ID     NAME                DRIVER    SCOPE
d31a51b8ef4e   autotpnet           bridge    local
```

实战

采用我们的测试平台项目来演示docker-compose的使用

项目准备dockerfile文件,内容和之前的一样。

```
FROM python:3.8
COPY . /opt
RUN  cd /opt && \
     sh auto_deploy.sh
CMD  cd /opt && \
     uwsgi uwsgi.ini && \
     tail -f uwsgi_server.log
```

更新auto_deploy.sh文件

```
mkdir -p /root/.config/pip && touch /root/.config/pip/pip.conf && \
echo "[global]\ntimeout = 60\nindex-url =
https://pypi.doubanio.com/simple\ntrusted-host = pypi.doubanio.com" >
/root/.config/pip/pip.conf && \
cat /root/.config/pip/pip.conf && \
pip install --upgrade pip  && \
pip install -r requirements.txt && \
pip install uwsgi
```

编写 `docker-compose.yml` 文件，这个是 Compose 使用的主模板文件。docker-compose工具会根据模板的内容来对容器进行编排，相当于我们平时用命令运行容器，这里把命令以yml格式放在配置文件中了。

```
version: "3"  # 表示compose版本，有1，2，3 最新是3 必须用字符串表示
services:
  #定义服务-容器启动相关参数
  frontend:   #服务的名称 可以自定义
    image: nginx:latest       # 指定容器启动所需的镜像
    container_name: mynginx   # 指定容器名称
    ports:                    # 端口映射  可以映射多个端口
    - 80:80                   # 端口映射不要加空格
    volumes:                  # 目录挂载  可以挂载多个
    - /root/conf:/etc/nginx
    - /root/html:/usr/share/nginx/html
    depends_on:               # 表示依赖的服务，在依赖的服务成功启动后才会启动
      - backend

  backend:
    build:                    # 构建镜像
      context: .              # 镜像上下文，相当于docker build [选项] <上下文路径/URL/->
      dockerfile: Dockerfile  # Dockerfile相对compose文件的相对路径
    container_name: autotpenv
    ports:
    - 8081:8081
```

介绍几个术语。

- 服务（`service`）：一个应用容器，实际上可以运行多个相同镜像的实例。
  - 以上backend和 frontend都是服务名
- 项目（`project`）：由一组关联的应用容器组成的一个完整业务单元。
  - 1个compose文件代表的就是1个项目

可见，一个项目可以由多个服务（容器）关联而成，`Compose` 面向项目进行管理。

修改`auto_deploy.sh`

```
mkdir -p /root/.config/pip && touch /root/.config/pip/pip.conf && \
echo "[global]\ntimeout = 60\nindex-url =
https://pypi.doubanio.com/simple\ntrusted-host = pypi.doubanio.com" >
/root/.config/pip/pip.conf && \
cat /root/.config/pip/pip.conf && \
pip install --upgrade pip  && \
pip install -r requirements.txt && \
pip install uwsgi
```

执行docker-compose命令启动。

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

一键构建镜像，并且拉起容器。

```
[root@localhost ~]# docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED        STATUS
      PORTS                                       NAMES
f0b896ddf43a   autotpsite_backend   "/bin/sh -c 'cd /opt…"   2 hours ago    Up 2
hours   0.0.0.0:8081->8081/tcp, :::8081->8081/tcp   autotpenv
55002022a044   nginx:latest         "/docker-entrypoint.…"   2 hours ago    Up 2
hours   0.0.0.0:80->80/tcp, :::80->80/tcp           mynginx
```

完善

现在我们将数据库也加入到容器组中。

备份数据库文件

```
mkdir /data/mysql # 创建文件夹
docker cp db_mysql:/etc/mysql /data/mysql/conf # 容器拷贝配置文件
docker cp db_mysql:/var/lib/mysql /data/mysql/data  #容器拷贝数据文件

scp -r root@192.168.21.142:/data/mysql /data/mysql  # 从192.168.21.142拷贝数据库文
件
```

compose文件增加数据库配置

```
version: "3"  # 表示compose版本，有1，2，3 最新是3 必须用字符串表示
services:
  #定义服务-容器启动相关参数
  frontend:   #服务的名称 可以自定义
    image: nginx:latest       # 指定容器启动所需的镜像
    container_name: mynginx   # 指定容器名称
    ports:                    # 端口映射  可以映射多个端口
    - 80:80                   # 端口映射不要加空格
    volumes:                  # 目录挂载  可以挂载多个
    - /root/conf:/etc/nginx
    - /root/html:/usr/share/nginx/html
    depends_on:               # 表示依赖的服务，在依赖的服务成功启动后才会启动
      - backend

  backend:
    build:                    # 构建镜像
      context: .              # 镜像上下文，相当于docker build [选项] <上下文路
径/URL/->
      dockerfile: Dockerfile  # Dockerfile相对compose文件的相对路径
    container_name: autotpenv
    ports:
    - 8081:8081

  db:
    image: mysql:5.7
    container_name: db_mysql
    ports:
    - 4498:3306
    environment:
      - MYSQL_ROOT_PASSWORD=devops
    volumes:
      - /data/mysql/conf:/etc/mysql
      - /data/mysql/data:/var/lib/mysql
```

更新settings文件的DATABASE配置部分

```
if not DEBUG:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': 'course_autotp',  # 数据库名称
            'USER': 'root',  # 用户名
            'PASSWORD': 'devops',  # 密码
            'HOST': 'db_mysql',
            'PORT': '3306',
        }
    }
else:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': 'course_autotp',  # 数据库名称
            'USER': 'root',  # 用户名
            'PASSWORD': 'devops',  # 密码
            'HOST': '192.168.21.140',
            'PORT': '3306',
            'TEST': {
                'CHARSET': 'utf8',  # 测试数据库的编码配置
                'COLLATION': 'utf8_general_ci',
            }
        }
    }
```

此时可以独立部署接口平台系统了，包括了数据库。

容器编排-Rancher

Rancher是一个开源的企业级容器管理平台。通过Rancher，企业再也不必自己使用一系列的开源软件去从头搭建容器服务平台。Rancher提供了在生产环境中使用的管理Docker和Kubernetes的全栈化容器部署与管理平台。

rancher使用最广泛的功能是容器编排与调度

搭建rancher平台

rancher由server-agent构成，都可以通过容器的方式来启动

启动server

```
docker run -d --restart=unless-stopped -p 8080:8080 rancher/server:stable
```

启动Rancher Server只需要几分钟时间。当日志中显示 `.... Startup Succeeded, Listening on port...` 的时候，Rancher UI就能正常访问了。配置一旦完成，这行日志就会立刻出现。需要注意的是，这一输出之后也许还会有其他日志，因此，在初始化过程中这不一定是最后一行日志。

Rancher UI的默认端口是 `8080`。所以为了访问UI，需打开 `http://<SERVER_IP>:8080`。需要注意的事，如果你的浏览器和Rancher Server是运行在同一主机上的，你需要通过主机的**真实IP地址**访问，比如 `http://192.168.1.100:8080` ，而不是 `http://localhost:8080` 或 `http://127.0.0.1:8080`，以防在添加主机的时候使用了不可达的IP而出现问题。

添加主机

在这里，为了简化操作，我们将添加运行着Rancher Server的主机为Rancher内的主机。在实际的生产环境中，请使用专用的主机来运行Rancher Server。

想要添加主机，首先你需要进入UI界面，点击**基础架构**，然后你将看到**主机**界面。点击**添加主机**，Rancher将提示你选择主机注册URL。这个URL是Rancher Server运行所在的URL，且它必须可以被所有你要添加的主机访问到——当Rancher Server会通过NAT防火墙或负载均衡器被暴露至互联网时，这一设定就非常重要了。如果你的主机有一个私有或本地的IP地址，比如 `192.168.*.*`，Rancher将提示一个警告信息，提醒你务必确保这个URL可以被主机访问到。

因为我们现在只会添加Rancher Server主机自身，你可以暂时忽略这些警告。点击**保存**。默认选择**自定义**选项，你将得到运行Rancher agent容器的Docker命令。这里还有其他的公有云的选项，使用这些选项，Rancher可以使用Docker Machine来启动主机。

Rancher UI会给你提供一些指示，比如你的主机上应该开放的端口，还有其他一些可供选择的信息。鉴于我们现在添加的是Rancher Server运行的主机，我们需要添加这个主机所使用的公网IP。页面上的一个选项提供输入此IP的功能，此选项会自动更新Docker命令中的环境变量参数以保证正确。

然后请在运行Rancher Server的主机上运行这个命令。

当你在Rancher UI上点击**关闭**按钮时，你会被返回到**基础架构->主机**界面。一两分钟之后，主机会自动出现在这里。

接下来可以在rancher上查看对应主机的容器运行情况。

> 若Agent与Server处于同一台主机，则需要主机手动开启防火墙，否则代理无法连接到主机

结合docker-compose容器编排

选择应用(Stack)>用户>添加新的应用

导入compose文件，由于rancher不支持version3，顾改为version:2

```
version: "2"  # 表示compose版本，有1，2，3 最新是3 必须用字符串表示
services:
  #定义服务-容器启动相关参数
  frontend:   #服务的名称 可以自定义
    image: nginx:latest       # 指定容器启动所需的镜像
    container_name: mynginx   # 指定容器名称
    ports:                    # 端口映射   可以映射多个端口
    - 80:80                   # 端口映射不要加空格
    volumes:                  # 目录挂载   可以挂载多个
    - /root/conf:/etc/nginx
    - /root/html:/usr/share/nginx/html
    depends_on:               # 表示依赖的服务，在依赖的服务成功启动后才会启动
      - backend

  backend:
    image: autotpsite007_backend:latest
    container_name: autotpenv
    ports:
    - 8081:8081
```

```
  db:
    image: mysql:5.7
    container_name: db_mysql
    ports:
    - 4498:3306
    environment:
      - MYSQL_ROOT_PASSWORD=devops
    volumes:
      - /data/mysql/conf:/etc/mysql
      - /data/mysql/data:/var/lib/mysql
```

此时，需要修改配置文件中，通过容器名来指定容器的方案，改成通过服务名指定容器

```
# nginx.conf

location /api {
       # proxy_pass    http://autotpenv:8081;
        proxy_pass    http://backend:8081;
}
```

```
# settings.py
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': 'course_autotp',  # 数据库名称
            'USER': 'root',  # 用户名
            'PASSWORD': 'devops',  # 密码
            'HOST': 'db',  #改成服务名
            'PORT': '3306',
        }
    }
```


附录-docker-compose模板命令

模板文件是使用 `Compose` 的核心，涉及到的指令关键字也比较多。但大家不用担心，这里面大部分指令跟 `docker run` 相关参数的含义都是类似的。

默认的模板文件名称为 `docker-compose.yml`，格式为 YAML 格式。

```
version: "3"
services:  webapp:    image: examples/web    ports:      - "80:80"    volumes:
   - "/data"
```

注意每个服务都必须通过 `image` 指令指定镜像或 `build` 指令（需要 Dockerfile）等来自动构建生成镜像。

如果使用 `build` 指令，在 `Dockerfile` 中设置的选项(例如：`CMD`，`EXPOSE`，`VOLUME`，`ENV` 等) 将会自动被获取，无需在 `docker-compose.yml` 中重复设置。

下面分别介绍各个指令的用法。

build

指定 `Dockerfile` 所在文件夹的路径（可以是绝对路径，或者相对 docker-compose.yml 文件的路径）。 `Compose` 将会利用它自动构建这个镜像，然后使用这个镜像。

```
version: '3'services:
  webapp:    build: ./dir
```

你也可以使用 `context` 指令指定 `Dockerfile` 所在文件夹的路径。

使用 `dockerfile` 指令指定 `Dockerfile` 文件名。

使用 `arg` 指令指定构建镜像时的变量。

```
version: '3'services:
  webapp:    build:      context: ./dir      dockerfile: Dockerfile-alternate
args:        buildno: 1
```

使用 `cache_from` 指定构建镜像的缓存

```
build:
  context: .
  cache_from:
    - alpine:latest
    - corp/web_app:3.14
```

cap_add, cap_drop

指定容器的内核能力（capacity）分配。

例如，让容器拥有所有能力可以指定为：

```
cap_add:
  - ALL
```

去掉 NET_ADMIN 能力可以指定为：

```
cap_drop:
  - NET_ADMIN
```

command

覆盖容器启动后默认执行的命令。

```
command: echo "hello world"
```

configs

仅用于 `Swarm mode`，详细内容请查看 `Swarm mode` 一节。

cgroup_parent

指定父 `cgroup` 组，意味着将继承该组的资源限制。

例如，创建了一个 cgroup 组名称为 `cgroups_1`。

```
cgroup_parent: cgroups_1
```

container_name

指定容器名称。默认将会使用 `项目名称_服务名称_序号` 这样的格式。

```
container_name: docker-web-container
```

> 注意: 指定容器名称后，该服务将无法进行扩展（scale），因为 Docker 不允许多个容器具有相同的名称。

deploy

仅用于 `Swarm mode`，详细内容请查看 `Swarm mode` 一节

devices

指定设备映射关系。

```
devices:
  - "/dev/ttyUSB1:/dev/ttyUSB0"
```

depends_on

解决容器的依赖、启动先后的问题。以下例子中会先启动 `redis` `db` 再启动 `web`

```
version: '3'
services:  web:    build: .    depends_on:      - db      - redis
  redis:    image: redis
  db:    image: postgres
```

> 注意：`web` 服务不会等待 `redis` `db` 「完全启动」之后才启动。

dns

自定义 `DNS` 服务器。可以是一个值，也可以是一个列表。

```
dns: 8.8.8.8
dns:  - 8.8.8.8  - 114.114.114.114
```

dns_search

配置 `DNS` 搜索域。可以是一个值，也可以是一个列表。

```
dns_search: example.com
dns_search:  - domain1.example.com  - domain2.example.com
```

tmpfs

挂载一个 tmpfs 文件系统到容器。

```
tmpfs: /run
tmpfs:
  - /run
  - /tmp
```

env_file

从文件中获取环境变量，可以为单独的文件路径或列表。

如果通过 `docker-compose -f FILE` 方式来指定 Compose 模板文件，则 `env_file` 中变量的路径会基于模板文件路径。

如果有变量名称与 `environment` 指令冲突，则按照惯例，以后者为准。

```
env_file: .env
env_file:  - ./common.env  - ./apps/web.env  - /opt/secrets.env
```

环境变量文件中每一行必须符合格式，支持 `#` 开头的注释行。

```
# common.env: Set development environment
PROG_ENV=development
```

environment

设置环境变量。你可以使用数组或字典两种格式。

只给定名称的变量会自动获取运行 Compose 主机上对应变量的值，可以用来防止泄露不必要的数据。

```
environment:  RACK_ENV: development  SESSION_SECRET:
environment:  - RACK_ENV=development  - SESSION_SECRET
```

如果变量名称或者值中用到 `true|false，yes|no` 等表达 布尔 含义的词汇，最好放到引号里，避免 YAML 自动解析某些内容为对应的布尔语义。这些特定词汇，包括

```
y|Y|yes|Yes|YES|n|N|no|No|NO|true|True|TRUE|false|False|FALSE|on|On|ON|off|Off|O
FF
```

expose

暴露端口，但不映射到宿主机，只被连接的服务访问。

仅可以指定内部端口为参数

```
expose:
 - "3000"
 - "8000"
```

external_links

> 注意：不建议使用该指令。

链接到 `docker-compose.yml` 外部的容器，甚至并非 `Compose` 管理的外部容器。

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

extra_hosts

类似 Docker 中的 `--add-host` 参数，指定额外的 host 名称映射信息。

```
extra_hosts:
 - "googledns:8.8.8.8"
 - "dockerhub:52.1.157.61"
```

会在启动后的服务容器中 `/etc/hosts` 文件中添加如下两条条目。

```
8.8.8.8 googledns
52.1.157.61 dockerhub
```

healthcheck

通过命令检查容器是否健康运行。

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 1m30s
  timeout: 10s
  retries: 3
```

image

指定为镜像名称或镜像 ID。如果镜像在本地不存在，`Compose` 将会尝试拉取这个镜像。

```
image: ubuntu
image: orchardup/postgresql
image: a4bc65fd
```

labels

为容器添加 Docker 元数据（metadata）信息。例如可以为容器添加辅助说明信息。

```
labels:
  com.startupteam.description: "webapp for a startup team"
  com.startupteam.department: "devops department"
  com.startupteam.release: "rc3 for v1.0"
```

links

> 注意：不推荐使用该指令。

logging

配置日志选项。

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

目前支持三种日志驱动类型。

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

`options` 配置日志驱动的相关参数。

```
options:
  max-size: "200k"
  max-file: "10"
```

network_mode

设置网络模式。使用和 `docker run` 的 `--network` 参数一样的值。

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

networks

配置容器连接的网络。

```
version: "3"services:
  some-service:    networks:     - some-network     - other-network
networks:  some-network:  other-network:
```

pid

跟主机系统共享进程命名空间。打开该选项的容器之间，以及容器和宿主机系统之间可以通过进程 ID 来相互访问和操作。

```
pid: "host"
```

ports

暴露端口信息。

使用宿主端口：容器端口 (HOST:CONTAINER) 格式，或者仅仅指定容器的端口（宿主将会随机选择端口）都可以。

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

注意：当使用 HOST:CONTAINER 格式来映射端口时，如果你使用的容器端口小于 60 并且没放到引号里，可能会得到错误结果，因为 YAML 会自动解析 xx:yy 这种数字格式为 60 进制。为避免出现这种问题，建议数字串都采用引号包括起来的字符串格式。

secrets

存储敏感数据，例如 mysql 服务密码。

```
version: "3.1"services:
mysql:  image: mysql  environment:    MYSQL_ROOT_PASSWORD_FILE:
/run/secrets/db_root_password  secrets:    - db_root_password    -
my_other_secret
secrets:  my_secret:    file: ./my_secret.txt  my_other_secret:    external:
true
```

security_opt

指定容器模板标签（label）机制的默认属性（用户、角色、类型、级别等）。例如配置标签的用户名和角色名。

```
security_opt:
    - label:user:USER
    - label:role:ROLE
```

stop_signal

设置另一个信号来停止容器。在默认情况下使用的是 SIGTERM 停止容器。

```
stop_signal: SIGUSR1
```

sysctls

配置容器内核参数。

```
sysctls:  net.core.somaxconn: 1024  net.ipv4.tcp_syncookies: 0
sysctls:  - net.core.somaxconn=1024  - net.ipv4.tcp_syncookies=0
```

ulimits

指定容器的 ulimits 限制值。

例如，指定最大进程数为 65535，指定文件句柄数为 20000（软限制，应用可以随时修改，不能超过硬限制） 和 40000（系统硬限制，只能 root 用户提高）。

```
  ulimits:
    nproc: 65535
    nofile:
      soft: 20000
      hard: 40000
```

volumes

数据卷所挂载路径设置。可以设置为宿主机路径(`HOST:CONTAINER`)或者数据卷名称(`VOLUME:CONTAINER`)，并且可以设置访问模式 （`HOST:CONTAINER:ro`）。

该指令中路径支持相对路径。

```
volumes:
 - /var/lib/mysql
 - cache/:/tmp/cache
 - ~/configs:/etc/configs/:ro
```

如果路径为数据卷名称，必须在文件中配置数据卷。

```
version: "3"
services:
my_src:
image: mysql:8.0
volumes:
- mysql_data:/var/lib/mysql
volumes:
mysql_data:
```

其它指令

此外，还有包括 `domainname, entrypoint, hostname, ipc, mac_address, privileged, read_only, shm_size, restart, stdin_open, tty, user, working_dir` 等指令，基本跟 `docker run` 中对应参数的功能一致。

指定服务容器启动后执行的入口文件。

```
entrypoint: /code/entrypoint.sh
```

指定容器中运行应用的用户名。

```
user: nginx
```

指定容器中工作目录。

```
working_dir: /code
```

指定容器中搜索域名、主机名、mac 地址等。

```
domainname: your_website.com
hostname: test
mac_address: 08-00-27-00-0C-0A
```

允许容器中运行一些特权命令。

```
privileged: true
```

指定容器退出后的重启策略为始终重启。该命令对保持服务始终运行十分有效，在生产环境中推荐配置为 `always` 或者 `unless-stopped`。

```
restart: always
```

以只读模式挂载容器的 root 文件系统，意味着不能对容器内容进行修改。

```
read_only: true
```

打开标准输入，可以接受外部输入。

```
stdin_open: true
```

模拟一个伪终端。

```
tty: true
```

读取变量

Compose 模板文件支持动态读取主机的系统环境变量和当前目录下的 `.env` 文件中的变量。

例如，下面的 Compose 文件将从运行它的环境中读取变量 `${MONGO_VERSION}` 的值，并写入执行的指令中。

```
version: "3"services:
db:  image: "mongo:${MONGO_VERSION}"
```

如果执行 `MONGO_VERSION=3.2 docker-compose up` 则会启动一个 `mongo:3.2` 镜像的容器；如果执行 `MONGO_VERSION=2.8 docker-compose up` 则会启动一个 `mongo:2.8` 镜像的容器。

若当前目录存在 `.env` 文件，执行 `docker-compose` 命令时将从该文件中读取变量。

在当前目录新建 `.env` 文件并写入以下内容。

```
# 支持 # 号注释
MONGO_VERSION=3.6
```

执行 `docker-compose up` 则会启动一个 `mongo:3.6` 镜像的容器。