1. Dockerfile

注意事项:

- 文件名首字母大写
- 存储Dockerfile的目录, 尽量是空目录
- 制作的镜像功能尽量单一
- 制作步骤要尽可能精简

```
RUN mkdir /home/go/hello
RUN mkdir /home/go/world

RUN mkdir /home/go/hello && mkdir /home/go/world
```

1.1 Dockerfile的构成

> dockerfile中的注释使用: #

- 基础镜像信息
  - 要制作的新的镜像, 基于那个镜像来制作的
  - 通过 docker images 查看
  - FROM 镜像名
- 维护者信息
  - 这个dockerfile是谁写的
  - MAINTAINTER 信息
- 镜像操作指令
  - 基于原始进行进行的操作

```
RUN
COPY
ADD
EXPOSE
```

- 容器启动指令
  - 基于第三步得到了新镜像
  - 新的镜像启动之后, 在容器中默认执行的指令

```
CMD
ENTRYPOINT
VOLUME
```

1.2 Dockerfile基础指令

- FROM

```
FROM 镜像名
FROM 镜像名:tag
# FROM必须要出现Dockerfile的第一行(除注释)，可以连续写多个FROM创建多个镜像.
# 如果指定的镜像名本地仓库没有，会从远程仓库pull到本地，远程仓库也没有 -> 报错.
```

- MAINTAINER

```
dockerfile维护者信息
MAINTAINER 维护人员信息
```

- RUN

```
# 构建镜像时候执行的shell命令，如果命令有确认操作，必须要加 -y
# 如果命令太长需要换行，行末尾需要加 \
RUN shell命令
    RUN mkdir /home/go/test -p
RUN ["mkdir", "/home/go/test", "-p"]
```

- EXPOSE

```
# 设置对外开放的端口
# 容器和外部环境是隔离的，如何向从外部环境访问到容器内部，需要容器开发端口
# 在使用的时候，让宿主机端口和容器开放端口形成一个映射关系，就可以访问了
#     docker run -itd -p 8888:80
EXPOSE 80
```

1.3 Dockerfile运行时指令

- CMD

```
# 新镜像已经被制作完毕，启动新镜像-> 得到一个容器
# 容器启动后默认执行的命令
# 一个dockerfile文件只能指定一个CMD指令
# 如果指定多个，只有最后一个有效
# 该CMD会被 docker run指定的shell命令覆盖
CMD shell命令
CMD ["shell命令", "命令参数1", "命令参数2"]
```

- ENTRYPOINT

```
# docker容器启动之后执行的命令，该命令不会被docker run指定的shell指令覆盖
# ENTRYPOINT只能指定一个，指定多个，只有最后一有效
# ENTRYPOINT 和 CMD可以同时指定
# 如果想被docker run覆盖，启动docker容器时可使用docker run --entrypoint
ENTRYPOINT shell命令
ENTRYPOINT ["shell命令", "命令参数1", "命令参数2"]
```

- CMD ENTRYPOINT 综合使用

```
docker run -itd ubuntu
# 任何docker run设置的命令参数或者CMD指令的命令，都将作为ENTRYPOINT 指令的命令参数，追加到ENTRYPOINT指令之后
ENTRYPOINT mkdir /home/go/a/b/c/d/e/f
CMD -p
mkdir /home/go/a/b/c/d/e/f -p ls
```

1.4 Dockerfile文件编辑指令

- ADD

```
# 将宿主机文件拷贝到容器目录中
# 如果宿主机文件是可识别的压缩包，会进行解压缩 -> tar
ADD 宿主机文件 容器目录/文件
# 实例
ADD ["宿主机文件", "容器目录"]
 - 宿主机文件一般放到Dockerfile对应的目录中
 - 容器目录，有可能存在，有可能不存在
    - 存在：直接拷贝到容器目录
    - 不存在：先在容器中创建一个，再拷贝
ADD ["a.txt", "/home/go/a.txt"]
 - 第二个参数如果指定的是一个文件名
   - 这个文件存在：直接覆盖
   - 不存在：直接拷贝
```

- COPY

```
# COPY 指令和ADD 指令功能和使用方式类似。只是COPY 指令不会做自动解压工作。
# 单纯复制文件场景，Docker 推荐使用COPY
COPY ["a.tar.gz", "/home/"]
```

- VOLUME

```
# 数据卷容器创建，挂载点为/backup
docker create -it --name contains -v /backup ubuntu bash
# 其他容器挂载到数据卷容器上
docker run -itd --volumes-from contains ubuntu bash

# VOLUME 指令可以在镜像中创建挂载点，这样只要通过该镜像创建的容器都有了挂载点
# 通过VOLUME 指定挂载点目录是自动生成的。
VOLUME ["/data"]

```

1.5 Dockerfile环境指令

- ENV

> 环境变量:
>
> - 系统级别
> - 用户级别
>
> 环境变量名大写

> Linux: env

```
# 设置环境变量，可以在RUN 之前使用，然后RUN 命令时调用，容器启动时这些环境变量都会被指定
ENV <key> <value>                         (一次设置一个环境变量)
ENV <key>=<value> ...                     (一次设置一个或多个环境变量)
ENV HELLO 12345
ENV HELLO=12345
ENV HELLO=12345 WORLD=12345 NIHAO=12345
ENV MYPATH=/a/b/c/d/e/f/g/h/....../z
mkdir /home/go $MYPATH
```

- WORKDIR

```
 # 切换目录，为后续的RUN、CMD、ENTRYPOINT 指令配置工作目录。相当于cd
 # 可以多次切换(相当于cd 命令),
 # 也可以使用多个WORKDIR 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。
 WORKDIR /path/to/workdir
 RUN a.sh
 WORKDIR /path
 WORKDIR /bin/abc
 WORKDIR to  # 相对路径 /bin/abc/to
 WORKDIR workdir # /bin/abc/to/workdir
 RUN pwd
 /bin/abc/to/workdir


 # 可执行程序 a.out
 # 现在在家目录下
 ./a.out
 # 工作目录进行了切换
 WORKDIR /home/go/test/work
 ./a.out
```

- USER

```
# 指定运行容器时的用户名和UID，后续的RUN 指令也会使用这里指定的用户。
# 如果不输入任何信息，表示默认使用root 用户
USER daemon

# /etc/passwd 文件的第一列就是用户名
```

- ARG

```
# ARG 指定了一个变量在docker build 的时候使用，可以使用--build-arg <varname>=<value>来指定参数的值。
ARG <name>[=<default value>]


#dockerfile写好之后
docker build -t 镜像名:镜像tag dockerfile的存储目录
```

1.6 Dockerfile触发器指令

- ONBUILD

```
# 当一个镜像A被作为其他镜像B的基础镜像时，这个触发器才会被执行，
# 新镜像B在构建的时候，会插入触发器中的指令。
ONBUILD [command]

# 原始镜像 -> 纯净版
    -> 修改 ONBUILD ["echo", "hello,linux"]

# 基于原始镜像制作新镜像 -> 镜像A
    -> 启动镜像A -> 不会输出hello, linux

# 基于镜像A制作了镜像B
    -> 启动镜像B -> 会输出 hello, linux
```

1.7 Dockerfile构建缓存

```
构建新镜像的时候不使用缓存机制
docker build -t 新镜像名:版本号 --no-cache
```

1.8 通过Dockerfile构建beego镜像

2. docker-compose

> Compose 是 Docker 容器进行编排的工具，定义和运行多容器的应用，可以一条命令启动多个容器，使用Docker Compose不再需要使用shell脚本来启动容器。
>
> Compose 通过一个配置文件来管理多个Docker容器，在配置文件中，所有的容器通过services来定义，然后使用docker-compose脚本来启动，停止和重启应用，和应用中的服务以及所有依赖服务的容器，非常适合组合使用多个容器进行开发的场景。
>
> - 知道yaml文件格式
> - docker-compose工具工作的时候需要使用一个配置文件
>   - 默认的名字: docker-compose.yaml/yml
> - docker-compose中常用关键字
> - docker-compose操作命令
>   - 启动, 关闭, 查看

2.1 docker-compose的安装

```
#安装依赖工具
sudo apt-get install python-pip -y
#安装编排工具
sudo pip install docker-compose
#查看编排工具版本
sudo docker-compose version
#查看命令帮助
docker-compose --help
```

2.2 yaml文件格式

> - YAML有以下基本规则： 1、大小写敏感 2、使用缩进表示层级关系 3、禁止使用tab缩进，**只能使用空格键** 4、缩进长度没有限制(只能使用空格缩进)，只要元素对齐就表示这些元素属于一个层级。 5、使用#表示注释 6、字符串可以不用引号标注
>   - "字符串"
>   - '字符串'
>   - 字符串
>   - 123 -> 整数
>   - 123a

yaml中的三种数据结构

- map - 散列表

```
# 使用冒号（: ）表示键值对，同一缩进的所有键值对属于一个map，示例:
age : 12
name : huang

# 使用json表示
{"age":12, "name":"huang"}
```

- list - 数组

```
# 使用连字符（-）表示:
# YAML表示
- a
- b
- 12

# 使用json表示
["a", "b", 12]
```

- scalar - 纯量

```
字符串
布尔值
    - true
    - false
整数
浮点数
    - 12.1
NULL
    - 使用 ~ 来表示
```

- 例子

```yaml
# 1
Websites:
 YAML: yaml.org
 Ruby: ruby-lang.org
 Python: python.org
 Perl: use.perl.org

# 使用json表示
{"Websites":{"YAML": "yaml.org", "Ruby": "ruby-lang.org" }}

# 2
languages:
 - Ruby
 - Perl
 - Python
 - c
# 使用json表示
{"languages":["Ruby", "Perl", "Python", "c"]}

# 3
-
  - Ruby
  - Perl
  - Python
-
  - c
  - c++
  - java
# 使用json表示
[["Ruby", "Perl", "Python"], ["c", "c++", "java"]]

# 4
-
  id: 1
  name: huang
-
  id: 2
  name: liao
# 使用json表示
[{"id":1, "name":"huang"}, {"id":2, "name":"liao"}]
```

2.3 docker-compose配置文件

```yaml
version: '2'    # docker-compose的版本

services:       # 服务
  web:          # 服务名，自己起的，每个服务器名对应一个启动的容器
    image: nginx:latest # 容器是基于那个镜像启动 的
    container_name: myweb
    ports:      # 向外开发的端口
      - 8080
    networks:   # 容器启动之后所在的网络
      - front-tier
    environment:  # ENV
      RACK_ENV: development
      SHOW: 'true'
      SESSION_SECRET: docker-compose
    command: tree -L 3
    extends:
      file: common.yml
      service: webapp

  lb:
    image: dockercloud/haproxy
    ports:
      - 80:80
    networks:
      - front-tier
      - back-tier
    volumes:    # 数据卷挂载 docker run -v xxxx:xxxx
      - /var/run/docker.sock:/var/run/docker.sock
    depend_on:
      - web
      - redis
      - lb

  redis:
    image: redis
    networks:
      - back-tier

networks:
  front-tier:
    driver: bridge
  back-tier:
    driver: bridge

```

> 一份标准配置文件应该包含
>
> - version
> - services
> - networks

> 三大部分，其中最关键的就是 services 和 networks 两个部分，下面先来看 services 的书写规则。

- **Image**

```
services:
  web:
    image: 镜像名/镜像ID
```

> 在 services 标签下的第二级标签是 web，这个名字是用户自己自定义，它就是服务名称。 image 则是指定服务的镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉取这个镜像。

- **command**

> 使用 command 可以覆盖容器启动后默认执行的命令。

```
command: tree -L 3
# 也可以写成类似 Dockerfile 中的格式:
command: [tree, -L, 3]
```

- **container_name**

> 容器启动之后的名字
>
> 如何查看:
>
> docker ps

- **depends_on**

> 一般项目容器启动的顺序是有要求的，如果直接从上到下启动容器，必然会因为容器依赖问题而启动失败。例如在没启动数据库容器的时候启动了应用容器，这时候应用容器会因为找不到数据库而退出，为了避免这种情况我们需要加入一个标签，就是 depends_on，这个标签解决了容器的依赖、启动先后的问题。

```
version: '2'
services:
  web:
    image: ubuntu
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: mysql
```

- **environment**

> environment 和 Dockerfile 中的 ENV 指令一样会把变量一直保存在镜像、容器中。

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET: docker-compose
```

- **ports**

> docker run -p 宿主机端口:容器端口
>
> 映射端口的标签。 使用HOST:CONTAINER格式或者只是指定容器的端口，宿主机会随机映射端口。

```
ports:
 - "3000"  # 宿主机的端口是随机分配的，3000是容器开发的对外端口  // -P
 - "8000:8000"   -> 一般这样写就可以
 - "127.0.0.1:8001:8001"
 - 55:55   -> 不推荐这样写
```

- **volumes**

> 挂载一个目录或者一个已存在的数据卷容器，可以直接使用 [HOST:CONTAINER] 这样的格式，或者使用 [HOST:CONTAINER:ro] 这样的格式，后者对于容器来说，数据卷是只读的，这样可以有效保护宿主机的文件系统。 Compose的数据卷指定路径可以是相对路径，使用 . 或者 .. 来指定相对目录。

```
 docker run -v /home/go:/xxx
 # 宿主机或容器的映射路径如果不存在，会自动创建出来
 volumes:
   # 这是宿主机目录，容器的映射目录会自动创建
   - /var/lib/mysql
   # 按照绝对路径映射
   - /opt/data:/var/lib/mysql
   # 相对路径的映射
   # ./ 目录是docker-compose配置文件所在的目录
   # 如果是相对路径，./是必须要写的，../
   - ./cache:/tmp/cache
   # 指定容器中对文件的操作权限，默认rw
   - /home/go/configs:/etc/configs/:ro
   # 文件映射
   - ./temp/a.txt:/temp/b.sh
```

- **volumes_from**

> 从其它容器或者服务挂载数据卷，可选的参数是 :ro或者 :rw，前者表示容器只读，后者表示容器对数据卷是可读可写的。默认情况下是可读可写的。

```
volumes_from:
  - service_name  # 服务名
  - service_name:ro
  - container:container_name  # 挂载容器
  - container:container_name:rw
```

- **extends**

> 这个标签可以扩展另一个服务，扩展内容可以是来自在当前文件，也可以是来自其他文件，相同服务的情况下，后来者会有选择地覆盖原有配置。

```
# 在一个yaml文件中引用另外一个yaml中的设置
extends:
  file: common.yml
  service: webapp
```

```
# docker-compose.yaml
version: '2'    # docker-compose的版本

services:       # 服务
  web:          # 服务名，自己起的，每个服务器名对应一个启动的容器
    extends:
      file: sub-compose.yaml
      service: demo1

   web1:            # 服务名，自己起的，每个服务器名对应一个启动的容器
    image: nginx:latest # 容器是基于那个镜像启动 的
    container_name: myweb
    ports:      # 向外开发的端口
      - 8080
    networks:   # 容器启动之后所在的网络
      - front-tier
      - back-tier
    environment:  # ENV
      RACK_ENV: development
      SHOW: 'true'
      SESSION_SECRET: docker-compose
    command: tree -L 3
```

```
# sub-compose.yaml
version: '2'    # docker-compose的版本

services:       # 服务
  demo1:            # 服务名，自己起的，每个服务器名对应一个启动的容器
    image: nginx:latest # 容器是基于那个镜像启动 的
    container_name: myweb
    ports:      # 向外开发的端口
      - 8080
    networks:   # 容器启动之后所在的网络
      - front-tier
      - back-tier
    environment:  # ENV
      RACK_ENV: development
      SHOW: 'true'
      SESSION_SECRET: docker-compose
    command: tree -L 3
    extends:
      file: sub-compose.yaml
      service: webapp

   demo2:           # 服务名，自己起的，每个服务器名对应一个启动的容器
```

```
23      image: nginx:latest # 容器是基于那个镜像启动 的
24      container_name: myweb
25      ports:      # 向外开发的端口
26        - 8080
27      networks:   # 容器启动之后所在的网络
28        - front-tier
29        - back-tier
30      environment:  # ENV
31        RACK_ENV: development
32        SHOW: 'true'
33        SESSION_SECRET: docker-compose
34      command: tree -L 3
```

- **networks**

> 加入指定网络，格式如下：

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

> 关于这个标签还有一个特别的子标签aliases，这是一个用来设置服务别名的标签，例如：

```
services:
  some-service:
    networks:
      some-network:
        aliases:
         - alias1
         - alias3
      other-network:
        aliases:
         - alias2
```

2.4 docker-compose 命令

- **compose服务启动、关闭、查看**

```
前提条件：docker-compose.yaml
# 启动docker容器
# -d 以守护进程的方式启动，不加会占用一个终端，用来输入启动过程中的日志信息
docker-compose up -d    # 前提配置文件叫做：docker-compose.yaml
# 如果配置文件不叫 docker-compose.yaml，叫temp.yaml,主要指定配置文件
docker-compose -f temp.yaml up -d
# 关闭，启动的容器被关闭，并且删除
# 前提配置文件叫做：docker-compose.yaml
docker-compose down
# 如果配置文件不叫 docker-compose.yaml，叫temp.yaml,主要指定配置文件
docker-compose -f temp.yaml down
# 查看 通过docker-compose启动的容器
docker-compose ps
```

- **容器开启、关闭、删除**

```
# 启动某一个容器
docker-compose start 服务名
# 容器的关闭，没有删除
# 关闭指定的容器
docker-compose stop 服务名
# 关闭所有的，一般不用，只是关闭不删除
docker rm $(docker ps -aq) -f
docker-compose stop
# 删除
docker-compose rm 服务名
```