Docker入门学习

Overlay Network

> **叠加网络**
>
> 参考链接：
>
> - https://blog.csdn.net/cuipengchong/article/details/78604575
> - https://blog.csdn.net/u012785382/article/details/70740052
> - https://juejin.im/entry/57cb68072e958a0068dc29d8

Docker四种网络模型

参考链接

> https://www.jianshu.com/p/7b6b0c87df40
>
> https://blog.csdn.net/huanongying123/article/details/73556634**(优)**
>
> https://blog.csdn.net/csdn066/article/details/77165269

封闭式容器

> `none`，创建容器时使用`--network none`参数

桥接模式(默认模式)

> `bridge`，创建容器时使用`--network bridge`参数，默认模式

主机模式

> `host`，创建容器时使用`--network host`参数

网络模式

> `container`，创建容器时使用`--network container:NAME_OR_ID`参数

Docker实现的底层依赖

参考链接

> https://blog.csdn.net/weixin_33910434/article/details/88271352
>
> - 系列文章
>   - (namespace_01)：https://coolshell.cn/articles/17010.html
>   - (namespace_02)：https://coolshell.cn/articles/17029.html
>   - (CGroup)：https://coolshell.cn/articles/17049.html
>   - (AUFS)：https://coolshell.cn/articles/17061.html
>   - (DeviceMapper)：https://coolshell.cn/articles/17200.html

- `namespaces`
- `cgroup`

OOME

> OOME(Out OF Memory Exception)，当内存够时linux内核会杀死进程，以释放内存。杀掉优先级最高的进程。Docker特意调整了`docker daemon`的OOM优先级，以免被内核杀死。

Docker限制资源

> 通过启动时候指定内存`-m`(ram和swap)和CPU的资源

制作镜像

- 基于容器制作镜像
- 使用Dockerfile制作镜像

Dockerfile

格式

- `# 注释`
- `指令 参数`，一般一行一个指令，指令不区分大小写，但是建议指令大写，参数小写

> 指令顺序执行，`Dockerfile`文件的第一个非注释行应该为`FROM 镜像`，表示该`Dockerfile`基于哪一个镜像。
>
> 制作镜像时，在一个特殊的工作的目录，且`Dockerfile`文件名**首字母必须大写**。依赖的文件也必须在当前目录中(或者子目录中)。

> 也可以创建一个`.dockerignore`文件，可以使用通配符，忽略打包的文件，类似于`.gitignore`不追踪的文件。(不包含`.dockerignore`本身)
>
> `${NAME:-tom}`，如果`NAMEE`不存在，则使用`tom`，否则使用`NAME`
>
> `${NAME:+tom}`，如果`NAMEE`存在，则使用`tom`，否则使用`NAME`

Docker指令

> 每个指令都会生成一个新的镜像层，所以多个指令可以合成一个则合并为一个。

- `FROM`：最重要且必须为`Dockerfile`文件开篇的第一个非注释行，用于映像文件构建过程中指定基准镜像。
  - `FROM <repository>[:<tag>]`
  - `FROM <repository>@<digest>`
    - `<reposotiry>`：指定作为base image的名称
    - `<tag>`：base image的标签，为可选项，默认省略为`latest`
- `MAINTANIER`：用于让Dockerfile制作者提供本人的详细信息，不限制该指令出现的位置，但是一般为`FROM`之后。
  - `MAINTINER <author's detail>`：例 `MAINTAINER <jianmo.pip@gmail.com>`
- `LABEL`：用于替代`MAINTANIER`，可以提供更加详细的信息
  - `LABEL <key>=<value> <key>=<value> <key>=<value> ...`
- `COPY`：用于从Docker主机复制文件到创建的新镜像文件
  - `COPY <src> … <dest>`
  - `COPY ["<src>", ... "<dest>"]`
    - `<src>`：要复制的原文件或者目录，支持使用通配符
    - `<dest>`：目标路径，即正在创建`images`的文件系统路径，建议`<dest>`使用绝对路径，否则，`COPY`指定则以`WORKDIR`为其起始路径。
  - 文件复制准则
    - `<src>`必须是build上下文中的路径，不能是其父目录中的文件；
    - 如果`<src>`是目录，则其内部文件或子目录会被递归复制，但`<src>`目录自身不会被复制；
    - 如果制定了多个`<src>`或在`<src>`中使用了通配符，则`<dest>`必须是一个目录，且必须以`/`结尾；
    - 如果`<dest>`事先不存在，它将会被自动创建，这包括其父目录路径。
- `ADD`：`ADD`指令类似于COPY指令，`ADD`指令支持使用`TAR`文件和`URL`路径
  - `ADD <src> ... <dest>`
  - `ADD {"<src>" ... "<dest>"}`
  - 文件操作准则
    - 同`COPY`指令
    - 如果`<src>`为`URL`且`<dest>`不以`/`结尾，则`<src>`指定的文件将被下载并且直接被创建为`<dest>`，如果`<dest>`以`/`结尾，则文件名URL指定的文件将被直接下载并保存为`<dest>/<filename>`；
    - 如果`<src>`是一个本地系统上的压缩格式的`tar`文件，它将被展开为一个目录，其行为类似于`tar -x`命令，然而，通过`URL`获取得到`tar`文件将不会自动展开；

        - 如果`<src>`有多个，或其间接或直接使用了通配符，则`<dest>`必须是一个以`/`结尾的目录路径；如果`<dest>`不以`/`结尾，则其被视为一个普通文件，`<src>`的内容将被直接写入到`<dest>`。
- `WORKDIR`：用于为`Dockerfile`中所有的RUN、CMD、ENTRYPOINT、COPY和ADD指定工作目录
    - `WORKDIR <dirpath>`
        - 在`Dockerfile`文件中，`WORKDIR`指令可出现多次，其路径也可以为相对路径，不过，其是相对此前一个`WORKDIR`指令指定的路径。
        - 例：`WORKDIR /var/log`和`WORKDIR $STATEPATH`
- `VOLUME`：用于在image中创建一个挂载点目录，以挂载`Docker host`上的卷或其他容器上的卷
    - `VOLUME <mountpoint>`
    - `VOLUME ["<mountpoint>"]`
    - 如果挂载点目录路径下此前在文件存在，`docker run`命令会在卷挂载完成后将此前的所有文件复制到新挂载的卷中
- `EXPOSE`：用于为容器打开指定要监听的端口以实现与外界通信
    - `EXPOSE <port>[/<protocol>] [<port>[/<protocol>] ...]`
        - `<protocol>`用于指定传输层协议，可为tcp或udp二者之一，默认为TCP协议
        - `EXPOSE`指令可以一次指定多个端口：`EXPOSE 11211/udp 11211/tcp`
    - 运行容器时，使用`-P`暴露需要暴露的端口，即`ENV`中需要的端口
- `ENV`：用于为镜像定义所需的环境变量，并可被Dockerfile文件中位于其后的其他指令`(ENV、ADD、COPY等)`所调用
    - 调用格式：`$variable_name`或`${variable_name}`
    - `ENV <key> <value>`
    - `ENV <key>=<value> ...`
    - 第一种格式，`<key>`之后的所有内容都会被视作`<value>`的组成部分，因此只能设置一个变量；
    - 第二种格式可以设置多个变量，每个变量为一个`<key>=<value>`的键值对，如果`<value>`中包含空格，可以以反斜杠`\`进行转义，也可通过对`<value>`加引号进行标识，另外反斜杠也可以用于续行；
    - 定义多个变量时，建议使用第二种方式，以便在同一层中完成所有功能。
- `RUN`：基于Dockerfile文件build镜像时候运行的命令
    - `RUN <command>`：通常是`<command>`是一个shell命令，且以`/bin/sh -c`来运行它，这意味着此进程在容器中的PID不为1，不能接收Unix信号，因此，当使用`docker stop <container>`命令停止容器时，此进程接收不到`SIGTERM`信号
    - `RUN ["<executable>", "<param1>", "<param2>"]`，语法格式中为一个JSON格式的数组，其中`<executable>`为要运行的命令，后面的为参数，这种格式指定的命令不会以`/bin/sh -c`来发起，因此不能使用`shell`操作如变量替换以及通配符`(?.*等)`替换不会进行，不过，如果要运行的命令依赖此shell特性可以使用`RUN["/bin/bash", "-c", "<executable>", "<>param1"]`
- `CMD`：RUN运行于映像文件的构建过程中，而`CMD`运行于基于`Dockerfile`构建出来的新印象文件启动一个容器时。`CMD`只能运行一个，可以有多但是只运行最后一个。
    - `CMD <command>`
    - `CMD ["<executable>", "<param1>", "<param2>"]`
    - `CMD ["<param1>", "<param2>"]`，主要为`ENTRYPOINT`指定提供默认参数
- `ENTRYPOINT`：类似与CMD指令的功能，用于为容器指定默认运行程序，从而使容器像是一个单独的可执行程序。
    - 与CMD不同的是，由`ENTRYPOINT`启动的程序不会被`docker run`命令指定的参数覆盖，而且，这些命令行参数会被当做参数传递给`ENTRYPOINT`指定的程序

    - 不过`docker run`命令的`--entrypoint`选型的参数可覆盖`ENTRYPOINT`指令指定的程序
    - `ENTRYPOINT <command>`
    - `ENTRYPOINT ["<executable>", "<param1>", "<param2>"]`
    - `docker run`命令传入的命令参数会覆盖`CMD`指令的内容并且附加到`ENTRYPOINT`命令作为其参数使用。
    - `Dockerfile`文件中也可以存在多个`ENTRYPOINT`指令，但仅有最后一个会生效。
- `USER`：用于指定运行`image`时的或运行`Dockerfile`中热河`RUN、CMD或ENTRYPOINT`指令指定的程序时的用户名或`UID`，默认用户为`root`，`root`用户属于内核。
    - `USER <UID>|<UserName>`，必须为`/etc/passwd`中的有效用户，否则会失败
- `HEALTHCHECK`：在`HEALTHCHECK`定义一个`CMD`，主要判断能否提供服务。
    - `HEALTHCHECK [OPTION] CMD command`
    - `--interval=DURATION(default:30s)`
    - `--time-out=DURATION(default:30s)`
    - `--start-period=DURATION(default:30s)`，等待进程启动后才进行检查
    - `--retries=N(default:3)`，重试次数
    - 相应值
        - `0`，成功
        - `1`，失败
        - `2`，未定义

```
HEALTHCHECK --interval=5m --timeout=3s \
CMD curl -f http://localhost/ || exit 1
```

- `SHELL`：指定程序运行时候的`shell`，linux的默认是`/bin/sh`
- `STOPSIGNAL`：指定发送给容器的信号
    - `STOPSIGNAL signal`
- `ARG`：参数传递，可以在`build`时传递参数
    - `docker build --build-arg author="jianmo" -t myweb:v0.3-9 ./`，替换`Dockerfile`中的`author`的值。
- `ONBUILD`：在`Dockerfile`中定义一个触发器，当别人使用你的镜像`build`一个新镜像时触发。(当你的镜像作为一个基础镜像时候触发执行)
    - `ONBUILD <NSTRUCTION>`
    - `ONBUILD`不能自我嵌套，且不会触发`FROM`和`MAINTAINER`指令
    - 在`ONBUILD`中使用`ADD`和`COPY`指令应当小心，因为新的构建过程中可能缺少指定的文件导致失败。