Docker

> 说明：本文是基于博文Docker — 从入门到实践进行学习总结；
> 更多文档请移步官方文档

1. 相关名词

1.1 LXC

> LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers.

`LXC` 是一组使用linux内核容器功能的用户接口；

1.2 cgroups

wiki：

> cgroups，其名称源自控制组群（control groups）的简写，是Linux内核的一个功能，用来限制、控制与分离一个进程组群的资源（如CPU、内存、磁盘输入输出等）

1.3 aufs

> aufs (short for advanced multi-layered unification filesystem) implements a `union mount` (操作系统中管理文件夹的方式) for Linux file systems.

1.4 UnionFs

中文：统一文件系统
作用：将多个文件系统（同行叫作 `分支` ）合并成一个统一的fs，具有相同路径的目录会全部展示在合并后的fs的对应目录位置，分支拥有优先级，如果不同分支中包含同名文件，分支合并（组合成UnionFs）后，高优先级分支中的文件会覆盖低优先级的；

> Unionfs is a filesystem service for Linux, FreeBSD and NetBSD which implements a union mount for other file systems. It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.
> `UnionFs` 是 `union mount` 的实现；
> aufs is a complete rewrite of the earlier UnionFS.

1.5 runc

`runc` 是一个创建和运行容器的客户端工具；

> runc is a CLI tool for spawning and running containers according to the OCI specification.

1.6 OCI

OCI是一个专门制定系统级虚拟化技术开源标准的组织；

> The *Open Container Initiative* (OCI) is a Linux Foundation project to design open standards for operating-system-

> level virtualization, most importantly Linux containers.

1.7 containerd

`containerd`是个生产级的容器运行时，主打简便性、稳定性和可移植性等特点；

> containerd is an industry-standard container runtime with an emphasis on simplicity, robustness and portability. It is available as a daemon for Linux and Windows, which can manage the complete container lifecycle of its host system: image transfer and storage, container execution and supervision, low-level storage and network attachments, etc.

2. 什么是Docker

2.1 开发语言

> Docker 使用 Google 公司推出的 Go 语言 进行开发实现

2.2 低层技术

> 最初实现是基于LXC，从 0.7 版本以后开始去除LXC，转而使用自行开发的`libcontainer`(容器管理工具，现已弃用，转为用runc)，从 1.11 开始，则进一步演进为使用runc和 containerd。

2.3 虚拟技术与容器

虚拟机技术:
虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；

容器：
应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。

2.3.1 容器的优点

1. 更高效

   > 不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高;

2. 启动更快

   > 传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。

3. 一致的运行时环境
4. 方便持续交付和部署
5. 方便迁移
6. 轻松维护和扩展

   > Docker 使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，基于基础镜像进一步扩展镜像也变得非常简单。

3. 概念

镜像、容器、仓库、数据卷

4. 安装

4.1 设置加速器

镜像加速器

5. 镜像

5.1 镜像操作

5.1.1 简单示例

以获取并运行Ubuntu镜像为例,参考:

1. 获取镜像

```
# 不添加标签（版本），默认获取latest标签的镜像
docker pull ubuntu

# 如果要添加标签，可以如下
docker pull ubuntu:18.04
```

2. 运行镜像

```
# -i 表示交互式执行命令
# -t 表示终端，因为要执行bash命令，所以需要交互式终端
# -rm 表示容器退出后随之将其删除：默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 docker rm
docker run -it --rm ubuntu:18.04  bash
```

3. 检验

```
# 查看系统信息
cat /etc/os-release
```

4. 退出

```
# 退出容器
exit
```

5.1.2 列出镜像

```
# 显示镜像列表
$ docker image ls
# 仓库名过滤，如库名Ubuntu
$ docker image ls ubuntu
# 仓库名+标签过滤，如：Ubuntu:18.04
$ docker image ls ubuntu:18.04
# 更加复杂的过滤，可以使用--filter或-f选项，如：mongo:3.2之后建立的镜像
$ docker image ls -f since=mongo:3.2
# 只列出镜像的ID
$ docker image ls -q
# 镜像列表的结构也可自定义，用到Go的模板语法
$ docker image ls --format "{{.ID}}: {{.Repository}}"
# 以表格等距显示，并且有标题行，和默认一样，不过自己定义列
$ docker image ls --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"

# 由于docker镜像是分层的，可能多个镜像共用同一个基础镜像，所以上面命令得到的镜像列表中所有镜像大小的总和并不是所有镜像实际占用的存储大小，可使用下面命令查看实际大小
$ docker system df
```

注：go-template-syntax

5.1.3 虚悬镜像

查看虚悬镜像列表：

```
$ docker image ls -f dangling=true
```

一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的，可以用下面的命令删除

```
$ docker image prune
```

5.1.4 中间层镜像

```
$ docker image ls -a
```

5.1.5 删除镜像

命令

```
$ docker image rm [OPTIONS] IMAGE [IMAGE...]
# 更多帮助信息
$ docker image rm --help
```

删除所有仓库名为 redis 的镜像:

```
$ docker image rm $(docker image ls -q redis)
```

5.2 commit镜像

> 当我们运行一个容器的时候（如果不使用卷的话），我们做的任何文件修改都会被记录于容器存储层里。
> Docker 提供了一个 docker commit 命令，可以将容器的存储层保存下来成为镜像。

5.2.1 命令

docker commit 的语法格式，请查看help信息：

```
$ docker commit --help
```

5.2.2 慎用commit

commit生成的镜像会越来越臃肿，详细参考

5.3 定制镜像

一般使用 `dockerfile` 脚本记录构建镜像的每一个步骤；

5.3.1 docker指令

5.3.1.1 FROM

必须以 `FROM` 开头，如下：以Ubuntu为基础镜像

```
FROM ubuntu
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
```

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像，以 scratch 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在：

```
FROM scratch
...
```

> 不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如 swarm、coreos/etcd。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM scratch 会让镜像体积更加小巧。使用 Go 语言 开发的应用很多会使用这种方式来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因之一。

5.3.1.2 RUN

> Union FS 是有最大层数限制的，比如 AUFS，曾经是最大不得超过 42 层，现在是不得超过 127 层

> 每一条RUN指令，就代表创建一层镜像；

所以，不要每条shell命令都添加RUN指令前缀，如下，创建了7层镜像：

```
FROM debian:stretch

RUN apt-get update
RUN apt-get install -y gcc libc6-dev make wget
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz"
RUN mkdir -p /usr/src/redis
RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
RUN make -C /usr/src/redis
RUN make -C /usr/src/redis install
```

正确的方式：

```
# 这是RUN的正确使用方式
# 并且，最后还进行了rm清理工作，将编译redis编译过程中使用得软件和包清理掉
RUN buildDeps='gcc libc6-dev make wget' \
    && apt-get update \
    && apt-get install -y $buildDeps \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz" \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -rf /var/lib/apt/lists/* \
    && rm redis.tar.gz \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps
```

5.3.1.3 更多指令

参考

5.3.2 构建镜像

构建命令：

```
$ docker build -t name:tag pathOfContext

# 更多help信息
$ docker  build --help
```

5.3.3 构建镜像上下文

参考

> 一般来说，应该会将 Dockerfile 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 .gitignore 一样的语法写一个 .dockerignore，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

5.3.4 其他构建方法