构建镜像和分发

这是本专栏的第三部分：镜像篇，共 8 篇。上一篇我为你介绍了镜像生命周期的管理。本篇和下一篇，我会为你介绍镜像的构建和分发，带你理解 Docker 所带来的优势及其所需技能。下面我们一起进入本篇的学习。

本地镜像存储

上一篇，我为你介绍了镜像生命周期管理，其中我为你介绍了可以通过 `docker pull`、`docker load`、`docker import`、`docker commit` 和 `docker build` 等方法，在本地新增镜像（可通过 `docker image ls` 查看）。

那么新增的镜像到哪里了呢？我们可以通过 `docker info` 看到 Docker Root Dir 的配置：

复制

```
(MoeLove) ➜  ~ docker info -f '{{.DockerRootDir}}'
/var/lib/docker
```

默认情况下，如果你没有对 `docker daemon` 配置 `--data-root` 参数的话，默认都是在上面这个 /var/lib/docker 目录中。

我们来看看该目录的结构：

复制

```
(MoeLove) ➜  ~ tree -L 1 /var/lib/docker/
/var/lib/docker/
├── builder
├── buildkit
├── containerd
├── containers
├── image
├── network
├── overlay2
├── plugins
├── runtimes
├── swarm
├── tmp
├── trust
└── volumes

13 directories, 0 files
```

可以看到其中有各种各样的文件目录，不过我们现在暂时先不去研究其他的目录，只关注于镜像所在的目录，也就是 image 目录：

复制

```
(MoeLove) ➜  ~ tree /var/lib/docker/image/
/var/lib/docker/image/
└── overlay2
    ├── distribution
    ├── imagedb
    │   ├── content
    │   │   └── sha256
    │   └── metadata
    │       └── sha256
    ├── layerdb
    └── repositories.json

8 directories, 1 file
```

看到目录结构后，我们暂且不进行解读，我们先来使用最常规的办法 `docker pull` 来给本地增加一个镜像，以 Alpine Linux 的 Docker 镜像为例（这个镜像体积比较小）：

复制

```
(MoeLove) ➜  ~ docker pull alpine:latest
latest: Pulling from library/alpine
...
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

查看在本地的镜像记录：

复制

```
(MoeLove) ➜  ~ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SI
alpine              latest              965ea09ff2eb        3 weeks ago         5.
```

我们再会过头看看刚才那个目录的变化：

复制

```
(MoeLove) ➜  tree /var/lib/docker/image/
/var/lib/docker/image/
└── overlay2
    ├── distribution
    ├── imagedb
    │   ├── content
    │   │   └── sha256
    │   │       └── 965ea09ff2ebd2b9eeec88cd822ce156f6674c7e99be082c7efac3c62
    │   └── metadata
    │       └── sha256
    ├── layerdb
    │   ├── sha256
    │   │   └── 77cae8ab23bf486355d1b3191259705374f4a11d483b24964d2f729dd8c07
    │   │       ├── cache-id
    │   │       ├── diff
    │   │       ├── size
    │   │       └── tar-split.json.gz
    │   └── tmp
    └── repositories.json

11 directories, 6 files
```

可以注意到该目录的第一层级，有一个名为 repositories.json 的文件，我们以此文件来入手，查看其内容。

复制

```
(MoeLove) ➜  jq . /var/lib/docker/image/overlay2/repositories.json
{
  "Repositories": {
    "alpine": {
      "alpine:latest": "sha256:965ea09ff2ebd2b9eeec88cd822ce156f6674c7e99be082c7ef
    }
  }
}
```

很明显，这个文件相当于是 Docker 本地镜像存储的索引文件。我们继续看看这个目录中 imagedb 目录的内容：

复制

```
# 单独再看一次，是为了便于大家观察
(MoeLove) ➜  tree /var/lib/docker/image/overlay2/imagedb/
/var/lib/docker/image/overlay2/imagedb/
├── content
│   └── sha256
│       └── 965ea09ff2ebd2b9eeec88cd822ce156f6674c7e99be082c7efac3c62f3ff652
└── metadata
    └── sha256

4 directories, 1 file
```

暂时先记住当前的目录结构及文件名，是不是感觉有点熟悉？

`docker image ls` **支持一个** `--no-trunc` **的参数**，我们来运行下看看结果：

复制

```
(MoeLove) ➜  docker image ls --no-trunc
REPOSITORY          TAG                 IMAGE ID
alpine              latest              sha256:965ea09ff2ebd2b9eeec88cd822ce156f66
```

注意：IMAGE ID 那一列，它跟 /var/lib/docker/image/overlay2/imagedb/content 目录中的结构是完全一致的。（你有没有觉得它们之间存在某些联系？）

我们看看这个文件的内容（考虑到篇幅原因，只看其中的 rootfs 字段了）：

复制

```
(MoeLove) ➜  jq .rootfs /var/lib/docker/image/overlay2/imagedb/content/sha256/965
{
  "type": "layers",
  "diff_ids": [
    "sha256:77cae8ab23bf486355d1b3191259705374f4a11d483b24964d2f729dd8c076a0"
  ]
}
```

可以看到，其中 rootfs 字段中内容的 type 是 layers，我们回过头再看看刚才目录中另一个文件夹的内容：

复制

```
(MoeLove) ➜  tree  /var/lib/docker/image/overlay2/layerdb/
/var/lib/docker/image/overlay2/layerdb/
├── sha256
│   └── 77cae8ab23bf486355d1b3191259705374f4a11d483b24964d2f729dd8c076a0
│       ├── cache-id
│       ├── diff
│       ├── size
│       └── tar-split.json.gz
└── tmp

3 directories, 4 files
```

注意这里目录的名称，对比上面 rootfs 中的内容，可以看到它们是完全一致的。

小结

前面的内容，我带你探究了 Docker 本地镜像存储的目录结构和内容，可以看到实际我们的镜像在运行着 Docker 服务的本地都是直接存储在磁盘中的，这也说明，我们可以通过一个运行中的 Docker 服务作为镜像的存储。（不过关于具体的存储实现及逻辑，不是本篇的重点，暂且跳过，后续存储篇会深入介绍）

无论直接使用磁盘存储，还是通过 `docker save/docker load` 方法进行镜像的打包、传输、加载，虽然都可以实现镜像分发的目标，但在实际使用的过程中也容易暴露出一些问题。

比如，在传输中可能出现文件数据损坏；如果镜像体积很大的话，传输大文件也不是很方便，另外，如果使用一个运行着 Docker 的机器来存储镜像的话，也可能会有单点问题，基于这些考虑，Docker 在诞生之初便提供了 `docker push/docker pull` 与 registry （比如：Docker Hub）交互，进行镜像的分发。

Docker 会帮我们完成镜像完整性的校验、镜像分层、并发传输、缓存校验等功能。所以在 Docker 推出后，立即受到了广大开发者的喜爱，大家纷纷使用 Docker Hub 来共享自己的 Docker 镜像，或者使用别人的镜像。这也是 Docker 成功的一个关键。

registry

前面已经介绍了，我们常规的进行镜像分发的方式，除了通过 `docker save` 保存镜像再分享外，更为优雅的方式是通过 registry 进行镜像的存储，用 `docker push` 和 `docker pull` 与其进行交互。

这里我为你介绍几个比较常用的 registry：

- Docker Hub：Docker 官方维护，在 Docker Hub 上你不仅可以找到很多 Docker 官方镜像，还有多数其他软件的官方镜像, 及用户自行分享的其他镜像，并且 Docker Hub 功能非常完善，非常适合新手使用；

- Docker Registry：Docker 官方开源的 registry，只需要一行 `docker run -d -p 5000:5000 --name registry registry:2` 便可启动该 registry 开始使用；
- Harbor：由 VMware 开源，后托管于 CNCF，当前项目比较活跃，可快速部署，作为私有 registry 使用。它支持较多功能，有不少企业在使用；
- Quay：日前由 RedHat 宣布开源，提供类似于 Docker Hub 这样的云服务，但功能较少；
- GitLab Container Registry：在 GitLab 中自带，独立使用或者配合 GitLab CI 使用均可。

镜像构建

现在镜像的分发方式已经介绍过了，但是我们还没学会如何构建镜像。接下来，我们进入镜像构建相关的内容的学习。

回顾上篇，已经有好几种办法可以“构建”出来镜像了。

docker commit

可以通过 `docker run` 命令启动任意容器，对其进行修改。之后通过 `docker commit` 命令，便可从此容器的修改创建出一个新的镜像。

- 使用此方式的优势在于简单，任何启动的容器均可进行 commit 操作；
- 劣势在于不可维护，没人能完全记住有哪些修改，不够直观，也不好做优化。

docker export & docker import

`docker export` 可以将任意容器的文件系统导出为一个 tar 包；而 `docker import` 则可以将此 tar 包导入为一个文件系统镜像。

但要注意的是，`docker export` 导出时只有文件系统，会丢失一些额外的配置信息，虽然在 `docker import` 的时候，可以通过 `-c` 参数额外地增加一些，但未免也过于麻烦，且这样子也不利于后续的维护。

docker build

这是使用最多、也最为推荐的方式。从一个 Dockerfile 文件构建出镜像，Dockerfile 中描述了镜像的构建过程及对其所做的修改。

类似这样：

复制

```
(MoeLove) ➜  mkdir build
(MoeLove) ➜  cd build
(MoeLove) ➜  build cat <<EOF > Dockerfile
FROM alpine:latest
RUN apk add --no-cache bash
EOF

(MoeLove) ➜  build docker build -t local/alpine:bash .
[+] Building 442.6s (6/6) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 149B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/alpine:latest
 => CACHED [1/2] FROM docker.io/library/alpine:latest
 => [2/2] RUN apk add --no-cache bash
 => exporting to image
 => => exporting layers
 => => writing image sha256:1baea8f38e49838dd071cdf5960171feb2e8473c9e2e72f216b43b
 => => naming to docker.io/local/alpine:bash
(MoeLove) ➜  build docker run --rm local/alpine:bash bash --version
GNU bash, version 5.0.0(1)-release (x86_64-alpine-linux-musl)
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

通过 FROM 指令，表示基于 `alpine:latest` 镜像进行构建；使用 RUN 指令，执行需要完成的动作。构建完成后，可以看到构建成功的镜像中已经包含了我们安装的 bash。

总结

本篇我为你介绍了镜像的本地存储，镜像的分发，几种常用的 registry 和镜像构建的几种基本方式。

由于镜像构建所使用的 Dockerfile 中包含的知识点比较多，所以我专门准备了下一篇，来集中地为你介绍 Dockerfile 的重点知识和一些扩展知识。后续的内容，我们会频繁的使用 `docker build` 命令，以及去深入了解其背后的原理。希望能让你在构建镜像时，能游刃有余。