title: Docker(三)镜像 tags: Docker categories: 技术

Docker 镜像

我们都知道，操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。而 Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:14.04 就包含了完整的一套 Ubuntu 14.04 最小系统的 root 文件系统。

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

分层存储

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 Union FS 的技术，将其设计为分层存储的架构。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

实现原理

Docker 镜像是怎么实现增量的修改和维护的？ 每个镜像都由很多层次构成，Docker 使用 Union FS 将这些不同的层结合到一个镜像中去。

通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个 disk 挂到同一个目录下,另一个更常用的就是将一个只读的分支和一个可写的分支联合在一起，Live CD 正是基于此方法可以允许在镜像不变的基础上允许用户在其上进行一些写操作。 Docker 在 AUFS 上构建的容器也是利用了类似的原理。

镜像的操作

列出镜像

要想列出已经下载下来的镜像，可以使用 docker images 命令。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
redis               latest              5f515359c7f8        5 days ago
```

```
183 MB
nginx               latest              05a60462f8ba        5 days ago
181 MB
mongo               3.2                 fe9198c04d62        5 days ago
342 MB
<none>              <none>              00285df0df87        5 days ago
342 MB
ubuntu              16.04               f753707788c5        4 weeks ago
127 MB
ubuntu              latest              f753707788c5        4 weeks ago
127 MB
ubuntu              14.04               1e0c3dd64ccd        4 weeks ago
188 MB
```

列表包含了仓库名、标签、镜像 ID、创建时间以及所占用的空间。

其中仓库名、标签在之前的基础概念章节已经介绍过了。**镜像 ID** 则是镜像的唯一标识，一个镜像可以对应多个**标签**。因此，在上面的例子中，我们可以看到 ubuntu:16.04 和 ubuntu:latest 拥有相同的 ID，因为它们对应的是同一个镜像。

镜像体积

如果仔细观察，会注意到，这里标识的所占用空间和在 Docker Hub 上看到的镜像大小不同。比如，ubuntu:16.04 镜像大小，在这里是 127 MB，但是在 Docker Hub 显示的却是 50 MB。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中镜像是保持着压缩状态的，因此 Docker Hub 所显示的大小是网络传输中更关心的流量大小。而 docker images 显示的是镜像下载到本地后，展开的大小，准确说，是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

另外一个需要注意的问题是，docker images 列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。

虚悬镜像

上面的镜像列表中，还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为 <none>。：

```
<none>              <none>              00285df0df87        5 days ago
342 MB
```

这个镜像原本是有镜像名和标签的，原来为 mongo:3.2，随着官方镜像维护，发布了新版本后，重新 docker pull mongo:3.2 时，mongo:3.2 这个镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消，从而成为了 <none>。除了 docker pull 可能导致这种情况，docker build 也同样可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 <none> 的镜像。这类无标签镜像也被称为 **虚悬镜像(dangling image)**，可以用下面的命令专门显示这类镜像：

```
$ docker images -f dangling=true
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
<none>              <none>              00285df0df87        5 days ago
342 MB
```

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 rmi $(docker images -q -f dangling=true)
```

中间层镜像

为了加速镜像构建、重复利用资源，Docker 会利用 **中间层镜像**。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 docker images 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加 -a 参数。

```
$ docker images -a
```

这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们。只要删除那些依赖它们的镜像后，这些依赖的中间层镜像也会被连带删除。

列出部分镜像

不加任何参数的情况下，docker images 会列出所有顶级镜像，但是有时候我们只希望列出部分镜像。docker images 有好几个参数可以帮助做到这个事情。

根据仓库名列出镜像

```
$ docker images ubuntu
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
ubuntu              16.04               f753707788c5        4 weeks ago
127 MB
ubuntu              latest              f753707788c5        4 weeks ago
127 MB
ubuntu              14.04               1e0c3dd64ccd        4 weeks ago
188 MB
```

列出特定的某个镜像，也就是说指定仓库名和标签

```
$ docker images ubuntu:16.04
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
ubuntu              16.04               f753707788c5        4 weeks ago
127 MB
```

除此以外，docker images 还支持强大的过滤器参数 --filter，或者简写 -f。之前我们已经看到了使用过滤器来列出虚悬镜像的用法，它还有更多的用法。比如，我们希望看到在 mongo:3.2 之后建立的镜像，可以用下面的命令：

```
$ docker images -f since=mongo:3.2
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
redis               latest              5f515359c7f8        5 days ago
183 MB
nginx               latest              05a60462f8ba        5 days ago
181 MB
```

想查看某个位置之前的镜像也可以，只需要把 since 换成 before 即可。

此外，如果镜像构建时，定义了 LABEL，还可以通过 LABEL 来过滤。

```
$ docker images -f label=com.example.version=0.1
...
```

以特定格式显示

默认情况下，docker images 会输出一个完整的表格，但是我们并非所有时候都会需要这些内容。比如，刚才删除虚悬镜像的时候，我们需要利用 docker images 把所有的虚悬镜像的 ID 列出来，然后才可以交给 docker rmi 命令作为参数来删除指定的这些镜像，这个时候就用到了 -q 参数。

```
$ docker images -q
5f515359c7f8
05a60462f8ba
fe9198c04d62
00285df0df87
f753707788c5
f753707788c5
1e0c3dd64ccd
```

--filter 配合 -q 产生出指定范围的 ID 列表，然后送给另一个 docker 命令作为参数，从而针对这组实体成批的进行某种操作的做法在 Docker 命令行使用过程中非常常见，不仅仅是镜像，将来我们会在各个命令中看到这类搭配以完成很强大的功能。因此每次在文档看到过滤器后，可以多注意一下它们的用法。

另外一些时候，我们可能只是对表格的结构不满意，希望自己组织列；或者不希望有标题，这样方便其它程序解析结果等，这就用到了 Go 的模板语法。

比如，下面的命令会直接列出镜像结果，并且只包含镜像ID和仓库名：

```
$ docker images --format "{{.ID}}: {{.Repository}}"
5f515359c7f8: redis
05a60462f8ba: nginx
fe9198c04d62: mongo
00285df0df87: <none>
f753707788c5: ubuntu
f753707788c5: ubuntu
1e0c3dd64ccd: ubuntu
```

或者打算以表格等距显示，并且有标题行，和默认一样，不过自己定义列：

```
$ docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"
IMAGE ID            REPOSITORY          TAG
5f515359c7f8        redis               latest
05a60462f8ba        nginx               latest
fe9198c04d62        mongo               3.2
00285df0df87        <none>              <none>
f753707788c5        ubuntu              16.04
f753707788c5        ubuntu              latest
1e0c3dd64ccd        ubuntu              14.04
```

获取镜像

Docker Hub 上有大量的高质量的镜像可以用，这里我们就说一下怎么获取这些镜像并运行。

从 Docker Registry 获取镜像的命令是 docker pull。其命令格式为：

```
docker pull [选项] [Docker Registry地址]<仓库名>:<标签>
```

具体的选项可以通过 docker pull --help 命令看到，这里我们说一下镜像名称的格式。

- Docker Registry地址：地址的格式一般是 <域名/IP>[:端口号]。默认地址是 Docker Hub。
- 仓库名：如之前所说，这里的仓库名是两段式名称，既 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。

比如：

```
$ docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu
bf5d46315322: Pull complete
9f13e0ac480c: Pull complete
e8988b5b3097: Pull complete
40af181810e7: Pull complete
e6f7c7e5c03e: Pull complete
Digest:
sha256:147913621d9cdea08853f6ba9116c2e27a3ceffecf3b492983ae97c3d643fbbe
Status: Downloaded newer image for ubuntu:14.04
```

上面的命令中没有给出 Docker Registry 地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 ubuntu:14.04，因此将会获取官方镜像 library/ubuntu 仓库中标签为 14.04 的镜像。

从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的 ID 的前 12 位。并且下载结束后，给出该镜像完整的 sha256 的摘要，以确保下载一致性。

在实验上面命令的时候，你可能会发现，你所看到的层 ID 以及 sha256 的摘要和这里的不一样。这是因为官方镜像是一直在维护的，有任何新的 bug，或者版本更新，都会进行修复再以原来的标签发布，这样可以确保任何使用这个标签的用户可以获得更安全、更稳定的镜像。

如果从 Docker Hub 下载镜像非常缓慢，可以参照第二章节中配置国内镜像源加速器。

运行

有了镜像后，我们就可以以这个镜像为基础启动一个容器来运行。以上面的 ubuntu:14.04 为例，如果我们打算启动里面的 bash 并且进行交互式操作的话，可以执行下面的命令。

```
$ docker run -it --rm ubuntu:14.04 bash
root@e7009c6ce357:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.5 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.5 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
root@e7009c6ce357:/# exit
exit
$
```

docker run 就是运行容器的命令，具体格式我们会在后面的章节讲解，我们这里简要的说明一下上面用到的参数。

- -it：这是两个参数，一个是 -i：交互式操作，一个是 -t 终端。我们这里打算进入 bash 执行一些命令并查看返回结果，因此我们需要交互式终端。
- --rm：这个参数是说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。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 --rm 可以避免浪费空间。
- ubuntu:14.04：这是指用 ubuntu:14.04 镜像为基础来启动容器。
- bash：放在镜像名后的是**命令**，这里我们希望有个交互式 Shell，因此用的是 bash。

进入容器后，我们可以在 Shell 下操作，执行任何所需的命令。这里，我们执行了 cat /etc/os-release，这是 Linux 常用的查看当前系统版本的命令，从返回的结果可以看到容器内是 Ubuntu 14.04.5 LTS 系统。

最后我们通过 exit 退出了这个容器。

利用 commit 理解镜像构成

镜像是容器的基础，每次执行 docker run 的时候都会指定哪个镜像作为容器运行的基础。在之前的例子中，我们所使用的都是来自于 Docker Hub 的镜像。直接使用这些镜像是可以满足一定的需求，而当这些镜像无法直接满足需求时，我们就需要定制这些镜像。

回顾一下，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是在以镜像为基础层，在其基础上加一层作为容器运行时的存储层。

现在让我们以定制一个 Web 服务器为例子，来讲解镜像是如何构建的。

```
docker run --name webserver -d -p 80:80 nginx
```

这条命令会用 nginx 镜像启动一个容器，命名为 webserver，并且映射了 80 端口，这样我们可以用浏览器去访问这个 nginx 服务器。

如果是在 Linux 本机运行的 Docker，或者如果使用的是 Docker for Mac、Docker for Windows，那么可以直接访问：http://localhost；如果使用的是 Docker Toolbox，或者是在虚拟机、云服务器上安装的 Docker，则需要将 localhost 换为虚拟机地址或者实际云服务器地址,我们这里采用了宿主机IP访问。

直接用浏览器访问的话，我们会看到默认的 Nginx 欢迎页面。

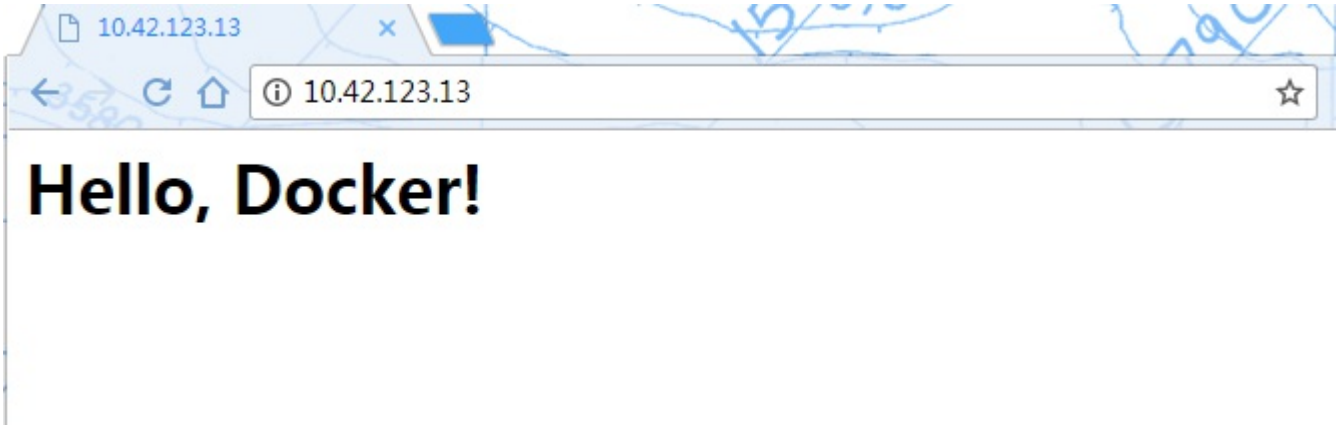
现在，假设我们非常不喜欢这个欢迎页面，我们希望改成欢迎 Docker 的文字，我们可以使用 docker exec 命令进入容器，修改其内容。

```
$ docker exec -it webserver bash
root@d414c7adba5a:/# echo '<h1>Hello, Docker!</h1>' >
/usr/share/nginx/html/index.html
root@d414c7adba5a:/# exit
exit
```

我们以交互式终端方式进入 webserver 容器，并执行了 bash 命令，也就是获得一个可操作的 Shell。

然后，我们用 <h1>Hello, Docker!</h1> 覆盖了 /usr/share/nginx/html/index.html 的内容。

现在我们再刷新浏览器的话，会发现内容被改变了。



我们修改了容器的文件，也就是改动了容器的存储层。我们可以通过 docker diff 命令看到具体的改动。

```
$ docker diff webserver
C /root
A /root/.bash_history
```

```
C /run
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
C /var
C /var/cache
C /var/cache/nginx
A /var/cache/nginx/client_temp
A /var/cache/nginx/fastcgi_temp
A /var/cache/nginx/proxy_temp
A /var/cache/nginx/scgi_temp
A /var/cache/nginx/uwsgi_temp
```

现在我们定制好了变化，我们希望能将其保存下来形成镜像。

要知道，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。而 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。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。

docker commit 的语法格式为：

```
docker commit [选项] <容器ID或容器名> [<仓库名>[:<标签>]]
```

我们可以用下面的命令将容器保存为镜像：

```
$ docker commit \
    --author "LT" \
    --message "update index.html" \
    webserver \
    nginx:v2
sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa1795214
```

其中 --author 是指定修改的作者，而 --message 则是记录本次修改的内容。这点和 git 版本控制相似，不过这里这些信息可以省略留空。

我们可以在 docker images 中看到这个新定制的镜像：

```
$ docker images nginx
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
nginx               v2                  07e334659748        9 seconds ago
181.5 MB
```

```
nginx               1.11                05a60462f8ba        12 days ago
181.5 MB
nginx               latest              b8efb18f159b        4 weeks ago
181.5 MB```

我们还可以用 `docker history` 具体查看镜像内的历史记录，如果比较 `nginx:latest` 的历史记录，我们会发现新增了我们刚刚提交的这一层。

```bash
$ docker history nginx:v2
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
07e334659748        54 seconds ago      nginx -g daemon off;
95 B                修改了默认网页
b8efb18f159b        4 weeks ago         /bin/sh -c #(nop)  CMD ["nginx" "-g"
"daemon    0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  EXPOSE 443/tcp
80/tcp        0 B
<missing>           4 weeks ago         /bin/sh -c ln -sf /dev/stdout
/var/log/nginx/   22 B
<missing>           4 weeks ago         /bin/sh -c apt-key adv --keyserver
hkp://pgp.   58.46 MB
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENV
NGINX_VERSION=1.11.5-1   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  MAINTAINER NGINX
Docker Ma   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]
0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD
file:23aa4f893e3288698c   123 MB
```

新的镜像定制好后，我们可以来运行这个镜像。

```
docker run --name web2 -d -p 81:80 nginx:v2
```

这里我们命名为新的服务为 web2，并且映射到 81 端口。如果是 Docker for Mac/Windows 或 Linux 桌面的话，我们就可以直接访问 http://localhost:81 看到结果，其内容应该和之前修改后的 webserver 一样。

至此，我们第一次完成了定制镜像，使用的是 docker commit 命令，手动操作给旧的镜像添加了新的一层，形成新的镜像，对镜像多层存储应该有了更直观的感觉。

慎用 docker commit

使用 docker commit 命令虽然可以比较直观的帮助理解镜像分层存储的概念，但是实际环境中并不会这样使用。

首先，如果仔细观察之前的 docker diff webserver 的结果，你会发现除了真正想要修改的 /usr/share/nginx/html/index.html 文件外，由于命令的执行，还有很多文件被改动或添加了。这还仅仅是最简
```

单的操作，如果是安装软件包、编译构建，那会有大量的无关内容被添加进来，如果不小心清理，将会导致镜像极为臃肿。

此外，使用 docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为**黑箱镜像**，换句话说，就是除了制作镜像的人知道执行过什么命令、怎么生成的镜像，别人根本无从得知。而且，即使是这个制作镜像的人，过一段时间后也无法记清具体在操作的。虽然 docker diff 或许可以告诉得到一些线索，但是远远不到可以确保生成一致镜像的地步。这种黑箱镜像的维护工作是非常痛苦的。

而且，回顾之前提及的镜像所使用的分层存储的概念，除当前层外，之前的每一层都是不会发生改变的，换句话说，任何修改的结果仅仅是在当前层进行标记、添加、修改，而不会改动上一层。如果使用 docker commit 制作镜像，以及后期修改的话，每一次修改都会让镜像更加臃肿一次，所删除的上一层的东西并不会丢失，会一直如影随形的跟着这个镜像，即使根本无法访问到™。这会让镜像更加臃肿。

docker commit 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保存现场等。但是，不要使用 docker commit 定制镜像，定制行为应该使用 Dockerfile 来完成。下面的章节我们就来讲述一下如何使用 Dockerfile 定制镜像。

使用 Dockerfile 定制镜像

从刚才的 docker commit 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内包含了一条条的**指令(Instruction)**，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

还以之前定制 nginx 镜像为例，这次我们使用 Dockerfile 来定制。

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

```
$ mkdir mynginx
$ cd mynginx
$ touch Dockerfile
```

其内容为：

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

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。

FROM 指定基础镜像

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 FROM 就是指定**基础镜像**，因此一个 Dockerfile 中 FROM 是必

备的指令，并且必须是第一条指令。

在 Docker Hub[^1] 上有非常多的高质量的官方镜像， 有可以直接拿来使用的服务类的镜像，如 nginx、redis、mongo、mysql、httpd、php、tomcat 等； 也有一些方便开发、构建、运行各种语言应用的镜像，如 node、openjdk、python、ruby、golang 等。 可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。 如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 ubuntu、debian、centos、fedora、alpine 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

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。

```
FROM scratch
...
```

如果你以 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。

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。

RUN 执行命令

RUN 指令是用来执行命令行命令的。由于命令行的强大能力，RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：

- *shell* 格式：RUN <命令>，就像直接在命令行中输入的命令一样。刚才写的 Dockrfile 中的 RUN 指令就是这种格式。

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

- *exec* 格式：RUN ["可执行文件", "参数1", "参数2"]，这更像是函数调用中的格式。

既然 RUN 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 RUN 呢？比如这样：

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y gcc libc6-dev make
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-
3.2.5.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
```

之前说过，Dockerfile 中每一个指令都会建立一层，RUN 也不例外。每一个 RUN 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，commit 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。 这是很多初学 Docker 的人常犯的一个错误。

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。

上面的 Dockerfile 正确的写法应该是这样：

```
FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \
    && apt-get update \
    && apt-get install -y $buildDeps \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-
3.2.5.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
```

首先，之前所有的命令只有一个目的，就是编译、安装 redis 可执行文件。因此没有必要建立很多层，这只是一层的事情。因此，这里没有使用很多个 RUN 对一一对应不同的命令，而是仅仅使用一个 RUN 指令，并使用 && 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。在撰写 Dockerfile 的时候，要经常提醒自己，这并不是在写 Shell 脚本，而是在定义每一层该如何构建。

并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 \ 的命令换行方式，以及行首 # 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 apt 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，一定要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉。

很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

构建镜像

好了，让我们再回到之前定制的 nginx 镜像的 Dockerfile 来。现在我们明白了这个 Dockerfile 的内容，那么让我们来构建这个镜像吧。

在 Dockerfile 文件所在目录执行：

```
$ docker build -t nginx:v3 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM nginx
 ---> b8efb18f159b
Step 2 : RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
 ---> Running in 0cfef91111ed
 ---> 706dd13cc995
Removing intermediate container 0cfef91111ed
Successfully built 706dd13cc995
```

从命令的输出结果中，我们可以清晰的看到镜像的构建过程。在 Step 2 中，如同我们之前所说的那样，RUN 指令启动了一个容器 0cfef91111ed，执行了所要求的命令，并最后提交了这一层 706dd13cc995，随后删除了所用到的这个容器 0cfef91111ed。

这里我们使用了 docker build 命令进行镜像构建。其格式为：

```
docker build [选项] <上下文路径/URL/->
```

在这里我们指定了最终镜像的名称 -t nginx:v3，构建成功后，我们可以像之前运行 nginx:v2 那样来运行这个镜像，其结果会和 nginx:v2 一样。

镜像构建上下文（Context）

如果注意，会看到 docker build 命令最后有一个 .。. 表示当前目录，而 Dockerfile 就在当前目录，因此不少初学者以为这个路径是在指定 Dockerfile 所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定**上下文路径**。那么什么是上下文呢？

首先我们要理解 docker build 的工作原理。Docker 在运行时分为 Docker 引擎（也就是服务端守护进程）和客户端工具。Docker 的引擎提供了一组 REST API，被称为 Docker Remote API，而如 docker 命令这样的客户端工具，则是通过这组 API 与 Docker 引擎交互，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种 docker 功能，但实际上，一切都是使用的远程调用形式在服务端（Docker 引擎）完成。也因为这种 C/S 设计，让我们操作远程服务器的 Docker 引擎变得轻而易举。

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就

会获得构建镜像所需的一切文件。

如果在 Dockerfile 中这么写：

```
COPY ./package.json /app/
```

这并不是要复制执行 docker build 命令所在的目录下的 package.json，也不是复制 Dockerfile 所在目录下的 package.json，而是复制 **上下文（context）** 目录下的 package.json。

因此，COPY 这类指令中的源文件的路径都是*相对路径*。这也是初学者经常会问的为什么 COPY ../package.json /app 或者 COPY /opt/xxxx /app 无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

现在就可以理解刚才的命令 docker build -t nginx:v3 . 中的这个 .，实际上是在指定上下文的目录，docker build 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像。

如果观察 docker build 输出，我们其实已经看到了这个发送上下文的过程：

```
$ docker build -t nginx:v3 .
Sending build context to Docker daemon 2.048 kB
...
```

理解构建上下文对于镜像构建是很重要的，避免犯一些不应该的错误。比如有些初学者在发现 COPY /opt/xxxx /app 不工作后，于是干脆将 Dockerfile 放到了硬盘根目录去构建，结果发现 docker build 执行后，在发送一个几十 GB 的东西，极为缓慢而且很容易构建失败。那是因为这种做法是在让 docker build 打包整个硬盘，这显然是使用错误。

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。

那么为什么会有人误以为 . 是指定 Dockerfile 所在目录呢？这是因为在默认情况下，如果不额外指定 Dockerfile 的话，会将上下文目录下的名为 Dockerfile 的文件作为 Dockerfile。

这只是默认行为，实际上 Dockerfile 的文件名并不要求必须为 Dockerfile，而且并不要求必须位于上下文目录中，比如可以用 -f ../Dockerfile.php 参数指定某个文件作为 Dockerfile。

当然，一般大家习惯性的会使用默认的文件名 Dockerfile，以及会将其置于镜像构建上下文目录中。

其它 docker build 的用法

直接用 Git repo 进行构建

或许你已经注意到了，docker build 还支持从 URL 构建，比如可以直接从 Git repo 中构建：

```
$ docker build https://github.com/twang2218/gitlab-ce-zh.git#:8.14
docker build https://github.com/twang2218/gitlab-ce-zh.git\#:8.14
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM gitlab/gitlab-ce:8.14.0-ce.0
8.14.0-ce.0: Pulling from gitlab/gitlab-ce
aed15891ba52: Already exists
773ae8583d14: Already exists
...
```

这行命令指定了构建所需的 Git repo，并且指定默认的 master 分支，构建目录为 /8.14/，然后 Docker 就会自己去 git clone 这个项目、切换到指定分支、并进入到指定目录后开始构建。

用给定的 tar 压缩包构建

```
$ docker build http://server/context.tar.gz
```

如果所给出的 URL 不是个 Git repo，而是个 tar 压缩包，那么 Docker 引擎会下载这个包，并自动解压缩，以其作为上下文，开始构建。

从标准输入中读取 Dockerfile 进行构建

```
docker build - < Dockerfile
```

或

```
cat Dockerfile | docker build -
```

如果标准输入传入的是文本文件，则将其视为 Dockerfile，并开始构建。这种形式由于直接从标准输入中读取 Dockerfile 的内容，它没有上下文，因此不可以像其他方法那样可以将本地文件 COPY 进镜像之类的事情。

从标准输入中读取上下文压缩包进行构建

```
$ docker build - < context.tar.gz
```

如果发现标准输入的文件格式是 gzip、bzip2 以及 xz 的话，将会使其为上下文压缩包，直接将其展开，将里面视为上下文，并开始构建。

[^1]: Docker Store是发现公共Docker内容，镜像发布和发行软件的新地方

删除本地镜像

如果要删除本地的镜像，可以使用 docker rmi 命令，其格式为：

```
docker rmi [选项] <镜像1> [<镜像2> ...]
```

注意 docker rm 命令是删除容器，不要混淆。

用 ID、镜像名、摘要删除镜像

其中，<镜像> 可以是 镜像短 ID、镜像长 ID、镜像名 或者 镜像摘要。

比如我们有这么一些镜像：

```
$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED
SIZE
centos                      latest              0584b3d2cf6d        3 weeks
ago         196.5 MB
redis                       alpine              501ad78535f0        3 weeks
ago         21.03 MB
docker                      latest              cf693ec9b5c7        3 weeks
ago         105.1 MB
nginx                       latest              b8efb18f159b        5 weeks
ago         181.5 MB
```

我们可以用镜像的完整 ID，也称为 长 ID，来删除镜像。使用脚本的时候可能会用长 ID，但是人工输入就太累了，所以更多的时候是用 短 ID 来删除镜像。docker images 默认列出的就已经是短 ID 了，一般取前3个字符以上，只要足够区分于别的镜像就可以了。

比如这里，如果我们要删除 redis:alpine 镜像，可以执行：

```
$ docker rmi 501
Untagged: redis:alpine
Untagged:
redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeea521a86d
Deleted:
sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7
Deleted:
sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b
Deleted:
sha256:32770d1dcf835f192cafd6b9263b7b597a1778a403a109e2cc2ee866f74adf23
Deleted:
sha256:127227698ad74a5846ff5153475e03439d96d4b1c7f2a449c7a826ef74a2d2fa
Deleted:
sha256:1333ecc582459bac54e1437335c0816bc17634e131ea0cc48daa27d32c75eab3
```

```
Deleted:
sha256:4fc455b921edf9c4aea207c51ab39b10b06540c8b4825ba57b3feed1668fa7c7
```

我们也可以用镜像名，也就是 <仓库名>:<标签>，来删除镜像。

```
$ docker rmi centos
Untagged: centos:latest
Untagged:
centos@sha256:b2f9d1c0ff5f87a4743104d099a3d561002ac500db1b9bfa02a783a46e0d366c
Deleted:
sha256:0584b3d2cf6d235ee310cf14b54667d889887b838d3f3d3033acd70fc3c48b8a
Deleted:
sha256:97ca462ad9eeae25941546209454496e1d66749d53dfa2ee32bf1faabd239d38
```

当然，更精确的是使用 镜像摘要 删除镜像。

```
$ docker images --digests
REPOSITORY                  TAG                 DIGEST
IMAGE ID            CREATED             SIZE
node                        slim
sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228
6e0c4c8e3913        3 weeks ago         214 MB

$ docker rmi
node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228
Untagged:
node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228
```

Untagged 和 Deleted

如果观察上面这几个命令的运行输出信息的话，你会注意到删除行为分为两类，一类是 Untagged，另一类是 Deleted。我们之前介绍过，镜像的唯一标识是其 ID 和摘要，而一个镜像可以有多个标签。

因此当我们使用上面命令删除镜像的时候，实际上是在要求删除某个标签的镜像。所以首先需要做的是将满足我们要求的所有镜像标签都取消，这就是我们看到的 Untagged 的信息。因为一个镜像可以对应多个标签，因此当我们删除了所指定的标签后，可能还有别的标签指向了这个镜像，如果是这种情况，那么 Delete 行为就不会发生。所以并非所有的 docker rmi 都会产生删除镜像的行为，有可能仅仅是取消了某个标签而已。

当该镜像所有的标签都被取消了，该镜像很可能会失去了存在的意义，因此会触发删除行为。镜像是多层存储结构，因此在删除的时候也是从上层向基础层方向依次进行判断删除。镜像的多层结构让镜像复用变动非常容易，因此很有可能某个其它镜像正依赖于当前镜像的某一层。这种情况，依旧不会触发删除该层的行为。直到没有任何层依赖当前层时，才会真实的删除当前层。这就是为什么，有时候会奇怪，为什么明明没有别的标签指向这个镜像，但是它还是存在的原因，也是为什么有时候会发现所删除的层数和自己 docker pull 看到的层数不一样的源。

除了镜像依赖以外，还需要注意的是容器对镜像的依赖。如果有用这个镜像启动的容器存在（即使容器没有运行），那么同样不可以删除这个镜像。之前讲过，容器是以镜像为基础，再加一层容器存储层，组成这样的多层存储结构去运行的。因此该镜像如果被这个容器所依赖的，那么删除必然会导致故障。如果这些容器是不需要的，应该先将它们删除，然后再来删除镜像。

用 docker images 命令来配合

像其它可以承接多个实体的命令一样，可以使用 docker images -q 来配合使用 docker rmi，这样可以成批的删除希望删除的镜像。比如之前我们介绍过的，删除虚悬镜像的指令是：

```
$ docker rmi $(docker images -q -f dangling=true)
```

我们在"镜像列表"章节介绍过很多过滤镜像列表的方式都可以拿过来使用。

比如，我们需要删除所有仓库名为 redis 的镜像：

```
$ docker rmi $(docker images -q redis)
```

或者删除所有在 mongo:3.2 之前的镜像：

```
$ docker rmi $(docker images -q -f before=mongo:3.2)
```

充分利用你的想象力和 Linux 命令行的强大，你可以完成很多非常赞的功能。

CentOS/RHEL 的用户需要注意的事项

在 Ubuntu/Debian 上有 UnionFS 可以使用，如 aufs 或者 overlay2，而 CentOS 和 RHEL 的内核中没有相关驱动。因此对于这类系统，一般使用 devicemapper 驱动利用 LVM 的一些机制来模拟分层存储。这样的做法除了性能比较差外，稳定性一般也不好，而且配置相对复杂。Docker 安装在 CentOS/RHEL 上后，会默认选择 devicemapper，但是为了简化配置，其 devicemapper 是跑在一个稀疏文件模拟的块设备上，也被称为 loop-lvm。这样的选择是因为不需要额外配置就可以运行 Docker，这是自动配置唯一能做到的事情。但是 loop-lvm 的做法非常不好，其稳定性、性能更差，无论是日志还是 docker info 中都会看到警告信息。官方文档有明确的文章讲解了如何配置块设备给 devicemapper 驱动做存储层的做法，这类做法也被称为配置 direct-lvm。

除了前面说到的问题外，devicemapper + loop-lvm 还有一个缺陷，因为它是稀疏文件，所以它会不断增长。用户在使用过程中会注意到 /var/lib/docker/devicemapper/devicemapper/data 不断增长，而且无法控制。很多人会希望删除镜像或者可以解决这个问题，结果发现效果并不明显。原因就是这个稀疏文件的空间释放后基本不进行垃圾回收的问题。因此往往会出现即使删除了文件内容，空间却无法回收，随着使用这个稀疏文件一直在不断增长。

所以对于 CentOS/RHEL 的用户来说，在没有办法使用 UnionFS 的情况下，一定要配置 direct-lvm 给 devicemapper，无论是为了性能、稳定性还是空间利用率。

或许有人注意到了 CentOS 7 中存在被 backports 回来的 overlay 驱动，不过 CentOS 里的这个驱动达不到生产环境使用的稳定程度，所以不推荐使用。