登录到 Harbor 仓库的 library 项目，可以查看到镜像已经上传，如图 6-13 所示。

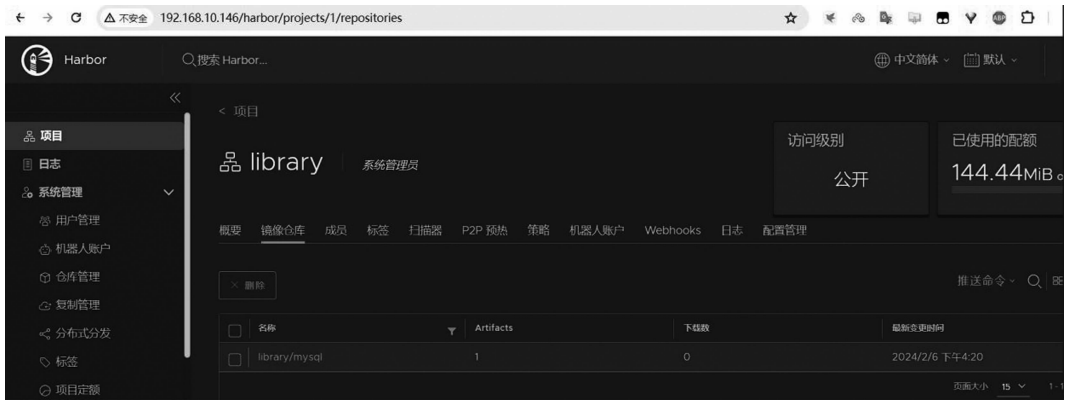


图 6-13　Harbor 仓库中的 library 项目

单击该镜像，可以进入镜像的详细信息页面。在该页面上，单击“拉取命令”按钮，以复制对应的 Docker 拉取命令到剪贴板。随后，可以将这个命令粘贴到 Docker 主机上执行，以便从 Harbor 仓库拉取该镜像，如图 6-14 所示。

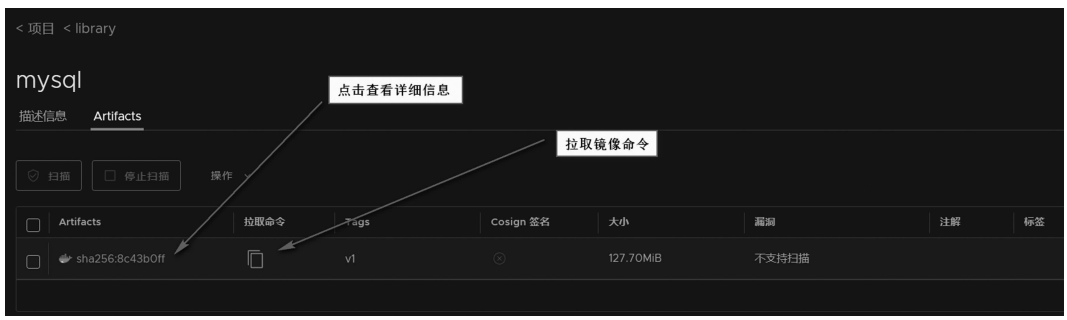


图 6-14　Harbor 仓库的镜像信息

6.2.5　素养小课堂

万豪酒店数据泄露：万豪国际酒店在 2018 年遭遇了一起大规模的数据泄漏事故，影响了多达 5 亿客户。攻击者能够访问客户的姓名、地址、电话号码、电子邮件地址、护照号码、出生日期、性别、到达和离店信息、预订日期和通信偏好等信息。这一事件对万豪酒店的声誉造成了严重损害，并导致了重大的经济损失。

当使用在线服务或注册会员时，大学生应仔细阅读隐私政策，了解企业如何收集、存储和使用个人信息。避免在不必要的场合提供过多个人信息，特别是不要在公共 Wi-Fi 等不安全环境中进行在线交易。

任务 6.3　制作 Dockerfile 镜像

6.3.1　学习目标

- 掌握 Docker 镜像的分层原理。
- 掌握 Dockerfile 文件的编写方式。

• 掌握通过 Dockerfile 构建镜像的方法。

6.3.2 知识点讲解

1. Dockerfile

Dockerfile 是一个用来构建 Docker 镜像的文本文件，其内容包含了一系列指令和说明，这些指令都会在原镜像层的基础上构建一个新的镜像层。通过使用 docker build 命令并指定 Dockerfile，系统会根据文件中的内容来构建镜像。因此，Dockerfile 中的每一条指令都详细描述了如何构建对应的镜像层。借助 Dockerfile，企业可以制作符合自己需求的 Docker 镜像。

2. Overlay2

Docker 的文件系统采用分层结构，这种结构依赖于 UnionFS(联合文件系统)技术。目前 Docker 支持的 UnionFS 实现方式有很多种，包括 AUFS、DeviceMapper、overlay、overlay2 和 VSF 等，这些实现方式统称为 Docker 的存储驱动。尽管它们在实现 UnionFS 的具体方式上有所不同，但最终效果相似，都是将文件系统的每次修改作为一层提交，这些层层层叠加，其中底层是只读层，顶层是读写层，最终所有这些层都被统一挂载到同一个虚拟文件系统下。对于使用较高版本 Linux 内核或不同 Linux 发行版的用户，建议使用 Overlay2 作为存储驱动，同时，为了使用 Overlay2，Docker 的版本也要高于 17.06.02。

Overlay2 主要由 MergedDir、LowerDir(镜像层)、UpperDir(容器层)和 WorkDir 组成。其中，LowerDir 对应底层文件系统，也就是镜像创建过程中提交的每一层，它是能被上层文件系统 UpperDir 所共享的只读层；WorkDir 则可以理解为是 Overlay2 运作的一个临时工作目录，用于完成写时复制(copy-on-write)等操作。Overlay2 运作时，会将 LowerDir、UpperDir 和 WorkDir 联合挂载到 MergedDir 目录，为用户提供一个“统一视图”，如图 6-15 所示。

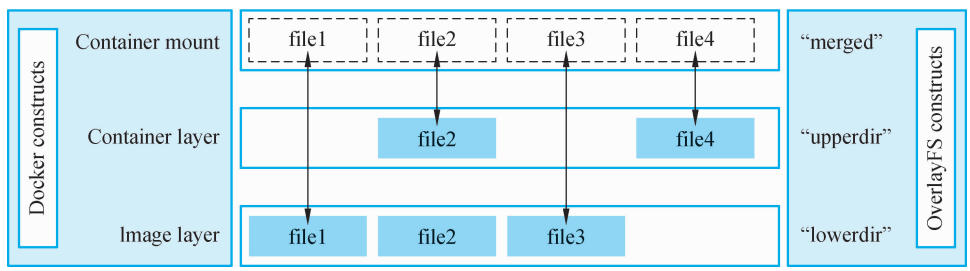


图 6-15 Overlay2 存储驱动示意图

当然，图 6-15 中的 LowerDir 不一定只有一层，可能有很多层。采用 Overlay2 存储驱动容器读取和修改文件的过程如下。

(1)读取文件。如果容器尝试读取一个文件，如果该文件在容器层(UpperDir)中不存在，系统会从镜像层(LowerDir)中读取该文件，这一过程产生的性能开销非常低。

如果容器尝试读取的文件仅存在于容器层(UpperDir)而不存在于镜像层(LowerDir)，那么该文件将直接从容器层被读取。

如果容器尝试读取的文件同时在镜像层和容器层中存在，系统将优先读取容器层中的文件。容器层中的文件会覆盖或隐藏镜像层中同名的文件。

(2)修改文件或目录。当容器首次写入一个已存在的文件，而该文件在容器层中尚不存在时，Overlay2 存储驱动会执行 copy_up 操作。这一操作会将文件从镜像层复制到容器层，从而创建一个新的文件副本。随后，容器将修改写入这个新副本中。然而，值得注意的是，OverlayFS 工作在文件级别而非块级别，这意味着即使只有文件的一小部分被修改，整个文件也会被复制。对于大文件而言，这种完整的文件复制会对容器的写入性能产生显著影响。但是，copy_up 操作仅发生在对该文件的首次写

入时，之后对该文件的任何修改都将直接作用于容器层中的这个副本，无需再次执行 copy_up 操作。

(3)删除文件和目录。在容器中删除文件时，会在容器层中创建一个空白文件。镜像层中的文件版本不会被删除(因为 LowerDir 是只读的)。当在容器中删除目录时，会在容器中创建一个不透明的目录(UpperDir)。

6.3.3 任务实施



视频

制作 Dockfile 镜像

通过 Dockerfile，用户可以自定义镜像的构建过程。Dockfile 文件中的每一行都代表一条指令，然后使用 docker build 命令根据 Dockerfile 中的指令来构建镜像。

Dockerfile 常见指令如下。

(1)FROM 指令。

①指令作用：指定构建镜像的基础镜像，一般都作为 Dockerfile 的第一条指令。

②语法格式：FROM <镜像：tag>。

③指令范例：FROM alpine:3.16.2。

(2)LABEL 指令。

①指令作用：以键值对形式指定镜像元数据，如镜像作者、描述信息等。

②语法格式：LABEL <键>=<值> <键>=<值> <键>=<值>……

③指令范例：LABEL maintainer="admin@testproject.org"

LABEL version="1.0"

(3)RUN 指令。

①指令作用：用于在 Docker 镜像构建过程中执行特定的命令，这些命令通常是 shell 命令。由于每次执行 RUN 指令时，Docker 都会创建一个新的镜像层来保存命令的执行结果，因此为了优化镜像的大小和层数，建议尽可能将多个相关的 RUN 指令合并成一条来执行，如指令范例中第二种写法。

②语法格式：RUN <shell 命名>。

RUN ["可执行文件", "参数 1", "参数 2"]

③指令范例：RUN apt install -y wget。

```
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz" \\
        && tar -xvf redis.tar.gz
RUN ["/bin/bash", "-c", "echo hello world"]
```

(4)COPY 指令。

①指令作用：用于将复制本地主机上的源文件(为 Dockerfile 所在目录的相对路径)复制到 Docker 镜像中。源文件支持使用通配符来表示多个文件或目录，如果源文件指定为一个目录，则 COPY 指令会复制该目录中的内容到镜像中。如果容器内的目标路径不存在，Docker 会自动创建路径。此外，可以通过--chown 设置复制后文件的所有者和所属组。

②语法格式：COPY [--chown=<user>:<group>] <源路径>... <目标路径>。

COPY [--chown=<user>:<group>] ["<src1>",... "<目标路径>"]

③指令范例：COPY test*　/mydir/。

COPY test.txt　/mydir/

(5)ADD 指令。

①指令作用：ADD 指令与 COPY 指令的作用相似，功能也相似，不同之处在于：如果源文件是 bzip2、gzip 等压缩格式，则会自动解压缩，同时 ADD 指令支持通过 URL 将网络资源添加到镜像中。

②语法格式：ADD [--chown=<user>:<group>] <源路径>... <目的路径>。

ADD [--chown=<user>:<group>] ["<源路径>",... "<目标路径>"]

③指令范例：ADD testfile　/mydir/。

ADD --chown=root:root　/mydir/

下面使用以上几个指令编写第一个 Dockerfile 文件来制作一个 nginx 镜像。首先拉取一个 ubuntu 基础镜像，然后创建需要复制到镜像中的文件。

```
#拉取 ubuntu 镜像
ubuntu@docker:~$ sudo docker pull ubuntu
ubuntu@docker:~$ sudo docker tag ubuntu ubuntu:v1
#创建工作目录
ubuntu@docker:~$ mkdir Dockerfile
ubuntu@docker:~$ cd Dockerfile/
#创建用于 COPY 指令的 index.html 文件和用于 ADD 指令的 addtest.tar.gz 压缩文件
ubuntu@docker:~/Dockerfile$ echo "Nginx build by dockerfile" >> index.html
ubuntu@docker:~/Dockerfile$ touch addtest1 addtest2
ubuntu@docker:~/Dockerfile$ mkdir addtest
ubuntu@docker:~/Dockerfile$ mv addtest1 addtest2 addtest
ubuntu@docker:~/Dockerfile$ tar -czvf addtest.tar.gz addtest
ubuntu@docker:~/Dockerfile$ vi sources.list          #准备镜像中需要的 apt 安装源配置文件
#添加如下内容，可以到清华镜像站 https://mirrors.tuna.tsinghua.edu.cn/help/ubuntu/网站复制
# 默认注释了源码镜像以提高 apt update 的速度，如有需要可自行取消注释
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal main restricted universe multiverse
# deb-src http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal main restricted universe multiverse
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-updates main restricted universe multiverse
# deb-src http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-updates main restricted universe multiverse
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-backports main restricted universe multiverse
# deb-src http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-backports main restricted universe multiverse
deb http://security.ubuntu.com/ubuntu/ focal-security main restricted universe multiverse
# deb-src http://security.ubuntu.com/ubuntu/ focal-security main restricted universe multiverse
# 预发布软件源，不建议启用
# deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-proposed main restricted universe multiverse
# # deb-src http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-proposed main restricted universe multiverse
ubuntu@docker:~/Dockerfile$ ls
addtest  addtest.tar.gz  index.html  sources.list
```

编写 Dockerfile 文件，内容如下。

```
ubuntu@docker:~/Dockerfile$ vi Dockerfile
#Dockerfile 中以#号为首的行为注释行
#添加以下指令
FROM ubuntu:v1
LABEL maintainer="testuser<root@testuser.com>"
RUN mv /etc/apt/sources.list /etc/apt/sources.list.bak
COPY ["sources.list","/etc/apt/"]
RUN apt-get update
RUN apt-get install -y nginx
COPY ["index.html","/var/www/html"]
ADD addtest.tar.gz /root/
```

然后通过 docker build 命令构建镜像，该命令的格式如下。

```
docker build [OPTIONS] PATH | URL | -
```

其中 OPTIONS 选项主要如下。

- -f:指定要使用的 Dockerfile 路径,默认当前目录下的 Dockerfile 文件。
- --tag, -t:制定镜像的名字及标签。
- --no-cache:创建镜像的过程不使用缓存。

```
#docker build 构建镜像,其中“.”表示当前目录,也即 Dockerfile 的上下文,即在当前目录中找构建镜像需要用到的资源
ubuntu@docker:~/Dockerfile$ sudo docker build -t mynginx:c7 .
Sending build context to Docker daemon  5.632kB
Successfully built 2747484fac17
Successfully tagged nginx:c7
#查看镜像,发现已被创建
ubuntu@docker:~/Dockerfile$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mynginx             c7                  f1be8ac4daf5        2 minutes ago       642MB
```

通过构建的镜像运行容器,查看 COPY 指令和 ADD 指令的效果。

```
ubuntu@docker:~/Dockerfile$ sudo docker run -it mynginx:c7
root@0bb2c7e03f4f:/# cd /root
root@0bb2c7e03f4f:~#ls
addtest                                          #ADD 添加的文件已经自动解压缩
root@0bb2c7e03f4f:~# cd /var/www/html/
root@0bb2c7e03f4f:/var/www/html# cat index.html
Nginx build by dockerfile
```

通过以上在容器中执行命令的返回结果,可以看到 ADD 指令不仅将压缩文件复制到容器中,还自动进行了解压缩。而 COPY 指令则是将 index.html 文件直接复制到镜像的指定位置。然而,如果直接使用 docker run -d 命令来通过该镜像运行容器,可能会因为没有长期运行的前台进程而导致容器立即退出。为了解决这个问题,需要在 Dockerfile 中继续完善,以确保容器能够持续运行。

(6)CMD 指令。

①指令作用:是定义容器启动时默认执行的命令。具体来说,如果在执行 docker run 时没有指定任何执行命令,且 Dockerfile 中也没有 ENTRYPOINT 指令,那么 Docker 会执行 CMD 指令中指定的默认命令。值得注意的是,每个 Dockerfile 中只能有一条 CMD 命令生效,如果有多条 CMD 指令被指定,只有最后一条会被执行。此外,如果用户在启动容器时通过 docker run 命令明确指定了其他运行命令,如 docker run -it alpine /bin/bash,则这个指定的命令会覆盖 CMD 指令中原有的默认命令。

②语法格式:CMD <shell 命令>。

```
CMD ["<可执行文件或命令>","<参数 1>","<参数 2>",...]
CMD ["<参数 1>","<参数 2>",...]          # 为 ENTRYPOINT 指令提供默认参数
```

③指令范例:CMD ping www.baidu.com

```
CMD ["nginx", "-g", "daemon off;"]
```

(7)ENTRYPOINT 指令。

①指令作用:类似于 CMD 指令,但它定义了一个容器启动时默认执行的程序,这个程序不会被 docker run 的命令行参数中直接指定的指令所直接覆盖。相反,docker run 命令行中的参数会被传递给由 ENTRYPOINT 指令指定的程序,作为该程序的启动参数。然而,如果在使用 docker run 时,明确指定了--entrypoint 选项,那么将覆盖 Dockerfile 中 ENTRYPOINT 指令所指定的程序。此外,需要注

意的是，如果 Dockerfile 中存在多个 ENTRYPOINT 指令，只有最后一个会生效。另外 CMD 指令的第三种用法（即作为 ENTRYPOINT 的默认参数）允许用户向 ENTRYPOINT 指令指定的程序传递额外的参数。

②语法格式：ENTRYPOINT ＜shell 命令＞。

ENTRYPOINT ["＜可执行文件或命令＞", "＜参数 1＞","＜参数 2＞",...]

③指令范例：ENTRYPOINT ["curl", "-s","https://www.baidu.com"]。

在原 Dockerfile 的基础上，通过 CMD 和 ENTRYPOINT 指令增加容器运行后执行的命令。

```
ubuntu@docker:~/Dockerfile$ vi Dockerfile
FROM ubuntu:v1
LABEL maintainer="testuser<root@testuser.com>"
RUN mv /etc/apt/sources.list /etc/apt/sources.list.bak
COPY ["sources.list","/etc/apt/"]
RUN apt-get update
RUN apt-get install -y nginx
COPY ["index.html","/var/www/html"]
ADD addtest.tar.gz /root/
ENTRYPOINT ["nginx","-g"]
CMD ["daemon off;"]
```

删除原镜像后，通过改写后的 Dockerfile 再次构建镜像并运行。

```
ubuntu@docker:~/Dockerfile$ sudo docker ps -a | grep mynginx:c7
0bb2c7e03f4f   mynginx:c7   "bash"   3 minutes ago   Exited (0) About a minute ago   elated_ardinghelli
ubuntu@docker:~/Dockerfile$  sudo docker rm -f 0b
0b
ubuntu@docker:~/Dockerfile$ sudo docker rmi mynginx:c7
ubuntu@docker:~/Dockerfile$  sudo docker build -t mynginx:c7 .
ubuntu@docker:~/Dockerfile$ sudo docker run -d mynginx:c7
4c790d172a7a780d767723a6efdf6c9d8de2fa1877be63ef672d05bdf4dbbf44
ubuntu@docker:~/Dockerfile$ sudo docker ps --format "table{{.ID}}\\t{{.Names}}\\t{{.Status}}\\t{{.
Command}}"
CONTAINER ID   NAMES              STATUS              COMMAND
4c790d172a7a   vigorous_dewdney   Up 2 minutes        "nginx -g 'daemon of…"
```

再次运行容器后，可以看到容器可以在主机的后台持续运行。在上面的 Dockerfile 中，通过 CMD 指令向 ENTRYPOINT 指令传递参数“daemon off;”，这两条指令的作用相当于运行容器后执行命令 nginx -g "daemon off;"，这样做的目的是让 nginx 作为前台进程运行，从而保持容器的活动状态。

小提示：如果有多个 CMD 指令，只有最后一个会生效；如果有多个 ENTRYPOINT 指令，只有最后一个会生效；如果 CMD 和 ENTRYPOINT 共存，只有 ENTRYPOINT 有效，且最后的 CMD 会作为 ENTRYPOINT 的参数。

(8)ENV 指令。

①指令作用：用于设置环境变量。一旦定义了环境变量，这些变量在后续的指令中就可以被引用和使用了。此外，这些环境变量在 docker build 镜像构建阶段和 docker run 容器运行阶段都有效。

②语法格式：ENV ＜键＞ ＜值＞。

ENV ＜键＞=＜值＞ ＜键＞=＜值＞...

③指令范例：ENV Version 5.0.3。

ENV website=www.baidu.com

(9)ARG 指令。

①指令作用:与 ENV 在功能上有所不同,尽管它们都用于设置变量,但它们的作用域和用途存在显著差异。ARG 指令用于在 Dockerfile 中定义构建时变量,这些变量仅在 Docker 镜像的构建阶段(docker build)内有效,并且不会被包含在最终构建好的镜像中。换句话说,ARG 设置的环境变量仅对 Dockerfile 内的指令有效,不会影响到镜像运行时的环境。在构建 Docker 镜像时,可以通过在 docker build 命令中使用--build-arg <参数名>=<值>来覆盖 ARG 指令中定义的默认值。

②语法格式:ARG <参数名>[=<默认值>]。

③指令范例:ARG tools。

ARG tools=net-tools

在上面的 Dockerfile 中增加 ARG 和 ENV 指令,步骤如下。

```
ubuntu@docker:~/Dockerfile$ vi Dockerfile
ARG tag=v1                                    #通过 ARG 指令设置 tag 变量
FROM ubuntu:$tag                              #使用 ARG 指令定义的变量 tag
LABEL maintainer="testuser<root@testuser.com>"
ENV tools=net-tools                           #通过 ENV 指令设置 tools 变量
ARG dest                                      #通过 ARG 指令设置未赋值变量 dest
RUN mv /etc/apt/sources.list /etc/apt/sources.list.bak
COPY ["sources.list","/etc/apt/"]
RUN apt-get update
RUN apt-get install -y nginx && apt-get install -y $tools     #使用 ENV 指令定义的 tools 变量
COPY ["index.html","/var/www/html/"]
ADD addtest.tar.gz $dest                      #使用 ARG 指令定义的 dest 变量
ENTRYPOINT ["nginx","-g"]
CMD ["daemon off;"]
```

然后删除镜像,重新根据 Dockerfile 构建镜像,步骤如下。

```
ubuntu@docker:~/Dockerfile$ sudo docker rm -f 4c
4c
ubuntu@docker:~/Dockerfile$ sudo docker rmi mynginx:c7
Untagged: mynginx:c7
#使用--build-arg 选项在 docker build 阶段指定 Dockerfile 中设置的 dest 的值为/root。
ubuntu@docker:~/Dockerfile$ sudo docker build --build-arg=dest=/root -t mynginx:c7 .
ubuntu@docker:~/Dockerfile$ sudo docker images
REPOSITORY      TAG       IMAGE ID        CREATED              SIZE
mynginx         c7        087fde06a8cc    About a minute ago   181MB
```

以上 docker build 构建镜像时,通过--build-arg=dest=/root 来指定 Dockerfile 中未定义变量 dest 的值。通过新的镜像运行容器。

```
ubuntu@docker:~/Dockerfile$ sudo docker run -d mynginx:c7
558a5137cfbcdc014c1a0fe88504e8fe3adaffa8668de020bd5c4a0cb86ee6c2
ubuntu@docker:~/Dockerfile$ sudo docker exec -it 55 /bin/bash
root@558a5137cfbc:/#cd /root/
#查看到当前目录中有 addtest 解压缩后的目录,说明 dest 变量生效
root@558a5137cfbc:~#ls
addtest
#容器中有 netstat 命令,说明 tools 变量生效
root@558a5137cfbc:~#netstat -ntpl
```

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1/nginx: master pro
tcp6       0      0 :::80                   :::*                    LISTEN      1/nginx: master pro
#输出 dest 变量为空,说明 ARG 指令定义的变量容器中不存在
root@558a5137cfbc:~# echo $dest
#输出 tools 变量仍然有效,说明 ENV 指令定义的变量容器中有效
root@558a5137cfbc:~# echo $tools
net-tools
```

(10)VOLUME 指令。

①指令作用:用于定义匿名数据卷。当启动容器时,如果忘记显式挂载数据卷,Docker 会自动为容器创建并挂载匿名卷。这些匿名卷通常会被存储在宿主机中的/var/lib/docker/volumes/<id>/_data 目录下。此外,通过 VOLUME 指令指定的容器目录会与这些匿名卷或显式指定的数据卷进行挂载。

②语法格式:VOLUME ["<路径 1>", "<路径 2>"...]。

VOLUME <路径>

③指令范例:VOLUME ["/data","/data2" ,…]。

(11)EXPOSE 指令。

①指令作用:用于告知 Docker 服务端容器内部将要暴露的端口号。需要注意的是,EXPOSE 仅仅声明了容器打算监听哪些端口,而并不会自动在宿主机上进行端口映射。因此,在启动容器时,如果想让宿主机上的端口能够访问到容器内部的服务,则需要通过-P 或-p 选项来指定端口映射关系。

②语法格式:EXPOSE <端口 1> [<端口 2>...]。

③指令范例:EXPOSE 80 443。

小提示:并不是只有在使用了 EXPOSE 指令声明了端口号之后,才能进行端口映射。实际上,该指令主要作用是声明容器内部的服务将通过哪个端口号对外提供服务。

(12)WORKDIR 指令。

①指令作用:用于为后续的 RUN、CMD、ENTRYPOINT 等指令配置工作目录。这意味着,当容器运行时,进入容器内的默认目录将是 WORKDIR 指定的工作目录。WORKDIR 指令在 Dockerfile 中设置的工作目录,会应用于构建镜像过程中的每一层,而不仅仅是最终层(WORKDIR 指定的工作目录,必须是提前创建好的)。在 docker build 构建镜像的过程中,每一个 RUN 命令通常都会基于前一层创建一个新的层,但由 WORKDIR 创建的目录会一直存在。

②语法格式:WORKDIR <工作目录路径>。

③指令范例:WORKDIR /root。

(13)USER 指令。

①指令作用:用于指定运行容器时所使用的用户名或 UID。后续的 RUN 指令也会使用指定用户,当服务不需要管理员权限时,可以通过 USER 指令指定运行用户,这个用户必须是事先建立好的,否则无法切换。

②语法格式:USER <user>[:<group>]。

USER <UID>[:<GID>]。

③指令范例:USER nginx:nginx

(14)ONBUILD 指令。

①指令作用:用于在基于当前镜像构建子镜像时自动触发执行预设的指令。具体来说,当在 Dockerfile 中使用 ONBUILD 指定命令时,这些命令在当前镜像的构建过程中不会执行。当有新的 Dockerfile 使用 FROM image:latest(假设 image:latest 为当前镜像的标签)来构建新镜像时,image:

latest Dockerfile 中所有的 ONBUILD 指令将自动执行。

②语法格式：ONBUILD ＜其他指令＞。

③指令范例：ONBUILD RUN yum install -y nginx。

使用以上指令继续完善构建 mynginx:c7 镜像的 Dockerfile，修改 Dockerfile 内容如下。

```
[root@docker Dockerfile]# vi Dockerfile
ARG tag=v1
FROM ubuntu: $ tag
LABEL maintainer="testuser＜root@testuser.com＞"
ENV tools=net-tools
ARG dest
RUN mv /etc/apt/sources.list /etc/apt/sources.list.bak
COPY ["sources.list","/etc/apt/"]
RUN apt-get update
RUN apt-get install -y nginx && apt-get install -y  $ tools
COPY ["index.html","/var/www/html"]
ADD addtest.tar.gz  $ dest
ENTRYPOINT ["nginx","-g"]
CMD ["daemon off;"]
VOLUME /var/www/html
EXPOSE 80
WORKDIR /var/www/html
```

使用重新修改后的 Dockerfile 构建镜像并运行。

```
ubuntu@docker:～/Dockerfile $ sudo docker rmi mynginx:c7
ubuntu@docker:～/Dockerfile $ sudo docker build --build-arg=dest=/root -t mynginx:c7 .
ubuntu@docker:～/Dockerfile $ sudo docker run -d -P mynginx:c7
ubuntu@docker:～/Dockerfile $ sudo docker ps --format "table{{.ID}}\\t{{.Image}}\\t{{.Status}}\\t{{.
Ports}}"
CONTAINER ID    IMAGE          STATUS          PORTS
867661ec6c2b    mynginx:c7     Up 2 minutes    0.0.0.0:49153->80/tcp, :::49153->80/tcp
```

通过访问 Docker 主机的“49153”端口访问容器，如图 6-16 所示。

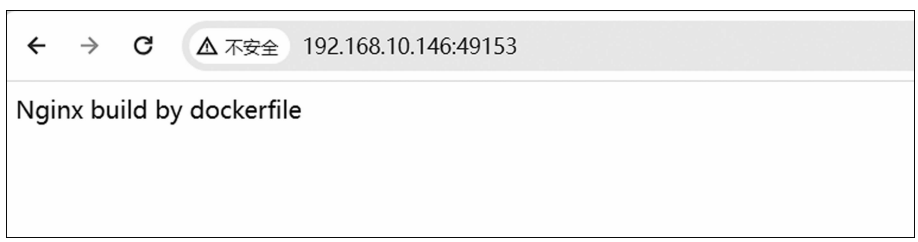


图 6-16　访问 mynginx:c7 镜像运行的容器

进入容器内部后，可以发现当前工作目录已经切换到通过 WORKDIR 指令在 Dockerfile 中指定的目录。此外，尽管在启动容器时没有通过-v 选项指定数据卷映射，但由于 Dockerfile 中包含了 VOLUME 指令，Docker 会在容器启动时自动为这些指定的路径创建匿名数据卷，并将它们挂载到容器内的相应路径上。

```
ubuntu@docker:~/Dockerfile$ sudo docker exec -it 86 /bin/bash
root@89830afa2e45:/var/www/html# pwd
/var/www/html
ubuntu@docker:~/Dockerfile$ sudo docker inspect 86
"Mounts": [
        {
            "Type": "volume",
            "Name": "7528bf06e2bde1413675cbb28d88dbc6c6e97f31cfc42b4c740fd57f06c9c1c9",
            "Source": "/var/lib/docker/volumes/7528bf06e2bde1413675cbb28d88dbc6c6e9
7f31cfc42b4c740fd57f06c9c1c9/_data",
            "Destination": "/var/www/html",
            "Driver": "local",
            "Mode": "",
            "RW": true,
            "Propagation": ""
        }
    ],
```

6.3.4 任务拓展

1. 任务内容

探究 Docker Overlay2 的存储结构。

2. 任务目标

掌握 Docker 的存储分层结构原理，掌握 Overlay2 的存储驱动原理。

3. 任务步骤

理解 Docker 的存储方式，有利于理解 Docker 镜像和容器的分层结构，也有利于更深入地理解 Dockerfile 制作的镜像过程。

(1)为了更清晰地查看 Docker 的 Overlay2 存储方式，在 Docker 主机上只保留 nginx:latest 镜像，以及通过该镜像运行的容器，同时删除所有其他不必要的镜像和容器。

```
ubuntu@docker:~$ sudo docker images
REPOSITORY      TAG         IMAGE ID        CREATED       SIZE
nginx           latest      2d389e545974    9 days ago    142MB
ubuntu@docker:~$ sudo docker ps -qa
498109119a00
```

(2)通过以下命令查看 nginx 镜像和该镜像运行容器的存储路径。

```
#查看镜像详细信息，找到 GraphDriver 属性中的以下内容
ubuntu@docker:~$ sudo docker image inspect nginx
"LowerDir": "
/var/lib/docker/overlay2/934be05f9b6b2belcf6f2fc0b7e48f5feaaa69d439f29dd2c0994a1eab74c1b8/diff:
/var/lib/docker/overlay2/d0c067c417f740d1c1bb83a962d1c6242a729a6c1086993bbff431efcbc71256/diff:
/var/lib/docker/overlay2/aeb00ba5ccabb2eee71731580686cd0bd4dc20feb987012458dc3511271a705b/diff:
/var/lib/docker/overlay2/0a21f586bb69ee7f50d5108f4b969afe295fcb05ba52595d1b8974302de4bd3e/diff:
/var/lib/docker/overlay2/3a621df640d9b6bd13fcf91b986228f3c72aad3c756c3dfc3b99001e3d9b84cd/diff",
"MergeDir": "
```