Docker 万字教程：从入门到掌握

1. Docker 介绍

1.1 Docker 简介

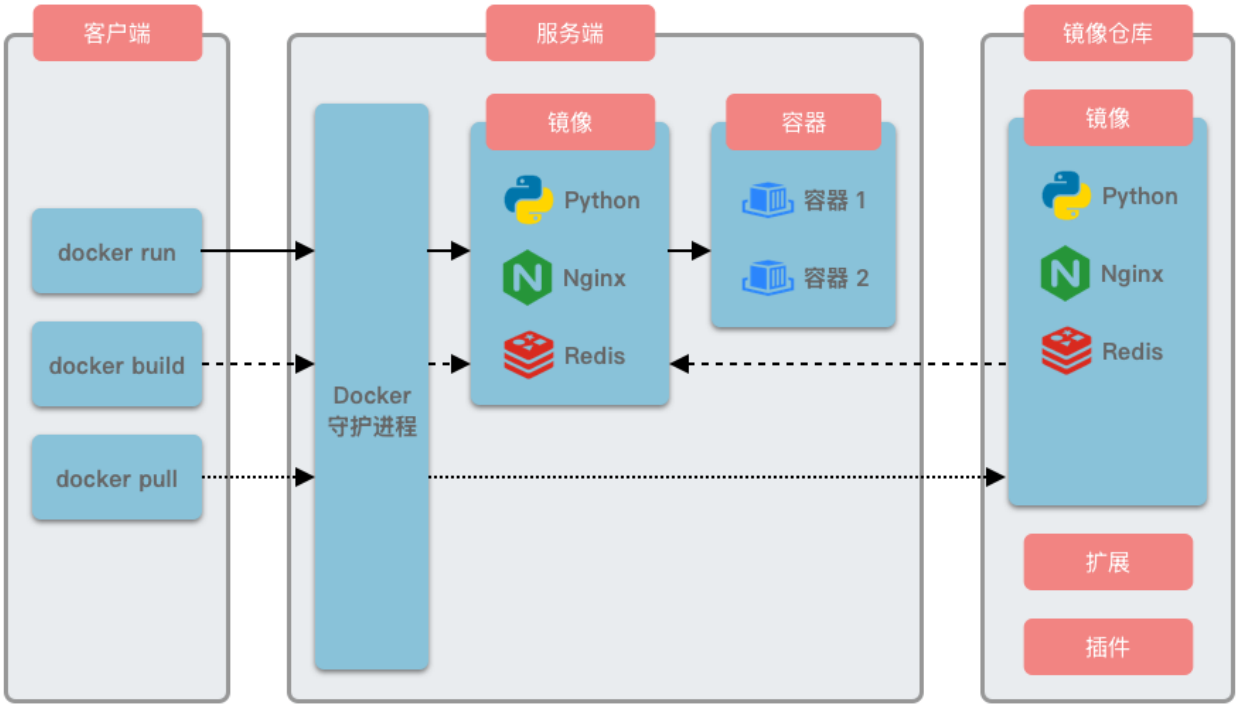
> **Docker**：Docker 是一个开源的应用容器引擎，让开发者可以将应用程序和所有依赖打包到一个轻量级、可移植的容器中，然后在任何支持 Docker 的环境中运行。

在传统的项目开发中，开发者经常遇到环境不一致的问题，比如代码在本地开发环境运行正常，但在测试或生产环境却出现各种错误，原因可能是操作系统版本、依赖库版本或配置差异。此外，传统部署方式需要手动安装和配置各种软件环境，过程繁琐且容易出错，不同服务器之间的环境也难以保持一致。

Docker 的核心目标就是解决这些问题，通过容器化技术将应用及其运行环境打包在一起，确保应用在不同环境中表现一致。Docker 的出现极大简化了开发、测试和部署的流程，成为现代 DevOps 和云计算中的重要工具。Docker 有几个显著特点：

1. **轻量性**：由于容器共享宿主机的操作系统内核，它们比传统虚拟机更小且启动更快，解决了传统虚拟化技术资源占用高、启动慢的问题。
2. **可移植性**：Docker 容器可以在任何支持 Docker 的平台上运行，无论是本地开发机、物理服务器还是云环境，彻底解决了「在我机器上能跑，线上却不行」的难题。
3. **隔离性**：每个容器拥有独立的文件系统、网络和进程空间，确保应用之间互不干扰，避免了传统部署中多个应用共用环境导致的依赖冲突问题。
4. **标准化**：Docker 提供统一的接口和工具链，使得构建、分发和运行容器变得简单高效，替代了传统部署中复杂的手动配置流程。

Docker 采用了客户端-服务器架构。Docker 客户端负责发送命令，Docker 守护进程（服务端）负责执行这些命令。它们可以运行在同一台机器上，也可以分开运行。客户端和守护进程通过 UNIX 套接字或网络接口进行通信，使用 REST API 交换数据。这种架构让用户可以通过本地或远程客户端管理 Docker 服务。Docker 的架构图如下所示：



1.2 Docker 核心概念

Docker 的核心概念包括容器、镜像、Dockerfile 和镜像仓库，这些是理解 Docker 技术的基础。

> **容器（Container）**：一种轻量级的虚拟化技术，它共享操作系统内核但保持独立的运行环境，这使得容器比传统虚拟机更快速且占用资源更少。

容器是 Docker 技术的核心运行单元，它是一种轻量级的虚拟化技术实现方式。与传统的虚拟机不同，容器不需要模拟完整的硬件环境，也不需要运行独立的操作系统内核。容器在运行时与其他容器和宿主机共享操作系统内核，这使得容器启动更快且占用资源更少。

容器之间是相互独立的。每个容器都拥有自己的文件系统、网络和进程空间，确保应用之间不会互相干扰。

> **镜像（Image）**：镜像是用于创建容器的模板，它包含了运行应用所需的代码、库和配置文件，用户可以从 Docker Hub 下载现成镜像或自己构建。

镜像是用来创建 Docker 容器的基础，镜像中包含了运行应用所需的代码、库、环境变量和配置文件，用户可以直接使用现成的镜像，比如从 Docker Hub 下载，也可以基于现有镜像定制自己的镜像。镜像采用分层存储结构，每一层代表一个修改步骤，这种设计使得镜像的构建和分发更高效。镜像中不包含任何的动态数据，其内容在构建之后不再变动。

> **Dockerfile**：Dockerfile 是一个文本文件，里面写明了如何一步步构建镜像，通过执行 Dockerfile 中的指令，Docker 能自动生成镜像。

Dockerfile 是用于定义镜像构建过程的脚本文件，它由一系列指令组成，比如 `FROM` 指定基础镜像，`RUN` 执行命令，`COPY` 复制文件等，Docker 会根据 Dockerfile 的指令自动构建镜像，这使得镜像的创建过程可重复且透明。

> **镜像仓库（Image Repository）**：用于集中存储和分发镜像的地方。

最常用的公共仓库是 Docker Hub，它提供了大量官方和社区维护的镜像。此外，我们也可以搭建公司内部 / 个人使用的私有镜像仓库，用于存放自己的镜像。当需要在另一台主机上使用该镜像时，只需要从仓库上下载即可。

容器、镜像和 Dockerfile、镜像仓库共同构成了 Docker 的工作流程：

1. 开发者编写 Dockerfile 定义环境，构建 Docker 镜像。
2. 将构建好的镜像推送到镜像仓库（公共仓库 / 私有仓库）。
3. 在任何支持 Docker 的机器上拉取镜像并运行容器。

整个过程标准化且高效。理解这些核心概念是掌握 Docker 的关键，它们解决了传统开发部署中的环境不一致问题，使应用在任何地方都能以相同的方式运行。

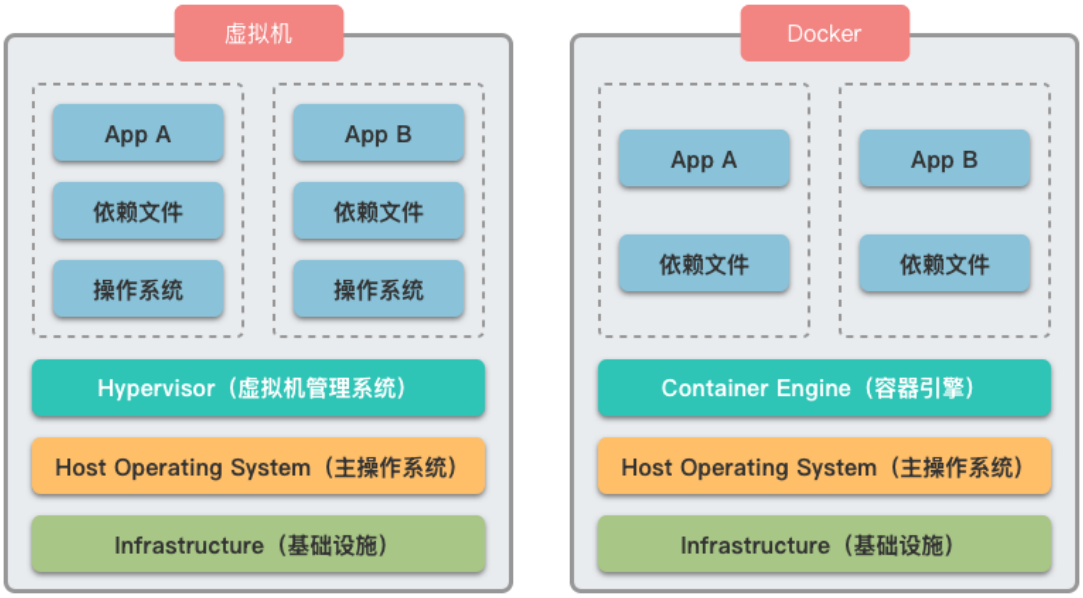
1.3 Docker 和虚拟机

Docker 和虚拟机都是用来隔离应用运行环境的技术，但它们的工作原理和资源占用方式完全不同。Docker 容器不需要模拟整个操作系统，而是直接运行在宿主机的内核上，这使得容器启动更快、占用资源更少，同时仍能提供良好的隔离性。

Docker 容器和虚拟机的具体区别如下：

- **Docker 容器**：使用 Docker 引擎进行调度和隔离，直接共享宿主机的操作系统内核，不需要额外的操作系统层，这使得容器启动更快、占用资源更少。
- **虚拟机**：通过 Hypervisor 软件模拟完整的硬件环境，管理每个虚拟机中操作系统的运行。每个虚拟机需要运行独立的操作系统、应用程序和必要的依赖文件等，因此启动速度慢且占用大量内存和存储。

虚拟机的隔离性更强，因为每个虚拟机有完全独立的操作系统，适合运行需要强隔离的不同类型应用。Docker 的隔离性主要依靠 Linux 内核的命名空间和控制组功能，虽然隔离程度不如虚拟机，但对于大多数应用场景已经足够。Docker 和虚拟机对比如图所示：



2. Docker 安装

2.1 CentOS Docker 安装

1. 清理旧版本 Docker

安装前需要清理系统中可能存在的旧版本 Docker 或其他冲突软件包。

```
$ sudo dnf remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-engine
```

2. 安装 yum 工具包

```
$ sudo yum install -y yum-utils
```

3. 配置 yum 软件源

```
# yum 国内源
$ sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
$ sudo sed -i 's/download.docker.com/mirrors.aliyun.com\/docker-ce/g' /etc/yum.repos.d/docker-ce.repo

# yum 官方源
# $ sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
```

4. 安装最新版 Docker

安装 Docker 需要多个组件，包括 Docker 引擎（docker-ce）、命令行工具（docker-ce-cli）、容器运行时（containerd.io）、构建镜像的插件工具（docker-buildx-plugin）、多容器应用的编排管理工具（docker-compose-plugin）等。

```
$ sudo yum install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

安装完成后可以通过 `docker --version` 命令检查版本信息确认安装是否成功。

```
$ docker --version
Docker version 28.0.4, build b8034c0
```

5. 启动 Docker

安装成功后需要启动 Docker 服务，运行下面的命令可以启动 Docker 守护进程。

```
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

> **注意**：如果安装过程中出现问题，可以尝试清理 yum 缓存，使用以下命令清除缓存后再重新安装。
>
> ```
> $ yum clean packages
> ```

2.2 macOS Docker 安装

在 macOS 系统上安装 Docker 需要下载并安装「Docker Desktop for Mac」 应用程序。Docker Desktop 是官方提供的图形化工具，它包含了 Docker 引擎、命令行工具和图形界面，让用户能够方便地在 macOS 上使用 Docker。

目前 macOS 上安装 Docker Desktop 有两种方法：「使用 Homebrew 安装」和「手动下载安装」。

2.2.1 使用 Homebrew 安装

第一种方法是使用 Homebrew 工具安装，打开终端并运行命令 `brew install --cask docker`，这个命令会自动下载并安装最新版的 Docker Desktop。

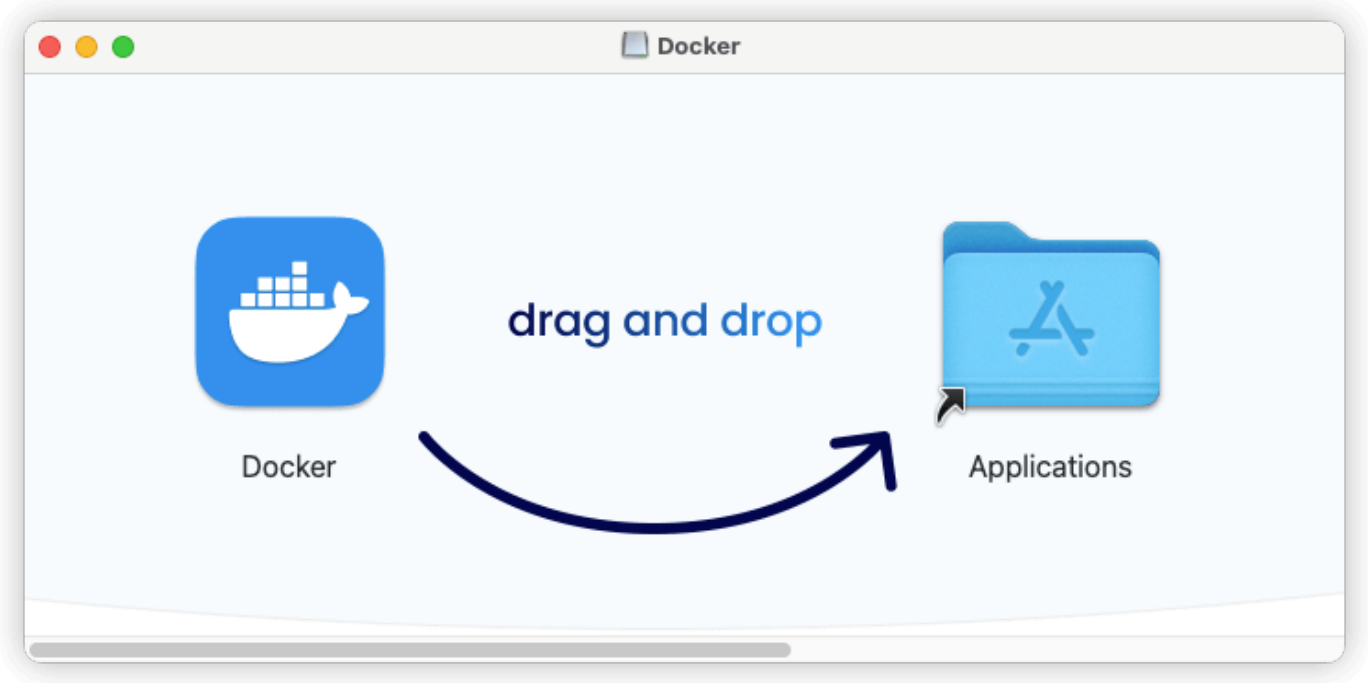
```
$ brew install --cask docker
==> Downloading https://formulae.brew.sh/api/cask_tap_migrations.jws.json
==> Downloading https://formulae.brew.sh/api/formula_tap_migrations.jws.json
==> Downloading https://raw.githubusercontent.com/Homebrew/homebrew-cask/b946820
######################################################################## 100.0%
==> Downloading https://desktop.docker.com/mac/main/arm64/187762/Docker.dmg
######################################################################## 100.0%
==> Installing Cask docker
==> Moving App 'Docker.app' to '/Applications/Docker.app'
==> Linking Binary 'docker-compose.zsh-completion' to '/opt/homebrew/share/zsh/s
==> Linking Binary 'docker-compose.fish-completion' to '/opt/homebrew/share/fish
==> Linking Binary 'compose-bridge' to '/usr/local/bin/compose-bridge'
==> Linking Binary 'docker' to '/usr/local/bin/docker'
==> Linking Binary 'docker-credential-desktop' to '/usr/local/bin/docker-credent
==> Linking Binary 'docker-credential-ecr-login' to '/usr/local/bin/docker-crede
```

```
==> Linking Binary 'docker-credential-osxkeychain' to
'/usr/local/bin/docker-cre
==> Linking Binary 'kubectl' to '/usr/local/bin/kubectl.docker'
==> Linking Binary 'docker-compose' to '/usr/local/cli-
plugins/docker-compose'
Password: # 如果需要输入密码，则输入电脑密码
==> Linking Binary 'docker.bash-completion' to
'/opt/homebrew/etc/bash_completio
==> Linking Binary 'docker.zsh-completion' to
'/opt/homebrew/share/zsh/site-func
==> Linking Binary 'docker.fish-completion'  to
'/opt/homebrew/share/fish/vendor_
==> Linking Binary 'hub-tool' to '/usr/local/bin/hub-tool'
==> Linking Binary 'docker-compose.bash-completion' to
'/opt/homebrew/etc/bash_c
🍺  docker was successfully installed!
```

2.2.2 手动下载安装

第二种方法是手动下载安装包。具体步骤如下：

1. 访问 Docker 官网的 下载页面，选择与你的 Mac 芯片类型匹配的版本下载。
2. 下载完成后双击 .dmg 文件，将 Docker 图标拖到 Applications 文件夹中完成安装。
3. 安装完成后，在应用程序文件夹中找到 Docker 并双击启动，首次启动时会提示你授予权限，需要输入系统密码确认。
4. 启动 Docker Desktop 后，你会在屏幕顶部菜单栏看到 Docker 的鲸鱼图标，这表示 Docker 正在运行。
5. 打开终端输入 `docker --version` 可以检查 Docker 是否安装成功，如果显示版本号说明安装正确。



Docker Desktop 默认会在开机时自动启动，你也可以在设置中调整这个选项。安装完成后，你可以直接使用 docker 命令在终端中操作容器，或者通过 Docker Desktop 的图形界面管理容器和镜像。需要注意的是，由于 macOS 的特殊性，Docker 在文件共享和网络配置方面与 Linux 系统有些差异，具体使用时可能需要调整相关设置。

2.3 Windows Docker 安装

在 Windows 系统上安装 Docker 需要下载并安装 Docker Desktop 应用程序。Docker Desktop 是官方提供的工具，它包含 Docker 引擎、命令行工具和图形界面，让用户能够在 Windows 上方便地使用 Docker。安装前需要确认系统版本，Windows 10 或 11 的 64 位专业版、企业版或教育版才能运行 Docker Desktop，家庭版需要通过额外步骤启用 Hyper-V 功能。具体安装步骤如下：

1. 打开 Docker 官网的 下载页面，点击下载 Windows 版安装包。
2. 下载完成后双击运行安装程序，安装过程中会提示启用 Windows 的 WSL 2 功能和 Hyper-V 虚拟化技术，勾选这些选项并继续安装。
3. 安装完成后在开始菜单中找到 Docker Desktop 并启动，首次启动需要等待 Docker 初始化完成，系统托盘会出现 Docker 的鲸鱼图标表示运行正常。
4. 打开命令提示符或 PowerShell 输入 `docker --version` 可以检查安装是否成功，如果显示版本号说明安装正确。

Docker Desktop 默认会随系统启动，可以在设置中修改这个选项。使用 Docker 时需要保持 Docker Desktop 在后台运行，通过命令行或图形界面都能管理容器和镜像。Windows 版 Docker 在文件共享方面需要注意，默认只有 C 盘用户目录下的文件可以直接挂载到容器中，其他盘符需要在 Docker 设置中手动添加共享路径。网络配置方面，Windows 版 Docker 使用 NAT 模式为容器提供网络连接，端口映射规则与 Linux 版本一致。如果遇到安装或运行问题，可以尝试重启 Docker 服务或电脑，检查防火墙设置是否阻止了 Docker 的网络连接。

3. Docker 使用

3.1 配置镜像源地址

Docker 下载镜像默认从国外的官网下载，在国内需要通过代理访问 / 更换国内镜像源。下面介绍一下更换国内镜像源的方法。

创建镜像源配置文件：

```
$ sudo tee /etc/docker/daemon.json <<-'EOF'
{
    "registry-mirrors": [
     "https://docker.m.daocloud.io",
     "https://dockerproxy.com",
     "https://docker.mirrors.ustc.edu.cn",
     "https://docker.nju.edu.cn"
   ]
}
EOF
```

更改镜像源配置文件后，需要重新加载配置文件。

```
$ sudo systemctl daemon-reload
```

然后重启 Docker 服务。

```
$ sudo systemctl restart docker
```

3.2 服务使用

Docker 服务管理是使用 Docker 的基础操作。下面介绍常用的 Docker 服务命令。

1. 启动 Docker 服务

```
$ systemctl start docker
```

2. 停止 Docker 服务

```
$ systemctl stop docker
```

3. 重启 Docker 服务

```
$ systemctl restart docker
```

4. 设置开机启动 Docker 服务

```
$ systemctl enable docker
```

5. 查看 Docker 服务状态

```
$ systemctl status docker
```

这些命令用于控制 Docker 服务的运行状态。启动服务后才能使用 Docker 的其他功能。停止服务会关闭所有正在运行的容器。重启命令通常用于应用配置变更后重新加载服务。设置开机启动可以确保系统重启后 Docker 自动运行。查看状态命令可以检查服务是否正常运行。

3.3 镜像使用

Docker 镜像是创建容器的基础模板，理解镜像的使用方法是掌握 Docker 的关键。镜像包含了运行应用所需的代码、库、环境变量和配置文件，用户可以直接使用现成的镜像，也可以基于现有镜像定制自己的镜像。下面以 nginx 为例，详细介绍一下镜像的基本操作。

3.3.1 官方镜像仓库

Docker Hub 是 Docker 官方的公共镜像仓库，提供了大量官方和社区维护的镜像。使用 Docker Hub 需要注册账号。在命令行中可以通过 `docker login` 命令登录 Docker Hub 账号。

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: your_username
Password:
Login Succeeded
```

登录后可以拉取和推送镜像。

3.3.2 拉取镜像

获取镜像最常用的方法是从 Docker Hub 或其他镜像仓库下载。Docker Hub 是 Docker 官方的公共镜像仓库，提供了大量官方和社区维护的镜像。从 Docker 镜像仓库获取镜像的命令是 `docker pull`，对应命令格式为：

```
$ docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

- OPTIONS（可选）：选项参数。
  - `--all-tags, -a`：下载指定镜像的所有标签。
  - `--disable-content-trust`：跳过镜像签名验证。
- NAME：镜像名称，通常包含注册表地址（比如 `docker.io/library/ubuntu`），不带注册表地址则默认从 Docker Hub 进行拉取。
- TAG（可选）：镜像标签，默认为 `latest`。
- DIGEST（可选）：镜像的 SHA256 摘要。

下载最新的 nginx 镜像可以运行 `docker pull nginx` 命令，Docker 会从配置的镜像源查找并下载该镜像：

```
$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
16c9c4a8e9ee: Pull complete
de29066b274e: Pull complete
2cf157fc31fe: Pull complete
450968563955: Pull complete
9b14c47aa231: Pull complete
```

```
 9  fd8a9ced9846: Pull complete
10  c96c7b918bd5: Pull complete
11  Digest:
    sha256:5ed8fcc66f4ed123c1b2560ed708dc148755b6e4cbd8b943fab094f2
    c6bfa91e
12  Status: Downloaded newer image for nginx:latest
13  docker.io/library/nginx:latest
14
15  What's next:
16      View a summary of image vulnerabilities and recommendations
    → docker scout quickview nginx
```

如果想下载特定版本的镜像，可以在镜像名后加上标签，比如下载 1.23 版本的 nginx。

```
1  $ docker pull nginx:1.23
```

3.3.3 镜像标签

镜像标签是用来标识和管理镜像版本的重要工具。每个镜像可以有多个标签，通常用于区分不同版本或环境。使用 `docker tag` 命令可以为镜像创建新标签。

以 nginx 镜像为例，默认情况下当我们拉取nginx镜像时，Docker 会自动使用 latest 标签，这表示最新稳定版。在实际开发中，我们需要更精确的控制镜像版本。例如使用下面的命令会将本地的 nginx 镜像标记为 `my-nginx:v1`，这样就能在项目中明确使用特定版本的镜像。

```
1  $ docker tag nginx:latest my-nginx:v1
```

3.3.4 推送镜像

要将本地镜像推送到 Docker Hub，需要使用登录的 Docker Hub 用户给镜像打标签。

```
1  $ docker tag my-image your_username/my-image:1.0
```

然后使用 `docker push` 命令推送到 Docker Hub。

```
1  $ docker push your_username/my-image:1.0
```

3.2.5 查看镜像

1. 查看所有镜像列表

下载完成后，对应镜像会存储在本地。通过 `docker images` 命令可以查看本地已有的镜像列表，这个命令会显示镜像的名称、标签、镜像 ID、创建时间和大小等信息。

```
$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED      SIZE
nginx         latest    1f94323bafb2   6 days ago   198MB
```

2. 查看指定镜像详细信息

如果想要查看某个镜像（比如 nginx）的详细信息，包括创建时间、环境变量、工作目录、暴露的端口等配置信息，可以通过 `docker inspect` 命令查看，这些信息对于了解镜像的运行方式和配置非常重要。

```
$ docker inspect nginx
```

如果想要查看镜像的构建历史，则可以通过 `docker history` 相关命令查看，这个命令会显示镜像每一层的创建命令和大小，帮助理解镜像是如何一步步构建出来的。

```
$ docker history nginx
```

对于正在运行的容器，可以使用 `docker logs` 相关命令查看容器的日志输出，这个命令对于排查容器运行问题很有帮助。

```
$ docker logs nginx
```

3.3.6 查找镜像

镜像的查找可以通过 Docker Hub 网站或命令行完成，使用 `docker search` 命令能在终端直接搜索公共镜像。比如使用下面命令可以列出所有 MySQL 相关镜像，结果会显示镜像名称、描述、星标数和是否官方认证等信息。官方镜像由 Docker 官方团队维护，通常更可靠安全，建议优先选择带有 "OFFICIAL" 标记的镜像。

```
$ docker search mysql
NAME                    DESCRIPTION
        STARS     OFFICIAL
mysql                   MySQL is a widely used, open-source
relation…   15752     [OK]
bitnami/mysql           Bitnami container image for MySQL
        133
circleci/mysql          MySQL is a widely used, open-source
relation…   31
bitnamicharts/mysql     Bitnami Helm chart for MySQL
       0
cimg/mysql
        3
ubuntu/mysql            MySQL open source fast, stable, multi-
thread…   67
……
```

3.3.7 镜像的导出和导入

在开发过程中，经常需要将 Docker 镜像从一个环境迁移到另一个环境。比如开发完成后，需要将测试通过的镜像部署到生产服务器，但生产服务器可能无法直接访问互联网下载镜像。这时可以使用镜像导出和导入功能。导出镜像使用 `docker save` 命令，这个命令会把镜像及其所有层打包成一个 tar 文件。

```
$ docker save -o nginx.tar nginx:latest
```

以上命令执行之后，会讲最新的 nginx 镜像保存到当前目录下的 nginx.tar 文件中。这个文件可以复制到其他服务器上。

导出镜像可以使用 `docker load` 命令。在其他服务器上，通过 `docker load -i nginx.tar` 命令可以将 `nginx.tar` 所对应的 nginx 镜像导入到服务器的 Docker 中。

```
$ docker load -i nginx.tar
c9b18059ed42: Loading layer
[==================================================>]
 100.2MB/100.2MB
cbd8457b9f28: Loading layer
[==================================================>]
 101.6MB/101.6MB
c648e944b17e: Loading layer
[==================================================>]
 3.584kB/3.584kB
966bd022be40: Loading layer
[==================================================>]
 4.608kB/4.608kB
b422fd70039f: Loading layer
[==================================================>]
2.56kB/2.56kB
486cd1e5e3be: Loading layer
[==================================================>]
5.12kB/5.12kB
cbaa47f9fe15: Loading layer
[==================================================>]
 7.168kB/7.168kB
Loaded image: nginx:latest
```

导入后使用 `docker images` 命令可以看到 nginx 镜像已经存在。

3.3.8 删除镜像

1. 删除单个镜像

当需要删除不再使用的 Docker 镜像时，可以使用 `docker rmi` 命令。这个命令需要指定镜像 ID 或镜像名称。比如要删除 nginx，可以使用 `docker rmi nginx` 命令。

```
$ docker rmi nginx
Untagged: nginx:latest
Deleted: sha256:1f94323bafb2ac98d25b664b8c48b884a8db9db3d9c98921b3b8ade588b2e676
Deleted: sha256:ca37bdd8ff5f2cbccaea502aa62460940bd5a2500a9fce4e931964e05a5f2ece
Deleted: sha256:2775bcda3310ca958798e032824be2d7a383c61cc4415c9ffd906be40eeab511
Deleted: sha256:c52a77a0a626e1e81d52b4ee7479be288a7b5430103e8caf99ea71c690084a41
Deleted: sha256:8f2e09717443cb341c6811b420d0d5129c92a1f2ec3af4795c0cdaf9d8f6ccdc
Deleted: sha256:58969d76cbbc7255c4f86d6c39a861f2e56e58c9f316133f988b821a9205bf32
Deleted: sha256:b4e77298dcd6ddc409f7e8d0ae3ccd9fe141f8844fd466ecf44dc927d9030ae6
Deleted: sha256:c9b18059ed426422229b2c624582e54e7e32862378c9556b90a99c116ae10a04
```

当镜像有多个标签时，删除一个标签只会移除该标签引用，不会真正删除镜像数据，需要使用 `docker rmi` 加上镜像 ID 才能彻底删除镜像文件。镜像 ID 可以通过 `docker images` 命令进行查看。

```
$ docker rmi 1f94323bafb2
```

2. 清除镜像缓存

Docker 使用过程中会积累大量未使用的镜像缓存，占用磁盘空间。通过 `docker image prune` 命令可以清理这些无用镜像。

不加参数时，这个命令只删除悬空镜像（没有标签且不被任何镜像引用的中间层）。加上 `-a` 参数会删除所有未被容器或镜像引用的镜像，包括构建缓存和旧版本镜像。

```
$ docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B
```

在执行清理前应该先用 `docker images` 查看镜像列表，确认哪些镜像可以删除。删除操作不可逆，重要的镜像需要提前备份。

清理完成后可以用 `docker system df` 查看磁盘使用情况，确认空间已经释放。定期清理镜像缓存能有效节省存储空间，保持 Docker 环境整洁高效。

```
$ docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          1         0         197.7MB   197.7MB (100%)
Containers      0         0         0B        0B
Local Volumes   2         0         41.46MB   41.46MB (100%)
Build Cache     46        0         300.5MB   300.5MB
```

3.4 容器使用

3.4.1 创建并启动容器

在获取镜像并下载完成之后，可以通过 `docker run` 命令运行容器，这个命令的基本格式为：

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

常用参数说明：

- `-d`：以后台模式运行容器。
- `-p`：端口映射，格式为 `主机端口号:容器端口号`，例如 `80:80`。
- `--name`：指定容器名称。
- `-it`：保持交互式终端连接。
- `-v`：挂载主机目录到容器，格式为 `-v 主机目录:容器目录`。
- `--rm`：容器停止时自动删除容器。

- `--env` 或 `-e`：设置环境变量，格式为 `--env 变量名=变量值`。
- `--network`：指定容器的网络模式。
- `--restart`：容器的重启策略。
- `-u`：指定用户运行，格式为 `-u 用户名`。

以 `nginx` 为例，如果我们想要启动一个 `nginx` 容器，可以通过以下命令进行启动。

```
$ docker run -d -p 80:80 --name nginx-container nginx
```

通过以上命令，Docker 会进行以下操作：

1. 使用指定的 `nginx` 镜像构建容器。
2. 允许容器在后台运行。
3. 将容器的 `80` 端口映射到主机的 `80` 端口上。
4. 指定容器名称为 `nginx-container`。
5. 启动容器。

在容器启动后，通过主机的 IP 地址就能看到 nginx 的默认欢迎页面了。

3.4.2 启动已经终止的容器

如果容器因为某种原因停止运行，可以使用 `docker start 容器ID` 命令重新启动它。

> **注意**：如果每次运行容器使用 `docker run` 命令，则每次都都会创建一个新的容器实例，即使使用相同的镜像和参数也会生成不同的容器。

3.4.3 终止容器

要终止正在运行的 Docker 容器，可以使用 `docker stop` 命令。这个命令会向容器发送 SIGTERM 信号，让容器有机会执行清理工作并正常关闭。如果容器在指定时间内没有停止，Docker 会强制发送 SIGKILL 信号终止容器。

```
$ docker stop nginx-container
nginx-container
```

3.4.4 查看容器

在 Docker 使用过程中，经常需要查看容器的运行状态和信息。Docker 提供了多个命令来查看容器。

1. 查看正在运行的容器

使用 `docker ps` 命令可以查看当前正在运行的容器。这个命令会显示容器的 ID、名称、使用的镜像、创建时间、状态和端口映射等信息。

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS          PORTS                    NAMES
a1b2c3d4e5f6   nginx     "/docker-entrypoint.…"   2 minutes ago
Up 2 minutes   0.0.0.0:80->80/tcp   nginx-container
```

2. 查看所有容器

`docker ps -a` 命令可以查看所有容器，包括已经停止的容器。这个命令的输出格式与 `docker ps` 相同，但会显示更多容器。

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED
       STATUS                      PORTS                  NAMES
a1b2c3d4e5f6   nginx          "/docker-entrypoint.…"   5 minutes
ago    Up 5 minutes                0.0.0.0:80->80/tcp   nginx-
container
b2c3d4e5f6a1   redis:alpine   "docker-entrypoint.s…"   2 days ago
      Exited (0) 2 days ago                            redis-
test
```

3. 查看容器详细信息

`docker inspect` 命令可以查看容器的详细信息。这个命令会返回 JSON 格式的数据，包含容器的配置、网络设置、挂载卷等完整信息。

```
$ docker inspect nginx-container
```

4. 查看容器日志

`docker logs` 命令可以查看容器的日志输出。这个命令对于排查容器运行问题很有帮助。加上 `-f` 参数可以实时查看日志输出。

```
$ docker logs nginx-container
$ docker logs -f nginx-container
```

5. 查看容器资源使用情况

`docker stats` 命令可以实时查看容器的资源使用情况，包括 CPU、内存、网络和磁盘 I/O。

```
$ docker stats
CONTAINER ID   NAME              CPU %     MEM USAGE / LIMIT
MEM %     NET I/O           BLOCK I/O         PIDS
a1b2c3d4e5f6   nginx-container   0.00%     2.5MiB / 1.952GiB
0.13%     1.45kB / 648B     0B / 0B           2
```

3.4.5 进入容器内部

当容器在后台运行时，我们可以通过 `docker exec` 命令来进入运行中的容器内部执行命令或查看状态。

```
$ docker exec -it nginx-container /bin/bash
```

以上命令会启动一个bash shell 让我们能与 `nginx-container` 容器进行交互，`-it` 参数会保持终端交互连接。

通过这种方式可以像操作普通 Linux 系统一样在容器内执行命令、查看文件或修改配置。容器启动后，Docker 会为它分配唯一的 ID 和名称，自动设置网络和存储，并根据镜像定义启动指定的进程。

3.4.6 容器的导出和导入

容器的导出和导入功能用于将容器及其当前状态保存为文件，便于迁移或备份。

导出容器可以使用 `docker export` 命令，这个命令会将容器的文件系统打包成 tar 归档文件，但不包含容器的元数据、网络配置或卷信息。

```
$ docker export -o nginx-container.tar nginx-container
```

使用上面的容器导出命令会将名为 nginx-container 的容器导出到当前目录下的 nginx-container.tar 文件中。

容器导出后的文件可以通过 `docker import` 命令重新导入为镜像，导入时需要指定镜像名称和标签。

```
$ docker import nginx-container.tar my-nginx:v1
```

使用上面的导入命令会创建一个名为 my-nginx、标签为 v1 的新镜像。

3.4.7 删除容器

当容器不再需要时，可以使用 `docker rm` 命令将其删除，这个命令需要指定容器 ID 或容器名称。

```
docker rm nginx-container
```

使用上面的删除容器命令，会删除名为 nginx-container 的容器。

> **注意**：如果容器正在运行，直接删除会失败，需要先使用 `docker stop` 停止容器，或者添加 `-f` 参数强制删除运行中的容器。

3.5 私有镜像仓库

除了使用公共仓库，可以搭建私有镜像仓库来存储内部镜像。Docker 官方提供了 Registry 镜像用于搭建私有仓库。

3.5.1 私有镜像仓库搭建

1. 拉取 Registry 镜像

运行命令拉取最新版 Registry 镜像。

```
$ docker pull registry:2
```

2. 运行 Registry 容器

使用以下命令启动 Registry 容器。

```
$ docker run -d \
  -p 5000:5000 \
  --name my-registry \
  -v /data/registry:/var/lib/registry \
  registry:2
```

参数说明：

- `-p 5000:5000`：将容器 5000 端口映射到主机 5000 端口。
- `-v /data/registry:/var/lib/registry`：挂载数据目录持久化存储镜像。
- `registry:2`：指定使用的镜像版本。

这会在本地 5000 端口启动一个私有仓库。

3. 验证私有仓库运行

检查容器是否正常运行：

```
$ docker ps
```

访问 `http://localhost:5000/v2/_catalog` 查看仓库内容，正常应返回空列表：

```
{"repositories":[]}
```

3.5.2 私有镜像仓库使用

1. 推送镜像到私有仓库

首先给本地镜像打标签，包含私有仓库地址。

```
$ docker tag my-image localhost:5000/my-image
```

然后推送镜像到仓库。

```
$ docker push localhost:5000/my-image
```

2. 从私有仓库拉取镜像

```
$ docker pull localhost:5000/my-image
```

3. 查看仓库中的镜像

通过 API 查看仓库中的镜像列表：

```
$ curl http://localhost:5000/v2/_catalog
{"repositories":["my-image"]}
```

查看特定镜像的标签：

```
curl http://localhost:5000/v2/my-image/tags/list
```

3.5.3 配置远程访问

默认仓库只能在本地访问。要允许远程访问需修改配置。

1. 服务器私有镜像仓库搭建

按照 3.5.1 的方式，在想要进行远程访问的服务器上搭建私有镜像仓库。

1. 修改 daemon.json 文件

编辑本地 `/etc/docker/daemon.json` 文件，添加仓库地址到 `insecure-registries` 中。

```
{
  "insecure-registries": ["your-server-ip:5000"]
}
```

2. 重启 Docker 服务

```
systemctl restart docker
```

3. 远程推送镜像

```
$ docker tag my-image your-server-ip:5000/my-image:v1
$ docker push your-server-ip:5000/my-image:v1
```

4. 创建 htpasswd 认证文件

安装 htpasswd 工具：

```
$ yum install httpd-tools  # CentOS
$ apt-get install apache2-utils  # Ubuntu
```

创建认证文件：

```
$ mkdir auth
$ htpasswd -Bbn testuser testpassword > auth/htpasswd
```

5. 使用 TLS 加密

使用下面命令生成自签名证书

```
$ mkdir certs
$ openssl req -newkey rsa:4096 -nodes -sha256 \
  -keyout certs/domain.key -x509 -days 365 \
  -out certs/domain.crt
```

6. 启动带认证和加密的仓库

```
docker run -d \
  -p 443:443 \
  --name my-registry \
  -v /data/registry:/var/lib/registry \
  -v $(pwd)/certs:/certs \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
  registry:2
```

7. 使用 Nginx 反向代理

示例 Nginx 配置：

```
server {
    listen 443 ssl;
    server_name registry.example.com;

    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    location / {
        proxy_pass http://registry:5000;
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

8. 使用 docker-compose 部署

创建 docker-compose.yml 文件，示例 docker-compose.yml 配置：

```
version: '3'

services:
  registry:
    image: registry:2
    ports:
      - "5000:5000"
    environment:
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
      REGISTRY_AUTH_HTPASSWD_REALM: Registry Realm
    volumes:
      - ./auth:/auth
      - ./data:/var/lib/registry
```

7. 使用 Nginx 反向代理

生产环境建议使用 Nginx 作为 Registry 的反向代理，提供 TLS 加密和认证。

示例 Nginx 配置：

```
server {
```

```
    listen 443 ssl;
    server_name registry.example.com;

    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    location / {
        proxy_pass http://registry:5000;
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

启动服务：

```
$ docker-compose up -d
```

4. Docker Dockerfile

4.1 Dockerfile 简单使用

> **Dockerfile**：Dockerfile 是用来构建 Docker 镜像的文本文件，它包含了一系列的指令和配置，用于告诉 Docker 如何构建一个镜像。

使用 Dockerfile 可以自动构建镜像，这样在不同的电脑上都能得到一样的镜像。Dockerfile 里可以写很多不同的指令，比如指定用什么基础镜像、往镜像里放什么文件、安装什么软件、设置什么环境变量、打开什么端口、运行什么命令等。每一条指令都会在镜像里新建一层，所有层叠在一起就是最终的镜像。这种分层的方式让镜像构建更快，也更容易分享和重复使用。

在实际开发中，虽然官方镜像提供了基础运行环境，但项目通常需要特定的配置、依赖、文件、程序代码，这时我们就需要用 Dockerfile 来定制自己的镜像。

以 Web 开发为例，我们可以在官方 nginx 镜像基础上，添加自定义的网页文件、修改配置文件、安装必要的工具等。这样就能保证开发、测试和线上环境完全一致，不会因为环境不同出问题。

具体操作如下：

1. 创建一个新的空白目录 `myweb`。
2. 在目录 `myweb` 下创建一个文本文件，命名为 `Dockerfile`，并在文件中添加如下内容。

```
FROM nginx:latest
RUN echo '<h1>My Custom Nginx Page</h1>' >
/usr/share/nginx/html/index.html
EXPOSE 80
```

这个 Dockerfile 做了三件事：第一行 `FROM nginx:latest` 指定使用最新版 nginx 作为基础镜像；第二行 `RUN` 命令将网页内容放到容器的 nginx 默认的网页目录中；第三行 `EXPOSE 80` 声明容器会使用 `80` 端口。

3. 在 `myweb` 目录下运行命令 `docker build -t my-web .` 进行自定义 nginx 镜像构建。

这样就完成了自定义 nginx 镜像的构建。其中 `-t my-web` 是给新镜像命名为 my-web，最后的点表示使用当前目录的 Dockerfile。

构建完成后可以用 `docker images` 查看新构建的镜像。

```
$ docker images
```

4. 运行 `docker run -d -p 80:80 my-web` 启动容器。

这时访问主机的 80 端口就能看到自定义网页了。

4.2 Dockerfile 常用指令

如果需要更复杂的定制，可以在 Dockerfile 中添加更多指令，Dockerfile 常用指令如下表所示。

命令	描述
FROM	指定基础镜像，用于后续指令构建。
LABEL	添加镜像的元数据，使用键值对的形式。
RUN	创建镜像时，在镜像中要执行的命令。
CMD	指定容器启动时默认要执行的命令（可以被覆盖）。如果执行 `docker run` 后面跟启动命令会被覆盖掉。
ENTRYPOINT	设置容器创建时的主要命令（不可被覆盖）。
SHELL	覆盖Docker中默认的shell，用于RUN、CMD和ENTRYPOINT指令。
EXPOSE	声明容器运行时监听的特定网络端口。
ENV	设置环境变量
COPY	将文件或目录复制到镜像中。
ADD	将文件、目录或远程 URL 复制到镜像中。
WORKDIR	指定工作目录
VOLUME	为容器创建挂载点或声明卷。
USER	切换执行后续命令的用户和用户组，但这个用户必需首先已使用 RUN 的命令进行创建好了。
ARG	定义在构建过程中传递给构建器的变量，可使用 "docker build" 命令设置。
ONBUILD	当该镜像被用作另一个构建过程的基础时，添加触发器。
STOPSIGNAL	设置发送给容器以退出的系统调用信号。
HEALTHCHECK	定义周期性检查容器健康状态的命令。

> **FROM 指令**：FROM 指令用于指定基础镜像，必须是 Dockerfile 的第一个指令。基础镜像可以是官方镜像如 `ubuntu:20.04`，也可以是用户自定义的镜像。如果不指定标签，默认使用 `latest` 标签。例如 `FROM nginx` 表示基于最新版 nginx 镜像构建。每个 Dockerfile 可以有多个 FROM 指令用于构建多阶段镜像，但最终只会保留最后一个 FROM 生成的镜像层。基础镜像的选择直接影响最终镜像的大小和安全性，通常推荐使用官方维护的最小化镜像如 `alpine` 版本。最后的 `AS name` 可以选择为新生成阶段指定名称。

-

- FROM 指令格式：

```
FROM <image>[:<tag>] [AS <name>]
```

- 参数说明：
  - `<image>`：必需，指定基础镜像名称。
  - `:<tag>`：可选，指定镜像版本标签。
  - `AS <name>`：可选，为构建阶段命名。
- FROM 指令示例：

```
FROM nginx:1.23
FROM python:3.9-alpine
FROM ubuntu:20.04 AS builder
```

> **LABEL 指令**：用于添加元数据到镜像，采用键值对格式。

- LABEL 指令格式：

```
LABEL <key>=<value>
```

- LABEL 指令示例：

```
LABEL version="1.0.1"
LABEL maintainer="admin@example.com"
```

> **RUN 指令**：在镜像构建过程中执行命令，每条 RUN 指令都会创建一个新的镜像层。RUN 有两种格式：Shell 格式（`RUN <command>`）和 Exec 格式（`RUN ["executable", "param1", "param2"]`）。Shell 格式默认使用 `/bin/sh -c` 执行，Exec 格式直接调用可执行文件。为了减少镜像层数，建议将多个命令合并到单个 RUN 指令中，用 `&&` 连接命令，用 `\` 换行。例如安装软件包时应该先更新包列表再安装，最后清理缓存。

- RUN 指令格式：

```
# Shell 格式
RUN <command>

# Exec 格式
RUN ["executable", "param1", "param2"]
```

- RUN 指令示例：

```
RUN yum -y install wget
RUN apt-get update && apt-get install -y \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/*
RUN ["/bin/bash", "-c", "echo Hello, Docker!"]
RUN ["yum", "-y", "install", "wget"]
```

> **CMD 指令**：指定容器启动时的默认命令，一个 Dockerfile 只能有一个有效的 CMD 指令。如果 `docker run` 指定了命令，CMD 会被覆盖。CMD 有三种格式：Exec 格式（推荐）、Shell 格式和作为 ENTRYPOINT 的参数。Exec 格式直接调用可执行文件，不经过 shell 处理环境变量；Shell 格式会通过 `/bin/sh -c` 执行。例如运行 Python 应用时可以使用 `CMD ["python", "app.py"]`。

- CMD 指令格式：

```
# Exec 格式（推荐）
CMD ["executable","param1","param2"]

# Shell 格式
CMD command param1 param2

# 作为 ENTRYPOINT 的参数
CMD ["param1","param2"]
```

- CMD 指令示例：

```
# Shell 格式示例：运行 Python 脚本
CMD python app.py

# Exec 格式示例：运行 Nginx
CMD ["nginx", "-g", "daemon off;"]

# 作为 ENTRYPOINT 参数
CMD ["--port=8080"]
```

> **ENTRYPOINT 指令**：用于设置容器启动时的主要命令，与 CMD 不同，它不会被 docker run 后面的命令覆盖。

- ENTRYPOINT 指令格式：

```
# Shell 格式
ENTRYPOINT command param1 param2

# Exec 格式
ENTRYPOINT ["executable", "param1", "param2"]
```

- ENTRYPOINT 指令示例：

```
# Shell 格式示例：运行 Python 脚本
ENTRYPOINT python app.py

# Exec 格式示例：运行 Nginx
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

> **SHELL 指令**：用于覆盖 Docker 默认的 shell 程序。默认情况下 Linux 使用 `["/bin/sh", "-c"]`，Windows 使用 `["cmd", "/S", "/C"]`。这个指令会影响 RUN、CMD 和 ENTRYPOINT 的 shell 格式执行方式。

- SHELL 指令格式

```
SHELL ["executable", "parameters"]
```

- SHELL 指令示例：

```
# 切换为 PowerShell
SHELL ["powershell", "-command"]
# 切换回默认 shell
SHELL ["/bin/sh", "-c"]
```

> **EXPOSE 指令**：声明容器运行时监听的网络端口，只是文档性质的说明，不会实际发布端口。实际端口映射需要在运行容器时通过 `-p` 参数设置。EXPOSE 可以指定 TCP 或 UDP 协议，默认是 TCP。例如 `EXPOSE 80/tcp` 表示容器会监听 80 端口。这个指令主要作用是帮助使用者理解容器提供的服务端口，同时被一些编排工具如 Docker Compose 使用。

- EXPOSE 指令格式：

```
EXPOSE <port> [<port>/<protocol>...]
```

- EXPOSE 指令示例：

```
# 声明单个端口
EXPOSE 80
# 声明多个端口
EXPOSE 80 443
# 指定协议
EXPOSE 53/udp
```

> **ENV 指令**：设置环境变量，这些变量会在构建阶段和容器运行时生效。变量可以被后续的 RUN、CMD 等指令使用，也会持久化到容器中。定义的环境变量会出现在 `docker inspect` 的输出中，也可以在容器运行时通过 `docker run --env` 参数覆盖。

- ENV 指令格式：

```
# 定义单个变量
ENV <key> <value>
# 一次性定义多个变量
ENV <key1>=<value1> <key2>=<value2>...
```

- ENV 指令示例：

```
# 设置单个变量
ENV NODE_VERSION=18.15.0
# 设置多个变量
ENV NODE_VERSION=18.15.0 NODE_ENV=production
```

> **COPY 指令**：将文件或目录从构建上下文复制到镜像中，源路径必须是相对路径（相对于 Dockerfile 所在目录），不能使用绝对路径或 `../` 父目录引用。目标路径可以是绝对路径或相对于 WORKDIR 的路径。COPY 会保留文件元数据（权限、时间戳），但不支持自动解压压缩包。与 ADD 指令相比，COPY 更推荐用于简单的文件复制操作，因为它的行为更可预测。例如复制项目代码到镜像的 `/app` 目录。

- COPY 指令格式：

```
COPY <src>... <dest>
```

- COPY 指令示例：

```
# 复制单个文件
COPY requirements.txt /app/
# 复制整个目录
COPY src /app/src
# 使用通配符复制多个文件
COPY *.sh /scripts/
```

> **ADD 指令**：ADD 指令功能类似 COPY，但增加了自动解压压缩包和处理远程 URL 的能力。当源路径是本地压缩文件（如 .tar、.gz）时，ADD 会自动解压到目标路径。源路径也可以是 URL，Docker 会下载文件到镜像中。例如 `ADD https://example.com/file.tar.gz /tmp` 会下载并解压文件。由于 ADD 行为较复杂，官方建议优先使用 COPY，除非明确需要解压或下载功能。

- ADD 指令格式：

```
ADD <src>... <dest>
```

- ADD 指令示例：

```
# 添加本地文件
ADD app.jar /opt/app/
# 自动解压压缩包
ADD project.tar.gz /app
# 从 URL 下载文件
ADD https://example.com/data.json /data
```

> **WORKDIR 指令**：设置工作目录，相当于 `cd` 命令的效果，如果目录不存在会自动创建。后续的 RUN、CMD、COPY 等指令都会在此目录下执行。WORKDIR 可以多次使用，路径可以是相对路径（基于前一个 WORKDIR）。例如 `WORKDIR /app` 后接 `WORKDIR src` 最终路径是 `/app/src`。使用 WORKDIR 可以避免在 RUN 指令中频繁使用 `cd` 命令，使 Dockerfile 更清晰。

- WORKDIR 指令格式：

```
WORKDIR <path>
```

- WORKDIR 指令示例：

```
WORKDIR /usr/src
WORKDIR app
RUN pwd  # 输出 /usr/src/app
```

> **VOLUME 指令**：创建挂载点或声明卷，会在容器运行时自动挂载匿名卷。主要用途是保留重要数据（如数据库文件）或共享目录。例如 `VOLUME /data` 确保 `/data` 目录的数据持久化。实际挂载的主机目录可以通过 `docker inspect` 查看。VOLUME 指令不能在构建阶段向挂载点写入数据，因为这些数据在运行时会被覆盖。数据卷可以在容器间共享和重用，即使容器被删除，卷数据仍然存在。VOLUME 声明后，运行时可以通过 -v 参数覆盖，但无法在构建阶段向挂载点写入数据（会被运行时覆盖）。

- VOLUME 指令格式：

```
VOLUME ["<path1>", "<path2>", ...]
```

- VOLUME 指令示例：

```
# 声明单个卷
VOLUME /data
# 声明多个卷
VOLUME ["/data", "/config"]
```

> **USER 指令**：切换执行后续指令的用户身份，用户必须已通过 RUN 指令创建。例如 `USER nobody` 让后续命令以 nobody 身份运行，增强安全性。该用户需要有足够的权限访问所需文件。USER 会影响 RUN、CMD、ENTRYPOINT 的执行身份。在运行时可以通过 `docker run -u` 覆盖此设置。典型的用法是在安装软件包后创建非 root 用户并切换，避免容器以 root 权限运行。

- USER 指令格式：

```
USER <user>[:<group>]
```

- USER 指令示例：

```
RUN groupadd -r app && useradd -r -g app appuser
USER appuser
CMD ["python", "app.py"]
```

> **ARG 指令**：指令定义构建时的变量。这些变量只在构建阶段有效，不会保留到容器运行时。可以通过 `docker build --build-arg <name>=<value>` 覆盖默认值。例如 `ARG VERSION=latest` 定义版本变量。ARG 变量可以用于控制构建流程，如选择不同的软件版本。常见的预定义变量包括 HTTP_PROXY 等代理设置。

- ARG 指令格式：

```
ARG <name>[=<默认值>]
```

- ARG 指令示例：

```
ARG NODE_VERSION=14
FROM node:${NODE_VERSION}
```

> **ONBUILD 指令**：设置触发器。当当前镜像被用作其他镜像的基础时，这些指令会被触发执行。例如 `ONBUILD COPY . /app` 会在子镜像构建时自动复制文件。ONBUILD 常用于创建基础镜像模板，子镜像可以继承特定的构建步骤。通过 `docker inspect` 可以查看镜像的 ONBUILD 触发器。

- ONBUILD 指令格式：

```
ONBUILD <其他指令>
```

- ONBUILD 指令示例：

```
ONBUILD RUN npm install
ONBUILD COPY . /app
```

> **STOPSIGNAL 指令**：设置容器停止时发送的系统信号。信号可以是数字（如 9）或信号名（如 SIGKILL）。默认的信号是 SIGTERM，允许容器优雅退出。如果需要强制终止，可以设置为 SIGKILL。例如 `STOPSIGNAL SIGTERM` 确保容器收到终止信号。这个设置会影响 `docker stop` 命令的行为。

- STOPSIGNAL 指令格式：

```
STOPSIGNAL <信号>
```

- STOPSIGNAL 指令示例：

```
STOPSIGNAL SIGQUIT
```

> **HEALTHCHECK 指令**：定义容器健康检查，Docker 会定期执行检查命令判断容器是否健康。检查命令返回 0 表示健康，1 表示不健康。选项包括 `--interval`（检查间隔）、`--timeout`（超时时间）、`--start-period`（启动宽限期）和 `--retries`（重试次数）。例如 `HEALTHCHECK --interval=30s CMD curl -f http://localhost/` 每30秒检查Web服务是否响应。健康状态可以通过 `docker ps` 查看。

- HEALTHCHECK 指令格式：

```
HEALTHCHECK [OPTIONS] CMD <command>
```

- HEALTHCHECK 指令示例：

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

4.3 Dockerfile 实际使用

4.3.1 构建 Python Web 应用镜像

假设有一个简单的 Python Flask 应用，需要将其打包成 Docker 镜像。以下是完整的 Dockerfile 示例：

```
# 使用官方 Python 基础镜像
FROM python:3.9-slim

# 设置工作目录
WORKDIR /app

# 复制当前目录下的文件到工作目录
COPY . .

# 安装依赖
RUN pip install --no-cache-dir -r requirements.txt

# 暴露端口
EXPOSE 5000

# 定义环境变量
ENV FLASK_APP=app.py
ENV FLASK_ENV=production

# 启动命令
CMD ["flask", "run", "--host=0.0.0.0"]
```

这个 Dockerfile 的执行步骤如下：

1. 使用 `python:3.9-slim` 作为基础镜像，这是一个轻量级的 Python 环境。
2. 设置工作目录为 `/app`。

3. 将当前目录的所有文件复制到容器的 `/app` 目录。
4. 运行 `pip install` 安装依赖包。
5. 声明容器运行时监听的端口为 5000。
6. 设置两个环境变量，指定 Flask 应用入口和运行环境。
7. 最后定义容器启动时执行的命令。

构建镜像的命令：

```
docker build -t my-python-app .
```

运行容器的命令：

```
docker run -d -p 5000:5000 my-python-app
```

4.3.2 构建 Node.js 应用镜像

下面是一个 Node.js 应用的 Dockerfile 示例：

```
# 使用官方 Node.js 基础镜像
FROM node:16-alpine

# 设置工作目录
WORKDIR /usr/src/app

# 复制 package.json 和 package-lock.json
COPY package*.json ./

# 安装依赖
RUN npm install

# 复制所有源代码
COPY . .

# 构建应用
RUN npm run build

# 暴露端口
```

```
EXPOSE 3000

# 启动命令
CMD ["npm", "start"]
```

这个 Dockerfile 的执行步骤如下：

1. 使用 `node:16-alpine` 作为基础镜像，这是一个基于 Alpine Linux 的 Node.js 环境。
2. 设置工作目录为 `/usr/src/app`。
3. 先复制 `package.json` 和 `package-lock.json` 文件，这样可以利用 Docker 的缓存层。
4. 运行 `npm install` 安装依赖。
5. 复制所有源代码到容器中。
6. 运行构建命令。
7. 声明容器运行时监听的端口为 3000。
8. 定义容器启动时执行的命令。

构建镜像的命令：

```
docker build -t my-node-app .
```

运行容器的命令：

```
docker run -d -p 3000:3000 my-node-app
```

4.3.3 构建 Nginx 静态网站镜像

如果需要部署一个静态网站，可以使用 Nginx 作为 Web 服务器。以下是 Dockerfile 示例：

```
# 使用官方 Nginx 基础镜像
FROM nginx:alpine

# 删除默认的 Nginx 配置
RUN rm -rf /etc/nginx/conf.d/default.conf

# 复制自定义配置
COPY nginx.conf /etc/nginx/conf.d/
```

```
9
10 # 复制静态网站文件
11 COPY dist/ /usr/share/nginx/html/
12
13 # 暴露端口
14 EXPOSE 80
15
16 # 启动 Nginx
17 CMD ["nginx", "-g", "daemon off;" ]
```

这个 Dockerfile 的执行步骤如下：

1. 使用 `nginx:alpine` 作为基础镜像，这是一个轻量级的 Nginx 环境。
2. 删除默认的 Nginx 配置文件。
3. 复制自定义的 Nginx 配置文件到容器中。
4. 将构建好的静态网站文件复制到 Nginx 的默认网站目录。
5. 声明容器运行时监听的端口为 80。
6. 定义容器启动时执行的命令，以前台模式运行 Nginx。

需要准备一个 `nginx.conf` 配置文件，例如：

```
1  server {
2      listen 80;
3      server_name localhost;
4
5      location / {
6          root /usr/share/nginx/html;
7          index index.html;
8          try_files $uri $uri/ /index.html;
9      }
10 }
```

构建镜像的命令：

```
1  docker build -t my-static-site .
```

运行容器的命令：

```
docker run -d -p 8080:80 my-static-site
```

5. Docker Compose

5.1 Docker Compose 简介

> **Docker Compose**：用于定义和管理多容器应用的工具。它通过一个 YAML 文件来配置所有服务配置，用一条命令就能启动整个应用。Docker Compose 解决了多个容器之间的依赖关系和启动顺序问题，主要用于开发环境、测试环境和 CI/CD 流程中。Docker Compose 文件包含了容器镜像、端口映射、数据卷、环境变量等配置，所有配置集中在一个文件中，便于版本控制和团队协作。

5.2 Docker Compose 安装

Docker Compose 可以通过多种方式安装。在 Linux 系统上，可以直接下载二进制文件进行安装。运行以下命令可以安装最新版本的 Docker Compose：

```
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

安装完成后，可以通过运行 `docker-compose --version` 来验证安装是否成功。对于 Windows 和 macOS 用户，Docker Desktop 已经包含了 Docker Compose，不需要单独安装。如果系统已经安装了 Docker Engine，通常也会包含 Docker Compose 插件，可以通过 `docker compose version` 命令检查。

5.3 Docker Compose 文件结构

Docker Compose 的核心是 `docker-compose.yml` 文件，它采用 YAML 格式定义服务、网络和卷。一个基本的 Compose 文件包含以下部分：

- **version**：指定 Compose 文件格式版本，例如 `'3.8'`。
- **services**：定义各个容器服务，每个服务对应一个容器。

- **networks**：配置自定义网络。
- **volumes**：声明数据卷。

YAML 文件使用缩进来表示层级关系，冒号表示键值对，连字符表示列表项。

以下是一个典型的 `docker-compose.yml` 示例，定义了一个 WordPress 的应用配置。

```yaml
version: '3.8'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: wordpress
    networks:
      - backend

  wordpress:
    image: wordpress:latest
    ports:
      - "80:80"
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: root
      WORDPRESS_DB_PASSWORD: password
    depends_on:
      - db
    networks:
      - frontend
      - backend

networks:
  frontend:
  backend:

volumes:
```

```
33    db_data:
34
```

这个 WordPress 的应用配置示例中展示了：

- 两个服务：MySQL 数据库和 WordPress。
- 自定义网络：frontend 和 backend。
- 声明数据卷：数据卷 db_data 持久化数据库。
- 设置环境变量：设置 MySQL 数据库和 WordPress 相关环境变量。
- 端口映射：将容器 80 端口映射到主机 80 端口。
- 服务依赖关系：确保数据库服务先启动。

下面详细说明每个部分的作用和配置方法。

> **version**：指定使用的 Compose 文件格式版本。不同版本支持的功能不同。目前常用的是 3.x 版本。版本号需要用引号包裹。

示例：

```
1  version: '3.8'
```

版本号影响可用功能。例如 3.8 版本支持更多配置选项。版本号必须与安装的 Docker Compose 版本兼容。如果不指定版本，默认使用最新支持版本。

> **services**：定义需要运行的容器服务，是 Compose 文件的核心部分。每个服务对应一个容器。服务名称自定义，作为容器的标识。

基本服务配置示例：

```
1  services:
2    web:
3      image: nginx:latest
4      ports:
5        - "80:80"
```

常用服务配置项：

- **image**：指定使用的镜像名称和标签。可以从 Docker Hub 获取或使用本地构建的镜像。

- **build**：如果使用本地 Dockerfile 构建镜像，需要指定构建路径。

```
build: ./dir
```

- **ports**：设置端口映射，格式为 `"主机端口:容器端口"`。

```
ports:
  - "8080:80"
```

- **volumes**：配置数据卷挂载，支持主机路径和命名卷。

```
volumes:
  - /host/path:/container/path
  - named_volume:/container/path
```

- **environment**：设置环境变量，可以用列表或键值对格式。

```
environment:
  - VAR1=value1
  - VAR2=value2
```

- **depends_on**：定义服务启动顺序，确保依赖服务先启动。

```
depends_on:
  - db
```

- **restart**：设置容器重启策略，比如 `always` 表示总是自动重启。

```
restart: always
```

- **command**：覆盖容器启动命令。

```
command: ["python", "app.py"]
```

完整服务示例：

```
services:
  web:
    build: .
    ports:
```

```
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
    depends_on:
      - redis
  redis:
    image: redis:alpine
    volumes:
      - redis_data:/data
```

> **networks**：定义自定义网络。容器通过网络名称互相通信。默认会创建 bridge 网络。

示例：

```
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
```

服务中使用网络：

```
services:
  web:
    networks:
      - frontend
  db:
    networks:
      - backend
```

自定义网络提供更好的隔离性。不同网络的容器默认不能互相访问。

> **volumes**：声明数据卷。数据卷用于持久化存储和容器间共享数据。

示例：

```yaml
volumes:
  db_data:
    driver: local
  app_data:
    driver: local
```

服务中使用卷：

```yaml
services:
  db:
    volumes:
      - db_data:/var/lib/mysql
```

卷数据独立于容器生命周期。删除容器不会删除卷数据。

5.4 Docker Compose 常用操作命令

Docker Compose 提供了一系列命令来管理多容器应用：

- **启动服务**：`docker-compose up -d` 会在后台启动所有服务，`-d` 表示 detached 模式。
- **停止服务**：`docker-compose down` 停止并移除所有容器、网络和卷。
- **查看状态**：`docker-compose ps` 显示各容器的运行状态。
- **查看日志**：`docker-compose logs` 输出容器日志，加 `-f` 可以跟踪实时日志。
- **构建镜像**：如果服务使用本地 Dockerfile，`docker-compose build` 会重新构建镜像。
- **重启服务**：`docker-compose restart`，重启所有服务或指定服务。。
- **单服务操作**：可以针对单个服务执行命令，例如 `docker-compose start wordpress`。

这些命令大大简化了多容器应用的管理工作。通过组合使用这些命令，可以轻松控制整个应用的运行状态。

6. Docker 常用命令

4.1 Docker 服务命令

1. 启动 Docker 服务

```
$ systemctl start docker
```

2. 停止 Docker 服务

```
$ systemctl stop docker
```

3. 重启 Docker 服务

```
$ systemctl restart docker
```

4. 设置开机启动 Docker 服务

```
$ systemctl enable docker
```

5. 查看 Docker 服务状态

```
$ systemctl status docker
```

4.2 Docker 镜像命令

1. 查看本地的镜像信息

```
$ docker images
```

2. 从镜像仓库中拉取或者更新指定镜像（默认的镜像仓库是官方的 Docker Hub）

```
$ docker pull NAME[:TAG]
```

3. 从镜像仓库查找镜像

```
$ docker search NAME
```

4. 根据本地 Dockerfile 文件，构建镜像

```
# docker build -t 镜像名:版本号 .
$ docker build -t my_image:1.0 .
```

5. 删除本地镜像

```
# docker rmi 镜像名:版本号
$ docker rmi mysql:5.7
```

6. 导入镜像

```
# docker load -i 指定要导入的镜像压缩包文件名
$ docker load -i image.tar
```

7. 导出镜像

```
# docker save -o 导出的镜像压缩包的文件名 要导出的镜像名:版本号
$ docker save -o image.tar target_image:tag
```

8. 清除多余镜像缓存

```
$ docker system prune -a
```

4.3 Docker 容器命令

1. 创建容器

```
# 常用参数列表
# -d: 后台运行容器，并返回容器 ID
# -p: 指定端口映射，格式为：主机(宿主)端口:容器端口
# -i: 以交互模式运行容器，通常与 -t 同时使用
# -t: 为容器重新分配一个伪输入终端，通常与 -i 同时使用
# --name=my_container: 为容器指定一个名称
# --dns 8.8.8.8: 指定容器使用的 DNS 服务器，默认和宿主一致
$ docker run -d --name=my_container -p 8080:8080 tomcat:latest
```

2. 查看容器列表

```
# 查看正在运行的容器列表
$ docker ps

# 查看最近一次创建的容器
$ docker ps -l
```

```
# 查看正在运行的容器 ID 列表
$ docker ps -q

# 查看全部容器(包括已经停止的容器)
$ docker ps -a

# 查看全部容器 ID 列表
$ docker ps -aq
```

3. 停止运行的容器

```
# 使用容器名停止
$ docker stop my_container

# 使用容器 ID 停止
$ docker stop container_id

# 使用容器 ID 停止多个正在运行的容器
$ docker ps
```

4. 启动已停止的容器

```
# 容器名
$ docker start my_container

# 容器 ID
$ docker start container_id

# 使用容器 ID 启动多个已停止的容器
$ docker start `docker ps -aq`
```

5. 删除容器

```
# 用容器名删除
$ docker rm my_container

# 用容器 ID 删除
$ docker rm container_id

# 删除多个未运行的容器, 运行中的无法删除
$ docker rm `docker ps -aq`
```

6. 进入容器（正在运行的容器才可以进入）

```
# 使用容器名
$ docker exec -it my_container /bin/bash

# 使用容器 ID
$ docker exec -it container_id /bin/bash
```

7. 查看容器信息

```
# 容器名
$ docker inspect my_container

# 容器 ID
$ docker inspect container_id
```

参考资料

- Docker 官方文档 - docker.com
- Docker 官方镜像仓库 - docker.com
- Docker 教程 - runoob.com
- Docker — 从入门到实践 - yeasy
- 万字长文带你看全网最详细Dockerfile教程
- 万字长文：编写 Dockerfiles 最佳实践