# Docker

## 1.1简介：你需要了解的Docker

在了解学习 Docker 之前我们就非常有必要介绍下 Docker 的前生 LXC（Linux Container）。



### LXC 介绍

**LXC** 可以提供轻量级的虚拟化，用来隔离进程和资源，和我们传统观念中的全虚拟化完全不一样，非常轻量级。LXC 可以将单个操作系统管理的资源划分到独立的组中，和传统的虚拟化技术相比，LXC 有如下一些优势：

* 和宿主机使用同一个内核，所以性能损耗小
* 不需要指令级模拟
* 不需要即时编译
* 容器可以在 CPU 核心的本地运行指令，不需要任何专门的解释机制
* 避免了虚拟化和系统调用中的复杂性
* 轻量级隔离，隔离的同时还可以和宿主机共享资源

LXC 有点像 [chroot](https://zh.wikipedia.org/zh-cn/Chroot)，提供了一个拥有自己进程和网络空间的虚拟环境，但是和虚拟机又不一样，因为 LXC 是一种操作系统层面上的资源的虚拟化。

**chroot 简介：**chroot（change root），在 Linux 系统中，系统默认的目录就都是以 / 也就是根目录开头的，chroot 的使用能够改变当前的系统根目录结构，通过改变当前系统的根目录，我们能够限制用户的权利，在新的根目录下并不能够访问旧系统根目录的结构个文件，也就建立了一个与原系统完全隔离的目录结构。

更多关于 chroot 的介绍，可以查看 [理解 chroot](https://www.ibm.com/developerworks/cn/linux/l-cn-chroot/index.html) 一文。

### 什么是 Docker

Docker 并不是 LXC 替代品，Docker 底层就是使用的 LXC 来实现，LXC 将 Linux 进程沙盒化，使得进程之间相互隔离，还可以共享宿主机的资源。在 LXC 的基础上，Docker 提供了一系列更加强大方便的功能，使得 Docker 成为了现在最火的虚拟化技术。

由于之前我们的后台在开发和运维阶段的环境是不一致的，这就导致了 Docker 的出现，因为我们通过 Docker 可以将程序运行的环境也一起打包到版本控制去了，这样就排除了因为环境不同造成的各种麻烦事情了，也不会出现在本地可以在线上却不行这样的窘境了。

Docker 是一个开源的应用容器引擎，基于 go 语言开发，可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 服务器。容器是一个沙箱机制，相互之间不会有影响（类似于我们手机上运行的 app），并且容器开销是很低的。

用官方的话来说，Docker 受欢迎，是因为以下几个特点：

* 灵活性：即使是最复杂的应用也可以集装箱化
* 轻量级：容器利用并共享主机内核
* 可互换：您可以即时部署更新和升级
* 便携式：您可以在本地构建，部署到云，并在任何地方运行
* 可扩展：您可以增加并自动分发容器副本
* 可堆叠：您可以垂直和即时堆叠服务

#### Docker 几个重要概念

在了解了 Docker 是什么之后，我们需要先了解下 Docker 中最重要的3个概念：镜像、容器和仓库。

**镜像** 是一个只读模板，带有创建 Docker 容器的说明，一般来说的，镜像会基于另外的一些基础镜像并加上一些额外的自定义功能来组成。比如，你可以构建一个基于 Centos 的镜像，然后在这个基础镜像上面安装一个 Nginx 服务器，这样就可以构成一个属于我们自己的镜像了。

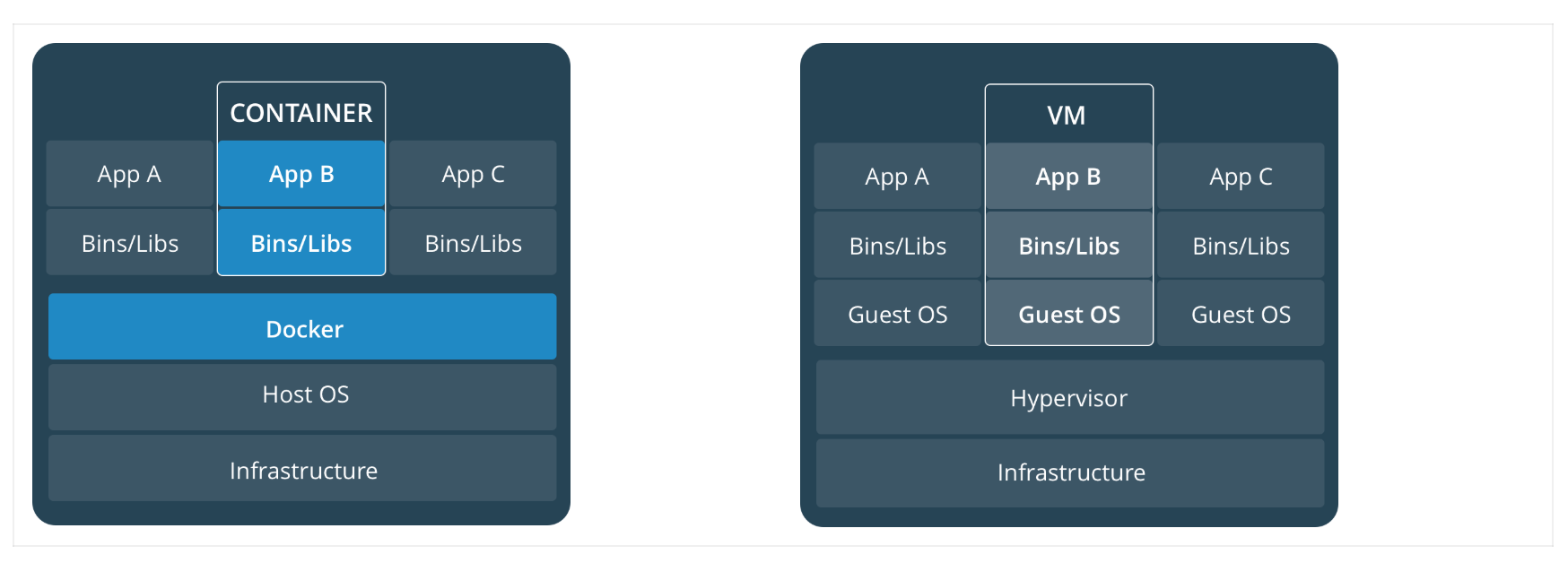
**容器** 是一个镜像的可运行的实例，可以使用 Docker REST API 或者 CLI 命令行工具来操作容器，容器的本质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。

**registry**是用来存储 Docker 镜像的仓库，Docker Hub 是 Docker 官方提供的一个公共仓库，而且 Docker 默认也是从 Docker Hub 上查找镜像的，当然你也可以很方便的运行一个私有仓库，当我们使用 docker pull 或者 docker run 命令时，就会从我们配置的 Docker 镜像仓库中去拉取镜像，使用 docker push 命令时，会将我们构建的镜像推送到对应的镜像仓库中，registry 可以理解为用于镜像的 github 这样的托管服务。

#### 容器和虚拟机

上面我们说到了容器是在 Linux 上本机运行，并与其他容器共享主机的内核，它运行一个独立的进程，不占用其他任何可执行文件的内存，非常轻量。

而虚拟机运行的是一个完整的操作系统，通过虚拟机管理程序对主机资源进行虚拟访问，相比之下需要的资源需要更多，但是非常安全，因为是独立的操作系统，独立的内核。



#### 支持 Docker 的底层技术

**Docker 本质就是宿主机的一个特殊进程**，Docker 是通过 namespace 实现资源隔离，通过cgroup 实现资源限制，通过写时复制技术（copy-on-write）实现了高效的文件操作（类似虚拟机的磁盘比如分配 500g 并不是实际占用物理磁盘 500g）

##### Namespaces

命名空间 (namespaces) 是 Linux 为我们提供的用于分离进程树、网络接口、挂载点以及进程间通信等资源的方法。在日常使用个人 PC 时，我们并没有运行多个完全分离的服务器的需求，但是如果我们在服务器上启动了多个服务，这些服务其实会相互影响的，每一个服务都能看到其他服务的进程，也可以访问宿主机器上的任意文件，一旦服务器上的某一个服务被入侵，那么入侵者就能够访问当前机器上的所有服务和文件，这是我们不愿意看到的，我们更希望运行在同一台机器上的不同服务能做到完全隔离，就像运行在多台不同的机器上一样。而 Docker 其实就通过 Linux 的 Namespaces 技术来实现的对不同的容器进行隔离。

当我们运行（docker run 或者 docker start）一个 Docker 容器时，Docker 会为该容器设置一系列的 namespaces，这些 namespaces 提供了一层隔离，容器的各个方面都在单独的 namespace 中运行，并且对其的访问仅限于该 namespace。

Docker 在 Linux 上使用以下几个命名空间（上面说的各个方面）：

* pid namespace：用于进程隔离（PID：进程ID）
* net namespace：管理网络接口（NET：网络）
* ipc namespace：管理对 IPC 资源的访问（IPC：进程间通信（信号量、消息队列和共享内存））
* mnt namespace：管理文件系统挂载点（MNT：挂载）
* uts namespace：隔离主机名和域名
* user namespace：隔离用户和用户组（3.8以后的内核才支持）

##### CGroups（Control Groups）

我们通过 Linux 的 namespaces 技术为新创建的进程隔离了文件系统、网络、进程等资源，但是 namespaces 并不能够为我们提供物理资源上的隔离，比如 CPU、内存、IO 或者网络带宽等，所以如果我们运行多个容器的话，则容器之间就会抢占资源互相影响了，所以对容器资源的使用进行限制就非常重要了，而 Control Groups（CGroups）技术就能够隔离宿主机上的物理资源。CGroups 由 7 个主要的子系统组成：分别是 cpuset、cpu、cpuacct、blkio、devices、freezer、memory，不同类型资源的分配和管理是由各个 CGroup 子系统负责完成的。

**CGroup 简介：**在 CGroup 中，所有的任务就是一个系统的一个进程，而 CGroup 就是一组按照某种标准划分的进程，在 CGroup 这种机制中，所有的资源控制都是以 CGroup 作为单位实现的，每一个进程都可以随时加入一个 CGroup 也可以随时退出一个 CGroup。

– [CGroup 介绍、应用实例及原理描述](https://www.ibm.com/developerworks/cn/linux/1506_cgroup/index.html)

– [Docker 背后的内核 Cgroups 机制](https://www.infoq.cn/article/docker-resource-management-cgroups/)

CGroup 具有以下几个特点：

* CGroup 的 API 以一个伪文件系统（/sys/fs/cgroup/）的实现方式，用户的程序可以通过文件系统实现 CGroup 的组件管理
* CGroup 的组件管理操作单元可以细粒度到线程级别，用户可以创建和销毁 CGroup，从而实现资源载分配和再利用
* 所有资源管理的功能都以子系统（cpu、cpuset 这些）的方式实现，接口统一子任务创建之初与其父任务处于同一个 CGroup 的控制组

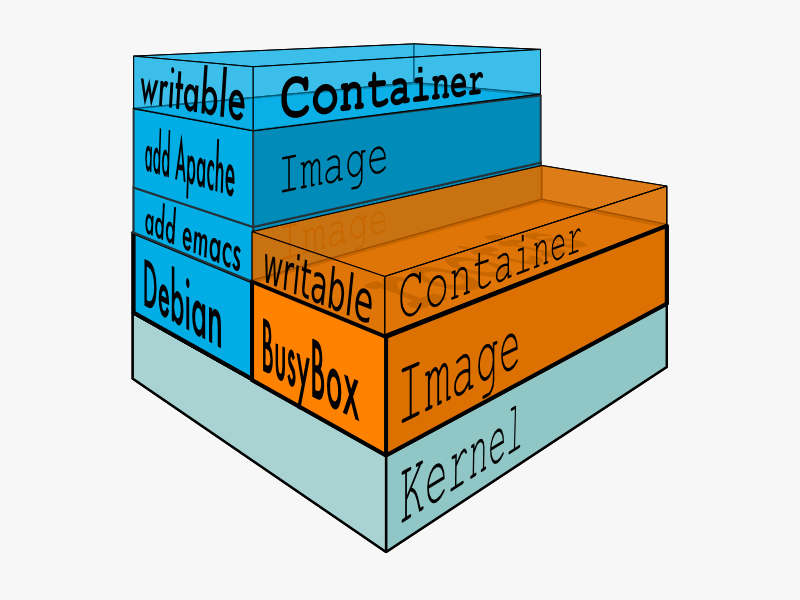
另外 CGroup 具有四大功能：

* 资源限制：可以对任务使用的资源总额进行限制
* 优先级分配：通过分配的 cpu 时间片数量以及磁盘 IO 带宽大小等，实际上相当于控制了任务运行优先级
* 资源统计：可以统计系统的资源使用量，如 cpu 时长，内存用量等
* 任务控制：cgroup 可以对任务执行挂起、恢复等操作

##### UnionFS

Linux 的命名空间和控制组分别解决了不同资源隔离的问题，前者解决了进程、网络以及文件系统的隔离，后者实现了 CPU、内存等资源的隔离，但是在 Docker 中还有另一个非常重要的问题需要解决 - 也就是镜像。

镜像到底是什么，它又是如何组成和组织的呢？而这其中最重要的概念就是镜像层(Layers)（如下图）的概念，而镜像层依赖于一系列的底层技术，比如文件系统(filesystems)、写时复制(copy-on-write)、联合挂载(union mounts)等。



Docker 镜像是由一系列的层组成的，每层代表 Dockerfile 中的一条指令，比如下面的 Dockerfile 文件：

FROM ubuntu:18.04

COPY . /app

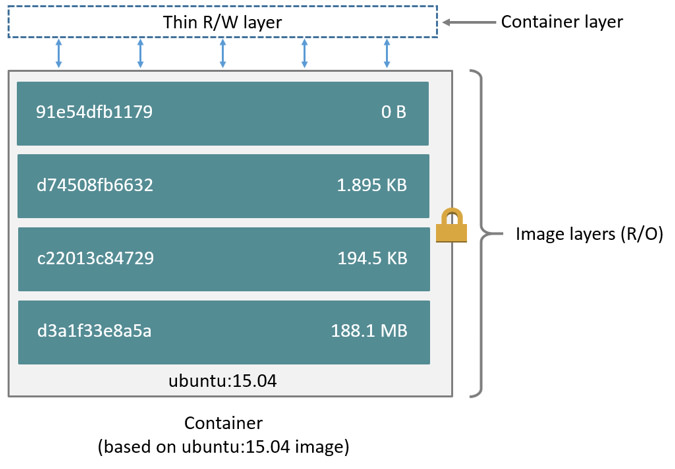
RUN make /app

CMD python /app/app.py

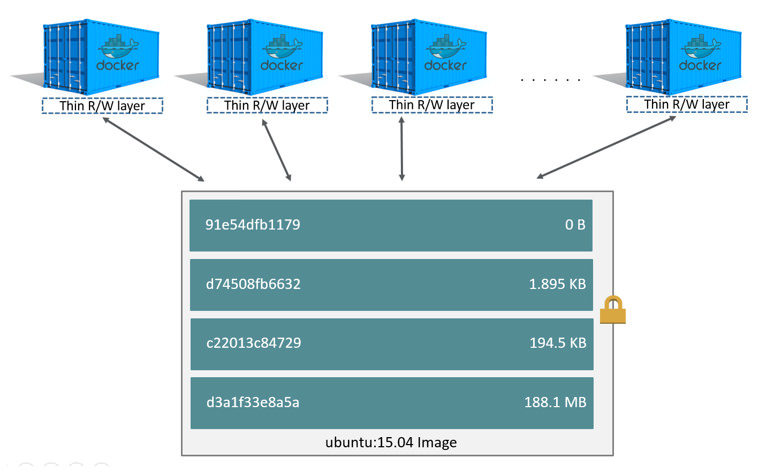
**Dockerfile 介绍：**Dockerfile 是一个文本文件，其内包含了一条条的指令，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。Dockerfile 是我们用来构建 Docker 镜像的一个说明文档，也是我们学习的重点，必须要要掌握如何编写 Dockerfile。

这里的 Dockerfile 包含4条命令，其中每一行就创建了一层，FROM 语句从 ubuntu:18.04 这个基础镜像创建一个层开始，COPY 命令从 Docker 客户端的当前目录添加一些新的文件，RUN 指令使用 make 命令构建应用，最后一层指定在容器中运行什么命令。

镜像就是由这些层一层一层堆叠起来的，镜像中的这些层都是只读的，当我们运行容器的时候，就可以在这些基础层之上添加新的可写层，也就是我们通常说的容器层，对于运行中的容器所做的所有更改（比如写入新文件、修改现有文件、删除文件）都将写入这个容器层，下面显示了基于 Ubuntu 15.04 镜像运行的容器层的结构：



容器和镜像之间的主要区别就是容器在镜像顶部由一个可写层，在容器中的所有操作都会存储在这个容器层中，删除容器后，容器层也会被删除，但是镜像不会变化。正因为每个容器都有自己的可写容器层，所有更改都存储在自己的容器层中，所以多个容器之间可以共享同一基础镜像的访问，但仍然具有自己的数据状态。如下图演示了多个容器共享同一镜像的请情况：



Docker 使用存储驱动程序来管理镜像层和可写容器层的内容，每个存储驱动程序的处理方式不同，但是所有的驱动都使用可堆叠的镜像层和写时复制（Cow）策略，这些驱动程序管理的这些层其实就是 UnionFS（联合文件系统），现在 Docker 主要支持的存储驱动有 aufs、devicemapper、overlay、overlay2、zfs 和 vfs 等等，在新的 Docker 版本中，overlay2 取代了 aufs 成为了推荐的存储驱动。

**Copy-on-write**

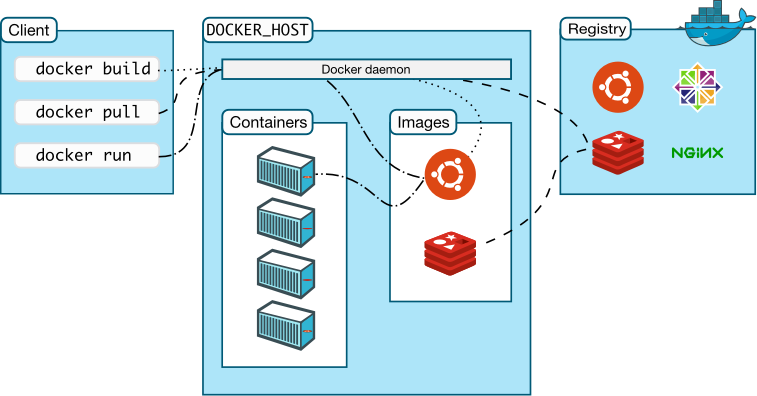
写时复制是一种共享和复制文件的策略，可以最大程度地提高效率，如果文件或目录位于镜像的较低层中，而另一层（包括可写层）需要对其进行读取访问，则它直接使用现有文件即可。另一层第一次需要修改文件时（在构建镜像或运行容器时），将文件复制到该层并进行修改。这样可以将 I/O 和每个后续层的大小最小化。

**选择存储驱动**

对于 Docker 如何选择一个合适的存储驱动程序，可以查看官方文档 [Docker storage drivers](https://docs.docker.com/storage/storagedriver/select-storage-driver/)。

#### Docker 架构

Docker 使用 C/S （客户端/服务器）体系的架构，Docker 客户端与 Docker 守护进程（Dockerd）通信，Docker 守护进程负责构建，运行和分发 Docker 容器。Docker 客户端和守护进程可以在同一个系统上运行，也可以将 Docker 客户端连接到远程 Docker 守护进程。Docker 客户端和守护进程使用 REST API 通过 UNIX 套接字或网络接口进行通信。



## [1.2 安装](https://www.qikqiak.com/k8strain2/)

[要安装 Docker CE（社区）版本，推荐使用 CentOS 7，使用 overlay2 存储驱动程序。](https://www.qikqiak.com/k8strain2/)

### [1.2.1. 卸载旧版本](https://www.qikqiak.com/k8strain2/)

[旧版本的 Docker 叫 docker 或者 docker-engine，现在的版本被称为docker-ce，如果之前安装过，则先卸载掉之前的版本：](https://www.qikqiak.com/k8strain2/)

[$ sudo yum remove docker \](https://www.qikqiak.com/k8strain2/)

[docker-client \](https://www.qikqiak.com/k8strain2/)

[docker-client-latest \](https://www.qikqiak.com/k8strain2/)

[docker-common \](https://www.qikqiak.com/k8strain2/)

[docker-latest \](https://www.qikqiak.com/k8strain2/)

[docker-latest-logrotate \](https://www.qikqiak.com/k8strain2/)

[docker-logrotate \](https://www.qikqiak.com/k8strain2/)

[docker-engine](https://www.qikqiak.com/k8strain2/)

### [1.2.2. 安装 Docker-CE](https://www.qikqiak.com/k8strain2/)

[要安装 Docker 有多种方法，大多数用户会设置 Docker 的存储仓库来进行安装，这种方式可以简化我们的安装和升级，当然也推荐使用这种方式。](https://www.qikqiak.com/k8strain2/)

[但是如果你的服务器无法访问互联网则可以使用手动下载 RPM 安装包进行安装，如果你要使用这种方式安装，可以在页面](https://www.qikqiak.com/k8strain2/)<https://download.docker.com/linux/centos/7/x86_64/stable/Packages/>[下载自己想要安装的 .rpm 文件然后直接安装即可。](https://www.qikqiak.com/k8strain2/)

[我们这里通过在线方式进行安装，首先安装仓库，先安装一些依赖包，其中yum-utils提供了yum-config-manager的一些实用的功能：](https://www.qikqiak.com/k8strain2/)

[$ sudo yum install -y yum-utils](https://www.qikqiak.com/k8strain2/)

[然后用下面的命令添加 stable 版本的 docker 仓库：](https://www.qikqiak.com/k8strain2/)

[$ sudo yum-config-manager \](https://www.qikqiak.com/k8strain2/)

[--add-repo \](https://www.qikqiak.com/k8strain2/)

[https://download.docker.com/linux/centos/docker-ce.repo](https://www.qikqiak.com/k8strain2/)

[添加完成后就可以安装了，如果要安装最新版本，直接执行下面的命令即可：](https://www.qikqiak.com/k8strain2/)

[$ yum install docker-ce docker-ce-cli containerd.io](https://www.qikqiak.com/k8strain2/)

[如果要安装指定的版本，可以通过如下命令来获取可安装的版本：](https://www.qikqiak.com/k8strain2/)

[$ yum list docker-ce --showduplicates | sort -r \* updates: mirrors.tuna.tsinghua.edu.cn](https://www.qikqiak.com/k8strain2/)

[Loading mirror speeds from cached hostfile](https://www.qikqiak.com/k8strain2/)

[Loaded plugins: fastestmirror, langpacks \* extras: mirrors.huaweicloud.com](https://www.qikqiak.com/k8strain2/)

[docker-ce.x86\_64 3:19.03.4-3.el7 docker-ce-stable](https://www.qikqiak.com/k8strain2/)

[......](https://www.qikqiak.com/k8strain2/)

[docker-ce.x86\_64 3:18.09.9-3.el7 docker-ce-stable](https://www.qikqiak.com/k8strain2/)

[docker-ce.x86\_64 3:18.09.8-3.el7 docker-ce-stable](https://www.qikqiak.com/k8strain2/)

[docker-ce.x86\_64 3:18.09.7-3.el7 docker-ce-stable](https://www.qikqiak.com/k8strain2/)

[docker-ce.x86\_64 3:18.09.6-3.el7 docker-ce-stable](https://www.qikqiak.com/k8strain2/)

[......](https://www.qikqiak.com/k8strain2/)

[docker-ce.x86\_64 18.06.0.ce-3.el7 docker-ce-stable](https://www.qikqiak.com/k8strain2/)

[该软件包名称是软件包名称（docker-ce）加上版本字符串（第二列），从第一个冒号（:）开始，直至第一个连字符，并用连字符（-）分隔 ）。 比如这里我们来安装 docker-ce-18.09.9 版本：](https://www.qikqiak.com/k8strain2/)

[$ sudo yum install docker-ce-18.09.9 docker-ce-cli-18.09.9 containerd.io -y](https://www.qikqiak.com/k8strain2/)

[安装完成，但是还没有启动，由于我这里的磁盘空间不大，所以需要更改下 Docker 的根目录，将默认的 /var/lib/docker 更改为 /data/docker 目录，添加如下配置文件：](https://www.qikqiak.com/k8strain2/)

[$ mkdir -p /etc/docker](https://www.qikqiak.com/k8strain2/)

[$ vi /etc/docker/daemon.json](https://www.qikqiak.com/k8strain2/)

[{](https://www.qikqiak.com/k8strain2/)

["registry-mirrors" : [](https://www.qikqiak.com/k8strain2/)

["https://ot2k4d59.mirror.aliyuncs.com/"](https://www.qikqiak.com/k8strain2/)

[],](https://www.qikqiak.com/k8strain2/)

["graph": "/data/docker"](https://www.qikqiak.com/k8strain2/)

[}](https://www.qikqiak.com/k8strain2/)

**[镜像加速器：](https://www.qikqiak.com/k8strain2/)**[使用加速器可以提升获取 Docker 官方镜像的速度，建议注册阿里云帐号，](https://www.qikqiak.com/k8strain2/)**[使用阿里云提供的镜像加速服务，地址：https://cr.console.aliyun.com/cn-beijing/instances/mirrors](https://www.qikqiak.com/k8strain2/)**[。](https://www.qikqiak.com/k8strain2/)

[然后执行下面的命令启动 docker：](https://www.qikqiak.com/k8strain2/)

[# 设置为开机启动](https://www.qikqiak.com/k8strain2/)

[$ systemctl enable docker](https://www.qikqiak.com/k8strain2/)

[$ systemctl daemon-reload](https://www.qikqiak.com/k8strain2/)

[# 启动 docker](https://www.qikqiak.com/k8strain2/)

[$ systemctl start docker](https://www.qikqiak.com/k8strain2/)

[启动成功后，可以通过如下命令查看 Docker 相关信息：](https://www.qikqiak.com/k8strain2/)

[# docker info](https://www.qikqiak.com/k8strain2/)

[Client:](https://www.qikqiak.com/k8strain2/)

[Context: default](https://www.qikqiak.com/k8strain2/)

[Debug Mode: false](https://www.qikqiak.com/k8strain2/)

[Server:](https://www.qikqiak.com/k8strain2/)

[Containers: 44](https://www.qikqiak.com/k8strain2/)

[Running: 16](https://www.qikqiak.com/k8strain2/)

[Paused: 0](https://www.qikqiak.com/k8strain2/)

[Stopped: 28](https://www.qikqiak.com/k8strain2/)

[Images: 104](https://www.qikqiak.com/k8strain2/)

[Server Version: 20.10.5](https://www.qikqiak.com/k8strain2/)

[Storage Driver: overlay2](https://www.qikqiak.com/k8strain2/)

[Backing Filesystem: extfs](https://www.qikqiak.com/k8strain2/)

[Supports d\_type: true](https://www.qikqiak.com/k8strain2/)

[Native Overlay Diff: true](https://www.qikqiak.com/k8strain2/)

[Logging Driver: json-file](https://www.qikqiak.com/k8strain2/)

[Cgroup Driver: cgroupfs](https://www.qikqiak.com/k8strain2/)

[Cgroup Version: 1](https://www.qikqiak.com/k8strain2/)

[Plugins:](https://www.qikqiak.com/k8strain2/)

[Volume: local](https://www.qikqiak.com/k8strain2/)

[Network: bridge host ipvlan macvlan null overlay](https://www.qikqiak.com/k8strain2/)

[Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog](https://www.qikqiak.com/k8strain2/)

[Swarm: inactive](https://www.qikqiak.com/k8strain2/)

[Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc](https://www.qikqiak.com/k8strain2/)

[Default Runtime: runc](https://www.qikqiak.com/k8strain2/)

[Init Binary: docker-init](https://www.qikqiak.com/k8strain2/)

[containerd version: 269548fa27e0089a8b8278fc4fc781d7f65a939b](https://www.qikqiak.com/k8strain2/)

[runc version: 12644e614e25b05da6fd08a38ffa0cfe1903fdec](https://www.qikqiak.com/k8strain2/)

[init version: de40ad0](https://www.qikqiak.com/k8strain2/)

[Security Options:](https://www.qikqiak.com/k8strain2/)

[seccomp](https://www.qikqiak.com/k8strain2/)

[Profile: default](https://www.qikqiak.com/k8strain2/)

[Kernel Version: 5.4.72-microsoft-standard-WSL2](https://www.qikqiak.com/k8strain2/)

[Operating System: Ubuntu 20.04.2 LTS](https://www.qikqiak.com/k8strain2/)

[OSType: linux](https://www.qikqiak.com/k8strain2/)

[Architecture: x86\_64](https://www.qikqiak.com/k8strain2/)

[CPUs: 4](https://www.qikqiak.com/k8strain2/)

[Total Memory: 12.41GiB](https://www.qikqiak.com/k8strain2/)

[Name: SZ-PC-00517](https://www.qikqiak.com/k8strain2/)

[ID: 4NP7:XXEH:PBR6:24NI:XMMW:M4AL:FCZH:VFB4:WYO6:B2ZM:KWCU:4NNO](https://www.qikqiak.com/k8strain2/)

[Docker Root Dir: /var/lib/docker](https://www.qikqiak.com/k8strain2/)

[Debug Mode: false](https://www.qikqiak.com/k8strain2/)

[Registry: https://index.docker.io/v1/](https://www.qikqiak.com/k8strain2/)

[Labels:](https://www.qikqiak.com/k8strain2/)

[Experimental: false](https://www.qikqiak.com/k8strain2/)

[Insecure Registries:](https://www.qikqiak.com/k8strain2/)

[127.0.0.0/8](https://www.qikqiak.com/k8strain2/)

[Live Restore Enabled: false](https://www.qikqiak.com/k8strain2/)

[Product License: Community Engine](https://www.qikqiak.com/k8strain2/)

[可以看到安装的版本是 20.10.5，默认的存储驱动程序是 overlay2，而且 Root Dir 也已经是我们更改后的目录了。](https://www.qikqiak.com/k8strain2/)

### [1.2.3. 测试](https://www.qikqiak.com/k8strain2/)

[Docker 启动成功后，我们可以使用下面的命令来运行一个测试容器：](https://www.qikqiak.com/k8strain2/)

[$ docker run hello-world](https://www.qikqiak.com/k8strain2/)

[Unable to find image 'hello-world:latest' locally](https://www.qikqiak.com/k8strain2/)

[latest: Pulling from library/hello-world1b930d010525: Pull complete](https://www.qikqiak.com/k8strain2/)

[Digest: sha256:c3b4ada4687bbaa170745b3e4dd8ac3f194ca95b2d0518b417fb47e5879d9b5f](https://www.qikqiak.com/k8strain2/)

[Status: Downloaded newer image for hello-world:latest](https://www.qikqiak.com/k8strain2/)

[Hello from Docker!](https://www.qikqiak.com/k8strain2/)

[This message shows that your installation appears to be working correctly.](https://www.qikqiak.com/k8strain2/)

[To generate this message, Docker took the following steps:](https://www.qikqiak.com/k8strain2/)

[1. The Docker client contacted the Docker daemon.](https://www.qikqiak.com/k8strain2/)

[2. The Docker daemon pulled the "hello-world" image from the Docker Hub.](https://www.qikqiak.com/k8strain2/)

[(amd64)](https://www.qikqiak.com/k8strain2/)

[3. The Docker daemon created a new container from that image which runs the](https://www.qikqiak.com/k8strain2/)

[executable that produces the output you are currently reading.](https://www.qikqiak.com/k8strain2/)

[4. The Docker daemon streamed that output to the Docker client, which sent it](https://www.qikqiak.com/k8strain2/)

[to your terminal.](https://www.qikqiak.com/k8strain2/)

[To try something more ambitious, you can run an Ubuntu container with:](https://www.qikqiak.com/k8strain2/)

[$ docker run -it ubuntu bash](https://www.qikqiak.com/k8strain2/)

[Share images, automate workflows, and more with a free Docker ID:](https://www.qikqiak.com/k8strain2/)

[https:](https://www.qikqiak.com/k8strain2/)*[//hub.docker.com/](https://www.qikqiak.com/k8strain2/)*

[For more examples and ideas, visit:](https://www.qikqiak.com/k8strain2/)

[https:](https://www.qikqiak.com/k8strain2/)*[//docs.docker.com/get-started/](https://www.qikqiak.com/k8strain2/)*

[看到如上的一些信息打印出来证明我们的 Docker 就已经安装成功了。](https://www.qikqiak.com/k8strain2/)

## [1.3. 基本操作（镜像和容器的常用操作）](https://www.qikqiak.com/k8strain2/)

[Docker 的基本原理我们已经了解了，也已经安装上了，接下来我们就一起来学习下 Docker 的常用操作，实际上主要就是 Docker CLI 的一些常用命令使用。](https://www.qikqiak.com/k8strain2/)

[](https://www.qikqiak.com/k8strain2/)

### [1.3.1. 镜像操作](https://www.qikqiak.com/k8strain2/)

[之前我们提到过 Docker 官方提供了一个公共的镜像仓库：](https://www.qikqiak.com/k8strain2/)[Docker Hub](https://hub.docker.com/explore/)[，我们就可以从这上面获取镜像，获取镜像的命令：docker pull，格式为：](https://www.qikqiak.com/k8strain2/)

[$ docker pull [选项] [Docker Registry 地址[:端口]/]仓库名[:标签]](https://www.qikqiak.com/k8strain2/)

* [Docker 镜像仓库地址：地址的格式一般是 <域名/IP>[:端口号]，默认地址是 Docker Hub 官方地址。](https://www.qikqiak.com/k8strain2/)
* [仓库名：这里的仓库名是两段式名称，即 <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：](https://www.qikqiak.com/k8strain2/)

[$ docker pull ubuntu:18.04](https://www.qikqiak.com/k8strain2/)

[18.04: Pulling from library/ubuntu](https://www.qikqiak.com/k8strain2/)

[7ddbc47eeb70: Pull complete](https://www.qikqiak.com/k8strain2/)

[c1bbdc448b72: Pull complete](https://www.qikqiak.com/k8strain2/)

[8c3b70e39044: Pull complete](https://www.qikqiak.com/k8strain2/)

[45d437916d57: Pull complete](https://www.qikqiak.com/k8strain2/)

[Digest: sha256:6e9f67fa63b0323e9a1e587fd71c561ba48a034504fb804fd26fd8800039835d](https://www.qikqiak.com/k8strain2/)

[Status: Downloaded newer image for ubuntu:18.04](https://www.qikqiak.com/k8strain2/)

[上面的命令中没有给出 Docker 镜像仓库地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 ubuntu:18.04，因此将会获取官方镜像 library/ubuntu 仓库中标签为 18.04 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。](https://www.qikqiak.com/k8strain2/)

[然后可以使用如下命令查看系统中已有的镜像：](https://www.qikqiak.com/k8strain2/)

[$ docker images](https://www.qikqiak.com/k8strain2/)

[REPOSITORY TAG IMAGE ID CREATED SIZE](https://www.qikqiak.com/k8strain2/)

[ubuntu 18.04 775349758637 19 hours ago 64.2MB](https://www.qikqiak.com/k8strain2/)

[busybox latest 020584afccce 42 hours ago 1.22MB](https://www.qikqiak.com/k8strain2/)

[hello-world latest fce289e99eb9 10 months ago 1.84kB](https://www.qikqiak.com/k8strain2/)

[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。镜像 ID 则是镜像的唯一标识，一个镜像可以对应多个标签。](https://www.qikqiak.com/k8strain2/)

[如果某个镜像不需要了，可以使用如下面命令删除镜像：](https://www.qikqiak.com/k8strain2/)

*[# 根据镜像名或者镜像ID删除都可以](https://www.qikqiak.com/k8strain2/)*

[$ docker rmi -f hello-world](https://www.qikqiak.com/k8strain2/)

[Untagged: hello-world:latest](https://www.qikqiak.com/k8strain2/)

[Untagged: hello-world@sha256:c3b4ada4687bbaa170745b3e4dd8ac3f194ca95b2d0518b417fb47e5879d9b5f](https://www.qikqiak.com/k8strain2/)

[Deleted: sha256:fce289e99eb9bca977dae136fbe2a82b6b7d4c372474c9235adc1741675f587e](https://www.qikqiak.com/k8strain2/)

[还可以给镜像重新打上一个 tag：](https://www.qikqiak.com/k8strain2/)

[$ docker tag nginx nginx:test](https://www.qikqiak.com/k8strain2/)

[另外我们还可以将镜像导出成一个独立的文件：](https://www.qikqiak.com/k8strain2/)

[$ docker save nginx >/tmp/nginx.tar.gz](https://www.qikqiak.com/k8strain2/)

[$ ls -la /tmp/nginx.tar.gz](https://www.qikqiak.com/k8strain2/)

[-rw-r--r--. 1 root root 130066944 Nov 2 02:58 /tmp/nginx.tar.gz](https://www.qikqiak.com/k8strain2/)

[对于无法访问外网的请情况下会经常使用这种方法导出镜像，然后使用 load 命令导入镜像：](https://www.qikqiak.com/k8strain2/)

[$ docker load </tmp/nginx.tar.gz](https://www.qikqiak.com/k8strain2/)

### [1.3.2. 运行容器](https://www.qikqiak.com/k8strain2/)

[有了镜像后，我们就能够以这个镜像为基础运行一个容器。以上面的 ubuntu:18.04 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：](https://www.qikqiak.com/k8strain2/)

[$ docker run -it --rm \](https://www.qikqiak.com/k8strain2/)

[ubuntu:18.04 \](https://www.qikqiak.com/k8strain2/)

[/bin/bash](https://www.qikqiak.com/k8strain2/)

[root@ec125fc290ca:/# cat /etc/os-release](https://www.qikqiak.com/k8strain2/)

[NAME="Ubuntu"](https://www.qikqiak.com/k8strain2/)

[VERSION="18.04.3 LTS (Bionic Beaver)"](https://www.qikqiak.com/k8strain2/)

[ID=ubuntu](https://www.qikqiak.com/k8strain2/)

[ID\_LIKE=debian](https://www.qikqiak.com/k8strain2/)

[PRETTY\_NAME="Ubuntu 18.04.3 LTS"](https://www.qikqiak.com/k8strain2/)

[VERSION\_ID="18.04"](https://www.qikqiak.com/k8strain2/)

[HOME\_URL="https://www.ubuntu.com/"](https://www.qikqiak.com/k8strain2/)

[SUPPORT\_URL="https://help.ubuntu.com/"](https://www.qikqiak.com/k8strain2/)

[BUG\_REPORT\_URL="https://bugs.launchpad.net/ubuntu/"](https://www.qikqiak.com/k8strain2/)

[PRIVACY\_POLICY\_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"](https://www.qikqiak.com/k8strain2/)

[VERSION\_CODENAME=bionic](https://www.qikqiak.com/k8strain2/)

[UBUNTU\_CODENAME=bionic](https://www.qikqiak.com/k8strain2/)

[docker run就是运行容器的命令，我们这里简要的说明一下上面用到的参数:](https://www.qikqiak.com/k8strain2/)

* **[-it：](https://www.qikqiak.com/k8strain2/)**[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。](https://www.qikqiak.com/k8strain2/)
* **[--rm：](https://www.qikqiak.com/k8strain2/)**[这个参数是说容器退出后随之将其删除。默认情况下，退出的容器并不会立即删除，除非手动 docker rm。我们这里只是执行个命令，看看结果，不需要保留结果，因此使用--rm可以避免浪费空间。](https://www.qikqiak.com/k8strain2/)
* **[ubuntu:18.04：](https://www.qikqiak.com/k8strain2/)**[这是指用 ubuntu:18.04 镜像为基础来启动容器。](https://www.qikqiak.com/k8strain2/)
* **[bash：](https://www.qikqiak.com/k8strain2/)**[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。](https://www.qikqiak.com/k8strain2/)

[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 16.04.4 LTS 系统。最后我们通过 exit 退出了这个容器。](https://www.qikqiak.com/k8strain2/)

[当利用docker run来创建容器时，Docker 在后台运行的流程如下所示：](https://www.qikqiak.com/k8strain2/)

* [检查本地是否存在指定的镜像，不存在就从公有仓库下载](https://www.qikqiak.com/k8strain2/)
* [利用镜像创建并启动一个容器](https://www.qikqiak.com/k8strain2/)
* [分配一个文件系统，并在只读的镜像层外面挂载一层可读写层](https://www.qikqiak.com/k8strain2/)
* [从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去](https://www.qikqiak.com/k8strain2/)
* [从地址池配置一个 ip 地址给容器](https://www.qikqiak.com/k8strain2/)
* [执行用户指定的应用程序](https://www.qikqiak.com/k8strain2/)
* [执行完毕后容器被终止](https://www.qikqiak.com/k8strain2/)

[另外需要注意的是容器管理的核心是容器执行的应用程序这个进程，所以如果这个进程不是常驻前台的话则执行后容器就会退出了，比如上面我们是执行的 /bin/bash 这个程序，这个程序会常驻前台，所以容器会一直存在，而且这个这个程序在容器中的进程ID=1：](https://www.qikqiak.com/k8strain2/)

[root@6bc39e8cd11c:/](https://www.qikqiak.com/k8strain2/)*[# ps](https://www.qikqiak.com/k8strain2/)*

[PID TTY TIME CMD](https://www.qikqiak.com/k8strain2/)

[1 pts/0 00:00:00 bash](https://www.qikqiak.com/k8strain2/)

[11 pts/0 00:00:00 ps](https://www.qikqiak.com/k8strain2/)

[但是如果我们运行一个普通的命令呢：](https://www.qikqiak.com/k8strain2/)

[$ docker run -it ubuntu:18.04 ls](https://www.qikqiak.com/k8strain2/)

[bin dev home lib64 mnt proc run srv tmp var](https://www.qikqiak.com/k8strain2/)

[boot etc lib media opt root sbin sys usr](https://www.qikqiak.com/k8strain2/)

[我们可以看到运行后就容器就直接退出了，这点非常重要，所以如果要在容器中执行 nginx 程序的话要记住不要用 daemon 模式了，因为执行后就退出到后台去了，Docker 就没办法管理了，就会退出容器了。](https://www.qikqiak.com/k8strain2/)

#### [1.3.2.1. 列出容器](https://www.qikqiak.com/k8strain2/)

[如果要查看当前系统中已经运行的容器，可以用如下命令：](https://www.qikqiak.com/k8strain2/)

[$ docker ps](https://www.qikqiak.com/k8strain2/)

[如果把已经退出的容器也列出来可以加上 -a 参数：](https://www.qikqiak.com/k8strain2/)

[$ docker ps -a](https://www.qikqiak.com/k8strain2/)

[CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES](https://www.qikqiak.com/k8strain2/)

[2275424275b6 ubuntu:18.04 "ls" 3 minutes ago Exited (0) 3 minutes ago ecstatic\_gates](https://www.qikqiak.com/k8strain2/)

[6e4a54862340 fce289e99eb9 "/hello" About an hour ago Exited (0) About an hour ago jovial\_khayyam](https://www.qikqiak.com/k8strain2/)

#### [1.3.2.2. 删除容器](https://www.qikqiak.com/k8strain2/)

[如果要删除或强制删除一个容器（包括已退出的）则可以使用如下命令：](https://www.qikqiak.com/k8strain2/)

[# 根据容器ID强制删除容器](https://www.qikqiak.com/k8strain2/)

[$ docker rm -f 2275424275b6](https://www.qikqiak.com/k8strain2/)

#### [1.3.2.3. 后台运行](https://www.qikqiak.com/k8strain2/)

[更多的时候，我们需要让 Docker 在后台运行而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加-d参数来实现。如果不使用-d参数运行容器：](https://www.qikqiak.com/k8strain2/)

[$ docker run ubuntu:18.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[容器会把输出的结果 (STDOUT) 打印到宿主机上面。如果使用了-d参数运行容器：](https://www.qikqiak.com/k8strain2/)

[$ docker run -d ubuntu:18.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"501f4d9538a0b01f0ac422089258ad71fa88c016f2662c1120c1499b5fbc930f](https://www.qikqiak.com/k8strain2/)

[此时容器会在后台运行并不会把输出的结果 (STDOUT) 打印到宿主机上面(输出结果可以用 docker logs 查看)：](https://www.qikqiak.com/k8strain2/)

[# docker logs -f [container ID or NAMES]$ docker logs -f 501f4d9538a0b01f0ac422089258ad71fa88c016f2662c1120c1499b5fbc930f](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[hello world](https://www.qikqiak.com/k8strain2/)

[注：容器是否会长久运行，是和 docker run 指定的命令有关，和 -d 参数无关。](https://www.qikqiak.com/k8strain2/)

[使用 -d 参数启动后会返回一个唯一的容器 id，当然也可以通过docker ps命令来查看容器信息。](https://www.qikqiak.com/k8strain2/)

#### [1.3.2.4. 终止容器](https://www.qikqiak.com/k8strain2/)

[另外我们可以使用docker stop [container ID or NAMES]来终止一个运行中的容器。此外，当 Docker 容器中指定的应用终结时，容器也自动终止。例如前面只启动了一个终端的容器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止。终止状态的容器可以用docker ps -a 命令看到：](https://www.qikqiak.com/k8strain2/)

[$ docker ps](https://www.qikqiak.com/k8strain2/)

[CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES501f4d9538a0 ubuntu:18.04 "/bin/sh -c 'while t…" About a minute ago Up About a minute nervous\_ganguly](https://www.qikqiak.com/k8strain2/)

[$ docker stop 501f4d9538a0501f4d9538a0](https://www.qikqiak.com/k8strain2/)

[$ docker ps](https://www.qikqiak.com/k8strain2/)

[CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES](https://www.qikqiak.com/k8strain2/)

[同样可以用docker start [container ID or NAMES] 命令来启动一个终止的容器：](https://www.qikqiak.com/k8strain2/)

[$ docker start 501f4d9538a0501f4d9538a0](https://www.qikqiak.com/k8strain2/)

[$ docker ps](https://www.qikqiak.com/k8strain2/)

[CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES501f4d9538a0 ubuntu:18.04 "/bin/sh -c 'while t…" 5 minutes ago Up 2 seconds nervous\_ganguly](https://www.qikqiak.com/k8strain2/)

### [1.3.3. 网络](https://www.qikqiak.com/k8strain2/)

### [1.3.4. 端口暴露](https://www.qikqiak.com/k8strain2/)

[Docker 容器更多情况下是用来运行 Web 应用的，所以要如何访问到容器中的 Web 服务呢？比如我们现在运行一个 nginx 容器服务：](https://www.qikqiak.com/k8strain2/)

[$ docker run --name webserver -d nginx](https://www.qikqiak.com/k8strain2/)

[Unable to find image 'nginx:latest' locally](https://www.qikqiak.com/k8strain2/)

[latest: Pulling from library/nginx](https://www.qikqiak.com/k8strain2/)

[8d691f585fa8: Pull complete](https://www.qikqiak.com/k8strain2/)

[5b07f4e08ad0: Pull complete](https://www.qikqiak.com/k8strain2/)

[abc291867bca: Pull complete](https://www.qikqiak.com/k8strain2/)

[Digest: sha256:922c815aa4df050d4df476e92daed4231f466acc8ee90e0e774951b0fd7195a4](https://www.qikqiak.com/k8strain2/)

[Status: Downloaded newer image for nginx:lateste8b034c01f4024162cefc45006738fce85b3fa1b717a6ff24520c0fcfabaf5b6](https://www.qikqiak.com/k8strain2/)

[nginx 镜像没有指定 tag 标签，则默认就是拉取nginx:latest镜像；其中--name参数指定容器的名称，不指定则是随机的容器名，运行成功后可以通过docker ps命令查看容器信息：](https://www.qikqiak.com/k8strain2/)

[$ docker ps](https://www.qikqiak.com/k8strain2/)

[CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES](https://www.qikqiak.com/k8strain2/)

[e8b034c01f40 nginx "nginx -g 'daemon of…" About a minute ago Up About a minute 80/tcp webserver](https://www.qikqiak.com/k8strain2/)

[但是我们要怎么去访问这个 nginx 服务呢？实际上在我们启动容器的时候，Docker 就会为我们的容器分配一个 IP 地址，我们可以通过如下命令来获取容器的 IP 地址：](https://www.qikqiak.com/k8strain2/)

[$ docker inspect webserver |grep IPAddress](https://www.qikqiak.com/k8strain2/)

["SecondaryIPAddresses": null,](https://www.qikqiak.com/k8strain2/)

["IPAddress": "172.17.0.4",](https://www.qikqiak.com/k8strain2/)

["IPAddress": "172.17.0.4",](https://www.qikqiak.com/k8strain2/)

[其中的 172.17.0.4就是容器 webserver 的地址，这个时候我们可以通过该地址访问到 nginx 服务：](https://www.qikqiak.com/k8strain2/)

[但是这样不够方便，因为启动容器的代价很小，所以容器的 IP 这些经常变动，是否能够通过宿主机的方式去访问呢？实际上是可以的，我们可以通过如下命令重新启动一个新的容器：](https://www.qikqiak.com/k8strain2/)

[$ docker rm -f webserver](https://www.qikqiak.com/k8strain2/)

[$ docker run --name webserver -d -p 8080:80 nginx](https://www.qikqiak.com/k8strain2/)

[我们在启动容器的时候添加了一个新的参数-p 8080:80，这个参数的意思是将宿主机的 8080 端口和容器的 80 端口进行绑定，这样我们就可以通过宿主机的 8080 端口来访问容器服务了。](https://www.qikqiak.com/k8strain2/)

[$ docker ps](https://www.qikqiak.com/k8strain2/)

[CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES89e105d56215 nginx "nginx -g 'daemon of…" About a minute ago Up About a minute 0.0.0.0:8080->80/tcp webserver](https://www.qikqiak.com/k8strain2/)

##### [1.3.4.1. Bridge 模式](https://www.qikqiak.com/k8strain2/)

[当 Docker 进程启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。从 docker0 子网中分配一个 IP 给容器使用，并设置 docker0 的 IP 地址为容器的默认网关。在主机上创建一对虚拟网卡veth pair设备，Docker 将 veth pair 设备的一端放在新创建的容器中，并命名为 eth0（容器的网卡），另一端放在主机中，以vethxxx这样类似的名字命名，并将这个网络设备加入到 docker0 网桥中。可以通过brctl show命令查看：](https://www.qikqiak.com/k8strain2/)

[$ brctl show](https://www.qikqiak.com/k8strain2/)

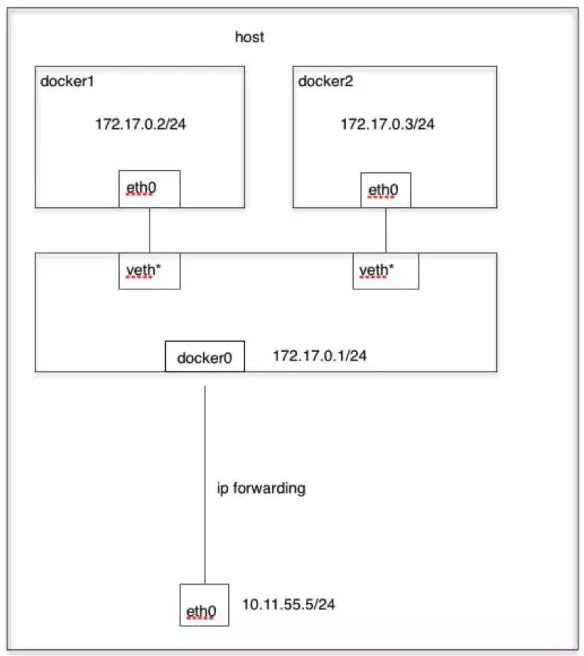
[bridge name bridge id STP enabled interfaces](https://www.qikqiak.com/k8strain2/)

[docker0 8000.024286df8f39 no veth1040b0a](https://www.qikqiak.com/k8strain2/)

[veth5a2ba56](https://www.qikqiak.com/k8strain2/)

[veth7aa7e71](https://www.qikqiak.com/k8strain2/)

[bridge 模式是 docker 的默认网络模式，使用docker run -p时，实际上是通过 iptables 做了DNAT规则，实现端口转发功能。可以使用iptables -t nat -vnL查看。bridge模式如下图所示：​​](https://www.qikqiak.com/k8strain2/)

[](https://www.qikqiak.com/k8strain2/)

[比如现在我们运行一个 busybox 容器：](https://www.qikqiak.com/k8strain2/)

[$ docker run -tid --net=bridge --name docker\_bri busybox top](https://www.qikqiak.com/k8strain2/)

[$ brctl show](https://www.qikqiak.com/k8strain2/)

[brctl show](https://www.qikqiak.com/k8strain2/)

[bridge name bridge id STP enabled interfaces](https://www.qikqiak.com/k8strain2/)

[docker0 8000.024286df8f39 no veth1040b0a](https://www.qikqiak.com/k8strain2/)

[veth27bc18a](https://www.qikqiak.com/k8strain2/)

[veth5a2ba56](https://www.qikqiak.com/k8strain2/)

[veth7aa7e71](https://www.qikqiak.com/k8strain2/)

[然后进入到容器内部去查看网络情况，这里我们需要使用到一个新的命令docker exec，用来进入容器内部，要记住我们要进行终端交互，所以要带上-it两个参数：](https://www.qikqiak.com/k8strain2/)

[$ docker exec -it docker\_bri /bin/sh](https://www.qikqiak.com/k8strain2/)

[/](https://www.qikqiak.com/k8strain2/) *[# ifconfig -a](https://www.qikqiak.com/k8strain2/)*

[eth0 Link encap:Ethernet HWaddr 02:42:AC:11:00:05](https://www.qikqiak.com/k8strain2/)

[inet addr:172.17.0.5 Bcast:172.17.255.255 Mask:255.255.0.0](https://www.qikqiak.com/k8strain2/)

[UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1](https://www.qikqiak.com/k8strain2/)

[RX packets:8 errors:0 dropped:0 overruns:0 frame:0](https://www.qikqiak.com/k8strain2/)

[TX packets:0 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0](https://www.qikqiak.com/k8strain2/)

[RX bytes:648 (648.0 B) TX bytes:0 (0.0 B)](https://www.qikqiak.com/k8strain2/)

[lo Link encap:Local Loopback](https://www.qikqiak.com/k8strain2/)

[inet addr:127.0.0.1 Mask:255.0.0.0](https://www.qikqiak.com/k8strain2/)

[UP LOOPBACK RUNNING MTU:65536 Metric:1](https://www.qikqiak.com/k8strain2/)

[RX packets:0 errors:0 dropped:0 overruns:0 frame:0](https://www.qikqiak.com/k8strain2/)

[TX packets:0 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0](https://www.qikqiak.com/k8strain2/)

[RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)](https://www.qikqiak.com/k8strain2/)

[/](https://www.qikqiak.com/k8strain2/) *[# route -n](https://www.qikqiak.com/k8strain2/)*

[Kernel IP routing table](https://www.qikqiak.com/k8strain2/)

[Destination Gateway Genmask Flags Metric Ref Use Iface](https://www.qikqiak.com/k8strain2/)

[0.0.0.0 172.17.0.1 0.0.0.0 UG 0 0 0 eth0](https://www.qikqiak.com/k8strain2/)

[172.17.0.0 0.0.0.0 255.255.0.0 U 0 0 0 eth0](https://www.qikqiak.com/k8strain2/)

[可以通过ip link show命令查看到对应的 veth pair 对名称。](https://www.qikqiak.com/k8strain2/)

[通过上面的命令可以验证我们前面提到的 bridge 模式原理。](https://www.qikqiak.com/k8strain2/)

###### [自定义网络](https://www.qikqiak.com/k8strain2/)

[另外我们可以通过自定义的 Docker 网络来连接多个容器，而不是使用--link命令，比如现在我们有一个新的容器想要和上面的 docker\_bri 容器建立互连关系，之前我们可以使用 --link 命令：](https://www.qikqiak.com/k8strain2/)

[$ docker run -tid --link docker\_bri --name docker\_bri1 busybox top2cba17dd1326c2c82d8fa415588a0169e7291d1a44d2accaff25d50216830777](https://www.qikqiak.com/k8strain2/)

[$ docker exec -it docker\_bri1 /bin/sh](https://www.qikqiak.com/k8strain2/)

[/](https://www.qikqiak.com/k8strain2/) *[# ping docker\_bri](https://www.qikqiak.com/k8strain2/)*

[PING docker\_bri (172.17.0.5): 56 data bytes](https://www.qikqiak.com/k8strain2/)

[64 bytes from 172.17.0.5: seq=0 ttl=64 time=0.194 ms](https://www.qikqiak.com/k8strain2/)

[64 bytes from 172.17.0.5: seq=1 ttl=64 time=0.156 ms](https://www.qikqiak.com/k8strain2/)

[我们可以看到在新创建的容器上可以访问到我们连接的容器，但是反过来却不行了，因为--link是单方面的：](https://www.qikqiak.com/k8strain2/)

[$ docker exec -it docker\_bri /bin/sh](https://www.qikqiak.com/k8strain2/)

[/](https://www.qikqiak.com/k8strain2/) *[# ping docker\_bri1](https://www.qikqiak.com/k8strain2/)*

[ping: bad address 'docker\_bri1'](https://www.qikqiak.com/k8strain2/)

[/](https://www.qikqiak.com/k8strain2/) *[#](https://www.qikqiak.com/k8strain2/)*

[这个时候我们可以通过自定义网络的方式来实现互联互通，首先创建一个自定义的网络：](https://www.qikqiak.com/k8strain2/)

[$ docker network create -d bridge my-net](https://www.qikqiak.com/k8strain2/)

[然后我们使用自定义的网络运行一个容器：](https://www.qikqiak.com/k8strain2/)

[$ docker run -it --rm --name busybox1 --network my-net busybox sh](https://www.qikqiak.com/k8strain2/)

[打开终端再运行一个容器：](https://www.qikqiak.com/k8strain2/)

[$ docker run -it --rm --name busybox2 --network my-net busybox sh](https://www.qikqiak.com/k8strain2/)

[然后我们通过 ping 来证明 busybox1 容器和 busybox2 容器建立了互联关系。 在 busybox1 容器输入以下命令：](https://www.qikqiak.com/k8strain2/)

[/ # ping busybox2](https://www.qikqiak.com/k8strain2/)

[PING busybox2 (172.19.0.3): 56 data bytes](https://www.qikqiak.com/k8strain2/)

[64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.072 ms](https://www.qikqiak.com/k8strain2/)

[64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.118 ms](https://www.qikqiak.com/k8strain2/)

[用 ping 来测试连接 busybox2 容器，它会解析成 172.19.0.3。 同理在 busybox2 容器执行 ping busybox1，也会成功连接到：](https://www.qikqiak.com/k8strain2/)

[/ # ping busybox1](https://www.qikqiak.com/k8strain2/)

[PING busybox1 (172.19.0.2): 56 data bytes](https://www.qikqiak.com/k8strain2/)

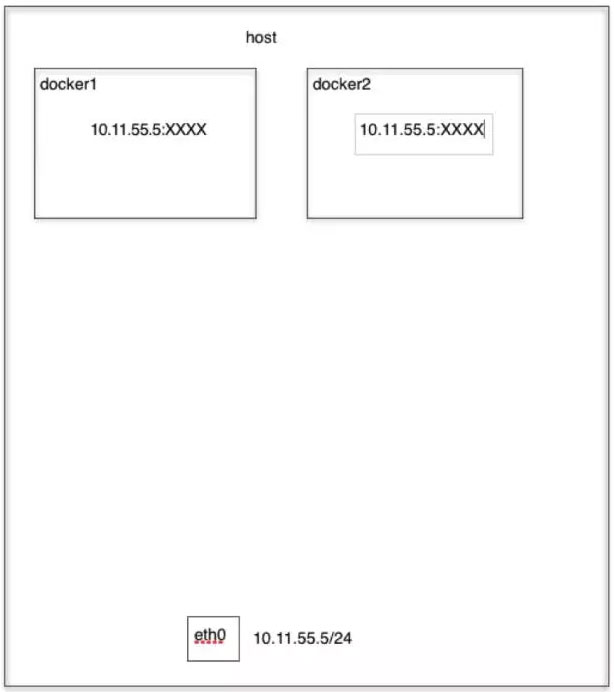
[64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.064 ms](https://www.qikqiak.com/k8strain2/)

[64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.143 ms](https://www.qikqiak.com/k8strain2/)

[这样，busybox1 容器和 busybox2 容器建立了互联关系，如果你有多个容器之间需要互相连接，推荐使用后面的 Docker Compose。](https://www.qikqiak.com/k8strain2/)

##### [1.3.4.2. Host 模式](https://www.qikqiak.com/k8strain2/)

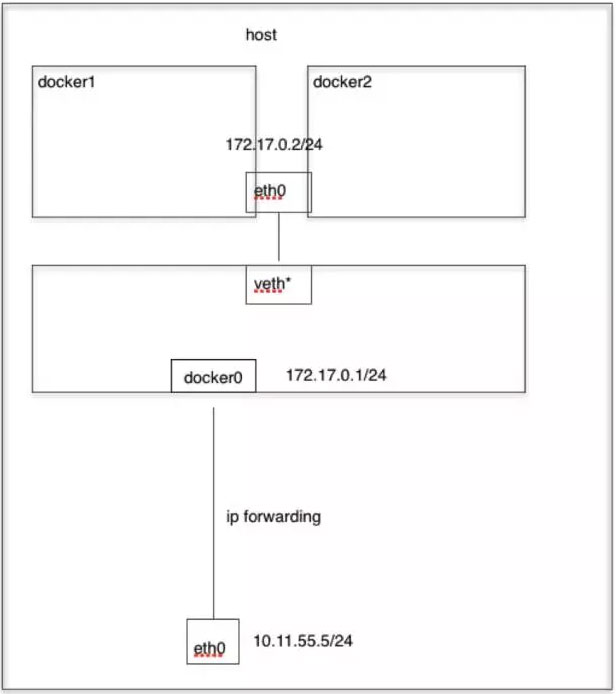
[如果启动容器的时候使用 host 模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个 Network Namespace。容器将不会虚拟出自己的网卡，配置自己的 IP 等，而是使用宿主机的 IP 和端口。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。 Host模式如下图所示：](https://www.qikqiak.com/k8strain2/)

[​​](https://www.qikqiak.com/k8strain2/)

[使用 host 模式也很简单，只需要在运行容器的时候指定 --net=host 即可。](https://www.qikqiak.com/k8strain2/)

##### [1.3.4.3. Container 模式](https://www.qikqiak.com/k8strain2/)

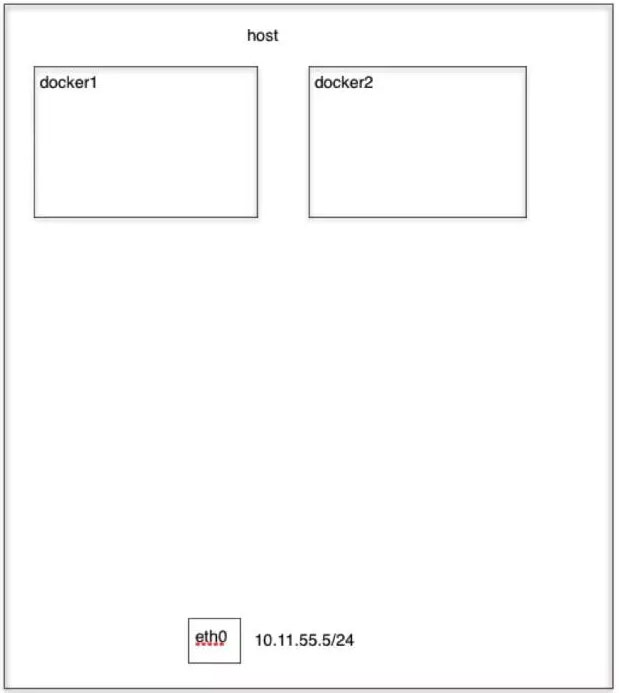
[这个模式指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。 Container 模式如下图所示：](https://www.qikqiak.com/k8strain2/)

[](https://www.qikqiak.com/k8strain2/)

[在运行容器的时候指定 --net=container:目标容器名 即可。实际上我们后面要学习的 Kubernetes 里面的 Pod 中容器之间就是通过 Container 模式链接到 pause 容器上面的，所以容器直接可以通过 localhost 来进行访问。](https://www.qikqiak.com/k8strain2/)

##### [1.3.4.4. None 模式](https://www.qikqiak.com/k8strain2/)

[使用 none模式，Docker 容器拥有自己的 Network Namespace，但是并不为Docker 容器进行任何网络配置。也就是说这个 Docker 容器没有网卡、IP、路由等信息。需要我们自己为 Docker 容器添加网卡、配置 IP 等。 None模式示意图如下所示：](https://www.qikqiak.com/k8strain2/)

[](https://www.qikqiak.com/k8strain2/)

[选择这种模式，一般是用户对网络有自己特殊的需求，不希望 docker 预设置太多的东西。](https://www.qikqiak.com/k8strain2/)

### [1.3.5. 数据共享与持久化](https://www.qikqiak.com/k8strain2/)

[接下来介绍如何在 Docker 内部以及容器之间管理数据，在容器中管理数据主要有两种方式：](https://www.qikqiak.com/k8strain2/)

* [数据卷（Data Volumes）](https://www.qikqiak.com/k8strain2/)
* [挂载主机目录 (Bind mounts)](https://www.qikqiak.com/k8strain2/)

##### [1.3.5.1. 数据卷（Data Volumes）](https://www.qikqiak.com/k8strain2/)

[数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：](https://www.qikqiak.com/k8strain2/)

* [数据卷可以在容器之间共享和重用](https://www.qikqiak.com/k8strain2/)
* [对数据卷的修改会立马生效](https://www.qikqiak.com/k8strain2/)
* [对数据卷的更新，不会影响镜像](https://www.qikqiak.com/k8strain2/)
* [数据卷默认会一直存在，即使容器被删除](https://www.qikqiak.com/k8strain2/)

**[数据卷：](https://www.qikqiak.com/k8strain2/)**[类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，显示的是挂载的 数据卷。](https://www.qikqiak.com/k8strain2/)

[可以通过如下命令创建一个数据卷：](https://www.qikqiak.com/k8strain2/)

[$ docker volume create my-vol](https://www.qikqiak.com/k8strain2/)

[查看数据卷：](https://www.qikqiak.com/k8strain2/)

[$ docker volume ls](https://www.qikqiak.com/k8strain2/)

[local my-vol](https://www.qikqiak.com/k8strain2/)

[同样可以通过 inspect 命令查看数据卷详细信息：](https://www.qikqiak.com/k8strain2/)

[$ docker volume inspect my-vol](https://www.qikqiak.com/k8strain2/)

[[](https://www.qikqiak.com/k8strain2/)

[{](https://www.qikqiak.com/k8strain2/)

["CreatedAt": "2019-11-02T03:08:02+08:00",](https://www.qikqiak.com/k8strain2/)

["Driver": "local",](https://www.qikqiak.com/k8strain2/)

["Labels": {},](https://www.qikqiak.com/k8strain2/)

["Mountpoint": "/data/docker/volumes/my-vol/\_data",](https://www.qikqiak.com/k8strain2/)

["Name": "my-vol",](https://www.qikqiak.com/k8strain2/)

["Options": {},](https://www.qikqiak.com/k8strain2/)

["Scope": "local"](https://www.qikqiak.com/k8strain2/)

[}]](https://www.qikqiak.com/k8strain2/)

[启动一个挂载数据卷的容器：在用docker run命令的时候，使用--mount或者-v标记来将数据卷挂载到容器里。下面创建一个名为 web 的容器，并加载一个数据卷到容器的 /usr/share/nginx/html 目录：](https://www.qikqiak.com/k8strain2/)

[# -v my-vol:/xxxx](https://www.qikqiak.com/k8strain2/)

[# --mount source=my-vol,target=/xxxx](https://www.qikqiak.com/k8strain2/)

[$ docker run -d -p 8080:80 --name web -v my-vol:/usr/share/nginx/html nginx](https://www.qikqiak.com/k8strain2/)

[运行完成后，可以查看数据卷目录下面已经有文件了：](https://www.qikqiak.com/k8strain2/)

[$ ls /data/docker/volumes/my-vol/\_data/](https://www.qikqiak.com/k8strain2/)

[50x.html index.html](https://www.qikqiak.com/k8strain2/)

[这个时候我们可以通过localhost:8080访问容器服务：](https://www.qikqiak.com/k8strain2/)

[如果我们把数据下面的 index.html 文件内容变更下：](https://www.qikqiak.com/k8strain2/)

[$ echo "Hello Docker" > /data/docker/volumes/my-vol/\_data/index.html](https://www.qikqiak.com/k8strain2/)

[这个时候重新访问就可以看到内容已经变化了：](https://www.qikqiak.com/k8strain2/)

[$ curl http:](https://www.qikqiak.com/k8strain2/)*[//localhost:8080](https://www.qikqiak.com/k8strain2/)*

[Hello Docker](https://www.qikqiak.com/k8strain2/)

[数据卷是被设计用来持久化数据的，它的生命周期独立于容器，Docker 不会在容器被删除后自动删除 数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用docker rm -v这个命令。 无主的数据卷可能会占据很多空间，要清理请使用以下命令：](https://www.qikqiak.com/k8strain2/)

[$ docker volume prune](https://www.qikqiak.com/k8strain2/)

##### [1.3.5.2. 挂载主机目录（Bind Mount）](https://www.qikqiak.com/k8strain2/)

[Docker 同样支持把宿主机上的目录挂载到容器中，同样可以使用 -v 或者 --mount 参数来进行挂载，如下所示，把宿主机的 /tmp 目录挂载到一个容器中：](https://www.qikqiak.com/k8strain2/)

[$ ls /tmp/](https://www.qikqiak.com/k8strain2/)

[nginx.tar.gz yum\_save\_tx.2019-11-01.23-56.ux9w\_4.yumtx](https://www.qikqiak.com/k8strain2/)

[$ ls /tmp/nginx.tar.gz](https://www.qikqiak.com/k8strain2/)

[/tmp/nginx.tar.gz](https://www.qikqiak.com/k8strain2/)

[$ docker run -it -v /tmp:/usr/tmp busybox /bin/sh](https://www.qikqiak.com/k8strain2/)

[/](https://www.qikqiak.com/k8strain2/) *[# ls /usr/tmp/](https://www.qikqiak.com/k8strain2/)*

[nginx.tar.gz yum\_save\_tx.2019-11-01.23-56.ux9w\_4.yumtx](https://www.qikqiak.com/k8strain2/)

[通过-v参数，冒号前为宿主机目录，必须为绝对路径，冒号后为容器内挂载的路径。这样容器内就可以共享宿主机里的文件了。](https://www.qikqiak.com/k8strain2/)

**[挂载权限：](https://www.qikqiak.com/k8strain2/)**[默认挂载的路径权限为读写。如果指定为只读可以用：ro，如：-v /tmp:/usr/tmp:ro。](https://www.qikqiak.com/k8strain2/)

[– 容器目录不可以为相对路径](https://www.qikqiak.com/k8strain2/)

[– 宿主机目录如果不存在，则会自动生成](https://www.qikqiak.com/k8strain2/)

[– 挂载宿主机已存在目录后，在容器内对其进行操作，报“Permission denied”。可通过两种方式解决：](https://www.qikqiak.com/k8strain2/)

[\* 1> 关闭selinux。](https://www.qikqiak.com/k8strain2/)

[临时关闭：`# setenforce 0`](https://www.qikqiak.com/k8strain2/)

[永久关闭：修改`/etc/sysconfig/selinux`文件，将 SELINUX 的值设置为disabled。](https://www.qikqiak.com/k8strain2/)

[\* 2> 以特权方式启动容器](https://www.qikqiak.com/k8strain2/)

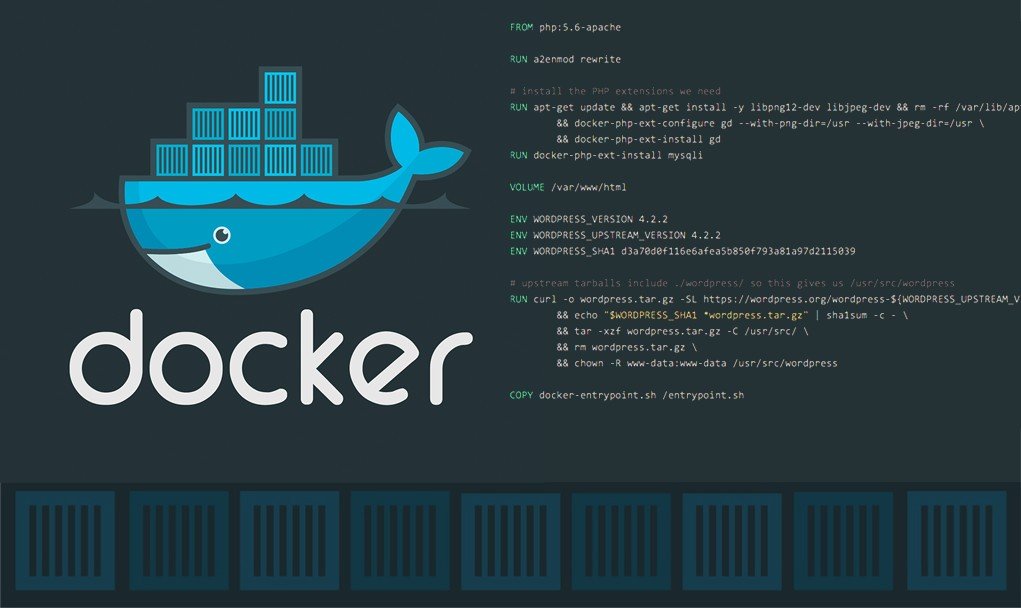
[指定`--privileged`参数，如：](https://www.qikqiak.com/k8strain2/)

[`# docker run -it --privileged=true -v /test:/soft centos /bin/bash`](https://www.qikqiak.com/k8strain2/)

[bind mount 和 volume 其实都是利用宿主机的文件系统，不同之处在于 volume 是 docker 自身管理的目录中的子目录，所以不存在权限引发的挂载的问题，并且目录路径是 docker 自身管理的，所以也不需要在不同的服务器上指定不同的路径，你不需要关心路径。它们之间的主要区别有如下几点：](https://www.qikqiak.com/k8strain2/)

* [volume 会引起 docker 目录膨胀，因为既要存镜像，又要存 volume，最好不要放在系统盘，将 docker 的安装目录配置到其他更大的挂载盘](https://www.qikqiak.com/k8strain2/)
* [两者有一个不同的行为：](https://www.qikqiak.com/k8strain2/)**[当容器外的对应目录是空的，volume 会先将容器内的内容拷贝到容器外目录，而 mount 会将外部的目录覆盖容器内部目录](https://www.qikqiak.com/k8strain2/)**
* [volume 还有一个不如 bind mount 的地方，不能直接挂载文件，例如挂载 nginx 容器的配置文件：nginx.conf](https://www.qikqiak.com/k8strain2/)

## **1.4. Dockerfile（使用 Dockerfile 定制镜像）**



镜像的定制实际上就是定制镜像的每一层所添加的配置、文件等信息，实际上当我们在一个容器中添加或者修改了一些文件后，我们可以通过docker commit命令来生成一个新的镜像，但是这个方法不够直观，没办法追溯我们镜像里面到底有哪些内容，所以实际定制镜像的过程我们很少采用这种方式。而是使用一个名为 Dockerfile 的文本文件来进行镜像定制，我们可以把镜像的每一层修改、安装、构建、操作的命令都写入到这个文件中，一行就对应镜像的一层，这种方法显然要更高级，因为我们有一个文件来直观反映我们的镜像内容，还可以作为版本记录进行跟踪。

### 1.4.1. 定制镜像

以我们之前的 nginx 镜像为例，我们现在来定制一个 nginx 镜像，要求默认的应用页面访问的内容是 Hello Docker。

首先我们新建一个目录 testnginx，然后在该目录下面新建一个名为Dockerfile的空白文本文件：

$ mkdir -p testnginx && cd testnginx && touch Dockerfile

然后向Dockerfile文件中添加如下内容：

FROM nginx

RUN echo 'Hello Docker!' > /user/share/nginx/html/index.html

这个文件很简单，一共就两行，其中包含两条指令，FROM 和 RUN。

#### 1.4.1.1. FROM 指令

定制镜像，那么肯定是一个镜像为基础，在其基础上进行定制，而 FROM 这个指令就是来指定基础镜像的，所以在一个 Dockerfile 文件中 FROM 指令是必备的，并且必须是第一条指令。

在 [Docker Hub](https://hub.docker.com/) 上有非常多的高质量的官方镜像，有一些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。

如果没有找到合适的基础镜像，则可以使用官方提供的一些更为基础的操作系统镜像，比如 ubuntu、debian、centos、alpine 等，这些基础镜像为我们提供了更大的扩展空间，就类似于平时我们在操作系统上面部署自己的服务一样的操作。

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，这个镜像是一个虚拟的镜像，并不实际存在，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。有的同学可能感觉很奇怪，没有任何基础镜像，我怎么去执行我的程序呢，其实对于 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，这也是为什么有很多观点认为 Go 是特别适合容器微服务架构的语言的原因之一。

#### 1.4.1.2. RUN 指令

RUN 指令是用来执行命令行命令的。由于命令行的强大能力，所以 RUN 指令是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 Dockerfile 中的 RUN 指令就是这种格式：

RUN echo 'Hello, Docker!' > /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 wget

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

*# 其他指令...*

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 就会新建立一层，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。这是完全没有必要的，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。

**UnionFS 层限制：**UnionFS 实际上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 wget' \

&& 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

*# 其他指令...*

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。所以我们这里没有使用很多个 RUN 指令来对应不同的命令，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下载的软件，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。

#### 1.4.1.3. WORKDIR 指令

WORKDIR 指令设置 Dockerfile 中的任何 RUN，CMD，ENTRPOINT，COPY 和 ADD 指令的工作目录。如果 WORKDIR 指定的目录不存在，即使随后的指令没有用到这个目录，都会创建该目录。

格式： WORKDIR /path/to/workdir

为了清晰性和可靠性，你应该总是在 WORKDIR 中使用绝对路径，而且单个 Dockerfile 可以使用多次WORKDIR。另外，我们应该使用 WORKDIR 来替代类似于 RUN cd ... && do-something 的指令，后者难以阅读、排错和维护。

#### 1.4.1.4. ADD & COPY 指令

Dockerfile 中的 COPY 指令和 ADD 指令都可以将主机上的资源复制或加入到容器镜像中，都是在构建镜像的过程中完成的。

COPY 指令和 ADD 指令的唯一区别在于**是否支持从远程 URL 获取资源**。COPY 指令只能从执行docker build所在的主机上读取资源并复制到镜像中。而 ADD 指令还支持通过 URL 从远程服务器读取资源并复制到镜像中。

一般来说满足同等功能的情况下，推荐使用COPY指令。ADD 指令更擅长读取本地 tar 文件并解压缩。

##### COPY 指令

COPY 指令能够将构建命令所在的主机本地的文件或目录，复制到镜像文件系统。COPY 指令同样也支持 exec 和 shell 两种格式：

* exec 格式用法：COPY ["<src>",... "<dest>"]，特别适合路径中带有空格的情况。
* shell 格式用法：COPY <src>... <dest>

##### ADD 指令

ADD 指令不仅能够将构建命令所在的主机本地的文件或目录，而且能够将远程 URL 所对应的文件或目录，作为资源复制到镜像文件系统。所以，可以认为 ADD 是增强版的 COPY，支持将远程 URL 的资源加入到镜像的文件系统。同样也支持 exec 和 shell 两种格式用法：   \* exec 格式用法：ADD ["<src>",... "<dest>"]，特别适合路径中带有空格的情况

* shell 格式用法：ADD <src>... <dest>

从远程 URL 获取资源，比如：

ADD http://foo.com/bar.go /tmp/main.go

不过需要注意的是对于从远程 URL 获取资源的情况，由于 ADD 指令不支持认证，如果从远程获取资源需要认证，则只能使用RUN wget 或 RUN curl 替代了。

有能力自动解压文件，比如：

ADD /foo.tar.gz /tmp/

上述指令会使 foo.tar.gz 压缩文件解压到容器的 /tmp 目录。

不过一般来说虽然 ADD 指令支持从远程获取资源，但是并不推荐使用，而是建议使用 RUN 指令去执行 wget 或 curl 命令。

比如前面我们定制的 nginx 镜像，可以改成下面的形式：

$ echo 'Hello Docker!' > index.html

然后修改 Dockerfile：

FROM nginx

*# COPY或者ADD指令都可以*

COPY index.html /user/share/nginx/html/index.html

##### 注意事项

COPY 指令和 ADD指令的用法非常相似，具体注意事项如下：

* 源路径可以有多个
* 源路径是相对于执行 build 的相对路径
* 源路径如果是本地路径，必须是构建上下文中的路径
* 源路径如果是一个目录，则该目录下的所有内容都将被加入到容器，但是该目录本身不会
* 目标路径必须是绝对路径，或相对于 WORKDIR 的相对路径
* 目标路径如果不存在，则会创建相应的完整路径
* 目标路径如果不是一个文件，则必须使用/结束
* 路径中可以使用通配符

### 1.4.2. 构建镜像

现在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:v1 .

Sending build context to Docker daemon 2.048 kB

Step 1 : FROM nginx

---> e43d811ce2f4

Step 2 : RUN echo 'Hello, Docker!' > /usr/share/nginx/html/index.html

---> Running in 9cdc27646c7b

---> 44aa4490ce2c

Removing intermediate container 9cdc27646c7b

Successfully built 44aa4490ce2c

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 9cdc27646c7b，执行了所要求的命令，并最后提交了这一层 44aa4490ce2c，随后删除了所用到的这个容器 9cdc27646c7b。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/->

在这里我们通过 -t 参数指定了最终镜像的名称 nginx:v1，构建成功后，我们就可以直接使用这个镜像来运行容器了。

#### 构建上下文

如果我们仔细观察的话会看到 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 Daemon 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:v1 .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:v1 .

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，可以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 执行时发送一个几十 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.Dev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。

### 1.4.3. 推送镜像

上面我们讲到了 Dockerfile 的基本写法以及构建镜像的时候一些注意事项，那么镜像构建完成后，如何把我们的镜像给到别人使用呢？第一种方法就是利用 Docker 官方提供的公共的 Docker Hub 仓库，我们可以将镜像推送上去，然后别人就可以直接拉取镜像。

#### 1.4.3.1. 注册

首先需要 [https://cloud.docker.com](https://cloud.docker.com/) 免费注册一个 Docker 账号。

#### 1.4.3.2. 登录

然后通过执行docker login命令交互式的输入用户名及密码来完成在命令行界面登录 Docker Hub。

$ 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: cnych

Password:

WARNING! Your password will be stored unencrypted in /root/.docker/config.json.Configure a credential helper to remove this warning. See

https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

#### 1.4.3.3. 注销

你可以通过docker logout退出登录。

#### 1.4.3.4. 推送镜像

用户在登录后可以通过docker push命令来将自己的镜像推送到 Docker Hub。以下命令中的 username 请替换为你的 Docker 账号用户名。

$ docker tag ubuntu:17.10 username/ubuntu:17.10

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu 17.10 275d79972a86 6 days ago 94.6MB

username/ubuntu 17.10 275d79972a86 6 days ago 94.6MB

$ docker push username/ubuntu:17.10

### 1.4.4. 私有仓库

有时候我们可能希望我们的镜像只在局域网范围内使用，不希望推送到 Docker Hub 这样的公共仓库，那么这个时候我们可以创建一个本地仓库供私人使用。

docker-registry 就是是官方提供的一个私有仓库工具，可以用于存储私有的镜像仓库。同样的我们可以通过获取官方 registry 镜像来直接运行：

$ docker run -d -p 5000:5000 --name registry registry:2

上面的命令会使用官方的 registry 镜像来启动私有仓库容器。默认情况下，仓库会被创建在容器的/var/lib/registry目录下。你可以通过 -v 参数来将镜像文件存放在本地的指定路径。例如下面的例子将上传的镜像放到本地的 /opt/data/registry 目录:

$ docker run -d \

-p 5000:5000 \

-v /opt/data/registry:/var/lib/registry \

registry:2

这样我们就运行了一个数据持久化的私有镜像仓库。

创建好私有仓库之后，然后可以使用docker tag来标记一个镜像，然后推送它到仓库。比如私有仓库地址为127.0.0.1:5000。先在本机查看已有的镜像。

$ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

ubuntu latest ba5877dc9bec 6 weeks ago 192.7 MB

使用docker tag将 ubuntu:latest 这个镜像标记为 127.0.0.1:5000/ubuntu:latest:

$ docker tag ubuntu:latest 127.0.0.1:5000/ubuntu:latest

$ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

ubuntu latest ba5877dc9bec 6 weeks ago 192.7 MB127.0.0.1:5000/ubuntu:latest latest ba5877dc9bec 6 weeks ago 192.7 MB

然后就可以使用docker push上传标记的镜像:

$ docker push 127.0.0.1:5000/ubuntu:latest

The push refers to repository [127.0.0.1:5000/ubuntu]

373a30c24545: Pushed

a9148f5200b0: Pushed

cdd3de0940ab: Pushed

fc56279bbb33: Pushed

b38367233d37: Pushed

2aebd096e0e2: Pushed

latest: digest: sha256:fe4277621f10b5026266932ddf760f5a756d2facd505a94d2da12f4f52f71f5a size: 1568

此外，我们还可以使用 registry 仓库提供的 API 来查看仓库中的镜像：

$ curl 127.0.0.1:5000/v2/\_catalog

{"repositories":["ubuntu"]}

这里可以看到 {"repositories":["ubuntu"]}，表明镜像已经被成功上传了。

先删除已有镜像，再尝试从私有仓库中下载这个镜像。

$ docker image rm 127.0.0.1:5000/ubuntu:latest

$ docker pull 127.0.0.1:5000/ubuntu:latest

Pulling repository 127.0.0.1:5000/ubuntu:latest

ba5877dc9bec: Download complete

511136ea3c5a: Download complete

9bad880da3d2: Download complete

25f11f5fb0cb: Download complete

ebc34468f71d: Download complete

2318d26665ef: Download complete

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE127.0.0.1:5000/ubuntu:latest latest ba5877dc9bec 6 weeks ago 192.7 MB

到这里我们就完成了把镜像上传到了私有仓库中的完整过程。

#### 注意事项

如果你不想使用 127.0.0.1:5000 作为仓库地址，比如想让本网段的其他主机也能把镜像推送到私有仓库。你就得把例如 192.168.199.100:5000 这样的内网地址作为私有仓库地址，这时你会发现无法成功推送镜像。

这是因为 Docker 默认不允许非 HTTPS 方式推送镜像。我们可以通过 Docker 的配置选项来取消这个限制，我们这里是 CentOS 7 系统，同样还是编辑文件/etc/docker/daemon.json，添加如下内容：

{

"registry-mirror": [

"https://registry.docker-cn.com"

],

"insecure-registries": [

"192.168.199.100:5000"

]}

其中的insecure-registries就是我们添加的内容，然后重启 Docker 之后就可以在局域网内使用我们的私有镜像仓库了。

默认的私有镜像仓库可以满足我们的很多需求，但是往往在企业中使用的话还有很安全性方面的功能的缺失，比如权限管理之类的，这对于企业来说是非常重要的，不过 registry 对于权限管理这一块做了一个对外暴露的接口，只要我们实现他暴露的安全接口就可以来实现权限管理相关的接口，其中注明的开源镜像管理软件 [Harbor](https://goharbor.io/) 就是这类应用的佼佼者，我们还在后面的课程中和大家学习 Harbor 的一个使用。

## 1.5. Dockerfile 实践

在前文中我们了解到了用于构建 Dockerfile 的基本方法，但是由于在编写 Dockerfile 的时候并没有一些强制要求，导致很多构建的镜像不符合一些最佳实践，典型的就是镜像构建的层数非常多，对一些基本指令的区别不是很清楚。本节主要介绍 Dockerfile 在实际使用中的一些最佳的实践方式。

第一个是前面重点提到的构建上下文，这个一定需要理解，然后是对于一些不需要提交构建的文件用.dockerignore来进行忽略。

### 1.5.1. 构建缓存

在镜像的构建过程中，Docker 根据 Dockerfile 指定的顺序执行每个指令。在执行每条指令之前，Docker 都会在缓存中查找是否已经存在可重用的镜像，如果有就使用现存的镜像，不再重复创建。当然如果你不想在构建过程中使用缓存，你可以在 docker build 命令中使用 --no-cache=true 选项。Docker 中构建缓存遵循的基本规则如下：

* 从一个基础镜像开始（FROM 指令指定），下一条指令将和该基础镜像的所有子镜像进行匹配，检查这些子镜像被创建时使用的指令是否和被检查的指令完全一样。如果不是，则缓存失效。
* 对于 ADD 和 COPY 指令，镜像中对应文件的内容也会被检查，每个文件都会计算出一个校验值。在缓存的查找过程中，会将这些校验和已存在镜像中的文件校验值进行对比。如果文件有任何改变，则缓存失效。
* 除了 ADD 和 COPY 指令，缓存匹配过程不会查看临时容器中的文件来决定缓存是否匹配。例如，当执行完 RUN apt-get -y update 指令后，容器中一些文件被更新，但 Docker 不会检查这些文件。这种情况下，只有指令字符串本身被用来匹配缓存。
* 一旦缓存失效，所有后续的 Dockerfile 指令都将产生新的镜像，缓存不会被使用。

### 1.5.2. 使用多阶段构建

多阶段构建可以让我们大幅度减小最终的镜像大小，而不需要去想办法减少中间层和文件的数量。因为镜像是在生成过程的最后阶段生成的，所以可以利用生成缓存来最小化镜像层。

例如，如果你的构建包含多个层，则可以将它们从变化频率较低（以确保生成缓存可重用）到变化频率较高的顺序排序：

* 安装构建应用程序所需的依赖工具
* 安装或更新依赖项
* 构建你的应用

比如我们构建一个 Go 应用程序的 Dockerfile 可能类似于这样：

FROM golang:1.11-alpine AS build

*# 安装项目需要的工具*

*# 运行 `docker build --no-cache .` 来更新依赖*

RUN apk add --no-cache git

RUN go get github.com/golang/dep/cmd/dep

*# 通过 Gopkg.toml 和 Gopkg.lock 获取项目的依赖*

*# 仅在更新 Gopkg 文件时才重新构建这些层*

COPY Gopkg.lock Gopkg.toml /go/src/project/

WORKDIR /go/src/project/*# 安装依赖库*

RUN dep ensure -vendor-only

*# 拷贝整个项目进行构建*

*# 当项目下面有文件变化的时候该层才会重新构建*

COPY . /go/src/project/

RUN go build -o /bin/project

*# 将打包后的二进制文件拷贝到 scratch 镜像下面，将镜像大小降到最低*

FROM scratch

COPY --from=build /bin/project /bin/project

ENTRYPOINT ["/bin/project"]

CMD ["--help"]

### 1.5.3. 避免安装不必要的包

为了降低复杂性、减少依赖、减小文件大小和构建时间，应该避免安装额外的或者不必要的软件包。例如，不要在数据库镜像中包含一个文本编辑器。

### 1.5.4. 应用解耦

每个容器应用只关心一个方面的事情。将多个应用解耦到不同容器中，可以更轻松地保证容器的横向扩展和复用。例如一个 web 应用程序可能包含三个独立的容器：web应用、数据库、缓存，每个容器都是独立的镜像，分开运行。但这并不是说一个容器就只能跑一个进程，因为有的程序可能会自行产生其他进程，比如 Celery 就可以有很多个工作进程。虽然每个容器跑一个进程是一条很好的法则，但这并不是一条硬性的规定。我们主要是希望一个容器只关注一件事情，尽量保持干净和模块化。

如果容器互相依赖，你可以使用 [Docker 容器网络](https://www.qikqiak.com/docker/basic/" \l "container) 来把这些容器连接起来，我们前面已经跟大家讲解过 Docker 的容器网络模式。

### 1.5.5. 最小化镜像层数

在很早之前的版本中尽量减少镜像层数是非常重要的，不过现在的版本已经有了一定的改善了：

* 只有 RUN、COPY 和 ADD 指令会创建层，其他指令会创建临时的中间镜像，但是不会直接增加构建的镜像大小了。
* 多阶段构建的支持，允许我们把需要的数据直接复制到最终的镜像中，这就允许我们在中间阶段包含一些工具或者调试信息了，而且不会增加最终的镜像大小。

### 1.5.6. 对多行参数排序

只要有可能，就将多行参数按字母顺序排序。这可以帮助你避免重复包含同一个包，更新包列表时也更容易，也更容易阅读和审查。**建议在反斜杠符号 \ 之前添加一个空格，可以增加可读性**。比如下面的例子：

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion

### 1.5.7. Dockerfile 指令

关于这些指令的使用建议可以帮助我们创建高效且可维护的 Dockerfile。

#### FROM 指令

前面已经介绍过，尽可能使用当前的官方镜像作为基础镜像。我们建议使用Alpine映像，因为它受到严格控制且较小（当前小于5 MB），同时仍是完整的 Linux 发行版。

#### LABEL 标签

你可以给镜像添加标签来帮助组织镜像、记录许可信息、辅助自动化构建等。每个标签一行，由 LABEL 开头加上一个或多个标签对。

下面的示例展示了各种不同的可能格式。#开头的行是注释内容。

**注意：**如果你的字符串包含空格，那么它必须被引用或者空格必须被转义。如果您的字符串包含内部引号字符（"），则也可以将其转义。

*# Set one or more individual labels*

LABEL com.example.version="0.0.1-beta"

LABEL vendor="ACME Incorporated"

LABEL com.example.release-date="2015-02-12"

LABEL com.example.version.is-production=""

一个镜像可以包含多个标签，当然以上内容也可以写成下面这样，但是不是必须的：

*# Set multiple labels at once, using line-continuation characters to break long lines*

LABEL vendor=ACME\ Incorporated \

com.example.is-production="" \

com.example.version="0.0.1-beta" \

com.example.release-date="2015-02-12"

#### RUN 指令

为了保持 Dockerfile 文件的可读性，以及可维护性，建议将长的或复杂的 RUN 指令用反斜杠\分割成多行。

RUN 指令最常见的用法是安装包用的 apt-get。因为RUN apt-get指令会安装包，所以有几个问题需要注意。

* 不要使用 RUN apt-get upgrade 或 dist-upgrade，如果基础镜像中的某个包过时了，你应该联系它的维护者。如果你确定某个特定的包，比如 foo，需要升级，使用 apt-get install -y foo 就行，该指令会自动升级 foo 包。
* 永远将 RUN apt-get update 和 apt-get install 组合成一条 RUN 声明，例如：

RUN apt-get update && apt-get install -y \

package-bar \

package-baz \

package-foo

* 将 apt-get update 放在一条单独的 RUN 声明中会导致缓存问题以及后续的 apt-get install 失败。比如，假设你有一个 Dockerfile 文件：

FROM ubuntu:14.04

RUN apt-get update

RUN apt-get install -y curl

构建镜像后，所有的层都在 Docker 的缓存中。假设你后来又修改了其中的 apt-get install 添加了一个包：

FROM ubuntu:14.04

RUN apt-get update

RUN apt-get install -y curl nginx

Docker 发现修改后的 RUN apt-get update 指令和之前的完全一样。所以，apt-get update 不会执行，而是使用之前的缓存镜像。因为 apt-get update 没有运行，后面的 apt-get install 可能安装的是过时的 curl 和 nginx 版本。

使用RUN apt-get update && apt-get install -y可以确保你的 Dockerfiles 每次安装的都是包的最新的版本，而且这个过程不需要进一步的编码或额外干预。这项技术叫作cache busting(缓存破坏)。

下面是一个 RUN 指令的示例模板，展示了所有关于 apt-get 的建议：

RUN apt-get update && apt-get install -y \

aufs-tools \

automake \

build-essential \

curl \

dpkg-sig \

libcap-dev \

libsqlite3-dev \

mercurial \

reprepro \

ruby1.9.1 \

ruby1.9.1-dev \

s3cmd=1.1.\* \

*# 其他操作*

&& rm -rf /var/lib/apt/lists/\*

其中 s3cmd 指令指定了一个版本号 1.1.\*。如果之前的镜像使用的是更旧的版本，指定新的版本会导致 apt-get udpate 缓存失效并确保安装的是新版本。 另外，清理掉 apt 缓存 var/lib/apt/lists 可以减小镜像大小。因为 RUN 指令的开头为 apt-get udpate，包缓存总是会在 apt-get install 之前刷新。

#### EXPOSE 指令

EXPOSE 指令用于指定容器将要监听的端口。因此，你应该为你的应用程序使用常见的端口。

例如，提供 Apache web 服务的镜像应该使用 EXPOSE 80，而提供 MongoDB 服务的镜像使用 EXPOSE 27017。

对于外部访问，用户可以在执行 docker run 时使用一个 -p 参数来指示如何将指定的端口映射到所选择的端口。

#### ENV 指令

为了方便新程序运行，你可以使用 ENV 指令来为容器中安装的程序更新 PATH 环境变量。例如使用ENV PATH /usr/local/nginx/bin:$PATH 来确保CMD ["nginx"]能正确运行。

ENV 指令也可用于为你想要容器化的服务提供必要的环境变量，比如 Postgres 需要的 PGDATA。 最后，ENV 也能用于设置常见的版本号，比如下面的示例：

ENV PG\_MAJOR 9.3

ENV PG\_VERSION 9.3.4

RUN curl -SL http://example.com/postgres-$PG\_VERSION.tar.xz | tar -xJC /usr/src/postgress && …

ENV PATH /usr/local/postgres-$PG\_MAJOR/bin:$PATH

类似于程序中的常量，这种方法可以让你只需改变 ENV 指令来自动的改变容器中的软件版本。

#### VOLUME 指令

VOLUME 指令用于暴露任何数据库存储文件，配置文件，或容器创建的文件和目录。强烈建议使用 VOLUME 来管理镜像中的可变部分和用户可以改变的部分。

#### USER 指令

如果某个服务不需要特权执行，建议使用 USER 指令切换到非 root 用户。先在 Dockerfile 中使用类似 RUN groupadd -r postgres && useradd -r -g postgres postgres 的指令创建用户和用户组。

**注意：**在镜像中，用户和用户组每次被分配的 UID/GID 都是不确定的，下次重新构建镜像时被分配到的 UID/GID 可能会不一样。如果要依赖确定的 UID/GID，你应该显示的指定一个 UID/GID。

你应该避免使用 sudo，因为它不可预期的 TTY 和信号转发行为可能造成的问题比它能解决的问题还多。如果你真的需要和 sudo 类似的功能（例如，以 root 权限初始化某个守护进程，以非 root 权限执行它），你可以使用 gosu。我们可以去查看官方的一些镜像，很多都是使用的 gosu。

为了减少层数和复杂度，避免频繁地使用 USER 来回切换用户。

#### WORKDIR 指令

为了清晰性和可靠性，你应该总是在 WORKDIR 中使用绝对路径。另外，你应该使用 WORKDIR 来替代类似于 RUN cd ... && do-something 的指令，后者难以阅读、排错和维护。

#### COPY 和 ADD 指令

虽然 ADD 和 COPY 功能类似，但一般优先使用 COPY。因为它比 ADD 更透明。COPY 只支持简单将本地文件拷贝到容器中，而 ADD 有一些并不明显的功能（比如本地 tar 提取和远程 URL 支持）。因此，ADD 的最佳用例是将本地 tar 文件自动提取到镜像中，例如ADD rootfs.tar.xz。

如果你的 Dockerfile 有多个步骤需要使用上下文中不同的文件。单独 COPY 每个文件，而不是一次性的 COPY 所有文件，这将保证每个步骤的构建缓存只在特定的文件变化时失效。例如：

COPY requirements.txt /tmp/

RUN pip install --requirement /tmp/requirements.txt

COPY . /tmp/

如果将COPY . /tmp/放置在 RUN 指令之前，只要 . 目录中任何一个文件变化，都会导致后续指令的缓存失效。

为了让镜像尽量小，最好不要使用 ADD 指令从远程 URL 获取包，而是使用 curl 和 wget。这样你可以在文件提取完之后删掉不再需要的文件来避免在镜像中额外添加一层。比如尽量避免下面的用法：

ADD http://example.com/big.tar.xz /usr/src/things/

RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things

RUN make -C /usr/src/things all

而是应该使用下面这种方法：

RUN mkdir -p /usr/src/things \

&& curl -SL http:*//example.com/big.tar.xz \*

| tar -xJC /usr/src/things \

&& make -C /usr/src/things all

上面使用的管道操作，所以没有中间文件需要删除。对于其他不需要 ADD 的自动提取功能的文件或目录，你应该使用 COPY。

#### CMD 与 ENTRYPOINT 指令

尽管 ENTRYPOINT 和 CMD 都是在容器里执行一条命令, 但是他们有一些微妙的区别，在绝大多数情况下, 你只要在这2者之间选择一个调用就可以，但是我们还是非常有必要来认真了解下二者的区别。

##### CMD 指令

CMD 指令是容器启动以后，默认的执行命令，需要重点理解下这个默认的含义，意思就是如果我们执行 docker run 没有指定任何的执行命令或者 Dockerfile 里面也没有指定 ENTRYPOINT，那么就会使用 CMD 指定的执行命令执行了。这也说明了 ENTRYPOINT 才是容器启动以后真正要执行的命令。

所以我们经常遇到 CMD 会被覆盖 的情况，为什么会被覆盖呢？主要还是因为 CMD 的定位就是默认，如果不额外指定，那么才会执行 CMD 命令，但是如果我们指定了的话那就不会执行 CMD 命令了，也就是说 CMD 会被覆盖。

CMD 总共有三种用法：

CMD ["executable", "param1", "param2"]

*# exec 形式*

CMD ["param1", "param2"]

*# 作为 ENTRYPOINT 的默认参数*

CMD command param1 param2

*# shell 形式*

其中 shell 形式，就是没有中括号的形式，命令 command 默认是在/bin/sh -c下执行的，比如：

FROM busybox

CMD echo "hello cmd shell form!"

我们将上面的 Dockerfile 打包成 cmdshell 镜像，然后直接启动一个容器：

$ docker build -t cmdshell .

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM busybox

---> 020584afccce

Step 2/2 : CMD echo "hello cmd shell form!"

---> Running in 651afaddb83d

Removing intermediate container 651afaddb83d

---> d26e4d6d9cdf

Successfully built d26e4d6d9cdf

Successfully tagged cmdshell:latest

$ docker run cmdshell

hello cmd shell form!

对于带有中括号的 exec 形式，命令没有在任何 shell 终端环境下，如果我们要执行 shell，必须把 shell 加入到中括号的参数中。将上面的例子修改为：

FROM busybox

CMD ["/bin/sh", "-c", "echo 'hello cmd exec form!'"]

同样将上面的 Dockerfile 打包成 cmdexec 镜像，然后直接启动一个容器：

$ docker build -t cmdexec .

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM busybox

---> 020584afccce

Step 2/2 : CMD ["/bin/sh", "-c", "echo 'hello cmd exec form!'"]

---> Running in 8c72d5e2ce35

Removing intermediate container 8c72d5e2ce35

---> c393bd1ab3c1

Successfully built c393bd1ab3c1

Successfully tagged cmdexec:latest

$ docker run cmdexec

hello cmd exec form!

需要注意，采用 exec 形式，第一个参数必须是命令的全路径才行。**一个 Dockerfile 如果有多个 CMD，只有最后一个生效，官网推荐采用这种方式**。

当然，以上都是体现了 CMD 的 **默认** 行为。如果我们在 run 时指定了命令或者有 ENTRYPOINT CMD 就会被覆盖。比如同样用上面两个镜像，在运行的时候指定一个命令：

$ docker run cmdexec echo 'hello docker'

hello docker

$ docker run cmdshell echo 'hello docker'

hello docker

可以看到，最终容器里面执行的是 run 命令后面的命令，而不是 CMD 里面定义的。

##### ENTRYPOINT 指令

根据官方定义来说 ENTRYPOINT 才是用于定义容器启动以后的执行程序的，允许将镜像当成命令本身来运行（用 CMD 提供默认选项），从名字也可以理解，是容器的**入口**。ENTRYPOINT 一共有两种用法：

ENTRYPOINT ["executable", "param1", "param2"] (exec 形式)

ENTRYPOINT command param1 param2 (shell 形式)

对应命令行 exec 模式，也就是带中括号的。和 CMD 的中括号形式是一致的，但是这里貌似是在shell的环境下执行的，与cmd有区别。如果 run 命令后面有执行命令，那么后面的全部都会作为 ENTRYPOINT 的参数。如果 run 后面没有额外的命令，但是定义了 CMD，那么 CMD 的全部内容就会作为 ENTRYPOINT 的参数，这同时是上面我们提到的 CMD 的第二种用法。所以说 ENTRYPOINT 不会被覆盖。当然如果要在 run 里面覆盖，也是有办法的，使用--entrypoint参数即可。

比如我们定义如下的 Dockerfile：

FROM busybox

CMD ["I am in cmd exec form"]

ENTRYPOINT ["echo"]

将上面的 Dockerfile 打包成镜像 entrypointest，然后直接运行，不带任何参数：

$ docker build -t entrypointest .

Sending build context to Docker daemon 2.048kB

Step 1/3 : FROM busybox

---> 020584afccce

Step 2/3 : CMD ["I am in cmd exec form"]

---> Running in 2d7b13b0dfe7

Removing intermediate container 2d7b13b0dfe7

---> 903d739ead9a

Step 3/3 : ENTRYPOINT ["echo"]

---> Running in c61682ea476e

Removing intermediate container c61682ea476e

---> 00b09a578d48

Successfully built 00b09a578d48

Successfully tagged entrypointest:latest

$ docker run entrypointest

I am in cmd exec form

我们可以看到打印的结果是 CMD 里面指定的内容，也就是默认情况将 CMD 部分作为 ENTRYPOINT 的参数了。但是如果我们在运行容器的时候如果指定了运行参数呢：

$ docker run entrypointest I am in run section

I am in run section

我们可以看到运行时指定的参数会覆盖掉 CMD 提供的默认参数，但是默认都是执行的 ENTRYPOINT 里面的命令。

对于 shell 模式的，任何 run 和 CMD 的参数都无法被传入到 ENTRYPOINT 里。官网推荐用上面一种用法。比如我们我们这里定义一个 Dockerfile 如下：

FROM busybox

CMD ["I am in cmd exec form and entrypoint shell form"]

ENTRYPOINT echo

将上面 Dockerfile 打包成镜像 entrypointshell，然后直接运行：

$ docker build -t entrypointshell .

Sending build context to Docker daemon 2.048kB

Step 1/3 : FROM busybox ---> 020584afccce

Step 2/3 : CMD ["I am in cmd exec form and entrypoint shell form"] ---> Running in 2aee7326f4cd

Removing intermediate container 2aee7326f4cd ---> e89cadeeecd3

Step 3/3 : ENTRYPOINT echo ---> Running in f359c8bb5025

Removing intermediate container f359c8bb5025 ---> f9d4e1d1b0a0

Successfully built f9d4e1d1b0a0

Successfully tagged entrypointshell:latest

$ docker run entrypointshell

我们可以发现 CMD 的参数并没有被打印出来，如果在运行的时候添加上参数呢：

$ docker run entrypointshell I am in run section

我们可以发现也没有将 run 命令后面的参数打印出来。所以一般情况下对于 ENTRYPOINT 来说使用中括号的 exec 形式更好。

**总结：**一般会用 ENTRYPOINT 的中括号形式作为 Docker 容器启动以后的默认执行命令，里面放的是不变的部分，可变部分比如命令参数可以使用 CMD 的形式提供默认版本，也就是 run 里面没有任何参数时使用的默认参数。如果我们想用默认参数，就直接 run，否则想用其他参数，就 run 里面加上参数。

### 1.5.8. 官方仓库示例

这些官方仓库的 Dockerfile 都是参考典范：<https://github.com/docker-library/docs>。

### 1.5.9. Wordpress 示例

这里我们以创建一个 Wordpress 镜像为例，我们知道默认情况下 Wordpress 是使用 MySQL 数据库的，如果使用 MySQL 数据库的不利于我们这里的演示，所以我们这里用 Sqlite 来作为 Wordpress 的存储数据库，要编写一个使用 Sqlite 数据库的 Wordpress 镜像，首先我们需要明白如果不使用镜像应该怎么来部署这样的一个程序，否则是不可能编写出 Dockerfile 的。

#### 获取 Wordpress

首先，我们需要获取到 Wordpress 的程序安装包，地址：<https://wordpress.org/download/>，下载最新的 Wordpress 安装包。

#### 获取 SQLite 插件

由于 Wordpress 默认使用的是 MySQL 数据库，如果要使用 SQLite 数据库的话需要使用 SQLite 插件，地址：<https://wordpress.org/plugins/sqlite-integration/>

#### 安装激活插件

将 SQLite 插件解压放置到 Wordpress 源码目录 wp-content/plugins 目录下面，然后拷贝 sqlite-integration 目录下面的 db.php 文件到 Wordpress 的 wordpress/wp-content 目录下面。

然后重新将 wordpress/wp-config-sample.php 重命名为 wordpress/wp-config.php，可以通过该文件里面的一些参数来配置数据库的名称和位置，默认情况下，数据库名为.ht.sqlite，存储在 wp-content/database/ 目录中。

然后安装上 Wordpress 运行的一些环境比较 PHP、Nginx 就可以运行了，最终的 Dockerfile 如下所示：

FROM nginx:latest

LABEL version="1.0"

LABEL author="阳明 <icnych@gmail.com>"

*# 设置成非交互式的*

ENV DEBIAN\_FRONTEND noninteractive

ENV DOCUMENT\_ROOT /usr/share/nginx/html

*# 安装 nginx php-fpm php-pdo unzip curl*

RUN apt-get update && apt-get -y install \

apt-utils \

curl \

php-fpm \

php-sqlite3 \

unzip

COPY wordpress-5.2.4.tar.gz .

RUN rm -rf ${DOCUMENT\_ROOT}/\* && \

tar -xzvf wordpress-5.2.4.tar.gz --strip-components=1 --directory ${DOCUMENT\_ROOT} && \

rm -rf wordpress-5.2.4.tar.gz

*# RUN rm -rf ${DOCUMENT\_ROOT}/\* && \*

*# curl -o wordpress.tar.gz https://wordpress.org/latest.tar.gz && \*

*# tar -xzvf /wordpress.tar.gz --strip-components=1 --directory ${DOCUMENT\_ROOT}*

COPY sqlite-integration.1.8.1.zip .

RUN unzip sqlite-integration.1.8.1.zip -d ${DOCUMENT\_ROOT}/wp-content/plugins/ && \

rm -rf sqlite-integration.1.8.1.zip*# RUN curl -o sqlite-plugin.zip https://downloads.wordpress.org/plugin/sqlite-integration.1.8.1.zip && \*

*# unzip sqlite-plugin.zip -d ${DOCUMENT\_ROOT}/wp-content/plugins/ && \*

*# rm sqlite-plugin.zip && \*

RUN cp ${DOCUMENT\_ROOT}/wp-content/plugins/sqlite-integration/db.php ${DOCUMENT\_ROOT}/wp-content && \

cp ${DOCUMENT\_ROOT}/wp-config-sample.php ${DOCUMENT\_ROOT}/wp-config.php

*# nginx 配置*

RUN sed -i -e"s/keepalive\_timeout\s\*65/keepalive\_timeout 2/" /etc/nginx/nginx.conf \

&& sed -i -e"s/keepalive\_timeout 2/keepalive\_timeout 2;\n\tclient\_max\_body\_size 10m/" /etc/nginx/nginx.conf \

&& sed -i -e "s|include /etc/nginx/conf.d/\\*.conf|include /etc/nginx/sites-enabled/\\*|g" /etc/nginx/nginx.conf \

&& echo "daemon off;" >> /etc/nginx/nginx.conf

*# php-fpm 配置# RUN php --ini # 找到fpm配置*RUN sed -i -e "s/;cgi.fix\_pathinfo=1/cgi.fix\_pathinfo=0/g" /etc/php/7.3/fpm/php.ini \

&& sed -i -e "s/upload\_max\_filesize\s\*=\s\*2M/upload\_max\_filesize = 10M/g" /etc/php/7.3/fpm/php.ini \

&& sed -i -e "s/post\_max\_size\s\*=\s\*8M/post\_max\_size = 10M/g" /etc/php/7.3/fpm/php.ini \

&& sed -i -e "s/;catch\_workers\_output\s\*=\s\*yes/catch\_workers\_output = yes/g" /etc/php/7.3/fpm/pool.d/www.conf \

&& sed -i -e "s/;listen.mode = 0660/listen.mode = 0666/g" /etc/php/7.3/fpm/pool.d/www.conf

COPY nginx.conf /etc/nginx/sites-available/default

RUN chown -R www-data.www-data ${DOCUMENT\_ROOT} \

&& mkdir -p /etc/nginx/sites-enabled \

&& ln -s /etc/nginx/sites-available/default /etc/nginx/sites-enabled/default \

&& apt-get purge -y --auto-remove curl unzip

EXPOSE 80

EXPOSE 443

CMD service php7.3-fpm start && nginx

由于一些网络原因，我们这里将安装包都放置到了 Dockerfile 目录下面，然后我们就可以来构建镜像：

$ docker build -t cnych/wordpress:sqlite .

然后用该镜像来运行一个 Wordpress 容器：

$ docker run -d --name wordpress -p 8888:80 cnych/wordpress:sqlite

运行成功后我们就可以通过 8888 端口来访问 Wordpress 应用了，而且我们没有 MySQL 相关的配置，通过一些配置就可以访问到应用了。

