Docker 从入门到实践

作者 yeasy

版本 1.10.0

日期 2026-05-31

Built with mdPress

- Docker 从入门到实践
  - 关于本书
  - 内容特色
  - 五分钟快速上手
  - 学习路线图
  - 在线阅读
  - 下载离线版本
  - 本地阅读
  - 社区交流
  - 推荐阅读
  - 参与贡献
  - 进阶学习
  - 支持鼓励
  - Star History
  - 许可证
- 修订记录
  - 如何贡献
  - 排版规范
- 第一章 Docker 简介
  - 本章内容
  - 学习目标
  - 1.1 快速上手
  - 1.2 什么是 Docker
  - 1.3 为什么要用 Docker
  - 本章小结
- 第二章 基本概念

    - 2.1 镜像
    - 2.2 容器
    - 2.3 仓库
    - 本章小结
    - 现在你已经了解了 Docker 的三个核心概念：镜像、容器 和仓库。接下来，让我们开始 安装 Docker，动手实践！
- 第三章 安装 Docker
    - 安装方式选择指南
    - 详细安装指南
    - 3.1 Ubuntu
    - 3.2 Debian
    - 3.3 Fedora
    - 3.4 CentOS
    - 3.5 Raspberry Pi
    - 3.6 Linux 离线安装
    - 3.7 macOS
    - 3.8 Windows 10/11
    - 3.9 镜像加速器
    - 3.10 开启实验特性
    - 本章小结
- 第四章 使用镜像

Built with mdPress

    - 本章内容
    - 4.1 获取镜像
    - 4.2 列出镜像
    - 4.3 删除本地镜像
    - 4.4 利用 commit 理解镜像构成
    - 4.5 使用 Dockerfile 定制镜像
    - 4.6 其它制作镜像的方式
    - 4.7 实现原理
    - 本章小结
- 第五章 操作容器
    - 版本号说明
    - 5.1 启动
    - 5.2 守护态运行
    - 5.3 终止
    - 5.4 进入容器
    - 5.5 导出和导入
    - 5.6 删除
    - 本章小结
- 第六章 访问仓库
    - 版本号说明
    - 为什么需要私有仓库?
    - 本章内容
    - 6.1 Docker Hub
    - 6.2 私有仓库
    - 6.3 私有仓库高级配置
    - 6.4 Nexus 3
    - 本章小结

Built with mdPress

- 第七章 Dockerfile 指令详解
  - 什么是 Dockerfile
  - Dockerfile 编写哲学
  - Dockerfile 基本结构
  - 使用 Dockerfile 构建镜像
  - 7.1 RUN 执行命令
  - 7.2 COPY 复制文件
  - 7.3 ADD 更高级的复制文件
  - 7.4 CMD 容器启动命令
  - 7.5 ENTRYPOINT 入口点
  - 7.6 ENV 设置环境变量
  - 7.7 ARG 构建参数
  - 7.8 VOLUME 定义匿名卷
  - 7.9 EXPOSE 暴露端口
  - 7.10 WORKDIR 指定工作目录
  - 7.11 USER 指定当前用户
  - 7.12 HEALTHCHECK 健康检查
  - 7.13 ONBUILD 为他人作嫁衣裳
  - 7.14 LABEL 为镜像添加元数据
  - 7.15 SHELL 指令
  - 7.16 参考文档
  - 7.17 多阶段构建
  - 7.18 实战多阶段构建 Laravel 镜像
  - 本章小结
- 第八章 数据管理

Built with mdPress

    - 8.1 数据卷
    - 8.2 挂载主机目录
    - 8.3 tmpfs 挂载
    - 本章小结
- 第九章 网络配置
    - 概述
    - 本章内容
    - 9.1 配置 DNS
    - 9.2 网络类型
    - 9.3 自定义网络
    - 9.4 容器互联
    - 9.5 外部访问容器
    - 9.6 网络隔离
    - 9.7 容器网络高级特性
    - 本章小结
- 第十章 Docker Buildx
    - 本章内容
    - 10.1 BuildKit
    - 10.2 使用 buildx 构建镜像
    - 10.3 使用 buildx 构建多种系统架构支持的 Docker 镜像
    - 本章小结
- 第十一章 Docker Compose

Built with mdPress

    - Docker Compose 解决什么问题?
    - 11.1 简介
    - 11.2 安装与卸载
    - 11.3 使用
    - 11.4 命令说明
    - 11.5 Compose 模板文件
    - 11.6 实战 Django
    - 11.7 实战 Rails
    - 11.8 实战 WordPress
    - 11.9 实战 LNMP
    - 本章小结
- 第十二章 底层实现
    - 本章内容
    - 12.1 基本架构
    - 12.2 命名空间
    - 12.3 控制组
    - 12.4 联合文件系统
    - 12.5 容器格式
    - 12.6 网络
    - 本章小结
- 第十三章 容器编排基础
    - 13.1 简介
    - 13.2 基本概念
    - 13.3 架构设计
    - 13.4 高级特性
    - 13.5 实战练习
    - 本章小结

Built with mdPress

- 第十四章 部署 Kubernetes
  - 14.1 使用 kubeadm 部署 Kubernetes
  - 14.2 使用 kubeadm 部署 Kubernetes：使用 Docker
  - 14.3 在 Docker Desktop 使用
  - 14.4 Kind - Kubernetes IN Docker
  - 14.5 K3s - 轻量级 Kubernetes
  - 14.6 一步步部署 Kubernetes 集群
  - 14.7 部署 Dashboard
  - 14.8 Kubernetes 命令行 kubectl
  - 本章小结
- 第十五章 Etcd 项目
  - 本章内容
  - 15.1 简介
  - 15.2 安装
  - 15.3 集群
  - 15.4 使用 etcdctl
  - 本章小结
- 第十六章 容器与云计算
  - 本章内容
  - 16.1 简介
  - 16.2 腾讯云
  - 16.3 阿里云
  - 16.4 亚马逊云
  - 16.5 多云部署策略
  - 本章小结
  - 同时，容器将作为与虚拟机类似的业务直接提供给用户使用，极大的丰富了应用开发和部署的场景。

Built with mdPress

- 第十七章 容器其它生态
  - 本章内容
  - 17.1 Fedora CoreOS 简介
  - 17.2 Fedora CoreOS 安装
  - 17.3 Podman - 下一代 Linux 容器工具
  - 17.4 Buildah - 容器镜像构建工具
  - 17.5 Skopeo - 容器镜像管理工具
  - 17.6 containerd - 核心容器运行时
  - 17.7 安全容器运行时
  - 17.8 WebAssembly 与容器
  - 本章小结
- 第十八章 安全
  - 容器安全的本质
  - 本章内容
  - 安全扫描清单
  - 18.1 内核命名空间
  - 18.2 控制组
  - 18.3 服务端防护
  - 18.4 内核能力机制
  - 18.5 其它安全特性
  - 18.6 容器镜像安全扫描与供应链安全
  - 本章小结
  - 另外，用户可以使用现有工具，比如 Apparmor，Seccomp，SELinux，GRSEC 来增强安全性；甚至自己在内核中实现更复杂的安全机制。
- 第十九章 容器监控与日志

Built with mdPress

    - 本章内容
    - 19.1 Prometheus
    - 19.2 ELK 套件
    - 19.3 容器性能优化与故障诊断
    - 本章小结
    - 扩展阅读：Docker 日志驱动
    - 19.4 日志平台选型对比与注意事项
    - 19.5 上线前检查清单
- 第二十章 实战案例 - 操作系统
    - 章节概述
    - 版本说明
    - 为什么选择合适的操作系统镜像很重要
    - 常用操作系统镜像对比
    - 学习目标
    - 章节内容导航
    - 20.1 Busybox
    - 20.2 Alpine
    - 20.3 Debian Ubuntu
    - 20.4 CentOS Fedora
    - 本章小结
- 第二十一章 实战案例 - DevOps

Built with mdPress

    - 版本说明
    - DevOps 背景介绍
    - Docker 在 DevOps 中的角色
    - CI/CD 管道的重要性
    - 本章学习目标
    - 章节内容导航
    - 21.1 DevOps 完整工作流
    - 21.2 GitHub Actions
    - 21.3 Drone
    - 21.4 Drone Demo
    - 21.5 在 IDE 中使用 Docker
    - 21.6 VS Code
    - 21.7 实战案例：Go/Rust/数据库/微服务
    - 本章小结
- 附录
    - 目录
- 附录一：常见问题与错误速查
    - 镜像相关
    - 容器相关
    - 仓库相关
    - 配置相关
    - Docker 与虚拟化
    - 其它
    - 常见错误速查表
- 附录二：热门镜像介绍

Built with mdPress

    - 目录
    - Ubuntu
    - CentOS
    - Nginx
    - PHP
    - Node.js
    - MySQL
    - WordPress
    - MongoDB
    - Redis
    - Minio
- 附录三：Docker 命令查询

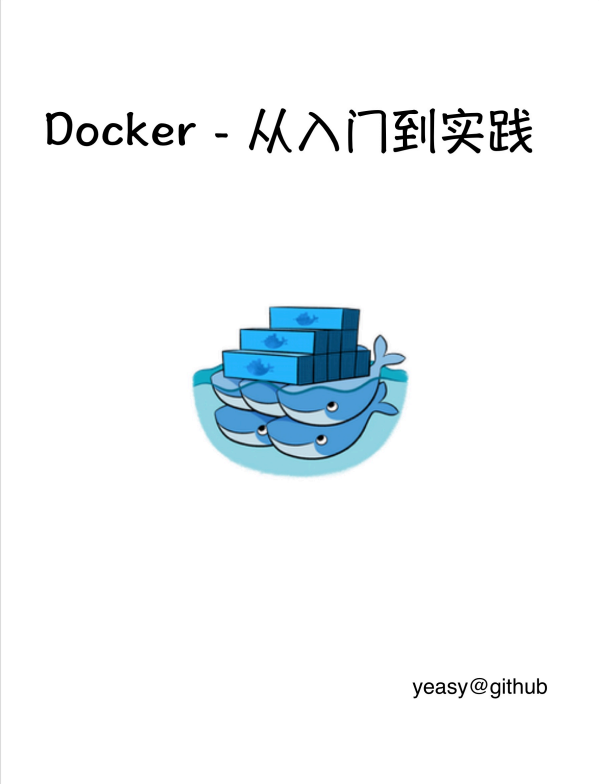
- 基本语法
- 客户端命令 - docker
- 服务端命令 - dockerd
- 附录四：Dockerfile 最佳实践
- 附录五：如何调试 Docker
- 附录六：资源链接
- 附录七：术语表
- A
- B
- C
- D
- E
- I
- K
- L
- M
- N
- O
- P
- R
- S
- U
- V
- 附录八：Docker 学习路线图与知识体系

Built with mdPress

Docker 从入门到实践

License CC BY-NC-SA 4.0 Stars 26k release v1.9.3 在线阅读 GitBook PDF 下载
Based Docker Engine v29.x

> 从零开始，系统掌握 Docker 容器技术的核心概念、原理与实战技巧



关于本书

Docker 是个划时代的开源项目，它彻底释放了计算虚拟化的威力，极大提高了应用的维护效率，降低了云计算应用开发的成本！使用 Docker，可以让应用的部署、测试和分发都变得前所未有的高效和轻松！

无论是应用开发者、运维人员、还是其他信息技术从业人员，都有必要认识和掌握 Docker，节约有限的生命。

本书既适用于具备基础 Linux 知识的 Docker 初学者，也希望可供理解原理和实现的高级用户参考。同时，书中给出的实践案例，可供在进行实际部署时借鉴。

内容特色

- **入门基础**：第 1 ~ 6 章为基础内容，帮助深入理解 Docker 的基本概念 (镜像、容器、仓库) 和核心操作。
- **进阶应用**：第 7 ~ 11 章涵盖 Dockerfile 指令详解、数据与网络管理、Buildx、Compose 等高级配置和管理操作。
- **深入原理**：第 12 ~ 17 章介绍其底层实现技术，深入探讨容器编排体系 (Kubernetes、Etcd)，并延伸涉及容器与云计算及其它关键生态项目 (Fedora CoreOS、Podman 等)。
- **实战扩展**：第 18 ~ 21 章重点讨论容器安全防护机制、监控与日志聚合系统 (Prometheus、ELK)，并展示操作系统、CI/CD 自动化构建等典型实践案例。

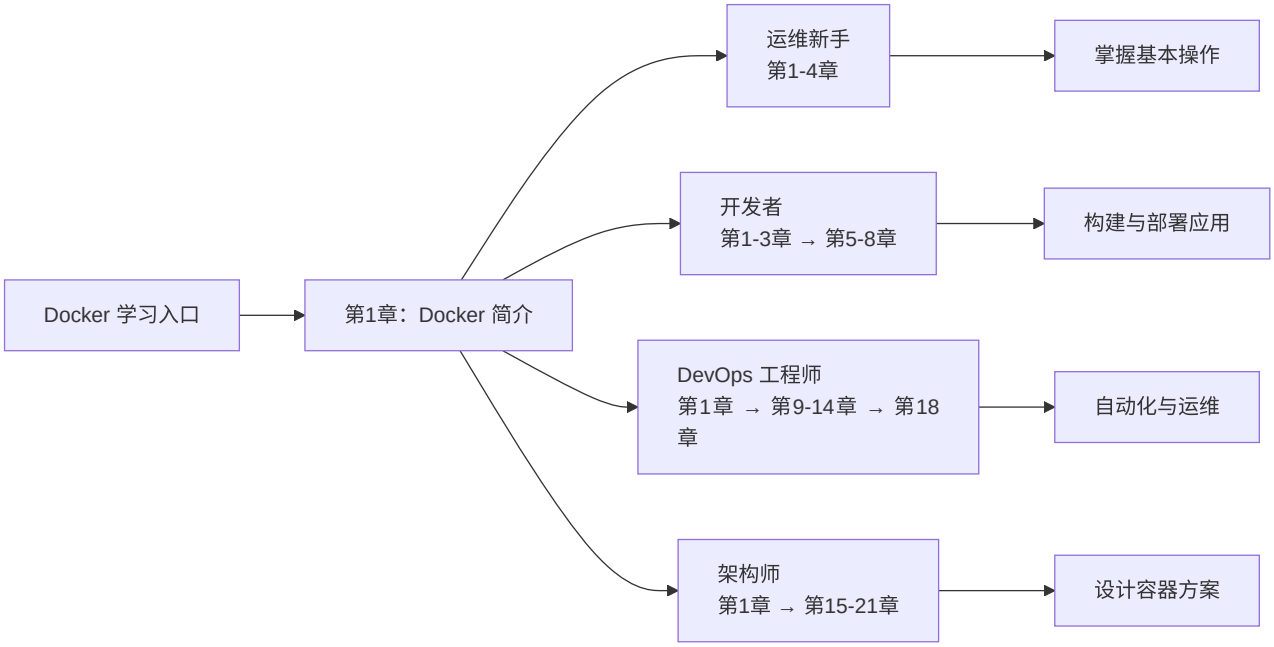
五分钟快速上手

“5分钟运行第一个容器”——跟随以下步骤快速体验 Docker：

1. **安装 Docker**（第3章）：根据操作系统完成 Docker 的安装与验证
2. **第一个容器**（第1章 `1.1`）：快速体验构建镜像与启动容器的完整流程
3. **交互式容器**（第5章）：执行 `docker run -it ubuntu bash`，进入容器内部与系统交互
4. **镜像与仓库**（第2章、第4章、第6章）：理解核心概念，并学会拉取、使用与管理镜像和仓库
5. **自定义镜像**（第7章）：学习如何编写 Dockerfile 创建自己的镜像

Built with mdPress

学习路线图



读者角色	学习重点	核心成果
运维新手	第1-4章	掌握容器的基本概念与操作
开发者	第1-3章 → 第5-8章	学会容器化应用的构建与部署
DevOps 工程师	第1章 → 第9-14章 → 第18章	实现容器编排与自动化部署流程
架构师	第1章 → 第15-21章	设计高可用、高性能的容器基础设施

在线阅读

本书在线阅读，可直接访问 GitBook。也可访问 GitHub 仓库目录 或 镜像站点。

下载离线版本

本书提供 PDF 版本供离线阅读，可前往 GitHub Releases 页面下载最新版本。

如需获取默认分支自动更新的预览版，可直接下载 docker_practice.pdf。该文件会随主线更新覆盖，不代表正式发布版本。

Built with mdPress

本地阅读

先安装 mdPress：

```
brew tap yeasy/tap && brew install mdpress
mdpress serve
```

或使用 Docker 镜像一条命令启动：

```
docker run -it --rm -p 4000:80 ccr.ccs.tencentyun.com/dockerpracticesig/docker_practice:vuepress
```

社区交流

- GitHub Discussions（技术问答、交流）
- GitHub Issues（内容错误、建议）

推荐阅读

本书是技术丛书的一部分。以下书籍与本书形成互补：

书名	与本书的关系
《前线部署工程实践指南》	讲容器、Kubernetes、GitOps、IaC、供应链、安全、观测与 SLO 如何组合成可交付、可审计、可运营的部署工程体系
《智能体 Harness 工程指南》	Agent 基础设施中的容器化部署与隔离
《大模型安全权威指南》	容器安全与 AI 系统安全的交叉实践
《区块链技术指南》	区块链节点的容器化部署

参与贡献

欢迎参与项目维护。

- 修订记录
- 贡献者名单

进阶学习

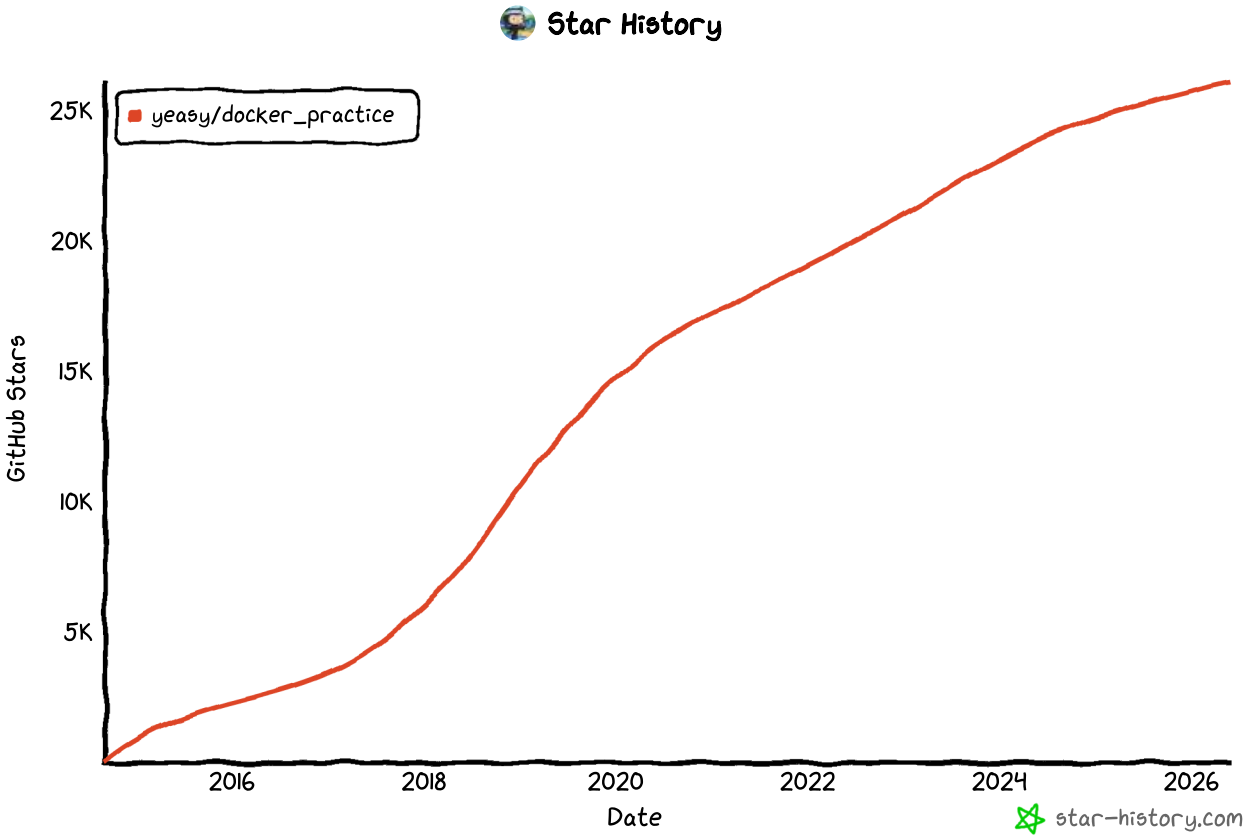
《Docker 技术入门与实战》已更新到第 4 版，讲解最新容器技术栈知识，欢迎大家阅读并反馈建议。京东图书 | 天猫图书

支持鼓励

欢迎鼓励项目一杯 coffee~



Star History



许可证

本书采用 CC BY-NC-SA 4.0 许可证。

您可以自由分享和演绎，但需署名、非商业使用、相同方式共享。

修订记录

- 1.9.2 2026-05-16
  - 刷新 Docker 生态说明，更新 Fedora CoreOS、Podman、containerd 等章节中的维护状态
  - 修复镜像事实、演示 TLS 密钥等后续审阅发现的问题
- 1.9.1 2026-05-08
  - 使用 `browser-actions/setup-chrome` 替代 Ubuntu runner 上不稳定的 Chromium snap 安装
  - 修正 Docker Engine 29 日期、TLS 协议、healthcheck 与交互式调试示例
- 1.9.0 2026-05-02
  - 更新 Docker Engine 29、nginx、MySQL 8.4 LTS、Node.js 22 LTS 与 Kubernetes 相关版本说明
  - 补充 ipvlan、nftables、Gateway API、Docker Scout、镜像安全与供应链安全内容
  - 修复 Docker Hub 限流、etcdctl API、Compose healthcheck、Docker Debug 等时效性内容
- 1.8.0 2026-04-27
  - 补全 Dockerfile 指令参考与多处章节编号、标题层级、代码块和表格格式
  - 增加预览 PDF 自动发布流程，修复 mdpress 端口和导出相关配置
- 1.7.5 2026-04-05
  - 将失效的 AtomHub 镜像替换为可用镜像源
- 1.7.4 2026-03-31
  - 修复标题层级格式
- 1.7.3 2026-03-29
  - 修复 Wikipedia URL 编码
- 1.7.2 2026-03-28

Built with mdPress

    - 修正 macOS、Windows、Compose 与 Kubernetes 章节中的时效性内容和错误前提
    - 收缩越界网络内容，补充 bind mount、tmpfs 与端口映射的关键限制说明
    - 统一 numbered section 的标题层级，清理正文末尾分散的参考资料小节
    - 补充生成物忽略规则，避免 `.mdpress` 与本地 HTML 导出误提交
- 1.7.1 2026-03-28
    - 对齐附录首页与目录结构，补全学习路线入口
    - 重组资源链接页，统一官方一手入口
    - 完善附录二导航页，提升热门镜像查阅体验
- 1.7.0 2026-03-25
    - 精简 CI 流程，移除遗留的 vuepress 构建，统一使用 mdpress
    - 升级 etcd 集群示例从 v3.4.0 到 v3.5.17
    - 更新 npm 镜像为 npmmirror.com，PHP 升级到 8.3
    - 移除 Compose 已废弃的 version 字段
    - 升级所有 CI Actions 到最新版本
- 1.6.1 2026-02-28
    - 修正数据卷 `--mount` 与 `-v` 的行为差异及数据卷管理说明
    - 补充 Docker Hub 限流机制说明，区分 pull rate limit 与 abuse rate limit
    - 完善安全权限警告，强化用户加入 docker 组等同于 root 的风险意识
    - 增补 Docker Engine v29 containerd image store 与 BuildKit provenance attestations 默认行为说明
- 1.6.0 2026-02-20
    - 全面统一使用 `docker compose` (V2) 为默认标准，提供 V1 迁移说明
    - 修复全书大量排版错误，建立附录与正文的双向索引与引用
    - 更新 Kubernetes 至 1.35 兼容说明及运行时环境提示
- 1.5.4 2026-02-15

    - 移除 combine.py
    - 修复若干问题
- 1.5.3 2026-02-15
    - 修复 CI 流程中的图片引用路径错误
    - 修复 CODEOWNERS 文件路径匹配问题
    - 更新项目配置版本号
- 1.5.0 2026-02-05
    - 全面重构章节目录结构 (01-15)
    - 支持 Docker Engine v29.x
    - 优化文档图片引用路径
- 1.4.0 2026-01-11
    - 全面支持 Docker Engine v29 新版本
    - 更新 Docker Compose 至 v2.40.x
    - 更新 Kubernetes 相关章节至 1.35 版本
    - BuildKit 已成为默认稳定构建器，移除实验特性说明
    - 新增 Docker Scout、Docker Init 相关内容
    - 更新镜像加速器配置
    - 添加 CentOS EOL 警告，推荐使用 Rocky Linux/AlmaLinux
    - 扩充安全章节和底层架构章节内容
- 1.3.0 2021-12-31
    - 全面支持 Docker v20.10 新版本
    - 新增 Docker Compose v2
    - Docker Hub 自动构建转为付费功能
- 1.2.0 2020-12-20
    - 错误修复
- 1.1.0 2019-12-31

Built with mdPress

    - 全面支持 Docker v19.03 新版本
    - 增加 `BuildKit`
    - 增加 `docker buildx` 命令使用说明
    - 增加 `docker manifest` 命令使用说明
    - 移除 `Ubuntu 14.04` `Debian 8` `Debian 7`
- 1.0.0: 2018-12-31
    - 全面支持 Docker v18.x 新版本
    - 添加如何调试 Docker
    - 错误修正
- 0.9.0: 2017-12-31
    - 对 v1.13.x 旧版本的最后支持
- 0.9.0-rc2: 2017-12-10

    - 增加 Docker 中文资源链接
    - 增加介绍基于 Docker 的 CI/CD 工具 `Drone`
    - 增加 `docker secret` 相关内容
    - 增加 `docker config` 相关内容
    - 增加 `LinuxKit` 相关内容
    - 更新 `CoreOS` 章节
    - 更新 `etcd` 章节，基于 3.x 版本
    - 删除 `Docker Compose` 中的 `links` 指令
    - 替换 `docker daemon` 命令为 `dockerd`
    - 替换 `docker ps` 命令为 `docker container ls`
    - 替换 `docker images` 命令为 `docker image ls`
    - 修改 `安装 Docker` 一节中部分文字表述
    - 移除历史遗留文件和错误的文件
    - 优化文字排版
    - 调整目录结构
    - 修复内容逻辑错误
    - 修复 `404` 链接
- 0.9.0-rc1: 2017-11-29

Built with mdPress

    - 根据最新版本 (v17.09) 修订内容
    - 增加 Dockerfile 多阶段构建 (multistage builds) Docker 17.05 新增特性
    - 增加 docker exec 子命令介绍
    - 增加 docker 管理子命令 container image network volume 介绍
    - 增加 树莓派单片电脑 安装 Docker
    - 增加 Docker 存储驱动 OverlayFS 相关内容
    - 更新 Docker CE v17.x 安装说明
    - 更新 Docker 网络 一节
    - 更新 Docker Machine 基于 0.13.0 版本
    - 更新 Docker Compose 基于 3 文件格式
    - 删除 Docker Swarm 相关内容，替换为 Swarm mode Docker 1.12.0 新增特性
    - 删除 docker run --link 参数
    - 精简 Docker Registry 一节
    - 替换 docker run -v 参数为 --mount
    - 修复 404 链接
    - 优化文字排版
    - 增加离线阅读功能
- 0.8.0: 2017-01-08
    - 修正文字内容
    - 根据最新版本 (1.12) 修订安装使用
    - 补充附录章节
- 0.7.0: 2016-06-12
    - 根据最新版本进行命令调整
    - 修正若干文字描述
- 0.6.0: 2015-12-24
    - 补充 Machine 项目
    - 修正若干 bug

Built with mdPress

- 0.5.0: 2015-06-29
    - 添加 Compose 项目
    - 添加 Machine 项目
    - 添加 Swarm 项目
    - 完善 Kubernetes 项目内容
    - 添加 Mesos 项目内容
- 0.4.0: 2015-05-08
    - 添加 Etcd 项目
    - 添加 Fig 项目
    - 添加 CoreOS 项目
    - 添加 Kubernetes 项目
- 0.3.0: 2014-11-25
    - 完成仓库章节
    - 重写安全章节
    - 修正底层实现章节的架构、命名空间、控制组、文件系统、容器格式等内容
    - 添加对常见仓库和镜像的介绍
    - 添加 Dockerfile 的介绍
    - 重新校订中英文混排格式
    - 修订文字表达
    - 发布繁体版本分支：zh-Hant
- 0.2.0: 2014-09-18
    - 对照官方文档重写介绍、基本概念、安装、镜像、容器、仓库、数据管理、网络等章节
    - 添加底层实现章节
    - 添加命令查询和资源链接章节
    - 其它修正
- 0.1.0: 2014-09-05

Built with mdPress

- 添加基本内容
- 修正错别字和表达不通顺的地方

如何贡献

领取或创建新的 Issue，如 issue 235，添加自己为 Assignee。

在 GitHub 上 fork 到自己的仓库，如 docker_user/docker_practice，然后 clone 到本地，并设置用户信息。

```
$ git clone git@github.com:docker_user/docker_practice.git

$ cd docker_practice
```

修改代码后提交，并推送到自己的仓库，注意修改提交消息为对应 Issue 号和描述。

```
# Update the content

$ git commit -a -s

# In commit msg dialog, add content like "Fix issue #235: describe ur change"

$ git push
```

在 GitHub 上提交 Pull Request，添加标签，并邀请维护者进行 Review。

定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/docker_practice

$ git fetch upstream

$ git rebase upstream/master

$ git push -f origin master
```

排版规范

本开源书籍遵循中文排版指南规范。

第一章 Docker 简介

本章将带领你进入 **Docker** 的世界。

> **版本提示**：本书内容及示例基于 **Docker Engine v29.x** 及以上版本。值得注意的是，自 Docker Engine v29 起，官方在全新安装场景下 **默认启用 `containerd image store` 作为镜像存储后端**（取代传统 classic store 路径下的 graph driver 体系）。这项底层革新极大增强了 Docker 对多架构镜像（Multi-platform）以及软件供应链安全元数据（Attestations, SBOM, Provenance）的本地支持原生性。

本章内容

- 快速上手
  - 通过一个简单的 Web 应用例子，带你快速体验 Docker 的核心流程：构建镜像、运行容器。
- 什么是 Docker
  - 介绍 Docker 的起源、发展历程以及其背后的核心技术 (Cgroups，Namespaces，UnionFS，以及 `containerd` 引擎的演进)。
  - 了解 Docker 是如何改变软件交付方式的。
- 为什么要用 Docker
  - 对比传统虚拟机技术，阐述 Docker 在启动速度、资源利用率、交付效率等方面的巨大优势。
  - 探讨 Docker 在 DevOps、微服务架构中的关键作用。

学习目标

通过本章的学习，你将能够：

1. 理解 Docker 的核心概念与架构。
2. 明白 Docker 解决了现代软件开发与运维中的哪些痛点。
3. 建立起对容器技术的初步认知，为后续的实战操作打下基础。

Built with mdPress

好吧，让我们带着问题开始这神奇之旅。

1.1 快速上手

> **版本说明**：本节示例基于 Docker v29.x 编写。示例中使用的 `nginx:alpine` 镜像标签为演示用途，请查阅 Docker Hub - nginx 确认最新可用版本。

本节将通过一个简单的 Web 应用例子，带你快速体验 Docker 的核心流程：构建镜像、运行容器。

为什么选择 Nginx + HTML 作为入门例子?

在学习 Docker 之前，我们先来理解为什么这个例子是最适合初学者的。Docker 的核心价值在于**一致性交付**——无论你在本地、云端还是他人的机器上运行容器，应用的行为都是完全一致的。这个 Nginx + 静态 HTML 的例子之所以被广泛采用，是因为它展现了 Docker 工作流的三个核心阶段：

1. **镜像定义（Image Layer）**：通过 Dockerfile 描述如何把应用打包成一个自包含的单元
2. **镜像构建（Build）**：执行 `docker build`，Docker 根据 Dockerfile 逐层构建镜像
3. **容器运行（Runtime）**：通过 `docker run` 启动容器实例，应用真正开始提供服务

Nginx 是一个轻量级、使用广泛的 Web 服务器，学习完这个例子后，你可以轻松扩展到部署 Node.js、Python、Go 等任何语言的应用。

1.1.1 准备代码

创建一个名为 `hello-docker` 的文件夹，并在其中创建一个 `index.html` 文件：

```
<h1>Hello, Docker!</h1>
```

1.1.2 编写 Dockerfile

在同级目录下创建一个名为 `Dockerfile` (无后缀) 的文件：

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

1.1.3 构建镜像

打开终端，进入该目录，执行构建命令：

```
$ docker build -t my-hello-world .
```

- `docker build`：构建命令
- `-t my-hello-world`：给镜像起个名字 (标签)
- `.`：指定上下文路径为当前目录

1.1.4 运行容器

使用刚才构建的镜像启动一个容器：

```
$ docker run -d -p 8080:80 my-hello-world
```

- `docker run`：运行命令
- `-d`：后台运行
- `-p 8080:80`：将宿主机的 8080 端口映射到容器的 80 端口

1.1.5 访问测试

打开浏览器访问 http://localhost:8080，你应该能看到 “Hello, Docker!”。

1.1.6 清理

停止并删除容器：

```
# 查看正在运行的容器 ID

$ docker ps

# 停止容器

$ docker stop <CONTAINER_ID>

# 删除容器

$ docker rm <CONTAINER_ID>
```

恭喜！你已经完成了第一次 Docker 实战。接下来请阅读 Docker 核心概念做深入了解。

1.2 什么是 Docker

Docker 是彻底改变了软件开发和交付方式的革命性技术。本节将从核心概念、与传统虚拟机的对比、技术基础以及历史生态等多个维度，带你深入理解什么是 Docker。

1.2.1 一句话理解 Docker

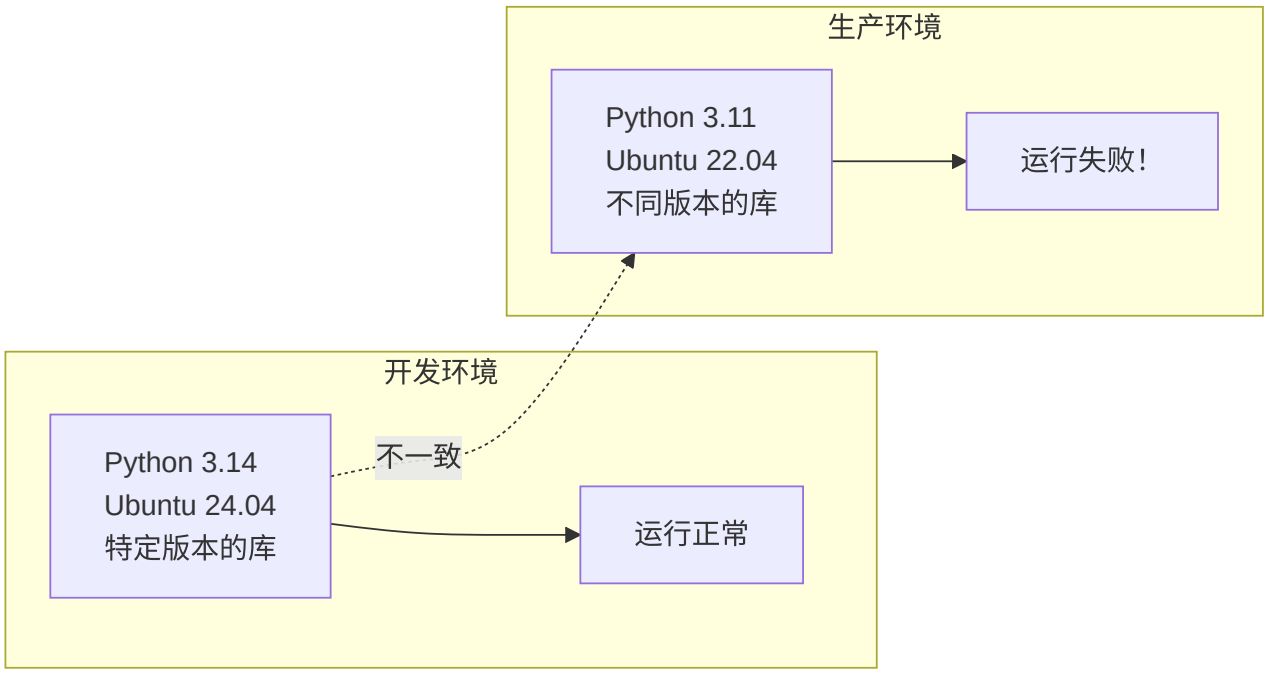
> **Docker 是一种轻量级的虚拟化技术，它让应用程序及其依赖环境可以被打包成一个标准化的单元，在任何地方都能一致地运行。** 如果用一个生活中的类比：**Docker 之于软件，就像集装箱之于货物。**

在集装箱发明之前，货物的运输是一件麻烦的事情——不同的货物需要不同的包装、不同的装卸方式，换一种运输工具就要重新装卸。集装箱的出现改变了这一切：无论里面装的是什么，集装箱的外形是标准的，可以用同样的方式装卸、堆放和运输。

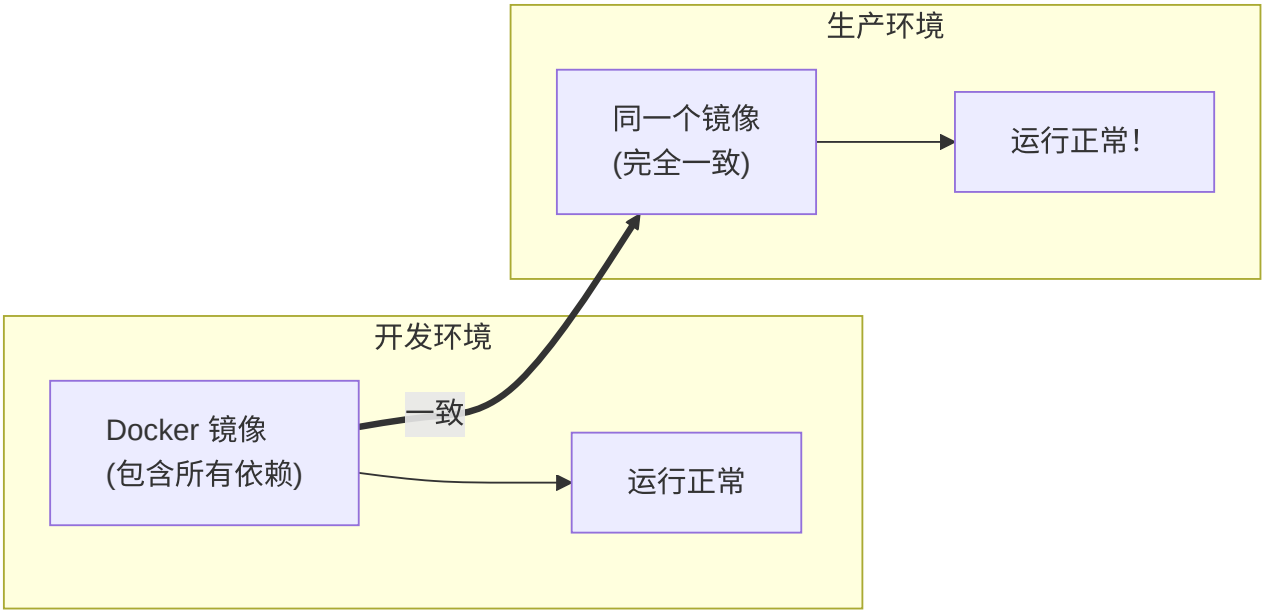
Docker 做的事情类似：无论你的应用是用 Python、Java、Node.js 还是其他语言写的，无论它需要什么样的依赖库和环境，一旦被打包成 Docker 镜像，就可以用同样的方式在任何支持 Docker 的机器上运行。

1.2.2 Docker 的核心价值

笔者认为，Docker 解决的是软件开发中最古老的问题之一：**“在我机器上明明能跑啊！”**



有了 Docker：



1.2.3 Docker vs 虚拟机

很多人第一次接触 Docker 时会问：“**这不就是虚拟机吗？**” 答案是：**不是，而且差别很大。**

传统虚拟机

传统虚拟机技术是虚拟出一套完整的硬件，在其上运行一个完整的操作系统，再在该系统上运行应用：


传统虚拟化

Docker 容器

而 Docker 容器内的应用直接运行于宿主的内核，容器内没有自己的内核，也没有进行硬件虚拟：


Docker

Built with mdPress

关键区别

特性	Docker 容器	传统虚拟机
启动速度	秒级	分钟级
资源占用	MB 级别	GB 级别
性能	接近原生	有明显损耗
隔离级别	进程级隔离	完全隔离
单机数量	可运行上千个	通常几十个

> 笔者经常用这个类比来解释：虚拟机像是每个应用都住在一栋独立的房子里 (有自己的地基、水电系统)，而容器像是大家住在同一栋公寓楼里的不同房间 (共享地基和水电系统，但各自独立)。

1.2.4 Docker 的技术基础

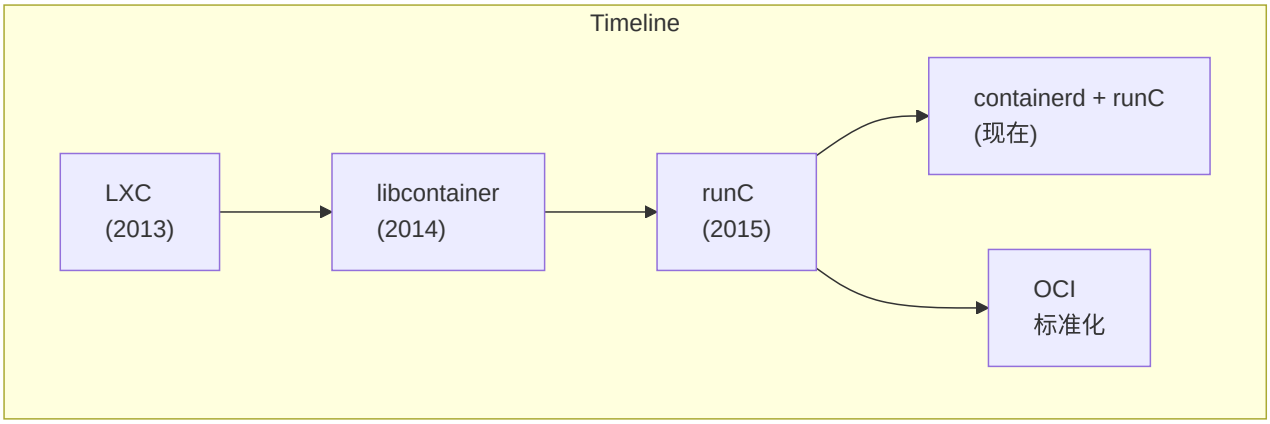
Docker 使用 Go 语言开发，基于 Linux 内核的以下技术：

- **Namespace**：实现资源隔离 (进程、网络、文件系统等)
- **Cgroups**：实现资源限制 (CPU、内存、I/O 等)
- **Union FS**：实现分层存储 (如 OverlayFS)

> 如果你对这些底层技术感兴趣，可以阅读本书的底层实现章节。

Docker 架构演进

Docker 的底层实现经历了多次演进：



- **LXC** (2013)：Docker 最初基于 Linux Containers
- **libcontainer** (2014，Docker 0.9)：Docker 自研的容器运行时
- **runC** (2015，Docker 1.11 整合)：捐献给 OCI 的标准容器运行时
- **containerd**：高级容器运行时，管理容器生命周期

Docker 架构

> `runc` 是一个 Linux 命令行工具，用于根据 OCI 容器运行时规范创建和运行容器。

> `containerd` 是一个守护程序，它管理容器生命周期，提供了在一个节点上执行容器和管理镜像的最小功能集。

1.2.5 Docker 的历史与生态

Docker 最初是 `dotCloud` 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目，于 2013 年 3 月以 Apache 2.0 授权协议开源。

Docker 的发展历程：

- **2013 年 3 月**：开源发布
- **2013 年底**：dotCloud 公司改名为 Docker，Inc。
- **2015 年**：成立开放容器联盟 (OCI)，推动容器标准化
- **至今**：GitHub 项目超过 7 万星标

Docker 的成功推动了整个容器生态的发展，催生了 Kubernetes、Podman 等众多相关项目。笔者认为，Docker 最大的贡献不仅是技术本身，更是它 **让容器技术从系统管理员的工具变成了每个开发者都能使用的标准工具**。

1.3 为什么要用 Docker

在回答 “为什么用 Docker” 之前，笔者想先问一个问题：**你有没有经历过这些场景？**

1.3.1 没有 Docker 的世界

在 Docker 出现之前，软件开发和运维面临着诸多棘手的问题。我们先来看看以下三个典型的痛点场景。

场景一：“在我电脑上明明能跑”

```
周五下午 5:00
├── 开发者：代码写完了，本地测试通过，提交！🎉
├── 周一早上 9:00
│   └── 测试："这个功能在测试环境跑不起来"
└── 开发者："不可能，在我电脑上明明能跑啊……"
```

笔者统计过，这个问题通常由以下原因导致：

- Python/Node/Java 版本不一致
- 依赖库版本不一致
- 操作系统配置不一致
- 某些环境变量没有设置
- “哦，忘了说我本地装了个 XXX”

场景二：环境配置的噩梦

```
新同事入职
├── Day 1：领电脑，配环境
├── Day 2：继续配环境，遇到问题
├── Day 3：换种方法配环境
├── Day 4：问老同事怎么配的，他也忘了
└── Day 5：终于能跑起来了！但不知道为什么……
```

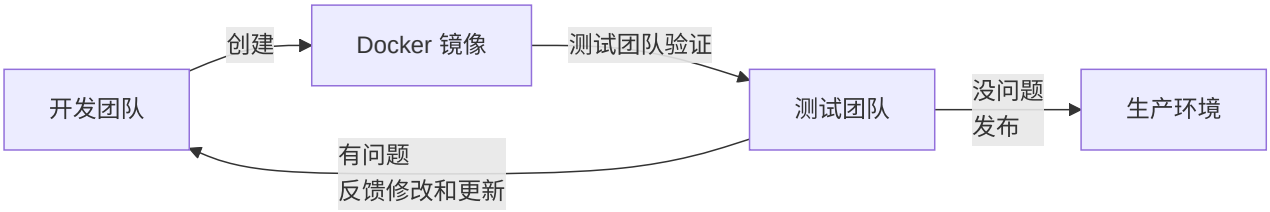
场景三：服务器迁移的恐惧

```
运维："我们需要把服务迁移到新服务器"
开发："旧服务器上的配置文档在哪？"
运维："当时是一个已经离职的同事配的……"
所有人：😱
```

1.3.2 Docker 如何解决这些问题

Docker 的出现为上述问题提供了完美的解决方案。它通过 “一次构建，到处运行” 的核心理念，从根本上改变了软件交付的方式。

核心理念：一次构建，到处运行



1.3.3 Docker 的核心优势

除了解决上述痛点，Docker 还拥有诸多显著的技术优势，包括环境一致性、秒级启动、高效的资源利用等。

1. 环境一致性

Docker 镜像包含了应用运行所需的 **一切**：代码、运行时、系统工具、库、配置。这意味着：

- ✅ 开发环境和生产环境完全一致
- ✅ 不会再有 “在我机器上能跑” 的问题
- ✅ 新人入职，一条命令就能启动开发环境

```
## 新同事入职第一天

$ git clone https://github.com/company/project.git
$ docker compose up

## 完整的开发环境就准备好了

...
```

2. 秒级启动

传统虚拟机启动需要几分钟 (引导操作系统)，而 Docker 容器启动通常只需要 **几秒甚至几百毫秒**。

笔者实测数据：

启动内容	虚拟机	Docker 容器
空系统	~60 秒	~0.5 秒
MySQL	~90 秒	~3 秒
完整 Web 应用	~120 秒	~5 秒

这个差异对以下场景尤为重要：

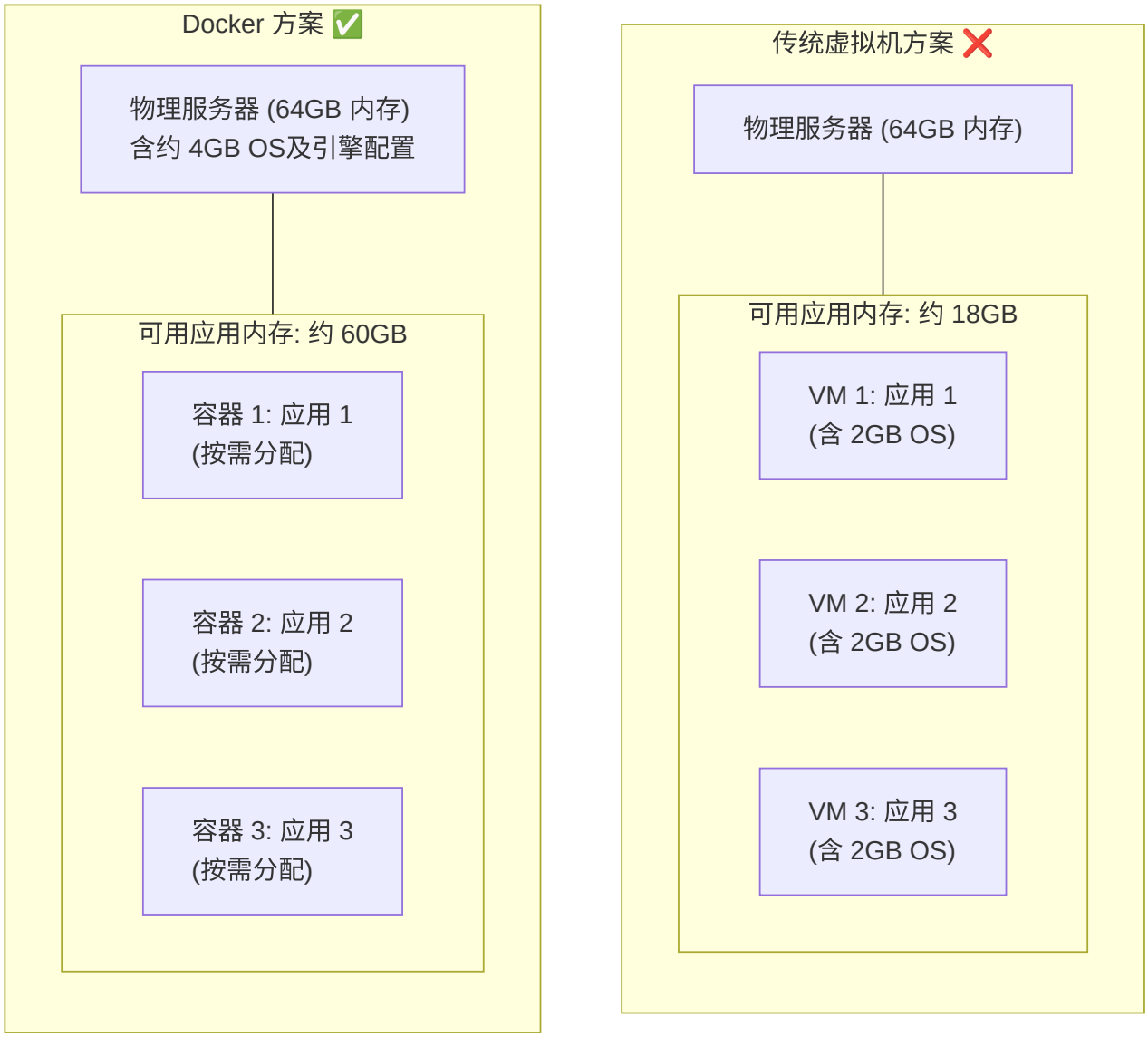
- **CI/CD 流水线**：每次构建节省几分钟，一天累积下来就是几小时
- **弹性扩容**：流量高峰时能快速启动更多实例
- **开发体验**：快速重启服务进行调试

3. 资源效率

Docker 容器共享宿主机内核，无需为每个应用运行完整的操作系统。以一台 64GB 内存的物理服务器为例：

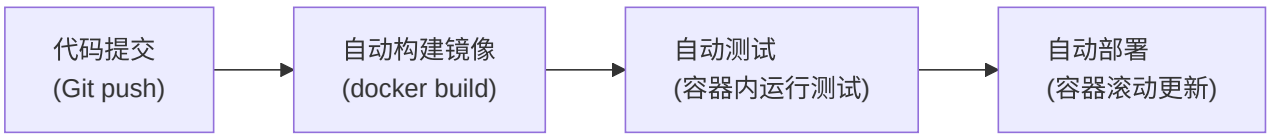
- **传统虚拟机方案**：每个虚拟机都需要运行完整的操作系统（每个额外占用如 2GB 内存），产生大量资源开销，实际可用于应用的内存可能只有约 18GB。
- **Docker 方案**：容器直接共享宿主机系统，只需付出很少的基础开销（OS 及引擎约 4GB），即可将约 60GB 的内存全部用于实际应用。

Built with mdPress



4. 持续交付和部署

Docker 完美契合 DevOps 的工作流程：



使用 Dockerfile 定义镜像构建过程，使得：

- 构建过程 **可重复、可追溯**
- 任何人都能从代码重建完全相同的镜像
- 配合 GitHub Actions 等 CI 系统实现自动化

Built with mdPress

5. 轻松迁移

Docker 可以在几乎任何平台上运行：

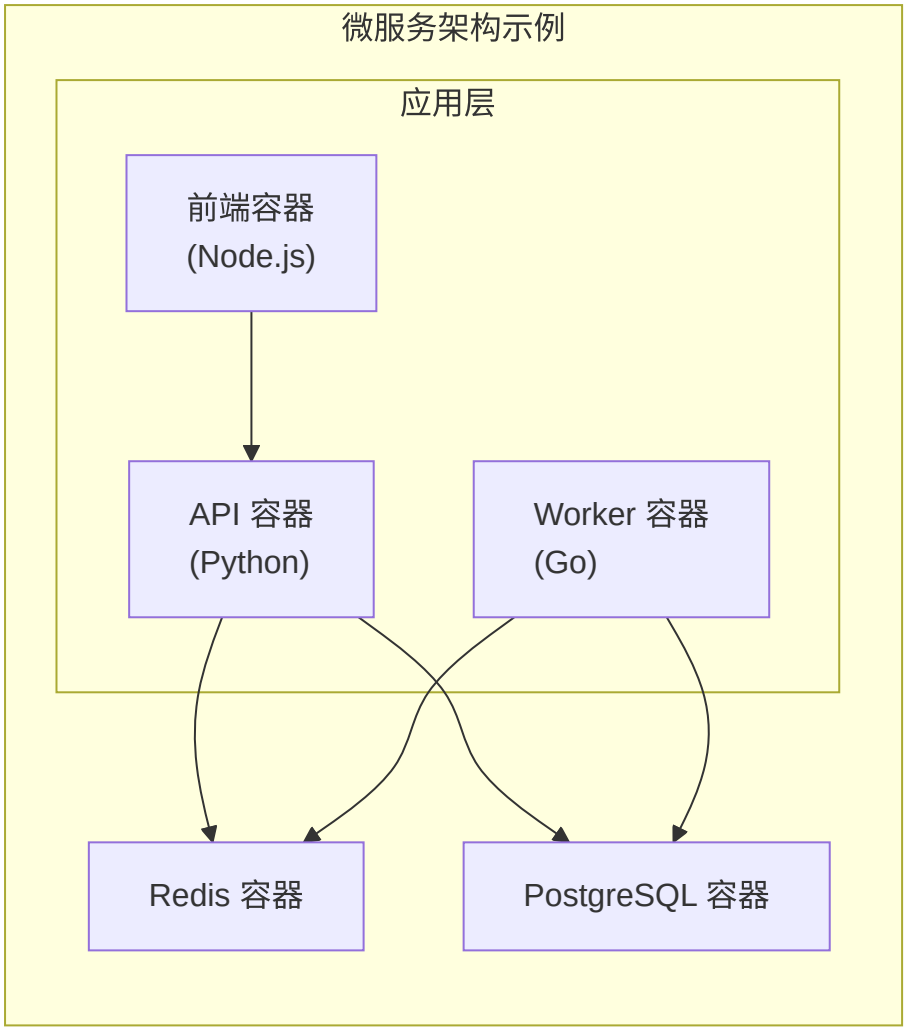
- ✅ 本地开发机 (macOS、Windows、Linux)
- ✅ 公有云 (AWS、Azure、GCP、阿里云、腾讯云)
- ✅ 私有云和自建数据中心
- ✅ 边缘设备和 IoT

同一个镜像，在任何地方运行结果都一致。 这让应用迁移变得前所未有的简单。

6. 微服务架构的基石

现代微服务架构几乎都依赖容器技术。Docker 让你可以：

- **隔离服务**：每个服务运行在独立容器中，互不干扰
- **独立扩展**：哪个服务负载高，就单独扩展哪个
- **独立部署**：更新一个服务不影响其他服务
- **技术多样**：不同服务可以用不同语言和框架



1.3.4 Docker 不适合的场景

笔者认为，技术选型要客观。Docker 并非银弹，以下场景可能不太适合：

- **需要完全隔离的场景**：容器共享宿主机内核，隔离性不如虚拟机。如果需要运行不受信任的代码，虚拟机可能更安全。
- **需要特殊内核的场景**：容器使用宿主机内核。如果应用需要特定版本的内核或内核模块，可能需要虚拟机。
- **Windows 原生应用**：虽然 Docker 支持 Windows 容器，但生态不如 Linux 容器成熟。传统 Windows 应用的容器化仍有挑战。
- **桌面应用**：Docker 主要面向服务端应用。桌面 GUI 应用的容器化虽然可行，但通常得不偿失。

1.3.5 与传统虚拟机的对比总结

关于容器与虚拟机的详细特性对比，请参阅 1.2.3 Docker vs 虚拟机 中的对比表。总结来说：

Built with mdPress

- **性能差异**：虚拟机通常有 5-20% 的性能损耗，而容器接近原生性能。
- **最佳场景**：Docker 容器适合微服务、CI/CD、开发环境；虚拟机适合多租户、高安全需求场景。

本章小结

- Docker 是一种轻量级虚拟化技术，核心价值是 **环境一致性**
- 与虚拟机相比，Docker 更轻量、更快速、资源利用率更高
- Docker 基于 Linux 内核的 Namespace、Cgroups 和 Union FS 技术
- Docker 推动了容器技术的标准化 (OCI) 和生态发展

Docker 的核心价值可以用一句话概括：**让应用的开发、测试、部署保持一致，同时极大提高资源利用效率。** 笔者认为，对于现代软件开发者来说，Docker 已经不是 “要不要学” 的问题，而是 **必备技能**。无论你是前端、后端、运维还是全栈开发者，掌握 Docker 都能让你的工作更高效。

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第二章 基本概念

> **版本说明**：本章内容及示例基于 Docker v29.x 编写。镜像标签相关示例请查阅官方文档以确认最新可用版本。

Docker 包括三个基本概念：

- **镜像** (`Image`)：Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数 (如匿名卷、环境变量、用户等)。镜像不包含任何动态数据，其内容在构建之后也不会被改变。
- **容器** (`Container`)：镜像 (`Image`) 和容器 (`Container`) 的关系，就像是面向对象程序设计中的 *类* 和 *实例* 一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。
- **仓库** (`Repository`)：镜像构建完成后，可以很容易的在当前宿主机上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。

理解了这三个概念，就理解了 **Docker** 的整个生命周期。

- Docker 镜像
- Docker 容器
- Docker 仓库

2.1 镜像

> **版本说明**：本节示例基于 Docker v29.x 编写。示例中使用的镜像标签（如 `ubuntu:24.04`、`nginx:latest`）为演示用途，建议查阅 Docker Hub 或相应镜像的官方页面确认最新可用版本。

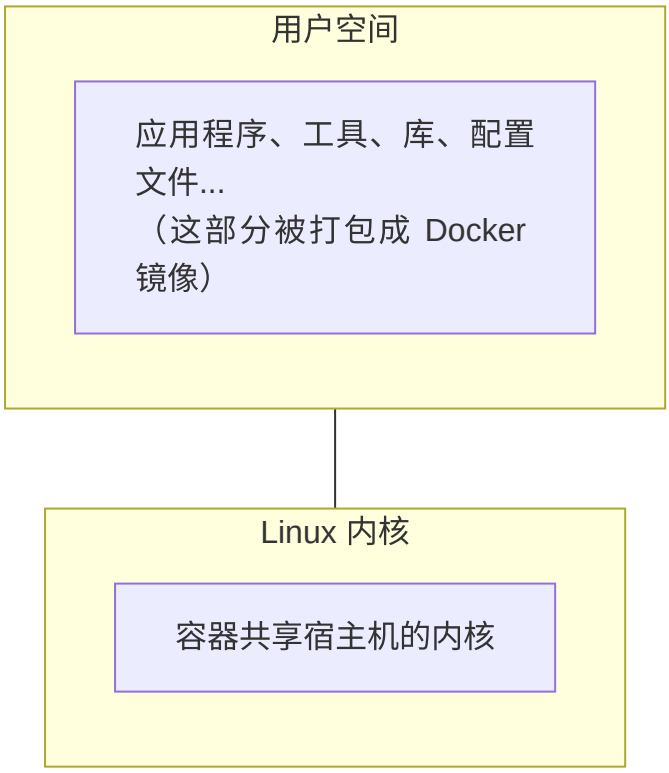
Docker 镜像作为容器运行的基石，其设计理念和实现机制至关重要。本节将深入探讨镜像的本质、与操作系统的关系、内容构成以及核心的分层存储机制。

2.1.1 一句话理解镜像

> **Docker 镜像是一个只读的模板，包含了运行应用所需的一切：代码、运行时、库、环境变量和配置文件。** 如果用一个类比：**镜像就像是一张光盘或 ISO 文件**。你可以用同一张光盘在不同电脑上安装系统，而光盘本身不会被修改。同样，一个镜像可以创建多个容器，而镜像本身保持不变。

2.1.2 镜像与操作系统的关系

我们都知道，操作系统分为 **内核** 和 **用户空间**：



对于 Linux 而言，内核启动后会挂载 `root` 文件系统来提供用户空间支持。**Docker 镜像** 本质上就是一个 `root` 文件系统。

例如，官方镜像 `ubuntu:24.04` 包含了一套完整的 Ubuntu 24.04 最小系统的 root 文件系统——但**不包含 Linux 内核** (因为容器共享宿主机的内核)。

2.1.3 镜像包含什么?

Docker 镜像是一个特殊的文件系统，包含：

内容类型	示例
程序文件	应用二进制文件、Python/Node 解释器
库文件	libc、OpenSSL、各种依赖库
配置文件	nginx.conf、my.cnf 等
环境变量	PATH、LANG 等预设值
元数据	启动命令、暴露端口、数据卷定义

关键特性：

- ✅ 镜像是 **只读** 的
- ✅ 镜像 **不包含** 动态数据
- ✅ 镜像构建后 **内容不会改变**

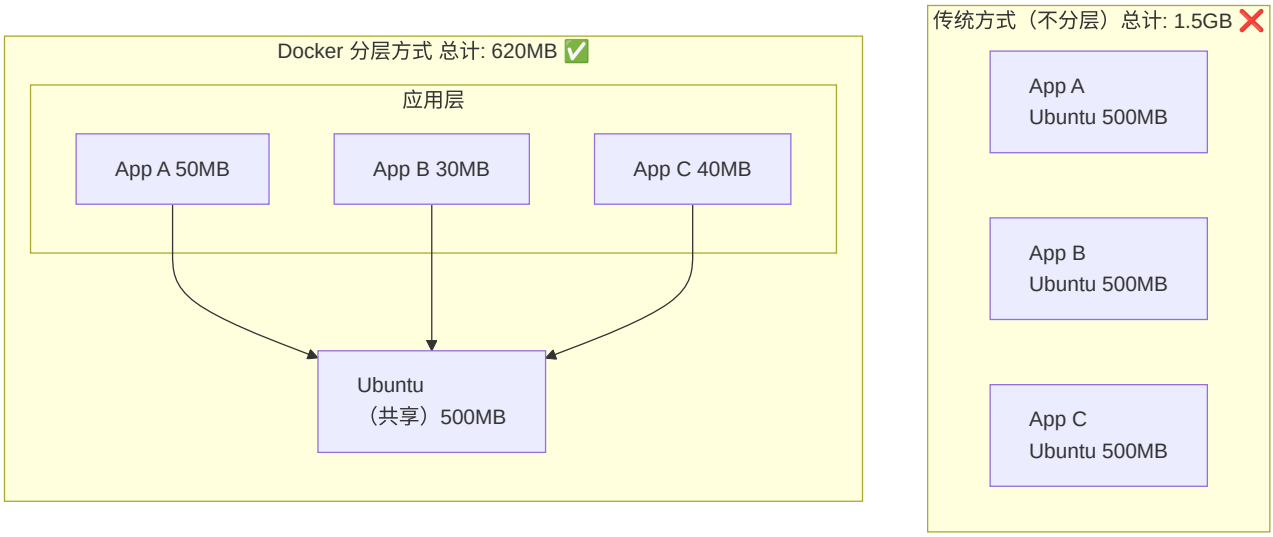
2.1.4 分层存储：镜像的核心设计

镜像的分层存储机制是 Docker 最具创新性的特性之一。通过 Union FS 技术，Docker 能够高效地构建和管理镜像。

为什么需要分层?

笔者认为，分层存储是 Docker 最巧妙的设计之一。

假设你有三个应用，都基于 Ubuntu 运行：



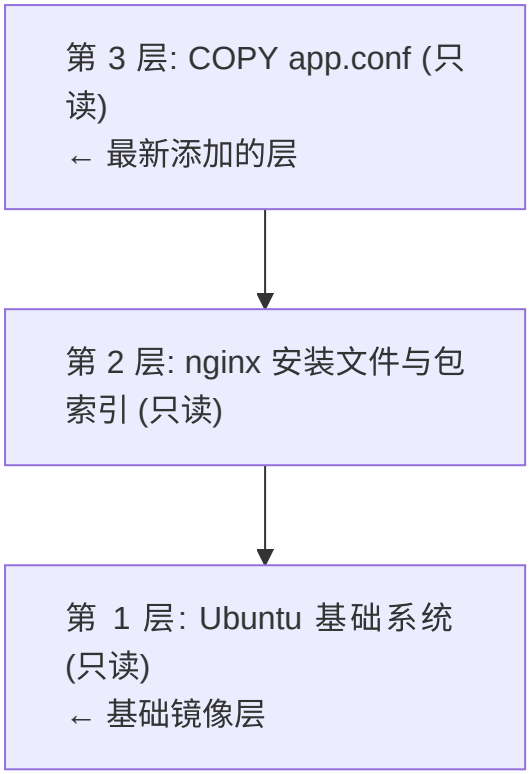
分层是如何工作的?

笔者用一个更贴近最佳实践的 Dockerfile 来解释分层：

```
FROM ubuntu:24.04                          # 第 1 层：基础系统（约 78MB）
RUN apt-get update && apt-get install -y nginx # 第 2 层：更新索引并安装 nginx
COPY app.conf /etc/nginx/                  # 第 3 层：复制配置文件
```

这里特意把 `apt-get update` 和 `apt-get install` 放在同一个 `RUN` 里，避免构建缓存导致包索引过期。

构建后的镜像结构：



每一层的特点：

- **只读**：构建完成后不可修改
- **可共享**：多个镜像可以共享相同的层
- **有缓存**：未变化的层不会重新构建

分层存储的 “陷阱”

> ⚠️ **笔者特别提醒**：理解这一点可以帮你避免构建出臃肿的镜像。**关键原理**：每一层的文件变化会被记录，但 **删除操作只是标记，不会真正减小镜像体积**。

```
## 错误示范 ❌

FROM ubuntu:24.04
RUN apt-get update
RUN apt-get install -y build-essential  # 安装编译工具（约 200MB）
RUN make && make install                # 编译应用
RUN apt-get remove build-essential      # 试图删除编译工具

## 结果：镜像仍然包含 200MB 的编译工具！
```

```
## 正确做法 ✅

FROM ubuntu:24.04
RUN apt-get update && \
    apt-get install -y build-essential && \
    make && make install && \
    apt-get remove -y build-essential && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*

## 在同一层完成安装、使用、清理
```

2.1.5 查看镜像的构建历史

```
## 查看镜像的历史（每层的构建记录）

$ docker history nginx:latest

IMAGE          CREATED       CREATED BY                                      SIZE
a6bd71f48f68   2 weeks ago   CMD ["nginx" "-g" "daemon off;"]                0B
<missing>      2 weeks ago   STOPSIGNAL SIGQUIT                              0B
<missing>      2 weeks ago   EXPOSE map[80/tcp:{}]                           0B
<missing>      2 weeks ago   ENTRYPOINT ["/docker-entrypoint.sh"]            0B
<missing>      2 weeks ago   COPY 30-tune-worker-processes.sh /docker-ent…   4.62kB
...
```

需要注意：`docker history` 展示的是镜像的构建历史。像 `CMD`、`ENTRYPOINT`、`EXPOSE`、`STOPSIGNAL` 这类 `0B` 项主要是镜像配置元数据，并不等同于新增了可见的文件系统内容；真正占用空间的通常是 `RUN`、`COPY`、`ADD` 等带来的文件变化。

2.1.6 镜像的标识

Docker 镜像有多种标识方式：

1. 镜像名称和标签

格式：`[仓库地址/]仓库名[:标签]`

```
## 完整格式

registry.example.com/myproject/myapp:v1.2.3

## 简写（使用 Docker Hub）

nginx:1.28
ubuntu:24.04

## 省略标签（默认使用 latest）

nginx  # 等同于 nginx:latest
```

2. 镜像 ID：Content-Addressable 标识

每个镜像有一个基于内容计算的唯一 ID：

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
nginx        latest    a6bd71f48f68   2 weeks ago    187MB
ubuntu       24.04     ca2b0f26964c   3 weeks ago    78.1MB
```

3. 镜像摘要

更精确的标识，基于镜像内容的 SHA256 哈希：

```
$ docker images --digests
REPOSITORY  TAG     DIGEST                                                                  IMAGE
ID
nginx       latest  sha256:6db391d1c0cfb30588ba0bf72ea999404f2764184d8b8d10d89e8a9c6... a6bd71f48f68
```

> 💡 笔者建议：在生产环境使用镜像摘要而非标签，因为标签可以被覆盖，但摘要是不可变的。

2.1.7 镜像的来源

Docker 镜像可以通过以下方式获取：

方式	说明	示例
从 Registry 拉取	最常用的方式	`docker pull nginx`
从 Dockerfile 构建	自定义镜像	`docker build -t myapp .`
从容器提交	保存容器状态 (不推荐)	`docker commit`
从文件导入	离线传输	`docker load < image.tar`

2.2 容器

容器是 Docker 技术的核心，是应用实际运行的载体。本节将从容器的本质、与虚拟机的区别、存储层机制以及生命周期管理等方面，全面解析 Docker 容器。

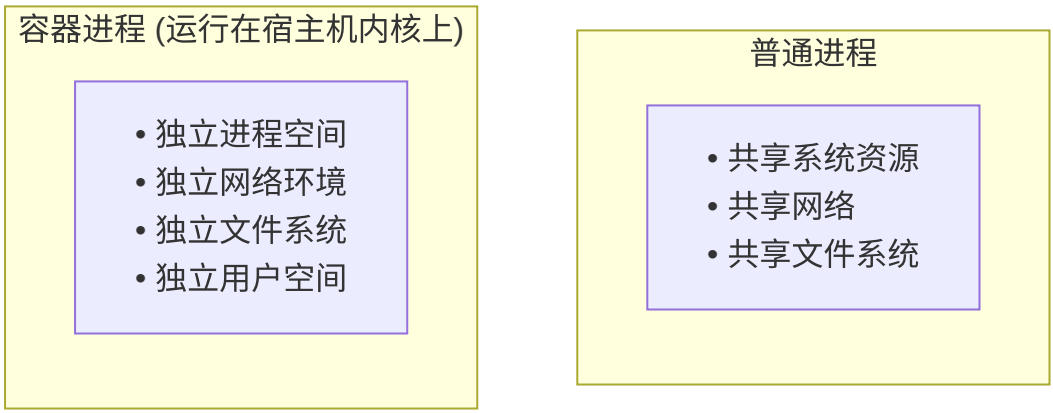
2.2.1 一句话理解容器

> **容器是镜像的运行实例。如果把镜像比作程序，那么容器就是进程。** 用面向对象编程的术语来说：**镜像是类 (Class)，容器是对象 (Instance)**。

- 一个镜像可以创建多个容器
- 每个容器相互独立，互不影响
- 容器可以被创建、启动、停止、删除、暂停

2.2.2 容器的本质

> 💡 笔者认为，理解这一点是理解 Docker 的关键：**容器的本质是一个特殊的进程**。

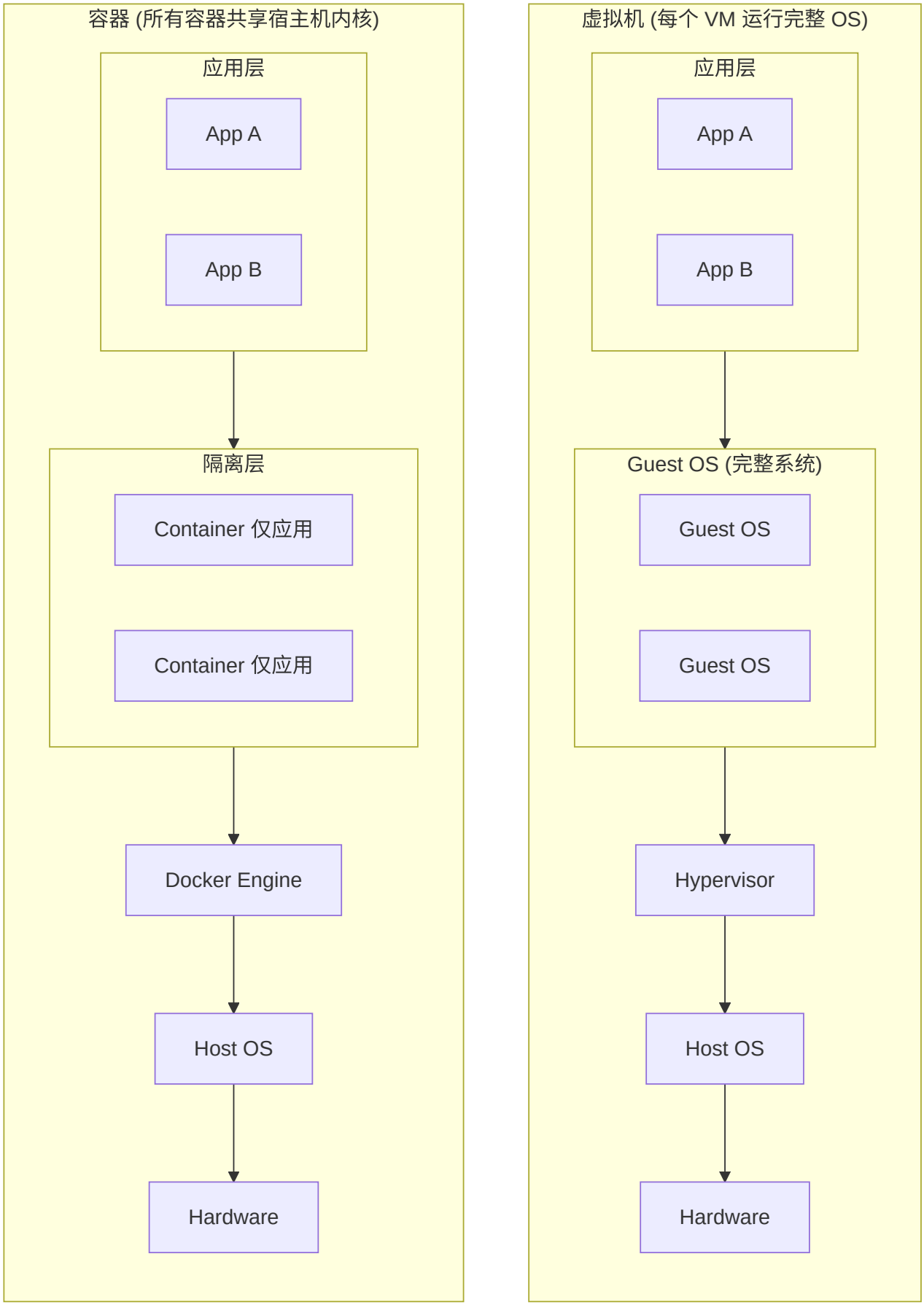


这种隔离主要通过 Linux 内核的 **Namespace** 实现，资源限制通常与 **cgroups** 配合。具体表现为：

- **进程空间**：容器看不到宿主机上的其他进程。
- **网络**：在默认网络模式下，容器通常拥有独立的网络命名空间，并可分配独立 IP；使用 `host` 或 `container:` 等模式时则例外。
- **文件系统**：容器拥有独立的 root 目录。
- **用户**：默认情况下，容器内的 `root` 仍是 `uid 0`，但通常只拥有受限 capabilities；如果启用 `userns-remap` 或 rootless 等机制，还会进一步映射为宿主机上的低权限用户。

2.2.3 容器 vs 虚拟机：核心区别

很多初学者会混淆容器和虚拟机。笔者用一张图来说明：



Built with mdPress

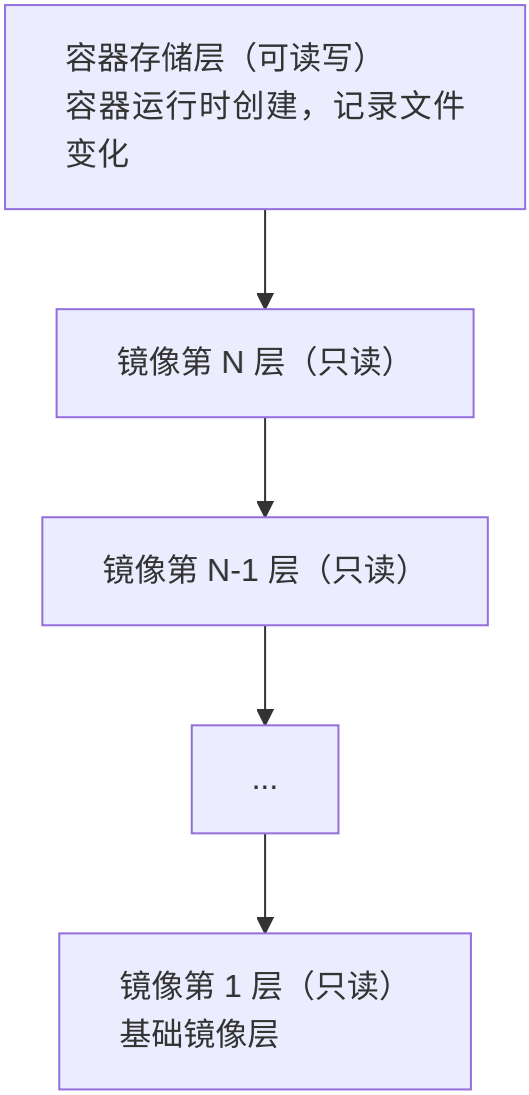
特性	容器	虚拟机
隔离级别	操作系统级 (Namespaces/cgroups)	硬件虚拟化级 (Hypervisor)
启动时间	通常更快	通常更慢
资源占用	通常更低	通常更高
运行开销	通常更接近原生	通常有更高虚拟化开销
内核	共享宿主机内核	各自独立内核

2.2.4 容器的存储层

理解容器的存储层机制对于数据的持久化和镜像的优化至关重要。本节将介绍容器的可写层以及 Copy-on-Write 机制。

镜像层 + 容器层

当容器运行时，Docker 会在镜像的只读层之上创建一个 **可写层** (容器存储层)：



Copy-on-Write：写时复制

当容器需要修改镜像层中的文件时：

1. Docker 将该文件 **复制** 到容器存储层
2. 在容器层中进行修改
3. 原始镜像层保持不变

```
读取文件：直接从镜像层读取（共享，高效）
修改文件：复制到容器层，然后修改（只有这个容器能看到修改）
```

Built with mdPress

⚠️ 容器存储层的生命周期

> **笔者特别强调**：这是新手最容易踩的坑！**容器存储层与容器生命周期绑定。容器删除，数据就没了！**

```
## 创建容器，写入数据

$ docker run -it ubuntu bash
root@abc123:/# echo "important data" > /data.txt
root@abc123:/# exit

## 删除容器

$ docker rm abc123

## 数据丢了！没有任何办法恢复！
```

正确的数据持久化方式

按照 Docker 最佳实践，容器存储层应该保持 **无状态**。需要持久化的数据应该使用：

方式	说明	适用场景
数据卷 (Volume)	Docker 管理的存储	数据库、应用数据
绑定挂载 (Bind Mount)	挂载宿主机目录	开发时共享代码

```
## 使用数据卷（推荐）

$ docker run -v mydata:/var/lib/mysql mysql

## 使用绑定挂载

$ docker run -v /host/path:/container/path nginx
```

这些位置的读写 **会跳过容器存储层**，直接写入宿主机，性能更好，也不会随容器删除而丢失。

2.2.5 容器的生命周期

掌握容器的生命周期对于管理和调试 Docker 应用非常重要。如图 2-1 所示，这里先聚焦最常见的创建、运行、暂停、停止和删除流转；Docker CLI 中还可能看到 `restarting`、`removing`、`dead` 等状态。

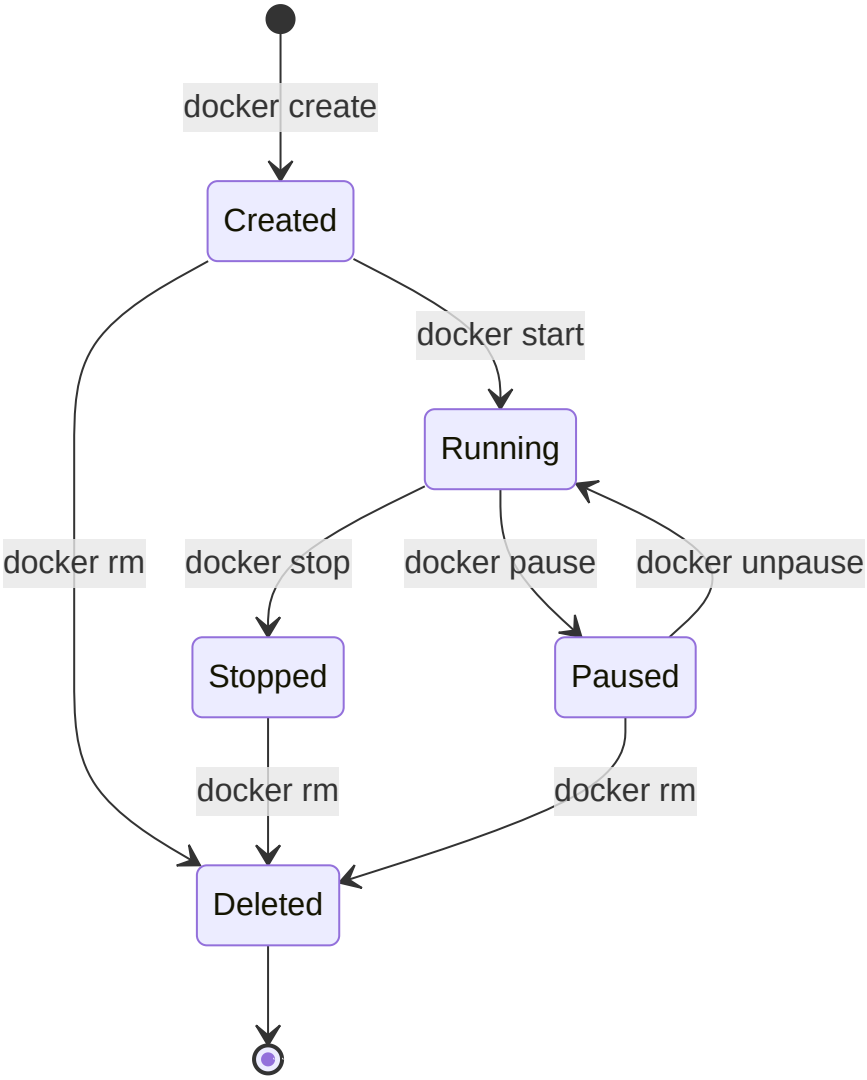


图 2-1：容器生命周期状态流转图

常用生命周期命令如下：

```
## 创建并启动容器（最常用）

$ docker run nginx

## 分步操作

$ docker create nginx     # 创建容器（不启动）
$ docker start abc123     # 启动容器

## 停止容器

$ docker stop abc123      # 优雅停止（发送 SIGTERM，等待后发送 SIGKILL）
$ docker kill abc123      # 强制停止（直接发送 SIGKILL）

## 暂停/恢复（不常用，但有时有用）

$ docker pause abc123     # 暂停容器内所有进程
$ docker unpause abc123   # 恢复

## 删除容器

$ docker rm abc123        # 删除已停止的容器
$ docker rm -f abc123     # 强制删除运行中的容器
```

2.2.6 容器与进程的关系

> **核心概念**：容器的生命周期 = 主进程 (PID 1) 的生命周期

```
## 主进程运行，容器运行

## 主进程退出，容器停止
```

这就是为什么：

```
## 这个容器会立即退出（bash 没有输入就退出了）

$ docker run ubuntu

## 这个容器会持续运行（官方 nginx 镜像让 nginx 在前台运行）

$ docker run nginx
```

官方 `nginx` 镜像默认使用 `nginx -g 'daemon off;'`，让主进程保持在前台运行，这样容器才会持续处于运行状态。

详细解释请参考后台运行章节。

2.2.7 容器的隔离性

Docker 容器通过以下 Namespace 实现隔离：

Namespace	隔离内容	效果
PID	进程 ID	容器内 PID 1 是应用进程，看不到宿主机其他进程
NET	网络	独立的网络栈、IP 地址、端口
MNT	文件系统	独立的根目录和挂载点
UTS	主机名	独立的主机名和域名
IPC	进程间通信	独立的信号量、消息队列
USER	用户	独立的用户和组 ID

> 想深入了解？请阅读底层实现 - 命名空间。

2.3 仓库

> **版本说明**：本节示例基于 Docker v29.x 和常见镜像版本编写。示例中的版本号（如 `nginx:1.28`、`mysql:8.4`、`mysql:5.7` 等）为演示用途。实际使用时请访问 Docker Hub 官方页面或相应镜像的发布页确认最新可用版本和标签。

Docker Registry 是镜像分发和管理的核心组件。本节将介绍 Registry 的基本概念、公共和私有服务的选择，以及镜像的安全管理。

2.3.1 一句话理解 Registry

> **Docker Registry 是存储和分发 Docker 镜像的服务，类似于代码的 GitHub 或包管理的 npm。**

镜像构建完成后，可以在当前机器上运行。但如果需要在其他服务器上使用这个镜像，就需要一个集中的存储和分发服务——这就是 Docker Registry。

2.3.2 核心概念

要熟练使用 Docker Registry，首先需要理清它与仓库 (Repository)、标签 (Tag) 之间的关系。

Registry、仓库、标签的关系

Docker Registry 中可以包含多个 Repository，每个 Repository 可以包含多个 Tag。如图 2-2 所示，它们之间具有清晰的层级关系。

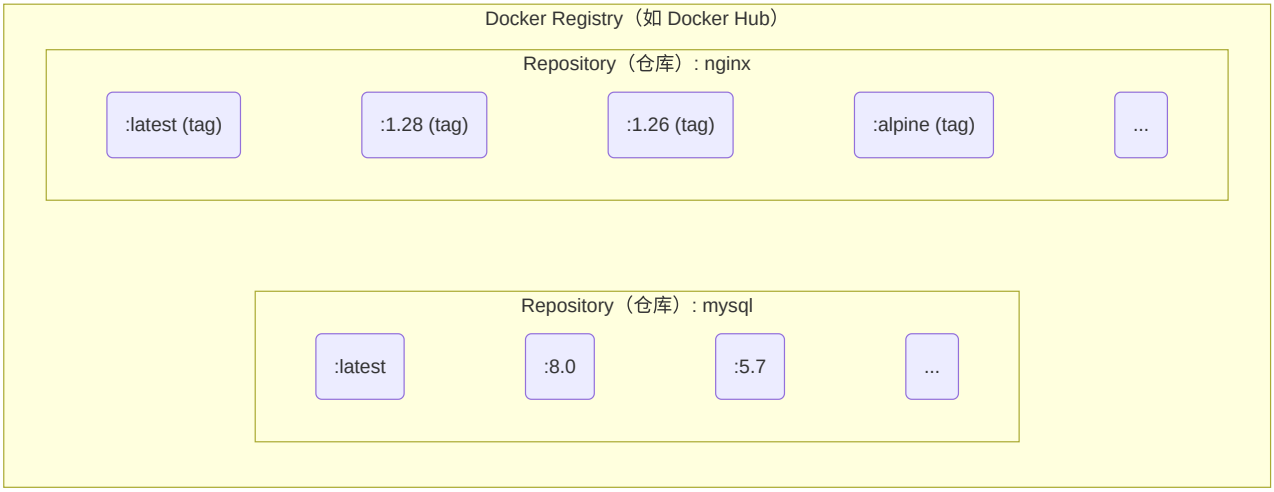


图 2-2：Registry、Repository 与 Tag 的层级关系

相关基本概念具体如下：

概念	说明	示例
Registry	存储镜像的服务	Docker Hub、ghcr.io
Repository (仓库)	同一软件的镜像集合	`nginx`、`mysql`、`mycompany/myapp`
Tag (标签)	仓库内的版本标识	`latest`、`1.28`、`alpine`

镜像的完整名称

一个完整的 Docker 镜像名称由 Registry 地址、用户名/组织名、仓库名和标签组成。了解其结构有助于我们更准确地定位镜像。基本格式如下：

```
[registry 地址/][用户名/]仓库名[:标签]
```

示例：

```
## 完整格式

registry.example.com/mycompany/myapp:v1.2.3
|                   |         |     |
|                   |         |     └── 标签
|                   |         └── 仓库名
|                   └── 用户名/组织名
└── Registry 地址

## Docker Hub 官方镜像（省略 registry 和用户名）

nginx:1.28
ubuntu:24.04

## Docker Hub 用户镜像

jwilder/nginx-proxy:latest

## 其他 Registry

ghcr.io/username/myapp:v1.0
us-west1-docker.pkg.dev/my-project/my-repo/myapp:v1.0
```

> 💡 **笔者提示**：如果不指定 Registry 地址，默认使用 Docker Hub。如果不指定标签，默认使用 `latest`。

2.3.3 公共 Registry 服务

公共 Registry 服务为开发者提供了便捷的镜像获取途径。其中最著名的是 Docker Hub。

默认的 Docker Hub

Docker Hub 是最大的公共 Registry，也是 Docker 的默认 Registry。

特点：

- 拥有大量官方镜像 (nginx、mysql、redis 等)
- 免费账户可以创建公开仓库
- 免费个人账户可创建 1 个私有仓库；更高套餐支持更多私有仓库

```
## 从 Docker Hub 拉取镜像

$ docker pull nginx              # 官方镜像
$ docker pull bitnami/redis      # 第三方镜像

## 推送镜像到 Docker Hub

$ docker login
$ docker push username/myapp:v1.0
```

其他公共 Registry

除了 Docker Hub，还有以下几个常见的公共 Registry：

Registry	地址	说明
GitHub Container Registry	ghcr.io	GitHub 提供，与 GitHub Actions 集成好
Google Artifact Registry	LOCATION-docker.pkg.dev	Google Cloud 当前推荐；也支持迁移后的 `gcr.io` 兼容域名
Quay.io	quay.io	Red Hat 提供
阿里云容器镜像服务	registry.cn-*.aliyuncs.com	国内访问快
腾讯云容器镜像服务	ccr.ccs.tencentyun.com	国内访问快

Google 的 Container Registry 已废弃并完成下线，当前应优先使用 Artifact Registry；如果已经完成迁移，部分 `gcr.io` 域名请求会被兼容到 Artifact Registry。

2.3.4 镜像加速器

由于网络原因，在国内直接访问 Docker Hub 可能会很慢。可以配置 **镜像加速器** (Registry Mirror) 来加速下载。配置示例如下：

```
// /etc/docker/daemon.json
{
  "registry-mirrors": [
    "https://your-accelerator-url"
  ]
}
```

详细配置方法请参考镜像加速器章节。

> ⚠️ **笔者提醒**：镜像加速器的可用性经常变化，使用前建议先测试是否可用。

2.3.5 私有 Registry

出于安全和隐私的考虑，企业往往需要搭建自己的私有 Registry。以下是几种常见的搭建方案。

官方 Registry 镜像

Docker 官方提供了 registry 镜像，可以快速搭建私有 Registry：

```
## 启动一个本地 Registry

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

## 推送镜像到本地 Registry

$ docker tag myapp:v1.0 localhost:5000/myapp:v1.0
$ docker push localhost:5000/myapp:v1.0

## 从本地 Registry 拉取

$ docker pull localhost:5000/myapp:v1.0
```

企业级解决方案

官方 Registry 功能较为基础，企业环境常用以下方案：

方案	特点
Harbor	CNCF 项目，功能全面 (用户管理、漏洞扫描、镜像签名)
Nexus Repository	支持多种制品类型 (Docker、Maven、npm 等)
云厂商服务	阿里云 ACR、腾讯云 TCR、AWS ECR 等

笔者建议：

- 小团队：可以先用官方 Registry，够用即可
- 中大型团队：推荐 Harbor，功能完善且开源免费
- 已使用云服务：直接用云厂商的 Registry 服务更省心

2.3.6 镜像的推送和拉取

掌握镜像的推送 (Push) 和拉取 (Pull) 是使用 Docker Registry 的基本功。

完整工作流程

如图 2-3 所示，镜像从开发环境构建后推送到 Registry，再由生产环境拉取并运行。

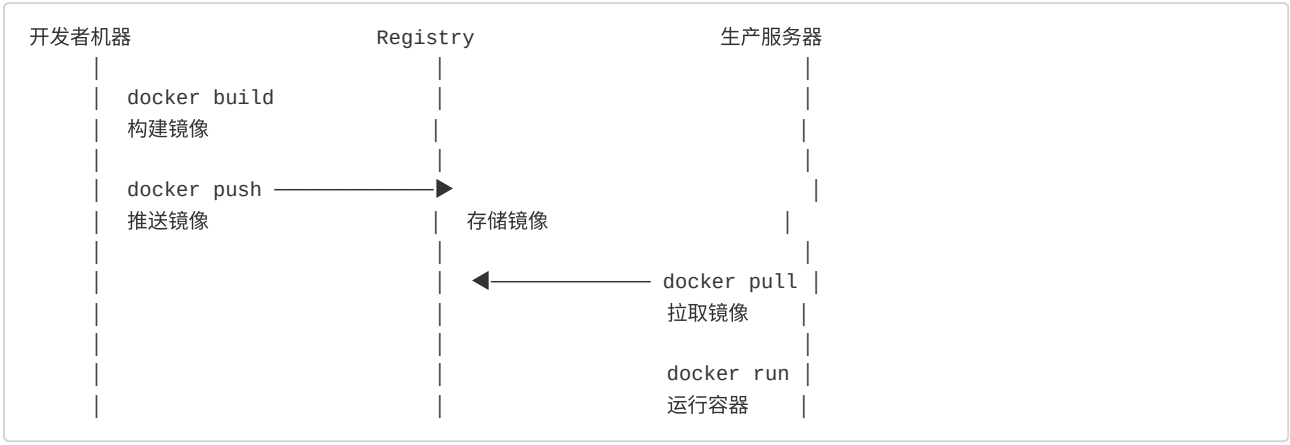


图 2-3：镜像构建、推送与拉取流程

常用命令

```
## 登录 Registry

$ docker login                      # 登录 Docker Hub
$ docker login registry.example.com # 登录其他 Registry

## 拉取镜像

$ docker pull nginx:1.28

## 标记镜像（准备推送）

$ docker tag myapp:latest registry.example.com/myteam/myapp:v1.0

## 推送镜像

$ docker push registry.example.com/myteam/myapp:v1.0

## 登出

$ docker logout
```

2.3.7 镜像的安全性

在使用公共镜像或维护私有镜像时，安全性是不容忽视的重要环节。

使用官方镜像

Docker Hub 的官方镜像 (标有 “Official Image” 标识) 经过 Docker 团队审核，相对更安全。示例如下：

```
## 官方镜像示例

nginx          # ✅ 官方
mysql          # ✅ 官方
redis          # ✅ 官方

## 第三方镜像（需要自行评估可信度）

bitnami/redis  # ⚠️ 需要评估
someuser/myapp # ⚠️ 需要评估
```

镜像签名

当前更推荐使用 Sigstore / Notation 体系进行镜像签名与验证。`Docker Content Trust (DCT)` 已进入弃用阶段：2025 年 8 月 8 日起最早的 DCT 签名证书开始过期，2025 年 9 月 30 日起不能在新 Registry 启用 DCT，2028 年 3 月 31 日将完全删除此功能。不建议作为新项目方案。

> 注意：Cosign 默认会把签名推送回镜像所在仓库，请使用你有推送权限的镜像地址。

```
## 准备一个你有写权限的镜像地址
$ export IMAGE=<你的仓库名>/nginx:1.28
$ docker pull nginx:1.28
$ docker tag nginx:1.28 $IMAGE
$ docker push $IMAGE

## 生成签名密钥（会生成 cosign.key / cosign.pub）
$ cosign generate-key-pair

## 使用 Cosign 签名与验证
$ cosign sign --key cosign.key $IMAGE
$ cosign verify --key cosign.pub $IMAGE
```

漏洞扫描

```
## 使用 Docker Scout 扫描镜像漏洞

$ docker scout cves nginx:latest

## 使用 Trivy（开源工具）

$ trivy image nginx:latest
```

本章小结

本章介绍了 Docker 的三个核心概念：镜像、容器和仓库。

概念	要点
镜像是什么	只读的应用模板，包含运行所需的一切
分层存储	多层叠加，共享基础层，节省空间
只读特性	构建后不可修改，保证一致性
层的陷阱	删除操作只是标记，不减小体积
容器是什么	镜像的运行实例，本质是隔离的进程
容器 vs 虚拟机	共享内核，更轻量，但隔离性较弱
存储层	可写层随容器删除而消失
数据持久化	使用 Volume 或 Bind Mount
生命周期	与主进程 (PID 1) 绑定
Registry	存储和分发镜像的服务
仓库 (Repository)	同一软件的镜像集合
标签 (Tag)	版本标识，默认为 latest
Docker Hub	默认的公共 Registry
私有 Registry	企业内部使用，推荐 Harbor

现在你已经了解了 Docker 的三个核心概念：镜像、容器 和仓库。接下来，让我们开始 安装 Docker，动手实践！

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第三章 安装 Docker

Docker Engine 主要提供 `stable` 和 `test` 两个更新频道；`test.docker.com` 对应测试频道，适合预发布验证，不建议直接用于生产环境。

官方网站上有各种环境下的安装指南，这里主要介绍 Docker 在 `Linux`、`Windows 10/11` 和 `macOS` 上的安装。

安装方式选择指南

在开始安装前，笔者建议你根据以下决策树选择最合适的安装方式：

生产环境 vs 开发环境

生产环境（服务器部署）：

- 优先使用**官方 APT/YUM 源安装**（Ubuntu、Debian、Fedora、CentOS）
- 优势：获得官方安全更新、长期技术支持、版本管理清晰
- 安装步骤稍多一些，但这种“麻烦”是值得的——它为你的生产系统争取了稳定性和可维护性

开发环境（本地开发机、测试服务器）：

- 使用**脚本自动安装**或**包管理器直接安装**
- 如果你想快速上手，官方脚本（`get.docker.com`）是最便捷的选择
- 国内用户注意：这一步一定要选对镜像源，否则网络卡顿会严重影响体验

国内用户的网络优化建议

值得注意的是，国内直接访问 Docker 官方源速度较慢，建议：

- **安装过程**：使用阿里云、腾讯云等国内镜像源
- **镜像拉取**：安装完成后配置 Docker 镜像加速器（详见 3.9 镜像加速器），这一步对日常开发的体验提升最明显

特殊场景

- **Raspberry Pi/ARM 平台**：见 3.5 Raspberry Pi
- **离线环境**：见 3.6 Linux 离线安装
- **macOS/Windows**：Docker Desktop 是官方推荐的一站式解决方案
- **需要实验特性**：见 3.10 开启实验特性

详细安装指南

- Ubuntu
- Debian
- Fedora
- CentOS
- Raspberry Pi
- Linux 离线安装
- macOS
- Windows 10/11
- 镜像加速器
- 开启实验特性

3.1 Ubuntu

Ubuntu 是 Docker 最常用的运行环境之一。本节将介绍如何在 Ubuntu 系统上安装 Docker，并配置国内镜像加速。

为什么推荐 APT 源安装而不是脚本?

虽然 Docker 官方提供了便捷的安装脚本（`get.docker.com`），但笔者在生产环境中**强烈推荐通过 Docker 官方 APT 仓库安装**，原因如下：

- **版本管理**：通过 APT 仓库安装后，可以像管理其他系统软件包一样升级、回滚和锁定版本
- **安全更新**：Docker 官方仓库会持续发布新版本和安全修复，适合长期维护
- **一致性**：团队更容易锁定同一版本，避免“在我的机器上可以运行”的问题
- **卸载干净**：APT 包管理系统会负责清理所有相关文件，脚本安装的清理往往不够彻底

如果你只是想快速尝试 Docker，脚本安装没有问题；但一旦涉及持久运维，APT 源是更成熟的选择。

> 警告：切勿在没有配置 Docker APT 源的情况下直接使用 apt 命令安装 Docker。

3.1.1 准备工作

在开始安装之前，我们需要确认系统版本是否满足要求，并清理可能存在的旧版本。

系统要求

根据 Docker 官方安装文档，当前受支持的 Ubuntu 64 位版本包括（具体以官方 安装文档 为准）：

- Ubuntu Resolute Raccoon 26.04 (LTS)
- Ubuntu Questing Quokka 25.10
- Ubuntu Noble 24.04 (LTS)
- Ubuntu Jammy 22.04 (LTS)

Built with mdPress

> **警告**：Ubuntu 20.04 LTS 已不在 Docker 当前支持列表中，不建议用于新部署。对于仍在运行 20.04 的生产系统，应尽快升级到 22.04 LTS 或 24.04 LTS；若短期内无法迁移，可通过 Ubuntu Pro 获取操作系统层面的扩展安全维护（ESM），但这并不改变 Docker 官方支持矩阵。

在 Ubuntu LTS 版本上，目前 Docker 支持 amd64、arm64、armhf、ppc64el、s390x 等 5 个平台；而非 LTS 版本支持的平台通常较少。同时，LTS 版本会获得 5 年的升级维护支持，这样的系统会获得更长期的安全保障，因此在生产环境中推荐使用 LTS 版本。

卸载旧版本

Docker 官方建议先卸载可能冲突的非官方软件包：

```
$ for pkg in docker.io docker-compose docker-compose-v2 docker-doc podman-docker containerd runc;
do
    sudo apt remove $pkg;
done
```

3.1.2 使用 APT 安装

先安装基础依赖，并准备 Docker 官方密钥目录：

```
$ sudo apt update

$ sudo apt install ca-certificates curl
$ sudo install -m 0755 -d /etc/apt/keyrings
```

如果企业内网已经维护了受信任的软件包镜像，可在后续步骤中替换 `URIs` 的域名；默认建议优先以 Docker 官方仓库为准。

为了确认所下载软件包的合法性，需要添加仓库签名密钥：

```
$ sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
$ sudo chmod a+r /etc/apt/keyrings/docker.asc
```

然后向 `apt` 添加 Docker 仓库：

```
$ sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
```

> 如果需要测试频道，可将 `Components: stable` 改为 `test`，或改用 `test.docker.com` 脚本在测试环境验证。

更新 APT 缓存，并安装 Docker Engine 及常用 CLI 插件：

```
$ sudo apt update

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

3.1.3 使用脚本自动安装

在测试或开发环境中，Docker 官方提供了便捷安装脚本，但官方明确不建议把它作为生产环境的标准安装方式。

在真正执行前，建议先用 `--dry-run` 预览脚本动作：

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh ./get-docker.sh --dry-run

# 若需要测试频道：
# curl -fsSL https://test.docker.com -o test-docker.sh
# sudo sh ./test-docker.sh
```

确认无误后，再执行 `sudo sh ./get-docker.sh` 安装稳定版。

3.1.4 启动 Docker

```
$ sudo systemctl enable --now docker
```

3.1.5 建立 docker 用户组

默认情况下，`docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好的做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

> **⚠️ 安全警告：docker 用户组等同于 root 权限**
>
> 将用户加入 docker 组免去了每次执行 docker 命令时输入 sudo 的繁琐，但这也意味着该用户可以轻易获取主机的最高 root 权限（例如通过挂载根目录运行容器）。 如果你在一个多用户共享的生产系统上配置，切勿随意将普通用户加入此组。此时，更安全的替代方案是使用官方提供的 **Rootless 模式 (Rootless mode)**，它允许在没有任何 root 权限的情况下运行 Docker 守护进程和容器。

建立 docker 组：

```
$ sudo groupadd docker
```

将当前用户加入 docker 组：

```
$ sudo usermod -aG docker $USER
```

退出当前终端并重新登录，进行如下测试。

3.1.6 测试 Docker 是否安装正确

```
$ docker run --rm hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

若能正常输出以上信息，则说明安装成功。

3.1.7 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.2 Debian

Debian 以其稳定性著称，是 Docker 的理想宿主系统。本节将指导你在 Debian 上完成 Docker 的安装。

APT 源安装的必要性

与 Ubuntu 类似，Debian 用户在安装 Docker 时同样应该优先选择 Docker 官方 APT 仓库。对于服务器环境，这种方式更利于版本控制、升级和长期维护。

> 警告：切勿在没有配置 Docker APT 源的情况下直接使用 apt 命令安装 Docker。

3.2.1 准备工作

安装前请仔细检查 Debian 版本支持情况，并卸载旧版本以避免冲突。

系统要求

Docker 支持以下版本的 Debian 操作系统（具体以官方 安装文档 为准）：

- Debian Trixie 13 (stable)
- Debian Bookworm 12 (oldstable，全面支持至 2026 年 6 月 10 日，LTS 至 2028 年 6 月 30 日)
- Debian Bullseye 11 (oldoldstable，LTS 支持至 2026 年 8 月 31 日)

> **注意**：Debian Bullseye 11 将于 2026 年 8 月底结束长期支持。建议新部署使用 Bookworm 12 或 Trixie 13。

卸载旧版本

Docker 官方建议先卸载可能冲突的非官方软件包：

```
$ for pkg in docker.io docker-compose docker-compose-v2 docker-doc podman-docker containerd runc; do
    sudo apt remove $pkg
done
```

3.2.2 使用 APT 安装

先安装基础依赖，并准备密钥目录：

```
$ sudo apt update

$ sudo apt install ca-certificates curl
$ sudo install -m 0755 -d /etc/apt/keyrings
```

如果公司内网维护了受信任镜像站，可在后续仓库地址中替换域名；默认建议优先使用 Docker 官方仓库。

为了确认所下载软件包的合法性，需要添加软件源的 GPG 密钥。

```
$ sudo curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc
$ sudo chmod a+r /etc/apt/keyrings/docker.asc
```

然后，我们需要向 `sources.list` 中添加 Docker 软件源：

> 在一些基于 Debian 的 Linux 发行版中，`$(. /etc/os-release && echo "$VERSION_CODENAME")` 可能不会返回 Debian 官方仓库使用的代号，例如 Kali Linux 等衍生发行版。此时请手动替换为对应的 Debian 代号，例如 `bookworm`。

```
$ sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/debian
Suites: $(. /etc/os-release && echo "$VERSION_CODENAME")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
```

> 如果使用 Kali 等衍生发行版，请把 `Suites` 对应的版本代号替换为映射到的 Debian 代号，例如 `bookworm`。如果需要测试频道，可将 `Components: stable` 改为 `test`。

更新 APT 缓存，并安装 Docker Engine 及常用 CLI 插件。

```
$ sudo apt update

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

3.2.3 使用脚本自动安装

在测试或开发环境中，Docker 官方提供了便捷安装脚本，但官方明确不建议把它作为生产环境的标准安装方式。

在真正执行前，建议先用 `--dry-run` 预览脚本动作：

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh ./get-docker.sh --dry-run

# 若需要测试频道：
# curl -fsSL https://test.docker.com -o test-docker.sh
# sudo sh ./test-docker.sh
```

确认无误后，再执行 `sudo sh ./get-docker.sh` 安装稳定版。

3.2.4 启动 Docker

```
$ sudo systemctl enable --now docker
```

3.2.5 建立 docker 用户组

默认情况下，`docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好的做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

> **⚠️ 安全警告：`docker` 用户组等同于 `root` 权限**
>
> 将用户加入 `docker` 组免去了每次执行 `docker` 命令时输入 `sudo` 的繁琐，但这也意味着该用户可以轻易获取主机的最高 root 权限（例如通过挂载根目录运行容器）。 如果你在一个多用户共享的生产系统上配置，切勿随意将普通用户加入此组。此时，更安全的替代方案是使用官方提供的 **Rootless 模式 (Rootless mode)**，它允许在没有任何 root 权限的情况下运行 Docker 守护进程和容器。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 `docker` 组：

```
$ sudo usermod -aG docker $USER
```

退出当前终端并重新登录，进行如下测试。

3.2.6 测试 Docker 是否安装正确

```
$ docker run --rm hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

若能正常输出以上信息，则说明安装成功。

3.2.7 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.3 Fedora

Fedora 作为技术前沿的 Linux 发行版，对 Docker 有着良好的支持。本节介绍在 Fedora 上的安装步骤。

YUM/DNF 源安装的策略建议

Fedora 的快速发布周期（每 6 个月发布新版本）决定了它的用户群体多为开发者和技术爱好者。虽然通过 DNF 可以直接安装 Docker，但笔者建议仍然通过 Docker 官方 YUM 源进行安装，原因是：Fedora 官方仓库的 Docker 版本往往滞后，而官方源能确保你获得最新的 Docker 功能和安全补丁。特别是在开发环境需要用到最新 Docker 特性时，这一点显得尤为重要。

> 警告：切勿在没有配置 Docker dnf 源的情况下直接使用 dnf 命令安装 Docker。

3.3.1 准备工作

确保你的 Fedora 版本在支持列表中，并清理旧版本。

系统要求

根据 Docker 官方安装文档，当前受支持的 Fedora 版本包括（具体以官方 安装文档 为准）：

- Fedora 44
- Fedora 43
- Fedora 42

卸载旧版本

旧版本的 Docker 称为 `docker` 或者 `docker-engine`，使用以下命令卸载旧版本：

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

3.3.2 使用 dnf 安装

使用 dnf 包管理器安装是推荐的方式，便于后续的更新和管理。

执行以下命令安装依赖包：

```
$ sudo dnf -y install dnf-plugins-core
```

默认建议优先使用 Docker 官方仓库；如果企业内网维护了受信任镜像站，可自行替换仓库 URL。

执行下面的命令添加 `dnf` 软件源：

```
$ sudo dnf config-manager \
    --add-repo \
    https://download.docker.com/linux/fedora/docker-ce.repo
```

如果需要测试版本的 Docker 请使用以下命令：

```
$ sudo dnf config-manager --set-enabled docker-ce-test
```

你也可以禁用测试版本的 Docker

```
$ sudo dnf config-manager --set-disabled docker-ce-test
```

安装 Docker

更新 `dnf` 软件源缓存，并安装 Docker Engine 及常用 CLI 插件。

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

你也可以使用以下命令安装指定版本的 Docker

```
$ dnf list docker-ce  --showduplicates | sort -r

docker-ce.x86_64          3:29.4.0-1.fc42                      docker-ce-stable

$ sudo dnf -y install docker-ce-<VERSION_STRING> docker-ce-cli-<VERSION_STRING>
```

3.3.3 使用脚本自动安装

在测试或开发环境中，Docker 官方提供了便捷安装脚本，但官方明确不建议把它作为生产环境的标准安装方式。

在真正执行前，建议先用 `--dry-run` 预览脚本动作：

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh ./get-docker.sh --dry-run

# 若需要测试频道：
# curl -fsSL https://test.docker.com -o test-docker.sh
# sudo sh ./test-docker.sh
```

确认无误后，再执行 `sudo sh ./get-docker.sh` 安装稳定版。

3.3.4 启动 Docker

```
$ sudo systemctl enable --now docker
```

3.3.5 建立 docker 用户组

默认情况下，`docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好的做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

> **⚠️ 安全警告：`docker` 用户组等同于 `root` 权限**
>
> 将用户加入 `docker` 组免去了每次执行 `docker` 命令时输入 `sudo` 的繁琐，但这也意味着该用户可以轻易获取主机的最高 root 权限（例如通过挂载根目录运行容器）。 如果你在一个多用户共享的生产系统上配置，切勿随意将普通用户加入此组。此时，更安全的替代方案是使用官方提供的 **Rootless 模式 (Rootless mode)**，它允许在没有任何 root 权限的情况下运行 Docker 守护进程和容器。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 `docker` 组：

```
$ sudo usermod -aG docker $USER
```

退出当前终端并重新登录，进行如下测试。

3.3.6 测试 Docker 是否安装正确

```
$ docker run --rm hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

若能正常输出以上信息，则说明安装成功。

3.3.7 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.4 CentOS

CentOS (及其替代品 Rocky Linux、AlmaLinux) 是企业级服务器常用的操作系统。本节介绍在这些系统上安装 Docker 的步骤。

企业级部署的版本选择

值得注意的是，**CentOS 8 已停止维护，CentOS 7 已停止支持**。如果你正在规划新的生产部署，强烈建议选择 Rocky Linux 或 AlmaLinux——这两个项目是由社区维护的 CentOS 替代品，延续了 CentOS 的企业级特性，同时提供了更长的生命周期承诺。选择稳定的基础系统，才能为 Docker 的长期运维奠定坚实基础。

> 警告：切勿在没有配置 Docker YUM 源的情况下直接使用 yum 命令安装 Docker。

3.4.1 准备工作

安装前请确认系统版本和内核版本满足 Docker 的运行要求。

系统要求

> **严重警告**：CentOS 7 已于 2024 年 6 月 30 日结束所有支持，不再接收任何安全更新。CentOS 8 已于 2021 年 12 月 31 日停止维护。强烈建议新项目使用 **Rocky Linux** 或 **AlmaLinux** 替代，这两个项目由社区维护，提供长期支持承诺。

Docker 官方当前安装文档覆盖的 CentOS 平台为 **CentOS Stream 9** 和 **CentOS Stream 10**（具体以官方 安装文档 为准）。Rocky Linux、AlmaLinux 等 RHEL 兼容发行版通常可以沿用相近步骤，但建议先在测试环境验证仓库与依赖是否匹配。

对于 Rocky Linux、AlmaLinux 或 CentOS Stream，推荐使用 `dnf` 包管理器。

卸载旧版本

旧版本的 Docker 称为 `docker` 或者 `docker-engine`，使用以下命令卸载旧版本：

```
$ sudo yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-selinux \
                  docker-engine-selinux \
                  docker-engine \
                  docker-ce-cli \
                  containerd.io
```

3.4.2 使用 yum 安装

使用 yum/dnf 安装是管理 Docker 生命周期的标准方式。

执行以下命令安装依赖包：

```
$ sudo dnf install -y dnf-plugins-core
```

默认建议优先使用 Docker 官方仓库；如果企业内网维护了受信任镜像站，可自行替换仓库 URL。

执行下面的命令添加 `yum` 软件源：

```
$ sudo dnf config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

如果需要测试版本的 Docker 请执行以下命令：

```
$ sudo dnf config-manager --set-enabled docker-ce-test
```

安装 Docker

更新 `dnf` 软件源缓存，并安装 Docker Engine 及常用 CLI 插件。

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

3.4.3 防火墙注意事项

Docker 官方提醒：如果主机使用 `ufw` 或 `firewalld` 管理防火墙，容器暴露出的端口可能绕过现有规则；同时 Docker 仅兼容 `iptables-nft` 和 `iptables-legacy`，不支持直接由 `nft` 创建的规则集。

因此，生产环境中更稳妥的做法是：

- 明确使用 `iptables-nft` 或 `iptables-legacy` 维护规则；
- 需要限制容器流量时，优先把自定义规则写入 `DOCKER-USER` 链；
- 在变更防火墙策略前，先用测试容器验证端口暴露和东西向流量是否符合预期。

3.4.4 使用脚本自动安装

在测试或开发环境中，Docker 官方提供了便捷安装脚本，但官方明确不建议把它作为生产环境的标准安装方式。

在真正执行前，建议先用 `--dry-run` 预览脚本动作：

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh ./get-docker.sh --dry-run

# 若需要测试频道：
# curl -fsSL https://test.docker.com -o test-docker.sh
# sudo sh ./test-docker.sh
```

确认无误后，再执行 `sudo sh ./get-docker.sh` 安装稳定版。

3.4.5 启动 Docker

```
$ sudo systemctl enable --now docker
```

3.4.6 建立 docker 用户组

默认情况下，`docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好的做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

> ⚠️ **安全警告：`docker` 用户组等同于 `root` 权限**
>
> 将用户加入 `docker` 组免去了每次执行 `docker` 命令时输入 `sudo` 的繁琐，但这也意味着该用户可以轻易获取主机的最高 root 权限（例如通过挂载根目录运行容器）。 如果你在一个多用户共享的生产系统上配置，切勿随意将普通用户加入此组。此时，更安全的替代方案是使用官方提供的 **Rootless 模式 (Rootless mode)**，它允许在没有任何 root 权限的情况下运行 Docker 守护进程和容器。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 docker 组：

```
$ sudo usermod -aG docker $USER
```

退出当前终端并重新登录，进行如下测试。

3.4.7 测试 Docker 是否安装正确

```
$ docker run --rm hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

若能正常输出以上信息，则说明安装成功。

3.4.8 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.4.9 添加内核参数

如果在 CentOS 使用 Docker 看到下面的这些警告信息：

```
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
```

请添加内核配置参数以启用这些功能。

```
$ sudo tee -a /etc/sysctl.conf <<-EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

然后重新加载 `sysctl.conf` 即可

```
$ sudo sysctl -p
```

3.5 Raspberry Pi

树莓派等 ARM 架构设备在物联网和边缘计算领域应用广泛。本节介绍如何在树莓派上安装 Docker。

树莓派运行 Docker 的现实考虑

树莓派是 Docker 在边缘计算的一个优秀用例。但在安装前，笔者建议你了解几个实际情况：

性能特性：

- Raspberry Pi 4 及以上才能流畅运行 Docker 和基本容器
- Raspberry Pi Zero 2 可以运行，但性能受限
- 磁盘 I/O 是主要瓶颈（特别是使用 microSD 卡时）

镜像可用性：

- 并非所有 Docker 镜像都有 ARM 版本（arm64 或 armv7）
- 官方镜像通常提供多架构支持，但第三方镜像可能没有
- 某些依赖 Intel 特定指令的应用无法在 ARM 上运行

存储和内存：

- 容器镜像会占用较多存储空间，128GB microSD 卡建议最多运行 3-4 个中等大小的容器
- 512MB 或 1GB 内存的树莓派运行多个容器会非常吃力

实践建议：树莓派 Docker 部署最适合轻量级应用——单个微服务、监控代理、Web 服务器等。复杂多容器应用还是应该部署在性能更强的硬件上。

> 警告：切勿在没有配置 Docker APT 源的情况下直接使用 apt 命令安装 Docker。

3.5.1 系统要求

Docker 对 ARM 架构有着良好的支持。

Docker 可以运行在多种 CPU 架构上，但本小节**只聚焦 Raspberry Pi OS 32-bit (armhf)** 的官方安装流程；如果你使用的是 64 位 Raspberry Pi OS，请直接参考 Debian `arm64` 安装说明。

Docker 官方目前单独提供的是 **Raspberry Pi OS 32-bit (armhf)** 安装页（具体以官方 安装文档 为准），支持情况如下：

- Raspberry Pi OS Bookworm
- Raspberry Pi OS Bullseye

> **重要提醒**：Docker Engine v28 是官方最后一个支持 Raspberry Pi OS 32-bit (armhf) 的大版本。从 v29 开始，32 位 Raspberry Pi OS 不再提供新主版本软件包。若你使用的是 64 位 Raspberry Pi OS，请直接参考 Debian `arm64` 安装方式。

注：`Raspberry Pi OS` 由树莓派的开发与维护机构树莓派基金会官方支持，并推荐用作树莓派的首选系统，其基于 `Debian`。

3.5.2 使用 APT 安装

推荐使用 APT 包管理器进行安装，以确保版本的稳定性和安全性。

先安装基础依赖，并准备密钥目录：

```
$ sudo apt-get update

$ sudo apt-get install \
     ca-certificates \
     curl \
     gnupg
```

默认建议优先使用 Docker 官方仓库；如果企业内网维护了受信任镜像站，可自行替换仓库 URL。

为了确认所下载软件包的合法性，需要添加软件源的 GPG 密钥。

```
$ sudo install -m 0755 -d /etc/apt/keyrings
$ sudo curl -fsSL https://download.docker.com/linux/raspbian/gpg -o /etc/apt/keyrings/docker.asc
$ sudo chmod a+r /etc/apt/keyrings/docker.asc
```

然后，我们需要向 `sources.list` 中添加 Docker 软件源：

```
$ sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/raspbian
Suites: $(. /etc/os-release && echo "$VERSION_CODENAME")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
```

> 以上命令会添加稳定版本的 Docker APT 源，如果需要测试版本的 Docker 请将 stable 改为 test。

报错解决办法

在 `Raspberry Pi OS Bullseye/Bookworm` 中，如果你使用 `add-apt-repository` 添加源，可能会出现如下报错。官方当前更推荐的做法是**不要继续回退到旧式单行 deb 配置**，而是直接使用上面的 `docker.sources` 文件方式写入仓库：

```
Traceback (most recent call last):
 File "/usr/bin/add-apt-repository", line 95, in <module>
   sp = SoftwareProperties(options=options)
 File "/usr/lib/python3/dist-packages/softwareproperties/SoftwareProperties.py", line 109, in __init
__
   self.reload_sourceslist()
 File "/usr/lib/python3/dist-packages/softwareproperties/SoftwareProperties.py", line 599, in reload
_sourceslist
   self.distro.get_sources(self.sourceslist)
 File "/usr/lib/python3/dist-packages/aptsources/distro.py", line 91, in get_sources
   raise NoDistroTemplateException(
aptsources.distro.NoDistroTemplateException: Error: could not find a distribution template for Raspb
ian/bullseye
```

如果之前已经写入了错误的源配置，建议先删除旧条目，再重新执行本节前面的 `docker.asc` + `docker.sources` 两步。

安装 Docker

更新 apt 软件包缓存，并安装 Docker Engine 及常用 CLI 插件。

```
$ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plu
gin
```

3.5.3 使用脚本自动安装

在测试或开发环境中，Docker 官方提供了便捷安装脚本，但官方明确不建议把它作为生产环境的标准安装方式。

在真正执行前，建议先用 `--dry-run` 预览脚本动作：

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh ./get-docker.sh --dry-run

# 若需要测试频道：
# curl -fsSL https://test.docker.com -o test-docker.sh
# sudo sh ./test-docker.sh
```

确认无误后，再执行 `sudo sh ./get-docker.sh` 安装稳定版。如果你运行的是 64 位 Raspberry Pi OS，更推荐直接迁移到 Debian `arm64` 安装路径，而不是继续停留在 32 位 armhf 流程上。

3.5.4 启动 Docker

```
$ sudo systemctl enable --now docker
```

3.5.5 建立 docker 用户组

默认情况下，`docker` 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 `root` 用户和 `docker` 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 `root` 用户。因此，更好的做法是将需要使用 `docker` 的用户加入 `docker` 用户组。

> **⚠️ 安全警告：`docker` 用户组等同于 `root` 权限**
>
> 将用户加入 `docker` 组免去了每次执行 `docker` 命令时输入 `sudo` 的繁琐，但这也意味着该用户可以轻易获取主机的最高 root 权限（例如通过挂载根目录运行容器）。 如果你在一个多用户共享的生产系统上配置，切勿随意将普通用户加入此组。此时，更安全的替代方案是使用官方提供的 **Rootless 模式 (Rootless mode)**，它允许在没有任何 root 权限的情况下运行 Docker 守护进程和容器。

建立 `docker` 组：

```
$ sudo groupadd docker
```

将当前用户加入 `docker` 组：

```
$ sudo usermod -aG docker $USER
```

退出当前终端并重新登录，进行如下测试。

3.5.6 测试 Docker 是否安装正确

```
$ docker run --rm hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
4ee5c797bcd7: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm32v7)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

若能正常输出以上信息，则说明安装成功。

*注意：*ARM 平台不能使用 `x86` 镜像，查看 Raspberry Pi OS 可使用镜像请访问 arm32v7 或者 arm64v8。

3.5.7 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.6 Linux 离线安装

生产环境中一般都是没有公网资源的，本文介绍如何在生产服务器上离线部署 Docker

括号内的字母表示该操作需要在哪些服务器上执行



3.6.1 CentOS/Rocky/AlmaLinux 离线安装 Docker

在无法连接外网的安全环境中，离线安装是唯一的选择。本节介绍如何在 RHEL 系发行版中进行离线安装。

> 注意：Docker 官方当前支持的 RHEL 兼容平台基线已是 **CentOS Stream 9/10**（具体以官方 安装文档 为准）。下面的离线示例建议统一按 `el9` 软件包和 `dnf` 流程准备，Rocky Linux 9、AlmaLinux 9 也可先在测试环境按相同思路验证。

3.6.1.1 本地 RPM 文件安装：推荐

推荐这种方式，是因为在生产环境中一般会选定某个指定的 Docker 软件版本使用。

查询可用的软件版本

```
$ sudo dnf -y install dnf-plugins-core
$ sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo
$ sudo dnf list docker-ce --showduplicates | sort -r
```

```
docker-ce.x86_64            3:29.4.0-1.el9                    docker-ce-stable
docker-ce.x86_64            3:29.3.1-1.el9                    docker-ce-stable
docker-ce.x86_64            3:29.3.0-1.el9                    docker-ce-stable
```

下载到指定文件夹

```
sudo dnf install --downloadonly --downloaddir=/tmp/docker_offline_install/ \
  docker-ce-<VERSION_STRING> \
  docker-ce-cli-<VERSION_STRING> \
  containerd.io \
  docker-buildx-plugin \
  docker-compose-plugin
```

下载完成后，把 `/tmp/docker_offline_install/` 目录中的全部 RPM 文件复制到离线目标服务器。

在目标服务器进入文件夹后安装

- 离线安装时，不要使用 `rpm --nodeps --force` 跳过依赖检查；应先把完整依赖包集复制到目标服务器，再让 `dnf` 从本地 RPM 安装。

```
$ sudo dnf install ./*.rpm
```

锁定软件版本（可选）

下载锁定版本软件

可参考下文的网络源搭建

```
$ sudo dnf install 'dnf-command(versionlock)'
```

锁定软件版本

```
$ sudo dnf versionlock add docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose
-plugin
```

查看锁定列表

```
$ sudo dnf versionlock list
```

锁定后无法再更新

```
$ sudo dnf upgrade docker-ce
```

解锁指定软件

```
$ sudo dnf versionlock delete docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-comp
ose-plugin
```

解锁所有软件

```
$ sudo dnf versionlock clear
```

3.6.1.2 本地仓库服务器搭建安装 Docker

挂载 ISO 镜像搭建本地 File 源

```
## 备份其他网络源

$ repo_backup="/etc/yum.repos.d/backup-$(date +%Y%m%d%H%M%S)"
$ sudo mkdir -p "$repo_backup"
$ sudo find /etc/yum.repos.d -maxdepth 1 -type f -name '*.repo' -exec mv {} "$repo_backup"/ \;

## 挂载光盘或者iso镜像

$ sudo mount /dev/cdrom /mnt
```

```
## 添加本地源

$ sudo tee /etc/yum.repos.d/local-base.repo <<EOF
[local_base]
name=local_base
baseurl=file:///mnt
enabled=1
gpgcheck=1
EOF
```

```
## 测试刚才的本地源,安装createrepo软件

$ sudo dnf clean all
$ sudo dnf install -y createrepo_c httpd
```

根据本地文件搭建 BaseOS/AppStream 网络源

```
## 安装apache 服务器

## 新建centos目录

$ sudo mkdir -p /var/www/html/base

## 复制光盘内的文件到刚才新建的目录

$ sudo cp -R /mnt/* /var/www/html/base/
$ sudo createrepo_c /var/www/html/base/
$ sudo systemctl enable --now httpd
```

下载 Docker CE 仓库内容

在有网络的服务器上同步 Docker CE RPM 包：

```
$ sudo dnf -y install dnf-plugins-core
$ sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.repo
$ mkdir -p /tmp/docker-ce
$ sudo dnf reposync --repo=docker-ce-stable --download-path=/tmp/docker-ce
```

创建仓库索引

把下载的 docker-ce 文件夹复制到离线的服务器

```
## 把 docker-ce 文件夹复制到 /var/www/html/docker-ce

$ sudo createrepo_c /var/www/html/docker-ce/
```

DNF 客户端设置

```
$ repo_backup="/etc/yum.repos.d/backup-$(date +%Y%m%d%H%M%S)"
$ sudo mkdir -p "$repo_backup"
$ sudo find /etc/yum.repos.d -maxdepth 1 -type f -name '*.repo' -exec mv {} "$repo_backup"/ \;
$ sudo tee /etc/yum.repos.d/local-files.repo <<EOF
[local_base]
name=local_base

## 改成仓库服务器地址

baseurl=http://x.x.x.x/base
enabled=1
gpgcheck=1
proxy=_none_
[docker-ce-stable]
name=docker-ce-stable

## 改成仓库服务器地址

baseurl=http://x.x.x.x/docker-ce
enabled=1
gpgcheck=1
proxy=_none_
EOF
```

安装 Docker

```
$ sudo dnf makecache
$ sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
$ sudo systemctl enable --now docker
$ sudo docker run hello-world
```

3.7 macOS

Mac 用户的特殊考虑

macOS 上没有原生 Linux 内核，Docker 需要运行在一个轻量级虚拟机中。Docker Desktop 完全封装了这个复杂性，让 Mac 用户可以像 Linux 用户一样使用 Docker。但有几点需要特别了解：

性能特性：

- Apple Silicon（M 系列芯片）比 Intel Mac 的性能更好，且拥有原生支持
- 文件 I/O 性能：macOS 与容器之间的卷挂载性能不如 Linux（这是虚拟化的代价）
- 内存使用：Docker Desktop 本身会消耗一定内存用于虚拟机管理

许可考虑：

- 小型企业（少于 250 名员工且年收入低于 1000 万美元）、个人使用、教育和非商业开源项目可免费使用
- 超出上述范围的商业用途需要付费订阅

3.7.1 系统要求

Docker Desktop for Mac 支持当前版本及前两个主要版本的 macOS（具体以官方 安装文档 为准），并且至少需要 4 GB 内存。对于 Apple Silicon 机型，若需要兼容部分 Intel 命令行工具，官方建议安装 Rosetta 2。

3.7.2 安装

> ⚠️ **Warning**
>
> **商业许可限制**：Docker Desktop 对小型企业（少于 250 名员工且年收入低于 1000 万美元）、个人使用、教育和非商业开源项目仍然免费。超出上述范围的商业用途需要付费订阅。企业用户请注意合规风险。

Docker Desktop 为 Mac 用户提供了标准的图形化安装体验。官方当前主要提供 DMG 交互安装和命令行/企业部署安装；这里先介绍最常见的 DMG 安装方式。

手动下载安装

如果需要手动下载，可直接使用 Docker Desktop for Mac Intel 版 安装包。

> 如果你的电脑搭载的是 Apple Silicon 芯片（`arm64` 架构），请使用 Docker Desktop for Mac Apple Silicon 版。

如同 macOS 其它软件一样，安装也非常简单，双击下载的 `.dmg` 文件，然后将那只叫 Moby 的鲸鱼图标拖拽到 `Application` 文件夹即可 (其间需要输入用户密码)。

图

3.7.3 运行

从应用中找到 Docker 图标并点击运行。

图

运行之后，会在右上角菜单栏看到多了一个鲸鱼图标，这个图标表明了 Docker 的运行状态。

图

每次点击鲸鱼图标会弹出操作菜单。

图

之后，你可以在终端通过命令检查安装后的 Docker 版本。

```
$ docker version
$ docker info
```

如果 `docker version` 和 `docker info` 都能正常返回信息，就可以尝试运行一个 Nginx 服务器：

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

服务运行后，可以访问 http://localhost。如果看到了 `Welcome to nginx!`，就说明 Docker Desktop for Mac 安装成功了。

图

要停止并删除 Nginx 容器，执行下面的命令：

```
$ docker stop webserver
$ docker rm webserver
```

3.7.4 替代容器运行时

Docker Desktop 并非 macOS 上运行容器的唯一选择。以下两个工具也广泛使用，各有侧重：

特性	Docker Desktop	OrbStack	Colima
启动速度	较慢（约 10–30 秒）	极快（约 2 秒）	中等（约 5–10 秒）
空闲内存占用	4–6 GB	200–300 MB	约 400 MB
图形界面	完整 GUI	轻量 GUI	仅命令行
Apple Silicon	支持	原生优化	支持
商业许可	大型企业需付费	个人免费，商业付费	MIT 开源
Kubernetes	内置	内置	通过 k3s 支持

OrbStack：以极低的资源占用和接近原生的 I/O 性能著称。对于 Apple Silicon 机型上需要频繁构建镜像的开发者，体验提升尤为明显。个人和教育用途免费。

Colima：完全开源的命令行方案，底层基于 Lima 虚拟机。适合偏好终端工作流、不需要 GUI 的开发者，或企业中希望避免商业许可限制的团队。

> 上述三种工具均兼容标准的 `docker` CLI 命令。从 Docker Desktop 切换到 OrbStack 或 Colima 时，已有的镜像和容器配置通常可以平滑迁移。

3.7.5 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.8 Windows 10/11

在 Windows 平台上，Docker Desktop 提供了完整的 Docker 开发环境。本节介绍在 Windows 10/11 上的安装和配置。

Windows 上的 Docker：运行原理理解

与 macOS 类似，Windows 也没有原生 Linux 容器支持。Docker Desktop for Windows 有两种运行后端可选：

WSL 2（Windows Subsystem for Linux 2） - 推荐：

- 利用 Hyper-V 虚拟化运行真正的 Linux 内核
- 性能更好，文件系统集成更深
- 现代 Windows 10/11 的标准选择
- 支持在 Linux 和 Windows 之间的无缝文件访问

Hyper-V - 传统方案：

- 纯虚拟化方式
- 性能略低于 WSL 2
- 在某些企业网络环境下仍被使用

实践建议：WSL 2 和 Hyper-V 在功能上都能满足 Docker Desktop 的日常开发需求，选择哪种后端应以机器能力、企业策略和你的工作流为准；如果系统只满足其中一种后端的前置条件，安装器才会自动选择可用的那一种。

3.8.1 系统要求

Docker Desktop for Windows 支持 Docker 官方文档列出的受支持 Windows 10/11 64 位版本（具体以官方 安装文档 为准）。若使用 WSL 2 后端，需要启用 WSL 2，并满足官方要求的 `WSL 2.1.5` 或更高版本；若使用 Hyper-V 后端，则需要启用 Hyper-V 和 Containers 功能。Windows 10 64 位支持 Enterprise、Pro 和 Education 22H2（build 19045），Windows 11 64 位支持 Enterprise、Pro 和 Education 23H2（build 22631）或更高版本，且官方建议主机至少具备 8 GB 内存。

3.8.2 安装

> ⚠️ **Warning**
>
> **商业许可限制**：Docker Desktop 对小型企业（少于 250 名员工且年收入少于 1000 万美元）、个人使用、教育和非商业开源项目仍然免费。对于其他商业用途，以及政府机构使用，需要付费订阅。企业用户请注意合规风险。

手动下载安装

官方当前提供三个主要入口：

- Docker Desktop for Windows x86_64 安装包
- Docker Desktop for Windows Microsoft Store 版本
- Docker Desktop for Windows Arm 早期访问版

下载好对应安装包后，双击 `Docker Desktop Installer.exe` 开始安装。

使用winget安装

```
$ winget install Docker.DockerDesktop
```

3.8.3 在 WSL2 运行 Docker

若你的环境使用 WSL 2 后端，请先确认 `wsl --version` 满足 Docker 官方的版本要求，并按 Docker Desktop 的 WSL 说明启用对应功能。

3.8.4 运行

在 Windows 搜索栏输入 **Docker** 点击 **Docker Desktop** 开始运行。

图

Docker 启动之后会在 Windows 任务栏出现鲸鱼图标。

图

等待片刻，当鲸鱼图标静止时，说明 Docker 启动成功，之后你可以打开 PowerShell 使用 Docker。

> 推荐使用 Windows Terminal 在终端使用 Docker。

3.8.5 镜像加速

如果在使用过程中发现拉取 Docker 镜像十分缓慢，可以配置 Docker 国内镜像加速。

3.9 镜像加速器

国内从 Docker Hub 拉取镜像有时会遇到困难，此时可以配置镜像加速器。

> ⚠️ **注意**：镜像加速器的可用性经常变化。配置前请先访问 docker-practice/docker-registry-cn-mirror-test 查看各镜像站的实时状态。

3.9.1 推荐配置方案

针对不同的使用场景，我们推荐以下几种镜像加速配置方案，以确保最佳的拉取速度。

> ⚠️ **重要提示**：国内大多数 Docker Hub 加速服务已于 2024 年中旬关闭（包括阿里云、腾讯云、网易云、百度云等）。如下推荐的镜像源可用性因人而异，建议先测试可用性再配置。

1. **云服务器用户**：优先使用所在云平台提供的内部加速器（见本页末尾“云服务商”部分）
2. **本地开发用户**：优先使用自建 pull-through cache；如果必须依赖社区镜像，请先用上面的测试仓库确认实时可用性
3. **代理方案**：如有条件，可配置 HTTP 代理直接访问 Docker Hub

更稳妥的长期方案是使用**自己控制的 pull-through cache / registry mirror**，或者优先使用云厂商提供的内网镜像。下文统一用 `https://<your-registry-mirror>` 作为占位符；如果你选择第三方公共站，请先确认可用性、服务条款和缓存策略。

3.9.2 Ubuntu 22.04+、Debian 12+、Rocky/Alma/CentOS Stream 9+

目前主流 Linux 发行版均已使用 systemd 进行服务管理，这里介绍如何在使用 systemd 的 Linux 发行版中配置镜像加速器。

请首先执行以下命令，查看是否在 `docker.service` 文件中配置过镜像地址。

```
$ systemctl cat docker | grep '\-\-registry\-mirror'
```

如果该命令有输出，那么请执行 `$ systemctl cat docker` 查看 `ExecStart=` 出现的位置，修改对应的文件内容去掉 `--registry-mirror` 参数及其值，并按接下来的步骤进行配置。

如果以上命令没有任何输出，那么就可以在 `/etc/docker/daemon.json` 中写入如下内容 (如果文件不存在请新建该文件)：

```
{
  "registry-mirrors": [
    "https://<your-registry-mirror>"
  ]
}
```

> 注意，一定要保证该文件符合 json 规范，否则 Docker 将不能启动。

之后重新启动服务。

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

3.9.3 Windows 10/11

对于使用 `Windows 10/11` 的用户，在任务栏托盘 Docker 图标内打开 `Settings`，在左侧导航菜单选择 `Docker Engine`，在右侧像下边一样编辑 JSON 文件，之后点击 `Apply & Restart` 保存后 Docker 就会重启并应用配置的镜像地址了。

```
{
  "registry-mirrors": [
    "https://<your-registry-mirror>"
  ]
}
```

3.9.4 macOS

对于使用 macOS 的用户，在任务栏点击 Docker Desktop 应用图标 -> `Settings...`，在左侧导航菜单选择 `Docker Engine`，在右侧像下边一样编辑 json 文件。修改完成之后，点击 `Apply & restart` 按钮，Docker 就会重启并应用配置的镜像地址了。

```
{
  "registry-mirrors": [
    "https://<your-registry-mirror>"
  ]
}
```

3.9.5 检查加速器是否生效

执行 `$ docker info`，如果从结果中看到了如下内容，说明配置成功。

```
Registry Mirrors:
 https://<your-registry-mirror>/
```

3.9.6 Kubernetes 官方镜像地址迁移

可以登录 阿里云容器镜像服务，在 **镜像中心** -> **镜像搜索** 中查找。

Kubernetes 社区已将官方镜像地址从 `k8s.gcr.io` 迁移到 `registry.k8s.io`。建议优先使用新地址。

一般情况下有如下对应关系：

```
$ docker pull registry.k8s.io/xxx
```

3.9.7 已停止服务的镜像列表

以下镜像源已停止服务，添加无用的镜像加速器会拖慢拉取速度，请从配置中删除：

- https://hub.atomgit.com （已于 2024 年底关闭）
- https://registry.cn-hangzhou.aliyuncs.com （阿里云 Docker 加速已于 2024 年关闭）
- https://dockerhub.azk8s.cn （已转为私有）
- https://reg-mirror.qiniu.com （已停止服务）
- https://registry.docker-cn.com （已停止服务）
- https://hub-mirror.c.163.com （网易云镜像已于 2024 年关闭）
- https://mirror.baidubce.com （百度云镜像已停止）

建议 **watch（页面右上角）** 镜像测试仓库 这个 GitHub 仓库，我们会持续更新各镜像源的可用状态。

3.9.8 云服务商

某些云服务商提供了 **仅供内部** 访问的镜像服务，当您的 Docker 运行在云平台时可以选择它们。

- 腾讯云 `https://mirror.ccs.tencentyun.com`

Built with mdPress

3.10 开启实验特性

一些 docker 命令或功能仅当 **实验特性** 开启时才能使用，请按照以下方法进行设置。

3.10.1 Docker CLI 的实验特性

CLI 的实验特性通常包含仍在开发中的新功能。幸运的是，在较新版本中这些特性已经更加易用。

从 `v20.10` 及更高版本开始，Docker CLI 所有实验特性的命令均默认开启，无需再进行配置或设置系统环境变量。

3.10.2 开启 dockerd 的实验特性

编辑 `/etc/docker/daemon.json`，新增如下条目

```
{
  "experimental": true
}
```

保存后重启 Docker daemon：

```
$ sudo systemctl restart docker
```

然后执行下面的命令验证服务端实验特性已经生效：

```
$ docker version
```

若输出中的 `Server` / `Engine` 部分出现 `Experimental: true`，说明 daemon 端实验特性已经启用。

本章小结

Docker 支持在多种平台上安装和使用，选择合适的安装方式是顺利使用 Docker 的第一步。

平台	推荐方式	说明
Ubuntu/Debian	官方 APT 仓库	最完善的支持，推荐首选
CentOS/Fedora	官方 DNF/YUM 仓库	注意验证防火墙与 `iptables` 兼容性
macOS	Docker Desktop	图形化安装，默认集成 Compose
Windows 10/11	Docker Desktop（WSL 2 或 Hyper-V）	按机器能力与企业策略选择后端
Raspberry Pi	官方 APT 仓库或 Debian `arm64` 方案	32 位系统已停止接收 v29+ 新主版本
离线环境	二进制包安装	适用于无法联网的服务器

安装后验证

安装完成后，运行以下命令验证 Docker 是否正常工作：

```
$ docker version
$ docker run --rm hello-world
```

延伸阅读

- 镜像加速器：解决国内拉取镜像慢的问题
- 开启实验特性：使用最新功能
- Docker Hub：官方镜像仓库

Built with mdPress

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第四章 使用镜像

在之前的介绍中，我们知道镜像是 Docker 的三大组件之一。

Docker 运行容器前需要本地存在对应的镜像，如果本地不存在该镜像，Docker 会从镜像仓库下载该镜像。

本章内容

本章将介绍更多关于镜像的内容，包括：

- 从仓库获取镜像
- 列出镜像
- 删除本地镜像
- 利用 commit 理解镜像构成
- 使用 Dockerfile 定制镜像
- 其它制作镜像的方式
- 镜像的实现原理

> **版本提示：镜像存储后端的变迁**
>
> 在 Docker Engine v29 及后续版本中，Docker 在**全新安装场景**默认启用 **containerd image store**（替代传统 classic store 路径）。这一底层架构级别的变迁，意味着 Docker 解锁了对 OCI Image Index 和 Attestations（例如原生的 provenance 来源证明与 SBOM 软件物料清单）的全量本地支持。 读者在执行类似 `docker buildx build --provenance=mode=min --sbom=true` 甚至使用后续审查工具（如 `docker buildx imagetools inspect`）时，其元数据能够与镜像数据一并完好地管理于本地存储系统中，为供应链安全验证补齐了最后一块拼图。

Built with mdPress

4.1 获取镜像

从 Docker 镜像仓库获取镜像可谓是 Docker 运作的第一步。本节将介绍如何使用 `docker pull` 命令下载镜像，以及如何理解下载过程。

> **版本号最佳实践**
>
> - **永远指定版本号**：避免使用 `latest` 标签，应指定具体的版本（如 `ubuntu:24.04`、`nginx:1.28`），以确保镜像内容稳定一致。
> - **在生产环境使用摘要**：优先使用镜像摘要（SHA256）而非标签，如 `nginx@sha256:abc123...`，因为摘要不可变。
> - **定期评估依赖**：即使指定了版本号，仍应定期检查依赖的基础镜像是否有安全更新。

4.1.1 docker pull 命令

从镜像仓库获取镜像的命令是 `docker pull`：

```
docker pull [选项] [Registry地址/]仓库名[:标签]
```

镜像名称格式

Docker 镜像名称由 Registry 地址、用户名、仓库名和标签组成。其标准格式如下：

```
docker.io / library / ubuntu : 24.04
──────┬──── ──────┬────── ──────┬──── ──────┬──────
      │           │             │           │
Registry地址   用户名        仓库名       标签
 (可省略)    (可省略)
```

组成部分	说明	默认值
Registry 地址	镜像仓库地址	`docker.io` (Docker Hub)
用户名	镜像所属用户/组织	`library` (官方镜像)
仓库名	镜像名称	必须指定
标签	版本标识	`latest`

示例

```
## 完整格式

$ docker pull docker.io/library/ubuntu:24.04

## 省略 Registry（默认 Docker Hub）

$ docker pull library/ubuntu:24.04

## 省略 library（官方镜像）

$ docker pull ubuntu:24.04

## 省略标签（默认 latest）

$ docker pull ubuntu

## 拉取第三方镜像

$ docker pull bitnami/redis:latest

## 从其他 Registry 拉取

$ docker pull ghcr.io/username/myapp:v1.0
```

4.1.2 下载过程解析

当我们执行 `docker pull` 命令时，Docker 会输出详细的下载进度。让我们以 `ubuntu:24.04` 为例来解析这些信息。

```
$ docker pull ubuntu:24.04
24.04: Pulling from library/ubuntu
92dc2a97ff99: Pull complete
be13a9d27eb8: Pull complete
c8299583700a: Pull complete
Digest: sha256:4bc3ae6596938cb0d9e5ac51a1152ec9dcac2a1c50829c74abd9c4361e321b26
Status: Downloaded newer image for ubuntu:24.04
docker.io/library/ubuntu:24.04
```

输出解读

输出内容	说明
`Pulling from library/ubuntu`	正在从官方 ubuntu 仓库拉取
`92dc2a97ff99: Pull complete`	各层的下载状态 (显示层 ID 前 12 位)
`Digest: sha256:...`	镜像内容的唯一摘要
`docker.io/library/ubuntu:24.04`	镜像的完整名称

分层下载

从输出可以看到，镜像是 **分层下载** 的：



如果本地已有相同的层，Docker 会跳过下载，节省带宽和时间。

4.1.3 常用选项

`docker pull` 命令支持多种选项来满足不同的下载需求，例如下载所有标签、指定平台架构等。

选项	说明	示例
`--all-tags, -a`	拉取所有标签	`docker pull -a ubuntu`
`--platform`	指定平台架构	`docker pull --platform linux/arm64 nginx`
`--quiet, -q`	静默模式	`docker pull -q nginx`

指定平台

在 Apple Silicon Mac 上拉取 x86 镜像：

```
$ docker pull --platform linux/amd64 nginx
```

4.1.4 拉取后运行

拉取镜像后，可以基于它启动容器：

```
## 拉取镜像

$ docker pull ubuntu:24.04

## 运行容器

$ docker run -it --rm ubuntu:24.04 bash
root@e7009c6ce357:/# cat /etc/os-release
PRETTY_NAME="Ubuntu 24.04 LTS"
...
root@e7009c6ce357:/# exit
```

> 本例使用 `ubuntu:24.04` 这样的具体版本标签是最佳实践。若无特殊需求，避免使用 `docker pull ubuntu` 或 `ubuntu:latest`，因为镜像内容可能在某个时刻发生变化。

参数说明：

参数	说明
`-it`	交互式终端模式
`--rm`	退出后自动删除容器
`bash`	启动命令

> 💡 `docker run` 在需要时会自动 `pull` 镜像，因此通常不需要单独执行 `docker pull`。

4.1.5 镜像加速

从 Docker Hub 下载可能较慢。可以配置镜像加速器：

```
// /etc/docker/daemon.json (Linux)
// ~/.docker/daemon.json (Docker Desktop)
{
  "registry-mirrors": [
    "https://your-accelerator-url"
  ]
}
```

配置后重启 Docker：

```
$ sudo systemctl restart docker  # Linux

## 或在 Docker Desktop 中重启
```

详见镜像加速器章节。

4.1.6 验证镜像完整性

为了确保下载的镜像没有被篡改且内容一致，我们可以校验镜像的摘要 (Digest)。

查看镜像摘要

```
$ docker images --digests ubuntu
REPOSITORY   TAG     DIGEST                                                                  IMAGE
ID
ubuntu       24.04   sha256:4bc3ae6596938cb0d9e5ac51a1152ec9dcac2a1c50829c74abd9c4361e321b26   ca2b0
f26964c
```

使用摘要拉取

用摘要拉取可确保获取完全相同的镜像：

```
$ docker pull ubuntu@sha256:4bc3ae6596938cb0d9e5ac51a1152ec9dcac2a1c50829c74abd9c4361e321b26
```

> 笔者建议：生产环境使用摘要而非标签，因为标签可能被覆盖，摘要则是不可变的。

4.1.7 常见问题

在使用 `docker pull` 过程中，可能会遇到下载速度慢、镜像不存在或磁盘空间不足等问题。以下是一些常见问题的排查思路。

Q：下载速度很慢

1. 配置镜像加速器
2. 检查网络连接
3. 尝试拉取更小的镜像版本 (如 `alpine` 变体)

Q：提示镜像不存在

```
Error: pull access denied, repository does not exist
```

可能原因：

- 镜像名拼写错误
- 私有镜像未登录 (需要 `docker login`)
- 镜像确实不存在

Q：磁盘空间不足

```
## 清理未使用的镜像

$ docker image prune

## 清理所有未使用资源

$ docker system prune
```

4.2 列出镜像

在下载了镜像后，我们可以使用 `docker image ls` 命令列出本地主机上的镜像。

4.2.1 基本用法

查看本地已下载的镜像：

```
$ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
redis        latest    5f515359c7f8   5 days ago     183MB
nginx        latest    05a60462f8ba   5 days ago     181MB
ubuntu       24.04     329ed837d508   3 days ago     78MB
ubuntu       noble     329ed837d508   3 days ago     78MB
```

> 💡 `docker images` 是 `docker image ls` 的简写，两者等效。

4.2.2 输出字段说明

`docker image ls` 命令默认输出的列表包含仓库名、标签、镜像 ID、创建时间和占用空间等信息。

字段	说明
REPOSITORY	仓库名
TAG	标签 (版本)
IMAGE ID	镜像唯一标识 (短 ID，前 12 位)
CREATED	创建时间
SIZE	本地占用空间

同一镜像多个标签

注意上面的 `ubuntu:24.04` 和 `ubuntu:noble` 拥有相同的 IMAGE ID——它们是同一个镜像的不同标签，只占用一份存储空间。

> **版本说明**：`ubuntu:24.04` 是具体版本号，`ubuntu:noble` 是发布代号（Ubuntu 24.04 的代号）。在 Dockerfile 中应优先使用版本号（如 `ubuntu:24.04`）而非发布代号，因为版本号在将来更易理解。

4.2.3 理解镜像大小

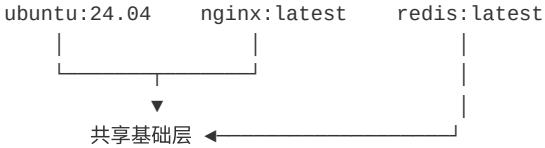
Docker 镜像的大小可能与我们通常理解的文件大小有所不同，这涉及到分层存储的概念。

本地大小 vs Hub 显示大小

位置	显示大小	说明
Docker Hub	29MB	压缩后的网络传输大小
docker image ls	78MB	本地解压后的实际大小

实际磁盘占用

由于镜像是分层存储，不同镜像可能共享相同的层：



因此，`docker image ls` 中各镜像大小之和 > 实际磁盘占用。

查看实际空间占用

```
$ docker system df
TYPE            TOTAL   ACTIVE   SIZE      RECLAIMABLE
Images          15      3        2.5GB     1.8GB (72%)
Containers      5       2        100MB     80MB (80%)
Local Volumes   8       2        500MB     400MB (80%)
Build Cache     0       0        0B        0B
```

4.2.4 过滤镜像

随着本地镜像数量的增加，我们需要更有效的方式来查找特定的镜像。Docker 提供了多种过滤方式。

按仓库名过滤

```
## 列出所有 ubuntu 镜像

$ docker images ubuntu
REPOSITORY   TAG     IMAGE ID       SIZE
ubuntu       24.04   329ed837d508   78MB
ubuntu       noble   329ed837d508   78MB
ubuntu       22.04   a1b2c3d4e5f6   72MB
```

按仓库名和标签过滤

```
$ docker images ubuntu:24.04
REPOSITORY   TAG     IMAGE ID       SIZE
ubuntu       24.04   329ed837d508   78MB
```

使用过滤器 --filter

过滤条件	说明	示例
`dangling=true`	虚悬镜像	`-f dangling=true`
`before=镜像`	在某镜像之前创建	`-f before=nginx:latest`
`since=镜像`	在某镜像之后创建	`-f since=nginx:latest`
`label=key=value`	按 LABEL 过滤	`-f label=version=1.0`
`reference=pattern`	按名称模式	`-f reference='*:latest'`

```
## 列出 nginx 之后创建的镜像

$ docker images -f since=nginx:latest

## 列出所有带 latest 标签的镜像

$ docker images -f reference='*:latest'

## 列出带特定 LABEL 的镜像

$ docker images -f label=maintainer=example@email.com
```

4.2.5 虚悬镜像

在镜像列表里，你可能会看到一些仓库名和标签都为 <none> 的镜像，这类镜像被称为虚悬镜像。

什么是虚悬镜像

仓库名和标签都显示为 <none> 的镜像：

```
$ docker images
REPOSITORY   TAG       IMAGE ID       SIZE
<none>       <none>    00285df0df87   342MB
```

产生原因

1. **镜像重新构建**：新镜像使用了旧镜像的标签，旧镜像标签被移除
2. **docker pull 更新**：拉取更新版本时，旧版本失去标签

处理虚悬镜像

```
## 列出虚悬镜像

$ docker images -f dangling=true

## 删除虚悬镜像

$ docker image prune
```

4.2.6 中间层镜像

除了虚悬镜像，`docker image ls` 默认列出的只是顶层镜像。还有一种镜像是为了加速镜像构建、重复利用资源而存在的中间层镜像。

查看所有镜像：包含中间层

```
$ docker images -a
```

会显示很多无标签镜像——这些是构建过程中产生的中间层，被其他镜像依赖。

> ⚠️ 不要删除中间层镜像。它们是其他镜像的依赖，删除会导致上层镜像无法使用。删除顶层镜像时会自动清理不再需要的中间层。

4.2.7 格式化输出

为了配合脚本使用或展示更关注的信息，我们可以使用 `--format` 参数来自定义输出格式。

只输出 ID

```
$ docker images -q
5f515359c7f8
05a60462f8ba
329ed837d508
```

常用于配合其他命令：

```
## 删除所有镜像

$ docker rmi $(docker images -q)

## 删除所有 redis 镜像

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

显示完整 ID

```
$ docker images --no-trunc
```

显示摘要

```
$ docker images --digests
REPOSITORY   TAG     DIGEST                    IMAGE ID
nginx        latest  sha256:b4f0e0bdeb5...    e43d811ce2f4
```

自定义格式

使用 Go 模板语法自定义输出：

```
## 只显示 ID 和仓库名

$ docker images --format "{{.ID}}: {{.Repository}}"
5f515359c7f8: redis
05a60462f8ba: nginx
329ed837d508: ubuntu

## 表格形式（带标题）

$ docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
REPOSITORY   TAG       SIZE
redis        latest    183MB
nginx        latest    181MB
ubuntu       24.04     78MB
```

可用模板字段

字段	说明
`.ID`	镜像 ID
`.Repository`	仓库名
`.Tag`	标签
`.Digest`	摘要
`.CreatedSince`	创建后经过的时间
`.CreatedAt`	创建时间
`.Size`	大小

Built with mdPress

4.2.8 常用命令组合

```
## 列出所有镜像及其大小，按大小排序（需要系统 sort 命令）

$ docker images --format "{{.Size}}\t{{.Repository}}:{{.Tag}}" | sort -h

## 查找大于 500MB 的镜像

$ docker images --format "{{.Size}}\t{{.Repository}}:{{.Tag}}" | grep -E "^[0-9]+GB|^[5-9][0-9]{2}MB"

## 导出镜像列表

$ docker images --format "{{.Repository}}:{{.Tag}}" > images.txt
```

4.3 删除本地镜像

当不再需要某个镜像时，我们可以将其删除以释放存储空间。本节介绍删除镜像的常用方法。

4.3.1 基本用法

使用 `docker image rm` 删除本地镜像：

```
$ docker image rm [选项] <镜像1> [<镜像2> ...]
```

> 💡 `docker rmi` 是 `docker image rm` 的简写，两者等效。

4.3.2 镜像标识方式

删除镜像时，可以使用多种方式指定镜像：

方式	说明	示例
短 ID	ID 的前几位 (通常 3-4 位)	`docker rmi 501`
长 ID	完整的镜像 ID	`docker rmi 501ad78535f0...`
镜像名:标签	仓库名和标签	`docker rmi redis:7.0`
镜像摘要	精确的内容摘要	`docker rmi nginx@sha256:...`

> **版本提示**：建议使用 **镜像名:标签** 的方式删除，特别是当需要明确清理特定版本的镜像时。例如 `docker rmi redis:7.0` 比 `docker rmi redis:latest` 更清晰且安全。

使用短 ID 删除

```
$ docker image ls
REPOSITORY   TAG     IMAGE ID       SIZE
redis        alpine  501ad78535f0   30MB
nginx        latest  e43d811ce2f4   142MB

## 只需输入足够区分的前几位

$ docker rmi 501
Untagged: redis:alpine
Deleted: sha256:501ad78535f0...
```

使用镜像名删除

```
$ docker rmi redis:alpine
Untagged: redis:alpine
Deleted: sha256:501ad78535f0...
```

使用摘要删除

摘要删除最精确，适用于 CI/CD 场景：

```
## 查看镜像摘要

$ docker images --digests
REPOSITORY   TAG    DIGEST                    IMAGE ID
nginx        latest sha256:b4f0e0bdeb5...     e43d811ce2f4

## 使用摘要删除

$ docker rmi nginx@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228
```

4.3.3 理解输出信息

执行删除命令后，Docker 会输出一系列的操作记录，理解这些信息有助于我们掌握镜像删除的机制。

删除镜像时会看到两类信息：**Untagged** 和 **Deleted**

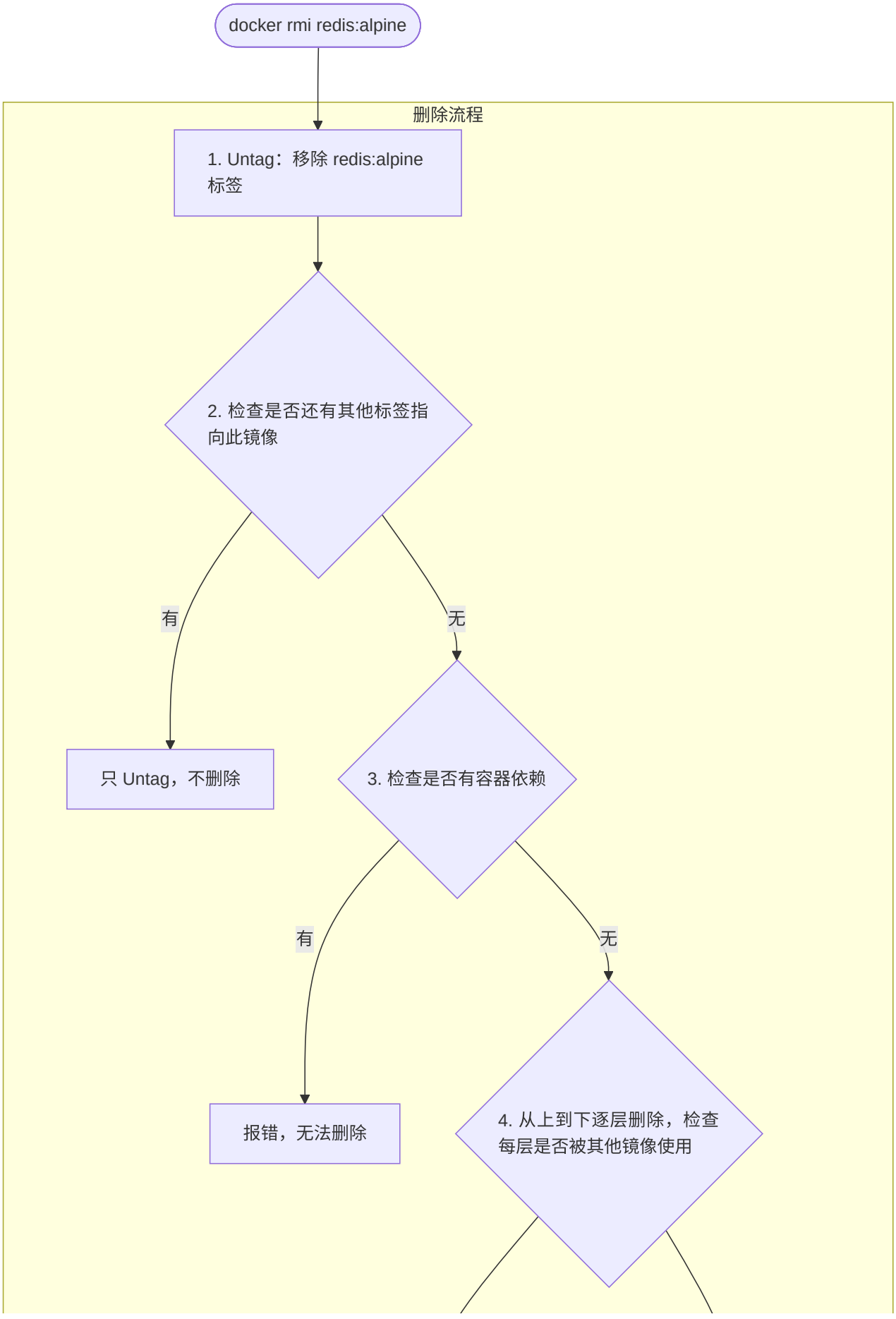
```
$ docker rmi redis:alpine
Untagged: redis:alpine
Untagged: redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeea521a86d
Deleted: sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7
Deleted: sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b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

Untagged vs Deleted

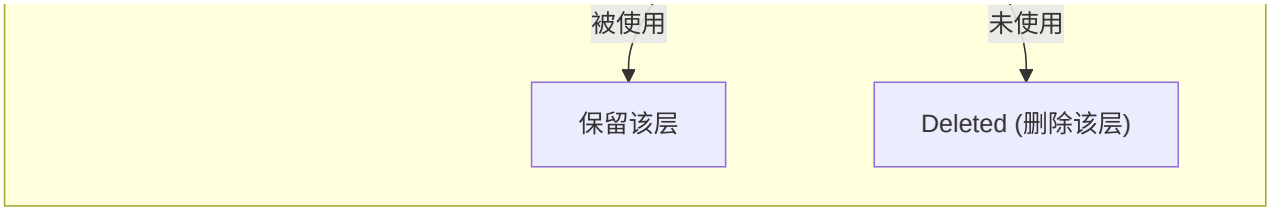
操作	含义
Untagged	移除镜像的标签
Deleted	删除镜像的存储层

删除流程

Docker 会检测镜像是否有容器依赖或其他标签指向，只有在确认为无用资源时才会真正删除存储层。



Built with mdPress



4.3.4 批量删除

手动一个一个删除镜像非常繁琐，Docker 提供了 `image prune` 命令和 shell 组合命令来实现批量清理。

删除所有虚悬镜像

虚悬镜像 (dangling)：没有标签的镜像，通常是旧版本被新版本覆盖后产生的

```
## 查看虚悬镜像

$ docker images -f dangling=true

## 删除虚悬镜像

$ docker image prune

## 不提示确认

$ docker image prune -f
```

删除所有未使用的镜像

```
## 删除所有没有被容器使用的镜像

$ docker image prune -a

## 保留最近 24 小时的

$ docker image prune -a --filter "until=24h"
```

按条件删除

```
## 删除所有 redis 镜像

$ docker rmi $(docker images -q redis)

## 删除 mongo:8.0 之前的所有镜像

$ docker rmi $(docker images -q -f before=mongo:8.0)

## 删除某个时间之前的镜像

$ docker image prune -a --filter "until=168h"  # 7天前
```

4.3.5 删除失败的常见原因

在删除镜像时，Docker 可能会提示错误并拒绝执行。这通常是为了防止误删正在使用的资源。

原因一：有容器依赖

```
$ docker rmi nginx
Error: conflict: unable to remove repository reference "nginx"
(must force) - container abc123 is using its referenced image
```

解决方案：

```
## 方案1：先删除依赖的容器

$ docker rm abc123
$ docker rmi nginx

## 方案2：强制删除镜像（容器仍可运行，但无法再创建新容器）

$ docker rmi -f nginx
```

原因二：多个标签指向同一镜像

```
$ docker images
REPOSITORY   TAG     IMAGE ID
ubuntu       24.04   ca2b0f26964c
ubuntu       latest  ca2b0f26964c   # 同一个镜像

$ docker rmi ubuntu:24.04
Untagged: ubuntu:24.04

## 只是移除标签，镜像仍存在（因为还有 ubuntu:latest 指向它）
```

当同一个镜像有多个标签时，`docker rmi` 只是删除指定的标签，不会删除镜像本身。

原因三：被其他镜像依赖：中间层

```
$ docker rmi some_base_image
Error: image has dependent child images
```

中间层镜像被其他镜像依赖，无法删除。需要先删除依赖它的镜像。

4.3.6 常用过滤条件

过滤条件	说明	示例
`dangling=true`	虚悬镜像	`-f dangling=true`
`before=镜像`	在某镜像之前	`-f before=mongo:3.2`
`since=镜像`	在某镜像之后	`-f since=mongo:3.2`
`label=key=value`	按标签过滤	`-f label=version=1.0`
`reference=pattern`	按名称模式	`-f reference='*:latest'`

4.3.7 清理策略

针对不同的环境 (开发环境 vs 生产环境)，我们应该采用不同的镜像清理策略。

开发环境

```
## 定期清理虚悬镜像

$ docker image prune -f

## 一键清理所有未使用资源

$ docker system prune -a
```

CI/CD 环境

```
## 只保留最近使用的镜像

$ docker image prune -a --filter "until=72h" -f
```

查看空间占用

```
$ docker system df
TYPE            TOTAL   ACTIVE   SIZE      RECLAIMABLE
Images          15      3        2.5GB     1.8GB (72%)
Containers      5       2        100MB     80MB (80%)
Local Volumes   8       2        500MB     400MB (80%)
Build Cache     0       0        0B        0B
```

4.4 利用 commit 理解镜像构成

> 注意：如果你是初学者，可以暂时跳过后面的内容，直接学习容器一节。

`docker commit` 除了帮助理解镜像分层之外，在少数场景下也可用于留存现场，例如事后分析被入侵容器的状态；但是，日常定制镜像不应依赖 `docker commit`，而应使用下一节介绍的 `Dockerfile`。

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

回顾一下之前我们学到的知识，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，它以镜像层为只读基础，并在最上方增加一层供运行时写入的容器层。

现在让我们以定制一个 Web 服务器为例子，来讲解镜像是如何构建的。

> **版本提示**：以下示例中 `nginx` 镜像使用默认 `latest` 标签。生产环境建议指定具体版本号（如 `nginx:1.28`），以避免镜像更新带来的不兼容性。

```
$ docker run --name webserver -d -p 8080:80 nginx
```

这条命令会用 `nginx` 镜像启动一个容器，命名为 `webserver`。其中，`-p 8080:80` 表示把宿主机的 `8080` 端口映射到容器内的 `80` 端口，这样我们就可以用浏览器去访问这个 `nginx` 服务器。

如果是在本机运行的 Docker，那么可以直接访问：`http://localhost:8080`；如果是在虚拟机、云服务器上安装的 Docker，则需要将 `localhost` 换为虚拟机地址或者实际云服务器地址，并保留 `8080` 端口。

直接用浏览器访问的话，我们会看到默认的 Nginx 欢迎页面。

图

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

```
$ docker exec -it webserver bash
root@3729b97e8226:/# echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
root@3729b97e8226:/# exit
exit
```

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

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

现在我们再刷新浏览器的话，会发现内容被改变了。

图

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

```
$ docker diff webserver
C /root
A /root/.bash_history
C /run
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
C /var
C /var/cache
C /var/cache/nginx
A /var/cache/nginx/client_temp
A /var/cache/nginx/fastcgi_temp
A /var/cache/nginx/proxy_temp
A /var/cache/nginx/scgi_temp
A /var/cache/nginx/uwsgi_temp
```

其中，`A` 表示新增（Added），`C` 表示变更（Changed），`D` 表示删除（Deleted）。

现在我们定制好了变化，我们希望能将其保存下来形成镜像。

要知道，当我们运行一个容器的时候 (如果不使用卷的话)，我们做的任何文件修改都会被记录于容器存储层里。而 Docker 提供了一个 `docker commit` 命令，可以将容器的存储层保存下来成为镜像。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。

`docker commit` 的语法格式为：

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

我们可以用下面的命令将容器保存为镜像。默认情况下，`docker commit` 会在提交时暂停容器进程，以降低数据损坏的风险；如果确实不希望暂停，可以显式指定 `--no-pause`：

```
$ docker commit \
    --author "Tao Wang <twang2218@gmail.com>" \
    --message "修改了默认网页" \
    webserver \
    nginx:v2
sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa1795214
```

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

我们可以在 `docker image ls` 中看到这个新定制的镜像。下面的输出仅为示例，标签、创建时间和大小会随着镜像版本和本地环境不同而变化：

```
$ docker image ls nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               v2                  07e334659748        9 seconds ago       181.5 MB
nginx               1.30                05a60462f8ba        12 days ago         181.5 MB
nginx               latest              e43d811ce2f4        4 weeks ago         181.5 MB
```

> **版本说明**：上面示例中 `nginx:1.30` 代表 1.30 系列的最新 patch 版本。在实际应用中应根据需求选择确切的版本号，而不是盲目使用 `latest`。

我们还可以用 `docker history` 具体查看镜像内的历史记录。例如先执行 `docker history nginx:v2`，再对比 `docker history nginx:latest`，就能看到我们刚刚提交出来的新层。

```
$ docker history nginx:v2
IMAGE               CREATED             CREATED BY                                      SIZE
COMMENT
07e334659748        54 seconds ago      nginx -g daemon off;                            95 B
修改了默认网页
e43d811ce2f4        4 weeks ago         /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon    0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  EXPOSE 443/tcp 80/tcp        0 B
<missing>           4 weeks ago         /bin/sh -c ln -sf /dev/stdout /var/log/nginx/   22 B
<missing>           4 weeks ago         /bin/sh -c apt-key adv --keyserver hkp://pgp.   58.46 MB
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENV NGINX_VERSION=1.27.0-1   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  MAINTAINER NGINX Docker Ma   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD file:23aa4f893e3288698c   123 MB
```

新的镜像定制好后，我们可以来运行这个镜像。

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

这里我们将新容器命名为 `web2`，并把宿主机的 `81` 端口映射到容器的 `80` 端口。访问 `http://localhost:81` 后，看到的内容应该和之前修改后的 `webserver` 一样。

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

4.4.1 慎用 `docker commit`

使用 `docker commit` 命令虽然可以比较直观地帮助理解镜像分层存储的概念，但它不应作为常规定制镜像的方式。

首先，如果仔细观察之前的 `docker diff webserver` 的结果，你会发现除了真正想要修改的 `/usr/share/nginx/html/index.html` 文件外，由于命令的执行，还有很多文件被改动或添加了。这还仅仅是最简单的操作，如果是安装软件包、编译构建，那会有大量的无关内容被添加进来，将会导致镜像极为臃肿。

此外，使用 `docker commit` 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为 **黑箱镜像**，换句话说，就是除了制作镜像的人知道执行过什么命令、怎么生成的镜像，别人根本无从得知。而且，即使是这个制作镜像的人，过一段时间后也无法记清具体的操作。这种黑箱镜像的维护工作是非常痛苦的。

而且，回顾之前提及的镜像所使用的分层存储的概念，除当前层外，之前的每一层都不会发生改变。换句话说，任何修改的结果仅仅是在当前层进行标记、添加、修改，而不会改动上一层。如果使用 `docker commit` 制作镜像，以及后期继续修改，那么每一次修改都会让镜像再膨胀一层；即使某些文件在更高层里被删除了，它们仍然存在于更低层中，只是在最终视图里被隐藏而已。因此，日常定制镜像应使用下一节介绍的 `Dockerfile`，把构建过程写成可重复执行、便于审查的文本。

4.5 使用 Dockerfile 定制镜像

从刚才的 `docker commit` 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复、镜像构建不透明、体积难以控制等问题就会更容易解决。这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内包含了一条条的 **指令 (Instruction)**。其中，会修改文件系统的指令通常会创建新层；而 `LABEL`、`CMD` 这类只修改镜像元数据的指令，则不会新增文件系统层。每一条指令的内容，都是在描述该镜像应当如何构建。

4.5.1 使用 docker init 快速创建：推荐

Docker 提供了 `docker init` 命令，可以根据项目类型自动生成 `Dockerfile`、`.dockerignore`、`compose.yaml` 和 `README.Docker.md` 等文件：

```
$ docker init
```

该命令会交互式地询问项目类型（支持 Go、Node.js、Python、Rust、Java、ASP.NET Core、PHP with Apache 等），并生成可作为起点的配置文件。对于新项目，这是一个很好的起步方式，但生成后的内容仍应结合项目实际情况继续调整。

4.5.2 手动创建 Dockerfile

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

在一个空白目录中，建立一个文本文件，并命名为 `Dockerfile`：

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

其内容为：

> **版本提示**：下面示例中 `FROM nginx` 使用的是 `latest` 标签。在实际应用中应使用明确的版本号（如 `FROM nginx:1.28`），以确保 Dockerfile 的可重现性和稳定性。

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

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

4.5.3 FROM 指定基础镜像

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 FROM 就是指定 **基础镜像**，因此一个 Dockerfile 中 FROM 是必备的指令，并且必须是第一条指令。

> **版本号最佳实践**：在 FROM 指令中 **务必指定具体版本号**（如 FROM ubuntu:24.04 或 FROM python:3.12-slim）而非 FROM ubuntu 或 FROM python:latest。这样可以确保 Dockerfile 在不同时间、不同环境下构建出的镜像内容一致，避免因基础镜像更新导致的不可预期的变化。

在 Docker Hub 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 nginx、redis、mongo、mysql、httpd、php、tomcat 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 node、eclipse-temurin、python、ruby、golang 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 ubuntu、debian、centos、fedora、alpine 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

```
FROM scratch
...
```

如果你以 scratch 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，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，这也是有人认为 Go 是特别适合容器微服务架构的语言的原因之一。

4.5.4 RUN 执行命令

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

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

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

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

在会修改文件系统的指令里，RUN 是最典型的一类。每一个 RUN 的行为，都可以类比为我们刚才手工建立镜像的过程：先基于当前结果启动一个临时构建环境，在其上执行这些命令，再把这一步产生的文件系统变化保存为新的结果层。

> **注意**
>
> 每一个 RUN 指令都会产生一个新的镜像层。为了减少镜像体积和层数，我们通常会将多个命令合并到一个 RUN 指令中执行。
>
> 更多关于 RUN 指令的详细用法、最佳实践 (如清理缓存、使用 pipefail 等) 及 Union FS 的层数限制等内容，请参阅 **第七章 Dockerfile 指令详解** 中的 **RUN 指令** 小节。

要想编写优秀的 Dockerfile，必须了解每一条指令的作用和副作用。在 **第七章 Dockerfile 指令详解** 中，我们将对 COPY，ADD，CMD，ENTRYPOINT 等指令进行详细讲解。

4.5.5 构建镜像

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

在 Dockerfile 文件所在目录执行：

```
$ docker build -t nginx:v3 .
```

在当前版本的 Docker 中，docker build 默认会通过 Buildx 调用 BuildKit，因此你更常看到的是 [+] Building ... 这类输出。为了帮助理解“每一步如何形成镜像历史”，下面仍展示一种较容易阅读的经典输出形式：

```
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM nginx
 ---> e43d811ce2f4
Step 2 : RUN echo '<h1>Hello, Docker!</h1>' > /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，我们可以清晰地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/->
```

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

4.5.6 镜像构建上下文

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

首先要理解 `docker build` 的工作原理。今天的 `docker build` 默认会通过 Buildx 向 BuildKit 后端发起构建请求；无论后端运行在本机还是远端，位置参数指定的都是 **构建上下文**，也就是构建器可以访问到的文件集合。

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，经常还需要把本地文件复制进镜像，比如通过 `COPY` 指令、`ADD` 指令等。因此，构建器必须能够访问这些文件，而它能访问的范围正是你传给 `docker build` 的那个上下文。

如果上下文是本地目录，那么这个目录中的文件和子目录就会成为可用输入；如果上下文是远端 Git 仓库或 tar 包，那么构建器会直接获取对应内容。对于本地目录，BuildKit 会按需读取构建过程中真正需要的文件，而不是让 Dockerfile 任意访问宿主机上的任意路径。

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路径都应该以构建上下文为基准来理解。对于 legacy builder，像 `COPY ../package.json /app` 这样的写法会直接报错；而在 BuildKit 下，前导的越界 `../` 会被剥离并重新解释为上下文内路径。无论是哪种情况，构建器都无法读取上下文之外的宿主机文件；如果真的需要那些文件，应该先把它们放进上下文目录，或重新选择合适的上下文。

现在就可以理解刚才的命令 `docker build -t nginx:v3 .` 中的这个 `.`，实际上是在指定上下文目录，而不是单纯指定 `Dockerfile` 所在目录。

如果观察 `docker build` 的经典输出，或 BuildKit 输出中的 `transferring context` 提示，我们其实都能看到上下文传输的过程：

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

理解构建上下文对于镜像构建是很重要的，避免犯一些不应该的错误。比如有些初学者在发现需要的文件不在上下文里后，干脆把上下文切到硬盘根目录去构建。这样做即使在 BuildKit 下也会让可见上下文变得过大，并且在使用 `COPY . .`、`ADD . /app` 之类写法时，仍可能触发大规模上下文传输，导致构建缓慢甚至失败。这显然是使用错误。

一般来说，应该会将 `Dockerfile` 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 `.gitignore` 一样的语法写一个 `.dockerignore`，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

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

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

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

Built with mdPress

4.5.7 其它 `docker build` 的用法

直接用 Git repo 进行构建

或许你已经注意到了，`docker build` 还支持从 URL 构建，也就是直接把远端 Git 仓库作为上下文。传统写法可以使用 URL 片段 `#ref:dir`，例如：

```
$ docker build https://github.com/user/myrepo.git#mybranch:docker
```

这行命令表示：把 Git 仓库作为构建上下文，使用 `mybranch` 分支中的 `docker/` 子目录来构建。在较新的 Buildx 中，也可以改用结构更清晰的查询参数写法，例如 `?branch=mybranch&subdir=docker`。

用给定的 tar 压缩包构建

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

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

从标准输入中读取 Dockerfile 进行构建

```
docker build - < Dockerfile
```

或

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

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

从标准输入中读取上下文压缩包进行构建

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

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

4.6 其它制作镜像的方式

除了标准的使用 Dockerfile 生成镜像的方法外，由于各种特殊需求和历史原因，还提供了一些其它方法用以生成镜像。

4.6.1 从 rootfs 压缩包导入

格式：`docker import [选项] <文件>|<URL>|- [<仓库名>[:<标签>]]`

压缩包可以是本地文件、远程 Web 文件，甚至是从标准输入中得到。压缩包将会在镜像 / 目录展开，并直接作为镜像第一层提交。

比如我们想要创建一个 OpenVZ 的 Ubuntu 16.04 模板的镜像：

> **版本提示**：noble 对应 Ubuntu 24.04 LTS。实际用于生产环境时，应选择仍在安全维护期内的发行版，并按团队的基础镜像更新策略定期重建。

```
$ docker import \
    http://download.openvz.org/template/precreated/ubuntu-16.04-x86_64.tar.gz \
    openvz/ubuntu:16.04

Downloading from http://download.openvz.org/template/precreated/ubuntu-16.04-x86_64.tar.gz
sha256:412b8fc3e3f786dca0197834a698932b9c51b69bd8cf49e100c35d38c9879213
```

这条命令自动下载了 ubuntu-16.04-x86_64.tar.gz 文件，并且作为根文件系统展开导入，并保存为镜像 openvz/ubuntu:16.04。

导入成功后，我们可以用 docker image ls 看到这个导入的镜像：

```
$ docker image ls openvz/ubuntu
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
openvz/ubuntu       16.04               412b8fc3e3f7        55 seconds ago      505MB
```

如果我们查看其历史的话，会看到描述中有导入的文件链接：

```
$ docker history openvz/ubuntu:16.04
IMAGE               CREATED              CREATED BY          SIZE                COMMENT
f477a6e18e98        About a minute ago                       214.9 MB            Imported from htt
p://download.openvz.org/template/precreated/ubuntu-16.04-x86_64.tar.gz
```

4.6.2 Docker 镜像的导入和导出 docker save 和 docker load

Docker 还提供了 `docker save` 和 `docker load` 命令，用以将镜像保存为一个文件，然后传输到另一个位置上，再加载进来。这是在没有 Docker Registry 时的做法，现在已经不推荐，镜像迁移应该直接使用 Docker Registry，无论是直接使用 Docker Hub 还是使用内网私有 Registry 都可以。

保存镜像

使用 `docker save` 命令可以将镜像保存为归档文件。

比如我们希望保存这个 `alpine` 镜像。

```
$ docker image ls alpine
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              baa5d63471ea        5 weeks ago         4.803 MB
```

> **版本提示：** `alpine:latest` 为最新版本的 Alpine Linux。如果需要特定版本号（如 `alpine:3.20`），可以明确指定以确保可重现性。保存镜像的命令为：

```
$ docker save alpine -o filename
$ file filename
filename: POSIX tar archive
```

这里的 filename 可以为任意名称甚至任意后缀名，但文件的本质都是归档文件

注意：如果同名则会覆盖 (没有警告)

若使用 `gzip` 压缩：

```
$ docker save alpine | gzip > alpine-latest.tar.gz
```

然后我们将 `alpine-latest.tar.gz` 文件复制到了到了另一个机器上，可以用下面这个命令加载镜像：

```
$ docker load -i alpine-latest.tar.gz
Loaded image: alpine:latest
```

如果我们结合这两个命令以及 `ssh` 甚至 `pv` 的话，利用 Linux 强大的管道，我们可以写一个命令完成从一个机器将镜像迁移到另一个机器，并且带进度条的功能：

```
docker save <镜像名> | bzip2 | pv | ssh <用户名>@<主机名> 'cat | docker load'
```

4.7 实现原理

Docker 镜像是怎么实现增量的修改和维护的？为什么容器启动如此之快？这一切都归功于 Docker 的镜像分层存储设计。

4.7.1 镜像与分层存储

在之前的章节中，我们一直强调镜像包含操作系统完整的 `root` 文件系统，其体积往往是庞大的。因此在 Docker 设计时，就充分利用 **Union FS** 的技术，将其设计为分层存储的架构。

Docker 镜像并不是一个单纯的文件，而是由一组文件系统叠加构成的。

最底层的镜像称为 **基础镜像 (Base Image)**，通常是各种 Linux 发行版的 root 文件系统，如 Ubuntu、Debian、CentOS 等。

当我们在基础镜像之上构建新的镜像时 (例如安装了 Nginx)，Docker 并不是复制一份基础镜像，而是在基础镜像之上，**新建一个层 (Layer)**，并在该层中仅记录为了安装 Nginx 而发生的文件变更 (添加、修改、删除)。

这种分层存储结构使得镜像的复用、分发变得非常高效：

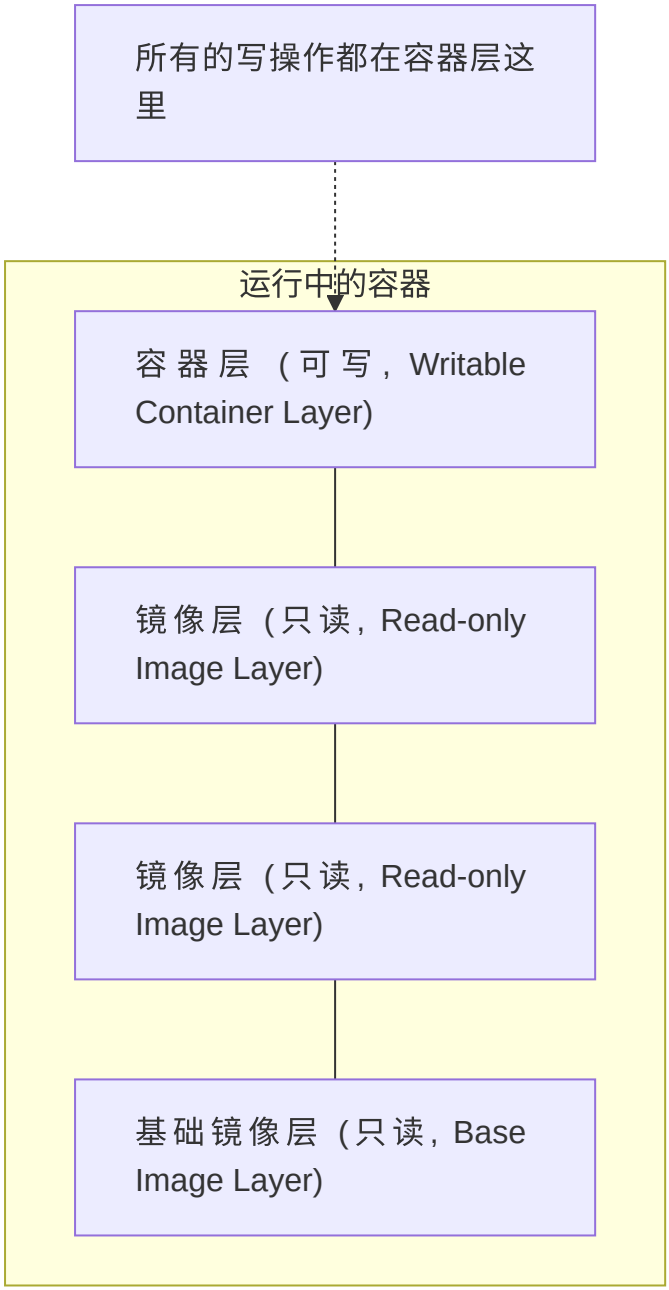
- **复用**：如果多个镜像都基于同一个基础镜像 (例如都基于 `ubuntu:24.04`)，那么宿主机只需要下载一份 `ubuntu:24.04`，所有镜像都可以共享它。
- **轻量分发**：镜像可以复用已有层，只传输和存储新增差异层；不过镜像是否足够小，仍然取决于基础镜像和新增内容本身。

4.7.2 容器层与读写

我们要理解的一个关键概念是：**镜像的每一层都是只读的 (Read-only)**。

那么，既然镜像只读，容器为什么能写文件呢？

当容器启动时，Docker 会在镜像的最上层，添加一个新的 **可写层 (Writable Layer)**，通常被称为 **容器层**。



- **读取文件**：当容器需要读取文件时，Docker 会从最上层 (容器层) 开始向下层 (镜像层) 寻找，直到找到该文件为止。
- **修改文件**：当容器需要修改某个文件时，Docker 会从下层镜像中将该文件复制到上层的容器层，然后对副本进行修改。这被称为 **写时复制 (Copy-on-Write，CoW)** 策略。
- **删除文件**：当容器删除某个文件时，Docker 并不是真的去下层删除它 (因为下层是只读的)，而是在容器层创建一个特殊的 “白障 (Whiteout)” 文件，用来标记该文件已被删除，从而在容器视图中隐藏它。

这就是为什么：

Built with mdPress

1. **容器删除后数据会丢失**：因为所有的数据修改都保存在最上层的容器层中，容器销毁时，这个层也就随之销毁了。(除非使用了数据卷，详见数据管理)。
2. **镜像不可变**：无论我们在容器里删除了多少文件，基础镜像的体积并不会减小，因为它们依然存在于底层的只读层中。

4.7.3 内容寻址与镜像 ID

Docker 镜像的每一层都有一个唯一的 ID，这个 ID 是根据该层的内容计算出来的哈希值 (SHA256)。这意味着：

- **内容即 ID**：只要层的内容有一丁点变化，ID 就会变。
- **安全性**：确保了镜像内容的完整性，下载过程中如果数据损坏，ID 校验就会失败。
- **去重**：如果两个不同的镜像 (甚至是不同来源的镜像) 包含相同的层 (ID 相同)，Docker 引擎在本地只会存储一份，绝不重复下载。

4.7.4 联合文件系统

Docker 使用联合文件系统 (Union FS) 与写时复制思路来实现这种分层挂载。传统的实现方式常见于 `overlay2`、`aufs`、`btrfs`、`zfs` 等存储驱动；而在 Docker Engine 29.0 及之后的全新安装中，默认镜像后端已经变为 containerd image store，它使用 snapshotter 来管理这些层。

> **版本背景**：Docker Engine 29.0.0 发布于 2025 年 11 月 10 日，是一个重要版本分界点。Docker Engine 29.0 及之后的全新安装默认使用 containerd image store；从更早版本升级的 daemon 会继续使用 legacy graph driver，直到显式启用 containerd image store。Docker Desktop 4.34 及之后也默认启用 containerd image store，实际环境仍应以当前配置为准。

虽然底层实现细节不同，但它们都遵循上述的 **分层 + CoW** 模型；因此，无论你看到的是 `overlay2` 还是 containerd snapshotter，理解镜像层、容器层和写时复制的方式都是一样重要的。

> 想要深入了解 Overlay2 等文件系统的具体实现原理，包括 WorkDir、UpperDir、LowerDir 等底层细节，请阅读 **第十二章 底层实现** 中的 **联合文件系统** 章节。

本章小结

本章介绍了 Docker 镜像的获取、列出、删除以及构建方式。

操作	命令
拉取镜像	`docker pull 镜像名:标签`
拉取所有标签	`docker pull -a 镜像名`
指定平台	`docker pull --platform linux/amd64 镜像名`
用摘要拉取	`docker pull 镜像名@sha256:...`
列出所有镜像	`docker images`
按仓库名过滤	`docker images nginx`
列出虚悬镜像	`docker images -f dangling=true`
只输出 ID	`docker images -q`
显示摘要	`docker images --digests`
自定义格式	`docker images --format "..."`
查看空间占用	`docker system df`
删除指定镜像	`docker rmi 镜像名:标签`
强制删除	`docker rmi -f 镜像名`
删除虚悬镜像	`docker image prune`
删除未使用镜像	`docker image prune -a`
批量删除	`docker rmi $(docker images -q -f ...)`

Built with mdPress

延伸阅读

- 获取镜像：从 Registry 拉取镜像
- 列出镜像：查看和过滤镜像
- 删除镜像：清理本地镜像
- 镜像加速器：加速镜像下载
- Docker Hub：官方镜像仓库
- 镜像：理解镜像概念
- 删除容器：清理容器
- 数据卷：清理数据卷

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

Built with mdPress

第五章 操作容器

容器是 Docker 又一核心概念。

简单的说，容器是独立运行的一个或一组应用，以及它们的运行态环境。对应的，虚拟机可以理解为模拟运行的一整套操作系统 (提供了运行态环境和其他系统环境) 和跑在上面的应用。

本章将具体介绍如何来管理一个容器，包括创建、启动和停止等。

版本号说明

本章示例涉及多个 Docker 镜像，遵循以下版本号最佳实践：

- **官方镜像**（如 `ubuntu`、`nginx`、`mysql`）：使用具体大版本号（如 `ubuntu:24.04`、`mysql:8.4`）而非 `latest`，确保示例的可重复性
- **镜像标签约定**：
  - `latest` 或 `v1.0.0` 等：带标签的自定义镜像，示例中指定具体版本
  - `24.04`、`8.4`：官方镜像的稳定版本分支
  - 生产环境建议：指定确切版本号（如 `nginx:1.28.0`、`mysql:8.4.4`）而非仅大版本号

- 启动容器
- 守护态运行
- 终止容器
- 进入容器
- 导出和导入容器
- 删除容器

Built with mdPress

5.1 启动

本节将详细介绍 Docker 容器的启动方式，包括新建启动和重新启动已停止的容器。

5.1.1 启动方式概述

启动容器有两种方式：

- **新建并启动**：基于镜像创建新容器
- **重新启动**：将已终止的容器重新运行

由于 Docker 容器非常轻量，实际使用中常常是随时删除和新建容器，而不是反复重启同一个容器。

5.1.2 新建并启动

基本语法

```
docker run [选项] 镜像 [命令] [参数...]
```

最简单的例子

输出 “Hello World” 后容器自动终止：

```
$ docker run ubuntu:24.04 /bin/echo 'Hello world'
Hello world
```

这与直接执行 `/bin/echo 'Hello world'` 几乎没有区别，但实际上已经启动了一个完整的 Ubuntu 容器来执行这条命令。

> **版本说明**：示例使用 `ubuntu:24.04`，这是最新 LTS 版本。如需其他版本，可替换为 `ubuntu:22.04`、`ubuntu:20.04` 等。

交互式容器

启动一个可以交互的 bash 终端：

```
$ docker run -it ubuntu:24.04 /bin/bash
root@af8bae53bdd3:/#
```

参数说明：

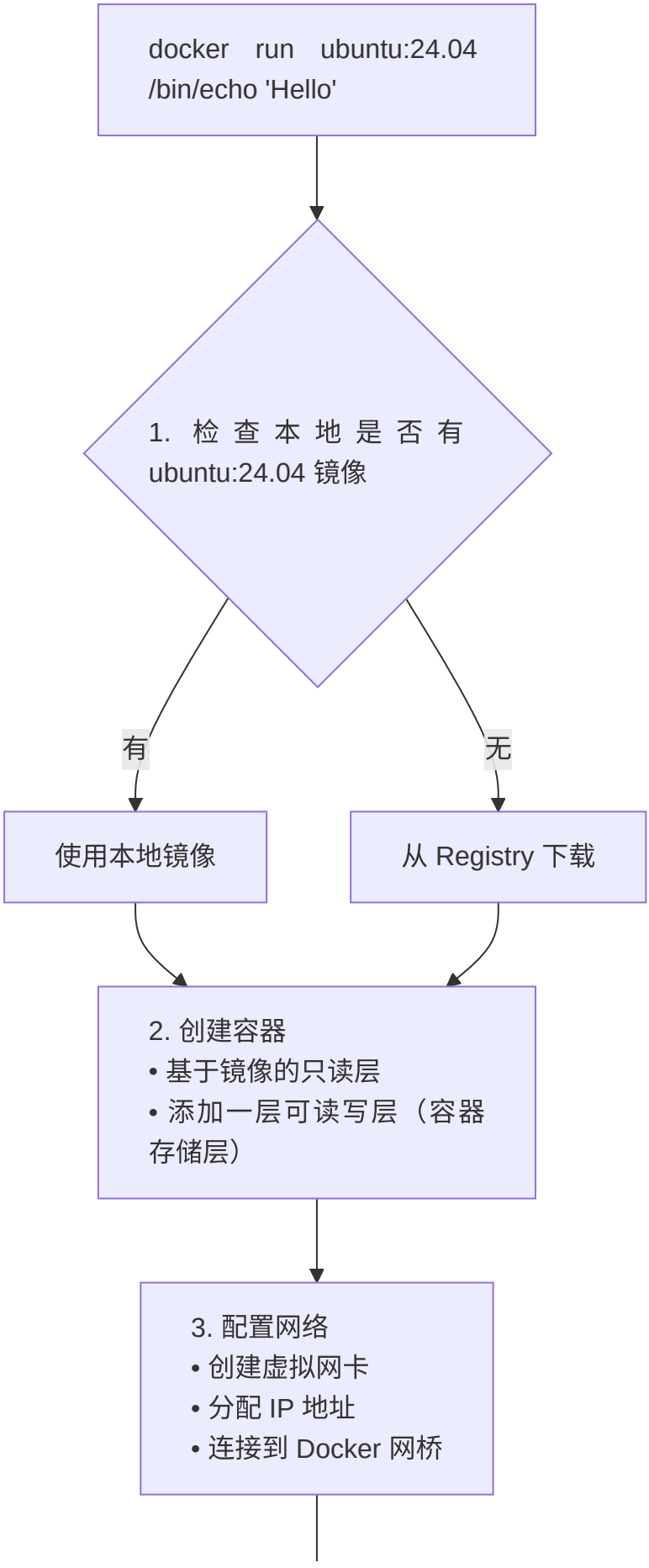
参数	作用
`-i`	保持标准输入 (stdin) 打开，允许输入
`-t`	分配伪终端 (pseudo-TTY)，提供终端界面
`-it`	两者组合使用，获得交互式终端

在交互模式下可以执行命令：

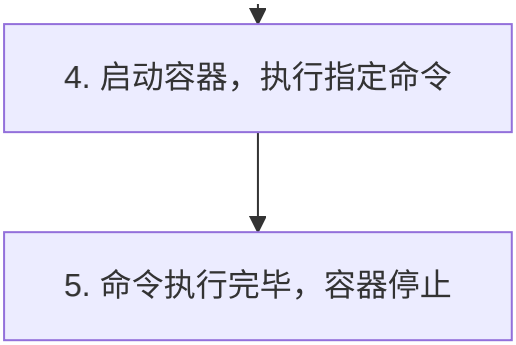
```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@af8bae53bdd3:/# exit  # 退出容器
```

5.1.3 docker run 的完整流程

执行 `docker run` 时，Docker 在后台完成以下操作：



Built with mdPress



5.1.4 常用启动选项

基础选项

选项	说明	示例
`-d`	后台运行 (detach)	`docker run -d nginx:latest`
`-it`	交互式终端	`docker run -it ubuntu:24.04 bash`
`--name`	指定容器名称	`docker run --name myapp nginx:latest`
`--rm`	退出后自动删除容器	`docker run --rm ubuntu:24.04 echo hi`

端口映射

```
## 将容器的 80 端口映射到宿主机的 8080 端口

$ docker run -d -p 8080:80 nginx:latest

## 随机映射端口

$ docker run -d -P nginx:latest

## 只绑定到 localhost

$ docker run -d -p 127.0.0.1:8080:80 nginx:latest
```

数据卷挂载

```
## 挂载命名卷

$ docker run -v mydata:/data nginx:latest

## 挂载宿主机目录

$ docker run -v /host/path:/container/path nginx:latest

## 只读挂载

$ docker run -v /host/path:/container/path:ro nginx:latest
```

环境变量

```
## 设置单个环境变量

$ docker run -e MYSQL_ROOT_PASSWORD=secret mysql

## 从文件加载环境变量

$ docker run --env-file .env myapp
```

资源限制

```
## 限制内存

$ docker run -m 512m nginx:latest

## 限制 CPU

$ docker run --cpus=1.5 nginx:latest
```

5.1.5 启动已终止容器

使用 `docker start` 重新启动已停止的容器：

```
## 查看所有容器（包括已停止的）

$ docker ps -a
CONTAINER ID  IMAGE   STATUS                     NAMES
af8bae53bdd3  ubuntu  Exited (0) 2 minutes ago   myubuntu

## 重新启动

$ docker start myubuntu

## 启动并附加终端

$ docker start -ai myubuntu
```

5.1.6 容器内进程的特点

容器内只运行指定的应用程序及其必需资源：

```
root@ba267838cc1b:/# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   11 ?        00:00:00 ps
```

可见容器中仅运行了 bash 进程。这种特点使得 Docker 对资源的利用率极高。

> 💡 笔者提示：容器内的 PID 1 进程很重要——它是容器的主进程，该进程退出则容器停止。详见后台运行章节。

5.1.7 常见问题

Q：容器启动后立即退出

原因：主进程执行完毕或无法保持运行

```
## 这个容器会立即退出（echo 执行完就结束了）

$ docker run ubuntu:24.04 echo "hello"

## 解决：使用能持续运行的命令

$ docker run -d nginx:latest  # nginx 是持续运行的服务
```

详细解释见后台运行。

Q：无法连接容器内的服务

原因：未正确映射端口

```
## 错误：没有 -p 参数，外部无法访问

$ docker run -d nginx:latest

## 正确：映射端口

$ docker run -d -p 80:80 nginx:latest
```

Q：容器内修改的文件丢失

原因：未使用数据卷，数据保存在容器存储层

```
## 使用数据卷持久化

$ docker run -v mydata:/app/data myapp
```

详见数据管理。

5.2 守护态运行

在生产环境中，我们通常需要容器持续运行，不受终端关闭的影响。本节将深入讲解如何让容器在后台运行，以及理解容器生命周期的核心概念。

5.2.1 核心概念：前台 vs 后台

当你在终端运行一个程序时，有两种模式：

- **前台运行**：程序占用当前终端，输出直接显示，关闭终端程序就停止
- **后台运行**：程序在后台执行，不占用终端，终端关闭也不影响程序

Docker 容器默认是 **前台运行** 的。使用 `-d` (detach) 参数可以让容器在后台运行。

5.2.2 基本使用

前台运行：默认

```
$ docker run ubuntu:24.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
hello world
hello world
hello world
hello world
```

容器会把输出的结果 (STDOUT) 打印到宿主机上面。此时：

- 终端被占用，无法执行其他命令
- 按 `Ctrl+C` 会终止容器
- 关闭终端窗口，容器也会停止

后台运行：使用 -d 参数

```
$ docker run -d ubuntu:24.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
77b2dc01fe0f3f1265df143181e7b9af5e05279a884f4776ee75350ea9d8017a
```

使用 `-d` 参数后：

- 容器在后台运行
- 返回容器的完整 ID
- 终端立即释放，可以继续执行其他命令
- 输出不会直接显示 (需要用 `docker logs` 查看)

5.2.3 深入理解：容器为什么会 “立即退出”？

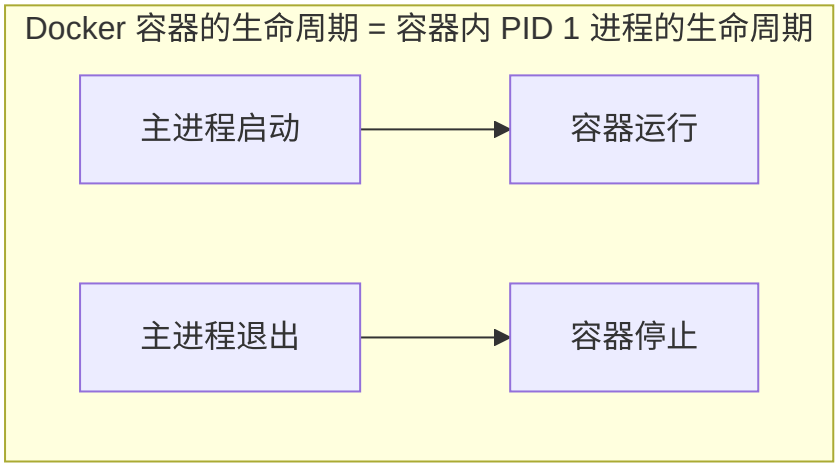
> **这是初学者最常遇到的困惑。** 理解这个问题，你就理解了 Docker 的核心设计理念。

很多人尝试这样启动容器：

```
$ docker run -d ubuntu:24.04
```

然后用 `docker ps` 查看，发现容器根本不在运行！这是为什么？

核心原理：容器的生命周期与主进程绑定



当你运行 `docker run -d ubuntu:24.04` 时：

1. 容器启动
2. 没有指定命令，默认执行 `/bin/bash`
3. 但没有交互式终端 (没有 `-it` 参数)，bash 发现没有输入源
4. bash 立即退出
5. 主进程退出，容器停止

关键理解：

- ❌ `-d` 参数 **不是** 让容器 “一直运行”
- ✅ `-d` 参数是让容器 “在后台运行”，能运行多久取决于主进程

常见的 “立即退出” 场景

场景	原因	解决方案
`docker run -d ubuntu`	默认 bash 无输入立即退出	指定长期运行的命令
`docker run -d nginx` 后改了配置	配置错误导致 nginx 启动失败	查看 `docker logs`
自定义镜像容器启动即退	Dockerfile 的 CMD 执行完毕	确保 CMD 是前台进程

5.2.4 查看后台容器

查看运行中的容器

```
$ docker container ls
CONTAINER ID  IMAGE         COMMAND               CREATED        STATUS       PORTS NAMES
77b2dc01fe0f  ubuntu:24.04  /bin/sh -c 'while tr  2 minutes ago  Up 1 minute        agitated_wright
```

查看容器输出日志

```
$ docker container logs 77b2dc01fe0f
hello world
hello world
hello world
...
```

实时查看日志 (类似 `tail -f`)：

```
$ docker container logs -f 77b2dc01fe0f
```

查看已停止的容器

```
$ docker container ls -a
```

加上 `-a` 参数可以看到所有容器，包括已停止的。这对于调试 “容器启动即退出” 的问题非常有用。

5.2.5 最佳实践

1. 长期运行的服务使用 -d

```
## Web 服务器

$ docker run -d -p 80:80 nginx:latest

## 数据库

$ docker run -d -p 3306:3306 mysql:8.4

## 缓存服务

$ docker run -d -p 6379:6379 redis:latest
```

> **版本说明**：示例使用常见的标签如 `latest` 或稳定大版本号如 `mysql:8.4`。具体版本可根据需求调整，生产环境建议明确指定版本号（如 `mysql:8.4.4`）而非使用 `latest`。

2. 调试时先用前台模式

当容器启动有问题时，**去掉 `-d` 参数** 可以直接看到输出和错误：

```
## 有问题的容器，先前台运行看看发生了什么

$ docker run myimage:v1.0.0
```

3. 使用 --rm 自动清理

对于一次性任务，使用 `--rm` 参数让容器退出后自动删除：

```
$ docker run --rm ubuntu:24.04 echo "Hello, World!"
Hello, World!

## 容器执行完后自动删除

...
```

4. 配合日志查看

```
## 后台启动

$ docker run -d --name myapp myimage:v1.0.0

## 查看最近 100 行日志

$ docker logs --tail 100 myapp

## 实时跟踪日志

$ docker logs -f myapp

## 查看带时间戳的日志

$ docker logs -t myapp
```

5.2.6 常见问题排查

Q：容器启动后立即退出

1. **查看退出状态码：**

```
$ docker ps -a --filter "name=mycontainer"
# 查看 STATUS 列，如 “Exited (1)” 表示异常退出
```

2. **查看容器日志：**

```
$ docker logs mycontainer
```

3. **以交互模式调试：**

```
# /bin/sh 覆盖镜像原本的启动命令，避免容器再次崩溃退出
# 进入 shell 后可手动执行原启动命令，定位具体报错原因
$ docker run -it myimage:v1.0.0 /bin/sh
```

Q：容器在后台运行但无法访问服务

1. **检查端口映射：**

```
$ docker port mycontainer
```

2. **检查容器内服务状态：**

```
$ docker exec mycontainer ps aux
```

Q：如何让已经在后台运行的容器回到前台？

使用 `docker attach`：

```
$ docker attach mycontainer
```

> **注意**：`attach` 会连接到容器的主进程。如果主进程不是交互式的，你可能只能看到输出。使用 `Ctrl+P Ctrl+Q` 可以安全退出而不停止容器。

5.2.7 延伸阅读

- 进入容器：如何进入正在运行的容器执行命令
- 容器日志：生产环境的日志管理最佳实践
- HEALTHCHECK 健康检查：自动检测容器内服务是否正常
- Docker Compose：管理多个后台容器的更好方式

5.3 终止

本节将介绍如何终止一个运行中的容器，以及几种不同的终止方式及其区别。

5.3.1 终止方式概述

终止容器有三种方式：

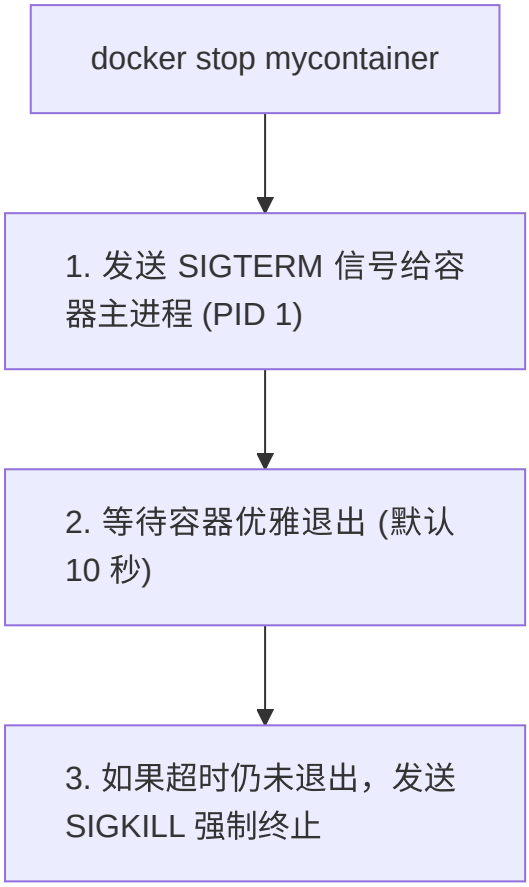
方式	命令	说明
优雅停止	`docker stop`	先发 SIGTERM，超时后发 SIGKILL
强制停止	`docker kill`	直接发 SIGKILL
自动终止	-	容器主进程退出时自动停止

5.3.2 docker stop：推荐

docker stop 基本用法

```
$ docker stop 容器名或ID
```

工作原理



自定义超时时间

```
## 等待 30 秒后强制终止

$ docker stop -t 30 mycontainer

## 立即发送 SIGKILL（相当于 docker kill）

$ docker stop -t 0 mycontainer
```

停止多个容器

```
## 停止多个指定容器

$ docker stop container1 container2 container3

## 停止所有运行中的容器

$ docker stop $(docker ps -q)
```

Built with mdPress

5.3.3 docker kill

基本用法

```
$ docker kill 容器名或ID
```

与 stop 的区别

命令	信号	使用场景
`docker stop`	SIGTERM → SIGKILL	正常停止，让应用优雅退出
`docker kill`	SIGKILL	应用无响应，强制终止

发送自定义信号

```
## 发送 SIGHUP（让进程重新加载配置）

$ docker kill -s HUP mycontainer

## 发送 SIGTERM

$ docker kill -s TERM mycontainer
```

5.3.4 容器自动终止

容器的生命周期与主进程绑定。主进程退出时，容器自动停止：

```
## 主进程是交互式 bash

$ docker run -it ubuntu bash
root@abc123:/# exit    # 退出 bash → 容器停止

## 主进程执行完毕

$ docker run ubuntu echo "Hello"    # echo 执行完 → 容器停止
```

5.3.5 查看已停止的容器

```
$ docker ps -a
CONTAINER ID   IMAGE    COMMAND       STATUS                      NAMES
ba267838cc1b   ubuntu   "/bin/bash"   Exited (0) 2 minutes ago    myubuntu
c5d3a5e8f7b2   nginx    "nginx"       Up 5 minutes                mynginx
```

STATUS 字段说明：

状态	说明
`Up X minutes`	运行中
`Exited (0)`	正常退出 (退出码 0)
`Exited (1)`	异常退出 (非零退出码)
`Exited (137)`	被 SIGKILL 终止 (128 + 9)
`Exited (143)`	被 SIGTERM 终止 (128 + 15)

5.3.6 重新启动容器

启动已停止的容器

```
$ docker start 容器名或ID

## 启动并附加终端

$ docker start -ai 容器名
```

重启运行中的容器

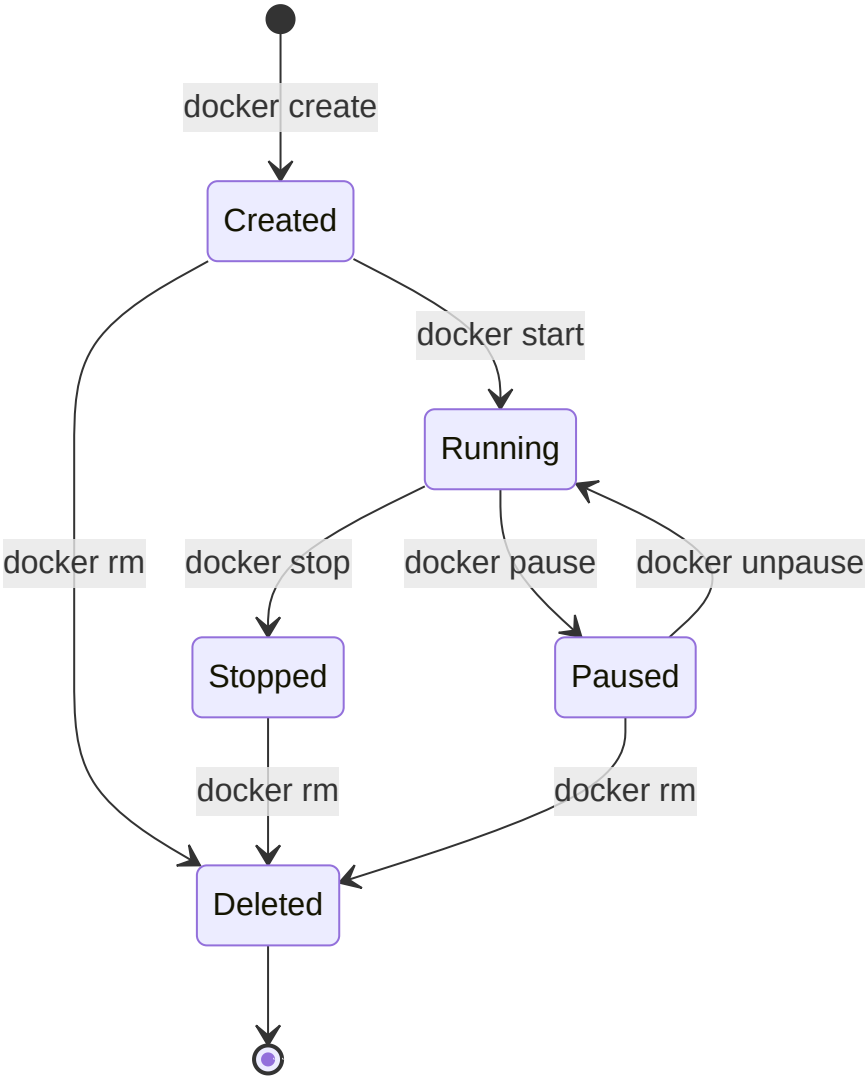
```
## 先停止再启动

$ docker restart 容器名

## 自定义停止超时

$ docker restart -t 30 容器名
```

5.3.7 生命周期状态图



5.3.8 批量操作

停止所有容器

```
$ docker stop $(docker ps -q)
```

删除所有已停止的容器

```
$ docker container prune
```

停止并删除所有容器

```
$ docker stop $(docker ps -q) && docker container prune -f
```

5.3.9 常见问题

Q：容器停止很慢

原因：应用没有正确处理 SIGTERM 信号，需要等待超时后强制终止。

解决方案：

1. 在应用中正确处理 SIGTERM
2. 使用 `docker stop -t 0` 立即终止
3. 检查 Dockerfile 中的 `STOPSIGNAL` 配置

Q：如何让容器优雅退出

确保容器主进程正确处理信号：

```
## Dockerfile 示例

FROM node:22-alpine
...

## 使用 exec 形式确保信号能传递给 node 进程

CMD ["node", "server.js"]
```

> **版本说明**：示例使用 `node:22-alpine`，这是一个精简的 Node.js 22 版本镜像。可根据需求替换为其他版本（如 `node:24-alpine`、`node:latest`）。

Q：容器无法停止

```
## 强制终止

$ docker kill 容器名

## 如果仍无法停止，检查系统资源

$ docker inspect 容器名
```

5.4 进入容器

5.4.1 为什么需要进入容器

使用 `-d` 参数启动容器后，容器在后台运行。以下场景需要进入容器内部操作：

场景	示例
调试问题	查看日志、检查配置、排查错误
临时操作	执行数据库迁移、清理缓存
检查状态	查看进程、网络连接、文件系统
开发测试	交互式测试命令、验证环境

5.4.2 两种进入方式

Docker 提供两种进入容器的命令：

命令	推荐程度	特点
`docker exec`	✅ **推荐**	启动新进程，退出不影响容器
`docker attach`	⚠️ 谨慎使用	附加到主进程，退出可能停止容器

5.4.3 docker exec：推荐

docker exec 基本用法

```
## 进入容器并启动交互式 shell

$ docker exec -it 容器名 /bin/bash

## 或使用 sh（适用于 Alpine 等精简镜像）

$ docker exec -it 容器名 /bin/sh
```

参数说明

参数	作用
`-i`	保持标准输入打开 (interactive)
`-t`	分配伪终端 (TTY)
`-it`	两者组合，获得完整交互体验
`-u`	指定用户 (如 `-u root`)
`-w`	指定工作目录
`-e`	设置环境变量

docker exec 示例

```
## 启动一个后台容器

$ docker run -dit --name myubuntu ubuntu
69d137adef7a...

## 进入容器（交互式 shell）

$ docker exec -it myubuntu bash
root@69d137adef7a:/# ls
bin  boot  dev  etc  home  lib  ...
root@69d137adef7a:/# exit

## 容器仍在运行!

$ docker ps
CONTAINER ID   IMAGE    STATUS         NAMES
69d137adef7a   ubuntu   Up 2 minutes   myubuntu
```

执行单条命令

不进入交互模式，直接执行命令：

```
## 查看容器内进程

$ docker exec myubuntu ps aux

## 查看配置文件

$ docker exec myubuntu cat /etc/nginx/nginx.conf

## 以 root 用户执行

$ docker exec -u root myubuntu apt update
```

只用 -i 不用 -t 的区别

```
## 只用 -i：可以执行命令，但没有提示符

$ docker exec -i myubuntu bash
ls            # 输入命令
bin           # 输出结果
boot
dev
...

## 用 -it：有完整的终端体验

$ docker exec -it myubuntu bash
root@69d137adef7a:/#    # 有提示符
```

> 💡 通常使用 `-it` 组合。只有在脚本中需要通过管道传入命令时才只用 `-i`。

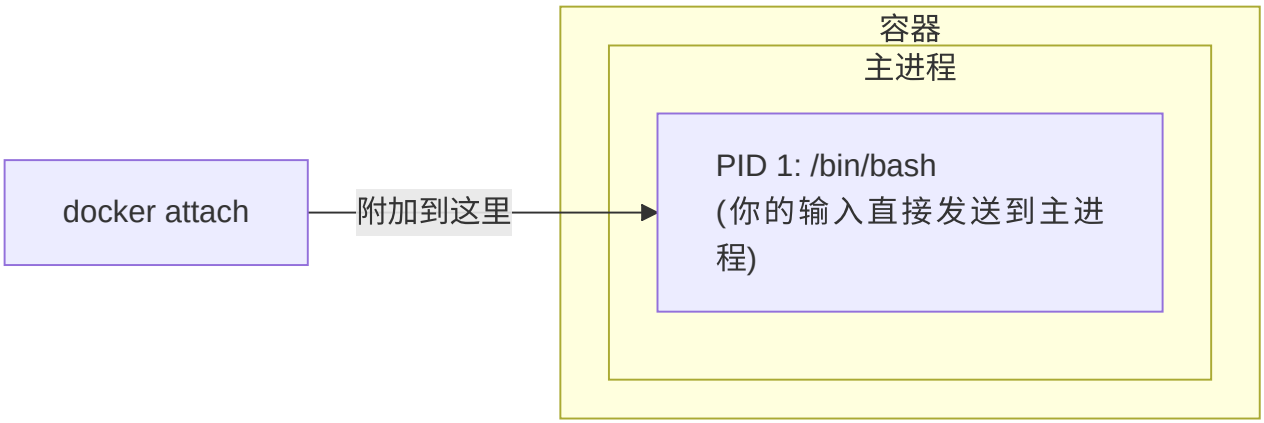
5.4.4 docker attach：谨慎使用

docker attach 基本用法

```
$ docker attach 容器名
```

工作原理

`attach` 会附加到容器的 **主进程** (PID 1) 的标准输入输出：



docker attach 示例

```
## 启动容器

$ docker run -dit --name myubuntu ubuntu
243c32535da7...

## 附加到容器

$ docker attach myubuntu
root@243c32535da7:/#
```

⚠️ 重要警告

从 attach 会话中输入 `exit` 或按 `Ctrl+D` 会导致容器停止！

```
$ docker attach myubuntu
root@243c32535da7:/# exit    # 这会停止容器!

$ docker ps
CONTAINER ID   IMAGE    STATUS                     NAMES
243c32535da7   ubuntu   Exited (0) 2 seconds ago   myubuntu
```

原因：attach 附加到主进程，退出主进程就等于退出容器。

安全退出 attach

使用 `Ctrl+P` 然后 `Ctrl+Q` 可以从 attach 会话中 **分离**，而不停止容器：

```
$ docker attach myubuntu
root@243c32535da7:/#

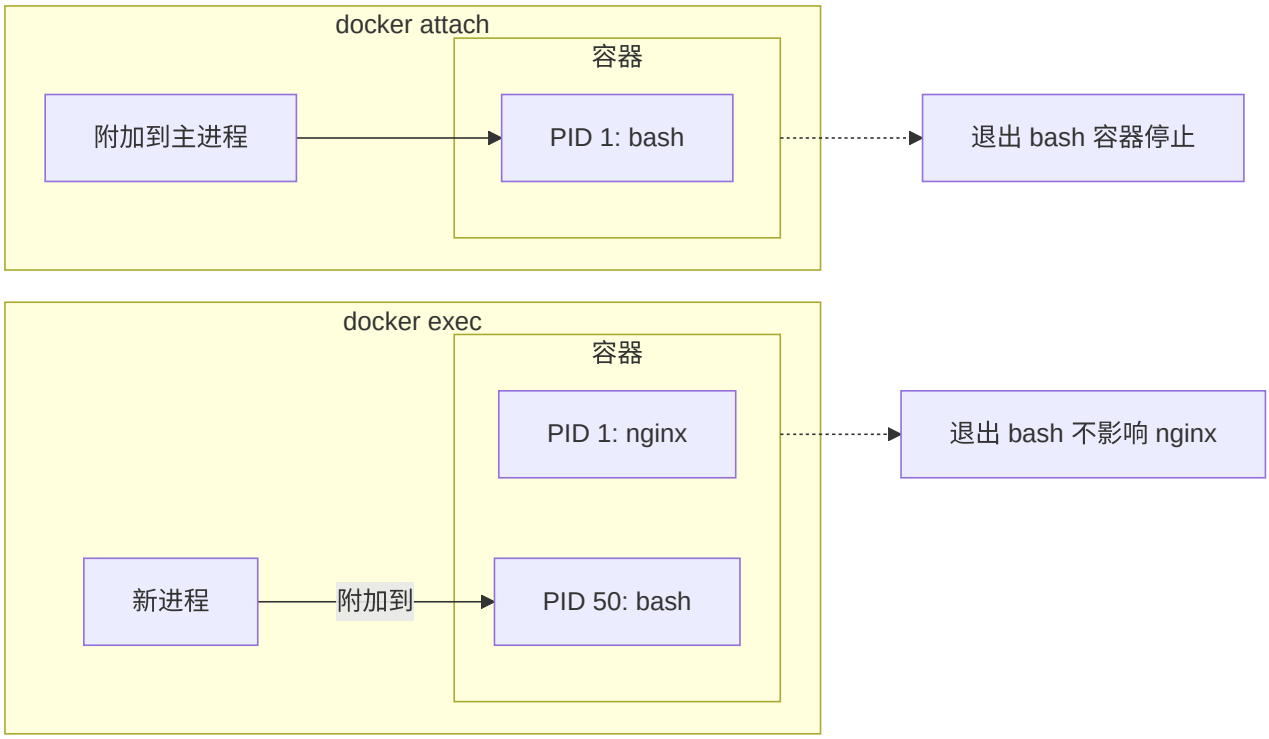
## 按 Ctrl+P 然后 Ctrl+Q

read escape sequence

$ docker ps    # 容器仍在运行
CONTAINER ID   IMAGE    STATUS         NAMES
243c32535da7   ubuntu   Up 5 minutes   myubuntu
```

5.4.5 exec vs attach 对比

特性	docker exec	docker attach
工作方式	在容器内启动新进程	附加到主进程
退出影响	不影响容器	可能停止容器
多终端	可以开多个	共享同一个会话
适用场景	调试、临时操作	查看主进程输出
推荐程度	✅ 推荐	⚠️ 特殊场景使用



5.4.6 最佳实践

1. 首选 docker exec

```
## 进入容器调试

$ docker exec -it myapp bash

## 查看日志

$ docker exec myapp tail -f /var/log/app.log

## 执行数据库迁移

$ docker exec myapp python manage.py migrate
```

2. 生产环境避免进入容器

笔者建议：生产环境应尽量避免进入容器直接操作，而是通过：

- 日志系统查看日志 (如 `docker logs` 或集中式日志)
- 监控系统查看状态
- 重新部署而非手动修改

3. 无 shell 镜像的处理

某些精简镜像 (如基于 `scratch` 或 `distroless`) 没有 shell：

```
## 这会失败

$ docker exec -it myapp bash
OCI runtime exec failed: exec failed: unable to start container process: exec: "bash": executable fi
le not found

## 解决方案：使用调试容器（需要 Docker Desktop Pro/Team/Business 订阅）

$ docker debug myapp
```

> **注意**：`docker debug` 是 Docker Desktop 4.33+ 提供的功能，需要 Pro、Team 或 Business 订阅。它会附加一个包含常用调试工具（vim、curl、htop 等）的工具箱到目标容器，即使目标镜像基于 `scratch` 也能使用。

5.4.7 常见问题

Q：exec 进入后看不到其他终端的操作

这是正常的。exec 启动的是独立进程，多个 exec 会话互不影响。

Q：容器没有 bash

尝试使用 sh：

```
$ docker exec -it myapp /bin/sh
```

Q：需要 root 权限

```
$ docker exec -u root -it myapp bash
```

5.5 导出和导入

当我们需要迁移容器或者备份容器时，可以使用 Docker 的导入和导出功能。本节将介绍这两个命令的使用方法。

5.5.1 导出容器

如果要导出本地某个容器，可以使用 `docker export` 命令。

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
7691a814370e        ubuntu:24.04        "/bin/bash"         36 hours ago        Exited (0) 21 hours
ago                       test
$ docker export 7691a814370e > ubuntu.tar
```

这样将导出容器快照到本地文件。

> **版本说明**：导出的容器快照不包含镜像版本历史，导入时可以自定义标签和版本号。

5.5.2 导入容器快照

可以使用 `docker import` 从容器快照文件中再导入为镜像，例如

```
$ cat ubuntu.tar | docker import - test/ubuntu:v1.0
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
test/ubuntu         v1.0                9d37a6082e97        About a minute ago   171.3 MB
```

此外，也可以通过指定 URL 或者某个目录来导入，例如

```
$ docker import http://example.com/exampleimage.tgz example/imagerepo
```

注：用户既可以使用 `docker load` 来导入镜像存储文件到本地镜像库，也可以使用 `docker import` 来导入一个容器快照到本地镜像库。这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息 (即仅保存容器当时的快照状态)，而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

5.6 删除

随着容器的创建和停止，系统中会积累大量的容器。本节将介绍如何删除不再需要的容器，以及如何清理所有停止的容器。

5.6.1 基本用法

使用 `docker rm` 删除已停止的容器：

```
$ docker rm 容器名或ID
```

> 💡 `docker rm` 是 `docker container rm` 的简写，两者等效。

5.6.2 删除选项

选项	说明	示例
无参数	删除已停止的容器	`docker rm mycontainer`
`-f`	强制删除运行中的容器	`docker rm -f mycontainer`
`-v`	同时删除关联的匿名卷	`docker rm -v mycontainer`

删除已停止的容器

```
$ docker rm mycontainer
mycontainer
```

强制删除运行中的容器

```
## 不加 -f 会报错

$ docker rm running_container
Error: cannot remove running container

## 加 -f 强制删除

$ docker rm -f running_container
running_container
```

> ⚠️ 强制删除会向容器发送 SIGKILL 信号，可能导致数据丢失。建议先 `docker stop` 优雅停止。

删除容器及其数据卷

```
## 删除容器时同时删除其匿名卷

$ docker rm -v mycontainer
```

> 注意：只删除匿名卷，命名卷不会被删除。

5.6.3 批量删除

删除所有已停止的容器

```
## 方式一：使用 prune 命令（推荐）

$ docker container prune

WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
abc123...
def456...
Total reclaimed space: 150MB

## 方式二：不提示确认

$ docker container prune -f
```

删除所有容器：包括运行中的

```
## 先停止所有容器，再删除

$ docker stop $(docker ps -q)
$ docker rm $(docker ps -aq)

## 或者直接强制删除

$ docker rm -f $(docker ps -aq)
```

按条件删除

```
## 删除所有已退出的容器

$ docker rm $(docker ps -aq -f status=exited)

## 删除名称包含 "test" 的容器

$ docker rm $(docker ps -aq -f name=test)

## 删除 24 小时前创建的容器

$ docker container prune --filter "until=24h"
```

5.6.4 常用过滤条件

`docker ps` 的过滤条件可以配合 `rm` 使用：

过滤条件	说明	示例
`status=exited`	已退出的容器	`-f status=exited`
`status=created`	已创建未启动	`-f status=created`
`name=xxx`	名称匹配	`-f name=myapp`
`ancestor=xxx`	基于某镜像创建	`-f ancestor=nginx`
`before=xxx`	在某容器之前创建	`-f before=mycontainer`
`since=xxx`	在某容器之后创建	`-f since=mycontainer`

示例

```
## 删除所有基于 nginx 镜像的容器

$ docker rm $(docker ps -aq -f ancestor=nginx)

## 删除所有创建后未启动的容器

$ docker rm $(docker ps -aq -f status=created)
```

5.6.5 容器与镜像的依赖关系

> 有容器依赖的镜像无法删除。

```
## 尝试删除有容器依赖的镜像

$ docker image rm nginx
Error: image is being used by stopped container abc123

## 需要先删除依赖该镜像的容器

$ docker rm abc123
$ docker image rm nginx
```

5.6.6 清理策略建议

开发环境

```
## 定期清理已停止的容器

$ docker container prune -f

## 一键清理所有未使用资源

$ docker system prune -f
```

生产环境

```
## 使用 --rm 参数运行临时容器

$ docker run --rm ubuntu echo "Hello"

## 容器退出后自动删除

## 定期清理（设置保留时间）

$ docker container prune --filter "until=168h"  # 保留 7 天内的
```

完整清理脚本

```
#!/bin/bash

## cleanup.sh - Docker 资源清理脚本

echo "清理已停止的容器..."
docker container prune -f

echo "清理未使用的镜像..."
docker image prune -f

echo "清理未使用的数据卷..."
docker volume prune -f

echo "清理未使用的网络..."
docker network prune -f

echo "清理完成！"
docker system df
```

5.6.7 常见问题

Q：容器无法删除

```
Error: container is running
```

解决：先停止容器，或使用 `-f` 强制删除

```
$ docker stop mycontainer
$ docker rm mycontainer

## 或

$ docker rm -f mycontainer
```

Q：删除后磁盘空间没释放

可能原因：

1. 容器的数据卷未删除 (使用 `-v` 参数)
2. 镜像未删除
3. 构建缓存未清理

解决：

```
## 查看空间占用

$ docker system df

## 完整清理

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

本章小结

本章介绍了 Docker 容器的启动、停止、进入和删除等核心操作。

操作	命令	说明
新建并运行	`docker run`	最常用的启动方式
交互式启动	`docker run -it`	用于调试或临时操作
后台运行	`docker run -d`	用于服务类应用
启动已停止的容器	`docker start`	重用已有容器
优雅停止	`docker stop`	先 SIGTERM，超时后 SIGKILL
强制停止	`docker kill`	直接 SIGKILL
重启	`docker restart`	停止后立即启动
停止全部	`docker stop $(docker ps -q)`	停止所有运行中容器
进入容器调试	`docker exec -it 容器名 bash`	推荐方式
执行单条命令	`docker exec 容器名 命令`	不进入交互模式
查看主进程输出	`docker attach 容器名`	慎用，退出可能停止容器
删除已停止容器	`docker rm 容器名`	需先停止
强制删除运行中容器	`docker rm -f 容器名`	直接删除
删除容器及匿名卷	`docker rm -v 容器名`	同时清理匿名卷
清理所有已停止容器	`docker container prune`	批量清理

Built with mdPress

延伸阅读

- 后台运行：理解 `-d` 参数和容器生命周期
- 进入容器：操作运行中的容器
- 网络配置：理解端口映射的原理
- 数据管理：数据持久化方案
- 删除镜像：清理镜像
- 数据卷：数据卷管理

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第六章 访问仓库

仓库 (Repository) 是集中存放镜像的地方。

一个容易混淆的概念是注册服务器 (Registry)。实际上注册服务器是管理仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。从这方面来说，仓库可以被认为是一个具体的项目或目录。例如对于仓库地址 `docker.io/ubuntu` 来说，`docker.io` 是注册服务器地址，`ubuntu` 是仓库名。

大部分时候，并不需要严格区分这两者的概念。

版本号说明

本章涉及的 Registry 和相关工具版本说明：

- **Docker Registry**：使用 `registry:2` 推荐版本，已停止维护的 `registry:1` 不建议使用
- **Nexus 3**：建议指定具体版本（如 `sonatype/nexus3:3.69`）而非 `latest`，避免自动升级带来的兼容性问题
- **镜像标签规范**：
  - 生产环境推送至仓库时应明确指定版本号（如 `myapp:v1.0.0`）
  - 避免依赖 `latest` 标签，因为其含义存在歧义且易导致版本混淆

为什么需要私有仓库?

在讨论具体的安装和配置前，让我们先理解：**什么时候你应该建设私有仓库?**

开发团队（有专利代码、不能公开）：

- 需要私有仓库存储内部镜像
- 涉及访问控制和审计
- 强烈推荐使用托管方案（如 Harbor 或云厂商提供的镜像仓库）

开源项目或个人学习：

- Docker Hub 公开仓库足够
- 无需自建私有仓库的成本

企业级部署：

- 需要高可用、备份、灾难恢复
- 推荐使用专业级方案（Nexus 3、Harbor）而非简单的 Registry

本章涵盖的方案从简到复杂：

- Docker Registry：最小化部署（适合简单场景）
- 私有仓库高级配置：添加认证、HTTPS 等生产必需项
- Nexus 3：企业级完整解决方案，支持权限管理、备份等

本章内容

- Docker Hub
- 私有仓库
- 私有仓库高级配置
- Nexus 3

Built with mdPress

6.1 Docker Hub

6.1.1 什么是 Docker Hub

Docker Hub 是 Docker 的中央镜像仓库，通过它您可以轻松地分享和获取 Docker 镜像。

Docker Hub 是 Docker 官方维护的公共镜像仓库，也是全球最大的容器镜像库。

它提供了：

- **官方镜像**：由 Docker 官方和软件厂商 (如 Nginx，MySQL，Node.js) 维护的高质量镜像。
- **个人/组织仓库**：用户可以上传自己的镜像。
- **自动构建**：与 GitHub/Bitbucket 集成 (需付费)。
- **Webhooks**：镜像更新时触发回调。

6.1.2 核心功能

1. 搜索镜像

我们可以通过 `docker search` 命令来查找官方仓库中的镜像，并利用 `docker pull` 命令来将它下载到本地。

除了网页搜索，也可以使用命令行：

```
$ docker search centos
NAME      DESCRIPTION                     STARS     OFFICIAL
centos    The official build of CentOS.   7000+     [OK]
```

> **技巧**：始终优先使用 `OFFICIAL` 标记为 `[OK]` 的镜像，安全性更有保障。

2. 拉取镜像

```
$ docker pull nginx:alpine
```

3. 推送镜像

需要先登录：

```
$ docker login

## 默认情况下，不带其它参数进行 docker login 会自动走 Device Code Web Flow (浏览器认证)
## 若在非交互 CI 环境中，推荐结合 --username 与 --password-stdin 参数使用

...
```

打标签并推送：

```
## 1. 标记镜像

$ docker tag myapp:v1 username/myapp:v1

## 2. 推送

$ docker push username/myapp:v1
```

6.1.3 限制与配额

镜像拉取限制

Docker Hub 对不同类型用户实施拉取速率限制（基于 6 小时周期）：

用户类型	限制
匿名用户 (未登录)	每 6 小时 100 次请求
免费账户 (已登录)	每 6 小时 200 次请求
Pro/Team/Business 账户	无限制（公平使用政策）

> **注意**：Docker 曾计划于 2025 年 4 月调整拉取限制策略，但在 2025 年 2 月宣布取消该计划。目前付费订阅用户享有无限制拉取额度，匿名用户和免费账户的限制保持不变。建议在 CI/CD 环境中始终配置 `docker login` 以获得更高的拉取额度。

滥用限流

除了上述针对特定账号拉取镜像数量的 Pull Rate Limit 之外，Docker Hub 对所有用户（包含已认证及付费用户）还实施了 **滥用保护限流 (Abuse Rate Limiting)**。它是根据网络出口 IP (IPv4 或 IPv6 /64 子网) 计算整体请求频率，阈值动态触发（通常为每分钟数千级别请求）。

两类的差异与排查方法：

- **Pull Rate Limit**：针对拉取量达到上限。报错返回 `429 Too Many Requests`，并且 HTTP 返回体/CLI 错误提示中会带有明确的 `toomanyrequests: You have reached your pull rate limit` 提示，常附有账户升级链接。
- **Abuse Rate Limit**：防范接口频率打击。报错仅返回简化的 `429 Too Many Requests`。这一限流不分付费与否，常发生在“多终端共享出口 IP”的企业局域网或者第三方云 CI 服务（如 GitHub Actions 等）中，即使你已正常配置 `docker login` 也依旧可能触发。

> **提示**：如果在 CI/CD 等环境遇到 429 错误，建议：
>
> 1. 先甄别具体是哪类限流：普通的 pull rate limit 只要在 CI 中配置 `docker login` (并使用有效账号) 就能解除匿名限制。
> 2. 如果是 Abuse 频控导致，应考虑搭建私有仓库作为拉取缓存代理 (Registry pull-through cache)，避免频繁直接请求官方 Hub。
> 3. 使用国内镜像加速器。

6.1.4 安全最佳实践

1. 启用 2FA：双因素认证

为了保护您的 Docker Hub 账号安全，我们建议采取以下措施。

在 Account Settings -> Security 中启用 2FA，保护账号安全。启用后，CLI 登录需要使用 **Access Token** 而非密码。

2. 使用 Access Token

> ⚠️ **警告**：绝不要在脚本或 CI/CD 系统中，直接使用 `-p` 参数传递密码或 Token (类似 `docker login -p xxx`)！这会导致凭证直接暴露在系统的命令历史、进程列表和终端输出中。

1. 在 Docker Hub -> Account Settings -> Security -> Access Tokens 创建 Token (PAT)。
2. 将 Token 保存在权限受限的本地文件或 CI secret 中，再通过标准输入 (stdin) 传递给 Docker，避免把真实 Token 写进脚本、命令历史或日志：

```
$ chmod 600 "$HOME/.dockerhub-token"
$ cat "$HOME/.dockerhub-token" | docker login --username username --password-stdin
```

3. 关注镜像漏洞

Docker Hub 提供 Docker Scout 安全扫描功能。官方镜像的漏洞扫描结果对所有用户免费可见。Docker Scout 的持续扫描功能在免费层可以覆盖 1 个私有仓库，付费用户可以扫描更多仓库。在镜像标签页可以看到漏洞扫描结果。

6.1.5 Webhooks

当镜像被推送时，可以自动触发 HTTP 回调 (例如通知 CI 系统部署)。

配置方法： 仓库页面 -> Webhooks -> Create Webhook。

6.1.6 自动构建

> ⚠️ 目前仅限付费用户 (Pro/Team) 使用。

链接 GitHub/Bitbucket 仓库后，当代码有提交或打标签时，Docker Hub 会自动运行构建。这保证了镜像总是与代码同步，且由可信的官方环境构建。

6.2 私有仓库

有时候使用 Docker Hub 这样的公共仓库可能不方便，用户可以创建一个本地仓库供私人使用。

本节介绍如何使用本地仓库。

Docker Registry 是官方提供的工具，可以用于构建私有的镜像仓库。本文内容基于 distribution/distribution v2.x 版本。

6.2.1 安装运行 docker-registry

容器运行

如果您需要搭建私有仓库，可以通过官方提供的 `registry` 镜像快速部署。

你可以使用官方 `registry` 镜像来运行。

```
$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

> **版本说明**：使用 `registry:2` 表示 Docker Registry 2.x 版本，这是当前推荐的版本。旧版本 Registry 1.x 已停止维护，不建议使用。 这将使用官方的 `registry` 镜像来启动私有仓库。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
```

6.2.2 在私有仓库上传、搜索、下载镜像

创建好私有仓库之后，就可以使用 `docker tag` 来标记一个镜像，然后推送它到仓库。例如私有仓库地址为 `127.0.0.1:5000`。

先在本机查看已有的镜像。

```
$ docker image ls
REPOSITORY                          TAG                 IMAGE ID            CREATED             VIRTUA
L 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`。

格式为 `docker tag IMAGE[:TAG] [REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]`。

```
$ docker tag ubuntu:latest 127.0.0.1:5000/ubuntu:latest
$ docker image ls
REPOSITORY                          TAG                 IMAGE ID            CREATED             VIRTUA
L SIZE
ubuntu                              latest              ba5877dc9bec        6 weeks ago         192.7
MB
127.0.0.1:5000/ubuntu:latest        latest              ba5877dc9bec        6 weeks ago         192.7
MB
```

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
```

用 `curl` 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             VIRTU
AL SIZE
127.0.0.1:5000/ubuntu:latest        latest              ba5877dc9bec        6 weeks ago         192.7
MB
```

6.2.3 配置非 https 仓库地址

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，或者查看下一节配置能够通过 `HTTPS` 访问的私有仓库。

Linux

默认情况下，Docker 强制使用 HTTPS 协议推送镜像。如果您搭建的私有仓库是 HTTP 协议，需要进行如下配置。

对于使用 `systemd` 的系统，请在 `/etc/docker/daemon.json` 中写入如下内容 (如果文件不存在请新建该文件)

```
{
  "registry-mirrors": [
    "https://docker.your-mirror.example.com"
  ],
  "insecure-registries": [
    "192.168.199.100:5000"
  ]
}
```

> 注意：该文件必须符合 `json` 规范，否则 Docker 将不能启动。镜像加速器地址请替换为实际可用的源，具体配置参见 3.9 镜像加速器。

6.2.4 其他

对于 Docker Desktop for Windows、Docker Desktop for Mac 在设置中的 `Docker Engine` 中进行编辑，增加和上边一样的字符串即可。

6.3 私有仓库高级配置

上一节我们搭建了一个具有基础功能的私有仓库，本小节我们来使用 Docker Compose 搭建一个拥有权限认证、TLS 的私有仓库。

新建一个文件夹，以下步骤均在该文件夹中进行。

6.3.1 准备站点证书

如果你拥有一个域名，国内各大云服务商均提供免费的站点证书。你也可以使用 openssl 自行签发证书。

这里假设我们将要搭建的私有仓库地址为 docker.domain.com，下面我们介绍使用 openssl 自行签发 docker.domain.com 的站点 SSL 证书。

第一步创建 CA 私钥。

```
$ openssl genrsa -out "root-ca.key" 4096
```

第二步利用私钥创建 CA 根证书请求文件。

```
$ openssl req \
          -new -key "root-ca.key" \
          -out "root-ca.csr" -sha256 \
          -subj '/C=CN/ST=Shanxi/L=Datong/O=Your Company Name/CN=Your Company Name Docker Registry C
A'
```

> 以上命令中 -subj 参数里的 /C 表示国家，如 CN；/ST 表示省；/L 表示城市或者地区；/O 表示组织名；/CN 通用名称。

第三步配置 CA 根证书，新建 root-ca.cnf。

```
[root_ca]
basicConstraints = critical,CA:TRUE,pathlen:1
keyUsage = critical, nonRepudiation, cRLSign, keyCertSign
subjectKeyIdentifier=hash
```

第四步签发根证书。

```
$ openssl x509 -req  -days 3650  -in "root-ca.csr" \
               -signkey "root-ca.key" -sha256 -out "root-ca.crt" \
               -extfile "root-ca.cnf" -extensions \
               root_ca
```

第五步生成站点 SSL 私钥。

```
$ openssl genrsa -out "docker.domain.com.key" 4096
```

第六步使用私钥生成证书请求文件。

```
$ openssl req -new -key "docker.domain.com.key" -out "site.csr" -sha256 \
          -subj '/C=CN/ST=Shanxi/L=Datong/O=Your Company Name/CN=docker.domain.com'
```

第七步配置证书，新建 `site.cnf` 文件。

```
[server]
authorityKeyIdentifier=keyid,issuer
basicConstraints = critical,CA:FALSE
extendedKeyUsage=serverAuth
keyUsage = critical, digitalSignature, keyEncipherment
subjectAltName = DNS:docker.domain.com, IP:127.0.0.1
subjectKeyIdentifier=hash
```

第八步签署站点 SSL 证书。

```
$ openssl x509 -req -days 750 -in "site.csr" -sha256 \
    -CA "root-ca.crt" -CAkey "root-ca.key"  -CAcreateserial \
    -out "docker.domain.com.crt" -extfile "site.cnf" -extensions server
```

配套 demo 中的 `ssl/README.md` 只保留本地生成证书的占位说明，真实私钥和证书不应提交到仓库。 这样已经拥有了 `docker.domain.com` 的网站 SSL 私钥 `docker.domain.com.key` 和 SSL 证书 `docker.domain.com.crt` 及 CA 根证书 `root-ca.crt`。

新建 `ssl` 文件夹并将 `docker.domain.com.key` `docker.domain.com.crt` `root-ca.crt` 这三个文件移入，删除其他文件。

> **安全提示**：这些私钥和证书应在本地或受控部署环境中生成，不要提交进 Git 仓库。示例目录只保留占位说明，运行前请按上面的步骤重新生成。

6.3.2 配置私有仓库

私有仓库默认的配置文件位于 `/etc/docker/registry/config.yml`，我们先在本地编辑 `config.yml`，之后挂载到容器中。

```
log:
  accesslog:
    disabled: true
  level: debug
  formatter: text
  fields:
    service: registry
    environment: staging
storage:
  delete:
    enabled: true
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
auth:
  htpasswd:
    realm: basic-realm
    path: /etc/docker/registry/auth/nginx.htpasswd
http:
  addr: :5000
  host: https://docker.domain.com
  headers:
    X-Content-Type-Options: [nosniff]
  http2:
    disabled: false
  tls:
    certificate: /etc/docker/registry/ssl/docker.domain.com.crt
    key: /etc/docker/registry/ssl/docker.domain.com.key
health:
  storagedriver:
    enabled: true
    interval: 10s
threshold: 3
```

6.3.3 生成 http 认证文件

```
$ mkdir auth

$ docker run --rm \
    --entrypoint htpasswd \
    httpd:2.4-alpine \
    -Bbn username password > auth/nginx.htpasswd
```

> 将上面的 `username` `password` 替换为你自己的用户名和密码。
>
> **安全提示**：上述命令会将密码明文暴露在 shell 历史记录和进程列表中。生产环境建议使用交互式方式输入密码（不带 `-b` 参数），或通过环境变量/文件传入。 配套 demo 的 `auth/README.md` 仅说明本地生成步骤，生成的 `auth/nginx.htpasswd` 已被忽略，不应提交。

> **版本说明**：使用 `httpd:2.4-alpine` 基于 Apache 2.4 的精简镜像。如需其他版本，可替换为 `httpd:latest` 或指定具体版本号如 `httpd:2.4.58-alpine`。

6.3.4 编辑 Docker Compose 配置

编辑 `compose.yaml` (或 `docker-compose.yml`) 配置如下：

```
services:
  registry:
    image: registry:2
    ports:
      - "443:5000"
    volumes:
      - ./:/etc/docker/registry
      - registry-data:/var/lib/registry

volumes:
  registry-data:
```

本书配套的 `06_repository/demo/` 也采用同样约定：容器内 registry 监听 `:5000`，宿主机通过 `443:5000` 暴露 HTTPS 服务。这样可以避免在容器内占用特权端口，同时仍让客户端使用 `https://docker.domain.com` 访问。

> **版本说明**：Compose 配置中明确指定 `registry:2` 版本。生产环境建议固定版本号（如 `registry:2.8.3`）而非使用 `latest`，以保证部署的可重复性。

6.3.5 修改 Hosts 文件

编辑 `/etc/hosts`

```
127.0.0.1 docker.domain.com
```

6.3.6 启动

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

这样我们就搭建好了一个具有权限认证、TLS 的私有仓库，接下来我们测试其功能是否正常。

6.3.7 测试私有仓库功能

由于自行签发的 CA 根证书不被系统信任，所以我们需要将 CA 根证书 `ssl/root-ca.crt` 移入 `/etc/docker/certs.d/docker.domain.com` 文件夹中。

```
$ sudo mkdir -p /etc/docker/certs.d/docker.domain.com

$ sudo cp ssl/root-ca.crt /etc/docker/certs.d/docker.domain.com/ca.crt
```

登录到私有仓库。

```
$ docker login docker.domain.com
```

尝试推送、拉取镜像。

```
$ docker pull ubuntu:24.04

$ docker tag ubuntu:24.04 docker.domain.com/username/ubuntu:24.04

$ docker push docker.domain.com/username/ubuntu:24.04

$ docker image rm docker.domain.com/username/ubuntu:24.04

$ docker pull docker.domain.com/username/ubuntu:24.04
```

> **版本说明**：示例使用 `ubuntu:24.04`，这是最新 LTS 版本。推送到私有仓库时保持了原有的版本标签，建议生产环境明确指定镜像版本号。 如果我们退出登录，尝试推送镜像。

```
$ docker logout docker.domain.com

$ docker push docker.domain.com/username/ubuntu:24.04

no basic auth credentials
```

发现会提示没有登录，不能将镜像推送到私有仓库中。

6.3.8 注意事项

如果你本机占用了 443 端口，你可以配置 Nginx 代理，这里不再赘述。

6.4 Nexus 3

使用 Docker 官方的 Registry 创建的仓库面临一些维护问题。比如某些镜像删除以后空间默认是不会回收的，需要一些命令去回收空间然后重启 Registry。在企业中把内部的一些工具包放入 Nexus 中是比较常见的做法，最新版本 Nexus3.x 全面支持 Docker 的私有镜像。所以使用 Nexus 3 一个软件来管理 Docker，Maven，Yum，PyPI 等是一个明智的选择。

6.4.1 启动 Nexus 容器

```
$ docker run -d --name nexus3 --restart=always \
    -p 8081:8081 \
    --mount src=nexus-data,target=/nexus-data \
    sonatype/nexus3:3.69
```

> **版本说明**：使用 sonatype/nexus3:3.69 指定稳定的 Nexus 版本。使用具体版本号而非 latest 可以避免意外升级带来的不兼容问题。 首次运行需等待 3-5 分钟，你可以使用 docker logs nexus3 -f 查看日志：

```
$ docker logs nexus3 -f

2021-03-11 15:31:21,990+0000 INFO  [jetty-main-1] *SYSTEM org.sonatype.nexus.bootstrap.jetty.JettySe
rver -
--------------------------------------------------

Started Sonatype Nexus OSS 3.69.0

--------------------------------------------------
```

> **版本说明**：上述日志中显示的是 Nexus 3.69 版本的启动日志。具体版本号会随着容器镜像版本而不同。 如果你看到以上内容，说明 Nexus 已经启动成功，你可以使用浏览器打开 http://YourIP:8081 访问 Nexus 了。

首次运行请通过以下命令获取初始密码：

```
$ docker exec nexus3 cat /nexus-data/admin.password

9266139e-41a2-4abb-92ec-e4142a3532cb
```

首次启动 Nexus 的默认账号是 `admin`，密码则是上边命令获取到的，点击右上角登录，首次登录需更改初始密码。

登录之后可以点击页面上方的齿轮按钮按照下面的方法进行设置。

6.4.2 创建仓库

创建一个私有仓库的方法：`Repository->Repositories` 点击右边菜单 `Create repository` 选择 `docker (hosted)`

- **Name**：仓库的名称
- **HTTP**：仓库单独的访问端口 (例如：**5001**)
- **Hosted -> Deployment policy**：请选择 **Allow redeploy** 否则无法上传 Docker 镜像。

其它的仓库创建方法请各位自己摸索，还可以创建一个 `docker (proxy)` 类型的仓库链接到 DockerHub 上。再创建一个 `docker (group)` 类型的仓库把刚才的 `hosted` 与 `proxy` 添加在一起。主机在访问的时候默认下载私有仓库中的镜像，如果没有将链接到 DockerHub 中下载并缓存到 Nexus 中。

6.4.3 添加访问权限

菜单 `Security->Realms` 把 Docker Bearer Token Realm 移到右边的框中保存。

添加用户规则：菜单 `Security->Roles->Create role` 在 `Privileges` 选项搜索 docker 把相应的规则移动到右边的框中然后保存。

添加用户：菜单 `Security->Users->Create local user` 在 `Roles` 选项中选中刚才创建的规则移动到右边的窗口保存。

6.4.4 NGINX 反向代理配置

证书的生成请参见 私有仓库认证 里面证书生成一节。

NGINX 示例配置如下

```
upstream register
{
    server "YourHostName OR IP":5001; #端口为上面添加私有镜像仓库时设置的 HTTP 选项的端口号
    check interval=3000 rise=2 fall=10 timeout=1000 type=http;
    check_http_send "HEAD / HTTP/1.0\r\n\r\n";
    check_http_expect_alive http_4xx;
}

server {
    server_name YourDomainName;#如果没有 DNS 服务器做解析，请删除此选项使用本机 IP 地址访问
    listen       443 ssl;

    ssl_certificate key/example.crt;
    ssl_certificate_key key/example.key;

    ssl_session_timeout  5m;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers  HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers   on;
    large_client_header_buffers 4 32k;
    client_max_body_size 300m;
    client_body_buffer_size 512k;
    proxy_connect_timeout 600;
    proxy_read_timeout   600;
    proxy_send_timeout   600;
    proxy_buffer_size    128k;
    proxy_buffers       4 64k;
    proxy_busy_buffers_size 128k;
    proxy_temp_file_write_size 512k;

    location / {
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://register;
        proxy_read_timeout 900s;

    }
    error_page   500 502 503 504  /50x.html;
}
```

6.4.5 Docker 主机访问镜像仓库

如果不启用 SSL 加密可以通过前面章节的方法添加非 https 仓库地址到 Docker 的配置文件中然后重启 Docker。

使用 SSL 加密以后程序需要访问就不能采用修改配置的方式了。具体方法如下：

```
$ openssl s_client -showcerts -connect YourDomainName OR HostIP:443 </dev/null 2>/dev/null|openssl x
509 -outform PEM >ca.crt
$ cat ca.crt | sudo tee -a /etc/ssl/certs/ca-certificates.crt
$ systemctl restart docker
```

使用 `docker login YourDomainName OR HostIP` 进行测试，用户名密码填写上面 Nexus 中设置的。

本章小结

本章介绍了 Docker Hub 的使用、私有仓库的搭建以及 Nexus 3 等企业级方案。

功能	说明
官方镜像	优先使用的基础镜像
拉取限制	匿名 100 次/6h，登录 200 次/6h
安全	推荐开启 2FA 并使用 Access Token
自动化	支持 Webhooks 和自动构建

延伸阅读

- 私有仓库：搭建自己的 Registry
- 镜像加速器：加速下载

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第七章 Dockerfile 指令详解

什么是 Dockerfile

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

在第四章中，我们通过 `docker commit` 学习了镜像的构成。但是，手动 `commit` 只能作为临时修补，并不适合作为生产环境镜像的构建方式。

使用 Dockerfile 构建镜像有以下优势：

- **自动化**：可以通过 `docker build` 命令自动构建镜像。
- **可重复性**：由于 Dockerfile 是文本文件，可以确保每次构建的结果一致。
- **版本控制**：Dockerfile 可以纳入版本控制系统 (如 Git)，便于追踪变更。
- **透明性**：任何人都可以通过阅读 Dockerfile 了解镜像的构建过程。

Dockerfile 编写哲学

在深入每个指令的细节之前，笔者想强调一个至关重要的原则：**Dockerfile 不是脚本，而是镜像的“设计图”**。这个区别决定了你如何思考每条指令的作用。

相比编写 Bash 脚本的思维（“按顺序执行这些命令”），Dockerfile 的思维应该是（“这一层镜像应该如何构建，下一层如何分层”）。这个思维转变会影响你的决策：

- **合并命令**：一个 `RUN apt-get update && apt-get install ...` 应该写在一起，而不是分开成多个 `RUN` 指令，因为它们是同一个“层”的逻辑
- **选择合适的指令**：`COPY` vs `ADD`、`CMD` vs `ENTRYPOINT` 这些选择不是随意的，而是根据镜像分层的语义来决定的
- **优化镜像大小**：最后才清理缓存、删除临时文件，让这些“瘦身”操作在同一层完成

这个章节将详细介绍各个指令。在学习指令语法时，请始终思考：“这个指令为什么要以这样的方式工作？如果我是 Docker，我应该如何设计它？”

Dockerfile 基本结构

Dockerfile 一般分为四部分：基础镜像信息、维护者信息、镜像操作指令和容器启动时执行指令。

指令详解

本章将详细讲解 Dockerfile 中的各个指令：

- RUN 执行命令
- COPY 复制文件
- ADD 更高级的复制文件
- CMD 容器启动命令
- ENTRYPOINT 入口点
- ENV 设置环境变量
- ARG 构建参数
- VOLUME 定义匿名卷
- EXPOSE 暴露端口
- WORKDIR 指定工作目录
- USER 指定当前用户
- HEALTHCHECK 健康检查
- ONBUILD 为他人作嫁衣裳
- LABEL 为镜像添加元数据
- SHELL 指令

高级特性

本章还将介绍 Dockerfile 的高级特性：

- 多阶段构建
- 多阶段构建实战：Laravel 应用

参考与最佳实践

此外，我们还将介绍 Dockerfile 的最佳实践和常见问题。

- 参考文档

Built with mdPress

使用 Dockerfile 构建镜像

构建镜像的基本命令格式为：

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

例如，在 Dockerfile 所在目录执行：

```
docker build -t my-image:1.0 .
```

关于版本号最佳实践

本章中的 Dockerfile 示例使用的基础镜像标签遵循以下原则：

- **通用标签**（如 `ubuntu:24.04`、`alpine`、`nginx`）：保持原样，无需修改
- **基础镜像版本号**（如 `node:22`、`python:3.12`）：使用主或次版本号而非完整版本号（patch），这样可以自动获取最新的补丁版本，确保获得安全更新
- **避免**：不建议使用 `latest` 标签和完整的 patch 版本号（如 `20.10.0`）作为基础镜像，因为这会导致构建的不可重现性或安全风险

读者在使用这些示例时，应根据实际生产环境需求选择合适的版本号。

更多关于 `docker build` 的用法，我们在实战中会结合具体指令进行演示。

7.1 RUN 执行命令

何时使用 RUN

`RUN` 是 Dockerfile 中最常用的指令，主要用于在镜像构建阶段执行命令来修改镜像。具体来说：

- **安装依赖**：`RUN apt-get install nginx`
- **编译程序**：`RUN gcc -o app main.c`
- **下载文件**：`RUN curl -O https://example.com/file.tar.gz`
- **配置系统**：`RUN mkdir -p /app/data`

理解 RUN 的核心是理解**镜像分层**：每一个 RUN 都会在当前层之上创建新的一层，这会影响镜像大小。因此，合理使用 RUN（特别是合并多个 RUN）是构建轻量级镜像的关键。

7.1.1 基本语法

```
RUN <command>
RUN ["executable", "param1", "param2"]
```

`RUN` 指令是 Dockerfile 中最常用的指令之一。它在 **当前镜像层** 之上创建一个新层，执行指定的命令，并提交结果。

7.1.2 两种格式对比

1. Shell 格式

```
RUN apt-get update
```

- **特点**：默认通过 `/bin/sh -c` 执行。
- **优势**：可以使用环境变量、管道、重定向等 Shell 特性。
- **示例**：

  ```
  RUN echo "Hello" > /test.txt
  ```

2. Exec 格式

```
RUN ["apt-get", "update"]
```

- **特点**：直接调用可执行文件，不经过 Shell。
- **优势**：避免 Shell 字符串解析问题，适用于参数中包含特殊字符的情况。
- **注意**：无法使用 $VAR 环境变量替换 (除非显式调用 shell)。

7.1.3 常见最佳实践

1. 组合命令：减少层数

每一个 RUN 指令都会新建一层镜像。为了减少镜像体积和层数，应使用 && 连接命令。

❌ **糟糕的写法** (创建 3 层)：

```
RUN apt-get update
RUN apt-get install -y nginx
RUN rm -rf /var/lib/apt/lists/*
```

✅ **推荐写法** (创建 1 层)：

```
RUN apt-get update && \
    apt-get install -y nginx && \
    rm -rf /var/lib/apt/lists/*
```

2. 清理缓存

在安装完软件后，立即清除缓存，可以显著减小镜像体积。

- **Debian/Ubuntu**:

  ```
  RUN apt-get update && apt-get install -y package-bar \
      && rm -rf /var/lib/apt/lists/*
  ```

- **Alpine**:

  ```
  RUN apk add --no-cache package-bar
  ```

3. 使用 `set -e` 和 `pipefail`

默认情况下，管道命令 `cmd1 | cmd2` 的返回值由最后一个命令 (`cmd2`) 决定，即使前面的命令失败了，整个 `RUN` 仍可能视为成功。

❌ **隐蔽的错误：**

```
## 如果下载失败，gzip 可能会报错，但如果不影响后续，构建可能继续

RUN wget http://error-url | gzip -d > file
```

✅ **推荐写法：**

```
SHELL ["/bin/bash", "-o", "pipefail", "-c"]
RUN wget http://url | gzip -d > file
```

7.1.4 常见问题

Q：为什么 `RUN cd /app` 不生效?

```
RUN cd /app
RUN touch hello.txt
```

结果：`hello.txt` 会出现在根目录 `/`，而不是 `/app`。**原因**：每个 `RUN` 都在一个新的 Shell/容器环境中执行。`cd` 只影响当前 `RUN` 的环境。**解决**：使用 `WORKDIR` 指令。

```
WORKDIR /app
RUN touch hello.txt
```

Q：环境变量不生效?

```
RUN export MY_VAR=hello
RUN echo $MY_VAR
```

结果：输出为空。**原因**：同上，环境变量只在当前 `RUN` 有效。**解决**：使用 `ENV` 指令，或在同一行 `RUN` 中导出。

```
ENV MY_VAR=hello
RUN echo $MY_VAR
```

7.1.5 高级技巧

1. 使用 BuildKit 的挂载缓存

BuildKit 支持在 `RUN` 指令中使用 `--mount` 挂载缓存，加速构建。

```
## 缓存 apt 包

RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
    --mount=type=cache,target=/var/lib/apt,sharing=locked \
    apt-get update && apt-get install -y gcc
```

```
## 缓存 Go 模块

RUN --mount=type=cache,target=/go/pkg/mod \
    go build -o app
```

2. 挂载密钥

安全地使用 SSH 密钥或 Token，而不将其记录在镜像中。

```
RUN --mount=type=secret,id=mysecret \
    my-private-tool --token-file /run/secrets/mysecret
```

不要在构建命令中 `cat`、`echo` 或打印密钥内容；应把 `/run/secrets/<id>` 路径交给真正消费密钥的工具。

3. Heredoc 语法

BuildKit 支持使用 heredoc 语法编写多行脚本，无需行末反斜杠 \ 连接：

```
RUN <<EOF
apt-get update
apt-get install -y \
  build-essential \
  curl \
  git
rm -rf /var/lib/apt/lists/*
EOF
```

优势：

- 可读性更强，不需要在每行末尾添加 \
- 避免在脚本中转义引号
- 支持多个 heredoc 块，可指定不同的 Shell

```
RUN <<EOF
echo "使用默认 /bin/sh"
EOF

RUN <<'EOF'
#!/bin/bash
set -euo pipefail
echo "使用 bash"
wget http://example.com/file.tar.gz
EOF
```

> 使用 heredoc 需要在 Dockerfile 首行声明 BuildKit 语法版本：`# syntax=docker/dockerfile:1`

7.2 COPY 复制文件

何时使用 COPY

COPY 是在构建镜像时，将构建上下文（Dockerfile 所在目录及其子目录）中的文件或目录复制到容器内的指令。它是处理应用代码、配置文件最常用的方式。

典型场景：

- 复制应用源码：`COPY . /app`
- 复制配置文件：`COPY nginx.conf /etc/nginx/nginx.conf`
- 复制静态资源：`COPY public /app/public`

为什么 COPY 比 ADD 更值得推荐？ 笔者建议在 90% 的情况下使用 COPY。原因是 COPY 的语义更清晰——它就是简单地复制文件。而 ADD 有额外的功能（自动解压、支持 URL），这些功能往往会带来意外的行为。除非你明确需要 ADD 的自动解压功能，否则用 COPY。详见 7.3 ADD 指令 中的详细对比。

7.2.1 基本语法

```
COPY [选项] <源路径>... <目标路径>
COPY [选项] ["<源路径1>", "<源路径2>", ... "<目标路径>"]
```

COPY 指令将构建上下文中的文件或目录复制到镜像内。

7.2.2 基本用法

复制单个文件

```
## 复制文件到指定目录

COPY package.json /app/

## 复制文件并重命名

COPY config.json /app/settings.json
```

复制多个文件

```
## 复制多个指定文件

COPY package.json package-lock.json /app/

## 使用通配符

COPY *.json /app/
COPY src/*.js /app/src/
```

复制目录

```
## 复制整个目录的内容（不是目录本身）

COPY src/ /app/src/
```

> ⚠️ **注意**：复制目录时，复制的是目录的 **内容**，不包含目录本身。

```
构建上下文:                  镜像内:
src/                        /app/src/
├── index.js      →         ├── index.js
└── utils.js                └── utils.js
```

7.2.3 通配符规则

COPY 支持 Go 的 `filepath.Match` 通配符规则：

通配符	说明	示例
`*`	匹配任意字符序列	`*.json`
`?`	匹配单个字符	`config?.json`
`[abc]`	匹配括号内任一字符	`[abc].txt`
`[a-z]`	匹配范围内字符	`file[0-9].txt`

```
COPY hom* /mydir/       # home.txt, homework.md 等
COPY hom?.txt /mydir/   # home.txt, homy.txt 等
COPY app[0-9].js /app/  # app0.js ~ app9.js
```

7.2.4 目标路径

绝对路径

```
COPY app.js /usr/src/app/
```

相对路径：基于 WORKDIR

```
WORKDIR /app
COPY package.json ./        # 复制到 /app/package.json
COPY src/ ./src/            # 复制到 /app/src/
```

自动创建目录

如果目标目录不存在，Docker 会自动创建：

```
## /app/config/ 不存在也会自动创建

COPY settings.json /app/config/
```

7.2.5 修改文件所有者

使用 `--chown` 选项设置文件的用户和组：

```
## 使用用户名和组名

COPY --chown=node:node package.json /app/

## 使用 UID 和 GID

COPY --chown=1000:1000 . /app/

## 只指定用户

COPY --chown=node . /app/
```

> 💡 结合 USER 指令使用，确保应用以非 root 用户运行。

7.2.6 保留文件元数据

COPY 会保留源文件的元数据：

- 读、写、执行权限
- 修改时间

这对于脚本文件特别重要：

```
## start.sh 的可执行权限会被保留

COPY start.sh /app/
```

7.2.7 COPY vs ADD

特性	COPY	ADD
复制本地文件	✅	✅
自动解压 tar	❌	✅
支持 URL	❌	✅ (不推荐)
推荐程度	✅ **推荐**	⚠️ 特殊场景使用

```
## 推荐：使用 COPY

COPY app.tar.gz /app/
RUN tar -xzf /app/app.tar.gz

## ADD 会自动解压（行为不明显，不推荐）

ADD app.tar.gz /app/
```

> 笔者建议：除非需要自动解压 tar 文件，否则始终使用 COPY。明确的行为比隐式的魔法更好。

7.2.8 多阶段构建中的 COPY

从其他构建阶段复制

```
## 构建阶段

FROM node:22 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

## 生产阶段

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

使用 --link 优化缓存

```
## 使用 --link 后，文件以独立层添加，不依赖前序指令

COPY --link --from=builder /app/dist /usr/share/nginx/html
```

`--link` 的优势：

- 更高效利用构建缓存
- 并行化构建过程
- 加速多阶段构建

> ⚠️ **注意**：使用 `--link` 需要满足以下条件：
>
> - 启用 BuildKit（Docker Engine 23.0+ 默认启用）
> - Dockerfile 语法版本 1.4 或更高（在首行声明 `# syntax=docker/dockerfile:1`）
> - 目标路径在前序指令中应不存在或为空目录

7.2.9 dockerignore

使用 `.dockerignore` 排除不需要复制的文件：

```
## .dockerignore

node_modules
.git
.env
*.log
Dockerfile
.dockerignore
```

这可以：

- 减小构建上下文大小
- 加速构建
- 避免复制敏感文件

7.2.10 最佳实践

1. 利用缓存，先复制依赖文件

```
## ✅ 好：先复制依赖定义，再安装，最后复制代码

COPY package.json package-lock.json ./
RUN npm install
COPY . .

## ❌ 差：一次性复制所有文件，代码变更会导致重新 npm install

COPY . .
RUN npm install
```

2. 使用 .dockerignore

```
## 确保 node_modules 不被复制

COPY . .

## .dockerignore 中应包含 node_modules

...
```

3. 明确复制路径

```
## ✅ 好：明确的路径

COPY src/ /app/src/
COPY package.json /app/

## ❌ 差：过于宽泛

COPY . .
```

> **🔥 踩坑实录**
>
> 某公司在优化 Node.js 应用的 Docker 镜像时，发现构建出来的镜像体积超过了 2GB，远远超过生产部署的需求。排查发现，Dockerfile 中使用了 `COPY . .` 把整个构建上下文复制进镜像，导致 `node_modules/`、`.git/` 目录和大量测试数据全部被打包进镜像。最初他们没有创建 `.dockerignore` 文件，默认会复制所有文件。解决方案是添加一个 `.dockerignore` 文件，排除这些不必要的目录，使镜像缩小到了 200MB。这个教训深刻地说明了：`.dockerignore` 和 `.gitignore` 一样重要，应该在项目初始化时就创建，而不是等到出现问题时才想起来。建议的标准做法是先复制 `package.json` 和 `package-lock.json` 安装依赖，再复制应用代码，同时在 `.dockerignore` 中明确列出 `node_modules`、`.git`、test 目录等。

7.3 ADD 更高级的复制文件

何时使用 ADD？何时用 COPY？

在开始前，让我们直言不讳：**在大多数情况下，你应该使用 COPY，而不是 ADD**。

ADD 在 COPY 基础上增加了两个额外功能，但这些功能往往引入复杂性而非便利：

1. 自动解压 tar 压缩包（有时你想复制一个 .tar.gz 本身，而 ADD 会意外地解压它）
2. 支持从 URL 下载文件（这个功能由于网络不稳定已被广泛认为是反模式）

实践中的建议：除非你明确需要自动解压功能（比如官方基础镜像构建根文件系统），否则始终使用 COPY。原因很简单——显式优于隐式。你的 Dockerfile 在 6 个月后被接手维护时，清晰的意图会让团队少走很多弯路。

7.3.1 基本语法

```
ADD [选项] <源路径>... <目标路径>
ADD [选项] ["<源路径>", ... "<目标路径>"]
```

ADD 在 COPY 基础上增加了两个功能：

1. 自动解压 tar 压缩包
2. 支持从 URL 下载文件 (不推荐)

7.3.2 ADD vs COPY 详细对比

特性	COPY	ADD
复制本地文件	✅	✅
自动解压 tar	❌	✅
支持 URL	❌	✅ (不推荐)
行为可预测性	✅ 高	⚠️ 低
推荐程度	✅ **优先使用**	仅解压场景

> **笔者建议**：除非需要自动解压 tar 文件，否则始终使用 COPY。明确的行为比隐式的魔法更好。

7.3.3 自动解压功能

基本用法：自动解压本地 tar

```
## 自动解压 tar.gz 到目标目录

ADD app.tar.gz /app/
```

ADD 会识别并解压以下格式：

- `.tar`
- `.tar.gz` / `.tgz`
- `.tar.bz2` / `.tbz2`
- `.tar.xz` / `.txz`

实际应用

官方基础镜像通常使用 ADD 解压根文件系统：

```
FROM scratch
ADD ubuntu-noble-core-cloudimg-amd64-root.tar.gz /
```

解压过程

```
ADD app.tar.gz /app/
        │
        ├─ 识别 .tar.gz 格式
        ├─ 自动解压
        └─ 内容放入 /app/

app.tar.gz 包含:          /app/ 目录结果:
├── src/                   ├── src/
│   └── main.py            │   └── main.py
└── config.json            └── config.json
```

7.3.4 URL 下载功能：不推荐

基本用法

```
## 从 URL 下载文件

ADD https://example.com/app.zip /app/app.zip
```

为什么不推荐

问题	说明
权限固定	下载的文件权限为 600，通常需要额外 RUN 修改
不会解压	URL 下载的压缩包不会自动解压
缓存问题	URL 内容变化时不会重新下载
层数增加	需要额外 RUN 清理

推荐替代方案

```
## ❌ 不推荐：使用 ADD 下载

ADD https://example.com/app.tar.gz /tmp/
RUN tar -xzf /tmp/app.tar.gz -C /app && rm /tmp/app.tar.gz

## ✅ 推荐：使用 RUN + curl

RUN curl -fsSL https://example.com/app.tar.gz | tar -xz -C /app
```

优势：

- 一条 RUN 完成下载、解压、清理
- 减少镜像层数
- 更清晰的构建意图

7.3.5 修改文件所有者

```
ADD --chown=node:node app.tar.gz /app/
ADD --chown=1000:1000 files/ /app/
```

7.3.6 何时使用 ADD

✅ 适合使用 ADD

```
## 解压本地 tar 文件

FROM scratch
ADD rootfs.tar.gz /

## 解压应用包

ADD dist.tar.gz /app/
```

❌ 不适合使用 ADD

```
## 复制普通文件（用 COPY）

ADD package.json /app/          # ❌
COPY package.json /app/         # ✅

## 下载文件（用 RUN + curl）

ADD https://example.com/file /  # ❌
RUN curl -fsSL ... -o /file     # ✅

## 需要保留 tar 不解压（用 COPY）

ADD archive.tar.gz /archives/   # ❌ 会解压
COPY archive.tar.gz /archives/  # ✅ 保持原样
```

7.3.7 缓存行为

ADD 可能导致构建缓存失效：

```
## 如果 app.tar.gz 内容变化，此层及后续层都需重建

ADD app.tar.gz /app/
RUN npm install
```

优化建议：

```
## 先复制依赖文件

COPY package*.json /app/
RUN npm install

## 再添加应用代码

ADD app.tar.gz /app/
```

7.3.8 最佳实践

1. 默认使用 COPY

```
## ✅ 大多数场景使用 COPY

COPY . /app/
```

2. 仅在需要解压时使用 ADD

```
## ✅ 自动解压场景

ADD app.tar.gz /app/
```

3. 不要用 ADD 下载文件

```
## ❌ 避免

ADD https://example.com/file.tar.gz /tmp/

## ✅ 推荐

RUN curl -fsSL https://example.com/file.tar.gz | tar -xz -C /app
```

4. 解压后清理

```
## 如果需要控制解压过程

COPY app.tar.gz /tmp/
RUN tar -xzf /tmp/app.tar.gz -C /app && \
    rm /tmp/app.tar.gz
```

7.4 CMD 容器启动命令

何时使用 CMD，何时使用 ENTRYPOINT?

在深入 CMD 的细节之前，我们需要理解一个关键问题：**CMD 和 ENTRYPOINT 应该在什么时候使用?**

这是 Dockerfile 使用中最常见的困惑之一。简单的答案是：

- **CMD**：定义容器的“默认命令”。如果用户在 `docker run` 时提供命令，CMD 会被覆盖
- **ENTRYPOINT**：定义容器的“入口脚本”。通常用于启动应用的某个特定部分

决策树：

1. **你的容器是否有不可变的启动逻辑?** 比如需要先做一些初始化工作，然后才运行应用 → 使用 ENTRYPOINT
2. **用户经常会在 `docker run` 时传入不同的命令吗?** → 使用 CMD（让用户灵活覆盖）
3. **大多数情况下，你希望容器始终以相同的方式启动?** → 使用 ENTRYPOINT

更多细节见 7.5 ENTRYPOINT 指令。

7.4.1 什么是 CMD

`CMD` 指令用于指定容器启动时默认执行的命令。它定义了容器的 “主进程”。

> **核心概念**：容器的生命周期 = 主进程的生命周期。CMD 指定的命令就是这个主进程。

7.4.2 语法格式

CMD 有三种格式：

格式	语法	推荐程度
exec 格式	`CMD ["可执行文件", "参数1", "参数2"]`	✅**推荐**
shell 格式	`CMD 命令 参数1 参数2`	⚠️ 简单场景
参数格式	`CMD ["参数1", "参数2"]`	配合 ENTRYPOINT

exec 格式：推荐

```
CMD ["nginx", "-g", "daemon off;"]
CMD ["python", "app.py"]
CMD ["node", "server.js"]
```

优点：

- 直接执行指定程序，是容器的 PID 1
- 正确接收信号 (如 SIGTERM)
- 无需 shell 解析

shell 格式

```
CMD echo "Hello World"
CMD nginx -g "daemon off;"
```

实际执行：会被包装为 `sh -c`

```
## 你写的

CMD echo $HOME

## 实际执行的

CMD ["sh", "-c", "echo $HOME"]
```

优点：可以使用环境变量、管道等 shell 特性

缺点：主进程是 sh，信号无法正确传递给应用

7.4.3 exec 格式 vs shell 格式

特性	exec 格式	shell 格式
主进程	指定的程序	`/bin/sh`
信号传递	✅ 正确	❌ 无法传递
环境变量	❌ 需要 shell 包装	✅ 自动解析
推荐使用	✅ 大多数场景	需要 shell 特性时

信号传递问题示例

```
## ❌ shell 格式：docker stop 会超时

CMD node server.js

## 实际是 sh -c "node server.js"

## SIGTERM 发给 sh，不会传递给 node

## ✅ exec 格式：docker stop 正常工作

CMD ["node", "server.js"]

## SIGTERM 直接发给 node

...
```

7.4.4 运行时覆盖 CMD

`docker run` 后的命令会覆盖 Dockerfile 中的 CMD：

```
## ubuntu 默认 CMD 是 /bin/bash

$ docker run -it ubuntu        # 进入 bash
$ docker run ubuntu cat /etc/os-release  # 覆盖为 cat 命令
```

```
Dockerfile:              docker run 命令:
CMD ["/bin/bash"]   +    cat /etc/os-release
        │                          │
        └──────→ 被覆盖 ←──────────┘
                    ↓
           执行: cat /etc/os-release
```

7.4.5 经典错误：容器立即退出

错误示例

```
## ❌ 容器启动后立即退出

CMD service nginx start
```

原因分析

```
1. CMD service nginx start
   ↓ 被转换为
2. CMD ["sh", "-c", "service nginx start"]
   ↓
3. sh 启动，执行 service 命令
   ↓
4. service 命令将 nginx 放到后台
   ↓
5. service 命令结束，sh 退出
   ↓
6. 容器主进程（sh）退出 → 容器停止
```

正确做法

```
## ✅ 让 nginx 在前台运行

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

7.4.6 CMD vs ENTRYPOINT

指令	用途	运行时行为
CMD	默认命令	`docker run` 参数会 **覆盖** 它
ENTRYPOINT	入口点	`docker run` 参数会 **追加** 到它后面

单独使用 CMD

```
## Dockerfile

CMD ["curl", "-s", "http://example.com"]
```

```
$ docker run myimage              # 执行默认命令
$ docker run myimage curl -v ...  # 完全覆盖
```

搭配 ENTRYPOINT

```
## Dockerfile

ENTRYPOINT ["curl", "-s"]
CMD ["http://example.com"]
```

```
$ docker run myimage              # curl -s http://example.com
$ docker run myimage http://other.com  # curl -s http://other.com（参数覆盖）
```

详见 ENTRYPOINT 入口点章节。

7.4.7 最佳实践

1. 优先使用 exec 格式

```
## ✅ 推荐

CMD ["python", "app.py"]

## ⚠️ 仅在需要 shell 特性时使用

CMD ["sh", "-c", "echo $PATH && python app.py"]
```

2. 确保应用在前台运行

```
## ✅ 前台运行

CMD ["nginx", "-g", "daemon off;"]
CMD ["apache2ctl", "-D", "FOREGROUND"]
CMD ["java", "-jar", "app.jar"]

## ❌ 不要使用后台服务命令

CMD service nginx start
CMD systemctl start nginx
```

3. 使用双引号

```
## ✅ 正确：双引号

CMD ["node", "server.js"]

## ❌ 错误：单引号（JSON 不支持）

CMD ['node', 'server.js']
```

4. 配合 ENTRYPOINT 使用

```
## 用于可配置参数的场景

ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8080"]

## 运行时可以覆盖端口

$ docker run myapp --port 9000

...
```

7.4.8 常见问题

Q：CMD 可以写多个吗?

不可以。多个 CMD 只有最后一个生效：

```
CMD ["echo", "first"]
CMD ["echo", "second"]  # 只有这个生效
```

Q：如何在 CMD 中使用环境变量?

```
## 方法1：使用 shell 格式

CMD echo "Port is $PORT"

## 方法2：显式使用 sh -c

CMD ["sh", "-c", "echo Port is $PORT"]
```

Q：为什么我的容器不响应 Ctrl+C?

可能是使用了 shell 格式，信号被 sh 吃掉了：

```
## ❌ 信号无法传递

CMD python app.py

## ✅ 信号正确传递

CMD ["python", "app.py"]
```

7.5 ENTRYPOINT 入口点

何时使用 ENTRYPOINT：从“容器”到“命令”

如果说 CMD 是“容器中的默认程序”，那么 ENTRYPOINT 就是“把容器变成一个命令”。这个思维转变决定了你何时使用 ENTRYPOINT。

使用 ENTRYPOINT 的典型场景：

1. **命令行工具**：你想让镜像像 `curl` 或 `wget` 一样使用

```
ENTRYPOINT ["curl"]
# docker run myimage http://example.com → curl http://example.com
```

2. **应用启动脚本**：你有一个初始化脚本，需要接收命令行参数

```
ENTRYPOINT ["/app/entrypoint.sh"]
# docker run myimage --debug → /app/entrypoint.sh --debug
```

3. **与 CMD 结合**：ENTRYPOINT 定义入口，CMD 定义默认参数

```
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8000"]
# docker run myimage → python app.py --port 8000
# docker run myimage --port 9000 → python app.py --port 9000
```

对比 CMD：如果没有这些“把容器当命令用”的需求，通常使用 CMD 就足够了。

7.5.1 什么是 ENTRYPOINT

`ENTRYPOINT` 指定容器启动时运行的入口程序。与 CMD 不同，ENTRYPOINT 定义的命令不会被 `docker run` 的参数覆盖，而是 **接收这些参数**。

> **核心作用**：让镜像像一个可执行程序一样使用，`docker run` 的参数作为这个程序的参数。

7.5.2 语法格式

格式	语法	推荐程度
exec 格式	`ENTRYPOINT ["可执行文件", "参数1"]`	✅**推荐**
shell 格式	`ENTRYPOINT 命令 参数`	⚠️ 不推荐

```
## exec 格式（推荐）
ENTRYPOINT ["nginx", "-g", "daemon off;"]
## shell 格式（不推荐）
ENTRYPOINT nginx -g "daemon off;"
```

7.5.3 ENTRYPOINT vs CMD

核心区别

特性	ENTRYPOINT	CMD
定位	固定的入口程序	默认参数
docker run 参数	追加为参数	完全覆盖
覆盖方式	`--entrypoint`	直接指定命令
适用场景	把镜像当命令用	提供默认行为

行为对比

```
## 只用 CMD
CMD ["curl", "-s", "http://example.com"]
```

```
$ docker run myimage              # curl -s http://example.com
$ docker run myimage -v           # 执行 -v（错误！）
$ docker run myimage curl -v ...  # curl -v ...（完全替换）
```

```
## 只用 ENTRYPOINT

ENTRYPOINT ["curl", "-s"]
```

```
$ docker run myimage                    # curl -s（缺参数）
$ docker run myimage http://example.com   # curl -s http://example.com ✓
```

```
## ENTRYPOINT + CMD 组合（推荐）

ENTRYPOINT ["curl", "-s"]
CMD ["http://example.com"]
```

```
$ docker run myimage                    # curl -s http://example.com（默认）
$ docker run myimage http://other.com     # curl -s http://other.com ✓
$ docker run myimage -v http://other.com  # curl -s -v http://other.com ✓
```

7.5.4 场景一：让镜像像命令一样使用

需求：启动前准备

创建一个查询公网 IP 的 “命令” 镜像。

使用 CMD 的问题

```
FROM ubuntu:24.04
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
CMD ["curl", "-s", "http://myip.ipip.net"]
```

```
$ docker run myip           # ✓ 正常工作
当前 IP：61.148.226.66

$ docker run myip -i        # ✗ 错误!
exec: "-i": executable file not found

## -i 替换了整个 CMD，被当作可执行文件

...
```

使用 ENTRYPOINT 解决

```
FROM ubuntu:24.04
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
ENTRYPOINT ["curl", "-s", "http://myip.ipip.net"]
```

```
$ docker run myip           # ✓ 正常工作
当前 IP: 61.148.226.66

$ docker run myip -i        # ✓ 添加 -i 参数
HTTP/1.1 200 OK
...
当前 IP: 61.148.226.66
```

交互图示

```
ENTRYPOINT ["curl", "-s", "http://myip.ipip.net"]
            |
docker run myip -i
            |
            ▼
curl -s http://myip.ipip.net -i
└──────────────────────────────┘
     ENTRYPOINT + docker run 参数
```

7.5.5 场景二：启动前的准备工作

需求

在启动主服务前执行初始化脚本 (如数据库迁移、权限设置)。

实现方式

```
# 建议使用 redis:7 或 redis:latest，具体版本号根据生产需求选择
FROM redis:7-alpine
COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["redis-server"]
```

docker-entrypoint.sh：

```
#!/bin/sh
set -e

## 准备工作

echo "Initializing..."

## 如果第一个参数是 redis-server，以 redis 用户运行

if [ "$1" = 'redis-server' ]; then
    chown -R redis:redis /data
    exec gosu redis "$@"
fi

## 其他命令直接执行

exec "$@"
```

工作流程

```
docker run redis                    docker run redis bash
        │                                   │
        ▼                                   ▼
docker-entrypoint.sh redis-server   docker-entrypoint.sh bash
        │                                   │
        ├─ 初始化                           ├─ 初始化
        ├─ chown -R redis:redis /data       │
        └─ exec gosu redis redis-server     └─ exec bash
           (以 redis 用户运行)                 (以 root 用户运行)
```

关键点

1. **exec “$@”**：用传入的参数替换当前进程，确保信号正确传递
2. **条件判断**：根据 CMD 不同执行不同逻辑
3. **用户切换**：使用 gosu 切换用户 (比 su 更适合容器)

7.5.6 场景三：带参数的应用

```
# 建议使用 python:3.12 或 python:3，具体版本号根据应用兼容性需求选择
FROM python:3.12-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt

ENTRYPOINT ["python", "app.py"]
CMD ["--host", "0.0.0.0", "--port", "8080"]
```

```
## 使用默认参数

$ docker run myapp

## 执行: python app.py --host 0.0.0.0 --port 8080

## 覆盖参数

$ docker run myapp --host 0.0.0.0 --port 9000

## 执行: python app.py --host 0.0.0.0 --port 9000

## 完全不同的参数

$ docker run myapp --help

## 执行: python app.py --help

...
```

7.5.7 覆盖 ENTRYPOINT

使用 `--entrypoint` 参数覆盖：

```
## 正常运行

$ docker run myimage

## 覆盖 ENTRYPOINT 进入 shell 调试

$ docker run --entrypoint /bin/sh myimage

## 覆盖 ENTRYPOINT 并传入参数

$ docker run --entrypoint /bin/cat myimage /etc/os-release
```

7.5.8 ENTRYPOINT 与 CMD 组合表

ENTRYPOINT	CMD	最终执行命令
无	无	无 (容器无法启动)
无	`["cmd", "p1"]`	`cmd p1`
`["ep", "p1"]`	无	`ep p1`
`["ep", "p1"]`	`["cmd", "p2"]`	`ep p1 cmd p2`
`ep p1` (shell)	`["cmd", "p2"]`	`/bin/sh -c "ep p1"` (CMD 被忽略)

> ⚠️ **注意**：shell 格式的 ENTRYPOINT 会忽略 CMD！

7.5.9 最佳实践

1. 使用 exec 格式

```
## ✅ 推荐
ENTRYPOINT ["python", "app.py"]

## ❌ 避免 shell 格式
ENTRYPOINT python app.py
```

2. 提供有意义的默认参数

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

3. 入口脚本使用 exec

```
#!/bin/sh

## 准备工作...

## 使用 exec 替换当前进程

exec "$@"
```

4. 处理信号

确保 ENTRYPOINT 脚本能正确传递信号：

```
#!/bin/bash
trap 'kill -TERM $PID' TERM INT

## 启动应用

app "$@" &
PID=$!

## 等待应用退出

wait $PID
```

7.6 ENV 设置环境变量

7.6.1 基本语法

```
## 格式一：单个变量

ENV <key> <value>

## 格式二：多个变量（推荐）

ENV <key1>=<value1> <key2>=<value2> ...
```

7.6.2 基本用法

设置单个变量

```
ENV NODE_VERSION 20
ENV APP_ENV production
```

设置多个变量

```
ENV NODE_VERSION=20 \
    APP_ENV=production \
    APP_NAME="My Application"
```

> 💡 包含空格的值用双引号括起来。

7.6.3 环境变量的作用

1. 后续指令中使用

```
ENV NODE_VERSION=20.10.0

## 在 RUN 中使用

RUN curl -fsSL https://nodejs.org/dist/v${NODE_VERSION}/node-v${NODE_VERSION}-linux-x64.tar.xz \
    | tar -xJ -C /usr/local --strip-components=1

## 在 WORKDIR 中使用

ENV APP_HOME=/app
WORKDIR $APP_HOME

## 在 COPY 中使用

COPY . $APP_HOME
```

2. 容器运行时使用

```
ENV DATABASE_URL=postgres://localhost/mydb
```

应用代码中可以读取：

```
import os
db_url = os.environ.get('DATABASE_URL')
```

```
const dbUrl = process.env.DATABASE_URL;
```

7.6.4 支持环境变量的指令

以下指令可以使用 `$变量名` 或 `${变量名}` 格式：

指令	示例
RUN	RUN echo $VERSION
CMD	CMD ["sh", "-c", "echo $HOME"]
ENTRYPOINT	同上
COPY	COPY . $APP_HOME
ADD	ADD app.tar.gz $APP_HOME
WORKDIR	WORKDIR $APP_HOME
EXPOSE	EXPOSE $PORT
VOLUME	VOLUME $DATA_DIR
USER	USER $USERNAME
LABEL	LABEL version=$VERSION
FROM	FROM node:$NODE_VERSION

> **注意**：FROM 是上表中的特例——它只能引用在第一条 FROM 之前用 ARG 声明的变量，**无法**使用 ENV 定义的环境变量（ENV 要进入构建阶段后才生效）。其余指令才能引用 ENV 环境变量。

7.6.5 运行时覆盖

使用 -e 或 --env 覆盖 Dockerfile 中定义的环境变量：

```
## 覆盖单个变量

$ docker run -e APP_ENV=development myimage

## 覆盖多个变量

$ docker run -e APP_ENV=development -e DEBUG=true myimage

## 从环境变量文件读取

$ docker run --env-file .env myimage
```

.env 文件格式

```
## .env

APP_ENV=development
DEBUG=true
DATABASE_URL=postgres://localhost/mydb
```

7.6.6 ENV vs ARG

特性	ENV	ARG
生效时间	构建时 + 运行时	仅构建时
持久性	写入镜像，运行时可用	构建后消失
覆盖方式	`docker run -e`	`docker build --build-arg`
适用场景	应用配置	构建参数 (如版本号)

组合使用

```
## ARG 接收构建时参数

ARG NODE_VERSION=20

## ENV 保存到运行时

ENV NODE_VERSION=$NODE_VERSION

## 后续指令使用

RUN curl -fsSL https://nodejs.org/dist/v${NODE_VERSION}/...
```

```
## 构建时指定版本

$ docker build --build-arg NODE_VERSION=18 -t myapp .
```

7.6.7 最佳实践

1. 统一管理版本号

```
## ✅ 好：版本集中管理

ENV NGINX_VERSION=1.30 \
    NODE_VERSION=22 \
    PYTHON_VERSION=3.12

RUN apt-get install nginx=${NGINX_VERSION}

## ❌ 差：版本分散在各处

RUN apt-get install nginx=1.30
```

2. 不要存储敏感信息

```
## ❌ 错误：密码写入镜像

ENV DB_PASSWORD=secret123

## ✅ 正确：运行时传入

## docker run -e DB_PASSWORD=xxx myimage

...
```

3. 为应用提供合理默认值

```
ENV APP_ENV=production \
    APP_PORT=8080 \
    LOG_LEVEL=info
```

4. 使用有意义的变量名

```
## ✅ 好：清晰的命名

ENV REDIS_HOST=localhost \
    REDIS_PORT=6379

## ❌ 差：模糊的命名

ENV HOST=localhost \
    PORT=6379
```

7.6.8 常见问题

Q：环境变量在 CMD 中不展开

exec 格式不会自动展开环境变量：

```
## ❌ 不会展开 $PORT

CMD ["python", "app.py", "--port", "$PORT"]

## ✅ 使用 shell 格式或显式调用 sh

CMD ["sh", "-c", "python app.py --port $PORT"]
```

Q：如何查看容器的环境变量

```
$ docker inspect mycontainer --format '{{json .Config.Env}}'
$ docker exec mycontainer env
```

Q：多行 ENV 还是多个 ENV

```
## ✅ 推荐：减少层数

ENV VAR1=value1 \
    VAR2=value2 \
    VAR3=value3

## ⚠️ 多个 ENV 会创建多层

ENV VAR1=value1
ENV VAR2=value2
ENV VAR3=value3
```

7.7 ARG 构建参数

7.7.1 基本语法

```
ARG <参数名>[=<默认值>]
```

ARG 指令定义构建时的变量，可以在 `docker build` 时通过 `--build-arg` 传入。

7.7.2 ARG vs ENV

特性	ARG	ENV
生效时间	仅构建时	构建时 + 运行时
持久性	构建后消失	写入镜像
覆盖方式	`docker build --build-arg`	`docker run -e`
适用场景	构建参数 (版本号等)	应用配置
可见性	`docker history` 可见	`docker inspect` 可见

```
构建时                          运行时
├─ ARG VERSION=1.0               │  (ARG 已消失)
├─ ENV APP_ENV=prod              │ APP_ENV=prod (仍存在)
└─ RUN echo $VERSION             │
```

> ⚠️ **安全提示**：不要用 ARG 传递密码等敏感信息，`docker history` 可以查看所有 ARG 值。

7.7.3 基本用法

定义和使用

```
## 定义有默认值的 ARG
## 建议使用主版本号，如 20 而非 20.10.0，这样可以自动获取最新的补丁版本

ARG NODE_VERSION=20

## 使用 ARG

FROM node:${NODE_VERSION}-alpine
RUN echo "Using Node.js $NODE_VERSION"
```

构建时覆盖

```
## 使用默认值

$ docker build -t myapp .

## 覆盖默认值

$ docker build --build-arg NODE_VERSION=18 -t myapp .
```

7.7.4 ARG 的作用域

FROM 之前的 ARG

```
## FROM 之前的 ARG 只能用于 FROM 指令

ARG REGISTRY=docker.io
ARG IMAGE_NAME=node

FROM ${REGISTRY}/${IMAGE_NAME}:20

## ❌ 这里无法使用上面的 ARG

RUN echo $REGISTRY  # 输出空
```

FROM 之后重新声明

```
ARG NODE_VERSION=20

FROM node:${NODE_VERSION}-alpine

## 需要再次声明才能使用

ARG NODE_VERSION
RUN echo "Node version: $NODE_VERSION"
```

多阶段构建中的 ARG

```
ARG BASE_VERSION=alpine

FROM node:22-${BASE_VERSION} AS builder

## 需要重新声明

ARG NODE_VERSION=20
RUN echo "Building with Node $NODE_VERSION"

FROM node:22-${BASE_VERSION}

## 每个阶段都需要重新声明

ARG NODE_VERSION=20
RUN echo "Running with Node $NODE_VERSION"
```

7.7.5 常见使用场景

1. 控制基础镜像版本

```
# 推荐使用主版本号（如 3），或指定次版本号（如 3.19），避免指定完整补丁版本
ARG ALPINE_VERSION=3
FROM alpine:${ALPINE_VERSION}
```

```
$ docker build --build-arg ALPINE_VERSION=3.19 .
```

2. 设置软件版本

```
# 使用次版本号 (1.30) 而非完整版本号 (1.30.0)，以便自动更新到最新补丁版本
ARG NGINX_VERSION=1.30

RUN curl -fsSL https://nginx.org/download/nginx-${NGINX_VERSION}.tar.gz | tar -xz
```

3. 配置构建环境

```
ARG BUILD_ENV=production
ARG ENABLE_DEBUG=false

RUN if [ "$ENABLE_DEBUG" = "true" ]; then \
        npm install --include=dev; \
    else \
        npm install --production; \
    fi
```

4. 配置私有仓库

```
ARG NPM_TOKEN

RUN echo "//registry.npmjs.org/:_authToken=${NPM_TOKEN}" > ~/.npmrc && \
    npm install && \
    rm ~/.npmrc
```

```
## 构建时传入 token

$ docker build --build-arg NPM_TOKEN=xxx .
```

7.7.6 将 ARG 传递给 ENV

如果需要在运行时使用 ARG 的值：

```
ARG VERSION=1.0.0

## 将 ARG 传递给 ENV

ENV APP_VERSION=$VERSION

## 运行时可用

CMD echo "App version: $APP_VERSION"
```

7.7.7 预定义 ARG

Docker 提供了一些预定义的 ARG，无需声明即可使用：

ARG	说明
HTTP_PROXY	HTTP 代理
HTTPS_PROXY	HTTPS 代理
NO_PROXY	不使用代理的地址
FTP_PROXY	FTP 代理

```
## 构建时使用代理

$ docker build --build-arg HTTP_PROXY=http://proxy:8080 .
```

7.7.8 最佳实践

1. 为 ARG 提供合理默认值

```
## ✅ 好：有默认值，建议使用主或次版本号而非完整版本号

ARG NODE_VERSION=20

## ⚠️ 需要每次传入

ARG NODE_VERSION
```

2. 不要用 ARG 存储敏感信息

```
## ❌ 错误：密码会被记录在镜像历史中

ARG DB_PASSWORD
RUN echo "password=$DB_PASSWORD" > /app/.env

## ✅ 正确：使用 secrets 或运行时环境变量

...
```

3. 使用 ARG 提高构建灵活性

```
# 推荐使用次版本号标签，允许Docker自动拉取最新的补丁版本
ARG BASE_IMAGE=python:3.12-slim
FROM ${BASE_IMAGE}

## 可以构建不同基础镜像的版本

## docker build --build-arg BASE_IMAGE=python:3-slim .

...
```

7.8 VOLUME 定义匿名卷

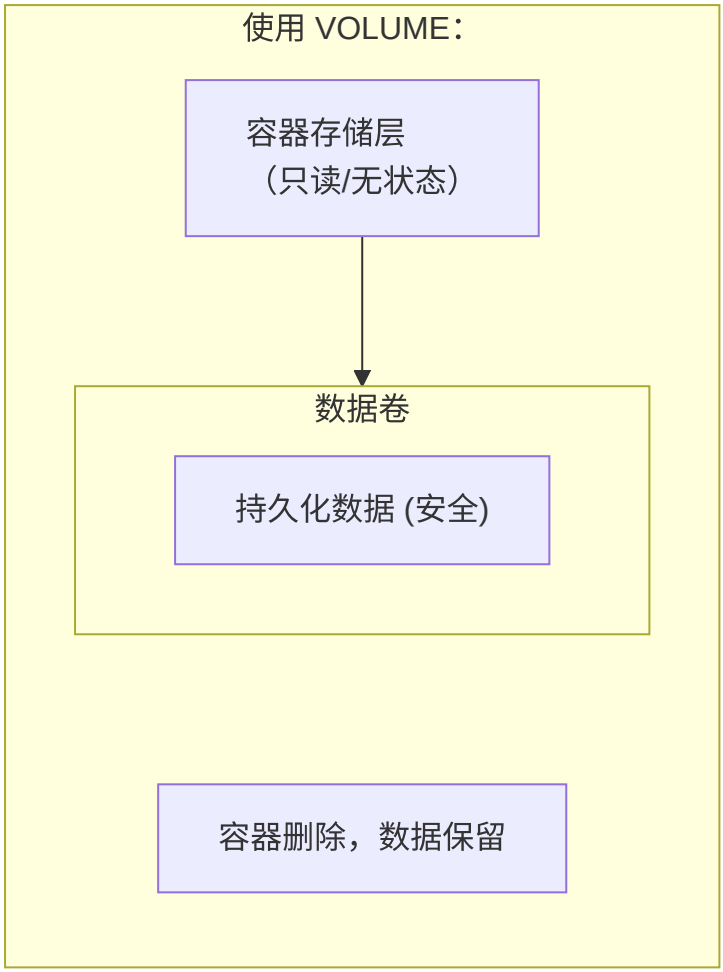
7.8.1 基本语法

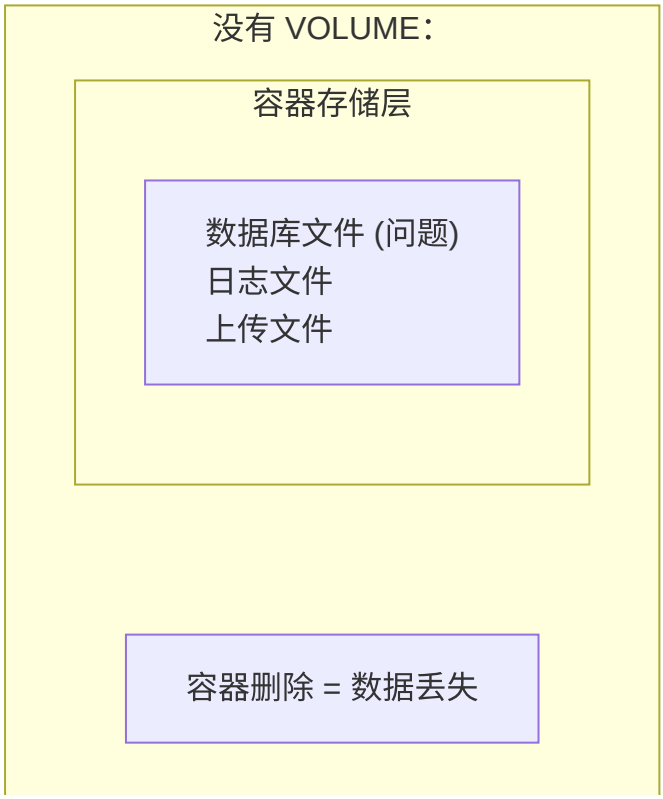
```
VOLUME ["/路径1", "/路径2"]
VOLUME /路径
```

VOLUME 指令创建挂载点，并标记为外部挂载的卷。

7.8.2 为什么使用 VOLUME

> **核心原则**：容器存储层应该保持无状态，任何运行时数据都应该存储在卷中。





Built with mdPress

7.8.3 基本用法

定义单个卷

```
FROM mysql:8.4
VOLUME /var/lib/mysql
```

定义多个卷

```
FROM myapp
VOLUME ["/data", "/logs", "/config"]
```

7.8.4 VOLUME 的行为

1. 自动创建匿名卷

如果运行时未指定挂载，Docker 会自动创建匿名卷：

```
$ docker run mysql:8.4
$ docker volume ls
DRIVER    VOLUME NAME
local     a1b2c3d4e5f6...  # 自动创建的匿名卷
```

2. 可被命名卷覆盖

```
## 使用命名卷替代匿名卷

$ docker run -v mysql_data:/var/lib/mysql mysql:8.4
```

3. 可被 Bind Mount 覆盖

```
## 使用宿主机目录替代

$ docker run -v /my/data:/var/lib/mysql mysql:8.4
```

7.8.5 VOLUME 在构建时的特殊行为

> ⚠️ **重要**：VOLUME 之后对该目录的修改会被丢弃！

```
FROM ubuntu
VOLUME /data

## ❌ 这个文件不会出现在镜像中！

RUN echo "hello" > /data/test.txt
```

原因：在构建过程中，VOLUME 指令会为该目录创建一个临时的匿名卷。后续 RUN 指令对该目录的写入实际发生在这个临时卷中，而非镜像层。当该 RUN 指令结束后，临时卷被丢弃，因此写入的内容不会保存到最终镜像中。注意：这与容器运行时创建的匿名卷是不同的——运行时创建的卷会在容器生命周期内持续存在。

正确做法

```
FROM ubuntu

## ✅ 先写入文件

RUN mkdir -p /data && echo "hello" > /data/test.txt

## 再声明 VOLUME

VOLUME /data
```

7.8.6 常见使用场景

数据库持久化

```
# 建议使用 postgres:16 或 postgres:latest，具体版本号根据数据库兼容性需求选择
FROM postgres:16
VOLUME /var/lib/postgresql/data
```

日志目录

```
FROM nginx
VOLUME /var/log/nginx
```

上传文件目录

```
FROM myapp
VOLUME /app/uploads
```

7.8.7 查看 VOLUME 定义

```
## 查看镜像定义的 VOLUME

$ docker inspect mysql:8.4 --format '{{json .Config.Volumes}}' | jq
{
  "/var/lib/mysql": {}
}

## 查看容器挂载的卷

$ docker inspect mycontainer --format '{{json .Mounts}}' | jq
```

7.8.8 VOLUME vs docker run -v

特性	Dockerfile VOLUME	docker run -v
定义时机	镜像构建时	容器运行时
默认行为	创建匿名卷	可指定命名卷或路径
灵活性	低 (固定路径)	高 (可任意指定)
适用场景	定义必须持久化的路径	灵活的数据管理

7.8.9 在 Compose 中

在 Compose 中配置如下：

```yaml
services:
  db:
    # 建议使用 postgres:16 或其他具体版本，避免 latest
    image: postgres:16
    volumes:
      # 命名卷（推荐）

      - postgres_data:/var/lib/postgresql/data
      # Bind Mount

      - ./init.sql:/docker-entrypoint-initdb.d/init.sql

volumes:
  postgres_data:  # 声明命名卷
```

7.8.10 安全注意事项

匿名卷可能导致数据丢失

```
## 使用 --rm 运行的容器，匿名卷会在容器删除时一起删除

$ docker run --rm mysql:8.4

## 容器停止后，数据丢失!

...
```

解决：始终使用命名卷

```
$ docker run -v mysql_data:/var/lib/mysql mysql:8.4
```

7.8.11 最佳实践

1. 定义必须持久化的路径

```
## 数据库必须使用卷

FROM postgres:16
VOLUME /var/lib/postgresql/data
```

2. 不要在 VOLUME 后修改目录

```
## ❌ 避免

VOLUME /app/data
RUN cp init-data.json /app/data/

## ✅ 正确

RUN mkdir -p /app/data && cp init-data.json /app/data/
VOLUME /app/data
```

3. 文档中说明 VOLUME 用途

```
## 持久化用户上传的文件

VOLUME /app/uploads

## 持久化数据库数据

VOLUME /var/lib/mysql
```

7.9 EXPOSE 暴露端口

7.9.1 基本语法

```
EXPOSE <端口> [<端口>/<协议>...]
```

EXPOSE 声明容器运行时提供服务的端口。这是一个 **文档性质的声明**，告诉使用者容器会监听哪些端口。

7.9.2 基本用法

```
## 声明单个端口

EXPOSE 80

## 声明多个端口

EXPOSE 80 443

## 声明 TCP 和 UDP 端口

EXPOSE 80/tcp
EXPOSE 53/udp
```

7.9.3 EXPOSE 的作用

1. 文档说明

告诉镜像使用者，容器将在哪些端口提供服务：

```
## 使用者一看就知道这是 web 应用

EXPOSE 80 443
```

```
## 查看镜像暴露的端口

$ docker inspect nginx --format '{{.Config.ExposedPorts}}'
map[80/tcp:{}]
```

2. 配合 -P 使用

使用 `docker run -P` 时，Docker 会自动映射 EXPOSE 的端口到宿主机随机端口：

```
## Dockerfile

EXPOSE 80
```

```
$ docker run -P nginx
$ docker port $(docker ps -q)
80/tcp -> 0.0.0.0:32768
```

7.9.4 EXPOSE vs -p

特性	EXPOSE	-p
位置	Dockerfile	docker run 命令
作用	声明/文档	实际端口映射
是否必需	否	是 (外部访问时)
映射发生时	不发生	运行时发生

EXPOSE 80
仅声明意图

docker run -p
实际端口映射
宿主机 ← → 容器

没有 EXPOSE 也能 -p

```
## 即使没有 EXPOSE，也可以使用 -p

FROM nginx

## 没有 EXPOSE

...
```

```
## 仍然可以映射端口

$ docker run -p 8080:80 mynginx
```

7.9.5 常见误解

误解：EXPOSE 会打开端口

```
## ❌ 错误理解：这不会让容器可从外部访问

EXPOSE 80
```

EXPOSE 不会：

- 自动进行端口映射
- 让服务可从外部访问
- 在容器启动时开启端口监听

EXPOSE 只是元数据声明。容器是否实际监听该端口，取决于容器内的应用。

正确理解

```
## Dockerfile

FROM nginx
EXPOSE 80    # 1. 声明：这个容器会在 80 端口提供服务
```

```
## 运行：需要 -p 才能从外部访问

$ docker run -p 8080:80 nginx    # 2. 映射：宿主机 8080 → 容器 80
```

7.9.6 最佳实践

1. 总是声明应用使用的端口

```
## Web 服务

FROM nginx
EXPOSE 80 443

## 数据库

FROM postgres
EXPOSE 5432

## Redis

FROM redis
EXPOSE 6379
```

2. 使用明确的协议

```
## 默认是 TCP

EXPOSE 80

## 明确指定 UDP

EXPOSE 53/udp

## 同时支持 TCP 和 UDP

EXPOSE 53/tcp 53/udp
```

3. 与应用实际端口保持一致

```
## ✅ 好：EXPOSE 与应用端口一致

ENV PORT=3000
EXPOSE 3000
CMD ["node", "server.js"]

## ❌ 差：EXPOSE 与应用端口不一致（误导）

EXPOSE 80
CMD ["node", "server.js"]  # 实际监听 3000
```

7.9.7 使用环境变量

```
ARG PORT=80
EXPOSE $PORT
```

7.9.8 在 Compose 中

在 Compose 中配置如下：

```
services:
  web:
    build: .
    ports:
      - "8080:80"    # 映射端口（类似 -p）
    expose:
      - "80"         # 仅声明（类似 EXPOSE）
```

expose 在 Compose 中仅用于容器间通信的文档说明，不进行端口映射。

7.10 WORKDIR 指定工作目录

7.10.1 基本语法

```
WORKDIR <工作目录路径>
```

WORKDIR 指定后续指令的工作目录。如果目录不存在，Docker 会自动创建。

7.10.2 基本用法

```
WORKDIR /app

RUN pwd          # 输出 /app
RUN echo "hello" > world.txt    # 创建 /app/world.txt
COPY . .         # 复制到 /app/
```

7.10.3 为什么需要 WORKDIR

常见错误

```
## ❌ 错误：cd 在下一个 RUN 中无效

RUN cd /app
RUN echo "hello" > world.txt    # 文件在根目录！
```

原因分析

```
RUN cd /app
    ↓
启动容器 → cd /app（仅内存变化）→ 提交镜像层 → 容器销毁
                                        |
                                        ↓ 工作目录未改变！
RUN echo "hello" > world.txt
    ↓
启动新容器（工作目录在 /）→ 创建 /world.txt
```

每个 RUN 都在新容器中执行，**前一个 RUN 的内存状态 (包括工作目录) 不会保留**。

正确做法

```
## ✅ 正确：使用 WORKDIR

WORKDIR /app
RUN echo "hello" > world.txt    # 创建 /app/world.txt
```

7.10.4 相对路径

WORKDIR 支持相对路径，基于上一个 WORKDIR：

```
WORKDIR /a
WORKDIR b
WORKDIR c

RUN pwd    # 输出 /a/b/c
```

7.10.5 使用环境变量

```
ENV APP_HOME=/app
WORKDIR $APP_HOME

RUN pwd    # 输出 /app
```

7.10.6 多阶段构建中的 WORKDIR

```
## 构建阶段
## 建议使用 node:22 或 node: 等具体版本标签，避免使用 latest

FROM node:22 AS builder
WORKDIR /build
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

## 生产阶段
## 建议使用 nginx:alpine 或其他具体版本

FROM nginx:alpine
WORKDIR /usr/share/nginx/html
COPY --from=builder /build/dist .
```

7.10.7 最佳实践

1. 尽早设置 WORKDIR

```
# 建议使用 node:22 等主/次版本号标签
FROM node:22
WORKDIR /app    # 尽早设置

COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "server.js"]
```

2. 使用绝对路径

```
## ✅ 推荐：绝对路径，意图明确

WORKDIR /app

## ⚠️ 避免：相对路径可能造成混淆

WORKDIR app
```

3. 不要用 RUN cd

```
## ❌ 避免

RUN cd /app && echo "hello" > world.txt

## ✅ 推荐

WORKDIR /app
RUN echo "hello" > world.txt
```

4. 适时重置 WORKDIR

```
WORKDIR /app

## ... 应用相关操作 ...

WORKDIR /data

## ... 数据相关操作 ...

...
```

7.10.8 与其他指令的关系

指令	WORKDIR 的影响
RUN	在 WORKDIR 中执行命令
CMD	在 WORKDIR 中启动
ENTRYPOINT	在 WORKDIR 中启动
COPY	相对目标路径基于 WORKDIR
ADD	相对目标路径基于 WORKDIR

```
WORKDIR /app

RUN pwd                 # /app
COPY . .                # 复制到 /app
CMD ["./start.sh"]      # /app/start.sh
```

7.10.9 运行时覆盖

使用 `-w` 参数覆盖工作目录：

```
$ docker run -w /tmp myimage pwd
/tmp
```

7.11 USER 指定当前用户

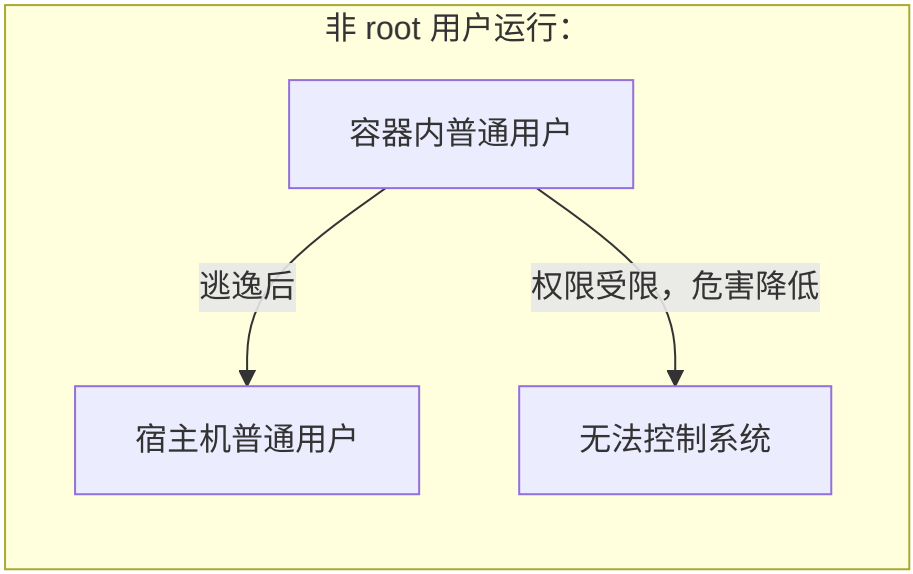
7.11.1 基本语法

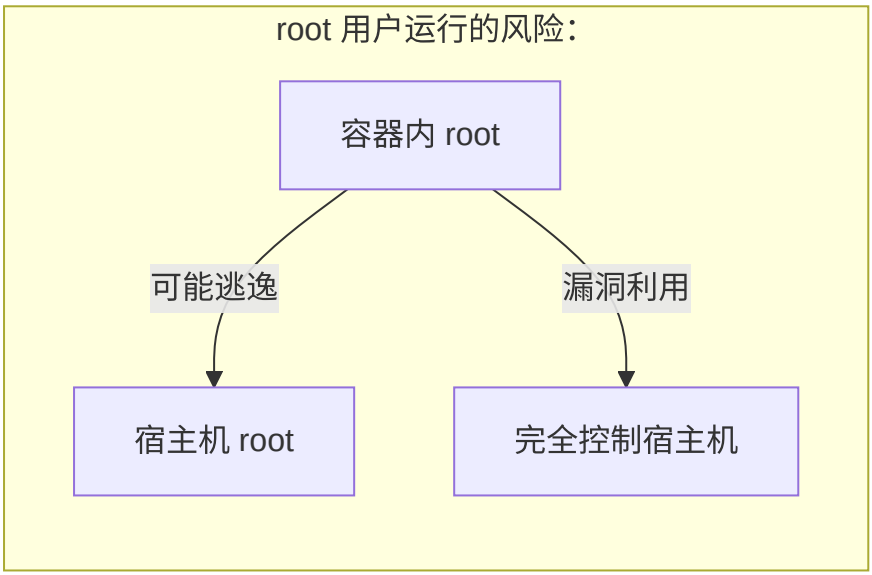
```
USER <用户名>[:<用户组>]
USER <UID>[:<GID>]
```

USER 指令切换后续指令 (RUN、CMD、ENTRYPOINT) 的执行用户。

7.11.2 为什么要使用 USER

> 笔者强调：以非 root 用户运行容器是最重要的安全实践之一。





Built with mdPress

7.11.3 基本用法

创建并切换用户

```
FROM node:22-alpine

## 1. 创建用户和组

RUN addgroup -g 1001 appgroup && \
    adduser -u 1001 -G appgroup -D appuser

## 2. 设置目录权限

WORKDIR /app
COPY --chown=appuser:appgroup . .

## 3. 切换用户

USER appuser

## 4. 后续命令以 appuser 身份运行

CMD ["node", "server.js"]
```

使用 UID/GID

```
## 也可以使用数字

USER 1001:1001
```

7.11.4 用户必须已存在

USER 指令只能切换到 **已存在** 的用户：

```
## ❌ 错误：用户不存在

USER nonexistent

## Error: unable to find user nonexistent

## ✅ 正确：先创建用户

RUN useradd -r -s /bin/false appuser
USER appuser
```

创建用户的方式

Debian/Ubuntu：

```
RUN groupadd -r appgroup && \
    useradd -r -g appgroup appuser
```

Alpine：

```
RUN addgroup -g 1001 -S appgroup && \
    adduser -u 1001 -S -G appgroup appuser
```

选项	说明
`-r` (useradd) / `-S` (adduser)	创建系统用户
`-g`	指定主组
`-G`	指定附加组
`-u`	指定 UID
`-s /bin/false`	禁用登录 shell

7.11.5 运行时切换用户

使用 gosu：推荐

在 ENTRYPOINT 脚本中切换用户时，不要使用 `su` 或 `sudo`，应使用 gosu：

```
FROM debian:bookworm

## 创建用户

RUN groupadd -r redis && useradd -r -g redis redis

## 安装 gosu

RUN apt-get update && apt-get install -y gosu && rm -rf /var/lib/apt/lists/*

COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["redis-server"]
```

docker-entrypoint.sh：

```
#!/bin/bash
set -e

## 以 root 执行初始化

chown -R redis:redis /data

## 用 gosu 切换到 redis 用户运行服务

exec gosu redis "$@"
```

为什么不用 su/sudo

问题	su/sudo	gosu
TTY 要求	需要	不需要
信号传递	不正确	正确
子进程	是	exec 替换
容器中使用	❌	✅

7.11.6 运行时覆盖用户

使用 `-u` 或 `--user` 参数：

```
## 以指定用户运行

$ docker run -u 1001:1001 myimage

## 以 root 运行（调试时）

$ docker run -u root myimage
```

7.11.7 文件权限处理

切换用户后，确保应用有权访问文件：

```dockerfile
FROM node:22-alpine

## 创建用户

RUN adduser -D -u 1001 appuser

WORKDIR /app

## 方式1：使用 --chown

COPY --chown=appuser:appuser . .

## 方式2：手动 chown（减少层数）

## COPY . .

## RUN chown -R appuser:appuser /app

USER appuser
CMD ["node", "server.js"]
```

7.11.8 最佳实践

1. 始终使用非 root 用户

```dockerfile
## ✅ 推荐

RUN adduser -D appuser
USER appuser
CMD ["myapp"]

## ❌ 避免

CMD ["myapp"]  # 以 root 运行
```

2. 使用固定 UID/GID

便于在宿主机和容器间共享文件：

```dockerfile
## 使用常见的非 root UID

RUN addgroup -g 1000 -S appgroup && \
    adduser -u 1000 -S -G appgroup appuser
USER 1000:1000
```

3. 多阶段构建中的 USER

```
## 构建阶段可以用 root

FROM node:22 AS builder
WORKDIR /app
COPY . .
RUN npm install && npm run build

## 生产阶段用非 root

FROM node:22-alpine
RUN adduser -D appuser
WORKDIR /app
COPY --from=builder --chown=appuser:appuser /app/dist .
USER appuser
CMD ["node", "server.js"]
```

7.11.9 常见问题

Q：权限被拒绝

```
permission denied: '/app/data.log'
```

解决：确保目录权限正确

```
RUN mkdir -p /app/data && chown appuser:appuser /app/data
```

Q：无法绑定低于 1024 的端口

非 root 用户无法绑定 80、443 等端口。

解决：

1. 使用高端口 (如 8080)
2. 在运行时映射端口：`docker run -p 80:8080`

7.12 HEALTHCHECK 健康检查

7.12.1 基本语法

```
HEALTHCHECK [选项] CMD <命令>
HEALTHCHECK NONE
```

HEALTHCHECK 指令告诉 Docker 如何判断容器状态是否正常。这是保障服务高可用的重要机制。

7.12.2 为什么需要 HEALTHCHECK

在没有 HEALTHCHECK 之前，Docker 只能通过 **进程退出码** 来判断容器状态。**问题场景**：

- Web 服务死锁，无法响应请求，但进程仍在运行
- 数据库正在启动中，尚未准备好接受连接
- 应用陷入死循环，CPU 爆满但进程存活

引入 HEALTHCHECK 后： Docker 定期执行指定的检查命令，根据返回值判断容器是否 “健康”。

```
容器状态转换：
Starting ──成功──> Healthy ──失败N次──> Unhealthy
                    ▲                       |
                    └─────────成功──────────┘
```

7.12.3 基本用法

Web 服务检查

```
# 注：nginx 镜像推荐使用具体的版本标签（如 nginx:1.28-alpine）
FROM nginx
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*

HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
  CMD curl -fs http://localhost/ || exit 1
```

命令返回值

- `0`：成功 (healthy)
- `1`：失败 (unhealthy)
- `2`：保留值 (不使用)

常用选项

选项	说明	默认值
`--interval`	两次检查的间隔	30s
`--timeout`	检查命令的超时时间	30s
`--start-period`	启动缓冲期 (期间失败不计入次数)	0s
`--retries`	连续失败多少次标记为 unhealthy	3

7.12.4 屏蔽健康检查

如果基础镜像定义了 HEALTHCHECK，但你不想使用它：

```
FROM my-base-image
HEALTHCHECK NONE
```

7.12.5 常见检查脚本

HTTP 服务

使用 `curl` 或 `wget`：

```
## 使用 curl

HEALTHCHECK CMD curl -f http://localhost/ || exit 1

## 使用 wget（Alpine 默认包含）

HEALTHCHECK CMD wget -q --spider http://localhost/ || exit 1
```

数据库

```
## MySQL

HEALTHCHECK CMD mysqladmin ping -h localhost || exit 1

## Redis

HEALTHCHECK CMD redis-cli ping || exit 1
```

自定义脚本

```
COPY healthcheck.sh /usr/local/bin/
HEALTHCHECK CMD ["healthcheck.sh"]
```

7.12.6 在 Compose 中使用

可以在 `compose.yaml` (或 `docker-compose.yml`) 中覆盖或定义健康检查：

```
services:
  web:
    image: nginx
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 1m30s
      timeout: 10s
      retries: 3
      start_period: 40s
```

带健康检查的依赖启动：

```yaml
services:
  web:
    depends_on:
      db:
        condition: service_healthy  # 等待 db 变健康才启动 web
  db:
    image: mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
```

7.12.7 查看健康状态

```
## 查看容器状态（包含健康信息）

$ docker ps
CONTAINER ID   STATUS
abc123         Up 1 minute (healthy)
def456         Up 2 minutes (unhealthy)

## 查看详细健康日志

$ docker inspect --format '{{json .State.Health}}' mycontainer | jq
{
  "Status": "healthy",
  "FailingStreak": 0,
  "Log": [
    {
      "Start": "...",
      "End": "...",
      "ExitCode": 0,
      "Output": "..."
    }
  ]
}
```

7.12.8 最佳实践

1. 避免副作用

健康检查会被频繁执行，不要在检查脚本中进行写操作或消耗大量资源的操作。

2. 使用轻量级工具

优先使用镜像中已有的工具 (如 `wget`)，避免为了健康检查安装庞大的依赖 (如 `curl`)。

3. 设置合理的 Start Period

应用启动可能需要时间 (如 Java 应用)。设置 `--start-period` 可以防止在启动阶段因检查失败而误判。

```
## 给应用 1 分钟启动时间

HEALTHCHECK --start-period=60s CMD curl -f http://localhost/ || exit 1
```

4. 只检查核心依赖

健康检查应主要关注 **当前服务** 是否可用，而不是检查其下游依赖 (数据库等)。下游依赖的检查应由应用逻辑处理。

5. 与容器编排平台的关系

不同平台对 HEALTHCHECK 的支持有所不同：

- **Docker Compose**：直接使用 Dockerfile 中定义的 HEALTHCHECK，也可在 `compose.yaml` 中覆盖
- **Docker Swarm**：使用 HEALTHCHECK 进行服务健康判断和滚动更新决策
- **Kubernetes**：**忽略** Dockerfile 中的 HEALTHCHECK，使用自己的 `livenessProbe` 和 `readinessProbe` 机制

如果应用同时部署在 Docker Compose 和 Kubernetes 环境中，建议在 Dockerfile 中保留 HEALTHCHECK（供 Compose 使用），同时在 Kubernetes 的 Pod 配置中定义对应的探针。

Built with mdPress

7.13 ONBUILD 为他人作嫁衣裳

7.13.1 基本语法

```
ONBUILD <其它指令>
```

ONBUILD 是一个特殊的指令，它后面跟的是其它指令 (如 RUN，COPY 等)，这些指令 **在当前镜像构建时不会执行**，只有当以当前镜像为基础镜像去构建下一级镜像时才会被执行。

7.13.2 为什么需要 ONBUILD

ONBUILD 主要用于制作 **语言栈基础镜像** 或 **框架基础镜像**。

场景：维护 Node.js 项目

假设你有多个 Node.js 项目，它们的构建流程都一样：

1. 创建目录
2. 复制 `package.json`
3. 执行 `npm install`
4. 复制源码
5. 启动应用

如果不使用 ONBUILD，每个项目的 Dockerfile 都要重复这些步骤，且通过 COPY 复制文件时，基础镜像无法预知子项目的文件名。

使用 ONBUILD 的解决方案

基础镜像 (my-node-base)：

```
FROM node:22-alpine
WORKDIR /app

## 这些指令将在子镜像构建时执行

ONBUILD COPY package*.json ./
ONBUILD RUN npm install
ONBUILD COPY . .

CMD ["npm", "start"]
```

子项目 Dockerfile：

```
FROM my-node-base

## 只需要一行!

## 构建时会自动执行 COPY 和 RUN

...
```

7.13.3 执行机制

```
基础镜像构建：
Dockerfile (含 ONBUILD) ──build──> 基础镜像 (记录了 ONBUILD 触发器)
                                    (指令未执行)

子镜像构建：
FROM 基础镜像 ──build──> 读取基础镜像触发器 ──> 执行触发器指令 ──> 继续执行子 Dockerfile
```

7.13.4 常见使用场景

1. 自动处理依赖安装

```
## Python 基础镜像

ONBUILD COPY requirements.txt ./
ONBUILD RUN pip install -r requirements.txt
```

2. 自动编译代码

```
## Go 基础镜像

ONBUILD COPY . .
ONBUILD RUN go build -o app main.go
```

3. 处理静态资源

```
## Nginx 静态网站基础镜像

ONBUILD COPY dist/ /usr/share/nginx/html/
```

7.13.5 注意事项

1. 继承性限制

`ONBUILD` 指令 **只会继承一次**。

- 镜像 A (含 ONBUILD)
- 镜像 B (FROM A) -> 触发 ONBUILD
- 镜像 C (FROM B) -> **不会** 再次触发 ONBUILD

2. 构建上下文

子镜像构建时，`ONBUILD COPY . .` 中的 `.` 指的是 **子项目** 的构建上下文，而不是基础镜像的上下文。

3. 不允许级联

`ONBUILD ONBUILD` 是非法的。你不能写 `ONBUILD ONBUILD COPY ...`。

4. 可能会导致构建失败

由于 `ONBUILD` 实际上是在子镜像中执行指令，如果子项目的上下文不满足要求 (例如缺少 `package.json`)，会导致子镜像构建失败，且错误信息可能比较隐晦。

7.13.6 最佳实践

1. 命名规范

建议在镜像标签中添加 `-onbuild` 后缀，明确告知使用者该镜像包含触发器。

```
node:22-onbuild
python:3.12-onbuild
```

2. 避免执行耗时操作

尽量不要在 `ONBUILD` 中执行过于耗时或不确定的操作 (如更新系统软件)，这会让子镜像构建变得缓慢且不可控。

3. 清理工作

如果 `ONBUILD` 指令产生了临时文件，最好在同一个指令链中清理，或者提供机制让子镜像清理。

7.14 LABEL 为镜像添加元数据

7.14.1 基本语法

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

LABEL 指令以键值对的形式给镜像添加元数据。这些数据不会影响镜像的功能，但可以帮助用户理解镜像，或被自动化工具使用。

7.14.2 为什么需要 LABEL

1. **版本管理**：记录版本号、构建时间、Git Commit ID
2. **联系信息**：维护者邮箱、文档地址、支持渠道
3. **自动化工具**：CI/CD 工具可以读取标签触发操作
4. **许可证信息**：声明开源协议

7.14.3 基本用法

定义单个标签

```
LABEL version="1.0"
LABEL description="这是一个 Web 应用服务器"
```

定义多个标签：推荐

```
LABEL maintainer="user@example.com" \
      version="1.2.0" \
      description="My App Description" \
      org.opencontainers.image.authors="Yeasy"
```

> 💡 包含空格的值需要用引号括起来。

7.14.4 常用标签规范

为了标准和互操作性，推荐使用 OCI Image Format Specification 定义的标准标签：

标签 Key	说明	示例
`org.opencontainers.image.created`	构建时间(RFC 3339)	`2024-01-01T00:00:00Z`
`org.opencontainers.image.authors`	作者/维护者	`support@example.com`
`org.opencontainers.image.url`	项目主页	`https://example.com`
`org.opencontainers.image.documentation`	文档地址	`https://example.com/docs`
`org.opencontainers.image.source`	源码仓库	`https://github.com/user/repo`
`org.opencontainers.image.version`	版本号	`1.0.0`
`org.opencontainers.image.licenses`	许可证	`MIT`
`org.opencontainers.image.title`	镜像标题	`My App`
`org.opencontainers.image.description`	描述	`Production ready web server`

示例

```
LABEL org.opencontainers.image.authors="yeasy" \
      org.opencontainers.image.documentation="https://yeasy.gitbooks.io" \
      org.opencontainers.image.source="https://github.com/yeasy/docker_practice" \
      org.opencontainers.image.licenses="MIT"
```

7.14.5 MAINTAINER 指令：已废弃

旧版本的 Dockerfile 中常看到 MAINTAINER 指令：

```
## ❌ 已弃用

MAINTAINER user@example.com
```

现在推荐使用 LABEL：

```
## ✅ 推荐

LABEL maintainer="user@example.com"

## 或

LABEL org.opencontainers.image.authors="user@example.com"
```

7.14.6 动态标签

配合 ARG 使用，可以在构建时动态注入标签：

```
ARG BUILD_DATE
ARG VCS_REF

LABEL org.opencontainers.image.created=$BUILD_DATE \
      org.opencontainers.image.revision=$VCS_REF
```

构建命令：

```
$ docker build \
  --build-arg BUILD_DATE=$(date -u +'%Y-%m-%dT%H:%M:%SZ') \
  --build-arg VCS_REF=$(git rev-parse --short HEAD) \
  .
```

7.14.7 查看标签

docker inspect

查看镜像的标签信息：

```
$ docker inspect nginx --format '{{json .Config.Labels}}' | jq
{
  "maintainer": "NGINX Docker Maintainers <docker-maint@nginx.com>"
}
```

过滤器

可以使用标签过滤镜像：

```
## 列出作者是 yeasy 的所有镜像

$ docker images --filter "label=org.opencontainers.image.authors=yeasy"

## 删除所有带有特定标签的镜像

$ docker rmi $(docker images -q --filter "label=stage=builder")
```

7.15 SHELL 指令

7.15.1 基本语法

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

SHELL 指令允许覆盖 Docker 默认的 shell。

- **Linux 默认**：`["/bin/sh", "-c"]`
- **Windows 默认**：`["cmd", "/S", "/C"]`

该指令会影响后续的 RUN，CMD，ENTRYPOINT 指令 (当它们使用 shell 格式时)。

7.15.2 为什么要用 SHELL 指令

1. 使用 bash 特性

默认的 /bin/sh (通常是 dash 或 alpine 的 ash) 功能有限。如果你需要使用 bash 的特有功能 (如数组、{} 扩展、pipefail 等)，可以切换 shell。

```
FROM ubuntu:24.04

## 切换到 bash

SHELL ["/bin/bash", "-c"]

## 现在可以使用 bash 特性了

RUN echo {a..z}
```

2. 增强错误处理

默认情况下，管道命令 cmd1 | cmd2 只要 cmd2 成功，整个指令就视为成功。这可能掩盖构建错误。

```
## ❌ 这里的 wget 失败了，但构建继续（因为 tar 成功了）

RUN wget -O - https://invalid-url | tar xz
```

使用 SHELL 启用 pipefail：

```
## ✅ 启用 pipefail

SHELL ["/bin/bash", "-o", "pipefail", "-c"]

## 如果 wget 失败，整个 RUN 就会失败

RUN wget -O - https://invalid-url | tar xz
```

3. Windows 环境

在 Windows 容器中，经常需要在 `cmd` 和 `powershell` 之间切换。

```
FROM mcr.microsoft.com/windows/servercore:ltsc2022

## 默认是 cmd

RUN echo Default shell is cmd

## 切换到 powershell

SHELL ["powershell", "-command"]
RUN Write-Host "Hello from PowerShell"

## 切回 cmd

SHELL ["cmd", "/S", "/C"]
```

7.15.3 作用范围

`SHELL` 指令可以出现多次，每次只影响其后的指令：

```
FROM ubuntu:24.04

## 使用默认 sh

RUN echo "Using sh"

SHELL ["/bin/bash", "-c"]

## 使用 bash

RUN echo "Using bash"

SHELL ["/bin/sh", "-c"]

## 回到 sh

RUN echo "Using sh again"
```

7.15.4 对其他指令的影响

`SHELL` 影响的是所有使用 **shell 格式** 的指令：

指令格式	是否受 SHELL 影响
`RUN command`	✅ 是
`RUN ["exec", "param"]`	❌ 否
`CMD command`	✅ 是
`CMD ["exec", "param"]`	❌ 否
`ENTRYPOINT command`	✅ 是
`ENTRYPOINT ["exec", "param"]`	❌ 否

7.15.5 最佳实践

1. 推荐开启 pipefail

对于使用 bash 的镜像，强烈建议开启 `pipefail`，以确保构建过程中的错误能被及时捕获。

```
SHELL ["/bin/bash", "-o", "pipefail", "-c"]
```

2. 明确意图

如果由于脚本需求必须更改 shell，最好在 Dockerfile 中显式声明，而不是依赖默认行为。

3. 尽量保持一致

避免在 Dockerfile 中频繁切换 SHELL，这会使构建过程难以理解和调试。尽量在头部定义一次即可。

7.16 参考文档

官方文档

- `Dockerfile` 官方参考手册：https://docs.docker.com/engine/reference/builder/
- `Dockerfile` 最佳实践指南：https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- `Docker` 官方镜像 `Dockerfile` 库：https://github.com/docker-library/docs

常用指令总结

Dockerfile 中的常用指令包括：

- **FROM**: 指定基础镜像，必须是第一条指令
- **RUN**: 在镜像中执行命令，用于安装软件包等
- **COPY**: 复制文件到镜像中
- **ADD**: 更高级的复制文件（支持 URL 和自动解压）
- **CMD**: 容器默认执行的命令
- **ENTRYPOINT**: 容器启动时的入口点
- **ENV**: 设置环境变量
- **ARG**: 构建时的参数变量
- **VOLUME**: 定义匿名卷挂载点
- **EXPOSE**: 声明容器监听的端口
- **WORKDIR**: 设置工作目录
- **USER**: 指定运行容器时的用户
- **HEALTHCHECK**: 配置容器健康检查
- **ONBUILD**: 设置触发器指令，在子镜像构建时执行
- **LABEL**: 为镜像添加元数据标签
- **SHELL**: 指定 RUN 等指令使用的 shell

Built with mdPress

最佳实践建议

1. 使用具体的基础镜像版本标签而非 latest
2. 最小化镜像层数，合并 RUN 指令
3. 使用 .dockerignore 文件排除不必要的文件
4. 安装必要的软件包后清理缓存
5. 使用多阶段构建减小最终镜像体积
6. 避免以 root 身份运行容器应用

相关资源

- Docker 官方镜像库：https://hub.docker.com/
- Docker 镜像构建最佳实践：https://docs.docker.com/build/building/best-practices/

Built with mdPress

7.17 多阶段构建

在 Docker 17.05 版本之前，我们构建 Docker 镜像时，通常会采用两种方式：

7.17.1 全部放入一个 Dockerfile

一种方式是将所有的构建过程包含在一个 `Dockerfile` 中，包括项目及其依赖库的编译、测试、打包等流程，这里可能会带来的一些问题：

- 镜像层次多，镜像体积较大，部署时间变长
- 源代码存在泄露的风险

例如，编写 `app.go` 文件，该程序输出 `Hello World!`

```
package main

import "fmt"

func main(){
    fmt.Printf("Hello World!");
}
```

编写 `Dockerfile.one` 文件

```
FROM golang:alpine

RUN apk --no-cache add git ca-certificates

WORKDIR /go/src/github.com/go/helloworld/

COPY app.go .

RUN go mod init helloworld \
  && go get github.com/go-sql-driver/mysql \
  && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app . \
  && cp /go/src/github.com/go/helloworld/app /root

WORKDIR /root/

CMD ["./app"]
```

构建镜像

```
$ docker build -t go/helloworld:1 -f Dockerfile.one .
```

7.17.2 分散到多个 Dockerfile

另一种方式，就是我们事先在一个 `Dockerfile` 将项目及其依赖库编译测试打包好后，再将其拷贝到运行环境中，这种方式需要我们编写两个 `Dockerfile` 和一些编译脚本才能将其两个阶段自动整合起来，这种方式虽然可以很好地规避第一种方式存在的风险，但明显部署过程较复杂。

例如，编写 `Dockerfile.build` 文件

```
FROM golang:alpine

RUN apk --no-cache add git

WORKDIR /go/src/github.com/go/helloworld

COPY app.go .

RUN go mod init helloworld \
  && go get github.com/go-sql-driver/mysql \
  && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

编写 `Dockerfile.copy` 文件

```
FROM alpine:3

RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY app .

CMD ["./app"]
```

新建 `build.sh`

```
#!/bin/sh
echo Building go/helloworld:build

docker build -t go/helloworld:build . -f Dockerfile.build

docker create --name extract go/helloworld:build
docker cp extract:/go/src/github.com/go/helloworld/app ./app
docker rm -f extract

echo Building go/helloworld:2

docker build --no-cache -t go/helloworld:2 . -f Dockerfile.copy
rm ./app
```

现在运行脚本即可构建镜像

```
$ chmod +x build.sh

$ ./build.sh
```

对比两种方式生成的镜像大小

```
$ docker image ls

REPOSITORY      TAG    IMAGE ID        CREATED         SIZE
go/helloworld   2      f7cf3465432c    22 seconds ago  6.47MB
go/helloworld   1      f55d3e16affc    2 minutes ago   295MB
```

7.17.3 使用多阶段构建

为解决以上问题，Docker v17.05 开始支持多阶段构建 (`multistage builds`)。使用多阶段构建我们就可以很容易解决前面提到的问题，并且只需要编写一个 `Dockerfile`：

例如，编写 `Dockerfile` 文件

```
FROM golang:alpine as builder

RUN apk --no-cache add git

WORKDIR /go/src/github.com/go/helloworld/

RUN go mod init helloworld \
  && go get github.com/go-sql-driver/mysql

COPY app.go .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:3 as prod

RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY --from=builder /go/src/github.com/go/helloworld/app .

CMD ["./app"]
```

构建镜像

```
$ docker build -t go/helloworld:3 .
```

对比三个镜像大小

```
$ docker image ls

REPOSITORY        TAG   IMAGE ID         CREATED             SIZE
go/helloworld     3     d6911ed9c846     7 seconds ago       6.47MB
go/helloworld     2     f7cf3465432c     22 seconds ago      6.47MB
go/helloworld     1     f55d3e16affc     2 minutes ago       295MB
```

很明显使用多阶段构建的镜像体积小，同时也完美解决了上边提到的问题。

> **Go Modules 最佳实践**：上述示例为简化演示在 Dockerfile 中临时执行 `go mod init`。在实际项目中，通常已在代码仓库中维护好 `go.mod` 和 `go.sum` 文件。推荐的 Dockerfile 写法是先拷贝这两个文件并执行 `go mod download` 以利用 Docker 层缓存，再拷贝源码并构建：
>
> ```
> COPY go.mod go.sum ./
> RUN go mod download
> COPY . .
> RUN go build -o app .
> ```

7.17.4 只构建某一阶段的镜像

我们可以使用 `as` 来为某一阶段命名，例如

```
FROM golang:alpine as builder
```

例如当我们只想构建 `builder` 阶段的镜像时，增加 `--target=builder` 参数即可

```
$ docker build --target builder -t username/imagename:tag .
```

7.17.5 构建时从其他镜像复制文件

上面例子中我们使用 `COPY --from=builder /go/src/github.com/go/helloworld/app .` 通过已命名阶段（`as builder`）从上一阶段的镜像中复制文件（命名引用比 `--from=0` 这种位置索引更易读，也是当前官方推荐的最佳实践）；我们也可以复制任意镜像中的文件。

```
COPY --from=nginx:1.28-alpine /etc/nginx/nginx.conf /nginx.conf
```

7.18 实战多阶段构建 Laravel 镜像

> 本节适用于 PHP 开发者阅读。Laravel 基于 8.x 版本，各个版本的文件结构可能会有差异，请根据实际自行修改。

7.18.1 准备

新建一个 Laravel 项目或在已有的 Laravel 项目根目录下新建 Dockerfile .dockerignore laravel.conf 文件。

在 .dockerignore 文件中写入以下内容。

```
.idea/
.git/

vendor/

node_modules/

public/js/
public/css/
public/mix-manifest.json

yarn-error.log

bootstrap/cache/*
storage/

## 自行添加其他需要排除的文件，例如 .env.* 文件

...
```

在 laravel.conf 文件中写入 nginx 配置。

```
server {
  listen 80 default_server;
  root /app/laravel/public;
  index index.php index.html;

  location / {
      try_files $uri $uri/ /index.php?$query_string;
  }

  location ~ .*\.php(\/.*)*$ {
    fastcgi_pass   laravel:9000;
    include        fastcgi.conf;

    # fastcgi_connect_timeout 300;

    # fastcgi_send_timeout 300;

    # fastcgi_read_timeout 300;

  }
}
```

7.18.2 前端构建

第一阶段进行前端构建。

```
# 注: node 镜像推荐使用具体的版本标签（如 node:22-alpine）
FROM node:22-alpine as frontend

COPY package.json /app/

RUN set -x ; cd /app \
      && npm install --registry=https://registry.npmmirror.com

COPY webpack.mix.js webpack.config.js tailwind.config.js /app/
COPY resources/ /app/resources/

RUN set -x ; cd /app \
      && touch artisan \
      && mkdir -p public \
      && npm run production
```

7.18.3 安装 Composer 依赖

第二阶段安装 Composer 依赖。

```
# 注：composer 镜像推荐使用具体的版本标签（如 composer:2.x）
FROM composer:2 as composer

COPY database/ /app/database/
COPY composer.json composer.lock /app/

RUN set -x ; cd /app \
      && composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/ \
      && composer install \
           --ignore-platform-reqs \
           --no-interaction \
           --no-plugins \
           --no-scripts \
           --prefer-dist
```

7.18.4 整合以上阶段所生成的文件

第三阶段对以上阶段生成的文件进行整合。

```
# 注：php 镜像版本号已包含 major.minor（8.3），生产环境可根据需要调整
FROM php:8.3-fpm-alpine as laravel

ARG LARAVEL_PATH=/app/laravel

COPY --from=composer /app/vendor/ ${LARAVEL_PATH}/vendor/
COPY . ${LARAVEL_PATH}
COPY --from=frontend /app/public/js/ ${LARAVEL_PATH}/public/js/
COPY --from=frontend /app/public/css/ ${LARAVEL_PATH}/public/css/
COPY --from=frontend /app/public/mix-manifest.json ${LARAVEL_PATH}/public/mix-manifest.json

RUN set -x ; cd ${LARAVEL_PATH} \
      && mkdir -p storage \
      && mkdir -p storage/framework/cache \
      && mkdir -p storage/framework/sessions \
      && mkdir -p storage/framework/testing \
      && mkdir -p storage/framework/views \
      && mkdir -p storage/logs \
      && chmod -R 777 storage \
      && php artisan package:discover
```

7.18.5 最后一个阶段构建 NGINX 镜像

```
# 注：nginx 镜像推荐使用具体的版本标签（如 nginx:1.28-alpine）
FROM nginx:1.28-alpine as nginx

ARG LARAVEL_PATH=/app/laravel

COPY laravel.conf /etc/nginx/conf.d/
COPY --from=laravel ${LARAVEL_PATH}/public ${LARAVEL_PATH}/public
```

7.18.6 构建 Laravel 及 Nginx 镜像

使用 `docker build` 命令构建镜像。

```
$ docker build -t my/laravel --target=laravel .

$ docker build -t my/nginx --target=nginx .
```

7.18.7 启动容器并测试

新建 Docker 网络

```
$ docker network create laravel
```

启动 laravel 容器，`--name=laravel` 参数设定的名字必须与 `nginx` 配置文件中的 `fastcgi_pass laravel:9000;` 一致

```
$ docker run -dit --rm --name=laravel --network=laravel my/laravel
```

启动 nginx 容器

```
$ docker run -dit --rm --network=laravel -p 8080:80 my/nginx
```

浏览器访问 `127.0.0.1:8080` 可以看到 Laravel 项目首页。

> 也许 Laravel 项目依赖其他外部服务，例如 redis、MySQL，请自行启动这些服务之后再进行测试，本小节不再赘述。

7.18.8 生产环境优化

本小节内容为了方便测试，将配置文件直接放到了镜像中，实际在使用时 **建议** 将配置文件作为 `config` 或 `secret` 挂载到容器中，请读者自行学习 `Kubernetes` 的相关内容。

由于篇幅所限本小节只是简单列出，更多内容可以参考 khs1994-docker/laravel-demo 项目。

7.18.9 附录

完整的 `Dockerfile` 文件如下。

```
# 注：生产环境推荐使用具体的版本标签，如 node:22-alpine、composer:2.x、php:8.3-fpm-alpine、nginx:1.28-alpin
e
FROM node:22-alpine as frontend

COPY package.json /app/

RUN set -x ; cd /app \
      && npm install --registry=https://registry.npmmirror.com

COPY webpack.mix.js webpack.config.js tailwind.config.js /app/
COPY resources/ /app/resources/

RUN set -x ; cd /app \
      && touch artisan \
      && mkdir -p public \
      && npm run production

FROM composer:2 as composer

COPY database/ /app/database/
COPY composer.json composer.lock /app/

RUN set -x ; cd /app \
      && composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/ \
      && composer install \
           --ignore-platform-reqs \
           --no-interaction \
           --no-plugins \
           --no-scripts \
           --prefer-dist

FROM php:8.3-fpm-alpine as laravel

ARG LARAVEL_PATH=/app/laravel

COPY --from=composer /app/vendor/ ${LARAVEL_PATH}/vendor/
COPY . ${LARAVEL_PATH}
COPY --from=frontend /app/public/js/ ${LARAVEL_PATH}/public/js/
COPY --from=frontend /app/public/css/ ${LARAVEL_PATH}/public/css/
COPY --from=frontend /app/public/mix-manifest.json ${LARAVEL_PATH}/public/mix-manifest.json

RUN set -x ; cd ${LARAVEL_PATH} \
      && mkdir -p storage \
      && mkdir -p storage/framework/cache \
      && mkdir -p storage/framework/sessions \
      && mkdir -p storage/framework/testing \
      && mkdir -p storage/framework/views \
      && mkdir -p storage/logs \
      && chmod -R 777 storage \
      && php artisan package:discover

FROM nginx:1.28-alpine as nginx

ARG LARAVEL_PATH=/app/laravel

COPY laravel.conf /etc/nginx/conf.d/
COPY --from=laravel ${LARAVEL_PATH}/public ${LARAVEL_PATH}/public
```

本章小结

本章详细介绍了 Dockerfile 的所有核心指令，以下是各指令要点的速查表。

指令	作用	关键要点
FROM	指定基础镜像	必须是第一条指令
RUN	在新层执行命令	合并命令、清理缓存以减小体积
COPY	复制文件	优先使用，支持 `--from`
ADD	更高级的复制	自动解压 tar，不推荐用于下载
CMD	容器启动默认命令	可被 `docker run` 参数覆盖
ENTRYPOINT	容器入口点	固定启动命令，CMD 作为默认参数
ENV	设置环境变量	构建时 + 运行时均生效
ARG	构建参数	仅构建时生效，FROM 后需重新声明
VOLUME	定义匿名卷	VOLUME 之后的修改会丢失
EXPOSE	声明端口	仅文档作用，不自动映射
WORKDIR	指定工作目录	替代 `RUN cd`，目录不存在会自动创建
USER	指定运行用户	用户必须已存在，推荐 gosu
HEALTHCHECK	健康检查	支持 starting/healthy/unhealthy 状态
ONBUILD	延迟执行指令	只继承一次，不可级联
LABEL	添加元数据	推荐 OCI 标准标签，替代 MAINTAINER
SHELL	更改默认 shell	推荐 `["/bin/bash", "-o", "pipefail", "-c"]`

Built with mdPress

生产镜像快速检查清单

在将镜像推向生产之前，建议逐条过一遍以下清单：

- ☐ 基础镜像选择了最小化版本（如 `alpine`、`distroless`）
- ☐ 使用了多阶段构建，最终镜像不含编译工具链
- ☐ 以非 root 用户运行（`USER` 指令）
- ☐ `COPY` 优先于 `ADD`，且仅复制必要文件
- ☐ `RUN` 指令合并了 `apt-get update && install && rm -rf /var/lib/apt/lists/*`
- ☐ 设置了 `HEALTHCHECK`
- ☐ 使用了 `.dockerignore` 排除 `.git`、`node_modules` 等无关文件
- ☐ 镜像标签使用了具体版本号或 commit hash，而非 `latest`

> 更完整的编写指南见附录：Dockerfile 最佳实践。

延伸阅读

- 使用 Dockerfile 定制镜像：Dockerfile 入门
- 多阶段构建：优化镜像大小
- Dockerfile 最佳实践：编写指南
- 安全：容器安全实践
- Compose 模板文件：Compose 中的配置

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第八章 数据管理

如图 8-1 所示，Docker 数据管理主要围绕三类挂载方式展开。

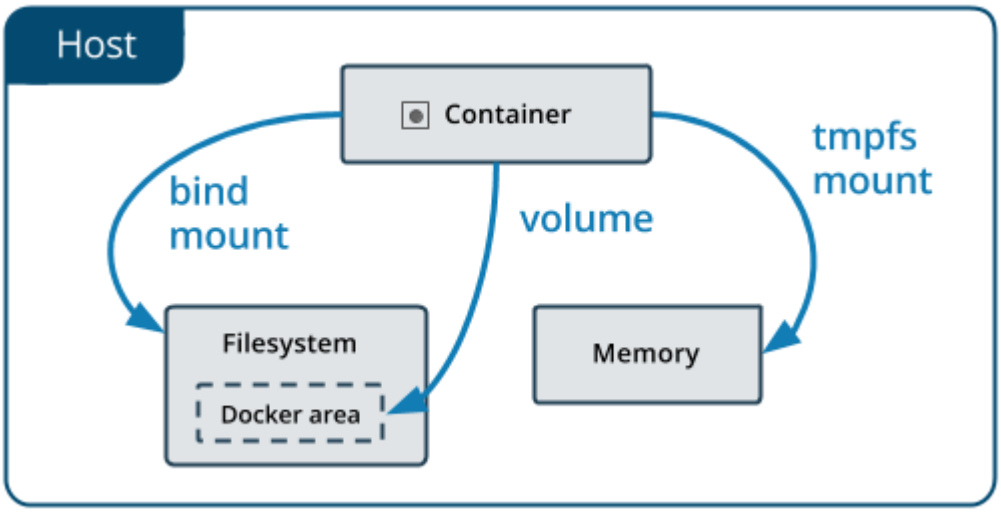


图 8-1：Docker 数据挂载类型示意图

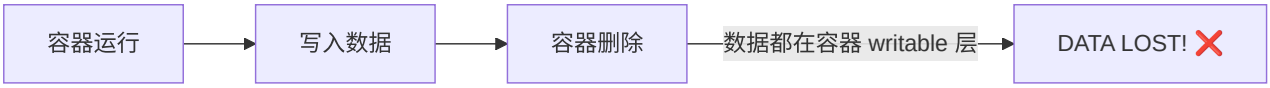
这一章介绍如何在 Docker 内部以及容器之间管理数据，在容器中管理数据主要有以下几种方式：

- 数据卷
- 挂载主机目录
- tmpfs 挂载

8.1 数据卷

8.1.1 为什么需要数据卷

容器的存储层有一个关键问题：**容器删除后，数据就没了**。



数据卷 (Volume) 解决了这个问题，它的生命周期独立于容器。

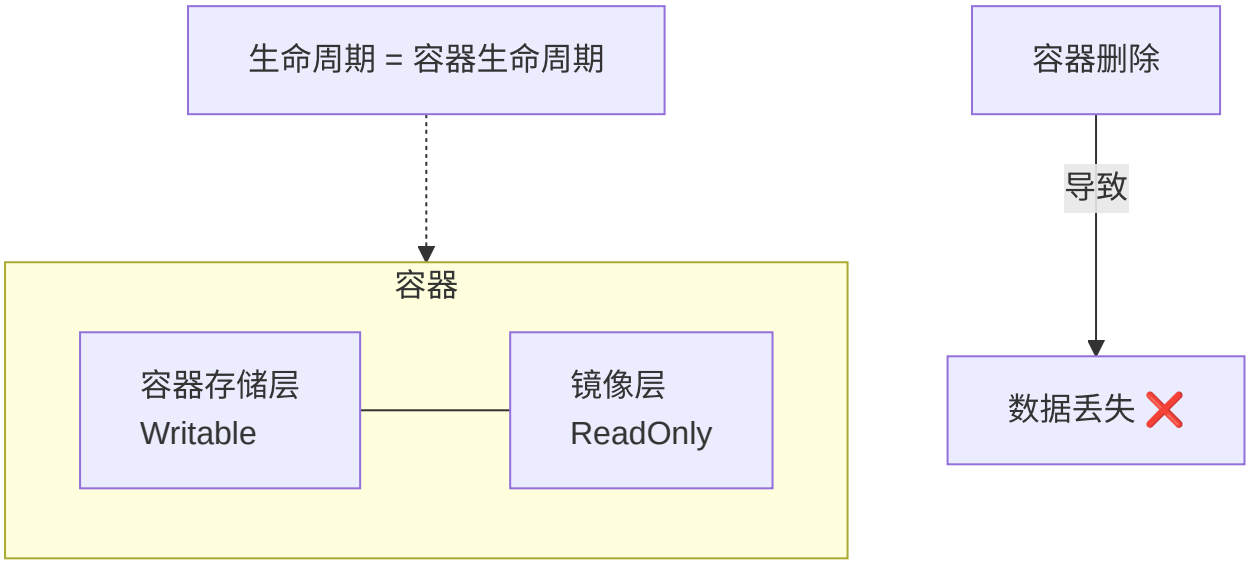
8.1.2 数据卷的特性

特性	说明
持久化	容器删除后数据仍然保留
共享	多个容器可以挂载同一个数据卷
即时生效	对数据卷的修改立即可见
不影响镜像	数据卷中的数据不会打包进镜像
性能更好	绕过 UnionFS，直接读写

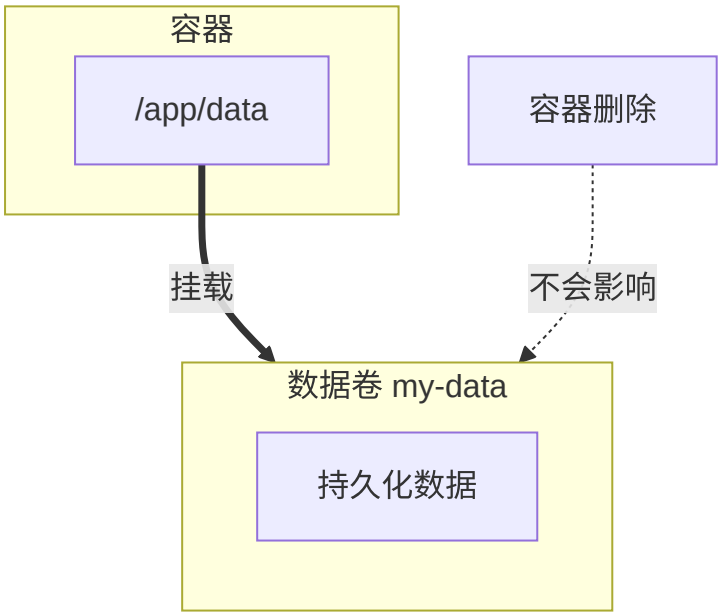
Built with mdPress

8.1.3 数据卷 vs 容器存储层

容器存储层：不推荐存储重要数据



数据卷：推荐



8.1.4 数据卷基本操作

创建数据卷

```
$ docker volume create my-vol
```

列出所有数据卷

```
$ docker volume ls
DRIVER    VOLUME NAME
local     my-vol
local     postgres_data
local     redis_data
```

查看数据卷详情

```
$ docker volume inspect my-vol
[
    {
        "CreatedAt": "2026-01-15T10:00:00Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

关键字段：

- `Mountpoint`：数据卷在宿主机上的实际存储位置
- `Driver`：存储驱动 (默认 local，也可以用第三方驱动)

8.1.5 挂载数据卷

方式一：--mount：推荐

```
$ docker run -d \
    --name web \
    --mount source=my-vol,target=/usr/share/nginx/html \
    nginx
```

参数说明：

参数	说明
`source`	数据卷名称 (不存在会自动创建)
`target`	容器内挂载路径
`readonly`	可选，只读挂载

方式二：-v：简写

```
$ docker run -d \
    --name web \
    -v my-vol:/usr/share/nginx/html \
    nginx
```

格式： `-v 数据卷名:容器路径[:选项]`

两种方式对比

特性	--mount	-v
语法	键值对，更清晰	冒号分隔，更简洁
数据卷 (Volume) 挂载行为	卷不存在会自动创建，与 `-v` 结果一致	卷不存在会自动创建
绑定挂载 (Bind Mount) 行为	⭐**宿主机路径不存在会报错**，不会自动创建	宿主机路径不存在会 **自动创建为目录**
推荐程度	✅ 推荐 (更明确安全，避免误创建)	常用 (更简洁)

> **提示**：官方更推荐使用 `--mount`。除了语法格式可读性更好之外，最重要的行为差异发生在 **绑定挂载 (Bind Mount)** 时：如果挂载的宿主机源路径尚未存在，`-v` 会擅自将其自动创建为一个空目录；而 `--mount` 则会严格检查并直接报错。这能有效避免因路径拼写错误而在宿主机上留下垃圾目录（以及导致的容器访问空目录问题）。而对于本节的 **数据卷 (Volume)** 挂载而言，两者在目标指定的卷不存在时皆会自动创建卷，产生的结果是 **完全一致** 的。

Built with mdPress

只读挂载

```
## --mount 方式

$ docker run -d \
    --mount source=my-vol,target=/data,readonly \
    nginx

## -v 方式

$ docker run -d \
    -v my-vol:/data:ro \
    nginx
```

8.1.6 使用场景示例

场景一：数据库持久化

```
## 创建数据卷

$ docker volume create postgres_data

## 启动 PostgreSQL，数据存储在数据卷中

$ docker run -d \
    --name postgres \
    -e POSTGRES_PASSWORD=secret \
    -v postgres_data:/var/lib/postgresql/data \
    postgres:16  # 确保与环境兼容的 PostgreSQL 版本

## 即使删除容器，数据仍然保留

$ docker rm -f postgres

## 重新启动，数据还在

$ docker run -d \
    --name postgres \
    -e POSTGRES_PASSWORD=secret \
    -v postgres_data:/var/lib/postgresql/data \
    postgres:16  # 确保与环境兼容的 PostgreSQL 版本
```

场景二：多容器共享数据

```
## 创建共享数据卷

$ docker volume create shared-data

## 容器 A 写入数据

$ docker run -d --name writer \
    -v shared-data:/data \
    alpine sh -c "while true; do date >> /data/log.txt; sleep 5; done"

## 容器 B 读取数据

$ docker run --rm \
    -v shared-data:/data \
    alpine cat /data/log.txt
```

场景三：配置文件持久化

```
## 将 nginx 配置存储在数据卷中

$ docker run -d \
    -v nginx-config:/etc/nginx/conf.d \
    -v nginx-logs:/var/log/nginx \
    -p 80:80 \
    nginx
```

8.1.7 数据卷管理

删除数据卷

```
## 删除指定数据卷

$ docker volume rm my-vol

## 删除容器时同时删除数据卷

$ docker rm -v container_name
```

清理未使用的数据卷

```
## 查看未被任何容器使用的数据卷

$ docker volume ls -f dangling=true

## 删除所有未使用的数据卷

$ docker volume prune

## 强制删除（不提示确认）

$ docker volume prune -f
```

> ⚠️ **注意**：数据卷不会自动垃圾回收。长期运行的系统应定期清理无用数据卷。

8.1.8 数据卷备份与恢复

备份数据卷

```
## 使用临时容器挂载数据卷，打包备份

$ docker run --rm \
    -v my-vol:/source:ro \
    -v $(pwd):/backup \
    alpine tar czf /backup/my-vol-backup.tar.gz -C /source .
```

原理：

1. 创建临时容器
2. 挂载要备份的数据卷到 `/source`
3. 挂载当前目录到 `/backup`
4. 使用 tar 打包

恢复数据卷

```
## 创建新数据卷

$ docker volume create my-vol-restored

## 解压备份到新数据卷

$ docker run --rm \
    -v my-vol-restored:/target \
    -v $(pwd):/backup:ro \
    alpine tar xzf /backup/my-vol-backup.tar.gz -C /target
```

备份脚本示例

```
#!/bin/bash

## backup-volume.sh

VOLUME_NAME=$1
BACKUP_DIR=${2:-/backups}
TIMESTAMP=$(date +%Y%m%d_%H%M%S)

docker run --rm \
    -v ${VOLUME_NAME}:/source:ro \
    -v ${BACKUP_DIR}:/backup \
    alpine tar czf /backup/${VOLUME_NAME}_${TIMESTAMP}.tar.gz -C /source .

echo "Backed up ${VOLUME_NAME} to ${BACKUP_DIR}/${VOLUME_NAME}_${TIMESTAMP}.tar.gz"
```

8.1.9 数据卷 vs 绑定挂载

Docker 有两种主要的数据持久化方式：

特性	数据卷 (Volume)	绑定挂载 (Bind Mount)
管理方式	Docker 管理	用户管理
存储位置	`/var/lib/docker/volumes/`	任意宿主机路径
可移植性	更好	依赖宿主机路径
适用场景	生产数据持久化	开发时同步代码
备份	需要工具	直接访问文件

```
## 数据卷

$ docker run -v mydata:/app/data nginx

## 绑定挂载

$ docker run -v /host/path:/app/data nginx
```

详见绑定挂载章节。

8.1.10 常见问题

Q：如何知道容器使用了哪些数据卷？

```
$ docker inspect container_name --format '{{json .Mounts}}' | jq
```

Q：数据卷的数据在哪里？

```
## 查看数据卷详情

$ docker volume inspect my-vol

## Mountpoint 字段显示实际路径

"Mountpoint": "/var/lib/docker/volumes/my-vol/_data"
```

> ⚠️ **注意**：不建议直接修改 Mountpoint 中的文件，应通过容器操作。

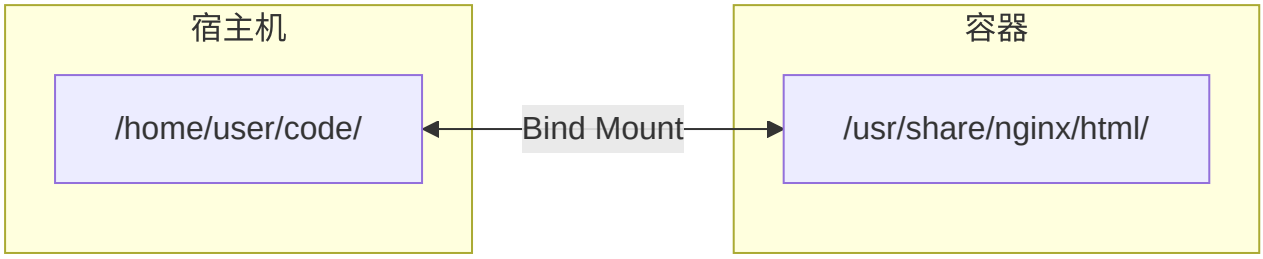
Q：如何在不同机器间迁移数据卷？

1. 在源机器备份：`docker run --rm -v mydata:/data -v $(pwd):/backup alpine tar czf /backup/data.tar.gz -C /data .`
2. 传输 tar.gz 文件
3. 在目标机器恢复

8.2 挂载主机目录

8.2.1 什么是绑定挂载

Bind Mount (绑定挂载) 将 **Docker daemon 所在主机** 上的目录或文件直接挂载到容器中。容器可以读写这台主机上的文件系统。



目录结构（同一份文件）：

```
/home/user/code/ (或 /usr/share/nginx/html/)
├── index.html
├── style.css
└── app.js
```

8.2.2 Bind Mount vs Volume

特性	Bind Mount	Volume
数据位置	宿主机任意路径	Docker 管理的目录
路径指定	必须是绝对路径	卷名
可移植性	依赖宿主机路径	更好 (Docker 管理)
性能	依赖宿主机文件系统	优化的存储驱动
适用场景	开发环境、配置文件	生产数据持久化
备份	直接访问文件	需要通过 Docker

选择建议

需求	推荐方案
开发时同步代码	Bind Mount
持久化数据库数据	Volume
共享配置文件	Bind Mount
容器间共享数据	Volume
备份方便	Bind Mount (直接访问)
生产环境	Volume

8.2.3 基本语法

使用 `--mount`：推荐

```
$ docker run -d \
    --mount type=bind,source=/宿主机路径,target=/容器路径 \
    nginx
```

使用 `-v`：简写

```
$ docker run -d \
    -v /宿主机路径:/容器路径 \
    nginx
```

两种语法对比

特性	--mount	-v
语法	键值对，更清晰	冒号分隔，更简洁
路径不存在时	直接报错 (Fail Fast)	静默自动创建 **目录**
推荐程度	✅ 推荐	常用

> ⚠️ **陷阱**：如果不小心挂载了一个不存在的主机路径，使用 `-v` 会在 **daemon 主机** 上静默创建一个空目录。对于本地 Docker Desktop 用户，这个主机通常就是本机；对于远程 daemon，这个主机就是远程机器。这也是 Docker 官方更推荐使用 `--mount` 的原因：它会直接报错，避免因路径拼写错误而挂错位置。

8.2.4 使用场景

场景一：开发环境代码同步

```
## 将本地代码目录挂载到容器

$ docker run -d \
    -p 8080:80 \
    --mount type=bind,source=$(pwd)/src,target=/usr/share/nginx/html \
    nginx

## 修改本地文件，容器内立即生效（热更新）

$ echo "Hello" > src/index.html

## 浏览器刷新即可看到变化

...
```

场景二：配置文件挂载

```
## 挂载自定义 nginx 配置

$ docker run -d \
    --mount type=bind,source=/path/to/nginx.conf,target=/etc/nginx/nginx.conf,readonly \
    nginx
```

场景三：日志收集

```
## 将容器日志输出到宿主机目录

$ docker run -d \
    --mount type=bind,source=/var/log/myapp,target=/app/logs \
    myapp
```

场景四：共享 SSH 密钥

```
## 挂载 SSH 密钥（只读）

$ docker run --rm -it \
    --mount type=bind,source=$HOME/.ssh,target=/root/.ssh,readonly \
    alpine ssh user@remote
```

8.2.5 只读挂载

防止容器修改宿主机文件：

```
## --mount 语法

$ docker run -d \
    --mount type=bind,source=/config,target=/app/config,readonly \
    myapp

## -v 语法

$ docker run -d \
    -v /config:/app/config:ro \
    myapp
```

容器内尝试写入会报错：

```
$ touch /app/config/new.txt
touch: /app/config/new.txt: Read-only file system
```

8.2.6 挂载单个文件

```
## 挂载 bash 历史记录

$ docker run --rm -it \
    --mount type=bind,source=$HOME/.bash_history,target=/root/.bash_history \
    ubuntu bash

## 挂载自定义配置文件

$ docker run -d \
    --mount type=bind,source=/path/to/my.cnf,target=/etc/mysql/my.cnf \
    mysql
```

> ⚠️ **注意**：挂载单个文件时，如果宿主机上的文件被编辑器替换 (而非原地修改)，容器内仍是旧文件的 inode。建议重启容器或挂载目录。

8.2.7 查看挂载信息

```
$ docker inspect mycontainer --format '{{json .Mounts}}' | jq
```

输出：

```
[
  {
    "Type": "bind",
    "Source": "/home/user/code",
    "Destination": "/app",
    "Mode": "",
    "RW": true,
    "Propagation": "rprivate"
  }
]
```

字段	说明
`Type`	挂载类型 (bind)
`Source`	宿主机路径
`Destination`	容器内路径
`RW`	是否可读写
`Propagation`	挂载传播模式

8.2.8 常见问题

Q：路径不存在报错

```
$ docker run --mount type=bind,source=/not/exist,target=/app nginx
docker: Error response from daemon: invalid mount config for type "bind":
bind source path does not exist: /not/exist
```

解决：确保源路径存在。若你确实需要自动创建目录，可改用 `-v`，但要先确认创建位置就是你想要的主机路径。

Q：权限问题

容器内用户可能无权访问挂载的文件：

```
## 方法1：确保宿主机文件权限允许容器用户访问

$ chmod -R 755 /path/to/data

## 方法2：以 root 运行容器

$ docker run -u root ...

## 方法3：使用相同的 UID

$ docker run -u $(id -u):$(id -g) ...
```

Q：macOS/Windows 性能问题

在 Docker Desktop 上，Bind Mount 性能通常不如 Volume，因为数据需要在宿主机文件系统和 Linux VM 之间同步：

```
## 使用 :cached 或 :delegated 提高性能（macOS，仅 Docker Desktop 4.5 及更早版本）

$ docker run -v /host/path:/container/path:cached myapp
```

选项	说明
`:cached`	宿主机权威，容器读取可能延迟
`:delegated`	容器权威，宿主机读取可能延迟
`:consistent`	默认，完全一致 (最慢)

> **注意**：Docker Desktop 4.6+ 默认使用 VirtioFS 文件共享引擎，上述 `:cached`/`:delegated` 选项已被静默忽略。如需优化文件同步性能，请参考 Docker Desktop 的 Synchronized file shares 功能。

8.2.9 最佳实践

1. 开发环境使用 Bind Mount

```
## 代码热更新

$ docker run -v $(pwd):/app -p 3000:3000 node npm run dev
```

2. 生产环境使用 Volume

```
## 数据持久化

$ docker run -v mysql_data:/var/lib/mysql mysql
```

3. 配置文件使用只读挂载

```
$ docker run -v /config/nginx.conf:/etc/nginx/nginx.conf:ro nginx
```

4. 注意路径安全

```
## ❌ 危险：挂载根目录或敏感目录

$ docker run -v /:/host ...

## ✅ 只挂载必要的目录

$ docker run -v /app/data:/data ...
```

8.3 tmpfs 挂载

`tmpfs` 挂载会把数据放在内存中，而不是写入容器可写层或数据卷。它只适用于 Linux 语义的容器环境，适合需要快速读写但不要求持久化的数据。

8.3.1 适用场景

- 临时缓存
- 会话数据
- 不希望落盘的敏感中间文件

8.3.2 基本用法

使用 `--mount` 语法（推荐）：

```
$ docker run --mount type=tmpfs,destination=/run,tmpfs-size=67108864,tmpfs-mode=1770 nginx
```

也可以使用 `--tmpfs` 简写语法：

```
$ docker run --tmpfs /run:size=64m nginx
```

> **注意：** `--tmpfs` 更适合简单场景；如果你希望显式描述挂载点、大小和权限，`--mount type=tmpfs,...` 的可读性更好，也更便于后续维护。

8.3.3 注意事项

- 容器停止后，`tmpfs` 数据会丢失。
- `tmpfs` 占用宿主机内存，建议显式限制大小。
- 不适合需要持久化的数据。
- `tmpfs` 不适合多个容器共享同一份数据，也不适合当作跨重启的缓存层。
- 在内存压力较高时，部分数据可能受系统交换机制影响，因此不要把 `tmpfs` 当作绝对不会落盘的安全边界。

8.3.4 与 Volume / Bind Mount 对比

类型	数据位置	持久化	典型用途
Volume	Docker 管理目录	是	数据库、长期业务数据
Bind Mount	宿主机指定目录	是	开发联调、配置文件共享
tmpfs	内存	否	高速临时数据、敏感临时文件

Built with mdPress

本章小结

本章介绍了 Docker 的三种数据管理方式：数据卷 (Volume)、绑定挂载 (Bind Mount) 和 tmpfs 挂载。

方式	特点	适用场景
数据卷 (Volume)	Docker 管理，生命周期独立于容器	数据库、应用数据（推荐生产环境）
绑定挂载 (Bind Mount)	挂载宿主机目录，更灵活	开发环境、配置文件、日志
tmpfs 挂载	仅存储在内存中，容器停止即消失	临时敏感数据、高速缓存

操作	命令
创建数据卷	`docker volume create name`
列出数据卷	`docker volume ls`
查看详情	`docker volume inspect name`
删除数据卷	`docker volume rm name`
清理未用	`docker volume prune`
挂载数据卷	`-v name:/path` 或 `--mount source=name,target=/path`

延伸阅读

- 数据卷：Docker 管理的持久化存储
- 绑定挂载：挂载宿主机目录
- tmpfs 挂载：内存中的临时存储
- 存储驱动：Docker 存储的底层原理
- Compose 数据管理：Compose 中的挂载配置

Built with mdPress

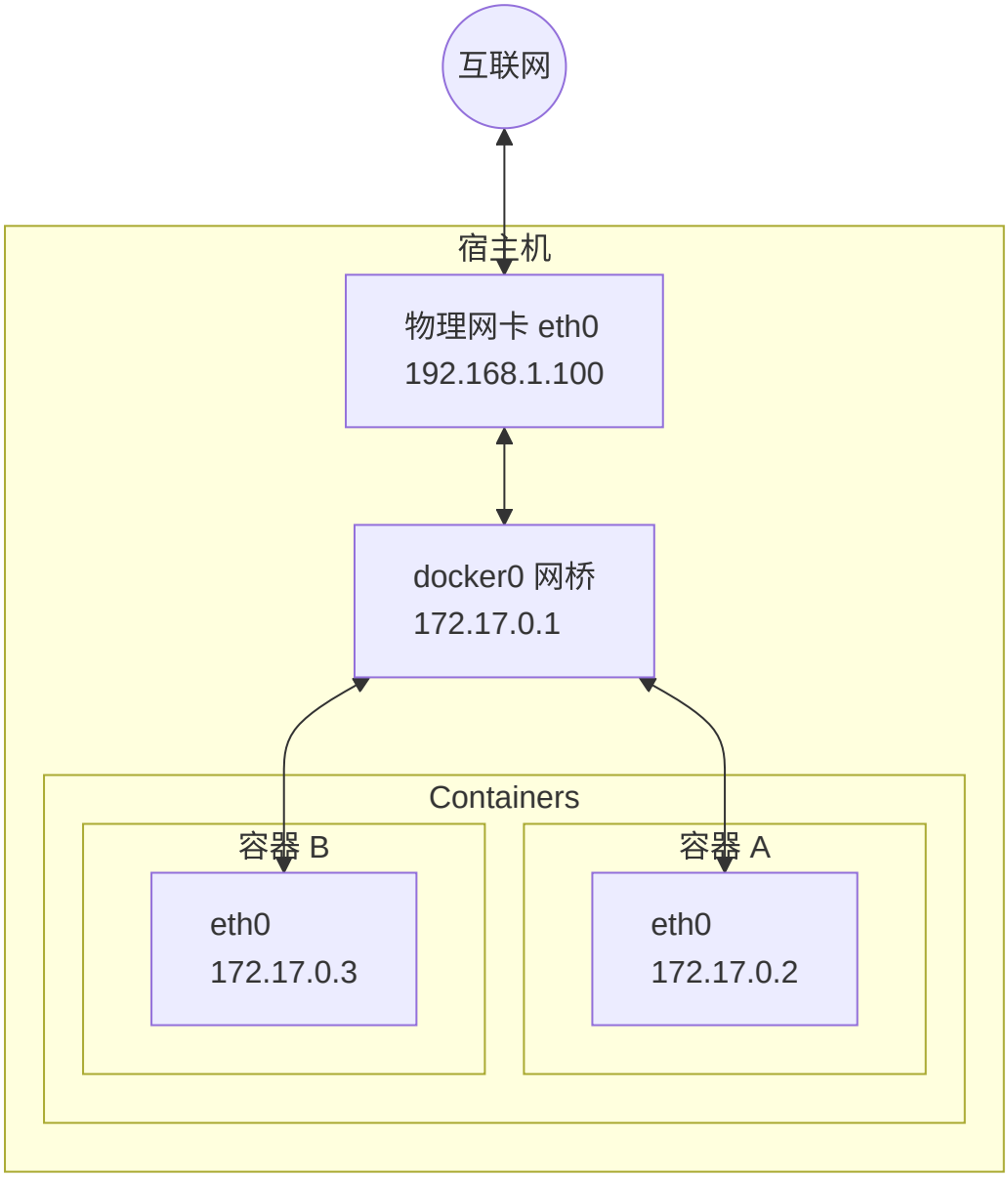
> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第九章 网络配置

Docker 容器需要网络来与外部世界通信、容器之间相互通信以及与宿主机通信。Docker 在安装时会自动配置网络基础设施，大多数情况下开箱即用。

概述

Docker 启动时自动创建以下网络组件：



本章将详细介绍 Docker 网络配置的各个方面。

Built with mdPress

本章内容

- 配置 DNS
- 网络类型
- 自定义网络
- 容器互联
- 外部访问容器
- 网络隔离
- 高级网络配置

9.1 配置 DNS

Docker 1.10.0 以后，内建了一个 DNS 服务器，使得容器可以直接通过容器名称通信。方法很简单，只要在创建容器时使用 `--name` 为容器命名即可。

但是使用 Docker DNS 有个前提条件，就是它只能在 **自定义网络** 中使用。也就是说，如果使用的是默认的 `bridge` 网络，是无法使用 DNS 的，所以我们就需要自定义网络。

9.1.1 容器的 DNS 机制

Docker 容器的 DNS 配置有两种情况：

1. **默认 Bridge 网络**：继承宿主机的 DNS 配置 (`/etc/resolv.conf`)。
2. **自定义网络**(推荐)：使用 Docker 嵌入式 DNS 服务器 (Embedded DNS)，支持通过 **容器名** 进行服务发现。

9.1.2 嵌入式 DNS

这是 Docker 网络最强大的功能之一。在自定义网络中，容器可以通过 “名字” 找到彼此，而不需要知道对方的 IP (因为 IP 可能会变)。

```
## 1. 创建自定义网络

$ docker network create mynet

## 2. 启动容器 web 并加入网络

$ docker run -d --name web --network mynet nginx

## 3. 启动容器 client 并尝试 ping web

$ docker run -it --rm --network mynet alpine ping web
PING web (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.074 ms
```

原理： Docker 守护进程在 `127.0.0.11` 运行了一个 DNS 服务器。容器内的 DNS 请求会被转发到这里。如果是容器名，解析为容器 IP；如果是外部域名 (如 google.com)，转发给上游 DNS。

9.1.3 配置 DNS 参数

如果你需要手动配置容器的 DNS (例如使用内网 DNS 服务器)，可以在 `docker run` 中使用以下参数：

1. --dns

指定 DNS 服务器 IP。

```
$ docker run -it --dns=114.114.114.114 ubuntu cat /etc/resolv.conf
nameserver 114.114.114.114
```

2. --dns-search

指定 DNS 搜索域。例如设置为 `example.com`，则 `ping host` 会尝试解析 `host.example.com`。

```
$ docker run --dns-search=example.com myapp
```

3. --hostname 与 -h

设置容器的主机名。

```
$ docker run -h myweb nginx
```

9.1.4 全局 DNS 配置

如果希望所有容器都使用特定的 DNS 服务器（而不是继承宿主机），可以修改 `/etc/docker/daemon.json`：

```
{
  "dns": [
    "114.114.114.114",
    "8.8.8.8"
  ]
}
```

修改后需要重启 Docker 服务：`systemctl restart docker`。

9.1.5 常见问题

以下是使用容器 DNS 时常见的问题及解决方法：

Q：容器无法解析域名

现象：`ping www.baidu.com` 失败，但 `ping 8.8.8.8` 成功。**解决**：

1. 宿主机的 `/etc/resolv.conf` 可能有问题 (例如使用了本地回环地址 127.0.0.53，特别是 Ubuntu 系统)。Docker 可能会尝试修复，但有时会失败。
2. 尝试手动指定 DNS：`docker run --dns 8.8.8.8 ...`
3. 检查防火墙是否拦截了 UDP 53 端口。

Q：无法通过容器名通信

现象：`ping db` 提示 `bad address 'db'`。**原因**：

- 你可能在使用 **默认的 bridge 网络**。默认 bridge 网络 **不支持** 通过容器名进行 DNS 解析 (这是一个历史遗留设计)。
- **解决**：使用自定义网络 (`docker network create ...`)。

Built with mdPress

9.2 网络类型

Docker 提供了多种网络驱动来满足不同的使用场景。安装 Docker 后，系统会自动创建三个默认网络：

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
abc123...      bridge    bridge    local
def456...      host      host      local
ghi789...      none      null      local
```

9.2.1 网络类型对比

各网络类型的特点和适用场景如下：

网络类型	说明	适用场景
bridge	默认类型，容器连接到虚拟网桥	大多数单机场景
host	容器直接使用宿主机网络栈	需要最高网络性能时
none	禁用网络	完全隔离的容器
overlay	跨主机网络	Docker Swarm 集群
macvlan	容器拥有独立 MAC 地址	需要直接接入物理网络
ipvlan	容器共享父接口 MAC，独享 IP	同网段大量容器、对 MAC 数量受限的网络

9.2.2 Bridge 网络：默认

Bridge 是 Docker 默认使用的网络模式。在原生 Linux Docker Engine 上，Docker 启动时通常会创建 `docker0` 虚拟网桥，未指定网络的容器会连接到这个网桥上。Docker Desktop 运行在虚拟机内，宿主机上不会直接看到 `docker0`，也不能假设宿主机可直接访问每个容器 IP。

核心组件如下：

组件	说明
docker0	虚拟网桥，充当交换机角色
veth pair	虚拟网卡对，一端在容器内，一端连接网桥
容器 eth0	容器内的网卡
IP 地址	自动从 172.17.0.0/16 网段分配

9.2.3 Host 网络

使用 `--network host` 参数启动的容器会直接使用宿主机的网络栈，不再拥有独立的网络命名空间。容器内的端口就是宿主机的端口，无需端口映射。

```
$ docker run -d --network host nginx
```

这种模式下网络性能最高，但容器之间和宿主机之间没有网络隔离。

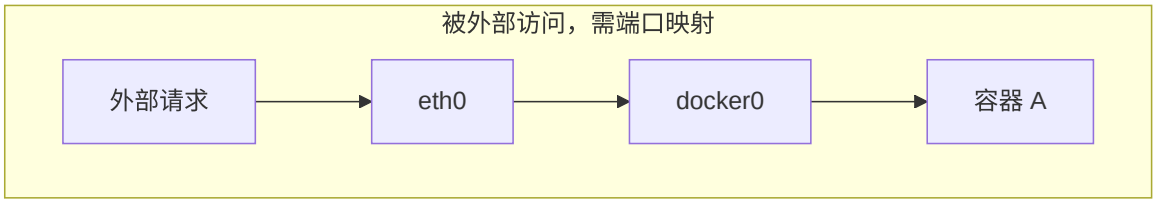
9.2.4 None 网络

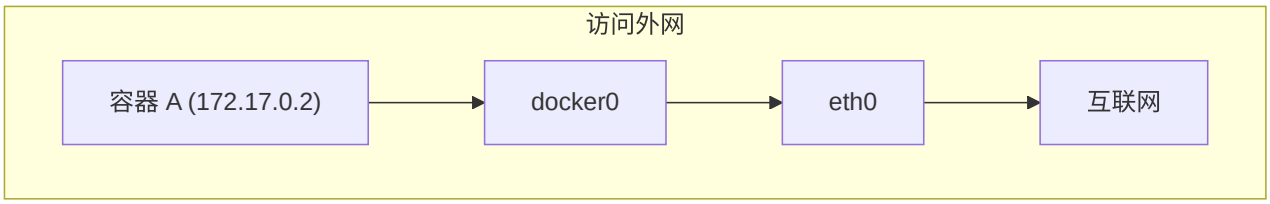
使用 `--network none` 参数启动的容器只有 `lo` 回环网卡，完全没有外部网络连接。适用于只需要运行计算任务、不需要网络的容器。

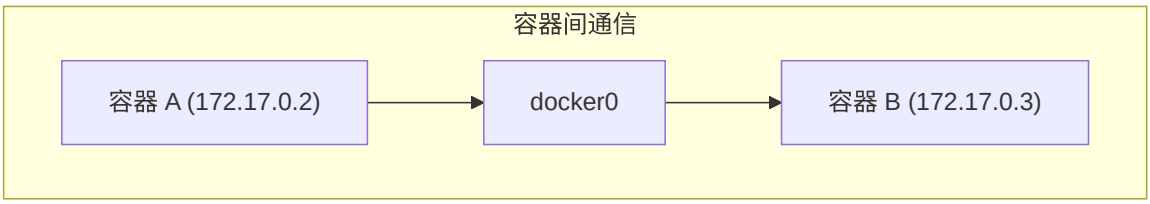
```
$ docker run -it --network none alpine ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> ...
    inet 127.0.0.1/8 scope host lo
```

9.2.5 数据流向

容器网络中的数据流向可以分为以下几种情况：







Built with mdPress

9.3 自定义网络

在生产环境中，推荐使用用户自定义网络代替默认的 bridge 网络。自定义网络提供了更好的隔离性和服务发现能力。

9.3.1 为什么要用自定义网络

默认 bridge 网络存在以下局限，而自定义网络可以很好地解决这些问题：

问题	自定义网络的优势
只能用 IP 通信	支持容器名 DNS 解析
所有容器在同一网络	更好的隔离性
需要 --link (已废弃)	原生支持服务发现

9.3.2 创建自定义网络

使用 `docker network create` 命令可以创建自定义网络：

```
## 创建网络

$ docker network create mynet

## 查看网络详情

$ docker network inspect mynet
```

9.3.3 使用自定义网络

启动容器时通过 `--network` 参数指定连接的网络：

```
## 启动容器并连接到自定义网络

$ docker run -d --name web --network mynet nginx
$ docker run -d --name db --network mynet postgres

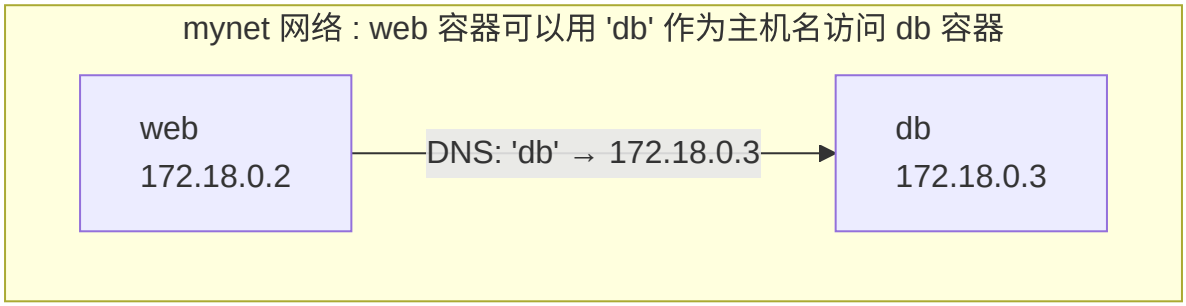
## 在 web 容器中可以直接用容器名访问 db

$ docker exec web ping db
PING db (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.083 ms
```

9.3.4 容器名 DNS 解析

自定义网络自动提供 DNS 服务。Docker 守护进程在 `127.0.0.11` 运行了一个嵌入式 DNS 服务器，容器内的 DNS 请求会被转发到这里：

- 如果是容器名，解析为容器 IP
- 如果是外部域名 (如 google.com)，转发给上游 DNS



9.3.5 常用网络命令

以下是 Docker 网络管理中常用的命令：

```
## 列出网络

$ docker network ls

## 创建网络

$ docker network create mynet

## 查看网络详情

$ docker network inspect mynet

## 连接容器到网络

$ docker network connect mynet mycontainer

## 断开网络连接

$ docker network disconnect mynet mycontainer

## 删除网络

$ docker network rm mynet

## 清理未使用的网络

$ docker network prune
```

Built with mdPress

🔥 踩坑实录

一个新手开发者通过 `docker compose` 部署了两个容器化服务：服务 A 和服务 B。他在服务 A 的代码中尝试用 `localhost:3000` 访问服务 B，结果始终连接超时。这个错误非常隐蔽——在本地单机开发时看不出问题，因为他可能在同一个进程中测试。排查时他错误地认为是防火墙或网络配置问题。实际原因是：每个容器都有独立的网络命名空间，`localhost` 在容器内部只指向容器自己，不是宿主机也不是其他容器。正确的做法是使用 Compose 自动创建的服务名作为主机名：`http://service-b:3000`。Compose 会自动在网络中注册服务名的 DNS，这样容器间通信才能正确解析。改动仅需一行代码，问题随之消失。

9.4 容器互联

容器之间的网络通信是 Docker 网络的核心功能之一。本节介绍容器互联的几种方式。

9.4.1 同一网络内的容器

同一自定义网络内的容器可以直接通过容器名通信，这是推荐的容器互联方式：

```
## 创建网络

$ docker network create app-net

## 启动应用和数据库

$ docker run -d --name redis --network app-net redis
$ docker run -d --name app --network app-net myapp

## app 容器中可以用 redis:6379 连接 Redis

...
```

9.4.2 连接到多个网络

一个容器可以同时连接到多个网络，这对于需要跨网络通信的中间件容器特别有用：

```
## 启动容器

$ docker run -d --name multi-net-container --network frontend nginx

## 再连接到另一个网络

$ docker network connect backend multi-net-container

## 查看容器的网络

$ docker inspect multi-net-container --format '{{json .NetworkSettings.Networks}}'
```

9.4.3 ⚠️ --link 已废弃

`--link` 是 Docker 早期用于容器互联的方式，**已经被废弃**，不建议在新项目中使用。请使用自定义网络替代：

```
## 旧方式（不推荐）

$ docker run --link db:database myapp

## 新方式（推荐）

$ docker network create mynet
$ docker run --network mynet --name db postgres
$ docker run --network mynet --name app myapp
```

使用自定义网络的优势在于：

- 原生支持 DNS 解析
- 不需要在容器启动时显式声明依赖
- 更灵活，可以动态 connect/disconnect

9.5 外部访问容器

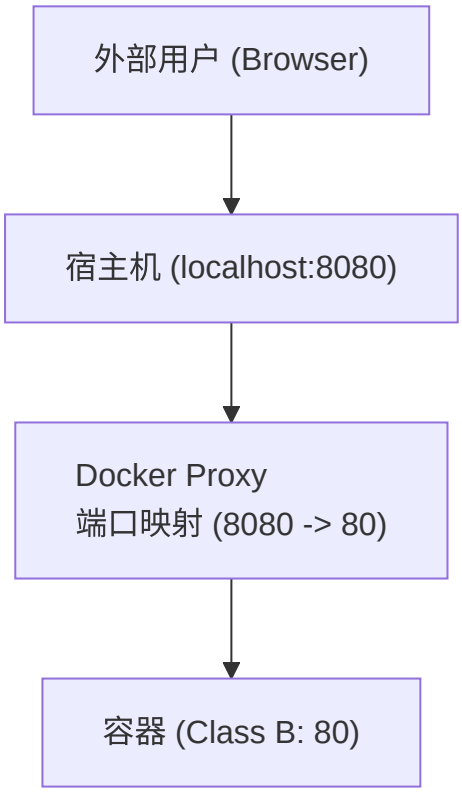
容器运行在自己的隔离网络环境中 (通常是 Bridge 模式)。为了让外部网络访问容器内的服务，我们需要将容器的端口映射到宿主机的端口。

9.5.1 为什么要映射端口

容器的网络访问规则如下：

- **容器之间**：可以通过 IP 或容器名 (自定义网络) 互通。
- **宿主机访问容器**：原生 Linux 环境下可按网络配置访问容器 IP；Docker Desktop 上不能依赖容器 IP，应通过端口映射、容器名（容器间）或 `host.docker.internal` 等机制访问。
- **外部网络访问容器**：❌ 默认无法直接访问。

为了让外部 (如你的浏览器、其他局域网机器) 访问容器内的服务，我们需要将容器的端口 **映射** 到宿主机的端口。



9.5.2 端口映射方式

Docker 提供了多种方式来指定端口映射。

1. 指定映射

使用 `-p <宿主机端口>:<容器端口>` 格式：

```
## 将宿主机的 8080 端口映射到容器的 80 端口

$ docker run -d -p 8080:80 nginx
```

此时访问 `http://localhost:8080` 即可看到 Nginx 页面。

多种格式：

格式	含义	示例
`ip:hostPort:containerPort`	绑定指定 IP 的特定端口	`-p 127.0.0.1:8080:80` (仅本机访问)
`ip::containerPort`	绑定指定 IP 的随机端口	`-p 127.0.0.1::80`
`hostPort:containerPort`	绑定所有 IP (0.0.0.0) 的特定端口	`-p 8080:80` (默认)
`containerPort`	绑定所有 IP 的随机端口	`-p 80`

2. 随机映射

如果不关心宿主机使用哪个端口，可以使用随机映射。使用 `-P` (大写) 参数，Docker 会把 Dockerfile 中 `EXPOSE` 指令暴露的所有端口发布到宿主机的随机高位端口。具体落在哪个端口，取决于宿主机当前可用的临时端口范围。

```
$ docker run -d -P nginx
```

查看映射结果：

```
$ docker ps
CONTAINER ID   PORTS
abc123456      0.0.0.0:49153->80/tcp
```

此时 Nginx 被映射到了宿主机的一个随机高位端口，例如 `49153`。

9.5.3 查看端口映射

可以使用以下命令查看容器的端口映射：

docker port

运行 `docker port` 可以查看到指定容器的端口映射情况：

```
$ docker port mycontainer
80/tcp -> 0.0.0.0:8080
80/tcp -> [::]:8080
```

docker ps

运行 `docker ps` 可以查看到所有容器的端口映射列表：

```
$ docker ps
CONTAINER ID   IMAGE     PORTS                   NAMES
abc123456      nginx     0.0.0.0:8080->80/tcp   web
```

9.5.4 最佳实践与安全

在配置端口映射时，需要注意以下安全事项：

1. 限制监听 IP

默认情况下，`-p 8080:80` 会监听 `0.0.0.0:8080`，这意味着任何人只要能连接你的宿主机 IP，就能访问该服务。

如果不希望对外暴露 (例如数据库服务)，应绑定到 `127.0.0.1`：

```
## 仅允许本机访问

$ docker run -d -p 127.0.0.1:3306:3306 mysql
```

2. 避免端口冲突

如果宿主机 8080 已经被占用了，容器将无法启动。

解决：

- 更换宿主机端口：`-p 8081:80`
- 让 Docker 自动分配：`-p 80`

3. UDP 映射

默认是 TCP 协议。如果要映射 UDP 服务 (如 DNS，Syslog)：

```
$ docker run -d -p 53:53/udp dns-server
```

9.5.5 实现原理

Docker 使用 `docker-proxy` 进程 (用户态) 或 `iptables` DNAT 规则 (内核态) 来实现端口转发。

当流量到达宿主机端口时，iptables 规则将其目标地址修改为容器 IP 并转发：

```
## 简化的 iptables 逻辑

iptables -t nat -A DOCKER -p tcp --dport 8080 -j DNAT --to-destination 172.17.0.2:80
```

这也是为什么你在容器内部看到的访问来源 IP 通常是网关 IP (如 172.17.0.1)，而不是真实的外部 Client IP (除非使用 host 网络模式)。

9.6 网络隔离

Docker 网络提供了天然的隔离能力，不同网络之间的容器默认无法通信。这是 Docker 网络安全的重要基础。

9.6.1 网络隔离原理

不同网络之间默认隔离，容器只能与同一网络中的容器直接通信：

```
## 创建两个网络

$ docker network create frontend
$ docker network create backend

## 容器 A 在 frontend

$ docker run -d --name web --network frontend nginx

## 容器 B 在 backend

$ docker run -d --name db --network backend postgres

## web 无法直接访问 db（不同网络）

$ docker exec web ping db
ping: db: Name or service not known
```

9.6.2 安全优势

这种隔离机制带来以下安全优势：

场景	说明
前后端分离	前端容器无法直接访问数据库网络
微服务隔离	不同微服务组可以使用不同网络
多租户	不同租户的容器在不同网络中完全隔离
最小权限	容器只能访问必要的网络资源

9.6.3 跨网络通信

如果确实需要某个容器跨网络通信，可以将其同时连接到多个网络：

```
## 创建一个中间件容器，连接到两个网络

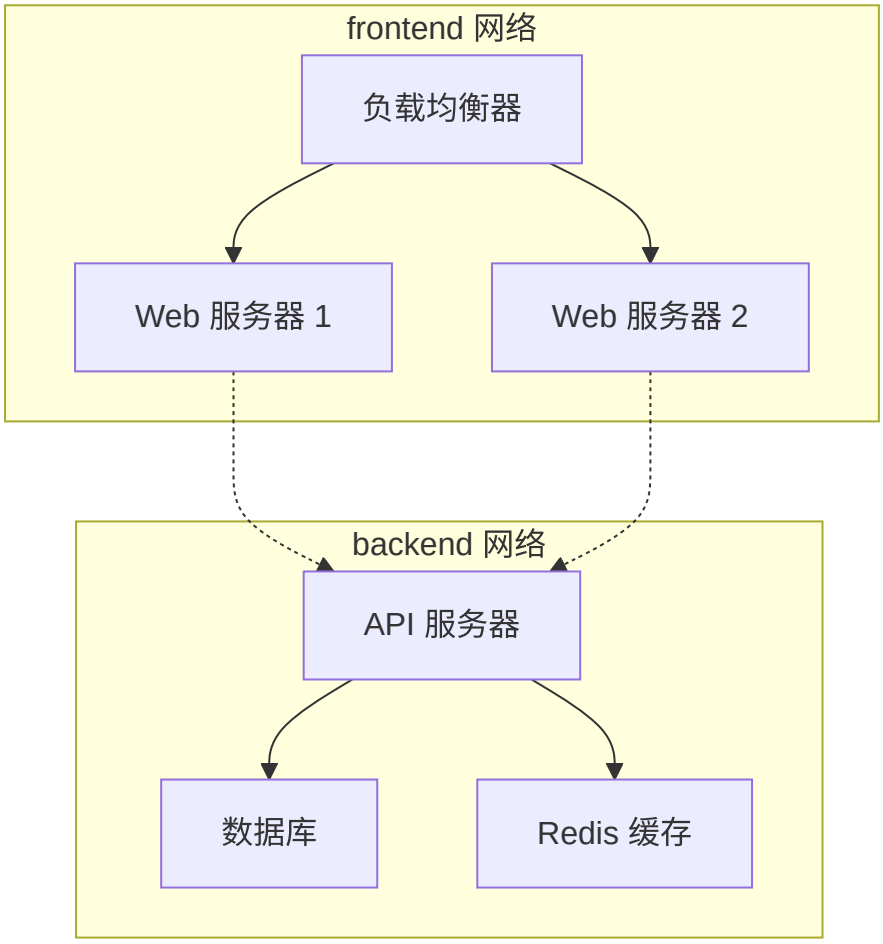
$ docker run -d --name api --network frontend myapi
$ docker network connect backend api

## 现在 api 容器既可以访问 frontend 中的 web，也可以访问 backend 中的 db
```

这种方式让你可以精确控制哪些容器可以跨网络通信，遵循最小权限原则。

9.6.4 典型网络架构

一个典型的多层应用网络架构如下：



在这种架构中，API 服务器同时连接到 `frontend` 和 `backend` 网络，充当两个网络之间的桥梁。负载均衡器和 Web 服务器无法直接访问数据库，增强了安全性。

9.7 容器网络高级特性

本节聚焦 Docker 自身可见的高级网络能力：Overlay 网络、服务发现和 DNS 解析。更偏 Kubernetes 编排网络的概念，如 CNI 和 NetworkPolicy，不在本章展开。

9.7.1 Overlay 网络原理与配置

Overlay 网络在现有网络基础上建立虚拟网络，允许容器跨宿主机通信。它是 Swarm 场景下的核心网络驱动之一。

Overlay 网络工作原理

Overlay 网络通过隧道封装技术（通常是 VXLAN）将容器网络流量封装在宿主机物理网络的 UDP 数据包中传输。Docker overlay 网络默认使用 `4789/udp` 作为数据通道端口，同时 Swarm 控制面与节点通信还需要相应开放 `2377/tcp`、`7946/tcp` 和 `7946/udp`。

```
容器 A (192.168.0.2)
    ↓
veth 对
    ↓
br-net (网桥)
    ↓
Docker 引擎 (VXLAN 封装)
    ↓
物理网络 (172.16.0.0/24)
    ↓
Docker 引擎 (VXLAN 解封装)
    ↓
br-net (网桥)
    ↓
veth 对
    ↓
容器 B (192.168.0.3，不同宿主机)
```

创建和使用 Overlay 网络

Overlay 网络依赖 Swarm。若要让独立容器加入 overlay 网络，需要把网络创建为 `attachable`。

```
# 初始化 Swarm
docker swarm init

# 创建 attachable overlay 网络
docker network create --driver overlay \
  --attachable \
  --subnet 192.168.0.0/24 \
  --opt com.docker.network.driver.mtu=1450 \
  my-overlay-net

# 验证网络创建
docker network ls
docker network inspect my-overlay-net

# 在 Swarm 服务中使用 overlay 网络
docker service create --name web \
  --network my-overlay-net \
  --replicas 3 \
  nginx  # 确保与环境兼容的 Nginx 版本

# 单机环境下也可以让普通容器加入 attachable overlay 网络
docker run -d --name container1 --network my-overlay-net busybox sleep 1d
docker run -d --name container2 --network my-overlay-net busybox sleep 1d

# 测试跨容器通信
docker exec container1 ping -c 1 container2
```

Overlay 网络性能优化

```
# 调整 MTU（Maximum Transmission Unit）避免分片
# VXLAN 开销 50 字节，物理 MTU 1500 时建议设置为 1450
docker network create --driver overlay \
  --attachable \
  --opt com.docker.network.driver.mtu=1450 \
  optimized-overlay

# 启用 IP 地址管理（IPAM）自定义
docker network create --driver overlay \
  --attachable \
  --subnet 10.0.9.0/24 \
  --aux-address "my-router=10.0.9.2" \
  my-custom-overlay
```

9.7.2 容器 DNS 解析机制

Docker 内置 DNS 服务器，但 DNS 解析涉及多个层面的配置。

DNS 解析流程

```
容器应用 (dig www.example.com)
    ↓
容器内 /etc/resolv.conf (127.0.0.11:53)
    ↓
Docker 内嵌 DNS 服务器 (127.0.0.11)
    ↓
用户自定义 DNS 或宿主机 /etc/resolv.conf
    ↓
外部 DNS 服务器 (8.8.8.8 等)
    ↓
DNS 响应 → 容器缓存 → 应用
```

配置容器 DNS

在运行时指定 DNS：

```
# 单个容器
docker run -d \
  --dns 8.8.8.8 \
  --dns 1.1.1.1 \
  --dns-search example.com \
  nginx  # 确保与环境兼容的 Nginx 版本

# DNS 选项
docker run -d \
  --dns-option ndots:2 \
  --dns-option timeout:1 \
  --dns-option attempts:3 \
  nginx  # 确保与环境兼容的 Nginx 版本

# 查看容器 DNS 配置
docker exec <container_id> cat /etc/resolv.conf
```

Docker Compose DNS 配置：

```yaml
services:
  web:
    image: nginx
    dns:
      - 8.8.8.8
      - 1.1.1.1
    dns_search:
      - example.com
      - local

  db:
    image: postgres
    networks:
      - backend
    hostname: postgres-db

networks:
  backend:
    driver: bridge

# 容器内 /etc/resolv.conf 将被自动配置
# search example.com local
# nameserver 8.8.8.8
# nameserver 1.1.1.1
```

Docker 守护进程级别配置：

```
{
  "dns": ["8.8.8.8", "1.1.1.1"],
  "dns-search": ["example.com"]
}
```

自定义服务发现

使用 Docker 内建 DNS 的服务发现：

```
# 创建自定义网络
docker network create mynet

# 运行服务
docker run -d --name web --network mynet nginx  # 确保与环境兼容的 Nginx 版本
docker run -d --name db --network mynet postgres  # 确保与环境兼容的 PostgreSQL 版本

# 在其他容器中通过服务名访问
docker run -it --network mynet busybox sh

# ping web  # 自动解析到 web 容器 IP
# ping db   # 自动解析到 db 容器 IP
```

Compose 服务名自动发现：

```yaml
services:
  frontend:
    image: nginx
    depends_on:
      - backend
    environment:
      BACKEND_URL: http://backend:8080

  backend:
    image: myapp
    depends_on:
      - database

  database:
    image: postgres
    environment:
      POSTGRES_DB: mydb

# frontend 容器可以直接访问 http://backend:8080
# backend 容器可以直接访问 postgres://database:5432
```

9.7.3 本章边界

Docker 的网络章节重点讨论 bridge、host、none、overlay 和 DNS 等能力。CNI、NetworkPolicy、Calico、Cilium 这类概念属于 Kubernetes 编排网络的范畴，后续在 Kubernetes 章节中再展开会更清晰。

本章小结

本章介绍了 Docker 网络配置的各个方面：

概念	要点
DNS 配置	自定义网络支持嵌入式 DNS，可通过容器名解析
网络类型	bridge (默认)、host、none、overlay、macvlan
自定义网络	推荐使用，支持容器名 DNS 解析和更好的隔离
容器互联	同一自定义网络内容器可直接通过容器名通信
端口映射	`-p 宿主机端口:容器端口` 暴露服务到外部
网络隔离	不同网络默认隔离，增强安全性
--link	已废弃，使用自定义网络替代

延伸阅读

- 配置 DNS：自定义 DNS 设置
- 网络类型：Bridge、Host、None 等网络模式
- 自定义网络：创建和管理自定义网络
- 容器互联：容器间通信方式
- 端口映射：高级端口配置
- 网络隔离：网络安全与隔离策略
- EXPOSE 指令：在 Dockerfile 中声明端口
- Compose 网络：Compose 中的网络配置

Built with mdPress

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十章 Docker Buildx

Docker Buildx 是一个 docker CLI 插件，其扩展了 docker 命令，支持 Moby BuildKit 提供的功能。提供了与 docker build 相同的用户体验，并增加了许多新功能。

> Buildx 需要 Docker v23.0+（该版本起 BuildKit 成为默认构建引擎）。推荐使用 Docker v28 及以上版本以获得最完整的 Buildx 功能支持。

本章内容

本章将详细介绍 Docker Buildx 的使用，包括：

- 使用 BuildKit 构建镜像
- 使用 Buildx 构建镜像
- 构建多种系统架构支持的 Docker 镜像

> **供应链安全与存储后端前瞻**：现代软件供应链中，镜像来源证明（Provenance，在 BuildKit 中默认以 `mode=min` 添加）和软件物料清单（SBOM，可通过 `--sbom=true` 显式开启）已经成为极其重要的构建产出。这些 Attestations 数据会作为 manifest 附着在 **镜像索引 (Image Index)** 上。正是基于此诉求，自 Docker Engine 29 起在**新安装场景**默认启用的 `containerd image store` 提供对 Image Index 的完美本地支持能力，解决了传统经典存储后端（Classic Store）无法有效处理带 Attestations 镜像索引的瓶颈。这使得你可以利用 `docker buildx imagetools inspect` 等手段，甚至做到无需拉取完整镜像内容即可在 Registry 或本地高效校验镜像的安全元数据。

10.1 BuildKit

BuildKit 是下一代的镜像构建组件，在 moby/buildkit 开源。

> **重要**：自 Docker 23 起，BuildKit 已成为 **默认稳定构建器**，无需手动启用。Docker Engine 29 进一步将 Containerd 镜像存储设为默认，提升与 Kubernetes 的互操作性。

目前，Docker Hub 自动构建已经支持 BuildKit，具体请参考 docker-practice/docker-hub-buildx。

10.1.1 `Dockerfile` 新增指令详解

BuildKit 引入了多项新指令，旨在优化构建缓存和安全性。以下将详细介绍这些指令的用法。

使用 BuildKit 后，我们可以使用下面几个新的 `Dockerfile` 指令来加快镜像构建。

要使用最新的 Dockerfile 语法特性，建议在 Dockerfile 开头添加语法指令：

```
# syntax=docker/dockerfile:1
```

这将使用最新的稳定版语法解析器，确保你可以使用所有最新特性。

`RUN --mount=type=cache`

目前，几乎所有的程序都会使用依赖管理工具，例如 Go 中的 `go mod`、`Node.js` 中的 `npm` 等等，当我们构建一个镜像时，往往会重复的从互联网中获取依赖包，难以缓存，大大降低了镜像的构建效率。

例如一个前端工程需要用到 `npm`：

```
FROM node:alpine as builder

WORKDIR /app

COPY package.json /app/

RUN npm i --registry=https://registry.npmmirror.com \
        && rm -rf ~/.npm

COPY src /app/src

RUN npm run build

FROM nginx:alpine

COPY --from=builder /app/dist /app/dist
```

使用多阶段构建，构建的镜像中只包含了目标文件夹 `dist`，但仍然存在一些问题，当 `package.json` 文件变动时，`RUN npm i && rm -rf ~/.npm` 这一层会重新执行，变更多次后，生成了大量的中间层镜像。

为解决这个问题，进一步的我们可以设想一个类似 **数据卷** 的功能，在镜像构建时把 `node_modules` 文件夹挂载上去，在构建完成后，这个 `node_modules` 文件夹会自动卸载，实际的镜像中并不包含 `node_modules` 这个文件夹，这样我们就省去了每次获取依赖的时间，大大增加了镜像构建效率，同时也避免了生成了大量的中间层镜像。

`BuildKit` 提供了 `RUN --mount=type=cache` 指令，可以实现上边的设想。

```
# syntax=docker/dockerfile:1

FROM node:alpine as builder

WORKDIR /app

COPY package.json /app/

RUN --mount=type=cache,target=/app/node_modules,id=my_app_npm_module,sharing=locked \
    --mount=type=cache,target=/root/.npm,id=npm_cache \
        npm i --registry=https://registry.npmmirror.com

COPY src /app/src

RUN --mount=type=cache,target=/app/node_modules,id=my_app_npm_module,sharing=locked \

## --mount=type=cache,target=/app/dist,id=my_app_dist,sharing=locked \

        npm run build

FROM nginx:alpine

## COPY --from=builder /app/dist /app/dist

## 为了更直观的说明 from 和 source 指令，这里使用 RUN 指令

RUN --mount=type=cache,target=/tmp/dist,from=builder,source=/app/dist \
    # --mount=type=cache,target=/tmp/dist,from=my_app_dist,sharing=locked \

    mkdir -p /app/dist && cp -r /tmp/dist/* /app/dist
```

第一个 `RUN` 指令执行后，`id` 为 `my_app_npm_module` 的缓存文件夹挂载到了 `/app/node_modules` 文件夹中。多次执行也不会产生多个中间层镜像。

第二个 `RUN` 指令执行时需要用到 `node_modules` 文件夹，`node_modules` 已经挂载，命令也可以正确执行。

第三个 `RUN` 指令将上一阶段产生的文件复制到指定位置，`from` 指明缓存的来源，这里 `builder` 表示缓存来源于构建的第一阶段，`source` 指明缓存来源的文件夹。

上面的 `Dockerfile` 中 `--mount=type=cache,...` 中指令作用如下：

Option	Description
`id`	`id` 设置一个标志，以便区分缓存。
`target` (必填项)	缓存的挂载目标文件夹。
`ro,readonly`	只读，缓存文件夹不能被写入。
`sharing`	有 `shared` `private` `locked` 值可供选择。`sharing` 设置当一个缓存被多次使用时的表现，由于 `BuildKit` 支持并行构建，当多个步骤使用同一缓存时 (同一 `id`) 会发生冲突。`shared` 表示多个步骤可以同时读写，`private` 表示当多个步骤使用同一缓存时，每个步骤使用不同的缓存，`locked` 表示当一个步骤完成释放缓存后，后一个步骤才能继续使用该缓存。
`from`	缓存来源 (构建阶段)，不填写时为空文件夹。
`source`	来源的文件夹路径。

`RUN --mount=type=bind`

该指令可以将一个镜像 (或上一构建阶段) 的文件挂载到指定位置。

```
# syntax=docker/dockerfile:1

RUN --mount=type=bind,from=php:alpine,source=/usr/local/bin/docker-php-entrypoint,target=/docker-php
-entrypoint \
        cat /docker-php-entrypoint
```

`RUN --mount=type=tmpfs`

该指令可以将一个 `tmpfs` 文件系统挂载到指定位置。

```
# syntax=docker/dockerfile:1

RUN --mount=type=tmpfs,target=/temp \
        mount | grep /temp
```

RUN --mount=type=secret

该指令可以将一个文件 (例如密钥) 挂载到指定位置。

```
# syntax=docker/dockerfile:1

RUN --mount=type=secret,id=aws,target=/root/.aws/credentials \
        test -s /root/.aws/credentials && echo "credentials mounted"
```

```
$ docker build -t test --secret id=aws,src=$HOME/.aws/credentials .
```

RUN --mount=type=ssh

该指令可以挂载 ssh 密钥。

```
# syntax=docker/dockerfile:1

FROM alpine
RUN apk add --no-cache openssh-client
RUN mkdir -p -m 0700 ~/.ssh && ssh-keyscan gitlab.com >> ~/.ssh/known_hosts
RUN --mount=type=ssh ssh git@gitlab.com | tee /hello
```

```
$ eval $(ssh-agent)
$ ssh-add ~/.ssh/id_rsa
(Input your passphrase here)
$ docker build -t test --ssh default=$SSH_AUTH_SOCK .
```

10.1.2 使用 `docker compose build` 与 BuildKit

Docker Compose 同样支持 BuildKit，这使得多服务应用的构建更加高效。

自 Docker 23 起，BuildKit 已默认启用，无需额外配置。如果使用旧版本，可设置 `DOCKER_BUILDKIT=1` 环境变量启用。

10.1.3 官方文档

- https://github.com/moby/buildkit/blob/master/frontend/dockerfile/docs/experimental.md

10.2 使用 buildx 构建镜像

10.2.1 使用

Buildx 的使用非常直观，绝大多数情况下可以替代 `docker build` 命令。

你可以直接使用 `docker buildx build` 命令构建镜像。

```
$ docker buildx build .
[+] Building 8.4s (23/32)
 => ...
```

Buildx 使用 BuildKit 引擎进行构建，支持许多新的功能，具体参考 Buildkit 一节。

需要注意的是，默认 `docker` driver 会把构建结果加载到本地镜像存储；使用 `docker-container`、remote、cloud 等 builder 时，如未指定 `--load`、`--push` 或 `--output`，结果通常只保留在构建缓存中。

构建前检查

Buildx 0.15 起支持构建检查：常规构建会默认检查 Dockerfile 与构建参数；如果只想做检查而不真正构建，可以使用 `--check`：

```
$ docker buildx build --check .
```

这适合作为 CI 的快速门禁：普通构建中的检查告警默认不会让构建失败，但 `--check` 发现问题会以非零状态退出；需要把告警提升为错误时，可在 Dockerfile 顶部配合 `# check=error=true`。

使用 bake

`docker buildx bake` 是一个高级构建命令，支持从 HCL、JSON 或 Compose 文件中定义构建目标，实现复杂的流水线构建。

```
## 从 Compose 文件构建所有服务

$ docker buildx bake

## 仅构建指定目标

$ docker buildx bake web
```

生成 SBOM

Buildx 支持在构建时直接生成 SBOM (Software Bill of Materials)，这对于软件供应链安全至关重要。

```
$ docker buildx build --sbom=true -t myimage .
```

该命令会在构建结果中包含 SPDX 或 CycloneDX 格式的 SBOM 数据。

> ⚠️ **注意与失败模式**： 要使 SBOM (或其它 attestation 元数据) 成功附着并可见，对底层的存储格式有前置要求：默认的 classic image store 不支持 manifest list/index 这种存放 attestation 的结构。
>
> 如果只简单运行上述命令，你可能会面临 **“命令成功执行，但本地镜像中看不到 SBOM”** 的体会落差。
>
> **正确的解决路径有两条**：
>
> 1. **推送到远端仓库**：使用 `docker buildx build --sbom=true --push -t myimage:tag` 时，SBOM 会正确保存到远端仓库。远端 OCI 兼容的镜像仓库能够完整存储这些元数据。
> 2. **启用 containerd image store**：在 Docker 守护进程中启用 `containerd image store` 特性（Docker 29+，新安装场景默认启用，Docker Desktop 上也更容易直接使用），可以在本地查看和管理 SBOM 等 attestation 元数据。

10.2.2 官方文档

- Docker buildx 命令文档

10.3 使用 buildx 构建多种系统架构支持的 Docker 镜像

Docker 镜像可以支持多种系统架构，这意味着你可以在 `x86_64`、`arm64` 等不同架构的机器上运行同一个镜像。这是通过一个名为 “manifest list” (或称为 “fat manifest”) 的文件来实现的。

10.3.1 Manifest List 是什么?

为了理解多架构镜像的原理，我们需要先了解 Manifest List 的概念。

Manifest list 是一个包含了多个指向不同架构镜像的 manifest 的文件。当你拉取一个支持多架构的镜像时，Docker 会自动根据你当前的系统架构选择并拉取对应的镜像。

例如，官方的 `hello-world` 镜像就支持多种架构。你可以使用 `docker manifest inspect` 命令来查看它的 manifest list：

```
$ docker manifest inspect hello-world
{
   "schemaVersion": 2,
   "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
   "manifests": [
      {
         "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
         "size": 525,
         "digest": "sha256:80852a401a974d9e923719a948cc5335a0a4435be8778b475844a7153a2382e5",
         "platform": {
            "architecture": "amd64",
            "os": "linux"
         }
      },
      {
         "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
         "size": 525,
         "digest": "sha256:3adea81344be1724b383d501736c3852939b33b3903d02474373700b25e5d6e3",
         "platform": {
            "architecture": "arm",
            "os": "linux",
            "variant": "v5"
         }
      },
      // ... more architectures
   ]
}
```

10.3.2 使用 `docker buildx` 构建多架构镜像

`docker buildx` 是构建多架构镜像的最佳实践工具，它屏蔽了底层的复杂性，提供了一键构建多架构镜像的能力。

在 Docker 19.03+ 版本中，`docker buildx` 是推荐的用于构建多架构镜像的工具。它使用 `BuildKit` 作为后端，可以大大简化构建过程（Docker 23+ 默认启用 BuildKit）。

新建 `builder` 实例

首先，你需要创建一个新的 `builder` 实例，因为它支持同时为多个平台构建。

```
$ docker buildx create --name mybuilder --use
$ docker buildx inspect --bootstrap
```

构建和推送

使用 `docker buildx build` 命令并指定 `--platform` 参数，可以同时构建支持多种架构的镜像。`--push` 参数会将构建好的镜像和 manifest list 推送到 Docker 仓库。

```
## Dockerfile

FROM --platform=$TARGETPLATFORM alpine

RUN uname -a > /os.txt

CMD cat /os.txt
```

```
$ docker buildx build --platform linux/amd64,linux/arm64,linux/arm/v7 -t your-username/multi-arch-im
age . --push
```

构建完成后，你就可以在不同架构的机器上拉取并运行 `your-username/multi-arch-image` 这个镜像了。

架构相关的构建参数

在 `Dockerfile` 中，你可以使用一些预定义的构建参数来根据目标平台定制构建过程：

- `TARGETPLATFORM`：构建镜像的目标平台，例如 `linux/amd64`。
- `TARGETOS`：目标平台的操作系统，例如 `linux`。
- `TARGETARCH`：目标平台的架构，例如 `amd64`。
- `TARGETVARIANT`：目标平台的变种，例如 `v7`。
- `BUILDPLATFORM`：构建环境的平台。
- `BUILDOS`：构建环境的操作系统。
- `BUILDARCH`：构建环境的架构。
- `BUILDVARIANT`：构建环境的变种。

例如，你可以这样编写 `Dockerfile` 来拷贝特定架构的二进制文件：

```
FROM scratch

ARG TARGETOS
ARG TARGETARCH

COPY bin/dist-${TARGETOS}-${TARGETARCH} /dist

ENTRYPOINT ["/dist"]
```

10.3.3 使用 `docker manifest`：底层工具

除了 `docker buildx`，我们也可以直接操作 Manifest List 来手动组合不同架构的镜像。

`docker manifest` 是一个更底层的命令，可以用来创建、检查和推送 manifest list。虽然 `docker buildx` 在大多数情况下更方便，但了解 `docker manifest` 仍然有助于理解其工作原理。

创建 manifest list

```
## 首先，为每个架构构建并推送镜像

$ docker buildx build --platform linux/amd64 -t your-username/my-app:amd64 . --push
$ docker buildx build --platform linux/arm64 -t your-username/my-app:arm64 . --push

## 然后，创建一个 manifest list，将它们组合在一起

$ docker manifest create your-username/my-app:latest \
    --amend your-username/my-app:amd64 \
    --amend your-username/my-app:arm64

## 最后，推送 manifest list

$ docker manifest push your-username/my-app:latest
```

检查 manifest list

你可以使用 `docker manifest inspect` 来查看一个 manifest list 的详细信息，如上文所示。

本章小结

Docker Buildx 是 Docker 构建系统的重要进化，提供了高效、安全且支持多平台的镜像构建能力。

概念	要点
BuildKit	下一代构建引擎，Docker 23+ 默认启用
缓存挂载	`RUN --mount=type=cache` 加速依赖安装
Secret 挂载	`RUN --mount=type=secret` 安全传递密钥
buildx build	替代 `docker build`，支持更多构建功能
构建检查	`--check` 可在不执行构建的情况下检查 Dockerfile 与构建参数
多架构构建	`--platform` 参数一键构建多种架构镜像
Manifest List	多架构镜像的索引文件
SBOM	通过 `--sbom=true` 生成软件物料清单

延伸阅读

- Dockerfile 指令详解：Dockerfile 编写基础
- 多阶段构建：优化镜像体积
- Dockerfile 最佳实践：编写高效 Dockerfile

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

Built with mdPress

第十一章 Docker Compose

`Docker Compose` 是 Docker 官方编排 (Orchestration) 项目之一，负责快速的部署分布式应用。

> ⚠️ **重要提示：Compose V1 已停止支持**
>
> 早期基于 Python 编写的 Compose V1（命令为 `docker-compose`）已于 2023 年中正式停止支持。现已全面升级为基于 Go 编写的 Compose V2，作为 Docker CLI 的官方插件提供（命令为 `docker compose`，中间为空格）。本书强烈推荐且后续章节均以 V2 为核心标准进行讲解。

Docker Compose 解决什么问题?

在学习 Compose 之前，笔者想强调它的真正价值。假设你正在开发一个微服务应用——前端、后端、数据库三个服务。如果你用 Docker 容器分别运行它们，你会遇到这些问题：

1. **启动顺序**：需要先启数据库，再启后端，最后启前端
2. **网络连接**：三个容器需要能彼此通信
3. **卷挂载**：本地代码需要映射到容器内
4. **环境变量**：每个服务的配置需要逐个设置

使用 `docker run` 逐个启动的话，需要记住 3 条复杂的命令。而 **Docker Compose 的核心价值就是用一个 YAML 文件来定义整个应用**，然后一条命令 `docker compose up` 启动所有服务。这是 Compose 被广泛采用的原因——它极大地简化了本地开发和测试的复杂性。

谁应该学 Compose? 任何使用 Docker 进行本地开发的人，以及需要快速部署多容器应用的团队。

本章将介绍 `Compose` 项目情况以及安装和使用。

- 简介
- 安装与卸载
- 使用
- 命令说明
- Compose 模板文件
- 实战 Django
- 实战 Rails
- 实战 WordPress
- 实战 LNMP

11.1 简介

Compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器的快速编排。从功能上看，跟 OpenStack 中的 Heat 十分类似。

其代码目前在 docker/compose 仓库 上开源。

Compose 定位是 “定义和运行多个 Docker 容器的应用 (Defining and running multi-container Docker applications)”，其前身是开源项目 Fig。

通过第一部分中的介绍，我们知道使用一个 Dockerfile 模板文件，可以让用户很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。

Compose 恰好满足了这样的需求。它允许用户通过一个单独的 compose.yaml (历史默认名也常见为 docker-compose.yml) 模板文件 (YAML 格式) 来定义一组相关联的应用容器为一个项目 (project)。

11.1.1 概述

Docker Compose 让用户能够以声明式方式定义和管理多容器应用。它的核心价值在于：用一个 YAML 文件取代一连串手动的 docker run 命令，使得复杂应用的启动、停止和重建变得一键可达。

对于开发团队而言，Compose 解决了三个关键问题：环境一致性（“在我机器上能跑”的问题）、服务依赖管理（确保数据库在应用之前启动）、以及开发-测试-生产的配置差异管理（通过 compose.override.yaml 实现多环境适配）。

11.1.2 模板文件规范

Compose 模板文件采用 YAML 格式，扩展名为 .yml 或 .yaml。

> **注意**：Compose Specification 的顶层 version 字段仅用于向后兼容，当前已标记为 obsolete。新文件建议直接省略该字段。

Docker Compose 默认使用 compose.yaml，也兼容 compose.yml、docker-compose.yaml、docker-compose.yml 等文件名。

Compose 中有两个重要的概念：

- 服务 (`service`)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
- 项目 (`project`)：由一组关联的应用容器组成的一个完整业务单元，在 Compose 文件中定义。

`Compose` 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

`Compose` 项目早期由 Python 编写，称为 Docker Compose V1。

现在的 Docker Compose V2 是一个 Go 语言编写的 Docker CLI 插件（当前版本号已演进至 v5.x 系列，以避免与旧 Compose 文件格式版本混淆）。Docker Desktop 默认包含它；在 Linux 上，也可以将它作为独立的 CLI 插件安装后直接通过 `docker compose` 命令使用。它提供了更快的性能和更好的集成体验。

只要所操作的平台支持 Docker API，就可以在其上利用 `Compose` 来进行编排管理。

Built with mdPress

11.2 安装与卸载

Compose 是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排。

当前的 Compose 以 `docker compose` 子命令的形式提供。Docker Desktop 在 macOS、Windows 和 Linux 上默认包含它；如果你已经在 Linux 上单独安装了 Docker Engine 和 Docker CLI，也可以再安装 Compose CLI 插件。

11.2.1 Linux

在 Linux 上，你可以通过 Docker 官方发布页安装 Compose CLI 插件。把二进制文件保存到 `$DOCKER_CONFIG/cli-plugins/docker-compose`，并赋予执行权限即可。

```
$ DOCKER_CONFIG=${DOCKER_CONFIG:-$HOME/.docker}
$ mkdir -p $DOCKER_CONFIG/cli-plugins
$ curl -SL https://github.com/docker/compose/releases/latest/download/docker-compose-linux-x86_64 -o
$DOCKER_CONFIG/cli-plugins/docker-compose
$ chmod +x $DOCKER_CONFIG/cli-plugins/docker-compose
```

11.2.2 测试安装

```
$ docker compose version
Docker Compose version v5.x.x
```

11.2.3 卸载

如果是二进制包方式安装的，删除二进制文件即可。

```
$ rm $DOCKER_CONFIG/cli-plugins/docker-compose
```

11.3 使用

本节将通过一个具体的 Web 应用案例，介绍 Docker Compose 的基本概念和常用操作。

11.3.1 术语

首先介绍几个术语。

- 服务 (service)：一个应用容器，实际上可以运行多个相同镜像的实例。
- 项目 (project)：由一组关联的应用容器组成的一个完整业务单元。

可见，一个项目可以由多个服务 (容器) 关联而成，Compose 面向项目进行管理。

11.3.2 准备示例

下面创建一个由 web 和 redis 组成的简单示例项目。

应用代码

新建文件夹，在该目录中编写 app.py 文件

```
from flask import Flask
from redis import Redis

app = Flask(__name__)
redis = Redis(host='redis', port=6379)

@app.route('/')
def hello():
    count = redis.incr('hits')
    return 'Hello World! 该页面已被访问 {} 次。\n'.format(count)

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

Dockerfile

编写 Dockerfile 文件，内容为

```
FROM python:3.12-alpine
COPY . /code
WORKDIR /code
RUN pip install redis flask
CMD ["python", "app.py"]
```

compose.yaml

编写 `compose.yaml` 文件，这是 Compose 推荐使用的主模板文件 (也兼容 `docker-compose.yml` 等历史文件名)。

```
services:
  web:
    build: .
    ports:
      - "5000:5000"

  redis:
    image: "redis:alpine"
```

11.3.3 启动与验证

启动项目

```
$ docker compose up
```

此时访问本地 `5000` 端口，每次刷新页面，计数就会加 1。

按下 `Ctrl-C` 停止项目。

后台运行与停止

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

查看日志

```
$ docker compose logs -f
```

进入服务

```
$ docker compose exec redis sh
/data # redis-cli
127.0.0.1:6379> get hits
"9"
```

11.3.4 常用命令

构建与重建

```
$ docker compose build
$ docker compose start
```

运行一次性命令

```
$ docker compose run web python app.py
```

验证与清理

```
$ docker compose config
$ docker compose down
```

> 💡 `docker compose down` 默认会删除容器和网络，但 **保留数据卷**。如需同时删除数据卷，请使用 `docker compose down -v`。

11.4 命令说明

Docker Compose 提供了丰富的命令来管理项目和容器。本节将详细介绍这些命令的使用格式和常用选项。

何时用哪个命令：场景化指南

在学习具体命令前，让我们从使用场景出发，这样可以帮助你更快地找到需要的命令：

项目启动与停止：

- `docker compose up`：第一次启动项目，拉取镜像、创建容器
- `docker compose start`：启动已停止的容器（项目已存在）
- `docker compose stop`：优雅地停止容器（不删除容器）
- `docker compose down`：完全清理，删除容器和网络（开发时常用）

调试与查看：

- `docker compose ps`：查看项目中的容器状态
- `docker compose logs`：查看容器日志（排查问题的第一步）
- `docker compose exec`：进入正在运行的容器执行命令

构建与更新：

- `docker compose build`：重新构建镜像（修改 Dockerfile 后）
- `docker compose pull`：更新所有镜像到最新版本

配置验证：

- `docker compose config`：验证 docker-compose.yml 格式是否正确

11.4.1 命令对象与格式

对于 Compose 来说，大部分命令的对象既可以是项目本身，也可以指定为项目中的服务或者容器。如果没有特别的说明，命令对象将是项目，这意味着项目中所有的服务都会受到命令影响。

执行 `docker compose [COMMAND] --help` 或者 `docker compose help [COMMAND]` 可以查看具体某个命令的使用格式。

`docker compose` 命令的基本的使用格式是

```
docker compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
```

11.4.2 命令选项

- `-f, --file FILE` 指定使用的 Compose 模板文件。默认会自动识别 `compose.yaml` (也兼容 `docker-compose.yml` 等)，并且可以多次指定。
- `-p, --project-name NAME` 指定项目名称，默认将使用所在目录名称作为项目名。
- `--verbose` 输出更多调试信息。(**已弃用**：在 Docker Compose V2 中，请改用 `docker --log-level debug compose ...` 或设置环境变量 `COMPOSE_DEBUG=1`。)
- `-v, --version` 打印版本并退出。

11.4.3 命令使用说明

`build`

格式为 `docker compose build [options] [SERVICE...]`。

构建 (重新构建) 项目中的服务容器。

服务容器一旦构建后，将会带上一个标记名，例如对于 web 项目中的一个 db 容器，可能是 web_db。

可以随时在项目目录下运行 `docker compose build` 来重新构建服务。

选项包括：

- `--force-rm` 删除构建过程中的临时容器。
- `--no-cache` 构建镜像过程中不使用 cache (这将加长构建过程)。
- `--pull` 始终尝试通过 pull 来获取更新版本的镜像。

`config`

验证 Compose 文件格式是否正确，若正确则显示配置，若格式错误显示错误原因。

`down`

此命令将会停止 `up` 命令所启动的容器，并移除网络

exec

进入指定的容器。

help

获得一个命令的帮助。

images

列出 Compose 文件中包含的镜像。

kill

格式为 `docker compose kill [options] [SERVICE...]`。

通过发送 `SIGKILL` 信号来强制停止服务容器。

支持通过 `-s` 参数来指定发送的信号，例如通过如下指令发送 `SIGINT` 信号。

```
$ docker compose kill -s SIGINT
```

logs

格式为 `docker compose logs [options] [SERVICE...]`。

查看服务容器的输出。默认情况下，docker compose 将对不同的服务输出使用不同的颜色来区分。可以通过 `--no-color` 来关闭颜色。

该命令在调试问题的时候十分有用。

pause

格式为 `docker compose pause [SERVICE...]`。

暂停一个服务容器。

port

格式为 `docker compose port [options] SERVICE PRIVATE_PORT`。

打印某个容器端口所映射的公共端口。

选项：

- `--protocol=proto` 指定端口协议，tcp (默认值) 或者 udp。
- `--index=index` 如果同一服务存在多个容器，指定命令对象容器的序号 (默认为 1)。

ps

格式为 `docker compose ps [options] [SERVICE...]`。

列出项目中目前的所有容器。

选项：

- `-q` 只打印容器的 ID 信息。

pull

格式为 `docker compose pull [options] [SERVICE...]`。

拉取服务依赖的镜像。

选项：

- `--ignore-pull-failures` 忽略拉取镜像过程中的错误。

push

推送服务依赖的镜像到 Docker 镜像仓库。

restart

格式为 `docker compose restart [options] [SERVICE...]`。

重启项目中的服务。

选项：

- `-t, --timeout TIMEOUT` 指定重启前停止容器的超时 (默认为 10 秒)。

rm

格式为 `docker compose rm [options] [SERVICE...]`。

删除所有 (停止状态的) 服务容器。推荐先执行 `docker compose stop` 命令来停止容器。

选项：

Built with mdPress

- `-f, --force` 强制直接删除，包括非停止状态的容器。一般尽量不要使用该选项。
- `-v` 删除容器所挂载的数据卷。

run

格式为 `docker compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]`。

在指定服务上执行一个命令。

例如：

```
$ docker compose run ubuntu ping docker.com
```

将会启动一个 ubuntu 服务容器，并执行 `ping docker.com` 命令。

默认情况下，如果存在关联，则所有关联的服务将会自动被启动，除非这些服务已经在运行中。

该命令类似启动容器后运行指定的命令，相关卷、链接等等都将会按照配置自动创建。

两个不同点：

- 给定命令将会覆盖原有的自动运行命令；
- 不会自动创建端口，以避免冲突。

如果不希望自动启动关联的容器，可以使用 `--no-deps` 选项，例如

```
$ docker compose run --no-deps web python manage.py shell
```

将不会启动 web 容器所关联的其它容器。

选项：

- `-d` 后台运行容器。
- `--name NAME` 为容器指定一个名字。
- `--entrypoint CMD` 覆盖默认的容器启动指令。
- `-e KEY=VAL` 设置环境变量值，可多次使用选项来设置多个环境变量。
- `-u, --user=""` 指定运行容器的用户名或者 uid。
- `--no-deps` 不自动启动关联的服务容器。
- `--rm` 运行命令后自动删除容器，`d` 模式下将忽略。
- `-p, --publish=[]` 映射容器端口到本地主机。
- `--service-ports` 配置服务端口并映射到本地主机。
- `-T` 不分配伪 tty，意味着依赖 tty 的指令将无法运行。

scale

在当前 Compose CLI 中，更稳妥的扩缩容写法是通过 `docker compose up --scale` 完成。

例如：

```
$ docker compose up -d --scale web=3 --scale db=2
```

将启动 3 个容器运行 `web` 服务，2 个容器运行 `db` 服务。

> **说明**：有些旧环境或实验性入口中仍可能出现 `docker compose scale`，但它不应再被视为当前 Compose 的通用稳定默认命令。

一般的，当指定数目多于该服务当前实际运行容器，将新创建并启动容器；反之，将停止容器。

常用搭配选项：

- `-d` 后台启动。
- `--scale SERVICE=NUM` 指定服务实例数量，可重复使用。

start

格式为 `docker compose start [SERVICE...]`。

启动已经存在的服务容器。

Built with mdPress

stop

格式为 `docker compose stop [options] [SERVICE...]`。

停止已经处于运行状态的容器，但不删除它。通过 `docker compose start` 可以再次启动这些容器。

选项：

- `-t, --timeout TIMEOUT` 停止容器时候的超时 (默认为 10 秒)。

top

查看各个服务容器内运行的进程。

unpause

格式为 `docker compose unpause [SERVICE...]`。

恢复处于暂停状态中的服务。

up

格式为 `docker compose up [options] [SERVICE...]`。

该命令十分强大，它将尝试自动完成包括构建镜像，(重新) 创建服务，启动服务，并关联服务相关容器的一系列操作。

链接的服务都将会被自动启动，除非已经处于运行状态。

可以说，大部分时候都可以直接通过该命令来启动一个项目。

默认情况，`docker compose up` 启动的容器都在前台，控制台将会同时打印所有容器的输出信息，可以很方便进行调试。

当通过 `Ctrl-C` 停止命令时，所有容器将会停止。

如果使用 `docker compose up -d`，将会在后台启动并运行所有的容器。一般推荐生产环境下使用该选项。

默认情况，如果服务容器已经存在，`docker compose up` 将会尝试停止容器，然后重新创建 (保持使用 `volumes-from` 挂载的卷)，以保证新启动的服务匹配 Compose 文件的最新内容。如果用户不希望容器被停止并重新创建，可以使用 `docker compose up --no-recreate`。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。如果用户只想重新部署某个服务，可以使用 `docker compose`

Built with mdPress

`up --no-deps -d <SERVICE_NAME>` 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。

选项：

- `-d` 在后台运行服务容器。
- `--no-color` 不使用颜色来区分不同的服务的控制台输出。
- `--no-deps` 不启动服务所链接的容器。
- `--force-recreate` 强制重新创建容器，不能与 `--no-recreate` 同时使用。
- `--no-recreate` 如果容器已经存在了，则不重新创建，不能与 `--force-recreate` 同时使用。
- `--no-build` 不自动构建缺失的服务镜像。
- `-t, --timeout TIMEOUT` 停止容器时候的超时 (默认为 10 秒)。

version

格式为 `docker compose version`。

打印版本信息。

watch

格式为 `docker compose watch [options] [SERVICE...]`。

启用开发模式，自动监视源代码并在文件发生变化时刷新服务。这需要项目中有 `compose.yaml` (或 `docker-compose.yml`)，且定义了 `x-develop` 或 `develop` 配置段。

例如：

```
services:
  web:
    build: .
    develop:
      watch:
        - action: sync
          path: ./web
          target: /src/web
          ignore:
            - node_modules/
        - action: rebuild
          path: package.json
```

选项：

- `--no-up` 不自动启动服务。
- `--quiet` 静默模式。

11.5 Compose 模板文件

模板文件是使用 Compose 的核心，涉及到的指令关键字也比较多。但大家不用担心，这里面大部分指令跟 docker run 相关参数的含义都是类似的。

默认的模板文件名称为 compose.yaml (也兼容 docker-compose.yml 等历史文件名)，格式为 YAML。

```
services:
  webapp:
    image: examples/web
    ports:
      - "80:80"
    volumes:
      - "/data"
```

注意每个服务都必须通过 image 指令指定镜像或 build 指令 (需要 Dockerfile) 等来自动构建生成镜像。

如果使用 build 指令，在 Dockerfile 中设置的选项 (例如：CMD、EXPOSE、VOLUME、ENV 等) 将会自动被获取，无需在 Compose 文件中重复设置。

下面分别介绍各个指令的用法。

11.5.1 build

指定 Dockerfile 所在文件夹的路径 (可以是绝对路径，或者相对 Compose 文件的路径)。Compose 将会利用它自动构建这个镜像，然后使用这个镜像。

```
services:
  webapp:
    build: ./dir
```

你也可以使用 context 指令指定 Dockerfile 所在文件夹的路径。

使用 dockerfile 指令指定 Dockerfile 文件名。

使用 arg 指令指定构建镜像时的变量。

```
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
```

使用 cache_from 指定构建镜像的缓存

```
build:
  context: .
  cache_from:
    - alpine:latest
    - corp/web_app:3.14
```

11.5.2 cap_add, cap_drop

指定容器的内核能力 (capacity) 分配。

例如，让容器拥有所有能力可以指定为：

```
cap_add:
  - ALL
```

去掉 NET_ADMIN 能力可以指定为：

```
cap_drop:
  - NET_ADMIN
```

11.5.3 command

覆盖容器启动后默认执行的命令。

```
command: echo "hello world"
```

11.5.4 configs

configs 来自 Compose Specification。它在 Swarm 中是原生对象；在本地 docker compose 模式下通常以文件挂载的形式实现，具体能力取决于 Compose 版本与运行平台。

11.5.5 cgroup_parent

指定父 cgroup 组，意味着将继承该组的资源限制。

例如，创建了一个 cgroup 组名称为 cgroups_1。

```
cgroup_parent: cgroups_1
```

11.5.6 container_name

指定容器名称。默认将会使用 项目名称_服务名称_序号 这样的格式。

```
container_name: docker-web-container
```

> 注意：指定容器名称后，该服务将无法进行扩展 (scale)，因为 Docker 不允许多个容器具有相同的名称。

11.5.7 deploy

deploy 用于描述副本数、更新策略、资源限制等部署参数。该字段在 Swarm 中支持最完整；在本地 docker compose up 场景下通常只有部分字段生效。

11.5.8 devices

指定设备映射关系。

```
devices:
  - "/dev/ttyUSB1:/dev/ttyUSB0"
```

11.5.9 depends_on

解决容器的依赖、启动先后的问题。以下例子中会先启动 redis db 再启动 web

```
services:
  web:
    build: .
    depends_on:
      - db
      - redis

  redis:
    image: redis

  db:
    image: postgres
```

> 注意：上述简写形式中，web 服务不会等待 redis db “完全启动” 之后才启动。

如果需要等待依赖服务就绪，可以使用 condition 字段配合 healthcheck：

```yaml
services:
  web:
    build: .
    depends_on:
      db:
        condition: service_healthy
      redis:
        condition: service_healthy

  redis:
    image: redis
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 5

  db:
    image: postgres
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 3s
      retries: 5
```

`condition` 支持三个值：

- `service_started`：容器启动即满足（默认）。
- `service_healthy`：容器的 `healthcheck` 状态为 healthy 时满足。
- `service_completed_successfully`：容器成功退出（退出码为 0）时满足，适用于初始化任务等一次性容器。

11.5.10 dns

自定义 DNS 服务器。可以是一个值，也可以是一个列表。

```yaml
dns: 8.8.8.8

dns:
  - 8.8.8.8
  - 114.114.114.114
```

11.5.11 dns_search

配置 DNS 搜索域。可以是一个值，也可以是一个列表。

```
dns_search: example.com

dns_search:
  - domain1.example.com
  - domain2.example.com
```

11.5.12 tmpfs

挂载一个 tmpfs 文件系统到容器。

```
tmpfs: /run
tmpfs:
  - /run
  - /tmp
```

11.5.13 env_file

从文件中获取环境变量，可以为单独的文件路径或列表。

如果通过 `docker compose -f FILE` 方式来指定 Compose 模板文件，则 `env_file` 中变量的路径会基于模板文件路径。

如果有变量名称与 `environment` 指令冲突，则按照惯例，以后者为准。

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

环境变量文件中每一行必须符合格式，支持 # 开头的注释行。

```
## common.env: Set development environment

PROG_ENV=development
```

11.5.14 environment

设置环境变量。你可以使用数组或字典两种格式。

只给定名称的变量会自动获取运行 Compose 主机上对应变量的值，可以用来防止泄露不必要的数据。

```
environment:
  RACK_ENV: development
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SESSION_SECRET
```

如果变量名称或者值中用到 `true|false，yes|no` 等表达布尔含义的词汇，最好放到引号里，避免 YAML 自动解析某些内容为对应的布尔语义。这些特定词汇，包括

```
y|Y|yes|Yes|YES|n|N|no|No|NO|true|True|TRUE|false|False|FALSE|on|On|ON|off|Off|OFF
```

11.5.15 expose

暴露端口，但不映射到宿主机，只被连接的服务访问。

仅可以指定内部端口为参数

```
expose:
 - "3000"
 - "8000"
```

11.5.16 external_links

> 注意：不建议使用该指令。

链接到 Compose 文件外部的容器，甚至并非 `Compose` 管理的外部容器。

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

11.5.17 extra_hosts

类似 Docker 中的 `--add-host` 参数，指定额外的 host 名称映射信息。

```
extra_hosts:
 - "googledns:8.8.8.8"
 - "dockerhub:52.1.157.61"
```

会在启动后的服务容器中 `/etc/hosts` 文件中添加如下两条条目。

```
8.8.8.8 googledns
52.1.157.61 dockerhub
```

11.5.18 `healthcheck`

通过命令检查容器是否健康运行。

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 1m30s
  timeout: 10s
  retries: 3
```

11.5.19 `image`

指定为镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉取这个镜像。

```
image: ubuntu
image: orchardup/postgresql
image: a4bc65fd
```

11.5.20 `labels`

为容器添加 Docker 元数据 (metadata) 信息。例如可以为容器添加辅助说明信息。

```
labels:
  com.startupteam.description: "webapp for a startup team"
  com.startupteam.department: "devops department"
  com.startupteam.release: "rc3 for v1.0"
```

11.5.21 `links`

> 注意：不推荐使用该指令。容器之间应通过 Docker 网络 (networks) 进行互联。

11.5.22 `logging`

配置日志选项。

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

目前支持三种日志驱动类型。

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

`options` 配置日志驱动的相关参数。

```
options:
  max-size: "200k"
  max-file: "10"
```

11.5.23 network_mode

设置网络模式。使用和 `docker run` 的 `--network` 参数一样的值。

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

11.5.24 networks

配置容器连接的网络。

```
services:

  some-service:
    networks:
     - some-network
     - other-network

networks:
  some-network:
  other-network:
```

11.5.25 pid

跟主机系统共享进程命名空间。打开该选项的容器之间，以及容器和宿主机系统之间可以通过进程 ID 来相互访问和操作。

```
pid: "host"
```

11.5.26 ports

暴露端口信息。

使用宿主端口：容器端口 (HOST:CONTAINER) 格式，或者仅仅指定容器的端口 (宿主将会随机选择端口) 都可以。

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

注意：当使用 HOST:CONTAINER 格式来映射端口时，如果你使用的容器端口小于 60 并且没放到引号里，可能会得到错误结果，因为 YAML 会自动解析 xx:yy 这种数字格式为 60 进制。为避免出现这种问题，建议数字串都采用引号包括起来的字符串格式。

11.5.27 secrets

存储敏感数据，例如 mysql 服务密码。

```
services:
  mysql:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
    secrets:
      - db_root_password
      - my_other_secret

secrets:
  db_root_password:
    file: ./db_root_password.txt
  my_other_secret:
    external: true
```

11.5.28 security_opt

指定容器模板标签 (label) 机制的默认属性 (用户、角色、类型、级别等)。例如配置标签的用户名和角色名。

```
security_opt:
    - label:user:USER
    - label:role:ROLE
```

11.5.29 stop_signal

设置另一个信号来停止容器。在默认情况下使用的是 SIGTERM 停止容器。

```
stop_signal: SIGUSR1
```

11.5.30 sysctls

配置容器内核参数。

```
sysctls:
  net.core.somaxconn: 1024
  net.ipv4.tcp_syncookies: 0

sysctls:
  - net.core.somaxconn=1024
  - net.ipv4.tcp_syncookies=0
```

11.5.31 ulimits

指定容器的 ulimits 限制值。

例如，指定最大进程数为 65535，指定文件句柄数为 20000 (软限制，应用可以随时修改，不能超过硬限制) 和 40000 (系统硬限制，只能 root 用户提高)。

```
  ulimits:
    nproc: 65535
    nofile:
      soft: 20000
      hard: 40000
```

11.5.32 volumes

数据卷所挂载路径设置。可以设置为宿主机路径 (`HOST:CONTAINER`) 或者数据卷名称 (`VOLUME:CONTAINER`)，并且可以设置访问模式 (`HOST:CONTAINER:ro`)。

该指令中路径支持相对路径。

```
volumes:
 - /var/lib/mysql
 - cache/:/tmp/cache
 - ~/configs:/etc/configs/:ro
```

如果路径为数据卷名称，必须在文件中配置数据卷。

```yaml
services:
  my_src:
    image: mysql:8.4
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

11.5.33 其它指令

此外，还有包括 `domainname`, `entrypoint`, `hostname`, `ipc`, `mac_address`, `privileged`, `read_only`, `shm_size`, `restart`, `stdin_open`, `tty`, `user`, `working_dir` 等指令，基本跟 `docker run` 中对应参数的功能一致。

指定服务容器启动后执行的入口文件。

```yaml
entrypoint: /code/entrypoint.sh
```

指定容器中运行应用的用户名。

```yaml
user: nginx
```

指定容器中工作目录。

```yaml
working_dir: /code
```

指定容器中搜索域名、主机名、mac 地址等。

```yaml
domainname: your_website.com
hostname: test
mac_address: 08-00-27-00-0C-0A
```

允许容器中运行一些特权命令。`privileged: true` 会显著扩大容器对宿主机设备和 Linux capability 的访问范围，生产环境应优先使用更窄的 `cap_add`、`devices` 或只读挂载；更多说明见 内核能力机制。

```yaml
privileged: true
```

指定容器退出后的重启策略。该命令对保持服务始终运行十分有效，在生产环境中推荐配置为 `always` 或者 `unless-stopped`。

策略	说明
`no`	默认值，不自动重启
`always`	无论退出码如何，始终重启；Docker 守护进程启动时也会重启
`on-failure[:max-retries]`	仅在非零退出码时重启，可指定最大重试次数
`unless-stopped`	类似 `always`，但手动停止的容器在守护进程重启后不会自动启动

```
restart: always
```

以只读模式挂载容器的 root 文件系统，意味着不能对容器内容进行修改。

```
read_only: true
```

打开标准输入，可以接受外部输入。

```
stdin_open: true
```

模拟一个伪终端。

```
tty: true
```

11.5.34 `profiles`

`profiles` 用于按场景选择性启动服务。未指定 `profiles` 的服务默认始终启动；标记了 `profiles` 的服务仅在激活对应 profile 时才启动。

```yaml
services:
  web:
    image: nginx

  debug:
    image: busybox
    profiles:
      - debug

  test:
    image: node
    profiles:
      - test
```

启动时通过 `--profile` 激活：

```
$ docker compose --profile debug up    # 启动 web + debug
$ docker compose up                    # 仅启动 web
```

11.5.35 读取变量

Compose 模板文件支持动态读取主机的系统环境变量和当前目录下的 `.env` 文件中的变量。

例如，下面的 Compose 文件将从运行它的环境中读取变量 `${MONGO_VERSION}` 的值，并写入执行的指令中。

```yaml
services:

db:
  image: "mongo:${MONGO_VERSION}"
```

如果执行 `MONGO_VERSION=3.2 docker compose up` 则会启动一个 `mongo:3.2` 镜像的容器；如果执行 `MONGO_VERSION=2.8 docker compose up` 则会启动一个 `mongo:2.8` 镜像的容器。

若当前目录存在 `.env` 文件，执行 `docker compose` 命令时将从该文件中读取变量。

在当前目录新建 `.env` 文件并写入以下内容。

```
## 支持 # 号注释

MONGO_VERSION=3.6
```

执行 `docker compose up` 则会启动一个 `mongo:3.6` 镜像的容器。

11.6 实战 Django

> 本小节内容适合 `Python` 开发人员阅读。

> **版本说明**：本示例使用以下镜像版本：
>
> - Python：3.12-slim（可替换为其他 3.x 版本）
> - PostgreSQL：16（可替换为其他 16.x、15.x 等版本）
> - Django：>=5.0,<6.0（可根据项目需求调整）

本节将使用 Docker Compose 配置并运行一个 **Django + PostgreSQL** 应用。笔者不仅会介绍具体步骤，还会解释每个配置项的作用，以及开发环境和生产环境的差异。

11.6.1 架构概览

在开始之前，先看整体架构 (如图 11-1 所示)：

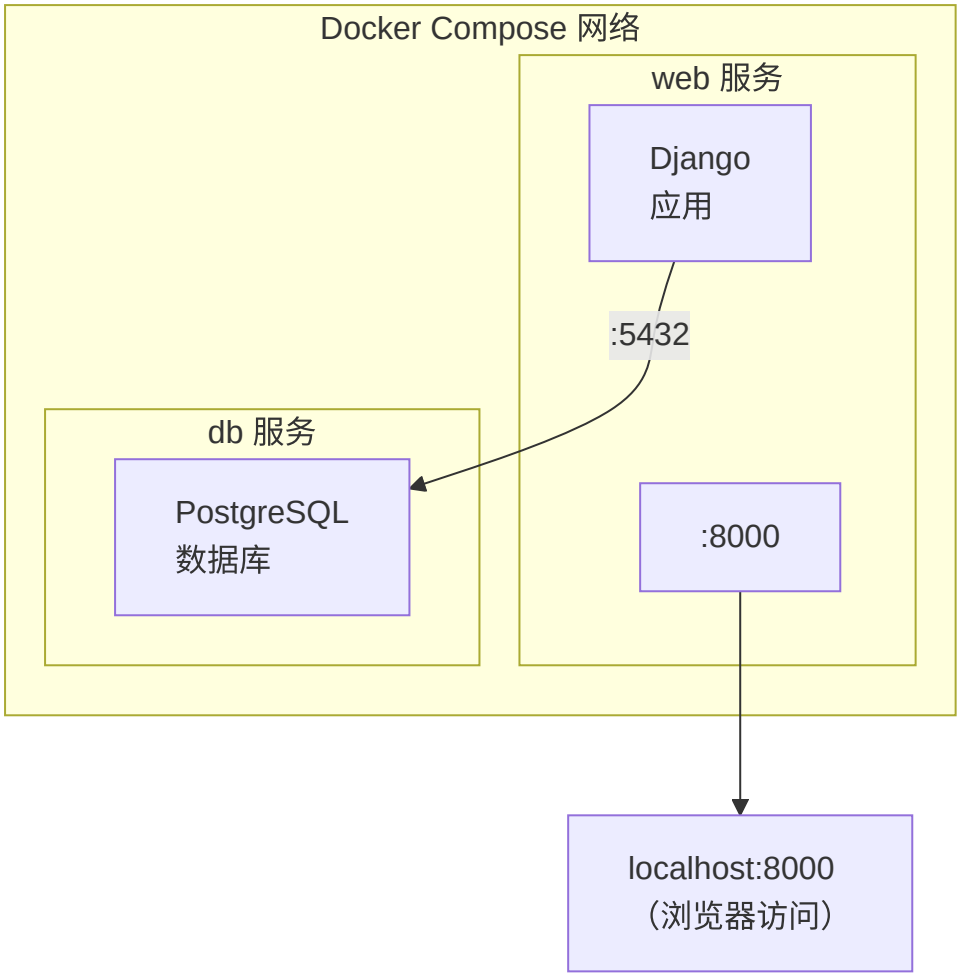


图 11-1：Django + PostgreSQL 的 Compose 架构

关键点：

- `web` 服务运行 Django 应用，对外暴露 8000 端口
- `db` 服务运行 PostgreSQL 数据库，只在内部网络可访问
- 两个服务通过 Docker Compose 自动创建的网络相互通信
- `web` 服务可以通过服务名 `db` 访问数据库 (Docker 内置 DNS)

11.6.2 准备工作

创建一个项目目录并进入：

```
$ mkdir django-docker && cd django-docker
```

我们需要创建三个文件：`Dockerfile`、`requirements.txt` 和 `compose.yaml`。

11.6.3 步骤 1：创建 Dockerfile

```
FROM python:3.12-slim

## 防止 Python 缓冲 stdout/stderr，让日志实时输出

ENV PYTHONUNBUFFERED=1

## 设置工作目录

WORKDIR /code

## 先复制依赖文件，利用 Docker 缓存加速构建

COPY requirements.txt /code/

## 安装依赖

RUN pip install --no-cache-dir -r requirements.txt

## 复制项目代码

COPY . /code/
```

逐行解释：

指令	作用	为什么这样写
`FROM python:3.12-slim`	基础镜像	`slim` 版本比完整版小很多，但包含运行 Python 所需的一切
`ENV PYTHONUNBUFFERED=1`	关闭输出缓冲	让 `print()` 和日志立即显示，便于调试
`WORKDIR /code`	设置工作目录	后续命令都在此目录执行
`COPY requirements.txt` 在前	分层复制	只有 requirements.txt 变化时才重新安装依赖，加速构建
`--no-cache-dir`	不缓存 pip 下载	减小镜像体积

> 💡 **笔者建议**：总是将变化频率低的文件先复制，变化频率高的后复制。这样可以最大化利用 Docker 的构建缓存。

11.6.4 步骤 2：创建 requirements.txt

```
Django>=5.0,<6.0
psycopg[binary]>=3.1,<4.0
gunicorn>=21.0,<22.0
```

依赖说明：

包名	作用
`Django`	Web 框架
`psycopg[binary]`	PostgreSQL 数据库驱动 (推荐使用 psycopg 3)
`gunicorn`	生产环境 WSGI 服务器 (可选，开发时可不用)

11.6.5 步骤 3：创建 compose.yaml

配置如下：

```yaml
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_DB: django_db
      POSTGRES_USER: django_user
      POSTGRES_PASSWORD: django_password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U django_user -d django_db"]
      interval: 5s
      timeout: 5s
      retries: 5

  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      db:
        condition: service_healthy
    environment:
      DATABASE_URL: postgres://django_user:django_password@db:5432/django_db

volumes:
  postgres_data:
```

配置详解：

db 服务

db 服务配置如下：

```
db:
  image: postgres:16                    # 使用官方 PostgreSQL 16 镜像
  environment:
    POSTGRES_DB: django_db              # 创建的数据库名
    POSTGRES_USER: django_user          # 数据库用户
    POSTGRES_PASSWORD: django_password  # 数据库密码
  volumes:
    - postgres_data:/var/lib/postgresql/data  # 持久化数据
  healthcheck:                          # 健康检查，确保数据库就绪
    test: ["CMD-SHELL", "pg_isready -U django_user -d django_db"]
    interval: 5s
```

> ⚠️ **笔者提醒**：volumes 配置很重要！没有它，每次容器重启数据都会丢失。笔者见过不少新手因为忘记这一步，导致开发数据全部丢失。

web 服务

web 服务配置如下：

```
web:
  build: .                              # 从当前目录的 Dockerfile 构建
  command: python manage.py runserver   # 启动 Django 开发服务器
  volumes:
    - .:/code                           # 挂载代码目录，支持热更新
  ports:
    - "8000:8000"                       # 映射端口
  depends_on:
    db:
      condition: service_healthy        # 等待数据库健康后再启动
```

关键配置说明：

配置项	作用	笔者建议
`volumes: .:/code`	代码挂载	开发时必备，修改代码无需重新构建镜像
`depends_on` + `healthcheck`	启动顺序	确保数据库就绪后 Django 才启动，避免连接错误
`environment`	环境变量	推荐用环境变量管理配置，避免硬编码

11.6.6 步骤 4：创建 Django 项目

运行以下命令创建新的 Django 项目：

```
$ docker compose run --rm web django-admin startproject mysite .
```

命令解释：

- `docker compose run`：运行一次性命令
- `--rm`：命令执行后删除临时容器
- `web`：在 web 服务环境中执行
- `django-admin startproject mysite .`：在当前目录创建 Django 项目

生成的目录结构：

```
django-docker/
├── compose.yaml
├── Dockerfile
├── requirements.txt
├── manage.py
└── mysite/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── asgi.py
    └── wsgi.py
```

> 💡 **Linux 用户注意**：如果遇到权限问题，执行 `sudo chown -R $USER:$USER .`

11.6.7 步骤 5：配置数据库连接

修改 `mysite/settings.py`，配置数据库连接：

```
import os

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB', 'django_db'),
        'USER': os.environ.get('POSTGRES_USER', 'django_user'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD', 'django_password'),
        'HOST': 'db',  # Docker Compose 服务名
        'PORT': 5432,
    }
}

## 允许的主机（开发环境）

ALLOWED_HOSTS = ['*']
```

为什么 HOST 是 `db` 而不是 `localhost`？

在 Docker Compose 中，各服务通过服务名相互访问。Docker 内置的 DNS 会将 `db` 解析为 db 服务容器的 IP 地址。这是 Docker Compose 的核心功能之一。

11.6.8 步骤 6：启动应用

```
$ docker compose up
```

你会看到：

1. 首先构建 web 镜像 (第一次运行)
2. 启动 db 服务，等待健康检查通过
3. 启动 web 服务

```
db-1   | PostgreSQL init process complete; ready for start up.
db-1   | LOG:  database system is ready to accept connections
web-1  | Watching for file changes with StatReloader
web-1  | Starting development server at http://0.0.0.0:8000/
```

打开浏览器访问 `http://localhost:8000`，可以看到 Django 欢迎页面！

11.6.9 常用开发命令

在另一个终端窗口执行：

```
## 执行数据库迁移

$ docker compose exec web python manage.py migrate

## 创建超级用户

$ docker compose exec web python manage.py createsuperuser

## 进入 Django shell

$ docker compose exec web python manage.py shell

## 进入 PostgreSQL 命令行

$ docker compose exec db psql -U django_user -d django_db
```

> 💡 笔者建议使用 exec 而不是 run。exec 在已运行的容器中执行命令，run 会创建新容器。

11.6.10 常见问题排查

Q1：数据库连接失败

错误信息：`django.db.utils.OperationalError: could not connect to server` **可能原因与解决方案：**

原因	解决方案
数据库还没启动完成	使用 `depends_on` + `healthcheck`
HOST 配置错误	确保使用服务名 `db` 而不是 `localhost`
网络未创建	运行 `docker compose down` 后重新 `up`

```
## 调试：检查数据库是否正常运行

$ docker compose ps
$ docker compose logs db
```

Q2：代码修改没有生效

可能原因：

1. **开发服务器没有自动重载**：确保使用 `runserver` 而不是 `gunicorn`
2. **Volume 挂载问题**：检查 `compose.yaml` 中的 volumes 配置
3. **缓存问题**：尝试 `docker compose restart web`

Q3：权限问题

```
## 如果容器内创建的文件 root 用户所有

$ sudo chown -R $USER:$USER .
```

11.6.11 开发 vs 生产：关键差异

笔者特别提醒，本节的配置是 **开发环境** 配置。生产环境需要以下调整：

配置项	开发环境	生产环境
Web 服务器	`runserver`	`gunicorn` + Nginx
DEBUG	`True`	`False`
密码管理	明文写在配置	使用 Docker Secrets 或环境变量
Volume	挂载代码目录	代码直接 COPY 进镜像
ALLOWED_HOSTS	`['*']`	具体域名

生产环境 Compose 文件示例：

```
## compose.prod.yaml

services:
  web:
    build: .
    command: gunicorn mysite.wsgi:application --bind 0.0.0.0:8000
    # 不挂载代码，使用镜像内的代码

    environment:
      DEBUG: 'False'
      ALLOWED_HOSTS: 'example.com,www.example.com'
    # ...
```

11.6.12 延伸阅读

- Compose 模板文件详解：深入理解 Compose 文件的所有配置项
- 使用 WordPress：另一个 Compose 实战案例
- Dockerfile 最佳实践：构建更小、更安全的镜像
- 数据管理：Volume 和数据持久化详解

11.7 实战 Rails

> 本小节内容适合 Ruby 开发人员阅读。

> **版本说明**：本示例使用以下镜像版本：
>
> - Ruby：3.2（可替换为其他 3.x 版本）
> - PostgreSQL：16（可替换为其他 16.x、15.x 等版本）
> - Rails：~> 7.1（可根据项目需求调整）

本节使用 Docker Compose 配置并运行一个 **Rails + PostgreSQL** 应用。

11.7.1 架构概览

如图 11-2 所示，Rails 与 PostgreSQL 在同一 Compose 网络中协同工作。

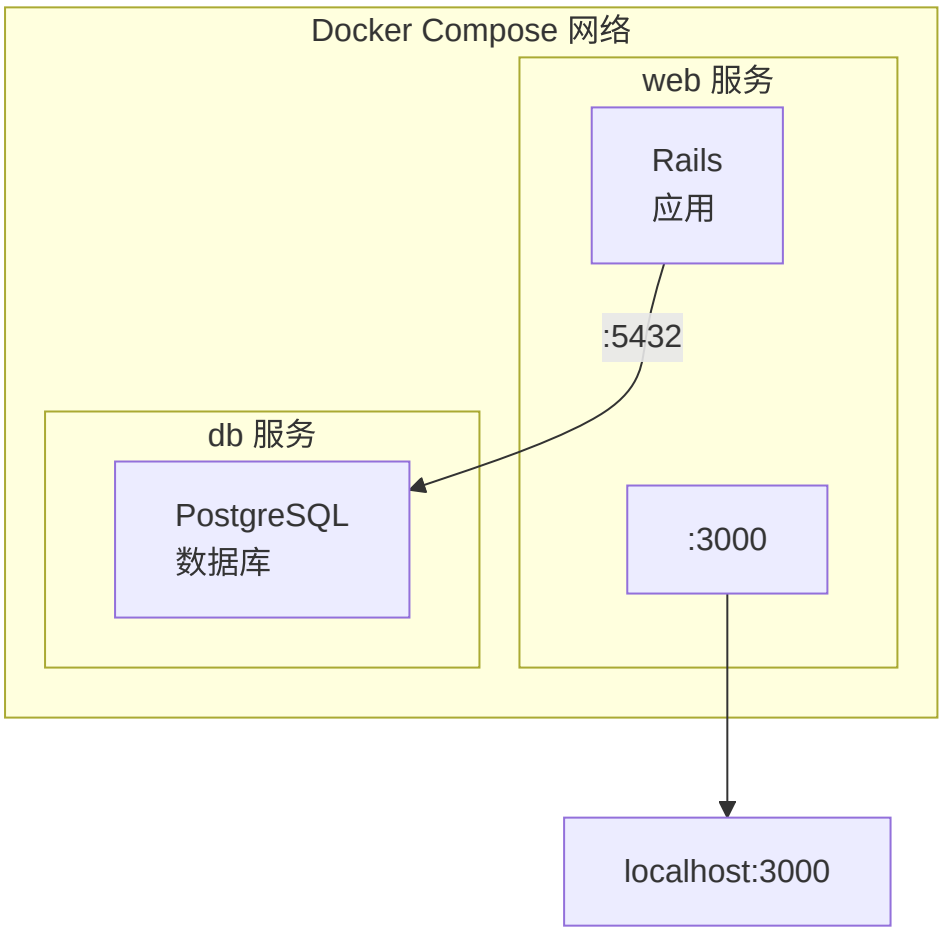


图 11-2：Rails + PostgreSQL 的 Compose 架构

11.7.2 准备工作

创建项目目录：

```
$ mkdir rails-docker && cd rails-docker
```

需要创建三个文件：`Dockerfile`、`Gemfile` 和 `compose.yaml`。

11.7.3 步骤 1：创建 Dockerfile

```
FROM ruby:3.2

## 安装系统依赖

RUN apt-get update -qq && \
    apt-get install -y build-essential libpq-dev nodejs && \
    rm -rf /var/lib/apt/lists/*

## 设置工作目录

WORKDIR /myapp

## 先复制 Gemfile，利用缓存加速构建

COPY Gemfile /myapp/Gemfile
COPY Gemfile.lock /myapp/Gemfile.lock
RUN bundle install

## 复制应用代码

COPY . /myapp
```

配置说明：

指令	作用
`build-essential`	编译原生扩展所需
`libpq-dev`	PostgreSQL 客户端库
`nodejs`	Rails Asset Pipeline 需要
先复制 Gemfile	只有依赖变化时才重新 `bundle install`

11.7.4 步骤 2：创建 Gemfile

创建一个初始的 `Gemfile`，稍后会被 `rails new` 覆盖：

```
source 'https://rubygems.org'
gem 'rails', '~> 7.1'
```

创建空的 `Gemfile.lock`：

```
$ touch Gemfile.lock
```

11.7.5 步骤 3：创建 compose.yaml

配置如下：

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5

  web:
    build: .
    command: bash -c "rm -f tmp/pids/server.pid && bundle exec rails s -p 3000 -b '0.0.0.0'"
    volumes:
      - .:/myapp
    ports:
      - "3000:3000"
    depends_on:
      db:
        condition: service_healthy
    environment:
      DATABASE_URL: postgres://postgres:password@db:5432/myapp_development

volumes:
  postgres_data:
```

配置详解：

配置项	说明
`rm -f tmp/pids/server.pid`	清理上次异常退出留下的 PID 文件
`volumes: .:/myapp`	挂载代码目录，支持热更新
`depends_on` + `condition`	等待数据库健康检查通过后再启动
`DATABASE_URL`	Rails 12-factor 风格的数据库配置

11.7.6 步骤 4：生成 Rails 项目

使用 `docker compose run` 生成项目骨架：

```
$ docker compose run --rm web rails new . --force --database=postgresql --skip-bundle
```

命令解释：

- `--rm`：执行后删除临时容器
- `--force`：覆盖已存在的文件
- `--database=postgresql`：配置使用 PostgreSQL
- `--skip-bundle`：暂不安装依赖 (稍后统一安装)

生成的目录结构：

```
$ ls
Dockerfile      Gemfile         Rakefile        config          lib             tmp
Gemfile.lock    README.md       app             config.ru       log             vendor
compose.yaml    bin             db              public
```

> ⚠️ **Linux 用户**：如遇权限问题，执行 `sudo chown -R $USER:$USER .`

11.7.7 步骤 5：重新构建镜像

由于生成了新的 Gemfile，需要重新构建镜像以安装完整依赖：

```
$ docker compose build
```

11.7.8 步骤 6：配置数据库连接

修改 `config/database.yml`：

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  url: <%= ENV['DATABASE_URL'] %>

development:
  <<: *default

test:
  <<: *default
  database: myapp_test

production:
  <<: *default
```

> 💡 使用 DATABASE_URL 环境变量配置数据库，符合 12-factor 应用原则，便于在不同环境间切换。

11.7.9 步骤 7：启动应用

```
$ docker compose up
```

输出示例：

```
db-1   | PostgreSQL init process complete; ready for start up.
db-1   | LOG:  database system is ready to accept connections
web-1  | => Booting Puma
web-1  | => Rails 7.1.0 application starting in development
web-1  | => Run `bin/rails server --help` for more startup options
web-1  | Puma starting in single mode...
web-1  | * Listening on http://0.0.0.0:3000
```

11.7.10 步骤 8：创建数据库

在另一个终端执行：

```
$ docker compose exec web rails db:create
Created database 'myapp_development'
Created database 'myapp_test'
```

访问 http://localhost:3000 查看 Rails 欢迎页面。

11.7.11 常用开发命令

```
## 数据库迁移

$ docker compose exec web rails db:migrate

## Rails 控制台

$ docker compose exec web rails console

## 运行测试

$ docker compose exec web rails test

## 生成脚手架

$ docker compose exec web rails generate scaffold Post title:string body:text

## 进入容器 Shell

$ docker compose exec web bash
```

11.7.12 常见问题

Q：数据库连接失败

检查 DATABASE_URL 环境变量格式是否正确，确保 db 服务已启动：

```
$ docker compose ps
$ docker compose logs db
```

Q：server.pid 文件导致启动失败

错误信息：`A server is already running`

已在 command 中添加 `rm -f tmp/pids/server.pid` 处理。如仍有问题：

```
$ docker compose exec web rm -f tmp/pids/server.pid
```

Q：Gem 安装失败

可能需要更新 bundler 或清理缓存：

```
$ docker compose run --rm web bundle update
```

11.7.13 开发 vs 生产

配置项	开发环境	生产环境
Rails 服务器	Puma (开发模式)	Puma + Nginx
代码挂载	使用 volumes	代码打包进镜像
静态资源	动态编译	预编译 (`rails assets:precompile`)
数据库密码	明文配置	使用 Secrets 管理

11.7.14 延伸阅读

- 使用 Django：Python Web 框架实战
- Compose 模板文件：配置详解
- 数据管理：数据持久化

Built with mdPress

11.8 实战 WordPress

> **版本说明**：本示例使用以下镜像版本：
>
> - MySQL：8.0（可替换为其他 8.x 版本，或使用 MariaDB 替代）
> - WordPress：latest（建议在生产环境指定具体版本，如 6.x）

WordPress 是全球最流行的内容管理系统 (CMS)。使用 Docker Compose 可以在几分钟内搭建一个包含数据库、Web 服务和持久化存储的生产级 WordPress 环境。

11.8.1 项目结构

```
wordpress/
├── compose.yaml
├── .env                # 非敏感环境变量（如版本、端口）
├── secrets/            # 本地密钥文件，生产环境应由密钥管理系统提供
│   ├── db_root_password.txt
│   └── db_password.txt
└── nginx/              # 可选：反向代理配置
    └── nginx.conf
```

11.8.2 编写 `compose.yaml`

这是一个可运行的单机最小配置，不是完整生产安全基线。正式部署前应固定镜像版本、放在反向代理/TLS 后面、限制公开端口、配置备份与监控，并按 WordPress 与插件生命周期做升级测试。

```yaml
services:
  # 数据库服务

  db:
    image: mysql:8.4
    container_name: wordpress_db
    restart: always
    command:
      # 启用原生密码认证（MySQL 8.4 默认禁用，旧版 WP 兼容性需要）

      - --mysql-native-password=ON
      - --character-set-server=utf8mb4
      - --collation-server=utf8mb4_unicode_ci
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_root_password
      - db_password
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - wp_net

  # WordPress 服务

  wordpress:
    # 示例保留 latest 以便读者快速体验；生产环境请固定到经过测试的明确版本标签
    image: wordpress:latest
    container_name: wordpress_app
    restart: always
    ports:
      # 本机调试入口；生产环境请通过反向代理发布 HTTPS
      - "127.0.0.1:8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
      WORDPRESS_DB_NAME: wordpress
    secrets:
      - db_password
    volumes:
      - wp_data:/var/www/html
      # 增加上传文件大小限制

      - ./uploads.ini:/usr/local/etc/php/conf.d/uploads.ini
    depends_on:
      - db
    networks:
      - wp_net

volumes:
  db_data:  # 数据库持久化
  wp_data:  # WordPress 文件(插件/主题/上传)持久化

networks:
  wp_net:
```

```
secrets:
  db_root_password:
    file: ./secrets/db_root_password.txt
  db_password:
    file: ./secrets/db_password.txt
```

11.8.3 配置文件详解

1. 密钥文件

不要把数据库密码写入 `compose.yaml`、`.env`、命令行或 Git。Compose 官方建议敏感值使用 secrets；本地练习可用只读密钥文件模拟：

```
mkdir -p secrets
printf '%s\n' 'somestrongrootpassword' > secrets/db_root_password.txt
printf '%s\n' 'somestronguserpassword' > secrets/db_password.txt
chmod 600 secrets/*.txt
```

把 `secrets/` 加入 `.gitignore`。生产环境应改用平台密钥管理能力，而不是把真实密码放在项目目录。

2. 数据持久化

我们定义了两个命名卷：

- `db_data`：确保 MySQL 容器重建后数据不丢失
- `wp_data`：保存 WordPress 的核心文件、插件、主题和上传的媒体文件

3. PHP 配置优化

默认的 WordPress 镜像上传文件限制较小 (通常 2MB)。创建 `uploads.ini`：

```
file_uploads = On
memory_limit = 256M
upload_max_filesize = 64M
post_max_size = 64M
max_execution_time = 600
```

11.8.4 启动与运行

1. 启动服务：

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

2. 访问安装界面： 打开浏览器访问 `http://localhost:8000`

3. 查看日志：

```
$ docker compose logs -f
```

11.8.5 生产环境最佳实践

1. 数据库备份

不要只依赖 Volume。建议定期备份数据库：

```
## 导出 SQL

$ docker exec wordpress_db mysqldump -u wordpress -pwordpress wordpress > backup.sql
```

或者添加一个自动备份容器：

```
  backup:
    image: tiredofit/db-backup
    volumes:
      - ./backups:/backup
    environment:
      - DB_TYPE=mysql
      - DB_HOST=db
      - DB_NAME=wordpress
      - DB_USER=wordpress
      - DB_PASS=${DB_PASSWORD}
      - DB_DUMP_FREQ=1440 # 每天备份一次
    depends_on:
      - db
    networks:
      - wp_net
```

2. 使用 Nginx 反向代理

在生产环境中，不要直接暴露 WordPress 端口，而是通过 Nginx 进行反向代理并配置 SSL。

3. 使用 Redis 缓存

WordPress 支持 Redis 缓存以提高性能。

```
  redis:
    image: redis:alpine
    restart: always
    networks:
      - wp_net
```

在 WordPress 容器环境变量中添加：

```
      WORDPRESS_REDIS_HOST: redis
```

并安装 Redis Object Cache 插件。

11.8.6 常见问题

Q：数据库连接错误

现象：访问页面显示 “Error establishing a database connection”。**排查**：

1. 检查 `docker compose logs wordpress`
2. 确认 `.env` 中的密码与 YAML 文件引用一致
3. 确认 `WORDPRESS_DB_HOST` 也是 `db` (服务名)
4. MySQL 8.4 可能需要几秒钟启动，WordPress 会自动重试，稍等片刻即可。

Q：无法上传大文件

解决：确保挂载了 `uploads.ini` 配置，并且重启了容器：

```
$ docker compose restart wordpress
```

11.8.7 延伸阅读

- Compose 模板文件：深入了解配置项
- 数据卷：理解数据持久化
- Docker Hub WordPress：官方镜像文档

11.9 实战 LNMP

什么是 LNMP

LNMP 是一个经典的 Web 应用栈，由以下四个开源软件组合而成：

- **L**：Linux（操作系统）
- **N**：Nginx（Web 服务器）
- **M**：MySQL（数据库服务器）
- **P**：PHP（脚本语言）

这个组合被广泛用于构建高性能的 Web 应用。

使用 Docker Compose 部署 LNMP

本项目的维护者 khs1994 的开源项目 khs1994-docker/lnmp 使用 Docker Compose 搭建了一套完整的 LNMP 环境。

参考项目

该项目中包含的服务：

- **Nginx**：Web 服务器，用于处理 HTTP 请求
- **MySQL/MariaDB**：关系型数据库服务
- **PHP-FPM**：PHP 处理器，与 Nginx 通过 Fast CGI 协议通信
- **Redis**：可选的内存缓存服务（用于会话或缓存）

学习资源

各位开发者可以参考该项目在以下场景中运行 LNMP：

- Docker 容器化部署
- Kubernetes 集群编排
- 开发环境快速搭建
- 生产环境配置参考

项目地址：khs1994-docker/lnmp

Built with mdPress

通过该项目，你可以学习到如何使用 Docker Compose 定义多个相互关联的服务，以及如何在容器化环境中管理应用的生命周期。

本章小结

Docker Compose 是管理多容器应用的利器，通过 YAML 文件声明式地定义服务、网络和数据卷。

概念	要点
核心概念	服务 (service) 和项目 (project)
配置文件	`compose.yaml` (推荐) 或 `docker-compose.yml`
版本	Compose V2 为 Go 编写的 CLI 插件，通过 `docker compose` 使用
启动	`docker compose up -d` 启动所有服务
停止	`docker compose down` 停止并移除容器
查看状态	`docker compose ps` 查看服务状态
查看日志	`docker compose logs` 查看服务日志
模板文件	支持 `services`、`networks`、`volumes` 等顶级配置

延伸阅读

- Compose 模板文件：详细模板语法参考
- Compose 命令说明：完整命令列表
- 网络配置：Docker 网络基础
- 数据管理：数据卷管理

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十二章 底层实现

Docker 底层的核心技术包括 Linux 上的命名空间 (Namespaces)、控制组 (Control groups)、Union 文件系统 (Union file systems) 和容器格式 (Container format)。

我们知道，传统的虚拟机通过在宿主主机中运行 hypervisor 来模拟一整套完整的硬件环境提供给虚拟机的操作系统。虚拟机系统看到的环境是可限制的，也是彼此隔离的。 这种直接的做法实现了对资源最完整的封装，但很多时候往往意味着系统资源的浪费。 例如，以宿主机和虚拟机系统都为 Linux 系统为例，虚拟机中运行的应用其实可以利用宿主机系统中的运行环境。

我们知道，在操作系统中，包括内核、文件系统、网络、PID、UID、IPC、内存、硬盘、CPU 等等，所有的资源都是应用进程直接共享的。 要想实现虚拟化，除了要实现对内存、CPU、网络 IO、硬盘 IO、存储空间等的限制外，还要实现文件系统、网络、PID、UID、IPC 等等的相互隔离。 前者相对容易实现一些，后者则需要宿主机系统的深入支持。

随着 Linux 系统对于命名空间功能的完善实现，程序员已经可以实现上面的所有需求，让某些进程在彼此隔离的命名空间中运行。大家虽然都共用一个内核和某些运行时环境 (例如一些系统命令和系统库)，但是彼此却看不到，都以为系统中只有自己的存在。这种机制就是容器 (Container)，利用命名空间来做权限的隔离控制，利用 cgroups 来做资源分配。

本章内容

- 基本架构
- 命名空间
- 控制组
- 联合文件系统
- 容器格式
- 网络

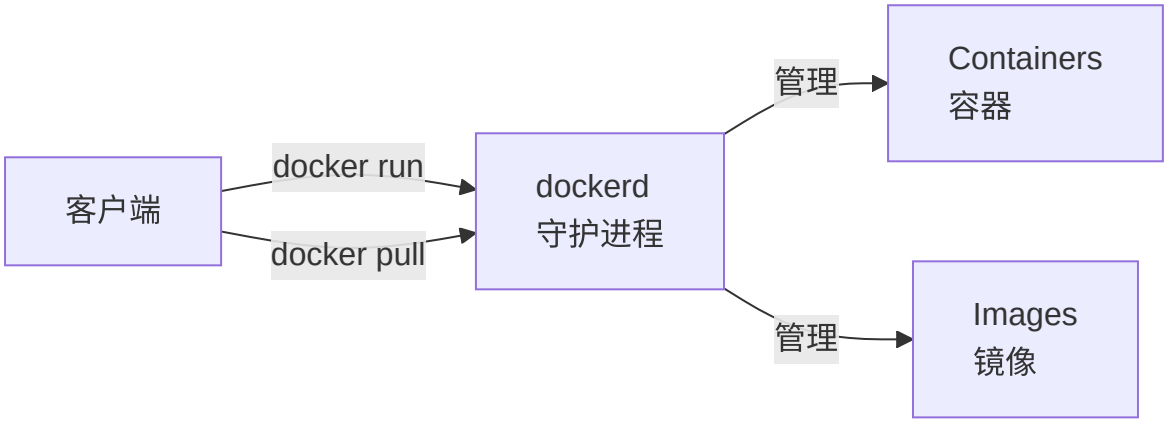
Built with mdPress

12.1 基本架构

Docker 的架构设计简洁而高效，主要由客户端和服务端两部分组成。

12.1.1 核心架构图

Docker 采用了 **C/S (客户端/服务端)** 架构。Client 向 Daemon 发送请求，Daemon 负责构建、运行和分发容器。



12.1.2 组件详解

Docker 的内部架构如同洋葱一样分层，每一层专注解决特定问题：

1. Docker CLI：客户端

用户与 Docker 交互的主要方式。它将用户命令 (如 `docker run`) 转换为 API 请求发送给 dockerd。

2. Dockerd：守护进程

Docker 的大脑。

- 监听 API 请求
- 管理 Docker 对象 (镜像、容器、网络、卷)
- 编排下层组件完成工作

3. Containerd：高级运行时

行业标准的容器运行时 (CNCF 毕业项目)。

Built with mdPress

- 管理容器的完整生命周期 (启动、停止)
- 镜像拉取与存储
- **不包含** 复杂的与容器无关的功能 (如构建、API)
- Kubernetes 也可以直接使用 containerd (跳过 Docker)

4. Runc：低级运行时

用于创建和运行容器的 CLI 工具。

- 直接与内核交互 (Namespaces，Cgroups)
- 遵循 OCI (Open Container Initiative) 规范
- **主要职责**：根据配置启动一个容器，然后退出 (将控制权交给容器进程)

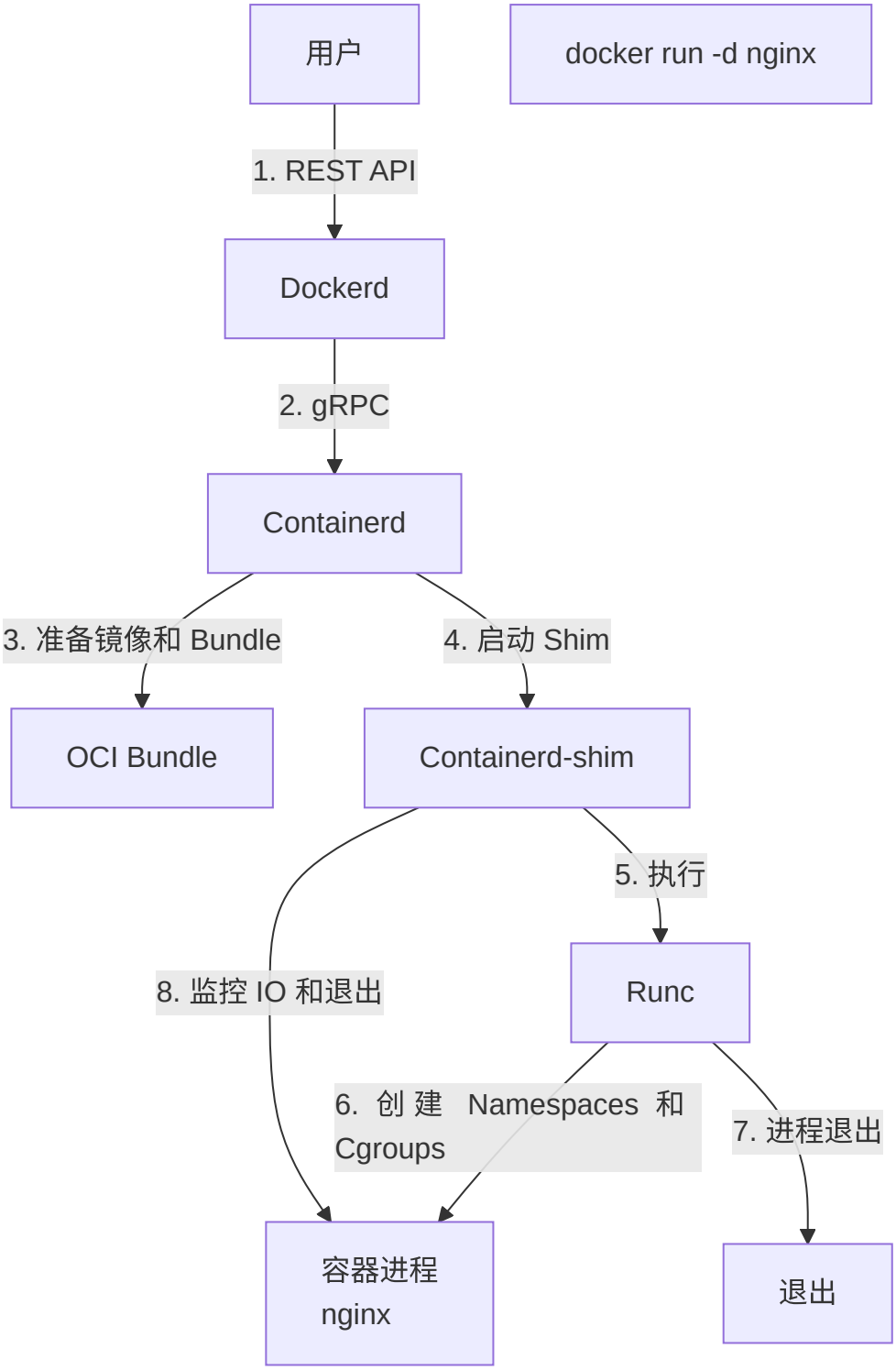
5. Shim

每个容器都有一个 shim 进程。

- **解耦**：允许 dockerd 重启而不影响容器运行
- **保持 IO**：维持容器的标准输入输出
- **状态汇报**：向 containerd 汇报容器退出状态

12.1.3 容器启动流程

当执行 `docker run -d nginx` 时，内部发生了什么？



Built with mdPress

1. **CLI** 发送请求给 **Dockerd**
2. **Dockerd** 解析请求，调用 **Containerd**
3. **Containerd** 准备镜像，转换为 OCI Bundle
4. **Containerd** 创建 **Shim** 进程
5. **Shim** 调用 **Runc**
6. **Runc** 与系统内核交互，创建 Namespaces 和 Cgroups
7. **Runc** 启动 nginx 进程后退出
8. **Shim** 接管容器 IO 和生命周期监控

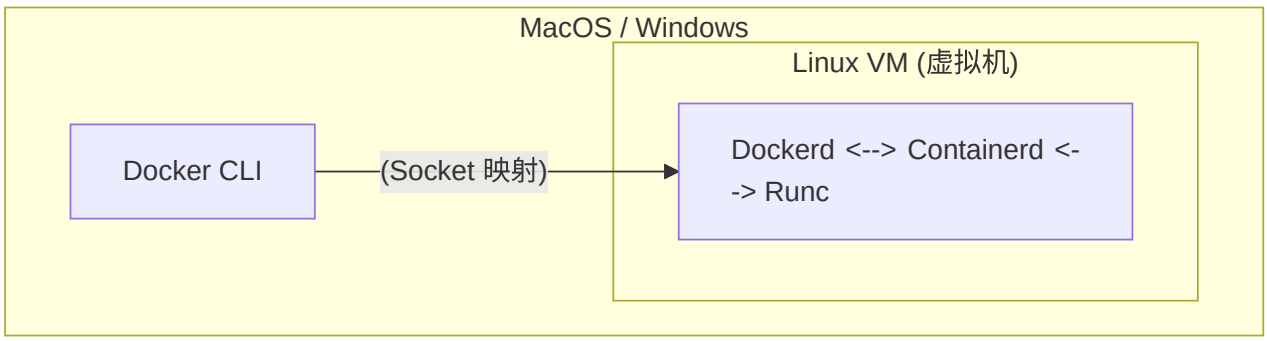
12.1.4 Docker Engine v29.x 变化

从 Docker Engine v29.x 开始，架构进一步简化和标准化：

- **Containerd 镜像存储 (Image Store)**：在 v29.x 的新安装场景中默认启用。Docker 直接使用 Containerd 的镜像管理能力，不再维护自己的一套 graphdriver。
  - **优势**：多平台镜像支持更好、镜像拉取更快 (lazy pulling)、与 K8s 共享镜像。
- **实验性 nftables 支持**：随着主流 Linux 发行版逐步弃用 iptables，Docker v29.x 引入了实验性 nftables 后端。启用方式为 `dockerd --firewall-backend=nftables`，可直接创建 nftables 规则而无需依赖 iptables-nft 转换层。生产环境请谨慎使用。

12.1.5 Docker Desktop 架构

在 macOS 和 Windows 上，因为内核差异，架构稍微复杂：



Built with mdPress

- 使用轻量级虚拟机 (Apple Virtualization / WSL 2) 运行 Linux 内核
- 文件挂载 (Bind Mount) 需要跨越 VM 边界 (这也是文件 I/O 慢的原因)
- 网络端口需要从宿主机转发到 VM

12.1.6 总结

组件	角色	关键职责
CLI	指挥官	发送指令，展示结果
Dockerd	大管家	API 接口，整体调度
Containerd	经理	容器生命周期，镜像管理
Shim	监工	保持 IO，允许无守护进程重启
Runc	工人	真正干活 (创建容器)，干完就走

12.1.7 延伸阅读

- 命名空间：Runc 如何隔离容器
- 控制组：Runc 如何限制资源
- 联合文件系统：镜像如何存储

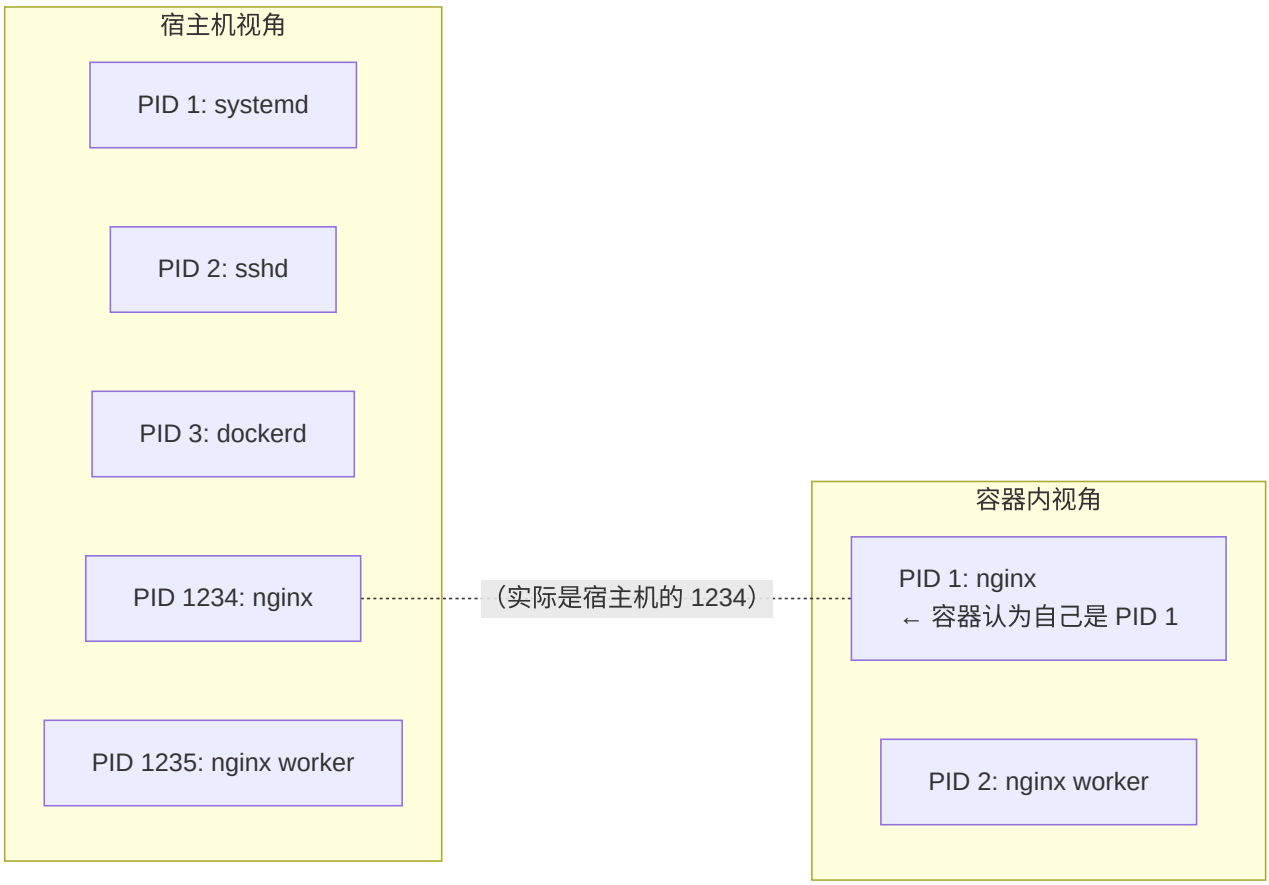
Built with mdPress

12.2 命名空间

命名空间是 Linux 内核一个强大的特性。每个容器都有自己单独的命名空间，运行在其中的应用都像是在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响。

12.2.1 什么是 Namespace

> **Namespace 是 Linux 内核提供的资源隔离机制，它让容器内的进程仿佛运行在独立的操作系统中。**Namespace 是容器技术的核心基础之一。它回答了一个关键问题：**如何让一个进程 “以为” 自己独占整个系统?**



12.2.2 Namespace 的类型

Linux 内核（5.6+）共提供 8 种 Namespace。Docker 容器默认使用其中 7 种（不含 Time）：

Namespace	隔离内容	容器中的效果
PID	进程 ID	容器内 PID 从 1 开始，看不到其他容器和宿主机进程
NET	网络栈	独立的网卡、IP 地址、端口、路由表
MNT	挂载点	独立的文件系统视图，自己的根目录
UTS	主机名	独立的主机名和域名
IPC	进程间通信	独立的信号量、消息队列、共享内存
USER	用户/组 ID	容器内的 root 可以映射为宿主机的普通用户
Cgroup	Cgroup 根目录	隔离 cgroup 层级视图 (Linux 4.6+)
Time	系统时钟	隔离 CLOCK_MONOTONIC 和 CLOCK_BOOTTIME (Linux 5.6+)

12.2.3 PID Namespace

PID Namespace 负责进程 ID 的隔离，使得容器内的进程彼此不可见。

PID 的作用

隔离进程 ID，让每个容器有自己的进程编号空间。

PID 隔离效果

```
## 宿主机上查看进程

$ ps aux | grep nginx
root     12345  0.0  0.1  nginx: master process
root     12346  0.0  0.1  nginx: worker process

## 容器内查看进程

$ docker exec mycontainer ps aux
PID   USER     COMMAND
  1   root     nginx: master process    ← 在容器内是 PID 1
  2   root     nginx: worker process
```

PID 关键点

- 容器内的 PID 1 进程特殊重要——它是容器的主进程，退出则容器停止
- 容器内无法看到宿主机或其他容器的进程
- 宿主机可以看到所有容器内的进程 (但 PID 不同)

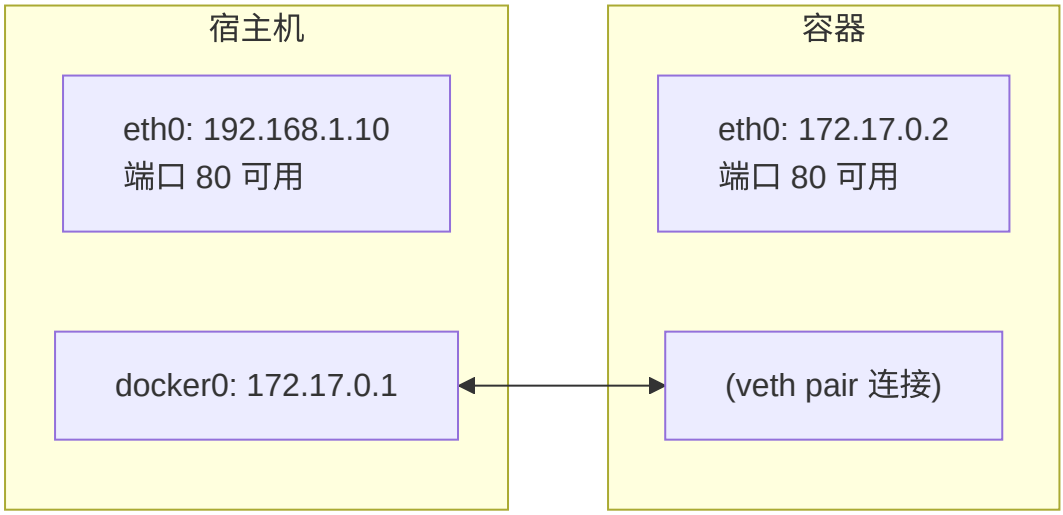
12.2.4 NET Namespace

NET Namespace 负责网络栈的隔离，包括网卡、路由表和 iptables 规则等。

NET 的作用

隔离网络栈，每个容器拥有独立的网络环境。

NET 隔离效果



NET 关键点

- 每个容器有独立的网卡、IP、路由表、iptables 规则
- 多个容器可以监听相同端口 (如都监听 80)
- Docker 使用 veth pair 连接容器网络和宿主机网桥

12.2.5 MNT Namespace

MNT Namespace 负责文件系统挂载点的隔离，确保容器看到独立的文件系统视图。

MNT 的作用

隔离文件系统挂载点，每个容器有自己的根目录。

MNT 隔离效果

```
宿主机文件系统:                    容器内看到的:
/                                  /  ← 容器的根目录
├── bin/                           ├── bin/
├── home/                          ├── home/
├── var/                           ├── var/
│   └── lib/                       │   └── lib/
│       └── docker/                │
│           └── overlay2/          │
│               └── merged/ ──────┼── 这个目录成为容器的 /
└── ...                            └── ...
```

与 chroot 的区别

特性	chroot	MNT Namespace
安全性	可以逃逸	更安全
挂载隔离	无	完全隔离
/proc/mounts	共享	独立

12.2.6 UTS Namespace

UTS Namespace 主要用于隔离主机名和域名。

UTS 的作用

隔离主机名和域名，让每个容器可以有自己的主机名。

UTS 隔离效果

```
## 宿主机

$ hostname
my-server

## 容器内

$ docker run --hostname mycontainer ubuntu hostname
mycontainer
```

UTS = “UNIX Time-sharing System”，是历史遗留的名称。

12.2.7 IPC Namespace

IPC Namespace 用于隔离进程间通信资源，如 System V IPC 和 POSIX 消息队列。

IPC 的作用

隔离 System V IPC 和 POSIX 消息队列。

隔离的资源

- 信号量 (semaphores)
- 消息队列 (message queues)
- 共享内存 (shared memory)

IPC 关键点

- 同一容器内的进程可以通过 IPC 通信
- 不同容器的进程无法通过 IPC 通信 (除非显式共享)

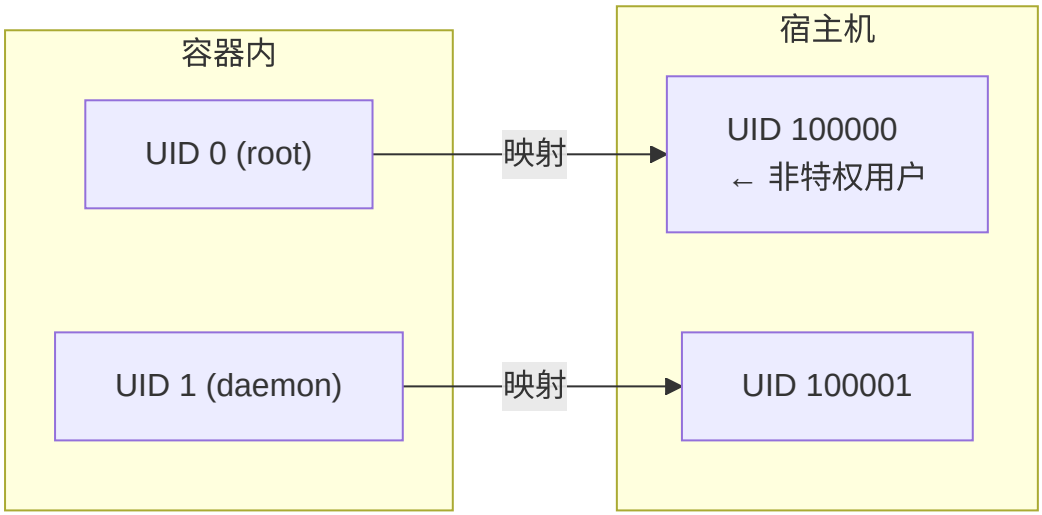
12.2.8 USER Namespace

USER Namespace 允许将容器内的用户 ID 映射到宿主机的不同用户 ID。

USER 的作用

隔离用户和组 ID，实现权限隔离。

USER 隔离效果



安全意义

容器内的 root 用户可以映射为宿主机上的普通用户，即使容器被突破，攻击者在宿主机上也只有普通权限。

> 💡 笔者建议：生产环境建议启用 User Namespace，增强安全性。

12.2.9 动手实验：体验 Namespace

使用 `unshare` 命令可以在不使用 Docker 的情况下体验 Namespace：

实验 1：UTS Namespace

```
## 创建新的 UTS namespace 并启动 shell

$ sudo unshare --uts /bin/bash

## 修改主机名（只影响这个 namespace）

$ hostname container-test
$ hostname
container-test

## 退出后查看宿主机主机名（未改变）

$ exit
$ hostname
my-server
```

实验 2：PID Namespace

```
## 创建新的 PID 和 MNT namespace

$ sudo unshare --pid --mount --fork /bin/bash

## 挂载新的 /proc

$ mount -t proc proc /proc

## 查看进程（只能看到当前 shell）

$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   8960  4516 pts/0    S    10:00   0:00 /bin/bash
root         8  0.0  0.0  10072  3200 pts/0    R+   10:00   0:00 ps aux
```

实验 3：NET Namespace

```
## 创建新的网络 namespace

$ sudo unshare --net /bin/bash

## 查看网络接口（只有 lo）

$ ip addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

12.2.10 Namespace 的局限性

Namespace 提供了隔离但不是安全边界：

方面	说明
共享内核	所有容器共享宿主机内核，内核漏洞可能影响所有容器
部分资源未隔离	/proc、/sys 部分内容仍可见；Time Namespace (Linux 5.6+) 虽已可用，但 Docker 默认不启用
非虚拟化	比虚拟机隔离性弱

> 需要更强隔离时，可考虑 gVisor、Kata Containers 等安全容器方案。

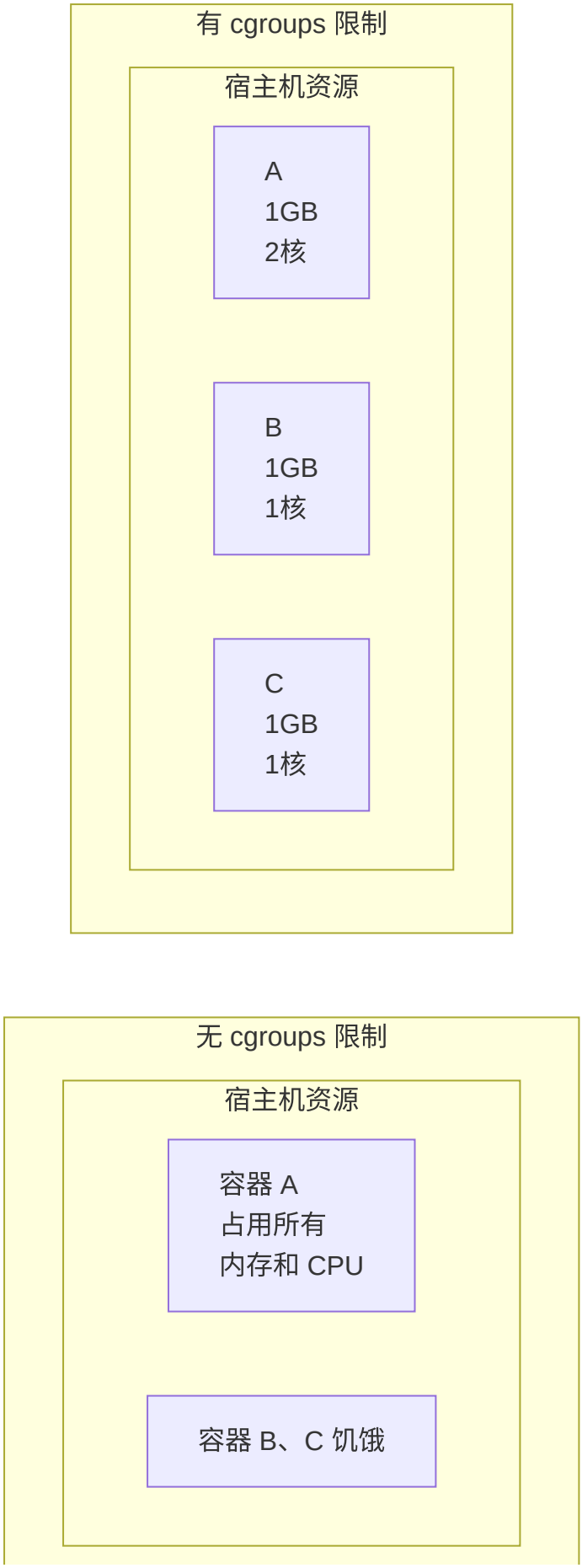
12.3 控制组

控制组 (Cgroups) 是 Linux 内核提供的另一种关键机制，主要用于资源的限制和审计。

12.3.1 什么是控制组

控制组 (Control Groups，简称 cgroups) 是 Linux 内核的一个特性，用于 **限制、记录和隔离** 进程组的资源使用 (CPU、内存、磁盘 I/O、网络等)。

> **核心作用**：让多个容器公平共享宿主机资源，防止单个容器耗尽系统资源。



Built with mdPress

12.3.2 cgroups 的历史

时间	事件
2006	Google 工程师提出 “process containers” 概念
2007	为避免与 Linux 容器概念混淆，更名为 “control groups” (cgroups)
2008	Linux 2.6.24（2008年1月）正式合并 cgroups v1
2016	Linux 4.5 引入 cgroups v2
现在	Docker 在宿主机支持 cgroups v2 时会自动使用 v2，否则回退到 v1

12.3.3 cgroups 可以限制的资源

资源类型	子系统	说明
CPU	`cpu`, `cpuset`	CPU 使用时间和核心分配
内存	`memory`	内存使用上限和 swap
块设备 I/O	`blkio`	磁盘读写速度限制
网络	`net_cls`, `net_prio`	网络带宽优先级
进程数	`pids`	限制进程/线程数量

Built with mdPress

12.3.4 Docker 中的资源限制

Docker 提供了丰富的参数来配置容器的资源限制，主要包括内存、CPU、磁盘 I/O 等。

内存限制

```
## 限制容器最多使用 512MB 内存

$ docker run -m 512m myapp

## 限制内存 + swap

$ docker run -m 512m --memory-swap 1g myapp

## 软限制（超过时警告，不会 OOM Kill）

$ docker run --memory-reservation 256m myapp
```

参数	说明
`-m` / `--memory`	硬限制 (超过会 OOM Kill)
`--memory-swap`	内存 + swap 总限制
`--memory-reservation`	软限制 (内存竞争时生效)
`--oom-kill-disable`	禁用 OOM Killer (谨慎使用)

CPU 限制

```
## 限制使用 1.5 个 CPU 核心

$ docker run --cpus=1.5 myapp

## 限制使用 CPU 0 和 1

$ docker run --cpuset-cpus="0,1" myapp

## 设置 CPU 使用权重（相对值，默认 1024）

$ docker run --cpu-shares=512 myapp
```

参数	说明
`--cpus`	限制 CPU 核心数 (如 1.5)
`--cpuset-cpus`	绑定到特定 CPU 核心
`--cpu-shares`	CPU 时间片权重 (相对值)
`--cpu-period` / `--cpu-quota`	精细控制 CPU 配额

磁盘 I/O 限制

```
## 限制设备写入速度为 10MB/s

$ docker run --device-write-bps /dev/sda:10mb myapp

## 限制设备读取速度

$ docker run --device-read-bps /dev/sda:10mb myapp

## 限制 IOPS

$ docker run --device-write-iops /dev/sda:100 myapp
```

进程数限制

```
## 限制最多 100 个进程

$ docker run --pids-limit=100 myapp
```

12.3.5 查看容器资源使用

```
## 实时监控所有容器的资源使用

$ docker stats
CONTAINER ID   NAME    CPU %   MEM USAGE / LIMIT   MEM %   NET I/O        BLOCK I/O
abc123         web     0.50%   45.5MiB / 512MiB    8.89%   1.2kB / 0B     0B / 0B
def456         db      2.30%   256MiB / 1GiB       25.00%  5.6kB / 3.2kB  4.1MB / 2.3MB

## 查看特定容器

$ docker stats mycontainer

## 查看容器的 cgroup 配置

$ docker inspect mycontainer --format '{{json .HostConfig}}' | jq
```

12.3.6 资源限制的效果

内存超限

```
## 启动限制 100MB 内存的容器

$ docker run -m 100m stress --vm 1 --vm-bytes 200M

## 容器会被 OOM Killer 杀死

$ docker ps -a
CONTAINER ID   STATUS                      NAMES
abc123         Exited (137) 5 seconds ago  hopeful_darwin

## 137 = 128 + 9, 表示被 SIGKILL (9) 杀死

...
```

CPU 限制验证

```
## 不限制 CPU

$ docker run --rm stress --cpu 4

## 占满所有 CPU

## 限制为 1 个核心

$ docker run --rm --cpus=1 stress --cpu 4

## 只能使用约 100% CPU (1 个核心)

...
```

12.3.7 cgroups v1 vs v2

特性	cgroups v1	cgroups v2
层级结构	多层级 (每个资源单独)	统一层级
管理复杂度	复杂	简化
资源分配	基于层级	基于子树
PSI (压力监控)	❌	✅
rootless 容器	部分支持	完整支持

Docker 对 cgroups v2 的支持

Docker 20.10+ 开始支持 cgroups v2（如果系统支持），提供更好的性能和资源隔离。如果需要明确控制或回退到 v1，可以通过 Docker 守护进程配置文件 `/etc/docker/daemon.json` 修改 `cgroup-driver` 参数：

```
{
  "cgroup-driver": "systemd"
}
```

常见的 `cgroup-driver` 值包括 `systemd` (推荐) 和 `cgroupfs`。重启 Docker 守护进程后生效。

检查系统使用的版本

```
## 查看 cgroup 版本

$ mount | grep cgroup
cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,relatime)

## 如果显示 cgroup2 表示 v2

## 或者

$ cat /proc/filesystems | grep cgroup
nodev   cgroup
nodev   cgroup2
```

12.3.8 在 Compose 中设置限制

在 Compose 中设置限制配置如下：

```
services:
  web:
    image: nginx
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.25'
          memory: 256M
```

12.3.9 最佳实践

在使用 Cgroups 限制资源时，遵循一些最佳实践可以避免潜在的问题。

1. 始终设置内存限制

```
## 防止 OOM 影响宿主机

$ docker run -m 1g myapp
```

2. 为关键应用设置 CPU 保证

```
$ docker run --cpus=2 --cpu-shares=2048 critical-app
```

3. 监控资源使用

```
## 配合 Prometheus + cAdvisor 监控

$ docker run -d --name cadvisor \
    -v /:/rootfs:ro \
    -v /var/run:/var/run:ro \
    -v /sys:/sys:ro \
    -v /var/lib/docker:/var/lib/docker:ro \
    ghcr.io/google/cadvisor
```

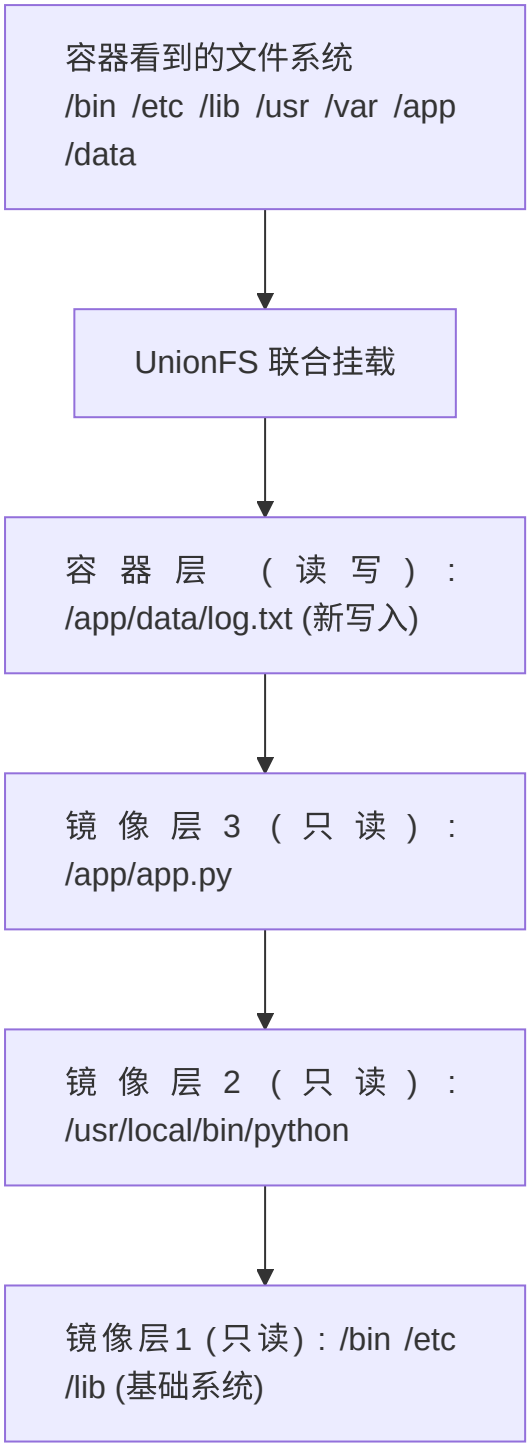
12.4 联合文件系统

联合文件系统 (UnionFS) 是 Docker 镜像分层存储的基础，它允许将多个目录挂载为同一个虚拟文件系统。

12.4.1 什么是联合文件系统

联合文件系统 (UnionFS) 是一种 **分层、轻量级** 的文件系统，它将多个目录 “联合” 挂载到同一个虚拟目录，形成一个统一的文件系统视图。

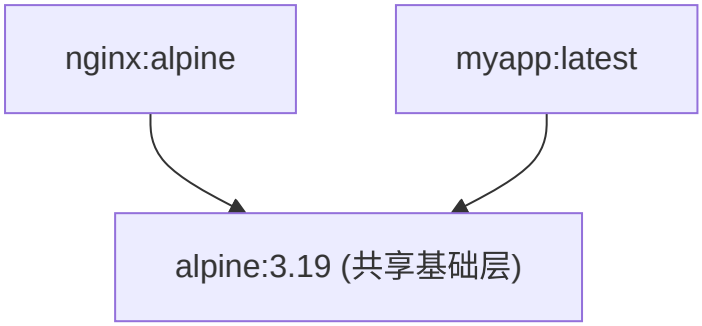
> **核心思想**：将多个只读层叠加，最上层可写，形成完整的文件系统。



12.4.2 为什么 Docker 使用联合文件系统

Docker 选择联合文件系统作为其存储驱动，主要基于以下几个核心优势。

1. 镜像分层复用



多个镜像共享相同的底层，节省磁盘空间。

2. 快速构建

每个 Dockerfile 指令创建一层，只有变化的层需要重建：

```
FROM node:22          # 层1：基础镜像
COPY package.json ./  # 层2：依赖定义
RUN npm install       # 层3：安装依赖
COPY . .              # 层4：应用代码
```

代码变化时，只需重建层 4，层 1-3 使用缓存。

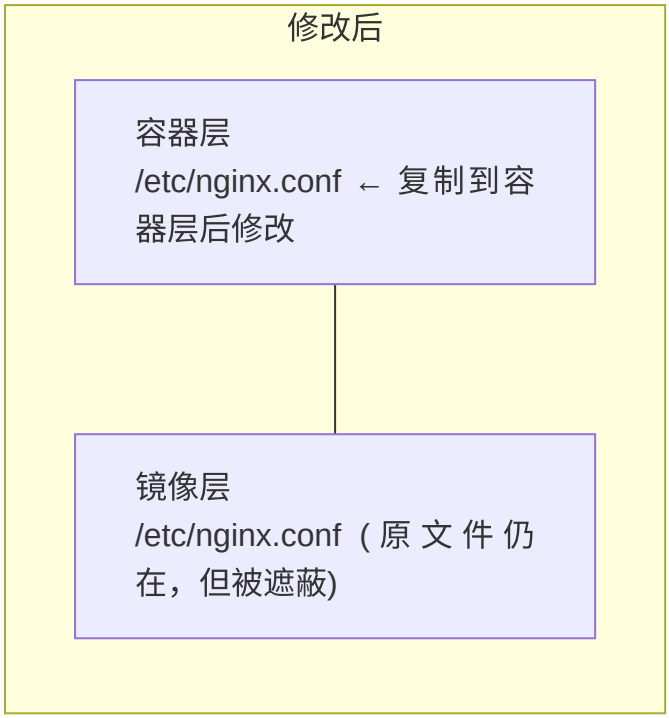
3. 容器启动快

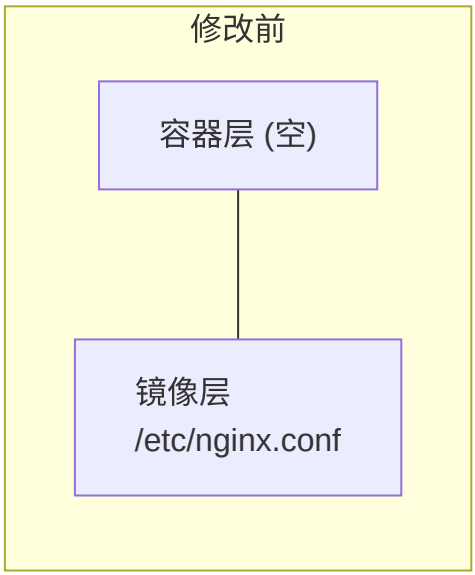
容器启动时不需要复制镜像，只需：

1. 在镜像层上创建一个薄的可写层
2. 联合挂载所有层

12.4.3 Copy-on-Write：写时复制

当容器修改只读层中的文件时：





流程：

1. 从只读层读取文件
2. 复制到容器的可写层
3. 在可写层中修改
4. 后续读取使用可写层的版本

Built with mdPress

12.4.4 Docker 支持的存储驱动

Docker 的存储驱动经历了从早期各式各样的机制（如 aufs, devicemapper），到被广泛使用的现代经典 graph driver (`overlay2`)，再到当下（Engine v29.x 及以后）**在新安装场景中默认启用的 containerd 镜像存储引擎（containerd image store）** 的演进。

存储后端 / 驱动	核心特性说明	推荐程度
containerd image store	(v29.x 新一代默认后端，新装默认) 基于 containerd 的 snapshotters，原生支持 OCI image index、多架构镜像与 Attestations 构建溯源元数据存储。	✅**强烈推荐 (现代默认)**
overlay2	(经典 Graph Driver) 传统架构下的现代 Linux 默认驱动，性能优秀，但在处理复杂溯源元数据（索引）时受限。	✅**推荐 (主要后备)**
aufs	早期默认，兼容性好	遗留系统
btrfs/zfs	使用原生稳定文件系统快照能力	特定场景
devicemapper	块设备级存储	遗留系统 (已被逐步弃用)
vfs	不使用 CoW，每层完整复制	仅测试

Classic Graph Drivers 与 Snapshotters 的核心差异

传统模型（如 `overlay2`）将镜像拉取解包的过程由 Docker 的 graph drivers 处理。而新的 `containerd image store` 则将这一职责彻底下放给了 `containerd` 自身的 `snapshotters`（底层在 Linux 发行版通常依然利用操作系统的 overlayfs）。这种架构改变带来了：

1. 本地免拉取查看多平台镜像 index manifest 与 attestations (SBOM、Provenance)。
2. 避免了以前绕过 CRI 获取本地镜像的问题，带来更好的原生 Kubernetes 生态兼容性。

Built with mdPress

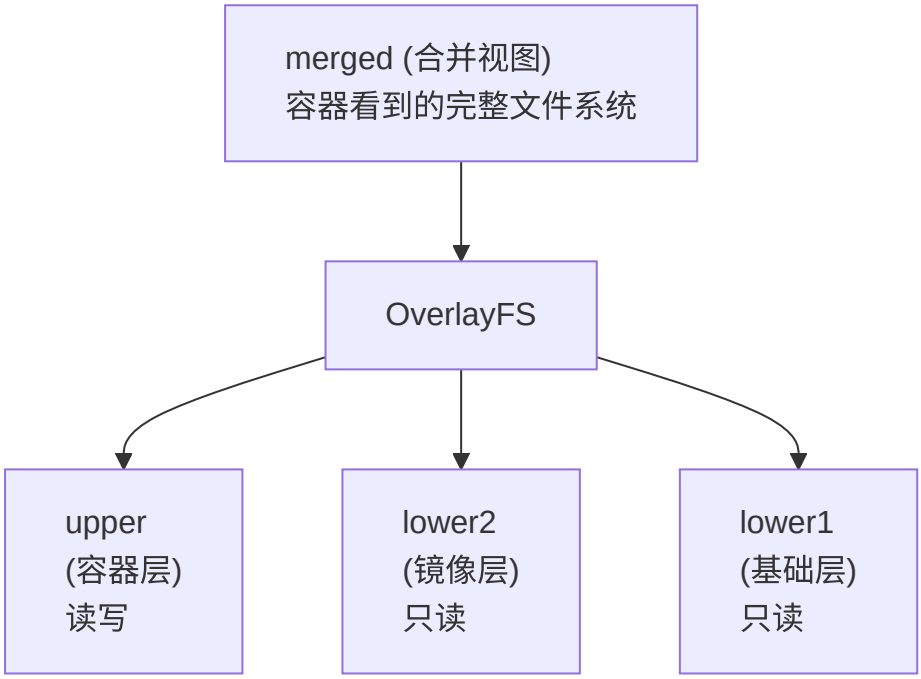
查看当前存储驱动与后端

```
## 查看默认存储驱动
$ docker info | grep "Storage Driver"
Storage Driver: overlay2

## 在 Engine v29.x 中，可以通过如下输出验证是否开启了 containerd 镜像后端：
$ docker info | grep "containerd image store"
 containerd image store: true
```

12.4.5 overlay2 工作原理

overlay2 是目前最推荐的存储驱动：



- **lowerdir**：只读的镜像层 (可以有多个)
- **upperdir**：可写的容器层
- **workdir**：OverlayFS 的工作目录
- **merged**：联合挂载后的视图

文件操作行为

操作	行为
读取	从上到下查找第一个匹配的文件
创建	在 upper 层创建
修改	如果在 lower 层，先复制到 upper 层再修改
删除	在 upper 层创建 whiteout 文件标记删除

12.4.6 查看镜像层

```
## 查看镜像的层信息

$ docker history nginx:alpine
IMAGE          CREATED       CREATED BY                                      SIZE
a6eb2a334a9f   2 weeks ago   CMD ["nginx" "-g" "daemon off;"]                0B
<missing>      2 weeks ago   STOPSIGNAL SIGQUIT                              0B
<missing>      2 weeks ago   EXPOSE map[80/tcp:{}]                           0B
<missing>      2 weeks ago   ENTRYPOINT ["/docker-entrypoint.sh"]            0B
<missing>      2 weeks ago   COPY 30-tune-worker-processes.sh /docker-ent…   4.62kB
...

## 查看层的存储位置

$ docker inspect nginx:alpine --format '{{json .GraphDriver.Data}}' | jq
{
  "LowerDir": "/var/lib/docker/overlay2/.../diff:/var/lib/docker/overlay2/.../diff",
  "MergedDir": "/var/lib/docker/overlay2/.../merged",
  "UpperDir": "/var/lib/docker/overlay2/.../diff",
  "WorkDir": "/var/lib/docker/overlay2/.../work"
}
```

12.4.7 最佳实践

为了构建高效、轻量的镜像，我们在使用联合文件系统时应注意以下几点。

1. 减少镜像层数

```
## ❌ 每条命令创建一层

RUN apt-get update
RUN apt-get install -y nginx
RUN rm -rf /var/lib/apt/lists/*

## ✅ 合并为一层

RUN apt-get update && \
    apt-get install -y nginx && \
    rm -rf /var/lib/apt/lists/*
```

2. 避免在容器中写入大量数据

容器层的写入性能低于直接写入。大量数据应使用：

- 数据卷 (Volume)
- 绑定挂载 (Bind Mount)

3. 使用 .dockerignore

排除不需要的文件可以：

- 减小构建上下文
- 避免创建不必要的层

12.5 容器格式

Docker 容器格式的演进

最初，Docker 采用了 LXC 中的容器格式。从 0.7 版本以后开始去除 LXC 的依赖，转而使用自行开发的 libcontainer。从 1.11 开始，则进一步演进为使用 runC 和 containerd。

关键组件说明

LXC

Docker 早期版本（0.1-0.7）直接使用 LXC 作为容器运行时，利用 Linux Namespaces 和 Cgroups 实现容器隔离。

libcontainer

- Docker 自行开发的容器库
- 提供了容器的通用接口
- 不依赖于特定的 Linux 容器实现
- 更灵活和可控

runC

- OCI（Open Container Initiative）标准实现
- 轻量级的容器运行时
- 独立的二进制文件，可单独使用
- 基于 libcontainer 发展而来

containerd

- Docker 开源的容器运行时
- 提供了容器的完整生命周期管理
- 支持 runC 和其他 OCI 兼容的运行时
- 在 Kubernetes 等编排系统中广泛使用

Built with mdPress

容器规范标准

Docker 积极参与 Open Container Initiative (OCI) 的制定，推动了以下规范的发展：

- **Image Spec**：容器镜像格式规范
- **Runtime Spec**：容器运行时接口规范
- **Distribution Spec**：容器镜像分发规范

架构演变的优势

从 LXC → libcontainer → runC/containerd 的演变提供了以下优势：

1. 减少外部依赖
2. 提高运行效率
3. 遵循行业标准（OCI）
4. 增强可移植性和互操作性
5. 支持多种容器运行时选择

Built with mdPress

12.6 网络

Docker 的网络实现其实就是利用了 Linux 上的网络命名空间和虚拟网络设备 (特别是 veth pair)。建议先熟悉了解这两部分的基本概念再阅读本章。

12.6.1 基本原理

首先，要实现网络通信，机器需要至少一个网络接口 (物理接口或虚拟接口) 来收发数据包；此外，如果不同子网之间要进行通信，需要路由机制。

Docker 中的网络接口默认都是虚拟的接口。虚拟接口的优势之一是转发效率较高。 Linux 通过在内核中进行数据复制来实现虚拟接口之间的数据转发，发送接口的发送缓存中的数据包被直接复制到接收接口的接收缓存中。对于本地系统和容器内系统看来就像是一个正常的以太网卡，只是它不需要真正同外部网络设备通信，速度要快很多。

Docker 容器网络就利用了这项技术。它在本地主机和容器内分别创建一个虚拟接口，并让它们彼此连通 (这样的一对接口叫做 `veth pair`)。

12.6.2 创建网络参数

Docker 创建一个容器的时候，会执行如下操作：

- 创建一对虚拟接口，分别放到本地主机和新容器中；
- 本地主机一端桥接到默认的 docker0 或指定网桥上，并具有一个唯一的名字，如 veth65f9；
- 容器一端放到新容器中，并修改名字作为 eth0，这个接口只在容器的命名空间可见；
- 从网桥可用地址段中获取一个空闲地址分配给容器的 eth0，并配置默认路由到桥接网卡 veth65f9。

完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

可以在 `docker run` 的时候通过 `--network` 参数来指定容器的网络配置，有 4 个可选值：

- `--network=bridge` 这个是默认值，连接到默认的网桥。
- `--network=host` 告诉 Docker 不要将容器网络放到隔离的命名空间中，即不要容器化容器内的网络。此时容器使用本地主机的网络，它拥有完全的本地主机接口访问权限。容器进程可以跟主机其它 root 进程一样可以打开低范围的端口，可以访问本地网络服务比如 D-bus，还可以让容器做一些影响整个主机系统的事情，比如重启主机。因此使用这个选项的时候要非常小心。如果进一步的使用 `--privileged=true`，容器会被允许直接配置主机的网络堆栈。
- `--network=container:NAME_or_ID` 让 Docker 将新建容器的进程放到一个已存在容器的网络栈中，新容器进程有自己的文件系统、进程列表和资源限制，但会和已存在的容器共享 IP 地址和端口等网络资源，两者进程可以直接通过 `lo` 环回接口通信。
- `--network=none` 让 Docker 将新容器放到隔离的网络栈中，但是不进行网络配置。之后，用户可以自己进行配置。

12.6.3 网络配置细节

用户使用 `--net=none` 后，可以自行配置网络，让容器达到跟平常一样具有访问网络的权限。通过这个过程，可以了解 Docker 配置网络的细节。

首先，启动一个 `/bin/bash` 容器，指定 `--net=none` 参数。

```
$ docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#
```

在本地主机查找容器的进程 id，并为它创建网络命名空间。

```
$ docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

检查桥接网卡的 IP 和子网掩码信息。

```
$ ip addr show docker0
21: docker0: ...
inet 172.17.0.1/16 scope global docker0
...
```

创建一对 “veth pair” 接口 A 和 B，绑定 A 到网桥 `docker0`，并启用它

```
$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up
```

将 B 放到容器的网络命名空间，命名为 eth0，启动它并配置一个可用 IP (桥接网段) 和默认网关。

```
$ sudo ip link set B netns $pid
$ sudo ip netns exec $pid ip link set dev B name eth0
$ sudo ip netns exec $pid ip link set eth0 up
$ sudo ip netns exec $pid ip addr add 172.17.0.99/16 dev eth0
$ sudo ip netns exec $pid ip route add default via 172.17.0.1
```

以上，就是 Docker 配置网络的具体过程。

当容器结束后，Docker 会清空容器，容器内的 eth0 会随网络命名空间一起被清除，A 接口也被自动从 `docker0` 卸载。

此外，用户可以使用 `ip netns exec` 命令来在指定网络命名空间中进行配置，从而配置容器内的网络。

本章小结

本章深入介绍了 Docker 的底层实现，包括命名空间、控制组和联合文件系统三大核心技术。

技术	作用	要点
Namespace	资源隔离	PID、NET、MNT、UTS、IPC、USER 六种命名空间
Cgroups	资源限制	限制 CPU、内存、磁盘 I/O、进程数
Union FS	分层存储	overlay2 为推荐驱动，支持 Copy-on-Write

Namespace	隔离内容	一句话说明
PID	进程 ID	容器有自己的进程树
NET	网络	容器有自己的 IP 和端口
MNT	文件系统	容器有自己的根目录
UTS	主机名	容器有自己的 hostname
IPC	进程间通信	容器间 IPC 隔离
USER	用户 ID	容器 root ≠ 宿主机 root

Built with mdPress

资源	限制参数	示例
内存	`-m`	`-m 512m`
CPU 核心数	`--cpus`	`--cpus=1.5`
CPU 绑定	`--cpuset-cpus`	`--cpuset-cpus="0,1"`
磁盘 I/O	`--device-write-bps`	`--device-write-bps /dev/sda:10mb`
进程数	`--pids-limit`	`--pids-limit=100`

延伸阅读

- 命名空间：资源隔离机制详解
- 控制组 (Cgroups)：资源限制机制
- 联合文件系统：分层存储的实现
- 安全：容器安全实践
- 镜像：理解镜像分层
- 容器：容器存储层
- 构建镜像：Dockerfile 层的创建

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十三章 容器编排基础

Kubernetes 是 Google 发起的开源容器编排系统，它支持多种云平台与私有数据中心。

Kubernetes 负责对容器工作负载进行调度与编排，其目的是让用户通过集群声明式地管理应用，而无需手动干预每个容器的生命周期细节。

Kubernetes 的最小调度单位是 Pod。一个 Pod 由一组紧密协作的容器构成，它们共享网络命名空间、IP 以及部分存储资源，也可以根据需要对 Pod 进行端口映射。

如果你已经熟悉 Docker，可以用以下对照来理解 Kubernetes 的核心概念：Docker 中的“容器”对应 Kubernetes 的 Pod（一个或多个容器的组合）；`docker-compose.yml` 的角色类似于 Kubernetes 的 `Deployment` + `Service` 声明；`docker run` 的端口映射和网络配置，在 Kubernetes 中由 `Service` 和 `Ingress` 接管。掌握这些映射关系，有助于从单机 Docker 平滑过渡到集群编排。

本章将分为 5 节介绍 Kubernetes：

- 简介
- 基本概念
- 架构设计
- 高级特性
- 实战练习

Built with mdPress

13.1 简介

如图 13-1 所示，Kubernetes 使用舵手图标作为项目标识。



图 13-1：Kubernetes 项目标识

13.1.1 什么是 Kubernetes

Kubernetes (常简称为 K8s) 是 Google 开源的容器编排引擎。如果说 Docker 解决了 “如何打包和运送集装箱” 的问题，那么 Kubernetes 解决的就是 “如何管理海量集装箱的调度、运行和维护” 的问题。

它不仅仅是一个编排系统，更是一个 **云原生应用操作系统**。

> **名字由来**：Kubernetes 在希腊语中意为 “舵手” 或 “飞行员”。K8s 是因为 k 和 s 之间有 8 个字母。

13.1.2 为什么需要 Kubernetes

当我们在单机运行几个容器时，Docker Compose 就足够了。但在生产环境中，我们需要面对：

- **多主机调度**：容器应该运行在哪台机器上？
- **自动恢复**：容器崩溃了怎么办？节点挂了怎么办？
- **服务发现**：容器 IP 变了，其他服务怎么找到它？
- **负载均衡**：流量大了，如何分发给多个副本？
- **滚动更新**：如何不中断服务升级应用？

Kubernetes 完美解决了这些问题。

Built with mdPress

13.1.3 核心概念

Kubernetes 的核心概念包括 Pod、Node、Deployment、Service 和 Namespace 等，这些是学习和使用 Kubernetes 的基础。

Pod：豆荚

Kubernetes 的最小调度单位。一个 Pod 可以包含一个或多个紧密协作的容器 (共享网络和存储)。就像豌豆荚里的豌豆一样。

Node：节点

运行 Pod 的物理机或虚拟机。

Deployment：部署

定义应用的期望状态 (如：需要 3 个副本，镜像版本为 v1)。K8s 会持续确保当前状态符合期望状态。

Service：服务

定义一组 Pod 的访问策略。提供稳定的 Cluster IP 和 DNS 名称，负责负载均衡。

Namespace：命名空间

用于多租户资源隔离。

13.1.4 Docker 用户如何过渡

如果你已经熟悉 Docker，学习 K8s 会很容易：

Docker 概念	Kubernetes 概念	说明
Container	Pod	K8s 增加了一层 Pod 包装
Volume	PersistentVolume	K8s 的存储更加抽象和强大
Network	Service/Ingress	K8s 的网络模型更扁平
Compose	Deployment + Service	声明式配置的理念是一致的

13.1.5 架构

Kubernetes 也是 C/S 架构，由 **Control Plane (控制平面)** 和 **Worker Node (工作节点)** 组成：

- **Control Plane**：负责决策 (API Server，Scheduler，Controller Manager，etcd)
- **Worker Node**：负责干活 (Kubelet，Kube-proxy，Container Runtime)

13.1.6 学习建议

Kubernetes 的学习曲线较陡峭。建议的学习路径：

1. **理解基本概念**：Pod，Deployment，Service
2. **动手实践**：使用 Minikube 或 Kind 在本地搭建集群
3. **部署应用**：编写 YAML 部署一个无状态应用
4. **深入原理**：网络模型、存储机制、调度算法

13.1.7 延伸阅读

- Minikube 安装：本地体验 K8s
- Kubernetes 官网：官方文档

Built with mdPress

13.2 基本概念

如图 13-2 所示，Kubernetes 由控制平面与工作节点构成。

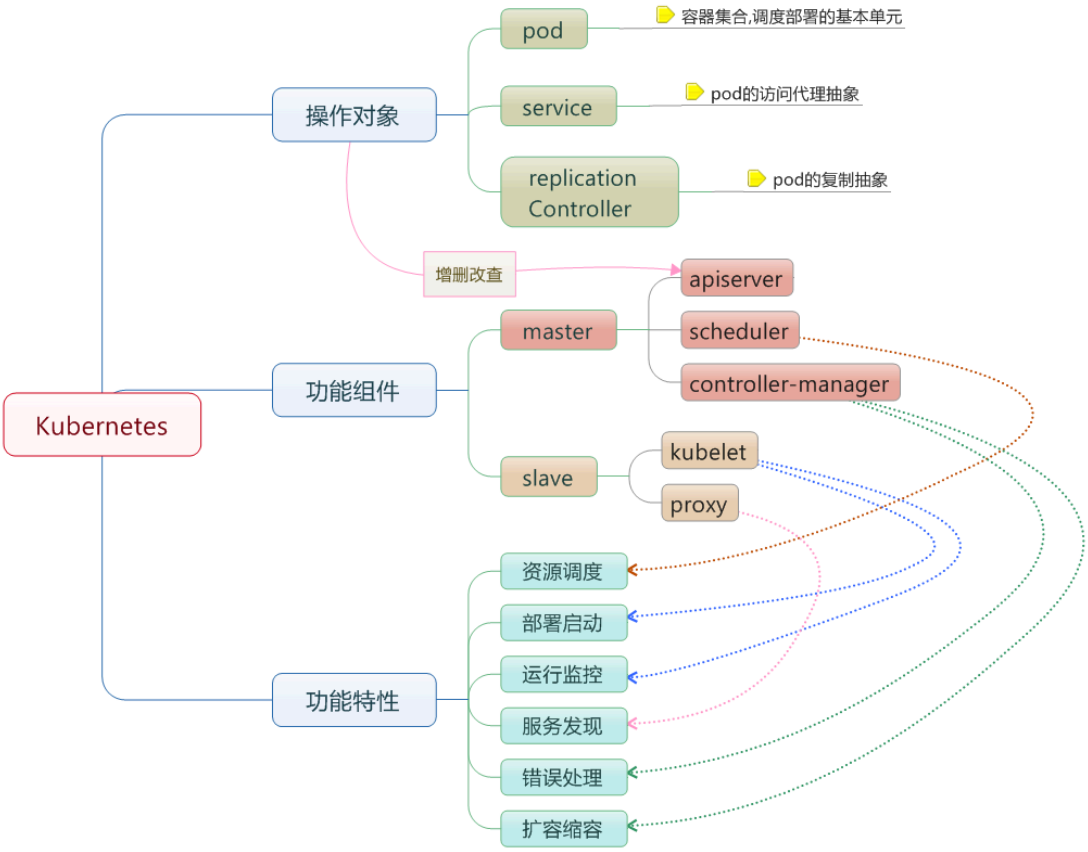


图 13-2：Kubernetes 基本概念示意图

- 节点 (`Node`)：一个节点是一个运行 Kubernetes 中的主机。
- 容器组 (`Pod`)：一个 Pod 对应于由若干容器组成的一个容器组，同个组内的容器共享一个存储卷 (volume)。
- 容器组生命周期 (`pod-states`)：包含所有容器状态集合，包括容器组状态类型，容器组生命周期，事件，重启策略，以及 replication controllers。
- Replication Controllers：主要负责指定数量的 pod 在同一时间一起运行。
- 服务 (`services`)：一个 Kubernetes 服务是容器组逻辑的高级抽象，同时也对外提供访问容器组的策略。
- 卷 (`volumes`)：一个卷就是一个目录，容器对其有访问权限。
- 标签 (`labels`)：标签是用来连接一组对象的，比如容器组。标签可以被用来组织和选择子对象。
- 接口权限 (`accessing_the_api`)：端口，IP 地址和代理的防火墙规则。
- web 界面 (`ux`)：用户可以通过 web 界面操作 Kubernetes。
- 命令行操作 (`cli`)：`kubectl` 命令。

13.2.1 节点

在 `Kubernetes` 中，节点是实际工作的点，节点可以是虚拟机或者物理机器，依赖于一个集群环境。每个节点都有一些必要的组件以运行容器组，并且它们都可以通过控制平面来管理。必要组件包括 kubelet、容器运行时（如 containerd 或 CRI-O）和 kube-proxy；Docker 不再是 Kubernetes 节点的默认基线运行时。

容器状态

容器状态用来描述节点的当前状态。现在，其中包含三个信息：

主机 IP

主机 IP 需要云平台来查询，`Kubernetes` 把它作为状态的一部分来保存。如果 `Kubernetes` 没有运行在云平台上，节点 ID 就是必需的。IP 地址可以变化，并且可以包含多种类型的 IP 地址，如公共 IP，私有 IP，动态 IP，ipv6 等等。

节点状态

注：`Pending` / `Running` / `Succeeded` / `Failed` / `Unknown` 是 **Pod 的生命周期阶段**（`pod.status.phase`），不是节点状态。节点本身没有 `Terminated` 状态。

Built with mdPress

节点的状态通过一组条件（Conditions）来描述。主要条件包括 `Ready`（kubelet 健康且可以接收 Pod）、`MemoryPressure`（内存不足）、`DiskPressure`（磁盘不足）和 `PIDPressure`（进程数过多）等。其中 `Ready` 条件最为关键：值为 `True` 表示节点健康可调度，`False` 表示节点异常，`Unknown` 表示节点控制器超过一定时间未收到心跳。被标记为 `Unknown` 较长时间的节点，其上的 Pod 会被驱逐重新调度到其他节点。

节点管理

节点并非 Kubernetes 创建，而是由云平台创建，或者就是物理机器、虚拟机。在 Kubernetes 中，节点仅仅是一条记录，节点创建之后，Kubernetes 会检查其是否可用。可以通过 `kubectl` 查看节点信息：

```
$ kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
control-plane  Ready    control-plane   10d   v1.36.0
worker-1       Ready    <none>          10d   v1.36.0
worker-2       Ready    <none>          10d   v1.36.0
```

每个节点的详细信息以如下结构保存：

```yaml
apiVersion: v1
kind: Node
metadata:
  name: worker-1
  labels:
    kubernetes.io/os: linux
status:
  capacity:
    cpu: "4"
    memory: 8Gi
  conditions:
    - type: Ready
      status: "True"
```

节点控制器

在 Kubernetes 控制平面中，节点控制器 (Node Controller) 负责管理节点的生命周期，主要包含：

- 集群范围内节点状态同步
- 单节点生命周期管理

节点控制器会持续监控节点的健康状态。当节点变为不可达时，控制器会等待一个超时期限，然后将该节点上的 Pod 标记为失败，并触发重新调度。可以使用 `kubectl` 来管理节点，例如标记节点为不可调度或排空节点上的工作负载：

```
## 标记节点为不可调度
$ kubectl cordon worker-1

## 排空节点上的 Pod
$ kubectl drain worker-1 --ignore-daemonsets
```

13.2.2 容器组

在 Kubernetes 中，使用的最小调度单位是容器组 (Pod)，它是创建、调度、管理的最小单位。一个 Pod 包含一个或多个紧密协作的容器，它们共享网络命名空间和存储卷。

Pod 通常不会被直接创建，而是通过 Deployment 等控制器来管理。当节点发生故障时，控制器会在其他可用节点上重新创建 Pod。

容器组设计的初衷

容器组 (Pod) 的设计主要是为了解决应用间的紧密协作和资源共享问题。

资源共享和通信

容器组主要是为了数据共享和它们之间的通信。

在一个容器组中，容器都使用相同的网络地址和端口，可以通过本地网络来相互通信。每个容器组都有独立的 IP，可用通过网络来和其他物理主机或者容器通信。

容器组有一组存储卷 (挂载点)，主要是为了让容器在重启之后可以不丢失数据。

容器组管理

容器组是一个应用管理和部署的高层次抽象，同时也是一组容器的接口。容器组是部署、水平放缩的最小单位。

容器组的使用

容器组可以通过组合来构建复杂的应用，典型的使用模式包含：

- 内容管理，文件和数据加载以及本地缓存管理等。
- 日志和检查点备份，压缩，快照等。
- 监听数据变化，跟踪日志，日志和监控代理，消息发布等。
- 代理，网桥
- 控制器，管理，配置以及更新

为什么不在一个容器里运行多个程序

1. **透明化**：为了使容器组中的容器保持一致的基础设施和服务，比如进程管理和资源监控。
2. **解耦依赖**：每个容器都可能独立地重新构建和发布。
3. **方便使用**：用户不必运行独立的程序管理，也不用担心每个应用程序的退出状态。
4. **高效**：考虑到基础设施有更多的职责，容器必须要轻量化。

容器组的生命状态

包括若干状态值：`Pending`、`Running`、`Succeeded`、`Failed`。

状态	说明
Pending	Pod 已被集群接受，但有一个或多个容器还没有运行起来（可能在拉取镜像）。
Running	Pod 已被调度到节点，并且所有容器都已启动。至少有一个容器处于运行状态。
Succeeded	Pod 中的所有容器都正常退出，且不会被重启。
Failed	Pod 中的所有容器都已终止，且至少有一个容器以失败状态退出。

容器组生命周期与重启策略

Pod 的重启策略 (`restartPolicy`) 决定了容器退出后的行为：

重启策略	容器正常退出	容器异常退出
Always (默认)	重启容器	重启容器
OnFailure	不重启	重启容器
Never	不重启	不重启

当节点故障或不可达时，节点控制器会将该节点上所有 Pod 的状态标记为 `Failed`。如果这些 Pod 由 Deployment 等控制器管理，控制器会自动在其他节点上重新创建。

Built with mdPress

13.2.3 Deployment 与 ReplicaSet

Deployment 是管理无状态应用的推荐方式，它通过 ReplicaSet 来确保指定数量的 Pod 副本始终在运行。

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.28
          ports:
            - containerPort: 80
```

Deployment 的核心能力包括：

- **副本管理**：确保始终有指定数量的 Pod 在运行
- **滚动更新**：逐步替换旧版本 Pod，实现零停机部署
- **回滚**：如果新版本出现问题，可以快速回滚到之前的版本

> 早期 Kubernetes 使用 Replication Controller (RC) 来管理副本，现已被 ReplicaSet/Deployment 取代。

13.2.4 StatefulSet

Deployment 适合无状态应用，而 **StatefulSet** 用于管理有状态应用（如数据库、消息队列）。与 Deployment 不同，StatefulSet 为每个 Pod 提供稳定的网络标识和持久化存储。

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:8.4
          volumeMounts:
            - name: data
              mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

StatefulSet 的核心特性：

- **稳定的网络标识**：Pod 名称按顺序编号（mysql-0, mysql-1, mysql-2），配合 Headless Service 提供可预测的 DNS 名称
- **有序部署与删除**：Pod 按序号顺序创建，逆序删除
- **持久化存储**：通过 `volumeClaimTemplates` 为每个 Pod 自动创建独立的 PVC

13.2.5 DaemonSet

DaemonSet 确保在集群的每个节点（或指定节点）上运行一个 Pod 副本。典型用途包括日志收集、监控代理和网络插件。

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
        - name: fluentd
          image: fluent/fluentd:v1.17
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

当新节点加入集群时，DaemonSet 会自动在该节点上创建 Pod；当节点被移除时，对应的 Pod 也会被回收。

13.2.6 Job 与 CronJob

Job 用于运行一次性任务，确保指定数量的 Pod 成功完成后自动退出。**CronJob** 则按照 cron 表达式周期性地创建 Job。

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-migration
spec:
  completions: 1
  template:
    spec:
      containers:
        - name: migrate
          image: myapp/migrate:latest
          command: ["python", "migrate.py"]
      restartPolicy: Never
  backoffLimit: 3
```

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-backup
spec:
  schedule: "0 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: myapp/backup:latest
          restartPolicy: OnFailure
```

Job 的 `backoffLimit` 控制失败重试次数，`completions` 指定需要成功完成的 Pod 数量。CronJob 适用于定时备份、报表生成等场景。

13.2.7 服务

服务 (Service) 定义了一组 Pod 的逻辑集合和访问策略。由于 Pod 的 IP 地址是动态分配的，Service 提供了一个稳定的访问入口。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

常见的 Service 类型：

类型	说明
ClusterIP	默认类型，仅集群内部可访问
NodePort	在每个节点上开放固定端口，集群外部可通过 `节点IP:端口` 访问
LoadBalancer	通过云平台的负载均衡器暴露服务

13.2.8 卷

卷 (Volume) 为 Pod 中的容器提供持久化存储。Kubernetes 支持多种卷类型：

卷类型	说明
emptyDir	临时存储，Pod 删除后数据丢失
hostPath	挂载节点上的文件或目录
PersistentVolumeClaim	使用持久卷声明，与底层存储解耦
configMap / secret	将配置或敏感数据挂载为文件

生产环境中，推荐使用 PersistentVolume (PV) 和 PersistentVolumeClaim (PVC) 来管理存储，实现存储资源与使用者的解耦。

13.2.9 标签

标签 (Label) 是附加到 Kubernetes 对象上的键值对，用于组织和选择对象子集。标签是 Kubernetes 中实现松耦合的关键机制。

```
## 为 Pod 添加标签
$ kubectl label pod my-pod env=production

## 通过标签选择器查询
$ kubectl get pods -l env=production
```

Service、Deployment 等资源都通过标签选择器 (`selector`) 来关联目标 Pod。

13.2.10 API 访问控制

Kubernetes API 的访问通过三个阶段进行控制：

1. **认证 (Authentication)**：验证请求者的身份（如证书、Token、OIDC）
2. **授权 (Authorization)**：判断请求者是否有权限执行操作（通常使用 RBAC）
3. **准入控制 (Admission Control)**：在请求被持久化之前对其进行校验或修改

13.2.11 Dashboard

Kubernetes Dashboard 是一个基于 Web 的用户界面，用于部署容器化应用、监控集群资源和排查问题。Dashboard 的部署方法详见部署 Dashboard 章节。

13.2.12 命令行工具 kubectl

`kubectl` 是 Kubernetes 的命令行工具，用于与集群进行交互。常用命令如下：

```
## 查看集群中的资源
$ kubectl get pods,deployments,services,nodes

## 创建资源
$ kubectl apply -f deployment.yaml

## 查看 Pod 日志
$ kubectl logs my-pod

## 进入 Pod 执行命令
$ kubectl exec -it my-pod -- /bin/sh

## 查看资源详情
$ kubectl describe pod my-pod
```

更多 kubectl 操作详见kubectl 命令行章节。

13.3 架构设计

任何优秀的项目都离不开优秀的架构设计。本小节将介绍 Kubernetes 在架构方面的设计考虑。

13.3.1 基本考虑

如果让我们自己从头设计一套容器管理平台，有如下几个方面是很容易想到的：

- 分布式架构，保证扩展性；
- 逻辑集中式的控制平面 + 物理分布式的运行平面；
- 一套资源调度系统，管理哪个容器该分配到哪个节点上；
- 一套对容器内服务进行抽象和 HA 的系统。

13.3.2 运行原理

如图 13-3 所示，该图完整展示了 Kubernetes 的运行原理。

Built with mdPress

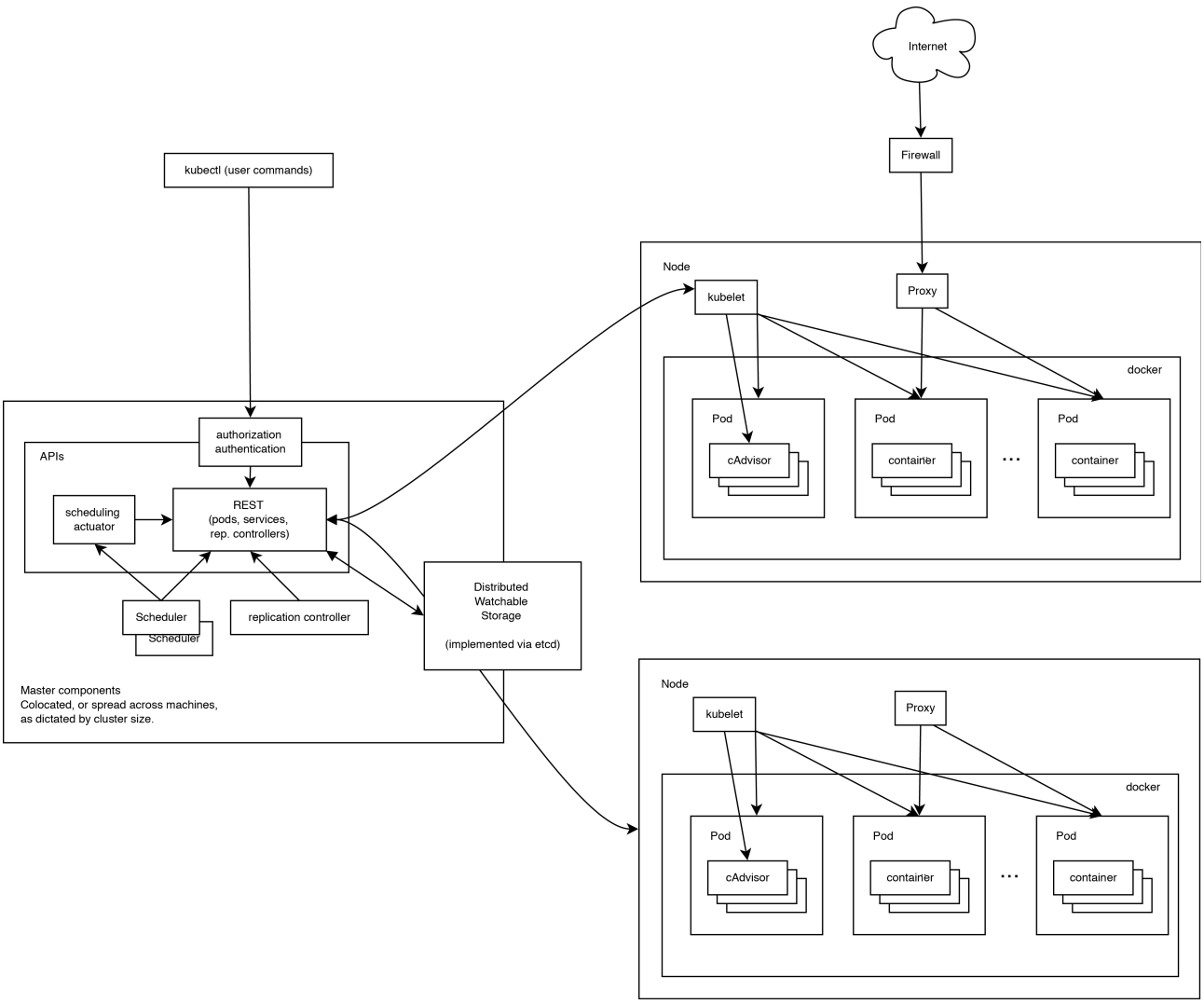


图 13-3：Kubernetes 运行原理图

可见，Kubernetes 首先是一套分布式系统，由多个节点组成，节点分为两类：一类是属于管理平面的主节点/控制节点 (Master Node)；一类是属于运行平面的工作节点 (Worker Node)。

显然，复杂的工作肯定都交给控制节点去做了，工作节点负责提供稳定的操作接口和能力抽象即可。

从这张图上，我们没有能发现 Kubernetes 中对于控制平面的分布式实现，但是由于数据后端自身就是一套分布式的数据库 Etcd，因此可以很容易扩展到分布式实现。

13.3.3 控制平面

控制平面 (Control Plane) 是 Kubernetes 集群的大脑，负责做出全局决策 (如调度) 以及检测和响应集群事件。

主节点服务

主节点上需要提供如下的管理服务：

- `apiserver` 是整个系统的对外接口，提供一套 RESTful 的 Kubernetes API，供客户端和其它组件调用；
- `scheduler` 负责对资源进行调度，分配某个 pod 到某个节点上。是 pluggable 的，意味着很容易选择其它实现方式；
- `controller-manager` 负责管理控制器，包括 endpoint-controller (刷新服务和 pod 的关联信息) 和 replication-controller (维护某个 pod 的复制为配置的数值)。

Etcd

这里 Etcd 即作为数据后端，又作为消息中间件。

通过 Etcd 来存储所有的主节点上的状态信息，很容易实现主节点的分布式扩展。

组件可以自动的去侦测 Etcd 中的数值变化来获得通知，并且获得更新后的数据来执行相应的操作。

13.3.4 工作节点

- kubelet 是工作节点执行操作的 agent，负责具体的容器生命周期管理，根据从数据库中获取的信息来管理容器，并上报 pod 运行状态等；
- kube-proxy 是一个简单的网络访问代理，同时也是一个 Load Balancer。它负责将访问到某个服务的请求具体分配给工作节点上的 Pod (同一类标签)。

Built with mdPress

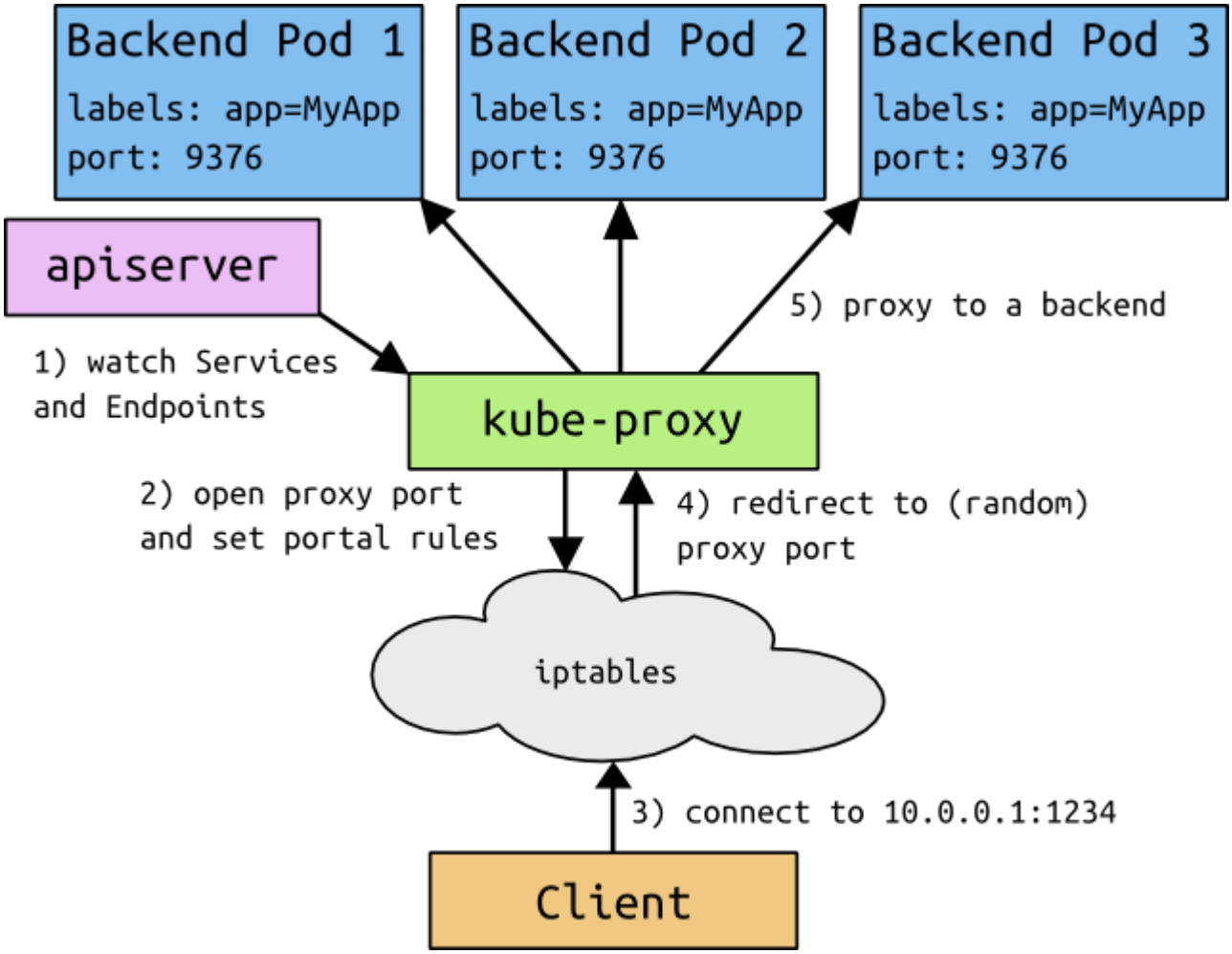


图 13-4：kube-proxy 请求转发示意图

13.4 高级特性

掌握了 Kubernetes 的核心概念 (Pod，Service，Deployment) 后，我们需要了解更多高级特性以构建生产级应用。

13.4.1 Helm - 包管理工具

Helm 被称为 Kubernetes 的包管理器 (类似于 Linux 的 apt/yum)。它将一组 Kubernetes 资源定义文件打包为一个 **Chart**。

- **安装应用**：`helm install my-release bitnami/mysql`
- **版本管理**：轻松回滚应用的发布版本。
- **模板化**：支持复杂的应用部署逻辑配置。

13.4.2 Gateway API 与 Ingress

Service 虽然提供了负载均衡，但通常是 4 层 (TCP/UDP)。集群需要 7 层 (HTTP/HTTPS) 路由能力来充当网关。

Gateway API（推荐）

> **重要**：Kubernetes 社区推荐使用 Gateway API 作为新一代流量管理标准。原 `kubernetes/ingress-nginx` 项目已于 2026 年 3 月退役停止维护，不再接收安全更新。

Gateway API 基于 CRD 实现，提供了比 Ingress 更强大和标准化的流量管理能力：

- **GatewayClass**：定义网关实现（类似 IngressClass）。
- **Gateway**：定义监听端口和协议（由基础设施团队管理）。
- **HTTPRoute**：定义 HTTP 路由规则（由应用团队管理）。
- **职责分离**：基础设施、集群运维和应用开发者各管各的资源。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: my-route
spec:
  parentRefs:
  - name: my-gateway
  hostnames:
  - "api.example.com"
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /api
    backendRefs:
    - name: api-svc
      port: 80
```

常见的 Gateway API 实现有 Envoy Gateway、Istio、Cilium、Traefik、Kong 等。

Ingress（传统方案）

Ingress 资源仍可正常使用，但建议新项目直接采用 Gateway API。已有 Ingress 配置可按需逐步迁移。

- **域名路由**：基于 Host 将请求转发不同服务。
- **路径路由**：基于 Path 将请求转发。
- **SSL/TLS**：集中管理证书。

13.4.3 Persistent Volume 与 StorageClass

容器内的文件是临时的。对于有状态应用 (如数据库)，需要持久化存储。

- **PVC (Persistent Volume Claim)**：用户申请存储的声明。
- **PV (Persistent Volume)**：实际的存储资源 (NFS，AWS EBS，Ceph 等)。
- **StorageClass**：定义存储类，支持动态创建 PV。
- **VolumeAttributesClass**：Kubernetes 1.34 起 GA，用于在 CSI 驱动支持 `ModifyVolume` 时动态调整卷属性，例如性能等级或服务质量参数。

13.4.4 Horizontal Pod Autoscaling

HPA 根据 CPU 利用率或其他指标 (如内存、自定义指标) 自动扩缩 Deployment 或 ReplicaSet 中的 Pod 数量。

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

13.4.5 ConfigMap 与 Secret

- **ConfigMap**：存储非机密的配置数据 (配置文件、环境变量)。
- **Secret**：存储机密数据 (密码、Token、证书)。Secret 值默认只是 Base64 编码并存入 etcd；生产环境应显式启用 etcd 静态加密或 KMS。

通过将配置与镜像分离，保证了容器的可移植性。

13.4.6 Pod Security Standards

> **注意**：PodSecurityPolicy (PSP) 已在 Kubernetes 1.25 中完全移除。

Kubernetes 使用 **Pod Security Standards** 定义三个安全级别，通过内置的 Pod Security Admission 控制器在命名空间级别执行：

- **Privileged**：不受限制，适用于系统级和基础设施工作负载。
- **Baseline**：防止已知的权限提升，适用于大多数工作负载。
- **Restricted**：严格限制，遵循 Pod 安全加固最佳实践。

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: my-app
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: restricted
```

13.4.7 Sidecar Containers

Kubernetes 1.33 起 Sidecar Containers 进入 GA。它们通过 `initContainers` 中带 `restartPolicy: Always` 的容器表达，既保留 init container 的启动顺序，又会在主容器生命周期内持续运行。对于旧集群或不需要启动顺序控制的场景，仍可使用普通多容器 Pod。

13.5 实战练习

本章将通过一个具体的案例：部署一个 Nginx 网站，并为其配置 Service 和 Ingress，来串联前面学到的知识。

13.5.1 目标

1. 部署一个 Nginx Deployment。
2. 创建一个 Service 暴露 Nginx。
3. (可选) 通过 Ingress 访问服务。

13.5.2 步骤 1：创建 Deployment

创建一个名为 `nginx-deployment.yaml` 的文件：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.28
        ports:
        - containerPort: 80
```

应用配置：

```
kubectl apply -f nginx-deployment.yaml
```

13.5.3 步骤 2：创建 Service

创建一个名为 `nginx-service.yaml` 的文件：

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort # 使用 NodePort 方便本地测试
```

应用配置：

```
kubectl apply -f nginx-service.yaml
```

查看分配的端口：

```
kubectl get svc nginx-service
```

如果输出端口是 `80:30080/TCP`，你可以通过 `http://<NodeIP>:30080` 访问 Nginx。

13.5.4 步骤 3：模拟滚动更新

修改 `nginx-deployment.yaml`，将镜像版本改为 `nginx:1.28-alpine`。

```
kubectl apply -f nginx-deployment.yaml
```

观察更新过程：

```
kubectl rollout status deployment/nginx-deployment
```

13.5.5 步骤 4：清理资源

练习结束后，记得清理资源：

```
kubectl delete -f nginx-service.yaml
kubectl delete -f nginx-deployment.yaml
```

本章小结

Kubernetes 是当前最主流的容器编排平台，其声明式管理模型和丰富的 API 为大规模容器化应用提供了坚实的基础。

概念	要点
Pod	最小调度单位，包含一组共享网络和存储的容器
Deployment	管理无状态应用的 Pod 副本集，支持滚动更新和回滚
StatefulSet	管理有状态应用，提供稳定的网络标识和持久化存储
DaemonSet	确保每个节点运行一个 Pod 副本，适用于日志、监控等场景
Job/CronJob	运行一次性或定时任务，确保任务成功完成
Service	为 Pod 提供稳定的网络访问入口和负载均衡
Namespace	资源隔离和多租户支持
ConfigMap/Secret	配置与敏感信息的管理
Master 节点	运行 API Server、Scheduler、Controller Manager
Worker 节点	运行 kubelet、kube-proxy 和容器运行时

延伸阅读

- 部署 Kubernetes：搭建 Kubernetes 集群
- Etcd：Kubernetes 使用的分布式存储
- 底层实现：容器技术原理

Built with mdPress

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十四章 部署 Kubernetes

目前，Kubernetes 支持在多种环境下使用，包括本地主机 (Ubuntu、Debian、CentOS、Fedora 等)、云服务 (腾讯云、阿里云、百度云等)。

你可以使用以下几种方式部署 Kubernetes，接下来的小节会对各种方式进行详细介绍。

- 使用 kubeadm 部署 (CRI 使用 containerd)
    - Kubernetes 也支持 CRI-O 等符合 CRI 的运行时；本文以 containerd 为主线。
- 使用 kubeadm 部署 (使用 Docker)
- 在 Docker Desktop 使用
- Kind - Kubernetes IN Docker
- K3s - 轻量级 Kubernetes
- 一步步部署 Kubernetes 集群
- 部署 Dashboard
- Kubernetes 命令行 kubectl

除了上述方式，企业生产环境中还有两个常见的部署工具值得关注：

- **KubeKey**：KubeSphere 社区开源的集群部署工具（CNCF 认证），支持一条命令从裸机部署到高可用集群，内置对 containerd 和多 Linux 发行版的适配，适合需要快速搭建私有化 Kubernetes 的团队。
- **RKE2**：SUSE Rancher 出品的安全加固型 Kubernetes 发行版，默认启用 CIS 基准合规、SELinux 支持和 etcd 自动快照，适合对安全审计有严格要求的企业场景。

Built with mdPress

14.1 使用 kubeadm 部署 Kubernetes

kubeadm 提供了 kubeadm init 以及 kubeadm join 这两个命令，作为快速创建 Kubernetes 集群的常用工具。

> **版本说明**：Kubernetes 版本更新较快 (约每 4 个月一个新版本)，本文档基于 Kubernetes 1.36 编写。更完整的安装和兼容性说明请以 Kubernetes 官方 kubeadm 文档 和 containerd 官方文档 为准。

14.1.1 安装 containerd

参考安装 Docker 一节添加 apt/yum 源，之后执行如下命令。

```
# debian 系

$ sudo apt install containerd.io

# rhel 系

$ sudo yum install containerd.io
```

14.1.2 配置 containerd

先生成默认配置文件，然后只调整与 kubeadm 相关的关键项：

```
$ sudo mkdir -p /etc/containerd
$ containerd config default | sudo tee /etc/containerd/config.toml > /dev/null
```

打开 /etc/containerd/config.toml，确认 runc 选项使用 systemd cgroup 驱动：

```
# containerd 2.x（当前默认版本）
[plugins."io.containerd.cri.v1.runtime".containerd.runtimes.runc.options]
  SystemdCgroup = true

# containerd 1.x（旧版本）使用以下路径，2.x 亦兼容
# [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
#   SystemdCgroup = true
```

> **提示**：`containerd config default` 生成的默认配置会自动使用当前版本的正确路径。如果你使用的是 containerd 2.x，配置中的插件路径将以 `io.containerd.cri.v1.runtime` 开头；如果仍在使用 1.x，则为 `io.containerd.grpc.v1.cri`。

默认配置里的其它内容通常可以保持不变。修改完成后重启 containerd：

```
$ sudo systemctl restart containerd
```

14.1.3 安装 kubelet、kubeadm、kubectl、cri-tools、kubernetes-cni

需要在每台机器上安装以下的软件包：

Ubuntu/Debian

```
$ K8S_MINOR="v1.36"

$ sudo apt-get update
$ sudo apt-get install -y ca-certificates curl gpg

$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL "https://pkgs.k8s.io/core:/stable:/${K8S_MINOR}/deb/Release.key" | sudo gpg --dearmor -
o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
$ sudo chmod a+r /etc/apt/keyrings/kubernetes-apt-keyring.gpg

$ echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stabl
e:/${K8S_MINOR}/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list > /dev/null

$ sudo apt-get update
$ sudo apt-get install -y kubelet kubeadm kubectl cri-tools kubernetes-cni

$ sudo apt-mark hold kubelet kubeadm kubectl
```

CentOS/Fedora

```
$ K8S_MINOR="v1.36"

$ cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/${K8S_MINOR}/rpm/
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/${K8S_MINOR}/rpm/repodata/repomd.xml.key
EOF

$ sudo yum install -y kubelet kubeadm kubectl cri-tools kubernetes-cni
```

14.1.4 修改内核的运行参数

加载内核模块

```
$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

$ sudo modprobe overlay
$ sudo modprobe br_netfilter
```

cgroup v2 要求：必须

Kubernetes v1.36 默认要求节点使用 cgroup v2。kubelet 在 cgroup v1 节点上默认会拒绝启动，但管理员可以在 kubelet 配置中设置 `failCgroupV1: false` 来兼容 cgroup v1（仅建议用于遗留系统过渡期）。验证节点是否支持 cgroup v2：

```
$ mount | grep cgroup2
```

如果输出包含 `cgroup2`，则系统已支持 cgroup v2。对于仍在使用 cgroup v1 的系统（如较旧的 RHEL 8），建议升级内核或更新系统配置以启用 cgroup v2。

禁用 swap：必须

kubelet 默认要求禁用 swap，否则可能导致初始化失败或节点无法加入集群。

```
$ sudo swapoff -a

# 如需永久禁用，可在 /etc/fstab 中注释 swap 对应行
```

```
$ cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# 应用配置

$ sysctl --system
```

14.1.5 配置 kubelet

为了让 kubelet 正确运行，我们需要对其进行一些必要的配置。

修改 `kubelet.service`（可选：IPVS 模式）

> **注意**：kube-proxy 的 IPVS 模式已在 Kubernetes 1.35 中被标记为弃用；Kubernetes 1.36 文档仍将其列为已弃用模式，并推荐迁移到 nftables。新部署应使用默认的 iptables 模式或 nftables 模式（Kubernetes 1.33+ 稳定）。以下 IPVS 配置仅供旧环境参考。

`/etc/systemd/system/kubelet.service.d/10-proxy-ipvs.conf` 写入以下内容

```
# 启用 ipvs 相关内核模块（已弃用，建议迁移至 nftables）

[Service]
ExecStartPre=-/sbin/modprobe ip_vs
ExecStartPre=-/sbin/modprobe ip_vs_rr
ExecStartPre=-/sbin/modprobe ip_vs_wrr
ExecStartPre=-/sbin/modprobe ip_vs_sh
```

执行以下命令应用配置。

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

14.1.6 部署

安装配置完成后，我们将分别在 Master 节点和 Worker 节点上进行部署操作。

master

```
$ sudo systemctl enable containerd

$ sudo systemctl start containerd

$ sudo kubeadm init \
      --image-repository registry.cn-hangzhou.aliyuncs.com/google_containers \
      --pod-network-cidr 10.244.0.0/16 \
      --cri-socket unix:///run/containerd/containerd.sock \
      --v 5
```

- `--pod-network-cidr 10.244.0.0/16` 参数与后续 CNI 插件有关，这里以 `flannel` 为例，若后续部署其他类型的网络插件请更改此参数。

> 若 `kubeadm` 预检失败，应按提示修复缺失依赖、内核参数、swap 或运行时配置。实验环境确需忽略预检时，只忽略明确理解且可接受的单项检查，不建议使用 `--ignore-preflight-errors=all`。

执行成功会输出

```
...
[addons] Applied essential addon: CoreDNS
I1116 12:35:13.270407   86677 request.go:538] Throttling request took 181.409184ms, request: POST:ht
tps://<CONTROL_PLANE_HOST>:6443/api/v1/namespaces/kube-system/serviceaccounts
I1116 12:35:13.470292   86677 request.go:538] Throttling request took 186.088112ms, request: POST:ht
tps://<CONTROL_PLANE_HOST>:6443/api/v1/namespaces/kube-system/configmaps
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join <CONTROL_PLANE_HOST>:6443 --token <TOKEN> \
    --discovery-token-ca-cert-hash sha256:<DISCOVERY_TOKEN_CA_CERT_HASH>
```

node 工作节点

在 **另一主机** 重复 **部署** 小节以前的步骤，安装配置好 kubelet。根据提示，加入到集群。

```
$ systemctl enable containerd

$ systemctl start containerd

$ kubeadm join <CONTROL_PLANE_HOST>:6443 \
    --token <TOKEN> \
    --discovery-token-ca-cert-hash sha256:<DISCOVERY_TOKEN_CA_CERT_HASH> \
    --cri-socket unix:///run/containerd/containerd.sock
```

其中 `<CONTROL_PLANE_HOST>`、`<TOKEN>` 和 `<DISCOVERY_TOKEN_CA_CERT_HASH>` 应使用你自己的 `kubeadm init` 输出，不要复用示例值。

14.1.7 查看服务

所有服务启动后，通过 `crictl` 查看本地实际运行的容器。这些服务大概分为三类：主节点服务、工作节点服务和其它服务。

```
CONTAINER_RUNTIME_ENDPOINT=unix:///run/containerd/containerd.sock crictl ps -a
```

主节点服务

- `apiserver` 是整个系统的对外接口，提供 RESTful 方式供客户端和其它组件调用；
- `scheduler` 负责对资源进行调度，分配某个 pod 到某个节点上；
- `controller-manager` 负责管理控制器，包括 endpoint-controller (刷新服务和 pod 的关联信息) 和 replication-controller (维护某个 pod 的复制为配置的数值)。

工作节点服务

- `proxy` 为 pod 上的服务提供访问的代理。

其它服务

- Etcd 是所有状态的存储数据库；

14.1.8 使用

将 `/etc/kubernetes/admin.conf` 复制到 `~/.kube/config`

执行 `$ kubectl get all -A` 查看启动的服务。

由于未部署 CNI 插件，CoreDNS 未正常启动。如何使用 Kubernetes，请参考后续章节。

14.1.9 部署 CNI

这里以 `flannel` 为例进行介绍。

flannel

检查 podCIDR 设置

```
$ kubectl get node -o yaml | grep CIDR

# 输出

    podCIDR: 10.244.0.0/16
    podCIDRs:
```

```
# 注意：以下以 v0.28.4 为示例；安装前请到 releases 页面核验当前版本
# 参见 https://github.com/flannel-io/flannel/releases
$ kubectl apply -f https://raw.githubusercontent.com/flannel-io/flannel/v0.28.4/Documentation/kube-flannel.yml
```

14.1.10 master 节点默认不能运行 pod

如果用 `kubeadm` 部署一个单节点集群，默认情况下无法使用，请执行以下命令解除限制

```
$ kubectl taint nodes --all node-role.kubernetes.io/control-plane-

# 较旧版本使用 master taint

# $ kubectl taint nodes --all node-role.kubernetes.io/master-

# 恢复默认值

# $ kubectl taint nodes NODE_NAME node-role.kubernetes.io/control-plane=true:NoSchedule

...
```

14.2 使用 kubeadm 部署 Kubernetes：使用 Docker

`kubeadm` 提供了 `kubeadm init` 以及 `kubeadm join` 这两个命令，作为快速创建 `Kubernetes` 集群的最佳实践。

> **版本说明**：本文档基于 Kubernetes v1.36 编写。Kubernetes 版本更新较快（约每 4 个月一个新版本），本文档中的第三方工具版本（如 cri-dockerd、flannel）仅为示例，请根据实际需求更新至当前版本。更完整的安装和兼容性说明请以 Kubernetes 官方文档 为准。

> ⚠️ **强烈提示：Docker 与 Kubernetes 环境的时代分界**
>
> 自 Kubernetes v1.24 起，内置的 `dockershim` 组件已被正式移除。这意味着 **Kubernetes 不再将 Docker Engine 作为默认内置的容器运行时**。虽然 Docker 仍然是你本地构建、管理镜像的绝佳工具，但它已不再是 kubelet 的默认运行时选项。
>
> 因此，**强烈推荐** 读者直接参考同目录下的《使用 kubeadm 部署 Kubernetes (CRI 使用 containerd)》作为主要的部署路线。
>
> 本文档保留，主要用于历史环境维护或特殊需求场景：如果你必须在较新的 Kubernetes 集群中继续使用 Docker Engine 作为底层运行时，你必须理解 CRI 层机制，额外部署并配置第三方兼容层 `cri-dockerd`，同时在部署时手动补充 `--cri-socket` 等参数约束。

14.2.1 安装 Docker

参考安装 Docker 一节安装 Docker。

14.2.2 安装并配置 cri-dockerd

由于 Kubernetes v1.24 移除了内置的 dockershim，需要额外安装 **cri-dockerd** 作为 Docker 与 kubelet 之间的 CRI（Container Runtime Interface）适配层。

Ubuntu/Debian

```
# 安装 cri-dockerd
# 注意：以下以 v0.4.3 为示例；安装前请到 releases 页面核验当前版本和校验值
# 参见 https://github.com/Mirantis/cri-dockerd/releases

$ cd /tmp
$ wget https://github.com/Mirantis/cri-dockerd/releases/download/v0.4.3/cri-dockerd-0.4.3.amd64.tgz
$ tar xzvf cri-dockerd-0.4.3.amd64.tgz
$ sudo mv cri-dockerd/cri-dockerd /usr/local/bin/

# 下载并安装 systemd service 文件

$ wget https://raw.githubusercontent.com/Mirantis/cri-dockerd/master/packaging/systemd/cri-docker.service
$ wget https://raw.githubusercontent.com/Mirantis/cri-dockerd/master/packaging/systemd/cri-docker.socket
$ sed -i -e 's,/usr/bin/cri-dockerd,/usr/local/bin/cri-dockerd,' cri-docker.service
$ sudo mv cri-docker.service cri-docker.socket /etc/systemd/system/

# 启动 cri-dockerd

$ sudo systemctl daemon-reload
$ sudo systemctl enable cri-docker
$ sudo systemctl start cri-docker

# 验证安装

$ sudo /usr/local/bin/cri-dockerd --version
```

CentOS/Fedora

```
# 安装 cri-dockerd
# 注意：以下以 v0.4.3 为示例；安装前请到 releases 页面核验当前版本和校验值
# 参见 https://github.com/Mirantis/cri-dockerd/releases

$ cd /tmp
$ wget https://github.com/Mirantis/cri-dockerd/releases/download/v0.4.3/cri-dockerd-0.4.3.amd64.tgz
$ tar xzvf cri-dockerd-0.4.3.amd64.tgz
$ sudo mv cri-dockerd/cri-dockerd /usr/local/bin/

# 下载并安装 systemd service 文件

$ wget https://raw.githubusercontent.com/Mirantis/cri-dockerd/master/packaging/systemd/cri-docker.service
$ wget https://raw.githubusercontent.com/Mirantis/cri-dockerd/master/packaging/systemd/cri-docker.socket
$ sed -i -e 's,/usr/bin/cri-dockerd,/usr/local/bin/cri-dockerd,' cri-docker.service
$ sudo mv cri-docker.service cri-docker.socket /etc/systemd/system/

# 启动 cri-dockerd

$ sudo systemctl daemon-reload
$ sudo systemctl enable cri-docker
$ sudo systemctl start cri-docker
```

14.2.3 安装 kubelet、kubeadm、kubectl

需要在每台机器上安装以下的软件包：

Ubuntu/Debian

```
$ K8S_MINOR="v1.36"

$ sudo apt-get update
$ sudo apt-get install -y ca-certificates curl gpg

$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL "https://pkgs.k8s.io/core:/stable:/${K8S_MINOR}/deb/Release.key" | sudo gpg --dearmor -
o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
$ sudo chmod a+r /etc/apt/keyrings/kubernetes-apt-keyring.gpg

$ echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stabl
e:/${K8S_MINOR}/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list > /dev/null

$ sudo apt-get update
$ sudo apt-get install -y kubelet kubeadm kubectl

$ sudo apt-mark hold kubelet kubeadm kubectl
```

CentOS/Fedora

```
$ K8S_MINOR="v1.36"

$ cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/${K8S_MINOR}/rpm/
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/${K8S_MINOR}/rpm/repodata/repomd.xml.key
EOF

$ sudo yum install -y kubelet kubeadm kubectl
```

14.2.4 修改内核的运行参数

cgroup v2 要求：必须

Kubernetes v1.36 默认要求节点使用 cgroup v2。kubelet 在 cgroup v1 节点上默认会拒绝启动，但管理员可以在 kubelet 配置中设置 `failCgroupV1: false` 来兼容 cgroup v1（仅建议用于遗留系统过渡期）。验证节点是否支持 cgroup v2：

```
$ mount | grep cgroup2
```

如果输出包含 `cgroup2`，则系统已支持 cgroup v2。对于仍在使用 cgroup v1 的系统（如较旧的 RHEL 8），建议升级内核或更新系统配置以启用 cgroup v2。

加载内核模块

```
$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

$ sudo modprobe overlay
$ sudo modprobe br_netfilter
```

swap 设置：现代集群仍建议优先关闭

在现代 Kubernetes 中，Linux 节点已经支持更细粒度的 swap 行为控制。对于多数 `kubeadm` 教学环境，**优先关闭 swap 仍然是最稳妥的默认做法**，因为这样最接近常见生产基线，也能避免不同发行版和 kubelet 配置差异带来的排障成本。

如果你明确要在启用 swap 的前提下运行节点，则应额外核对 kubelet 的 `NoSwap` / `LimitedSwap` 配置与当前 Kubernetes 版本文档，而不要把“开着 swap 也能跑”直接当作通用默认路径。

```
$ sudo swapoff -a

# 如需永久禁用，可在 /etc/fstab 中注释 swap 对应行
```

```
$ cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# 应用配置

$ sysctl --system
```

14.2.5 配置 kubelet

为了让 kubelet 正确运行，我们需要对其进行一些必要的配置。

修改 `kubelet.service`（可选：IPVS 模式）

> **注意**：kube-proxy 的 IPVS 模式已在 Kubernetes 1.35 中被标记为弃用；Kubernetes 1.36 文档仍将其列为已弃用模式，并推荐迁移到 nftables。新部署应使用默认的 iptables 模式或 nftables 模式（Kubernetes 1.33+ 稳定）。以下 IPVS 配置仅供旧环境参考。

`/etc/systemd/system/kubelet.service.d/10-proxy-ipvs.conf` 写入以下内容

```
# 启用 ipvs 相关内核模块（已弃用，建议迁移至 nftables）

[Service]
ExecStartPre=-/sbin/modprobe ip_vs
ExecStartPre=-/sbin/modprobe ip_vs_rr
ExecStartPre=-/sbin/modprobe ip_vs_wrr
ExecStartPre=-/sbin/modprobe ip_vs_sh
```

执行以下命令应用配置。

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

14.2.6 部署

安装配置完成后，我们将分别在 Master 节点和 Worker 节点上进行部署操作。

master

```
$ sudo kubeadm init --image-repository registry.cn-hangzhou.aliyuncs.com/google_containers \
      --pod-network-cidr 10.244.0.0/16 \
      --cri-socket unix:///var/run/cri-dockerd.sock \
      --v 5
```

- `--cri-socket unix:///var/run/cri-dockerd.sock` 参数指定使用 cri-dockerd 作为容器运行时接口。
- `--pod-network-cidr 10.244.0.0/16` 参数与后续 CNI 插件有关，这里以 `flannel` 为例，若后续部署其他类型的网络插件请更改此参数。

> 若 `kubeadm` 预检失败，应按提示修复缺失依赖、内核参数、swap 或运行时配置。实验环境确需忽略预检时，只忽略明确理解且可接受的单项检查，不建议使用 `--ignore-preflight-errors=all`。

执行成功会输出

```
...
[addons] Applied essential addon: CoreDNS
I1116 12:35:13.270407   86677 request.go:538] Throttling request took 181.409184ms, request: POST:ht
tps://<CONTROL_PLANE_HOST>:6443/api/v1/namespaces/kube-system/serviceaccounts
I1116 12:35:13.470292   86677 request.go:538] Throttling request took 186.088112ms, request: POST:ht
tps://<CONTROL_PLANE_HOST>:6443/api/v1/namespaces/kube-system/configmaps
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join <CONTROL_PLANE_HOST>:6443 --token <TOKEN> \
    --discovery-token-ca-cert-hash sha256:<DISCOVERY_TOKEN_CA_CERT_HASH>
```

node 工作节点

在 **另一主机** 重复 **部署** 小节以前的步骤，安装配置好 kubelet。根据提示，加入到集群。

```
$ kubeadm join <CONTROL_PLANE_HOST>:6443 --token <TOKEN> \
    --discovery-token-ca-cert-hash sha256:<DISCOVERY_TOKEN_CA_CERT_HASH> \
    --cri-socket unix:///var/run/cri-dockerd.sock
```

其中 `<CONTROL_PLANE_HOST>`、`<TOKEN>` 和 `<DISCOVERY_TOKEN_CA_CERT_HASH>` 应使用你自己的 `kubeadm init` 输出，不要复用示例值。

14.2.7 查看服务

所有服务启动后，查看本地实际运行的 Docker 容器。这些服务大概分为三类：主节点服务、工作节点服务和其它服务。

主节点服务

- `apiserver` 是整个系统的对外接口，提供 RESTful 方式供客户端和其它组件调用；
- `scheduler` 负责对资源进行调度，分配某个 pod 到某个节点上；
- `controller-manager` 负责管理控制器，包括 endpoint-controller (刷新服务和 pod 的关联信息) 和 replication-controller (维护某个 pod 的复制为配置的数值)。

工作节点服务

- `proxy` 为 pod 上的服务提供访问的代理。

其它服务

- Etcd 是所有状态的存储数据库；

14.2.8 使用

将 `/etc/kubernetes/admin.conf` 复制到 `~/.kube/config`

执行 `$ kubectl get all -A` 查看启动的服务。

由于未部署 CNI 插件，CoreDNS 未正常启动。如何使用 Kubernetes，请参考后续章节。

14.2.9 部署 CNI

这里以 `flannel` 为例进行介绍。

flannel

检查 podCIDR 设置

```
$ kubectl get node -o yaml | grep CIDR

# 输出

    podCIDR: 10.244.0.0/16
    podCIDRs:
```

```
# 注意：以下以 v0.28.4 为示例；安装前请到 releases 页面核验当前版本
# 参见 https://github.com/flannel-io/flannel/releases
$ kubectl apply -f https://raw.githubusercontent.com/flannel-io/flannel/v0.28.4/Documentation/kube-flannel.yml
```

14.2.10 master 节点默认不能运行 pod

如果用 `kubeadm` 部署一个单节点集群，默认情况下无法使用，请执行以下命令解除限制

```
$ kubectl taint nodes --all node-role.kubernetes.io/control-plane-

# 较旧版本使用 master taint

# $ kubectl taint nodes --all node-role.kubernetes.io/master-

# 恢复默认值

# $ kubectl taint nodes NODE_NAME node-role.kubernetes.io/control-plane=true:NoSchedule

...
```

14.3 在 Docker Desktop 使用

使用 Docker Desktop 可以很方便的启用 Kubernetes。

14.3.1 启用 Kubernetes

在 Docker Desktop 设置页面，点击 `Kubernetes`，选择 `Enable Kubernetes`，稍等片刻，看到左下方 `Kubernetes` 变为 `running`，Kubernetes 启动成功。

图

> 注意：Kubernetes 的镜像存储在 `registry.k8s.io`，如果国内网络无法直接访问，可以在 Docker Desktop 配置中的 `Docker Engine` 处配置镜像加速器，或者利用国内云服务商的镜像仓库手动拉取镜像并 retag。

14.3.2 测试

```
$ kubectl version
```

如果正常输出信息，则证明 Kubernetes 成功启动。

14.4 Kind - Kubernetes IN Docker

Kind (Kubernetes in Docker) 是一个使用 Docker 容器作为节点运行本地 Kubernetes 集群的工具。主要用于测试 Kubernetes 本身，也非常适合本地开发和 CI 环境。

> **版本说明**：本文档基于 Kind v0.31.0 编写。Kind 会根据下载时的默认行为自动拉取对应的 Kubernetes 版本。建议定期更新 Kind 到最新版本以获得最新的 Kubernetes 支持。详见 Kind Releases。

14.4.1 为什么选择 Kind

Kind 相比其他本地集群方案 (如 Minikube) 有以下显著优势：

- **轻量便捷**：只要有 Docker 环境即可，无需额外虚拟机。
- **多集群支持**：可以轻松在本地启动多个集群。
- **多版本支持**：支持指定 Kubernetes 版本进行测试。
- **HA 支持**：支持模拟高可用集群 (多 Control Plane)。

14.4.2 安装 Kind

Kind 是一个二进制文件，并在 PATH 中即可使用。以下是不同系统的安装方法。

macOS

```
brew install kind
```

Linux / Windows

可以下载二进制文件：

```
# Linux AMD64
# 注意：v0.31.0 为编写本文档时的最新版本，请根据需要替换为当前版本
# 参见 https://github.com/kubernetes-sigs/kind/releases

curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.31.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

14.4.3 创建集群

最简单的创建方式：

```
kind create cluster
```

指定集群名称：

```
kind create cluster --name my-cluster
```

14.4.4 与集群交互

Kind 会自动将 kubeconfig 合并到 `~/.kube/config`。

```
kubectl cluster-info --context kind-kind
kubectl get nodes
```

14.4.5 高级用法：配置集群

创建一个 `kind-config.yaml` 来定制集群，例如映射端口到宿主机：

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 8080
    protocol: TCP
- role: worker
- role: worker
```

应用配置：

```
kind create cluster --config kind-config.yaml
```

14.4.6 删除集群

```
kind delete cluster
```

14.5 K3s - 轻量级 Kubernetes

K3s 是一个轻量级的 Kubernetes 发行版，由 Rancher Labs 开发。它专为边缘计算、物联网、CI、ARM 等资源受限的环境设计。K3s 被打包为单个二进制文件，只有不到 100MB，但通过了 CNCF 的一致性测试。

> **版本说明**：K3s 版本与 Kubernetes 版本对应。安装脚本会自动拉取最新稳定版本。如需指定特定版本，可在安装时设置 `INSTALL_K3S_VERSION` 环境变量。详见 K3s Releases。

14.5.1 核心特性

- **轻量级**：移除过时的、非必须的 Kubernetes 功能 (如传统的云提供商插件)，使用 SQLite 作为默认数据存储 (也支持 Etcd/MySQL/Postgres)。
- **单一二进制**：所有组件 (API Server，Controller Manager，Scheduler，Kubelet，Kube-proxy) 打包在一个进程中运行。
- **开箱即用**：内置 Helm Controller、Traefik Ingress controller、ServiceLB、Local-Path-Provisioner。
- **安全**：默认启用安全配置，基于 TLS 通信。

14.5.2 安装

K3s 的安装非常简单，官方提供了便捷的安装脚本。

脚本安装

K3s 提供了极为便捷的安装脚本。该命令会从网络下载脚本并直接交给 `sh` 执行，生产环境建议先下载审查脚本内容，并按官方文档固定版本或安装参数：

```
curl -sfL https://get.k3s.io | sh -
```

安装完成后，K3s 会自动启动并配置好 `systemd` 服务。

查看状态

```
sudo k3s kubectl get nodes
```

输出类似：

```
NAME          STATUS   ROLES                  AGE   VERSION
k3s-master    Ready    control-plane,master   1m    v1.36.0+k3s1
```

> **版本说明**：输出中的 `v1.36.0+k3s1` 为编写本文档时的版本示例。实际输出版本号取决于你所安装的 K3s 版本。更多信息请参见 K3s Releases。

14.5.3 快速使用

K3s 内置了 `kubectl` 命令（通过 `k3s kubectl` 调用），为了方便，通常会建立别名或配置 KUBECONFIG。

```
## 读取 K3s 的配置文件

export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

## 现在可以直接使用 kubectl

kubectl get pods -A
```

14.5.4 清理卸载

```
/usr/local/bin/k3s-uninstall.sh
```

14.6 一步步部署 Kubernetes 集群

14.6.1 概述

部署 Kubernetes 集群涉及多个组件的安装和配置，包括 Master 节点和 Worker 节点。本章介绍如何使用 systemd 管理这些服务的生命周期。

14.6.2 Kubernetes 主要组件

Master 节点组件

- **kube-apiserver**：API 服务器，Kubernetes 集群的中心
- **kube-controller-manager**：控制器管理器
- **kube-scheduler**：调度器，负责 Pod 调度
- **etcd**：分布式键值存储，存储集群数据

Worker 节点组件

- **kubelet**：节点代理，管理容器生命周期
- **kube-proxy**：网络代理，处理服务网络
- **Container Runtime**：容器运行时（Docker、containerd 等）

14.6.3 使用 systemd 管理 Kubernetes 服务

服务单元文件

为了让 systemd 管理 Kubernetes 服务，需要创建相应的 `.service` 文件，例如：

```
/etc/systemd/system/kubelet.service
/etc/systemd/system/kube-proxy.service
/etc/systemd/system/kube-apiserver.service
```

常用命令

```
# 启动服务
sudo systemctl start kubelet

# 停止服务
sudo systemctl stop kubelet

# 重启服务
sudo systemctl restart kubelet

# 查看服务状态
sudo systemctl status kubelet

# 设置开机自启
sudo systemctl enable kubelet
```

如果希望查看更完整的 systemd 部署案例，可以参考 `opsnull/follow-me-install-kubernetes-cluster` 这类社区项目，再结合本章前文的 kubeadm 与组件配置说明理解整体流程。

14.6.4 推荐学习路径

1. 理解 Kubernetes 架构和各组件的作用
2. 准备所需的系统环境（Linux 主机、网络配置等）
3. 按步骤安装各个 Kubernetes 组件
4. 配置 systemd 服务单元文件
5. 验证集群健康状态

14.7 部署 Dashboard

Kubernetes Dashboard 是基于网页的 Kubernetes 用户界面。

> **注意**：原 `kubernetes/dashboard` 项目已于 2026 年 1 月 21 日归档停止维护。推荐使用 Headlamp 等替代方案。以下内容仅供历史参考，基于归档前的最新 Helm 安装方式。

图

14.7.1 部署

Dashboard 7.0+ 版本仅支持通过 Helm 安装：

```
$ helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/

$ helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard \
  --create-namespace --namespace kubernetes-dashboard
```

14.7.2 访问

通过端口转发访问 Dashboard：

```
$ kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443
```

然后在浏览器打开 `https://localhost:8443` 即可访问。

14.7.3 登录

为历史 Dashboard 创建只读服务账户并获取短期登录令牌。不要为 Dashboard 创建 `cluster-admin` 绑定；如果确实需要临时管理员权限，应走单独的 break-glass 审批和审计流程。

```
$ kubectl create sa dashboard-readonly -n kubernetes-dashboard

$ kubectl create clusterrole dashboard-readonly \
  --verb=get,list,watch \
  --resource=pods,deployments,services,configmaps,namespaces,nodes

$ kubectl create clusterrolebinding dashboard-readonly \
  --clusterrole=dashboard-readonly \
  --serviceaccount=kubernetes-dashboard:dashboard-readonly

$ kubectl create token dashboard-readonly -n kubernetes-dashboard --duration=1h
```

将输出的令牌粘贴到登录页面，即可登录。

14.8 Kubernetes 命令行 kubectl

kubectl 是 Kubernetes 自带的客户端，可以用它来直接操作 Kubernetes。

kubectl 的基本用法格式为：

```
kubectl [command] [resource] [name] [flags]
```

其中 command 是操作命令（如 get、apply、delete），resource 是资源类型（如 pod、deployment、service），name 是资源名称（可选），flags 是各种选项参数。

14.8.1 get

显示一个或多个资源。最常用的命令之一，用于查询集群中的资源状态。

```
# 查看所有 Pod
$ kubectl get pods

# 查看指定命名空间中的 Deployment
$ kubectl get deployment -n kube-system

# 查看所有资源
$ kubectl get all

# 使用更详细的输出格式
$ kubectl get pods -o wide

# 输出为 YAML 格式
$ kubectl get pod my-pod -o yaml

# 根据标签选择资源
$ kubectl get pods -l app=nginx
```

14.8.2 describe

显示资源的详细信息，包括事件、状态、配置等。用于调试和查看资源的完整信息。

```
# 查看特定 Pod 的详情
$ kubectl describe pod my-pod

# 查看 Node 的详情
$ kubectl describe node node-1

# 查看 Deployment 的详情
$ kubectl describe deployment nginx-deployment
```

14.8.3 apply

从文件或标准输入应用配置。这是声明式资源管理的标准方式，可用于创建或更新资源。

```
# 应用 YAML 文件
$ kubectl apply -f deployment.yaml

# 应用目录中的所有 YAML 文件
$ kubectl apply -f ./manifests/

# 从标准输入应用配置
$ cat deployment.yaml | kubectl apply -f -
```

14.8.4 create

从文件或标准输入创建资源。与 apply 不同，create 仅用于创建新资源，如果资源已存在会报错。

```
# 创建资源
$ kubectl create -f pod.yaml

# 从标准输入创建
$ kubectl create -f - < deployment.yaml
```

14.8.5 delete

删除一个或多个资源。支持按名称、标签、资源类型等多种方式删除。

```
# 按名称删除
$ kubectl delete pod my-pod

# 删除整个 Deployment（同时删除其管理的 Pod）
$ kubectl delete deployment nginx-deployment

# 按标签删除
$ kubectl delete pods -l app=nginx

# 从文件删除
$ kubectl delete -f deployment.yaml
```

14.8.6 logs

查看 Pod 中容器的日志。用于调试应用和查看容器输出。

```
# 查看 Pod 的日志
$ kubectl logs my-pod

# 查看 Pod 中特定容器的日志
$ kubectl logs my-pod -c container-name

# 实时跟踪日志（类似 tail -f）
$ kubectl logs -f my-pod

# 查看最近 100 行日志
$ kubectl logs my-pod --tail=100

# 查看 Pod 启动前的日志（适用于已崩溃的容器）
$ kubectl logs my-pod --previous
```

14.8.7 exec

在运行中的容器内部执行命令。用于调试、排查问题或执行应用内的操作。

```
# 进入容器的交互式 shell
$ kubectl exec -it my-pod -- /bin/sh

# 在容器中执行命令
$ kubectl exec my-pod -- ls -la /app

# 在 Pod 中的特定容器执行命令
$ kubectl exec -it my-pod -c container-name -- /bin/bash
```

14.8.8 port-forward

将本地端口转发到 Pod 的端口。用于本地访问集群内的服务，无需暴露 Service。

```
# 将本地 8080 端口转发到 Pod 的 80 端口
$ kubectl port-forward pod/my-pod 8080:80

# 使用随机本地端口
$ kubectl port-forward pod/my-pod :80

# 转发到 Service
$ kubectl port-forward svc/my-service 8080:80
```

14.8.9 rollout

对 Deployment、DaemonSet、StatefulSet 等资源执行滚动更新、暂停、继续或回滚操作。

```
# 查看滚动更新状态
$ kubectl rollout status deployment/nginx-deployment

# 暂停滚动更新
$ kubectl rollout pause deployment/nginx-deployment

# 继续滚动更新
$ kubectl rollout resume deployment/nginx-deployment

# 查看更新历史
$ kubectl rollout history deployment/nginx-deployment

# 回滚到前一个版本
$ kubectl rollout undo deployment/nginx-deployment

# 回滚到特定版本
$ kubectl rollout undo deployment/nginx-deployment --to-revision=2
```

14.8.10 label

向资源添加、修改或删除标签。标签用于组织和选择资源。

```
# 为 Pod 添加标签
$ kubectl label pod my-pod env=production

# 修改已有标签
$ kubectl label pod my-pod env=staging --overwrite

# 删除标签
$ kubectl label pod my-pod env-

# 为多个资源添加标签
$ kubectl label pods -l app=nginx version=v1
```

14.8.11 annotate

向资源添加或修改注解。注解用于存储任意元数据，不用于资源选择。

```
# 添加注解
$ kubectl annotate pod my-pod description="Production pod"

# 修改注解
$ kubectl annotate pod my-pod description="Staging pod" --overwrite

# 删除注解
$ kubectl annotate pod my-pod description-
```

14.8.12 edit

直接编辑 Kubernetes 资源。编辑器由 `EDITOR` 环境变量指定。

```
# 编辑 Pod
$ kubectl edit pod my-pod

# 编辑 Deployment
$ kubectl edit deployment nginx-deployment
```

14.8.13 config

管理 kubectl 的配置文件（通常位于 `~/.kube/config`）。

```
# 查看当前配置
$ kubectl config view

# 切换上下文
$ kubectl config use-context my-cluster

# 查看所有上下文
$ kubectl config get-contexts

# 设置默认命名空间
$ kubectl config set-context --current --namespace=my-namespace
```

14.8.14 cluster-info

显示集群的连接信息和组件状态。

```
# 显示集群信息
$ kubectl cluster-info

# 显示完整的集群状态（注：componentstatuses 自 1.19 起已弃用，建议使用下方替代命令）
$ kubectl get componentstatuses
# 推荐替代：
$ kubectl get --raw='/readyz?verbose'
```

14.8.15 version

显示 kubectl 客户端和 Kubernetes API server 的版本信息。

```
# 显示版本
$ kubectl version

# 仅显示客户端版本
$ kubectl version --client

# 以 JSON 格式输出
$ kubectl version --output=json
```

14.8.16 api-versions

列出 API server 支持的所有 API 版本，格式为 “组/版本”。

```
$ kubectl api-versions
```

14.8.17 explain

解释资源的字段含义。用于了解 YAML 配置文件中各字段的作用。

```
# 查看 Pod 资源的字段说明
$ kubectl explain pod

# 查看 Pod 下 spec 字段的说明
$ kubectl explain pod.spec

# 查看具体字段的详细说明
$ kubectl explain pod.spec.containers
```

14.8.18 auth

检查用户权限，用于验证 RBAC 配置。

```
# 检查当前用户是否有权限执行操作
$ kubectl auth can-i create pods

# 检查特定用户的权限
$ kubectl auth can-i create pods --as=other-user
```

14.8.19 help

显示任何命令的帮助信息。

```
# 显示 kubectl 的一般帮助
$ kubectl help

# 显示特定命令的帮助
$ kubectl get --help

# 显示资源的 API 文档
$ kubectl explain deployment
```

本章小结

部署 Kubernetes 集群有多种方式，应根据使用场景选择合适的方案。

部署方式	适用场景	特点
kubeadm	生产环境	官方推荐的集群部署工具
Docker Desktop	本地开发	一键启用，开箱即用
Kind	CI/CD 测试	Kubernetes IN Docker，快速创建集群
K3s	边缘计算/IoT	轻量级，资源占用少
手动部署	学习原理	逐步配置每个组件，加深理解

延伸阅读

- 容器编排基础：Kubernetes 核心概念
- Dashboard：部署可视化管理界面
- kubectl：命令行工具使用指南

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十五章 Etcd 项目

etcd 是 CoreOS 团队发起的一个管理配置信息和服务发现 (Service Discovery) 的项目，在这一章里面，我们将基于 etcd 3.5 系列版本介绍该项目的目标，安装和使用，以及实现的技术。

> **版本说明：** 本章示例基于 etcd 3.5 系列版本编写。etcd 官方维护最新两个次版本（当前为 3.5 和 3.6）。请访问 etcd 官方发布页 获取最新版本信息。

本章内容

- 简介
- 安装
- 集群
- 使用 etcdctl

Built with mdPress

15.1 简介

> **版本说明：** 本章内容基于 etcd 3.5 系列版本编写。官方维护最新两个次版本（当前为 3.5 和 3.6）。请访问 etcd 官方发布页 获取最新版本信息。

如图 15-1 所示，etcd 项目使用该标识。



图 15-1：etcd 项目标识

etcd 是 CoreOS 团队于 2013 年 6 月发起的开源项目，它的目标是构建一个高可用的分布式键值(key-value)数据库，基于 Go 语言实现。我们知道，在分布式系统中，各种服务的配置信息的管理分享，服务的发现是一个很基本同时也是很重要的问题。CoreOS 项目就希望基于 etcd 来解决这一问题。

etcd 目前在 github.com/etcd-io/etcd 进行维护。

受到 Apache ZooKeeper 项目和 doozer 项目的启发，etcd 在设计的时候重点考虑了下面四个要素：

- 简单：具有定义良好、面向用户的 API (gRPC)
- 安全：支持 HTTPS 方式的访问
- 快速：支持并发 10 k/s 的写操作
- 可靠：支持分布式结构，基于 Raft 的一致性算法

Apache ZooKeeper 是一套知名的分布式系统中进行同步和一致性管理的工具。

doozer 是一个一致性分布式数据库。

Raft 是一套通过选举主节点来实现分布式系统一致性的算法，相比于大名鼎鼎的 Paxos 算法，它的过程更容易被人理解，由 Stanford 大学的 Diego Ongaro 和 John Ousterhout 提出。更多细节可以参考 raftconsensus.github.io。

Built with mdPress

一般情况下，用户使用 etcd 可以在多个节点上启动多个实例，并添加它们为一个集群。同一个集群中的 etcd 实例将会保持彼此信息的一致性。

15.2 安装

本节将介绍 etcd 的几种常见安装方式，包括二进制安装、Docker 镜像运行以及在 macOS 上的安装。

etcd 基于 Go 语言实现，因此，用户可以从项目主页下载源代码自行编译，也可以下载编译好的二进制文件，甚至直接使用制作好的 Docker 镜像文件来体验。

> 注意：etcd 官方仅维护最新两个次版本（当前为 3.5 和 3.6）。etcd 3.4 已于 2026 年 5 月结束支持（EOL），仍在使用 3.4 的用户应尽快升级。本章示例基于 etcd 3.5.x 版本编写。etcd 3.6.x 可用于新部署。请访问 etcd 官方发布页 获取最新版本。

15.2.1 二进制文件方式下载

编译好的二进制文件都在 github.com/etcd-io/etcd/releases 页面，用户可以选择需要的版本，或通过下载工具下载。

例如，使用 curl 工具下载压缩包，并解压。

```
# 下载 etcd v3.5.29 版本（请访问 https://github.com/etcd-io/etcd/releases 获取最新版本）
$ curl -L https://github.com/etcd-io/etcd/releases/download/v3.5.29/etcd-v3.5.29-linux-amd64.tar.gz
-o etcd-v3.5.29-linux-amd64.tar.gz

## 国内用户可选择就近的网络加速方式（以可用镜像站为准）

$ tar xzvf etcd-v3.5.29-linux-amd64.tar.gz
$ cd etcd-v3.5.29-linux-amd64
```

解压后，可以看到文件包括

```
$ ls
Documentation README-etcdctl.md README.md READMEv2-etcdctl.md etcd etcdctl
```

其中 etcd 是服务主文件，etcdctl 是提供给用户的命令客户端，其他文件是支持文档。

下面将 etcd etcdctl 文件放到系统可执行目录 (例如 /usr/local/bin/)。

```
$ sudo cp etcd* /usr/local/bin/
```

默认 2379 端口处理客户端的请求，2380 端口用于集群各成员间的通信。启动 etcd 显示类似如下的信息：

```
$ etcd
...
2017-12-03 11:18:34.411579 I | embed: listening for peers on http://localhost:2380
2017-12-03 11:18:34.411938 I | embed: listening for client requests on localhost:2379
```

此时，可以使用 `etcdctl` 命令进行测试，设置和获取键值 `testkey: "hello world"`，检查 `etcd` 服务是否启动成功：

```
$ ETCDCTL_API=3 etcdctl member list
8e9e05c52164694d, started, default, http://localhost:2380, http://localhost:2379

$ ETCDCTL_API=3 etcdctl put testkey "hello world"
OK

$ ETCDCTL_API=3 etcdctl get testkey
testkey
hello world
```

说明 etcd 服务已经成功启动了。

15.2.2 Docker 镜像方式运行

镜像名称为 `quay.io/coreos/etcd`，可以通过下面的命令启动 `etcd` 服务监听到 `2379` 和 `2380` 端口。

> **版本说明：** 示例中使用 `v3.5.29` 标签。请访问 etcd 官方发布页 获取最新可用版本标签。

```
$ docker run \
-p 2379:2379 \
-p 2380:2380 \
--mount type=bind,source=/tmp/etcd-data.tmp,destination=/etcd-data \
--name etcd-gcr-v3.5.29 \
quay.io/coreos/etcd:v3.5.29 \
/usr/local/bin/etcd \
--name s1 \
--data-dir /etcd-data \
--listen-client-urls http://0.0.0.0:2379 \
--advertise-client-urls http://<HOST_IP>:2379 \
--listen-peer-urls http://0.0.0.0:2380 \
--initial-advertise-peer-urls http://<HOST_IP>:2380 \
--initial-cluster s1=http://<HOST_IP>:2380 \
--initial-cluster-token tkn \
--initial-cluster-state new \
--log-level info \
--logger zap \
--log-outputs stderr
```

其中 `listen-*` 可以绑定 `0.0.0.0` 监听所有网卡，但 `advertise-*` 应填写其他节点或客户端**实际可访问的主机地址**，不能直接写成 `0.0.0.0`。

打开新的终端按照上一步的方法测试 etcd 是否成功启动。

15.2.3 macOS 中运行

```
$ brew install etcd

$ etcd

$ ETCDCTL_API=3 etcdctl member list
```

15.3 集群

> **版本说明：** 本节示例使用 etcd v3.5.29。请访问 etcd 官方发布页 获取最新版本信息。

下面我们使用 Docker Compose 模拟启动一个 3 节点的 `etcd` 集群。

编辑 `compose.yaml` (或 `docker-compose.yml`) 文件

```
services:

  node1:
    image: quay.io/coreos/etcd:v3.5.29
    volumes:
      - node1-data:/etcd-data
    expose:
      - 2379
      - 2380
    networks:
      cluster_net:
        ipv4_address: 172.16.238.100
    environment:
      - ETCDCTL_API=3
    command:
      - /usr/local/bin/etcd
      - --data-dir=/etcd-data
      - --name
      - node1
      - --initial-advertise-peer-urls
      - http://172.16.238.100:2380
      - --listen-peer-urls
      - http://0.0.0.0:2380
      - --advertise-client-urls
      - http://172.16.238.100:2379
      - --listen-client-urls
      - http://0.0.0.0:2379
      - --initial-cluster
      - node1=http://172.16.238.100:2380,node2=http://172.16.238.101:2380,node3=http://172.16.238.10
2:2380
      - --initial-cluster-state
      - new
      - --initial-cluster-token
      - docker-etcd

  node2:
    image: quay.io/coreos/etcd:v3.5.29
    volumes:
      - node2-data:/etcd-data
    networks:
      cluster_net:
        ipv4_address: 172.16.238.101
    environment:
      - ETCDCTL_API=3
    expose:
      - 2379
      - 2380
    command:
      - /usr/local/bin/etcd
      - --data-dir=/etcd-data
      - --name
      - node2
      - --initial-advertise-peer-urls
      - http://172.16.238.101:2380
      - --listen-peer-urls
      - http://0.0.0.0:2380
      - --advertise-client-urls
      - http://172.16.238.101:2379
      - --listen-client-urls
      - http://0.0.0.0:2379
```

```
      - --initial-cluster
      - node1=http://172.16.238.100:2380,node2=http://172.16.238.101:2380,node3=http://172.16.238.10
2:2380
      - --initial-cluster-state
      - new
      - --initial-cluster-token
      - docker-etcd

  node3:
    image: quay.io/coreos/etcd:v3.5.29
    volumes:
      - node3-data:/etcd-data
    networks:
      cluster_net:
        ipv4_address: 172.16.238.102
    environment:
      - ETCDCTL_API=3
    expose:
      - 2379
      - 2380
    command:
      - /usr/local/bin/etcd
      - --data-dir=/etcd-data
      - --name
      - node3
      - --initial-advertise-peer-urls
      - http://172.16.238.102:2380
      - --listen-peer-urls
      - http://0.0.0.0:2380
      - --advertise-client-urls
      - http://172.16.238.102:2379
      - --listen-client-urls
      - http://0.0.0.0:2379
      - --initial-cluster
      - node1=http://172.16.238.100:2380,node2=http://172.16.238.101:2380,node3=http://172.16.238.10
2:2380
      - --initial-cluster-state
      - new
      - --initial-cluster-token
      - docker-etcd

volumes:
  node1-data:
  node2-data:
  node3-data:

networks:
  cluster_net:
    driver: bridge
    ipam:
      driver: default
      config:
      -
        subnet: 172.16.238.0/24
```

使用 `docker compose up` 启动集群之后使用 `docker exec` 命令登录到任一节点测试 `etcd` 集群。

```
/ # etcdctl member list
daf3fd52e3583ff, started, node3, http://172.16.238.102:2380, http://172.16.238.102:2379
422a74f03b622fef, started, node1, http://172.16.238.100:2380, http://172.16.238.100:2379
ed635d2a2dbef43d, started, node2, http://172.16.238.101:2380, http://172.16.238.101:2379
```

15.4 使用 etcdctl

> **版本说明：** 本节示例基于 etcd 3.5 系列版本。请访问 etcd 官方发布页 获取最新版本信息。

`etcdctl` 是一个命令行客户端，它能提供一些简洁的命令，供用户直接跟 `etcd` 服务打交道，而无需基于 `HTTP API` 方式。这在某些情况下将很方便，例如用户对服务进行测试或者手动修改数据库内容。我们也推荐在刚接触 `etcd` 时通过 `etcdctl` 命令来熟悉相关的操作，这些操作跟 `HTTP API` 实际上是对应的。

`etcd` 项目二进制发行包中已经包含了 `etcdctl` 工具，没有的话，可以从 github.com/etcd-io/etcd/releases 下载。

`etcdctl` 支持如下的命令，大体上分为数据库操作和非数据库操作两类，后面将分别进行解释。

```
NAME:
        etcdctl - A simple command line client for etcd3.

USAGE:
        etcdctl

VERSION:
        3.5.29

API VERSION:
        3.5


COMMANDS:
        get                     Gets the key or a range of keys
        put                     Puts the given key into the store
        del                     Removes the specified key or range of keys [key, range_end)
        txn                     Txn processes all the requests in one transaction
        compaction              Compacts the event history in etcd
        alarm disarm            Disarms all alarms
        alarm list              Lists all alarms
        defrag                  Defragments the storage of the etcd members with given endpoints
        endpoint health         Checks the healthiness of endpoints specified in `--endpoints` flag
        endpoint status         Prints out the status of endpoints specified in `--endpoints` flag
        watch                   Watches events stream on keys or prefixes
        version                 Prints the version of etcdctl
        lease grant             Creates leases
        lease revoke            Revokes leases
        lease timetolive        Get lease information
        lease keep-alive        Keeps leases alive (renew)
        member add              Adds a member into the cluster
        member remove           Removes a member from the cluster
        member update           Updates a member in the cluster
        member list             Lists all members in the cluster
        snapshot save           Stores an etcd node backend snapshot to a given file
        snapshot restore        Restores an etcd member snapshot to an etcd directory
        snapshot status         Gets backend snapshot status of a given file
        make-mirror             Makes a mirror at the destination etcd cluster
        migrate                 Migrates keys in a v2 store to a mvcc store
        lock                    Acquires a named lock
        elect                   Observes and participates in leader election
        auth enable             Enables authentication
        auth disable            Disables authentication
        user add                Adds a new user
        user delete             Deletes a user
        user get                Gets detailed information of a user
        user list               Lists all users
        user passwd             Changes password of user
        user grant-role         Grants a role to a user
        user revoke-role        Revokes a role from a user
        role add                Adds a new role
        role delete             Deletes a role
        role get                Gets detailed information of a role
        role list               Lists all roles
        role grant-permission   Grants a key to a role
        role revoke-permission  Revokes a key from a role
        check perf              Check the performance of the etcd cluster
        help                    Help about any command

OPTIONS:
```

```
      --cacert=""                              verify certificates of TLS-enabled secure servers us
ing this CA bundle
      --cert=""                                identify secure client using this TLS certificate fi
le
      --command-timeout=5s                     timeout for short running command (excluding dial ti
meout)
      --debug[=false]                          enable client-side debug logging
      --dial-timeout=2s                        dial timeout for client connections
      --endpoints=[127.0.0.1:2379]             gRPC endpoints
      --hex[=false]                            print byte strings as hex encoded strings
      --insecure-skip-tls-verify[=false]       skip server certificate verification
      --insecure-transport[=true]              disable transport security for client connections
      --key=""                                 identify secure client using this TLS key file
      --user=""                                username[:password] for authentication (prompt if pa
ssword is not supplied)
  -w, --write-out="simple"                     set the output format (fields, json, protobuf, simpl
e, table)
```

15.4.1 数据库操作

数据库操作围绕对键值和目录的 CRUD (符合 REST 风格的一套操作：Create) 完整生命周期的管理。

etcd 在键的组织上采用了层次化的空间结构 (类似于文件系统中目录的概念)，用户指定的键可以为单独的名字，如 `testkey`，此时实际上放在根目录 `/` 下面，也可以为指定目录结构，如 `cluster1/node2/testkey`，则将创建相应的目录结构。

> 注：CRUD 即 Create，Read，Update，Delete，是符合 REST 风格的一套 API 操作。

put

```
$ etcdctl put /testdir/testkey "Hello world"
OK
```

get

获取指定键的值。例如

```
$ etcdctl put testkey hello
OK
$ etcdctl get testkey
testkey
hello
```

支持的选项为

`--sort-by` 指定排序字段（CREATE / KEY / MODIFY / VALUE / VERSION）

`--order` 指定排序顺序（ASCEND / DESCEND）

`--consistency` 指定一致性级别（`l` 线性一致，`s` 串行）

del

删除某个键值。例如

```
$ etcdctl del testkey
1
```

15.4.2 非数据库操作

watch

监测一个键值的变化，一旦键值发生更新，就会输出最新的值。

例如，用户更新 `testkey` 键值为 `Hello world`。

```
$ etcdctl watch testkey
PUT
testkey
2
```

member

通过 `list`、`add`、`update`、`remove` 命令列出、添加、更新、删除 etcd 实例到 etcd 集群中。

例如本地启动一个 `etcd` 服务实例后，可以用如下命令进行查看。

```
$ etcdctl member list
422a74f03b622fef, started, node1, http://172.16.238.100:2380, http://172.16.238.100:2379
```

本章小结

etcd 是 Kubernetes 的核心存储组件，为分布式系统提供可靠的键值存储和服务发现能力。

概念	要点
定位	分布式键值存储系统，用于配置管理和服务发现
协议	基于 Raft 一致性算法，保证数据强一致
API	提供 gRPC 和 HTTP API
集群	建议使用奇数节点 (3 或 5 个) 部署
etcdctl	命令行管理工具，支持 put/get/del/watch 等操作
安全	支持 TLS 加密通信和 RBAC 访问控制

延伸阅读

- 容器编排基础：Kubernetes 如何使用 etcd
- 部署 Kubernetes：在集群中部署 etcd

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十六章 容器与云计算

> **版本说明**：云平台的 Kubernetes 版本和容器服务功能更新迅速。本章示例使用通用的 API 版本（如 `apps/v1`），建议查阅各云厂商官方文档获取最新的服务版本和配置指南。

Docker 目前已经得到了众多公有云平台的支持，并成为除虚拟机之外的核心云业务。

除了 AWS、Google、Azure 等，国内的各大公有云厂商，基本上都同时支持了虚拟机服务和基于 Kubernetes 的容器云业务。有的还推出了其他服务，例如容器镜像服务让用户在云上享有安全高效的镜像托管、分发等服务。

本章内容

- 简介
- 腾讯云
- 阿里云
- 亚马逊云
- 多云部署策略

16.1 简介

> **版本说明**：各云平台的容器服务版本更新频繁。本章示例适用于常见的 Kubernetes API 版本。建议访问各云厂商官方文档（如 AWS EKS、Azure AKS、Google GKE、阿里云 ACK、腾讯云 TKE）获取最新信息。

随着容器技术的普及，目前主流的云计算服务商都提供了成熟的容器服务。与容器相关的云计算服务主要分为以下几种类型：

16.1.1 容器编排托管服务

这是目前最主流的形式。云厂商托管 Kubernetes 的控制平面 (Master 节点)，用户只需管理工作节点 (Worker Node)。

- **优势**：降低了 Kubernetes 集群的维护成本，高可用性由厂商保证。
- **典型服务**：AWS EKS，Azure AKS，Google GKE，阿里云 ACK，腾讯云 TKE。

16.1.2 容器实例服务

这一类服务通常被称为 CaaS (Container as a Service)。用户无需管理底层服务器 (EC2/CVM)，只需提供镜像和配置即可运行容器。

- **优势**：极致的弹性，按秒计费，零运维。
- **典型服务**：AWS Fargate，Azure Container Instances，Google Cloud Run，阿里云 ECI。

16.1.3 镜像仓库服务

提供安全、可靠的私有 Docker 镜像存储服务，通常与云厂商的 CI/CD 流水线深度集成。

- **典型服务**：AWS ECR，Azure ACR，Google GCR/GAR，阿里云 ACR。

本章将介绍如何在几个主流云平台上使用 Docker 和 Kubernetes 服务。

16.2 腾讯云

腾讯云容器服务 TKE 是腾讯云提供的 Kubernetes 托管服务，适合把容器化应用部署到云上。官方文档见 腾讯云容器服务。



图 16-1：腾讯云标识

下面的示例只保留创建集群、部署应用、管理镜像和配置加速器这几类最常见操作。



图 16-2：腾讯云容器服务示意图

腾讯云容器服务：TKE 简介

腾讯云容器服务 (TKE, Tencent Kubernetes Engine) 是一款容器编排平台，基于原生 Kubernetes 提供，支持自动扩展、负载均衡、多可用区高可用等企业级功能。TKE 帮助开发者快速部署和管理容器化应用，消除集群运维的复杂度。

基本使用步骤

1. 创建集群

登录腾讯云控制台，进入容器服务模块：

- 选择 “创建集群”，配置集群名称、地域和网络
- 选择节点配置（云服务器规格和数量）
- 设置 Kubernetes 版本和安全组
- 完成创建后获得集群 kubeconfig 文件

```
# 下载 kubeconfig 文件后，配置本地环境
export KUBECONFIG=/path/to/kubeconfig.yaml
kubectl cluster-info
```

2. 部署容器应用

创建 Deployment 部署应用：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

应用配置文件：

```
kubectl apply -f deployment.yaml
kubectl get pods
kubectl get svc
```

3. 管理镜像

使用腾讯云容器镜像服务 (TCR) 存储和分发私有镜像：

```
# 登录腾讯云镜像仓库
docker login ccr.ccs.tencentyun.com -u <username>

# 标记本地镜像
docker tag my-app:latest ccr.ccs.tencentyun.com/namespace/my-app:latest

# 推送镜像到腾讯云
docker push ccr.ccs.tencentyun.com/namespace/my-app:latest
```

腾讯云 Docker 镜像加速器配置

如果你的账号开通了镜像加速器，可以把控制台给出的地址写入 Docker 配置。

Linux 系统配置

编辑 `/etc/docker/daemon.json` 文件（如果不存在则创建）：

```
# 创建或编辑配置文件
sudo mkdir -p /etc/docker
sudo nano /etc/docker/daemon.json
```

添加以下内容：

```
{
  "registry-mirrors": [
    "https://mirror.ccs.tencentyun.com"
  ],
  "insecure-registries": []
}
```

重启 Docker 服务：

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

验证配置：

```
docker info | grep -A 5 "Registry Mirrors"
```

Windows/Mac 配置

对于 Docker Desktop，在设置界面中打开 `Docker Engine`，把上述 `registry-mirrors` 字段写入 JSON 后重启即可。

腾讯云容器镜像服务：TCR

TCR 提供私有镜像仓库、访问控制和镜像分发能力。一个最小示例如下：

```
# 登录到腾讯云 TCR
docker login ccr.ccs.tencentyun.com --username <username>

# 构建并推送镜像
docker build -t my-app:v1.0 .
docker tag my-app:v1.0 ccr.ccs.tencentyun.com/my-namespace/my-app:v1.0
docker push ccr.ccs.tencentyun.com/my-namespace/my-app:v1.0
```

TKE 集群中使用 TCR 镜像

配置镜像拉取凭证后，在 Deployment 中直接引用 TCR 镜像：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      imagePullSecrets:
      - name: tcr-secret
      containers:
      - name: my-app
        image: ccr.ccs.tencentyun.com/my-namespace/my-app:v1.0
        ports:
        - containerPort: 8080
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "500m"
```

16.3 阿里云

如图 16-3：所示，阿里云是国内主流云服务平台之一。



图 16-3：阿里云标识

阿里云创立于 2009 年，是中国较早的云计算平台。阿里云致力于提供安全、可靠的计算和数据处理能力。

阿里云的客户群体中，活跃着微博、虎牙、魅族、优酷等一大批明星互联网公司。在天猫双 11 全球狂欢节等极富挑战的应用场景中，阿里云保持着良好的运行纪录。

阿里云容器服务 Kubernetes 版 ACK 提供了高性能、可伸缩的容器应用管理服务，支持在一组云服务器上通过 Docker 容器来进行应用生命周期管理。容器服务极大简化了用户对容器管理集群的搭建工作，无缝整合了阿里云虚拟化、存储、网络和安全能力。容器服务提供了多种应用发布方式和流水线般的持续交付能力，原生支持微服务架构，助力用户无缝上云和跨云管理。

图 16-4：阿里云容器服务示意图（请访问 阿里云容器服务 ACK 官方文档 查看最新界面）

阿里云容器服务 ACK 简介

阿里云容器服务 Kubernetes 版 (ACK, Container Service for Kubernetes) 是一款托管式 Kubernetes 服务，基于开源 Kubernetes 构建，提供企业级的容器编排和管理能力。ACK 集成了阿里云存储、网络和安全能力，支持多种应用部署模式和持续交付流程。

基本使用步骤

1. 创建集群

登录阿里云控制台，进入容器服务 > Kubernetes 集群：

- 点击 “创建集群”，选择集群配置
- 配置集群名称、地域、可用区和节点类型
- 选择节点规格和数量（支持弹性伸缩）
- 配置网络参数和安全设置
- 完成创建，下载 kubeconfig 文件

```
# 配置本地 kubectl
export KUBECONFIG=/path/to/kubeconfig.yaml
kubectl get nodes
```

2. 部署容器应用

通过 Deployment 部署应用示例：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web
        image: registry.cn-hangzhou.aliyuncs.com/myapp/web:v1
        ports:
        - containerPort: 8080
        resources:
          limits:
            memory: "512Mi"
            cpu: "500m"
```

部署应用：

```
kubectl apply -f deployment.yaml
kubectl get pods -o wide
kubectl logs <pod-name>
```

3. 暴露服务

创建 Service 暴露应用：

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: web
```

应用：

```
kubectl apply -f service.yaml
kubectl get svc web-service
```

阿里云 Docker 镜像加速器配置

为了加快从阿里云镜像源拉取官方镜像的速度，可以配置镜像加速器。阿里云为容器服务 ACK 用户提供了免费的镜像加速服务。

获取加速器地址

登录阿里云容器镜像服务控制台，在 “镜像工具” > “镜像加速器” 中可获取个人的加速器地址（类似于 `https://xxxxxx.mirror.aliyuncs.com`）。

Linux 系统配置

编辑或创建 `/etc/docker/daemon.json` 文件：

```
sudo mkdir -p /etc/docker
sudo nano /etc/docker/daemon.json
```

添加或修改以下内容（替换为你的加速器地址）：

```
{
  "registry-mirrors": [
    "https://xxxxxx.mirror.aliyuncs.com"
  ]
}
```

重新加载并重启 Docker：

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

验证配置生效：

```
docker info | grep -A 5 "Registry Mirrors"
```

Windows/Mac 配置

在 Docker Desktop 的 Settings 中：

1. 进入 “Docker Engine” 标签
2. 编辑 JSON 配置，添加 `registry-mirrors` 字段
3. 点击 “Apply & Restart”

测试加速效果

```
# 从加速器拉取镜像（速度应该明显提升）
docker pull nginx:latest
time docker pull alpine:latest
```

阿里云容器镜像服务：ACR

阿里云容器镜像服务 (ACR, Container Registry) 是企业级的容器镜像存储和分发平台：

- **私有镜像仓库**：支持多个命名空间，细粒度权限控制
- **镜像构建**：云端编译和构建，支持自动化 CI/CD
- **镜像扫描**：自动检测镜像中的漏洞和恶意代码
- **跨地域复制**：支持镜像在多个地域的同步和加速
- **集成 ACK**：与 ACK 无缝集成，自动身份认证
- **镜像版本管理**：标签管理、镜像过期清理、保留策略

完整推送/拉取示例

```
# 登录阿里云镜像仓库（使用 Docker 登录）
# 使用阿里云账户 ID 和 RAM 访问密钥或密码
docker login registry.cn-hangzhou.aliyuncs.com \
  --username=<阿里云账户ID>

# 拉取阿里云公开镜像
docker pull registry.cn-hangzhou.aliyuncs.com/library/nginx:latest

# 构建本地镜像
docker build -t my-app:v1.0 .

# 标记镜像为阿里云仓库地址
docker tag my-app:v1.0 \
  registry.cn-hangzhou.aliyuncs.com/myapp/my-app:v1.0

# 推送镜像到阿里云 ACR
docker push registry.cn-hangzhou.aliyuncs.com/myapp/my-app:v1.0

# 在 Dockerfile 中使用 ACR 镜像
FROM registry.cn-hangzhou.aliyuncs.com/myapp/my-app:v1.0
COPY . /app
RUN echo "已成功使用阿里云镜像"
```

ACK 集群中使用 ACR 镜像

在 ACK 集群中，需要先配置镜像拉取凭证（Secret），然后在 Deployment 中引用：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      # 如果是私有镜像，需配置镜像拉取凭证
      imagePullSecrets:
      - name: acr-secret
      containers:
      - name: web
        image: registry.cn-hangzhou.aliyuncs.com/myapp/web:v2.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "500m"
      affinity:
        # 配置 Pod 反亲和性，分散到不同节点
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web
              topologyKey: kubernetes.io/hostname
```

创建镜像拉取凭证

在 ACK 集群中创建 Secret，用于拉取私有镜像：

```
# 创建镜像拉取 Secret
kubectl create secret docker-registry acr-secret \
  --docker-server=registry.cn-hangzhou.aliyuncs.com \
  --docker-username=<阿里云账户ID> \
  --docker-password=<RAM访问密钥或密码> \
  --docker-email=<邮箱地址>

# 查看创建的 Secret
kubectl get secret acr-secret
kubectl describe secret acr-secret
```

ACR 优势

- 在 ACK 集群中与镜像仓库无缝集成，简化身份认证
- 支持 Helm Chart 存储和版本管理，方便应用交付
- 提供完整的图形化镜像仓库管理界面
- 完整的审计日志和操作追踪功能
- 支持镜像自动扫描和漏洞报告

16.4 亚马逊云

如图 16-5：所示，AWS 是全球主流云服务平台之一。



图 16-5：AWS 标识

AWS，即 Amazon Web Services，是亚马逊 (Amazon) 公司的 IaaS 和 PaaS 平台服务。AWS 提供了一整套基础设施和应用程序服务，使用户几乎能够在云中运行一切应用程序：从企业应用程序和大数据项目，到社交游戏和移动应用程序。AWS 面向用户提供包括弹性计算、存储、数据库、应用程序在内的一整套云计算服务，能够帮助企业降低 IT 投入成本和维护成本。

在容器领域，AWS 目前主流能力可以按场景分为四类：

1. `Amazon EKS`：托管 Kubernetes 控制平面，适合标准云原生工作负载。
2. `Amazon ECS`：AWS 原生容器编排服务，适合深度集成 AWS 生态 (IAM、ALB、CloudWatch) 场景。
3. `AWS Fargate`：无服务器容器运行时，可与 EKS/ECS 结合使用，减少节点运维。
4. `Amazon ECR`：镜像仓库服务，提供私有镜像管理、扫描与访问控制。

实践建议：

- 团队已具备 Kubernetes 经验，优先选择 EKS；
- 追求更低运维复杂度且业务主要运行在 AWS，可优先 ECS + Fargate；
- 无论编排方案如何，都建议使用 ECR 统一管理镜像生命周期。

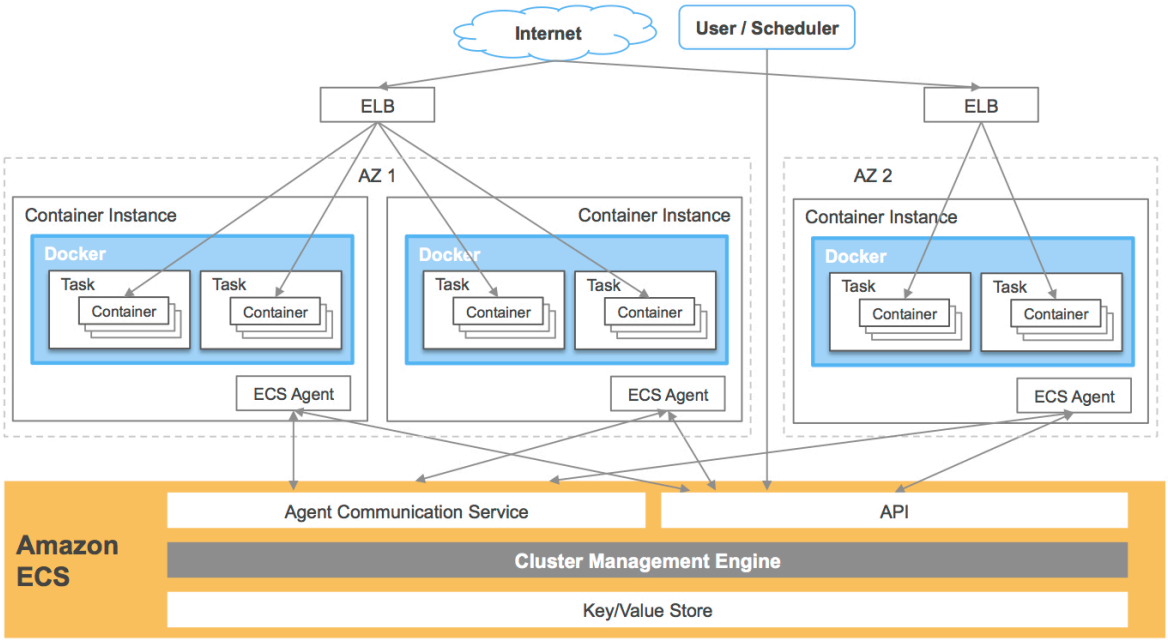


图 16-6：AWS 容器服务示意图

16.5 多云部署策略

企业在选择容器云平台时，通常会在 AWS EKS，Azure AKS，Google GKE 以及国内的阿里云 ACK，腾讯云 TKE 之间进行权衡。

16.5.1 三大公有云 Kubernetes 服务对比

特性	Google GKE	AWS EKS	Azure AKS
版本更新	最快，通常是 K8s 新特性的首发地	相对保守，注重稳定性	跟随社区，更新速度适中
控制平面管理	全托管，自动升级，$0.10/h$（有$FreeTier$抵扣），托管 0.10/h	全托管；Free 层适合开发/测试，Standard/Premium 层默认包含 Uptime SLA	
节点管理	GKE Autopilot 模式完全托管节点	Managed Node Groups 简化管理	Virtual Machine Scale Sets
网络模型	VPC-native, 性能优秀	AWS VPC CNI, Pod 直接获取 VPC IP	Azure CNI (消耗 IP 多) 或 Kubenet
集成度	与 GCP 数据分析、AI 服务集成紧密	与 AWS IAM, ALB, CloudWatch 集成深度高	与 Active Directory, Azure DevOps 集成好

16.5.2 多云部署策略

随着企业业务的扩展，单一云平台可能无法满足所有需求，多云部署成为趋势。

1. 跨云灾备：Active-Passive

主要业务运行在一个云 (如 AWS)，数据实时复制到另一个云 (如阿里云)。当主云发生故障时，流量切换到备云。

Built with mdPress

- **优点**：架构相对简单，数据一致性好控制。
- **缺点**：资源闲置浪费，切换可能有 RTO。

2. 多活部署：Active-Active

业务同时在多个云上运行，通过全局流量管理 (DNS/GSLB) 分发流量。

- **优点**：高可用，就近接入提升用户体验。
- **缺点**：数据同步复杂，跨云网络延迟问题。

3. 混合云

核心数据和敏感业务保留在私有云 (IDC)，弹性业务或前端业务部署在公有云。

- **工具**：Google Anthos，AWS Outposts，Azure Arc 都是为了解决混合云统一管理而生。

16.5.3 建议

- **技术选型**：尽量使用标准的 Kubernetes API，避免过度依赖特定云厂商的 CRD 或专有服务，以保持应用的可移植性。
- **IaC 管理**：使用 Terraform 或 Pulumi 等工具统一管理多云基础设施。

本章小结

本章介绍了公有云服务对 Docker 的积极支持，以及新出现的容器云平台。

事实上，Docker 技术的出现自身就极大推动了云计算行业的发展。

通过整合公有云的虚拟机和 Docker 方式，可能获得更多的好处，包括

- 更快速的持续交付和部署能力；
- 利用内核级虚拟化，对公有云中服务器资源进行更加高效地利用；
- 利用公有云和 Docker 的特性更加方便的迁移和扩展应用。

同时，容器将作为与虚拟机类似的业务直接提供给用户使用，极大的丰富了应用开发和部署的场景。

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第十七章 容器其它生态

> **版本说明（核验日期：2026-05-16）**：本章介绍的工具和运行时（Podman、Buildah、Skopeo、containerd、Kata Containers、gVisor、WasmEdge 等）都保持活跃的开发。建议：
>
> - 查阅各项目官方文档获取最新版本
> - 在生产环境使用前验证版本兼容性
> - 关注官方发布说明了解重大变更

本章将介绍 Docker 和 Kubernetes 之外的容器生态技术。

同时，Docker 自身的生态也在向云构建、AI 本地推理和企业级桌面安全扩展。当前需要额外关注：

- **Docker Model Runner**：在 Docker Desktop / Docker Engine 中管理、运行和服务本地 AI 模型，支持 OpenAI 与 Ollama 兼容 API，并可将 GGUF、Safetensors 等模型文件作为 OCI Artifact 管理。
- **Docker Build Cloud**：通过远程 BuildKit 和共享构建缓存加速本地与 CI 构建，适合多平台镜像和团队共享缓存场景。
- **Docker Offload**：把容器构建和运行卸载到云端，适合 VDI、受限本机或不支持嵌套虚拟化的开发环境。
- **Hardened Docker Desktop / Enhanced Container Isolation (ECI)**：通过更强的命名空间隔离、敏感挂载保护和系统调用限制降低桌面容器逃逸风险。

本章内容

- Fedora CoreOS 简介
  - 专为容器化工作负载设计的操作系统。
- Fedora CoreOS 安装与配置
  - CoreOS 的安装方式与基本配置。
- Podman
  - 兼容 Docker CLI 的下一代无守护进程容器引擎。
- Buildah
  - 无需守护进程的 OCI 容器镜像构建工具。
- Skopeo
  - 远程检查和管理容器镜像的利器。
- containerd
  - 作为现代容器生态基石的核心容器运行时。
- 安全容器运行时
  - 通过提供更强隔离性来保证安全的技术方案（如 Kata Containers、gVisor）。
- WebAssembly
  - 一种极具潜力的轻量级跨平台二进制指令格式。

Built with mdPress

17.1 Fedora CoreOS 简介

> **版本说明**：Fedora CoreOS 定期发布更新。建议访问 官方下载页面 获取最新版本和兼容性信息。

Fedora CoreOS 是一个自动更新的，最小的，整体的，以容器为中心的操作系统，不仅适用于集群，而且可独立运行，并针对运行 Kubernetes 进行了优化。它旨在结合 CoreOS Container Linux 和 Fedora Atomic Host 的优点，将 Container Linux 中的 Ignition 与 rpm-ostree 和 Project Atomic 中的 SELinux 强化等技术相集成。其目标是提供最佳的容器主机，以安全，大规模地运行容器化的工作负载。

17.1.1 FCOS 特性

一个最小化操作系统

FCOS 被设计成一个基于容器的最小化的现代操作系统。它比现有的 Linux 安装平均节省 40% 的 RAM (大约 114M) 并允许从 PXE 或 iPXE 非常快速的启动。

系统初始化

Ignition 是一种配置实用程序，可读取配置文件 (JSON 格式) 并根据该配置配置 FCOS 系统。可配置的组件包括存储，文件系统，systemd 和用户。

Ignition 在系统首次启动期间 (在 initramfs 中) 仅运行一次。由于 Ignition 在启动过程中的早期运行，因此它可以在用户空间开始启动之前重新对磁盘分区，格式化文件系统，创建用户并写入文件。当 systemd 启动时，systemd 服务已被写入磁盘，从而加快了启动时间。

自动更新

FCOS 使用 rpm-ostree 系统进行事务性升级。无需像 yum 升级那样升级单个软件包，而是 rpm-ostree 将 OS 升级作为一个原子单元进行。新的 OS 部署在升级期间进行，并在下次重新引导时生效。如果升级出现问题，则一次回滚和重新启动会使系统返回到先前的状态。确保了系统升级对群集容量的影响降到最小。

容器工具

对于诸如构建，复制和其他管理容器的任务，FCOS 用一组容器工具代替了 **Docker CLI**。**podman CLI** 工具支持许多容器运行时功能，例如运行，启动，停止，列出和删除容器和镜像。**skopeo CLI**

工具可以复制，认证和签名镜像。您还可以使用 **crictl CLI** 工具来处理 CRI-O 容器引擎中的容器和镜像。

17.2 Fedora CoreOS 安装

17.2.1 下载 ISO

在下载页面 `Bare Metal & Virtualized` 标签页下载 ISO。

17.2.2 编写 Butane 配置

> **注意**：Fedora CoreOS 配置工具已从 `fcct` (Fedora CoreOS Config Transpiler) 更名为 **Butane**。新版本使用 `.bu` 扩展名和更新的 spec 版本。

```
## example.bu

variant: fcos
version: 1.6.0
passwd:
  users:
    - name: core
      ssh_authorized_keys:
        - ssh-rsa AAAA...
```

将 `ssh-rsa AAAA...` 替换为自己的 SSH 公钥 (位于 `~/.ssh/id_rsa.pub`)。

17.2.3 转换 Butane 配置为 Ignition

```
$ docker run -i --rm quay.io/coreos/butane:release --pretty --strict < example.bu > example.ign
```

17.2.4 挂载 ISO 启动虚拟机并安装

> 虚拟机需要分配 3GB 以上内存，否则会无法启动。

在虚拟机终端执行以下命令安装：

```
$ sudo coreos-installer install /dev/sda --ignition-file example.ign
```

安装之后重新启动即可使用。

17.2.5 使用

```
$ ssh core@虚拟机IP

$ podman --version
```

17.3 Podman - 下一代 Linux 容器工具

> **版本说明**：Podman 保持活跃的开发和发布周期。建议访问 Podman 官方文档 和 GitHub Releases 获取最新版本。

Podman 是一个无守护进程、与 Docker 命令高度兼容的下一代 Linux 容器工具。它由 Red Hat 开发，旨在提供一个更安全的容器运行环境。

17.3.1 Podman vs Docker

Podman 和 Docker 在设计理念上存在显著差异，主要体现在架构和权限模型上。

特性	Docker	Podman
架构	C/S 架构，依赖守护进程 (`dockerd`)	无守护进程 (Daemonless)
权限	默认需要 root 权限 (虽有 Rootless 模式)	默认支持 Rootless (非 root 用户运行)
生态	完整的生态系统 (Compose, Swarm)	专注单机容器，配合 Kubernetes 使用
镜像构建	`docker build`	`podman build` 或 `buildah`

17.3.2 安装

Podman 支持多种操作系统，安装过程也相对简单。

CentOS / RHEL

```
$ sudo yum -y install podman
```

macOS

macOS 上需要安装 Podman Desktop 或通过 Homebrew 安装：

```
$ brew install podman
$ podman machine init
$ podman machine start
```

17.3.3 基本使用

podman 的命令行几乎与 docker 完全兼容，大多数情况下，你只需将 docker 替换为 podman 即可。

运行容器

```
## $ docker run -d -p 80:80 nginx:alpine

$ podman run -d -p 80:80 nginx:alpine
```

列出容器

```
$ podman ps
```

构建镜像

```
$ podman build -t myimage .
```

17.3.4 Pods 的概念

与 Docker 不同，Podman 支持“Pod”的概念 (类似于 Kubernetes 的 Pod)，允许你在同一个网络命名空间中运行多个容器。

```
## 创建一个 Pod

$ podman pod create --name mypod -p 8080:80

## 在 Pod 中运行容器

$ podman run -d --pod mypod --name webbing nginx
```

17.3.5 迁移到 Podman

如果你习惯使用 docker 命令，可以简单地设置别名：

```
$ alias docker=podman
```

Systemd 集成

Podman 可以生成 systemd 单元文件，让容器像普通系统服务一样管理。

```
## 创建容器

$ podman run -d --name myweb -p 8080:80 nginx

## 生成 systemd 文件

$ podman generate systemd --name myweb --files --new

## 启用并启动服务

$ systemctl --user enable --now container-myweb.service
```

Podman Compose

虽然 Podman 兼容 Docker Compose，但在某些场景下你可能需要明确使用 `podman-compose`。

```
$ pip3 install podman-compose
$ podman-compose up -d
```

17.4 Buildah - 容器镜像构建工具

> **版本说明**：Buildah 与 Podman 和 Skopeo 共同维护。建议查阅 Buildah 官方文档 和 GitHub Releases 了解最新版本。

本节介绍 Buildah，包括其基础概念、应用场景以及基本指令。

17.4.1 Buildah 简介

Buildah 是一个用于构建 OCI（Open Container Initiative）兼容格式容器镜像的开源命令行工具。与 Docker 需要一直运行的守护进程（daemon）不同，Buildah 的设计初衷是无需守护进程（daemonless）即可工作，并且也不强制要求 root 权限（rootless）。这使得在持续集成/持续部署（CI/CD）环境中构建镜像时能够更加轻量且安全。

Buildah 由 Red Hat 主导开发，通常和 Podman、Skopeo 一起使用，被认为是构建、运行和管理容器的一套现代化工具链。在很多需要增强安全性和无需依赖守护进程的场景中，Buildah 是 `docker build` 命令的最佳替代方案。

17.4.2 核心特性

- **无守护进程（Daemonless）**：Buildah 直接通过系统调用拉取、构建和推送镜像，减少了单点故障的风险和资源开销。
- **构建效率高**：可以挂载镜像的根文件系统到本地，并直接利用宿主机的工具对其进行操作，非常灵活。
- **兼容性**：不仅支持处理传统的 Dockerfile，还能完全兼容 OCI（Open Container Initiative）标准和 Docker 格式。
- **与 Podman 集成**：Podman 自身构建镜像的命令 `podman build` 底层实际上也是依赖 Buildah 库来实现的。

17.4.3 安装 Buildah

在许多主流的 Linux 发行版中都可以通过包管理器直接安装 Buildah。

以 Fedora/CentOS/RHEL 为例：

```
$ sudo dnf install -y buildah
```

以 Ubuntu/Debian 为例（需引入官方源后）：

```
$ sudo apt-get update
$ sudo apt-get -y install buildah
```

17.4.4 基础用法示例

1. 从现有的 Dockerfile 构建镜像

Buildah 最常见的用法就是像 Docker 一样根据 `Dockerfile` 来构建镜像，可以直接使用 `buildah bud`（或者 `buildah build-using-dockerfile`）命令：

```
$ buildah bud -t my-app:latest .
```

可以看到在这点上，它与 `docker build` 的体验完全一致。

2. 交互式从空镜像开始构建

除了使用 Dockerfile，Buildah 最强大的功能来自于它的交互式和脚本化构建机制。我们可以从一个极简的镜像（或基础镜像）开始构建：

```
# 获取一个基础镜像
$ container=$(buildah from alpine:latest)

# 获取挂载点，并查看其路径
$ mnt=$(buildah mount $container)
$ echo $mnt
/var/lib/containers/storage/overlay/xxx/merged

# 利用宿主机直接创建文件，而不需要在容器内部运行命令
$ echo "Hello Buildah" > $mnt/hello.txt

# 添加一些配置和命令
$ buildah config --cmd "cat /hello.txt" $container

# 将容器提交为镜像
$ buildah commit $container my-hello-image:latest

# 构建完成后可以卸载并清理容器上下文
$ buildah unmount $container
$ buildah rm $container
```

这种模式在自动化流水线中极为有用，因为我们可以将上述过程编写成标准的 bash 脚本，无需为了构建镜像而撰写只在其独立语法中运行的 Dockerfile 指令。

3. 查看和推送镜像

通过 `buildah images` 可以查看当前环境中的镜像。推送镜像到外部 Registry 也十分安全方便：

```
# 查看本地构建的镜像
$ buildah images

# 推送镜像到 Docker Hub（注意需要先登录）
$ buildah push my-hello-image:latest docker://docker.io/username/my-hello-image:latest
```

结合其无需特权和灵活脚本的优点，Buildah 正变得越来越受到构建和分发 OCI 镜像的用户喜爱。

17.5 Skopeo - 容器镜像管理工具

> **版本说明**：Skopeo 属于 containers 项目生态。建议访问 Skopeo 官方仓库 和 GitHub Releases 获取最新版本信息。

本节介绍 Skopeo，包括其基础概念、应用场景以及基本指令。

17.5.1 Skopeo 简介

Skopeo 是一个由 Red Hat 赞助开源的命令行工具，它可以在不需要运行容器守护进程（如 Docker Daemon）的前提下，对容器镜像进行极其高效的操作和管理，包括：检查、复制、删除和签名等操作。

Skopeo 最大的特点是其可以在“不将镜像拉取到本地”的情况下，直接在远端 Registry（镜像仓库）之间完成检查和搬运，从而大幅度节省带宽和磁盘空间。这也是它在容器运维和分发领域非常受欢迎的原因。

17.5.2 核心特性

- **远程巡检**：通过 `skopeo inspect` 可以查看远端仓库中镜像的元数据（例如包含哪些层、环境变量信息、入口命令等），而完全无需拉取该镜像。
- **镜像复制与同步**：支持各种格式之间的相互传输，如在不同的容器仓库之间、或者从仓库拉取到本地的目录、或者存储为 OCI 布局结构等。
- **镜像删除**：可以远程删除仓库中的镜像（需要拥有权限）。
- **签名验证**：支持在分发镜像前进行数字签名以保障安全性。

17.5.3 安装 Skopeo

类似于 Buildah，Skopeo 也直接包含在大部分主流的 Linux 源中。

在 Fedora/CentOS/RHEL 等发行版中：

```
$ sudo dnf install -y skopeo
```

在 Ubuntu/Debian 中：

```
$ sudo apt-get update
$ sudo apt-get -y install skopeo
```

如果是 macOS 环境，可以通过 Homebrew 安装：

```
$ brew install skopeo
```

17.5.4 基础用法示例

1. 远程检查镜像

有时候我们只想知道远端镜像的详细信息，并不想拉取它。下面是检查 Docker Hub 上的 Alpine 镜像的例子：

```
$ skopeo inspect docker://docker.io/library/alpine:latest
```

这个命令会返回一段 JSON 格式的数据，其中包含了诸如镜像摘要（Digest）、创建时间、架构（Architecture）、标签（Tags）等丰富信息。在自动化的工具和系统审查环境中，这是一个不可或缺的利器。

2. 同步与复制镜像

`skopeo copy` 是使用最为广泛的命令。它可以在不同的仓库、不同的格式之间无缝搬运镜像。

例如，从一个公共仓库直接搬运镜像到一个私有企业仓库（无需将其先 `docker pull` 到本地，再 `docker push`）：

```
$ skopeo copy docker://docker.io/library/alpine:latest docker://registry.example.com/library/alpine:
latest
```

又或者，你可以将远程镜像拉取到本地，只为了检查其拆解后的格式，比如将镜像解压到本地某个目录下以 OCI 规范存放：

```
$ skopeo copy docker://docker.io/library/alpine:latest oci:alpine-oci
```

如果我们要将本地的某个目录下的打包好的镜像再次推向 Registry 或转换为其它存储类型也是完全支持的，诸如：

- `docker://` 远端 Registry
- `docker-archive:` / `docker-daemon:` Docker 对应的归档文件或本地守护进程
- `oci:` / `oci-archive:` OCI 相关文件格式
- `dir:` 本地纯目录

3. 校验机制与安全

如果你不信任公开仓库上的镜像，或是需要通过特定的 TLS 证书和鉴权，Skopeo 的功能也是能很好的支持的，比如它可以直接传递诸如 `--src-creds` 或 `--dest-tls-verify=false` 等参数，这在进行网络隔离的复杂镜像搬运操作中常常会用到。

对于复杂的容器环境或那些纯粹用于镜像资产管理的节点来说，Skopeo 提供了一个直接而强大的数据“搬运工”。

Built with mdPress

17.6 containerd - 核心容器运行时

> **版本说明**：containerd 和 nerdctl 保持活跃的发布周期。建议查阅 containerd 官方文档 和 nerdctl GitHub Releases 获取最新版本信息。

本节介绍 containerd，它是现代容器技术栈中最为核心的基础组件之一。了解 containerd 有助于更深入地理解 Docker 和 Kubernetes 的底层运行机制。

17.6.1 containerd 简介

containerd 是一个行业标准的容器运行时，它最初是由 Docker 引擎中剥离出来的一个核心组件，后来 Docker 将其捐赠给了云原生计算基金会（CNCF），目前已经是一个 CNCF 毕业（Graduated）项目。

它的主要职责是管理单个宿主机上完整的容器生命周期，包括：

- 镜像的传输和存储
- 容器执行和管理
- 存储和网络接口的管理

简单来说，当你在使用 Docker 或者 Kubernetes 时，真正去底层调用操作系统接口（如 Linux 的 Namespace 和 Cgroups）来启动和管理容器进程的，往往是 containerd（及它所调用的 runc 组件）。

17.6.2 与 Docker 和 Kubernetes 的关系

> **版本说明**：本章节涉及 containerd、Docker 和 Kubernetes 的多个版本。建议查阅官方文档了解最新的兼容性信息。

理解 containerd，首先要理清它与用户日常操作的 Docker 以及 Kubernetes 的关系。

Docker 的架构

在早期，Docker 引擎是一个包含了所有功能的单体架构。随着技术的发展和标准化要求，Docker 将底层关于容器运行时的部分解耦出来，形成了 `containerd` 和 `runc`。

当你执行一个 `docker run` 命令时，调用链路大致如下：

1. Docker Client 发送请求给 Docker Daemon（`dockerd`）。
2. `dockerd` 将请求转发给 `containerd`。
3. `containerd` 准备好镜像和容器的必要环境，然后调用 `runc`。
4. `runc` 负责按照 OCI（Open Container Initiative）标准，拉起并运行真正的容器进程。

因此，Docker 现在实际上是构建在 containerd 之上的一个包含更多开发者友好特性（如构建镜像、Compose 管理等）的增强平台。

Kubernetes 与 CRI

Kubernetes 作为一个容器编排系统，为了屏蔽底层不同容器运行时的实现差异，引入了 CRI（Container Runtime Interface）标准。

- 早期版本中，Kubernetes 默认使用 docker 作为运行时，通过一个名为 `dockershim` 的桥接组件对接 Docker，Docker 再对接 containerd。
- 随着 containerd 原生支持了 CRI 插件，Kubernetes 开始直接与 containerd 通信，去掉了 `dockershim` 和 `dockerd` 的中间层。这就是为什么从 Kubernetes 1.24+ 开始“弃用 Docker”引发了广泛关注，实际上 Kubernetes 只是弃用 `dockershim`，底层可以使用 containerd、CRI-O 等符合 CRI 的运行时。参见 Kubernetes 官方文档。
- containerd 2.0+ 移除了已弃用的 CRI v1alpha2 接口，仅保留 CRI v1（Kubernetes 1.26+ 仅支持 CRI v1）。如果集群中仍有依赖 CRI v1alpha2 的组件，升级 containerd 2.x 前需先完成迁移。containerd 2.3+ 是 2.x 系列首个 LTS 版本，支持从 1.7 LTS 直接升级，生产环境推荐使用。详见 containerd 发布说明。

17.6.3 为什么直接使用 containerd?

对普通应用开发者来说，Docker 依然是本地开发和测试的首选。但对于构建云平台、自动化流水线或深度管理 Kubernetes 集群的系统工程师来说，直接使用 containerd 可以带来：

- **更高的性能与更少的开销**：去掉了 Docker Daemon 等附加组件的资源占用，链路更短。
- **更强的稳定性**：作为专注于运行时的底层组件，它的核心功能极为稳定且更新受控。
- **直接符合 Kubernetes CRI 标准**：在生产级 Kubernetes 集群中作为标准配置。

17.6.4 基础用法与工具介绍

不同于 `docker` 命令行工具，containerd 提供了不同的客户端来满足不同的使用场景：

- **ctr**：containerd 自带的调试用客户端。它功能比较基础，主要用于开发者在开发 containerd 时进行快速调试，一般不作为最终用户的日常管理工具。
- **crictl**：Kubernetes 提供的 CRI 命令行工具。它用于排查 Kubernetes 节点上的容器和沙箱（Pod）问题。
- **nerdctl**：这是一个由 containerd 项目维护者开发的，完全兼容 Docker CLI 体验的 containerd 命令行客户端。对于习惯了 `docker run/ps/build` 命令的用户来说，`nerdctl` 可以作为直接操作 containerd 的理想替代品，并且它还支持直接构建镜像（依赖 BuildKit）。

nerdctl 使用示例

安装完 containerd 和 nerdctl 后，你可以体验到几乎与 Docker 完全一致的命令行：

```
# 启动一个 nginx 容器
$ nerdctl run -d -p 8080:80 --name my-nginx nginx:alpine

# 查看运行中的容器
$ nerdctl ps

# 查看本地镜像
$ nerdctl images
```

对于那些希望在生产服务器上剥离 Docker 庞大体积，但又想要保留类似 Docker 方便的命令行体验的用户，`containerd` + `nerdctl` 是一个极佳的组合。

17.7 安全容器运行时

> **版本说明**：Kata Containers、gVisor 和 Firecracker 均为活跃开发的项目。建议查阅各项目官方文档获取最新版本和兼容性信息：Kata Containers、gVisor、Firecracker。

本节介绍容器技术生态中的安全运行时机制，主要探讨在隔离性和安全性上比标准 Linux 容器更进一步的方案，重点介绍 Kata Containers 和 gVisor。

17.7.1 为什么需要安全容器?

标准的 Linux 容器（如 Docker、Podman 或基础的 containerd/runc 所提供的）依赖于 Linux 内核的 **Namespace（命名空间）** 和 **Cgroups（控制组）** 来实现进程级别的隔离与资源限制。这种轻量级的虚拟化方式共享同一个宿主机的内核。

尽管这种方式在性能和启动速度上拥有巨大优势，但也带来了一个显著的缺点：**隔离性（Isolation）不足**。如果某个容器内的恶意进程利用了宿主机内核的漏洞完成了越狱（Privilege Escalation），它将对整个宿主机以及其上运行的所有其他容器造成毁灭性威胁。

如果在公有云环境（多租户场景）或运行不可信的第三方代码时，共享内核显然是不够安全的。为了解决这一问题，社区推出了“安全容器（Secure Containers/Sandboxed Containers）”的概念。安全容器的核心理念是：提供类似虚拟机的强隔离性，同时保持类似容器的轻量、快速启动和标准化管理。

17.7.2 什么是 Kata Containers?

Kata Containers 是一个开源项目，由 OpenStack Foundation（现更名为 OpenInfra Foundation）托管。它将早期的两个项目——Intel Clear Containers 和 Hyper runV 结合而成。

Kata Containers 的核心思路是：**使用轻量级的虚拟机（Lightweight VM）来运行每一个容器或者 Pod**。

工作原理

当使用 Kata Containers 时，它不是在宿主机上启动一个普通的独立进程，而是调用一个精简高度优化的虚拟机管理程序（如 QEMU、Firecracker 或 Cloud Hypervisor）启动一个小型的虚拟机。容器内的应用进程运行在这个虚拟机拥有独立特制内核的沙箱中。

- **高度隔离**：因为拥有自己的独立内核，即使容器内的应用利用内核漏洞溢出，它也只能破坏虚拟机虚拟出来的内核，根本无法触及宿主机真正的内核。
- **兼容性**：Kata Containers 完全实现了 OCI（Open Container Initiative）规范和 CRI（Container Runtime Interface）标准。这意味着它可以作为 `containerd` 或 `Docker` 的底层运行时无缝替换默认的 `runc`。
- **与 Kubernetes 集成**：在 Kubernetes 中，你可以为一个特定的 Pod 指定 `runtimeClassName: kata`，让高敏感的任务自动运行在虚拟机级别的隔离环境中。

17.7.3 什么是 gVisor?

gVisor 是由 Google 开发并开源的一种不同流派的沙箱容器运行时方案。

它采取了与 Kata Containers 完全不同的技术路线，它不是启动完整的虚拟机，而是提供了一个 **应用态内核（User-space Kernel）**。

工作原理

gVisor 的核心组件是一个名为 **Sentry** 的用户空间进程。Sentry 扮演了一个“内核代理”的角色。

- 当容器内的应用想要进行系统调用（System Call，比如读写文件、网络通信）时，这些调用会被 Sentry 拦截并进行一层虚拟化处理，然后再由 Sentry 把经过安全过滤和转换的请求转发给宿主机内核。
- 因为 Sentry 在用户态实现了一套 Linux 系统调用接口，它极大减少了应用直接接触底层操作系统内核的表面积。这样就有效防御了利用底层内核漏洞突破隔离的攻击方式。
- gVisor 同样兼容 OCI 规范，其核心运行时组件称为 `runsc`，可以作为底层运行时与 Docker 或 Kubernetes 进行集成。

Built with mdPress

17.7.4 总结与对比

特性	标准 Linux 容器 (runc)	Kata Containers	gVisor (runsc)
隔离技术	Namespace & Cgroups	轻量级虚拟机 (独立内核)	用户态内核 (系统调用拦截)
安全性	较低（共享宿主机内核）	极高（硬件级虚拟化隔离）	高（减少内核攻击面）
性能开销	极小（原生进程）	中等（因轻量化而优于传统 VM）	中等到较高（取决于系统调用频率开销）
启动速度	极快（毫秒/秒级）	快（秒级之内）	快（接近原生容器）
兼容性	完美（所有系统调用均支持原始实现）	极好（拥有完整内核）	好（但在极少数复杂的未被拦截支持的系统调用中可能报错）

如今，诸如 AWS 这样的云厂商也推出了针对无服务器容器功能（Serverless Containers）的高度优化的轻量级虚拟机管理器 **Firecracker**，可以看作是安全容器生态中与 Kata 类似方案的底层基石。

对于普通的微服务开发来说可能不需要考虑使用安全容器，但在提供多租户平台即服务（PaaS）、运行无状态边缘计算函数（FaaS）等对安全隔离要求极高的场景中，以 Kata Containers 和 gVisor 为代表的安全容器技术展现出了巨大的价值。

Built with mdPress

17.8 WebAssembly 与容器

> **版本说明**：WebAssembly 和相关的 Wasm 运行时（如 WasmEdge、Spin）处于快速演进阶段。建议查阅 WebAssembly 官方文档、WasmEdge 和 Spin 官方文档获取最新信息。

本节介绍 WebAssembly (简写为 Wasm) 以及它为何成为现代容器生态中备受瞩目的前沿技术路线。

17.8.1 什么是 WebAssembly?

WebAssembly (Wasm) 最初是由 W3C 主导的一项为了解决网页中 JavaScript 性能瓶颈而发明的技术标准。它是一种小体积的、加载极快的、提供安全沙盒的高效二进制格式的指令集架构。通过将 C/C++、Rust、Go 等高级语言编译成 `.wasm` 格式，这些程序可以直接在所有现代的浏览器中以接近原生代码的速度安全地运行。

然而，一项原本用于前端领域的技术，为何如今却与容器云计算生态产生了强烈的化学反应?

因为开源社区很快意识到，Wasm 所具备的核心特性完美契合了云原生后端的诉求。人们制定了诸如 **WASI (WebAssembly System Interface)** 这样的标准，将其能力从浏览器扩展到了服务器端操作系统上。

17.8.2 Wasm 与容器特性的完美契合

将 Wasm 应用于服务端时，它展现出了一些可能比传统 Linux 容器更为优异的特性：

1. **极速的冷启动性能**： 传统 Linux 容器虽然比虚拟机轻量很多，但它启动依然需要建立 Namespace 和 Cgroups 以及一整套文件系统，通常需要近百毫秒到几秒。而 Wasm 模块不需要这样庞杂的环境初始化，能够在几毫秒之内完成从加载到执行，这对于无服务器函数（Serverless Functions）而言是巨大的提升。
2. **跨平台性 (Write Once, Run Anywhere)**： 我们知道 Docker 等容器通常是绑定架构的。如果是 x86_64 平台上打出的镜像，通常无法直接在基于 ARM 的系统（如苹果 M 系列芯片甚至树莓派）上直接运行原生代码，除非使用 QEMU 进行低效转译或者专门构建多架构（Multi-arch）镜像和 manifests。 而 Wasm 二进制本身是平台无关的平台中间语言代码形式！你编译出来的一份 `.wasm` 可执行模块，不用做任何修改，就可以在 x86 的 Linux 服务器、ARM 的边缘设备、甚至是 Windows 和 macOS 上直接通过 Wasm 运行时来驱动和执行。真正做到了 “编写一次，到处运行”。
3. **天然的安全沙箱机制**： Wasm 设计之初就是在不被信任的浏览器沙盒环境中运行未知代码的，因此执行环境非常安全，采用了极好的能力导向安全模型。应用只能访问它被明确授予权限的文件或能力。其默认安全隔离性比起依靠 Namespaces 机制的共享内核的 Linux 容器更加坚固。
4. **极小的包体积**： Linux 容器需要打包一整套依赖甚至是简化的 OS 根目录结构。而一个编译好的功能完善的 Wasm 模块体积常常不到几兆甚至仅仅几十 Kb，极大地加快了存储及网络利用效率。

17.8.3 当 Docker 遇上 WebAssembly

在现代的容器生态系统中，Wasm 并不被看作是要被取代传统的 Docker 或者 Kubernetes 的技术，而是成为了一种 **与 Linux 容器互补并且共生** 的全新工作负载类型。

目前，这通过 OCI (Open Container Initiative) 和 CRI 标准实现了集成上的统一：

1. **将 Wasm 打包为 OCI 镜像**：虽然内容并非传统的 Linux RootFS，但是通过标准化的打包工具同样可以将应用程序及其 `.wasm` 构建结果转化为一个可以被推送至 Docker Hub 或其他 registry 的标准化镜像规范。
2. **通过容器运行时直接执行**：Docker 已经与如 WasmEdge 和 Spin 等高性能的企业级 Wasm 运行时进行了官方集成合作。

如今在 Docker Desktop 或者集成了 `containerd` 的环境中，我们可以十分简易地以类似普通镜像的形式去拉取并运行一个基于 Wasm 编译的后端服务（通过指定相应的 `--platform` 或者是特别的 `--runtime=io.containerd.wasmedge.v1` 设置），将其如同对待一个标准应用进程一样让 Docker 为其

接管日志、配置相关的网络端口映射，甚至通过 Docker Compose 将一个普通的数据库容器实例与一个 Wasm 微服务实例协同起来混布。

17.8.4 总结

随着技术底座如 WASI 规范不断的成熟完善（例如提供完备的套接字网络支持以及系统资源访问支持），我们有理由相信不仅是边缘计算与无服务器调用，会有越来越多对于速度和安全性有极高指标要求的云原生后端微服务开始采用这一颠覆传统边界的轻量级“微型智能体”架构。在可见的将来，Wasm 势必成为云原生与 Docker 生态的重要拼图。

Built with mdPress

本章小结

Docker 并非容器生态的唯一选择，了解其他工具有助于根据场景做出合适的技术选型。

项目	定位	特点
Fedora CoreOS	容器化操作系统	自动更新、不可变基础设施、专为运行容器设计
Podman	容器管理引擎	无守护进程、兼容 Docker CLI、支持 Rootless 模式、支持原生 Pod
Buildah	镜像构建工具	Daemonless 工作模式、灵活的脚本化构建能力
Skopeo	镜像仓库管理	无需拉取即可检查远端镜像、跨仓库/格式无缝迁移镜像
containerd	核心底层运行时	稳定高效、符合 CRI 规范、是 Docker 的基石之一
安全容器	强隔离沙箱运行	利用轻量级虚拟机 (Kata) 或用户态内核 (gVisor) 防止越狱，极其安全
Wasm	新型工作负载	体积极小、冷启动超快且具备跨平台及高度特征化沙盒能力的后端架构新方向

Podman vs Docker

两者的主要区别：

Built with mdPress

对比项	Docker	Podman
守护进程	需要 dockerd	无需守护进程
权限	默认需要 root	原生支持 Rootless
CLI 兼容	-	与 Docker 命令兼容
Pod 支持	不支持	原生支持 Pod 概念
Compose	docker compose	podman-compose 或兼容模式

延伸阅读

- 底层实现：容器技术的内核基础
- 安全：容器安全实践

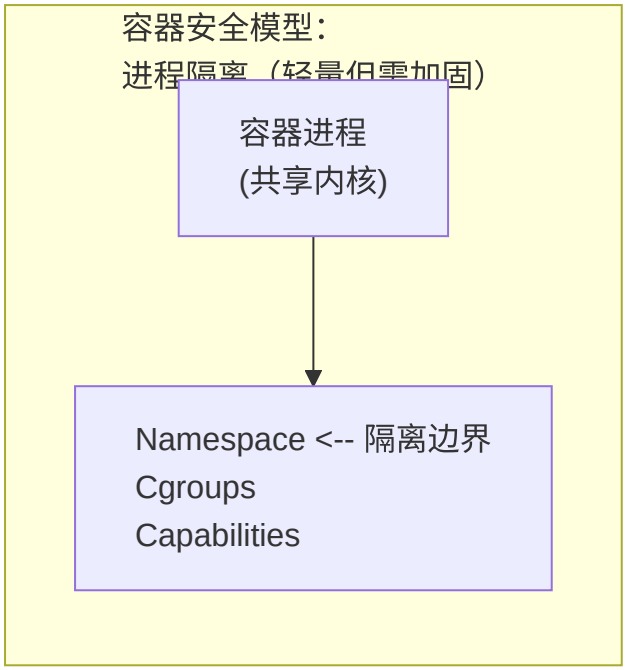
> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

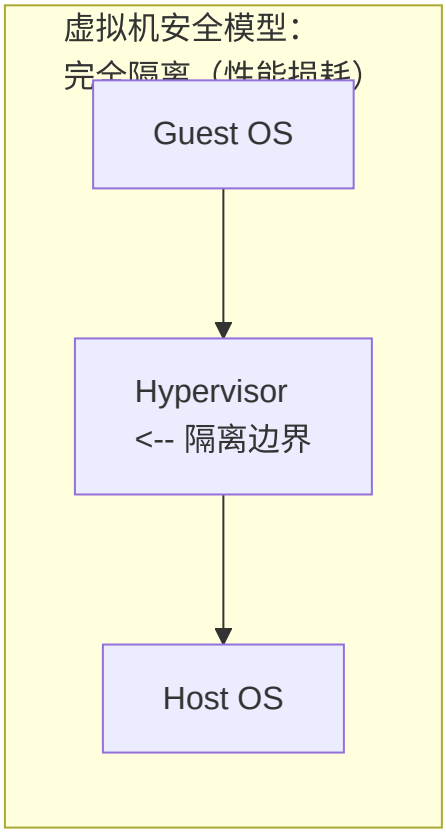
第十八章 安全

容器安全是生产环境部署的核心考量。本章介绍 Docker 的安全机制和最佳实践。

容器安全的本质

> **核心问题**：容器共享宿主机内核，隔离性弱于虚拟机。如何在便利性和安全性之间取得平衡？





本章内容

本章涵盖 Docker 安全的多个层面，从内核隔离机制到运行时防护和供应链安全。

- 内核命名空间
  - 命名空间的安全意义、User Namespace 与提权防护。
- 控制组
  - 通过 Cgroups 限制容器资源使用，防止资源耗尽攻击。
- 服务端防护
  - Docker 守护进程的安全配置与网络访问控制。
- 内核能力机制
  - Linux Capabilities 的细粒度权限控制。
- 其它安全特性
  - 镜像安全（漏洞扫描、签名验证）、运行时安全（非 root 运行、只读文件系统、Seccomp、AppArmor）、Dockerfile 安全实践、软件供应链安全（SBOM、SLSA）。
- 镜像安全
  - 容器镜像的安全扫描、漏洞检测与签名验证。

安全扫描清单

部署前检查：

Built with mdPress

检查项	命令/方法
漏洞扫描	`docker scout cves` 或 `trivy`
非 root 运行	检查 Dockerfile 中的 `USER`
资源限制	检查 `-m`, `--cpus` 参数
只读文件系统	检查 `--read-only`
无特权模式	确认没有 `--privileged`
最小能力	检查 `--cap-drop=all`
网络隔离	检查网络配置
敏感信息	确认无硬编码密码

Built with mdPress

18.1 内核命名空间

命名空间 (Namespace) 是 Linux 容器隔离的基础，它确保了容器内的进程无法直接干扰主机或其他容器。虽然在本书第 12 章中我们已经从底层实现的角度介绍了 Namespace，但在本节中，我们将重点探讨其 **安全意义** 及相关配置。

18.1.1 隔离的安全本质

Docker 守护进程在启动容器时，会在后台为容器创建一套独立的命名空间。命名空间提供了最基础也是最直接的隔离：

- **PID Namespace**：防止容器内的进程查看或终止宿主机或其他容器的进程。恶意攻击者即使在容器内获得了 root 权限，也无法通过 `kill` 命令影响宿主机上的关键服务。
- **NET Namespace**：每个容器都有自己独立的网络栈。如果没有显式地进行端口映射或将容器连接到同一网络，容器之间无法网络互通，从而限制了横向移动的能力。
- **MNT Namespace**：为容器提供独立的文件系统视图。这可以防止容器不经意或恶意地修改宿主机的重要系统文件（如 `/etc/passwd`）。

18.1.2 命名空间不是绝对安全的护城河

尽管命名空间提供了很好的隔离性，但我们必须认识到：**所有的容器依然共享同一个宿主机的 Linux 内核**。

这意味着，一旦宿主机的内核存在提权漏洞（如著名的 Dirty COW 漏洞），攻击者有可能通过突破 Namespace 的限制，直接在内核层面执行恶意代码，从而实现“容器逃逸”。

> ⚠️ **Warning**
>
> 为了缓解内核漏洞带来的威胁，生产环境务必保持宿主机 Linux 内核的及时修补与更新，或者借助诸如 gVisor、Kata Containers 等提供了独立内核的安全容器技术。同时，需要及时修补容器运行时（如 runC）的漏洞。2025 年 11 月披露的一系列 runC 容器逃逸漏洞（CVE-2025-31133、CVE-2025-52565、CVE-2025-52881）就表明，即使内核保持更新，运行时层的缺陷仍然可能导致容器隔离被突破。

通过命名空间，Docker 也能限制进程从外部环境获取信息。 例如，由于进程环境被隔离，进程在内部其实是无法感知到外部宿主机的存在的。它既不能获取其他容器的进程列表，也无法通过网络与其他系统进行交互（除非经过配置）。

18.1.3 用户命名空间与提权防护

在所有的 Namespace 中，**User Namespace** 对安全的影响尤为关键。

在默认情况下，容器内的 `root` 用户（UID=0）就是宿主机上的 `root` 用户。如果攻击者设法突破了容器的其他隔离机制获取了宿主机的访问权限，他将拥有宿主机的最高系统权限。

通过启用 **User Namespace Remapping (用户命名空间映射)**，我们可以将容器内的 `root` 用户映射到宿主机上的一个无特权普通用户。

如何配置 User Namespace

要在 Docker 服务端启用这一特性，需要修改 Docker 的配置文件 `/etc/docker/daemon.json`。

1. **设置映射策略**

编辑配置文件，添加 `userns-remap` 配置项：

```
{
  "userns-remap": "default"
}
```

使用 `default` 值时，Docker 会自动在宿主机上创建一个名为 `dockremap` 的用户和用户组。

2. **验证子 UID 和子 GID 分配**

Docker 会通过 `/etc/subuid` 和 `/etc/subgid` 文件为 `dockremap` 分配一个高位的 UID 范围：

```
$ cat /etc/subuid
dockremap:165536:65536
```

这意味着：容器内的 UID `0`（root 用户）在宿主机上实际被映射成了 UID `165536`。如果是容器内的 UID `1`，对应宿主机的 `165537`，以此类推。

3. **重启 Docker 守护进程**

```
$ sudo systemctl restart docker
```

验证映射效果

我们可以运行一个简单的容器并执行 `sleep` 命令，同时在宿主机上观察进程的所有者：

```
## 在容器内以 root 身份运行
$ docker run -d --name userns_test alpine sleep 3600

## 在宿主机上查看该 sleep 进程
$ ps aux | grep sleep
165536    12345  0.0  0.0   1568     4 ?        Ss   14:20   0:00 sleep 3600
```

你会发现，尽管在容器内该进程是由 root 启动的，但在宿主机上，它的属主是 `165536`（一个完全没有特权的用户）。

> 💡 **Tip**
>
> 启用 User Namespace 会对容器共享宿主机数据卷（Bind Mount）产生权限影响。你需要确保映射后的高位 UID 对宿主机上的挂载目录具有合适的读写权限。

18.1.4 总结

内核命名空间从 Linux 2.6.15 版本 (2006 年) 被引入，十余年间，这些机制的可靠性在诸多大型生产系统中被实践验证。通过合理利用命名空间（尤其是 User Namespace），可以极大地收窄攻击面，显著提升容器部署的安全性。

18.2 控制组

控制组 (Cgroups) 是 Linux 容器机制的另外一个关键组件。如果说命名空间 (Namespace) 决定了容器能 **看到** 什么，那么控制组就决定了容器能 **使用** 多少资源。

在安全领域中，资源的不可用性本身就是一种安全威胁。控制组负责实现资源的审计和限制，这对于抵御资源耗尽型攻击（如拒绝服务攻击 DoS）至关重要。

18.2.1 为什么资源限制关乎安全?

默认情况下，Docker 容器对系统资源的使用是没有限制的：一个容器理论上可以使用宿主机所有的 CPU 计算能力、吃光所有的内存、耗尽所有的系统 PID。

想象一下以下场景：

- 一个恶意用户向你暴露在公网的应用发起海量并发请求。
- 应用程序逻辑中存在内存泄漏漏洞。
- 黑客在入侵容器后，在里面运行了挖矿木马程序。

如果没有 Cgroups 的限制，某个容器内的异常行为（或恶意攻击）将会榨干宿主机的资源，导致宿主机上其他健康的容器甚至 Docker 守护进程自身因为 OOM（Out Of Memory）崩溃或 CPU 饥饿而停止响应。

18.2.2 核心资源限制实战

为了确保多租户平台（如公有或私有的 PaaS 平台）的稳定性，或者在生产环境防止服务级联故障，我们要养成在启动容器时 **显式声明资源上限** 的习惯。

1. 内存限制

限制内存可以防止应用程序因内存泄漏或恶意载荷导致宿主机 OOM。

关键参数：

- `-m, --memory=""`：硬限制，容器可使用的最大内存量。
- `--memory-swap=""`：限制容器可使用的内存与 Swap 总量。

实战示例：

限制容器最多只能使用 512MB 内存，并且禁用 Swap（将 memory 和 memory-swap 设置成一样的值即可）：

```
$ docker run -d \
  --name web_app \
  --memory="512m" \
  --memory-swap="512m" \
  nginx:alpine
```

如果该容器内的应用尝试分配超过 512MB 的内存，该进程将会被内核的 OOM Killer 杀掉，但绝不会波及到宿主机的其他部分。

2. CPU 限制

限制 CPU 可以防止个别计算密集型的容器垄断 CPU 时间片，保证系统的调度公平性。

关键参数：

- `--cpus=<value>`：指定容器可以使用的 CPU 核心数量（可以是小数）。
- `-c, --cpu-shares=0`：软限制，设置容器使用 CPU 的相对权重（默认是 1024）。

实战示例：

限制容器最多使用 1.5 个 CPU 核心的算力：

```
$ docker run -d \
  --name worker_app \
  --cpus="1.5" \
  busybox \
  md5sum /dev/urandom
```

即使上面的命令是一个死循环的哈希计算进程，容器也永远无法吃满双核 CPU 系统的全部算力。

3. 进程数限制

进程炸弹（Fork Bomb）是一种典型的拒绝服务攻击方式，它通过不断 `fork()` 新进程来耗尽系统的进程表条目，导致系统无法创建任何新任务。

关键参数：

- `--pids-limit=<number>`：限制容器内允许创建的最大进程数。

实战示例：

一个常规的 Web 服务进程数通常在几十到上百之间。我们可以设定一个合理的上限来防范 Fork 炸弹：

```
$ docker run -d \
  --name app_service \
  --pids-limit=100 \
  python:alpine python app.py
```

当容器内的进程总数达到 100 时，任何尝试派生新进程的操作都会失败并返回 `Resource temporarily unavailable`，从而挫败相关的攻击行为。

18.2.3 最佳实践建议

在生产环境中，不仅要在单机使用 Docker 命令时设置这些参数，更应当在集群编排工具中将资源配额制度化。

例如，在 Kubernetes 中，强烈建议为每个 Pod 设置 `requests` 和 `limits`：

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

通过 Cgroups 的资源边界控制，你可以从根本上切断一条导致整个系统雪崩的脆弱链路。这也进一步使得 Docker 以及容器技术成为了现代高可用服务的基础设施首选。

18.3 服务端防护

Docker 守护进程（`dockerd`）是容器生命周期的核心驱动力。默认情况下，Docker 服务的运行需要极高的系统特权（root 权限），因此其安全性关系到整台宿主机的生死存亡。

如果 Docker 守护进程的访问控制没有做好，恶意攻击者可以通过 Docker API 轻易地启动一个特权容器，并将宿主机的根目录（`/`）挂载到容器中，从而完全接管服务器。

为了加强对服务端的保护，我们需要从访问控制、通信加密和权限最小化三个维度进行加固。

18.3.1 限制 API 访问

Docker 客户端（`docker` 命令）通过 REST API 与守护进程进行通信。

在早期版本中，Docker 有时会绑定在 `127.0.0.1` 的 TCP 套接字上，但这容易遭遇跨站脚本（跨协议）攻击。现在的发行版默认使用 Unix Domain Socket（`/var/run/docker.sock`）并依赖文件系统的权限控制。

原则 1：决不可将无认证的 TCP 端口暴露在公网

这是最常见的 Docker 被入侵抓去挖矿的原因！绝不能在没有任何安全控制的情况下强行开启 `-H tcp://0.0.0.0:2375`。

如果业务确实需要远程访问 Docker 守护进程，**必须启用 TLS 认证机制**，让客户端和服务端互相进行证书校验。

开启 TLS 认证双向加密

利用安全机制，确保只有经过授权的主机网络，并在强证书保护下进行通信：

1. 首先使用 `openssl` 或基于本地 CA 工具生成一套客户端与服务器的证书。
2. 配置 Docker 守护进程（通常是 `daemon.json` 或 `dockerd` 启动参数），指定证书路径：

```
dockerd \
  --tlsverify \
  --tlscacert=ca.pem \
  --tlscert=server-cert.pem \
  --tlskey=server-key.pem \
  -H=0.0.0.0:2376
```

3. 客户端想要连接时，也必须出示客户端证书。

> 💡 **Tip**
>
> 配置 TLS 生成证书的完整步骤可以查阅 Docker 官方 TLS 文档。在现代编排系统（如 Kubernetes）中，通常会有自动化方案管理这些凭据。

18.3.2 保护本地 Socket 访问

哪怕不开启网络端口，本地的 `/var/run/docker.sock` 也需要谨慎对待。

任何被授予该 Socket 读写权限的用户（通常被加入 `docker` 用户组），等同于拥有了对宿主机的零成本提权途径，即“无需密码的免密 `sudo` 权限”。

> 🔴 **Caution**
>
> 永远不要将不可信的普通用户加入到 `docker` 用户组中。同样，在容器编排时尽量避免将宿主机的 `/var/run/docker.sock` 直接映射给普通容器使用，这种模式被称为 Docker-in-Docker (DinD) 或 Docker-out-of-Docker (DooD)，存在极高的越权风险。

18.3.3 Rootless 模式：非特权运行

为了从根本上解决“拥有 Docker socket 就是 root”的问题，Docker 自 19.03 版本（2019 年）起提供了 **Rootless 模式**。

Rootless 模式允许在完全局限于非 `root` 用户的环境中运行 Docker 守护进程（`dockerd`）和容器。该模式利用了现代 Linux 内核的 User Namespace 技术和非特权网络命名空间实现。

配置运行 Rootless Docker

要在非 root 环境中运行 Docker，需要先满足宿主机条件：安装 `newuidmap` / `newgidmap`，并在 `/etc/subuid` 与 `/etc/subgid` 中为该用户分配足够的 subordinate UID/GID。若系统级 Docker daemon 仍在运行，命令行也仍可能连到 rootful socket，因此应明确切换 Docker context 或 `DOCKER_HOST`。

1. 安装必要的依赖（通常是 `uidmap` 工具包以便系统支持 `newuidmap` 和 `newgidmap`）：

```
$ sudo apt-get install uidmap
```

2. 切换到一个没有任何 `sudo` 权限的普通用户（假设用户名为 `testuser`）：

```
$ su - testuser
```

3. 下载并检查 Docker 官方提供的 Rootless 安装脚本，确认来源和内容后再执行：

```
$ curl -fsSL https://get.docker.com/rootless -o install-rootless-docker.sh
$ less install-rootless-docker.sh
$ sh install-rootless-docker.sh
```

4. 配置环境变量指向新创建的私有 socket：

```
$ export DOCKER_HOST=unix:///run/user/1000/docker.sock
$ docker version
```

安装并暴露相应的配置后，该用户的环境将能独立启动属于他自己的 Docker Daemon。即使由于某些未知 0-Day 漏洞使得攻击者突破了容器，他们也只会受限于 `testuser` 这个非特权用户所在的有限系统环境内。

> ℹ️ **Note**
>
> Rootless 模式不是无条件替代 rootful Docker。端口绑定、网络、cgroup、存储驱动和系统服务自启动能力都受发行版、内核与 systemd 用户服务配置影响；生产环境应先验证具体工作负载，并用 `loginctl enable-linger <user>` 等方式显式配置开机自启动。

18.3.4 授权插件（Authorization Plugin）与访问策略

在企业环境中，对 Docker 守护进程的访问控制往往不仅限于文件系统权限，还需要更细粒度的授权策略。**Authorization Plugin** 机制允许在 API 层级对请求进行拦截和审批。

常见的授权插件包括：

- **OPA/Conftest**：开放策略引擎，支持声明式策略定义。
- **Prisma Cloud（Twistlock）**：商业容器安全平台。
- 自定义脚本：根据请求内容（镜像、命令、用户等）做出允许/拒绝决定。

配置 Authorization Plugin

在 `daemon.json` 中指定授权插件：

```
{
  "authorization-plugins": ["myauthorizer"]
}
```

此后，Docker 守护进程会在执行任何 API 请求前，将请求转发给授权插件进行审批。

> 🔴 **Caution**
>
> 重要的安全事项：在配置 Authorization Plugin 时，务必确保插件本身的可靠性和及时更新。历史上已发现多个 AuthZ 验证绕过漏洞，可能导致攻击者绕过授权检查并获得宿主机访问权限。建议：
>
> - 定期审计授权插件的日志，检查是否有可疑的请求被错误允许。
> - 使用来自可信来源的授权插件，并保持其版本最新。
> - 将授权检查结果与其他安全措施（如 TLS 认证、Rootless 模式）结合使用，构建纵深防御。

18.3.5 结语

保障 Docker 服务端的安全主要是做减法：关闭不必要的网络监听点，严管 Socket 访问权限。在基础系统、网络和存储约束都验证通过后，Rootless 模式是一项值得优先评估的安全加固选择。

18.4 内核能力机制

传统 Linux 的权限模型非常粗放：进程分为“特权进程”（以 root 用户 `UID 0` 运行）和“非特权进程”（其他 UID 运行）。这带来了一个致命问题——只要一个后台服务需要一个微小的特权（例如绑定低于 1024 的端口），就必须被赋予所有的 root 权限。一旦该服务被攻陷，系统便会全面沦陷。

为了解决这一问题，Linux 引入了 **能力机制（Capabilities）**。它将传统的全能 root 权限划分为几十个细粒度的操作能力。

18.4.1 容器内置的 Capability 白名单

在默认情况下，即便一个容器是在以 `root` 用户运行，Docker 也只为其内核授予了所有可用能力中的 **一小部分“白名单”能力**。

常见的 Linux Capabilities 包含：

- `CAP_CHOWN`: 修改文件所有者。
- `CAP_NET_BIND_SERVICE`: 绑定特权端口（即 1024 以下的端口）。
- `CAP_NET_ADMIN`: 网络管理的最高权限（例如调整路由配置，设置防火墙规则等）。
- `CAP_SYS_ADMIN`: 被誉为“Linux 内核的特权网管”，允许各种高危操作（挂载磁盘、访问敏感设备等）。

为了在 **“最小特权原则”** 的指导下加强安全，Docker 默认 **移除了** 大量可能导致容器大范围破坏宿主机的能力，例如：

- 完全禁止了任何通过 `CAP_SYS_ADMIN` 进行的核心挂载或设备操作。
- 禁止修改内核模块。
- 禁止直接访问硬件套接字。

这种“非完整”的 root 用户能保证大部分应用在拥有其所需权限的同时，把恶意行为对系统的影响降到最低。

18.4.2 实战：添加与剥夺能力

当启动一个 Docker 容器时，我们可以利用 `--cap-add`（增加特权）和 `--cap-drop`（剥夺特权）两个参数精细地控制进程环境。

实战场景一：构建极限安全的 Web 靶机

假设你正在提供一个公共的 Web 容器。你不希望里面的任何恶意脚本修改进程权限或者创建设备节点，你可以通过命令先移除 **所有** 默认能力，然后再按需授权该守护进程一个仅仅能绑端口的能力。

```
$ docker run -d \
  --name max_secure_web \
  --cap-drop ALL \
  --cap-add NET_BIND_SERVICE \
  nginx:alpine
```

这里的 `--cap-drop ALL` 是实现“特权最小化”的最强杀手锏。此时，即便某黑客利用 0-Day 手段拿到了 Web 服务的容器 root Shell，当他试图改变任何不属于他自己的进程配置或者所有权时，系统都会报错拒绝访问。

实战场景二：需要捕获网络数据包的网络实验

假设容器内的主程序是一个网络嗅探器（如 tshark 或 tcpdump），这显然不在 Docker 提供的默认白名单之内，因为该程序试图直接操纵底层网卡流量，会触发 Permission Denied。

此时，我们需要给它适当补发缺失的部分核心管理能力：

```
$ docker run -it --rm \
  --name network_sniffer \
  --cap-add NET_ADMIN \
  --cap-add NET_RAW \
  tshark-image /bin/bash
```

我们只授予了所需的网络管理控制（NET_ADMIN）和侦听底层套接字的权限（NET_RAW），而免去了赋予整个容器终极杀器 `--privileged` 参数。

> ⚠️ **Warning**
>
> 大量开发人员遇到了“权限遭到拒绝”的错误时，往往习惯性图省事添加 `--privileged` 这个核选项。但这将把 **宿主机上一切特权和所有访问设备完全投射给容器内的根用户**，其危险性等价于根本没有做隔离！请务必查明进程出错的实际原因，精准施加必要的隔离 `CAP_*` 能力。

18.4.3 总结

利用能力机制（Capabilities）是进行精细化系统级访问控制的关键一环。遵循“**白名单剥夺一切不必要权利（--cap-drop ALL）**”的极端配置并不过分，这将使得即便程序本身漏洞百出，攻击面也被死死压缩在一个几乎毫无后续伸展潜力的受限维度中。

18.5 其它安全特性

除了上述的命名空间、控制组以及能力机制，Linux 内核与云原生生态还提供了大量安全增强功能，它们共同筑成了一道防御纵深的“马奇诺防线”。本节主要介绍强制访问控制、系统调用拦截以及自动化的容器漏洞扫描技术。

18.5.1 系统调用过滤

`Seccomp`（Secure Computing mode）是 Linux 内核的一个安全机制，用于限制进程能够发起的系统调用数量。

一个普通的 Linux 内核提供了 300 多个系统调用，而一个正常运行的容器化应用（例如 Nginx 服务）通常只会用到几十个调用，这就给攻击者留下了大量的闲置入口点来进行内核层的缓冲区溢出攻击。

Docker 默认启用了 Seccomp，并使用预置的 默认配置文件 作为 allowlist：默认拒绝未显式允许的系统调用，并额外允许常见应用所需的调用。Docker 官方文档将其描述为默认禁用约 44 个系统调用（内核与 Docker 版本不同会有差异），例如与内核模块、系统重启或特权命名空间操作相关的调用。

如果你对应用的系统调用特征了如指掌，你可以为容器定制专属规则。

实战：禁用 chmod 系统调用过滤

首先，编写一个 `no-chmod.json` 的策略文件：

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
            "name": "chmod",
            "action": "SCMP_ACT_ERRNO"
        }
    ]
}
```

在启动时告诉 Docker 载入这套过滤配置：

```
$ docker run --rm -it \
  --security-opt seccomp=no-chmod.json \
  alpine sh

/ # chmod 777 /etc/passwd
chmod: /etc/passwd: Operation not permitted
```

应用只要被劫持进行越界尝试，其操作系统层命令便会立刻吃瘪。

18.5.2 强制访问控制：AppArmor / SELinux

传统的 Linux 模型遵循 DAC（自主访问控制），这意味着如果一个文件被赋予了全员读写权限（777），普通隔离下任何人便都能修改。但 **MAC（强制访问控制）** 技术，诸如 AppArmor (常用于 Ubuntu/Debian) 或 SELinux (常用于 CentOS/RHEL)，可以制定比“文件所有权”更宏观且优先的策略控制模块。

在开启了上述机制的机器上：

- **AppArmor**: 在启用 AppArmor 的系统上，Docker 默认为容器加载 `docker-default` profile；如果需要自定义策略，先用 `apparmor_parser` 加载 profile，再通过 `--security-opt apparmor=<profile>` 指定。
- **SELinux**: 在启用 SELinux 集成的系统上，容器与挂载目录需要正确的 SELinux label。绑定挂载时常用 `:z` 表示多个容器共享，`:Z` 表示该挂载只给单个容器使用；不要对 `/home`、`/usr` 等系统目录随意使用 `:Z`，否则可能破坏宿主机标签。

如果想为某些受信任应用施加特定的外部强化文件策略，可以通过如下方法指派规则表：

```
$ docker run --rm -it \
  --security-opt apparmor=custom-nginx-profile \
  nginx

$ docker run --rm -it \
  -v "$PWD/html":/usr/share/nginx/html:Z \
  nginx
```

18.5.3 容器镜像漏洞静态扫描

现代防护的防御已经不仅仅在运行阶段，而向“左”延伸至了构建与分发时期控制。很多安全隐患并不是用户代码中的直接逻辑异常，而是打包环境或者引入库的基础 APT 安装层面潜伏了开源界众所周知的历史漏洞。

使用 Trivy 识别风险

Trivy 是由 Aqua Security 发行的一款针对容器技术的快速镜像漏洞扫描利器。在分发应用前通过它的扫描可以规避绝大多数风险。

```
## 如果使用本地命令行扫描容器镜像
$ trivy image alpine:3.20

2024-03-01T10:05:07.124Z        INFO    Number of language-specific files: 1
2024-03-01T10:05:07.124Z        INFO    Detecting vulnerabilities...

alpine:3.20 (alpine 3.20.0)
===========================
Total: 2 (UNKNOWN: 0, LOW: 0, MEDIUM: 1, HIGH: 1, CRITICAL: 0)

+---------+------------------+----------+-------------------+--------------+-----------------------
---------------+
| LIBRARY | VULNERABILITY ID | SEVERITY | INSTALLED VERSION | FIXED VERSION | TITLE
|
+---------+------------------+----------+-------------------+--------------+-----------------------
---------------+
| busybox | CVE-2022-28391   | HIGH     | 1.30.1-r3         | 1.30.1-r4     | busybox: out-of-bounds
read in...     |
...
```

只要确保所有上传给私有或公共仓库分发服务的产物先被引入至 CI/CD 流水线，如果出现 `HIGH` 及 `CRITICAL` 严重的报错记录强行阻拦部署，这本身便是构建环节极其有力的自动化安全大门保障。除 Trivy 外，最新的 Docker 版本也已内置支持官方扫描利刃 **Docker Scout**。

18.5.4 容器核心层基石结语

到这里，Docker 为保障宿主和容器界限安全的几个护城河：从 **资源剥离限制**(`Cgroups`) 到 **进程/网络/身份蒙蔽**(`Namespace`)、到 **特权能力回收**(`Capabilities`) 再到 **内核强制策略拦截管制**(`Seccomp/AppArmor`) 已悉数交代完毕。虽然绝没有“100% 免疫网络穿刺的防线”，只要开发者牢记 **权限最小化原则**，容器的堡垒就可以做到令攻击者望洋兴叹。

18.6 容器镜像安全扫描与供应链安全

在 DevOps 流程中，容器镜像安全已经成为不容忽视的关键环节。从开发、构建、存储到部署，镜像的整个生命周期都需要安全防护。本节深入讨论镜像漏洞扫描、软件物料清单（SBOM）、镜像签名验证等供应链安全实践。

18.6.1 容器镜像漏洞扫描工具对比

Trivy - 轻量级通用扫描器

Trivy 是由 Aqua Security 开发的开源漏洞扫描器，以其轻量级、快速、准确而闻名，已成为业界标准。

优点：

- 零依赖，单个二进制文件
- 扫描速度快（秒级）
- 支持镜像、文件系统、Git 仓库多种扫描源
- 数据库每日自动更新
- 支持多种输出格式（JSON、表格、SBOM 等）

安装与基本使用：

```
# 安装 Trivy
curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh -o install-tr
ivy.sh
less install-trivy.sh  # 先审阅脚本内容
sudo sh install-trivy.sh -b /usr/local/bin

# 扫描本地镜像
trivy image nginx:latest

# 生成 JSON 格式报告
trivy image -f json -o report.json nginx:latest

# 扫描文件系统
trivy fs /path/to/project

# 扫描 Git 仓库
trivy repo https://github.com/aquasecurity/trivy
```

在 CI/CD 中集成：

```
# 设置严重程度过滤
trivy image --severity HIGH,CRITICAL \
  --exit-code 1 \
  myregistry.com/myapp:v1.0.0
```

Grype - 支持多种软件包的扫描器

Grype 由 Anchore 开发，支持更广泛的软件包管理器和语言。

优点：

- 支持 Java、Python、Go、Ruby、JavaScript 等多种语言的依赖检测
- 与 Syft（SBOM 生成器）配合效果好
- 可自定义漏洞数据库源
- 支持离线扫描模式

安装与使用：

```
# 安装 Grype
curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh -o install-grype.sh
less install-grype.sh
sudo sh install-grype.sh -b /usr/local/bin

# 扫描镜像
grype docker:nginx:latest

# 与 Syft 配合生成 SBOM
syft docker:nginx:latest -o json > sbom.json
grype sbom:sbom.json

# 扫描特定目录
grype dir:/path/to/app
```

Snyk - 完整的安全平台

Snyk 提供了商业级的安全扫描服务，特别适合企业环境。

特点：

- 支持开源漏洞、许可证扫描和修复建议
- 与多个 Git 平台深度集成（GitHub、GitLab、Bitbucket）
- 提供修复建议和自动化修复 PR
- 支持 Kubernetes 部署后安全监控

基本使用：

```
# 安装 Snyk CLI
npm install -g snyk

# 认证
snyk auth

# 扫描镜像
snyk container test docker-archive://image.tar

# 监控仓库
snyk monitor --docker
```

此外，Docker 官方提供的 **Docker Scout**（`docker scout` CLI）可以直接在 Docker Desktop 或命令行中对镜像进行漏洞扫描和依赖分析，适合日常开发流程中快速检查：

```
# 扫描本地镜像
$ docker scout cves myapp:latest

# 查看镜像的依赖和漏洞摘要
$ docker scout quickview myapp:latest
```

工具对比表：

特性	Docker Scout	Trivy	Grype	Snyk
Docker 集成	✓（原生）	需安装	需安装	需安装
零依赖	✗	✓	✗	✗
离线模式	✗	✓	✓	✗
许可证扫描	✓	✓	✓	✓
自动修复建议	✓	✗	✗	✓
开源免费	部分	✓	✓	部分
IDE 集成	✓	✓	✓	✓

18.6.2 SBOM（软件物料清单）生成与管理

SBOM（Software Bill of Materials）是一份详细列表，记录了软件中使用的所有组件、依赖库及其版本信息。SBOM 在供应链安全中至关重要，特别是在发现新的安全漏洞时，能快速定位受影响的应用。

Syft - SBOM 生成工具

Syft 是 Anchore 推出的专业 SBOM 生成工具。

安装：

```
curl -sSfL https://raw.githubusercontent.com/anchore/syft/main/install.sh -o install-syft.sh
less install-syft.sh
sudo sh install-syft.sh -b /usr/local/bin
```

生成 SBOM：

```
# 从镜像生成 SBOM（多种格式）
syft docker:nginx:latest -o json > sbom.json
syft docker:nginx:latest -o spdx > sbom.spdx
syft docker:nginx:latest -o cyclonedx > sbom.xml

# 从本地文件系统生成
syft dir:/path/to/app -o json > sbom.json

# 从 OCI 镜像档案生成
syft oci-archive:image.tar -o json > sbom.json
```

CycloneDX 与 SPDX 格式

两种主流的 SBOM 格式：

CycloneDX 格式示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<bom xmlns="http://cyclonedx.org/schema/bom/1.4" version="1">
  <components>
    <component type="library">
      <name>openssl</name>
      <version>1.1.1k</version>
      <purl>pkg:deb/debian/openssl@1.1.1k-1+deb11u5</purl>
    </component>
    <component type="library">
      <name>curl</name>
      <version>7.74.0-1.3+deb11u1</version>
      <purl>pkg:deb/debian/curl@7.74.0-1.3+deb11u1</purl>
    </component>
  </components>
</bom>
```

SPDX 格式示例：

```
{
  "SPDXID": "SPDXRef-DOCUMENT",
  "spdxVersion": "SPDX-2.2",
  "creationInfo": {
    "created": "2024-03-01T12:00:00Z",
    "creators": ["Tool: syft"]
  },
  "packages": [
    {
      "SPDXID": "SPDXRef-Package-openssl",
      "name": "openssl",
      "versionInfo": "1.1.1k",
      "downloadLocation": "NOASSERTION"
    }
  ]
}
```

SBOM 的应用场景

漏洞关联：

当新的 CVE 被发现时，可快速查询受影响的应用：

```
# 使用 Grype 针对 SBOM 进行漏洞扫描
grype sbom:sbom.json --add-cpes-if-none
```

合规性报告：

将 SBOM 保存为构建产物，用于审计和合规性检查。

依赖升级决策：

通过分析 SBOM 中的依赖版本，制定安全升级计划。

18.6.3 镜像签名与验证

镜像签名确保镜像的来源可信且未被篡改。两种主流方案是 Cosign 和 Notary。

Cosign - 现代签名解决方案

Cosign 是 Sigstore 项目的核心工具，支持无密钥签名，适合现代 CI/CD 流程。

安装：

```
wget https://github.com/sigstore/cosign/releases/latest/download/cosign-linux-amd64
chmod +x cosign-linux-amd64
sudo mv cosign-linux-amd64 /usr/local/bin/cosign
```

生成密钥对（传统方式）：

```
cosign generate-key-pair

# 生成 cosign.key 和 cosign.pub
```

签名镜像：

```
# 先推送镜像，再使用不可变 digest 签名
IMAGE_DIGEST="myregistry.com/myapp@sha256:<digest>"
cosign sign --key cosign.key "$IMAGE_DIGEST"

# 系统会提示输入私钥密码
```

验证签名：

```
# 使用公钥验证
cosign verify --key cosign.pub "$IMAGE_DIGEST"

# 输出结果示例
# Verification successful!
# {
#   "critical": {
#     "identity": {...},
#     "image": {...},
#     "type": "cosign container image signature"
#   },
#   "optional": {...}
# }
```

Keyless 签名（推荐用于 CI/CD）：

```
# 在 GitHub Actions 等 CI 中无需存储密钥
cosign sign --yes "$IMAGE_DIGEST"

# 验证时自动使用 OIDC 令牌验证身份
cosign verify "$IMAGE_DIGEST" \
  --certificate-identity https://github.com/myorg/myrepo/.github/workflows/build.yml@refs/heads/main
\
  --certificate-oidc-issuer https://token.actions.githubusercontent.com
```

Docker Content Trust 与 Notary

> **注意：DCT 退役时间线**
>
> Docker 已宣布退役 Content Trust，Docker Official Images 的部分 DCT 签名证书自 2025 年 8 月起已陆续过期；完整退役节奏应以 Docker 官方公告和你所使用的 registry 服务商文档为准。不要把云厂商特定日期直接当作 Docker 全局时间线。
>
> 新项目应优先使用上文介绍的 **Cosign (Sigstore)** 或 registry 原生签名/证明能力；现有 DCT 用户应先盘点依赖、验证替代方案，再制定迁移计划。

Docker Content Trust 使用 Notary 实现镜像签名，是 Docker 官方的传统签名解决方案。

历史用法示例（不建议新项目采用）：

```
# 在环境中启用 DCT
export DOCKER_CONTENT_TRUST=1

# 此后所有 docker push/pull 都需要签名
docker push myregistry.com/myapp:v1.0.0

# 如果镜像未签名，操作会被拒绝

# 禁用 DCT（仅用于特定操作）
docker push --disable-content-trust myregistry.com/myapp:v1.0.0
```

签名密钥管理：

```
# 首次推送时会提示创建 Delegation Key
# 密钥存储在 ~/.docker/trust/private/root_keys/ 和 ~/.docker/trust/private/tuf_keys/

# 查看 DCT/Notary 信任数据；RepoDigests 只是内容摘要，不等于签名验证
docker trust inspect --pretty myregistry.com/myapp:v1.0.0
```

18.6.4 供应链安全最佳实践

1. 基础镜像安全

```
# ❌ 不推荐：使用 latest 标签
FROM ubuntu:latest
RUN apt-get update && apt-get install -y curl

# ✓ 推荐：固定基础镜像版本和摘要
FROM ubuntu:22.04@sha256:a6d2b38300ce017add71440577d5b0a90460d0e6e0e14...（完整 64 位哈希）
RUN apt-get update && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*
```

2. 构建时扫描

在 CI/CD 构建阶段集成安全扫描，避免在 Dockerfile 里从分支 URL 下载并执行远程安装脚本：

```
docker run --rm \
  -v "$PWD:/work" \
  -w /work \
  aquasec/trivy:latest \
  fs . --exit-code 1 --severity HIGH,CRITICAL
```

3. 运行时镜像扫描策略

```
# 镜像构建完成后立即扫描
trivy image --severity HIGH,CRITICAL \
  --exit-code 1 \
  --timeout 30m \
  $IMAGE_NAME:$IMAGE_TAG

# 定期扫描已部署的镜像
trivy image --scanners vuln,misconfig registry:5000/myapp:latest
```

4. 镜像仓库安全配置

Harbor（私有镜像仓库）的安全扫描：

```
# harbor.yml 配置示例
trivy:
  enabled: true
  # 启用镜像扫描
  image_source: "Official"

# 默认扫描配置
scan_on_push: true  # 推送时自动扫描
scan_all: true      # 扫描仓库中的所有镜像
```

5. 政策执行

在 Kubernetes 环境中使用 Admission Webhook 强制镜像签名和扫描：

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: image-security-policy
webhooks:
- name: image-security.example.com
  clientConfig:
    service:
      name: image-security-webhook
      namespace: security
      path: "/validate"
  rules:
  - operations: ["CREATE", "UPDATE"]
    apiGroups: [""]
    apiVersions: ["v1"]
    resources: ["pods"]
  admissionReviewVersions: ["v1"]
  sideEffects: None
```

18.6.5 CI/CD 中集成安全扫描

GitHub Actions 工作流示例

```yaml
name: Build and Scan Image

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-scan:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
      security-events: write
      id-token: write

    steps:
    - name: Checkout code
      uses: actions/checkout@v6

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v4

    - name: Install Cosign
      if: github.event_name == 'push'
      uses: sigstore/cosign-installer@v3

    - name: Build Docker image for scan
      uses: docker/build-push-action@v7
      with:
        context: .
        push: false
        load: true
        tags: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:latest

    - name: Run Trivy vulnerability scan
      # 安全提醒：2026 年 3 月 19 日 Trivy GitHub Actions 遭受供应链攻击，
      # 76 个版本标签被劫持。务必使用不可变的 commit SHA 引用，而非可变标签。
      # 使用前请到 https://github.com/aquasecurity/trivy-action/releases 核实 SHA 对应正确版本。
      uses: aquasecurity/trivy-action@57a97c7e7821a5776cebc9bb87c984fa69cba8f1 # v0.35.0
      with:
        image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:latest
        format: 'sarif'
        output: 'trivy-results.sarif'
        severity: 'HIGH,CRITICAL'

    - name: Upload Trivy results to GitHub Security tab
      uses: github/codeql-action/upload-sarif@v3
      with:
```

```
        sarif_file: 'trivy-results.sarif'

    - name: Generate SBOM
      uses: anchore/sbom-action@v0
      with:
        image: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:latest
        format: cyclonedx-json
        output-file: sbom-cyclonedx.json

    - name: Upload SBOM
      uses: actions/upload-artifact@v4
      with:
        name: sbom
        path: sbom-cyclonedx.json

    - name: Login to Registry and Push
      if: github.event_name == 'push'
      uses: docker/login-action@v4
      with:
        registry: ${{ env.REGISTRY }}
        username: ${{ github.actor }}
        password: ${{ secrets.GITHUB_TOKEN }}

    - name: Push image
      if: github.event_name == 'push'
      id: build-push
      uses: docker/build-push-action@v7
      with:
        context: .
        push: true
        tags: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:latest

    - name: Sign image with Cosign
      if: github.event_name == 'push'
      run: |
        cosign sign --yes ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}@${{ steps.build-push.outputs.dig
est }}
```

GitLab CI 工作流示例

```
stages:
  - build
  - scan
  - push
  - sign

variables:
  REGISTRY: registry.gitlab.com
  IMAGE_NAME: $REGISTRY/$CI_PROJECT_PATH

build:
  stage: build
  image: docker:<version>-cli@sha256:<docker-cli-digest>
  services:
    - name: docker:<version>-dind@sha256:<docker-dind-digest>
  script:
    - docker build -t $IMAGE_NAME:$CI_COMMIT_SHA .
    - docker save $IMAGE_NAME:$CI_COMMIT_SHA > image.tar

scan:trivy:
  stage: scan
  image: aquasec/trivy:<version>@sha256:<trivy-digest>
  script:
    - trivy image --severity HIGH,CRITICAL --exit-code 1 docker-archive://image.tar
  allow_failure: false

scan:grype:
  stage: scan
  image: anchore/grype:<version>@sha256:<grype-digest>
  script:
    - grype docker-archive://image.tar

generate:sbom:
  stage: scan
  image: anchore/syft:<version>@sha256:<syft-digest>
  script:
    - syft docker-archive://image.tar -o cyclonedx > sbom.xml
  artifacts:
    reports:
      sbom: sbom.xml

push:
  stage: push
  image: docker:<version>-cli@sha256:<docker-cli-digest>
  services:
    - name: docker:<version>-dind@sha256:<docker-dind-digest>
  script:
    - docker load < image.tar
    - echo "$REGISTRY_PASSWORD" | docker login --username "$REGISTRY_USER" --password-stdin "$REGIST
RY"
    - docker push $IMAGE_NAME:$CI_COMMIT_SHA
    - DIGEST=$(docker buildx imagetools inspect "$IMAGE_NAME:$CI_COMMIT_SHA" --format '{{.Manifest.D
igest}}')
    - echo "$IMAGE_NAME@$DIGEST" > image-digest.txt
  artifacts:
    paths:
      - image-digest.txt
```

```
  only:
    - main

sign:
  stage: sign
  image: gcr.io/projectsigstore/cosign:<version>@sha256:<cosign-digest>
  script:
    - cosign sign --key $COSIGN_KEY "$(cat image-digest.txt)"
  only:
    - main
```

18.6.6 常见问题与最佳实践

Q: 扫描报告中有过时的 CVE，如何处理?

A: 某些 CVE 可能已经被修复但数据库未更新，可以：

- 手动验证安全补丁是否已应用
- 使用工具的忽略列表功能（如 Trivy 的 `.trivyignore`）
- 定期更新扫描工具和漏洞数据库

Q: 如何平衡镜像大小和安全性?

A:

- 使用多阶段构建减少最终镜像大小
- 使用精简基础镜像（Alpine、Distroless）
- 定期更新依赖而不是一味求小
- 优先安全性，体积次之

Q: 如何管理和轮换签名密钥?

A:

- 在密钥管理系统（如 HashiCorp Vault）中存储密钥
- 定期轮换密钥（建议每 90 天）
- 使用 Keyless 签名消除密钥管理复杂性
- 保留密钥轮换的审计日志

本章小结

Docker 的安全性依赖于多层隔离机制的协同工作，同时需要用户遵循最佳实践。本章涵盖的核心安全维度包括：

维度	关键措施
内核隔离	Namespace 隔离进程/网络/文件系统，Cgroups 限制资源使用
权限控制	非 root 运行、`--cap-drop ALL` 最小能力集、`--read-only` 只读根文件系统
镜像安全	使用可信基础镜像、定期扫描漏洞（Trivy / Snyk）、启用 Docker Content Trust 签名验证
运行时防护	Seccomp 系统调用过滤、AppArmor / SELinux 强制访问控制
网络隔离	自定义 bridge 网络隔离容器通信、限制容器对宿主机网络的访问

总体来看，Docker 容器还是十分安全的，特别是在容器内不使用 root 权限来运行进程的话。

另外，用户可以使用现有工具，比如 Apparmor，Seccomp，SELinux，GRSEC 来增强安全性；甚至自己在内核中实现更复杂的安全机制。

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

Built with mdPress

第十九章 容器监控与日志

在生产环境中，容器化应用部署完成后，实时掌握容器的运行状态以及应用日志非常重要。本章将以 Docker/Compose 的场景为主，介绍容器监控与日志管理的落地思路与最小实践闭环。

对于 Kubernetes 场景，可观测性链路与组件选择通常会有所不同（例如使用 Prometheus Operator、日志采集 DaemonSet 等）。本章会在关键点给出迁移提示，但不会展开为完整的 Kubernetes 教程。

我们将重点探讨以下内容：

- **容器监控**：以 Prometheus 为主，讲解如何采集和展示容器性能指标。
- **日志管理**：以 ELK (Elasticsearch, Logstash, Kibana) 套件为例，介绍集中式日志收集平台。

为了让读者能够在生产环境中真正用起来，本章会补齐以下“最小闭环”：

- 关键指标与日志的验证方法
- 常见故障排查路径
- 最小告警闭环 (Prometheus -> Alertmanager -> 接收端)
- 日志容量治理的最小实践

本章内容

- Prometheus 监控
  - 容器监控基础、指标采集与告警配置。
- ELK 日志管理
  - 集中式日志收集、存储与检索。
- 性能优化
  - 容器和应用性能优化实践。

Built with mdPress

19.1 Prometheus

Prometheus 和 Grafana 是目前最流行的开源监控组合，前者负责数据采集与存储，后者负责数据可视化。

Prometheus 是一个开源的系统监控和报警工具包。它受 Google Borgmon 的启发，由 SoundCloud 在 2012 年创建。

19.1.1 架构简介

Prometheus 的主要组件包括：

- **Prometheus Server**：核心组件，负责收集和存储时间序列数据。
- **Exporters**：负责向 Prometheus 暴露监控数据 (如 Node Exporter，cAdvisor)。
- **Alertmanager**：处理报警发送。
- **Pushgateway**：用于支持短生命周期的 Job 推送数据。

19.1.2 快速部署

我们可以使用 Docker Compose 快速部署一套 Prometheus + Grafana 监控环境。

本节示例使用了：

- `node-exporter`：采集宿主机指标 (CPU、内存、磁盘、网络等)。
- `cAdvisor`：采集容器指标 (容器 CPU/内存/网络 IO、文件系统等)。

在生产环境中，建议将 Prometheus 的数据目录做持久化，并显式配置数据保留周期。

1. 准备配置文件

创建 `prometheus.yml`：

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']

rule_files:
  - /etc/prometheus/rules.yml
```

2. 编写 Docker Compose 文件

创建 `compose.yaml` (或 `docker-compose.yml`)。下面示例使用的是相对较新的稳定版本。**镜像版本提示**：本示例中 Prometheus、Grafana、node-exporter、cAdvisor 的版本号仅为参考。生产环境部署前，请查阅以下官方资源确认最新推荐版本：

- Prometheus 官方文档
- Grafana 官方发布
- node-exporter 发布页
- cAdvisor 发布页

```yaml
services:
  prometheus:
    image: prom/prometheus:v3.11.2
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - ./rules.yml:/etc/prometheus/rules.yml
      - prometheus_data:/prometheus
    ports:
      - "9090:9090"
    command:
      - --config.file=/etc/prometheus/prometheus.yml
      - --storage.tsdb.path=/prometheus
      - --storage.tsdb.retention.time=15d
    networks:
      - monitoring

  grafana:
    image: grafana/grafana:13.0.1
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    networks:
      - monitoring
    depends_on:
      - prometheus

  node-exporter:
    image: prom/node-exporter:v1.11.1
    ports:
      - "9100:9100"
    networks:
      - monitoring

  cadvisor:
    image: ghcr.io/google/cadvisor:v0.56.2
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    networks:
      - monitoring

networks:
  monitoring:

volumes:
  prometheus_data:
```

3. 启动服务

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

启动后，访问以下地址：

- Prometheus: `http://localhost:9090`
- Grafana：`http://localhost:3000` (默认账号密码：admin/admin，首次登录后务必立即修改密码)

19.1.3 配置 Grafana 面板

1. 在 Grafana 中添加 Prometheus 数据源，URL 填写 `http://prometheus:9090`。
2. 导入现成的 Dashboard 模板，例如 Node Exporter Full (ID：1860) 和 Docker Container (ID：193)。

这样，你就拥有了一个直观的容器监控大屏。

19.1.4 生产要点与告警闭环

完成部署后，建议补齐以下生产要点。

指标采集的“最小闭环”

1. 在 Prometheus 页面打开 **Status -> Targets**，确认 `prometheus`、`node-exporter`、`cadvisor` 的 `State` 均为 `UP`。
2. 在 **Graph** 中尝试查询：
    - `up`
    - `rate(container_cpu_usage_seconds_total[5m])`
3. 在 Grafana Dashboard 中重点关注：
    - 宿主机 CPU/Load/内存/磁盘
    - 容器 CPU/内存使用率、容器重启次数

如果你发现“面板为空”，通常不是 Grafana 的问题，而是 Prometheus 没抓到数据或查询标签与 Dashboard 不匹配。

Built with mdPress

常见问题排查

- **Target down**：检查容器网络是否互通，端口是否暴露到同一网络，以及 exporter 是否在容器内正常监听。
- **cAdvisor 无数据或报错**：确认挂载了 Docker 目录与宿主机的 `/sys`、`/var/run` 等路径，并确保宿主机上 Docker 运行正常。
- **指标缺失**：确认你的 Docker/内核版本与 cAdvisor 兼容；对于 containerd 等运行时，采集方式会不同。

关键指标速查：节点/容器

在生产环境排障时，建议优先关注下面几类指标，并在 Grafana 面板中建立对应的常用视图。

- **节点 CPU 使用率**：`100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)`
- **节点内存使用率**：`(1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)) * 100`
- **节点磁盘空间使用率**：`(1 - (node_filesystem_avail_bytes{fstype!~"tmpfs|overlay"} / node_filesystem_size_bytes{fstype!~"tmpfs|overlay"})) * 100`
- **容器 CPU**：`sum by (namespace, pod, container) (rate(container_cpu_usage_seconds_total[5m]))`
- **容器内存**：`sum by (namespace, pod, container) (container_memory_working_set_bytes)`

说明：不同采集路径的 label 命名不同。Docker Compose 中独立部署的 cAdvisor 常见容器标签是 `name`；Kubernetes kubelet 的 `/metrics/cadvisor` 中，`container_cpu_usage_seconds_total` 等稳定指标使用 `container`、`pod`、`namespace`。如果查询为空，先直接查询 `container_cpu_usage_seconds_total` 样本并在 Prometheus 图形界面查看实际 label，不要假设存在 `container_name`。

Targets down 排错清单

当 **Status -> Targets** 出现 `DOWN` 时，建议按以下顺序排查：

Built with mdPress

1. **网络连通性**：Prometheus 容器是否能解析并访问目标 (同一 Docker network、DNS、端口)。
2. **端口/路径**：确认 exporter 监听端口与 Prometheus 配置一致；必要时在 Prometheus 容器内 `curl http://node-exporter:9100/metrics`。
3. **权限/挂载**：cAdvisor 需要访问宿主机 `/sys`、`/var/lib/docker` 等挂载路径，缺失会导致指标不全或报错。
4. **时间问题**：宿主机与容器时间偏差过大可能导致“数据看起来断档”，需要检查 NTP/时区配置。
5. **目标本身异常**：确认 exporter 容器是否在重启，查看 `docker logs`。

Alertmanager 告警建议

生产环境建议引入 Alertmanager 做告警聚合与路由，并在 Prometheus 中配置 `alerting` 与 `rule_files`。

为了保持“最小告警闭环”，建议至少覆盖两类告警：

- **采集链路告警**：例如 `up == 0`，用于发现 exporter 或网络故障。
- **资源风险告警**：例如节点磁盘空间不足，用于提前发现容量风险。

1. 准备告警规则文件

创建 `rules.yml`：

```
groups:
  - name: docker_practice
    rules:
      - alert: PrometheusTargetDown
        expr: up == 0
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "Prometheus 抓取目标不可达"
          description: "Job={{ $labels.job }}, Instance={{ $labels.instance }}"

      - alert: HostDiskSpaceLow
        expr: |
          (node_filesystem_avail_bytes{fstype!~"tmpfs|overlay"} / node_filesystem_size_bytes{fstype!~"tmpfs|overlay"}) < 0.10
        for: 10m
        labels:
          severity: critical
        annotations:
          summary: "磁盘可用空间不足"
          description: "Instance={{ $labels.instance }}, Mountpoint={{ $labels.mountpoint }}"
```

说明：这里的规则是“可用空间低于 10%”的阈值告警，并非“未来 24 小时写满”的预测。生产环境建议针对特定文件系统与挂载点做更精确的过滤。

2. 配置 Prometheus 加载规则并接入 Alertmanager

修改 `prometheus.yml`，增加：

```
rule_files:
  - /etc/prometheus/rules.yml

alerting:
  alertmanagers:
    - static_configs:
        - targets: ["alertmanager:9093"]
```

并在 Compose 中挂载规则文件。

3. 部署 Alertmanager

创建 `alertmanager.yml`：

```
route:
  receiver: default

receivers:
  - name: default
    webhook_configs:
      - url: http://example.com/webhook
```

再在 `compose.yaml` 增加服务：

```
  alertmanager:
    image: prom/alertmanager:v0.32.0
    volumes:
      - ./alertmanager.yml:/etc/alertmanager/alertmanager.yml
    ports:
      - "9093:9093"
    networks:
      - monitoring
```

生产环境中，建议将告警发送到可追踪的渠道 (如 IM 机器人、事件平台、工单系统)，并在告警中附带 Dashboard 链接与排障入口，避免告警成为噪声。

建议的文件清单

为了避免示例难以复现，建议在同一目录下准备以下文件：

- `compose.yaml`：Prometheus、Grafana、exporters、Alertmanager 的部署文件
- `prometheus.yml`：Prometheus 抓取配置与告警配置
- `rules.yml`：告警规则
- `alertmanager.yml`：告警路由与接收器配置

Built with mdPress

19.2 ELK 套件

ELK (Elasticsearch，Logstash，Kibana) 是目前业界最流行的开源日志解决方案。而在容器领域，由于 Fluentd 更加轻量级且对容器支持更好，EFK (Elasticsearch，Fluentd，Kibana) 组合也变得非常流行。

19.2.1 方案架构

我们将采用以下架构：

1. **Docker Container**：容器将日志输出到标准输出 (stdout/stderr)。
2. **Fluentd**：作为 Docker 的 Logging Driver 或运行为守护容器，收集容器日志。
3. **Elasticsearch**：存储从 Fluentd 接收到的日志数据。
4. **Kibana**：从 Elasticsearch 读取数据并进行可视化展示。

19.2.2 部署流程

我们将使用 Docker Compose 来一键部署整个日志堆栈。

1. 编写 Compose 文件

1. 编写 `compose.yaml` (或 `docker-compose.yml`) 配置如下。**版本提示**：本示例使用 Elasticsearch 9.x、Kibana 9.x、Fluentd 等组件的特定版本号，仅作为参考。生产环境部署前，请查阅 Elastic 官方文档 和 Fluentd 官方文档 确认最新版本与配置兼容性。

```yaml
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:9.4.0
    container_name: elasticsearch
    environment:
      - "discovery.type=single-node"
      - "xpack.security.enabled=false"
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ports:
      - "9200:9200"
    volumes:
      - es_data:/usr/share/elasticsearch/data
    networks:
      - logging

  kibana:
    image: docker.elastic.co/kibana/kibana:9.4.0
    container_name: kibana
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - "5601:5601"
    links:
      - elasticsearch
    networks:
      - logging

  fluentd:
    # elasticsearch8 插件通过 REST API 与 ES 9.x 兼容
    image: fluent/fluentd-kubernetes-daemonset:v1.17-debian-elasticsearch8-1
    container_name: fluentd
    environment:
      - "FLUENT_ELASTICSEARCH_HOST=elasticsearch"
      - "FLUENT_ELASTICSEARCH_PORT=9200"
      - "FLUENT_ELASTICSEARCH_SCHEME=http"
      - "FLUENT_UID=0"
    ports:
      - "24224:24224"
      - "24224:24224/udp"
    links:
      - elasticsearch
    volumes:
      - ./fluentd/conf:/fluentd/etc
    networks:
      - logging

volumes:
  es_data:

networks:
  logging:
```

2. 配置 Fluentd

创建 `fluentd/conf/fluent.conf`：

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

<match *.**>
  @type copy
  <store>
    @type elasticsearch
    host elasticsearch
    port 9200
    logstash_format true
    logstash_prefix docker
    logstash_dateformat %Y%m%d
    include_tag_key true
    tag_key @log_name
    flush_interval 1s
  </store>
  <store>
    @type stdout
  </store>
</match>
```

3. 配置应用容器使用 fluentd 驱动

启动一个测试容器，指定日志驱动为 `fluentd`：

```
docker run -d \
  --log-driver=fluentd \
  --log-opt fluentd-address=localhost:24224 \
  --log-opt tag=nginx-test \
  --name nginx-test \
  nginx
```

注意：确保 `fluentd` 容器已经启动并监听在 `localhost:24224`。在生产环境中，如果你是在不同机器上，需要将 `localhost` 替换为运行 fluentd 的主机 IP。

4. 在 Kibana 中查看日志

1. 访问 `http://localhost:5601`。
2. 进入 **Management**->**Kibana**->**Index Patterns**。
3. 创建新的 Index Pattern，输入 `docker-*` (我们在 fluent.conf 中配置的前缀)。
4. 选择 `@timestamp` 作为时间字段。
5. 去 **Discover** 页面，你就能看到 Nginx 容器的日志了。

Kibana 建索引模式常见坑

首次接入 EFK/ELK 时，“Elasticsearch 有数据但 Kibana 看不到”很常见，通常是 Kibana 配置或时间窗口问题：

- **Index Pattern 不匹配**：确认 Kibana 的 Index Pattern 与实际索引前缀一致。可以先用 `_cat/indices` 查看真实索引名。
- **时间字段选择错误**：若索引里包含 `@timestamp`，一般选择它；如果选择了错误的字段，会导致 Discover 无法按时间筛选。
- **时间窗口/时区**：Discover 右上角的时间范围默认可能是最近 15 分钟，且时区可能影响显示。建议先把范围扩大到最近 24 小时再验证。
- **数据解析失败**：若日志是非结构化文本，仍可入库但字段不可用；生产环境建议输出 JSON 并在采集端解析。

5. 验证日志是否写入 Elasticsearch：生产排错必备

当你在 Kibana 看不到日志时，建议先跳过 UI，从存储端直接验证“日志是否入库”。

1. 查看索引是否创建：

   ```
   curl -s http://localhost:9200/_cat/indices?v
   ```

   如果 Fluentd 使用了 `logstash_format true` 且 `logstash_prefix docker`，通常会看到形如 `docker-YYYY.MM.DD` 的索引。

2. 查看最近一段时间的日志文档：

   ```
   curl -s -H 'Content-Type: application/json' \
     http://localhost:9200/docker-*/_search \
     -d '{"size":1,"sort":[{"@timestamp":"desc"}]}'
   ```

如果 Elasticsearch 中已经有文档，但 Kibana 仍然为空，常见原因是：

- Index Pattern 没匹配到索引 (例如写成了 `docker-*` 但实际索引前缀不同)。
- 时间字段没选对或时区不一致，导致 Discover 时间窗口内看不到数据。

19.2.3 总结

通过 Docker 的日志驱动机制，结合 ELK/EFK 强大的收集和分析能力，我们可以轻松构建一个能够处理海量日志的监控平台，这对于排查生产问题至关重要。

Built with mdPress

19.2.4 生产要点

在生产环境中，日志系统往往比监控系统更容易因为“容量与写入压力”出问题，建议特别关注：

- **容量规划**：日志增长速度与磁盘占用直接相关。建议设置日志保留周期与索引生命周期策略 (ILM)，避免 Elasticsearch 因磁盘水位触发只读或不可用。
- **资源配置**：Elasticsearch 对 JVM Heap 较敏感。除示例中的 `ES_JAVA_OPTS` 外，生产环境需要结合节点内存、分片规模、查询压力做评估。
- **链路可靠性**：采集端到存储端要考虑网络抖动、背压与重试策略；当 Elasticsearch 写入变慢时，采集端的缓冲与落盘策略决定了是否会丢日志。
- **日志格式**：推荐应用输出结构化日志 (JSON) 并包含关键字段 (如 `trace_id`、`request_id`、`service`、`env`)，以便快速过滤与关联分析。

索引与保留策略的落地建议

无论是 EFK 还是 ELK，生产上都需要回答两个问题：

- 日志保留多久？
- 保留期内的日志如何保证可查询、不过度占用存储？

建议按环境与业务重要性对日志分层，并制定不同的保留周期，例如：

- **生产环境**：7～30 天
- **测试环境**：1～7 天

实现方式通常有两类：

- **按天滚动索引**：如 `docker-YYYY.MM.DD`，再定期删除过期索引。
- **使用 ILM**：定义 Hot/Warm/Cold/删除阶段，按时间与容量自动滚动与回收。

对于中小规模集群，先把“按天滚动 + 过期删除”做扎实，往往就能解决 80% 的容量问题；当日志量上来、查询压力变大后，再逐步引入 ILM、分层存储与更精细的分片规划。

最小可用的“过期索引清理”示例

如果你采用按天滚动索引 (例如 `docker-YYYY.MM.DD`)，可以通过 Elasticsearch API 定期清理过期索引。

下面示例仅用于演示思路：获取所有 `docker-` 前缀索引并删除指定索引。生产环境建议基于日期计算、灰度验证与权限控制后再执行自动化清理。

Built with mdPress

1. 列出索引：

```
curl -s http://localhost:9200/_cat/indices/docker-*?v
```

2. 删除某个过期索引 (示例)：

```
curl -X DELETE http://localhost:9200/docker-2026.02.01
```

如果你希望更自动化的治理能力，可以进一步使用 ILM 为索引配置滚动与删除策略。

Built with mdPress

19.3 容器性能优化与故障诊断

容器的轻量级特性不代表性能问题会自动消失。在实际运维中，性能瓶颈可能来自 CPU 限制、内存溢出、磁盘 I/O、网络拥塞等多个层面。本节深入讨论容器性能监控、诊断方法和优化策略。

19.3.1 容器性能监控指标

核心性能指标体系

容器性能监控涉及以下关键指标：

CPU 相关指标：

- `cpu.usage_usec`：容器 CPU 使用时间（微秒）
- `cpu.stat.nr_throttled`：CPU 限流发生次数
- `cpu.stat.throttled_usec`：CPU 限流总时间
- `cpu_percent`：CPU 使用百分比
- `cpu_quota`：CPU 配额设置（微秒）

内存相关指标：

- `memory.usage_bytes`：当前内存使用量
- `memory.max_usage_bytes`：内存使用峰值
- `memory.limit_in_bytes`：内存限制
- `memory.fail_cnt`：OOM（Out of Memory）失败次数
- `memory.stat.cache`：页面缓存占用
- `memory.stat.rss`：实际内存占用（RSS）
- `memory.stat.swap`：SWAP 使用量

网络相关指标：

- `rx_bytes`：接收字节数
- `tx_bytes`：发送字节数
- `rx_packets`：接收包数
- `tx_packets`：发送包数
- `rx_errors`：接收错误数
- `tx_errors`：发送错误数
- `rx_dropped`：接收丢包数
- `tx_dropped`：发送丢包数

I/O 相关指标：

- `io_service_bytes`：I/O 操作字节数
- `io_service_time`：I/O 操作耗时
- `io_queued`：I/O 队列长度
- `fs_limit_bytes`：文件系统限制
- `fs_usage_bytes`：文件系统使用量

19.3.2 使用 docker stats 实时监控

`docker stats` 是最基础但强大的监控工具，提供实时的容器资源使用情况。

基本使用：

```
# 实时监控所有运行中的容器
docker stats

# 输出示例：
# CONTAINER ID   NAME      CPU %   MEM USAGE / LIMIT     MEM %     NET I/O       BLOCK I/O
# abc123def456   nginx     0.45%   24.3 MiB / 256 MiB    9.49%     1.2kB / 3.4kB 0 B / 0 B
# def789ghi012   redis     0.23%   12.5 MiB / 512 MiB    2.44%     2.1kB / 1.5kB 0 B / 0 B

# 只监控特定容器
docker stats nginx redis

# 一次性输出不进入交互模式
docker stats --no-stream

# 每 2 秒采样一次（docker stats 没有 --interval 选项）
while true; do
  docker stats --no-stream
  sleep 2
done

# 格式化输出（使用 Go 模板）
docker stats --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}" --no-stream

# 导出为 JSON 格式用于日志记录
docker stats --format json --no-stream > stats.json
```

在脚本中使用：

```
#!/bin/bash

# 持续监控并记录到文件
while true; do
    timestamp=$(date '+%Y-%m-%d %H:%M:%S')
    docker stats --no-stream --format "{{.Container}},{{.CPUPerc}},{{.MemUsage}}" | \
        awk -v ts="$timestamp" '{print ts","$0}' >> container_stats.log
    sleep 10
done
```

性能指标解读：

```
# CPU % 超过 80%：需要增加 CPU 限制或优化应用
# MEM % 接近 100%：容器即将 OOM，需要增加内存或排查内存泄漏
# 如果 NET I/O 中 dropped 为非零：网络拥塞或丢包
```

19.3.3 cAdvisor 容器监控系统

cAdvisor 是 Google 开发的容器监控工具，提供比 `docker stats` 更详细的性能数据。

> ⚠️ **安全权衡提示**：下面的示例为简化部署使用了 `privileged: true`，与 第 18 章 中“最小权限 / `cap_drop=all`”的原则相冲突。生产环境建议改为按需授予能力（如 `cap_add: [SYS_ADMIN]` 加 `device_cgroup_rules` 与精确的 `devices`、`volumes` 挂载），并将 cAdvisor 部署在独立的监控网络中。如何选择请参考 18.4 节 关于内核能力（capabilities）的细化授权。

Docker Compose 部署 cAdvisor：

```yaml
services:
  cadvisor:
    image: ghcr.io/google/cadvisor:v0.56.2
    container_name: cadvisor
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
      - /dev/disk/:/dev/disk:ro
    privileged: true
    devices:
      - /dev/kmsg
    networks:
      - monitoring

networks:
  monitoring:
    driver: bridge
```

启动后访问 `http://localhost:8080` 查看：

- 容器性能统计
- 系统资源使用情况
- 历史性能数据

从 cAdvisor 提取指标：

```
# 获取所有容器的 JSON 格式性能数据
curl http://localhost:8080/api/v1.3/machine | jq .

# 获取特定容器信息
curl http://localhost:8080/api/v1.3/docker | jq '.docker | keys' | head -5

# 获取容器统计信息
curl http://localhost:8080/api/v1.3/docker/abc123/ | jq '.stats[-1]'
```

与 Prometheus 集成：

```yaml
# prometheus.yml 配置
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['localhost:8080']
    metrics_path: '/metrics'
```

19.3.4 Prometheus 容器监控配置

使用 Prometheus 和 node-exporter 进行长期的容器性能监控。

完整监控栈部署：

> **💡 Tip**
>
> 以下示例中的镜像标签（如 `prom/prometheus:v3.11.2`、`prom/node-exporter:v1.11.1`、`ghcr.io/google/cadvisor:v0.56.2`、`grafana/grafana:13.0.1`）仅为参考。在生产环境部署前，请访问各项目的官方发布页或文档获取最新版本号。

```yaml
services:
  prometheus:
    image: prom/prometheus:v3.11.2
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--storage.tsdb.retention.time=30d'
    extra_hosts:
      - "host.docker.internal:host-gateway"
    networks:
      - monitoring

  node-exporter:
    image: prom/node-exporter:v1.11.1
    container_name: node-exporter
    ports:
      - "9100:9100"
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.sysfs=/host/sys'
      - '--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($$|/)'
    networks:
      - monitoring

  cadvisor:
    image: ghcr.io/google/cadvisor:v0.56.2
    container_name: cadvisor
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    privileged: true
    networks:
      - monitoring

  grafana:
    image: grafana/grafana:13.0.1
    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
      - GF_INSTALL_PLUGINS=grafana-piechart-panel
    volumes:
      - grafana_data:/var/lib/grafana
    networks:
      - monitoring
```

```
volumes:
  prometheus_data:
  grafana_data:

networks:
  monitoring:
    driver: bridge
```

Prometheus 配置文件（prometheus.yml）：

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']

  - job_name: 'docker'
    static_configs:
      - targets: ['host.docker.internal:9323']
```

常用的 Prometheus 查询（PromQL）：

```
# 容器 CPU 使用百分比
rate(container_cpu_usage_seconds_total[5m]) * 100

# 容器内存使用百分比
(container_memory_usage_bytes / container_spec_memory_limit_bytes) * 100

# 容器网络入站流量（MB/s）
rate(container_network_receive_bytes_total[5m]) / 1024 / 1024

# 容器网络出站流量（MB/s）
rate(container_network_transmit_bytes_total[5m]) / 1024 / 1024

# 容器磁盘读取速率（MB/s）
rate(container_fs_reads_bytes_total[5m]) / 1024 / 1024

# CPU 限流情况
rate(container_cpu_cfs_throttled_seconds_total[5m])

# 内存缓存占比
container_memory_cache_bytes / container_memory_usage_bytes

# 按镜像统计容器数
count(container_memory_usage_bytes) by (image)
```

19.3.5 容器 OOM 排查与内存限制调优

OOM 问题诊断

```
# 检查容器是否因 OOM 被杀死
docker inspect <container_id> | grep OOMKilled

# 查看容器退出码：137 表示被 OOM 杀死
docker ps -a --format "{{.ID}}\t{{.Status}}" | grep "137"

# 查看容器日志中的 OOM 信息
docker logs <container_id> 2>&1 | grep -i "out of memory\|oom"

# 从宿主机日志查看 OOM 事件
dmesg | grep -i "oom\|kill"
journalctl -u docker -n 100 | grep -i "oom"
```

内存泄漏检测

使用专项工具分析应用内存使用：

Python 应用内存泄漏检测：

```
# Dockerfile
FROM python:3.14-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt memory_profiler tracemalloc

COPY app.py .
CMD ["python", "-m", "memory_profiler", "app.py"]
```

```
# app.py - 内存泄漏示例
from memory_profiler import profile
import tracemalloc

@profile
def memory_leak():
    # 不断创建未释放的列表
    data = []
    while True:
        data.append([0] * 1000000)
        print(f"List size: {len(data)}")

# 使用 tracemalloc 跟踪内存分配
tracemalloc.start()

# 执行可能泄漏的代码
# ...

current, peak = tracemalloc.get_traced_memory()
print(f"Current: {current / 1024 / 1024:.2f} MB")
print(f"Peak: {peak / 1024 / 1024:.2f} MB")
```

Java 应用内存分析：

```
# 在容器中启用 JVM 远程调试
docker run -e JAVA_OPTS="-Xmx512m -Xms256m -XX:+UseG1GC" \
  -p 5005:5005 \
  myapp:latest

# 使用 jstat 检查垃圾回收情况
jstat -gc <pid> 1000  # 每秒采样一次

# 输出示例:
#  S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC       MU    CCSC   CCSU
# 6144   6144      0   6144   39424    12288   149504     84320     50552   47689  6464   5989
```

内存限制最佳实践

```
# 为容器设置内存限制
docker run -m 512m --memory-swap 1g myapp:latest

# 参数说明：
# -m / --memory：内存限制（这里是 512MB）
# --memory-swap：内存+SWAP 总额（这里是 1GB，意味着 SWAP 为 512MB）
# 如果不设置 --memory-swap，则等于 --memory 值

# Docker Compose 配置
services:
  app:
    image: myapp:latest
    deploy:
      resources:
        limits:
          memory: 512M
        reservations:
          memory: 256M
```

内存超额提交（Memory Overcommit）：

```
# 在 Docker Compose 中区分限制和预留
# limits：绝不能超过的最大值
# reservations：Compose 排期时的参考值

services:
  web:
    memory: 512M  # 限制
    memswap_limit: 1G  # SWAP 限制

  db:
    memory: 2G
    memory_reservation: 1G  # 预留 1GB，允许突发到 2GB
```

19.3.6 镜像体积优化与多阶段构建

镜像体积分析工具

使用 dive 分析镜像层：

```
# 安装 dive
wget https://github.com/wagoodman/dive/releases/download/v0.13.1/dive_0.13.1_linux_amd64.deb
sudo apt install ./dive_0.13.1_linux_amd64.deb

# 分析镜像
dive myapp:latest

# 输出详细的分层信息，显示每一层的大小和内容
```

使用 Dockerfile 分析工具：

```
# 安装 hadolint
curl https://github.com/hadolint/hadolint/releases/download/v2.14.0/hadolint-Linux-x86_64 -L -o hado
lint
chmod +x hadolint

# 检查 Dockerfile 最佳实践
./hadolint Dockerfile
```

多阶段构建最佳实践

Go 应用的最小化镜像构建：

```
# Stage 1: 构建阶段
FROM golang:1.26-alpine AS builder

WORKDIR /build

# 安装依赖
RUN apk add --no-cache git ca-certificates tzdata

COPY go.mod go.sum ./
RUN go mod download

COPY . .

# 构建静态二进制（支持 scratch 基础镜像）
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
    -a -installsuffix cgo \
    -ldflags="-w -s" \
    -o app .

# Stage 2: 运行阶段
FROM scratch

# 从 builder 复制必要的文件
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
COPY --from=builder /usr/share/zoneinfo /usr/share/zoneinfo
COPY --from=builder /build/app /app

EXPOSE 8080
ENTRYPOINT ["/app"]

# 最终镜像大小通常 < 15MB（相比 golang:1.26-alpine 的 ~1GB）
```

Node.js 应用的多阶段构建：

```dockerfile
# Stage 1: 依赖安装
FROM node:24-alpine AS dependencies

WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production && \
    npm cache clean --force

# Stage 2: 构建阶段
FROM node:24-alpine AS builder

WORKDIR /app
COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

# Stage 3: 运行阶段
FROM node:24-alpine

WORKDIR /app

# 从依赖阶段复制 node_modules
COPY --from=dependencies /app/node_modules ./node_modules

# 从构建阶段复制构建产物
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./

# 删除开发依赖和不必要的文件
RUN rm -rf src tests *.config.js

USER node
EXPOSE 3000

CMD ["node", "dist/index.js"]

# 镜像大小对比:
# 不优化: ~500MB
# 多阶段构建后: ~120MB (减少 76%)
```

Python 应用的多阶段构建:

```dockerfile
# Stage 1: 构建阶段
FROM python:3.14-slim AS builder

WORKDIR /build

RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Stage 2: 运行阶段
FROM python:3.14-slim

WORKDIR /app

# 从 builder 复制虚拟环境
COPY --from=builder /root/.local /root/.local

# 设置 PATH
ENV PATH=/root/.local/bin:$PATH \
    PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1

COPY . .

USER nobody
EXPOSE 5000

CMD ["python", "app.py"]
```

镜像体积优化检查清单

```
# 检查清单
□ 使用精简基础镜像（Alpine、Distroless）
□ 清理包管理器缓存（apt-get clean、rm -rf /var/cache/*）
□ 在同一 RUN 指令中安装和清理依赖
□ 使用 .dockerignore 排除不必要的文件
□ 多阶段构建避免构建依赖污染最终镜像
□ 去除调试符号：-ldflags="-w -s"（Go）、strip 命令（C/C++）
□ 压缩静态资源和应用文件
□ 使用 BuildKit 缓存优化加速构建

# 优化示例：
FROM ubuntu:24.04

# ❌ 不推荐
RUN apt-get update
RUN apt-get install -y curl wget git
RUN apt-get clean

# ✓ 推荐
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    curl \
    wget \
    git && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

19.3.7 常见性能问题及解决方案

问题 1: 容器频繁被 OOM 杀死

症状：容器进程被无故杀死，exit code 137 解决方案：

```
# 增加内存限制
docker update -m 1g <container_id>

# 排查内存泄漏
docker exec <container_id> ps aux | grep -E "VSZ|RSS"

# 使用 docker stats 实时监控
docker stats <container_id>

# 启用内存交换（作为最后手段）
docker run -m 512m --memory-swap 1g myapp:latest
```

问题 2: CPU 被限流（CPU Throttling）

症状：应用性能突然下降，但 CPU 使用率不高 诊断：

```
# 查看 CPU 限流统计
docker exec <container_id> cat /sys/fs/cgroup/cpu/cpu.stat

# 如果 throttled_time > 0，说明发生了 CPU 限流
# 解决方案：增加 CPU 限制
docker update --cpus 2 <container_id>
```

问题 3: 网络丢包或延迟高

诊断：

```
# 进入容器检查网络状态
docker exec <container_id> ip -s link show

# 检查路由和 DNS
docker exec <container_id> cat /etc/resolv.conf

# 测试网络延迟
docker exec <container_id> ping 8.8.8.8

# 检查容器网络驱动
docker inspect <container_id> | grep -A 10 NetworkSettings

# 解决方案：更换网络驱动或调整 MTU
# host 网络可降低网络栈开销，但会放弃容器网络隔离；仅在明确接受安全边界变化时使用
docker run --net=host myapp:latest
```

本章小结

本章从两个维度介绍了容器可观测性：

- **指标监控**：以 Prometheus + Grafana 为主，完成指标采集、存储与可视化。
- **日志管理**：以 EFK/ELK 为例，完成容器日志的集中采集、检索与分析。

生产环境中，建议将“可观测性”当成一个完整闭环：**采集 -> 存储 -> 展示 -> 告警 -> 排错 -> 容量治理**。

扩展阅读：Docker 日志驱动

Docker 提供了多种日志驱动 (Log Driver)，用于将容器标准输出的日志转发到不同后端。

常见的日志驱动包括：

- `json-file`：默认驱动，将日志以 JSON 格式写入本地文件。
- `syslog`：将日志转发到 syslog 服务器。
- `journald`：将日志写入 systemd journal。
- `fluentd`：将日志转发到 fluentd 收集器。
- `gelf`：支持 GELF 协议的日志后端 (如 Graylog)。
- `awslogs`：发送到 Amazon CloudWatch Logs。

生产建议：无论采用哪种驱动，都要明确日志的保留周期、容量上限与传输可靠性，避免“日志把磁盘写满”或“链路抖动导致丢日志”。

19.4 日志平台选型对比与注意事项

日志平台通常由“采集/处理/存储/查询展示”几部分组成。常见选型包括：

- **EFK/ELK**：Elasticsearch + Fluentd/Logstash + Kibana，适合全文检索与结构化查询。
- **Loki + Grafana**：更偏“日志像指标一样存储”的思路，部署与成本可能更友好，但查询能力与使用习惯不同。

选型时建议关注：

- **写入压力与背压**：当存储端变慢时，采集端是否会缓冲、落盘、重试，是否会影响业务。
- **容量治理**：是否具备按天/按大小滚动、保留策略、生命周期管理 (ILM) 等能力。
- **安全与合规**：鉴权、TLS、审计、敏感字段脱敏。
- **可运维性**：升级策略、备份恢复、告警指标是否齐全。

19.5 上线前检查清单

你可以用下面的清单快速检查“是否具备最小生产可用性”：

- Prometheus 数据目录已持久化，并设置了合理的保留周期。
- Prometheus Targets 全部为 UP，并且关键查询 (CPU/内存/容器指标) 有数据。
- Grafana 已导入面板并能定位到具体实例/容器；默认账号密码已修改。
- 至少有一条关键告警已打通 Alertmanager 的接收链路，并验证告警能被正确发送与抑制。
- Elasticsearch 数据目录已持久化，并有明确的日志保留周期与容量上限策略。
- Kibana 能查询到最新日志；当 UI 异常时能用 Elasticsearch API 验证入库。
- 可观测性组件未直接暴露到公网，访问已加鉴权或置于内网。

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第二十章 实战案例 - 操作系统

章节概述

本章将介绍 Docker 在不同操作系统镜像场景下的实战案例。当你构建容器化应用时，选择合适的基础镜像至关重要。不同的操作系统镜像在大小、功能和性能方面各有特点，适用于不同的使用场景。本章通过具体的案例，详细讲解如何在 Docker 中使用主流操作系统镜像，包括轻量级镜像 (Busybox、Alpine) 和完整功能镜像 (Debian、Ubuntu、CentOS 等)。

版本说明

本章示例中使用的操作系统镜像版本遵循以下原则：

- **Alpine、Debian、Ubuntu、CentOS** 等操作系统镜像采用大版本或次版本标签（如 `alpine:3.21`、`ubuntu:26.04`），避免使用 `latest` 标签确保构建的可再现性
- **OS 大版本保留**，以便获得最新的安全补丁和修复
- 在生产环境中，建议根据实际需求选择合适的版本，并定期更新以获得安全修复

为什么选择合适的操作系统镜像很重要

在容器化应用开发中，选择合适的基础操作系统镜像直接影响容器的大小、启动速度、安全性和运行性能。不同的镜像提供了不同的功能集和资源占用：

- **轻量级镜像** (Busybox、Alpine) - 镜像大小仅几 MB，启动快速，适合微服务、IoT 设备和对资源敏感的环境。Busybox 是最小的选择，集成了常见的 Unix 工具；Alpine 则提供了完整的包管理器，方便安装额外工具。
- **通用镜像** (Debian、Ubuntu) - 提供完整的 Linux 功能和丰富的软件生态，镜像大小通常在 100-300 MB 之间。适合需要灵活安装各种依赖和工具的应用场景。
- **企业级镜像** (CentOS、Fedora) - 基于 Red Hat 生态，广泛应用于企业环境和复杂系统应用。提供了 yum 包管理器和强大的系统管理工具。

选择镜像的关键原则是 “小而够用” ——选择满足应用需求的最小镜像。这样可以减少安全漏洞表面积、加快镜像拉取和推送速度、降低存储成本，同时也使容器更便于分发和部署。

常用操作系统镜像对比

镜像	大小	包管理器	适用场景	优势
Busybox	~1 MB	无	最小化工具集、initrd	极致轻量，启动秒级
Alpine	~5 MB	apk	微服务、静态应用	体积小，有包管理器
Debian	~100 MB	apt-get	通用应用、开发环境	软件包丰富，稳定性强
Ubuntu	~80 MB	apt-get	类似 Debian，现代化系统	更新频繁，用户多
CentOS	~200 MB	yum	企业应用、兼容性需求	企业级支持，稳定性高
Fedora	~200 MB	dnf	新特性需求、开发环境	最新技术栈，创新性强

学习目标

通过学习本章内容，你将能够：

- 理解不同操作系统镜像的特点、大小和适用场景
- 掌握在 Docker 中使用各类操作系统镜像的方法和最佳实践
- 学习如何根据实际需求选择合适的基础镜像，实现镜像优化
- 了解如何在不同操作系统容器中安装、配置和管理应用程序
- 掌握多阶段构建等高级技巧，最小化最终镜像大小
- 学会使用 Docker Compose 编排多个操作系统容器环境

Built with mdPress

章节内容导航

- Busybox — 超轻量级工具集镜像，适合嵌入式和最小化容器
- Alpine — 轻量级 Linux 镜像，广泛用于生产环境微服务
- Debian Ubuntu — 功能完整的通用 Linux 镜像，生态丰富
- CentOS Fedora — 企业级 Linux 镜像，适合复杂系统应用
- 本章小结

20.1 Busybox

20.1.1 简介

下图直观地展示了本节内容：



BusyBox 是一个集成了一百多个最常用 Linux 命令和工具 (如 cat、echo、grep、mount、telnet 等) 的精简工具箱，它只需要几 MB 的大小，很方便进行各种快速验证，被誉为 “Linux 系统的瑞士军刀”。

BusyBox 可运行于多款 POSIX 环境的操作系统中，如 Linux (包括 Android)、Hurd、FreeBSD 等。

20.1.2 获取官方镜像

可以使用 `docker pull` 指令下载 `busybox:latest` 镜像：

```
$ docker pull busybox:latest
latest: Pulling from library/busybox
5c4213be9af9: Pull complete
Digest: sha256:c6b45a95f932202dbb27c31333c4789f45184a744060f6e569cc9d2bf1b9ad6f
Status: Downloaded newer image for busybox:latest
docker.io/library/busybox:latest
```

下载后，可以看到 busybox 镜像只有 **2.433 MB**：

```
$ docker image ls
REPOSITORY                  TAG                 IMAGE ID            CREATED             VIRTUAL SIZ
E
busybox                     latest              e72ac664f4f0        6 weeks ago         2.433 MB
```

20.1.3 运行 busybox

启动一个 busybox 容器，并在容器中执行 grep 命令。

```
$ docker run -it busybox
/ # grep
BusyBox v1.22.1 (2014-05-22 23:22:11 UTC) multi-call binary.

Usage: grep [-HhnlLoqvsriwFE] [-m N] [-A/B/C N] PATTERN/-e PATTERN.../-f FILE [FILE]...

Search for PATTERN in FILEs (or stdin)

        -H      Add 'filename:' prefix
        -h      Do not add 'filename:' prefix
        -n      Add 'line_no:' prefix
        -l      Show only names of files that match
        -L      Show only names of files that don't match
        -c      Show only count of matching lines
        -o      Show only the matching part of line
        -q      Quiet. Return 0 if PATTERN is found, 1 otherwise
        -v      Select non-matching lines
        -s      Suppress open and read errors
        -r      Recurse
        -i      Ignore case
        -w      Match whole words only
        -x      Match whole lines only
        -F      PATTERN is a literal (not regexp)
        -E      PATTERN is an extended regexp
        -m N    Match up to N times per file
        -A N    Print N lines of trailing context
        -B N    Print N lines of leading context
        -C N    Same as '-A N -B N'
        -e PTRN Pattern to match
        -f FILE Read pattern from file
```

查看容器内的挂载信息。

```
/ # mount
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/BOTCI5RF24AMC4A2UWF4N6ZWF
P:/var/lib/docker/overlay2/l/TWVP5T5DMKJGXZOROR7CAPWGFP,upperdir=/var/lib/docker/overlay2/801ef0bf6c
ce35288dbb8fe00a4f9cc47760444693bfdf339ed0bdcf926e12a3/diff,workdir=/var/lib/docker/overlay2/801ef0b
f6cce35288dbb8fe00a4f9cc47760444693bfdf339ed0bdcf926e12a3/work)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,relatime,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup (ro,nosuid,nodev,noexec,relatime,xattr,release_agent=/l
ib/systemd/systemd-cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (ro,nosuid,nodev,noexec,relatime,net_cls,net_p
rio)
cgroup on /sys/fs/cgroup/freezer type cgroup (ro,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (ro,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/cpuset type cgroup (ro,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/blkio type cgroup (ro,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/perf_event type cgroup (ro,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/memory type cgroup (ro,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/devices type cgroup (ro,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/pids type cgroup (ro,nosuid,nodev,noexec,relatime,pids)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k)
/dev/vda1 on /etc/resolv.conf type ext3 (rw,noatime,data=ordered)
/dev/vda1 on /etc/hostname type ext3 (rw,noatime,data=ordered)
/dev/vda1 on /etc/hosts type ext3 (rw,noatime,data=ordered)
devpts on /dev/console type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
proc on /proc/bus type proc (ro,relatime)
proc on /proc/fs type proc (ro,relatime)
proc on /proc/irq type proc (ro,relatime)
proc on /proc/sys type proc (ro,relatime)
proc on /proc/sysrq-trigger type proc (ro,relatime)
tmpfs on /proc/acpi type tmpfs (ro,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/keys type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/timer_list type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/sched_debug type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /sys/firmware type tmpfs (ro,relatime)
```

busybox 镜像虽然小巧，但包括了大量常见的 Linux 命令，读者可以用它快速熟悉 Linux 命令。

20.1.4 相关资源

- Busybox 官网：https://busybox.net/
- Busybox 官方仓库：https://git.busybox.net/busybox/
- Busybox 官方镜像：https://hub.docker.com/_/busybox/
- Busybox 官方仓库：https://github.com/docker-library/busybox

20.2 Alpine

20.2.1 简介

下图直观地展示了本节内容：



Alpine 操作系统是一个面向安全的轻型 Linux 发行版。它不同于通常 Linux 发行版，Alpine 采用了 musl libc 和 busybox 以减小系统的体积和运行时资源消耗，但功能上比 busybox 又完善的多，因此得到开源社区越来越多的青睐。在保持瘦身的同时，Alpine 还提供了自己的包管理工具 apk，可以通过 Alpine Packages 网站上查询包信息，也可以直接通过 apk 命令直接查询和安装各种软件。

Alpine 由非商业组织维护的，支持广泛场景的 Linux 发行版，它特别为资深/重度 Linux 用户而优化，关注安全，性能和资源效能。Alpine 镜像可以适用于更多常用场景，并且是一个优秀的可以适用于生产的基础系统/环境。

Alpine Docker 镜像也继承了 Alpine Linux 发行版的这些优势。相比于其他 Docker 镜像，它的容量非常小，压缩后仅约 **5 MB**（对比 Ubuntu 系列镜像约 28 MB），且拥有非常友好的包管理机制。官方镜像来自 docker-alpine 项目。

目前 Docker 官方已开始推荐使用 Alpine 替代之前的 Ubuntu 做为基础镜像环境。这样会带来多个好处。包括镜像下载速度加快，镜像安全性提高，主机之间的切换更方便，占用更少磁盘空间等。

下表是官方镜像的大小比较（压缩后大小，实际数据以 Docker Hub 为准）：

```
REPOSITORY          TAG           COMPRESSED SIZE
alpine              latest        ~5 MB
debian              latest        ~30 MB
ubuntu              latest        ~28 MB
```

> **提示**：镜像大小会随版本更新而变化，精确值请查阅 Docker Hub 各镜像的 Tags 页面。Alpine 的优势不仅在于体积小，还在于更少的预装包带来更小的攻击面。

20.2.2 获取并使用官方镜像

由于镜像很小，下载时间往往很短，读者可以直接使用 docker run 指令直接运行一个 Alpine 容器，并指定运行的 Linux 指令，例如：

```
$ docker run alpine echo '123'
123
```

20.2.3 迁移至 Alpine 基础镜像

目前，大部分 Docker 官方镜像都已经支持 `Alpine` 作为基础镜像，可以很容易进行迁移。

例如：

- `ubuntu/debian` -> `alpine`
- `python:3` -> `python:3-alpine`
- `ruby:2.6` -> `ruby:2.6-alpine`

另外，如果使用 `Alpine` 镜像替换 `Ubuntu` 基础镜像，安装软件包时需要用 `apk` 包管理器替换 `apt` 工具，如

```
$ apk add --no-cache <package>
```

`Alpine` 中软件安装包的名字可能会与其他发行版有所不同，可以在 `https://pkgs.alpinelinux.org/packages` 网站搜索并确定安装包名称。如果需要的安装包不在主索引内，但是在测试或社区索引中。那么可以按照以下方法使用这些安装包。

```
$ echo "http://dl-cdn.alpinelinux.org/alpine/edge/testing" >> /etc/apk/repositories
$ apk --update add --no-cache <package>
```

由于在国内访问 `apk` 仓库较缓慢，建议在使用 `apk` 之前先替换仓库地址为国内镜像。

```
RUN sed -i "s/dl-cdn.alpinelinux.org/mirrors.aliyun.com/g" /etc/apk/repositories \
      && apk add --no-cache <package>
```

20.2.4 相关资源

- `Alpine` 官网：https://www.alpinelinux.org/
- `Alpine` 官方仓库：https://github.com/alpinelinux
- `Alpine` 官方镜像：https://hub.docker.com/_/alpine/
- `Alpine` 官方镜像仓库：https://github.com/alpinelinux/docker-alpine

20.3 Debian Ubuntu

Debian 和 Ubuntu 都是目前较为流行的 **Debian 系** 的服务器操作系统，十分适合研发场景。Docker Hub 上提供了官方镜像，国内各大容器云服务也基本都提供了相应的支持。

20.3.1 Debian 系统简介

下图直观地展示了本节内容：



Debian 是由 GPL 和其他自由软件许可协议授权的自由软件组成的操作系统，由 **Debian 计划 (Debian Project)** 组织维护。**Debian 计划** 是一个独立的、分散的组织，由 3000 人志愿者组成，接受世界多个非盈利组织的资金支持，Software in the Public Interest 提供支持并持有商标作为保护机构。Debian 以其坚守 Unix 和自由软件的精神，以及其给予用户的众多选择而闻名。现时 Debian 包括了超过 25,000 个软件包并支持 12 个计算机系统结构。

Debian 作为一个大的系统组织框架，其下有多种不同操作系统核心的分支计划，主要为采用 Linux 核心的 Debian GNU/Linux 系统，其他还有采用 GNU Hurd 核心的 Debian GNU/Hurd 系统、采用 FreeBSD 核心的 Debian GNU/kFreeBSD 系统，以及采用 NetBSD 核心的 Debian GNU/NetBSD 系统。甚至还有利用 Debian 的系统架构和工具，采用 OpenSolaris 核心构建而成的 Nexenta OS 系统。在这些 Debian 系统中，以采用 Linux 核心的 Debian GNU/Linux 最为著名。

众多的 Linux 发行版，例如 Ubuntu、Knoppix 和 Linspire 及 Xandros 等，都基于 Debian GNU/Linux。

使用 Debian 官方镜像

Debian 是一个常用的基础镜像。

官方提供了大家熟知的 debian 镜像以及面向科研领域的 neurodebian 镜像。可以使用 docker run 直接运行 Debian 镜像。

```
$ docker run -it debian bash
root@668e178d8d69:/# cat /etc/issue
Debian GNU/Linux 13
```

Debian 镜像很适合作为基础镜像，构建自定义镜像。

20.3.2 Ubuntu 系统简介

下图直观地展示了本节内容：



Ubuntu 是一个以桌面应用为主的 GNU/Linux 操作系统，其名称来自非洲南部祖鲁语或豪萨语的 “ubuntu” 一词 (官方译名 “友帮拓”，另有 “吾帮托”、“乌班图”、“有奔头” 或 “乌斑兔” 等译名)。Ubuntu 意思是 “人性” 以及 “我的存在是因为大家的存在”，是非洲传统的一种价值观，类似华人社会的 “仁爱” 思想。Ubuntu 基于 Debian 发行版和 GNOME/Unity 桌面环境，与 Debian 的不同在于它每 6 个月会发布一个新版本，每 2 年推出一个长期支持 **(Long Term Support，LTS)** 版本；LTS 版本通常提供 5 年标准安全维护，Ubuntu Pro/ESM 可进一步延长安全覆盖。

使用 Ubuntu 官方镜像

Ubuntu 是目前最流行的 Linux 发行版之一。

下面以 ubuntu:26.04 为例，演示如何使用该镜像安装一些常用软件。

首先使用 -ti 参数启动容器，登录 bash，查看 ubuntu 的发行版本号。

```
$ docker run -ti ubuntu:26.04 /bin/bash
root@7d93de07bf76:/# cat /etc/os-release
PRETTY_NAME="Ubuntu 26.04 LTS"
NAME="Ubuntu"
VERSION_ID="26.04"
VERSION="26.04 LTS (Resolute Raccoon)"
VERSION_CODENAME=resolute
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
```

当试图直接使用 apt-get 安装一个软件的时候，会提示 E: Unable to locate package。

```
root@7d93de07bf76:/# apt-get install curl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
E: Unable to locate package curl
```

这并非系统不支持 apt-get 命令。Docker 镜像在制作时为了精简清除了 apt 仓库信息，因此需要先执行 apt-get update 命令来更新仓库信息。更新信息后即可成功通过 apt-get 命令来安装软件。

```
root@7d93de07bf76:/# apt-get update
Get:1 http://archive.ubuntu.com/ubuntu resolute InRelease [256 kB]
Get:2 http://security.ubuntu.com/ubuntu resolute-security InRelease [126 kB]
...
Fetched 25.8 MB in 8s (3215 kB/s)
Reading package lists... Done
```

首先，安装 `curl` 工具。

```
root@7d93de07bf76:/# apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  ca-certificates krb5-locales libasn1-8-heimdal libcurl4 libgssapi-krb5-2 libgssapi3-heimdal libhcr
ypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
  libk5crypto3 libkeyutils1 libkrb5-26-heimdal libkrb5-3 libkrb5support0 libldap-2.4-2 libldap-commo
n libnghttp2-14 libpsl5 libroken18-heimdal librtmp1 libsasl2-2 libsasl2-modules libsasl2-modules-db
libsqlite3-0 libssl1.1 libwind0-heimdal openssl publicsuffix
...
root@7d93de07bf76:/# curl
curl: try 'curl --help' or 'curl --manual' for more information
```

接下来，再安装 apache 服务。

```
root@7d93de07bf76:/# apt-get install -y apache2
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  apache2-bin apache2-data apache2-utils file libapr1 libaprutil1 libaprutil1-dbd-sqlite3 libaprutil
1-ldap libexpat1 libgdbm-compat4 libgdbm5 libicu60 liblua5.2-0 libmagic-mgc libmagic1 libperl5.26 li
bxml2 mime-support netbase perl perl-modules-5.26 ssl-cert xz-utils
...
```

启动这个 apache 服务，然后使用 `curl` 来测试本地访问。

```
root@7d93de07bf76:/# service apache2 start
 * Starting web server apache2
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.1
7.0.2. Set the 'ServerName' directive globally to suppress this message
 *
root@7d93de07bf76:/# curl 127.0.0.1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtm
l1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <!--
    Modified from the Debian original for Ubuntu
    Last updated: 2016-11-16
    See: https://launchpad.net/bugs/1288690
  -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Apache2 Ubuntu Default Page: It works</title>
    <style type="text/css" media="screen">
...
```

配合使用 `-p` 参数对外映射服务端口，可以允许容器外来访问该服务。

20.3.3 相关资源

- `Debian` 官网：https://www.debian.org/
- `Neuro Debian` 官网：http://neuro.debian.net/
- `Debian` 官方仓库：https://github.com/Debian
- `Debian` 官方镜像：https://hub.docker.com/_/debian/
- `Debian` 官方镜像仓库：https://github.com/tianon/docker-brew-debian/
- `Ubuntu` 官网：https://ubuntu.com
- `Ubuntu` 官方仓库：https://github.com/ubuntu
- `Ubuntu` 官方镜像：https://hub.docker.com/_/ubuntu/
- `Ubuntu` 官方镜像仓库：https://github.com/tianon/docker-brew-ubuntu-core

20.4 CentOS Fedora

20.4.1 CentOS 系统简介

CentOS 和 Fedora 都是基于 Redhat 的常见 Linux 分支。传统 CentOS Linux 曾是企业级服务器的常用操作系统，但相关 Docker 官方镜像已停止维护；新部署通常应评估 Rocky Linux、AlmaLinux、CentOS Stream 或直接使用 RHEL/UBI 等替代方案。Fedora 则主要面向个人桌面用户。

CentOS 操作系统

CentOS (Community Enterprise Operating System，中文意思是：社区企业操作系统)，它是基于 Red Hat Enterprise Linux 源代码编译而成。由于 CentOS 与 Redhat Linux 源于相同的代码基础，所以很多成本敏感且需要高稳定性的公司就使用 CentOS 来替代商业版 Red Hat Enterprise Linux。CentOS 自身不包含闭源软件。

使用 CentOS 官方镜像

CentOS 官方镜像的使用非常简单。

注意：CentOS 8 已于 2021 年 12 月 31 日停止维护 (EOL)，CentOS 7 已于 2024 年 6 月 30 日停止维护。对于新部署，推荐使用 Rocky Linux、AlmaLinux 或 CentOS Stream 等替代发行版。

使用 docker run 直接运行 CentOS 7 镜像，并登录 bash。

```
$ docker run -it centos:7 bash
Unable to find image 'centos:7' locally
7: Pulling from library/centos
3d8673bd162a: Pull complete
Digest: sha256:a66ffcb73930584413de83311ca11a4cb4938c9b2521d331026dad970c19adf4
Status: Downloaded newer image for centos:7
[root@43eb3b194d48 /]# cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
```

20.4.2 Fedora 系统简介

下图直观地展示了本节内容：

Fedora 操作系统

Fedora 由 Fedora Project 社区开发，红帽公司赞助的 Linux 发行版。它的目标是创建一套新颖、多功能并且自由和开源的操作系统。Fedora 的功能对于用户而言，它是一套功能完备的，可以更新的

免费操作系统，而对赞助商 Red Hat 而言，它是许多新技术的测试平台。被认为可用的技术最终会加入到 Red Hat Enterprise Linux 中。

使用 Fedora 官方镜像

使用 docker run 命令直接运行 Fedora 官方镜像，并登录 bash。

```
$ docker run -it fedora bash
Unable to find image 'fedora:latest' locally
latest: Pulling from library/fedora
2bf01635e2a0: Pull complete
Digest: sha256:64a02df6aac27d1200c2572fe4b9949f1970d05f74d367ce4af994ba5dc3669e
Status: Downloaded newer image for fedora:latest
[root@196ca341419b /]# cat /etc/redhat-release
Fedora release 43 (Forty Three)
```

20.4.3 相关资源

- Fedora 官网：https://getfedora.org/
- Fedora 官方仓库：https://github.com/fedora-infra
- Fedora 官方镜像：https://hub.docker.com/_/fedora/
- Fedora 官方镜像仓库：https://github.com/fedora-cloud/docker-brew-fedora
- CentOS 官网：https://www.centos.org
- CentOS 官方仓库：https://github.com/CentOS
- CentOS 官方镜像：https://hub.docker.com/_/centos/
- CentOS 官方镜像仓库：https://github.com/CentOS/CentOS-Dockerfiles

本章小结

本章讲解了典型操作系统镜像的下载和使用。

除了官方的镜像外，在 `Docker Hub` 上还有许多第三方组织或个人上传的 Docker 镜像。

读者可以根据具体情况来选择。一般来说：

- 官方镜像体积都比较小，只带有一些基本的组件。精简的系统有利于安全、稳定和高效的运行，也适合进行个性化定制。
- 出于安全考虑，几乎所有官方制作的镜像都没有安装 SSH 服务，无法通过用户名和密码直接登录到容器中。

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

第二十一章 实战案例 - DevOps

版本说明

本章示例中使用的镜像版本遵循以下原则：

- **GitHub Actions** (`actions/checkout@v6` 等) 采用大版本标签，代表最新的功能版本与稳定性的平衡
- **编程语言镜像** (`golang:1.26-alpine`、`rust:1.95-alpine` 等) 采用次版本标签，确保 API 稳定性同时获得安全更新
- **数据库镜像** (`postgres:16-alpine`、`redis:8-alpine` 等) 采用大版本标签，便于获得修复和改进
- **基础镜像** (`alpine:3.21`、`debian:12` 等) 采用大版本或次版本标签，避免 `latest` 以确保可再现性
- 实际项目中应根据需求调整版本，生产环境建议定期更新以获取安全补丁

DevOps 背景介绍

DevOps 是一种重要的开发和运维文化，强调开发团队和运维团队之间的协作和自动化。它致力于通过自动化和流程优化，加快软件交付速度，同时提高系统的稳定性和可靠性。Docker 作为容器化技术的领导者，已成为现代 DevOps 工作流中不可或缺的工具。通过容器化应用，开发团队可以确保“一次构建，处处运行”，消除开发、测试和生产环境的差异，大大简化了部署流程。

Docker 在 DevOps 中的角色

Docker 在 DevOps 工作流中承担多个关键角色。首先，它标准化了应用的开发和部署环境，使得团队成员在相同的 Docker 容器中工作，避免了“在我的机器上可以运行”的问题。其次，Docker 与 CI/CD 流程无缝集成，通过自动化的镜像构建、测试和部署，实现快速的迭代周期。此外，Docker 还支持微服务架构和容器编排，使团队能够更灵活地扩展应用和管理基础设施。

CI/CD 管道的重要性

持续集成与持续部署 (CI/CD) 是现代 DevOps 的核心。通过自动化的代码检测、测试、构建和部署流程，团队可以更加频繁地发布新版本，同时保持系统的稳定性和可靠性。Docker 在 CI/CD 中扮演了重要角色：

- **标准化构建环境** - Docker 确保开发、测试和生产环境完全一致，消除了环境差异带来的问题
- **加速流水线** - 容器的快速启动和轻量级特性，大幅加快了 CI/CD 流程的执行效率
- **灵活的测试框架** - 可以轻松创建短生命周期的测试容器，并行运行多个测试
- **自动化镜像发布** - CI/CD 工具可以自动构建、扫描、标记和推送 Docker 镜像到仓库
- **蓝绿部署和金丝雀发布** - 利用容器的隔离性和可重复性，实现高级发布策略

本章将通过介绍 GitHub Actions、Drone 等流行的 CI/CD 工具，展示如何在实际项目中构建完整的自动化流水线。我们还将演示如何在本地开发环境中集成 Docker，使用 IDE 的容器开发插件，加快本地迭代周期。

本章学习目标

通过学习本章内容，你将能够：

- 理解 DevOps 文化、CI/CD 流程和容器化的紧密关系
- 掌握完整的 Docker 工作流，从代码提交到线上部署的每一个环节
- 学习如何使用 GitHub Actions 实现自动化 CI/CD，以及工作流的编写和优化
- 了解 Drone 等第三方 CI/CD 工具的架构、部署和配置方式
- 学会在本地 IDE (VS Code) 中集成 Docker，利用容器开发工具提升开发效率
- 掌握实战中常见的 DevOps 场景、最佳实践和故障排查方法

章节内容导航

- DevOps 完整工作流 — 从代码到部署的全流程
- GitHub Actions — 使用 GitHub Actions 实现 CI/CD
- Drone — Drone CI/CD 平台简介和配置
- Drone Demo — Drone 实战演示和应用
- 在 IDE 中使用 Docker — IDE 与 Docker 集成的好处
- VS Code — Visual Studio Code 容器开发指南
- 实战例子 — 真实项目中的 DevOps 应用案例
- 本章小结

Built with mdPress

21.1 DevOps 完整工作流

本章将演示一个基于 Docker、Kubernetes 和 Jenkins/GitLab CI 的完整 DevOps 工作流。

21.1.1 工作流概览

1. **Code**：开发人员提交代码到 GitLab。
2. **Build**：GitLab CI 触发构建任务。
3. **Test**：运行单元测试和集成测试。
4. **Package**：构建 Docker 镜像并推送到 Harbor/Registry。
5. **Deploy (Staging)**：自动部署到测试环境 Kubernetes 集群。
6. **Verify**：人工或自动化验证。
7. **Release (Production)**：审批后自动部署到生产环境。

21.1.2 关键配置示例

本节通过一组最小可用的片段，展示典型 DevOps 流程中与 Docker 相关的关键配置。

1. Dockerfile 多阶段构建

使用 Docker 多阶段构建可以有效减小镜像体积。

```
## Build stage

FROM golang:1.26 AS builder
WORKDIR /app
COPY . .
RUN go build -o main .

## Final stage

FROM alpine:3.21
WORKDIR /app
COPY --from=builder /app/main .
CMD ["./main"]
```

2. GitLab CI 配置

GitLab CI（`.gitlab-ci.yml`）配置如下：

```yaml
stages:
  - test
  - build
  - deploy

unit_test:
  stage: test
  image: golang:1.26
  script:
    - go test ./...

build_image:
  stage: build
  image: docker:29
  services:
    - docker:29-dind
  script:
    - echo "$CI_REGISTRY_PASSWORD" | docker login -u "$CI_REGISTRY_USER" --password-stdin $CI_REGIST
RY
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA

deploy_staging:
  stage: deploy
  image: dtzar/helm-kubectl
  script:
    - printf '%s' "$KUBE_CA_PEM" > kube-ca.crt
    - kubectl config set-cluster k8s --server=$KUBE_URL --certificate-authority=kube-ca.crt --embed-
certs=true
    - kubectl config set-credentials admin --token=$KUBE_TOKEN
    - kubectl config set-context default --cluster=k8s --user=admin
    - kubectl config use-context default
    - kubectl set image deployment/myapp myapp=$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
      -n staging
  only:
    - develop
```

21.1.3 最佳实践

1. **不可变基础设施**：一旦镜像构建完成，在各个环境（Dev、Staging、Prod）中都应该使用同一个镜像 tag（通常是 commit hash），而不是重新构建。
2. **配置分离**：使用 ConfigMap 和 Secret 管理环境特定的配置，不要打包进镜像。
3. **应对 Docker Hub 限额 (Rate Limits)**：
    - Docker Hub 对匿名拉取实施了严格的限制 (6 小时内约 100 次)。若在 CI/CD 中频繁构建，极易触发 `toomanyrequests` 错误。
    - **最佳策略**：
        - 在流水线开头始终执行安全的身份认证 (使用 PAT，而非密码)。
        - 将常用的基础镜像缓存到自建的 Harbor/Nexus，使用 Pull-Through Cache。
        - 开启 Docker 构建引擎 (BuildKit) 的 inline 或 registry 缓存功能，以降低全量拉取频率。
4. **GitOps**：考虑引入 ArgoCD，将部署配置也作为代码存储在 Git 中，实现 Git 驱动的部署同步。

21.2 GitHub Actions

GitHub Actions 是 GitHub 推出的一款 CI/CD 工具。

我们可以在每个 `job` 的 `step` 中使用 Docker 执行构建步骤。

21.2.1 最小可用示例

更多语法、权限模型和可用 action，请以 GitHub Actions 官方文档 为准。

在仓库根目录创建 `.github/workflows/ci.yml`：

```yaml
name: CI

on:
  push:
  pull_request:

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: docker/setup-buildx-action@v4
      - uses: docker/build-push-action@v7
        with:
          context: .
          push: false
          tags: local/test:ci
```

该示例会在 GitHub Actions 中构建当前仓库的 Docker 镜像（不推送到 registry）。

21.2.2 构建并推送到 Registry

实际项目中通常需要在 CI 中构建镜像并推送到容器 Registry。以下示例展示了多阶段构建 + 登录 + 推送的完整流程：

```yaml
name: Build and Push

on:
  push:
    branches: [main]

permissions:
  contents: read
  packages: write

jobs:
  build-push:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6

      - uses: docker/login-action@v4
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - uses: docker/setup-buildx-action@v4

      - uses: docker/build-push-action@v7
        with:
          context: .
          push: true
          tags: |
            ghcr.io/${{ github.repository }}:${{ github.sha }}
            ghcr.io/${{ github.repository }}:latest
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

关键说明：

- `docker/login-action` 负责认证，支持 Docker Hub、GHCR、ECR 等主流 Registry。
- `cache-from` / `cache-to` 使用 GitHub Actions 原生缓存（`type=gha`），无需额外配置即可加速增量构建。
- 标签同时使用 commit hash 和 `latest`，兼顾版本追溯与部署便利。

21.2.3 最佳实践

- 固定 action 的主版本（例如 `@v4` / `@v6`），避免使用 `@master` 这类浮动引用。
- 设置最小权限（例如 `contents: read`），需要写入权限时再打开。
- 需要依赖缓存时，优先使用官方支持的缓存方案（例如针对语言包管理器的 cache 或 BuildKit cache）。
- 敏感凭据（Registry 密码、Deploy Key 等）一律通过 `secrets` 注入，禁止硬编码。
- 多平台构建可在 `build-push-action` 中添加 `platforms: linux/amd64,linux/arm64`。

如果你需要在某个步骤里直接运行容器镜像（而不是构建镜像），可以使用 `docker://` 语法：

```
- name: Run container step
  uses: docker://golang:alpine
  with:
    args: go version
```

21.3 Drone

基于 Docker 的 CI/CD 工具 Drone，所有编译、测试的流程都在容器中进行。

开发者只需在项目中包含 .drone.yml 文件，将代码推送到 git 仓库，Drone 就能够自动化地进行编译、测试、发布。

本小节以 GitHub + Drone 来演示 Drone 的工作流程。 当然在实际开发过程中，你的代码也许不在 GitHub 托管，那么你可以尝试使用 Gogs + Drone 来进行 CI/CD。

21.3.1 关联项目

在 GitHub 新建一个名为 drone-demo 的仓库。

打开我们已经部署好的 Drone 网站或者 Drone Cloud， 使用 GitHub 账号登录，在界面中关联刚刚新建的 drone-demo 仓库。

21.3.2 编写项目源代码

初始化一个 git 仓库：

```
mkdir drone-demo
cd drone-demo

git init

git remote add origin git@github.com:username/drone-demo.git
```

这里以一个简单的 Go 程序为例，该程序输出 Hello World!

编写 app.go 文件：

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello World!\n")
}
```

编写 .drone.yml 文件：

```
kind: pipeline
type: docker
name: build

steps:
  - name: build
    image: golang:alpine
    pull: if-not-exists
    environment:
      KEY: VALUE
    commands:
      - echo $KEY
      - pwd
      - ls
      - CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
      - ./app

trigger:
  branch:
    - master
```

现在目录结构如下：

```
.
├── .drone.yml
└── app.go
```

21.3.3 推送项目源代码到 GitHub

```
git add .

git commit -m "test drone ci"

git push origin master
```

21.3.4 查看项目构建过程及结果

打开我们部署好的 Drone 网站或者 Drone Cloud，即可看到构建结果。

图

当然我们也可以把构建结果上传到 GitHub、Docker Registry、 云服务商提供的对象存储，或者生产环境中。

21.4 Drone Demo

21.4.1 Demo 项目说明

这是一个基于 Go 语言编写的简单 Web 应用示例，用于演示 Drone CI 的持续集成流程。

21.4.2 目录结构

- `drone_demo.app.go`：简单的 Go Web 服务器代码。
- `drone_demo.drone.yml`：Drone CI 的配置文件，定义了构建和测试流程。

21.4.3 如何使用

1. 确保本地已安装 Docker 环境。
2. 将示例文件重命名为 Drone 期望的文件名：

   ```
   cp drone_demo.app.go app.go
   cp drone_demo.drone.yml .drone.yml
   ```

3. 将 `app.go` 与 `.drone.yml` 推送到你的 `drone-demo` 仓库，即可在 Drone 中看到构建结果。

21.5 在 IDE 中使用 Docker

使用 IDE 进行开发，往往要求本地安装好工具链。一些 IDE 支持 Docker 容器中的工具链，这样充分利用了 Docker 的优点，而无需在本地安装。

本节关注一个核心目标：**把“开发依赖”放进容器，把“源码编辑体验”留在本地 IDE**。

21.5.1 适用场景

- 团队希望统一开发环境（Go/Node/Python 版本、系统依赖、编译链）。
- 本地系统不方便安装依赖（例如 Windows、公司管控环境）。
- 项目依赖较重（例如需要 `gcc`、数据库客户端、特定系统库）。

不太适合的场景：强依赖本机 GPU/USB 设备、或需要非常低延迟文件 IO 的工程（此时可能需要额外调优挂载/同步策略）。

21.5.2 最小可用模式：docker compose + 开发容器

下面用一个“长期运行的开发容器”作为例子（以 Go 为例，你可以替换为 Node/Python）。

1. 在项目中创建 `compose.yaml`（或复用你已有的 compose 文件）：

```
services:
  dev:
    image: golang:1.26
    working_dir: /work
    volumes:
      - ./:/work
    command: sleep infinity
```

2. 启动开发容器：

```
docker compose up -d
```

3. 进入容器安装依赖/执行命令：

```
docker compose exec dev bash
go version
go test ./...
```

这个模式的优点是“简单直接、IDE 无关”，缺点是 IDE 需要额外配置 （例如配置远程解释器/语言服务，或使用 VS Code Dev Containers）。

21.5.3 目录挂载与权限建议

- Linux 下如果遇到容器内写文件权限问题，优先确保容器内用户与宿主机 UID/GID 对齐。 VS Code Dev Containers 支持自动处理；手写 Dockerfile/compose 时也可以显式设置用户。
- 如果遇到文件变更监听不生效（常见于 macOS/Windows 的虚拟化文件系统）， 优先使用语言/工具支持的轮询模式或提高 watcher 限制。

Built with mdPress

21.6 VS Code

VS Code 的 Dev Containers 可以把“开发环境”放进容器，同时保留 VS Code 的编辑、补全、调试体验。

本节提供一个最小可用示例：把任意项目（以 Go 为例）变成“打开即开发”的容器化环境。

21.6.1 前置条件

- 安装 Docker Desktop（或 Linux 上的 Docker Engine）。
- VS Code 安装扩展：Dev Containers（`ms-vscode-remote.remote-containers`）。

21.6.2 最小示例：.devcontainer/devcontainer.json

在项目根目录创建 `.devcontainer/devcontainer.json`：

```
{
  "name": "docker-practice-dev",
  "image": "golang:1.26",
  "workspaceFolder": "/work",
  "workspaceMount": "source=${localWorkspaceFolder},target=/work,type=bind",
  "customizations": {
    "vscode": {
      "extensions": [
        "golang.Go"
      ]
    }
  },
  "postCreateCommand": "go version"
}
```

然后在 VS Code 命令面板选择：

- `Dev Containers: Reopen in Container`

VS Code 会拉取镜像并启动容器，随后你就可以在容器内运行：

```
go test ./...
```

21.6.3 结合 Docker Compose（可选）

如果项目同时依赖数据库/缓存（例如 Postgres/Redis），可以使用 `dockerComposeFile` 把依赖一起拉起。

Built with mdPress

示例（devcontainer.json 片段）：

```
{
  "name": "compose-dev",
  "dockerComposeFile": [
    "../docker-compose.yml"
  ],
  "service": "dev",
  "workspaceFolder": "/work"
}
```

注意：service 需要对应 compose 里的服务名。

21.7 实战案例：Go/Rust/数据库/微服务

本节通过实际项目案例演示如何为不同类型的应用构建最优化的 Docker 镜像，以及如何使用 Docker Compose 构建完整的开发和生产环境。

21.7.1 Go 应用的最小化镜像构建

Go 语言因其编译为静态二进制和快速启动而特别适合容器化。以下展示如何构建极小的 Go 应用镜像。

超小 Go Web 服务

应用代码（main.go）：

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "os"
)

func healthHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        fmt.Fprintf(w, `{"status":"healthy","version":"1.0.0"}`)
}

func helloHandler(w http.ResponseWriter, r *http.Request) {
        hostname, _ := os.Hostname()
        w.Header().Set("Content-Type", "application/json")
        fmt.Fprintf(w, `{"message":"Hello from %s","version":"1.0.0"}`, hostname)
}

func main() {
        http.HandleFunc("/health", healthHandler)
        http.HandleFunc("/hello", helloHandler)
        http.HandleFunc("/", helloHandler)

        port := ":8080"
        log.Printf("Server starting on %s", port)

        if err := http.ListenAndServe(port, nil); err != nil {
                log.Fatalf("Server failed: %v", err)
        }
}
```

多阶段 Dockerfile：

```dockerfile
# Stage 1: 构建阶段
FROM golang:1.26-alpine AS builder

WORKDIR /build

# 安装构建依赖
RUN apk add --no-cache git ca-certificates tzdata

# 复制模块文件（利用缓存）
COPY go.mod go.sum ./
RUN go mod download

# 复制源代码
COPY . .

# 构建静态二进制
# -ldflags="-w -s" 去除调试符号减小体积
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
    -a -installsuffix cgo \
    -ldflags="-w -s -X main.Version=1.0.0 -X main.BuildTime=$(date -u +'%Y-%m-%dT%H:%M:%SZ')" \
    -o app .

# Stage 2: 运行阶段（scratch 镜像）
FROM scratch

# 复制 CA 证书（用于 HTTPS）
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/

# 复制时区数据（用于时间处理）
COPY --from=builder /usr/share/zoneinfo /usr/share/zoneinfo

# 复制应用二进制
COPY --from=builder /build/app /app

EXPOSE 8080

# 使用绝对路径作为 ENTRYPOINT
ENTRYPOINT ["/app"]
```

构建和测试：

```
# 构建镜像
docker build -t go-app:latest .

# 检查镜像大小
docker images go-app

# 运行容器
docker run -d -p 8080:8080 --name go-demo go-app:latest

# 测试应用
curl http://localhost:8080/health | jq .

# 进入容器验证
docker exec go-demo ls -la /

# 只包含 /app 和系统必要文件

# 镜像大小通常 < 10MB（相比 golang:1.26 基础镜像的 ~900MB）
docker history go-app:latest
```

go.mod 和 go.sum 示例：

```
module github.com/example/go-app

go 1.26

require (
    // 如果需要依赖
)
```

带依赖的 Go 应用

应用代码（使用 Gin 框架）：

```go
package main

import (
        "github.com/gin-gonic/gin"
        "log"
)

func main() {
        router := gin.Default()

        router.GET("/health", func(c *gin.Context) {
                c.JSON(200, gin.H{
                        "status": "ok",
                })
        })

        router.GET("/api/users", func(c *gin.Context) {
                c.JSON(200, gin.H{
                        "users": []string{"alice", "bob"},
                })
        })

        log.Fatal(router.Run(":8080"))
}
```

优化的 Dockerfile：

```dockerfile
FROM golang:1.26-alpine AS builder

WORKDIR /src

RUN apk add --no-cache git ca-certificates tzdata

COPY go.mod go.sum ./
RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 \
    go build -a -installsuffix cgo \
    -ldflags="-w -s" \
    -o app .

# 最终镜像
FROM alpine:3.21

RUN apk add --no-cache ca-certificates tzdata

WORKDIR /root/

COPY --from=builder /src/app .

EXPOSE 8080

CMD ["./app"]
```

21.7.2 Rust 应用的最小化镜像构建

Rust 因其性能和安全性在系统级应用中备受青睐。

应用代码（main.rs）：

```
use actix_web::{web, App, HttpServer, HttpResponse};
use std::sync::Mutex;

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    println!("Starting server on 0.0.0.0:8080");

    HttpServer::new(|| {
        App::new()
            .route("/health", web::get().to(health))
            .route("/hello", web::get().to(hello))
    })
    .bind("0.0.0.0:8080")?
    .run()
    .await
}

async fn health() -> HttpResponse {
    HttpResponse::Ok().json(serde_json::json!({
        "status": "healthy"
    }))
}

async fn hello() -> HttpResponse {
    HttpResponse::Ok().json(serde_json::json!({
        "message": "Hello from Rust"
    }))
}
```

Cargo.toml：

```
[package]
name = "rust-app"
version = "0.1.0"
edition = "2021"

[[bin]]
name = "rust-app"
path = "src/main.rs"

[dependencies]
actix-web = "4.13"
tokio = { version = "1.35", features = ["full"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

多阶段构建 Dockerfile：

```
# Stage 1: 编译
FROM rust:1.95-alpine AS builder

RUN apk add --no-cache musl-dev

WORKDIR /src

COPY Cargo.* ./
COPY src ./src

# 构建优化的发布版本
RUN cargo build --release

# Stage 2: 运行镜像
FROM alpine:3.21

RUN apk add --no-cache ca-certificates

COPY --from=builder /src/target/release/rust-app /app

EXPOSE 8080

CMD ["/app"]
```

构建和验证：

```
docker build -t rust-app:latest .
docker run -d -p 8080:8080 rust-app:latest
curl http://localhost:8080/health | jq .

# Rust 应用通常比 Go 更小：5-20MB（取决于依赖）
docker images rust-app
```

21.7.3 数据库容器化最佳实践

PostgreSQL 生产部署

自定义 PostgreSQL 镜像：

```
FROM postgres:16-alpine

# 安装额外工具
RUN apk add --no-cache \
    postgresql-contrib \
    pg-stat-monitor \
    curl

# 复制初始化脚本
COPY init-db.sql /docker-entrypoint-initdb.d/
COPY health-check.sh /

RUN chmod +x /health-check.sh

HEALTHCHECK --interval=10s --timeout=5s --start-period=40s --retries=3 \
    CMD /health-check.sh

EXPOSE 5432
```

初始化脚本（init-db.sql）：

```
-- 创建自定义用户
CREATE USER appuser WITH PASSWORD 'secure_password';

-- 创建数据库
CREATE DATABASE myappdb OWNER appuser;

-- 创建扩展
\c myappdb

CREATE EXTENSION IF NOT EXISTS uuid-ossp;
CREATE EXTENSION IF NOT EXISTS hstore;
CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- 创建表
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    username VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 创建索引
CREATE INDEX idx_users_username ON users (username);
CREATE INDEX idx_users_email ON users (email);

-- 授予权限
GRANT CONNECT ON DATABASE myappdb TO appuser;
GRANT USAGE ON SCHEMA public TO appuser;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO appuser;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO appuser;
```

健康检查脚本（health-check.sh）：

```
#!/bin/bash

PGPASSWORD=$POSTGRES_PASSWORD pg_isready \
  -h localhost \
  -U $POSTGRES_USER \
  -d $POSTGRES_DB \
  -p 5432 > /dev/null 2>&1

exit $?
```

Docker Compose 配置：

```yaml
services:
  postgres:
    build:
      context: .
      dockerfile: Dockerfile.postgres
    container_name: postgres-db
    environment:
      POSTGRES_DB: myappdb
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in .env}
      POSTGRES_INITDB_ARGS: "--encoding=UTF8 --locale=en_US.UTF-8"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./backups:/backups
    ports:
      # 只暴露给本机调试；生产环境优先不发布数据库端口
      - "127.0.0.1:5432:5432"
    networks:
      - backend
    restart: unless-stopped
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

  # 备份服务
  backup:
    image: postgres:16-alpine
    depends_on:
      - postgres
    environment:
      PGPASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in .env}
    volumes:
      - ./backups:/backups
    command: |
      sh -c 'while true; do
        pg_dump -h postgres -U postgres -d myappdb > /backups/backup_$$(date +%Y%m%d_%H%M%S).sql
        echo "Backup completed at $$(date)"
        sleep 86400
      done'
    networks:
      - backend

volumes:
  postgres_data:
    driver: local

networks:
  backend:
    driver: bridge
```

性能优化配置：

```yaml
services:
  postgres:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: myappdb
    command:
      - "postgres"
      - "-c"
      - "max_connections=200"
      - "-c"
      - "shared_buffers=256MB"
      - "-c"
      - "effective_cache_size=1GB"
      - "-c"
      - "maintenance_work_mem=64MB"
      - "-c"
      - "checkpoint_completion_target=0.9"
      - "-c"
      - "wal_buffers=16MB"
      - "-c"
      - "default_statistics_target=100"
      - "-c"
      - "random_page_cost=1.1"
      - "-c"
      - "effective_io_concurrency=200"
      - "-c"
      - "work_mem=1310kB"
      - "-c"
      - "min_wal_size=1GB"
      - "-c"
      - "max_wal_size=4GB"
      - "-c"
      - "max_worker_processes=4"
      - "-c"
      - "max_parallel_workers_per_gather=2"
      - "-c"
      - "max_parallel_workers=4"
      - "-c"
      - "max_parallel_maintenance_workers=2"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres_data:
```

Built with mdPress

MySQL/MariaDB 部署

```
FROM mariadb:11

# 复制自定义配置
COPY my.cnf /etc/mysql/conf.d/custom.cnf

# 初始化脚本
COPY init.sql /docker-entrypoint-initdb.d/

EXPOSE 3306

HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
    CMD mariadb-admin ping -h localhost || exit 1
```

自定义 my.cnf：

```
[mysqld]

# 性能优化
max_connections = 200
default_storage_engine = InnoDB
innodb_buffer_pool_size = 1GB
innodb_log_file_size = 256MB
query_cache_type = 0
query_cache_size = 0

# 日志配置
log_error = /var/log/mysql/error.log
slow_query_log = 1
slow_query_log_file = /var/log/mysql/slow.log
long_query_time = 2

# 复制配置
server_id = 1
log_bin = mysql-bin
binlog_format = ROW
```

Redis 缓存部署

```
FROM redis:8-alpine

# 复制 Redis 配置
COPY redis.conf /usr/local/etc/redis/redis.conf

# 使用配置文件启动
CMD ["redis-server", "/usr/local/etc/redis/redis.conf"]

EXPOSE 6379

HEALTHCHECK --interval=5s --timeout=3s --retries=5 \
    CMD redis-cli ping || exit 1
```

redis.conf 配置：

```
# 绑定地址
bind 0.0.0.0

# 端口
port 6379

# 密码保护
requirepass your_secure_password

# 内存管理
maxmemory 512mb
maxmemory-policy allkeys-lru

# 持久化
save 900 1
save 300 10
save 60 10000

# AOF 持久化
appendonly yes
appendfsync everysec

# 日志
loglevel notice
logfile ""

# 客户端输出缓冲限制
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit replica 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
```

21.7.4 微服务架构的 Docker Compose 编排

三层微服务架构示例：

```yaml
services:
  # 前端服务
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: frontend
    ports:
      - "3000:3000"
    environment:
      REACT_APP_API_URL: http://localhost:8000
      NODE_ENV: production
    depends_on:
      - api
    networks:
      - frontend-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000"]
      interval: 30s
      timeout: 10s
      retries: 3

  # API 服务
  api:
    build:
      context: ./api
      dockerfile: Dockerfile
    container_name: api
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgresql://appuser:password@postgres:5432/myappdb
      REDIS_URL: redis://redis:6379
      LOG_LEVEL: info
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
    networks:
      - frontend-network
      - backend-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 512M
        reservations:
          cpus: '0.5'
          memory: 256M

  # PostgreSQL 数据库
  postgres:
```

```
    image: postgres:16-alpine
    container_name: postgres
    environment:
      POSTGRES_DB: myappdb
      POSTGRES_USER: appuser
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - backend-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U appuser -d myappdb"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Redis 缓存
  redis:
    image: redis:8-alpine
    container_name: redis
    command: redis-server --appendonly yes --requirepass redispass
    volumes:
      - redis_data:/data
    networks:
      - backend-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "redis-cli", "--raw", "incr", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Nginx 反向代理
  nginx:
    image: nginx:alpine
    container_name: nginx
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/conf.d:/etc/nginx/conf.d:ro
      - ./ssl:/etc/nginx/ssl:ro
    depends_on:
      - frontend
      - api
    networks:
      - frontend-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "wget", "--quiet", "--tries=1", "--spider", "http://localhost/health"]
      interval: 30s
      timeout: 10s
      retries: 3

volumes:
  postgres_data:
    driver: local
```

```
  redis_data:
    driver: local

networks:
  frontend-network:
    driver: bridge
  backend-network:
    driver: bridge
```

nginx.conf 配置：

```
upstream frontend {
    server frontend:3000;
}

upstream api {
    server api:8000;
}

server {
    listen 80;
    server_name localhost;
    client_max_body_size 100M;

    # 健康检查端点
    location /health {
        access_log off;
        return 200 "OK\n";
        add_header Content-Type text/plain;
    }

    # 前端应用
    location / {
        proxy_pass http://frontend;
        proxy_set_header Host $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;
    }

    # API 接口
    location /api/ {
        proxy_pass http://api/;
        proxy_set_header Host $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;
        proxy_redirect off;

        # WebSocket 支持
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }

    # 静态资源缓存
    location ~* ^.+\.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2|ttf|eot)$ {
        proxy_pass http://frontend;
        expires 30d;
        add_header Cache-Control "public, immutable";
    }
}
```

21.7.5 使用 VS Code Dev Containers

Dev Containers 让整个开发环境容器化，提升团队一致性。

.devcontainer/devcontainer.json：

```
{
  "name": "Python Dev Environment",
  "image": "mcr.microsoft.com/devcontainers/python:3.14",

  "features": {
    "ghcr.io/devcontainers/features/docker-in-docker:2": {},
    "ghcr.io/devcontainers/features/git:1": {}
  },

  "customizations": {
    "vscode": {
      "extensions": [
        "ms-python.python",
        "ms-python.vscode-pylance",
        "ms-python.pylint",
        "charliermarsh.ruff",
        "ms-vscode-remote.remote-containers"
      ],
      "settings": {
        "python.linting.enabled": true,
        "python.linting.pylintEnabled": true,
        "python.formatting.provider": "black",
        "[python]": {
          "editor.formatOnSave": true,
          "editor.defaultFormatter": "ms-python.python"
        }
      }
    }
  },

  "postCreateCommand": "pip install -r requirements.txt && pip install pytest black pylint",

  "forwardPorts": [8000, 5432, 6379],
  "portsAttributes": {
    "8000": {
      "label": "Application",
      "onAutoForward": "notify"
    },
    "5432": {
      "label": "PostgreSQL",
      "onAutoForward": "ignore"
    },
    "6379": {
      "label": "Redis",
      "onAutoForward": "ignore"
    }
  },

  "mounts": [
    "source=${localEnv:HOME}/.ssh,target=/home/vscode/.ssh,readonly"
  ],

  "remoteUser": "vscode"
}
```

.devcontainer/Dockerfile：

```
FROM mcr.microsoft.com/devcontainers/python:3.14

# 安装额外工具
RUN apt-get update && apt-get install -y \
    postgresql-client \
    redis-tools \
    curl \
    git \
    && rm -rf /var/lib/apt/lists/*

# 创建虚拟环境
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

WORKDIR /workspace
```

Docker Compose 用于 Dev Containers：

```
# .devcontainer/docker-compose.yml
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      DATABASE_URL: postgresql://dev:dev@postgres:5432/myapp
      REDIS_URL: redis://redis:6379
    volumes:
      - ..:/workspace:cached
    ports:
      - "8000:8000"
    depends_on:
      - postgres
      - redis

  postgres:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: dev
      POSTGRES_PASSWORD: dev
      POSTGRES_DB: myapp
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:8-alpine

volumes:
  postgres_data:
```

本章小结

本章通过一个完整的 DevOps 工作流示例，串联了从代码提交、自动化测试、镜像构建、 到部署发布的一整套实践路径。

在落地时，建议重点把握以下原则：

- **不可变交付物**：同一份产物（镜像）在不同环境中流转，使用 commit hash 等方式标识版本。
- **最小权限与密钥管理**：CI 平台权限尽量收敛，敏感信息使用 Secret 管理并定期轮换。
- **流水线可观测与可回滚**：为关键步骤保留日志与制品，发布失败时能快速定位并回滚。
- **开发环境一致性**：通过 Dev Containers / compose 开发容器减少“在我电脑上可以”的问题。

下一步你可以根据团队现状选择一条主线深入：

- 已在用 GitHub：优先补全 Actions 的缓存、制品、发布策略。
- 自建体系：结合私有 Registry、Kubernetes 与 GitOps 工具完善部署与审计。

> 📝 **发现错误或有改进建议？** 欢迎提交 Issue 或 PR。

Built with mdPress

附录

本附录汇总了 Docker 相关的常见问题、镜像参考、命令速查、最佳实践、调试方法、官方资源、术语表与学习路线，便于按需查阅。

目录

- **附录一：常见问题与错误速查**：汇总学习和使用 Docker 过程中的常见问题与错误解决方案。
- **附录二：热门镜像介绍**：介绍常用官方镜像的用途、基本用法与适用场景。
- **附录三：Docker 命令查询**：速查 Docker 客户端和服务端的常用命令。
- **附录四：Dockerfile 最佳实践**：提供编写高效、安全 Dockerfile 的指导原则。
- **附录五：如何调试 Docker**：介绍 Docker 调试技巧和工具。
- **附录六：资源链接**：提供官方文档、发布页和参考入口。
- **附录七：术语表**：统一全书中英文术语、缩写与命令写法。
- **附录八：Docker 学习路线图与知识体系**：给出阶段化学习路径和知识地图。

Built with mdPress

附录一：常见问题与错误速查

更多错误排查条目见 常见错误处理。

镜像相关

如何批量清理临时镜像文件?

答：可以使用 `docker image prune` 命令。

如何查看镜像支持的环境变量?

答：可以使用 `docker run IMAGE env` 命令。

本地的镜像文件都存放在哪里?

答：与 Docker 相关的本地资源默认存放在 `/var/lib/docker/` 目录下，以 `overlay2` 文件系统为例，其中 `containers` 目录存放容器信息，`image` 目录存放镜像信息，`overlay2` 目录下存放具体的镜像层文件。

构建 Docker 镜像应该遵循哪些原则?

答：整体原则上，尽量保持镜像功能的明确和内容的精简，要点包括

- 尽量选取满足需求但较小的基础系统镜像，例如大部分时候可以选择 `alpine` 镜像，仅有不足六兆大小；
- 清理编译生成文件、安装包的缓存等临时文件；
- 安装各个软件时候要指定准确的版本号，并避免引入不需要的依赖；
- 从安全角度考虑，应用要尽量使用系统的库和依赖；
- 如果安装应用时候需要配置一些特殊的环境变量，在安装后要还原不需要保持的变量值；
- 使用 Dockerfile 创建镜像时候要添加 .dockerignore 文件或使用干净的工作目录。

更多内容请查看 Dockerfile 最佳实践

Built with mdPress

碰到网络问题，无法 pull 镜像，命令行指定 http_proxy 无效?

答：先区分代理要作用在哪一层。Docker daemon 拉取镜像时，推荐在 `daemon.json` 的 `proxies` 字段或 systemd drop-in 中配置 `HTTP_PROXY` / `HTTPS_PROXY` / `NO_PROXY`，然后重启 Docker 服务。Docker CLI、构建过程和容器内应用的代理应分别使用 `~/.docker/config.json`、`--build-arg` 或 `docker run --env` 配置；不要把 `export http_proxy=...` 当作 `daemon.json` 内容写入。

容器相关

容器退出后，通过 docker container ls 命令查看不到，数据会丢失么?

答：容器退出后会处于终止 (exited) 状态，此时可以通过 `docker container ls -a` 查看。其中的数据也不会丢失，还可以通过 `docker start` 命令来启动它。只有删除掉容器才会清除所有数据。

如何停止所有正在运行的容器?

答：可以使用 `docker stop $(docker container ls -q)` 命令。

如何批量清理已经停止的容器?

答：可以使用 `docker container prune` 命令。

如何获取某个容器的 PID 信息?

答：可以使用

```
docker inspect --format '{{ .State.Pid }}' <CONTAINER ID or NAME>
```

如何获取某个容器的 IP 地址?

答：可以使用

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' <CONTAINER ID or NAME>
```

如何给容器指定一个固定 IP 地址，而不是每次重启容器 IP 地址都会变?

答：使用以下命令启动容器可以使容器 IP 固定不变

```
$ docker network create -d bridge --subnet 172.25.0.0/16 my-net

$ docker run --network=my-net --ip=172.25.3.3 -itd --name=my-container busybox
```

如何临时退出一个正在交互的容器的终端，而不终止它?

答：按 `Ctrl-p Ctrl-q`。如果按 `Ctrl-c` 往往会让容器内应用进程终止，进而会终止容器。

使用 `docker port` 命令映射容器的端口时，系统报错 “Error：No public port ‘80’ published for xxx”？

答：

- 创建容器时用 `-p HOST_PORT:CONTAINER_PORT` 或 `--publish` 显式发布端口，例如 `docker run -p 8080:80 nginx`；
- 只想把镜像声明的 `EXPOSE` 端口随机发布到宿主机端口时，用 `-P` / `--publish-all`，之后通过 `docker ps` 查看实际端口；
- `Dockerfile` 中的 `EXPOSE` 只是镜像元数据，不会自动发布端口。

可以在一个容器中同时运行多个应用进程么?

答：一般并不推荐在同一个容器内运行多个应用进程。如果有类似需求，可以通过一些额外的进程管理机制，比如 `supervisord` 来管理所运行的进程。可以参考 Docker 官方说明。

如何控制容器占用 CPU、内存等系统资源的份额?

答：在使用 `docker create` 命令创建容器或使用 `docker run` 创建并启动容器的时候，可以使用 -c|--cpu-shares[=0] 参数来调整容器使用 CPU 的权重；使用 -m|--memory[=MEMORY] 参数来调整容器使用内存的大小。

仓库相关

仓库、注册服务器、注册索引有何关系?

首先，仓库是存放一组关联镜像的集合，比如同一个应用的不同版本的镜像。

注册服务器是存放实际的镜像文件的地方。注册索引则负责维护用户的账号、权限、搜索、标签等的管理。因此，注册服务器利用注册索引来实现认证等管理。

配置相关

Docker 的配置文件放在哪里，如何修改配置?

答：使用 `systemd` 的系统 (如 Ubuntu 22.04+、Debian 12+、Rocky/Alma/CentOS Stream 9+) 的配置文件在 `/etc/docker/daemon.json`。

如何更改 Docker 的默认存储位置?

答：Docker 的默认存储位置是 `/var/lib/docker`，如果希望将 Docker 的本地文件存储到其他分区，可以使用 Linux 软连接的方式来完成，或者修改配置文件 `/etc/docker/daemon.json` 的 `data-root` 项。可以使用 `docker info | grep "Docker Root Dir"` 查看当前使用的存储位置。

例如，如下操作将默认存储位置迁移到 /storage/docker。

```
[root@s26 ~]# df -h
Filesystem                    Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup-lv_root   50G  5.3G   42G  12% /
tmpfs                          48G  228K   48G   1% /dev/shm
/dev/sda1                     485M   40M  420M   9% /boot
/dev/mapper/VolGroup-lv_home  222G  188M  210G   1% /home
/dev/sdb2                     2.7T  323G  2.3T  13% /storage
[root@s26 ~]# service docker stop
[root@s26 ~]# cd /var/lib/
[root@s26 lib]# mv docker /storage/
[root@s26 lib]# ln -s /storage/docker/ docker
[root@s26 lib]# ls -la docker
lrwxrwxrwx. 1 root root 15 11月 17 13:43 docker -> /storage/docker
[root@s26 lib]# service docker start
```

使用内存和 swap 限制启动容器时候报内核不支持警告?

答：如果遇到 `WARNING: Your kernel does not support cgroup swap limit` 等警告，这是因为系统默认没有开启对内存和 swap 使用的统计功能，引入该功能会带来性能的下降。要开启该功能，可以采取如下操作：

- 编辑 `/etc/default/grub` 文件 (Ubuntu 系统为例)，配置 `GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"`
- 更新 grub：`$ sudo update-grub`
- 重启系统，即可。

Docker 与虚拟化

Docker 与 LXC 有何不同?

答：LXC 利用 Linux 上相关技术实现了容器。Docker 则在如下的几个方面进行了改进：

- 移植性：通过抽象容器配置，容器可以实现从一个平台移植到另一个平台；
- 镜像系统：基于 OverlayFS 的镜像系统为容器的分发带来了很多的便利，同时共同的镜像层只需要存储一份，实现高效率的存储；
- 版本管理：类似于 Git 的版本管理理念，用户可以更方便的创建、管理镜像文件；
- 仓库系统：仓库系统大大降低了镜像的分发和管理的成本；
- 周边工具：各种现有工具 (配置管理、云平台) 对 Docker 的支持，以及基于 Docker 的 PaaS、CI 等系统，让 Docker 的应用更加方便和多样化。

Docker 与 Vagrant 有何不同?

答：两者的定位完全不同。

- Vagrant 类似 Boot2Docker (一款运行 Docker 的最小内核)，是一套虚拟机的管理环境。Vagrant 可以在多种系统上和虚拟机软件中运行，可以在 Windows，Mac 等非 Linux 平台上为 Docker 提供支持，自身具有较好的包装性和移植性。
- 原生的 Docker 自身只能运行在 Linux 平台上，但启动和运行的性能都比虚拟机要快，往往更适合快速开发和部署应用的场景。

简单说：Vagrant 适合用来管理虚拟机，而 Docker 适合用来管理应用环境。

开发环境中 Docker 和 Vagrant 该如何选择?

答：Docker 不是虚拟机，而是进程隔离，对于资源的消耗很少，但是目前需要 Linux 环境支持。Vagrant 是虚拟机上做的封装，虚拟机本身会消耗资源。

如果本地使用的 Linux 环境，推荐都使用 Docker。

如果本地使用的是 macOS 或者 Windows 环境，那就需要开虚拟机，单一开发环境下 Vagrant 更简单；多环境开发下推荐在 Vagrant 里面再使用 Docker 进行环境隔离。

Built with mdPress

其它

Docker 能在非 Linux 平台上运行么？比如 Windows 或 macOS

答：完全可以。安装方法请查看安装 Docker 一节

如何将一台宿主主机的 Docker 环境迁移到另外一台宿主主机?

答：停止 Docker 服务。将整个 Docker 存储文件夹复制到另外一台宿主主机，然后调整另外一台宿主主机的配置即可。

如何进入 Docker 容器的网络命名空间?

答：Docker 在创建容器后，删除了宿主主机上 `/var/run/netns` 目录中的相关的网络命名空间文件。因此，在宿主主机上是无法看到或访问容器的网络命名空间的。

用户可以通过如下方法来手动恢复它。

首先，使用下面的命令查看容器进程信息，比如这里的 1234。

```
$ docker inspect --format='{{ .State.Pid }}' $container_id
1234
```

接下来，在 `/proc` 目录下，把对应的网络命名空间文件链接到 `/var/run/netns` 目录。

```
$ sudo ln -s /proc/1234/ns/net /var/run/netns/
```

然后，在宿主主机上就可以看到容器的网络命名空间信息。例如

```
$ sudo ip netns show
1234
```

此时，用户可以通过正常的系统命令来查看或操作容器的命名空间了。例如修改容器的 IP 地址信息为 `172.17.0.100/16`。

```
$ sudo ip netns exec 1234 ifconfig eth0 172.17.0.100/16
```

如何获取容器绑定到本地那个 veth 接口上?

答：Docker 容器启动后，会通过 veth 接口对连接到本地网桥，veth 接口命名跟容器命名毫无关系，十分难以找到对应关系。

最简单的一种方式是通过查看接口的索引号，在容器中执行 `ip a` 命令，查看到本地接口最前面的接口索引号，如 `205`，将此值加上 `1`，即 `206`，然后在本地主机执行 `ip a` 命令，查找接口索引号为 `206` 的接口，两者即为连接的 veth 接口对。

Built with mdPress

常见错误处理

常见错误速查表

错误信息 / 现象	可能原因	解决方案
`Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?`	Docker 服务未启动	Linux: `sudo systemctl start docker` Mac/Win: 启动 Docker Desktop
`permission denied while trying to connect to the Docker daemon socket`	当前用户无权访问 Docker socket	临时使用 `sudo docker ...`，或确认风险后执行 `sudo usermod -aG docker $USER` 并重新登录；`docker` 用户组等同授予 root 级权限，安全要求较高的环境优先评估 Rootless mode
`manifest for ... not found: manifest unknown`	镜像 tag 不存在	检查 Docker Hub 该镜像是否存在该 tag，或拼写是否正确
`connection refused` (pull image)	网络不通或镜像源无法访问	检查网络，配置镜像加速器
`Bind for 0.0.0.0:8080 failed: port is already allocated`	端口被占用	检查占用端口的进程 (`lsof -i:8080`) 并杀掉，或换个端口映射 (`-p 8081:80`)
`exec user process caused "exec format error"`	架构不匹配 (如在 x86 上跑 ARM 镜像)	使用 `docker buildx` 构建多架构镜像，或拉取对应架构的镜像
`standard_init_linux.go:211: exec user process caused`	找不到解释器或依赖库	检查 `ENTRYPOINT`/`CMD` 脚本开头的 shebang (`#!/bin/sh` vs

错误信息 / 现象	可能原因	解决方案
`"no such file or directory"`		`#!/bin/bash`)，或确认二进制文件是否依赖缺失 (Alpine 常见缺少 glibc)
`iptables: No chain/target/match by that name`	防火墙规则缺失或冲突	重启 Docker 服务重置 iptables 链: `sudo systemctl restart docker`
容器内无法访问外网	DNS 配置或转发问题	检查 `/etc/docker/daemon.json` 中的 DNS 配置

附录二：热门镜像介绍

本附录介绍常用官方镜像的用途、基本用法与适用场景，便于快速查阅。

目录

- Ubuntu：Ubuntu 官方镜像的使用方法。
- CentOS：CentOS 镜像与替代发行版的说明。
- Nginx：Nginx 官方镜像的使用方法。
- PHP：PHP 官方镜像的使用方法。
- Node.js：Node.js 官方镜像的使用方法。
- MySQL：MySQL 官方镜像的使用方法。
- WordPress：WordPress 官方镜像的使用方法。
- MongoDB：MongoDB 官方镜像的使用方法。
- Redis：Redis 官方镜像的使用方法。
- MinIO：MinIO 官方镜像的使用方法。

Ubuntu

基本信息

Ubuntu 是流行的 Linux 发行版，其自带软件版本往往较新一些。

该仓库位于 Docker Hub 的 Ubuntu 官方镜像页。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

默认会启动一个最小化的 Ubuntu 环境。

```
$ docker run --name some-ubuntu -it ubuntu:24.04
root@523c70904d54:/#
```

Dockerfile

请到 Ubuntu 官方镜像文档目录 查看。

Built with mdPress

CentOS

基本信息

CentOS 是流行的 Linux 发行版，其软件包大多跟 RedHat 系列保持一致。

> ⚠️ **重要提示**：CentOS 8 已于 2021 年 12 月 31 日停止维护 (EOL)，CentOS 7 也已于 2024 年 6 月 30 日 **完全结束支持**。Docker Hub 上的 CentOS 官方镜像 **已停止更新** 且存在未修复的安全漏洞。
>
> 2026 年了，对于任何新项目，**强烈建议** 使用以下生产级替代方案：
>
> - Rocky Linux：CentOS 原创始人发起的社区驱动项目，目前主流为 Rocky Linux 9。
> - AlmaLinux：由 CloudLinux 支持的企业级发行版，提供长期支持。
> - CentOS Stream：RHEL 的上游开发分支，镜像已迁移至 Quay.io (适合开发测试，不建议用于生产环境)。

该仓库位于 Docker Hub 的 CentOS 官方镜像页，提供了 CentOS 从 5 ~ 8 各个版本的镜像（仅作为历史归档，不再更新）。

使用方法

使用 Rocky Linux 9 替代 (**推荐**)：

```
$ docker run --name rocky -it rockylinux:9 bash
```

使用旧版 CentOS 7 (**仅用于维护旧项目，不推荐**)：

```
$ docker run --name centos -it centos:7 bash
```

Dockerfile

请到 CentOS 官方镜像文档目录 查看。

Nginx

基本信息

Nginx 是开源的高效的 Web 服务器实现，支持 HTTP、HTTPS、SMTP、POP3、IMAP 等协议。

该仓库位于 Docker Hub 的 Nginx 官方镜像页。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

下面的命令将作为一个静态页面服务器启动。

```
$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx
```

用户也可以不使用这种映射方式，通过利用 Dockerfile 来直接将静态页面内容放到镜像中，内容为

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

之后生成新的镜像，并启动一个容器。

```
$ docker build -t some-content-nginx .
$ docker run --name some-nginx -d some-content-nginx
```

开放端口，并映射到本地的 `8080` 端口。

```
$ docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

Nginx 的默认配置文件路径为 `/etc/nginx/nginx.conf`，可以通过映射它来使用本地的配置文件，例如

```
$ docker run -d \
    --name some-nginx \
    -p 8080:80 \
    -v /path/nginx.conf:/etc/nginx/nginx.conf:ro \
    nginx
```

Dockerfile

请到 Nginx 官方镜像文档目录 查看。

PHP

基本信息

PHP (Hypertext Preprocessor 超文本预处理器的字母缩写) 是一种被广泛应用的开放源代码的多用途脚本语言，它可嵌入到 HTML 中，尤其适合 web 开发。

该仓库位于 Docker Hub 的 PHP 官方镜像页。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

下面的命令将运行一个已有的 PHP 脚本。

```
$ docker run -it --rm -v "$PWD":/app -w /app php:alpine php your-script.php
```

Dockerfile

请到 PHP 官方镜像文档目录 查看。

Node.js

基本信息

Node.js 是基于 JavaScript 的可扩展服务端和网络软件开发平台。

该仓库位于 `https://hub.docker.com/_/node/`。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

在项目中创建一个 Dockerfile。

```
FROM node:22

## replace this with your application's default port

EXPOSE 8888
```

然后创建镜像，并启动容器。

```
$ docker build -t my-nodejs-app
$ docker run -it --rm --name my-running-app my-nodejs-app
```

也可以直接运行一个简单容器。

```
$ docker run -it --rm \
    --name my-running-script \
    # -v "$ ":/usr/src/myapp \

    --mount type=bind,src="$(pwd)",target=/usr/src/myapp \
    -w /usr/src/myapp \
    node:22-alpine \
    node your-daemon-or-script.js
```

Dockerfile

请到 Node 官方镜像文档目录 查看。

MySQL

基本信息

MySQL 是开源的关系数据库实现。

该仓库位于 Docker Hub 的 MySQL 官方镜像页。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

默认会在 `3306` 端口启动数据库。

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
```

之后就可以使用其它应用来连接到该容器。

首先创建网络

```
$ docker network create my-mysql-net
```

然后启动 MySQL 容器

```
$ docker run --name some-mysql -d --network my-mysql-net -e MYSQL_ROOT_PASSWORD=mysecretpassword mys
ql
```

最后启动应用容器

```
$ docker run --name some-app -d --network my-mysql-net application-that-uses-mysql
```

或者通过 `mysql` 命令行连接。

```
$ docker run -it --rm \
    --network my-mysql-net \
    mysql \
    sh -c 'exec mysql -hsome-mysql -P3306 -uroot -pmysecretpassword'
```

Dockerfile

请到 MySQL 官方镜像文档目录 查看。

WordPress

基本信息

WordPress 是开源的 Blog 和内容管理系统框架，它基于 PHP 和 MySQL。

该仓库位于 `https://hub.docker.com/_/wordpress/`。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

启动容器需要 MySQL 的支持，默认端口为 `80`。

首先创建网络

```
$ docker network create my-wordpress-net
```

启动 MySQL 容器

```
$ docker run --name some-mysql -d --network my-wordpress-net -e MYSQL_ROOT_PASSWORD=mysecretpassword
mysql
```

启动 WordPress 容器

```
$ docker run --name some-wordpress -d --network my-wordpress-net -e WORDPRESS_DB_HOST=some-mysql -e
WORDPRESS_DB_PASSWORD=mysecretpassword wordpress
```

启动 WordPress 容器时可以指定的一些环境变量包括：

- `WORDPRESS_DB_HOST`：MySQL 服务的主机名
- `WORDPRESS_DB_USER`：MySQL 数据库的用户名
- `WORDPRESS_DB_PASSWORD`：MySQL 数据库的密码
- `WORDPRESS_DB_NAME`：WordPress 要使用的数据库名

Dockerfile

请到 WordPress 官方镜像文档目录 查看。

MongoDB

基本信息

MongoDB 是开源的 NoSQL 数据库实现。

该仓库位于 `https://hub.docker.com/_/mongo/`。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

默认会在 `27017` 端口启动数据库。

```
$ docker run --name mongo -d mongo
```

使用其他应用连接到容器，首先创建网络

```
$ docker network create my-mongo-net
```

然后启动 MongoDB 容器

```
$ docker run --name some-mongo -d --network my-mongo-net mongo
```

最后启动应用容器

```
$ docker run --name some-app -d --network my-mongo-net application-that-uses-mongo
```

或者通过 `mongosh`（MongoDB 6.0+ 已移除旧版 `mongo` 命令行工具）

```
$ docker run -it --rm \
    --network my-mongo-net \
    mongo \
    mongosh "some-mongo:27017/test"
```

Dockerfile

请到 Mongo 官方镜像文档目录 查看。

Redis

基本信息

Redis 是开源的内存 Key-Value 数据库实现。

该仓库位于 Docker Hub 的 Redis 官方镜像页。具体可用版本以 Docker Hub 上的 tags 列表为准。

使用方法

默认会在 6379 端口启动数据库。

```
$ docker run --name some-redis -d -p 6379:6379 redis
```

另外还可以启用持久存储。

```
$ docker run --name some-redis -d -p 6379:6379 redis redis-server --appendonly yes
```

默认数据存储位置在 VOLUME/data。可以使用 --volumes-from some-volume-container 或 -v /docker/host/dir:/data 将数据存放到本地。

使用其他应用连接到容器，首先创建网络

```
$ docker network create my-redis-net
```

然后启动 redis 容器

```
$ docker run --name some-redis -d --network my-redis-net redis
```

最后启动应用容器

```
$ docker run --name some-app -d --network my-redis-net application-that-uses-redis
```

或者通过 redis-cli

```
$ docker run -it --rm \
    --network my-redis-net \
    redis \
    sh -c 'exec redis-cli -h some-redis'
```

Dockerfile

请到 Redis 官方镜像文档目录 查看。

Minio

MinIO 是一个基于 Apache License v2.0 开源协议的对象存储服务。它兼容亚马逊 S3 云存储服务接口，非常适合于存储大容量非结构化的数据，例如图片、视频、日志文件、备份数据和容器/虚拟机镜像等，而一个对象文件可以是任意大小，从几 kb 到最大 5T 不等。

MinIO 是一个非常轻量的服务，可以很简单的和其他应用的结合，类似 NodeJS，Redis 或者 MySQL。

官方文档

简单使用

测试、开发环境下不考虑数据存储的情况下可以使用下面的命令快速开启服务。

```
$ docker run -d -p 9000:9000 -p 9090:9090 minio/minio server /data --console-address ':9090'
```

离线部署

许多生产环境是一般是没有公网资源的，这就需要从有公网资源的服务器上把镜像导出，然后导入到需要运行镜像的内网服务器。

导出镜像

在有公网资源的服务器上下载好 `minio/minio` 镜像

```
$ docker save -o minio.tar minio/minio:latest
```

> 使用 docker save 的时候，也可以使用 image id 来导出，但是那样导出的时候，就会丢失原来的镜像名称，推荐，还是使用镜像名字+tag 来导出镜像

导入镜像

把压缩文件复制到内网服务器上，使用下面的命令导入镜像

```
$ docker load -i minio.tar
```

运行 minio

- 把 `/mnt/data` 改成要替换的数据目录
- 替换 `MINIO_ROOT_USER` 的值
- 替换 `MINIO_ROOT_PASSWORD` 的值
- 替换 name,minio1 (可选)
- 如果 9000、9090 端口冲突，替换端口前面的如 `9009:9000`

```
$ sudo docker run -d -p 9000:9000 -p 9090:9090 --name minio1 \
  -e "MINIO_ROOT_USER=改成自己需要的" \
  -e "MINIO_ROOT_PASSWORD=改成自己需要的" \
  -v /mnt/data:/data \
  --restart=always \
  minio/minio server /data --console-address ':9090'
```

访问 web 管理页面

打开 `http://<server-ip>:9090` 访问 Web 控制台。

附录三：Docker 命令查询

基本语法

Docker 命令有两大类，客户端命令和服务端命令。前者是主要的操作接口，后者用来启动 Docker Daemon。

- 客户端命令：基本命令格式为 `docker [OPTIONS] COMMAND [arg...]`；
- 服务端命令：基本命令格式为 `dockerd [OPTIONS]`。

可以通过 `man docker` 或 `docker help` 来查看这些命令。

接下来的小节对这两个命令进行介绍。

Built with mdPress

客户端命令 - docker

客户端命令选项

- `--config=""`：指定客户端配置文件，默认为 `~/.docker`；
- `-D=true|false`：是否使用 debug 模式。默认不开启；
- `-H, --host=[]`：指定命令对应 Docker 守护进程的监听接口，可以为 unix 套接字 `unix:///path/to/socket`，文件句柄 `fd://socketfd` 或 tcp 套接字 `tcp://[host[:port]]`，默认为 `unix:///var/run/docker.sock`；
- `-l, --log-level="debug|info|warn|error|fatal"`：指定日志输出级别；
- `--tls=true|false`：是否对 Docker 守护进程启用 TLS 安全机制，默认为否；
- `--tlscacert=/.docker/ca.pem`：TLS CA 签名的可信证书文件路径；
- `--tlscert=/.docker/cert.pem`：TLS 可信证书文件路径；
- `--tlskey=/.docker/key.pem`：TLS 密钥文件路径；
- `--tlsverify=true|false`：启用 TLS 校验，默认为否。

客户端命令

可以通过 `docker COMMAND --help` 来查看这些命令的具体用法。

- `attach`：依附到一个正在运行的容器中；
- `build`：从一个 Dockerfile 创建一个镜像；
- `commit`：从一个容器的修改中创建一个新的镜像；
- `cp`：在容器和本地宿主系统之间复制文件中；
- `create`：创建一个新容器，但并不运行它；
- `diff`：检查一个容器内文件系统的修改，包括修改和增加；
- `events`：从服务端获取实时的事件；
- `exec`：在运行的容器内执行命令；
- `export`：导出容器内容为一个 `tar` 包；
- `history`：显示一个镜像的历史信息；
- `images`：列出存在的镜像；
- `import`：导入一个文件 (典型为 `tar` 包) 路径或目录来创建一个本地镜像；
- `info`：显示一些相关的系统信息；
- `inspect`：显示一个容器的具体配置信息；
- `kill`：关闭一个运行中的容器 (包括进程和所有相关资源)；
- `load`：从一个 tar 包中加载一个镜像；
- `login`：注册或登录到一个 Docker 的仓库服务器；
- `logout`：从 Docker 的仓库服务器登出；
- `logs`：获取容器的 log 信息；
- `network`：管理 Docker 的网络，包括查看、创建、删除、挂载、卸载等；
- `node`：管理 swarm 集群中的节点，包括查看、更新、删除、提升/取消管理节点等；
- `pause`：暂停一个容器中的所有进程；
- `port`：查找一个 nat 到一个私有网口的公共口；
- `ps`：列出主机上的容器；
- `pull`：从一个 Docker 的仓库服务器下拉一个镜像或仓库；
- `push`：将一个镜像或者仓库推送到一个 Docker 的注册服务器；
- `rename`：重命名一个容器；

Built with mdPress

- `restart`：重启一个运行中的容器；
- `rm`：删除给定的若干个容器；
- `rmi`：删除给定的若干个镜像；
- `run`：创建一个新容器，并在其中运行给定命令；
- `save`：保存一个镜像为 tar 包文件；
- `search`：在 Docker index 中搜索一个镜像；
- `service`：管理 Docker 所启动的应用服务，包括创建、更新、删除等；
- `start`：启动一个容器；
- `stats`：输出 (一个或多个) 容器的资源使用统计信息；
- `stop`：终止一个运行中的容器；
- `swarm`：管理 Docker swarm 集群，包括创建、加入、退出、更新等；
- `tag`：为一个镜像打标签；
- `top`：查看一个容器中的正在运行的进程信息；
- `unpause`：将一个容器内所有的进程从暂停状态中恢复；
- `update`：更新指定的若干容器的配置信息；
- `version`：输出 Docker 的版本信息；
- `volume`：管理 Docker volume，包括查看、创建、删除等；
- `wait`：阻塞直到一个容器终止，然后输出它的退出符。

一张图总结 Docker 的命令

如图 16-1 所示，Docker 常用客户端命令可按功能分组理解。

Docker 命令总结

图 A-1：Docker 客户端命令分类示意图

参考

- 官方文档

服务端命令 - dockerd

使用说明

`dockerd` 参数会随版本变化。建议优先在目标机器上执行 `dockerd --help`，并以 `daemon.json` 为主进行持久化配置。

常用选项：Docker Engine 29.x

- `--config-file="/etc/docker/daemon.json"`：指定 daemon 配置文件路径；
- `--data-root=""`：Docker 数据目录 (默认 `/var/lib/docker`)；
- `-H, --host=[]`：指定 daemon 监听地址 (Unix socket / TCP)；
- `-D, --debug`：开启调试日志；
- `-l, --log-level="debug|info|warn|error|fatal"`：日志级别；
- `--group=""`：Unix socket 所属用户组 (默认 `docker`)；
- `--containerd=""`：指定 containerd socket；
- `--exec-opt=[]`：运行时执行选项 (如 cgroup 驱动)；
- `--default-ulimit=[]`：设置容器默认 ulimit；
- `--dns=[]` / `--dns-search=[]` / `--dns-opt=[]`：DNS 配置；
- `--registry-mirror=[]`：镜像加速地址；
- `--insecure-registry=[]`：允许访问不安全仓库；
- `--iptables=true|false` / `--ip-forward=true|false` / `--ip-masq=true|false`：网络转发与 NAT 规则控制；
- `--ipv6=true|false`：启用 IPv6；
- `--storage-driver=""` / `--storage-opt=[]`：存储驱动及参数；
- `--log-driver=""` / `--log-opt=[]`：容器日志驱动与参数；
- `--authorization-plugin=[]`：鉴权插件；
- `--selinux-enabled=true|false`：启用 SELinux 集成 (依赖发行版策略)；
- `--userns-remap=...`：用户命名空间映射；
- `--tls` / `--tlscacert` / `--tlscert` / `--tlskey` / `--tlsverify`：TLS 安全配置。

历史参数提示

以下参数已移除或不建议继续使用：

- `--graph`：请改用 `--data-root`；
- `--cluster-store` / `--cluster-advertise` / `--cluster-store-opt`：已移除；
- `--disable-legacy-registry`：已移除。

参考

- 官方文档

Built with mdPress

附录四：Dockerfile 最佳实践

本附录是笔者对 Docker 官方文档中 Best practices for writing Dockerfiles 的理解与翻译。

一般性的指南和建议

容器应该是短暂的

通过 `Dockerfile` 构建的镜像所启动的容器应该尽可能短暂 (生命周期短)。“短暂” 意味着可以停止和销毁容器，并且创建一个新容器并部署好所需的设置和配置工作量应该是极小的。

使用 `.dockerignore` 文件

使用 `Dockerfile` 构建镜像时最好是将 `Dockerfile` 放置在一个新建的空目录下。然后将构建镜像所需要的文件添加到该目录中。为了提高构建镜像的效率，你可以在目录下新建一个 `.dockerignore` 文件来指定要忽略的文件和目录。`.dockerignore` 文件的排除模式语法和 Git 的 `.gitignore` 文件相似。

使用多阶段构建

在现代 Docker 版本中，你可以使用多阶段构建来减少所构建镜像的大小。该能力最早在 Docker 17.05 引入，如今已是编写生产镜像的默认实践之一；新项目还应结合 BuildKit 缓存挂载、secret 挂载和多平台构建能力一起评估。

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、节约构建时间，你应该避免安装任何不必要的包。例如，不要在数据库镜像中包含一个文本编辑器。

一个容器只运行一个进程

应该保证在一个容器中只运行一个进程。将多个应用解耦到不同容器中，保证了容器的横向扩展和复用。例如 web 应用应该包含三个容器：web 应用、数据库、缓存。

如果容器互相依赖，你可以使用 Docker 自定义网络来把这些容器连接起来。

镜像层数尽可能少

你需要在 `Dockerfile` 可读性 (也包括长期的可维护性) 和减少层数之间做一个平衡。

将多行参数排序

将多行参数按字母顺序排序 (比如要安装多个包时)。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。也便于 PRs 阅读和审查。建议在反斜杠符号 \ 之前添加一个空格，以增加可读性。

下面是来自 buildpack-deps 镜像的例子：

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

构建缓存

在镜像的构建过程中，Docker 会遍历 Dockerfile 文件中的指令，然后按顺序执行。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。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。

但是，如果你想在构建的过程中使用缓存，你得明白什么时候会，什么时候不会找到匹配的镜像，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。
- 在大多数情况下，只需要简单地对比 Dockerfile 中的指令和子镜像。然而，有些指令需要更多的检查和解释。
- 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和。文件的最后修改时间和最后访问时间不会纳入校验。在缓存的查找过程中，会将这些校验和与已存在镜像中的文件校验和进行对比。如果文件有任何改变，比如内容和元数据，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。

Dockerfile 指令

下面针对 `Dockerfile` 中各种指令的最佳编写方式给出建议。

FROM

尽可能使用当前官方仓库作为你构建镜像的基础。推荐使用 Alpine 镜像，因为它被严格控制并保持最小尺寸 (目前小于 5 MB)，但它仍然是一个完整的发行版。

LABEL

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：如果你的字符串中包含空格，必须将字符串放入引号中或者对空格使用转义。如果字符串内容本身就包含引号，必须对引号使用转义。

```
## 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，但建议将多个标签放入到一个 LABEL 指令中。

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

LABEL vendor=ACME\ Incorporated \
      com.example.is-beta= \
      com.example.is-production="" \
      com.example.version="0.0.1-beta" \
      com.example.release-date="2015-02-12"
```

关于标签可以接受的键值对，参考 Understanding object labels。关于查询标签信息，参考 Managing labels on objects。

RUN

为了保持 `Dockerfile` 文件的可读性，可理解性，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。

apt-get

`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`，因为许多基础镜像中的 “必须” 包不会在一个非特权容器中升级。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:24.04

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:24.04

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`。你也可以显示指定一个包的版本号来达到 `cache-busting`，这就是所谓的固定版本，例如：

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo=1.3.*
```

固定版本会迫使构建过程检索特定的版本，而不管缓存中有什么。这项技术也可以减少因所需包中未预料到的变化而导致的失败。

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 \
    git \
    redis-server \
 && rm -rf /var/lib/apt/lists/*
```

其中 `redis-server` 是示例包。确保安装的是最新版本。

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 update`，包缓存总是会在 `apt-get install` 之前刷新。

> 注意：官方的 Debian 和 Ubuntu 镜像会自动运行 apt-get clean，所以不需要显式的调用 apt-get clean。

CMD

`CMD` 指令用于执行目标镜像中包含的软件，可以包含参数。`CMD` 大多数情况下都应该以 `CMD ['executable', 'param1', 'param2'...]` 的形式使用。因此，如果创建镜像的目的是为了部署某个服务 (比如 `Apache`)，你可能会执行类似于 `CMD ['apache2', '-DFOREGROUND']` 形式的命令。我们建议任何服务镜像都使用这种形式的命令。

多数情况下，`CMD` 都需要一个交互式的 `shell` (bash，Python，perl 等)，例如 `CMD ['perl', '-de0']`，或者 `CMD ['PHP', '-a']`。使用这种形式意味着，当你执行类似 `docker run -it python`

时，你会进入一个准备好的 shell 中。CMD 应该在极少的情况下才能以 CMD ['param', 'param'] 的形式与 ENTRYPOINT 协同使用，除非你和你的镜像使用者都对 ENTRYPOINT 的工作方式十分熟悉。

EXPOSE

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 时使用一个标志来指示如何将指定的端口映射到所选择的端口。

ENV

为了方便新程序运行，你可以使用 ENV 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/postgres && …

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。

ADD 和 COPY

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`。

ENTRYPOINT

`ENTRYPOINT` 的最佳用处是设置镜像的主命令，允许将镜像当成命令本身来运行 (用 `CMD` 提供默认选项)。

例如，下面的示例镜像提供了命令行工具 `s3cmd`：

```
ENTRYPOINT ["s3cmd"]

CMD ["--help"]
```

现在直接运行该镜像创建的容器会显示命令帮助：

```
$ docker run s3cmd
```

或者提供正确的参数来执行某个命令：

```
$ docker run s3cmd ls s3://mybucket
```

这样镜像名可以当成命令行的参考。

ENTRYPOINT 指令也可以结合一个辅助脚本使用，和前面命令行风格类似，即使启动工具需要不止一个步骤。

例如，Postgres 官方镜像使用下面的脚本作为 ENTRYPOINT：

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

> 注意：该脚本使用了 Bash 的内置命令 exec，所以最后运行的进程就是容器的 PID 为 1 的进程。这样，进程就可以接收到任何发送给容器的 Unix 信号了。

该辅助脚本被拷贝到容器，并在容器启动时通过 ENTRYPOINT 执行：

```
COPY ./docker-entrypoint.sh /

ENTRYPOINT ["/docker-entrypoint.sh"]
```

该脚本可以让用户用几种不同的方式和 Postgres 交互。

你可以很简单地启动 Postgres：

```
$ docker run postgres
```

也可以执行 Postgres 并传递参数：

```
$ docker run postgres postgres --help
```

最后，你还可以启动另外一个完全不同的工具，比如 Bash：

```
$ docker run --rm -it postgres bash
```

VOLUME

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

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，你应该显式的指定一个 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。

最后，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

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。

官方镜像示例

这些官方镜像的 Dockerfile 都是参考典范，详见 docker-library/docs。

附录五：如何调试 Docker

开启 Debug 模式

在 dockerd 配置文件 daemon.json (默认位于 /etc/docker/) 中添加

```
{
  "debug": true
}
```

重启守护进程。

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

此时 dockerd 会在日志中输出更多信息供分析。

检查内核日志

```
$ sudo dmesg |grep dockerd
$ sudo dmesg |grep runc
```

Docker 不响应时处理

可以杀死 dockerd 进程查看其堆栈调用情况。

```
$ sudo kill -SIGUSR1 $(pidof dockerd)
```

重置 Docker 本地数据

注意，本操作会移除所有的 Docker 本地数据，包括镜像和容器等。

更安全的替代方式是优先使用以下命令进行清理：

```
$ docker system prune
```

如果你只是想 “恢复出厂设置”，在 Docker Desktop 里也提供了相应入口。

只有在确认要丢弃本机全部镜像、容器、卷和构建缓存，并且已经停止 Docker 服务、完成备份或导出后，才考虑删除 Docker 数据目录：

Built with mdPress

```
$ sudo systemctl stop docker containerd
$ sudo cp -a /var/lib/docker /var/lib/docker.backup
$ sudo rm -rf /var/lib/docker
$ sudo systemctl start docker
```

常见故障排查

容器启动失败

如果容器启动后立即退出，可以使用 `docker logs` 查看原因。

- **Exit Code 1**：应用程序错误。通常是配置错误或依赖缺失。
- **Exit Code 137**：OOM (Out Of Memory)。容器内存不足被内核杀掉。
  - 检查宿主机内存。
  - 调整容器内存限制 (`--memory`)。
- **Exit Code 127**：命令未找到。可能是 `ENTRYPOINT` 或 `CMD` 指定的命令不存在。

网络连接问题

容器内部无法联网

1. 检查 Docker DNS 配置 (`/etc/docker/daemon.json`)。
2. 检查宿主机防火墙 (iptables/firewalld) 是否拦截了转发。
3. 容器内测试：`ping 8.8.8.8` (测试连通性)，`nslookup google.com` (测试 DNS)。

端口映射不通

1. 检查容器端口是否正确监听：`netstat -tunlp` (宿主机) 或 `docker exec <container> netstat -tunlp`。
2. 确认应用监听地址是 `0.0.0.0` 而不是 `127.0.0.1`。
   - 如果应用监听在 `127.0.0.1`，只有容器内部能访问，映射到宿主机外部也无法被外部请求访问。

Built with mdPress

镜像拉取失败

- **connection refused**：检查网络或代理设置。
- **image not found**：检查镜像名称和 Tag 拼写。
- **EOF / timeout**：网络不稳定，尝试配置镜像加速器。

附录六：资源链接

本页只保留适合作为一手核验入口的官方资源，便于查版本、查命令、查发布说明和查最佳实践。

官方入口

- Docker 文档
- Docker 入门指南
- Docker 发行说明
- Docker Desktop 发行说明
- Docker Compose 安装
- Docker Hub 文档
- Docker Blog
- Docker Roadmap
- Kubernetes 文档
- Kubernetes 发布页
- kubeadm 安装文档
- containerd 文档
- GitHub Actions 文档
- Drone 文档

参考文档

- Docker CLI 参考
- Dockerfile 参考
- Docker 构建最佳实践
- Docker 远端应用 API
- Docker 存储文档
- Docker 网络文档

Built with mdPress

源码仓库

- Moby 源代码仓库

附录七：术语表

本附录整理了本书中常见的一些专业术语及其解释。

A

- **Alpine**：一个轻量级的 Linux 发行版，常作为基础镜像用于构建体积较小的 Docker 镜像。
- **API (Application Programming Interface)**：应用程序编程接口，Docker Daemon 提供 RESTful API 供客户端或外部程序与之交互。

B

- **Base Image (基础镜像)**：没有父镜像的镜像，通常是操作系统的最小安装集合（如 `ubuntu` 或 `alpine`）。
- **BuildKit**：Docker 下一代的构建引擎，提供了更高的构建性能、更好的缓存处理和并发构建支持。
- **Buildx**：Docker CLI 的一个插件，扩展了构建功能，支持 BuildKit 的所有高级特性，例如多系统架构镜像构建。

C

- **Cgroups (Control Groups)**：控制组，Linux 内核特性，用于限制、记录、隔离进程组使用的物理资源（如 CPU、内存、磁盘 I/O 等）。
- **Cluster (集群)**：一组协同工作的节点（如主机、虚拟机等），在容器领域常指 Kubernetes 集群。
- **Compose (Docker Compose)**：用于定义和运行多容器 Docker 应用程序的工具，通过 YAML 文件配置应用服务。
- **Container (容器)**：镜像的运行实例，带有额外的可写文件层，具有独立性。
- **Containerd**：行业标准的容器运行时，核心功能是管理宿主机上容器的生命周期（创建、启动、停止、销毁）。

Built with mdPress

D

- **Daemon (守护进程)**：Docker 的后台守护进程，负责接收和处理 Docker API 请求，并管理镜像、容器、网络和数据卷等对象。
- **Docker**：开源的应用容器引擎，让开发者可以打包应用程序及其依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 或 Windows 机器上。
- **Docker Desktop**：包含 Docker Engine、Docker CLI 客户端、Docker Compose 和 Kubernetes 等的桌面应用程序，适用于 macOS 和 Windows。
- **Docker Hub**：Docker 官方的公共镜像仓库服务，提供容器镜像的存储和分发。
- **Dockerfile**：包含用于组合镜像的命令的文本文件，Docker 通过读取 `Dockerfile` 中的指令即可自动完成镜像构建。

E

- **Etcd**：一个高可用、强一致性的分布式键值存储系统，常用于容器集群（如 Kubernetes）的服务发现和状态配置管理。

I

- **Image (镜像)**：Docker 镜像是一个只读模板，带有创建 Docker 容器的说明。

K

- **Kubernetes (K8s)**：开源的容器编排引擎，用于自动化容器化应用程序的部署、扩展和管理。

L

- **Layer (镜像层)**：Docker 镜像由多个只读层叠合而成，每一层通常代表 Dockerfile 中的一条指令的操作结果，通过联合文件系统（UFS）叠加在一起形成完整的文件系统。

M

- **Multistage Build (多阶段构建)**：Dockerfile 中的特性，允许在同一个 Dockerfile 中使用多个 `FROM` 语句，从一个阶段复制所需的构建产物到另一个阶段，从而大幅减小最终镜像的体积。

Built with mdPress

N

- **Namespace (命名空间)**：Linux 内核特性，用于隔离各种系统资源，如进程、网络、挂载点等，使容器看起来就像是一个独立的操作系统。
- **Node (节点)**：容器集群（如 Kubernetes）中的一台工作机器，可以是物理机或虚拟机。

O

- **OCI (Open Container Initiative)**：开放容器规范，由多家行业领头企业共同制定的容器运行时和镜像格式的行业标准。
- **Orchestration (编排)**：自动化部署、管理、扩展和网络配置容器的系统和技术（如 Kubernetes）。

P

- **Pod**：Kubernetes 中最小的、可部署的计算单元，包含一个或多个紧密相关的容器，共享相同的网络命名空间和存储。
- **Prometheus**：开源的系统监控和告警工具包，广泛应用于云原生的监控体系中。

R

- **Registry (注册服务器)**：提供 Docker 镜像下载和上传等存储分发服务的服务器。
- **Repository (仓库)**：集中存放某个应用的所有镜像的地方，通常由镜像名定义。一个 Registry 中可以包含多个 Repository。

S

- **Swarm (Docker Swarm)**：Docker 原生的集群和编排管理工具，可将多个 Docker 主机组合成一个统一的虚拟 Docker 主机池。维护节点时通常将节点可用性设为 `Drain`，这只影响 Swarm service 调度，不会停止该节点上独立运行的容器。

U

- **UFS (Union File System)**：联合文件系统，一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改一层层叠加。

Built with mdPress

V

- **Volume (数据卷)**：专为绕过联合文件系统而设计的特殊目录，用于实现容器数据的持久化，或在多个容器之间提供文件共享。

附录八：Docker 学习路线图与知识体系

本附录为学习者提供清晰的学习路线、知识点依赖关系、认证指南和常见面试题，帮助快速成长为 Docker 和 DevOps 专家。

学习阶段划分

Docker 学习可分为四个递进阶段，每个阶段都有明确的学习目标和时间投入。

第一阶段：基础入门（0-2 周）

学习目标：

- 理解容器化的基本概念
- 能够运行、管理基本的容器
- 了解镜像和仓库的基本操作

核心内容：

```
Docker 简介
├── 为什么需要 Docker
├── 容器 vs 虚拟机 vs 云计算
└── Docker 的三大核心概念
    ├── 镜像（Image）
    ├── 容器（Container）
    └── 仓库（Repository）

基础命令
├── docker run / create / start / stop / rm
├── docker ps / logs / exec / inspect
├── docker pull / push / tag
└── docker build -t

Docker 安装配置
├── Linux 平台安装
├── macOS 和 Windows 安装
├── 镜像加速器配置
└── 权限和用户配置
```

学习资源：

- 官方教程
- 本书第 1-3 章：入门篇基础概念
- Docker CLI 参考

时间投入：

- 理论学习：3-4 小时
- 实操练习：8-10 小时
- 总计：2 周

验证学习成果：

```
# 完成以下任务说明基础入门完成
1. 运行官方 nginx 镜像，访问 http://localhost
2. 使用 docker exec 进入容器修改首页
3. 提交修改为新镜像
4. 推送镜像到 Docker Hub（需创建账户）
```

第二阶段：核心开发（2-6 周）

学习目标：

- 掌握 Dockerfile 编写
- 能够构建自己的应用镜像
- 理解数据管理和网络配置
- 熟悉 Docker Compose 编排

核心内容：

```
Dockerfile 指令详解
├── FROM / RUN / COPY / ADD
├── WORKDIR / ENV / ARG
├── EXPOSE / CMD / ENTRYPOINT
├── VOLUME / USER / HEALTHCHECK
└── 最佳实践和性能优化
    ├── 分层缓存机制
    ├── 减少镜像体积
    ├── 多阶段构建
    └── 安全最佳实践

容器数据管理
├── 数据卷（Volume）
│   ├── 命名卷
│   ├── 匿名卷
│   └── 卷挂载最佳实践
├── 绑定挂载（Bind Mount）
│   ├── 宿主机路径映射
│   └── 权限和隔离
└── tmpfs 挂载
    └── 临时文件系统

容器网络
├── 网络类型
│   ├── bridge（默认）
│   ├── host
│   ├── overlay
│   └── macvlan
├── 端口映射
├── 容器互联
├── DNS 配置
└── 自定义网络

Docker Compose
├── compose.yml/docker-compose.yml 编写
├── services 定义
├── volumes 配置
├── networks 配置
├── 依赖关系
├── 环境变量
└── 命令操作
    ├── up / down / ps / logs
    ├── exec / run
    └── build / push
```

学习资源：

- 本书第 4-11 章：进阶篇
- Docker 官方最佳实践
- Dockerfile 参考

时间投入：

- 理论学习：8-10 小时
- 实操练习：30-40 小时（多个实战项目）
- 总计：4-6 周

项目实战：

```
项目 1: Python Web 应用（Flask/Django）
- 编写多阶段 Dockerfile
- 使用 Compose 配置数据库
- 实现热重载开发环境

项目 2: Node.js 微服务
- 优化镜像大小
- 配置 Compose 多个服务
- 设置网络和环保境变量

项目 3: 数据库容器化
- PostgreSQL/MySQL 配置
- 数据持久化
- 备份恢复策略
```

第三阶段：生产优化（6-12 周）

学习目标：

- 掌握容器安全最佳实践
- 理解性能监控和优化
- 学会容器编排（Kubernetes 基础）
- 熟悉 CI/CD 集成

核心内容：

```
容器安全
├── 镜像安全
│   ├── 漏洞扫描（Trivy/Grype/Snyk）
│   ├── 镜像签名和验证（Cosign）
│   ├── SBOM 生成和管理
│   └── 供应链安全
├── 运行时安全
│   ├── 用户和权限
│   ├── Linux 能力机制
│   ├── AppArmor 和 SELinux
│   ├── Rootless 容器
│   └── 安全的 Docker socket 访问
└── 宿主机安全
    ├── API 访问控制
    ├── TLS 认证
    └── 审计日志

性能监控和优化
├── 监控指标体系
│   ├── CPU / 内存 / 网络 / I/O
│   └── 应用级指标
├── 监控工具
│   ├── docker stats
│   ├── cAdvisor
│   ├── Prometheus
│   └── Grafana
├── 性能优化
│   ├── 镜像大小优化
│   ├── 内存和 CPU 限制
│   ├── OOM 诊断和处理
│   └── 网络性能优化
└── 日志管理
    ├── 日志驱动配置
    ├── ELK Stack
    └── 日志聚合

容器编排基础
├── Kubernetes 核心概念
│   ├── Pod / Deployment / Service
│   ├── ConfigMap / Secret
│   └── 健康检查和自动恢复
├── 容器执行环境
│   ├── containerd
│   ├── CRI-O
│   └── Docker
├── 网络插件
│   ├── CNI 标准
│   ├── Calico / Flannel / Cilium
│   └── 网络策略
└── 存储和有状态应用
    ├── PV / PVC
    ├── StorageClass
    └── StatefulSet

CI/CD 集成
├── GitHub Actions
│   ├── 镜像构建和推送
│   ├── 安全扫描
│   └── 自动化测试
├── GitLab CI
```

```
├── Jenkins Docker 集成
└── Drone

生态工具
├── Buildx（多架构构建）
├── Skopeo（镜像管理）
├── Podman（替代方案）
├── Buildah（镜像构建）
└── Kollabot
```

学习资源：

- 本书第 12-21 章：深入篇和实战篇
- Kubernetes 官方文档
- CNCF 学习路线

时间投入：

- 理论学习：15-20 小时
- 实操练习：60-80 小时（多个生产级项目）
- 总计：6-12 周

项目实战：

```
项目 1: 安全镜像构建流程
- 集成 Trivy 扫描
- 镜像签名和验证
- 生成 SBOM 文档

项目 2: 完整监控栈
- 搭建 Prometheus + Grafana
- 配置告警规则
- 性能数据采集和分析

项目 3: CI/CD 流程
- GitHub Actions 或 GitLab CI 配置
- 自动化镜像构建
- 安全扫描和合规检查
- 自动化部署到 Kubernetes

项目 4: Kubernetes 集群部署
- 本地 K3s/Kind 集群
- 部署有状态应用
- 配置持久化存储
```

第四阶段：专家深造（12+ 周）

学习目标：

- 掌握 Kubernetes 高级特性
- 理解容器运行时底层实现
- 能够设计和优化大规模容器平台
- 贡献开源社区

核心内容：

```
Kubernetes 高级特性
├── 集群管理
│   ├── 节点管理和驱逐
│   ├── 集群自动扩缩容
│   └── 节点亲和性和污点容忍
├── 存储编排
│   ├── 动态存储配置
│   ├── 有状态应用管理（StatefulSet）
│   └── 备份和灾难恢复
├── 服务网格（Service Mesh）
│   ├── Istio / Linkerd / Cilium
│   ├── 流量管理
│   └── 可观测性增强
├── 安全和多租户
│   ├── RBAC（角色访问控制）
│   ├── Network Policy 深入
│   ├── Pod Security Policy
│   └── 准入控制器（Admission Controller）
└── 性能和扩展性
    ├── 大规模集群优化
    ├── 自定义 Operator
    └── 集群联邦

容器运行时底层
├── Linux 内核机制
│   ├── Namespace 详解
│   ├── Cgroup v1 和 v2
│   ├── OverlayFS 和 UnionFS
│   └── SELinux 和 AppArmor
├── 容器运行时
│   ├── containerd 源码阅读
│   ├── runc 实现
│   ├── gVisor 和 Kata
│   └── Firecracker
└── OCI 标准
    ├── Image Spec
    └── Runtime Spec

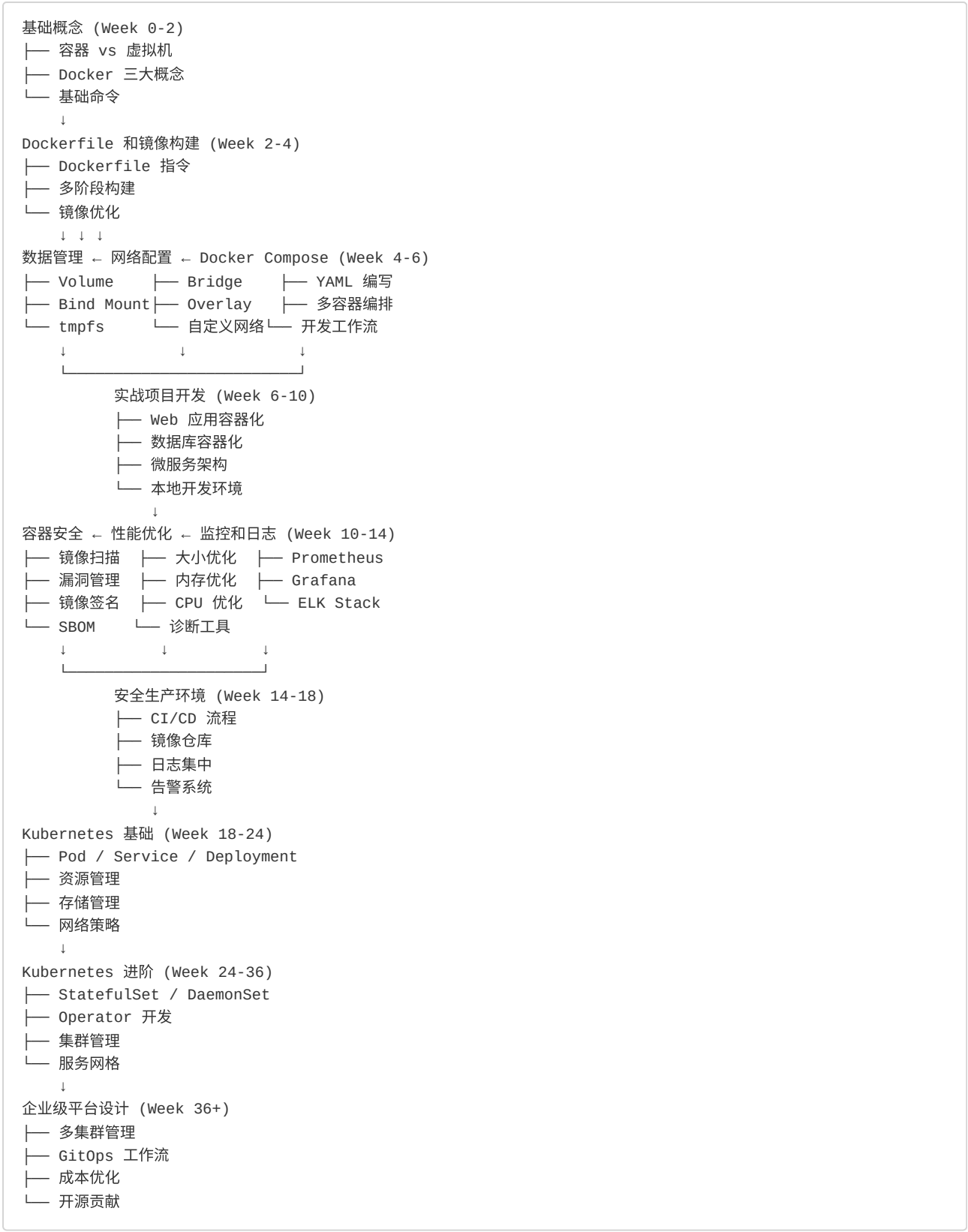
DevOps 工程化
├── 大规模集群管理
│   ├── Helm / Kustomize
│   ├── GitOps（Flux / ArgoCD）
│   └── 配置管理
├── 灾难恢复和高可用
│   ├── 多集群部署
│   ├── 故障转移
│   └── 备份策略
├── 成本优化
│   ├── 资源申请和限制
│   ├── 自动扩缩容
│   └── 成本监控
└── 团队协作
    ├── GitFlow 工作流
    ├── 代码审查
    └── 文档和最佳实践传播
```

贡献机会：

Built with mdPress

- Kubernetes
- Cilium
- Prometheus
- Docker/Moby

知识点依赖关系



Built with mdPress

推荐学习资源

官方文档

资源	URL	推荐程度
Docker 官方文档	docs.docker.com	⭐⭐⭐⭐⭐
Docker Hub	hub.docker.com	⭐⭐⭐⭐⭐
Kubernetes 官方	kubernetes.io/docs	⭐⭐⭐⭐⭐
CNCF 景观	landscape.cncf.io	⭐⭐⭐⭐

在线课程

- **Udemy**：Docker 和 Kubernetes 完整课程（70-100 小时）
- **Linux Academy**：Linux 和容器管理
- **A Cloud Guru**：AWS/Azure 容器服务
- **Pluralsight**：Docker 和容器生态系统

书籍推荐

- 《Docker 深入浅出》- 本书的原版
- 《Kubernetes 权威指南》- 深入 Kubernetes 的必读书
- 《容器技术核心技术与应用》- 理解底层实现
- 《SRE Google 运维之道》- 生产环境最佳实践

博客和社区

- Docker 官方博客
- Kubernetes 官方博客
- CNCF 博客
- DZone Cloud Architecture

Built with mdPress

认证指南

Docker 认证

Docker Certified Associate (DCA)

考试信息：

- 题目数：55 道
- 时间限制：90 分钟
- 及格分数：73%（约 41 道题）
- 费用：$199 USD
- 有效期：2 年

考试内容比例：

```
镜像和仓库（20%）
- 镜像构建和管理
- 镜像层和缓存
- 私有仓库配置

容器运行（15%）
- 容器生命周期
- 资源限制
- 容器隔离

网络（15%）
- 网络驱动
- 容器通信
- 端口映射

存储（10%）
- Volume 管理
- 数据持久化
- 绑定挂载

编排（20%）
- Docker Compose
- Docker Swarm 基础

安全（15%）
- 用户和权限
- 密钥管理
- 镜像安全
- 守护进程安全

和日志（5%）
- Logging drivers
- 事件处理
```

准备建议：

```
# 1. 学习本书第 1-11 章（基础到中级）
# 2. 完成 20+ 个实战项目
# 3. 参考官方学习指南
# 参考 KodeKloud DCA 认证指南：https://kodekloud.com/blog/docker-certified-associate-guide/

# 4. 模拟考试
- Linux Academy DCA 练习题
- Whizlabs DCA 模拟考试

# 5. 重点掌握的命令
docker build / push / pull / tag
docker run / exec / logs / inspect / ps
docker volume / network / service
docker compose up / down / logs / ps
docker stats / events / inspect
```

Kubernetes 认证

认证路径：

1. **CKA - Certified Kubernetes Administrator**
    - 难度：高
    - 时间：3 小时（实操）
    - 费用：$395
    - 内容：集群安装、管理、故障排查
2. **CKAD - Certified Kubernetes Application Developer**
    - 难度：中
    - 时间：2 小时（实操）
    - 费用：$395
    - 内容：应用开发和部署
3. **CKS - Certified Kubernetes Security Specialist**
    - 难度：很高
    - 时间：2 小时（实操）
    - 费用：$395
    - 内容：安全最佳实践

常见面试题与答案要点

基础概念面试题

Q1: Docker 容器和虚拟机有什么区别?

A（要点）:

```
虚拟机:
- 完整的操作系统环境（GB 级）
- 启动时间：分钟级
- 隔离级别：完全硬件隔离
- 性能开销：高（5-20%）

容器:
- 共享内核，包含应用和依赖（MB 级）
- 启动时间：秒级
- 隔离级别：进程级隔离（Namespace/Cgroup）
- 性能开销：低（1-5%）

总结：容器更轻量、更快、密度更高
```

Q2: 什么是 Docker 镜像？它如何存储的?

A（要点）:

```
镜像本质:
- 只读的文件系统快照
- 分层存储结构
- 每一层是前一层的增量

存储方式:
- Union FS：多个只读层 + 一个可写层
- 每个 RUN/COPY/ADD 指令创建一层
- 层之间通过 diff 增量存储，节省空间

优点:
- 共享基础层减少存储
- 层级缓存加快构建
- 支持高效分发
```

Q3: 容器如何实现隔离?

A（要点）:

```
技术手段：
1. Namespace（资源隔离）：
   - PID Namespace：进程隔离
   - Network Namespace：网络隔离
   - Mount Namespace：文件系统隔离
   - UTS Namespace：主机名隔离
   - IPC Namespace：进程间通信隔离

2. Cgroup（资源限制）：
   - 限制 CPU 使用
   - 限制内存使用
   - 限制磁盘 I/O
   - 限制网络带宽

3. Linux 能力机制（权限控制）：
   - 削减不必要的 root 权限
   - 限制容器能力

4. SELinux / AppArmor（强制访问控制）
```

Dockerfile 面试题

Q4: 如何优化 Docker 镜像大小?

A（要点）：

```
1. 选择合适的基础镜像：
   scratch < alpine:3.21 < python:3.14-slim < python:3.14

2. 多阶段构建：
   - 构建阶段只保留编译工具
   - 运行阶段只包含最终二进制
   - 典型场景：Go、Node.js、Java

3. 清理包管理器缓存：
   apt-get clean && rm -rf /var/lib/apt/lists/*
   yum clean all && rm -rf /var/cache/yum
   pip install --no-cache-dir

4. 合并 RUN 指令：
   减少镜像层数

5. 使用 .dockerignore：
   排除不必要的构建上下文

6. 去除调试符号：
   Go: -ldflags="-w -s"
   C/C++: strip binary

7. 压缩资源：
   gzip 静态文件，压缩图片
```

Q5: CMD 和 ENTRYPOINT 有什么区别?

A（要点）：

```
CMD:
- 定义容器默认命令
- 容器运行时可被覆盖：docker run image_name custom_cmd
- 可以有多个 CMD，只有最后一个生效

ENTRYPOINT:
- 定义容器的可执行程序
- 容器运行时参数追加而非覆盖
- 与 CMD 配合使用

推荐用法:
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8000"]

# 运行 docker run image --debug 会执行:
# python app.py --debug
```

网络和存储面试题

Q6: Docker 网络驱动的区别?

A（要点）:

```
Bridge（默认）:
- 虚拟网桥，容器间通过网桥通信
- 支持端口映射
- 隔离性好，性能适中

Host:
- 使用宿主机网络栈
- 性能最优，隔离性最差
- 容器端口直接映射到宿主机

Overlay:
- 跨主机通信，基于 VXLAN
- Docker overlay 网络默认使用 UDP 4789 传输数据
- Swarm 标准；Kubernetes 通常由 CNI 插件实现跨主机网络
- 性能略低，支持分布式

macvlan:
- 容器获得 MAC 地址
- 表现为物理机，性能好
- 用于物理网络集成

None:
- 无网络，完全隔离
```

Q7: Volume 和 Bind Mount 有什么区别?

A（要点）:

```
Volume:
- Docker 管理，存储位置：/var/lib/docker/volumes/
- 跨平台兼容，隔离性好
- 支持驱动，可扩展
- 推荐在生产环境使用

Bind Mount:
- 宿主机管理，任意位置
- 跨平台兼容性一般
- 性能好，用于开发环境
- 权限管理复杂

tmpfs:
- 内存文件系统，不持久化
- 用于临时文件、敏感数据
- 性能最好，重启丢失
```

安全和生产面试题

Q8: 如何提高 Docker 安全性?

A（要点）：

```
镜像安全:
- 使用官方镜像或可信镜像源
- 定期扫描漏洞（Trivy/Grype）
- 镜像签名验证（Cosign）
- 生成和管理 SBOM

容器运行:
- 以非 root 用户运行
- 使用 read-only 文件系统
- 限制 Linux 能力
- 使用 AppArmor 或 SELinux

宿主机安全:
- 启用 TLS 认证 API
- 不暴露 /var/run/docker.sock
- 使用 Rootless 容器
- 定期更新 Docker

网络安全:
- 使用自定义网络隔离
- 配置网络策略
- 限制出入站流量
```

Q9: 容器被 OOM 杀死，如何诊断和解决?

A（要点）：

```
诊断：
1. 检查容器是否被 OOM 杀死：
   docker inspect <container> | grep OOMKilled

2. 查看宿主机日志：
   dmesg | grep -i oom
   journalctl -u docker | grep -i oom

3. 监控内存使用：
   docker stats <container>
   docker exec <container> ps aux --sort=-%mem

解决：
1. 增加内存限制：
   docker update -m 2g <container>

2. 检查内存泄漏：
   使用内存分析工具（heapdump、pprof）

3. 优化应用：
   - 增加垃圾回收频率
   - 减少缓存大小
   - 使用对象池模式

4. 使用内存交换（最后手段）：
   docker run -m 512m --memory-swap 1g
```

Q10: 如何在 CI/CD 中集成 Docker?

A（要点）：

构建阶段：
- 触发器：Push / PR 事件
- 构建镜像：docker build
- 标记：git sha、版本号
- 扫描：Trivy 漏洞扫描
- 签名：Cosign 镜像签名

存储阶段：
- 推送到镜像仓库：docker push
- 记录 SBOM 和扫描报告

部署阶段：
- 验证镜像签名
- 获取镜像摘要
- 更新部署配置
- 触发 GitOps 工作流

监控阶段：
- 收集应用日志
- 监控性能指标
- 告警异常情况

示例工作流：
1. GitHub Actions / GitLab CI 监听 push
2. 运行单元测试
3. 构建 Docker 镜像
4. 推送到 Docker Hub / ECR
5. 触发 ArgoCD / Flux 自动部署
6. 监控部署状态

Built with mdPress

学习进度跟踪模板

```
# Docker 学习进度跟踪

## 第一阶段：基础入门（目标：2 周）
- [ ] 学完第 1-3 章（6 小时）
- [ ] 完成基础命令练习（10 小时）
- [ ] 运行官方镜像
- [ ] 创建和推送第一个镜像到 Docker Hub
- [ ] 完成度：___%

## 第二阶段：核心开发（目标：4-6 周）
- [ ] 学完第 4-11 章（15 小时）
- [ ] 完成 3 个 Dockerfile 最佳实践项目
- [ ] 掌握 Docker Compose（5 个项目）
- [ ] 学习数据管理和网络（8 小时）
- [ ] 完成度：___%

## 第三阶段：生产优化（目标：6-12 周）
- [ ] 学完第 12-21 章（25 小时）
- [ ] 镜像安全扫描和签名
- [ ] 搭建完整监控栈
- [ ] 配置 CI/CD 流程
- [ ] Kubernetes 基础（30 小时）
- [ ] 完成度：___%

## 第四阶段：专家深造（目标：12+ 周）
- [ ] Kubernetes 高级特性
- [ ] 服务网格学习
- [ ] 底层实现研究
- [ ] 贡献开源项目
- [ ] 完成度：___%

## 证书目标
- [ ] Docker DCA 认证
- [ ] CKA 认证
- [ ] CKAD 认证

## 实战项目清单
- [ ] Python Web 应用容器化
- [ ] Node.js 微服务
- [ ] 数据库容器化
- [ ] 完整微服务架构
- [ ] 监控和日志系统
- [ ] CI/CD 流程实现
```

快速参考速查表

常用命令速查：

```
# 镜像管理
docker build -t image:tag .              # 构建镜像
docker images                             # 列出镜像
docker rmi image:tag                      # 删除镜像
docker tag source:tag target:tag          # 标记镜像
docker push registry/image:tag            # 推送镜像
docker pull image:tag                     # 拉取镜像
docker history image:tag                  # 查看镜像历史
docker inspect image:tag                  # 查看镜像详情

# 容器管理
docker run [OPTIONS] image                # 运行容器
docker ps [-a]                            # 列出容器
docker stop/start/restart container       # 容器生命周期
docker rm container                       # 删除容器
docker logs [-f] container                # 查看日志
docker exec -it container cmd             # 进入容器
docker inspect container                  # 查看容器详情
docker stats [container]                  # 查看资源使用

# 网络管理
docker network ls                         # 列出网络
docker network create name                # 创建网络
docker network connect/disconnect         # 连接/断开网络
docker network inspect name               # 查看网络详情

# 卷管理
docker volume ls                          # 列出卷
docker volume create name                 # 创建卷
docker volume rm name                     # 删除卷
docker volume inspect name                # 查看卷详情

# Docker Compose
docker compose up [-d]                    # 启动服务
docker compose down                       # 停止服务
docker compose ps                         # 列出服务
docker compose logs [-f] [service]        # 查看日志
docker compose exec service cmd           # 在服务中执行命令
docker compose build                      # 构建服务镜像
```

Built with mdPress