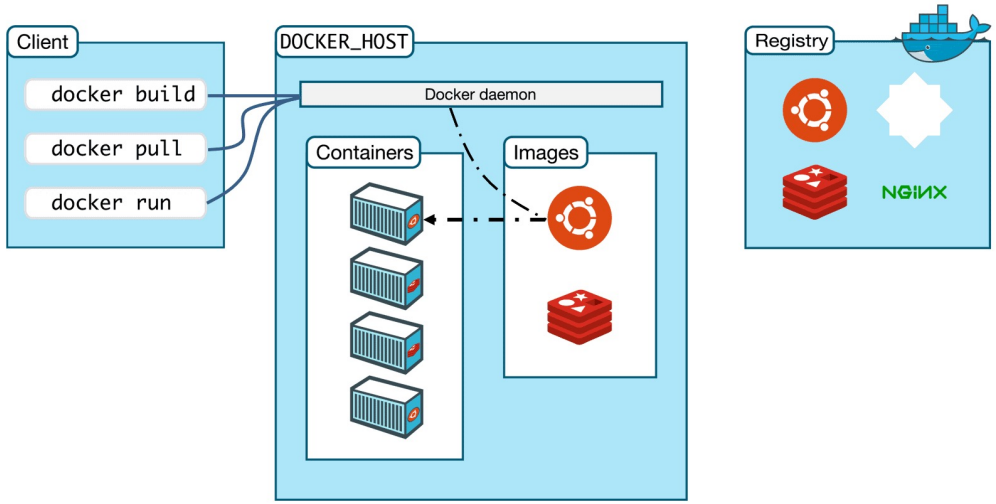
# 容器的基本概念

# Docker 的定义和特点

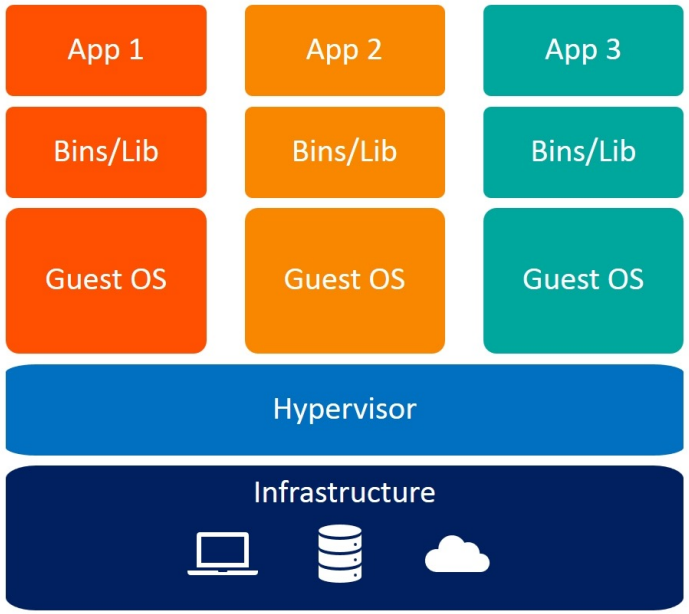
- 轻量和可移植性: Docker容器具有较小的资源消耗，共享操作系统内核，提供高度可移植的环境，使应用程序可以在不同的平台和环境中快速部署和运行。
- 隔离性和安全性: Docker利用容器技术实现应用程序的隔离，每个容器都拥有独立的文件系统、进程空间和网络栈，确保应用程序之间相互隔离，提高安全性。
- 可伸缩和弹性: Docker容器可以根据需要快速扩展或缩减，实现应用程序的弹性伸缩，以适应不同负载下的需求变化，提高系统的可伸缩性。
- 快速部署和启动: Docker容器可以快速部署和启动，减少了应用程序的部署时间，提高了开发和交付的效率。
- 版本控制和复用: Docker容器可以被版本化和存储，方便管理和复用，确保应用程序的一致性和可重复性，简化了开发、测试和部署过程。

# Docker 的架构和组件

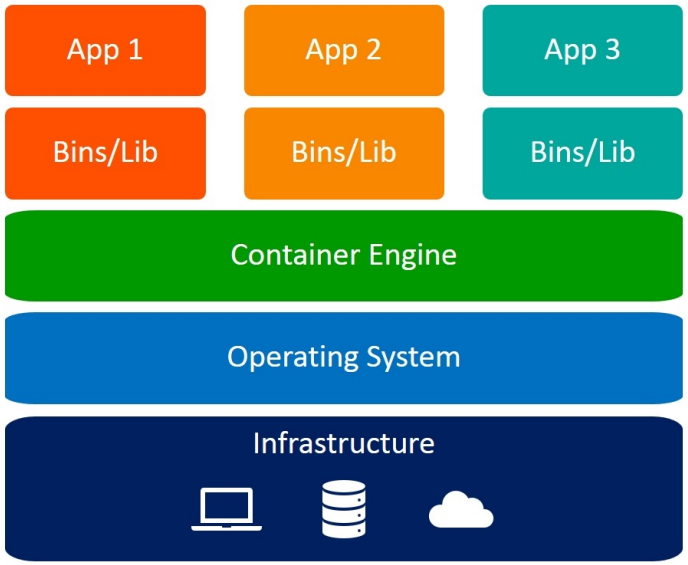



- Docker Daemon: Docker Daemon是Docker的后台服务，管理和运行容器。
- Docker Clients: Docker Clients是用于与Docker Daemon通信的工具或接口。
- Docker Registry: Docker Registry是用于存储和分发Docker镜像的中央仓库。
- Docker Images: Docker images是Docker容器的静态快照，包含了运行容器所需的文件系统和应用程序。

# Docker vs VM




Virtual Machines

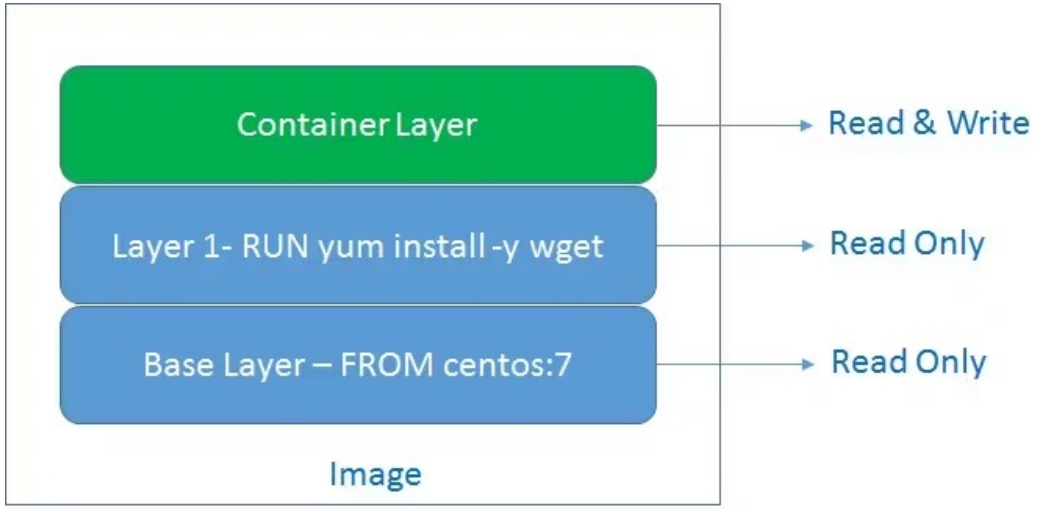



Containers

# Docker vs VM

| | Docker | VM |
|---|---|---|
| 启动时间 | 几秒钟内启动 | 需要几分钟来启动 |
| 运行环境 | 利用 Docker 引擎运行 | 利用虚拟化技术中的hypervisor运行 |
| 内存效率 | 由于无需虚拟化，需要的内存较少 | 需要加载整个操作系统，内存利用效率较低 |
| 隔离性 | 隔离机制较弱，容易受到外部影响 | 高效的隔离机制，最小化可能的干扰 |
| 部署 | 使用单个容器化镜像在各个平台上部署简单 | 部署相对较长，需要单独的实例来执行 |
| 使用 | 复杂的使用机制，涉及第三方和Docker管理的工具 | 使用工具简单易用 |

# Docker 镜像分层




- Docker 镜像是由多个分层（layers）组成的。每个分层都包含了镜像的一部分文件系统或配置信息。
- 每个 Docker 镜像都有一个基础镜像（base image），它包含了操作系统的核心组件和一些基本的工具。
- 当你修改或更新一个镜像时，Docker 只会更新相应的层，而不是整个镜像。
- 当你使用 Docker 运行容器时，Docker 会将这些镜像层堆叠在一起，以创建一个容器的文件系统。

# Docker 镜像的构建

### 编写 Dockerfile

Dockerfile 是一个文本文件，它包含了一组用户可以调用的指令来创建一个镜像

```
FROM alpine:latest
RUN apk update && apk add python3
WORKDIR /web
COPY .. /web
ENTRYPOINT [ "python3", "-m", "http.server"]
CMD [ "8080" ]
```

### 构建 Docker 镜像

```
docker build -t myapp:latest .
```

### 运行 Docker 镜像

```
docker run -p 8080:8080 my-web-server
```

# Docker 镜像的构建

### 重载 CMD

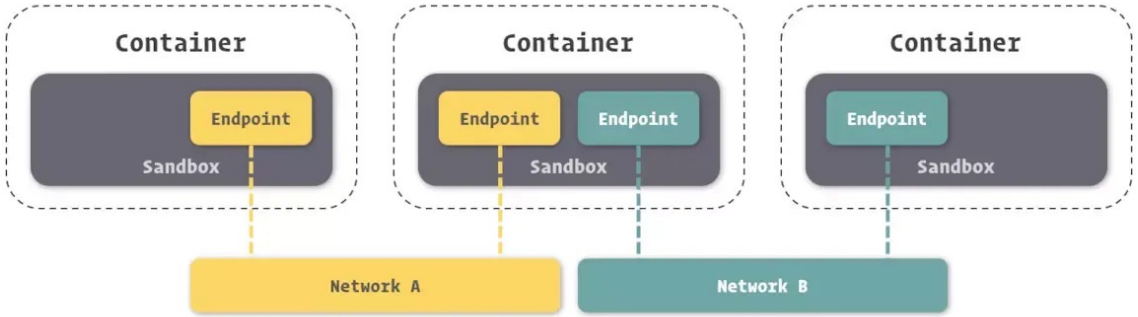
```
docker run -p 8080:9090 my-web-server 9090
```

### CMD 和 ENTRYPOINT 区别

对于 `docker run` 命令行参数的处理方式：`CMD` 可以被覆盖，而 `ENTRYPOINT` 不能（除非使用 `--entrypoint`）。 在设计 Dockerfile 时，一般将那些需要用户能够覆盖的命令或参数放在 `CMD` 中，将那些核心的、不希望用户轻易改动的命令放在 `ENTRYPOINT` 中。

# Docker 网络

**架构**




- 沙盒（Sandbox）： 提供了容器的虚拟网络栈，也即端口套接字、IP路由表、防火墙等内容。隔离容器网络与宿主机网络，形成了完全独立的容器网络环境。
- 网络（Network）： 可以理解为Docker内部的虚拟子网，网络内的参与者相互可见并能够进行通讯。Docker的虚拟网络和宿主机网络是存在隔离关系的，其目的主要是形成容器间的安全通讯环境。
- 端点（Endpoint）：是指容器在网络上的终结点，用于与其他容器或主机进行通信。每个容器都有一个或多个端点，用于接收和发送网络流量。

# Docker 网络

### EMQX 端口

```
EXPOSE 1883 8081 8083 8084 8883 11883 18083 4370 5369
```

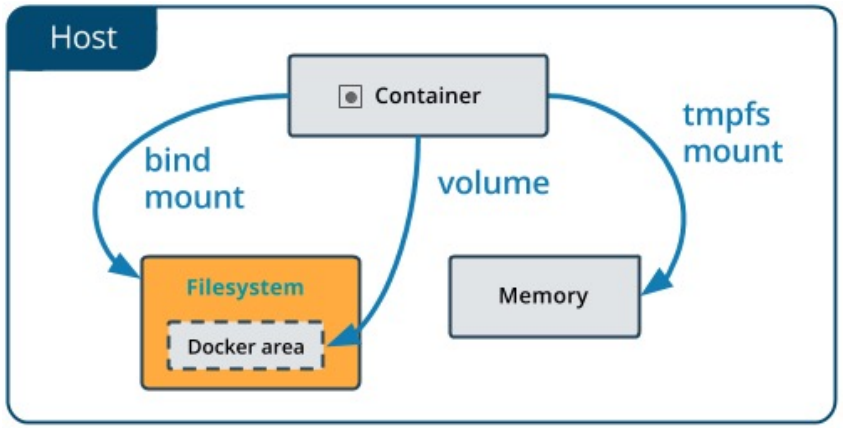
EXPOSE 指令用于声明端口。但它并不会自动映射到主机上的相应端口。要实现端口映射，需要在运行容器时使用-p选项来将容器的端口映射到主机上的端口。

### 映射端口

```
docker run -d --name emqx -p 1883:1883 -p 8083:8083 -p 8084:8084 -p 8883:8883 -p 18083:18083  emqx:5.0.26
```

# Docker 存储

**3种基本方式**




- 数据卷（Volume）：它是由 Docker 管理的特殊目录，专门用于在容器之间共享和存储数据。数据卷提供了一种方便的方式来将数据与容器分离，使得容器可以在重新创建或迁移时保留数据。
- 绑定挂载（Bind Mount）： Docker 中一种将主机文件或目录直接挂载到容器中的机制。通过绑定挂载，可以实现容器与主机之间的文件共享和数据交互。
- tmpfs：它可以将一部分系统内存（RAM）用作文件系统的存储空间。在 Docker 中，tmpfs 可以用来创建一个临时的的文件系统，用于在容器内部存储临时数据，该文件系统在容器停止或重新启动后将被清空。

# Docker 存储

### EMQX 持久化

```
VOLUME ["/opt/emqx/log", "/opt/emqx/data"]
```

在容器中创建了两个挂载点，分别是/opt/emqx/log和/opt/emqx/data。

如果在运行容器时没有使用–v选项显式指定挂载点，而Dockerfile中使用了VOLUME指令创建了挂载点，那么Docker会自动创建匿名卷。

如果需要在运行容器时指定挂载点的位置，并且需要与主机上的目录进行映射和共享，那么建议使用–v选项显式指定挂载点的位置和配置。

### 挂载点映射

```
docker run -d --name emqx  -v /opt/emqx/log:/opt/emqx/log -v /opt/emqx/data:/opt/emqx/data emqx:5.0.26
```

# Docker Compose

Docker Compose 是一个用于编排和运行多容器的工具

## Docker Compose VS Kubernetes

| 功能/特性 | Docker Compose | Kubernetes |
| --- | --- | --- |
| 容器编排 | 适用于本地开发和简单应用程序的单主机容器编排工具 | 适用于大规模生产环境的容器编排和管理平台 |
| 部署方式 | 单主机或Swarm集群 | 多节点集群 |
| 缩放性 | 有限的横向扩展能力，适用于小规模应用程序 | 高度可扩展，适用于大规模和复杂应用程序 |
| 自动发现 | 基于DNS服务自动发现和连接容器 | 使用Kubernetes服务发现机制进行容器间通信 |
| 健康检查 | 支持基本的健康检查 | 强大的容器健康检查和自动恢复能力 |
| 存储管理 | 有限的本地存储管理支持 | 丰富的存储管理选项，包括持久卷、存储类和动态卷等 |
| 网络管理 | 有限的本地网络管理支持 | 强大的网络管理和服务间通信能力 |
| 配置管理 | 环境变量和.env文件 | 配置映射、ConfigMap和Secrets等 |
| 水平扩展和滚动更新 | 有限的水平扩展和滚动更新支持，需要手动操作 | 内置的水平扩展和滚动更新支持，自动管理容器的创建和销毁 |

# Docker Troubleshooting

**容器启动后立即停止**：

- 使用命令 docker ps –a 检查容器的状态，确认容器是否已停止。
- 运行 docker logs 查看容器的日志输出，寻找有关容器停止的错误消息。
- 如果发现容器因为某些错误而停止，尝试修复错误并重新启动容器。

**无法通过网络访问容器**：

- 确保容器的端口映射设置正确。使用 docker ps 查看容器的端口映射情况，确保端口号匹配并且端口是打开的。
- 检查主机防火墙设置，确保端口在防火墙中被允许通过。
- 检查容器内部的应用程序是否正在侦听正确的端口。

# Docker Troubleshooting

**容器内应用程序无法正常工作**：

- 使用 docker logs 查看容器的日志输出，查找与应用程序相关的错误消息。
- 检查容器内应用程序所需的配置参数是否正确提供。确保命令、环境变量和卷挂载等设置正确。
- 尝试进入容器，使用 docker exec –it bash 进入容器的交互式终端，然后手动运行应用程序并查看输出和错误消息。

**容器运行缓慢或资源限制**：

- 使用 docker stats 查看容器的资源使用情况，包括CPU、内存和网络等。
- 如果发现资源限制不足，可以使用 docker update ––cpu–shares ––memory 命令来调整容器的CPU和内存限制。

# Docker 最佳实践

- 使用官方镜像：尽可能使用官方提供的Docker镜像，这些镜像由Docker团队或相关项目维护。官方镜像通常会定期更新并包含最新的安全补丁。
- 最小化镜像大小：尽量使用基于Alpine Linux等轻量级发行版构建的基础镜像，避免使用过于庞大的基础镜像。这可以减小镜像大小，提高下载速度和容器启动时间。
- 多阶段构建：使用多阶段构建技术，可以在构建过程中使用不同的镜像和环境，以减小最终生成的镜像的大小。这可以在构建阶段使用临时镜像来编译和构建应用程序，并在最终生成的镜像中只包含运行时所需的部分。
- 每个容器应该只有一个关注点: 每个容器只应该有一个主要的运行进程。例如，不要在一个容器中同时运行一个web服务器和数据库。
- 不要在容器中运行以root用户身份运行进程: 由于安全原因，你不应该使用root用户运行容器中的进程。例如：USER emqx
- 理解并合理使用Docker缓存

# Docker 最佳实践

**多阶段构建**

```
# 第一阶段：构建阶段
ARG BUILD_FROM=ghcr.io/emqx/emqx-builder/5.0-16:1.13.4-24.2.1-1-debian11
ARG RUN_FROM=debian:11-slim
FROM ${BUILD_FROM} AS builder
...
# 第二阶段：安装构建依赖
FROM $RUN_FROM
COPY --from=builder /emqx-rel/emqx /opt/emqx
...
```

1. ARG BUILD_FROM=ghcr.io/emqx/emqx-builder/5.0-16:1.13.4-24.2.1-1-debian11 和 ARG RUN_FROM=debian:11-slim：这两行定义了两个构建参数。第一个参数定义了用于构建EMQ X的基础镜像，第二个参数定义了运行EMQ X的基础镜像。
2. FROM ${BUILD_FROM} AS builder：这个指令开始了第一个构建阶段。它使用了BUILD_FROM参数定义的镜像，并给这个阶段命名为builder。
3. COPY --from=builder /emqx-rel/emqx /opt/emqx：从第一阶段（builder）复制EMQ X的构建结果到这个阶段的/opt/emqx目录。

# Docker 最佳实践

**理解并合理使用Docker缓存**

```
FROM node:14
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
CMD ["node", "app.js"]
```

在这个例子中，我们先复制 package.json 和 package-lock.json 到镜像中，然后运行 npm install。这是因为我们的 package.json 和 package-lock.json 文件不会像我们的源代码那么频繁地改变。所以我们希望只要 package.json 和 package-lock.json 没有改变，我们就复用之前缓存的层，而不是每次都运行 npm install。

然后，我们再复制我们的源代码到镜像中。源代码可能会经常改变，所以我们把 COPY . . 放在 Dockerfile 的后面。

# Q & A