中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

# Lecture 08：轻量级虚拟化

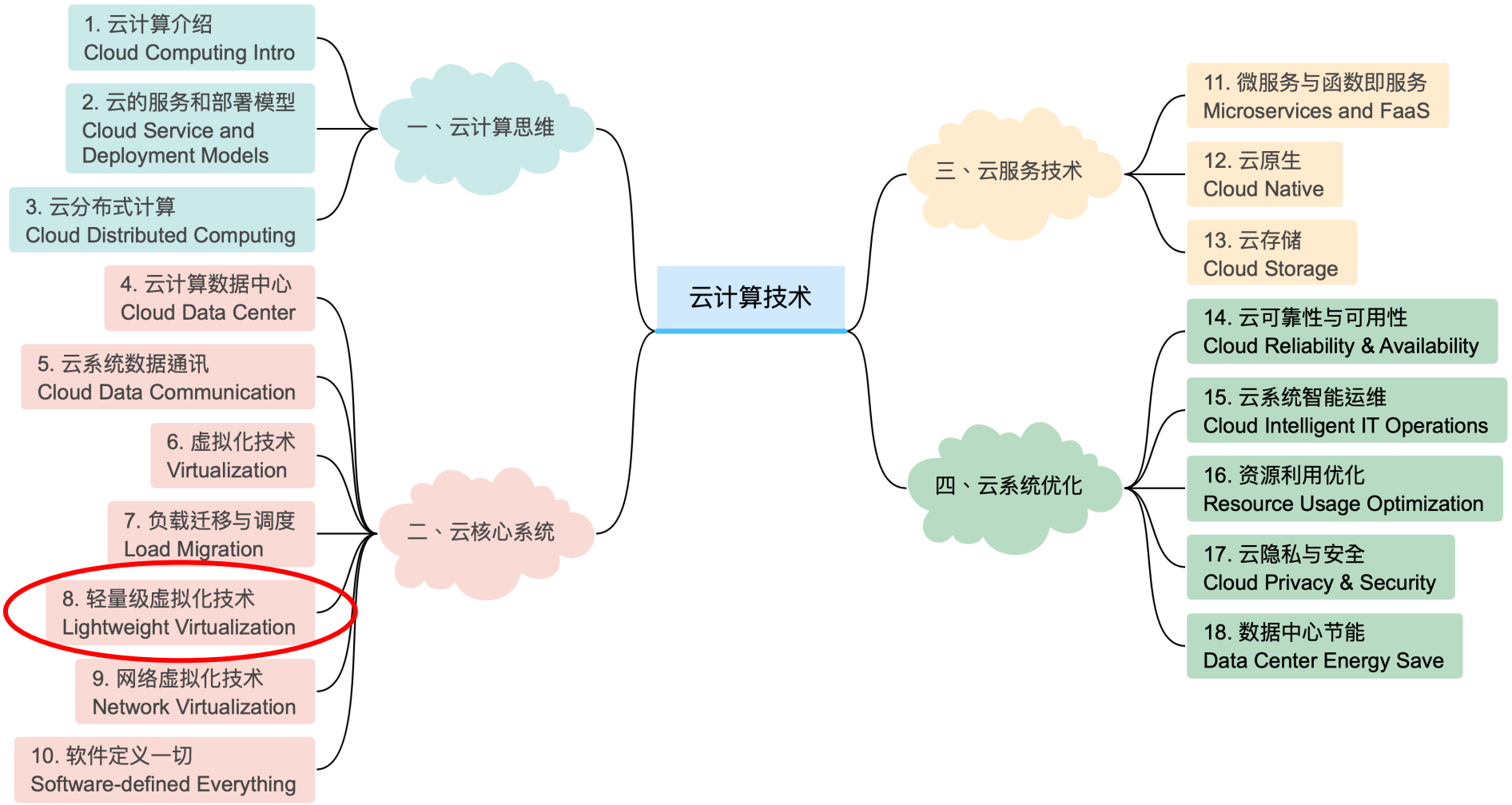
## SSE316：云计算技术
## Cloud Computing Technologies

---


陈壮彬

软件工程学院

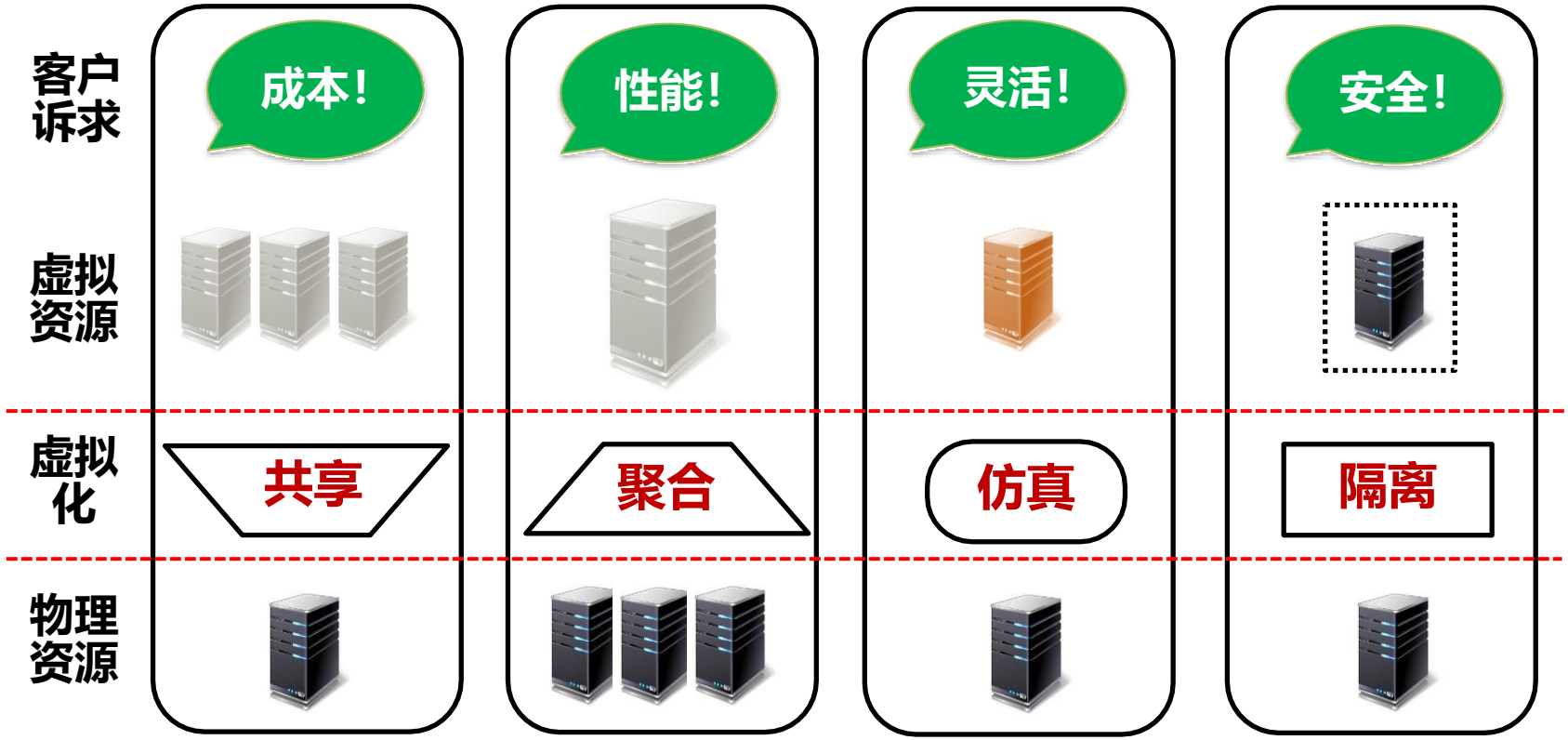
chenzhb36@mail.sysu.edu.cn


云计算技术
一、云计算思维
1. 云计算介绍
Cloud Computing Intro
2. 云的服务和部署模型
Cloud Service and
Deployment Models
3. 云分布式计算
Cloud Distributed Computing
二、云核心系统
4. 云计算数据中心
Cloud Data Center
5. 云系统数据通讯
Cloud Data Communication
6. 虚拟化技术
Virtualization
7. 负载迁移与调度
Load Migration
8. 轻量级虚拟化技术
Lightweight Virtualization
9. 网络虚拟化技术
Network Virtualization
10. 软件定义一切
Software-defined Everything
三、云服务技术
11. 微服务与函数即服务
Microservices and FaaS
12. 云原生
Cloud Native
13. 云存储
Cloud Storage
四、云系统优化
14. 云可靠性与可用性
Cloud Reliability & Availability
15. 云系统智能运维
Cloud Intelligent IT Operations
16. 资源利用优化
Resource Usage Optimization
17. 云隐私与安全
Cloud Privacy & Security
18. 数据中心节能
Data Center Energy Save

# Today’s topics

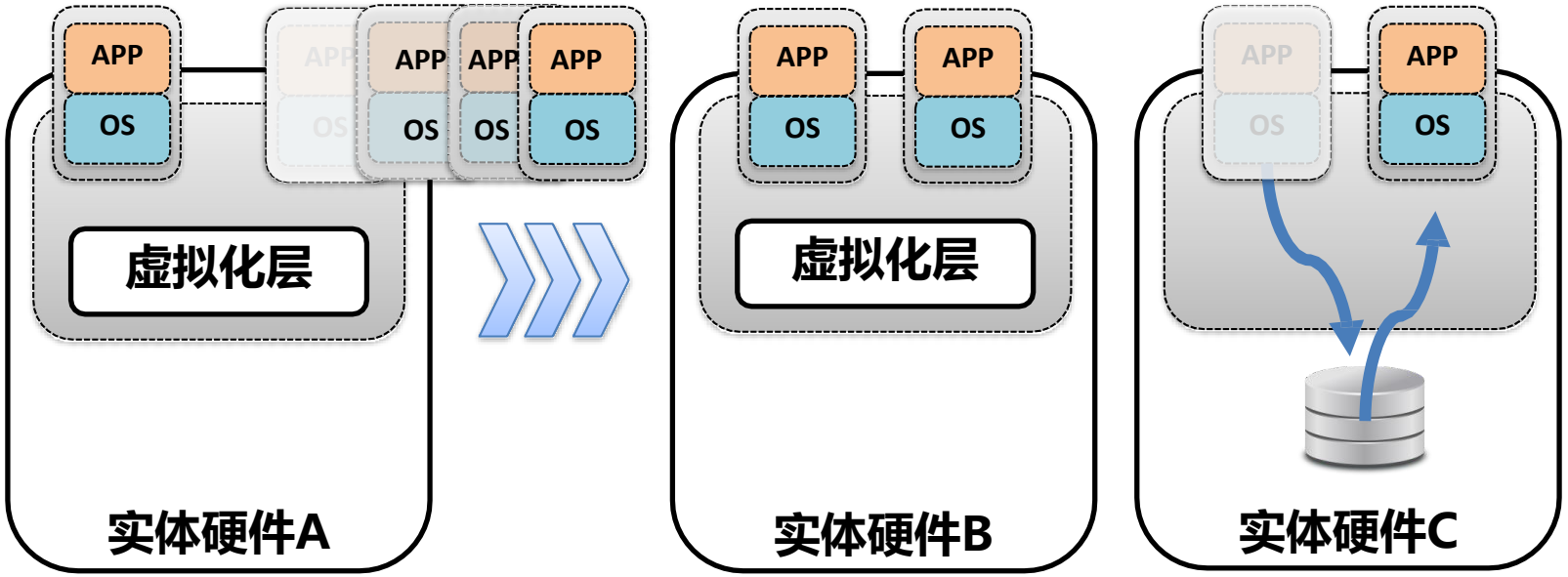
- 轻量级虚拟化
- Linux 控制组和命名空间
- 容器
- Docker

# 虚拟机优势 – 静态角度




**虚拟化为IT资源利用提供全新维度，是云计算重要基础**

# 虚拟机优势 – 动态优势




VM实例的动态迁移　　VM实例的状态回溯

虚拟机状态可以**数据形式存储**，支持**时空层面**的灵活管理

# 从虚拟**一台机器**到虚拟**一个运行环境**

口这一讲不再围绕 hypervisor 本身，而是围绕**应用**、**运行时**和**交付方式**重新看虚拟化

**上节课**

## VM / Hypervisor 视角

- 重点是硬件抽象、guest OS、强隔离边界。
- 关注“如何把一台物理机切成多台逻辑机”。
- 典型对象是虚拟 CPU、虚拟内存、虚拟设备。

更像“把机器服务化”

**这节课**

## Container / Runtime 视角

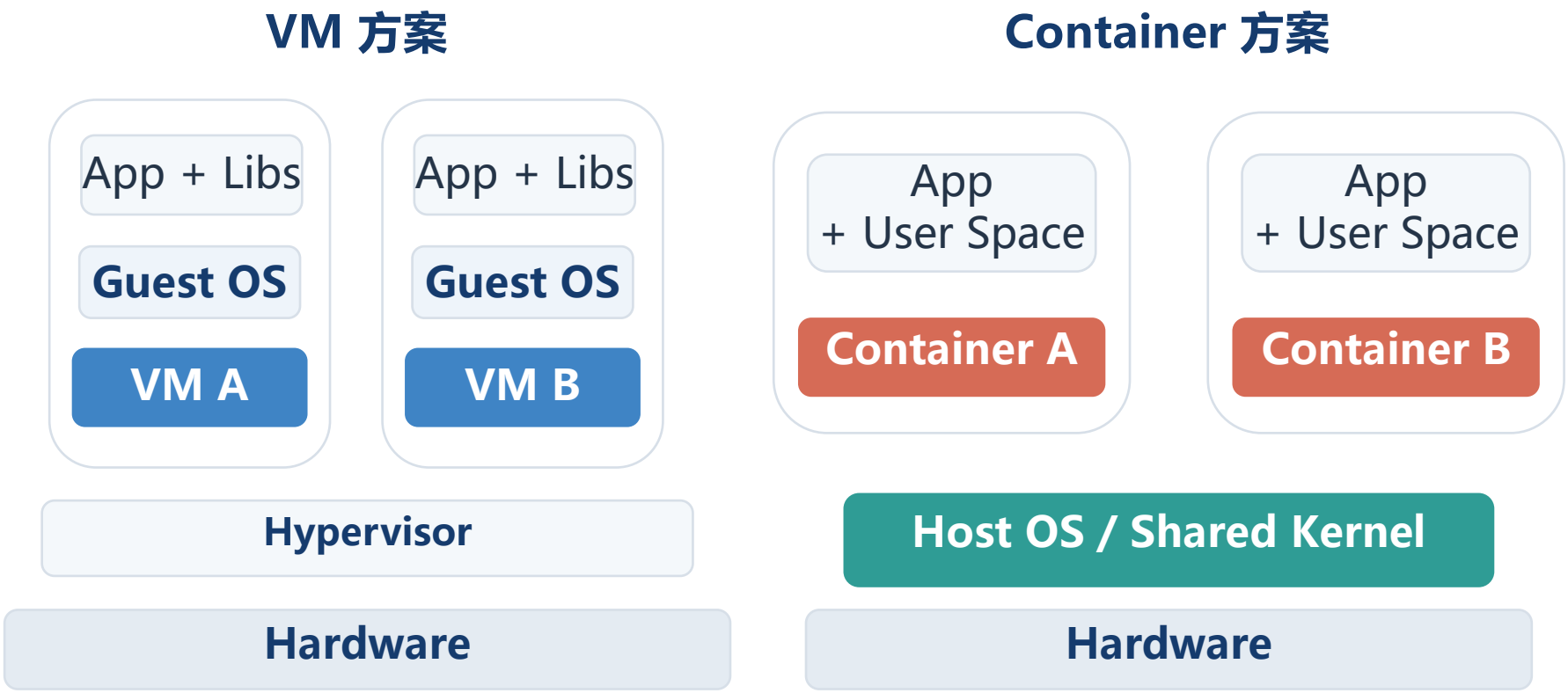
- 重点是应用交付、环境一致性、启动速度和密度。
- 关注“如何给进程一个像机器一样的独立运行空间”。
- 核心对象变成 image、container、registry、orchestrator。

更像“把运行环境服务化”

今天的关键不是“再造一台电脑”，而是“给应用造一个隔离、可搬运、可批量管理的舞台”。

# 轻量级虚拟化：容器 (container)

- VM 和容器都在做隔离，但隔离的粒度和代价不一样




容器真正“省”的不是应用本身，而是每个实例都不用再带着一整套独立内核重复上路

Part I • Why Lightweight?

# 为什么会出现轻量级虚拟化

如果 VM 解决的是“隔离一台机器”，容器更像在解决“快速交付一个运行环境”

# 为什么很多云平台都离不开**更轻**的运行时？

口解决的主要是运维与交付问题，而不只是“把开销再压小一点”

**Speed**

### 启动更快

不必为每个实例再完整启动一套 guest OS。秒级拉起更适合弹性扩缩、CI 和短任务。

适合 burst 场景

**Density**

### 密度更高

多个容器共享宿主机内核，重复的系统开销更少，同一台 host 能塞下更多工作负载。

更像提升承载率

**Delivery**

### 更贴近应用

把应用与依赖一起交付，开发、测试、上线更容易围绕同一种对象协作。

从“管机器”转向“管应用”

轻量级不是白拿的性能红利，它是通过更多资源共享换来的

# 轻量级虚拟化到底在缓解什么？

**Dev**

**开发机能跑，服务器却跑不起来**

库版本、路径和系统差异一变，部署就容易变成排障大会

**Teach**

**课程实验环境难统一**

200 名同学的机器条件不同，助教很难保证每个人都装对同一套依赖

**Scale**

**业务高峰来得太快**

VM 不是不能扩，但准备和装配成本让突发流量显得更棘手

**Ops**

**机器越多，环境越难管**

补丁、依赖、版本和回滚一多，真正累人的往往不是代码而是环境

容器之所以流行，不是因为大家突然更爱虚拟化了，而是因为环境问题终于有了更顺手的对象。

# 容器流行之前，发布应用到底难在哪？

- 容器真正击中的痛点，不只是隔离，而是软件在不同环境之间移动时的摩擦

**过去**

### 像重新搬家

- 同一应用要为不同发行版和版本反复打包
- 依赖库一变，部署脚本就容易和现实脱节
- “在我机器上能跑”并不等于上线就能跑

问题常常出在环境差异，而不是代码本身

**容器**

### 给出统一承诺

- 应用和依赖一起封装，环境跟着应用走
- 启动、停止、更新的入口更统一
- CI/CD 更容易围绕镜像形成稳定流程

软件交付第一次有了更标准的“装载箱”

容器的最大势能，是“把环境当成可分发对象”

# 案例：课程实验平台为什么会偏爱容器？

**没有容器**

## 助教会碰到什么

- 不同同学装的是不同版本的 Python / Node / JDK
- 排错时很难判断是代码错、系统错，还是依赖没装齐
- 每次重置环境都像重新整理一间被不同人住过的宿舍

最耗精力的通常不是写代码，而是对齐环境

**用了容器**

## 平台会更顺什么

- 统一发放同一份镜像，所有同学看到的是同一套依赖
- 实验结束后可以快速回收并恢复到干净状态
- 助教讨论的是“镜像版本”，不再是“你那台电脑上到底改过什么”

教学里，容器最先解决的是一致性和可重置性

如果教学目标是“学应用环境”，容器往往比给每个人发一整台 VM 更轻巧

# 问题思考

**如果学校要同时给 200 名同学发统一实验环境，为什么平台会先想到容器，但又不会彻底抛弃 VM?**

?

Part II • Kernel Primitives

# 容器为什么看起来像一台小机器

容器不是“迷你 VM”，它本质上还是进程，重点是要跟其他进程隔离开来

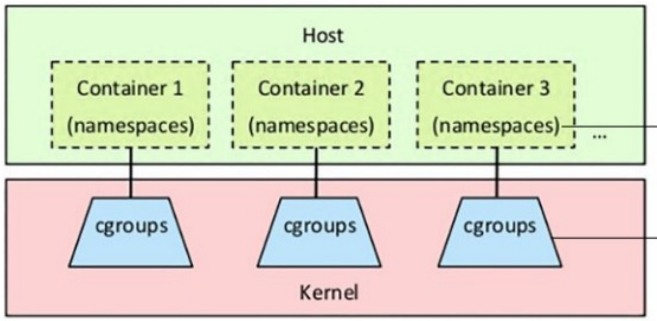
# 如何实现轻量级虚拟化？思考一下🤔

**cgroups**
**控制组**

**namespaces**
**命名空间**

- 最初是为了加强**进程之间的隔离**而开发的，与虚拟化无关
- 可利用它们创建一种新形式的轻量级虚拟化，且没有完全虚拟化相关的开销

- **Controllable properties**
  - CPU and Disk quotas
  - Network isolation
  - I/O rate and Memory limit
  - File system isolation
  - …




**控制你能看到什么资源**

**控制你能使用多少资源**

# 一句话理解容器：进程+隔离视图+资源额度

口容器不是迷你 VM，它更像是“被内核精心布置过的进程舞台”

**Container = Process + Namespaces + cgroups + Image**

**View (namespaces)**

**看见什么**

进程树、网卡、挂载点、主机名都像是“自己的”。机器感首先来自视图隔离。

**Budget (cgroups)**

**能用多少**

CPU、内存、I/O 与进程数可以被限额、统计和治理。容器并不是无限拿资源。

**Source (image)**

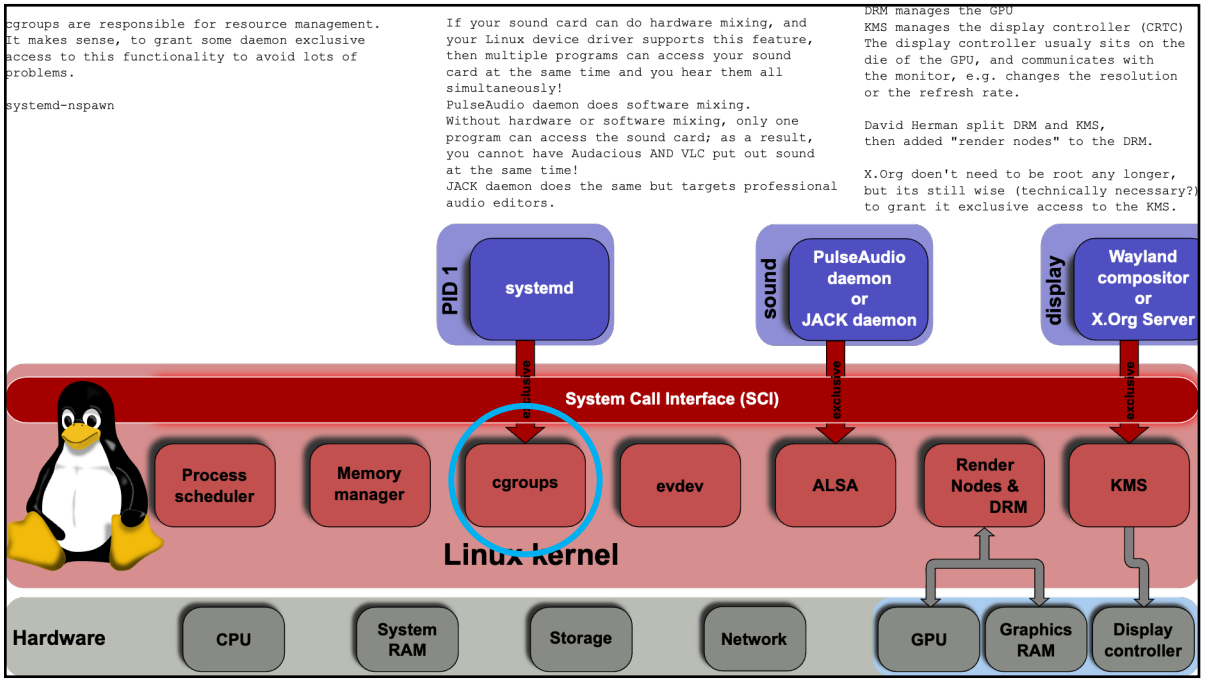
**从哪启动**

镜像提供用户态文件、元数据和启动入口，让环境可以被复制、版本化和分发。

很多“像一台机器”的感觉，其实来自视图和配额；
硬件并没有真的被重新造一遍

# Linux cgroups (control groups)

一种 Linux 内核功能，用于限制，记账，隔离或拒绝**进程**或**进程组**对资源的使用

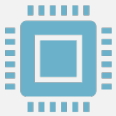



**细粒度控制单个进程和资源的基本块，提供了实现操作系统级虚拟化的方法**

# cgroups 特征

**资源限制**
**Resource limiting**

可以设定进程组在一段时间内可以使用的最大资源



**优先级调整**
**Prioritization**

可以调整进程组访问某些资源的优先级

**资源记账**
**Accounting**

可以跟踪进程组的资源使用情况，进行分析和统计

**进程控制**
**Control**

可以冻结、恢复或者重新启动进程组

# cgroup CPU：它不是让每个人都一样快，而是控制谁能用多少

**没有配额时：谁抢得凶，谁就可能吃掉更多时间片**

同一台主机上，两个容器都想要 CPU

**Host CPU 时间片池**

**批处理容器**
**压测 / 编译**

**API 容器**
**在线请求**

结果：批处理疯狂争抢时，API 也会跟着抖。

批处理吃掉大部分时间片

**有 cgroup 时：平台可以明确“谁最多吃多少”**

quota 像上限，shares 像争抢时的优先级

**Host CPU 100%**

**API 容器**
**quota 50%**
**优先保障**

**批处理容器**
**quota 50%**
**超了就限速**

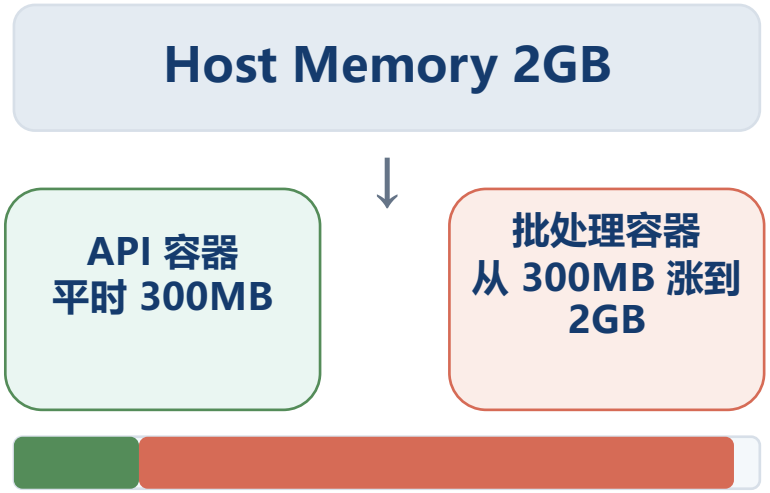
结果：平台不保证每个人都开心，但能保证秩序更稳定。

CPU cgroup 的价值，不是追求绝对平均，而是让共享主机时的秩序更可控

# cgroup Memory：它在提醒你，内存不是“写满了再说”

**没有边界时：一个容器涨太猛，别的服务也会被连带拖垮**

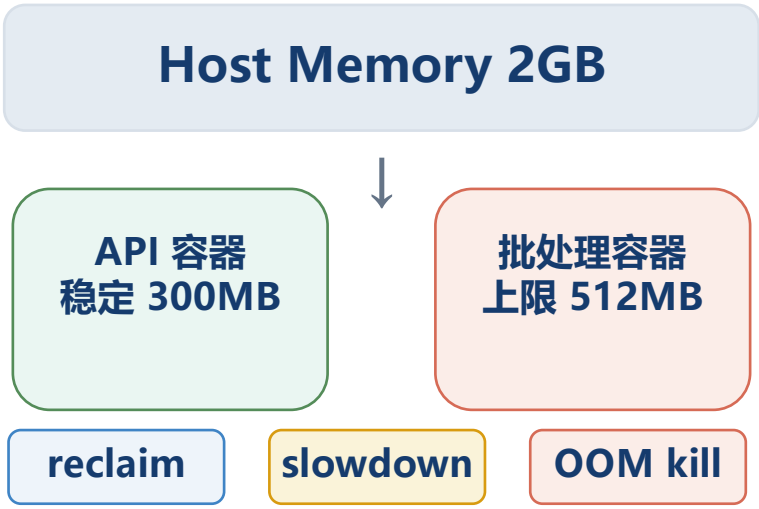
共享主机上，API 和批处理都在吃同一池内存




结果：一旦共享内存池被批处理吃满，API 和整机都可能一起抖

**有 cgroup 时：平台先画一堵墙，越界问题尽量留在局部**

memory.max 像上限，超了以后先回收、再抖动，最后才 OOM kill




结果：API 尽量稳住，越界代价优先留在出问题的容器内

内存 cgroup 像一面墙：它会让风险尽量留在局部，但也会把越界问题更快暴露出来。

# cgroups 案例：限制CPU开销

```
netlab@VM:~$ sudo apt install cgroup-tools

# Copy and past this text (till second EOF) in a terminal
# to create a script that does an infinite loop
netlab@VM:~$ cat <<'EOF' > infinite_full.sh
#!/bin/bash
while true
do
    i=i+1
done
EOF

netlab@VM:~$ chmod +x infinite_full.sh

netlab@VM:~$ cp infinite_full.sh infinite_half1.sh
netlab@VM:~$ cp infinite_full.sh infinite_half2.sh

netlab@VM:~$ sudo cgcreate -g cpu:/cpufullspeed
netlab@VM:~$ sudo cgcreate -g cpu:/cpuhalfspeed

netlab@VM:~$ sudo cgset -r cpu.shares=1024 cpufullspeed
netlab@VM:~$ sudo cgset -r cpu.shares=512 cpuhalfspeed

netlab@VM:~$ sudo cgexec -g cpu:cpufullspeed ./infinite_full.sh &
netlab@VM:~$ sudo cgexec -g cpu:cpuhalfspeed ./infinite_half1.sh &
netlab@VM:~$ sudo cgexec -g cpu:cpuhalfspeed ./infinite_half2.sh &
```

简单的循环加程序

设置两个 cgroups

限制 CPU 使用配额

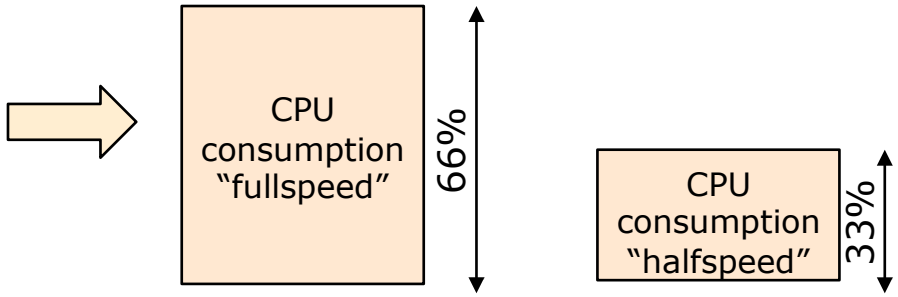
运行程序并添加到相应控制组

# cgroups 案例：限制CPU开销

Output of the 'top' command (on a **single-core** machine)

```
Tasks: 165 total,   4 running, 161 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    1990.8 total,    410.9 free,    491.1 used,   1088.7 buff/cache
MiB Swap:     711.4 total,    711.4 free,      0.0 used.   1309.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5179 root      20   0   22612   3492   3256 R  66.3   0.2   1:30.24 infinite_full.sh
 5222 root      20   0   22612   3560   3332 R  16.2   0.2   0:20.08 infinite_half1.sh
 5152 root      20   0   22612   3484   3260 R  16.7   0.2   0:19.57 infinite_half2.sh
```

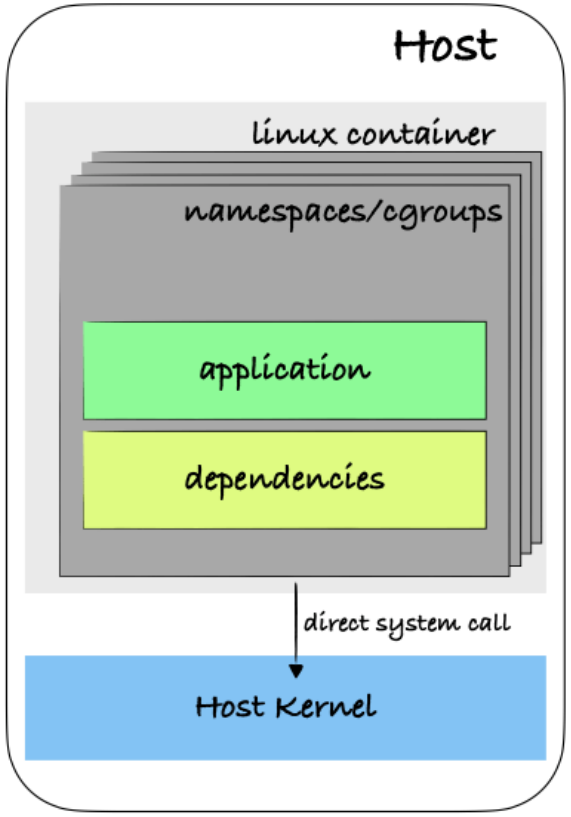


# Linux namespaces

- Linux 内核功能，技术层面不属于 cgroups，但高度相关
  - 为进程组创建**隔离且独立**的虚拟环境

- 两个命名空间可具有独立的
  - 网络堆栈，如虚拟接口、IP 地址等
  - 文件系统，如不同的 /etc/ 文件夹等
  - …

- 当前定义了七种不同类型的命名空间
  - 每种类型的命名空间中可以创建多个不同的空间

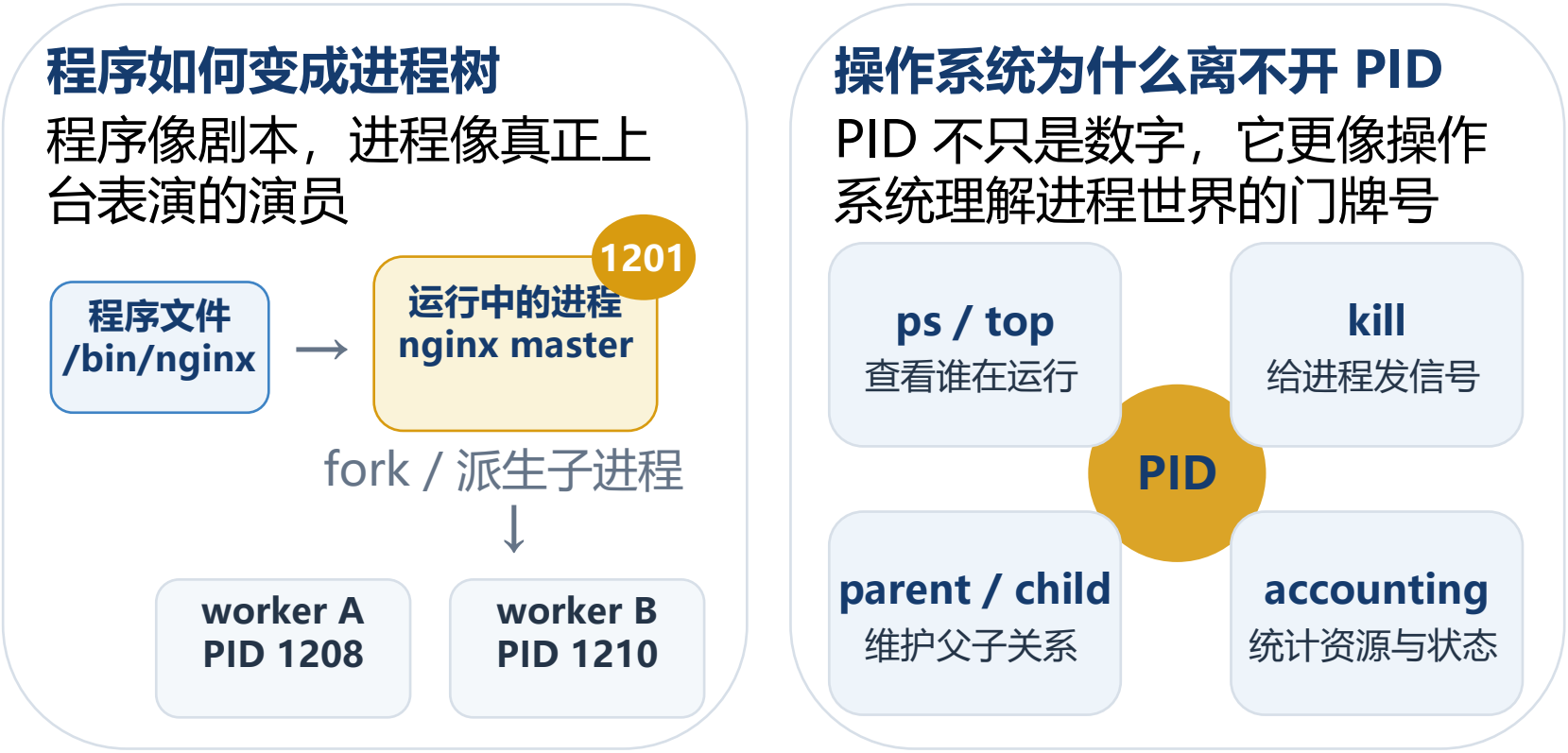


# Linux namespaces 总结

| Namespaces | Constant | Isolates |
|---|---|---|
| IPC | CLONE_NEWIPC | System V IPC, POSIX message queues |
| Network | CLONE_NEWNET | Network devices, stacks, ports, etc. |
| Mount | CLONE_NEWNS | Mount points |
| PID | CLONE_NEWPID | Process IDs |
| User | CLONE_NEWUSER | User and group IDs |
| UTS | CLONE_NEWUTS | Hostname and NIS domain name |
| Cgroup | CLONE_NEWCGROUP | Control groups |

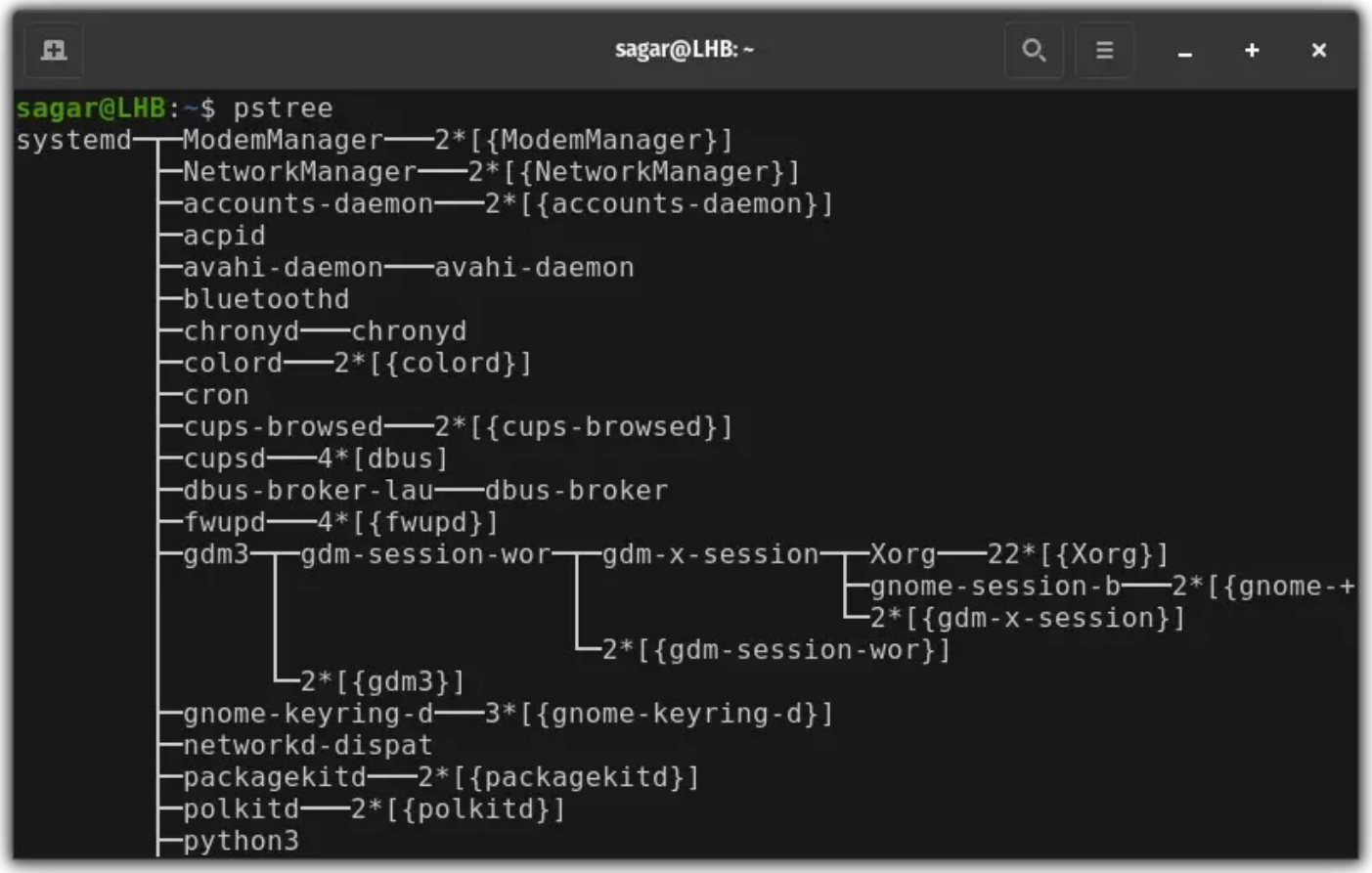
# 先认识 PID：Linux 为什么要给每个进程编号?

- PID: Process ID (系统分不清名字，但靠 PID 能精准区分每一个进程)



# PID: Process ID

- 在Linux中，进程源自单个进程树（Process tree）
  - 每个进程都有一个父进程，直到 init(1)，PID=1
  - 一个进程在需要并发处理任务、执行另一个不同程序或隔离运行风险时，会主动通过系统调用（如 Linux 中的 fork() 或 clone()）来创建子进程



# PID namespace

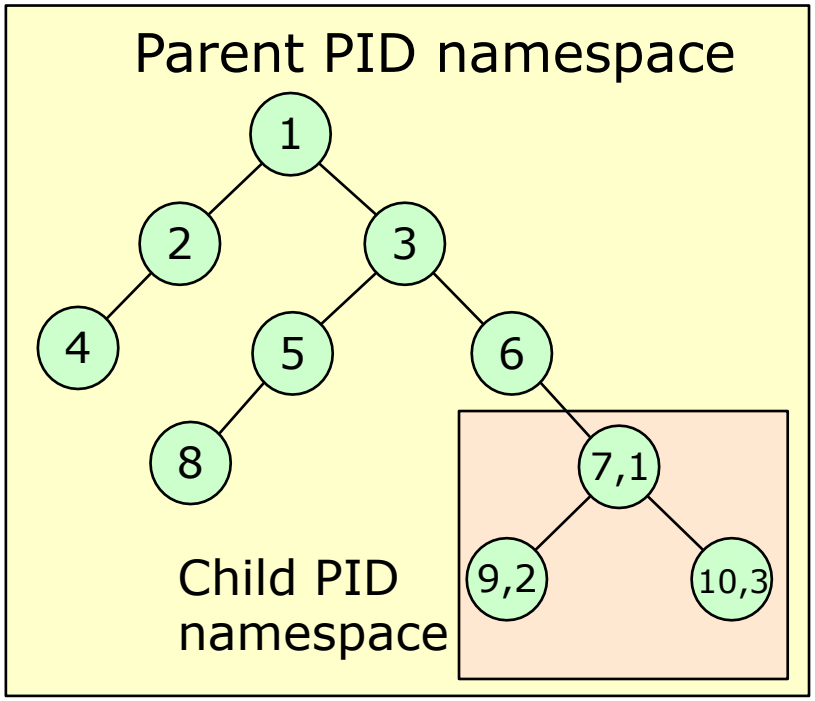
□PID namespace：让容器里的应用觉得自己是“主角进程”

**定义**

**它隔开的是什么**

- 容器里的进程只优先看到自己的那棵进程树
- 因此应用会觉得“我有自己的 1 号进程、子进程和退出关系”
- **这让容器更像一台小机器，而不是裸跑在宿主机上的散乱进程**

本质是进程视图被裁切，而不是宿主机真的消失了

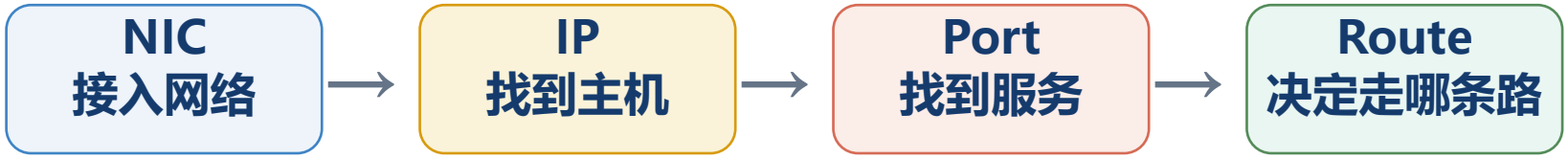



容器里看到的 PID 1，更像是“局部世界的 1 号人物”，不是全系统唯一的 1 号进程，无法对其他进程进行操作（如删除）

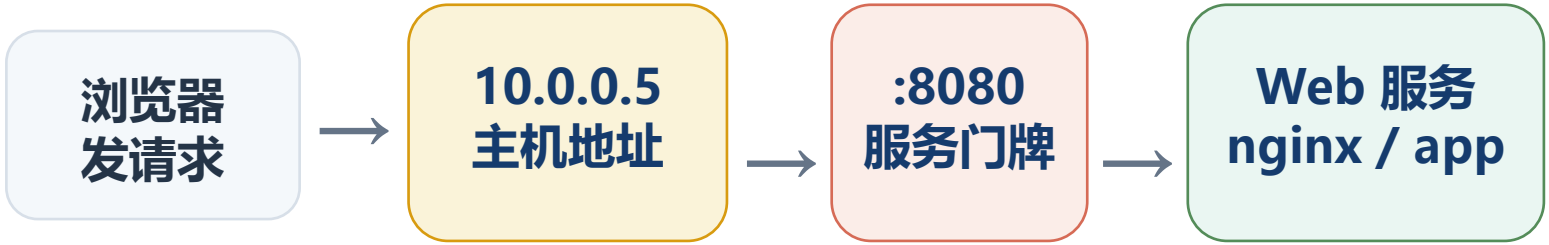
# 先认识网络视角：网卡、IP、端口、路由

## 网络世界的四个角色

可以把它们理解成：接口、地址、门牌和道路规则。




## 一个访问请求是怎么一步步找到服务的



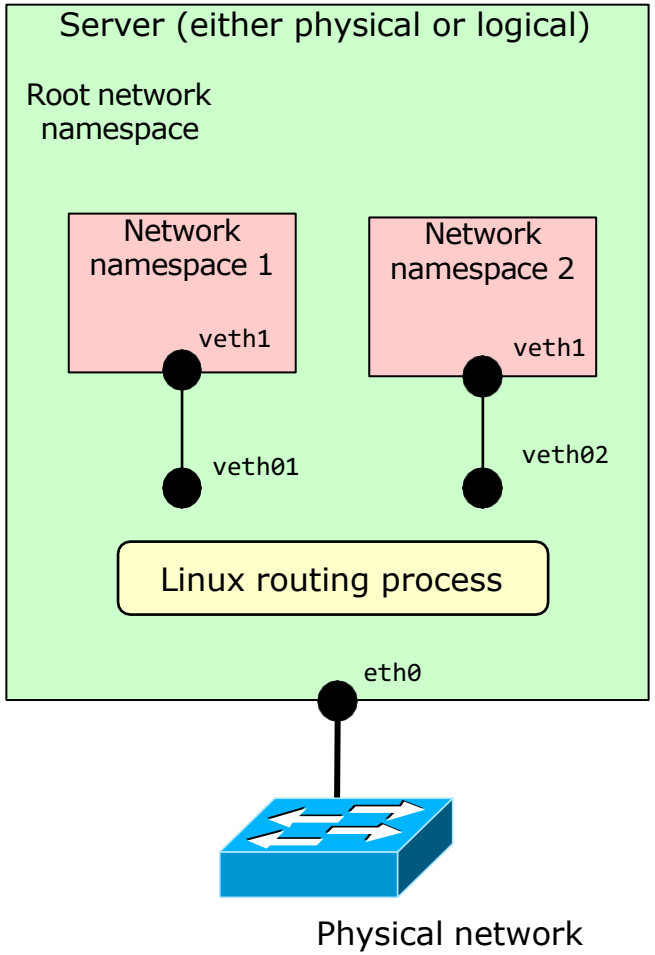

路由表像路标：它决定发往不同地址的流量该从哪张网卡、哪条路出去。

network namespace 不是把网络重新造一遍，而是给进程单独发了一张小型网络地图

# #2: Network namespace

- 网络命名空间允许两个进程**感知完全不同的网络设置**
  - 不同的接口、路由表、防火墙规则等
  - 甚至环回接口也不一样

- 一旦创建了网络命名空间，就应该创建跨多个空间的额外虚拟网络接口
  - 虚拟接口（veth）是一个网络抽象，它是一条两端连接不同命名空间的“网线”
  - Veth 允许流量跨越命名空间边界并传递到另一个命名空间
  - 根命名空间中的桥接/路由进程使流量能够传递到目的地



# Network namespace

□Network namespace：给进程一套像**独立主机**的网络视角

**定义**

**它隔开的是什么**

- 容器可以拥有自己的网卡、IP、路由表和端口视图
- 因此容器内部会觉得自己像在一台单独的主机上监听服务
- 外界如何找到它，还要靠 bridge、veth、NAT 或 overlay 等转发机制

隔离的是网络视图，不是直接把网络问题全部消灭。

**例子**

**同一台宿主机上，两个容器都监听 80**

- 容器 A 内部监听 80，容器 B 内部也监听 80，它们在各自的网络命名空间里并不冲突
- 对外时，可以把它们分别映射成宿主机的 8080 和 8081
- 所以学生看到的现象常常是：容器里都写 80，但宿主机访问入口却不同

这正是“内部世界相似，对外接口可重排”的典型例子。

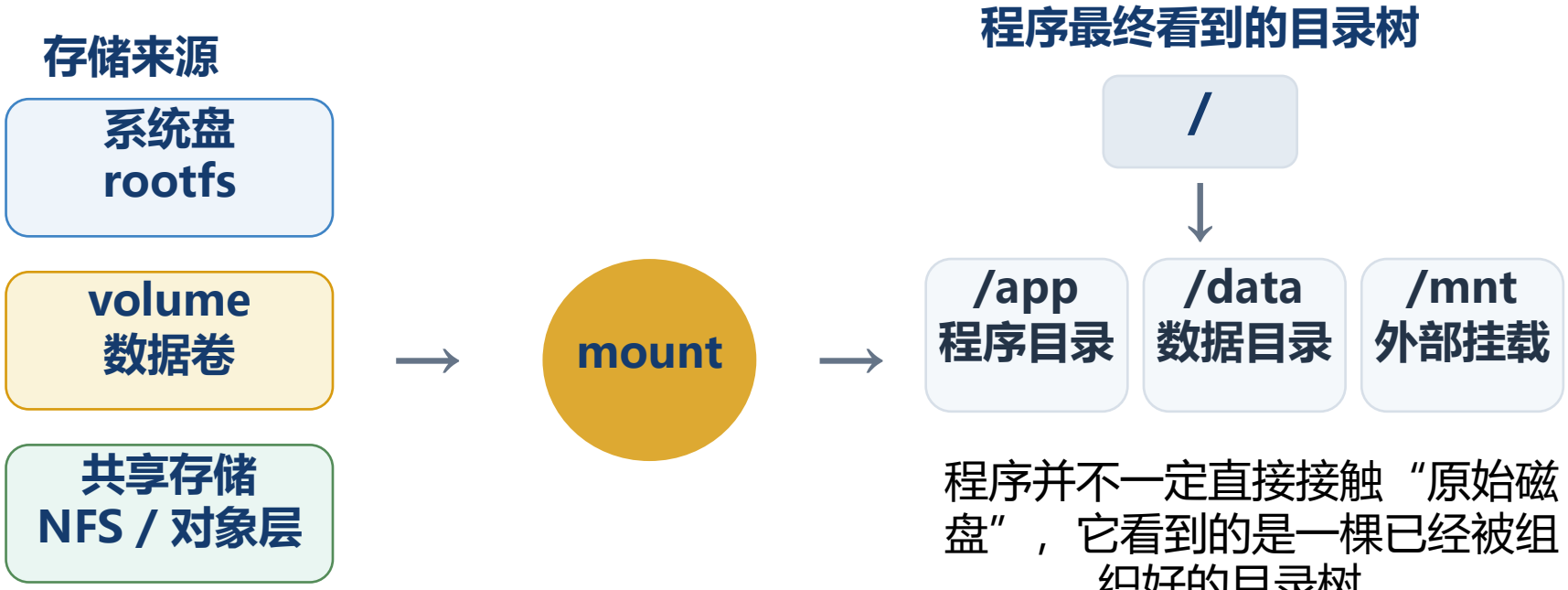
Network namespace 让每个容器像拥有自己的小网卡，但对外连通仍然需要平台来搭桥

# 先认识挂载（mount）

- 将独立的存储设备（如磁盘、U盘）**“拼接”到 Linux 统一的目录树中**（即挂载点），让系统和用户能够像访问普通文件夹一样去读写该设备里的数据

## 把“不同来源的内容”接进同一棵目录树

目录树像展厅，mount 像把不同货架接进这座展厅




程序并不一定直接接触“原始磁盘”，它看到的是一棵已经被组织好的目录树。

# Mount namespace

□Mount namespace：让容器看到一套“属于自己的文件系统布局”

**定义**

**它隔开的是什么**

- 容器会看到自己的根文件系统和挂载点，不必直接暴露宿主机全貌
- 因此应用会感觉自己待在一套独立目录树里，好像拥有自己的磁盘视图
- 但这套视图背后，很多时候仍建立在镜像层和宿主机挂载之上

文件系统的“独立感”，很多来自挂载视图的重新组织。

**例子**

**为什么重建容器后某些文件会不见**

- 如果文件只写在容器的可写层里，删掉容器时，这一层往往也跟着消失
- 真正想长期保留的数据，更适合挂到 volume 或外部存储里
- 所以容器更像旅馆房间：临时用品能放，长期家当最好寄存在专门仓库

Mount namespace 让目录树看起来像“自己的”，但数据是否能长期留下，还要看你把它放在什么层。

# 共享内核：容器强大与脆弱都来自这里

好处

## 共享带来的收益

- 系统重复开销少，启动更快、密度更高。
- 统一运维接口，适合批量部署与快速回收。
- 非常适合微服务、CI、教学实验和短任务。

效率与敏捷性，往往来自更多共享

代价

## 共享带来的约束

- 不能任意换成完全不同的 guest OS / kernel。
- 内核漏洞、配置错误的影响半径可能更大。
- 强多租户场景通常仍想再加一层 VM 或 microVM。

越轻，越要认真看边界落在哪里

容器不是“更好的 VM”，它只是把隔离边界挪到了另一个层级

# 共享内核意味着：有些 VM 能做的事，容器并不能原样照搬

**OS**

**不能随意换 guest OS**

容器共享宿主机内核，所以它不像 VM 那样天然适合跑完全不同的操作系统语义。

**Risk**

**内核问题的影响半径更大**

一旦共享层出现漏洞或配置失误，多个容器可能同时受影响。

**Tenant**

**强多租户更谨慎**

面对不可信代码、监管要求或强隔离场景，平台往往还会叠加 VM 或 microVM。

越轻，通常越依赖共享；越依赖共享，就越要认真看边界落在哪里。

# cgroups 和 namespaces 总结

- 强大的进程隔离在现代计算世界中很重要
  - 更有效地利用计算资源
  - 降低了操作成本（例如，更新不同虚拟机中单独的内核）
  - 允许多个进程共存，具有（强）隔离属性

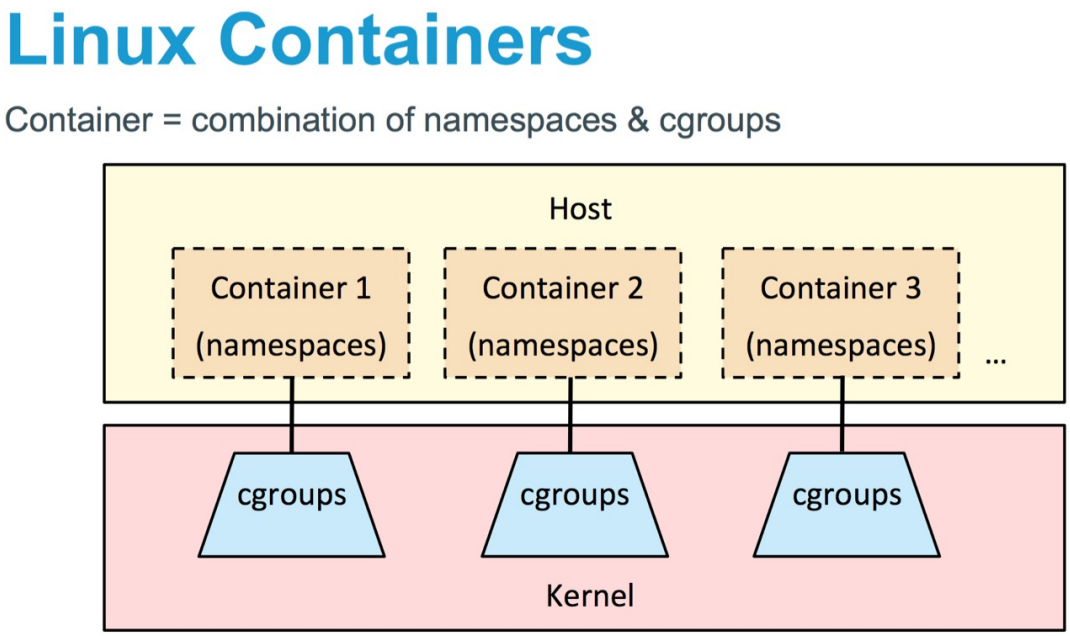
- 然而，实现隔离需要将它们全部结合起来，复杂性很高！

因此，我们需要工具降低配置复杂性并添加新功能

# Linux containers (LXC)

- LXC = cgroups+namespaces（+ user-friendly config）
  - 不需要内核补丁，原生内核版本即可

- 是一个**内核功能集合**，用来以不同的方式隔离进程
- 也是一个**用户空间工具**，使用所有这些功能来创建成熟的容器



# Linux Containers (LXC)


- LXC 的历史定位：它像容器世界的一台“早期样机”
- 可以把它理解成“把零件拼成整机”的第一代工具

**它是什么**

### LXC 在做什么

- 把 namespaces、cgroups 这些原始能力组合起来，形成比较像样的容器体验
- 通过配置和用户空间工具，让“容器化 Linux 系统”变得更可操作
- 它更接近“把一套 Linux 环境装起来”，而不是只盯一个应用进程

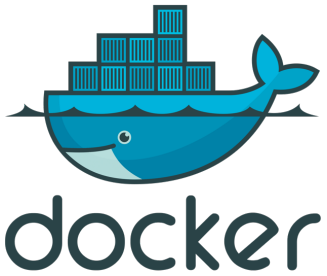
**历史位置**

### 为什么它值得讲

- 它说明容器的底层原语早就存在，真正难的是把体验做顺
- 很多后来的产品，并不是发明隔离，而是把隔离变成可传播、可协作的对象

LXC 的意义，不只是它本身多强，而是它证明了“容器可以被做成产品”

# Docker




## 为什么后来是 Docker 更出圈，而不是 LXC?

### LXC

**它更像什么**

- 更像轻量级系统容器，偏向“把一个 Linux 环境装起来”。
- 对开发者而言，镜像分发、版本协作和统一工作流还不够顺手。
- 学习门槛和日常使用体验也更偏运维风格。

它更像一套能力集合，而不是完整协作平台

### Docker

**它抓住了什么**

- 抓住了 image、registry、Dockerfile、CLI 这套最容易传播的协作接口。
- 开发、测试、上线都能围绕同一种对象协作，所以生态起得很快。
- 真正让它出圈的，不只是隔离本身，而是交付体验第一次被做顺了。

标准化对象一旦形成，生态增长会比单点技术能力更快

Docker 赢得更快，不只是因为它“更会做容器”，而是因为它更会做协作

Part III • Docker & Ecosystem

# 从 LXC 到 Docker：真正改变了什么

很多同学以为 Docker 发明了容器；更准确地说，它把容器做成了可传播、可复用、可自动化的产品体验。

# 容器不是突然出现的

口先有隔离原语，再有好用产品，最后才有大规模编排

1

**1979**
**chroot**

先把进程视野限制在某个目录树里

→

2

**2000s**
**Namespaces**
**cgroups**

隔离“看见什么”和“能用多少”

→

3

**2008**
**LXC**

把零散能力拼成可配置容器工具

→

4

**2013**
**Docker**

镜像、仓库、CLI 与 Dockerfile 一起出圈

→

5

**2014**
**K8s**

容器从单机走向集群级编排

# Docker in a nutshell

# Docker 目标

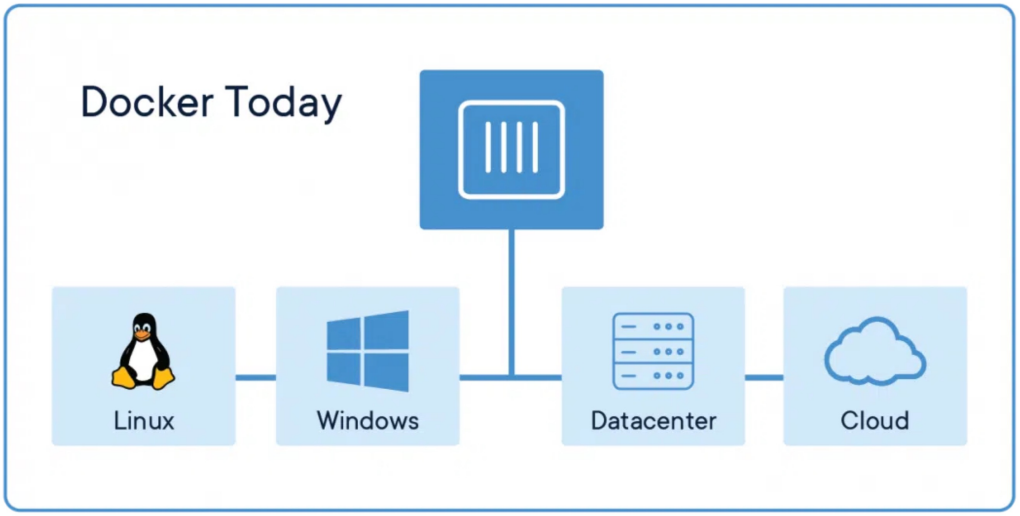
- 应用程序隔离
  - 通过定义资源进行沙盒隔离
  - 环境（不同进程）的清晰分离
- 统一环境处理应用程序生命周期
  - 所有应用使用相同的命令（运行，停止等）和相同的环境
- 透明且无缝的网络
  - Docker 网桥、DHCP、透明 NAT 等

**Docker 专注于应用程序，简化了它们的部署和执行**

# Docker 可靠且一致地部署应用程序

- 应用程序能在任何地方运行，且具有相同的行为
  - 无论是哪个 Linux 发行版
  - 无论是哪个内核版本
  - 物理或虚拟，云或非云

**Package Software into Standardized Units for Development, Shipment and Deployment**



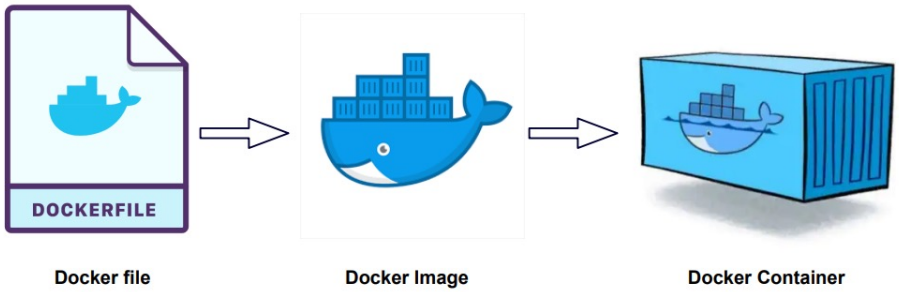
# Docker 不是什么

- 不是虚拟化引擎（利用现有基元，如 cgroups 和 namespaces）
- 不支持不同的内核
- 不利用硬件基元（例如 CPU 扩展）

# 镜像和容器

## 口镜像（image）

- 容器的不可变模版
- 可以从 Registry 拉取和推送
- 镜像名称的格式为 [registry/] [user/] name [:tag]
- tag 的默认值为 latest




## 口容器（container）

- 镜像的实例
- 可以启动、停止、重新启动
- 维护文件系统中的更改
- 可以从当前容器状态创建新的镜像（但不推荐，建议使用 Dockerfile）

# Docker Registry

- 类似于软件存储库，用于保存可用的 Docker 镜像
- 可以是私人的或公开的，如 Docker Hub


- 常用命令
  - 在 Registry 中搜索镜像

```
docker search <term>
```

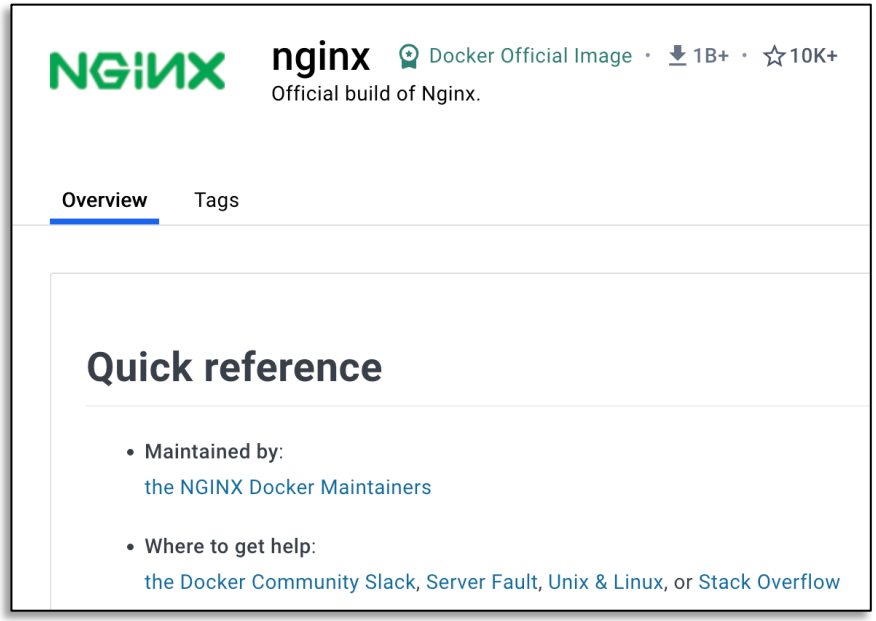
  - 从 Registry 下载或更新镜像（并在本地缓存）

```
docker pull <image>
```

  - 将镜像推送到 Registry

```
docker push <image>
```

# Docker Registry




```
                                    docker pull nginx:1.25.4
1.25.4: Pulling from library/nginx
26070551e657: Pull complete
5745264e68a8: Pull complete
6f07c61775e7: Pull complete
c4f29c7c07f7: Pull complete
5a639d96fbc1: Pull complete
3ba04a51efe1: Pull complete
716495aa6d18: Pull complete
Digest: sha256:9ff236ed47fe39cf1f0acf349d0e5137f8b8a6fd0b46e5117a
401010e56222e1
Status: Downloaded newer image for nginx:1.25.4
docker.io/library/nginx:1.25.4

What's Next?
  1. Sign in to your Docker account → docker login
  2. View a summary of image vulnerabilities and recommendations
→ docker scout quickview nginx:1.25.4
```

```
                              docker search nginx --limit 3
NAME                    DESCRIPTION                                     STARS     OFFICIAL
nginx                   Official build of Nginx.                        19767     [OK]
unit                    Official build of NGINX Unit: Universal Web …   26        [OK]
nginx/nginx-ingress     NGINX and _NGINX Plus Ingress Controllers fo…   89
```

# Docker Images (locally)

- Pull 下来的 images 缓存在本地，可以执行

- 常用命令：
  - 列出已下载的镜像

```
docker images
```

```
                                docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
nginx        1.25.4    48b4217efe5e   8 weeks ago     192MB
mysql        8.0.33    a5e6f938c138   8 months ago    587MB
```

  - 删除一个本地镜像

```
docker rmi <image> (or)
docker image rm <image>
```

  - 运行镜像

```
docker run [options] <image>
```

# Docker run：常见选项

- 运行容器时，有两个非常有用的选项
  - 暴露主机上容器的 TCP/UDP 端口
  - 将主机上的文件夹挂载到容器文件系统中

- docker run 命令的简单选项
  - 将容器 nginx 的 80 端口发布为主机上的 8080 端口

```
docker run -p 8080:80 nginx
```

  - 将本地目录/html挂载为容器 nginx 中的 /usr/nginx/html 目录

```
docker run -v /html:/usr/nginx/html mynginx
```

- 在短暂数据的情况下是有用的，对于持久性数据则不是
  - 当容器在特定的主机上启动时，才能访问到本地文件夹
  - 在这种情况下，应优先考虑网络共享

# 列出和检查运行容器的常用命令

- 显示正在运行的容器

```
docker ps
```

```
                                  % docker ps
CONTAINER ID   IMAGE          COMMAND
d453a3f286c2   mysql:8.0.33   "docker-entrypoint.s…"
b10ade99cc74   nginx:latest   "/docker-entrypoint.…"
```

- 显示所有容器（包括已终止的容器）

```
docker ps -a
```

```
                                  % docker ps -a
CONTAINER ID   IMAGE          COMMAND
d453a3f286c2   mysql:8.0.33   "docker-entrypoint.s…"
b10ade99cc74   nginx:latest   "/docker-entrypoint.…"
d87e3713998d   nginx          "/docker-entrypoint.…"
```

- 显示正在运行的容器的元数据
  - 返回一个包含容器所有信息的 JSON（当前IP/MAC地址、镜像名称等）

```
docker inspect <container>
```

```
                                % docker inspect mynginx
[
    {
        "Id": "d87e3713998d84573e5dc67bc260f1e0c6347c3c270a8a6d0f26ae73fc585c2a",
        "Created": "2024-04-14T07:46:44.471112716Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
            "nginx",
            "-g",
            "daemon off;"
        ],
```

  - 仅显示特定的元数据

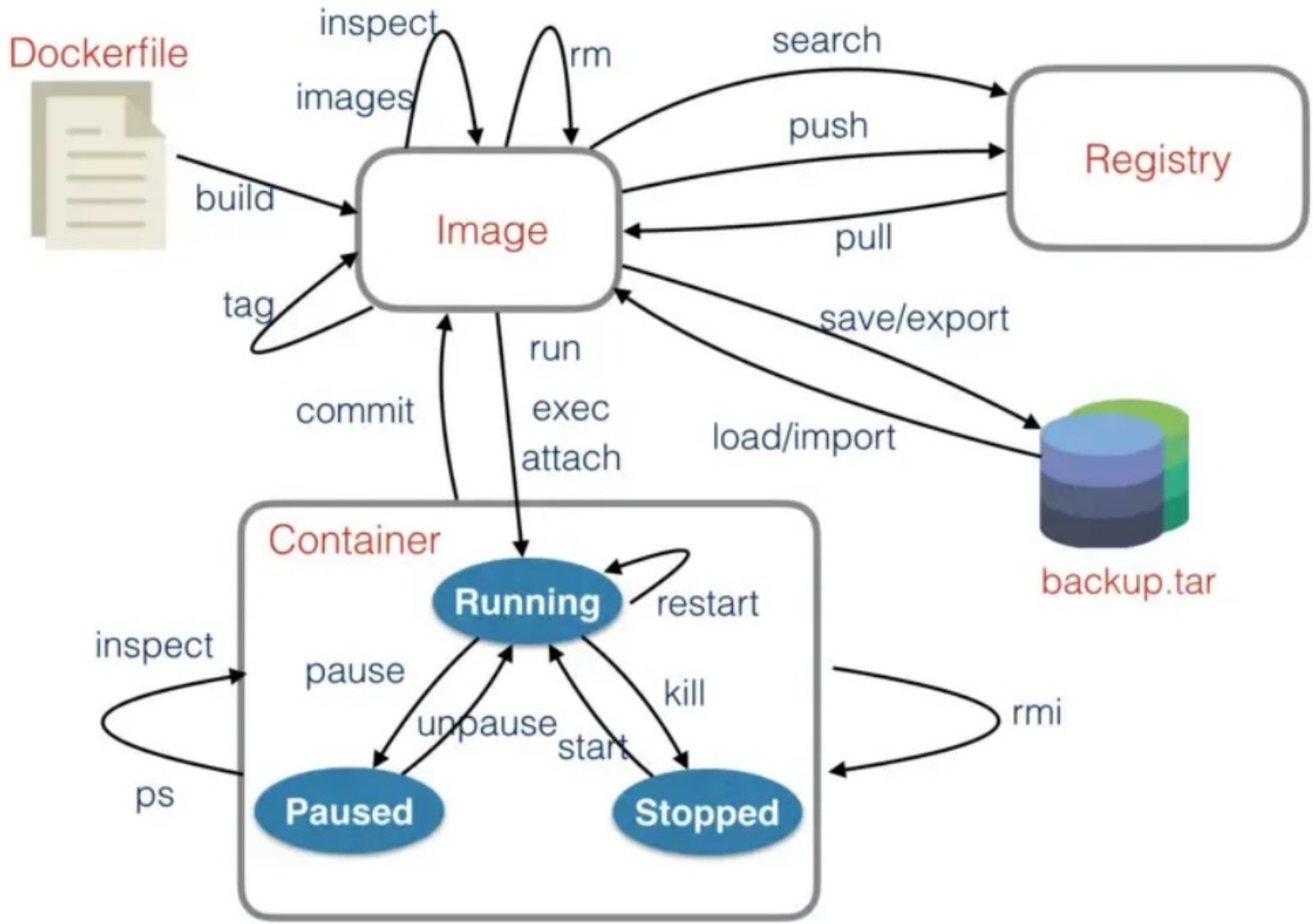
```
docker inspect --format='{{.Image}}' <container>
```

# 控制容器生命周期的常用命令

- 启动一个容器：docker run <container>
- 启动一个已停止的容器：docker start <container>
- 使用 SIGTERM 信号优雅地停止容器： docker stop <container>
- 使用 SIGKILL 信号杀死正在运行的容器：docker kill <container>

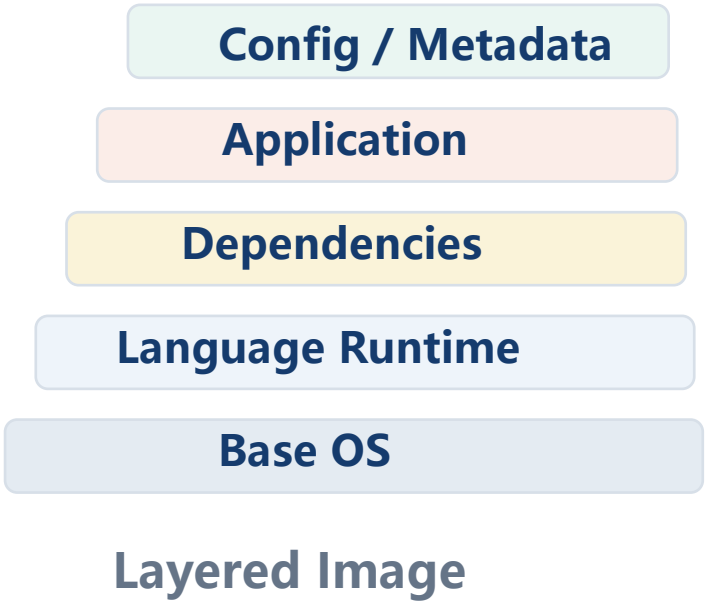
- docker rm <container>
  - 从"docker ps –a"列表中移除容器，但不移除镜像
  - 容器的状态将会丢失，阻止用户将其保存在另一个容器中
  - "docker start" 不再适用于已"移除"的容器
  - 它不会移除镜像；"docker run"仍然可用

# Docker 生命周期


Dockerfile
inspect
images
rm
search
push
pull
Registry
build
Image
tag
save/export
run
commit
exec
attach
load/import
backup.tar
Container
Running
restart
inspect
pause
kill
unpause
start
rmi
ps
Paused
Stopped

# 镜像分层

- 为什么同样的基础环境不用每次都从头重传
- 容器生态之所以适合持续交付，一个关键原因就是层的复用、缓存和回滚




**Reuse 共享层不用重复传**

多个镜像常共享底层基础层，网络传输和存储开销都因此下降。

**Cache 改动越靠上层，构建越快**

基础层不变时，新增应用代码不需要把整套运行时重新构建一遍。

**Recipe Dockerfile 让变化可描述**

把“怎么构建环境”写回配方，比在运行中的容器里手工修补可靠得多。

镜像分层让“软件环境”第一次像代码一样可复用、可缓存、可版本化。

# Python Web 应用例子

口一个 Python Web 应用的镜像，会怎么一层层长出来?

1 **基础层**

先选一个轻量基础镜像，例如 python:3.11-slim

→

2 **依赖层**

复制 requirements.txt，再安装 Flask、Gunicorn 等依赖

→

3 **应用层**

把你的 app.py、templates 和静态资源复制进去

→

4 **配置层**

声明端口、环境变量和默认启动命令

→

5 **运行实例**

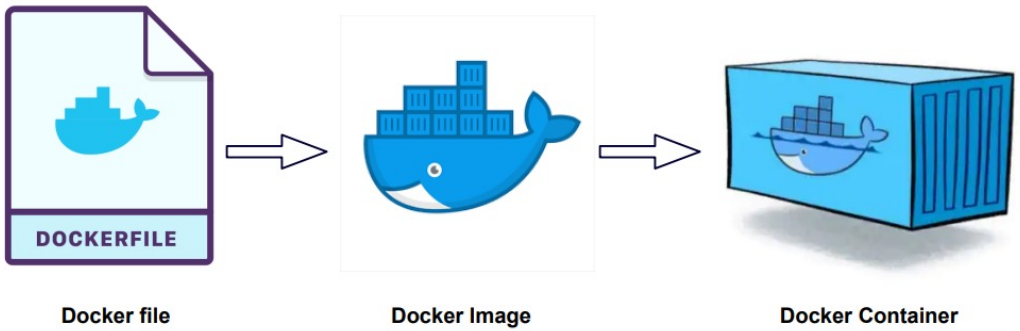
最后由同一份镜像，反复启动出多个容器副本

不必死记 Dockerfile，只要记住：镜像像千层盒饭，
改上层比改底座便宜得多

# Docker 自动构建

- Docker 允许从容器的组成元素开始创建一个镜像

- 用于创建镜像的配方保存在 Dockerfile 文件中




如果我们从一个极简的基础镜像（比如一个精简版的操作系统镜像）开始构建进程，就能创建一个只包含所需软件的镜像，避免那些在系统中会默认安装的软件

# Dockerfile example

## 包含一系列指令的文本文件，定义 Docker 镜像的构建步骤

```
##############################################################
# Dockerfile to build MongoDB container images, based on Ubuntu

# Set the base image to Ubuntu, with a specific tag (1910)
FROM Ubuntu:1910

# File Author / Maintainer
MAINTAINER Example McAuthor

# Update the repository sources list. Not strictly needed, but
good practice RUN apt-get update

################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the Ubuntu default repository sources list
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' |
tee /etc/apt/sources.list.d/mongodb.list
```

https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images

# Dockerfile example

```
# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db

##################### INSTALLATION END #####################

# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

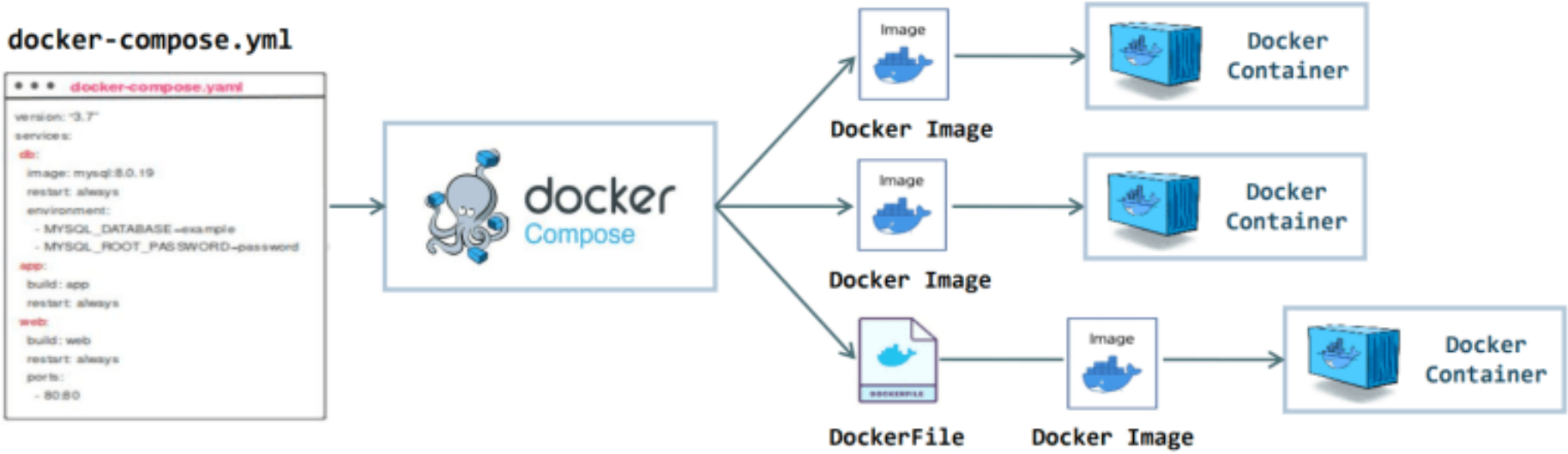
Now we have to build the image and give it a name:

```
docker build --pull -t mymongo .
```

`--pull`: Pull a newer version of the image
`-t mymongo`: name the new image "mymongo"
`.`: look for the "Dockerfile" in the current folder

# Docker Compose

- 一个用于定义和运行多容器 Docker 应用程序的工具
- 使用 Compose 文件（YAML 格式）来配置应用程序的各个服务及依赖关系
- 然后使用 Docker Compose 命令来启动、停止和重建服务，以及查看服务的状态和运行的服务




中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

# 谢谢

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn