



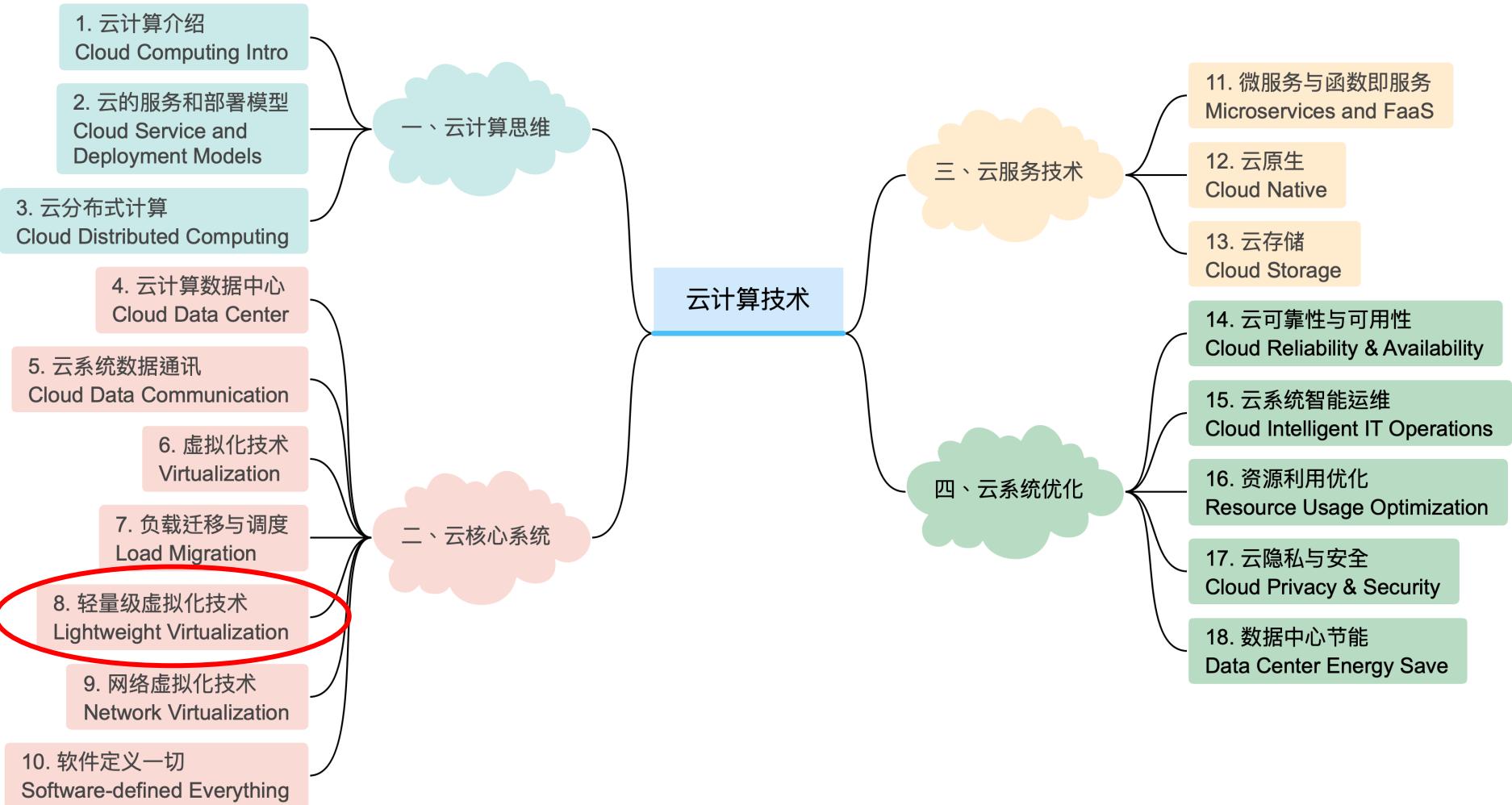
Lecture 08: 轻量级虚拟化

SSE316: 云计算技术
Cloud Computing Technologies

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn

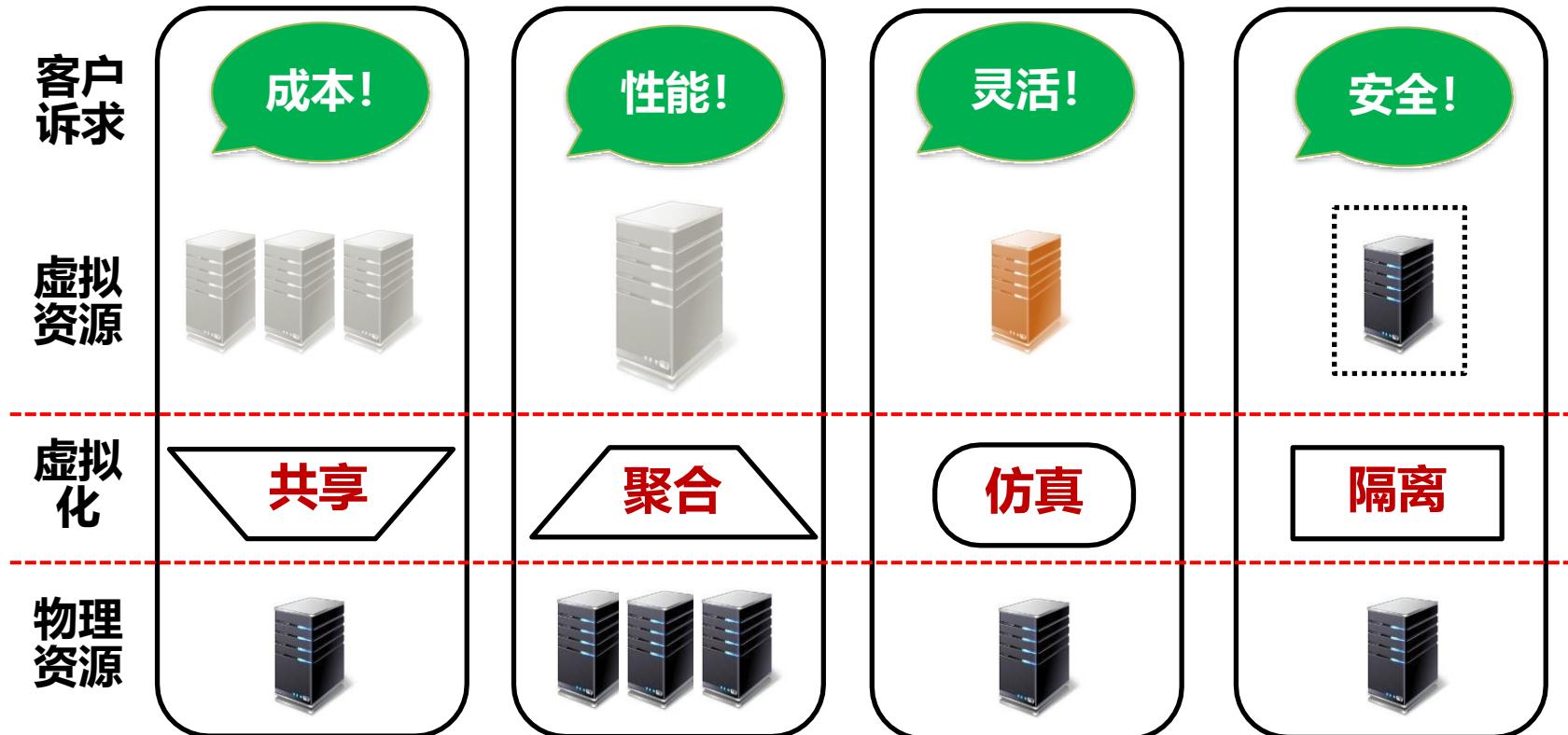


Today's topics

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

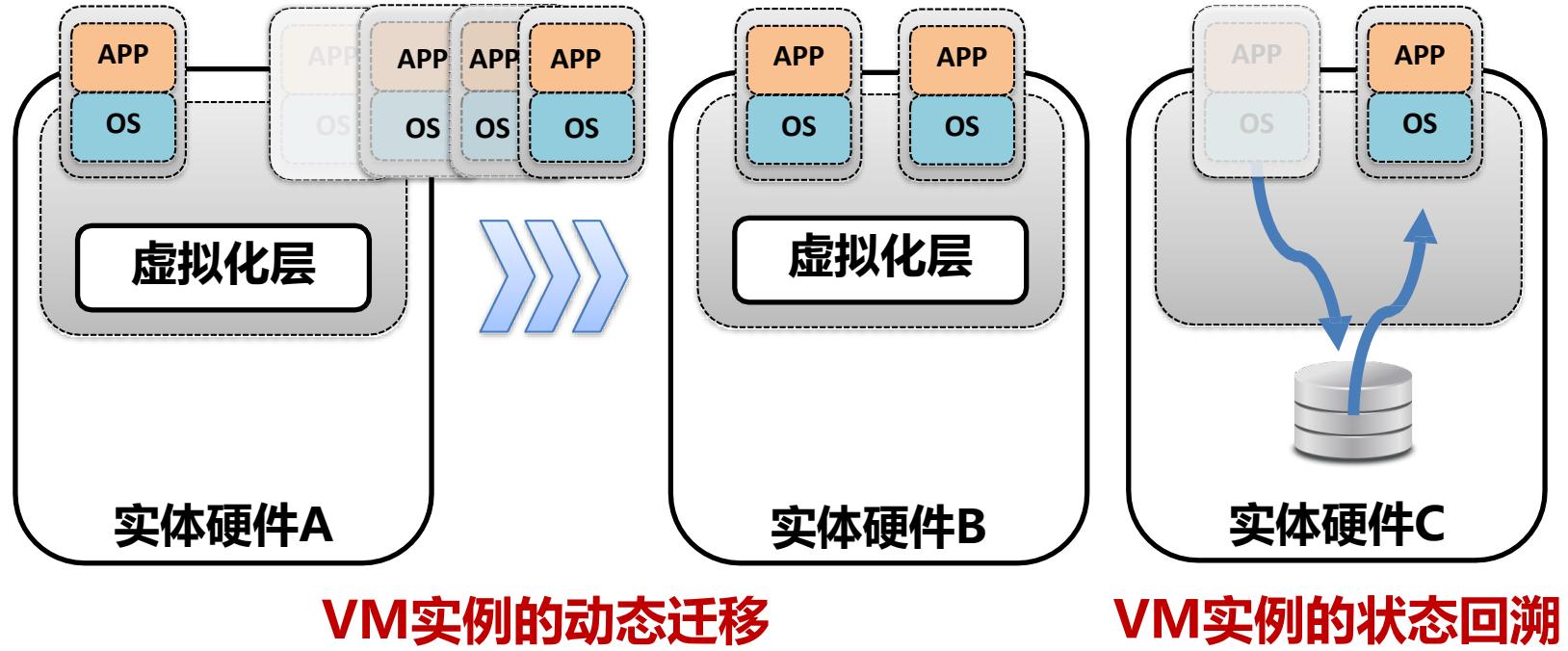
轻量级虚拟化

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



虚拟机状态可以数据形式存储，支持**时空层面**的灵活管理

基于全虚拟化的虚拟机

优势

- 与现有应用程序兼容
- 支持不同的操作系统
- 每个应用程序（即 VM）都有自己的执行环境，如内核版本、库
- 出色的隔离，有硬件机制支持（Memory, CPU）

劣势

- 执行客户操作系统的必要性
开销（Memory, CPU）
- 必须配置客户操作系统的每个实例并保持更新
- “运营”在现实生活中是一项不可忽视的成本
- 操作系统**启动时间长**（数十秒或更长）

轻量级虚拟化背后的理念

创建一个系统，保证计算机虚拟化的良好特性
(可扩展性、弹性、隔离性)，但资源消耗更少

口轻量级虚拟化适用于以下情况：

- 不需要完整的操作系统
- 经典虚拟机的**开销是不可接受的**
- 希望有一个能够快速部署、迁移和删除的隔离环境，且几乎没有开销
- 希望在垂直方向和水平方向进行**快速扩展**

轻量级虚拟化

口 使用操作系统级虚拟化或应用程序级虚拟化

- 不依赖硬件虚拟化
- 两者都是软件虚拟化技术

口 在操作系统级虚拟化中，VMM 就是 Linux 内核本身

- 不再需要专用的虚拟机监控程序

口 虚拟环境可以取代传统的虚拟机

- 虚拟环境也称为**虚拟专用服务器、监狱 (jails)、容器**
- 虚拟环境具有一定程度的资源管理和隔离功能，通常少于虚拟机所能实现的功能
- **应用程序能在这些虚拟隔离环境中执行**

轻量级虚拟化的要求

□ 物理资源的细粒度控制

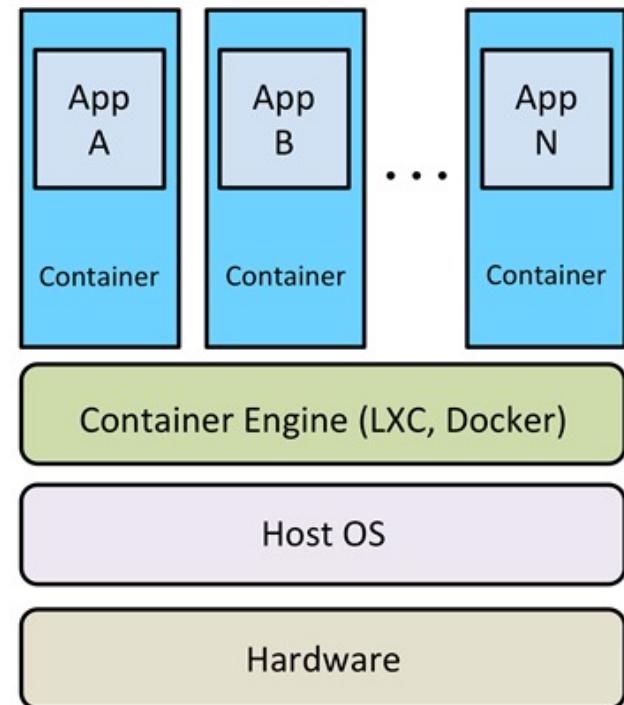
- 在不同环境中划分和控制资源，如 CPU 核心、RAM

□ 安全和隔离保证

- 每个虚拟环境分配给不同的应用程序/用户，并避免影响其他人

□ 将整个数据中心作为一个实体进行管理

- 例如使用云工具包，简化操作并提高效率
- 能够将轻量级虚拟化与云工具包集成，以便根据请求灵活部署 VM、容器等



如何实现轻量级虚拟化?

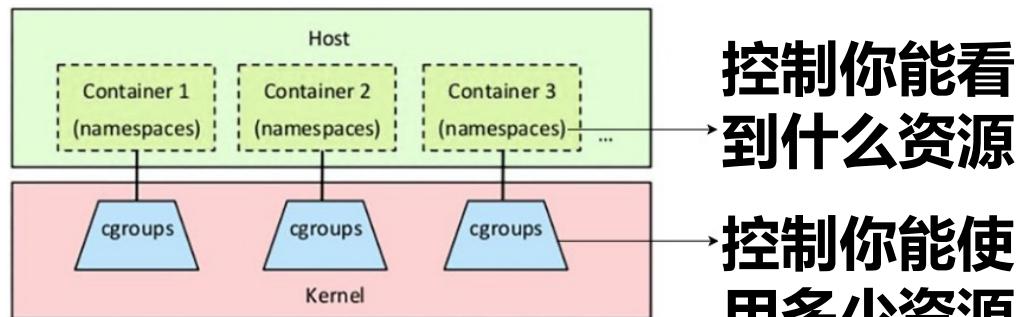
cgroups
控制组

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

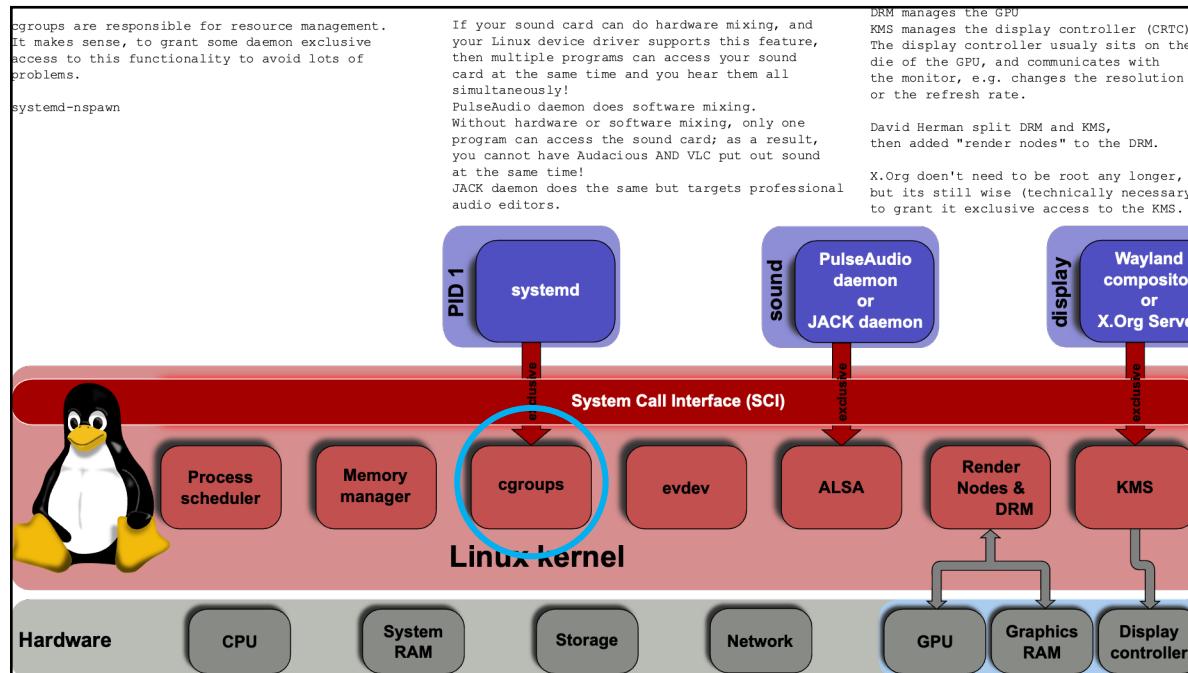
- CPU and Disk quotas
- Network isolation
- I/O rate and Memory limit
- File system isolation
- ...



Linux 控制组和命名空间

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

Linux cgroups

由两部分组成

- Kernel feature: 内核中实现进程组对资源的使用限制或隔离
- User-space tools: 处理内核控制组机制的用户空间工具

Kernel feature

User-space Tools

与 nice 或 cpulimit 相比，其优势在于可限制一组进程，而不仅是单个进程

- ✓ nice 命令用于调整单个进程的优先级
- ✓ cpulimit 命令用于限制单个进程的 CPU 使用率

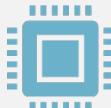
cgroups 特征



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

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
```

简单的循环加程序

```
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
```

设置两个 cgroups

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

限制 CPU 使用配额

```
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 &
```

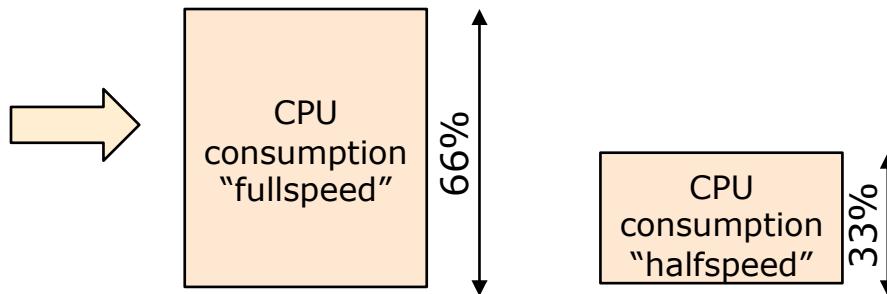
运行程序并添加到相应控制组

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
```

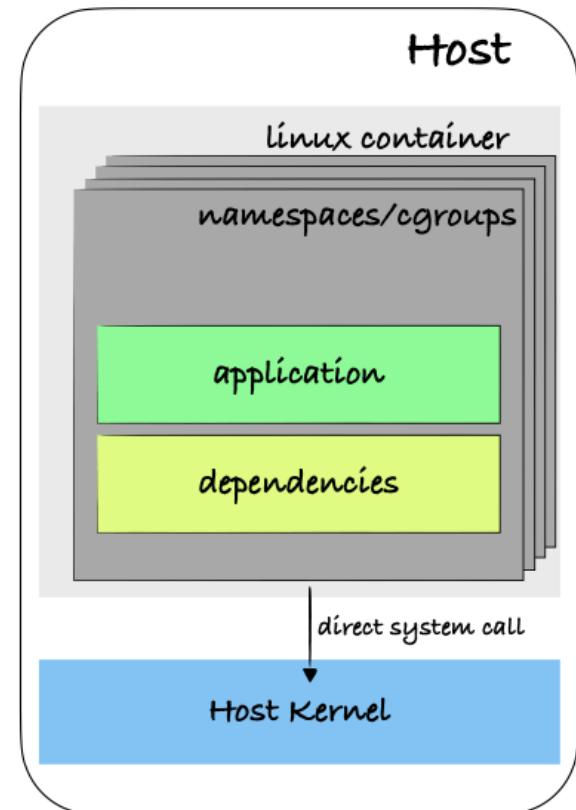


Linux namespaces

- 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



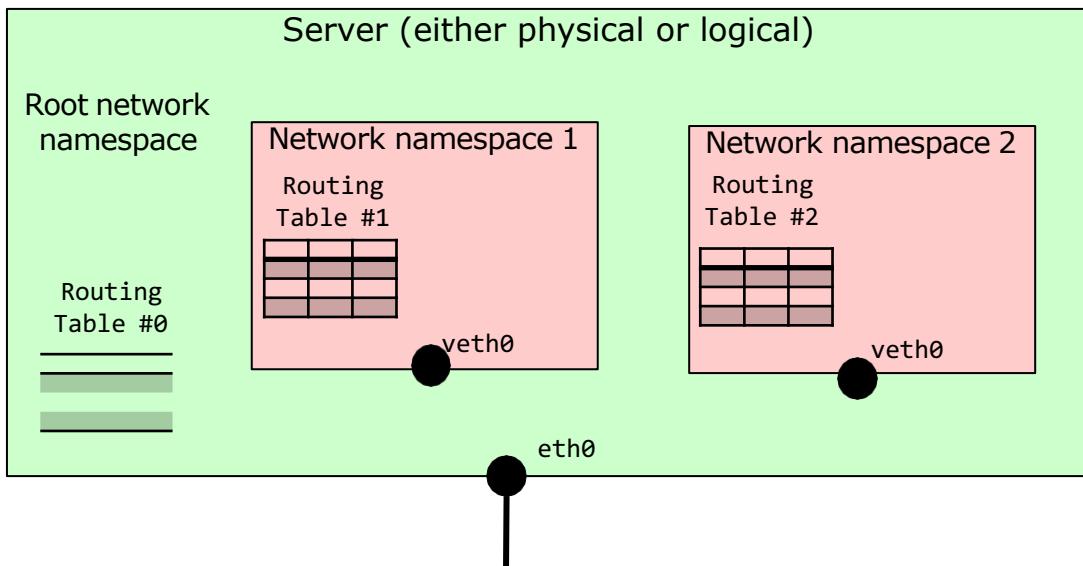
示例：不同命名空间的内核分区

□ 内核必须为不同空间创建相同数据结构的不同独立实例

- 例如，网络命名空间的路由表

□ 每个对象（如进程）与给定的命名空间关联

- 可访问属于同一空间的对象
- 当请求访问给定的数据结构时，内核使用 namespaceid 从正确的结构中检索数据



Kernel data structures

	ns root	ns #1	ns #2
Routing table	RT #0	RT #1	RT #2
Interface list	eth0 (Intel, gigabit Ethernet)	veth0 (Virtual Ethernet)	veth0 (Virtual Ethernet)

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

#1: Process namespace

□ 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
- 进程可能有子进程，比如调用 fork()

□ PID 命名空间启用多个“嵌套”进程树

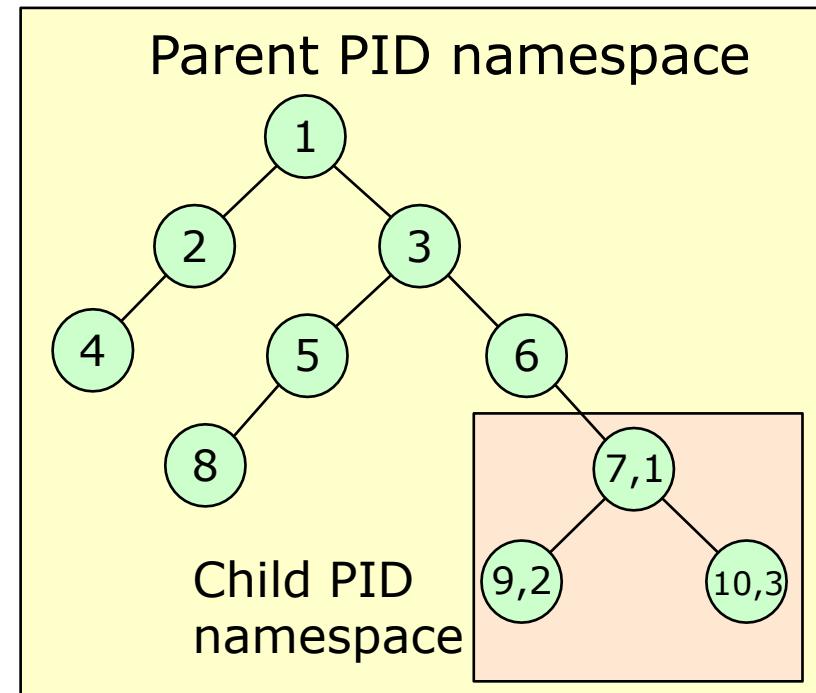
- 每个进程树表示一组完全独立的流程
- 进程树的进程不知道其他并行进程树的存在，因此无法检查或杀死其他进程树中的进程

```
warda@warda:~$ pstree -u
systemd--ModemManager---2*[{ModemManager}]
          |----NetworkManager---2*[{NetworkManager}]
          |----accounts-daemon---2*[{accounts-daemon}]
          |----acpid
          |----avahi-daemon(avahi)---avahi-daemon
          |----colord(colord)---2*[{colord}]
          |----containerd---8*[{containerd}]
          |----cron
          |----cups-browsed---2*[{cups-browsed}]
          |----cupsd---4*[dbus(lp)]
          |----dbus-daemon(messagebus)
          |----fwupd---4*[{fwupd}]
          |----gdm3---gdm-session-wor---gdm-x-session(warda)---Xorg---5*[{Xorg}]
          |          |----gnome-session-b---ssh+---2*[{gdm-x-session}]
          |          |----2*[{gdm-session-wor}]
          |          |----2*[{gdm3}]
          |          |----gnome-keyring-d(warda)---3*[{gnome-keyring-d}]
          |          |----2*[kerneloops(kernoops)]
          |          |----networkd-dispat
          |          |----nmbd
          |          |----polkitd---2*[{polkitd}]
```

#1: Process namespace

口 PID 命名空间允许一个进程创建一个新的树，且拥有自己的 PID=1 进程

- 第一个进程仍然在原始树中，并且了解其子进程
- 父命名空间中的进程能查看子命名空间中的进程，就像它们是父命名空间中的任何其他进程一样
- 子进程成为其自己进程树的根，且对源进程树一无所知



#1: Process namespace

一个进程现在可以有多个 PID 与之关联，它所属的每个命名空间都有一个 PID

```
https://elixir.bootlin.com/linux/latest/source/include/linux/pid.h

/*
 * struct upid is used to get the id of the struct pid, as it is
 * seen in particular namespace. Later the struct pid is found with
 * find_pid_ns() using the int nr and struct pid_namespace *ns.
 */

struct upid {
    int nr;                      //PID value
    struct pid_namespace *ns;    //Namespace where this is relevant
};

struct pid
{
    refcount_t count;
    unsigned int level;
    /* lists of tasks that use this pid */
    struct hlist_head tasks[PIDTYPE_MAX];
    /* wait queue for pidfd notifications */
    wait_queue_head_t wait_pidfd;
    struct rcu_head rcu;
    struct upid numbers[1]; //Array of uPIDs
};
```

#1: Process namespace

Source code

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static int child_fn(void *arg) {
    printf("Child PID: %ld\n", (long) getpid());
    printf("Parent PID: %ld\n", (long) getppid());
    sleep(1000);
    return 0;
}

int main() {
    pid_t child_pid = clone(child_fn, child_stack + sizeof(child_stack),
                           CLONE_NEWPID | SIGCHLD, NULL);
    printf("clone() = %ld\n", (long) child_pid);
    printf("Main PID: %ld\n", (long) getpid());

    waitpid(child_pid, NULL, 0);
    return 0;
}
```

Output example

```
netlab@VM:~/sample/bin$ sudo
./sample clone() = 10381
Main PID: 10380
Child PID: 1
Parent PID: 0
```

```
netlab@VM:~/sample/bin$ ps ax |grep sample
10379 pts/1    S+        0:00 sudo ./sample
10380 pts/1    S+        0:00 ./sample
10381 pts/1    S+        0:00 ./sample
```

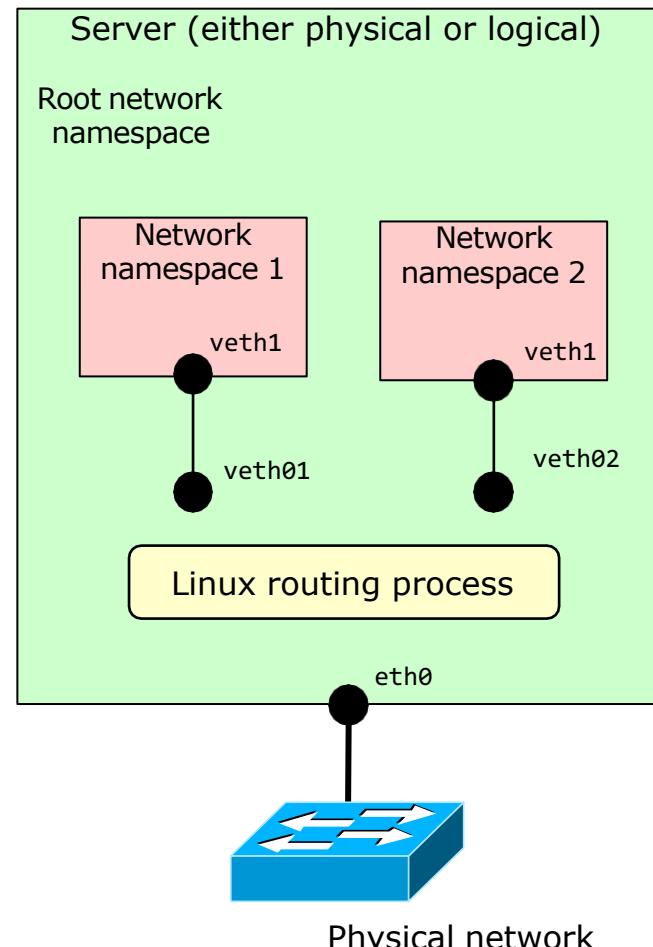
#2: Network namespace

□ 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



#2: Network namespace

□ Creation commands

```
sudo ip netns add <netnsname>
```

Create a new network namespace named <netnsname> (e.g., ns1)

```
sudo ip link add veth0-root type veth peer name veth0-ns
```

```
sudo ip link set veth0-ns netns <netnsname>
```

(or)

Create the first virtual Ethernet link (i.e., 'veth' pair). The two endpoints are virtual interfaces called 'veth0-root' (in the root namespace) and 'veth0-ns' (in the new namespace)

```
sudo ip link add name veth0-root type veth peer name veth0-ns netns <netnsname>
```

Push the 'veth0-ns' endpoint in the new namespace

```
sudo ip link add name veth0-root type veth peer name veth0-ns netns <pid>
```

Alternative version, when the namespace has already been created by a process. <pid> is the Process ID of the process in the child namespace as observed by the parent

#2: Network namespace

口上述命令在两个命名空间之间建立类似管道的连接

- 父空间保留 veth0 根设备，并将 veth0 ns 设备传递给子空间
- 任何进入其中一端的消息都会到达另一端，类似两个真实节点用以太网连接
- 此虚拟以太网连接的两侧都可以提供 IP 地址



#2: Network namespace

口一些例子

```
netlab@VM:~$ sudo ip netns add ns1
```

```
netlab@VM:~$ sudo ip link add name veth0 type veth peer name veth1 netns ns1
```

```
netlab@VM:~$ ip link  
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000  
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff  
3: veth0@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether 6e:ce:ab:3d:19:9a brd ff:ff:ff:ff:ff:ff link-netns ns1
```

```
netlab@VM:~$ sudo ip netns exec ns1 ip link
```

```
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
2: veth1@if6: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether 9a:a4:a6:77:fb:94 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

```
netlab@VM:~$ sudo ip link add name veth2 type veth peer name veth3
```

```
netlab@VM:~$ ip link  
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000  
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff  
3: veth0@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether 6e:ce:ab:3d:19:9a brd ff:ff:ff:ff:ff:ff link-netns ns1  
4: veth3@veth2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether 92:c7:93:32:db:21 brd ff:ff:ff:ff:ff:ff  
5: veth2@veth3: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether c2:29:ab:43:85:07 brd ff:ff:ff:ff:ff:ff
```

The veth is created with one end in the root namespace, the other in namespace ns1

The veth is created entirely in the root namespace, hence both ends are here. We need another command "sudo ip link set veth3 netns ns1" to push it in namespace

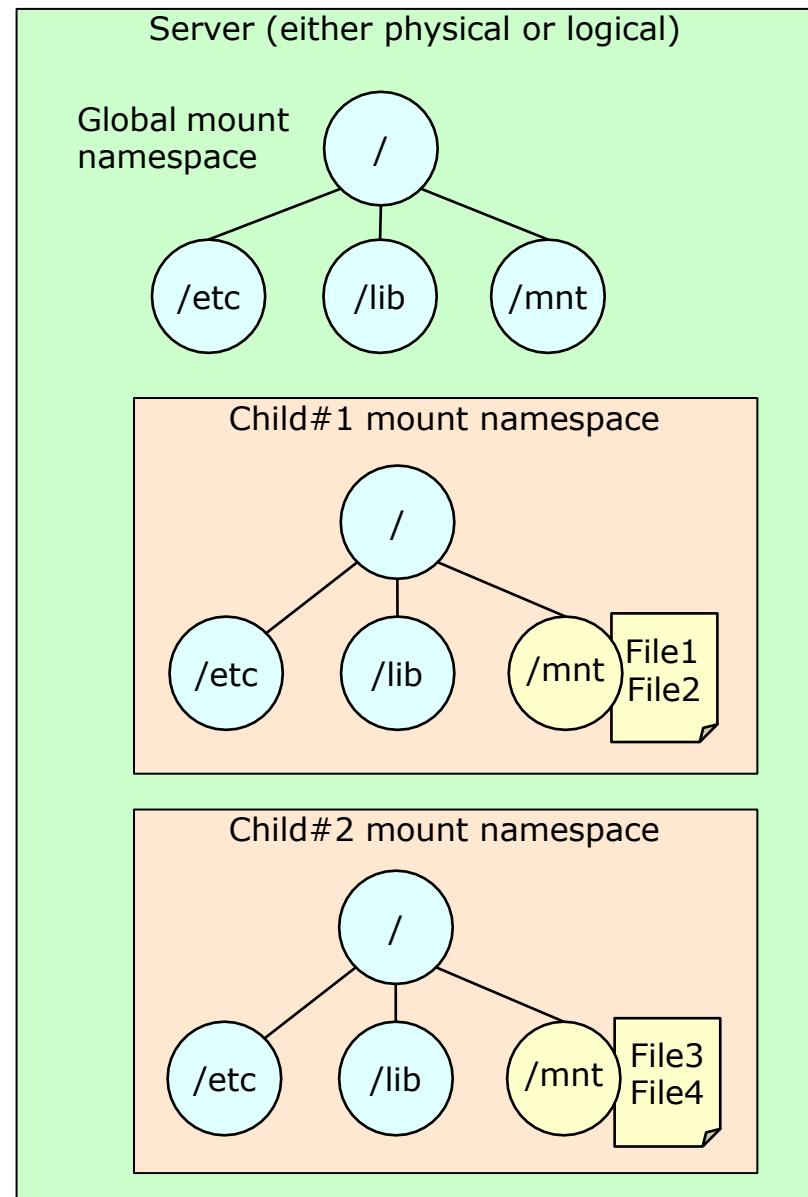
#3: Mount namespace

□ 允许创建具有所需结构的全新文件系统

- 例如磁盘分区(以及权限)、装载点、文件夹等

□ 类似 chroot，但更灵活

- chroot 创建了一个全新的(空的)根，必须从头开始填充
- 装载命名空间最初克隆父进程的装载数据结构
- 然后，此新进程可在**不影响其他进程的情况下更改其装载点**



#3: Mount namespace

口例子

```
# Become superuser
netlab@VM:~$ sudo su

# Start a shell in a new mount namespace
root@VM:/home/netlab# unshare --mount

# Mount the /usr/bin/ folder into /mnt/
root@VM:/home/netlab# mount --bind /usr/bin/ /mnt/

# Look for file 'bash' in /mnt
root@VM:/home/netlab# ls -la /mnt/bash
-rwxr-xr-x 1 root root 1166912 May  3 18:31 /mnt/bash

# Create a folder 'testfolder' under /mnt
root@VM:/mnt# mkdir /mnt/testfolder

# Exit the shell, and hence the mount namespace
root@VM:/mnt# exit
logout

# List what happened to the newly created folder in the root namespace
root@VM:/home/netlab# ls -la /usr/bin/testfolder
total 8
drwxr-xr-x  2 root root 4096 Oct 29 16:36 .
drwxr-xr-x 25 root root 4096 Oct 29 16:36 ..

root@VM:/home/netlab# ls -la /mnt/testfolder
ls: cannot access '/mnt/testfolder': No such file or directory
```

#4: UTS namespace

口 UTS 命名空间提供了两个系统标识符的隔离

- 主机名 (nodename)
- NIS 域名 (Network Information Service domainname)
- 应用： 每个容器通常拥有独立的 UTS namespace，使得容器内部的主机名与宿主机或其他容器不同，避免冲突

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/utsname.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static void print_utsname() {
    struct utsname utsname;

    // Struct utsname contains other info such as kernel version, etc
    // However the UTS namespace isolates only domainname and nodename
    // http://man7.org/linux/man-pages/man2/uname.2.html

    uname(&utsname);
    printf("utsname.domainname: %s\n", utsname.domainname);
    printf("utsname.nodename:   %s\n", utsname.nodename);
}

static int child_fn(void *arg) {
    printf("\nNew UTS namespace:\n");
    print_utsname();
}

int main() {
    printf("\nOriginal UTS namespace:\n");
    print_utsname();

    pid_t child_pid = clone(child_fn, child_stack + sizeof(child_stack),
                           CLONE_NEWUTS | SIGCHLD, NULL);

    sleep(1);

    printf("\nOriginal UTS namespace:\n");
    print_utsname();

    waitpid(child_pid, NULL, 0);

    return 0;
}
```

#4: UTS namespace

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/utsname.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static void print_utsname() {
    struct utsname utsname;

    // Struct utsname contains other info such as kernel version, etc
    // However the UTS namespace isolates only domainname and nodename
    // http://man7.org/linux/man-pages/man2/uname.2.html

    uname(&utsname);
    printf("utsname.domainname: %s\n", utsname.domainname);
    printf("utsname.nodename: %s\n", utsname.nodename);
}

static int child_fn(void *arg) {
    printf("\nNew UTS namespace:\n");
    print_utsname();
}

int main() {
    printf("\nOriginal\n");
    print_utsname();

    pid_t child_pid = fork();
    if (child_pid == -1) {
        perror("fork");
        exit(1);
    }

    if (child_pid == 0) {
        sleep(1);

        printf("\nOriginal\n");
        print_utsname();

        waitpid(child_pid,
                &status, WNOHANG);
        return 0;
    }

    printf("Changing data inside new UTS namespace\n");

    sethostname("CustomVM", strlen("CustomVM"));
    setdomainname("CustomDomain", strlen("CustomDomain"));

    printf("\nNew UTS namespace:\n");
    print_utsname();
    return 0;
}
```

root@VM:/home/netlab# ./uts_namespace_sample

Original UTS namespace:
utsname.domainname: (none)
utsname.nodename: VM

New UTS namespace:
utsname.domainname: (none)
utsname.nodename: VM

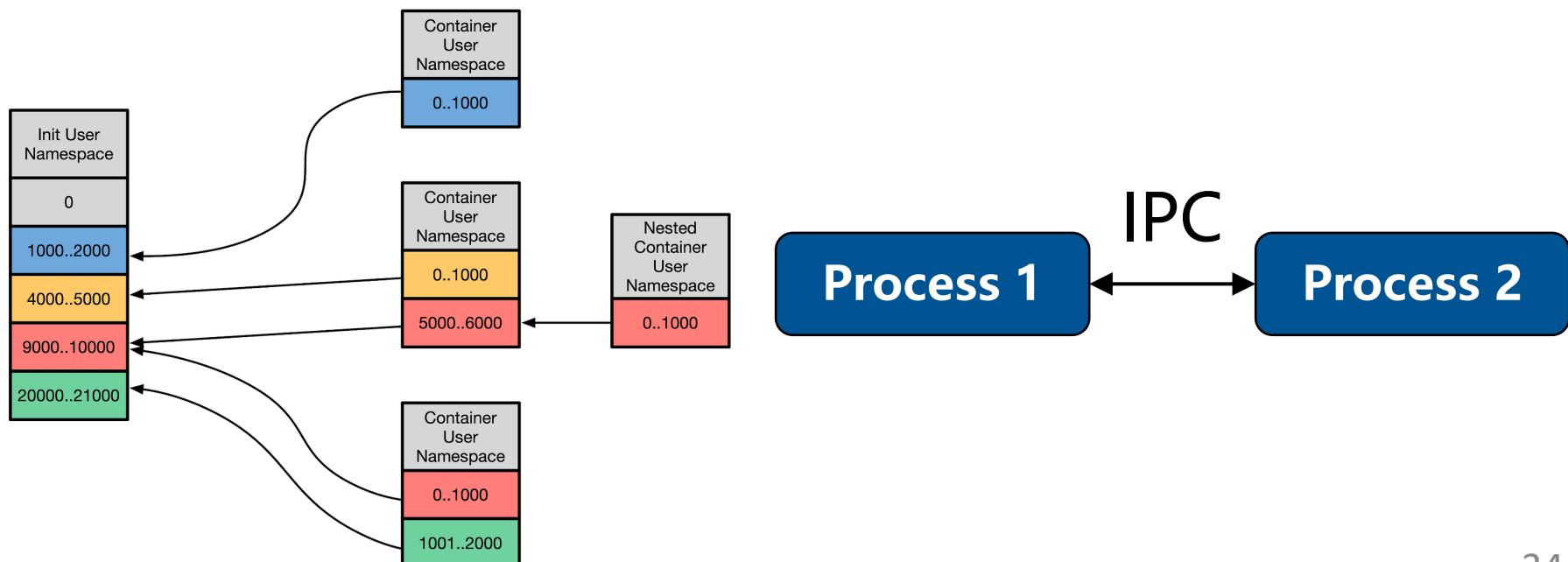
Changing data inside new UTS namespace

New UTS namespace:
utsname.domainname: CustomDomain
utsname.nodename: CustomVM

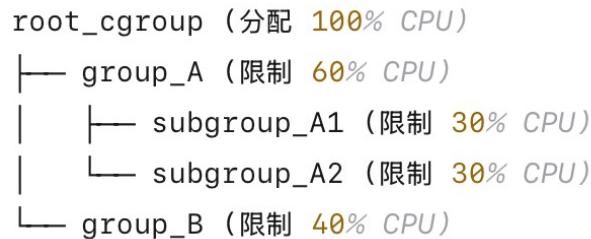
Original UTS namespace:
utsname.domainname: (none)
utsname.nodename: VM

#5, #6: User namespace, IPC namespace

- 用户命名空间：允许进程在命名空间内拥有 root 权限，而不允许其访问命名空间外的进程
- IPC (Inter-Process Communication) 命名空间：为隔离进程创建专用进程间通信资源（例如 System V IPC、POSIX 消息）



#7: cgroup namespace



□ cgroup 命名空间用于隔离 cgroup 根目录的视图

- 命名空间内的进程只能看到自己的 cgroup 层次结构，不能看到系统上其他的 cgroup 信息
- 让容器内的进程认为自己是在一个隔离的环境中运行

□ 例如，两个进程共享相同的 cgroup 命名空间

- `#cat /proc/self/cgroup` (显示当前进程所属的cgroup信息)
- `0:::/batchjobs/container_id1` (该进程在cgroup层次结构中的位置为 `/batchjobs/container_id1`)

□ 取消共享新命名空间后，视图变为

- `#cat /proc/self/cgroup`
- `0::/` (该进程在cgroup层次结构中的位置变为根)

cgroups 和 namespaces 的局限

□ 非常灵活，但**难以使用**

- 打开一个完整的容器需要大量的命令/脚本

□ 无法保证应用程序的**可移植性**

- 虚拟机可以在任何服务器上打包和启动
- “打包”一个独立的应用程序并使其在另一台服务器上运行并不是一种简单的方法

□ 仅针对单个服务器上的“虚拟化”

- 不能按原样用于处理整个数据中心

cgroups 和 namespaces 总结

口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

口然而，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 Linux 容器和 Docker，
降低配置复杂性并添加新功能

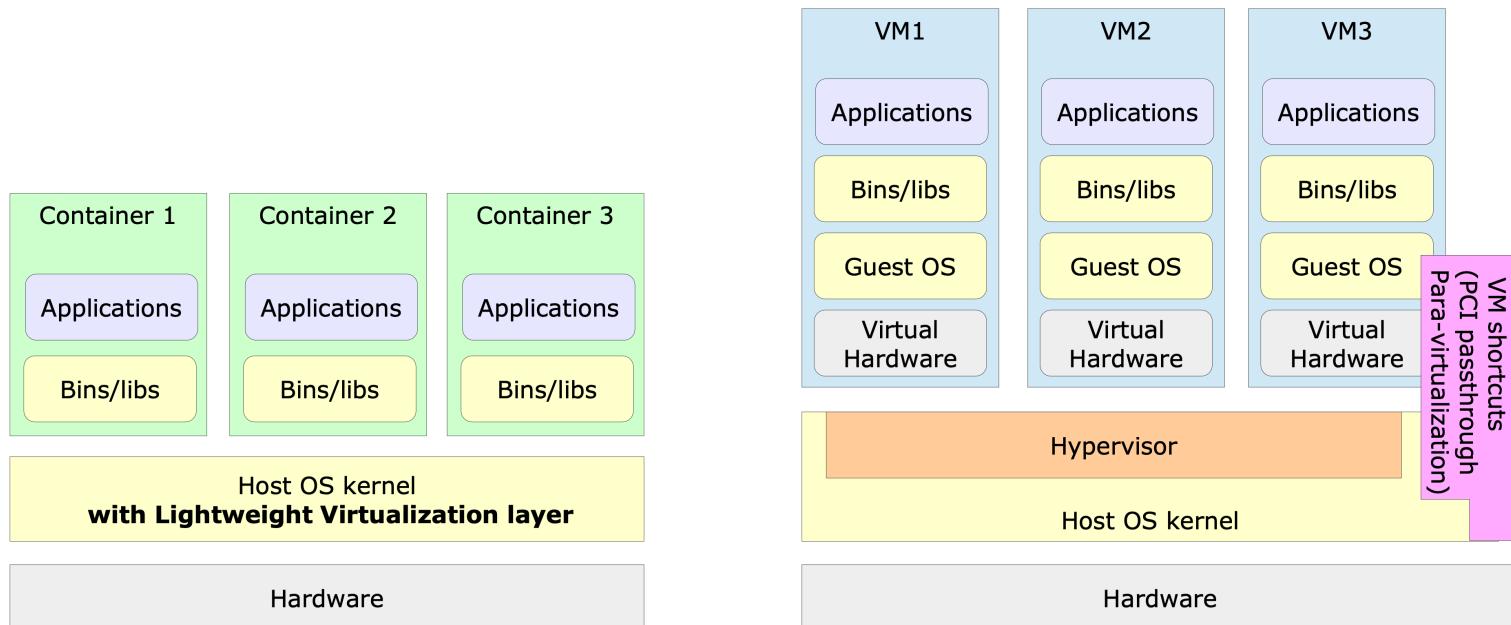
Containers overview

□ 容器提供了轻量级虚拟化

- 允许隔离进程和资源，且没有完全虚拟化的复杂性

□ Linux 容器是一种操作系统级虚拟化方法

- 用于在单个主机上运行多个独立的 Linux 系统（容器）
- Linux 内核在所有容器中共享



容器

□ What they do

- 将进程分组在独立的容器中 (可以向其分配不同的资源)
- 与主机共享相同的操作系统
- 在容器内：像一个虚拟机
- 在容器外：像正常的进程

□ What they don't

- 不模仿硬件
- 不运行不同的内核或 OS (与主机共享相同的内核)
- 安全性不是理所当然的功能

Containers vs VMs

□ 容器

- **比虚拟机快**
 - 启动、冻结、删除和编排
- **比虚拟机轻**
 - 更少的 CPU 和内存，没有虚拟化开销（例如，指令仿真）
 - 容器虚拟化技术实际上可以实现**与本机执行相同的性能**

□ 虚拟机

- 提供更好的隔离（例如，还可以防止内核漏洞）
- 由于攻击点有限（VMM 通常非常小），因此安全性更好
- 系统兼容性，允许使用不同的操作系统
- 网络配置更复杂且强大，能模拟出完整的网络环境

容器特征

□ 敏捷的应用程序创建和部署

- 与虚拟机镜像相比，容器镜像创建更加容易和高效

□ 可移植性

- 在 Ubuntu、RHEL、CoreOS 和其他任何地方运行

□ 以应用程序为中心的管理

- 提高了从在虚拟硬件上运行操作系统到使用逻辑资源在操作系统上运行应用程序的抽象级别

□ 资源隔离

- 可预测的应用程序性能

□ 资源利用率

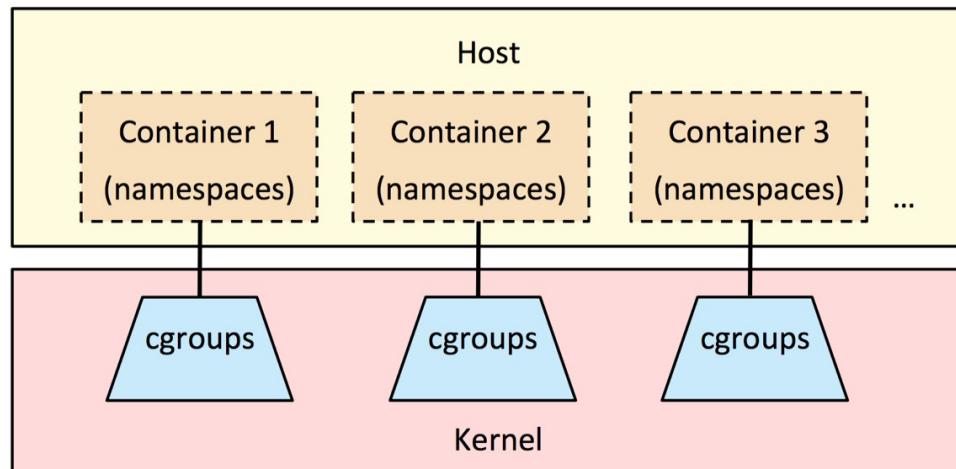
- 高效率和高密度

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

Container = combination of namespaces & cgroups



Why LXC?

□ 使用 LXC 比使用单个组件更容易，例如

- cgroups 仅提供**资源管理**
- 如果还想要**隔离**，还得添加 namespaces
- 想要**安全性**则需要添加 Apparmor 和/或 SELinux
- 还有**在线迁移**等...

□ LXC 并未提供所有功能，但以 (相对) 友好的方式提供部分功能

- LXC 使用**配置文件**指定所需功能列表来配置容器
- 可以将不同的资源分配给不同的容器
- 可以在不同的容器（和主机）之间进行**不同级别的隔离**

LXC 局限

口检查点和迁移 (Checkpointing and migration)

- 原生 Linux 内核中缺少的功能
- 需要借助其他工具如 CRIU，但也不能 100% 正常工作

口资源隔离 (Resource isolation)

- 必须配置容器使其“看不见”资源
- 并非总是能得到保证；例如，容器的资源配置可能会受到其他行为的影响

口不保证可移植性 (Portability)

- 在第一台服务器上创建的 LXC 不能移植到其他服务器
- 而虚拟机可移植（至少在具有相同 VMM 的服务器之间）

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

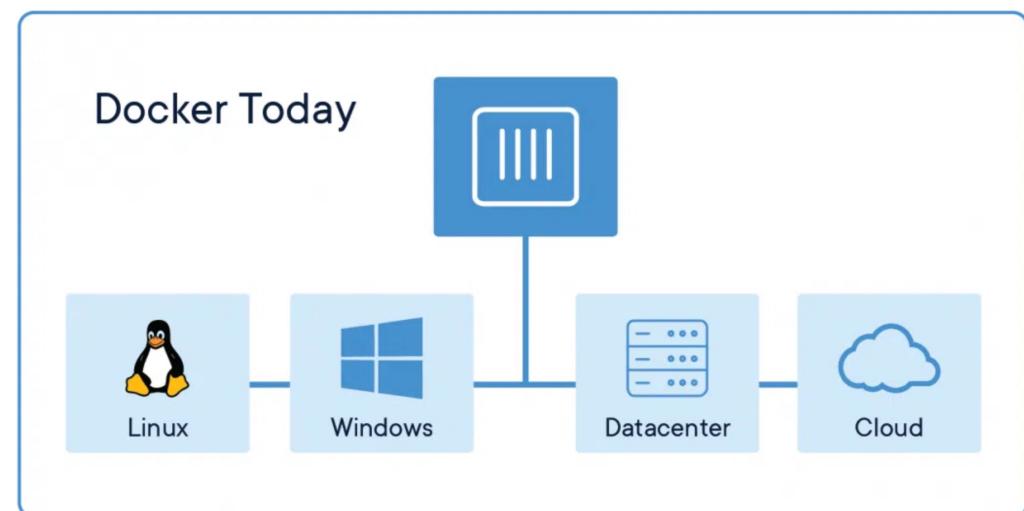
Docker 专注于应用程序，简化了它们的部署和执行

Docker 可靠且一致地部署应用程序

口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,
Shipmment 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）



关注点分离

Developer (inside the container)

- App
- Libraries and dependencies
- “Manifest” (e.g., used TCP/UDP ports)
- [Code]
- [Data]

DevOps (outside the container)

- Logging
- Remote access
- Network configuration
- Monitoring

Docker vs LXC

口应用程序隔离

- 均使用 cgroups 和 namespaces 来隔离应用程序和系统
- Docker 更专注于单个应用程序的隔离
- LXC 则更像是轻型的虚拟机，可以运行完整的 Linux 系统

口可移植性

- Docker 具有更高的可移植性
- LXC 可移植性相对较差，依赖于特定的 Linux 内核特性

口用户体验

- Docker 有一个用户友好的命令行界面和 API，使得创建和管理容器非常容易
- 而 LXC 的用户界面相对更为复杂

Docker: 与 LXC 相比的额外功能

- Application portability
- Union file system
- Automatic build
- Focus on running applications
- Versioning
- Component re-use
- Sharing (Docker server)
- Better documentation and ecosystem
- Integration with OpenStack, Kubernetes and all cloud vendors
- Resource management and isolation (already available also in LXC)

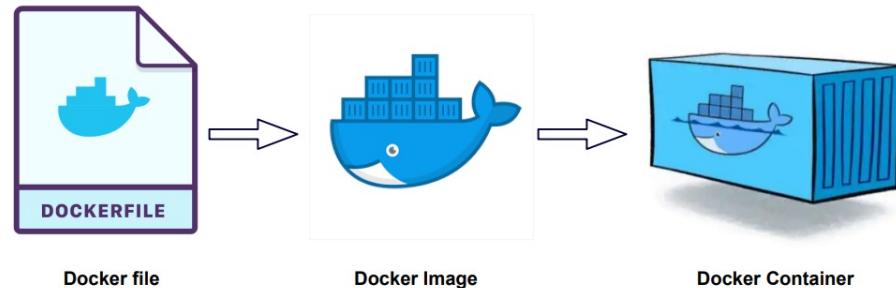
镜像和容器

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

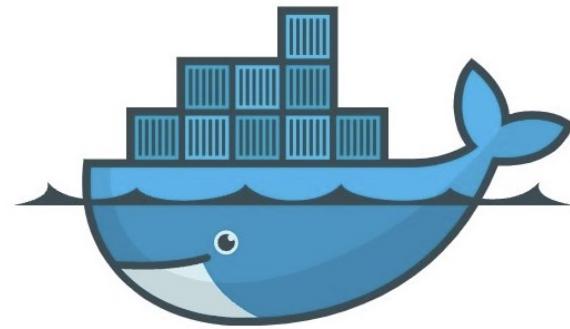
口容器 (container)

- 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](#)



Docker Registry

口常用命令

- 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

The screenshot shows the Docker Hub page for the official Nginx image. At the top, there's a large green "NGINX" logo. Below it, the image name "nginx" is displayed with a "Docker Official Image" badge, a download count of "1B+", and a star rating of "10K+". A subtext says "Official build of Nginx." Below this, there are two tabs: "Overview" (which is underlined in blue) and "Tags". In the "Quick reference" section, there are two bullet points: "Maintained by: the NGINX Docker Maintainers" and "Where to get help: the Docker Community Slack, Server Fault, Unix & Linux, or Stack Overflow".

```
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

口运行容器时，有两个非常有用的选择

- 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

口 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`

口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

口显示正在运行的容器

```
docker ps
```

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

口显示所有容器（包括已终止的容器）

```
docker ps -a
```

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

口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

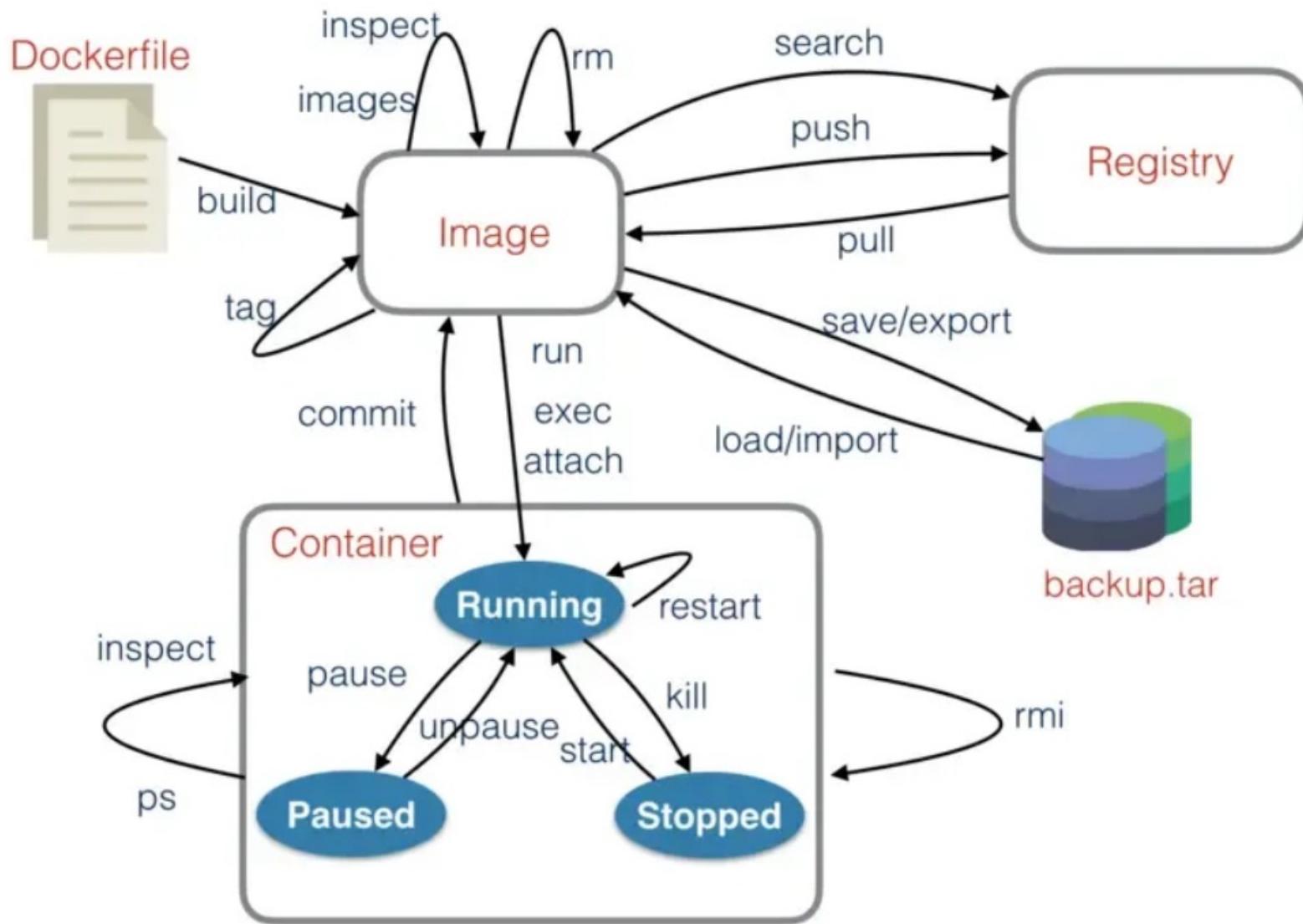
```
docker inspect --format='{{.Image}}' <container>
```

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



用于交互和调试的常用命令

口在现有的容器中运行一个命令，例如，启动一个 shell

```
docker exec <container> <command>
```

```
docker exec -it <container> bash
```

口查看容器中的日志 (stdout)

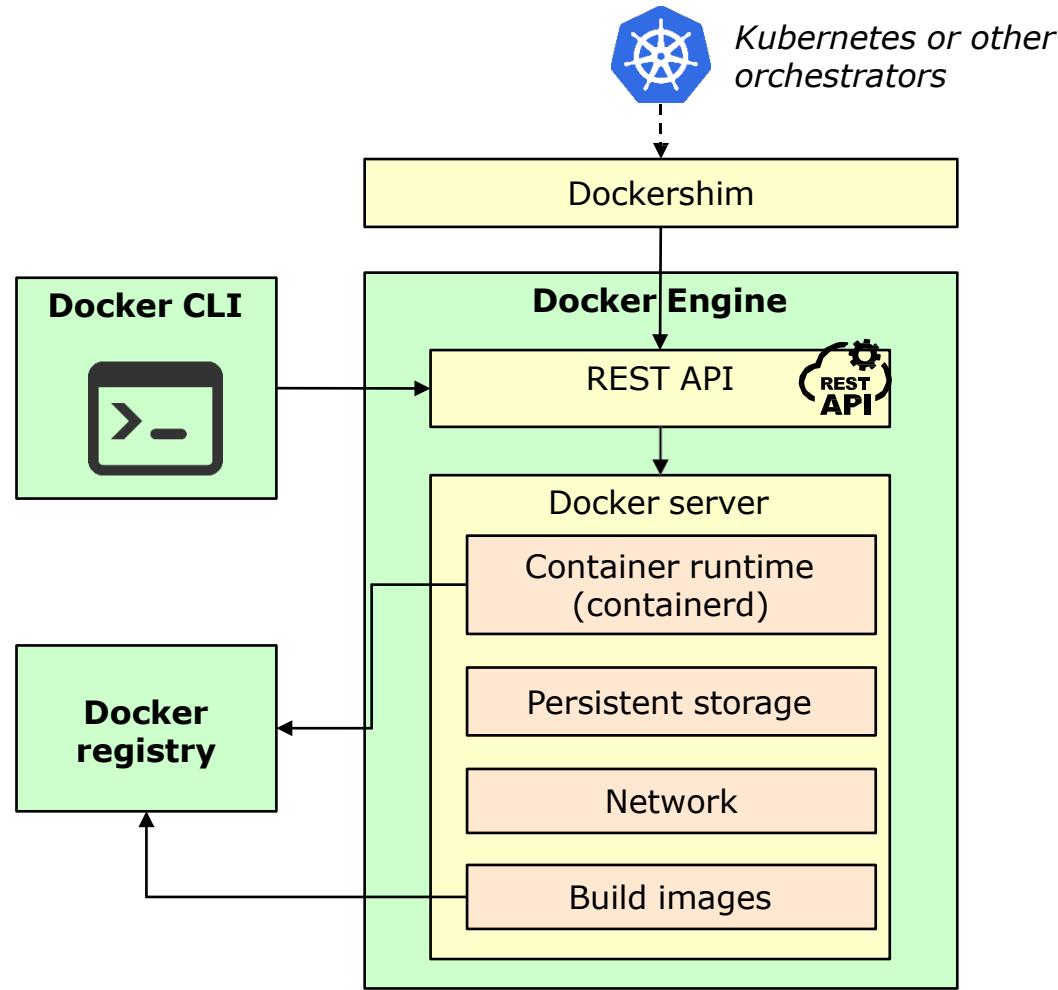
```
docker logs -f <container>
```

口从 Docker 容器中复制文件到另一个 Docker 容器

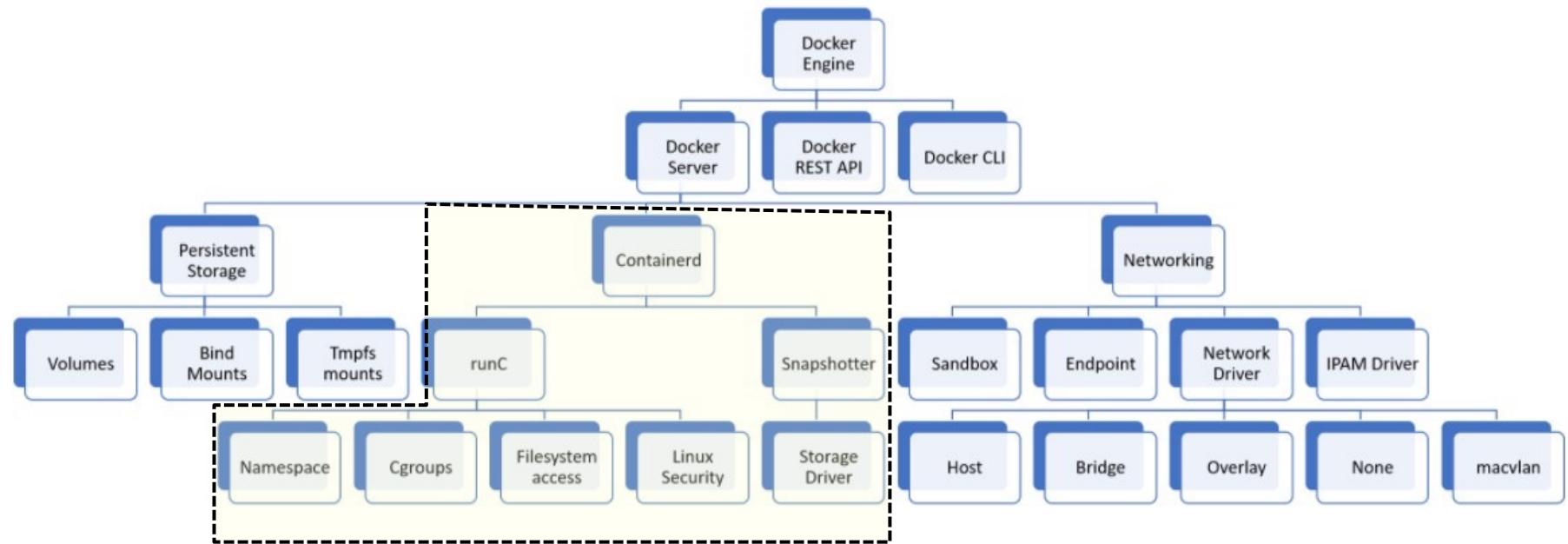
```
docker cp <source> <destination>
```

```
docker cp my_webserver:/etc/nginx/nginx.conf ~/
```

Docker 的软件架构



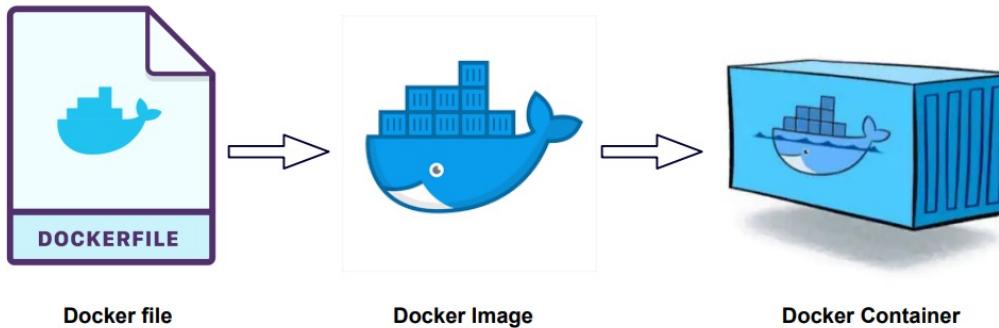
Docker full components tree



Mandatory components
(others can be provider by an
external orchestrator)

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
```

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 .
```

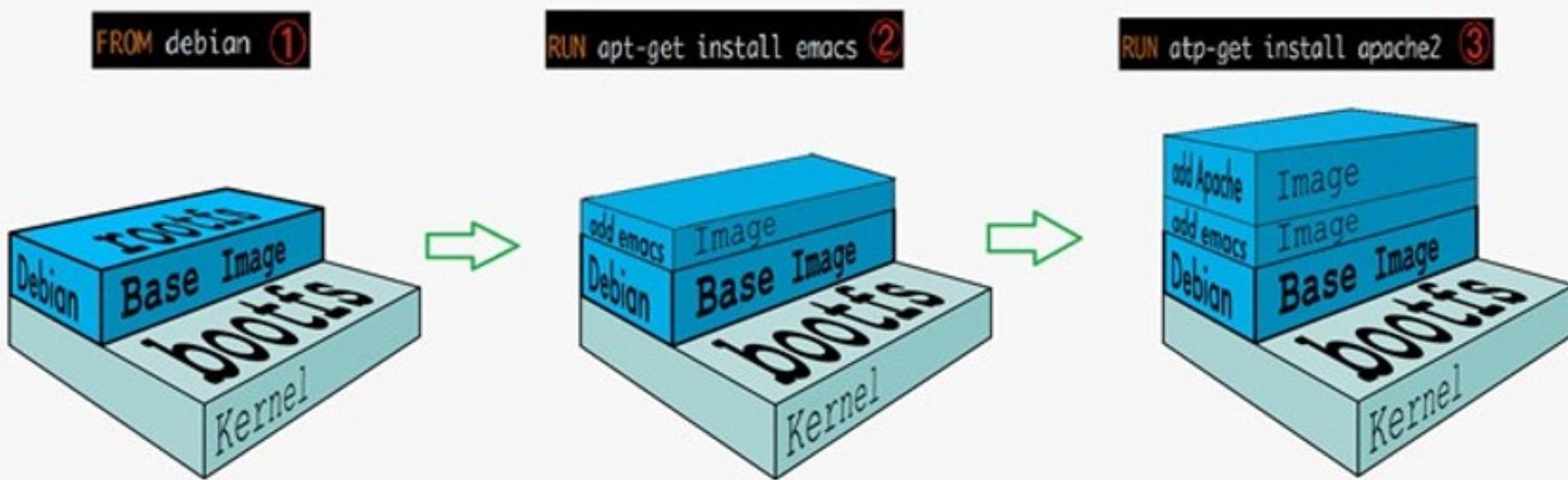
--pull: Pull a newer version of the image
-t mymongo: name the new image "mymongo"
. : look for the "Dockerfile" in the current folder

镜像底层原理

□ Docker 镜像为什么小?

- 底层很多模块是共用的，共用部分不会重复下载
- 通过分层的方式不断在这个镜像上去叠加应用，添加的应用又可以生成一个新的镜像，最底层是 bootfs 和 rootfs
- 底层使用的联合文件系统 UnionFS，可以简单理解为搭积木，不断在上面搭建你想要的应用

```
[root@kuangshen home]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
54fec2fa59d0: Pull complete
9c94e11103d9: Pull complete
04ab1bf453f: Pull complete
a22fde870392: Pull complete
def16cac9f02: Pull complete
1604f5999542: Pull complete
Digest: sha256:f7ee67d8d9050357a6ea362e2a7e8b65a6823d9b612bc430d057416788ef6df9
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```



Docker 资源开销举例

口内存和 CPU

- 相比原始操作系统，一个包含 “hello world” 程序的容器需要额外约 670KB 的RAM
- 相同程序在容器中运行的额外 CPU 消耗可以忽略不计

口磁盘

- Ubuntu 14.04 LTS 的容器可能会占用 250MB 的磁盘空间
- 安装 openjdk-7-jre 会额外增加约 140MB

为什么 Docker 会被广泛使用？

□ 持续交付

- 更频繁且更少错误地交付软件
- 无需花费时间进行开发到运维的交接

□ 提高安全性

- 容器有助于隔离系统的每个部分，并更好地控制组件

□ 何时何地都能运行

- 所有语言，所有数据库，所有操作系统，任何分发版本，任何云服务，任何机器

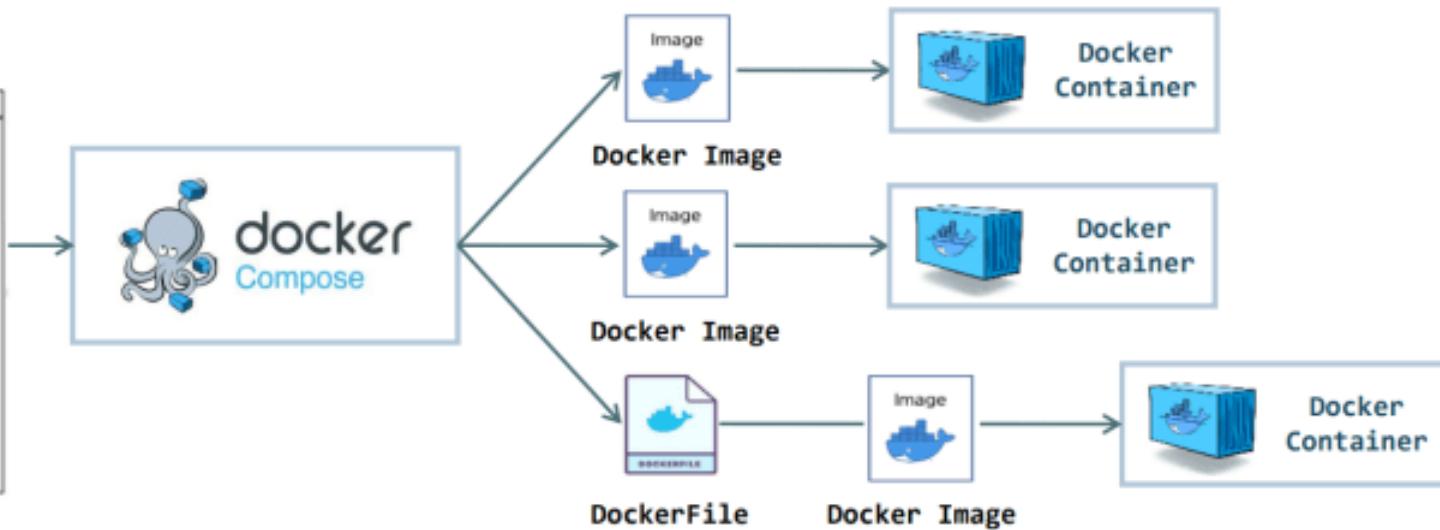
□ 可重复性

- 减少“它在我的机器上明明可以工作”的情况

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

```
*** docker-compose.yml
version: "3.7"
services:
  db:
    image: mysql:8.0.19
    restart: always
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD=password
  app:
    build: app
    restart: always
  web:
    build: web
    restart: always
    ports:
      - 80:80
```



你喜欢容器吗？



容器总结

□ 轻量级虚拟化无疑非常重要，在大企业中的应用日益增多

- 更有效地利用计算资源
- 降低运营成本（例如，更新不同虚拟机中的每个单独内核）
- 使得多个进程能够共存，并具有（强）隔离属性

□ Docker 是容器虚拟化的成熟技术

- 尽管其受欢迎是因为改变了软件被打包的方式，而不是轻量级虚拟化...



20M+

monthly developers

7M+

applications

20B+

monthly image pulls



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