Docker技术

1. 启程
   1. 容器生态系统
      1. 容器核心技术

容器规范

runtime spec

image format spec

容器 runtime

lxc

Docker 最初也是用 lxc 作为 runtime

runc

Docker 自己开发的容器 runtime，符合 oci 规范，也是现在 Docker 的默认 runtime

rkt

CoreOS 开发的容器 runtime，符合 oci 规范，因而能够运行 Docker 的容器

容器管理工具

lxd

lxd 是 lxc 对应的管理工具

docker engine

deamon

cli

rkt cli

rkt 的管理工具是 rkt cli

容器定义工具

docker image

docker 容器的模板，runtime 依据 docker image 创建容器。

dockerfile

包含若干命令的文本文件，可以通过这些命令创建出 docker image

ACI (App Container Image)

docker image 类似，只不过它是由 CoreOS 开发的 rkt 容器的 image 格式

Registries

Docker Registry

企业可以用 Docker Registry 构建私有的 Registry

Docker Hub

Docker 为公众提供的托管 Registry

https://hub.docker.com

Quay.io

https://quay.io/

是另一个公共托管 Registry

容器 OS

CoreOS

atomic

ubuntu core

* + 1. 容器平台技术

容器编排引擎

docker swarm

Docker 开发的容器编排引擎

kubernetes

Google 领导开发的开源容器编排引擎，同时支持 Docker 和 CoreOS 容器

mesos

mesos 与 marathon 一起提供容器编排引擎功能

容器管理平台

Rancher

ContainerShip

基于容器的 PaaS

Deis

Flynn

Dokku

* + 1. 容器支持技术

容器网络

docker network

docker network 是 Docker 原生的网络解决方案

flannel

weave

calico

服务发现

etcd

consul

zookeeper

监控

docker ps/top/stats

Docker 也提供了 stats API，用户可以通过 HTTP 请求获取容器的状态信息

sysdig

cAdvisor/Heapster

Weave Scope

数据管理

Flocker

日志管理

docker logs

logspout

对日志提供了路由功能，它可以收集不同容器的日志并转发给其他工具进行后处理。

安全性

OpenSCAP

OpenSCAP 能够对容器镜像进行扫描，发现潜在的漏洞

* 1. 覆盖知识范围
  2. 准备实验环境

[root@localhost Documents]# docker run -d -p 80:80 httpd

Unable to find image 'httpd:latest' locally

Trying to pull repository docker.io/library/httpd ...

latest: Pulling from docker.io/library/httpd

f2b6b4884fc8: Pull complete

b58fe2a5c9f1: Pull complete

e797fea70c45: Pull complete

6c7b4723e810: Pull complete

02074013c987: Pull complete

4ad329af1f9e: Pull complete

0cc56b739fe0: Pull complete

Digest: sha256:b54c05d62f0af6759c0a9b53a9f124ea2ca7a631dd7b5730bca96a2245a34f9d

Status: Downloaded newer image for docker.io/httpd:latest

d25318e3fa1ce167a4bb059206a7e730bc10929906def21b3b6903cb9f1a5938

Docker 客户端执行 docker run 命令。

Docker daemon 发现本地没有 httpd 镜像。

daemon 从 Docker Hub 下载镜像。

下载完成，镜像 httpd 被保存到本地。

Docker daemon 启动容器

docker images 可以查看到 httpd 已经下载到本地。

[root@localhost ~]# docker run -d -p 80:80 httpd

255f154c74d6980abadd3935dcd10d482d1cc0f5dbddbd660981efc7217c9e59

[root@localhost ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/httpd latest 805130e51ae9 3 weeks ago 178 MB

docker ps 或者 docker container ls 显示容器正在运行。

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

255f154c74d6 httpd "httpd-foreground" 5 minutes ago Up 5 minutes 0.0.0.0:80->80/tcp thirsty\_fermi

[root@localhost ~]# docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

255f154c74d6 httpd "httpd-foreground" 5 minutes ago Up 5 minutes 0.0.0.0:80->80/tcp thirsty\_fermi

* 1. 运行一个容器

1. 容器技术
   1. 容器核心知识
      1. what

什么是容器

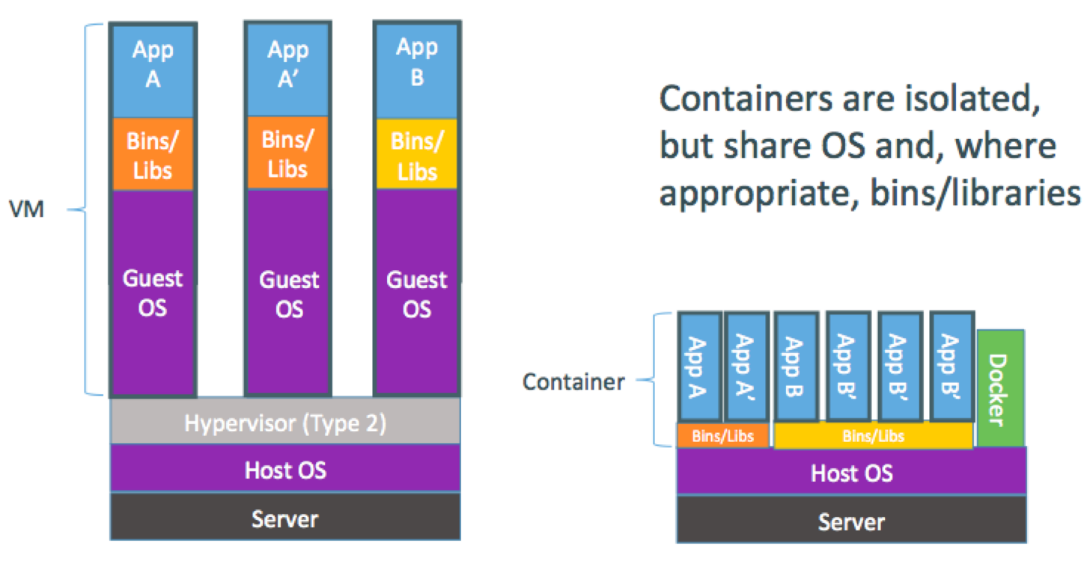
容器是一种轻量级、可移植、自包含的软件打包技术，使应用程序可以在几乎任何地方以相同的方式运行。开发人员在自己笔记本上创建并测试好的容器，无需任何修改就能够在生产系统的虚拟机、物理服务器或公有云主机上运行。

容器由两部分组成：

1.应用程序本身

2.依赖：比如应用程序需要的库或其他软件

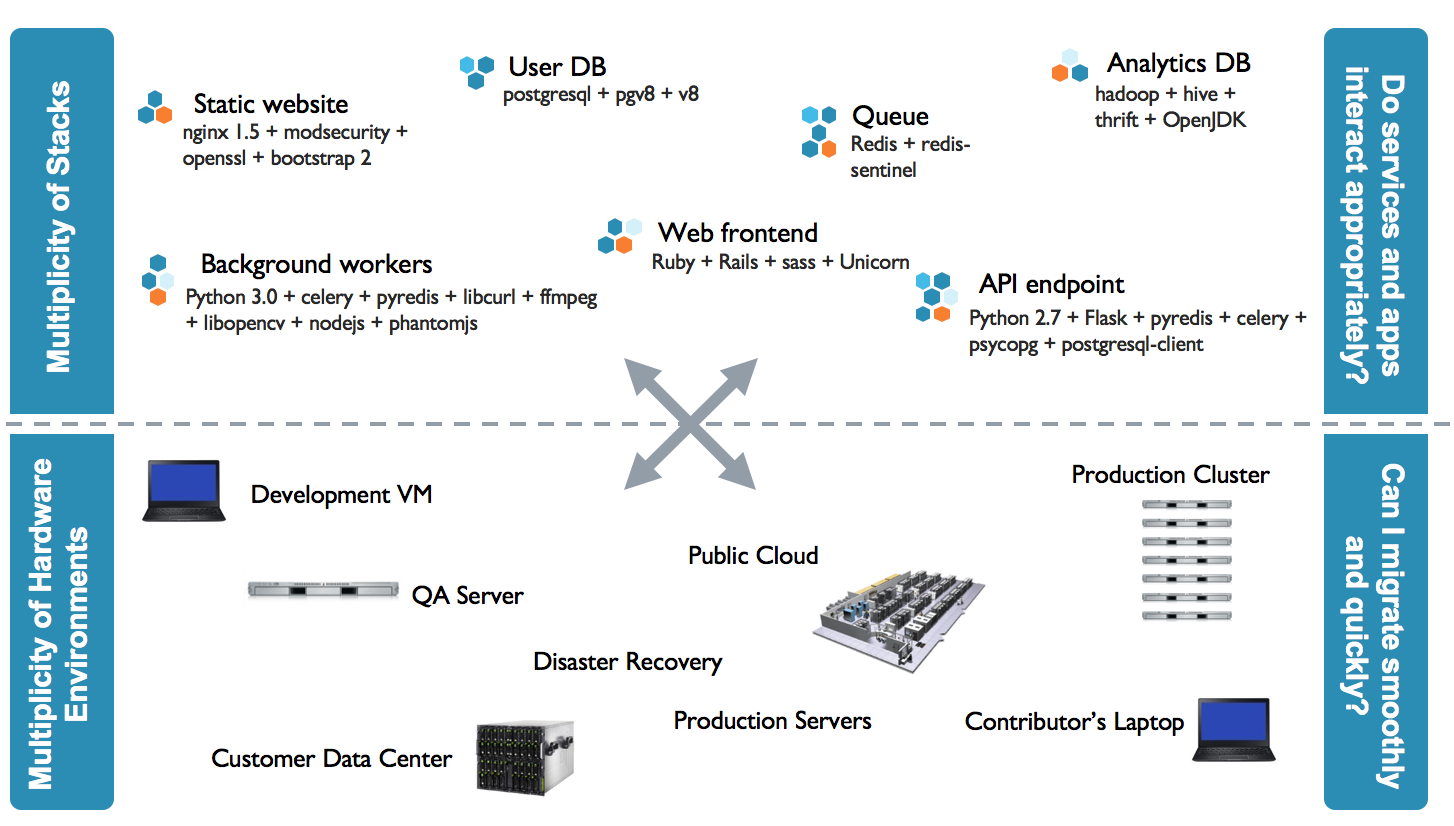
容器在 Host 操作系统的用户空间中运行，与操作系统的其他进程隔离。



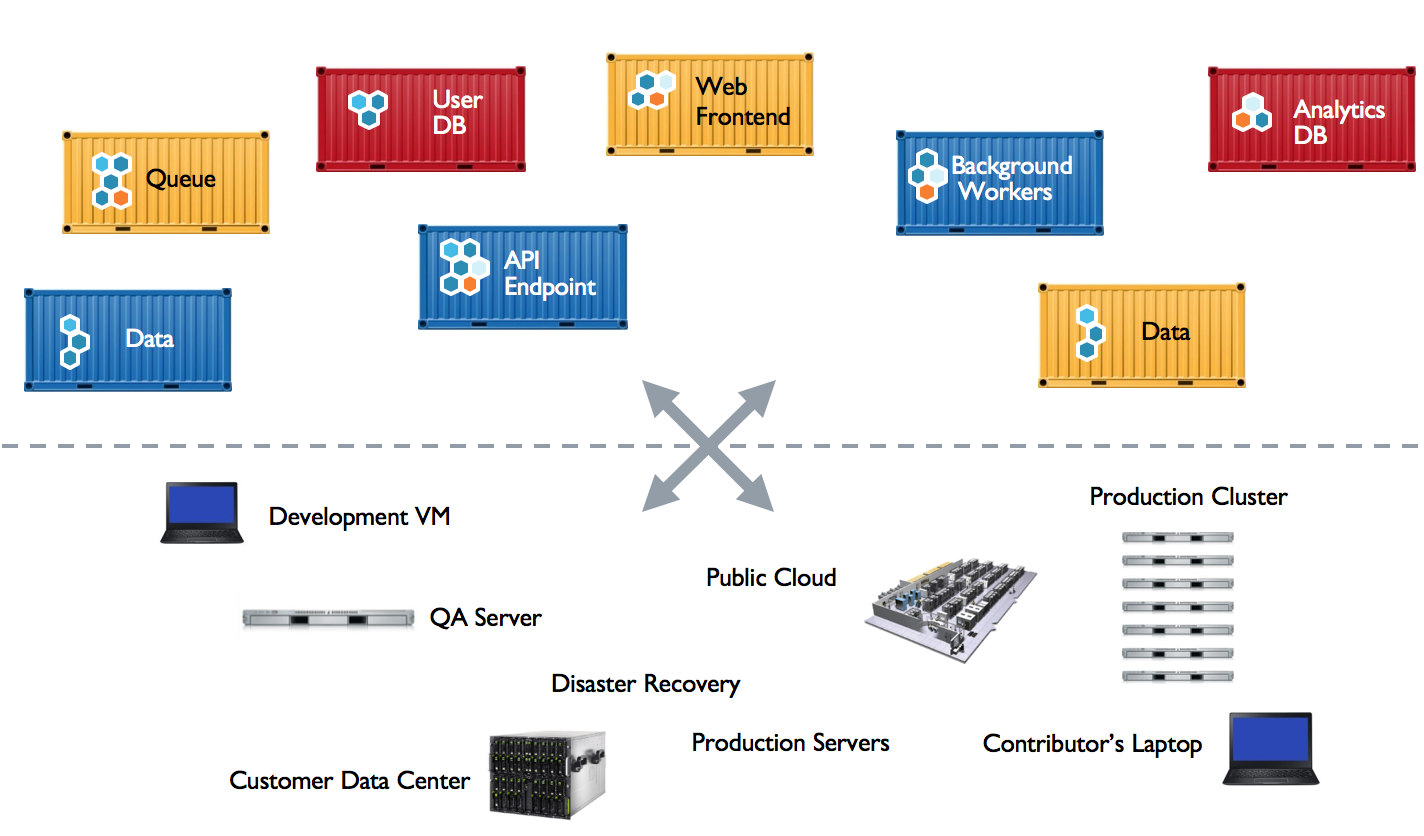
如图所示，由于所有的容器共享同一个 Host OS，这使得容器在体积上要比虚拟机小很多。另外，启动容器不需要启动整个操作系统，所以容器部署和启动速度更快，开销更小，也更容易迁移。

* + 1. why

为什么需要容器



==================================================



==================================================



* + 1. how

架构

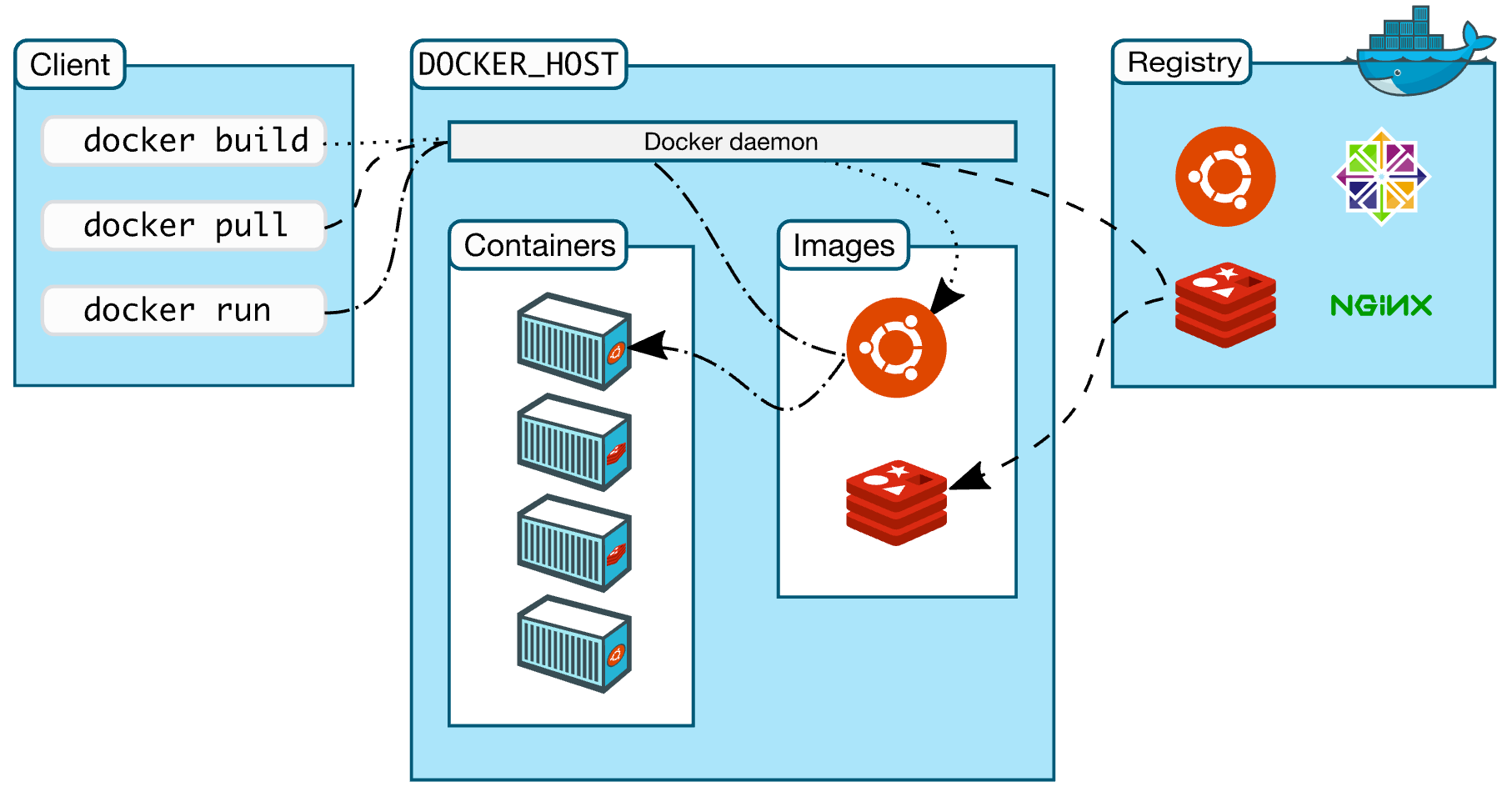
镜像

容器

网络

存储

* + 1. 容器核心知识



Docker 客户端

最常用的 Docker 客户端是 docker 命令。通过 docker 我们可以方便地在 Host 上构建和运行容器

Docker 服务器

Docker daemon 是服务器组件，以 Linux 后台服务的方式运行

Docker daemon 运行在 Docker host 上，负责创建、运行、监控容器，构建、存储镜像

默认配置下，Docker daemon 只能响应来自本地 Host 的客户端请求。如 果要允许远程客户端请求，需要在配置文件中打开 TCP 监听

编辑配置文件 /etc/systemd/system/multi-user.target.wants/docker.service，在环境变量 ExecStart 后面添加 -H tcp://0.0.0.0，允许来自任意 IP 的客户端连接

[root@localhost system]# pwd

/usr/lib/systemd/system

[root@localhost system]# cd /etc/systemd/system/multi-user.target.wants/

[root@localhost multi-user.target.wants]# ln -s /usr/lib/systemd/system/docker.service docker.service

[root@localhost multi-user.target.wants]# ls docker.service

docker.service

重启 Docker daemon

[root@localhost system]# systemctl daemon-reload

[root@localhost system]# systemctl restart docker.service

Docker 镜像

[root@localhost ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/hello-world latest e38bc07ac18e 8 days ago 1.85 kB

docker.io/httpd latest 805130e51ae9 4 weeks ago 178 MB

docker 运行

[root@localhost ~]# docker run hello-world

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

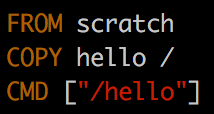
Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/engine/userguide/

Dockerfile



FROM scratch

此镜像是从白手起家，从 0 开始构建。

COPY hello /

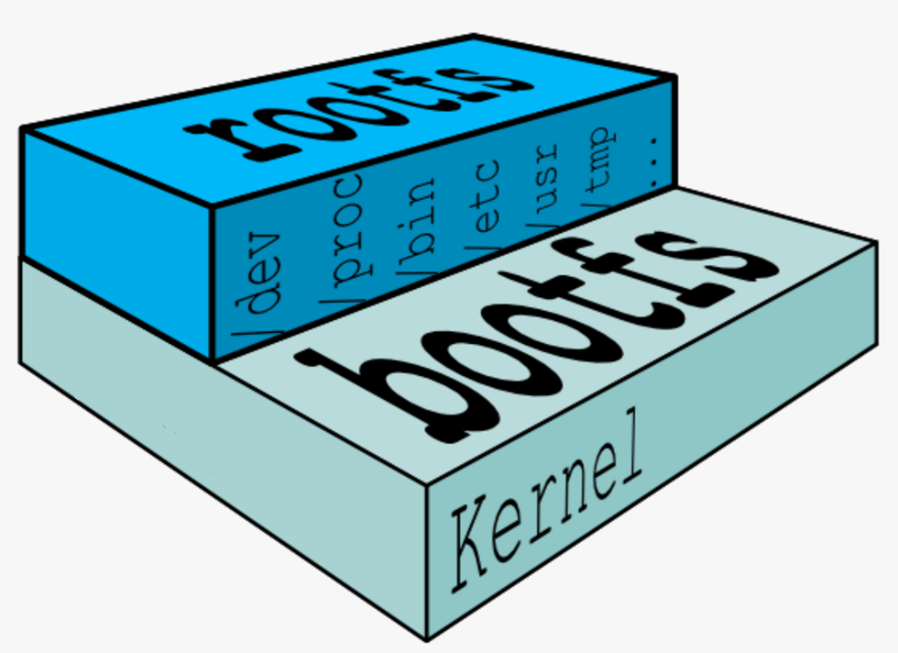
将文件“hello”复制到镜像的根目录。

CMD ["/hello"]

容器启动时，执行 /hello

base 镜像

Linux 操作系统由内核空间和用户空间组成



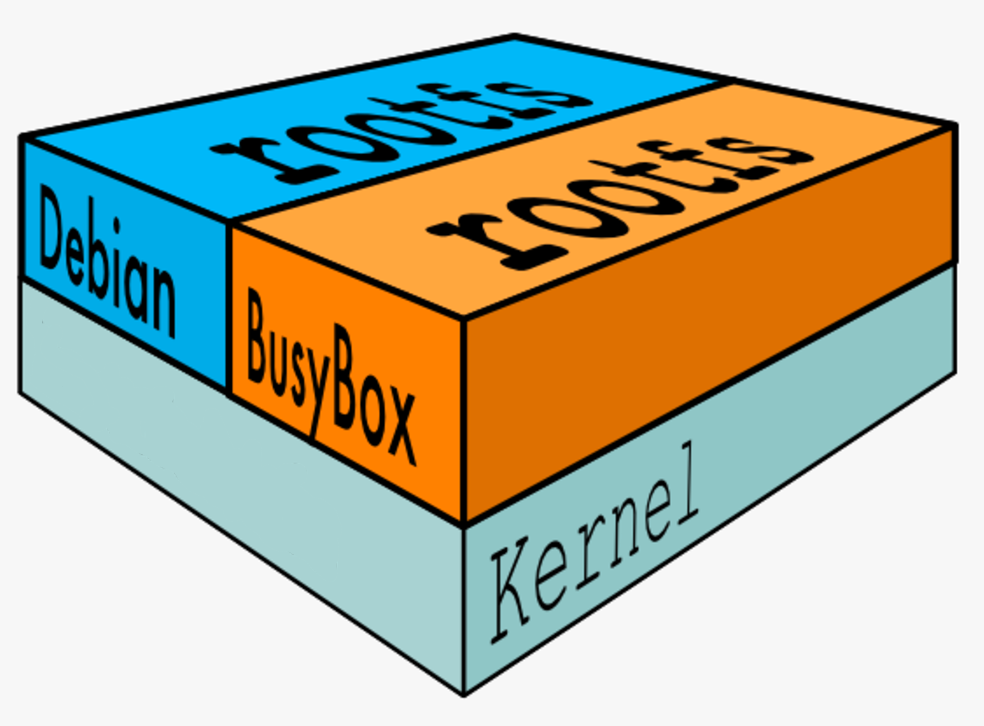
Linux 操作系统由内核空间和用户空间组成

rootfs

用户空间的文件系统是 rootfs，包含我们熟悉的 /dev, /proc, /bin 等目录

对于 base 镜像来说，底层直接用 Host 的 kernel，自己只需要提供 rootfs 就行了

不同 Linux 发行版的区别主要就是 rootfs



bootfs

内核空间是 kernel，Linux 刚启动时会加载 bootfs 文件系统，之后 bootfs 会被卸载掉

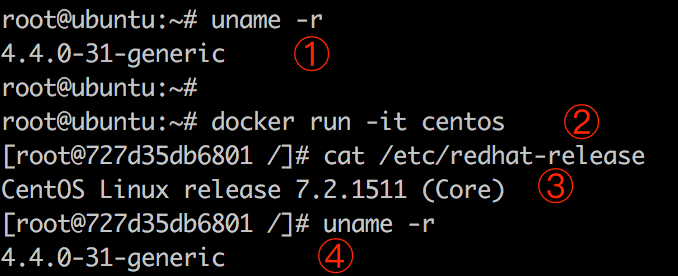
base镜像含义

不依赖其他镜像，从 scratch 构建

其他镜像可以之为基础进行扩展

docker注意事项

base 镜像只是在用户空间与发行版一致，kernel 版本与发型版是不同的



① Host kernel 为 4.4.0-31

② 启动并进入 CentOS 容器

③ 验证容器是 CentOS 7

④ 容器的 kernel 版本与 Host 一致

容器只能使用 Host 的 kernel，并且不能修改

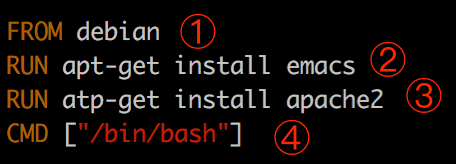
所有容器都共用 host 的 kernel，在容器中没办法对 kernel 升级。如果容器对 kernel 版本有要求（比如应用只能在某个 kernel 版本下运行），则不建议用容器，这种场景虚拟机可能更合适

docker扩展镜像

如果多个容器共享一份基础镜像，当某个容器修改了基础镜像的内容，比如 /etc 下的文件，这时其他容器的 /etc 是否也会被修改？

答案是不会！

修改会被限制在单个容器内。

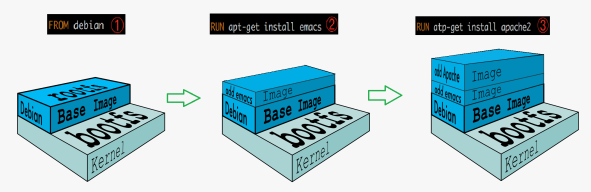


① 新镜像不再是从 scratch 开始，而是直接在 Debian base 镜像上构建

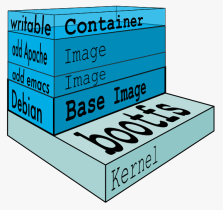
② 安装 emacs 编辑器

③ 安装 apache2

④ 容器启动时运行 bash



可写的容器层



所有对容器的改动 - 无论添加、删除、还是修改文件都只会发生在容器层中

只有容器层是可写的，容器层下面的所有镜像层都是只读的

镜像层数量可能会很多，所有镜像层会联合在一起组成一个统一的文件系统。 如果不同层中有一个相同路径的文件，比如 /a，上层的 /a 会覆盖下层的 /a，也就是说用户只能访问到上层中的文件 /a。在容器层中，用户看到的是一个叠加之后的文件系统。

添加文件

在容器中创建文件时，新文件被添加到容器层中

读取文件

在容器中读取某个文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到， 立即将其复制到容器层，然后打开并读入内存。

修改文件

在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到， 立即将其复制到容器层，然后修改之

删除文件

在容器中删除文件时，Docker 也是从上往下依次在镜像层中查找此文件。找到后， 会在容器层中记录下此删除操作

容器层保存的是镜像变化的部分，不会对镜像本身进行任何修改

创建Docker

docker commit 命令

运行容器

docker run -it ubuntu

-it 以交互模式进入容器

用户后的id为容器内部ID

修改容器

例如安装软件

将容器保存为新的镜像

docker ps 查看当前运行的容器，nams是系统自动分配的docker名字

docker commit silly\_goldberg ubuntu-with-vi

docker images

docker run -it ubuntu-with-vi

Dockerfile 构建文件

用 Dockerfile 创建上节的 ubuntu-with-vi

from ubuntu run apt-get update && apt-get install -y vim

① root@ubuntu:~# pwd

当前目录为 /root

② root@ubuntu:~# ls

Dockerfile 准备就绪

③ root@ubuntu:~# docker build -t ubuntu-with-vi-dockerfile .

运行 docker build 命令，-t 将新镜像命名为 ubuntu-with-vi-dockerfile，命令末尾的 . 指明 build context 为当前目录。Docker 默认会从 build context 中查找 Dockerfile 文件，我们也可以通过 -f 参数指定 Dockerfile 的位置

④ Sending build context to Docker daemon 32.26 kB

首先 Docker 将 build context 中的所有文件发送给 Docker daemon。build context 为镜像构建提供所需要的文件或目录。

Dockerfile 中的 ADD、COPY 等命令可以将 build context 中的文件添加到镜像。此例中，build context 为当前目录 /root，该目录下的所有文件和子目录都会被发送给 Docker daemon。

build context 就得小心了，不要将多余文件放到 build context

⑤ Step 1 : FROM ubuntu

执行 FROM，将 ubuntu 作为 base 镜像

⑥ Step 2 : RUN apt-get update && apt-get install -y vim

执行 RUN，安装 vim，具体步骤为 ⑦、⑧、⑨

⑦ ---> Running in 9f4d4166f7e3

启动 ID 为 9f4d4166f7e3 的临时容器，在容器中通过 apt-get 安装 vim

⑧ ---> 35ca89798937

安装成功后，将容器保存为镜像，其 ID 为 35ca89798937

类似 docker commit 的命令

⑨ Removing intermediate container 9f4d4166f7e3

删除临时容器 9f4d4166f7e3

⑩ Successfully built 35ca89798937

镜像构建成功

查看镜像分层结构

docker history

docker history 会显示镜像的构建历史，也就是 Dockerfile 的执行过程。

表示无法获取 IMAGE ID，通常从 Docker Hub 下载的镜像会有这个问题

镜像的缓存

Using cache

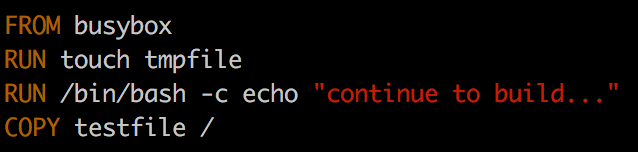
如果我们希望在构建镜像时不使用缓存，可以在 docker build 命令中加上 --no-cache 参数

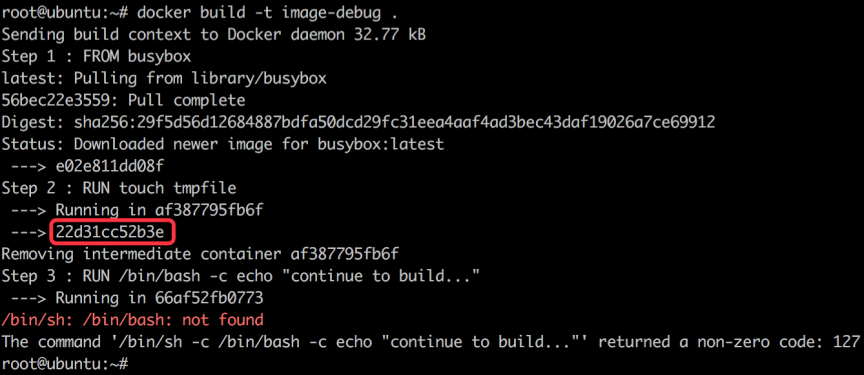
Dockerfile 中每一个指令都会创建一个镜像层，上层是依赖于下层的。无论什么时候，只要某一层发生变化 ，其上面所有层的缓存都会失效。

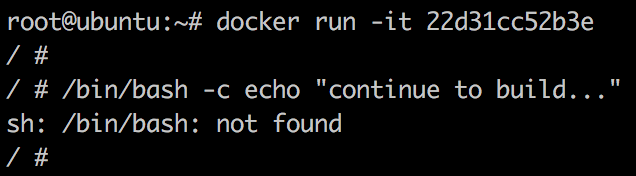
也就是说，如果我们改变 Dockerfile 指令的执行顺序，或者修改或添加指令，都会使缓存失效

debug Dockerfile

docker build







Docker 创建过程

从 base 镜像运行一个容器。

执行一条指令，对容器做修改

执行类似 docker commit 的操作，生成一个新的镜像层

Docker 再基于刚刚提交的镜像运行一个新容器

重复 2-4 步，直到 Dockerfile 中的所有指令执行完毕。

Dockerfile 中最常用的指令

FROM

指定 base 镜像

MAINTAINER

设置镜像的作者

COPY

COPY src dest

COPY ["src", "dest"]

注意：src 只能指定 build context 中的文件或目录

ADD

ENV

设置环境变量，环境变量可被后面的指令使用

EXPOSE

指定容器中的进程会监听某个端口，Docker 可以将该端口暴露出来

VOLUME

将文件或目录声明为 volume

WORKDIR

为后面的 RUN, CMD, ENTRYPOINT, ADD 或 COPY 指令设置镜像中的当前工作目录

RUN

在容器中运行指定的命令

CMD

容器启动时运行指定的命令

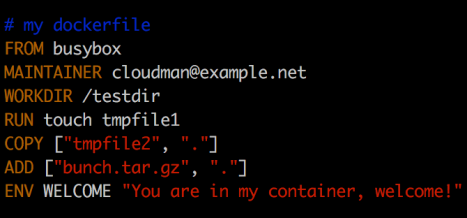
Dockerfile 中可以有多个 CMD 指令，但只有最后一个生效。CMD 可以被 docker run 之后的参数替换

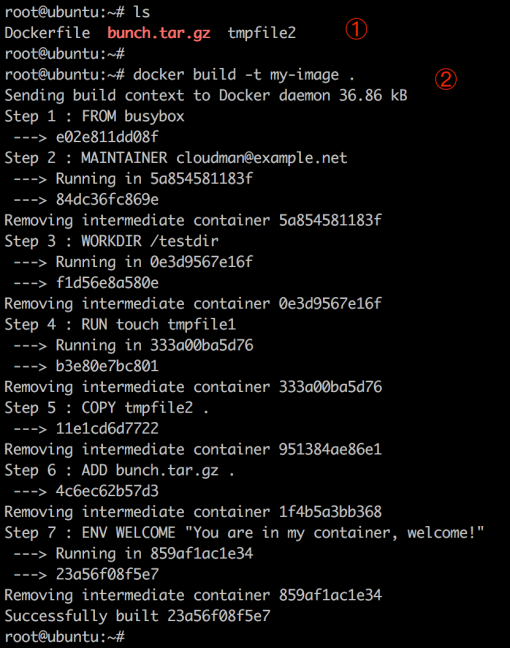
ENTRYPOINT

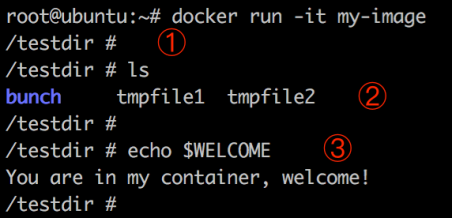
设置容器启动时运行的命令

Dockerfile 中可以有多个 ENTRYPOINT 指令，但只有最后一个生效。CMD 或 docker run 之后的参数会被当做参数传递给 ENTRYPOINT。

编写的Dockerfile

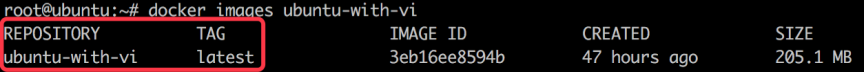






为镜像命名

docker build -t ubuntu-with-vi



实际上一个特定镜像的名字由两部分组成：repository 和 tag

[image name] = [repository]:[tag]

如果执行 docker build 时没有指定 tag，会使用默认值 latest。

docker build -t ubuntu-with-vi:latest

atest 其实并没有什么特殊的含义。当没指明镜像 tag 时，Docker 会使用默认值 latest，仅此而已。

tag 使用最佳实践



docker tag myimage-v1.9.1 myimage:1

docker tag myimage-v1.9.1 myimage:1.9

docker tag myimage-v1.9.1 myimage:1.9.1

docker tag myimage-v1.9.1 myimage:latest



docker tag myimage-v1.9.2 myimage:1

docker tag myimage-v1.9.2 myimage:1.9

docker tag myimage-v1.9.2 myimage:1.9.2

docker tag myimage-v1.9.2 myimage:latest



docker tag myimage-v2.0.0 myimage:2

docker tag myimage-v2.0.0 myimage:2.0

docker tag myimage-v2.0.0 myimage:2.0.0

docker tag myimage-v2.0.0 myimage:latest

镜像的常用操作子命令

images 显示镜像列表

history 显示镜像构建历史

commit 从容器创建新镜像

build 从 Dockerfile 构建镜像

tag 给镜像打 tag

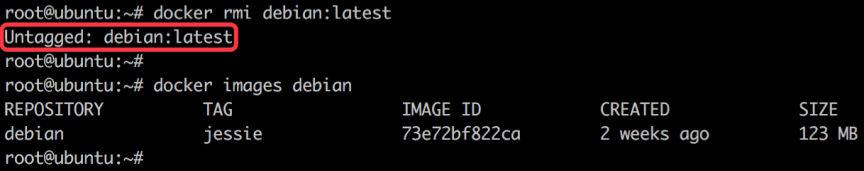
pull 从 registry 下载镜像

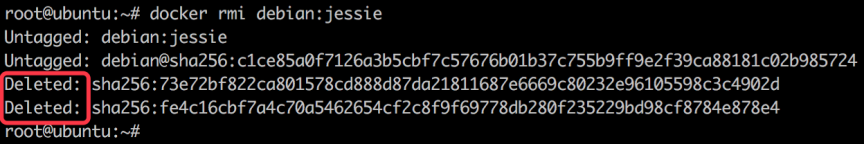
push 将 镜像 上传到 registry

rmi 删除 Docker host 中的镜像

mi 只能删除 host 上的镜像，不会删除 registry 的镜像

如果一个镜像对应了多个 tag，只有当最后一个 tag 被删除时，镜像才被真正删除。





search 搜索 Docker Hub 中的镜像

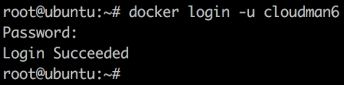
search 让我们无需打开浏览器，在命令行中就可以搜索 Docker Hub 中的镜像

Registry

何用 Docker Hub 存取我们的镜像

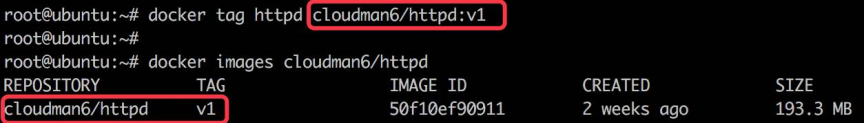
首先得在 Docker Hub 上注册一个账号

在 Docker Host 上登录



修改镜像的 repository 使之与 Docker Hub 账号匹配

Docker Hub 为了区分不同用户的同名镜像，镜像的 registry 中要包含用户名，完整格式为：[username]/xxx:tag



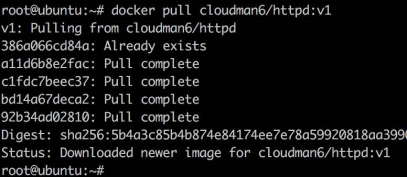
Docker 官方自己维护的镜像没有用户名，比如 httpd

通过 docker push 将镜像上传到 Docker Hub

Docker 会上传镜像的每一层。因为 cloudman6/httpd:v1 这个镜像实际上跟官方的 httpd 镜像一模一样，Docker Hub 上已经有了全部的镜像层，所以真正上传的数据很少。同样的，如果我们的镜像是基于 base 镜像的，也只有新增加的镜像层会被上传。如果想上传同一 repository 中所有镜像，省略 tag 部分就可以了

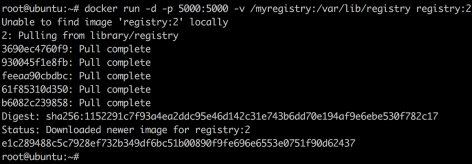


这个镜像可被其他 Docker host 下载使用了



搭建本地的 Registry

启动 registry 容器



-d 是后台启动容器

-p 将容器的 5000 端口映射到 Host 的 5000 端口。5000 是 registry 服务端口。

-v 将容器 /var/lib/registry 目录映射到 Host 的 /myregistry，用于存放镜像数据。

通过 docker tag 重命名镜像，使之与 registry 匹配

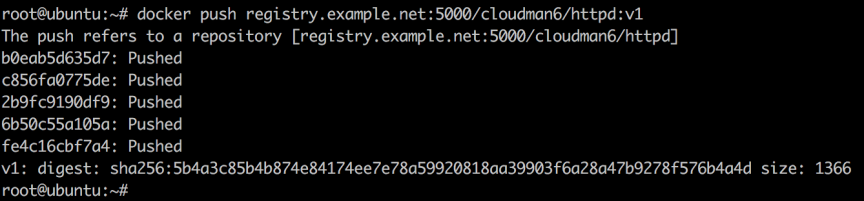


们在镜像的前面加上了运行 registry 的主机名称和端口

而 repository 的完整格式为：[registry-host]:[port]/[username]/xxx

只有 Docker Hub 上的镜像可以省略 [registry-host]:[port]

通过 docker push 上传镜像



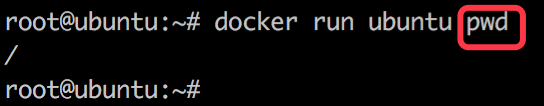
现在已经可通过 docker pull 从本地 registry 下载镜像了



Docker 容器

运行容器

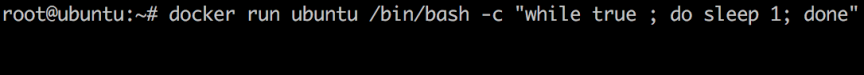
docker run 是启动容器的方法



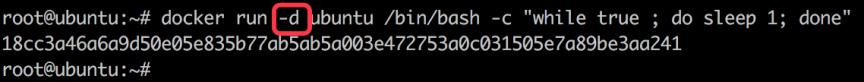
docker ps 或 docker container ls 可以查看 Docker host 中当前运行的容器

docker ps -a 或 docker container ls -a 会显示所有状态的容器

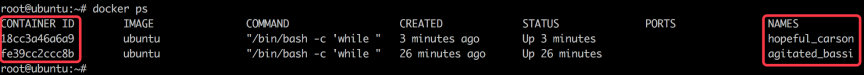
让容器长期运行



不过这种方法有个缺点：它占用了一个终端



-d 以后台方式启动容器

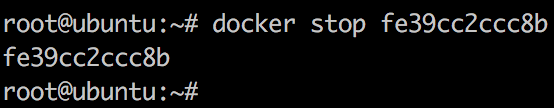


CONTAINER ID

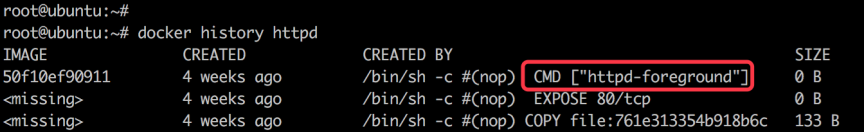
容器的 “短ID”，前面启动容器时返回的是 “长ID”。短ID是长ID的前12个字符

NAMES

段显示容器的名字，在启动容器时可以通过 --name 参数显示地为容器命名，如果不指定，docker 会自动为容器分配名字。

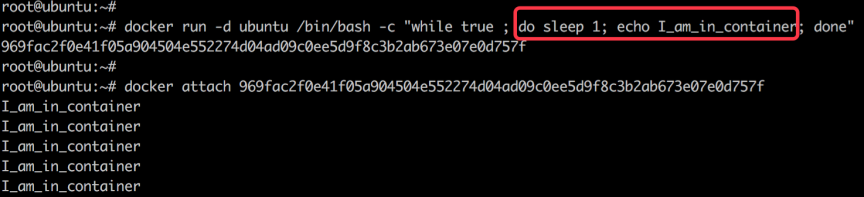






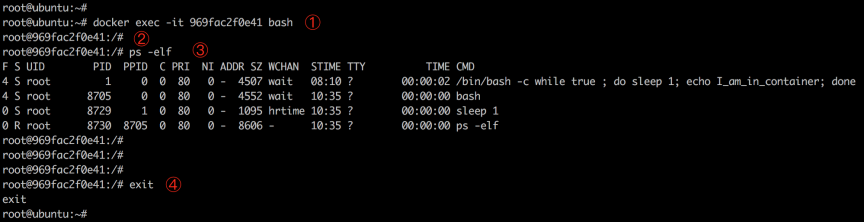
进入容器

docker attach



可通过 Ctrl+p 然后 Ctrl+q 组合键退出 attach 终端

docker exec



① -it 以交互模式打开 pseudo-TTY，执行 bash，其结果就是打开了一个 bash 终端

② 进入到容器中，容器的 hostname 就是其 “短ID”。

③ 可以像在普通 Linux 中一样执行命令。ps -elf 显示了容器启动进程while 以及当前的 bash 进程。

④ 执行 exit 退出容器，回到 docker host

docker exec -it <container> bash|sh 是执行 exec 最常用的方式。

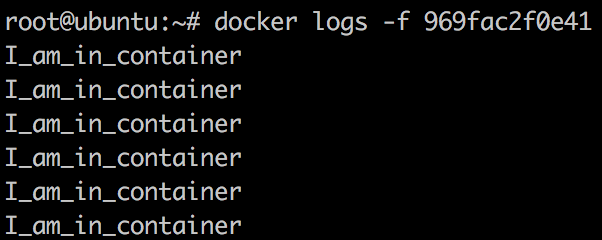
attach VS exec

attach 直接进入容器 启动命令 的终端，不会启动新的进程。

exec 则是在容器中打开新的终端，并且可以启动新的进程。

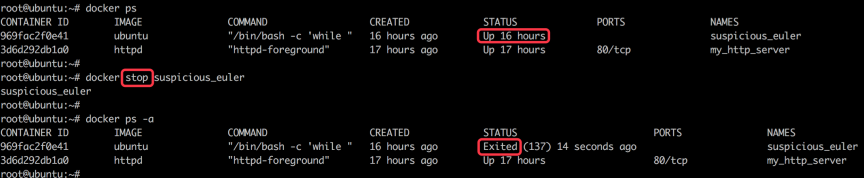
如果想直接在终端中查看启动命令的输出，用 attach；其他情况使用 exec

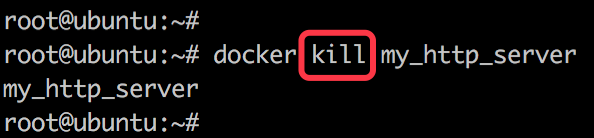
当然，如果只是为了查看启动命令的输出，可以使用 docker logs



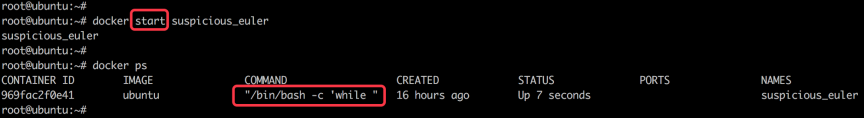
容器的其他常用操作

通过 docker stop 可以停止运行的容器

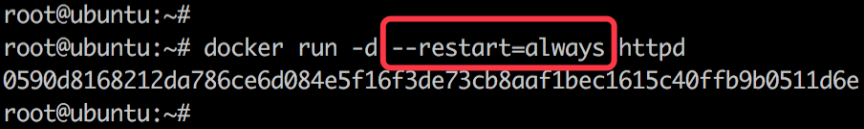




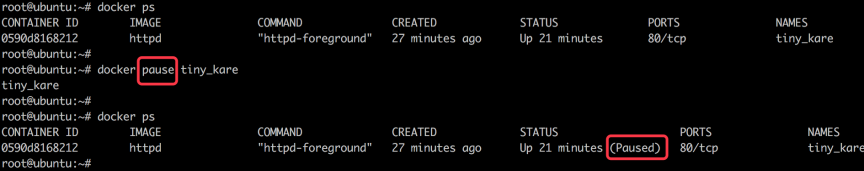
docker start 重新启动

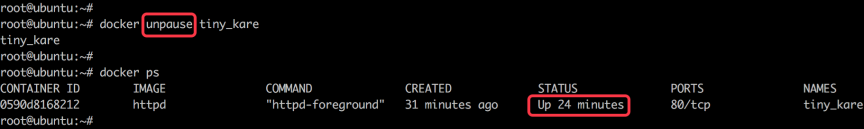


docker restart 可以重启容器



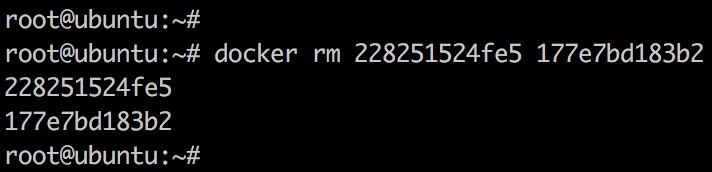
docker pause 望暂时让容器暂停工作一段时间，比如要对容器的文件系统打个快照





删除容器





docker rm 一次可以指定多个容器，如果希望批量删除所有已经退出的容器

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

docker rm 是删除容器，而 docker rmi 是删除镜像

补充



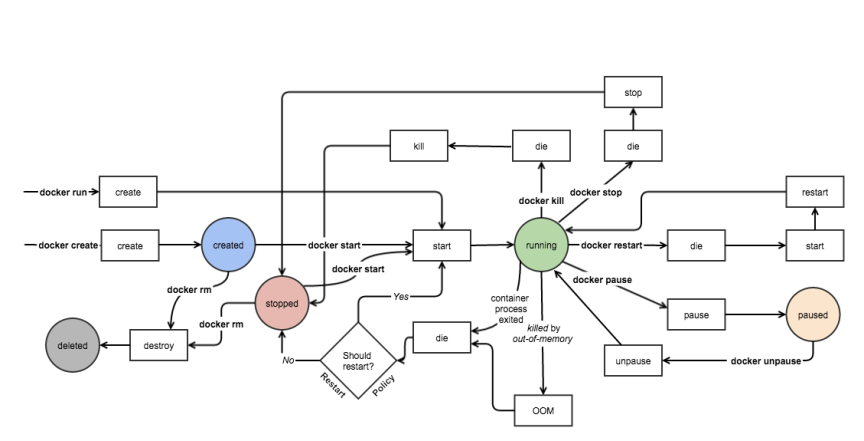
① docker create 创建的容器处于 Created 状态

② docker start 将以后台方式启动容器。 docker run 命令实际上是 docker create 和 docker start 的组合

只有当容器的启动进程 退出 时，--restart 才生效

退出包括正常退出或者非正常退出。这里举了两个例子：启动进程正常退出或发生 OOM，此时 docker 会根据 --restart 的策略判断是否需要重启容器。但如果容器是因为执行 docker stop 或docker kill 退出，则不会自动重启

命令概要



限制容器

内存限额

物理内存

-m 或 --memory：设置内存的使用限额

swap

--memory-swap：设置 内存+swap 的使用限额

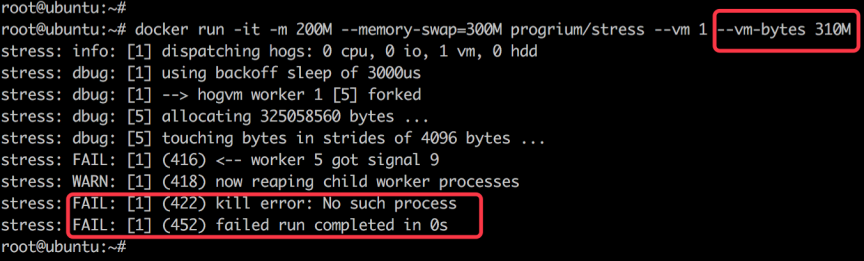
docker run -m 200M --memory-swap=300M ubuntu

docker run -it -m 200M --memory-swap=300M progrium/stress --vm 1 --vm-bytes 280M

--vm 1：启动 1 个内存工作线程

--vm-bytes 280M：每个线程分配 280M 内存





如果在启动容器时只指定 -m 而不指定 --memory-swap，那么 --memory-swap 默认为 -m 的两倍

限制容器对CPU的使用

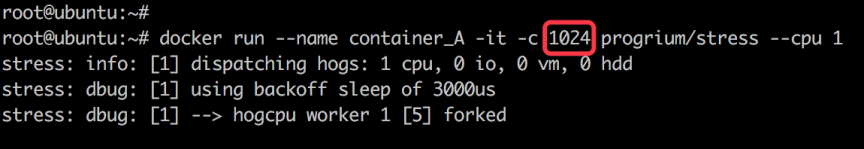
docker run --name "container\_A" -c 1024 ubuntu

docker run --name "container\_B" -c 512 ubuntu

这种按权重分配 CPU 只会发生在 CPU 资源紧张的情况下

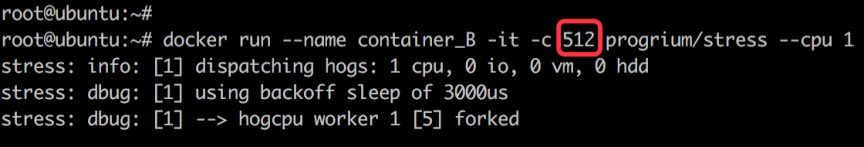
案例

启动 container\_A，cpu share 为 1024

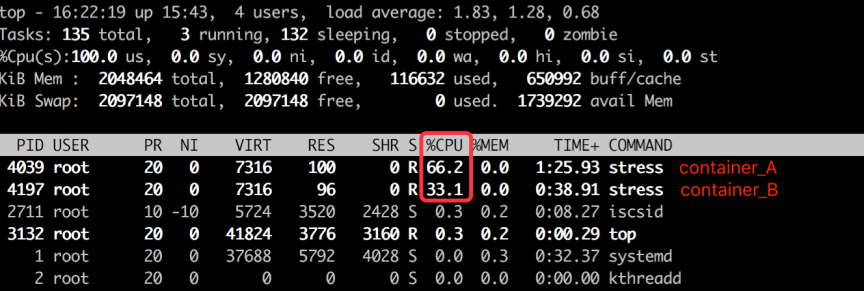


--cpu 用来设置工作线程的数量。因为当前 host 只有 1 颗 CPU

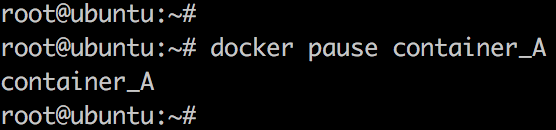
启动 container\_B，cpu share 为 512



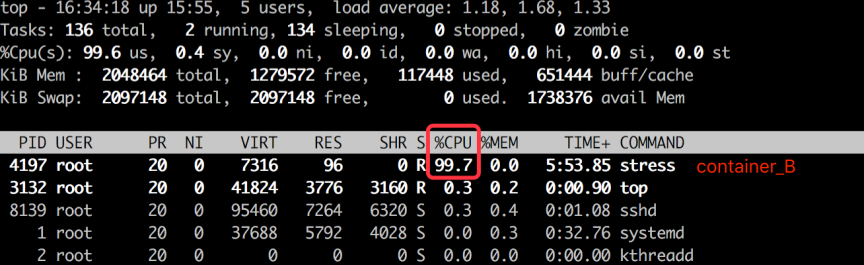
在 host 中执行 top，查看容器对 CPU 的使用情况



现在暂停 container\_A



top 显示 container\_B 在 container\_A 空闲的情况下能够用满整颗 CPU



限制容器的 Block IO

block IO 权重

所有容器能平等地读写磁盘，可以通过设置 --blkio-weight 参数来改变容器 block IO 的优先级

--blkio-weight 与 --cpu-shares 类似，设置的是相对权重值，默认为 500。

docker run -it --name container\_A --blkio-weight 600 ubuntu

docker run -it --name container\_B --blkio-weight 300 ubuntu

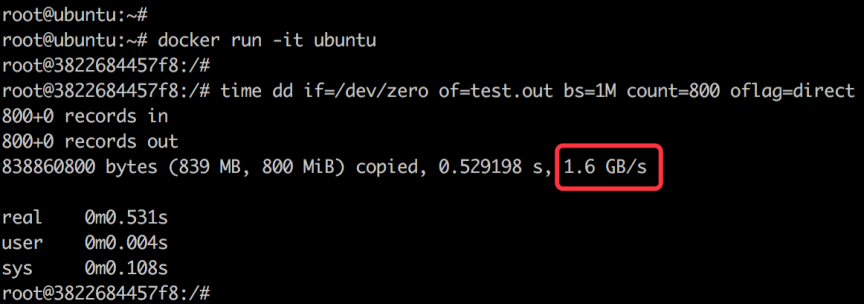
限制 bps 和 iops

bps 是 byte per second，每秒读写的数据量

--device-read-bps，限制读某个设备的 bps

--device-write-bps，限制写某个设备的 bps





iops 是 io per second，每秒 IO 的次数

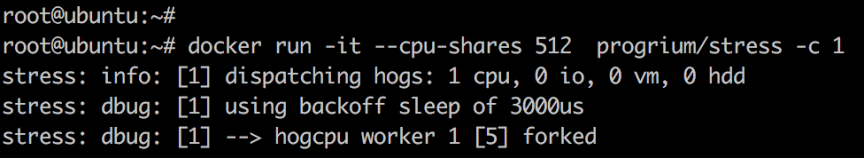
--device-read-iops，限制读某个设备的 iops

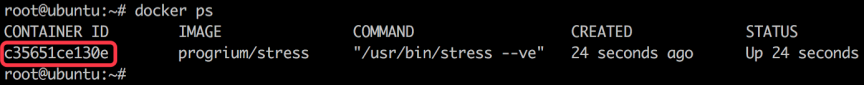
--device-write-iops，限制写某个设备的 iops

底层实现技术

cgroup 实现资源限额

--cpu-shares、-m、--device-write-bps 实际上就是在配置 cgroup







/sys/fs/cgroup

/sys/fs/cgroup/memory/docker

/sys/fs/cgroup/blkio/docker

namespace 实现资源隔离

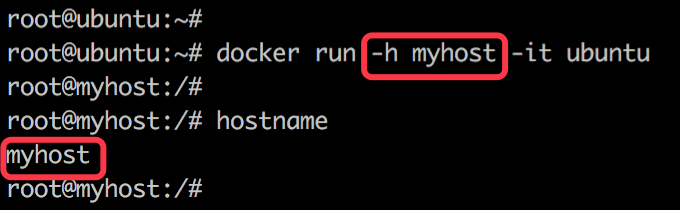
Mount namespace

Mount namespace 让容器看上去拥有整个文件系统

容器有自己的 / 目录，可以执行 mount 和 umount 命令。当然我们知道这些操作只在当前容器中生效，不会影响到 host 和其他容器

UTS namespace

让容器有自己的 hostname。 默认情况下，容器的 hostname 是它的短ID，可以通过 -h 或 --hostname 参数设置。

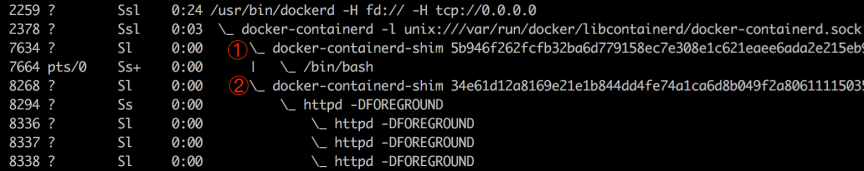


IPC namespace

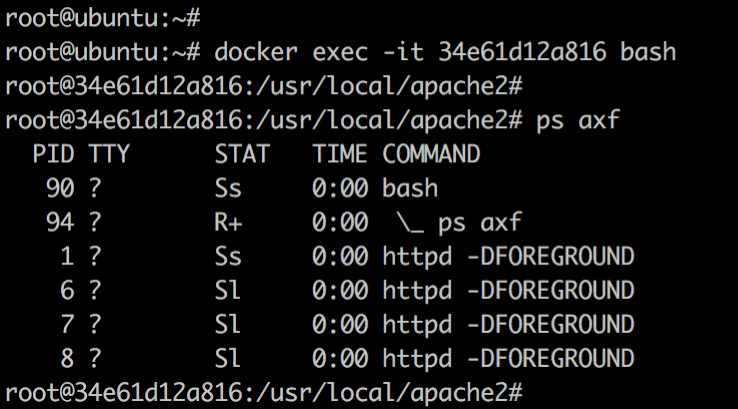
让容器拥有自己的共享内存和信号量（semaphore）来实现进程间通信，而不会与 host 和其他容器的 IPC 混在一起。

PID namespace

容器拥有自己独立的一套 PID



所有容器的进程都挂在 dockerd 进程下，同时也可以看到容器自己的子进程。



Network namespace

容器拥有自己独立的网卡、IP、路由等资源

User namespace

让容器能够管理自己的用户，host 不能看到容器中创建的用户

* 1. 容器进阶知识
     1. multi-host
     2. 容器网络
     3. 监控
     4. 数据管理
     5. 日志管理
     6. 安全性

1. 容器平台技术
   1. 容器编排引擎
      1. docker swarm
      2. kubernetes
      3. mesos + marathon
   2. 容器管理平台
      1. Rancher
      2. ContainerShip
   3. 基于容器的PaaS
      1. Deis
      2. Flynn
      3. Dokku