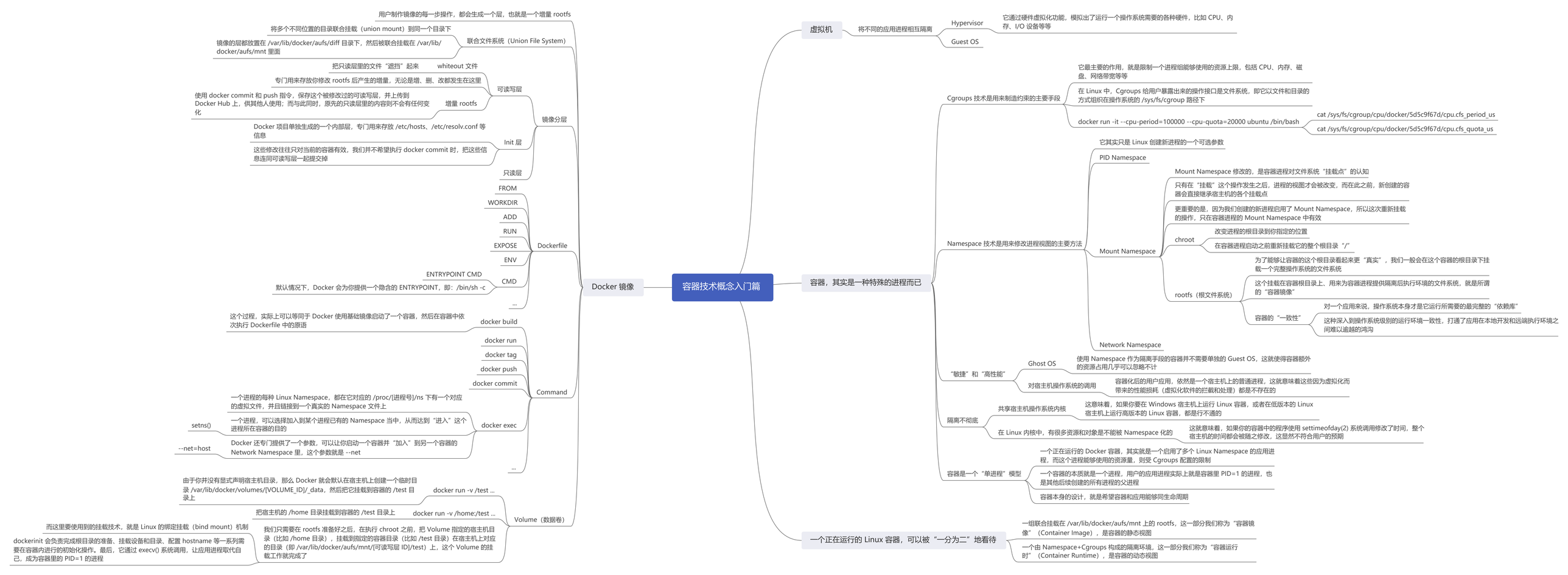
容器技术概念入门篇
Docker 镜像
用户制作镜像的每一步操作，都会生成一个层，也就是一个增量 rootfs
联合文件系统（Union File System）
将多个不同位置的目录联合挂载（union mount）到同一个目录下
镜像的层都放置在 /var/lib/docker/aufs/diff 目录下，然后被联合挂载在 /var/lib/docker/aufs/mnt 里面
镜像分层
可读写层
whiteout 文件
把只读层里的文件“遮挡”起来
专门用来存放你修改 rootfs 后产生的增量，无论是增、删、改都发生在这里
增量 rootfs
使用 docker commit 和 push 指令，保存这个被修改过的可读写层，并上传到 Docker Hub 上，供其他人使用；而与此同时，原先的只读层里的内容则不会有任何变化
Init 层
Docker 项目单独生成的一个内部层，专门用来存放 /etc/hosts、/etc/resolv.conf 等信息
这些修改往往只对当前的容器有效，我们并不希望执行 docker commit 时，把这些信息连同可读写层一起提交掉
只读层
Dockerfile
FROM
WORKDIR
ADD
RUN
EXPOSE
ENV
CMD
ENTRYPOINT CMD
默认情况下，Docker 会为你提供一个隐含的 ENTRYPOINT，即：/bin/sh -c
...
Command
这个过程，实际上可以等同于 Docker 使用基础镜像启动了一个容器，然后在容器中依次执行 Dockerfile 中的原语
docker build
docker run
docker tag
docker push
docker commit
docker exec
一个进程的每种 Linux Namespace，都在它对应的 /proc/[进程号]/ns 下有一个对应的虚拟文件，并且链接到一个真实的 Namespace 文件上
一个进程，可以选择加入到某个进程已有的 Namespace 当中，从而达到“进入”这个进程所在容器的目的
setns()
Docker 还专门提供了一个参数，可以让你启动一个容器并“加入”到另一个容器的 Network Namespace 里，这个参数就是 --net
--net=host
...
Volume（数据卷）
由于你并没有显式声明宿主机目录，那么 Docker 就会默认在宿主机上创建一个临时目录 /var/lib/docker/volumes/[VOLUME_ID]/_data，然后把它挂载到容器的 /test 目录上
docker run -v /test ...
把宿主机的 /home 目录挂载到容器的 /test 目录上
docker run -v /home:/test ...
我们只需要在 rootfs 准备好之后，在执行 chroot 之前，把 Volume 指定的宿主机目录（比如 /home 目录），挂载到指定的容器目录（比如 /test 目录）在宿主机上对应的目录（即 /var/lib/docker/aufs/mnt/[可读写层 ID]/test）上，这个 Volume 的挂载工作就完成了
而这里要使用到的挂载技术，就是 Linux 的绑定挂载（bind mount）机制
dockerinit 会负责完成根目录的准备、挂载设备和目录、配置 hostname 等一系列需要在容器内进行的初始化操作。最后，它通过 execv() 系统调用，让应用进程取代自己，成为容器里的 PID=1 的进程
虚拟机
将不同的应用进程相互隔离
Hypervisor
它通过硬件虚拟化功能，模拟出了运行一个操作系统需要的各种硬件，比如 CPU、内存、I/O 设备等等
Guest OS
容器，其实是一种特殊的进程而已
Cgroups 技术是用来制造约束的主要手段
它最主要的作用，就是限制一个进程组能够使用的资源上限，包括 CPU、内存、磁盘、网络带宽等等
在 Linux 中，Cgroups 给用户暴露出来的操作接口是文件系统，即它以文件和目录的方式组织在操作系统的 /sys/fs/cgroup 路径下
docker run -it --cpu-period=100000 --cpu-quota=20000 ubuntu /bin/bash
cat /sys/fs/cgroup/cpu/docker/5d5c9f67d/cpu.cfs_period_us
cat /sys/fs/cgroup/cpu/docker/5d5c9f67d/cpu.cfs_quota_us
Namespace 技术是用来修改进程视图的主要方法
它其实只是 Linux 创建新进程的一个可选参数
PID Namespace
Mount Namespace
Mount Namespace 修改的，是容器进程对文件系统“挂载点”的认知
只有在“挂载”这个操作发生之后，进程的视图才会被改变，而在此之前，新创建的容器会直接继承宿主机的各个挂载点
更重要的是，因为我们创建的新进程启用了 Mount Namespace，所以这次重新挂载的操作，只在容器进程的 Mount Namespace 中有效
chroot
改变进程的根目录到你指定的位置
在容器进程启动之前重新挂载它的整个根目录“/”
rootfs（根文件系统）
为了能够让容器的这个根目录看起来更“真实”，我们一般会在这个容器的根目录下挂载一个完整操作系统的文件系统
这个挂载在容器根目录上、用来为容器进程提供隔离后执行环境的文件系统，就是所谓的“容器镜像”
容器的“一致性”
对一个应用来说，操作系统本身才是它运行所需要的最完整的“依赖库”
这种深入到操作系统级别的运行环境一致性，打通了应用在本地开发和远端执行环境之间难以逾越的鸿沟
Network Namespace
“敏捷”和“高性能”
Ghost OS
使用 Namespace 作为隔离手段的容器并不需要单独的 Guest OS，这就使得容器额外的资源占用几乎可以忽略不计
对宿主机操作系统的调用
容器化后的用户应用，依然是一个宿主机上的普通进程，这就意味着这些因为虚拟化而带来的性能损耗（虚拟化软件的拦截和处理）都是不存在的
隔离不彻底
共享宿主机操作系统内核
这意味着，如果你要在 Windows 宿主机上运行 Linux 容器，或者在低版本的 Linux 宿主机上运行高版本的 Linux 容器，都是行不通的
在 Linux 内核中，有很多资源和对象是不能被 Namespace 化的
这就意味着，如果你的容器中的程序使用 settimeofday(2) 系统调用修改了时间，整个宿主机的时间都会被随之修改，这显然不符合用户的预期
容器是一个“单进程”模型
一个正在运行的 Docker 容器，其实就是一个启用了多个 Linux Namespace 的应用进程，而这个进程能够使用的资源量，则受 Cgroups 配置的限制
一个容器的本质就是一个进程，用户的应用进程实际上就是容器里 PID=1 的进程，也是其他后续创建的所有进程的父进程
容器本身的设计，就是希望容器和应用能够同生命周期
一个正在运行的 Linux 容器，可以被“一分为二”地看待
一组联合挂载在 /var/lib/docker/aufs/mnt 上的 rootfs，这一部分我们称为“容器镜像”（Container Image），是容器的静态视图
一个由 Namespace+Cgroups 构成的隔离环境，这一部分我们称为“容器运行时”（Container Runtime），是容器的动态视图