docker

- image
  - 分层
    - 可读写层
      - 可读写层。它是这个容器的 rootfs 最上面的一层（6e3be5d2ecccae7cc），它的挂载方式为：rw。在没有写入文件之前，这个目录是空的。而一旦在容器里做了写操作，你修改产生的内容就会以增量的方式出现在这个层中。
    - init层
      - 专门用来存放特殊信息的，比如ubuntu镜像的/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。comiit时不会提交该层的信息。
    - 只读层
      - 只读层。它是这个容器的 rootfs 最下面的几层
    - 分层的意义
      - 如果要改变镜像的内容，会在可读写层创建一个 whiteout 文件，把只读层里的文件"遮挡"起来。比如，你要删除只读层里一个名叫 foo 的文件，那么这个删除操作实际上是在可读写层创建了一个名叫.wh.foo 的文件。这样，当这两个层被联合挂载之后，foo 文件就会被.wh.foo 文件"遮挡"起来，"消失"了
      - 我们修改后，可以使用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。这，就是增量rootfs 的好处
- network
  - 类型
    - bridge
      - 此模式会为每一个容器分配、设置IP等，并将容器连接到一个docker0虚拟网桥，通过docker0网桥以及Iptables nat表配置与宿主机通信
        - 子主题
    - Container
      - 和一个指定的容器共享IP、端口范围
        - 和已经存在的一个容器共享一个Network Namespace。所以两个容器的进程可以通过lo网卡设备通信
        - 两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的
    - none
      - 该模式关闭了容器的网络功能
    - host
      - 使用宿主机的IP和端口
        - 如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace
        - 不用任何NAT转换，就如直接跑在宿主机中一样
        - 容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的
  - 组网类型
    - overlay
      - overlay网络是在传统网络上虚拟出一个虚拟网络，承载的底层网络不再需要做任何适配。它唯一的要求是主机之间的IP连接。
        - L2 overlay
          - 大二层。构建在底层物理网络上的L2网络。VXLAN就是L2 overlay网络的典型实现，其通过在UDP包中封装原始L2报文，实现了 容器的跨主机通信
        - L3 overlay
          - 类似L2 overlay，但会在节点上增加一个网关。每个节点上的容器都在同一个子网内，可以直接进行二层通信
          - 跨节点的容器间通信只能走L3，都会经过网关转发，性能相比于L2 overlay较弱
          - 但是更灵活。可以存在于不同的网段中
            - flannel的UDP模式采用的就是L3 overlay模型。 L3 overlay网络容器在主机间迁移时可能需要改变其IP地址
    - underlay
      - 传统的网络组网就是underlay类型
        - L2 underlay
          - L2 underlay网络就是链路层（L2）互通的底层网络
        - L3 underlay
          - 在各个虚拟网络和主机网络之间进行不同网络报文的路由转发工作。只要父接口相同，即使虚拟机/容器不在同一个网络，也可以互相ping通对方
            - flannel的host-gw模式和Calico的BGP组网方式都是L3 underlay类型的网络
  - 组网方案
    - 隧道方案
      - 隧道网络也称为overlay网络。
        - 隧道方案很重要一点是进行封包，它们主要解决的问题是同一个问题，即在容器网络里，主机间不知道对方的目的地址，没有办法把IP包投递到正确的地方。
    - 路由方案
      - 传统的三层网络是用路由来互相访问的，不需要封包
        - Calico在每一个计算节点，利用Linux Kernel实现高效的vRouter来负责数据转发，而每个 vRouter通过BGP把自己节点上的工作负载的路由信息向整个Calico网络传播。小规模部署 可以直接互联，大规模下可通过指定的BGP Route Reflector完成。保证最终所有的容器之 间的数据流量都是通过IP路由的方式完成互联的
- 容器通讯方式
  - 主机间通信
  - 跨主机通信
  - 容器与主机间通信
- volume
  - 工作原理
    - 当容器进程被创建之后，尽管开启了 Mount Namespace，但是在它执行 chroot（或者 pivot_root）之前，容器进程一直可以看到宿主机上的整个文件系统
    - 所以，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
    - docker的挂载，使用的是Linux 的绑定挂载（bind mount）机制
      - 在该挂载点上进行的任何操作，只是发生在被挂载的目录或者文件上，而原挂载点的内容则会被隐藏起来且不受影响。
        - mount --bind /home /test，会将 /home 挂载到 /test 上。其实相当于将 /test 的 dentry，重定向到了 /home 的 inode。这样当我们修改 /test 目录时，实际修改的是 /home 目录的 inode。
        - 这也就是为何，一旦执行 umount 命令，/test 目录原先的内容就会恢复：因为修改真正发生在的，是在宿主机的/home 目录里。不会影响容器镜像的内容
  - 类型
    - volume
      - 在主机的/var/lib/docker/volumes/创建一个目录挂载进去。
        - 可以在多个容器共享使用，且不会自动删除
        - 容器目录原有文件不会被Volume覆盖
    - Bind mounts
      - 宿主机上对应的文件或者目录事先不需要存在，否则会覆盖掉容器内原有的内容。用时会自动创建
    - tmpfs
      - 仅存储在容器的内存中，永远不会写入文件系统
- dockerfile
  - COPY与ADD区别
    - ADD会对本地压缩文件（tar, gzip, bzip2, etc）做提取和解压操作。
    - 如果是远程，则只会下载，不会展开 ADD http://nginx.org.tar.gz
  - ARG
    - 仅仅在build docker image的过程中（包括CMD和ENTRYPOINT）有效。需要在构建时通过 --build-arg传入
  - CMD与ENTRYPOINT区别
    - CMD的命令会被 docker run 的命令覆盖而ENTRYPOINT不会
    - CMD和ENTRYPOINT都存在时，CMD的指令变成了ENTRYPOINT的参数
  - 优化
    - 多段构建镜像，减小最终镜像的体积
    - 将频繁改动的往最后放，减少重新构建的内容
  - .dockerignore
    - 构建时要忽略的对象
- 容器状态
  - created
  - running
  - stopped
  - paused
  - deleted
- registry
  - 关于k8s拉取镜像
    - K8s实现了自动拉取镜像的功能。这个功能的核心，是把 docker.json 内容编码，并以Secret 的方式作为 Pod 定义的一部分传给 Kubelet。
      - Kubelet 调用 docker 创建容器且把 .dockerconfigjson 传给 docker；最后 docker 使用解码出来的账户密码拉取镜像。
- 默认命名空间为moby
  - 子主题
- 架构
  - docker daemon
    - server、engine
  - driver
    - graphdriver
      - rootfs
    - networkdriver
      - brige、ip、port等
    - execdriver
      - cgroup、capabilitoes、namespace等
  - cli
- 与传统虚拟化的区别
  - 复用了宿主机的操作系统
  - 资源利用率高
  - 子主题
- 缓存
  - 子主题
  - 子主题
- 容器原理
  - 隔离
    - 启动时，docker修改了当前进程的PID namespace
      - 让当前进程只能看到自己的PID，所以认为自己是1
    - 启动时，修改当前进程的mount namespace
      - 只能看到自己的挂载点
    - 启动时，修改当前进程的network namespace
      - 只能看到自己的网络设备
  - 限制
    - 在宿主机上，容器与其他所有进程之间依然是平等的竞争关系
      - 为了控制容器的资源使用，使用Cgroup进行控制
  - 挂载
    - 除了声明要启用 MountNamespace 之外，我们还可以告诉容器进程，有哪些目录需要重新挂载
    - 因为启用了 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。如果在宿主机上用 mount -l 来检查一下这个挂载，你会发现它是不存在的
    - chroot可以改变进程的根目录，比如将rootfs拷贝到宿主机的$HOME/test 。然后使用chroot改变进程根目录到$HOME/test 。那么启动后，就可以看到rootfs里面的目录了。
      - 当然，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
  - 本质
    - 容器，其实是一种特殊的进程
      - 真正对隔离环境负责的其实还是宿主机操作系统本身，docker只是调用了一下
  - 问题
    - linux内核中很多资源和对象是不能被 Namespace 化的，最典型的例子就是：时间。如果通过容器修改了时间，那么整个宿主机的时间也被修改了
    - 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
  - 容器启动流程
    - 1. 启用 Linux Namespace 配置；2. 设置指定的 Cgroups 参数；3. 切换进程的根目录（Change Root）4. 启动用户进程。这些工作都是dockerinit做的
      - dockerinit不是应用进程 (ENTRYPOINT + CMD)。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。
  - 一个正在运行的 Linux 容器，其实可以被"一分为二"地看待
    - rootfs，这一部分我们称为"容器镜像"（Container Image），是容器的静态视图
    - 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
- rootfs
  - rootfs 只是一个操作系统所包含的文件和目录，并不包括操作系统内核
  - 实际上，同一台机器上的所有容器，都共享宿主机操作系统的内核
- 资源限制
  - 内存
    - 不可压缩资源
      - 可以调整swap、limit等
  - cpu
    - 可压缩资源
      - limit、绑定CPU、设置调度周期等
- 部分命令原理
  - exec
    - 每个进程在它的/proc/[进程号]/ns 下都有所以namespace的对应的虚拟文件，并且链接到一个真实的 Namespace 文件上。我们可以根据这些文件进行一些操作，比如加入到一个已经存在的 Namespace 当中