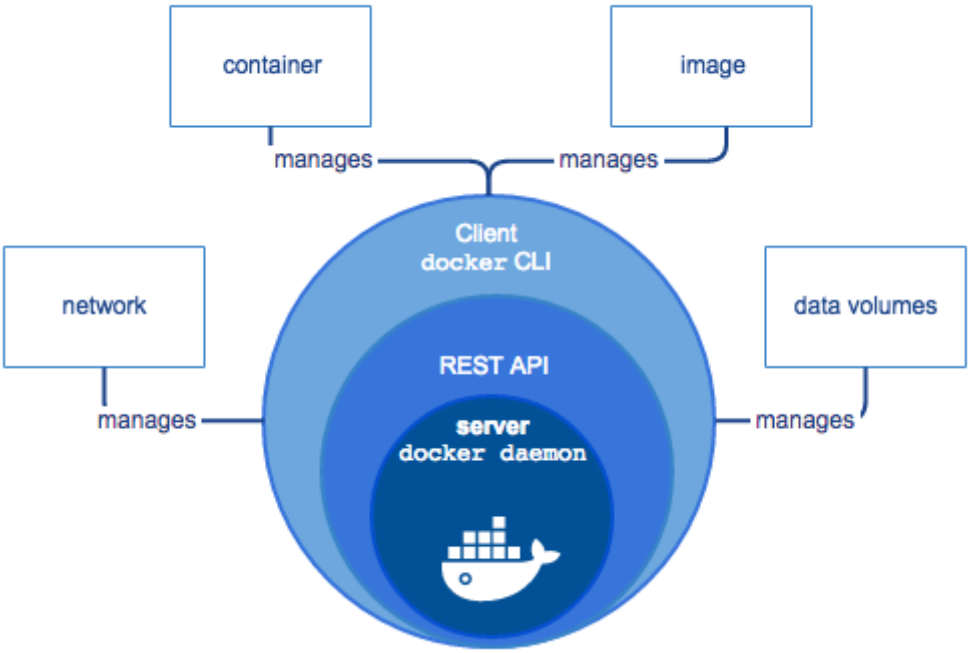
0.docker原理

1.docker是一个c/s架构主要组件包括

1. 常驻后台进程dockerd
2. 一个和dockerd交互的rest api server
3. 命令行cli接口 通过和rest api进行交互



- docker daemon : dockerd用来监听docker api的请求和管理docker镜像 容器 网络 卷
- docker client 是交互工具 如docker run docker build等命令
- image： 镜像是一个制度模板 带有镜像的说明
- container： 容器是一个镜像的可运行的实例 可以使用rest api或者cli来操作容器 容器的实质是进程 但与直接在宿主执行的实例进程不同，容器进程属于自己独立的namespace 因此容器用于自己的root文件系统 网络配置 进程空间 用户id

文章链接： https://i4t.com/4248.html

1.docker常见问题

- ```
  rrent[11684]: Error starting daemon: SELinux is not supported with the
  : docker.service: main process exited, code=exited, status=1/FAILURE
  : Failed to start Docker Application Container Engine.
  : Unit docker.service entered failed state.
  : docker.service failed.
  ```
- 原因：此linux内核中的selinux不支持overlay2 graph driver ，解决办法有两个
  - 启动一个新内核
  - 在docker里禁用selinux（--selinux-enabled=false）

    修改 /etc/sysconfig/docker中的--selinux-enabled=false
  - 重启docker

2.docker 自带仓库 registry
  ```

- 启动docker仓库容器

```
docker run -d -p 5000:5000 --restart=always --name registry registry
```

- 指定本地路径

```
$ docker run -d \
    -p 5000:5000 \
    -v /opt/data/registry:/var/lib/registry \
    registry
```

- 打标记

```
$ docker tag centos:latest 127.0.0.1:5000/centos:latest
```

- 使用 `docker push` 上传标记的镜像

```
$ docker push 127.0.0.1:5000/centos:latest
```

- docker登录Harbor

```
docker login 10.2.21.11:32023  #输入用户名密码

#1.标记镜像
docker tag {镜像名}:{tag} {Harbor地址}:{端口}/{Harbor项目名}/{自定义镜像名}:{自定义tag}
#eg:docker tag vmware/harbor-adminserver:v1.1.0
192.168.2.108:5000/test/harbor-adminserver:v1.1.0

#2.push 到Harbor
docker push {Harbor地址}:{端口}/{自定义镜像名}:{自定义tag}
#eg:docker push 192.168.2.108:5000/test/harbor-adminserver:v1.1.0

3.pull 到本地
docker pull 192.168.2.108：5000/test/harbor-adminserver:v1.1.0
```

3.docker 命令

1.docker基本命令

1. **attach**
    1.
2. **build**
    1. --rm=false 不删除临时镜像
    2. -f 使用自定义Dockerfile
    3. -t 添加标签
3. **commit**

    1. -a --autohr作者
    2. -c --change 改变参数 可更改参数：
       CMD|ENTRYPOINT|ENV|EXPOSE|LABEL|ONBUILD|USER|VOLUME|WORKDIR
    3. -m --message 提交一个commit备注信息
    4. -pause=false or true 在执行commit操作时 容器内所有进程是处于暂停状态 false表示不暂停
4. **cp**
    1. docker cp :/tmp/a.txt $(home)/dockerdata
5. **create**
    1. create命令只是创建一个container 并且在容器文件层的最上面添加一个读写层 容器里所有数据的变化都发生在读写层 当使用create命令后 在使用start命令启动这个容器
6. **diff**
7. **events**
    1. docker events --since '2020-07-01' or '2020-07-01T15:49:30'
    2. docker events --filter 'event=stop'
    3. docker events --filter 'container=id'
8. **exec**
    1. exec是容器内部执行命令的命令 在docker使用过程中 很少出现启动容器时直接创建pty的情况 因为在启动容器时创建pty 退出终端时 容器也会关闭
    2. docker exec id ps -ef
    3. docker exec id touch /tmp/a.sh
9. **logs**
    1. docker logs -f log不会退出 容器新产生的日志会显示出来
10. **rename**
    1. 重命名
11. **run**
    1. **docker run --pid=host reh17 strace -p 1234 xxxxxx** 通过这个命令可以让新容器访问主机pid=1234的容器的strace进程了（其他还包括 --ipc --uts）
    2. **docker run -d --cidfile=/tmp/id.log busybox** 通过指定cidfile docker运行后 将id写入cidfile中
    3. --**restart** : no always on-failure(在容器遭遇异常原因退出时--restart=on-failure:50)查看重启次数 **docker inspect -f '{{.HostConfig.RestartPolicy}}' mybusybox**
    4. **a**.--dns 设置一个dns服务器为容器（默认情况复用主机dns）**b**.--net **c**.--add-host **d**.--mac-address
    5. **内存 cpu等限制**
        1. -m 300M --memory-swap -1 : --memory-swap没有限制 默认memory+swap=2*memory
        2. -m 300M --memory-swap 1G : 允许swap+memory 等于1G
        3. 如果容器进程出现oom docker会强制杀死错误进程 想关闭此功能 使用 **--oom-kill-disable** 使用此功能时 容器要内存限制 否则有可能耗尽主机内存 非常危险
    6. docker run --privileged 那么docker将允许访问主机除了AppArmor和selinux之外的所有进程
12. **start**
    1. docker start -a xxxxx --attach docker会把容器中的stderr和stdout重定向到主机的stderr和stdout流中
13. **stats**
    1. 监控查看容器资源的命令 stats命令可以统计cpu使用率 内存使用率 网络吞吐量

    2. docker stats --no-stream a68fc01cbe4e
14. **tag**
    1. docker tag box newbox
15. **top**
    1. top命令统计容器资源状态 包括pid ppid 如果有必要kill某个进程 可以配合exec发送kill命令

2.docker资源命令

1. **export导出容器**
    1. docker export box > box.tar
    2. docker export --output="box.tar" box
    3. export 命令不会导出挂载卷组的数据

       docker run -it --rm -v 'pwd':/mountdir box:1.0 /bin/bash

       创建后查看mountdir目录 应该是和当前目录一致的

       导出tar包 发现mountdir为空
2. **images**
    1. --all 展示所有images
    2. --filter 筛选参数 key=value形式 目前key只有dangling和label，dangling表示悬空。意思是查找没有父文件层的临时文件层。这些文件主要产生的原因是在共建images时 修改Dockerfile中某一个命令 造成此命令之后所有临时文件层失效 虽然docker不会使用这些失效的临时文件层 但又没有删除 长久会产生很多没有父文件层的镜像数据
    3. docker images --filter "dangling=true"可以筛选出这些镜像
    4. 可以把镜像ID找出来一一删除 docker rmi $(docker images -f "dangling=true" -q)
3. **import**
    1. cat box.tar | docker import - image:tag(用户指定tag)
4. **load**
    1. load命令是加载镜像归档文件的命令 import加载容器导出的归档文件 load加载镜像归档文件 export导出的归档文件不包括文件系统的历史记录 只有最近一次的读写层数据 load命令导入image归档文件时会维护历史记录
    2. docker load --input box.tar
5. save
    1. docker save --output test.tar stefanprodan/podinfo

3.DockerFile

1. **FROM**
2. **MAINTAINER**
3. **RUN**
4. **CMD**
5. **LABEL**
6. **EXPOSE**
7. **ENV**
8. **ADD**
9. **COPY**
10. **ENTRYPOINT**
11. **VOLUME**
12. **USER**
13. **WORKDIR**
14. **ONBUILD**

4.docker namespace

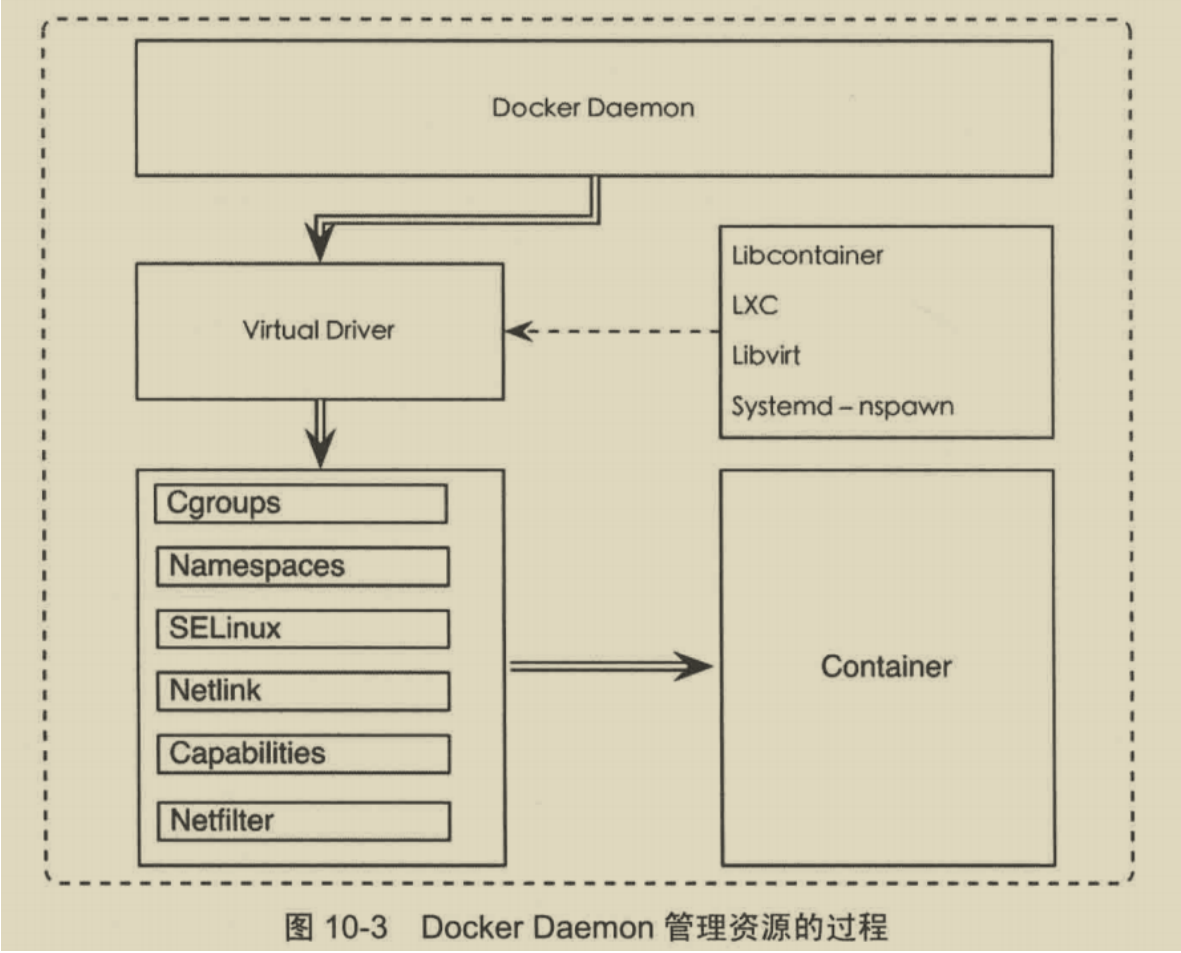


图 10-3　Docker Daemon 管理资源的过程

docker daemon 管理资源的过程

namespace包括： mount IPC Network PID User UTS

- NameSpace

1. IPC：linux进程通信方式包括：信号量 消息队列和共享内存 容器内部的进程通信 对于宿主机来说，就是具有相同PID的进程间通信。 因此，docker首先为容器创建一个IPC namespace 允许容器内所有进程通过全局唯一的32位标识符访问共享资源。

   需要注意的是 docker自身通信是通过tcp 或者socket进行 所以ipc namespace并不是为了容器自身使用 更多是为了容器内部的应用预留的 如果容器内部运行的应用需要使用message queue 就可以在同一宿主环境创建多个message queue 而不会产生干扰

2. PID namespace： 容器之间的进程树相互不可见 通过PID namespace 每个容器都会有一个进程号计数器 容器内所有的进程号会被重新编号 宿主机内核会维护各个容器中的进程树 在树最顶端的进程号是1 也就是init进程 此进程会作为容器内其他所有进程的父进程

   在宿主机中 次init进程只是一个普通的进程 docker cli发送stop或者kill命令的本质 就是向init进程发送SIGSTOP信号或者SIGKILL信号。一旦容器处于顶点的Init进程被销毁 那么和其处于同一个PID namespace的所有进程都将会受到内核发送的SIGSTOP和SIGKILL信号被销毁

3. UTS namespace 每个容器都拥有独立的主机名和域名资源

4. network namespace 一个典型的network namespace包括 ip协议栈 ip路由 端口信息 物理设备（网卡）在linux系统中，一个物理设备最多只能被包括在一个network namespace中， docker为每个容器的network创建一对虚拟网络设备 一个名为eth0 放置在容器中 另一个vethN连接docker0上

5. user namespace 隔离资源包括：用户ID 用户组ID 用户权限 但无论容器中的用户怎么变 始终对应的是宿主机环境中所创建容器的用户 而且这种关联关系通过 /proc/[pid]/uid_map 和 /proc/[pid]/gid_map 这两个文件予以保存
6. mount namespace 通过格力文件挂载点的方式提供隔离的文件系统 docker 为每个容器创建一个其独有的目录 并且将此容器所依赖的镜像文件层按照先父后子的顺序 逐层挂载到此目录当中 docker 会将当前目录设置为read-only模式 对此目录所作出的所有写操作都将体现到另一个目录 这个目录就是我们说的writable文件层

5.docker cgroup

1. **任务 task** 一个任务对应宿主机环境当中的一个 进程
2. **子系统 subsystem** 没一个子系统是对某一项具体物理资源的控制 cup子系统 memory子系统
3. **控制组 control group** cgroup中最基本的控制单元 一个group包含若干个任务（对应宿主环境的进程） 并且此group也会包含若干个子系统 用来控制group内的任务在指定子系统上面的资源使用
4. **层级树 hierarchy** cgroup的调度单位 由一个或多个group组成的树状结构 每个层级树通过绑定对应的子系统进行资源调度 同时子节点继承父节点的属性

- **cgroup共有十个子系统**
    - **blkio 块设备**（磁盘 硬盘 usb） 设备io限制
    - **cpuacct** cgroup中任务生成cpu资源使用报告
    - **cupset** 在多cpu系统中 为cgroup中的任务分配独立的cup和内存节点
    - **devices** 设置任务对物理设备的访问权限
    - **freezer** 挂起或者恢复cgroup中的任务
    - **memory** 设定cgroup中任务使用的内存限制 同时生成任务的内存资源使用报告
    - **net_cls** 使用等级识别符 标记网络数据包 同时使用linux流量控制程序识别从具体cgroup中生成的数据包
    - **net_prio** 对应用程序设置网络传输优先级 类似于socket 选项中的SO_PRIORITY
    - **HugeTLB** HugeTLB页的资源控制功能

通过查看 /proc/cgroups 可以查看当前操作系统支持的子系统

4.容器运行时 安全工具

- 开源工具

Anchore（https://anchore.com/ ）：漏洞分析与镜像扫描。
Apparmor（https://gitlab.com/apparmor/ ）：RASP功能。
Cilium（https://cilium.io/ ）：网络及HTTP层安全。
Coreos Clair（https://coreos.com/clair/docs/latest/ ）：静态代码分析。
Dagda（https://github.com/eliasgranderubio/dagda ）：静态漏洞分析与监视。
Saucelabs（https://saucelabs.com/open-source ）：免费实时自动化代码测试。

- 主流供应商

Alertlogic（https://www.alertlogic.com/solutions/container-security/）：管理容器身份和日志分析。
AquaSec（https://www.aquasec.com/）：RASP、审计、镜像扫描和容器IDS
Flawcheck（https://www.tenable.com/products/tenable-io/container-security）：被Tenable收购并融入其容器镜像扫描器以利用其Nessus安全专业技术。
Twistlock（https://www.twistlock.com/platform/）：RASP和附加机器学习防护。
Threatstack（https://www.threatstack.com/securing-containerized-environments）：作为漏洞监视工具融入其云安全平台。

]

5.docker 的空间使用分析和清理

```
#根据使用的存储驱动的不同，相应目录会有所不同
$du -h --max-depth=1 |sort
#Docker 的内置 CLI 指令 docker system df ，可用于查询镜像（Images）、容器（Containers）和本地卷（Local Volumes）等空间使用大户的空间占用情况
$docker system df
#可以进一步通过 -v 参数查看空间占用细节，以确定具体是哪个镜像、容器或本地卷占用了过高空间。示例输出如下
$docker system df -v
#清理空间
$docker system prune
WARNING! This will remove:
        - all stopped containers
        - all networks not used by at least one container
        - all images without at least one container associated to them
        - all build cache
####手动清理

#镜像清理
# 删除所有悬空镜像，但不会删除未使用镜像
$docker rmi $(docker images -f "dangling=true" -q)
# 删除所有未使用镜像和悬空镜像。
# 【说明】：轮询到还在被使用的镜像时，会有类似"image is being used by xxx container"的告警信息，所以相关镜像不会被删除，忽略即可
$docker rmi $(docker images -q)

#卷清理
$docker system df -v #查看卷 对于未被任何容器调用的卷（-v 结果信息中，"LINKS" 显示为 0）
# 删除所有未被任何容器关联引用的卷：
$docker volume rm $(docker volume ls -qf dangling=true)

# 也可以直接使用如下指令，删除所有未被任何容器关联引用的卷（但建议使用上面的方式）
# 【说明】轮询到还在使用的卷时，会有类似"volume is in use"的告警信息，所以相关卷不会被删除，忽略即可。
$docker volume rm $(docker volume ls -q)

#容器清理
# 删除所有已退出的容器
docker rm -v $(docker ps -aq -f status=exited)
# 删除所有状态为 dead 的容器
```

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

$docker ps -s #分析容器占用空间
# 如下容器的原始镜像占用了422MB空间，实际运行过程中只占用了2B空间：
CONTAINERID  IMAGE   COMMAND  CREATED   STATUS  PORTS   NAMES        SIZE
ac3912      word    ntrypoin      3      11   80/tcp  Web_web_4
2B(virtual422MB)

#使用Device Mapper 存储驱动限制容器磁盘空间 如果使用device mapper 作为底层驱动 则可以通过
Docker daemon 参数 全局限制单个容器的空间大小
--storage-opt dm.basesize=20G

#使用btrfs   btrfs驱动主要使用btrfs所提供的的subvolume功能实现
$ btrfs qgroup limit -e 50G /var/lib/docker/btrfs/subvolumes/<CONTAINER_ID>

#文章链接：https://yq.aliyun.com/articles/272173（其中包括 如何给容器服务的Docker增加
数据盘）
```

6.载出和载入镜像

```
$ docker save -o calico_node_v3.8.2.tar calico/node:v3.8.2
$ docker load --input calico_node_v3.8.2.tar
```

7.docker-compose network网段设置

```
# docker version:  18.06.0+
# docker-compose version: 1.23.2+
# OpenSSL version: OpenSSL 1.1.0h
version: "3.7"
services:
  web:
    image: alenx/walle-web:2.1
    container_name: walle-nginx
    hostname: nginx-web
    ports:
      # 如果宿主机80端口被占用，可自行修改为其他port(>=1024)
      # 0.0.0.0:要绑定的宿主机端口:docker容器内端口80
      - "80:80"
    depends_on:
      - python
    networks:
      - walle-net
    restart: always

  python:
    image: alenx/walle-python:2.1
    container_name: walle-python
    hostname: walle-python
    env_file:
      # walle.env需和docker-compose在同级目录
      - ./walle.env
    command: bash -c "cd /opt/walle_home/ && /bin/bash admin.sh migration &&
python waller.py"
    expose:
      - "5000"
    volumes:
```

```
      - /opt/walle_home/plugins/:/opt/walle_home/plugins/
      - /opt/walle_home/codebase/:/opt/walle_home/codebase/
      - /opt/walle_home/logs/:/opt/walle_home/logs/
      - /root/.ssh:/root/.ssh/
    depends_on:
      - db
    networks:
      - walle-net
    restart: always

  db:
    image: mysql
    container_name: walle-mysql
    hostname: walle-mysql
    env_file:
      - ./walle.env
    command: [ '--default-authentication-plugin=mysql_native_password', '--
character-set-server=utf8mb4', '--collation-server=utf8mb4_unicode_ci']
    ports:
      - "3306:3306"
    expose:
      - "3306"
    volumes:
      - /data/walle/mysql:/var/lib/mysql
    networks:
      - walle-net
    restart: always

networks:
  walle-net
    driver: bridge
    ipam:
      config:
        - subnet: 172.33.255.1/24
```

8.docker daemon 远程访问

1.docker daemon的连接方式

1. Unix域套接字
    1. 默认方式 会生成一个 /var/run/docker.sock文件 Unix域套接字用于本地进程间的通信
2. tcp端口监听
    1. 服务端开启端口监听 dockerd -H IP:port 客户端通过docker -H IP:port 访问（不安全）

2.更改docker配置

- 在/etc/docker/daemon.json 中添加如下内容（现在docker的配置信息都在daemon文件中配置 没有需要创建）

```
{
  "hosts" : ["unix:///var/run/docker.sock", "tcp://0.0.0.0:2375"]
}
#"unix:///var/run/docker.sock" unix socket ：本地客户端通过这个连接docker daemon
#"tcp://0.0.0.0:2375" tcp  ：  表示允许任何远程客户端通过2375访问（2376加密）
```

3.更改系统服务配置

- 通过**systemctl edit docker** 来调用文本编辑器修改单元 新建或者修改 /etc/systemd/system/docker.service.d/override.conf

```
##Add this to the file for the docker daemon to use different ExecStart
parameters (more things can be added here)
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd
```

- 默认情况下使用systemd时， docker.service的设置是 ExecStart=/usr/bin/dockerd -H fd:// ，这将覆盖daemon.json中的任何hosts 通过override.conf 文件将ExecStart=/usr/bin/dockerd 这将会使用在daemon.json中设置的hosts 这个文件中的第一行必须有 ExecStart= 用于清除默认的 ExecStart参数

4.重新加载daemon和重启docker服务

```
systemctl daemon-reload
systemctl restart docker
```