一、Docker 简介

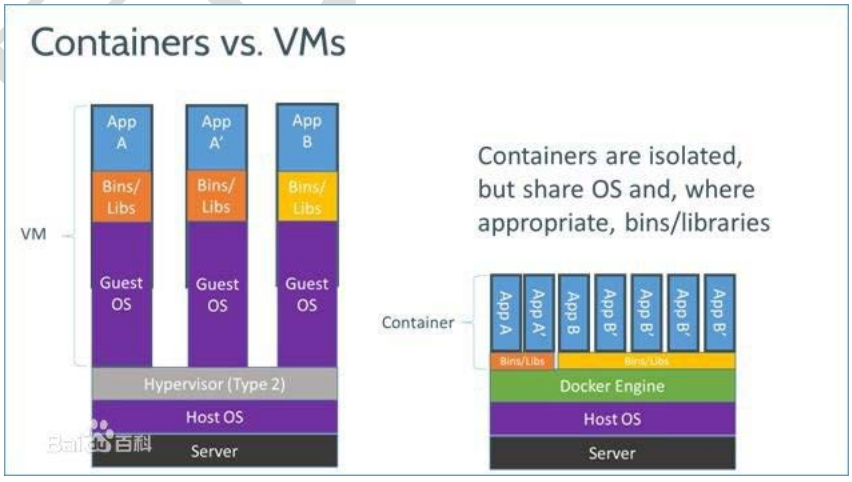
1、Docker 诞生

Docker 是 dotcloud 公司开源的一款产品 dotcloud 是 2010 年新成立的一家公司，主要基于 PAAS (Platfrom as a Service) 平台为开发者提供服务。2013 年 10 月 dotcloud 公司改名为 Docker 股份有限公司

2、Docker 相关解释

- Linux Container 是一种内核虚拟化技术，可以提供轻量级的虚拟化，以便隔离进程和资源
- Docker 是 PAAS 提供商 dotCloud 开源的一个基于 LXC 的高级容器引擎，源代码托管在 Github 上，基于 go 语言并遵从 Apache2.0 协议开源
- Docker 设想是交付运行环境如同海运，OS 如同一个货轮，每一个在 OS 基础上的软件都如同一个集装箱，用户可以通过标准化手段自由组装运行环境，同时集装箱的内容可以由用户自定义，也可以由专业人员制造

3、Docker 与传统虚拟化对比



4、Docker 的构成

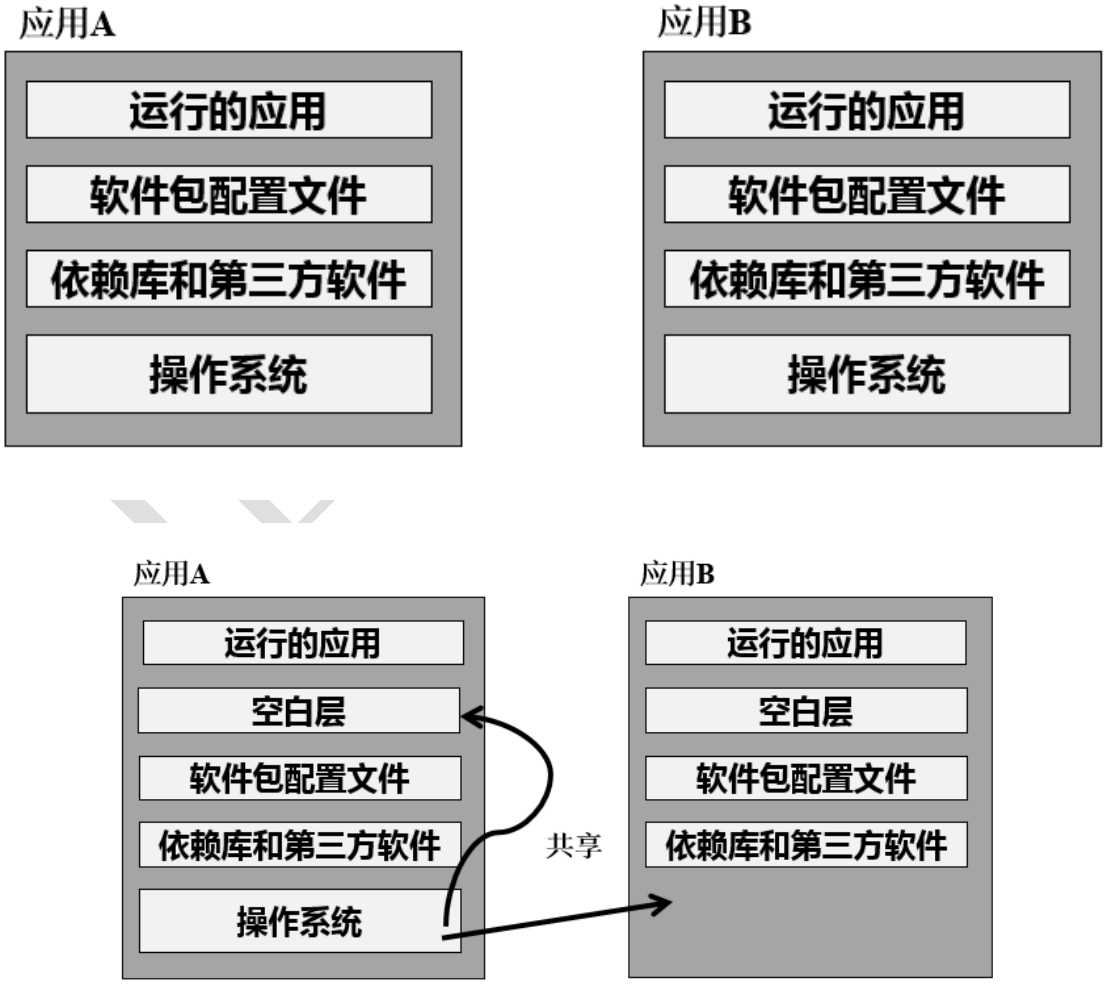
- **Docker 仓库：https://hub.docker.com**
- **Docker 自身组件**

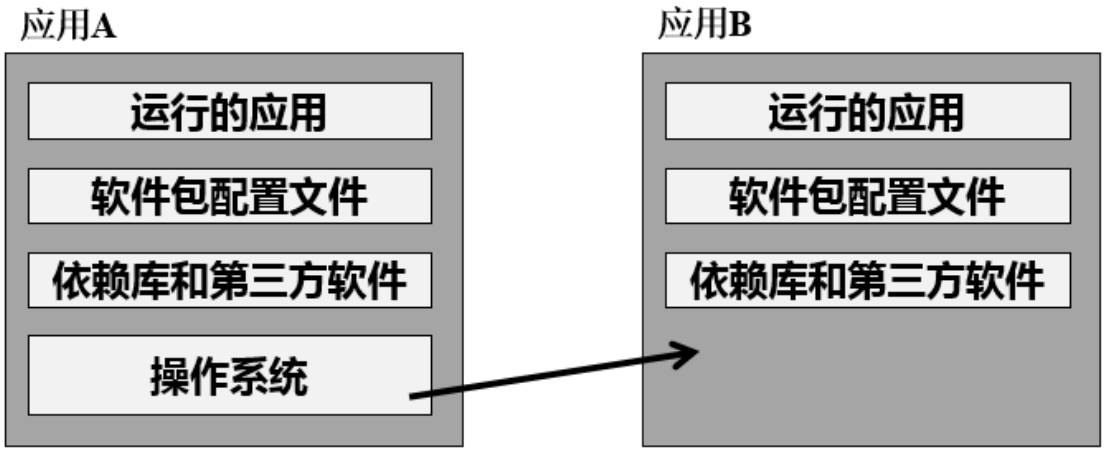
  > Docker Client：Docker 的客户端

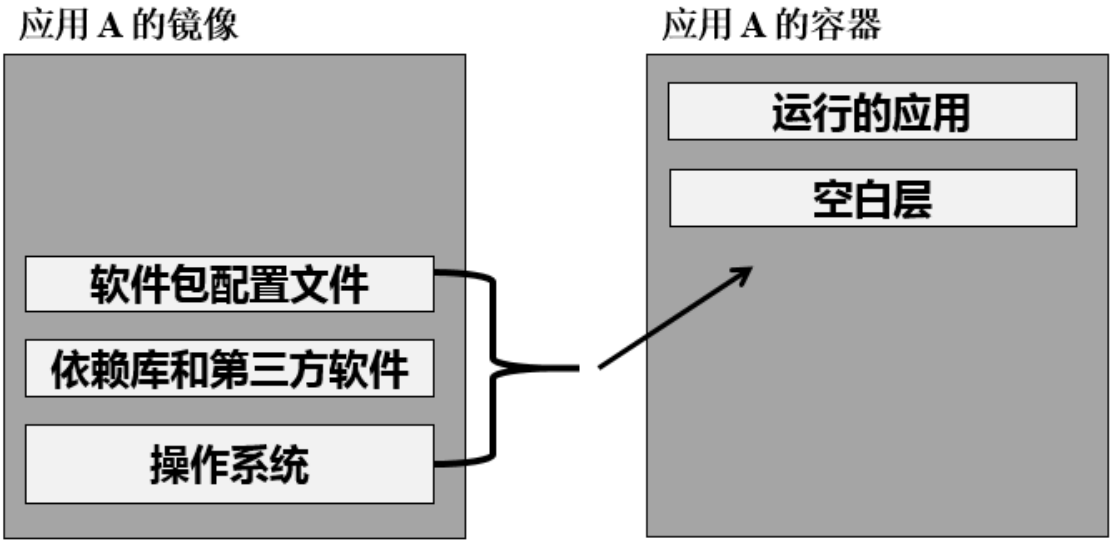
  > Docker Server：Docker daemon 的主要组成部分，接受用户通过 Docker Client 发出的请求，并按照相应的路由规则实现路由分发

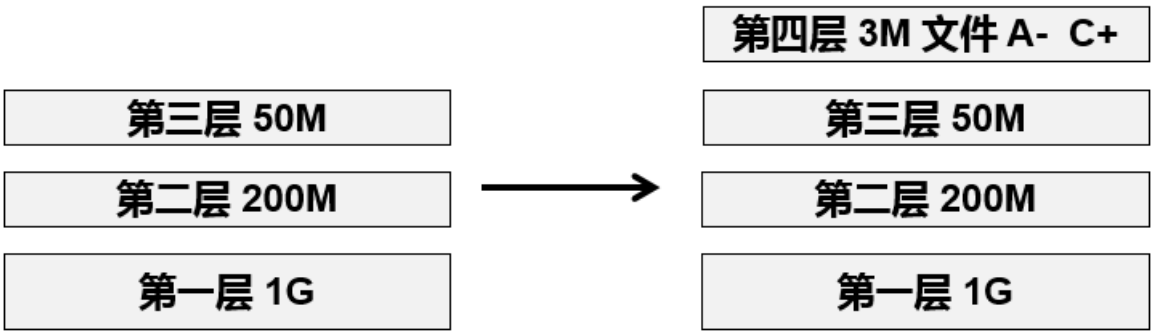
  > Docker 镜像：Docker 镜像运行之后变成容器（docker run）

5、Docker 历经过程




应用A
运行的应用
软件包配置文件
依赖库和第三方软件
操作系统
应用B
运行的应用
软件包配置文件
依赖库和第三方软件


应用 A 的镜像
软件包配置文件
依赖库和第三方软件
操作系统
应用 A 的容器
运行的应用
空白层


第三层 50M
第二层 200M
第一层 1G
第四层 3M 文件 A- C+
第三层 50M
第二层 200M
第一层 1G

二、Docker 安装

1、Docker 的安装方式

Script Install

```
yum update
$ curl -sSL https://get.docker.com/ | sh
systemctl start docker
systemctl enable docker
docker run hello-world
```

YUM Install

```
yum update
cat >/etc/yum.repos.d/docker.repo <<-EOF
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg EOF
yum install docker
```

RPM Install

https://download.docker.com/linux/centos/7/x86_64/stable/Packages/

.

2、Docker 镜像加速配置

```
cp /lib/systemd/system/docker.service    /etc/systemd/system/docker.service
chmod 777 /etc/systemd/system/docker.service
vim       /etc/systemd/system/docker.service
      ExecStart=/usr/bin/dockerd-current    --registry-mirror=https://kfp63jaj.mirror.aliyuncs.com \
systemctl     daemon-reload
systemctl     restart     docker
ps     -ef     |     grep docker
```

阿里云 Docker 官网：https://dev.aliyun.com/search.html

3、Docker 化应用体验

环境分析

WordPress 运行环境需要如下软件的支持：

- PHP 5.6 或更新软件
- MySQL 5.6 或 更新版本
- Apache 和 mod_rewrite 模块

代码展现

```
docker run --name db --env MYSQL_ROOT_PASSWORD=example -d mariadb
docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

三、Docker 容器管理

1、Docker 基础概念

Docker 三个重要概念：仓库 (Repository)、镜像 (image) 和 容器 (Container)

docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress

Docker 指令的基本用法：

docker + 命令关键字(COMMAND) + 一系列的参数

2、Docker 基础命令

docker info	**守护进程的系统资源设置**
docker searchDocker	**仓库的查询**
docker pull Docker	**仓库的下载**
docker images	**Docker 镜像的查询**
docker rmi	**Docker 镜像的删除**
docker ps	**容器的查询**
docker run	**容器的创建启动**
docker start/stop	**容器启动停止**

 # Docker 指令除了单条使用外，还支持赋值、解析变量、嵌套使用

3、单一容器管理命令

每个容器被创建后，都会分配一个 **CONTAINER ID** 作为容器的唯一标示，后续对容器的启动、停止、修改、删除等所有操作，都是通过 **CONTAINER ID** 来完成，偏向于数据库概念中的主键

docker ps --no-trunc	**查看**
docker stop/start CONTAINERID	**停止**
docker start/stop MywordPress	**通过容器别名启动/停止**
docker inspect MywordPress	**查看容器所有基本信息**
docker logs MywordPress	**查看容器日志**
docker stats MywordPress	**查看容器所占用的系统资源**
docker exec 容器名 容器内执行的命令	**容器执行命令**
docker exec -it 容器名 /bin/bash	**登入容器的 bash**

4、Run 常用的一些参数

--restart=always	容器的自动启动
-h x.xx.xx	设置容器主机名
-dns xx.xx.xx.xx	设置容器使用的 DNS 服务器
--dns-search	DNS 搜索设置
--add-host hostname:IP	注入 hostname <> IP 解析
--rm	服务停止时自动删除

5、Docker-Compose

Docker-compose installl

```
curl -L https://github.com/docker/compose/releases/download/1.14.0/docker-compose-`uname
-s`-`uname -m` > /usr/local/bin/docker-compose
docker version
```

Docker-compose 用法

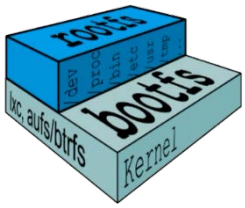
-f	指定使用的 yaml 文件位置	**up -d**	启动容器项目
ps	显示所有容器信息	**pause**	暂停容器
restart	重新启动容器	**unpause**	恢复暂停
logs	查看日志信息	**rm**	删除容器
config -q	验证 yaml 配置文件是否正确		
stop	停止容器		
start	启动容器		

演示代码记录

```
version: '2'

services:
    db:
      image: mysql:5.7
```

```
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    restart: always
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
```

四、Docker 镜像管理



1、镜像的特性

容器创建时需要指定镜像，每个镜像都由唯一的标示 Image ID ，和容器的 Container ID 一样，默认 128 位，可以使用前 16 为缩略形式，也可以使用镜像名与版本号两部分组合唯一标示，如果省略版本号，默认使用最新版本标签（latesr）

镜像的分层：Docker 的镜像通过联合文件系统（union filesystem）将各层文件系统叠加在一起

> bootfs：用于系统引导的文件系统，包括 bootloader 和 kernel，容器启动完成后会被卸载以节省内存资源

> roofs：位于 bootfs 之上，表现为 Docker 容器的跟文件系统

>> 传统模式中，系统启动时，内核挂载 rootfs 时会首先将其挂载为“只读”模式，完整性自检完成后将其挂载为读写模式

>> Docker 中，rootfs 由内核挂载为“只读”模式，而后通过 UFS 技术挂载一个“可写” 层

2、容器转换为镜像

docker commit CID xx.xx.xx

3、Dockerfile

Dockfile 是一种被 Docker 程序解释的脚本，Dockerfile 由一条一条的指令组成，每条指令对应 Linux 下面的一条命令。Docker 程序将这些 Dockerfile 指令翻译真正的 Linux 命令。Dockerfile 有自己书写格式和支持的命令，Docker 程序解决这些命令间的依赖关系，类似于 Makefile。Docker 程序将读取 Dockerfile，根据指令生成定制的 image

转换命令

docker build -t wangyang/jdk-tomcat .

Dockerfile 语法

1、FROM（指定基础 image）：
构建指令，必须指定且需要在 Dockerfile 其他指令的前面。后续的指令都依赖于该指令指定的 image。FROM 指令指定的基础 image 可以是官方远程仓库中的，也可以位于本地仓库

example：
    FROM centos:7.2
    FROM centos

2、MAINTAINER（用来指定镜像创建者信息）：
构建指令，用于将 image 的制作者相关的信息写入到 image 中。当我们对该 image 执行 docker inspect 命令时，输出中有相应的字段记录该信息。

example：
    MAINTAINER wangyang "wangyang@itxdl.cn"

3、RUN（安装软件用）：
构建指令，RUN 可以运行任何被基础 image 支持的命令。如基础 image 选择了 Centos，那么软件管理部分只能使用 Centos 的包管理命令

example：

RUN cd /tmp && curl -L 'http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.8/bin/apache-tomcat-7.0.8.tar.gz' | tar -xz

RUN ["/bin/bash", "-c", "echo hello"]

4、CMD（设置 container 启动时执行的操作）：

设置指令，用于 container 启动时指定的操作。该操作可以是执行自定义脚本，也可以是执行系统命令。该指令只能在文件中存在一次，如果有多个，则只执行最后一条

example：

CMD echo “Hello, World!”

5、ENTRYPOINT（设置 container 启动时执行的操作）：

设置指令，指定容器启动时执行的命令，可以多次设置，但是只有最后一个有效。

example：

ENTRYPOINT ls -l

#该指令的使用分为两种情况，一种是独自使用，另一种和 CMD 指令配合使用。当独自使用时，如果你还使用了 CMD 命令且 CMD 是一个完整的可执行的命令，那么 CMD 指令和 ENTRYPOINT 会互相覆盖只有最后一个 CMD 或者 ENTRYPOINT 有效

CMD 指令将不会被执行，只有 ENTRYPOINT 指令被执行

CMD echo “Hello, World!”

ENTRYPOINT ls -l

#另一种用法和 CMD 指令配合使用来指定 ENTRYPOINT 的默认参数，这时 CMD 指令不是一个完整的可执行命令，仅仅是参数部分；ENTRYPOINT 指令只能使用 JSON 方式指定执行命令，而不能指定参数

FROM ubuntu

CMD ["-l"]

ENTRYPOINT ["/usr/bin/ls"]

6、USER（设置 container 容器的用户）：

设置指令，设置启动容器的用户，默认是 root 用户

example：

USER daemon = ENTRYPOINT ["memcached", "-u", "daemon"]

7、EXPOSE（指定容器需要映射到宿主机器的端口）：设置指令，该指令会将容器中的端口映射成宿主机器中的某个端口。当你需要访问容器的时候，可以不是用容器的 IP 地址而是使用宿主机器的 IP 地址和映射后的端口。要完成整个操作需要两个步骤，首先在 Dockerfile 使用 EXPOSE 设置需要映射的容器端口，然后在运行容器的时候指定-p 选项加上 EXPOSE 设置的端口，这样 EXPOSE 设置的端

口号会被随机映射成宿主机器中的一个端口号。也可以指定需要映射到宿主机器的那个端口，这时要确保宿主机器上的端口号没有被使用。EXPOSE 指令可以一次设置多个端口号，相应的运行容器的时候，可以配套的多次使用-p 选项。

example：

映射一个端口

EXPOSE 22

相应的运行容器使用的命令

docker run -p port1 image

映射多个端口

EXPOSE port1 port2 port3

相应的运行容器使用的命令

docker run -p port1 -p port2 -p port3 image

还可以指定需要映射到宿主机器上的某个端口号

docker run -p host_port1:port1 -p host_port2:port2 -p host_port3:port3 image

8、ENV（用于设置环境变量）：构建指令，在 image 中设置一个环境变量

example：

设置了后，后续的 RUN 命令都可以使用，container 启动后，可以通过 docker inspect 查看这个环境变量，也可以通过在 docker run --env key=value 时设置或修改环境变量。假如你安装了 JAVA 程序，需要设置 JAVA_HOME，那么可以在 Dockerfile 中这样写：

ENV JAVA_HOME /path/to/java/dirent

9、ADD（从 src 复制文件到 container 的 dest 路径）

example：

ADD <src> <dest>

<src> 是相对被构建的源目录的相对路径，可以是文件或目录的路径，也可以是一个远程的文件 url;

<dest> 是 container 中的绝对路径

10、COPY （从 src 复制文件到 container 的 dest 路径）

example：

COPY <src> <dest>

10、VOLUME（指定挂载点）：

设置指令，使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使用，也可以共享给其他容器使用。我们知道容器使用的是 AUFS，这种文件系统不能持久化数据，当容器关闭后，

所有的更改都会丢失。当容器中的应用有持久化数据的需求时可以在 Dockerfile 中使用该指令

examp：

```
FROM base
VOLUME ["/tmp/data"]
```

11、WORKDIR（切换目录）：设置指令，可以多次切换(相当于 cd 命令)，对 RUN,CMD,ENTRYPOINT 生效

example：

```
WORKDIR /p1 WORKDIR p2 RUN vim a.txt
```

12、ONBUILD（在子镜像中执行）：ONBUILD 指定的命令在构建镜像时并不执行，而是在它的子镜像中执行

example：

```
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
```

4、镜像的导出以及导入

导出：docker save -o xx.xx.xx xx.xx.xx.tar

导入：docker load -i xx.xx.xx.tar

五、镜像仓库构建

1、官方仓库构建方式

仓库服务器配置：

```
docker run -d -v /opt/registry:/var/lib/registry -p 5000:5000 --restart=always registry

vim /etc/docker/daemon.json
{
    "insecure-registries": ["10.10.10.11:5000"]
}
```

客户机设置：

vim /etc/sysconfig/docker
--insecure-registry 10.10.10.11:5000　　增加

curl -XGET http://10.10.10.11:5000/v2/_catalog　　查看已有镜像

2、Harbor 构建

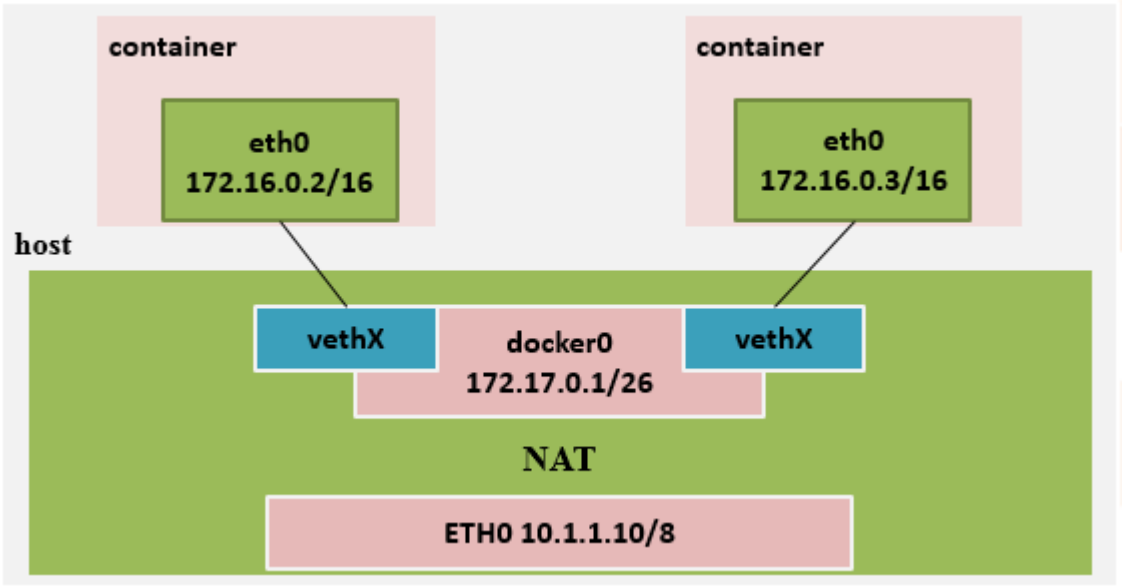
详情见目录下 Harbor 构建 pdf

六、Docker 中的网络

1、Linux 内核中的 NameSpace

namespace	系统调用参数	隔离内容	内核版本
UTS	CLONE_NEWUTS	主机名和域名	2.6.19
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存	2.6.19
PID	CLONE_NEWPID	进程编号	2.6.24
NetWork	CLONE_NEWNET	网络设备、网络栈、端口等	2.6.29
Mount	CLONE_NEWNS	挂载点（文件系统）	2.4.19
User	CLONE_NEWUSER	用户和用户组	3.8

2、Docker 网络通讯示意图



3、Docker 中的防火墙规则

容器访问外部网络

iptables -t nat -A POSTROUTING -s 172.17.0.0/16 -o docker0 -j MASQUERADE

外部网络访问容器

docker run -d -p 80:80 apache

iptables -t nat -A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER

iptables -t nat -A DOCKER ! -i docker0 -p tcp -m tcp --dport 80 -j DNAT --to-destination 172.17.0.2:80

4、Docker 网络的修改

Docker 进程网络修改

-b, --bridge=”” 指定 Docker 使用的网桥设备，默认情况下 Docker 会自动创建和使用 docker0 网桥设备，通过此参数可以使用已经存在的设备

--bip 指定 Docker0 的 IP 和掩码，使用标准的 CIDR 形式，如 10.10.10.10/24

--dns 配置容器的 DNS，在启动 Docker 进程是添加，所有容器全部生效

Docker 容器网络修改

--dns 用于指定启动的容器的 DNS

--net 用于指定容器的网络通讯方式，有以下四个值

- **bridge：Docker 默认方式，网桥模式**
- **none：容器没有网络栈**
- **container：使用其它容器的网络栈，Docker 容器会加入其它容器的 network namespace**
- **host：表示容器使用 Host 的网络，没有自己独立的网络栈。容器可以完全访问 Host 的网络，不安全**

5、端口的暴露方式

-p / P 选项的使用格式

> -p <ContainerPort>：将制定的容器端口映射至主机所有地址的一个动态端口

> -p <HostPort>:<ContainerPort>：映射至指定的主机端口

> -p <IP>::<ContainerPort>：映射至指定的主机的 IP 的动态端口

> -p <IP>:<HostPort>:<ContainerPort>：映射至指定的主机 IP 的主机端口

> -P（大）：暴露所需要的所有端口

*** docker port ContainerName 可以查看容器当前的映射关系**

6、网络隔离

基础命令说明

docker network ls 查看当前可用的网络类型

docker network create -d 类型 网络空间名称

类型分为：

overlay network
bridge network

使用网络名字空间进行隔离代码记录

```
docker network create -d bridge --subnet "172.26.0.0/16" --gateway "172.26.0.1" my-bridge-network

docker run -d --network=my-bridge-network --name test1    hub.c.163.com/public/centos:6.7-tools

docker run -d --name test2    hub.c.163.com/public/centos:6.7-tools
```

使容器配置上独立 IP 进行通讯

A、配置真是网桥

```
[root@localhost network-scripts]# vi ifcfg-eth0
        DEVICE=eth0
        HWADDR=00:0C:29:06:A2:35
        TYPE=Ethernet
        UUID=34b706cc-aa46-4be3-91fc-d1f48c301f23
        ONBOOT=yes
        BRIDGE=br0
        NM_CONTROLLED=yes
        BOOTPROTO=static

[root@localhost network-scripts]# vi ifcfg-br0
        //改成这样
        DEVICE=br0
        TYPE=Bridge
        ONBOOT=yes
        BOOTPROTO=static
        IPADDR=192.168.216.131
        NETMASK=255.255.255.0
        GATEWAY=192.168.216.2
```

DNS=8.8.8.8

B、使用工具分配地址

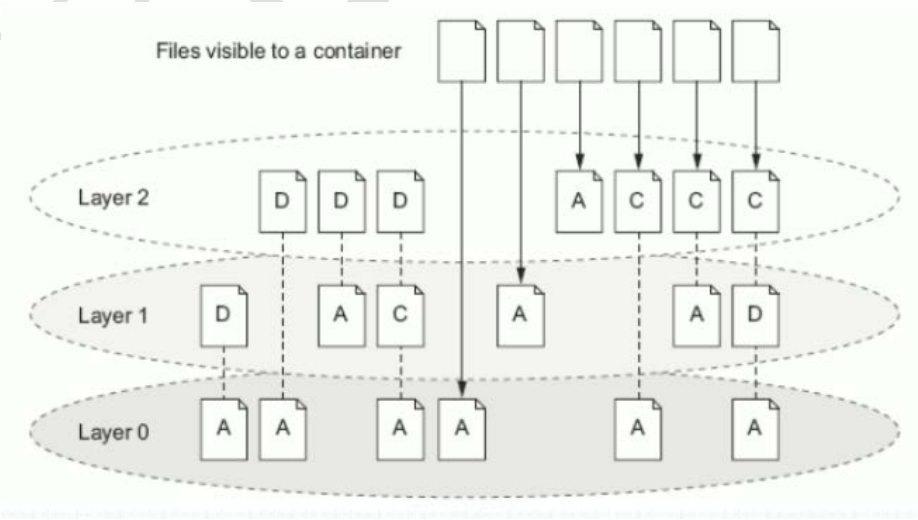
```
[root@localhost network-scripts]# yum install -y git

[root@localhost network-scripts]# git clone https://github.com/jpetazzo/pipework

[root@localhost network-scripts]# cp pipework/pipework /usr/local/bin/

[root@localhost network-scripts]# docker run -itd --net=none --name=ff centos-6-x86 bash
[root@localhost network-scripts]# pipework br0 fl 192.168.216.135/24
```

七、数据存储

1、数据卷特性

> Docker 镜像由多个只读层叠加而成，启动容器时，Docker 会加载只读镜像层并在镜像栈顶部添加一个读写层

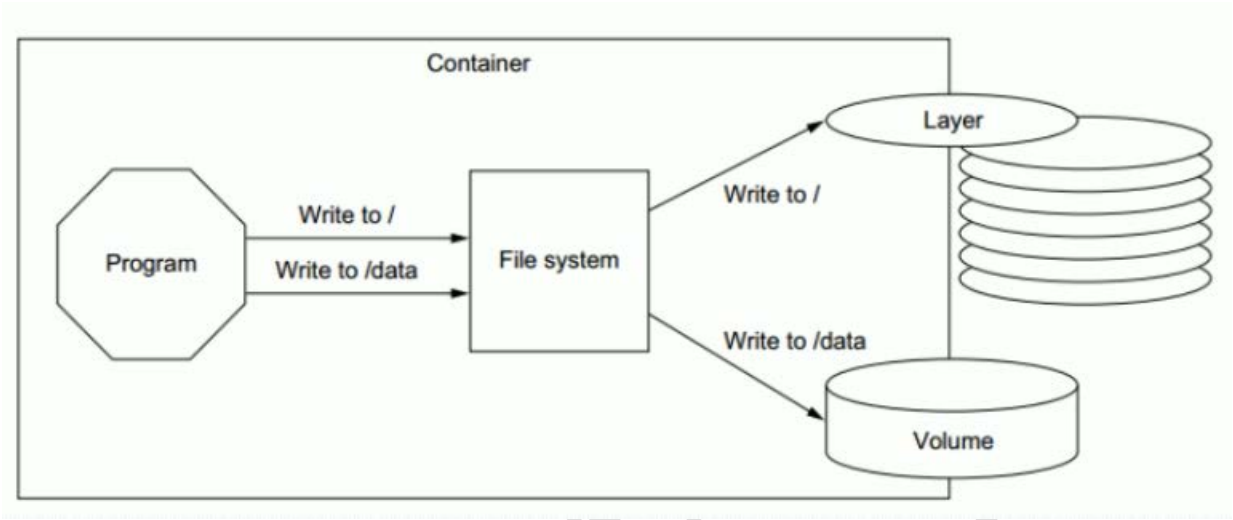
> 如果运行中的容器修改了现有的一个已经存在的文件，那么该文件将会从读写层下面的的只读层复制到读写层，该文件的只读版本仍然存在，只是已经被读写层中该文件的副本所隐藏，次即“写时复制”机制



2、数据卷的意义

> Volume 可以在运行容器时即完成创建与绑定操作。当然，前提需要拥有对应的申明

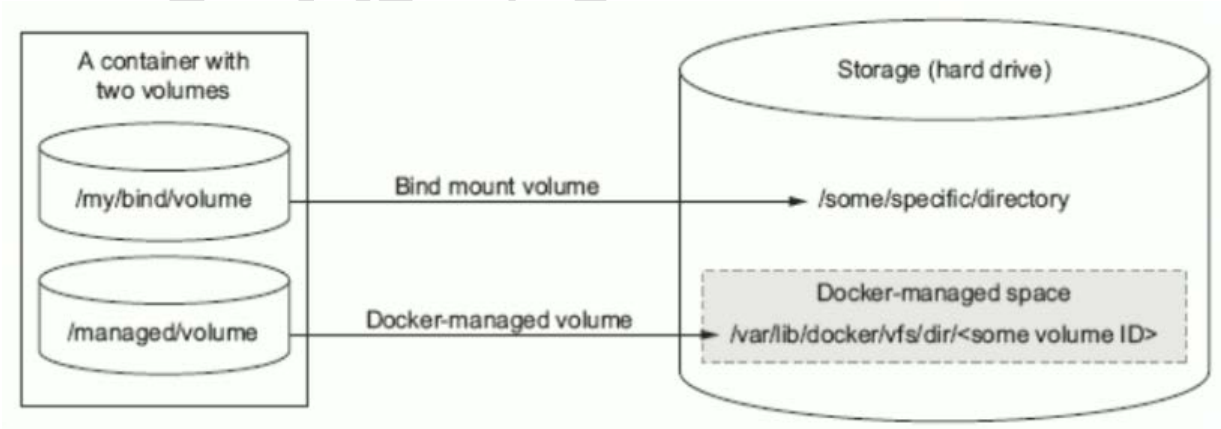
> Volume 的初衷就是数据持久化



3、数据卷的类型

> Bind mount volume

> Docker-managed volume



4、容器中使用数据卷的方法

> Docker-managed Volume

>> docker run -it --name roc -v MOUNTDIR roc/lamp:v1.0

>> docker inspect -f {{.Mounts}} roc

> Bind-mount Volume

>> docker run -it --name roc -v HOSTDIR:VOLUMEDIR roc/lamp:v1.0

> Union Volume

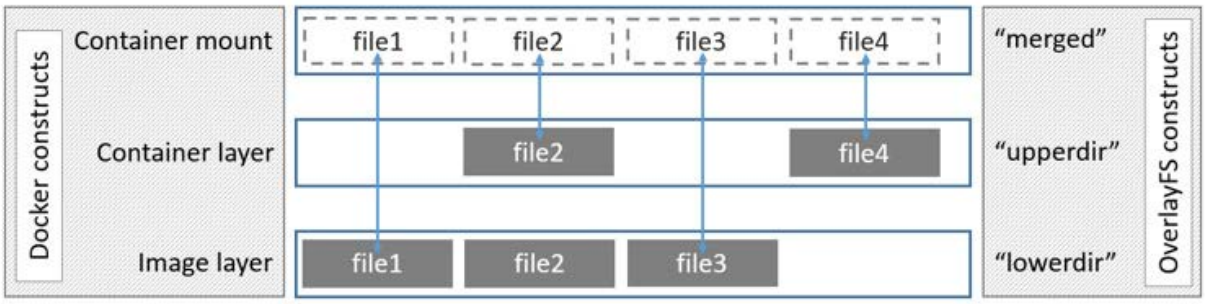
>> docker run -it --name roc --volumes-from ContainerName roc/lamp:v1.0

5、存储驱动

Docker 存储驱动（storage driver）是 Docker 的核心组件，它是 Docker 实现分成镜像的基础

- device mapper (DM)：性能和稳定性存在问题，不推荐生产环境使用
- btrfs：社区实现了 btrfs driver，稳定性和性能存在问题
- overlayfs：内核 3.18 overlayfs 进入主线，性能和稳定性优异，第一选择

Docker overlayfs driver



mount -t overlay overlay -olowerdir=./low,upperdir=./upper,workdir=./work ./merged

修改为 overlayfs 存储驱动

```
echo "overlay" > /etc/modules-load.d/overlay.conf
cat /proc/modules|grep overlay
reboot
vim /etc/systemd/system/docker.service
    --storage-driver=overlay \
```

八、资源限制

1、内存资源限制

相关说明

> CGroup 是 Control Groups 的缩写，是 Linux 内核提供的一种可以限制、记录、隔离进程组 (process groups) 所使用的物力资源 (如 cpu memory i/o 等等) 的机制。2007 年进入 Linux 2.6.24 内核，CGroups 不是全新创造的，它将进程管理从 cpuset 中剥离出来，作者是 Google 的 Paul Menage

> 默认情况下，如果不对容器做任何限制，容器能够占用当前系统能给容器提供的所有资源

> Docker 限制可以从 Memory、CPU、Block I/O 三个方面

> OOME：Out Of Memory Exception

>> 一旦发生 OOME，任何进程都有可能被杀死，包括 docker daemon 在内

>> 为此，Docker 调整了 docker daemon 的 OOM 优先级，以免被内核关闭

重点提示

> 为应用做内存压力测试，理解正常业务需求下使用的内存情况，然后才能进入生产环境使用

> 一定要限制容器的内存使用上限

> 尽量保证主机的资源充足，一旦通过监控发现资源不足，就进行扩容或者对容器进行迁移

> 如果可以（内存资源充足的情况），尽量不要使用 swap，swap 的使用会导致内存计算复杂，对调度器非常不友好

2、内存限制设置方式

在 docker 启动参数中，和内存限制有关的包括（参数的值一般是内存大小，也就是一个正数，后面跟着内存单位 b、k、m、g，分别对应 bytes、KB、MB、和 GB）：

- -m --memory：容器能使用的最大内存大小，最小值为 4m

- **--memory-swap：容器能够使用的 swap 大小**
- **--memory-swappiness：默认情况下，主机可以把容器使用的匿名页（anonymous page）swap 出来，你可以设置一个 0-100 之间的值，代表允许 swap 出来的比例**
- **--memory-reservation：设置一个内存使用的 soft limit,设置值小于 －m 设置**
- **--kernel-memory：容器能够使用的 kernel memory 大小，最小值为 4m。**
- **--oom-kill-disable：是否运行 OOM 的时候杀死容器。只有设置了 -m，才可以把这个选项设置为 false，否则容器会耗尽主机内存，而且导致主机应用被杀死**

3、参数示意图

--memory-swap	--memory	功能
正数S	正数M	容器可用总空间为S，其中ram为M，swap为(S-M)，若S=M，则无可用swap资源
0	正数M	相当于未设置swap (unset)
unset	正数M	若主机(Docker Host)启用了swap，则容器的可用swap为2*M
-1	正数M	若主机(Docker Host)启用了swap，则容器可使用最大至主机上的所有swap空间的swap资源
注意：在容器内使用free命令可以看到的swap空间并不具有其所展现出的空间指示意义。		

4、CPU 资源限制

相关说明

Docker 提供的 CPU 资源限制选项可以在多核系统上限制容器能利用哪些 vCPU。而对容器最多能使用的 CPU 时间有两种限制方式：

- **一是有多个 CPU 密集型的容器竞争 CPU 时，设置各个容器能使用的 CPU 时间相对比例**
- **二是以绝对的方式设置容器在每个调度周期内最多能使用的 CPU 时间**

CPU 限制方式

--cpuset-cpus="" 允许使用的 CPU 集，值可以为 0-3,0,1

-c,--cpu-shares=0 CPU 共享权值（相对权重），默认值 1024

--cpuset-mems="" 允许在上执行的内存节点（MEMs）

--cpu-period=0 即可设置调度周期，CFS 周期的有效范围是 1ms~1s，对应的--cpu-period 的数值范围是 1000~1000000

--cpu-quota=0 设置在每个周期内容器能使用的 CPU 时间，容器的 CPU 配额必须不小于 1ms，即 --cpu-quota 的值必须 >= 1000，单位微秒

--cpus 能够限制容器可以使用的主机 CPU 个数，并且还可以指定如 1.5 之类的小数

Example

```
docker run -it --cpu-period=50000 --cpu-quota=25000 ubuntu:16.04 /bin/bash
docker run -it --cpu-period=10000 --cpu-quota=20000 ubuntu:16.04 /bin/bash
```

5、限制性实验

```
> docker run --name stress -it --rm -m 256m lorel/docker-stress-ng:latest stress -vm 2
> docker run --name stress -it --rm --cpus 2 lorel/docker-stress-ng:latest stress --cpu 8
> docker run --name stress -it --rm --cpuset-cpus 0 lorel/docker-stress-ng:latest stress --cpu 8
```

九、补充

1、配置 Docker 远程访问

修改 /etc/docker/daemon.json 文件

```
"hosts": ["tcp://0.0.0.0:2375","unix://var/run/docker.sock"]
```

example:

docker -H Ip:Port Command

2、容器标准化结构

Open Container Initiative

> 由 Linux 基金会主导于 2015 年 6 月创立

> 旨在围绕容器格式和运行时定制一个开放的工业化标准

Contains two specifications

> the Runtime Specification (runtime-spec)

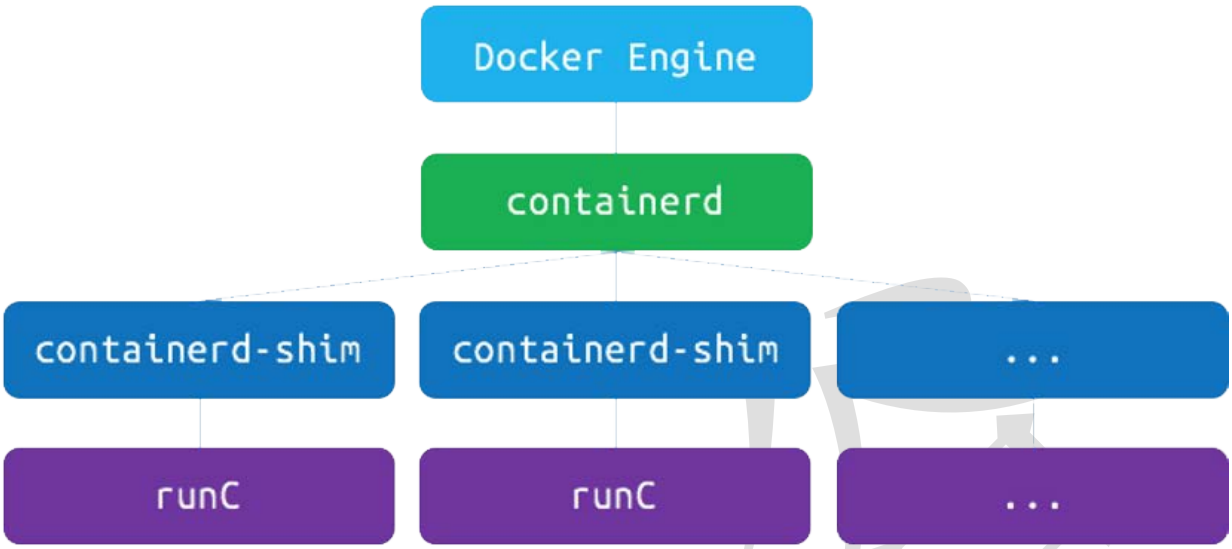
> the Image Specification (runtime-spec)

RUNC ？

是对于 OCI 标准的一个参考实现，是一个可以用于创建和运行容器的 CLI (command-line interface)工具。runC 直接与容器所依赖的 cgroup/linux kernel 等进行交互，负责为容器配置 cgroup/namespace 等启动容器所需的环境，创建启动容器的相关进程。

为了兼容 OCI 标准，Docker 也做了架构调整。将容器运行时相关的程序从 Docker daemon 剥离出来，形成了 Containerd。Containerd 向 Docker 提供运行容器的 API，二者通过 grpc 进行交互。Containerd 最后会通过 Runc 来实际运行容器

RUNC 架构图



3、常见命令关联图