Doker基础

走进容器云（一）

@Tranq

2019/4

目录

CONTENTS

1

蓝鲸鱼展头

容器技术发展历程

2013 dotCloud开源docker，Docker开源项目诞生，迅速崛起并在云平台产业与群雄对抗，将容器技术应用推向成熟。

容器技术发展历程

- 1979 UNIX chroot
- 2001 Linux VServer
- 2007 cgroups(google)
- 2008 LXC(namespace, cgroups)

- 2014后 docker项目崛起（镜像）（14~15）。
- CoreOS Rocket
- 2014 Docker Swarm
- 2014 Kubernets

- 2016 Windows原生Docker诞生
- 2016 Docker放弃Swarm
- 2016后 k8s社区繁荣，百家争鸣 (CNCF)
- 2017 Containerd捐献给CNCF
- 2017 Moby, Docker-CE, Docker-EE

➢ docker开源（dotCloud）

➢ RunC (libcontainer)
➢ OCI(Open Container Initivative)

➢ containerd行业标准引擎

2013 → 2015 → 2018

◆ 08~13 PaaS领域群雄逐鹿 (Gboogle IBM, RedHat)
◆ CloudFundry 标志性PaaS

◆ “CaaS” vs. CloudFundry
◆ PaaS深入人心市场开拓，用户诟病经典PaaS产品 。
◆ 2014 Docker平台化商业计划

◆ Google、RedHat等基础设施厂家联合成立CNCF.
◆ Docker, K8s, Mesos容器编排”三足鼎立“（15~17底）
◆ 2017.10 Docker-EE内置k8s。

◆ 2018.1 RedHat收购CoreOS
◆ 2018.3 Solomon辞职

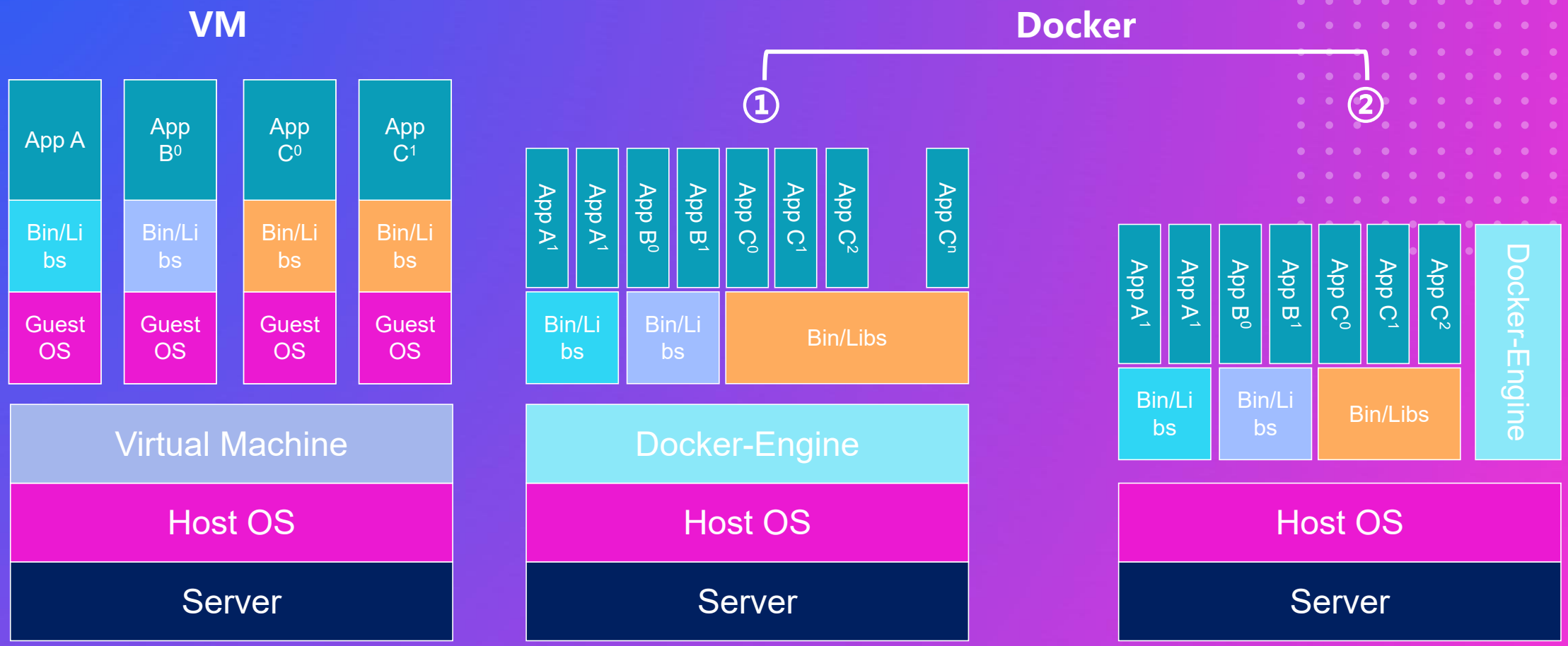
2

核心原理

Docker核心原理：
隔离、控制、打包

三驾马车：namespace， cgroups, rootfs

Docker与虚拟机



- 相同：为应用程序创建了相互隔离的“沙箱”
- 区别：VM对物理硬件的抽象；Docker对OS的抽象。

Docker之进程隔离：namespace

- 宏观角度看隔离：
  - 我的眼里只有我（进程）
  - 感知不到周围的人（其他进程）存在
  - 资源（文件、网络）都为我所用

- 微观角度（OS）：

路径/proc/[pid]/ns

namespace	隔离资源
IPC	信号量、消息队列、共享内存、POSIX message queues
Network	网络设备、协议栈、端口等
Mount	文件系统挂载点
PID	进程编号
User	用户合用户组
UTS	主机名与NIS域名

Docker之进程资源控制：cgroups

- Linux control group(cgroups)用来限制一个进程组能够使用的资源上线，包括CPU、内存、网络带宽、磁盘等。
- cgroups以/sys/fs/cgroup下的文件系统作为用户接口。下属cpuset, cpu, memory等子目录为cgroup子系统。
- cgroup每个子系统（子目录）下包含了改子系统提供的资源控制项目；
- cgroup每个子系统下可以创建子目录，子目录创建时自动创建对应子系统支持的控制项目。该子目录形成一个“控制组”。

例如：docker run -it --cpu-period=100000 --cpu-quota=20000 ubuntu /bin/bash

- docker在cpu子系统下创建一个目录（例如container_0）并将cpu-period, cpu-quota等值写入对应的控制文件。
- 待业务进程起来后，将该进程PID（宿主机中真实PID）写入tasks文件

Docker之rootfs

- 让应用“打包”更容易（系统、库、配置依赖）
- 作为一个整体任意搬运：开发、测试、生产部署

- 进程的根文件系统（rootfs），早期有chroot.
- 如何解决依赖复用?
- 联合文件系统（Union File System）
  - aufs
  - overlay/overlay2
  - devicemapper

Docker之精髓：分层镜像（Union File System-based）

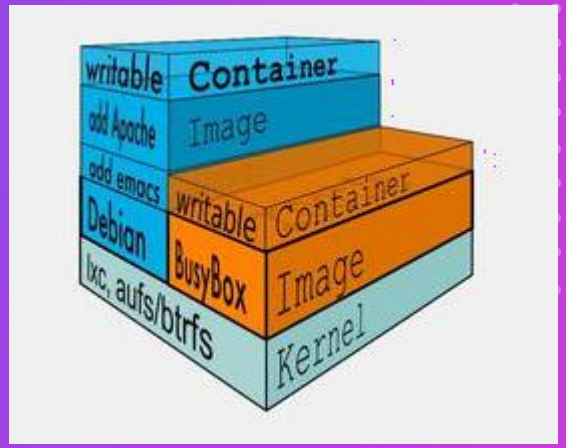


图1：分层镜像

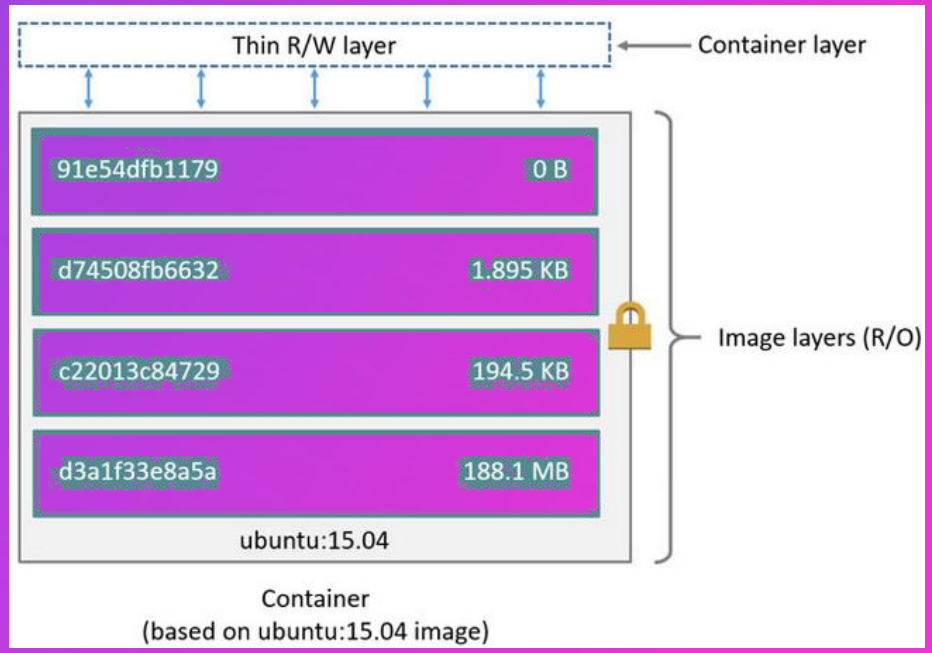


图2：
容器文件系统由只读镜像层+可读写容器层

Docker镜像构建（1）

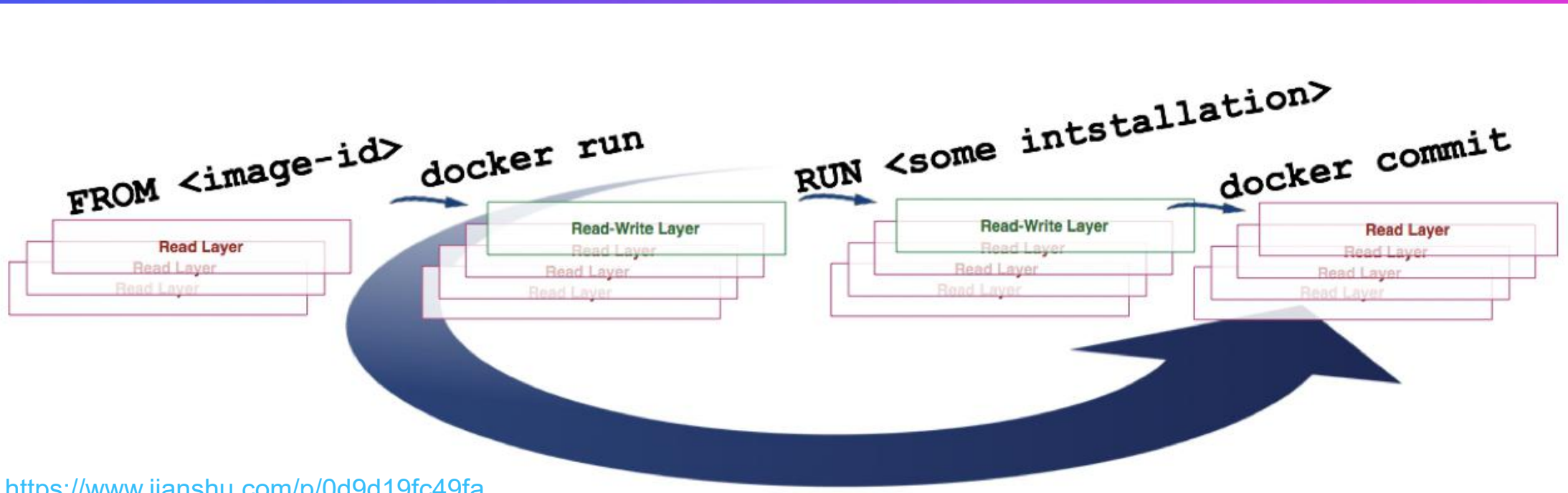
两种构建镜像方法：

- docker commit <container_id>
- Dockerfile
  - FROM 指令指定base image
  - 三类base image: 原始镜像，服务镜像，空镜像(scratch)
  - 每个指令都会生产一个Layer（需防止Layer太多）

- Dockerfile指令
  - FROM (必需，且必须为第一条)
  - RUN
  - CMD
- Docker主进程 vs. daemon进程
  - 容器启动时运行的应用进程如果是daemon,那么容器会立即结束。

Docker镜像构建（2）

- docker commit VS. Dockerfile
- Dockerfile 自动化执行多个指令



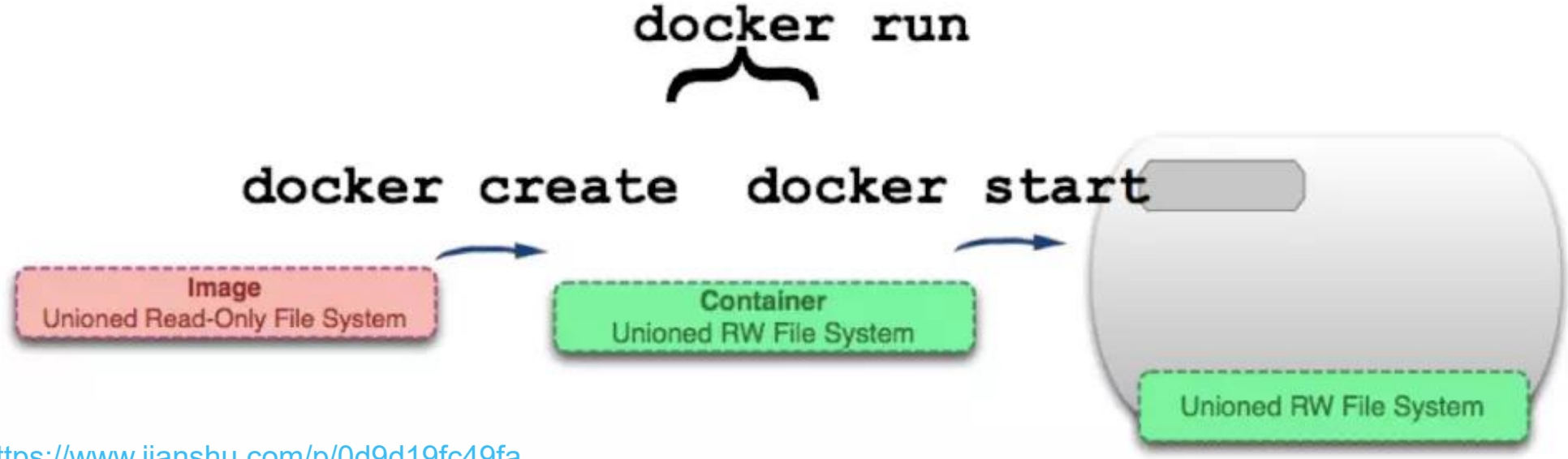
https://www.jianshu.com/p/0d9d19fc49fa

Docker镜像与容器

- 抽象的角度：容器是镜像的动态表现（类似 程序 vs. 进程）
- 文件系统角度：容器在分层镜像基础之上增加可读写层（参考）。

由镜像创建容器：

- docker run （创建后并运行）
- docker create （仅创建）
  - docker start <container_id>



https://www.jianshu.com/p/0d9d19fc49fa

3

常用命令

docker cli 常用命令

docker help | docker <cmd> help

Docker常用命令

- docker images
- docker commit <container_id>
- docker build
- docker pull
- docker save/load
- docker export/import

- docker create <image>
- docker start/restart/stop
- docker run
- docker ps [-a]
- docker exec -it <container_id>
- docker top <container_id>
- docker help / CMD help

4

其他及讨论

小结、学与用

容器云另外两大基石：平台化（k8s），容器云中的网络。

小结

- Docker vs. VM
  - Docker: 面向操作系统内核的抽象
  - VM:面向机器的抽象
- Docker vs. Other container
  - docker镜像解决应用发布、部署的易用性问题。
  - 采用分层镜像解决了不同容器共享相同依赖问题。
  - 镜像、容器的联合文件系统。

Docker 存在的问题/使用限制

- 隔离不彻底
  - /proc, /sys, /dev/sd*等项目未完全隔离；例如，cat /proc/meminfo在容器内看到内容与宿主机中一样。
  - selinux, time, syslog等未隔离。例如，date修改时间影响所有容器。
- cgroups只能限制资源使用最大量，不能保证不被其他进程占用。
- 资源高受限环境中程序自动配置
  - JavaSE(<8u131)需要手工设定Heap最大值以防OOM。

学与用：思考题

- Linux Docker中能否运行window的原生应用?
- 宿主机Linux内核4.0，其Docker中能否运行需要Linux 内核4.4依赖的应用?
- 在一个Docker集群中，Registry是否必须?

一套容器云还需要：

- 相关容器直接的“合理”部署、启/停
- 部署、迁移自动化
- 服务发现（容器实现的业务）
- 日志收集
- 监控&安全

处处留心皆学问，
问遍千家事必明！