窥探 Docker

---

renyl 2015/10/26

目录

1 Docker 背景

1) 随着云计算的快速发展，虚拟化技术显得愈发重要。由于传统的 VM 存在启动慢、占用资源多、效率利用低等缺点，使得一种新的技术—容器（Container）得以快速发展。

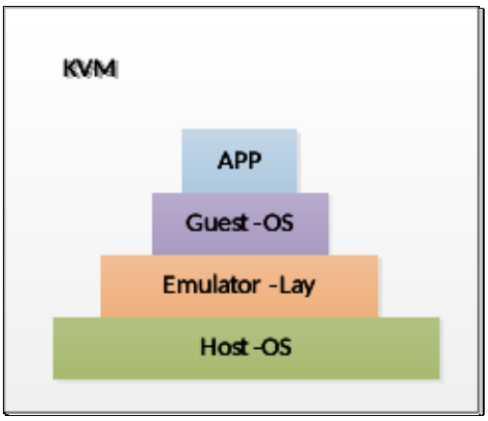
2) Linux Containers 是一种操作系统级（内核轻量级）虚拟化技术，相对传统虚拟化技术 KVM，具有如下特点：
   a) Container 与 Host 共用一个内核。
   b) Container 相当于一个轻量级 APP sandbox。
   c) Container 作为一个普通进程在 Host 上运行。
   d) Container 不需要 Host 的 CPU 支持虚拟化。

3) Container 相对于 KVM 具有如下优缺点：

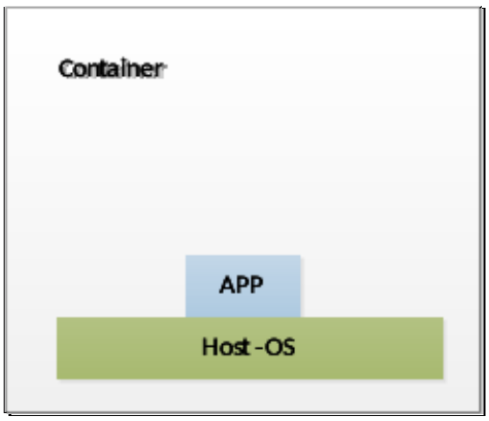
ID	ITEM	Container	KVM
1	Performance	Great	Normal
2	OS support	Linux only	No limit
3	Security	Normal	Great
4	Completeness	Low	High
5	Complexity	Low	High

Container 和 KVM 的系统架构图如下所示：

KVM 系统架构



Container 系统架构



4) Linux Containers 相对传统的虚拟化技术，虽然解决了性能相关的问题，但是在“云”时代环境下，应用程序的快速部署、可移植性、版本控制等重要问题仍没有解决。在这种背景下，一种基于 Linux Containers 的新理念产品 Docker 应运而生。

2 Docker 介绍

1) Docker 是 Docker.Inc 公司开源的一个基于 LXC 技术之上构建的 Container 容器引擎，源代码托管在 GitHub 上，基于 Google 公司推出的 Go 语言并遵从 Apache2.0 协议开源。

2) Docker 是一个开源的应用容器引擎，它能帮助开发者打包应用以及应用的依赖包，并构建为一个可移植的容器，从而发布到任何流行的 Linux、Windows 等操作系统上。

3) Docker 利用 Linux 内核的一些核心技术（如 Cgroup、Namespaces 和 SELinux）来支撑容器之间的隔离。起初 Docker 只是 LXC 容器管理子系统的前端，不过在其版本稳定后（0.9 版），使用 Go 语言重写了一套类 LXC 接口实现（即 libcontainer 项目），这个原生的 Go 语言库提供了用户空间和内核之间的接口，为 Docker 的多平台发展打了下基础。

4) Docker 将自己定位为一个“分发应用的开放平台”，其官方网站上也明确地提到了 Docker 的典型应用场景：

   a) 对应用进行自动打包和部署(Automating the packaging and deployment of applications)

   b) 创建轻量、私有的 PaaS 环境(Creation of lightweight, private PaaS environments)

   c) 自动化测试和持续整合与部署(Automated testing and continuous integration/deployment)

   d) 部署和扩展 Web 应用、数据库和后端服务(Deploying and scaling web apps, databases and backend services)

5) Docker 的发展过程：

   a) 2013 年 3 月 20 日，Docker0.1 首次发布，拉开了基于云计算平台发布产品方式的变革序幕。

   b) 2014 年 6 月 10 日，Docker1.0 正式版发布，标志着 Docker 自身已经转变为一个分发应用开放平台。

   c) 2014 年 12 月 4 日，Docker 发布了首个商业产品---“跨容器的分布式应用编排服务”，由 Docker Machine、Docker Swarm 以及 Docker Composer 3 个组件组成，编排服务开放了原生的接口，可以保证应用的可移植性。

6) Docker 的发展催生了新的 Linux 发行版，如：CoreOS 和 Project Atomic（Red Hat 公司推出的），它们设计成能运行容器的最小环境系统。近日，基于容器技术发展起来的 CoreOS 推出了自己的容器技术 Rocket 欲与 Docker 竞争，不过目前来说 Docker 为容器这一领域的王者。

7) Docker 已被各大 IT 公司（如 Google、Facebook、Amazon、IBM、Microsoft、Alibaba）支持在其“云”上运行。百度的 BAE（Baidu App Engine）更是以 Docker 作为其 PaaS 云基础。

8) Docker 是基于 Image 文件来创建 Container 的，使得 Docker 可以简化部署多种应用实例环境，如：Web 应用、后台应用、数据库应用、大数据应用、消息队列等都可以打包成一个 Image 文件来进行部署。

9) Docker 是由构建（build）、交付（ship）、运行（run）三部分组成的。Docker 成功地将“交付”和“运行”解耦，这样源自任意 Docker 版本的镜像都可以和其它任意不同版本一起工作（向前和后向均可兼容），这就为 Docker 应用提供了稳定的基础，以应对快速的变化。

10) Docker 出现之前，可以使用 Pupet、Chef、Ansible 等配置管理工具把复杂的配置管理起来。但在 Docker 技术之后，，结合 Docker 的开发部署工具 Fig，可以通过镜像的方式简化环境的安装，系统的依赖问题得了到彻底的解决。

11) Docker 容器要比虚拟机更加效率，这是因为容器可以共享系统内核和相关的库。

12) Docker 把 LXC 技术实现了商业化，让 Container 具有可移植性、版本控制、组件重用、易于部署、可共享等特点。Docker 相对于 LXC 和 KVM 具有如下优缺点：

ID	ITEM	Docker	LXC	KVM
1	Performance※1	Great	Great	Normal
2	OS support	Not only Linux※2	Linux only	No limit
3	Security	Normal	Normal	Great
4	Completeness	Low	Low	High
5	Complexity	Low	Low	High
6	Workloads	Low	Low	High
7	Share	Yes	No	No
8	Portability	Yes	No	No
9	Deployment	Easy	Difficult	Difficult

※1：

IBM 的研究团队针对 Linux Containers 和 Virtual Machines 的性能情况发布了一个报告，表明容器各方面的性能都要优于虚拟机，详细可参考地址：
http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

※2：

a) 目前 Docker 可在 Linux、Windows、Mac OS X 上运行，但是由于 Docker Engine 使用了 Linux 内核相关特性（Linux-specific kernel features），使得 Docker 在 Windows 和 Mac OS X 上运行需要安装一个叫“Boot2Docker”的程序来建立一个轻量级的虚拟机。

b) Docker 已和微软已达成合作，确定了在下一个版本的 Windows Server 中自带 Docker Engine，使得 Docker 的跨平台移植得以实现。

注 1：Red Hat 在 RHEL7 中开始正式支持 Docker。
注 2：本文在如下平台下进行 Docker 研究及测试。

-	描述
os	RHEL7.0_x86_64
kernel	kernel-3.10.0-110.el7.x86_64
cpu	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
glibc	glibc-2.17-52.el7.x86_64。
docker	Docker version 1.4.1

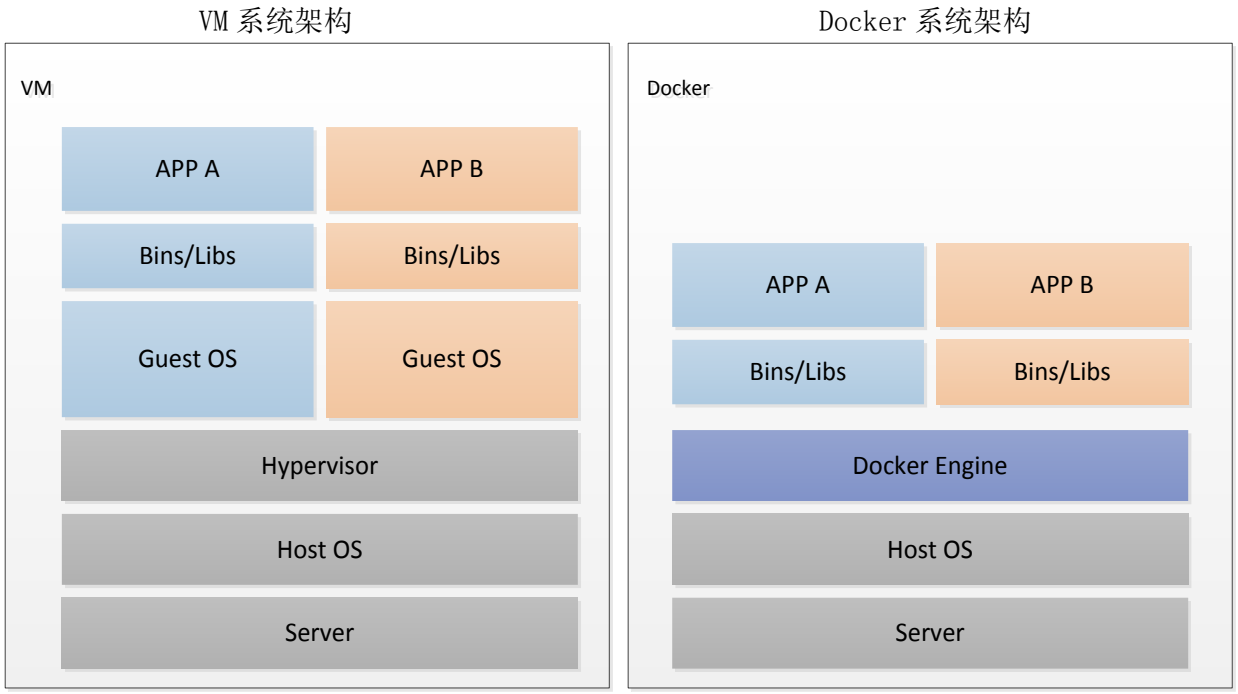
3 Docker 原理

3.1 Docker Feature

Docker 是一个开放平台（open platform）用来构建（build）、交付（ship）、运行（run）分布式 APP，其与传统的 VM 相比有如下特点：

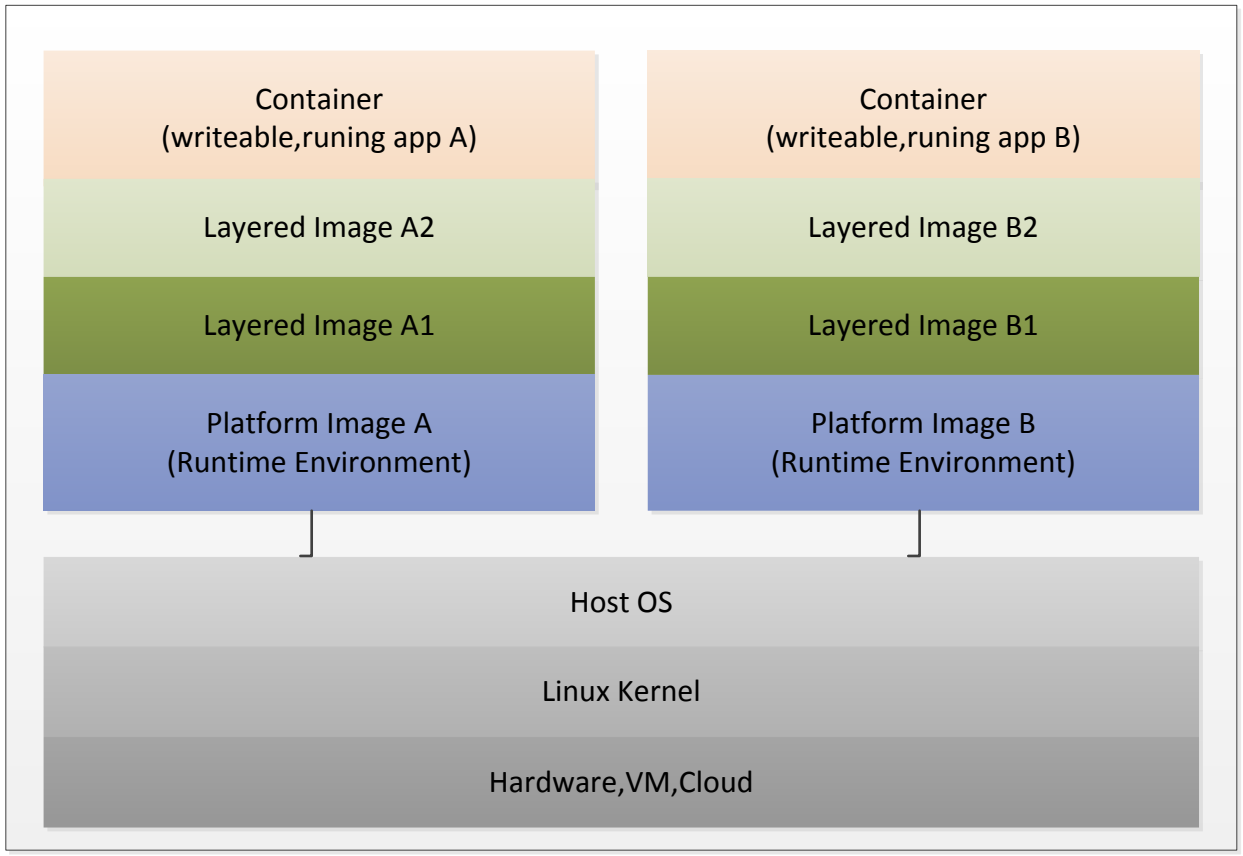
1) Docker Container 与 Host OS 共享内核。
2) Docker Container 作为一个独立的进程运行在用户空间。
3) Docker Container 是轻量级的，运行效率高、执行速度快。
4) Docker Container 启动速度快（基本 1 秒钟即可）。

Docker 和 VM 的系统架构如下所示：



3.2 Docker Image

Docker 是基于 image 文件来创建 Container 的，其架构如下所示：



说明：

1) Container 包含 APP 运行所需要的一切环境，其是基于 image 文件创建的。当基于 image 文件启动一个 Container 时将会在这个 image 文件之上加一个 writeable layer。

2) Image 是个 read-only layer，从不会被修改，所有的修改都是在最上面的 writeable layer。每次 commit 一个 Container 时都会在原来的 image 文件之上添加一个新的 image layer 来进行存储，采用的是 Copy-On-Write 机制。

   注：
   在 Container 中做任何修改（如添加、修改、删除文件）后不进行 commit 操作的话，下次重新启动 Container 时会发现之前在 Container 中所做的修改都没有保存下来。因为 Container 都是基于 image 文件启动的，使用相同的 image 启动 Container 后生成的 writeable-lay 内容就不会改变。

3) 每个 Image 都依赖一个或多个 parent images，唯一没有 parent image 的就是 Platform image。

4) Platform Image 包含了 APP 运行时所必须的运行时环境（the runtime environment）、相关包和相关工具。

3.3 Docker Technology

Docker 是使用 Go 语言编写而成的，其使用了 Linux Kernel 内核的 Namespaces、Cgroup 和 SELinux & AppArmor 技术以及 AUFS 机制，详细介绍如下：

3.3.1 Namespaces

Linux Containers 依赖 Namespaces 技术对 Container 进行隔离，通过如下 6 个 namespace 来实现：

1) Mount namespace
用于隔离文件系统挂载相关信息。在 mount namespace 内的进程进行 mount/umount 操作将只会在该 mount namespace 内可见，因此可为进程提供独有的文件系统层次结构。

2) Uts namespace
用于隔离 Container OS 版本相关信息。每个 uts namespace 拥有自己的 ostype, osrelease, version, hostname, domainname。

3) Ipc namespace
用于隔离进程间的通信。处于同一 ipc namespace 的进程才可以互相通信，由于不同 Container 不在同一 ipc namespace，因此不同 Container 中的进程无法互相通信。

4) Net namespace
用于隔离网络相关的资源。每个 net namespace 拥有自己的 net device、IP address、防火墙规则、路由规则等。

5) Pid namespace
用于隔离进程的 PID。Host 和 Container 中可以存在同样的 PID，在 Container 只能查看 Container 中的进程，无法查看其它 Container 或者 Host 上的进程。但在 Host 上可以查看到 Container 下的进程，不过它们被分配不同的 PID。

注：pid namespace 在内核中实现为进程分层结构。如：父 pid namespace 可以看到子 pid namespace 的所有进程，但子 pid namespace 看不到父 pid namespace 的信息。在创建进程时，除了在进程所属的 pid namespace 中申请 pid 外，还需要在父 pid namespace 中申请 pid。

6) User namespace
主要是为了解决安全问题。通过将 Container 上的 root 用户映射为 Host 上的普通用户，可以防止 Container 上的 root 用户进行加载/卸载模块等会对 Host 造成影响的操作。

注：Namespace 相关的 System Call 有 clone、unshare、setns。

3.3.2 Cgroup

Linux Containers 依赖 Cgroup 技术对 Container 的资源进行控制，Cgroup 有多个子系统组成，每个子系统实现不同的功能，如下所示：

ID	Sub_System	Description
1	blkio	设置块设备设定输入/输出限制（如物理设备）。
2	cpu	使用调度程序提供对 CPU 的 cgroup 任务访问(控制 CPU 的利用率)。
3	cpuacct	自动生成 cgroup 中任务所使用的 CPU 报告。
4	cpuset	为 cgroup 中的任务分配独立 CPU 和内存节点。
5	devices	允许或者拒绝 cgroup 中的任务访问设备。
6	freezer	挂起或者恢复 cgroup 中的任务。
7	memory	设置每个 cgroup 的内存限制以及产生内存资源报告
8	net_cls	标记每个网络包以供 cgroup 方便使用

注：
关于 Cgroup 的详细使用方法请参考：https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/pdf/Resource_Management_Guide/Red_Hat_Enterprise_Linux-6-Resource_Management_Guide-en-US.pdf

3.3.3 SELinux & AppArmor

1) SELinux（Security-Enhance Linux）是 Linux 内核针对 MAC（mandatory access control）机制、MLS（multi-level security）、MCS（multi-category security）实现的一个机制。

2) AppArmor 类似于 SELinux，是一个基于 MAC 机制（name-based mandatory access controls）实现的安全模块，能够针对不同的可执行程序设置不同的访问控制权限。

3) 仅通过 Cgroup 和 Namespaces 无法保证 Container 中的 root 进程对 Container 外部的进程进行“干涉”，这时候就需要 SELinux 机制或 AppArmor 机制来对 Container 进行安全隔离。

4) 采用 SELinux 机制时，Container 在被创建的时候，会根据 SELinux policy 自动为 Container 分配一个 SELinux Context。

注：
a) 关于 SELinux 的详细信息，可参考：
https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/SELinux_Users_and_Administrators_Guide/index.html

b) 关于 AppArmor 的详细信息，可参考：
http://wiki.apparmor.net/index.php/Main_Page#Kernel

3.3.4 AUFS

Docker 使用 AUFS（Another Union FS）机制来实现 image 文件的管理是 Docker 实现轻量级虚拟化（lightweight virtual）的基础。

针对 AUFS 的详细说明如下：

1) Union FS 主要有两个用途：
   a) 可以不借助 LVM、RAID 机制将多个 Disk 挂载到同一目录下。
   b) 可以将一个 read-only 的目录和一个 writeable 的目录联合在一起。

2) AFUS 是一种 Union FS，其支持将不同目录挂载到同一个虚拟文件系统下，即支持为每一个成员目录设定不同的权限（read-only 或 read-write）。

3) Linux 系统在启动时，首先将 rootfs 设置为 read-only，然后进行一系列检查，最后再将其切换为 "read-write"供用户使用。

4) Docker Container 初始化时将 rootfs 以 read-only 方式加载并检查，然后利用 union mount 方式将一个 read-write 文件系统挂载在 read-only 的 rootfs 之上，并且允许再次将下层的文件系统设定为 read-only 并且向上叠加，这样一组 read-only 和一个 writeable 构成了一个 Container 的运行时环境。

5) AUFS 的特性是每一个对 read-only 层的修改都只会存在于上层的 writeable 层中。这样由于不存在竞争,多个 Container 可以共享 read-only 的 FS 层。

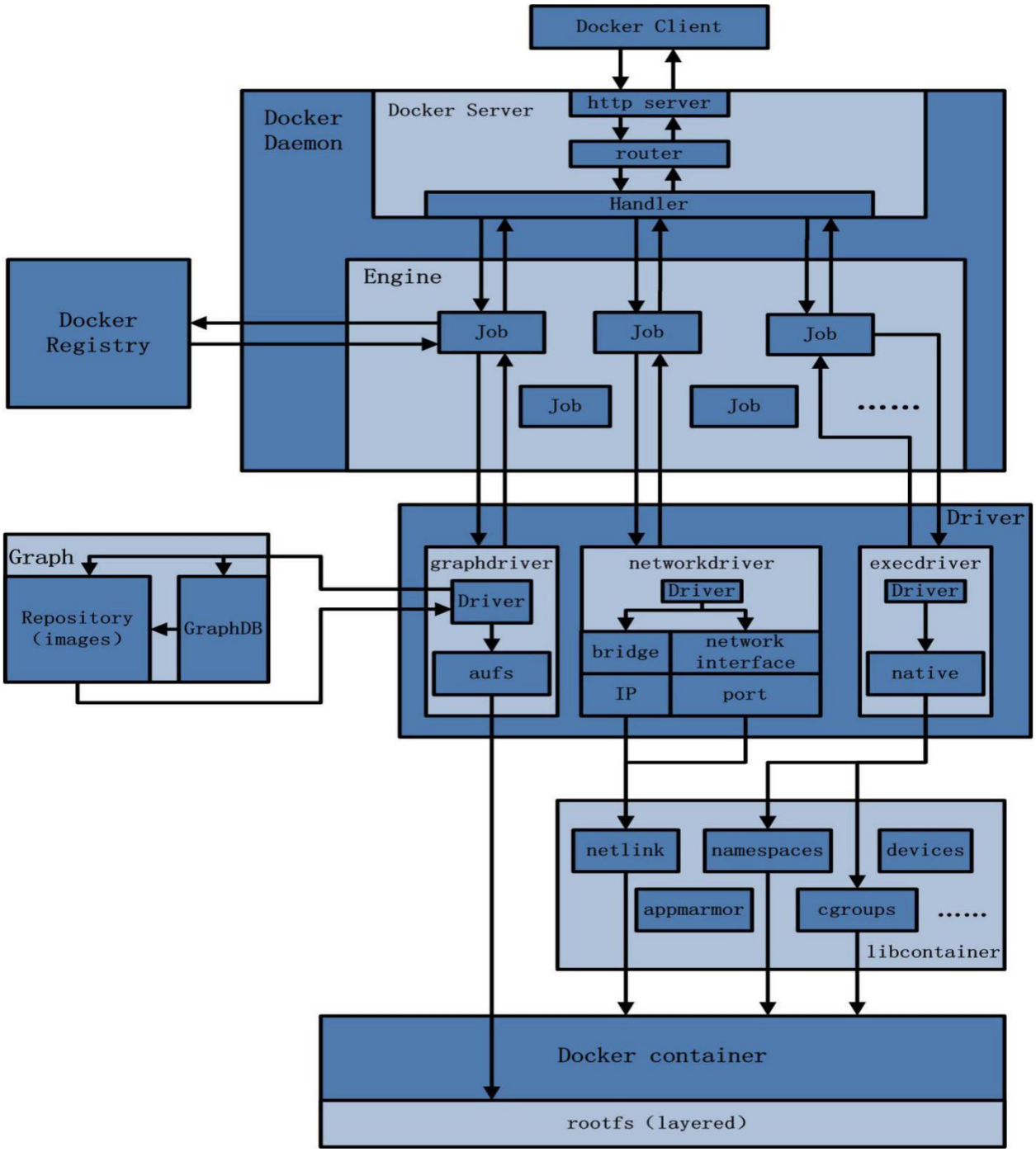
6) Docker 将 read-only 的 FS 层称作 “image”，对于 Container 来说整个 rootfs 都是 read-write 的，但事实上所有的修改都写入最上层的 writeable 层中，image 不保存用户状态，只用于模板、新建和复制使用。

7) 上层的 image 依赖下层的 image，因此 Docker 中把下层的 image 称作父 image，没有父 image 的 image 称作 base image。当想要从一个 image 启动一个 Container 时，Docker 会加载这个 image 和依赖的父 images 以及 base image，用户的进程运行在 writeable 的 layer 中。

4 Docker 架构

4.1 Docker Architecture

Docker 对用户来说是一个 C/S 模式的架构，其后端是一个非常松耦合的架构，模块各司其职并有机组合来支撑 Docker 的运行。Docker 架构如下图所示：



说明：

1) 从上图可以看出，用户通过使用 Docker Client 与 Docker Daemon 建立通信。

2) Docker Daemon 作为 Docker 架构中的主体部分，首先提供 Server 的功能使其可以接受 Docker Client 的请求，而后 Engine 执行 Docker 内部的一系列工作，其中每一项工作都是以一个 Job 的形式存在。

3) Job 的运行过程中，根据 Job 的不同执行不同的流程：

   a) 当需要容器镜像时，则从 Docker Registry 中下载镜像，并通过镜像管理驱动 graph driver 将下载的镜像以 Graph 的形式存储。

   b) 当需要为 Docker 创建网络环境时，通过网络管理驱动 network driver 创建并配置 Docker 容器网络环境。

   c) 当需要限制 Docker 容器运行资源或执行用户指令等操作时，则通过 exec driver 来完成。

4) libcontainer 是一项独立的容器管理包，network driver 以及 exec driver 都是通过其来实现对容器进行的具体操作。

5) 当执行完运行容器的命令后，Docker 容器就处于运行状态，该容器拥有独立的文件系统、安全的运行环境等。

4.2 Docker Module

Docker 架构主要由 7 个模块组成：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer 以及 Docker container。接下来，对各个模块进行详细的介绍。

4.2.1 Client

1) Docker Client 是 Docker 架构中用户用来和 Docker Daemon 建立通信的客户端。用户使用的可执行文件为 docker，通过 docker 命令行工具可以发起众多管理 Container 的请求。

2) Docker Client 可以通过三种方式和 Docker Daemon 建立通信，如下：
   a) tcp://host:port
   b) unix://path_to_socket
   c) fd://socketfd

3) Docker Client 与 Docker Daemon 建立连接并传输请求的时候，Docker Client 可以通过设置命令行 flag 参数的形式设置安全传输层协议(TLS)的有关参数，来保证传输的安全性。

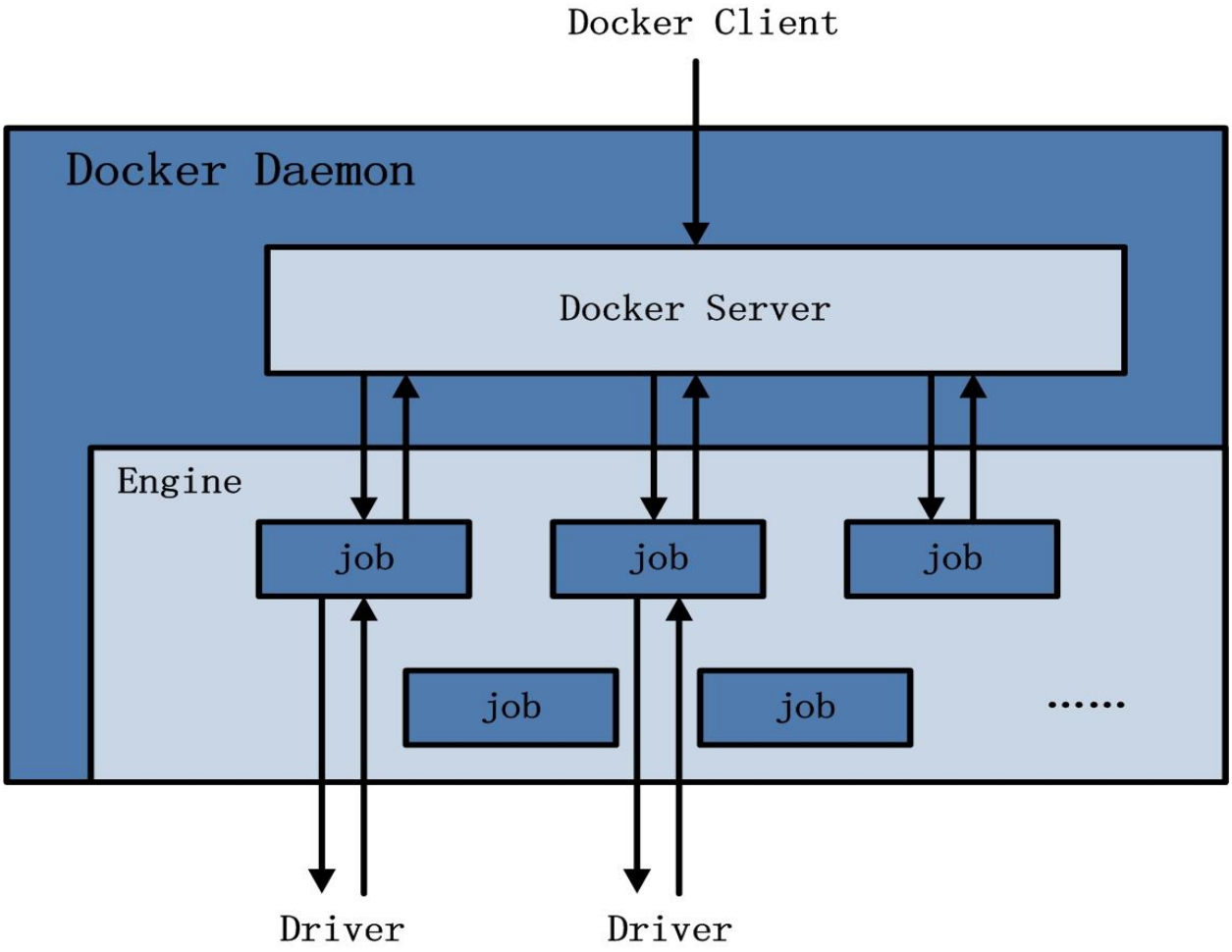
4) Docker Client 发送容器管理请求后，由 Docker Daemon 接受并处理请求，当 Docker Client 接收到返回的请求相应并简单处理后，Docker Client 一次完整的生命周期就结束了。当需要继续发送容器管理请求时，用户必须再次通过 docker 可执行文件创建 Docker Client。

4.2.2 Daemon

1) Docker Daemon 启动所使用的可执行文件也为 docker，与 Docker Client 启动所使用的可执行文件 docker 相同。在 docker 命令执行时，通过传入的参数来判别 Docker Daemon 与 Docker Client。

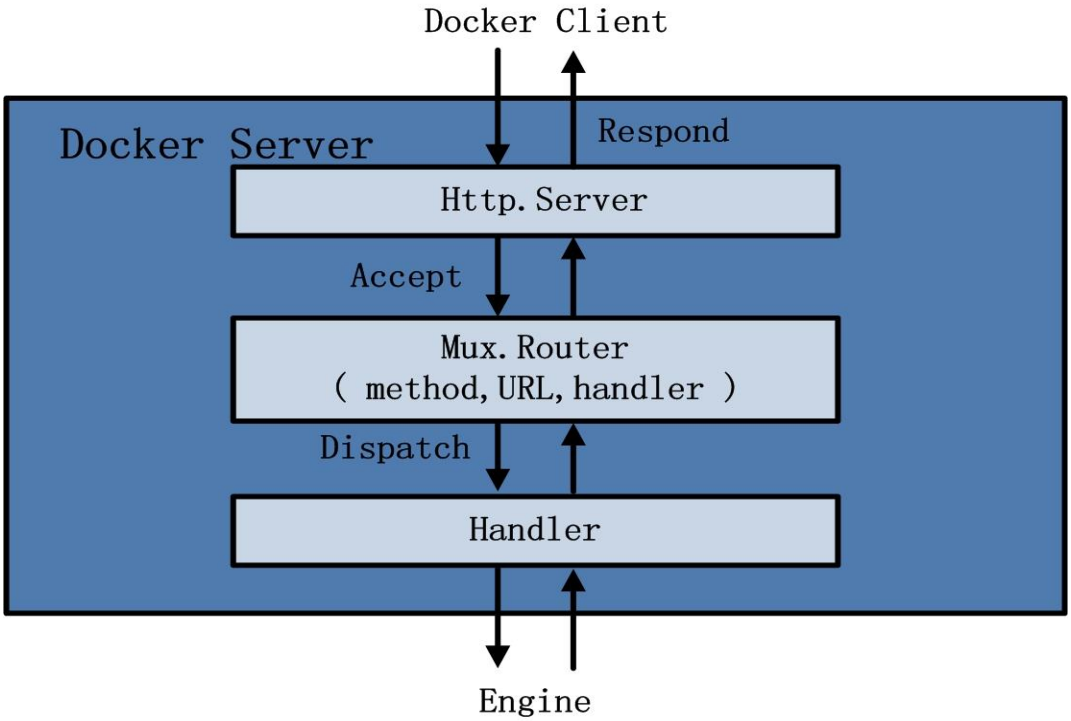
2) Docker Daemon 是 Docker 架构中一个常驻在后台的系统进程，其功能是接受并处理 Docker Client 发送的请求：
   a) Docker Daemon 在后台启动一个 Server，Server 负责接受 Docker Client 发送的请求。
   b) Server 接受请求后，通过路由与分发调度找到相应的 Handler 来执行请求。

3) Docker Daemon 的架构大致可以分为 2 部分：Docker Server 和 Engine，如下图所示：



4.2.2.1 Docker Server

Docker Server 在 Docker 架构中是专门服务于 Docker Client 的 server，其功能是接受并调度分发 Docker Client 发送的请求。Docker Server 的架构图如下所示：



说明：

1) Docker Server 在启动过程中，会创建一个 Mux.Router 来提供请求的路由功能。该 Mux.Router 中的每一个路由项由 Method(HTTP 请求方法：PUT、POST、GET 或 DELETE）、URL、Handler 三部分组成。

2) Docker Server 接受 Docker Client 的访问请求后，会创建一个 goroutine 来服务该请求。在 goroutine 中，首先读取请求内容做解析工作，接着找到相应的路由项，随后调用相应的 Handler 来处理该请求，最后 Handler 处理完请求之后回复该请求。

3) 需要注意的是：Docker Server 只是众多 Job 中的一个，但是为了强调 Docker Server 的重要性以及为其他 Job 服务的重要特性，将其单独抽离出来分析，理解为 Docker Server。

4.2.2.2 Engine

1) Engine 是 Docker 架构中的运行引擎，同时也 Docker 运行的核心模块。它扮演 Docker Container 存储仓库的角色，并且通过执行 job 的方式来操纵管理这些容器。

2) 在 Engine 数据结构的设计与实现过程中，有一个 handler 对象，该 handler 对象存储的都是关于众多特定 Job 的 handler 处理访问。如：Engine 的 handler 对象中有一项为：{"create": daemon.ContainerCreate,}，则当名为"create"的 Job 在运行时，执行的是 daemon.ContainerCreate 的 handler。

3) Job 是 Engine 内部最基本的工作执行单元，Docker 做的每一项工作，都可以抽象为一个 Job。如：在 Container 内部运行一个进程，这就是一个 Job。

4.2.3 Registry

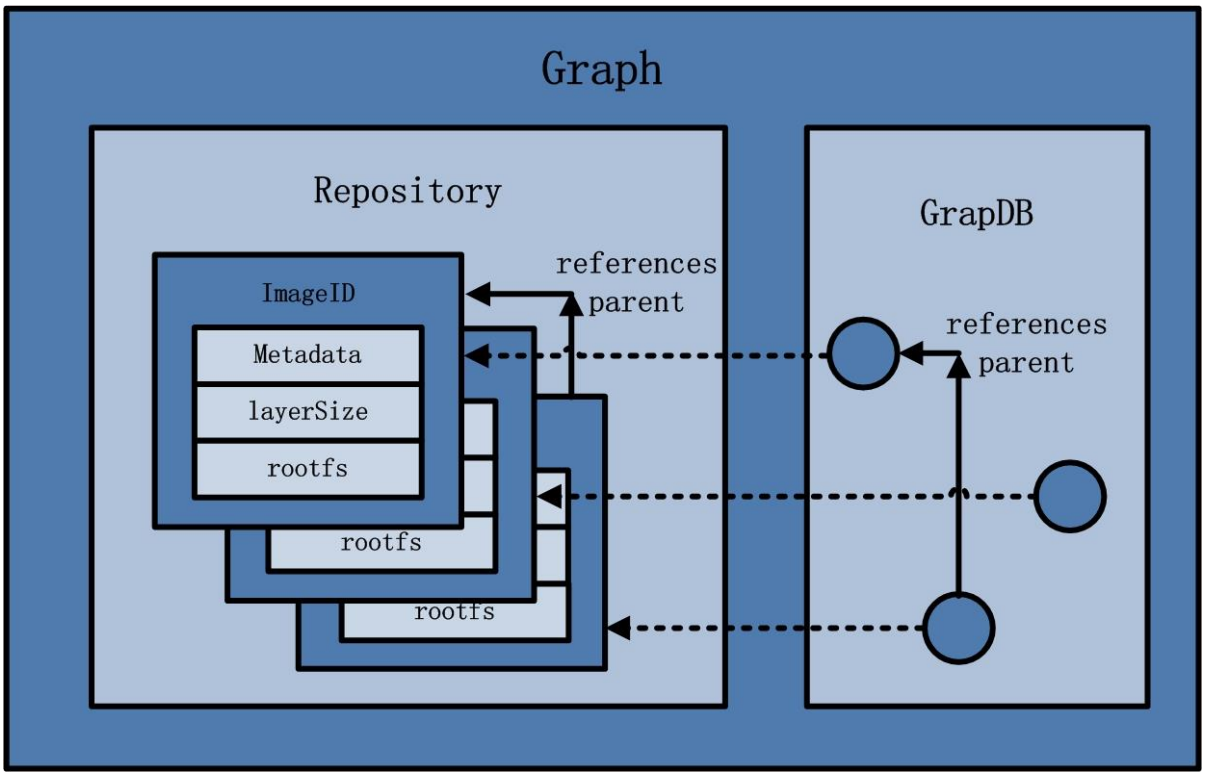
1) Docker Registry 是一个存储容器镜像的仓库。容器镜像是在容器被创建时，被加载用来初始化容器的文件架构与目录。

2) Docker 在运行过程中，Docker Daemon 会与 Docker Registry 进行通信，并实现镜像搜索、下载、上传三个功能。

3) Docker 可以通过互联网访问共有的 Docker Registry(即 Docker Hub)来获取 Docker 的容器镜像文件。同时 Docker 也允许用户构建本地私有的 Docker Registry，这样可以保证容器镜像的获取在内网完成。

4.2.4 Graph

Graph 在 Docker 架构中扮演容器镜像的保管者以及容器镜像之间关系的记录者：一方面，Graph 存储着本地具有版本信息的文件系统镜像，另一方面 Graph 通过 GraphDB 记录着所有文件系统镜像之间的关系。Graph 的架构图如下所示：



说明：

1) GraphDB 是一个构建在 SQLite 之上的小型图数据库，实现了节点的命名以及节点之间关系的记录。

2) 在 Graph 的本地目录中，关于每一个容器镜像具体存储的信息有：该容器镜像的元数据，容器镜像的大小信息，以及该容器镜像所代表的具体 rootfs。

4.2.5 Driver

1) Docker 通过 Driver 可以实现对 Docker 容器执行环境的定制。

2) Docker 为了将容器的管理从 Docker Daemon 内部业务逻辑中区分开来，设计了 Driver 层驱动来接管所有这部分请求。

3) Docker Driver 在实现中分为三类驱动：graphdriver、networkdriver 和 execdriver。

4.2.5.1 graph driver

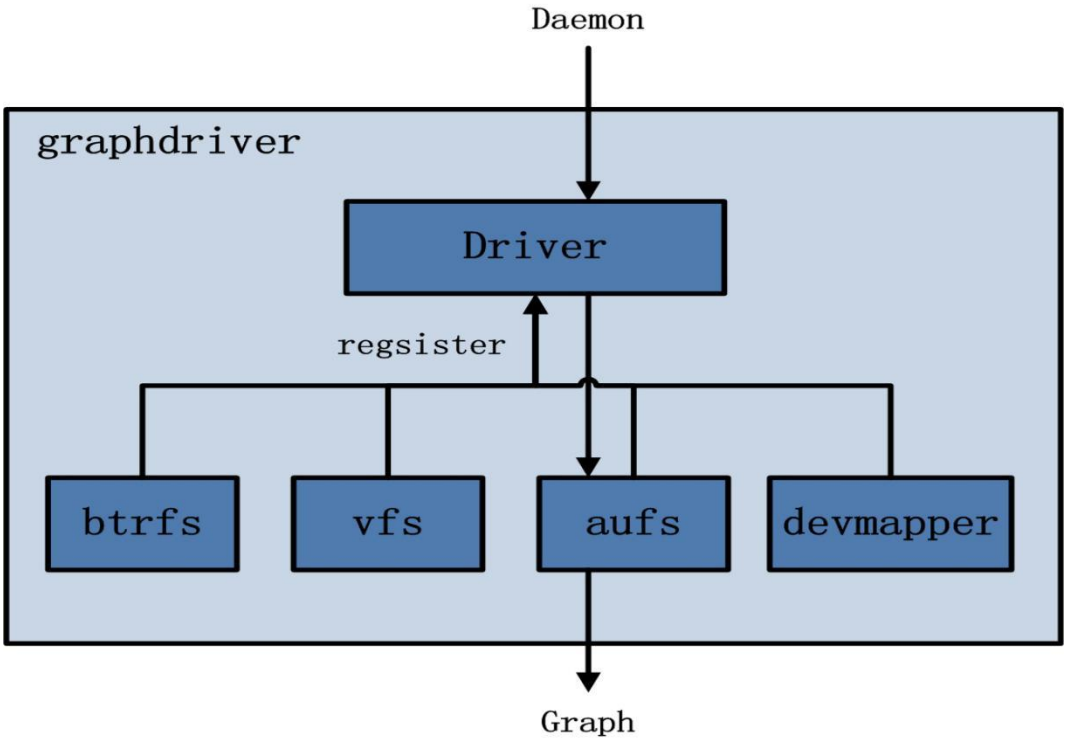
1) graph driver 主要用于完成容器镜像的管理，包括存储与获取。

2) graph driver 在用户需要下载指定的容器镜像时，会将容器镜像存储在本地的指定目录；同时 graph driver 在用户需要使用指定的容器镜像来创建容器的 rootfs 时，会从本地镜像存储目录中获取指定的容器镜像。

3) graph driver 在初始化之前，有 4 种文件系统或类文件系统在其内部注册，它们分别是 btrfs、vfs、aufs 和 devmapper。

4) Docker 在初始化之时，通过获取系统环境变量“DOCKER_DRIVER”来提取所使用 driver 的指定类型。之后所有的 graph 操作，都使用该 driver 来执行。

graph driver 的架构图如下所示：

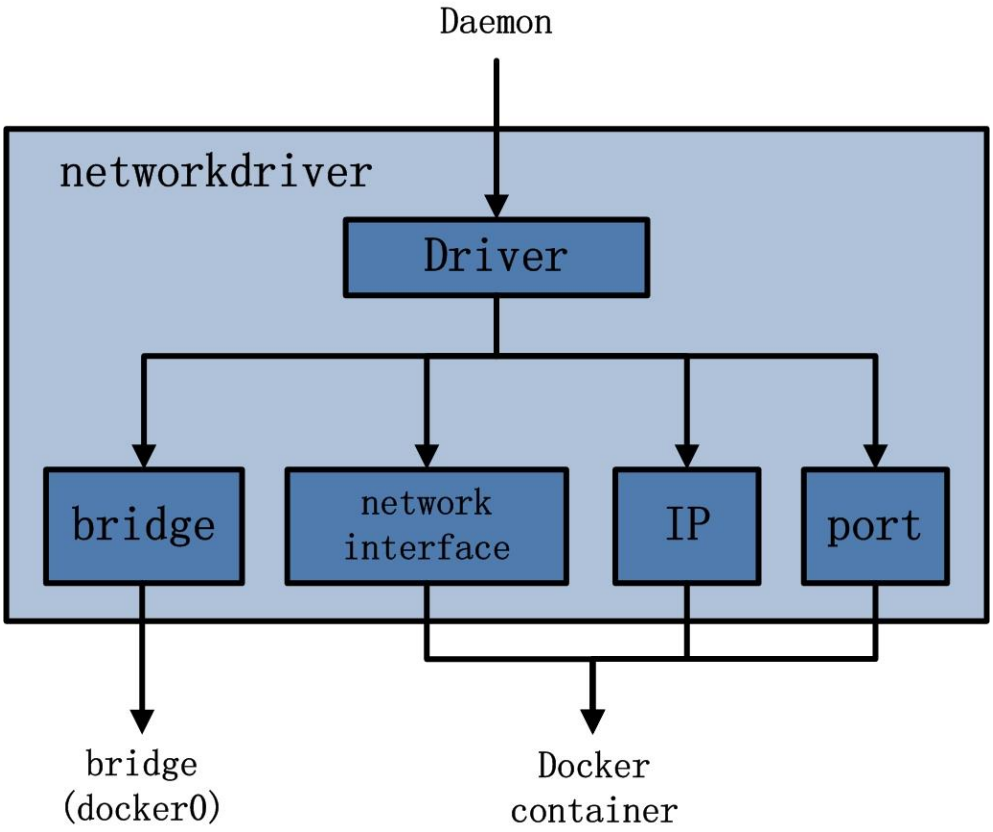


4.2.5.2 network driver

network driver 的用途是完成 Docker 容器网络环境的配置，具体包括：

1) Docker 启动时为 Docker 环境创建网桥。
2) Docker 容器创建时为其创建专属虚拟网卡设备。
3) Docker 容器 IP 与端口的分配，以及设置与宿主机做端口映射、容器防火墙策略等。

network driver 的架构图如下所示：



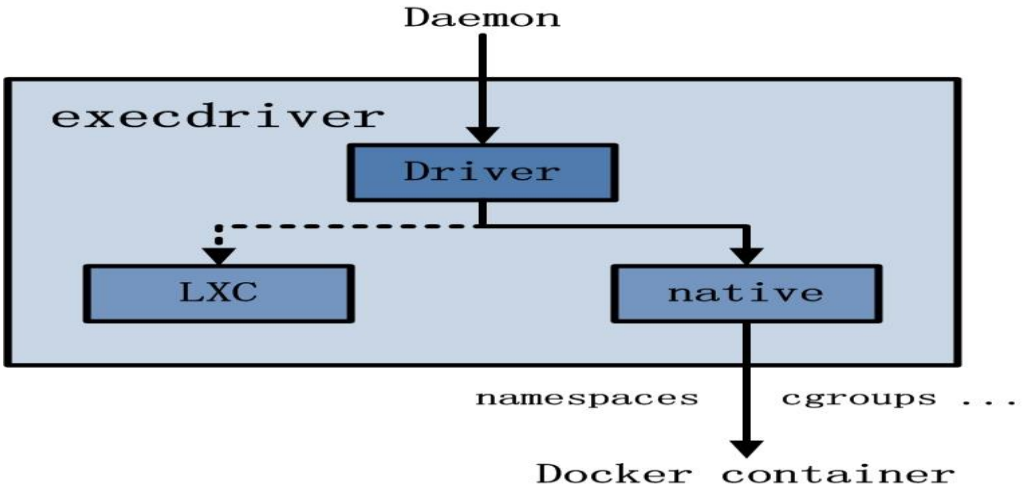
4.2.5.3 exec driver

1) exec driver 作为 Docker 容器的执行驱动，负责创建容器运行命名空间、容器资源使用的统计与限制，容器内部进程的真正运行等。

2) exec driver 之前使用 LXC 驱动调用 LXC 的接口来操纵容器的配置以及生命周期。

3) exec driver 在 1.2 版之后默认使用 native 驱动，从而不再依赖于 LXC。具体体现在 Daemon 启动过程中加载的 ExecDriverflag 参数，该参数在配置文件已经被设为"native"。

exec driver 架构图如下所示：



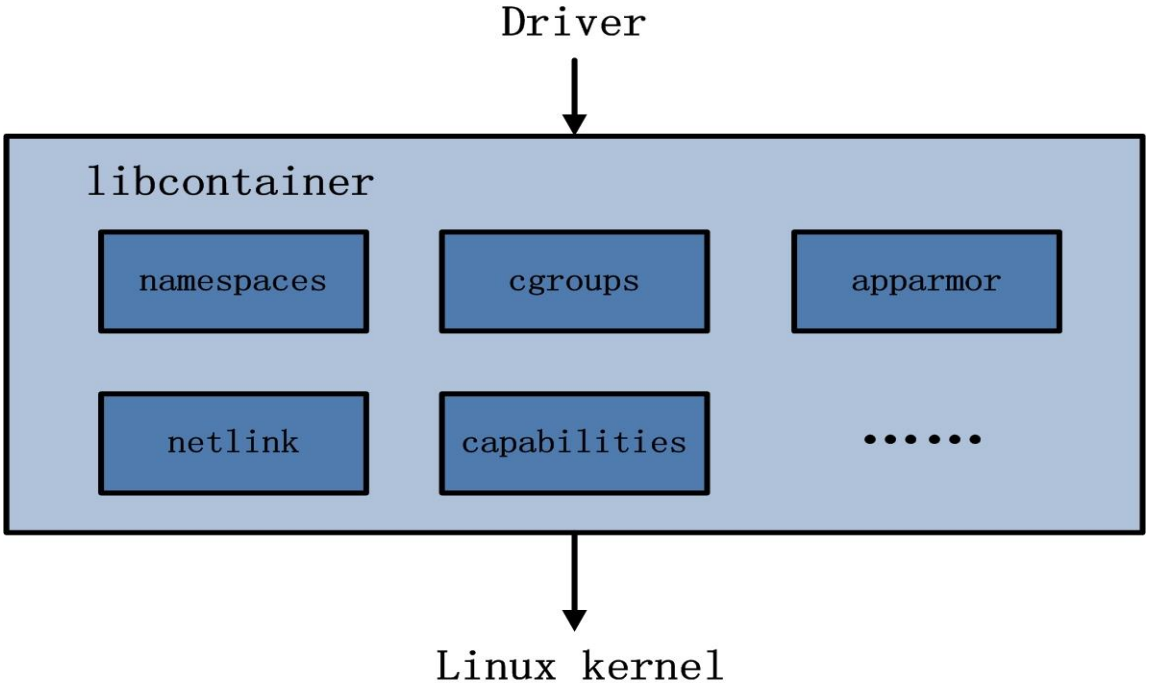
4.2.6 Libcontainer

1) libcontainer 是 Docker 架构中一个使用跨平台的语言 Go 实现的库，设计初衷是希望该库可以不依靠任何依赖，能够直接访问内核中与容器相关的 API。

2) libcontainer 屏蔽了 Docker 上层对容器的直接管理，其提供了一整套标准的接口来满足上层对容器管理的需求。

3) Libcontainer 使得 Docker 可以直接调用内核中与容器相关的 API，从而最终操纵容器的 namespaces、cgroup、apparmor、网络设备以及防火墙规则等。

libcontainer 架构图如下所示：

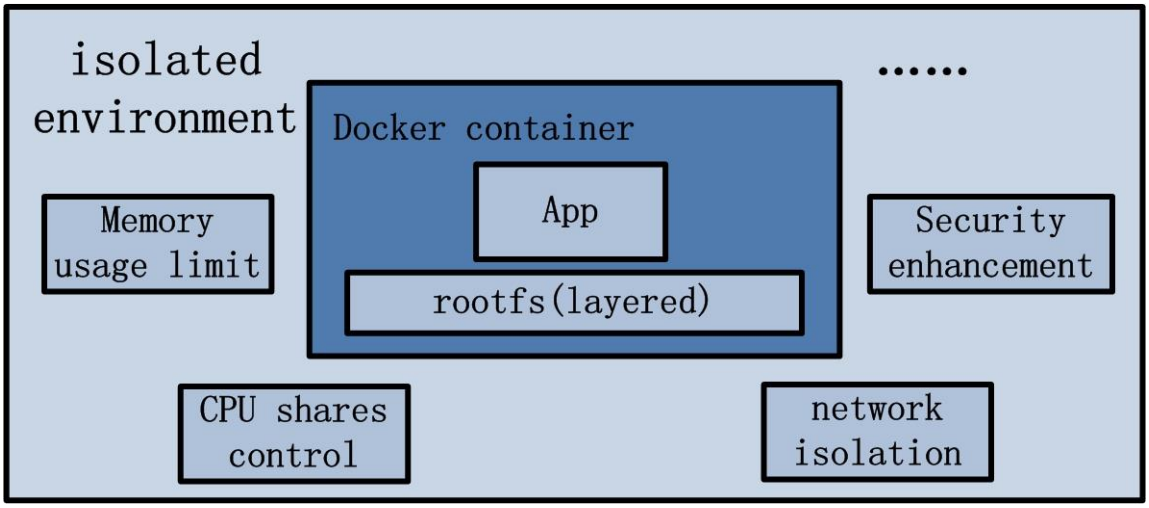


4.2.7 Container

1) Docker Container 是 Docker 架构中服务交付的最终体现形式。

2) Docker 可按照用户的需求与指令，订制相应的 Docker 容器，如：
   a. 用户通过指定容器镜像，使得 Docker Container 可以自定义 rootfs 等文件系统；
   b. 用户通过指定计算资源的配额，使得 Docker Container 使用指定的计算资源；
   c. 用户通过配置网络及其安全策略，使得 Docker Container 拥有独立且安全的网络环境；
   d. 用户通过指定运行的命令，使得 Docker Container 执行指定的工作。

Container 的架构图如下所示：



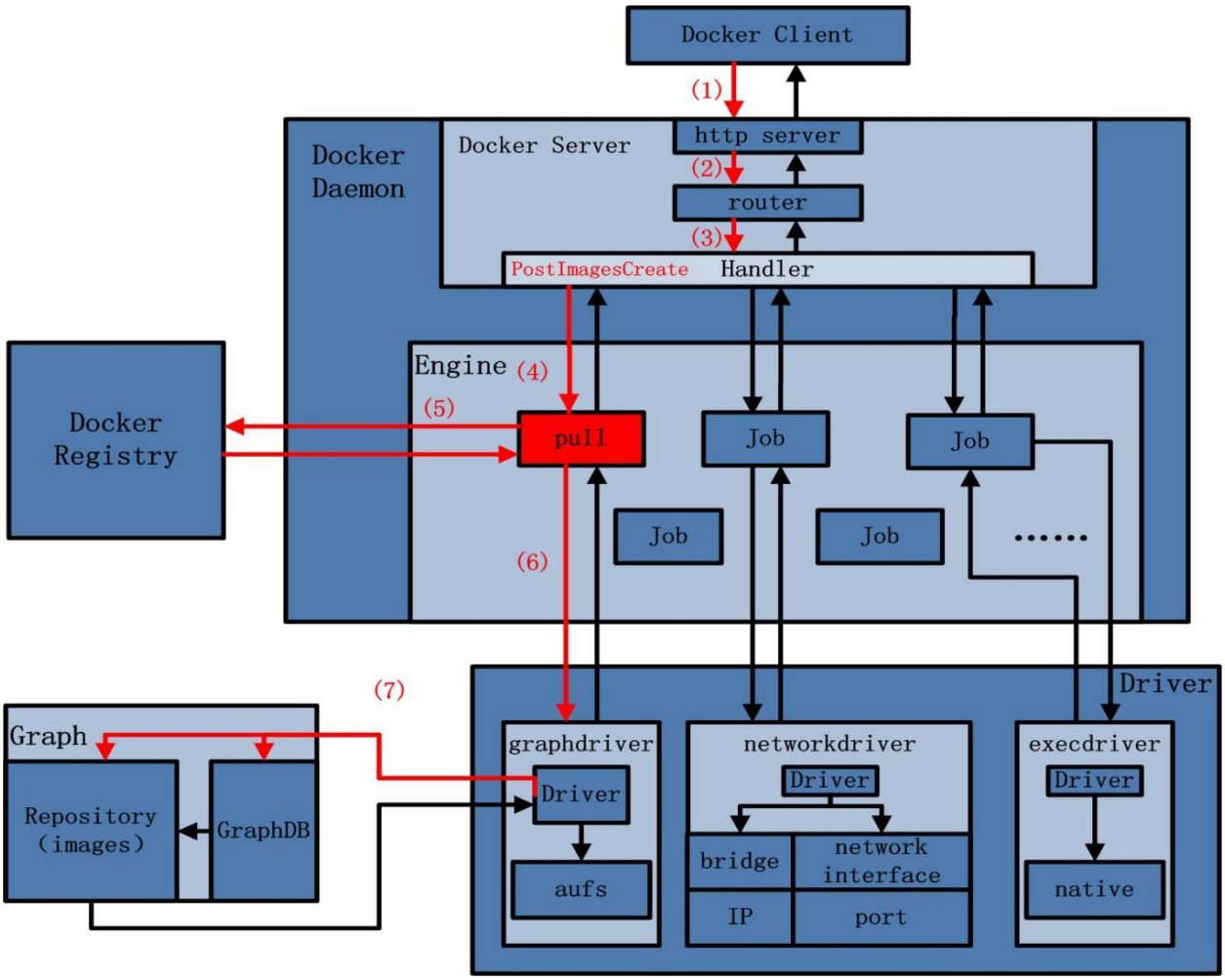
4.3 Docker Flow

上节独立的介绍了 Docker 架构中各个模块，本节讲通过两个 docker 命令：docker pull 和 docker run 来介绍各个模块如何有机组合、协同合作来支撑着 Docker 的运行。

4.3.1 docker pull

docker pull 命令将会从 Docker Registry 中下载指定的容器镜像，并存储在本地的 graph 中，以备后续创建 Docker 容器时使用。

docker pull 命令的执行流程图如下所示：



步骤：（图中标记的红色箭头表示 docker pull 命令在发起后所做的一系列操作）

1) Docker Client 接受 docker pull 命令，解析完请求以及收集完请求参数之后，发送一个 HTTP 请求给 Docker Server。HTTP 请求方法为 POST，请求 URL 为"/images/create? "+"xxx"。

2) Docker Server 接受以上 HTTP 请求，并交给 mux.Router，mux.Router 通过 URL 以及请求方法来确定执行该请求的具体 handler。

3) mux.Router 将请求路由分发至相应的 handler，具体为 PostImagesCreate。

4) 在 PostImageCreate 这个 handler 之中，一个名为"pull"的 Job 被创建，并开始执行。

5) 名为"pull"的 Job 在执行过程中，执行 pull Repository 操作，即从 Docker Registry 中下载相应的一个或者多个 image。

6) 名为"pull"的 job 将下载的 image 交给 graph driver。

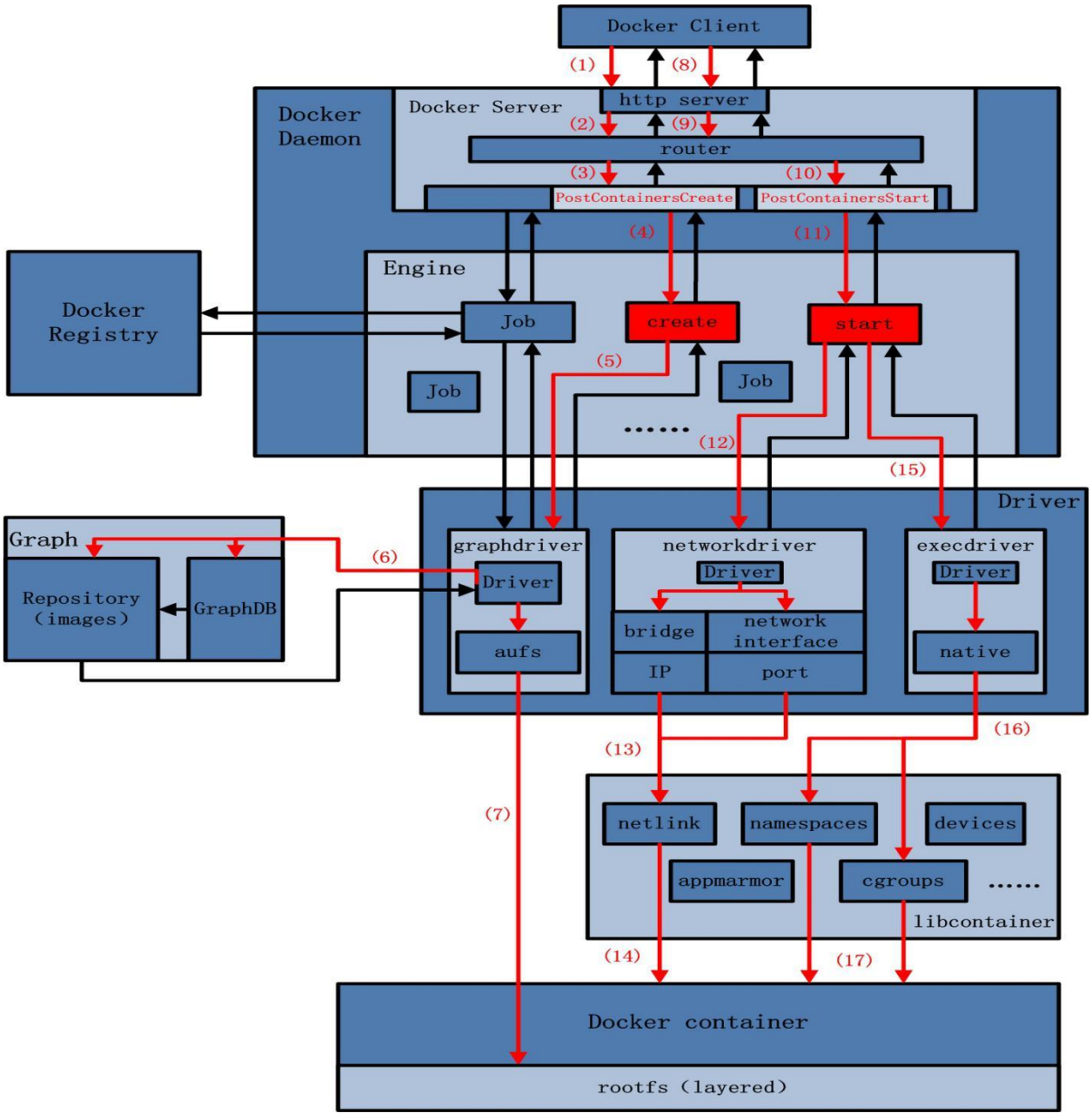
7) graph driver 负责将 image 进行存储，一方面创建 graph 对象，另一方面在 GraphDB 中记录 image 之间的关系。

4.3.2 docker run

1) docker run 命令将在一个全新的 Docker Container 内部运行一条指令。Docker 在执行这条指令的时候，所做工作可以分为两部分：
   a. 创建 Docker 容器所需的 rootfs 以及网络等运行环境。
   b. 运行用户所指定的指令。

2) docker run 在整个执行流程中，Docker Client 给 Docker Server 发送了两次 HTTP 请求。第二次请求的发起取决于第一次请求的返回状态。

Docker run 命令的执行流程图如下所示：



步骤：（图中标记的红色箭头表示 docker pull 命令在发起后所做的一系列操作）

1) Docker Client 接受 docker run 命令，解析完请求以及收集完请求参数之后，发送一个 HTTP 请求给 Docker Server。HTTP 请求方法为 POST，请求 URL 为"/containers/create? "+"xxx"。

2) Docker Server 接受以上 HTTP 请求，并交给 mux.Router，mux.Router 通过 URL 以及请求方法来确定执行该请求的具体 handler。

3) mux.Router 将请求路由分发至相应的 handler，具体为 PostContainersCreate。

4) 在 PostImageCreate 这个 handler 之中，一个名为"create"的 Job 被创建，并开始运行。

5) 名为"create"的 Job 在运行过程中会执行 Container.Create 操作，该操作需要获取容器镜像来为 Docker Container 创建 rootfs，即调用 graph driver。

6) graph driver 从 Graph 中获取创建 Docker Container rootfs 所需要的所有的镜像。

7) graph driver 将 rootfs 所有镜像加载安装至 Docker Container 指定的文件目录下。

8) 若以上操作全部正常执行，没有返回错误或异常，则 Docker Client 收到 Docker Server 返回状态之后发起第二次 HTTP 请求。请求方法为"POST"，请求 URL 为 "/containers/"+container_ID+"/start"。

9) Docker Server 接受以上 HTTP 请求，并交给 mux.Router，mux.Router 通过 URL 以及请求方法来确定执行该请求的具体 handler。

10) mux.Router 将请求路由分发至相应的 handler，具体为 PostContainersStart。

11) 在 PostContainersStart 这个 handler 之中，名为"start"的 Job 被创建，并开始执行。

12) 名为"start"的 Job 执行完初步配置工作后，调用 network driver 配置与创建网络环境。

13) network driver 需要指定的 Docker Container 创建网络接口设备，并为其分配 IP、port 以及设置防火墙规则，相应的操作转交至 libcontainer 中的 netlink 包来完成。

14) netlink 完成 Docker Container 的网络环境配置与创建。

15) 返回至名为"start"的 Job，执行完一些辅助性操作后，Job 调用 exec driver 开始执行用户指令。

16) exec driver 将初始化 Docker Container 内部的运行环境,如:命名空间、资源控制与隔离以及用户命令的执行，相应的操作转交至 libcontainer 来完成。

17) Libcontainer 将完成 Docker Container 内部的运行环境初始化，并最终执行用户要求的命令。

5 Docker 安装

Docker 的安装在官网上针对不同的操作系统分别进行了描述，这里以 RHEL7 为例来介绍 Docker 的几种安装方法，如下所示：

1) 官网推荐安装

首先使用命令“subscription-manager repos --enable=rhel-7-server-extras-rpms”配置好 yum 源，然后使用命令“yum install docker”即可安装 Docker。

注：
a) 使用 Red Hat 的 yum 源需注册订阅（付费）。
b) 同样，Red Hat 没有在 Docker Hub 上提供镜像，想下载 Red Hat 发布的官方镜像需注册订阅(付费)才能在其官网上下载。

详细说明地址如下：
https://docs.docker.com/installation/rhel/

2) 二进制安装（推荐）

通过二进制安装是最简单的方法，不过该方法要求系统的 Kernel 版本不能低于 3.8、iptables 版本不能低于 1.4 等条件。

具体要求可查看地址：https://docs.docker.com/installation/binaries/

二进制文件下载地址如下：
https://get.docker.com/builds/Linux/x86_64/docker-latest.tgz

3) RPM 安装

可直接下载相应版本的 RPM 包进行安装。需要注意的是，安装 RPM 包时因系统的不同可能会因依赖原因导致安装失败。

RPM 包下载地址如下：
http://rpmfind.net/linux/rpm2html/search.php?query=docker&submit=Search+...&system=&arch=

4) 源码安装

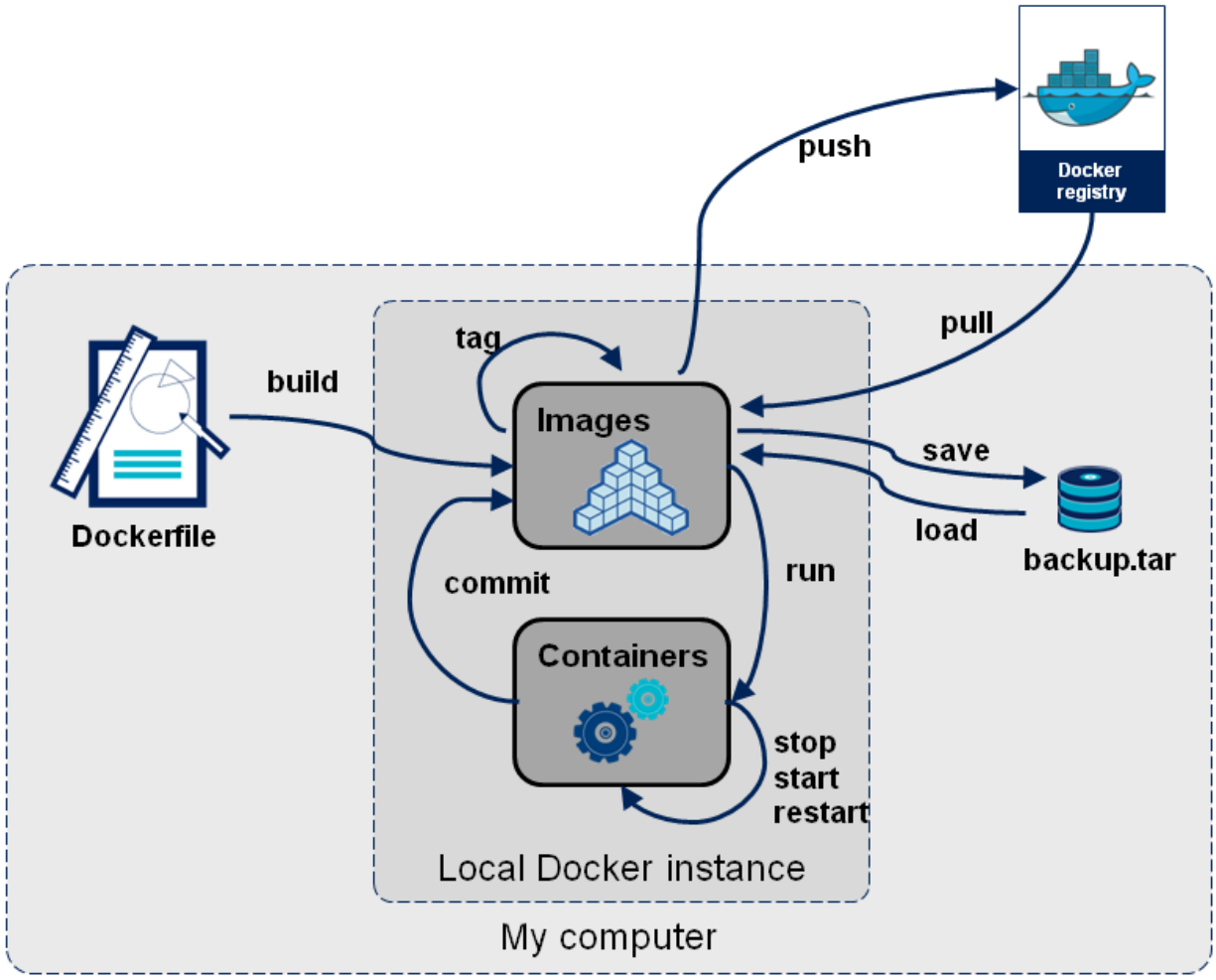
由于 Docker 是使用 Go 语言来写的，在 Linux 下使用源码进行安装较为复杂，不推荐使用。

注：Docker 官网上只提供了 64 位的 docker 安装，如果想要安装 32 位的 Docker 则只能使用源码编译安装。

6 Docker 管理

1) Docker 对用户来说是一个 C/S 模式的架构，其使用同一文件来运行 Client 端和 Server 端。Docker 在命令执行时，通过传入的参数来判别 Client 端和 Server 端。

2) Docker 可以把任务模块化，然后做成特定的 image 文件来运行需要的应用程序。针对每个 image 可操作的场景类似如下：



3) 接下来，针对 Docker 的命令使用、私有仓库创建、Docker 管理工具进行分别介绍。

6.1 Command

Docker 提供了丰富的参数选项和命令，查看其参数选项和命令列表可以直接运行命令“docker”即可，如下所示：

```
[root@localhost renyl]# docker
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.

Options:
  --api-enable-cors=false               Enable CORS headers in the remote API
  -b, --bridge=""                       Attach containers to a pre-existing network bridge
                                          use 'none' to disable container networking
  --bip=""                              Use this CIDR notation address for the network bridge's IP,
not compatible with -b
  -D, --debug=false                     Enable debug mode
  -d, --daemon=false                    Enable daemon mode
  --dns=[]                              Force Docker to use specific DNS servers
  --dns-search=[]                       Force Docker to use specific DNS search domains
  -e, --exec-driver="native"            Force the Docker runtime to use a specific exec driver
  --fixed-cidr=""                       IPv4 subnet for fixed IPs (ex: 10.20.0.0/16)
                                          this subnet must be nested in the bridge subnet (which is
defined by -b or --bip)
  -G, --group="docker"                  Group to assign the unix socket specified by -H when running
in daemon mode
                                          use '' (the empty string) to disable setting of a group
  -g, --graph="/var/lib/docker"         Path to use as the root of the Docker runtime
  -H, --host=[]                         The socket(s) to bind to in daemon mode or connect to in
client mode, specified using one or more tcp://host:port, unix:///path/to/socket, fd://* or
fd://socketfd.
  --icc=true                            Allow unrestricted inter-container and Docker daemon host
communication
  --insecure-registry=[]                Enable insecure communication with specified registries (no
certificate verification for HTTPS and enable HTTP fallback) (e.g., localhost:5000 or 10.20.0.0/16)
  --ip=0.0.0.0                          Default IP address to use when binding container ports
  --ip-forward=true                     Enable net.ipv4.ip_forward
  --ip-masq=true                        Enable IP masquerading for bridge's IP range
  --iptables=true                       Enable Docker's addition of iptables rules
  -l, --log-level="info"                Set the logging level
  --label=[]                            Set key=value labels to the daemon (displayed in `docker
info`)
  --mtu=0                               Set the containers network MTU
                                          if no value is provided: default to the default route MTU
or 1500 if no default route is available
  -p, --pidfile="/var/run/docker.pid"   Path to use for daemon PID file
  --registry-mirror=[]                  Specify a preferred Docker registry mirror
  -s, --storage-driver=""               Force the Docker runtime to use a specific storage driver
  --selinux-enabled=false               Enable selinux support. SELinux does not presently support
the BTRFS storage driver
  --storage-opt=[]                      Set storage driver options
  --tls=false                           Use TLS; implied by --tlsverify flag
  --tlscacert="/root/.docker/ca.pem"    Trust only remotes providing a certificate signed by the CA
given here
```

```
  --tlscert="/root/.docker/cert.pem"      Path to TLS certificate file
  --tlskey="/root/.docker/key.pem"        Path to TLS key file
  --tlsverify=false                       Use TLS and verify the remote (daemon: verify client,
client: verify daemon)
  -v, --version=false                     Print version information and quit

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders from a container's filesystem to the host path
    create    Create a new container
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    exec      Run a command in a running container
    export    Stream the contents of a container as a tar archive
    history   Show the history of an image
    images    List images
    import    Create a new filesystem image from the contents of a tarball
    info      Display system-wide information
    inspect   Return low-level information on a container
    kill      Kill a running container
    load      Load an image from a tar archive
    login     Register or log in to a Docker registry server
    logout    Log out from a Docker registry server
    logs      Fetch the logs of a container
    port      Lookup the public-facing port that is NAT-ed to PRIVATE_PORT
    pause     Pause all processes within a container
    ps        List containers
    pull      Pull an image or a repository from a Docker registry server
    push      Push an image or a repository to a Docker registry server
    restart   Restart a running container
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save an image to a tar archive
    search    Search for an image on the Docker Hub
    start     Start a stopped container
    stop      Stop a running container
    tag       Tag an image into a repository
    top       Lookup the running processes of a container
    unpause   Unpause a paused container
    version   Show the Docker version information
    wait      Block until a container stops, then print its exit code

Run 'docker COMMAND --help' for more information on a command.
[root@localhost renyl]#
```

从上述的输出可以看到，Docker 提供了多个参数选项和子命令（每个命令又会提供多个参数选项）。其中参数选项被用于 Docker Server 启动时来进行相关设置，Docker Client 通过使用子命令来与 Docker Server 进行通信。

参数选项详细信息如下：

ID	Parameter	Description
1	--api-enable-cors=false	开放远程 API 调用的 CORS 头信息。 这个接口开关对想进行二次开发的上层应用提供了支持。
2	-b, --bridge=""	挂载已存在的网桥设备到 Docker 容器里。 注意，使用 none 可以停用容器里的网络。
3	--bip=""	使用 CIDR 地址来设定网络桥的 IP。 注意，此参数和 -b 不能一起使用。
4	-D, --debug=false	开启 Debug 模式。 例如：docker -d -D
5	-d, --daemon=false	开启 Daemon 模式。 例如：docker -d
6	--dns=[]	强制容器使用 DNS 服务器。 例如： docker -d --dns 8.8.8.8
7	--dns-search=[]	强制容器使用指定的 DNS 搜索域名。 例如： docker -d --dns-search example.com
8	-e, --exec-driver="native"	强制容器使用指定的运行时驱动。 例如：docker -d -e lxc
9	--fixed-cidr=""	针对 ip 子网设置固定的 ip 地址。
10	-G, --group="docker"	赋予指定的 Group 到相应的 unix socket 上。 注意，此参数--group 赋予空字符串时，将去除组信息。
11	-g, --graph="/var/lib/docker"	配置 Docker 运行时的根目录

12	-H, --host=[]	在后台模式下指定 socket 绑定，可以绑定一个或多个 tcp://host:port, unix:///path/to/socket, fd://* 或 fd://socketfd。
		例如：
		$ docker -H tcp://0.0.0.0:2375 ps 或者
		$ export DOCKER_HOST="tcp://0.0.0.0:2375"
		$ docker ps

13	--icc=true	启用内联容器的通信。
14	--insecure-registry=[]	指定的 registry 为不安全的通信。
15	--ip="0.0.0.0"	绑定容器端口时，默认使用的 IP 地址。
16	--ip-forward=true	启动容器的 net.ipv4.ip_forward。
17	--ip-masq=true	针对网桥的 IP 范围启用伪装 IP。
18	--iptables=true	启动 Docker 容器自定义的 iptable 规则。
19	-l, --log-level="info"	设置 log 等级。
20	--label=[]	设置 key=value 标志，使用“docker info”即可查看到。
21	--mtu=0	设置容器网络的 MTU 值。如果没这个参数，默认用 route MTU，如果没有默认 route，就设置成常量值 1500。
22	-p, --pidfile="/var/run/docker.pid"	后台进程 PID 文件路径。
23	--registry-mirror=[]	指定一个 preferred Docker 库镜像。
24	-s, --storage-driver=""	强制容器运行时使用指定的存储驱动。 例如：docker -d -s devicemapper
25	--selinux-enabled=false	启用 selinux 支持。
26	--storage-opt=[]	配置存储驱动的参数。
27	--tls=false	启动 TLS 认证开关。
28	--tlscacert="/Users/dxiao/.docker/ca.pem"	通过 CA 认证过的 certificate 文件路径。
29	--tlscert="/Users/dxiao/.docker/cert.pem"	TLS 的 certificate 文件路径。
30	--tlskey="/Users/dxiao/.docker/key.pem"	TLS 的 key 文件路径。
31	--tlsverify=false	使用 TLS 并做后台进程与客户端通讯的验证。
32	-v, --version=false	显示版本。

各命令详细信息如下：（根据功能进行划分）

ID	Function	Command	Description
1	环境信息相关	info	显示 system-wide 信息。
2		version	显示 docker 版本相关信息。
3	系统运行相关	attach	连接一个正在运行的容器。
4		build	根据 Dockerfile 文件创建 image。
5		commit	根据容器的改变创建一个新的 image。
6		cp	从容器中拷贝文件/目录到 host 上。
		create	创建一个 Container，但不运行任何命令。
7		diff	查看容器的文件系统变化。
		exec	在一个已运行的 Container 中执行命令。
8		export / save	把容器的文件系统/image 文件导出到标准输出流。
9		images	列出 image。
10		import / load	导入/加载一个 image。
11		inspect	查看 image 或者 container 的配置信息。
12		kill	杀掉正在运行的 container
13		port	查看 Container 与 Host 的端口映射。
14		pause / unpause	container 中的所有进程都被 pause/unpause。
15		ps	列出 container。
16		rm	删除一个或多个 containers。
17		rmi	删除一个或多个 images。
18		run	启动一个新的 Container，且执行一个命令。
19		start / stop / restart	启动/停止/重新启动 一个 Container。
20		tag	给 image 打个 tag 标记，类似于重命名。
21		top	查看 Container 中正在运行的进程。
22		wait	阻塞式等待一个 container 停止。
23	日志信息相关	events	从 Docker Server 获取一些实时的事件。
24		history	显示一个 image 的 history。
25		logs	获取 container 的 logs。
26	Docker Hub 服务相关	login / logout	登录/退出 Docker 官网的 Registry Server。
27		pull / push	下载/上传一个 image。
28		search	搜索 image。

在对每个命令进行详细介绍前，以下几点需要知道：

1) 单个字符的参数可以分开配置，也可以放在一起组合配置。如下所示

```
docker run -t -i image /bin/bash
docker run -ti image  /bin/bash
```

2) 参数选项如果使用符号“[]”,那么该参数可以被定义多次，否则只能被定义一次。如下所示：
(参数 hostname 和 publish 使用不同的符号，--hostname="", --publish=[])

```
docker run -ti --rm --hostname="Linux" kulong0105 /bin/bash
docker run -ti --publish 22 --publish 23  image  /bin/bash
```

3) docker 默认会监听 unix：///var/run/docker.sock 文件，只允许本地的 root 用户连接。要想从远端连接，在启动 docker 服务时，可以使用-H 参数来设置监听的 sock 文件(如：docker -d -H tcp://0.0.0.0:2375)，但是这种方式并不被推荐，因为容易造成安全隐患，可能使得远程攻击者获取 Host 的 root 权限。

6.1.1 info & version

语法：

Usage	Description
docker info	显示 system-wide 信息。
docker version	显示 docker 版本相关信息。

举例：

```
[root@localhost renyl]# docker info
Containers: 18
Images: 19
Storage Driver: devicemapper
 Pool Name: docker-253:0-27269-pool
 Pool Blocksize: 65.54 kB
 Data file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata file: /var/lib/docker/devicemapper/devicemapper/metadata
 Data Space Used: 1.039 GB
 Data Space Total: 107.4 GB
 Metadata Space Used: 4.788 MB
 Metadata Space Total: 2.147 GB
 Library Version: 1.02.82-git (2013-10-04)
Execution Driver: native-0.2
Kernel Version: 3.10.0-123.el7.x86_64
Operating System: Red Hat Enterprise Linux Server 7.0 (Maipo)
CPUs: 1
Total Memory: 990.8 MiB
Name: localhost.localdomain
ID: GJWW:PJZO:AIMS:2EBG:IATE:7ZQZ:2E7P:S5H4:TB4C:IXFQ:MKWS:UUXP
[root@localhost renyl]# docker version
Client version: 1.4.1
Client API version: 1.16
Go version (client): go1.3.3
Git commit (client): 5bc2ff8
OS/Arch (client): linux/amd64
Server version: 1.4.1
Server API version: 1.16
Go version (server): go1.3.3
Git commit (server): 5bc2ff8
[root@localhost renyl]#
```

6.1.2 attach

语法：

Usage	Description
docker attach [OPTIONS] CONTAINER	用来 attach 一个正在后台运行的容器。

OPTIONS	Description
--no-stdin=false	不 attach 输入流（STDIN）。
--sig-proxy=true	代理处理所有接收到的信号（除了 SIGCHLD, SIGKILL, and SIGSTOP 信号）。

举例：

```
[root@localhost bin]# docker ps
CONTAINER ID    IMAGE           COMMAND    CREATED    STATUS    PORTS    NAMES
2419ca389137 kulong0105:latest    "/bin/bash"    3 hours ago  Up 3 hours
tender_poincare
[root@localhost bin]# docker attach 241
pwd
/tmp
bash-4.2#
```

注：
在 Container 环境下，如果使用命令“exit”不仅会退出 Container 环境，同时也会关闭该 Container。要想退出 Container 环境，但不关闭该 Container，可以使用组合按键“ctrl+p + ctrl+q”来实现。

6.1.3 build

语法：

Usage	Description
docker build [OPTIONS] PATH | URL | -	根据 Dockerfile 创建一个新的 image。

OPTIONS	Description
--force-rm=false	强制删除 intermediate containers。
--no-cache=false	创建 image 过程中不使用 cache。
--pull=false	总是尝试 pull 新版的 image。
-q, --quiet=false	减少向终端打印 verbose 信息。
--rm=true	image 创建成功后删除 intermediate containers。
-t, --tag=""	给 Repository 设置一个 tag。

举例：

```
[root@localhost testdir]# docker  images
REPOSITORY        TAG          IMAGE ID        CREATED          VIRTUAL SIZE
kulong0105        latest       2157a84fbc40     5 days ago          458.4 MB
[root@localhost testdir]# docker build  -q -t kulong0106  .
Sending build context to Docker daemon  5.12 kB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
Step 1 : ENV MYNAME Allen Ren
 ---> Running in c637a8692040
 ---> 3a3c4d2b7d42
Removing intermediate container c637a8692040
```

```
Step 2 : RUN echo $MYNAME
 ---> Running in c11f6e9353e3
 ---> 8f2515161a87
Removing intermediate container c11f6e9353e3
Successfully built 8f2515161a87
[root@localhost testdir]# docker  images
REPOSITORY        TAG          IMAGE ID          CREATED                VIRTUAL SIZE
kulong0106        latest       8f2515161a87      About a minute ago       458.4 MB
kulong0105        latest       2157a84fbc40        5 days ago             458.4 MB
[root@localhost testdir]#
```

6.1.4 commit

语法：

Usage	Description
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]	根据 Container 的改变来创建一个新的 image。

OPTIONS	Description
-a, --author=""	设置 commit 的 author。
-m, --message=""	设置 commit 的 message。
-p, --pause=true	在 commit 期间暂停 container 的运行。

举例：

```
[root@localhost testdir]# docker run -ti kulong0106 touch /testfile
[root@localhost testdir]# docker ps -l
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
94f8e696438b        kulong0106:latest   "touch /testfile"   5 seconds ago       Exited
(0) 4 seconds ago                       serene_lovelace
[root@localhost testdir]# docker commit  -a "Allen"  -m "just test"  94f kulong0107
82f22eea5c4187632cbf6ead970f0583487688533f6dfadd0bb8cf1a5c2ee7e4
[root@localhost testdir]# docker images
REPOSITORY            TAG            IMAGE ID        CREATED          VIRTUAL SIZE
kulong0107            latest         82f22eea5c41    4 seconds ago       458.4 MB
kulong0106            latest         8f2515161a87    10 minutes ago      458.4 MB
kulong0105            latest         2157a84fbc40    5 days ago          458.4 MB
[root@localhost testdir]#
```

注：

1) container_id 可以使用前 3 位，只要这 3 位可以唯一确定一个 Container。
2) 针对/etc/hosts, /etc/hostname, /etc/resolve.conf 三个文件的修改都是临时的，只在运行的容器中保留，容器终止或重启后并不会被保存下来，也不会被 docker commit 提交。

6.1.5 cp

语法：

Usage	Description
docker cp CONTAINER:PATH HOSTPATH	从 Container 中拷贝文件/目录到 Host 上。

举例：

```
[root@localhost testdir]# docker run -ti kulong0107 /bin/bash
bash-4.2# ls /home/renyl/
test.txt  testfile
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID      IMAGE               COMMAND           CREATED          STATUS
PORTS             NAMES
f787f98ba1f8      kulong0107:latest   "/bin/bash"       28 seconds ago    Up 27 seconds
                  jovial_galileo
[root@localhost testdir]# ls
Dockerfile  Dockerfile.bak  unuseful.txt  useful.txt
[root@localhost testdir]# docker cp f78:/home/renyl/testfile   ./
[root@localhost testdir]# ls
Dockerfile  Dockerfile.bak  testfile  unuseful.txt  useful.txt
[root@localhost testdir]#
```

6.1.6 create

语法：

Usage	Description
docker create [OPTIONS] IMAGE [COMMAND] [ARG...]	创建一个 Container，但不运行任何命令。

注：
[OPTIONS]与命令“docker run”中的[OPTIONS]基本一致，具体可参看命令“docker run”中[OPTIONS]的介绍。

举例：

```
[root@localhost testdir]# docker images
REPOSITORY              TAG       IMAGE ID            CREATED             VIRTUAL SIZE
kulong0105              latest    2157a84fbc40       8 days ago          458.4 MB
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED          STATUS
PORTS               NAMES
[root@localhost testdir]# docker create -ti kulong0105
6eeed4c22827c43591177e31d1efd3b7abe1ece901f62d6530d7fe0b3a2b88bc
 [root@localhost testdir]# docker ps -l
CONTAINER ID        IMAGE               COMMAND              CREATED         STATUS
PORTS               NAMES
6eeed4c22827        kulong0105:latest   "yum install passwd" 19 seconds ago
lonely_bohr
[root@localhost testdir]# docker start 6eee
```

注：如果需要启动这个 Container 的话，可以使用命令“docker start”来实现。

6.1.7 diff

语法：

Usage	Description
docker diff CONTAINER	查看 Container 中文件系统的变化。

举例：

```
[root@localhost testdir]# docker run -ti  kulong0107 /bin/bash
bash-4.2# cd /home/renyl/
bash-4.2# ls
test.txt  testfile
bash-4.2# echo "hello world" > testfile
bash-4.2# touch testfile2
bash-4.2# rm -rf test.txt
bash-4.2# ls
testfile  testfile2
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID     IMAGE            COMMAND        CREATED             STATUS
PORTS            NAMES
a993d9b49222     kulong0107:latest "/bin/bash"   51 seconds ago      Up 50 seconds
                 sad_einstein
 [root@localhost testdir]# docker diff a99
C /home
C /home/renyl
C /home/renyl/testfile
A /home/renyl/testfile2
D /home/renyl/test.txt
[root@localhost testdir]#
```

注：针对 Container 中文件系统的不同变化用不同字母表示：C=Change，A=Add，D=Delete。

6.1.8 exec

语法：

Usage	Description
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]	在一个已运行的 Container 中执行命令。

OPTIONS	Description
-d, --detach=false	设置 commit 的 author。
-i, --interactive=false	设置 commit 的 message。
-t, --tty=false	在 commit 期间暂停 container 的运行。

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID     IMAGE            COMMAND        CREATED             STATUS
```

```
PORTS               NAMES
a993d9b49222        kulong0107:latest "/bin/bash"   51 seconds ago      Up 50 seconds
                    sad_einstein
[root@localhost testdir]# docker exec a99 ls /home/renyl/
testfile
testfile2
[root@localhost testdir]#
```

6.1.9 export / save

语法：

Usage	Description
docker export CONTAINER	把 Container 的文件系统作为一个 tar 文档输出到标准输出流（STDOUT）。

Usage	Description
docker save [OPTIONS] IMAGE [IMAGE...]	保存 image 到一个 tar 文档,默认写到标准输出流（STDOUT）。

OPTIONS	Description
-o, --output=""	写到文件，来替换写到标准输出终端（STDOUT）

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED         STATUS
PORTS               NAMES
8d1390a0e10d        kulong0105:latest   "/bin/bash"   2 minutes ago   Up About a minute
                    cranky_elion
[root@localhost testdir]# docker export 8d1 > kulong0105_container.tar
[root@localhost testdir]# docker stop 8d13
8d13
[root@localhost testdir]# docker images
REPOSITORY       TAG       MAGE ID           CREATED           VIRTUAL SIZE
kulong0105       latest    2157a84fbc40      8 days ago        458.4 MB
[root@localhost testdir]# docker save -o kulong0105_image.tar  215
[root@localhost testdir]# ls
Dockerfile  kulong0105_container.tar  kulong0105_image.tar  testfile
[root@localhost testdir]#
```

6.1.10 images

语法：

Usage	Description
docker images [OPTIONS] [REPOSITORY]	列出系统中 images。

OPTIONS	Description
-a, --all=false	列出系统中所有 images，默认情况下会过滤中间层的 image。
-f, --filter=[]	根据过滤设置列出系统中的 images。
--no-trunc=false	不截断输出信息。
-q, --quiet=false	只显示 images 的 ID。

举例：

```
[root@localhost testdir]# docker images
REPOSITORY        TAG          IMAGE ID            CREATED             VIRTUAL SIZE
kulong0105        latest       2157a84fbc40        5 days ago          458.4 MB
[root@localhost testdir]#
```

注：

1) “IMAGE ID”是由 64 个十六进制的字符所组成来唯一标识，使用命令“docker images --no-trunc”可以查看到完整的“IMAGE ID”。

2) image 文件存储在/var/lib/docker/* 目录下。但若该目录使用 volume 进行管理，采用 B-Tree filesystem（如：Brtrfs 文件系统），SELinux 是不被支持的。

6.1.11 import / load

语法：

Usage	Description
docker import URL|- [REPOSITORY[:TAG]]	创建一个空的文件系统 image，然后导入 tarball 的内容文件到这个 image。

Usage	Description
docker load [OPTIONS]	从标准输入流（STDIN）中加载一个 image。

OPTIONS	Description
-i, --input=""	从 tar 文档读取，替代从标准输入流（SDTIN）。

举例：

```
[root@localhost testdir]# docker images
REPOSITORY            TAG            IMAGE ID        CREATED             VIRTUAL SIZE
kulong0105            latest         2157a84fbc40    8 days ago          458.4 MB
[root@localhost testdir]# ls
Dockerfile  kulong0105_container.tar  kulong0105_image.tar  testfile
[root@localhost testdir]# cat kulong0105_container.tar |docker import - kulong0106
376f8a35c078e3bb6f9cb7014783310c486c07ffb5d7ba92aa307d458c3643b6
[root@localhost testdir]# docker images
REPOSITORY          TAG          IMAGE ID            CREATED             VIRTUAL SIZE
kulong0106          latest       376f8a35c078        26 seconds ago      384.3 MB
kulong0105          latest       2157a84fbc40        8 days ago          458.4MB
[root@localhost testdir]# docker load -i kulong0105_image.tar
```

```
[root@localhost testdir]#
```

注：

1) 命令“docker export”和命令“docker import”搭配使用，命令“docker save”和命令“docker load”搭配使用，不可混用。

2) 除了从存在的 tar 包上创建一个 image，还可以通过官方提供的脚本来创建一个 base-image，参考地址：https://github.com/docker/docker/blob/master/contrib/mkimage.sh ．当然还可以通过简单的命令来创建一个 base-image，如：

```
# debootstrap raring raring > /dev/null
# tar -C raring -c . | sudo docker import - raring
```

6.1.12 inspect

语法：

Usage	Description
docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]	查看 image 或者 container 的配置信息。

OPTIONS	Description
-f, --format=""	根据指定 format 打印配置信息。

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
a993d9b49222        kulong0107:latest   "/bin/bash"         28 minutes ago      Up 28 minutes
sad_einstein
[root@localhost testdir]# docker inspect  --format='{{.NetworkSettings.IPAddress}}' a99
172.17.0.30
[root@localhost testdir]#
```

6.1.13 kill

语法：

Usage	Description
docker kill [OPTIONS] CONTAINER [CONTAINER...]	杀掉正在运行的 container 或者给 container 发送一个指定的信号。

OPTIONS	Description
-s, --signal="KILL"	发送指定的信号给 container。

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
a993d9b49222        kulong0107:latest   "/bin/bash"         34 minutes ago      Up 34 minutes
```

```
sad_einstein
[root@localhost testdir]# docker kill a99
a99
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
[root@localhost testdir]#
```

6.1.14 port

语法：

Usage	Description
docker port CONTAINER [PRIVATE_PORT[/PROTO]]	查看 Container 与 Host 的端口映射。

举例：

```
[root@localhost testdir]# docker  run -ti -p 23:22 kulong0107 /bin/bash
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID      IMAGE               COMMAND          CREATED             STATUS
PORTS             NAMES
87f02ce151f2      kulong0107:latest   "/bin/bash"      6 seconds ago       Up 5 seconds
0.0.0.0:23->22/tcp   high_hodgkin
[root@localhost testdir]# docker port 87f
22/tcp -> 0.0.0.0:23
[root@localhost testdir]#
```

6.1.15 pause / unpause

语法：

Usage	Description
docker pause CONTAINER	container 中的所有进程都被 pause。
docker unpause CONTAINER	container 中的所有进程都被 unpause。

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID      IMAGE               COMMAND          CREATED          STATUS
PORTS             NAMES
87f02ce151f2      kulong0107:latest   "/bin/bash"      4 minutes ago    Up 4 minutes
0.0.0.0:23->22/tcp   high_hodgkin
[root@localhost testdir]# docker pause 87f
87f
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND        CREATED       STATUS
PORTS               NAMES
87f02ce151f2        kulong0107:latest   "/bin/bash" 4 minutes ago  Up 4 minutes (Paused)
```

```
0.0.0.0:23->22/tcp  high_hodgkin
[root@localhost testdir]# docker unpause  87f
87f
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND         CREATED         STATUS
PORTS               NAMES
87f02ce151f2        kulong0107:latest "/bin/bash"   4 minutes ago   Up 4 minutes
0.0.0.0:23->22/tcp  high_hodgkin
[root@localhost testdir]#
```

6.1.16 ps

语法：

Usage	Description
docker ps [OPTIONS]	列出 container。

OPTIONS	Description
-a, --all=false	列出所有的 container。默认情况下只列出正在运行的 container。
--before=""	只列出指定 ID 或 name 之前的 container（包括不正在运行的 container）。
-f, --filter=[]	根据过滤条件列出相应的 container。
-l, --latest=false	只列出 latest 创建的 container（包括不在运行的 container）。
-n=-1	只列出 n last 创建的 container（包括不在运行的 container）。
--no-trunc=false	不截断输出信息。
-q, --quiet=false	只列出 container 的 ID。
-s, --size=false	列出文件大小。
--since=""	只列出指定 ID 或 name 之后的 container（包括不正在运行的 container）。

举例：

```
[root@localhost testdir]# docker ps -n 2
CONTAINER ID     IMAGE              COMMAND        CREATED          STATUS
PORTS            NAMES
87f02ce151f2     kulong0107:latest  "/bin/bash"    19 minutes ago   Up 19 minutes
0.0.0.0:23->22/tcp   high_hodgkin
383f2f9aac5b     kulong0107:latest  "/bin/bash"    22 minutes ago   Exited (0) 19
minutes ago          jolly_albattani
[root@localhost testdir]#
```

6.1.17 rm

语法：

Usage	Description
docker rm [OPTIONS] CONTAINER [CONTAINER...]	删除一个或多个 containers。

OPTIONS	Description
-f, --force=false	强制删除一个正在运行的 container。
-l, --link=false	删除 link 标记。
-v, --volumes=false	删除 volume 标记。

举例：

```
[root@localhost testdir]# docker ps -n 2
CONTAINER ID     IMAGE              COMMAND        CREATED          STATUS
PORTS            NAMES
87f02ce151f2     kulong0107:latest  "/bin/bash"    19 minutes ago   Up 19 minutes
0.0.0.0:23->22/tcp   high_hodgkin
383f2f9aac5b     kulong0107:latest  "/bin/bash"    22 minutes ago   Exited (0) 19
minutes ago          jolly_albattani
[root@localhost testdir]# docker rm 383
383
[root@localhost testdir]# docker ps -n 2
CONTAINER ID     IMAGE              COMMAND        CREATED          STATUS
PORTS            NAMES
87f02ce151f2     kulong0107:latest  "/bin/bash"    19 minutes ago   Up 19 minutes
0.0.0.0:23->22/tcp   high_hodgkin
3433877c3bce     kulong0107:latest  "/bin/bash"    36 minutes ago   Exited (0) 32
minutes ago          angry_tesla
[root@localhost testdir]#
```

注：命令“docker rm ` docker ps -a -q`”可以删除当前系统中所有 Container。

6.1.18 rmi

语法：

Usage	Description
docker rmi [OPTIONS] IMAGE [IMAGE...]	删除一个或多个 images。

OPTIONS	Description
-f, --force=false	强制删除一个 image。
--no-prune=false	不删除没有打 tag 标记的 parent image。

举例：

```
[root@localhost testdir]# docker images
REPOSITORY          TAG         IMAGE ID        CREATED              VIRTUAL SIZE
```

```
kulong0107          latest      25e655be4072    About an hour ago    458.4 MB
kulong0106          latest      8f2515161a87    About an hour ago    458.4 MB
kulong0105          latest      2157a84fbc40    5 days ago           458.4 MB
[root@localhost testdir]# docker rmi kulong0106
Error response from daemon: Conflict, cannot delete 8f2515161a87 because the container
87f02ce151f2 is using it, use -f to force
FATA[0000] Error: failed to remove one or more images
[root@localhost testdir]# docker rmi -f kulong0106
Untagged: kulong0106:latest
[root@localhost testdir]# docker images
REPOSITORY          TAG         IMAGE ID        CREATED              VIRTUAL SIZE
kulong0107          latest      25e655be4072    About an hour ago    458.4 MB
kulong0105          latest      2157a84fbc40    5 days ago           458.4 MB
[root@localhost testdir]#
```

6.1.19 run

语法：

Usage	Description
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]	启动一个新的 Container，且执行一个命令。

OPTIONS	Description
-a, --attach=[]	连接到 STDIN, STDOUT or STDERR。
--add-host=[]	增加一个自定义的 host-to-IP 映射。
-c, --cpu-shares=0	设定共享 CPU。
--cap-add=[]	设定 Container 中可以使用的功能。
--cap-drop=[]	设定 Container 中不可以使用的功能。
--cidfile=""	Container 的 ID 写到指定文件中。
--cpuset=""	指定 Container 可运行的 cpuset。
-d, --detach=false	设定 Container 为后台运行模式。
--device=[]	增加 host 的设备到 Container 中。
--dns=[]	设定自定义的 DNS 服务器。
--dns-search=[]	设定自定义的 DNS 搜索域。
-e, --env=[]	设置环境变量。
--entrypoint=""	覆盖 image 中默认的 ENTRYPOINT。
--env-file=[]	设定行分隔的文件中读取环境变量。
--expose=[]	expose container 的端口，但不 publish 到 host。
-h, --hostname=""	设定 Container 的 host name。
-i, --interactive=false	设定 STDIN 为打开状态。
--ipc=""	设定 IPC 方式（默认使用一个私有的 IPC 命名间）。
--link=[]	增加一个 link 到另一个 Container。
--lxc-conf=[]	增加自定义的 lxc 选项（必须用 lxcexec-driver）。
-m, --memory=""	设定内存大小限制。
--mac-address=""	设定 Container 的 Mac 地址。

--name=""	设定 Container 的 name。
--net="bridge"	设置 Container 的网络模式。
-P, --publish-all=false	publish container 所有 expose 的端口号到 host。
-p, --publish=[]	publish container 的端口号到 host。
--privileged=false	给 container 设置更多权利。Docker 中运行 docker。
--restart=""	设定重新启动的 policy。
--rm=false	当 container 退出时，自动删除该 container。
--security-opt=[]	设置安全选项。
--sig-proxy=true	代理接收信号并处理。
-t, --tty=false	分配一个伪终端。
-u, --user=""	设定 Username 或 UID。
-v, --volume=[]	绑定挂载卷。
--volumes-from=[]	从指定的 container 挂载卷。
-w, --workdir=""	设定 Container 的工作目录

命令“docker run”中的参数选项众多，接下来，选取 4 个重点参数进行举例介绍，如下：

举例 1：（-cap-drop=[] 选项）

```
[root@localhost testdir]# docker run --cap-drop="CHOWN" -ti kulong0105 /bin/bash
bash-4.2# cd /home/renyl/
bash-4.2# ls -al
total 12
drwxr-xr-x  2 root root 4096 Dec 24 09:24 .
drwxr-xr-x. 3 root root 4096 Dec 24 09:22 ..
-rw-r--r--  1 root root    9 Dec 24 09:24 test.txt
bash-4.2# chown  kulong0105:kulong0105  test.txt
chown: changing ownership of 'test.txt': Operation not permitted
bash-4.2# exit
exit
[root@localhost testdir]#
```

注：参数选项“--privileged”可以完成同样的功能。

举例 2：（-p, --publish=[]、-P 选项）

```
[root@localhost testdir]# docker run -ti --name Allen -p 80:80 kulong0105 /bin/bash
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND        CREATED          STATUS
PORTS               NAMES
48ae14c39dff        kulong0105:latest   "/bin/bash"    6 seconds ago    Up 5 seconds
0.0.0.0:80->80/tcp   Allen
[root@localhost testdir]# docker stop 48ae14c39dff
[root@localhost testdir]# docker run -ti --name Lucky  -P kulong0105 /bin/bash
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND         CREATED          STATUS
PORTS               NAMES
8df8cca54107        kulong0105:latest   "/bin/bash"     30 seconds ago   Up 29 seconds
0.0.0.0:49153->5000/tcp   Lucky
```

```
[root@localhost testdir]#
```

注：

1) -p 选项(小写字母)默认是绑定 TCP 协议的，要想绑定 UDP 协议，可以这样使用：“-p 5000:5000/udp”。
2) -p 选项（小写字母）可以绑定到一个动态的端口号，可以这样使用“-p 127.0.0.1：：5000”。
3) -P 选项（大写字母）用来自动映射 Container 中 expose 端口号到 Host 上一个随机（49153~65535）的端口号。

举例 3：（--link=[] 选项）

```
[root@localhost /]#  docker images
REPOSITORY          TAG          MAGE ID            CREATED             VIRTUAL SIZE
kulong0106          latest       e0686584e193       33 minutes ago      458.4 MB
kulong0105          latest       b4abd4fc3ef0       21 hours ago        458.4 MB
[root@localhost /]#  docker inspect --format='{{.Config.ExposedPorts}}'  kulong0105
map[5000/tcp:map[]]
[root@localhost /]# docker inspect   --format='{{.Config.ExposedPorts}}'  kulong0106
map[8080/tcp:map[]]
[root@localhost /]# docker run -ti  --rm  --name source kulong0105 /bin/bash
bash-4.2# env
HOSTNAME=282751e0777f
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
_=/usr/bin/env
bash-4.2# ifconfig
eth0: flags=67<UP,BROADCAST,RUNNING>  mtu 1500
        inet 172.17.0.8  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:8  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:08  txqueuelen 0  (Ethernet)
        RX packets 6  bytes 508 (508.0 B)
…
bash-4.2# [root@localhost /]#docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED             STATUS
PORTS               NAMES
282751e0777f        kulong0105:latest "/bin/bash"  About a minute ago  Up About a minute
5000/tcp            source
[root@localhost /]# docker run -ti --rm --name target --link source:mylink kulong0106
/bin/bash
bash-4.2# env
MYLINK_PORT_5000_TCP=tcp://172.17.0.8:5000
MYLINK_PORT=tcp://172.17.0.8:5000
HOSTNAME=ee465e70bd38
TERM=xterm
MYLINK_PORT_5000_TCP_PORT=5000
MYLINK_PORT_5000_TCP_PROTO=tcp
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
```

```
MYLINK_NAME=/target/mylink
SHLVL=1
HOME=/root
MYLINK_PORT_5000_TCP_ADDR=172.17.0.8
_=/usr/bin/env
bash-4.2# ifconfig
eth0: flags=67<UP,BROADCAST,RUNNING>  mtu 1500
        inet 172.17.0.10  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:a  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:0a  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 648 (648.0 B)
…
bash-4.2# cat /etc/hosts
172.17.0.10     ee465e70bd38
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.8      mylink
bash-4.2# ping -c 3 mylink
PING mylink (172.17.0.8) 56(84) bytes of data.
64 bytes from mylink (172.17.0.8): icmp_seq=1 ttl=64 time=0.101 ms
64 bytes from mylink (172.17.0.8): icmp_seq=2 ttl=64 time=0.032 ms
64 bytes from mylink (172.17.0.8): icmp_seq=3 ttl=64 time=0.032 ms

--- mylink ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.032/0.055/0.101/0.032 ms
bash-4.2#[root@localhost /]# docker ps
CONTAINER ID     IMAGE              COMMAND      CREATED              STATUS
PORTS            NAMES
ee465e70bd38     kulong0106:latest  "/bin/bash"  About a minute ago   Up About a minute
8080/tcp         target
282751e0777f     kulong0105:latest   "/bin/bash" 4 minutes ago        Up 4 minutes
5000/tcp         source
[root@localhost /]#
```

注：

1) Container 之间的通信除了通过端口映射的方式，Docker 还提供了一种称为“linking Containers”的方式。
2) “linking Container“方式提供了两种方式来暴露相互 source Container 到 target Container 的连接信息，分别为“环境变量”和“/etc/hosts 文件”。同时，通过 ssh 等产生新 shell 的方式将看不到“环境变量”。
3) “linking Container“方式提供的通道是安全和私有的，不需要 publish 任何端口号。
4) 官网介绍，source Container 重启后，target Container 中的“/etc/hosts”文件会自动根据 source Container 中的 IP 变化而变化，经过实测发现不行，仍需要重启 target Container。
5) 被链接的容器必须运行在同一个 Docker 宿主机上，不同 Docker 宿主机上运行的容器无法连接。

举例 4：（-v, --volume=[]、--volumes-from=[] 选项）

```
[root@localhost testdir]# pwd
/home/renyl/testdir
[root@localhost testdir]# ls
Dockerfile  testfile
[root@localhost testdir]# docker run -ti -v /home/renyl/testdir:/tmp/  kulong0105
/bin/bash
bash-4.2# pwd
/tmp
bash-4.2# ls
Dockerfile  testfile
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID     IMAGE               COMMAND      CREATED             STATUS
PORTS            NAMES
297282ae62e4     kulong0105:latest   "/bin/bash"  About a minute ago  Up About a minute
                 stupefied_goodall
[root@localhost testdir]# docker run -ti --volumes-from 297  centos /bin/bash
[root@4b82a6b1a52a /]# cd /tmp/
[root@4b82a6b1a52a tmp]# ls
Dockerfile  testfile
[root@4b82a6b1a52a tmp]# exit
exit
[root@localhost testdir]#
```

注：

1) -v 选项后的参数 container 即使停止了，仍然能够共享 volume 中的数据。
2) -v 选项还可以针对共享的目录设置只读或读写权限，格式为：“-v host_dir:conainter_dir:access_rights”，其中 access_right 使用“ro”表示只读权限，“rw”表示读写权限。

6.1.20 start / stop / restart

语法：

Usage	Description
docker stop [OPTIONS] CONTAINER [CONTAINER...]	通过发送 SIGTERM 信号来停止一个正在运行的 container，在 grace period 之后如果 container 还未停止，发送 SIGKILL 信号来停止 container。

OPTIONS	Description
-t, --time=10	停止一个正在运行的 container 之前等待多少秒。

Usage	Description
docker start [OPTIONS] CONTAINER [CONTAINER...]	重新启动一个已停止的 container。

OPTIONS	Description

-a, --attach=false	连接容器的标准输出流和标准错误输出流(STDOUT and STDERR)且处理所有被 forward 的信号。
-i, --interactive=false	连接容器的标准输入流（STDIN）。

Usage	Description
docker restart [OPTIONS] CONTAINER [CONTAINER...]	重新启动一个已停止的 container。

OPTIONS	Description
-t, --time=10	杀掉正在运行的 container 之前等待多少秒。

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND        CREATED             STATUS
PORTS               NAMES
1e853d245115        kulong0107:latest   "/bin/bash"    21 minutes ago      Up 8 minutes
modest_bohr
[root@localhost testdir]# time docker stop 1e8
1e8

real    0m10.056s
user    0m0.016s
sys     0m0.007s
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
[root@localhost testdir]# docker start -ai 1e8
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND        CREATED             STATUS
PORTS               NAMES
1e853d245115        kulong0107:latest   "/bin/bash"    22 minutes ago      Up 9 seconds
modest_bohr
[root@localhost testdir]# time docker restart 1e8
1e8

real    0m10.376s
user    0m0.015s
sys     0m0.008s
[root@localhost testdir]# docker ps
CONTAINER ID        IMAGE               COMMAND         CREATED             STATUS
PORTS               NAMES
1e853d245115        kulong0107:latest   "/bin/bash"     23 minutes ago      Up 5 seconds
modest_bohr
[root@localhost testdir]#
```

6.1.21 tag

语法：

Usage	Description
docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]	给 image 打个 tag 标记，类似于重命名。

OPTIONS	Description
-f, --force=false	强制执行。

举例：

```
[root@localhost testdir]# docker images
REPOSITORY              TAG          IMAGE ID          CREATED           VIRTUAL SIZE
kulong0107              latest       25e655be4072      15 hours ago      458.4 MB
kulong0105              latest       2157a84fbc40      5 days ago        458.4 MB
[root@localhost testdir]# docker tag kulong0107   allen/kulong0108
[root@localhost testdir]# docker images
REPOSITORY             TAG          IMAGE ID          CREATED           VIRTUAL SIZE
allen/kulong0108       latest       25e655be4072      15 hours ago      458.4 MB
kulong0107             latest       25e655be4072      15 hours ago      458.4 MB
kulong0105             latest       2157a84fbc40      5 days ago        458.4 MB
[root@localhost testdir]#
```

6.1.22 top

语法：

Usage	Description
docker top CONTAINER [ps OPTIONS]	查看 Container 中正在运行的进程。

举例：

```
[root@localhost testdir]# docker run -ti kulong0107 /bin/bash
bash-4.2# sleep 1000 &
[1] 7
bash-4.2# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 02:54 ?        00:00:00 /bin/bash
root         7     1  0 02:54 ?        00:00:00 sleep 1000
root         8     1  0 02:54 ?        00:00:00 ps -ef
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID    IMAGE              COMMAND        CREATED             STATUS
PORTS           NAMES
78c0fa0be753    kulong0107:latest  "/bin/bash"    15 seconds ago      Up 14 seconds
ecstatic_wright
[root@localhost testdir]# docker top 78c
UID     PID      PPID       C      STIME       TTY        TIME          CMD
root    8643     24789      0      21:54       pts/2      00:00:00      /bin/bash
```

```
root     8686     8643        0       21:54        pts/2       00:00:00       sleep 1000
[root@localhost testdir]#
```

6.1.23 wait

语法：

Usage	Description
docker wait CONTAINER [CONTAINER...]	阻塞式等待一个 container 停止，然后打印它的退出码。

举例：

```
[root@localhost testdir]# docker ps
CONTAINER ID    IMAGE              COMMAND          CREATED          STATUS
PORTS           NAMES
78c0fa0be753    kulong0107:latest  "/bin/bash"      8 minutes ago    Up 8 minutes
ecstatic_wright
[root@localhost testdir]# docker wait 78c      #这里会一直被阻塞，除非 container 停止
-1                    #在另一终端执行命令“docker stop 78c”之后，这里才打印出退出码
[root@localhost testdir]#
```

6.1.24 events

语法：

Usage	Description
docker events [OPTIONS]	从 Docker Server 获取一些实时的事件。

OPTIONS	Description
-f, --filter=[]	提供过滤条件，如 event=stop。
--since=""	显示从指定 timestamp 之后的事件。
--until=""	显示到指定 timestamp 之前的事件。

举例：

```
 [root@localhost testdir]# docker events
2014-12-29T22:12:07.000000000-05:00 8658fc03c26d: (from kulong0107:latest) die
2014-12-29T22:14:01.000000000-05:00 fb0811ca9eb2: (from kulong0107:latest) start
^C[root@localhost testdir]#
```

注：
命令“docker events”是阻塞式的，上述输出是由于在另一个终端下执行命令“docker stop container_id”和“docker start container_id”产生的。

6.1.25 history

语法：

Usage	Description

docker history [OPTIONS] IMAGE	显示一个 image 的 history。

OPTIONS	Description
--no-trunc=false	不截断输出信息。
-q, --quiet=false	只显示数值 ID。

举例：

```
[root@localhost testdir]# docker history kulong0107
IMAGE          CREATED         CREATED BY                                      SIZE
25e655be4072   16 hours ago    touch /home/renyl/testfile                      0 B
8f2515161a87   16 hours ago    /bin/sh -c echo $MYNAME                         0 B
3a3c4d2b7d42   16 hours ago    /bin/sh -c #(nop) ENV MYNAME=Allen Ren          0 B
9a3156510299   5 days ago      /bin/sh -c #(nop) ADD file:df8681be7b081bfa9c   9 B
b6bde356e1d2   5 days ago      /bin/sh -c #(nop) USER [root]                   0 B
0a0ffa6b2654   5 days ago      /bin/sh -c #(nop) ADD file:df98cc3fd9c30a8d70   8 B
4a607e6eaca6   5 days ago      /bin/sh -c #(nop) ENV MYNAME=Allen Ren          0 B
6492b81a2b6a   5 days ago      /bin/sh -c #(nop) WORKDIR /tmp/                 0 B
43af81d7ef8b   5 days ago      /bin/sh -c #(nop) USER [root]                   0 B
e1786518ed85   9 weeks ago     yum install passwd                              14.11 MB
324ec5f6c9c5   9 weeks ago     /bin/bash                                       14.2 MB
5c2458ba8931   9 weeks ago     yum install openssh*                            60.12 MB
001a512fad7a   9 weeks ago     yum install sysstat                             13.09 MB
f7668685064f   9 weeks ago     yum install net-tools*                          12.58 MB
d207cdb5fdfa   9 weeks ago     yum install gcc                                 120.3 MB
87e5b6b3ccc1   12 weeks ago    /bin/sh -c #(nop) ADD file:9b13ab24098a9148d6   224 MB
5b12ef8fd570   12 weeks ago    /bin/sh -c #(nop) MAINTAINER The CentOS Proje   0
[root@localhost testdir]#
```

注：目前 docker 中的一个 image 文件最多只能支持 127 层。

6.1.26 logs

语法：

Usage	Description
docker logs [OPTIONS] CONTAINER	获取 container 的 logs。

OPTIONS	Description
-f, --follow=false	阻塞式运行，除非 container 停止运行了。
-t, --timestamps=false	显示 timestap。
--tail="all"	显示指定行数的 logs，默认显示所有 logs。

举例：

```
[root@localhost testdir]# docker run -ti kulong0105 /bin/bash
bash-4.2# cd /home/renyl/
bash-4.2# ls
test.txt
```

```
bash-4.2# touch testfile
bash-4.2# [root@localhost testdir]# docker ps
CONTAINER ID   IMAGE              COMMAND          CREATED              STATUS
PORTS          NAMES
661abe2503bf   kulong0105:latest  "/bin/bash"      17 seconds ago       Up 16 seconds
evil_lumiere
[root@localhost testdir]# docker logs  -t 661
2014-12-30T03:47:25.477607872Z bash-4.2# cd /home/renyl/
2014-12-30T03:47:26.151650389Z bash-4.2# ls
2014-12-30T03:47:26.169867298Z test.txt
2014-12-30T03:47:33.970084808Z bash-4.2# touch testfile
[root@localhost testdir]#
```

6.1.27 login / logout

语法：

Usage	Description
docker login [OPTIONS] [SERVER]	如果 Server 没有指定，默认会登录 Docker 官网的 Registry Server。

OPTIONS	Description
-e, --email=""	指定 email。
-p, --password=""	指定密码。
-u, --username=""	指定用户名。

举例：

```
[root@localhost ~]# docker login
Username: kulong0105
Password:
Email: kulong0105@gmail.com
Login Succeeded
[root@localhost ~]# docker logout
Remove login credentials for https://index.docker.io/v1/
[root@localhost ~]#
```

6.1.28 pull / push

语法：

Usage	Description
docker pull [OPTIONS] NAME[:TAG]	从 registry 中下载一个 image 或者 repository。

OPTIONS	Description
-a, --all-tags=false	下载 repository 中所有被 tagged 的 images。

Usage	Description
docker push NAME[:TAG]	上传一个 image 或者 repository 到 registry 中。

举例：

```
[root@localhost ~]# docker images
REPOSITORY          TAG         IMAGE ID            CREATED             VIRTUAL SIZE
kulong0107          latest      25e655be4072        38 hours ago        458.4 MB
kulong0105          latest      2157a84fbc40        6 days ago          458.4 MB
[root@localhost ~]# docker pull centos
centos:latest: The image you are pulling has been verified
34943839435d: Pull complete
34943839435d: Pulling fs layer
511136ea3c5a: Already exists
Status: Downloaded newer image for centos:latest
[root@localhost ~]# docker images
REPOSITORY          TAG         IMAGE ID            CREATED             VIRTUAL SIZE
kulong0107          latest      25e655be4072        38 hours ago        458.4 MB
kulong0105          latest      2157a84fbc40        6 days ago          458.4 MB
centos              latest      34943839435d        4 weeks ago         224 MB
[root@localhost ~]# docker tag centos   kulong0105/centos_test
[root@localhost ~]# docker images
REPOSITORY                TAG         IMAGE ID            CREATED             VIRTUAL SIZE
kulong0107                latest      25e655be4072        38 hours ago        458.4 MB
kulong0105                latest      2157a84fbc40        6 days ago          458.4 MB
kulong0105/centos_test    latest      34943839435d        4 weeks ago         224 MB
centos                    latest      34943839435d        4 weeks ago         224 MB
[root@localhost ~]# docker push kulong0105/centos_test
The push refers to a repository [kulong0105/centos_test] (len: 1)
Sending image list
Pushing repository kulong0105/centos_test (1 tags)
511136ea3c5a: Image already pushed, skipping
5b12ef8fd570: Image already pushed, skipping
34943839435d: Image already pushed, skipping
Pushing tag for rev [34943839435d] on {https://cdn-registry-
1.docker.io/v1/repositories/kulong0105/centos_test/tags/latest}
[root@localhost ~]#
```

注：

1) 命令“docker pull centos”默认会从 Docker 官网的 Docker Hub 中下载镜像，然而由于 GFW 原因，网络时常会断开，导致无法 pull 一个完整的 image。不过，国内不少好心人士，已把官方的镜像都同步到自己的网站并提供下载（但不提供上传服务），如：可以使用“docker pull docker.cn/docker/centos”下载官方的 centos 镜像。

2) 命令“docker pull -”默认会把镜像上传到 Docker 官网的 Docker Hub 中，因此需要先到官网上注册一个账户 username，并且 image 的格式必须是“username/repository:tag”。

6.1.29 search

语法：

Usage	Description
docker search [OPTIONS] TERM	在 Docker Hub 中搜索 image。

OPTIONS	Description
--automated=false	只显示 automated builds 的 image。
--no-trunc=false	不截断输出。
-s, --stars=0	只显示至少拥有指定的 star 数。

举例：

```
[root@localhost ~]# docker search ubunt
NAME                DESCRIPTION                     STARS     OFFICIAL   AUTOMATED
ubuntu              Official Ubuntu base image      1105      [OK]
dockerfile/ubuntu   Trusted automated Ubuntu        37                   [OK]
tutum/ubuntu        Ubuntu image with SSH access.   28                   [OK]
…
[root@localhost ~]# docker search kulong0105
NAME                                              DESCRIPTION   STARS     OFFICIAL   AUTOMATED
kulong0105/centos_gcc_net_sar_ssh_passwd                        0
kulong0105/centos_test                                          0
[root@localhost ~]#
```

注：

1) 一个 image 完整的名称是“username/repository:tag”，如果 username 没有写，则被认为是官方认证过的 image，如果 tag 没有写，则被认为 tag 是 lastest。

2) 命令“docker search”在搜索 image 时，是通过 Docker 中的 Index 进行搜索的，Index 主要存储的用户信息、image 的 checksum 信息等。

6.2 Registry

Docker 官方提供了 Docker Hub 作为一个公共的仓库。然后，本地访问 Docker Hub 速度往往很慢，加上 GFW 原因使得 push/pull 一个 image 变的很困难。为此，本节介绍如何搭建一个私有仓库供内网使用。

6.2.1 install

Docker 官方提供了一个简易的方法，使用“docker-registry”项目来实现内网搭建一个私有仓库。安装“docker-registry”项目有如下两种方法：

1) 直接执行运行命令“docker run -p 5000:5000 registry”，这样 Docker 会自动从 Docker Hub 中拉取 registry 镜像。

2) 从 github 网站上下载 docker-registry 项目，然后通过 Dockerfile 来构建镜像（使用命令“docker build”来拉取 registry 镜像），如下所示：

```
#git clone https://github.com/docker/docker-registry.git
#cd docker-registry
#docker build -t registry
```

6.2.2 deploy

在拉取好 registry 镜像后，运行命令“docker run -p 5000:5000 registry”会启动一个 registry 服务器，但是所有上传的镜像都是由 docker 容器管理，存放在容器的/var/lib/docker/*目录下。一旦删除容器，镜像也会被删除。因此，需要把镜像放在 Host 上，有如下两种方法：

1) 直接运行如下命令：

```
# docker run -d -e SETTINGS_FLAVOR=dev -e STORAGE_PATH=/tmp/registry -v
/home/renyl/registry:/tmp/registry  -p 5000:5000 registry
```

说明：

a) 环境变量“SETTINGS_FLAVOR”设置镜像存储的后端

b) 环境变量“STORAGE_PATH”设置镜像存储在容器中的目录。

c) 选项“-v /home/renyl/registry:/tmp/registry ”设置 Container 与 Host 的共享目录。

2) 从“docker-registry”项目中拷贝文件 config_sample.yml 到 Host 上，然后修改配置文件中的参数选项，最后设置环境变量“DOCKER_REGISTRY_CONFIG”来启动 container。执行命令步骤如下所示：

```
#cp /docker-registry/config/config_sample.yml   /home/renyl/docker/my_config.yml
#vi /home/renyl/docker/my_config.yum   #可以修改相关设置
#docker run -d -e DOCKER_REGISTRY_CONFIG= /home/renyl/docker/my_config.yml -v
/home/renyl/registry:/tmp/registry  -p 5000:5000 registry
```

注：

关于“docker-registry”项目的详细信息以及相关配置可参考地址：https://github.com/docker/docker-registry/.

6.2.3 apply

针对私有仓库的应用，演示如下：

仓库端机器:

```
[root@localhost /]# docker -d &  &> /dev/null        #启动 docker 服务
[root@localhost /]# docker run -d -p 5000:5000 -v /tmp/registry:/tmp/registry
registry  >& /dev/null
[root@localhost /]# docker ps
CONTAINER ID     IMAGE             COMMAND               CREATED        STATUS
PORTS            NAMES
48d873a3b6fc     registry:latest   "/bin/sh -c 'exec do  3 seconds ago  Up 3 seconds
0.0.0.0:5000->5000/tcp   happy_cori
[root@localhost /]#
```

非仓库端机器:

```
[root@localhost /]# docker  --insecure-registry  193.168.249.121:5000  -d  &  >&
/dev/null
[root@localhost /]# docker images
REPOSITORY          TAG           IMAGE ID            CREATED             VIRTUAL SIZE
kulong0105          latest        2157a84fbc40        8 days ago          458.4 MB
[root@localhost /]# docker tag kulong0105  193.168.249.121:5000/myregistry/first_image
[root@localhost /]# docker images
REPOSITORY          TAG           IMAGE ID            CREATED             VIRTUAL SIZE
kulong0105          latest        2157a84fbc40        8 days ago          458.4 MB
193.168.249.121:5000/myregistry/first_image   latest              2157a84fbc40
8 days ago          458.4 MB
[root@localhost /]# docker push 193.168.249.121:5000/myregistry/first_image
The push refers to a repository [193.168.249.121:5000/myregistry/first_image] (len: 1)
Sending image list
Pushing repository 193.168.249.121:5000/myregistry/first_image (1 tags)
Image 511136ea3c5a already pushed, skipping
…
Pushing tag for rev [2157a84fbc40] on
{http://193.168.249.121:5000/v1/repositories/myregistry/first_image/tags/latest}
[root@localhost /]# docker search  193.168.249.121:5000/first
NAME                    DESCRIPTION   STARS     OFFICIAL   AUTOMATED
myregistry/first_image                0
[root@localhost ~]#
```

说明：

1) 由于 Docker 在新版本中针对私有仓库连接时需要进行 CA 认证，因此在“非仓库端机器”启动 docker 服务时需要使用参数选项“--insecure-registry 193.168.249.121:5000”来避开 CA 认证，其中：“193.168.249.121”为“仓库端机器”的 IP 地址，“5000”为“仓库端机器”registry 服务开放的端口号。

2) 在“非仓库端机器”执行命令“docker push”时需要注意，必须先使用命令“docker tag”把准备 push 到仓库的 image 给重命名，重命名后的格式应该是“IP：PORT/REPOSITORY：TAG”。

6.3 Orchestration

2014 年 12 月，Docker 发布首个 Orchestration ，由 Docker Machine、Docker Swarm 以及 Docker Composer 3 个组件组成，编排服务开放了原生的接口，保证应用的可移植性，以及对容器的集群管理。

除了 Docker 官方发布 Orchestration 之前，已有多个针对容器的集群管理系统，如：

1) Chef，Puppet，Jenkins 等传统的配置工具，可以很好对容器进行集群部署。

2) Twitter 的 Apache Mesos，一套资源管理调度集群系统，生产环境使用它可以实现应用集群。

3) CenturyLink 的 Panamax，一个容器集群管理工具，可以用于管理单机的 Docker 容器。

4) Google 的 Kubernetes，一个开源容器集群管理系统，其提供应用部署、维护、 扩展机制等功能，利用 Kubernetes 能方便地管理跨机器运行容器化的应用，其主要功能如下：
   a) 使用 Docker 对应用程序包装(package)、实例化(instantiate)、运行(run)。
   b) 以集群的方式运行、管理跨机器的容器。
   c) 解决 Docker 跨机器容器之间的通讯问题。
   d) Kubernetes 的自我修复机制使得容器集群总是运行在用户期望的状态。

   注：
   1) 当前 Kubernetes 支持 GCE、vShpere、CoreOS、OpenShift、Azure 等平台，也可以直接运行在物理机上。
   2) 关于 Kubernetes 的基本原理和使用，可参考地址：
      Kuberneteshttp://dockerpool.com/article/1419409920 .

7 Dockerfile 制作

1) Dockerfile 文件由一系列指令组成，使用其可以自动构建 image 文件。

2) 使用命令“docker build .”构建 image 文件时，Docker 会逐个读取每个指令并执行，其中需要注意的是：
   a) 每条指令是被独立运行的。
   b) 每条指令执行后，都会创建一个新的 image 文件，新的 image 文件将作为 base-image 继续执行后续的指令。

3) Dockerfile 文件的命令格式为“INSTRUCTION arguments”, INSTRUCTION 列表如下所示：

ID	INSTRUCTION	DESCRIPTION
1	FROM	指定基本镜像(base-image)。
2	MAINTAINER	指定作者的姓名和联系方式。
3	RUN	在基本镜像里面执行命令并提交结果。
4	CMD	启动 Container 没有指定命令时会执行 CMD 指定的命令。
5	EXPOSE	Container 运行时 listen 的端口。
6	ENV	设置环境变量。
7	COPY & ADD	复制文件。
8	ENTRYPOINT	Container 启动时执行的命令。
9	VOLUME	创建一个挂载点用于共享目录。
10	USER	设置用户名。
11	WORKDIR	设置工作目录。
12	ONBUILD	设置一个动作在以该 image 为基本镜像时被触发。

说明：
   a) 指令大小写都可以，但是建议使用大写。
   b) 注释内容使用符号#开头。

7.1 Command

7.1.1 FROM

语法：

```
FROM  <image>:[tag]
```

举例：

```
FROM  centos
```

说明：

1) FROM 指令指定了基本镜像（base-image）。因该镜像为后续的指令所使用，所以该指令必须为 Dockerfile 文件的第一个指令。

2) FROM 指令如果没有指定 tag，那么默认的 tag 就是 latest。

3) FROM 指令能够多次出现在一个 Dockerfile 文件中，这样就可以创建多个 images。

7.1.2 MAINTAINER

语法：

```
MAINTAINER  <name>
```

举例：

```
MAINTAINER  Allen Ren <kulong0105@gmail.com>
```

说明:

MAINTAINER 指令用来指定作者的姓名和联系方式。

7.1.3 RUN

语法：

```
RUN  <command>  #(the command is run in a shell - /bin/sh -c  - shell from)
```

或者

```
RUN  [“executable”, “param1”, “param2”, …]   #(exec from)
```

举例：

```
RUN  echo hello
```

或者

```
RUN  [“/bin/bash”, “-c”, “echo hello”]
```

说明：

1) RUN 指令会在 base-image 里执行命令，然后提交结果，新生成的 image 会作为 base-image 供后续的命令使用。类似于执行如下命令：

```
docker run  image command
docker commit container_id
```

2) 为了不使用/bin/sh 作为 shell，而使用/bin/bash 作为 shell，可以使用“exec form”模式来完成，如:RUN [“/bin/bash”, “-c”, “echo hello”]。

3) 在“exec form”模式下必须使用双引号来，不能使用单引号，因为该模式下会被解析为 JSON array。

4) 在“exce form”模式下，不会唤醒 一个 command shell，这使得一些常规的 shell 处理将不会发生，如：RUN [“echo”, “$HOME”]将不会对$HOME 进行解析，除非你指定使用一个 shell 去执行才会进行处理，如：RUN [“sh”, “-c”, “echo”, “$HOME”]。

7.1.4 CMD

语法：

```
CMD [” executable”,“param1”,“param2”, ...]#(exec from, this is the preferred form)
```

或者

```
CMD [“param1”, “param2”] #(as default parameters to ENTRYPOINT)
```

或者

```
CMD command param1 param2  #(shell from)
```

举例：

```
CMD [“/bin/bash”, “-c”, “echo hello”]
```

或者

```
CMD [“-a”, “-l”]
```

或者

```
CMD echo hello
```

说明：

1) 当使用 docker run image 启动一个 Container 时不指定任何命令时，且没有使用 ENTRYPOINT 指令，那么这个时候就会执行指令 CMD 所指定的命令。

2) Dockerfile 文件中只能有一个 CMD 指令，如果出现多个，只有最后一个生效。

7.1.5 EXPOSE

语法：

```
EXPOSE <port> [<port> ...]
```

举例：

```
EXPOSE 12306 12345
```

说明：

1) EXPOSE 指令告知 Docker 在 Container 运行时会 listen 指定的网络端口号。

2) EXPOSE 指令并不会 expose 其端口号给 Host，需要使用命令 docker run -p/-P 建立端口映射才能 expose 其端口号。

7.1.6 ENV

语法：

```
ENV  <key>  <value>
```

或者

```
ENV  <key> =<value> ...
```

举例：

```
ENV  myName  Allen  Ren
ENV  myLove  Lucky  Xiong
```

或者

```
ENV  myName=“Allen\ Ren”  myLove=Lucky\ Xiong
```

说明：

1) ENV 指令用于设置环境变量，该环境变量可被用在后续的 RUN 指令中。

2) “ENV <key> <value>”这种模式一次只能设置一个环境变量，<key>第一个空格之后都被作为<value>,包括空格和引号。

3) “ENV <key> =<value> …”这种模式一次可设置多个环境变量,可通过反引号来设置一些特殊字符（如，空格、双引号、单引号）。

4) 通过 ENV 指令设置的环境变量,可以通过命令 docker inspect <image>来查看。当 Container 以该 image 启动时，设置的环境变量将继续在 Container 中生效，除非使用命令 docker run --env <key>=<value> 进行修改。

注：Docker 默认会提供环境变量 PATH。

7.1.7 COPY & ADD

语法：

```
COPY  <src> ... <dest>
ADD   <src> ... <dest>
```

举例：

```
COPY  hom*  et*  /mydir/          #add all files staring with “hom” and “et”
ADD   hom*  www.github.com/kulong0105/testfile  /mydir/
```

说明：

1) COPY 指令与 ADD 指令相同点：
   a) 都用于复制文件或目录从<src>到 Container 的<dest>目录。
   b) 都可以指定多个<src>，只能指定一个<dest>。
   c) <dest>都必须是 Container 里面的绝对路径。
   d) 复制到 Container 中的新文件或目录的 UID 和 GID 都为 0。
   e) <src>如果是文件或者目录，那么<src>都必须是源目录的相对路径。（如，通过命令 docker build github.com/creack/docker-firefox 命令创建 image 时，<src>所指定的源目录就是 docker-firefox 目录，该目录也称为 context directory）
   f) <src>的路径必须在 context directory 的内部，即不能有“../something/something”这样的形式。
   g) <src>如果是一个目录，那么目录下所有的内容都会被拷贝，包括文件系统元数据，不过目录本身并不会被拷贝。
   h) 如果指定了多个<src>,那么<dest>必须以正斜杠/结尾表明其是个目录。如果 dest 目录不存在，将会自动在 Container 中创建。
   i) 如果<dest>没有以正斜杠/结尾表明其是个常规文件，那么<src>将覆盖写数据到<dest>。

2) COPY 指令与 ADD 指令不同点：
   a) ADD 指令中<src>可为一个 URL，而 COPY 指令中<src>不可为一个 URL。因此，当使用 STDIN（如，docker build - <somefile）来创建 image 时，由于不会有 build context 使得不能使用 COPY 指令，但是仍然可使用 ADD 指令（src 必须为一个 URL）。
   b) ADD 指令中<src>若为一个压缩文件（如，gzip、bzip2）会自动进行解压缩，而 COPY 指令不会自动进行解压缩。同时，ADD 指令对来自 URL 中的压缩文件也不会自动解压。

7.1.8 ENTRYPOINT

语法：

```
ENTRYPOINT [”executable”,“param1”,“param2”, …] #(the preferred exec from)
```

或者

```
ENTRYPOINT command param1 param2 #(shell form)
```

举例：

```
ENTRYPOINT [“/usr/bin/ls”, /home/”]
```

或者

```
ENTRYPOINT ls /home
```

说明：

1) ENTRYPOINT 指令设置了容器启动时执行的命令。如果出现多个 ENTRYPOINT 指令，只有最后一个生效。

2) ENTRYPOINT 指令在“exec form”模式下，具有如下特点：
   a) 如果同时使用了 CMD 指令，那么 CMD 指令所指定的参数将会覆盖 ENTRYPOINT 指令中所有的参数。
   b) 如果使用了命令行参数（如，docker run base-image parameter），那么 CMD 指令将无效，命令行参数 parameter 将会被追加到 ENTRYPOINT 指令中参数的后面。

3) ENTRYPOINT 指令在“shell form”模式下，具有如下特点：
   a) CMD 指令参数和命令行参数不会被使用。
   b) 由于使用了 shell form 模式，那么 ENTRYPOINT 指令所指定的可执行程序（如，Mytest）将会作为“/bin/sh -c”的一个子程序运行。那么程序 Mytest 的 PID 将不可能为 1，这样的话，使用 docker stop <Container>去关闭 Container 的时候，程序 Mytest 将接受不到 SIGTERM 信号，Docker 会在 timeout 时发送一个 SIGKILL 信号关闭 Container。

4) 可以使用命令 docker run 的参数- -entrypoint 来重新指定容器启动时执行的命令，不过—entrypoint 只能指定一个二进制文件，不可以包含参数。

7.1.9 VOLUME

语法：

```
VOLUME ["/data"]
```

举例：

```
VOLUME ["/var/log"]
```

说明：
VOLUME 指令用于创建一个挂载点用于 Container 之间共享目录。与命令“docker run”中的参数选项--volume 不同的是，该命令只能用于共享 Container 中的目录，无法共享 Host 上的目录。

7.1.10 USER

语法：

```
USER <name | uid>
```

举例：

```
USER mysql
```

说明：
USER 指令用于设置用户名或者 UID，并以该用户运行 RUN 指令、CMD 指令、ENTRYPOINT 指令。

7.1.11 WORKDIR

语法：

```
WORKDIR <path>
```

举例：

```
WORKDIR /home/
```

说明：
1) WORKDIR 指令用来设置 RUN 指令、CMD 指令、ENTRYPOINT 指令的当前工作目录。

2) WORKDIR 指令能够在 Dockerfile 中多次使用，要想使用变量的话，只能使用 ENV 指令设置环境变量，如下：

```
ENV  DIRPATH  /home/
WORKDIR  $DIRPATH
WORKDIR  renyl
RUN pwd
```

结果：pwd 命令的输出为“/home/renyl”。

3) 如果不通过 USER 指令指定用户，默认用户为 root。

7.1.12 ONBUILD

语法：

```
ONBUILD  [INSTRUCTION]
```

举例：

```
ONBUILD  ADD  ./testfile   /testdir/
ONBUILD  WORKDIR  /home/
ONBUILD  RUN   ls  -al
```

说明：

1) ONBUILD 指令在本次使用 Dockerfile 创建 image（假设 image 名为 parent_image）时并不会执行，而在以 child_image 为基础镜像创建一个新的镜像时，ONBUILD 指令才会被触发。举例如下：

Dockerfile（第一个）	Dockerfile(第二个)
FROM centos USER root ENV MY_PATH /home/renyl ONBUILD ADD ./hello.c /home/renyl/ RUN ls -al $MY_PAT	FROM parent_image USER root ENV MY_PATH /home/renyl RUN ls -al $MY_PAT

在第一个 Dockerfile 文件所在目录下使用命令“docker build -t parent_image ./”创建 parent_image 镜像（使用选项-t 指定），但是这个时候/home/renyl/目录下不会有 hello.c 文件。

在第二个 Dockerfile 文件所在目录下使用命令“docker build -t child_image ./”创建 child_image 镜像，此时由于会触发第一个 Dockerfile 中的 ONBUILD 指令，这样生成的 child_image 镜像的/home/renyl/目录下存在 hello.c 文件。

2) 使用 ONBUILD 指令不能嵌套 ONBUILD 指令（如：ONBUILD ONBUILD），同时 ONBUILD 指令不能触发 FROM 指令和 MAINTAINER 指令。

7.2 Skill

Docker 官方针对用户制作 Dockerfile 提供了一些建议和技巧，具体如下：

1) 在制作 Dockerfile 文件时，应该避免安装一些非必须安装的包，这样可以减少 image 文件的创建时间和大小。

2) 在制作 Dockerfile 文件时，应该考虑每个 Container 最好只运行一个 APP，这样便于 Container 的重用和水平扩展。

3) 在制作 Dockerfile 文件时，应该在 Dockerfile 的可读性和 image lays 之间做个平衡，尽量使 image lays 数量降到最低。（每执行一条指令，就会在 base-image 上加一层）

4) 在制作 Dockerfile 文件时，尽量使用 COPY 指令，复制的文件如果需要 URL 指定可以使用 wget 或者 curl，除非复制的文件需要自动解压缩才使用 ADD 指令。

5) 在制作 Dockerfile 文件时，在运行多参数时，为了便于阅读，最好按字母进行分类和用多行表示。

   示例如下：

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

6) 使用 Dockerfile 构建 image 文件时，Docker 会读取其每条指令并执行（每条执行都会创建一个新的 image 文件），在执行每条指令时，Docker 会首先在其 Cache 中检查是否存在即将创建的 image 文件，如果存在的话就重用它，不存在的话就重新创建。

   示例如下：
   （使用同一 Dockerfile 文件构建，第二次会利用 Cache 中的 image，不重新构建 image）

```
[root@localhost testdir]# docker build -t kulong0106 .
Sending build context to Docker daemon 5.632 kB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
Step 1 : ENV MYNAME Allen Ren
 ---> Running in 1f5eb1927c05
 ---> a2eba81bdad2
Removing intermediate container 1f5eb1927c05
Step 2 : RUN echo $MYNAME
 ---> Running in 4dce44f96fdd
Allen Ren
 ---> ff6a6c1186a1
Removing intermediate container 4dce44f96fdd
Successfully built ff6a6c1186a1
```

```
[root@localhost testdir]# docker build -t kulong0107 .
Sending build context to Docker daemon 5.632 kB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
Step 1 : ENV MYNAME Allen Ren
 ---> Using cache
 ---> a2eba81bdad2
Step 2 : RUN echo $MYNAME
 ---> Using cache
 ---> ff6a6c1186a1
Successfully built ff6a6c1186a1
[root@localhost testdir]# docker  images
REPOSITORY        TAG          IMAGE ID        CREATED          VIRTUAL SIZE
kulong0107        latest       ff6a6c1186a1    14 seconds ago       458.4 MB
kulong0106        latest       ff6a6c1186a1    14 seconds ago       458.4 MB
kulong0105        latest       2157a84fbc40    3 hours              458.4 MB
[root@localhost testdir]#
```

注：如果不想使用 Cache，在使用命令“docker build”时添加参数选项“--no-cache”即可。

7) Docker 在对 Dockerfile 文件中的每条指令执行前进行 Cache 匹配时，仅仅通过每条指令（包括参数）是否相同来判断是否 Cache 命中，但是除了 ADD 指令和 COPY 指令，这两个指令还要对复制的文件内容进行判断，如果文件内容发生了改变，那么 Docker 会判断 Cache 匹配失败。

示例如下：

```
[root@localhost testdir]# ls
Dockerfile  test.txt
[root@localhost testdir]# cat Dockerfile
FROM kulong0105

RUN echo hello
ADD ./test.txt  /home/renyl/
RUN echo welcome

[root@localhost testdir2]# cat test.txt
hello world
[root@localhost testdir2]# docker build -t kulong0106 .
Sending build context to Docker daemon 4.608 kB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
Step 1 : RUN echo hello
 ---> Running in 7577200044ac
hello
 ---> 1579855c3ba3
Removing intermediate container 7577200044ac
```

```
Step 2 : ADD ./test.txt /home/renyl/
 ---> c6fd4dd8f95f
Removing intermediate container 159a6793c883
Step 3 : RUN echo welcome
 ---> Running in eabf074424fc
welcome
 ---> c0e85f8495e6
Removing intermediate container eabf074424fc
Successfully built c0e85f8495e6
[root@localhost testdir2]# echo "hello" > test.txt
[root@localhost testdir2]# docker build -t kulong0107 .
Sending build context to Docker daemon 4.608 kB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
Step 1 : RUN echo hello
 ---> Using cache
 ---> 1579855c3ba3
Step 2 : ADD ./test.txt /home/renyl/
 ---> b1b842b553bc
Removing intermediate container e313237778e5
Step 3 : RUN echo welcome
 ---> Running in 7efc144957a0
welcome
 ---> eef8018e9dad
Removing intermediate container 7efc144957a0
Successfully built eef8018e9dad
```

说明：

a) 指令“RUN yum -y update”虽然同样会对 Container 中的文件进行修改，但是 Docker 并不会针对 update 来检查 Container 中的文件是否发生了变化，仅仅通过指令和参数来判断是否 Cache 命中。因此在 Dockerfile 中不要把“RUN yum -y update”这样的命令单独放在一行，这样很可能会利用 Cache 中的 image 导致更新失败。

b) Dockerfile 中如果前一条指令 Cache 匹配失败，那么后面的指令都会匹配失败，因为每条指令构建的 image 文件都是在上条指令构建的 image 基础再次构建的。

c) 如果针对 Dockerfile 添加新指令构建 image 时，把新指令放在最后面，这样重新构建 image 时能够最大利用 Cache 从而加快 image 的构建。

8) 使用 Dockerfile 生成 image 时需要上传 build context 到 Docker daemon，为了加快上传速度以及命令“docker build”的执行效率，可以使用 .dockerfile 文件来过滤一些不必上传的文件，从而加速 image 的生产。

示例如下：

```
[root@localhost testdir]# ls -alh
total 40M
drwxr-xr-x  2 root root  79 Dec 24 04:43 .
drwxr-xr-x. 7 root root  83 Dec 23 04:26 ..
-rw-r--r--  1 root root 116 Dec 24 04:42 Dockerfile
-rw-r--r--  1 root root 40M Dec 24 03:44 unuseful.txt
-rw-r--r--  1 root root   9 Dec 24 04:24 useful.txt
[root@localhost testdir]# docker build -t kulong0106 .
Sending build context to Docker daemon 40.96 MB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
…
[root@localhost testdir]# echo unuseful.txt  > .dockerignore
[root@localhost testdir]# docker build -t kulong0107 .
Sending build context to Docker daemon 4.608 kB
Sending build context to Docker daemon
Step 0 : FROM kulong0105
 ---> 2157a84fbc40
…
[root@localhost testdir]#
```

7.3 Example

这里列出 Docker 官方项目 docker-registry 的 Dockerfile 让大家感受一下，如下：
（地址：https://github.com/docker/docker-registry/blob/master/Dockerfile）

```
# VERSION 0.1
# DOCKER-VERSION  0.7.3
# AUTHOR:         Sam Alba <sam@docker.com>
# DESCRIPTION:    Image with docker-registry project and dependecies
# TO_BUILD:       docker build -rm -t registry .
# TO_RUN:         docker run -p 5000:5000 registry

# Latest Ubuntu LTS
FROM ubuntu:14.04

# Update
RUN apt-get update \
# Install pip
    && apt-get install -y \
        swig \
        python-pip \
# Install deps for backports.lmza (python2 requires it)
        python-dev \
        libssl-dev \
        liblzma-dev \
        libevent1-dev \
    && rm -rf /var/lib/apt/lists/*

COPY . /docker-registry
COPY ./config/boto.cfg /etc/boto.cfg

# Install core
RUN pip install /docker-registry/depends/docker-registry-core

# Install registry
RUN pip install file:///docker-registry#egg=docker-registry[bugsnag,newrelic,cors]

RUN patch \
 $(python -c 'import boto; import os; print os.path.dirname(boto.__file__)')/connection.py
\
 < /docker-registry/contrib/boto_header_patch.diff

ENV DOCKER_REGISTRY_CONFIG /docker-registry/config/config_sample.yml
ENV SETTINGS_FLAVOR dev

EXPOSE 5000

CMD ["docker-registry"]
```

8 Docker 局限与未来

8.1 Limit

1) 容器技术在性能、部署、效率上相对于传统虚拟机都有着巨大优势，但基于容器技术的“云”目前还没有成为主流，其中很大一部分原因还是因安全性问题。

2) 传统虚拟机可以利用来自硬件的机制来提升安全性，如：虚拟机可以使用 ring-1 特权的硬件隔离技术（如 VT-d 和 VT-x）来防止虚拟机 breaking out，从而提高其安全性。

3) 容器采用了独特的设计，没有采用任何形式的硬隔离技术，仅仅从操作系统级别（使用 Cgroup、Namespaces 和 SELinux 技术）来对容器进行隔离。正是因为这种设计使得其性能性、效率上比传统虚拟机有着巨大优势，但是由于这种设计没有采集任何硬件技术进行隔离，也使得它容易受到漏洞的利用，无法保证其足够高的安全性。

4) 同时，从 Docker 最新发布的 1.4.0 版本中修复了大量安全相关的 Bug，也可以看出随着 Docker 的越来越流行其安全问题越来越来突出。

8.2 Future

1) Docker 公司已经建立了清晰的道路：发展核心能力（libcontainer）、跨业务管理（libswarm）和容器间消息（libchan）。

2) Docker 近期又发布了“跨容器的分布式应用编排服务”（由 Docker Machine、Docker Swarm 以及 Docker Composer 3 个组件组成）来帮助用户来在“云”上方便的管理 Container。

3) Docker 接下来将会针对企业用户推出“Docker Hub Enterprise”，使得 Docker 能够在 Firewall 之后进行运行，并且提供实时分析等功能。其商业化越来越接近。

4) Docker 目前已和各大 IT 公司（如：IBM、Google、Facebook、Microsoft、RedHat 等）进行合作，并且这些公司还为其贡献代码。

5) 值得注意的是，传统虚拟化技术的著名公司 VMware 也和 Docker 进行了合作，或许可以表明虚拟机和容器之间并非只有竞争，也可以有合作，如：可以在虚拟机中运行 Docker，可以在 Docker 中运行虚拟机，利用虚拟机来提高安全性，利用 Docker 提高性能。

6) Docker 才推出不到 2 年，以取得如此大的成功，而且社区依然异常活跃，版本更新速度很快，基本每个月发一个版本。由于 Docker 超前的设计理论，加上出色的性能、部署、效率等，相信接下来会发展的更好，让我们拭目以待。

9 参考地址

1) https://www.docker.com/

2) https://docker.cn/

3) http://dockerone.com/

4) http://www.infoq.com/

5) http://special.csdncms.csdn.net/BeDocker/

6) https://code.csdn.net/u010702509/docker/file/Docker.md