# 窥探 Docker

-----------------------------------------------------------------------------------------

renyl 2015/10/26

目录

[窥探 Docker 1](#_Toc477390091)

[目录 2](#_Toc477390092)

[1 Docker背景 4](#_Toc477390093)

[2 Docker介绍 5](#_Toc477390094)

[3 Docker原理 7](#_Toc477390095)

[3.1 Docker Feature 7](#_Toc477390096)

[3.2 Docker Image 8](#_Toc477390097)

[3.3 Docker Technology 9](#_Toc477390098)

[3.3.1 Namespaces 9](#_Toc477390099)

[3.3.2 Cgroup 10](#_Toc477390100)

[3.3.3 SELinux & AppArmor 10](#_Toc477390101)

[3.3.4 AUFS 11](#_Toc477390102)

[4 Docker架构 12](#_Toc477390103)

[4.1 Docker Architecture 12](#_Toc477390104)

[4.2 Docker Module 13](#_Toc477390105)

[4.2.1 Client 13](#_Toc477390106)

[4.2.2 Daemon 14](#_Toc477390107)

[4.2.3 Registry 16](#_Toc477390108)

[4.2.4 Graph 16](#_Toc477390109)

[4.2.5 Driver 17](#_Toc477390110)

[4.2.6 Libcontainer 19](#_Toc477390111)

[4.2.7 Container 20](#_Toc477390112)

[4.3 Docker Flow 20](#_Toc477390113)

[4.3.1 docker pull 20](#_Toc477390114)

[4.3.2 docker run 22](#_Toc477390115)

[5 Docker安装 24](#_Toc477390116)

[6 Docker管理 25](#_Toc477390117)

[6.1 Command 26](#_Toc477390118)

[6.1.1 info & version 30](#_Toc477390119)

[6.1.2 attach 30](#_Toc477390120)

[6.1.3 build 31](#_Toc477390121)

[6.1.4 commit 32](#_Toc477390122)

[6.1.5 cp 33](#_Toc477390123)

[6.1.6 create 33](#_Toc477390124)

[6.1.7 diff 34](#_Toc477390125)

[6.1.8 exec 34](#_Toc477390126)

[6.1.9 export / save 35](#_Toc477390127)

[6.1.10 images 35](#_Toc477390128)

[6.1.11 import / load 36](#_Toc477390129)

[6.1.12 inspect 37](#_Toc477390130)

[6.1.13 kill 37](#_Toc477390131)

[6.1.14 port 38](#_Toc477390132)

[6.1.15 pause / unpause 38](#_Toc477390133)

[6.1.16 ps 39](#_Toc477390134)

[6.1.17 rm 40](#_Toc477390135)

[6.1.18 rmi 40](#_Toc477390136)

[6.1.19 run 41](#_Toc477390137)

[6.1.20 start / stop / restart 45](#_Toc477390138)

[6.1.21 tag 47](#_Toc477390139)

[6.1.22 top 47](#_Toc477390140)

[6.1.23 wait 48](#_Toc477390141)

[6.1.24 events 48](#_Toc477390142)

[6.1.25 history 48](#_Toc477390143)

[6.1.26 logs 49](#_Toc477390144)

[6.1.27 login / logout 50](#_Toc477390145)

[6.1.28 pull / push 50](#_Toc477390146)

[6.1.29 search 52](#_Toc477390147)

[6.2 Registry 53](#_Toc477390148)

[6.2.1 install 53](#_Toc477390149)

[6.2.2 deploy 53](#_Toc477390150)

[6.2.3 apply 54](#_Toc477390151)

[6.3 Orchestration 55](#_Toc477390152)

[7 Dockerfile制作 56](#_Toc477390153)

[7.1 Command 57](#_Toc477390154)

[7.1.1 FROM 57](#_Toc477390155)

[7.1.2 MAINTAINER 57](#_Toc477390156)

[7.1.3 RUN 57](#_Toc477390157)

[7.1.4 CMD 58](#_Toc477390158)

[7.1.5 EXPOSE 58](#_Toc477390159)

[7.1.6 ENV 59](#_Toc477390160)

[7.1.7 COPY & ADD 59](#_Toc477390161)

[7.1.8 ENTRYPOINT 60](#_Toc477390162)

[7.1.9 VOLUME 61](#_Toc477390163)

[7.1.10 USER 61](#_Toc477390164)

[7.1.11 WORKDIR 61](#_Toc477390165)

[7.1.12 ONBUILD 62](#_Toc477390166)

[7.2 Skill 63](#_Toc477390167)

[7.3 Example 67](#_Toc477390168)

[8 Docker局限与未来 68](#_Toc477390169)

[8.1 Limit 68](#_Toc477390170)

[8.2 Future 68](#_Toc477390171)

[9 参考地址 69](#_Toc477390172)

# 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：
   1. Container与Host共用一个内核。
   2. Container相当于一个轻量级APP sandbox。
   3. Container作为一个普通进程在Host上运行。
   4. 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](javascript:void(0);) | 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



1. 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](http://www.docker.io/)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](https://github.com/dotcloud/docker)托管在 [GitHub](https://github.com/docker/docker) 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](http://selinuxproject.org/page/Main_Page)）来支撑容器之间的隔离。起初Docker只是[LXC](https://linuxcontainers.org/)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](https://github.com/docker/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：
5. 对应用进行自动打包和部署(Automating the packaging and deployment of applications)
6. 创建轻量、私有的PaaS环境(Creation of lightweight, private PaaS environments)
7. 自动化测试和持续整合与部署(Automated testing and continuous integration/deployment)
8. 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)
9. Docker的发展过程：
10. 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。

1. 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。

1. 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。
2. Docker的发展催生了新的Linux发行版，如：[CoreOS](https://coreos.com/)和[Project Atomic](http://www.projectatomic.io/)（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。
3. 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。
4. 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。
5. 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。
6. 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。
7. 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。
8. 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](javascript:void(0);) | Low | Low | High |
| 5 | Complexity | Low | Low | High |
| 6 | Workloads | Low | Low | High |
| 7 | Share | Yes | No | No |
| 8 | [Portability](file:///C:\Program%20Files\Youdao\Dict4\5.4.43.3217\resultui\app:ds: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：

1. 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。
2. 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Docke](http://server.cnw.com.cn/server-os/htm2014/20140616_303249.shtml)r。

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：

VM系统架构 Docker系统架构



## 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。

1. 每个Image都依赖一个或多个parent images，唯一没有parent image的就是Platform image。
2. 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。

1. 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。

1. 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。

1. 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。

1. 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。

1. 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：

1. 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>

1. 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。

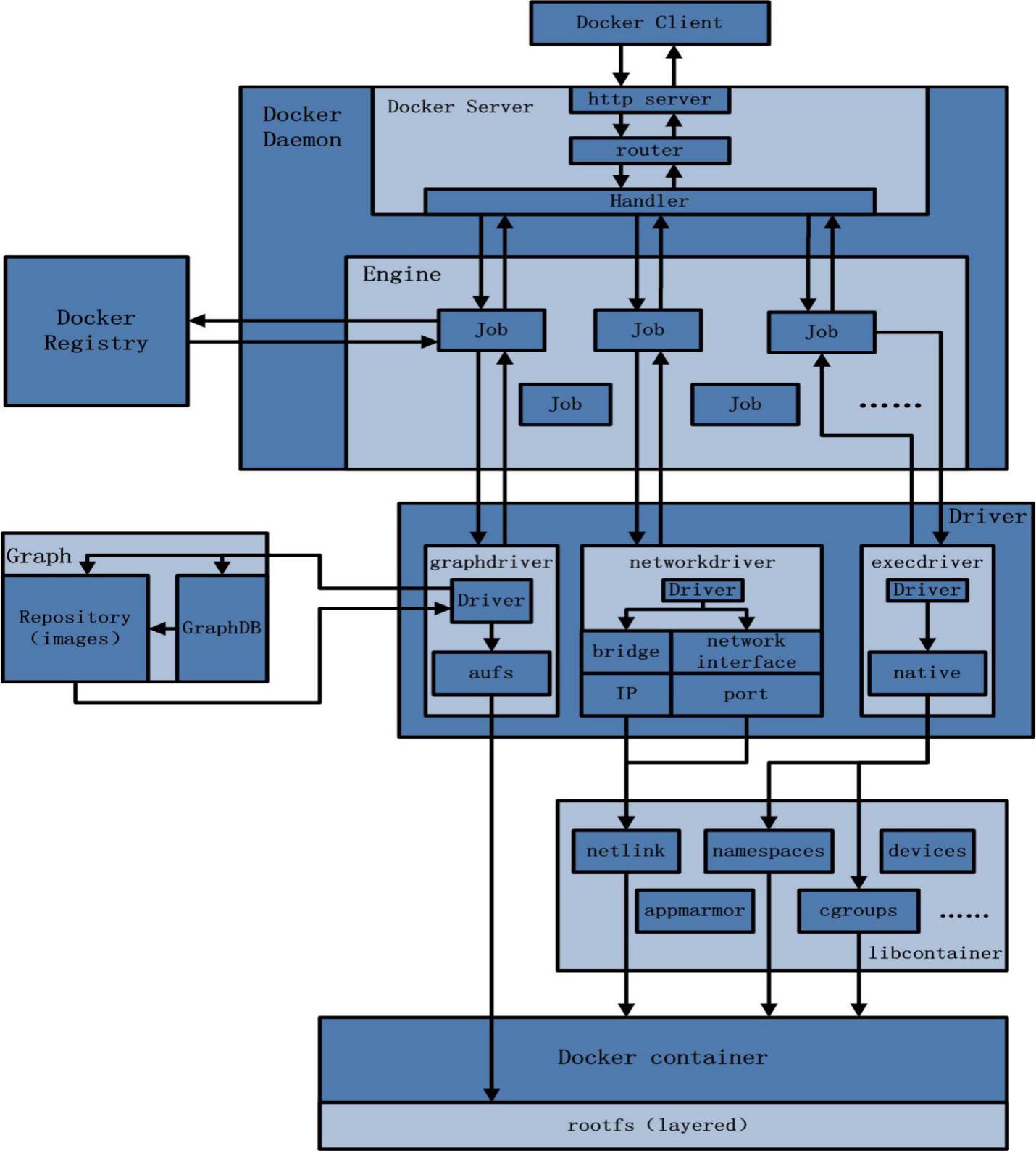
针对AUFS的详细说明如下：

1. Union FS主要有两个用途：
   1. 可以不借助LVM、RAID机制将多个Disk挂载到同一目录下。
   2. 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：

1. 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。
2. 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。

1. 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。
2. 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。
3. 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

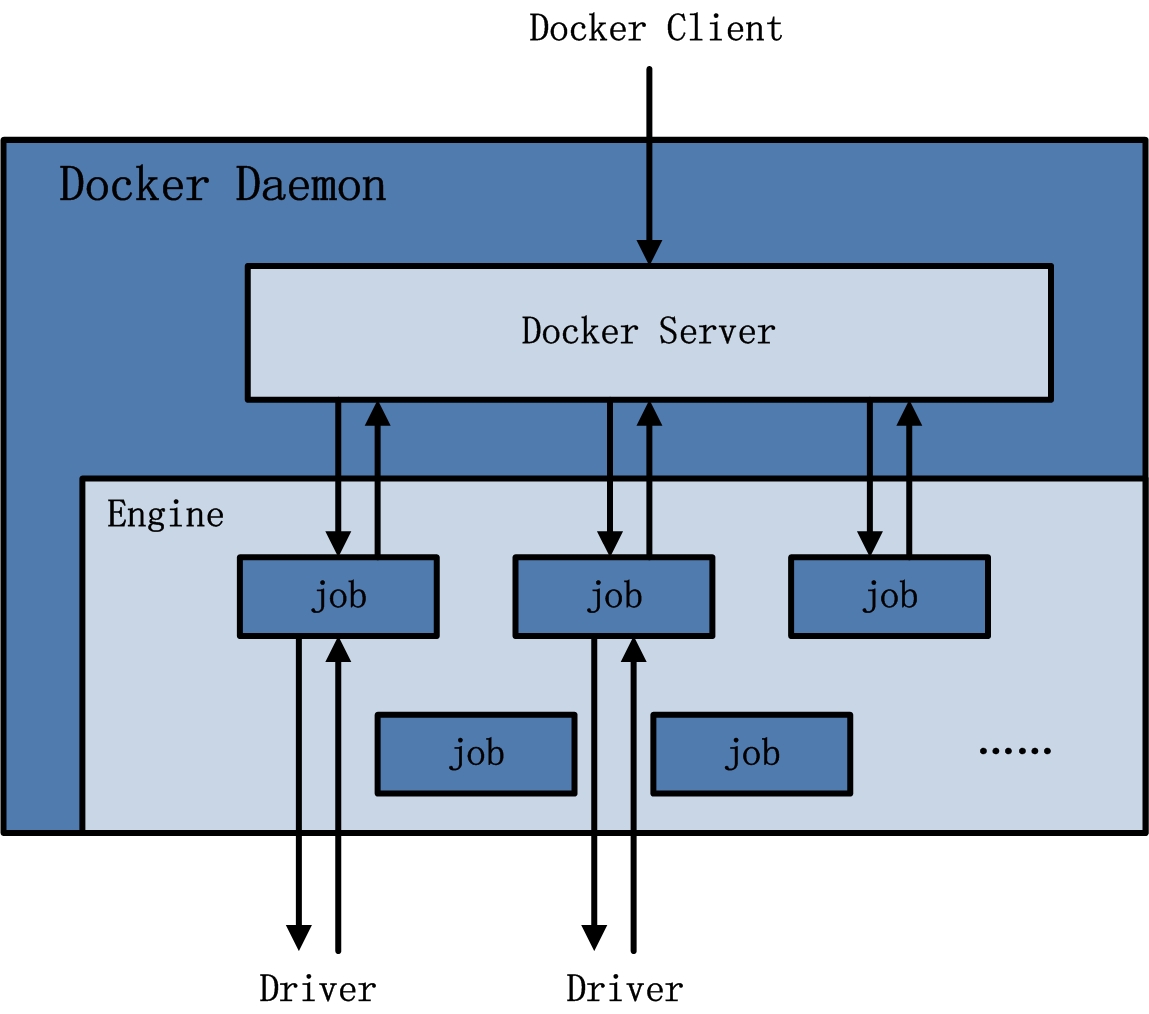
Docker架构主要由7个模块组成：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer以及Docker container。接下来，对各个模块进行详细的介绍。

### 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：
3. tcp://host:port
4. unix://path\_to\_socket
5. fd://socketfd
6. 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。
7. 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。

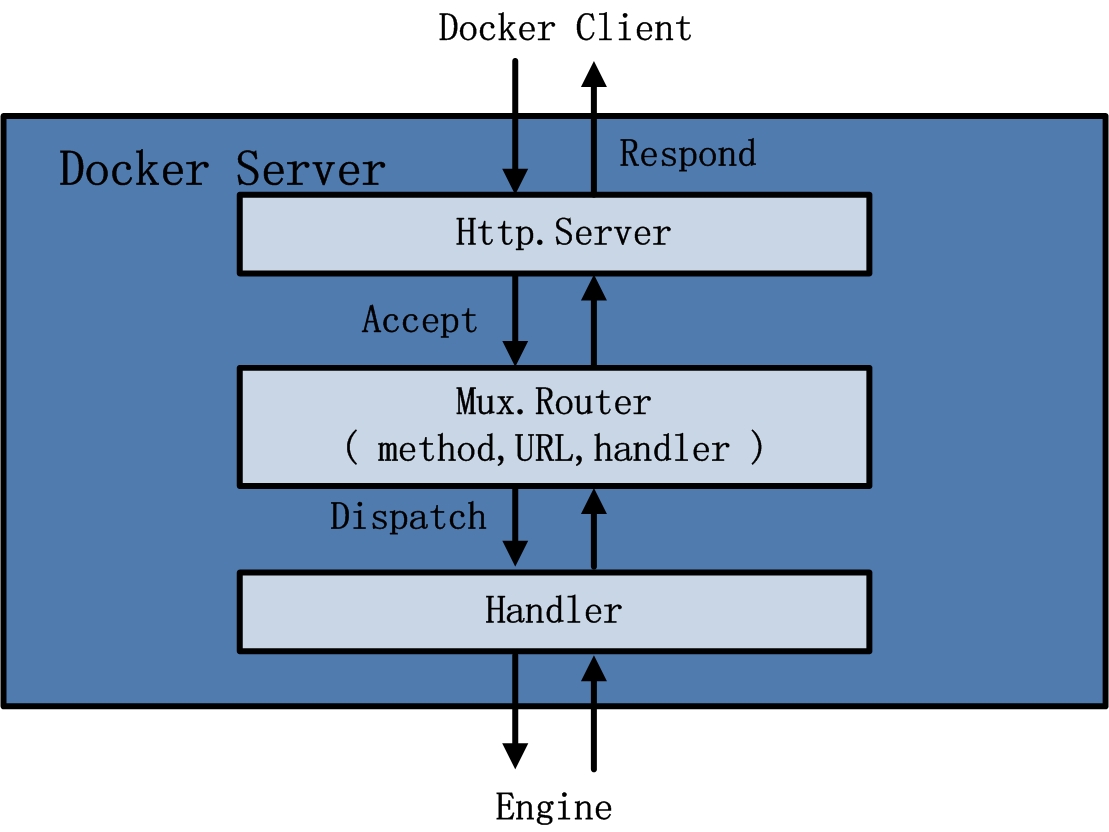
### 4.2.2 Daemon

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：
3. Docker Daemon在后台启动一个Server，Server负责接受Docker Client发送的请求。
4. 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。
5. 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

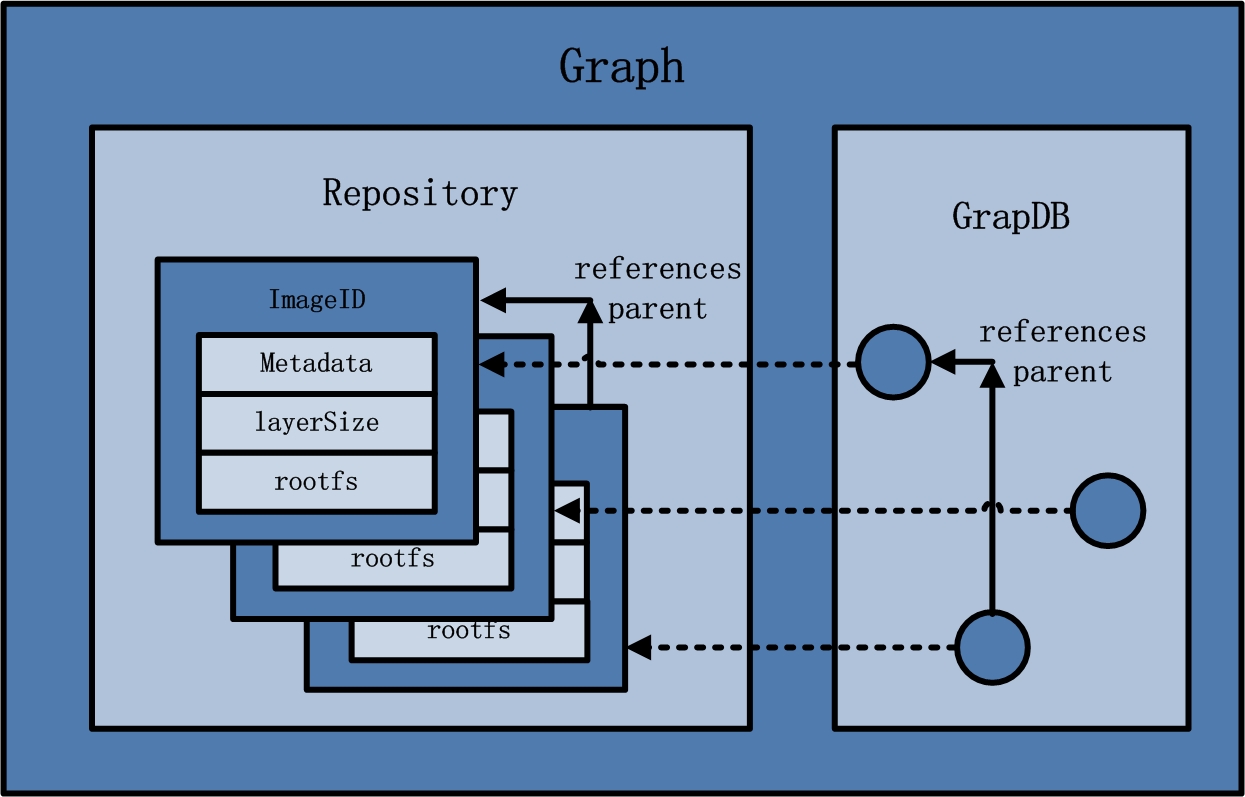
1. Engine是Docker架构中的运行引擎，同时也Docker运行的核心模块。它扮演Docker Container存储仓库的角色，并且通过执行job的方式来操纵管理这些容器。
2. 在Engine数据结构的设计与实现过程中，有一个handler对象，该handler对象存储的都是关于众多特定Job的handler处理访问。如：Engine的handler对象中有一项为：{"create": daemon.ContainerCreate,}，则当名为"create"的Job在运行时，执行的是daemon.ContainerCreate的handler。
3. Job是Engine内部最基本的工作执行单元，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。

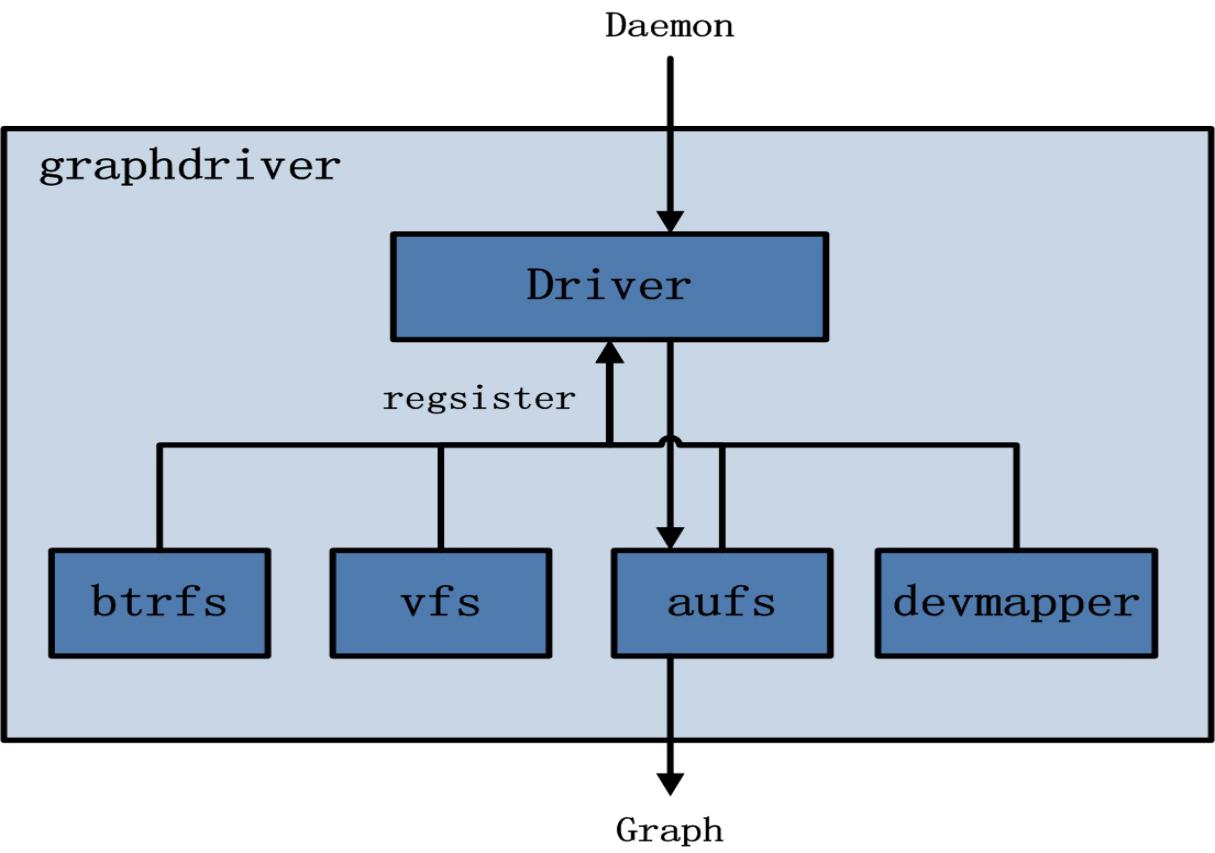
### 4.2.5 Driver

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：

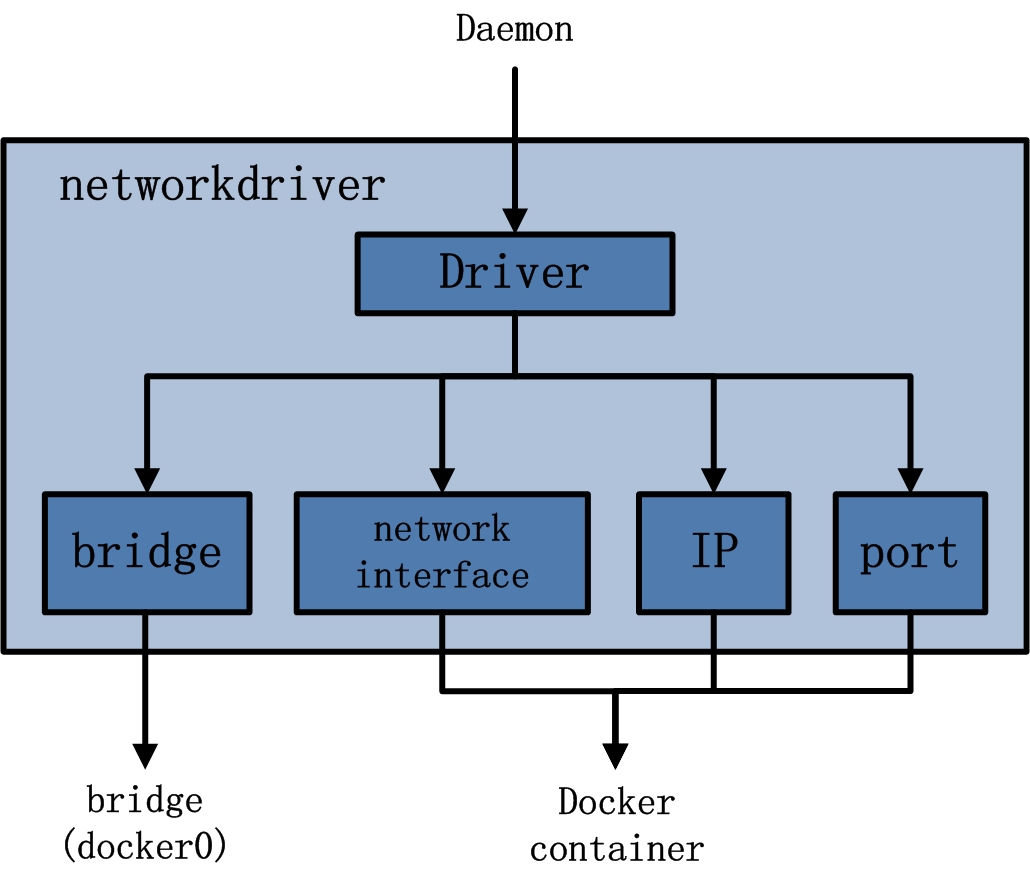


#### 4.2.5.2 network driver

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。

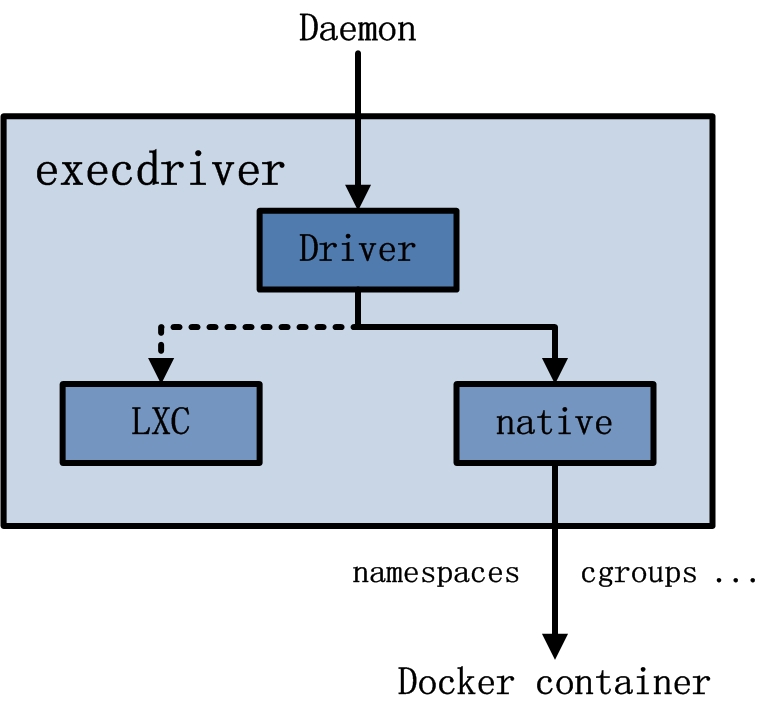
network driver的架构图如下所示：



#### 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"。

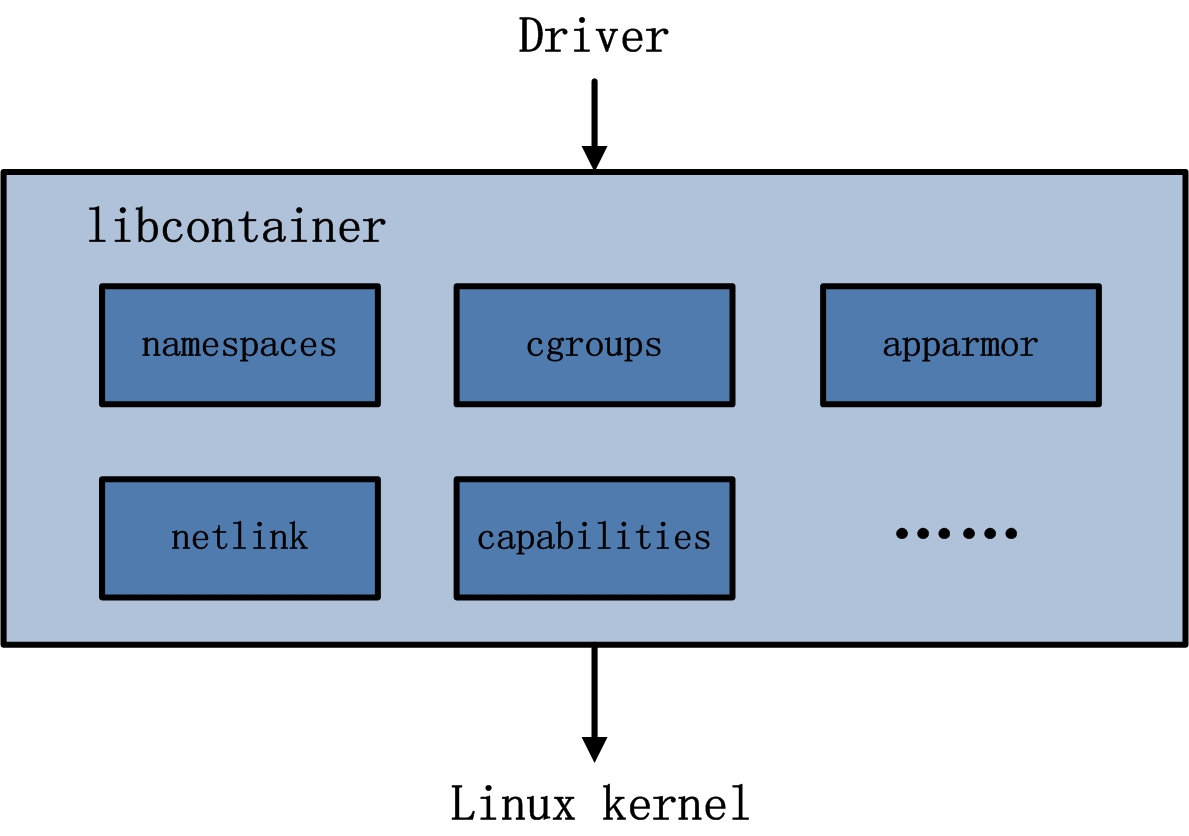
exec driver架构图如下所示：



### 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。

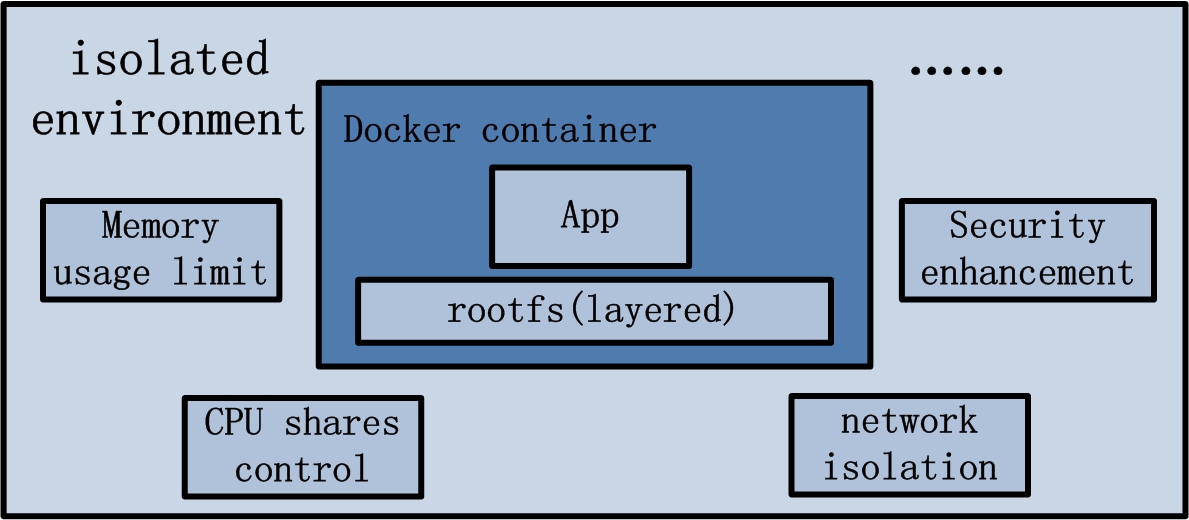
libcontainer架构图如下所示：



### 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：
3. 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；
4. 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；
5. 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；
6. 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：



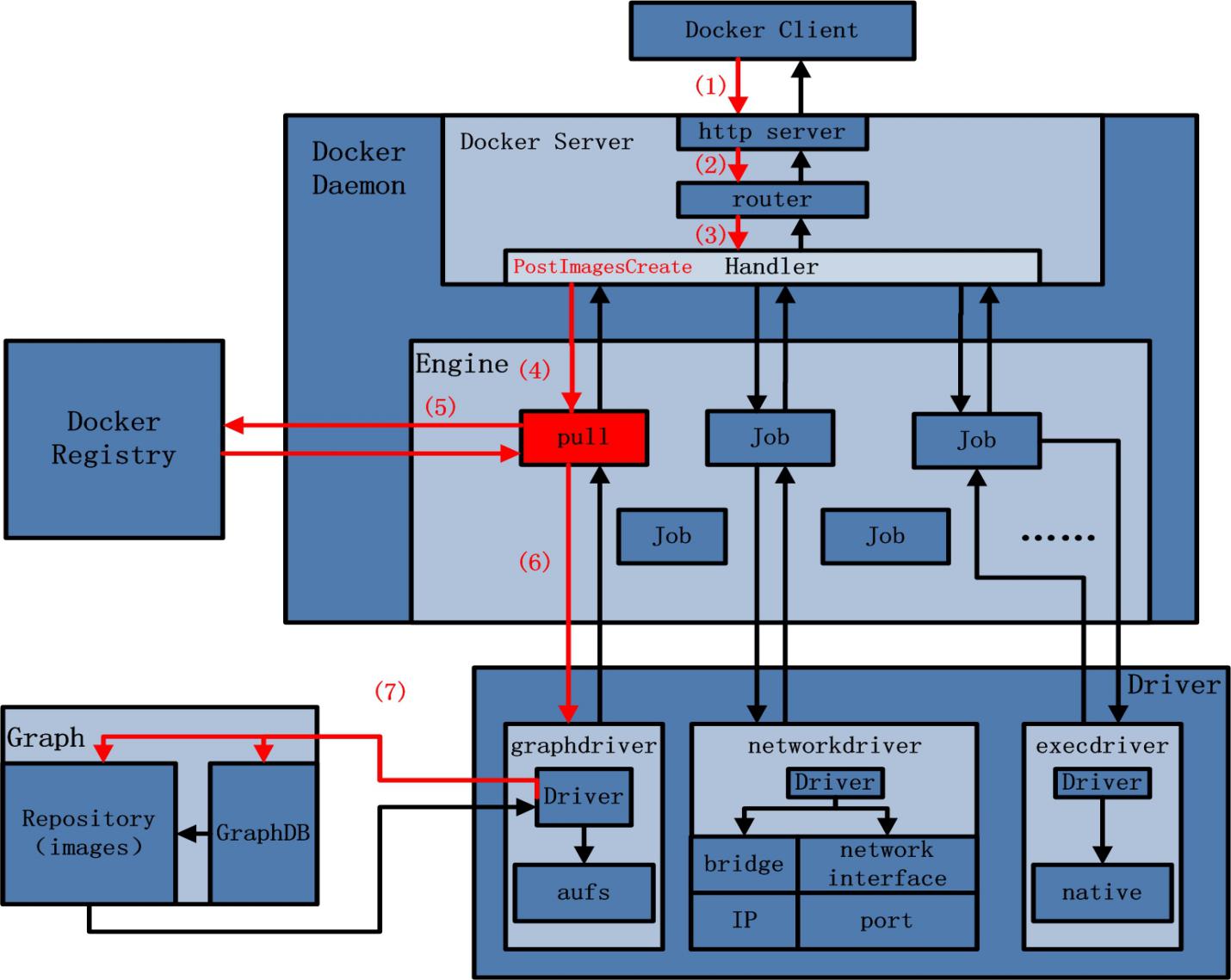
## 4.3 Docker Flow

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：



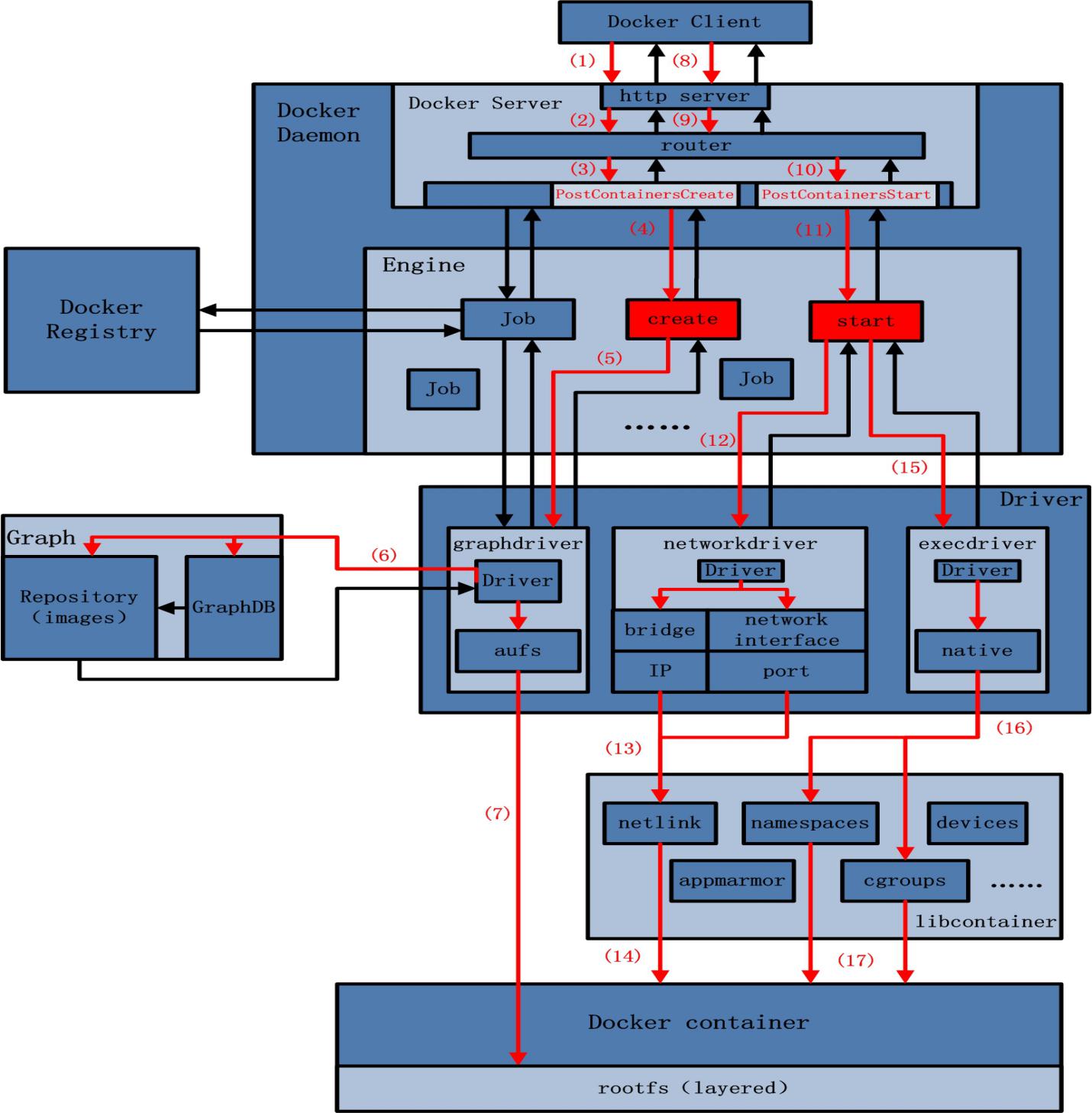
步骤：（图中标记的红色箭头表示docker pull命令在发起后所做的一系列操作）

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：
2. 创建Docker容器所需的rootfs以及网络等运行环境。
3. 运行用户所指定的指令。
4. 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：

1. 使用Red Hat的yum源需注册订阅（付费）。
2. 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/>

1. 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>

1. 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=>

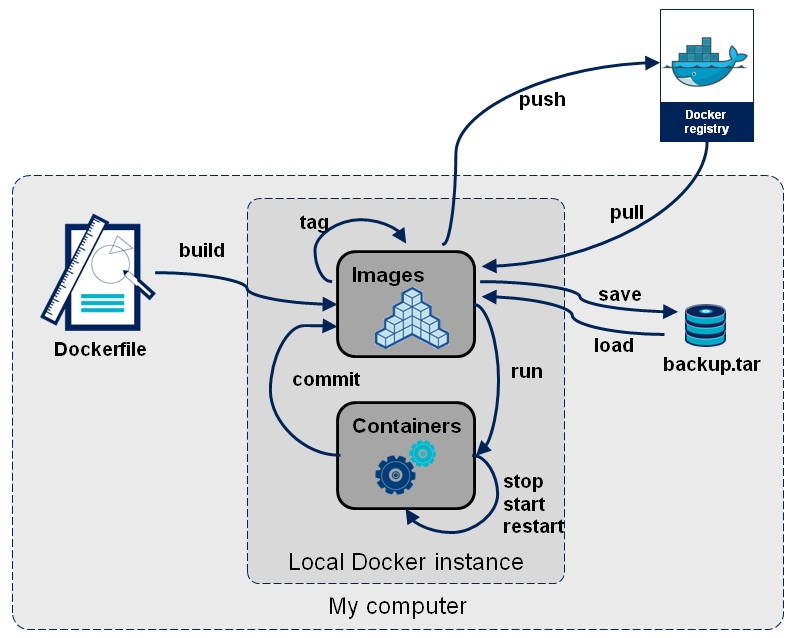
1. 源码安装

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：



1. 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 |

1. 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 |

1. 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：

1. 环境变量“SETTINGS\_FLAVOR”设置镜像存储的后端
2. 环境变量“STORAGE\_PATH”设置镜像存储在容器中的目录。
3. 选项“-v /home/renyl/registry:/tmp/registry ”设置Container与Host的共享目录。
4. 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：
5. 使用Docker对应用程序包装(package)、实例化(instantiate)、运行(run)。
6. 以集群的方式运行、管理跨机器的容器。
7. 解决Docker跨机器容器之间的通讯问题。
8. 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：
3. 每条指令是被独立运行的。
4. 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。
5. 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：

1. 指令大小写都可以，但是建议使用大写。
2. 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](mailto:kulong0105@gmail.com) 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 |

1. 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”]。
2. 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。
3. 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：
   1. 都用于复制文件或目录从<src>到Container的<dest>目录。
   2. 都可以指定多个<src>，只能指定一个<dest>。
   3. <dest>都必须是Container里面的绝对路径。
   4. 复制到Container中的新文件或目录的UID和GID都为0。
   5. <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）
   6. <src>的路径必须在context directory的内部，即不能有“../something/something”这样的形式。
   7. <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。
   8. 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。
   9. 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：
   1. 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）。
   2. 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：
   1. 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。
   2. 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：
   1. CMD指令参数和命令行参数不会被使用。
   2. 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”。

1. 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。

1. 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 |

1. 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。

1. 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：

1. 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。
2. 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。
3. 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。
4. 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/](http://www.infoq.com/cn/dockers/?utm_source=infoq&utm_medium=header_graybar&utm_campaign=topic_clk)
5. <http://special.csdncms.csdn.net/BeDocker/>
6. <https://code.csdn.net/u010702509/docker/file/Docker.md>