**1 启动docker服务**

**首先需要知道启动docker服务是：**

**service docker start**

**1**

**或者：**

**systemctl start docker**

**1**

**2 关闭docker服务**

**关闭docker服务是：**

**service docker stop**

**1**

**或者：**

**systemctl stop docker**

**1**

**3 启动docker某个image（镜像）的container（容器）**

**Docker的镜像称为image，容器称为container。**

**对于Docker来说，image是静态的，类似于操作系统快照，而container则是动态的，是image的运行实例。**

**比如，有一个image名称为ubuntu，那么比如现在我们启动这个image的container并且进入到这个container的bash命令行中：**

**docker run -t -i ubuntu /bin/bash**

**1**

**官网是这么说的：**

**docker run: runs a container.**

**ubuntu: is the image you would like to run.**

**-t: flag assigns a pseudo-tty or terminal inside the new container.**

**-i: flag allows you to make an interactive connection by grabbing the standard in (STDIN) of the container.**

**/bin/bash: launches a Bash shell inside our container.**

**理解很简单：**

**docker run：启动container**

**ubuntu：你想要启动的image**

**-t：进入终端**

**-i：获得一个交互式的连接，通过获取container的输入**

**/bin/bash：在container中启动一个bash shell**

**这样就进入container的内部了：**

**root@af8bae53bdd3:/#**

**1**

**如果有运行中的container，可以在container所在的外部操作系统中运行：**

**docker ps**

**1**

**查看到这个container。**

**如果想看到所有的container，包括运行中的，以及未运行的或者说是沉睡镜像，则运行：**

**docker ps -a**

**1**

**如果要退出就：**

**Ctrl-D**

**或：**

**root@af8bae53bdd3:/# exit**

**1**

**如果想再次打开这个container，运行：**

**docker start goofy\_almeida**

**1**

**其中“goofy\_almeida”是容器的名称。**

**4 进入container（容器）**

**4.1 使用“docker attach”命令进入**

**这个时候container运行在后台，如果想进入它的终端，则：**

**docker attach goofy\_almeida**

**1**

**就可以了。**

**4.2 使用“docker exec -it”命令进入**

**使用“docker attach”命令进入container（容器）有一个缺点，那就是每次从container中退出到前台时，container也跟着退出了。**

**要想退出container时，让container仍然在后台运行着，可以使用“docker exec -it”命令。每次使用这个命令进入container，当退出container后，container仍然在后台运行，命令使用方法如下：**

**docker exec -it goofy\_almeida /bin/bash**

**1**

**goofy\_almeida：要启动的container的名称**

**/bin/bash：在container中启动一个bash shell**

**这样输入“exit”或者按键“Ctrl + C”退出container时，这个container仍然在后台运行，通过：**

**docker ps**

**1**

**就可以查找到。**

**5 退出container**

**输入：**

**exit**

**1**

**或者按键：**

**Ctrl + D**

**---------------------**

**作者：我是干勾鱼**

**来源：CSDN**

**原文：https://blog.csdn.net/dongdong9223/article/details/52998375**

**版权声明：本文为博主原创文章，转载请附上博文链接！**

**1 从容器拷贝文件到宿主机**

**拷贝方式为：**

**docker cp 容器名：容器中要拷贝的文件名及其路径 要拷贝到宿主机里面对应的路径**

**例如，将容器：**

**mycontainer**

**中路径：**

**/opt/testnew/**

**下的文件：**

**file.txt**

**拷贝到宿主机：**

**/opt/test/**

**路径下，在宿主机中执行命令如下：**

**docker cp mycontainer：/opt/testnew/file.txt /opt/test/**

**1**

**2 从宿主机拷贝文件到容器**

**拷贝方式为：**

**docker cp 宿主机中要拷贝的文件名及其路径 容器名：要拷贝到容器里面对应的路径**

**例如，将宿主机中路径：**

**/opt/test/**

**下的文件：**

**file.txt**

**拷贝到容器：**

**mycontainer**

**的：**

**/opt/testnew/**

**路径下，同样还是在宿主机中执行命令如下：**

**docker cp /opt/test/file.txt mycontainer：/opt/testnew/**

**---------------------**

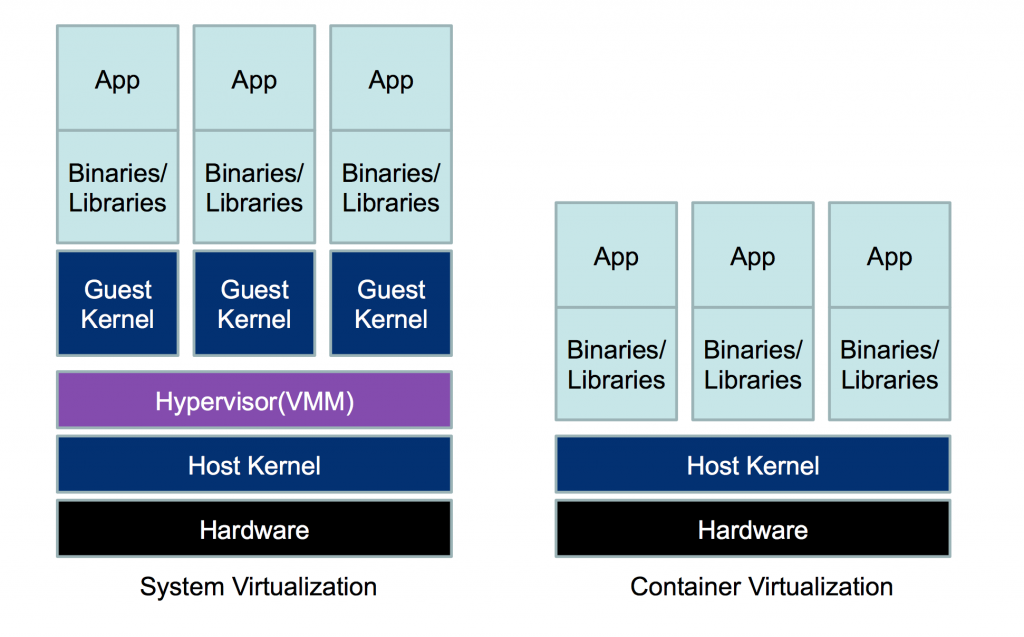
**作者：我是干勾鱼**

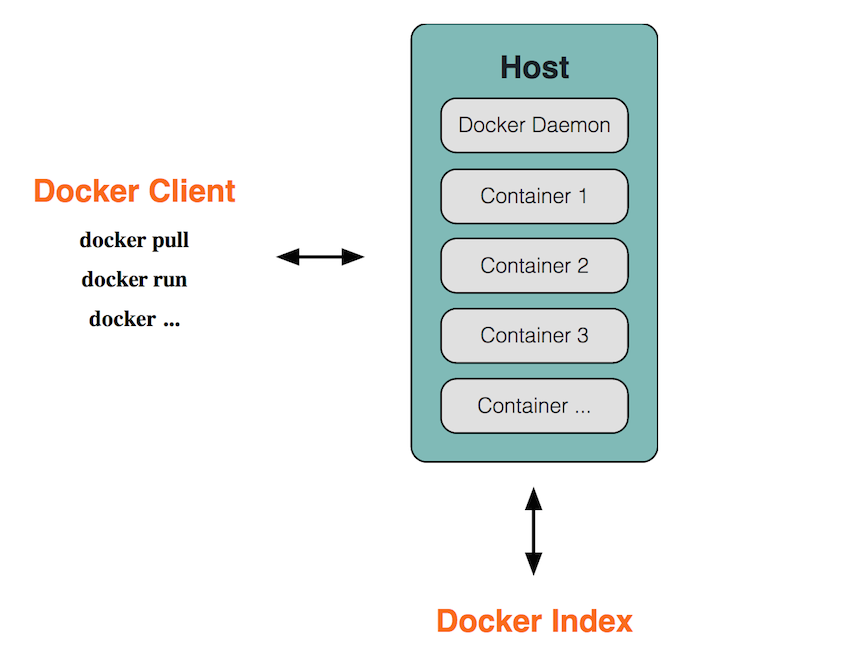
**来源：CSDN**

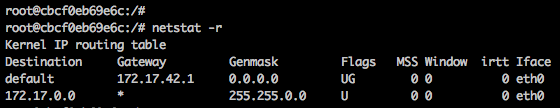
**原文：https://blog.csdn.net/dongdong9223/article/details/71425077**

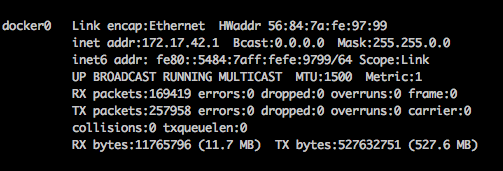
**版权声明：本文为博主原创文章，转载请附上博文链接！**

**https://m.jb51.net/os/other/409447.html**

**什么是Docker？**docker的英文本意是码头工人，也就是搬运工，这种搬运工搬运的是集装箱（Container），集装箱里面装的可不是商品货物，而是任意类型的App，Docker把App（叫Payload）装在Container内，通过Linux Container技术的包装将App变成一种标准化的、可移植的、自管理的组件，这种组件可以在你的latop上开发、调试、运行，最终非常方便和一致地运行在production环境下。  
Docker的核心底层技术是LXC（Linux Container），Docker在其上面加了薄薄的一层，添加了许多有用的功能。这篇stackoverflow上的问题和答案很好地诠释了Docker和LXC的区别，能够让你更好的了解什么是Docker， 简单翻译下就是以下几点：  
Docker提供了一种可移植的配置标准化机制，允许你一致性地在不同的机器上运行同一个Container；而LXC本身可能因为不同机器的不同配置而无法方便地移植运行；  
Docker以App为中心，为应用的部署做了很多优化，而LXC的帮助脚本主要是聚焦于如何机器启动地更快和耗更少的内存；  
Docker为App提供了一种自动化构建机制（Dockerfile），包括打包，基础设施依赖管理和安装等等；  
Docker提供了一种类似git的Container版本化的机制，允许你对你创建过的容器进行版本管理，依靠这种机制，你还可以下载别人创建的Container，甚至像git那样进行合并；  
Docker Container是可重用的，依赖于版本化机制，你很容易重用别人的Container（叫Image），作为基础版本进行扩展；  
Docker Container是可共享的，有点类似github一样，Docker有自己的INDEX，你可以创建自己的Docker用户并上传和下载Docker Image；  
Docker提供了很多的工具链，形成了一个生态系统；这些工具的目标是自动化、个性化和集成化，包括对PAAS平台的支持等；  
那么Docker有什么用呢？对于运维来说，Docker提供了一种可移植的标准化部署过程，使得规模化、自动化、异构化的部署成为可能甚至是轻松简单的事情；而对于开发者来说，Docker提供了一种开发环境的管理方法，包括映像、构建、共享等功能。  
**Docker vs. VM**从下图可以看出，VM是一个运行在宿主机之上的完整的操作系统，VM运行自身操作系统会占用较多的CPU、内存、硬盘资源。Docker不同于VM，只包含应用程序以及依赖库，基于libcontainer运行在宿主机上，并处于一个隔离的环境中，这使得Docker更加轻量高效，启动容器只需几秒钟之内完成。由于Docker轻量、资源占用少，使得Docker可以轻易的应用到构建标准化的应用中。但Docker目前还不够完善，比如隔离效果不如VM，共享宿主机操作系统的一些基础库等；网络配置功能相对简单，主要以桥接方式为主；查看日志也不够方便灵活。  


另外，IBM发表了一篇关于虚拟机和Linux container性能对比的论文，论文中实际测试了虚拟机和Linux container在CPU、内存、存储IO以及网络的负载情况，结果显示Docker容器本身几乎没有什么开销，但是使用AUFS会一定的性能损耗，不如使用Docker Volume，Docker的NAT在较高网络数据传输中会引入较大的工作负载，带来额外的开销。不过container的性能与native相差不多，各方面的性能都一般等于或者优于虚拟机。Container和虚拟机在IO密集的应用中都需要调整优化以更好的支持IO操作，两者在IO密集型的应用中都应该谨慎使用。  
**Docker组件  
**

Docker是CS架构，主要由下面三部分组成：  
Docker daemon: 运行在宿主机上，Docker守护进程，用户通过Docker client(Docker命令)与Docker daemon交互  
Docker client: Docker 命令行工具，是用户使用Docker的主要方式，Docker client与Docker daemon通信并将结果返回给用户，Docker client也可以通过socket或者RESTful api访问远程的Docker daemon  
Docker hub/registry: 共享和管理Docker镜像，用户可以上传或者下载上面的镜像，官方地址为https://registry.hub.docker.com/，也可以搭建自己私有的Docker registry。  
了解了Docker的组成，再来了解一下Docker的两个主要概念：  
Docker image：镜像是只读的，镜像中包含有需要运行的文件。镜像用来创建container，一个镜像可以运行多个container；镜像可以通过Dockerfile创建，也可以从Docker hub/registry上下载。  
Docker container：容器是Docker的运行组件，启动一个镜像就是一个容器，容器是一个隔离环境，多个容器之间不会相互影响，保证容器中的程序运行在一个相对安全的环境中。  
**Docker网络**Docker的网络功能相对简单，没有过多复杂的配置，Docker默认使用birdge桥接方式与容器通信，启动Docker后，宿主机上会产生docker0这样一个虚拟网络接口， docker0不是一个普通的网络接口， 它是一个虚拟的以太网桥，可以为绑定到docker0上面的网络接口自动转发数据包，这样可以使容器与宿主机之间相互通信。每次Docker创建一个容器，会产生一对虚拟接口，在宿主机上执行ifconfig，会发现多了一个类似veth\*\*\*\*这样的网络接口，它会绑定到docker0上，由于所有容器都绑定到docker0上，容器之间也就可以通信。  
在宿主机上执行ifconfig，会看到docker0这个网络接口， 启动一个container，再次执行ifconfig, 会有一个类似veth\*\*\*\*的interface，每个container的缺省路由是宿主机上docker0的ip，在container中执行netstat -r可以看到如下图所示内容：  


容器中的默认网关跟docker0的地址是一样的：  


当容器退出之后，veth\*虚拟接口也会被销毁。  
除bridge方式，Docker还支持host、container、none三种网络通信方式，使用其它通信方式，只要在Docker启动时，指定--net参数即可，比如:

docker run -i -t --net=host ubuntu /bin/bash

host方式可以让容器无需创建自己的网络协议栈，而直接访问宿主机的网络接口，在容器中执行ip addr会发现与宿主机的网络配置是一样的，host方式让容器直接使用宿主机的网络接口，传输数据的效率会更加高效，避免bridge方式带来的额外开销，但是这种方式也可以让容器访问宿主机的D-bus等网络服务，可能会带来意想不到的安全问题，应谨慎使用host方式；container方式可以让容器共享一个已经存在容易的网络配置； none方式不会对容器的网络做任务配置，需要用户自己去定制。  
**Docker生态**随着Docker迅速火遍全球， 以Docker为基础的生态系统也迅速的发展起来，从以部署和运行container为基础的CoreOS到各种各样的管理工具和PaaS软件，Docker以及生态产品都在迅猛发展，以下介绍几个代表性的软件。  
首先介绍CoreOS，它的出现极大地推动了Docker技术的推广和发展，CoreOS是专门为大规模服务部署而设计的一种新的Linux发行版，通过运行轻量级的容器方便扩展和维护大规模的服务。它具有以下特点：  
CoreOS使用container管理服务(容器即服务)，即以容器的角度去管理服务，服务的代码和依赖都打包到容器里，打包后的容器直接在CoreOS上运行管理。通过容器用户不再需要关注虚拟机环境等，极大地降低了服务和系统环境的耦合性。另外部署在CoreOS的多个容器都运行在各自独立的环境中，不会相互影响。  
CoreOS专门为cluster等大规模部署而设计，提供了Etcd进行服务发现，以及Fleet管理容器保证服务可用。  
CoreOS更加精简，比如RAM使用比普通Linux低40%。  
CoreOS采用双分区模式(Dual-Partition)，主分区为主动模式，负责系统运行，被动模式分区负责系统更新，更新时将整个CoreOS系统下载下来。  
CoreOS是为集群服务而设计的，提供了Etcd、Fleet等管理工具管理容器和服务。Etcd是一种类似Zookeeper的分布式key/value存储服务，用于服务发现和配置管理。Fleet是容器管理工具，保证服务的可用性，当某个机器的服务不可用时，Fleet会将服务迁移到其它机器上运行。  
Docker生态中还有一个非常重要的容器管理工具--Kubernetes，它是Google开源的用于在集群环境中管理、维护、自动扩展容器，通过Kubernetes可以很方便地在多个机器上管理和部署容器服务。现在已经得到IBM、Microsoft、RedHat等多个大公司的支持。  
在Kubernetes中pod是一个基本单元，一个pod可以是提供相同功能的多个container，这些容器会被部署在同一个minion上。Replication controller定义了多个pod或者容器需要运行，如果当前集群中运行的pod或容器达不到配置的数量，replication controller会调度容器在多个minion上运行，保证集群中的pod数量。service则定义真实对外提供的服务，一个service会对应后端运行的多个container。Kubernetes的架构由一个master和多个minion组成，master通过api提供服务，接受kubectl的请求来调度管理整个集群。minion是运行Kubelet的机器，它接受master的指令创建pod或者容器。  
最后介绍一下基于Docker实现的PaaS软件，Docker PaaS软件中以Deis和Flynn最为知名。Deis是基于Docker和CoreOS实现的轻量级的PaaS，受到Heroku的启发，遵循“十二要素”构建应用方法。Deis是以应用程序为中心设计的，分为build、release、run三个阶段，用户执行"git push"后，Deis使用Docker 容器编译并将编译结果保存在Docker镜像；发布阶段，一次build和配置文件产生一个数字标识的发布镜像，将发布镜像保存到Docker registry中以供后续发布到线上运行；运行阶段应用镜像会被调度到主机上运行，并更新相应的路由。Flynn与Deis类似，也是以应用为中心，Flynn组件分为两层，layer0是底层资源的抽象，主要负责资源调度以及服务发现等，为上层应用容器的运行提供底层资源调度支持；layer1处理具体应用，通过Docker容器编译、部署和维护上层应用程序。