# 文 并存共生or相爱相杀？容器、虚拟机与Docker概念全解析 (/a/1190000004681086)

docker (https://segmentfault.com/t/docker/blogs) 容器 (https://segmentfault.com/t/容器/blogs) 虚拟机 (https://segmentfault.com/t/虚拟机/blogs)

**数人云** (https://segmentfault.com/u/shurenyun) 3 天前发布

> 当小数看到这篇文章时内心是激动的，因为或许介绍Docker容器的文章有无数，但是如此清晰易懂、对小白如此友好的却不多见。本文立足于新手，从容器和虚拟机两个大的概念入手，由浅入深，由宏转微，为我们解析了Docker的方方面面。来吧朋友们，理解它，热爱它，然后更好地使用它。

作为程序员或者技术人员，大家肯定听说过Docker的鼎鼎大名——这款工具能够帮助我们高效打包、发布及运行承载着应用程序的“容器”系统。其发展如火如荼——从开发者到运维人员，每个人都在关注着这位技术新贵。即使是像谷歌、VMware与Amazon这样的技术巨擘也在构建相关服务为其提供支持。

无论大家是否有意立即使用Docker，我们都应当对其基础概念加以了解，并明确区分容器与虚拟机之间的差异所在。目前网络上已经存在大量对二者关联与区别的阐述性文章，不过在今天的综述中，我们将立足于新手对其加以剖析。

下面首先聊聊虚拟机与容器究竟是什么。

## 容器与虚拟机究竟是什么？

容器与虚拟机拥有着类似的使命：对应用程序及其关联性进行隔离，从而构建起一套能够随处运行的自容纳单元。

此外，容器与虚拟机还摆脱了对物理硬件的需求，允许我们更为高效地使用计算资源，从而提升能源效率与成本效益。

容器与虚拟机之间的核心差异在于其架构方法。下面一起进行深入了解。

## 虚拟机

虚拟机在本质上就是在模拟一台真实的计算机设备，同时遵循同样的程序执行方式。虚拟机能够利用“虚拟机管理程序”运行在物理设备之上。反过来，虚拟机管理程序则可运行在主机设备或者“裸机”之上。

下面用更直白的表达来说明：

虚拟机管理程序可表现为软件、固件或者硬件，并作为虚拟机的运行基础。虚拟机管理程序本身运行在物理计算机之上，我们也将这种底层硬件称为“主机设备”。主机设备为虚拟机提供资源，包括内存与CPU。这些资源由不同虚拟机共享，并根据需要进行随意分配。因此如果一套虚拟机运行有需求大量资源的高强度应用程序，那么我们可以在同一主机设备上为其分配远高于其它虚拟机的资源配额。

运行在主机设备上的虚拟机（当然，需要配合虚拟机管理程序）通常被称为一套“客户机”。这套客户机容纳有应用程序及其运行所必需的各类组件（例如系统二进制文件及库）。它同时还包含有完整的虚拟硬件堆栈，其中包括虚拟网络适配器、存储以及CPU——这意味着它也拥有自己的完整访客操作系统。着眼于内部，这套客户机自成体系并拥有专用资源。而从外部来看，这套虚拟机使用的则是由主机设备提供的共享资源。

如上所述，客户机可以运行主机虚拟机管理程序或者裸机虚拟机管理程序。二者之间存在着多种重要区别。

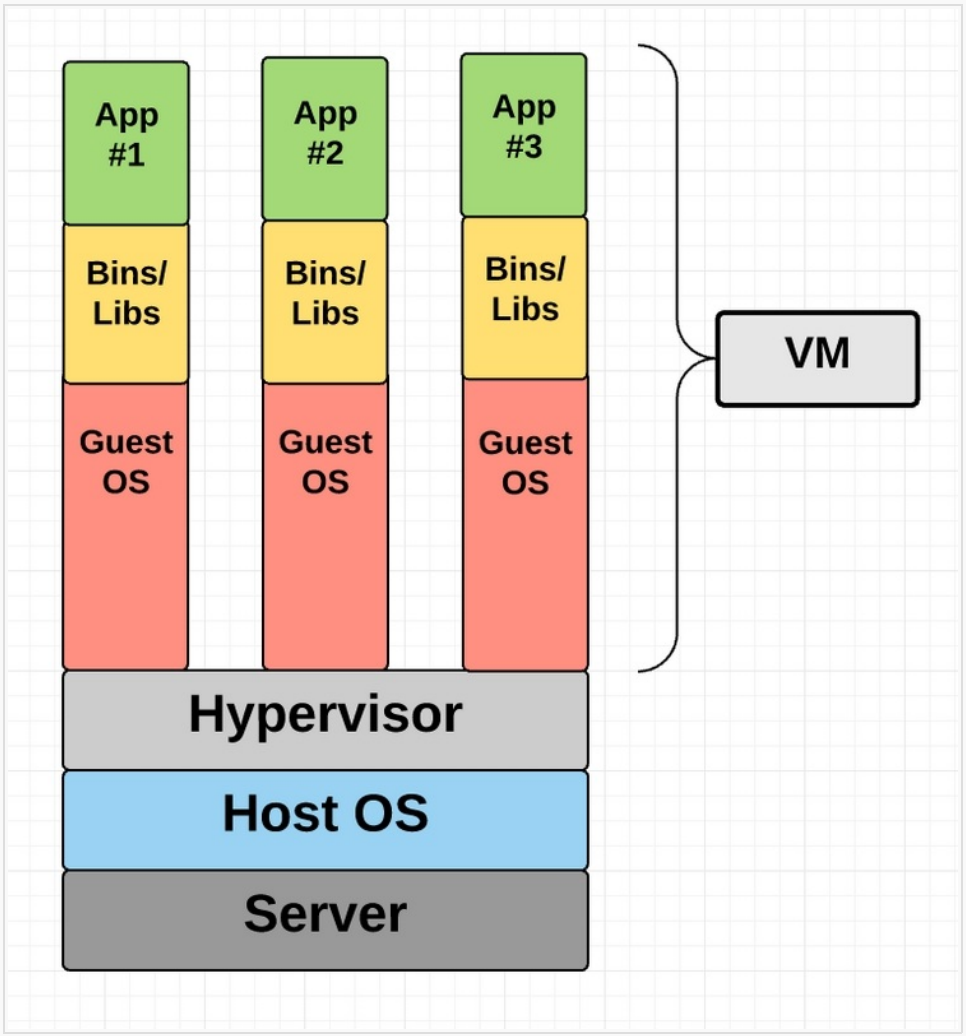
首先，主机虚拟机管理程序运行在主机设备的操作系统之上。举例来说，一台运行有OS X的计算机可以在操作系统之上安装虚拟机（例如VirtualBox或者VMware Workstation 8）。该虚拟机并不会直接访问硬件，因此其需要经由主机操作系统（也就是Mac OS X）实现资源获取。

主机虚拟机管理程序的优势在于，其基本摆脱了对底层硬件的要求。该主机操作系统负责提供硬件驱动程序，而非由虚拟机管理程序自身提供，因此我们认为其具备更理想的“硬件兼容性”。在另一方面，这种介于硬件与虚拟机管理程序之间的额外层会带来更多资源消耗，进而降低虚拟机性能表现。

裸机虚拟机管理程序则将虚拟机直接安装并运行在主机设备硬件之上以改善性能表现。由于其直接接入底层硬件，因此我们不再需要主机操作系统作为辅助。在这种情况下，我们可以直接在硬件上安装虚拟机管理程序并将其作为操作系统。与主机虚拟机管理程序不同，裸机虚拟机管理程序拥有自己的设备驱动程序及接口，从而直接支持各类I/O、处理或者操作系统特定任务相关组件。这种方式能够带来更理想的性能水平、可扩展性以及稳定性。但代价是其硬件兼容性比较有限，因为虚拟机管理程序只能包含一部分设备驱动程序。

说到这里，大家可能提出疑问：为什么我们非得在虚拟机与主机设备之间添加“虚拟机管理程序”呢？

这个嘛，因为虚拟机本身拥有一套虚拟操作系统，而虚拟机管理程序则负责为虚拟机提供平台以管理并运行这套访客操作系统。如此一来，主机计算机就能够为运行于其上的各虚拟机分配共享资源了。



虚拟机原理示意图

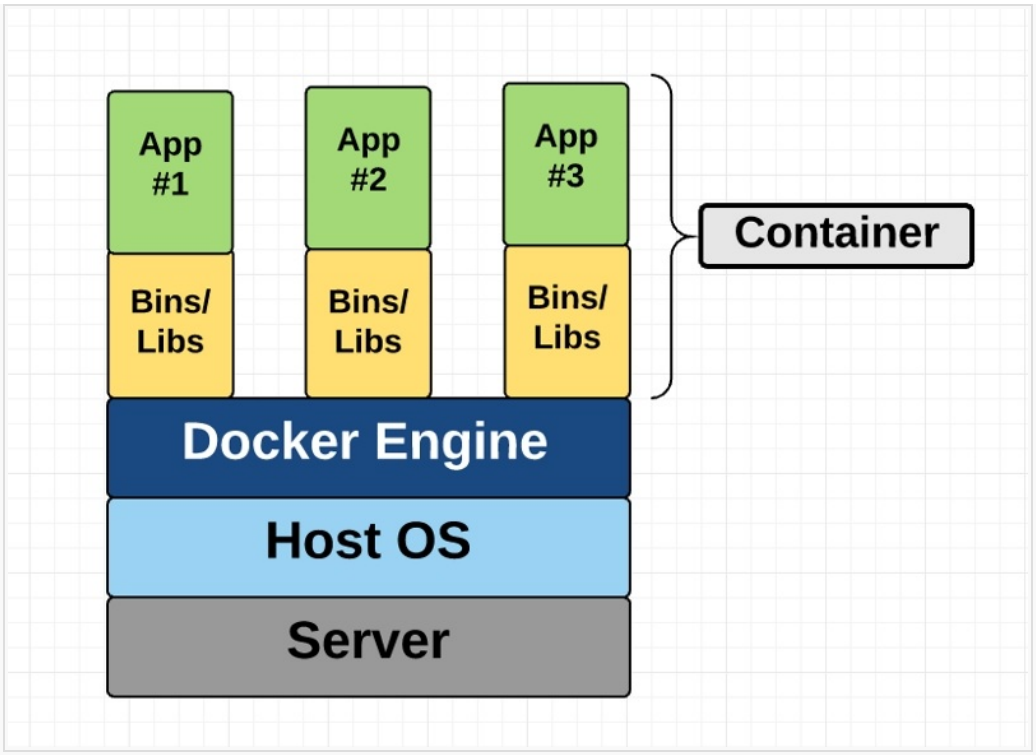
如大家所见，虚拟机会将虚拟硬件、内核（即操作系统）以及用户空间打包在新虚拟机当中。

## 容器

与提供硬件虚拟化机制的虚拟机不同，容器通过对“用户空间”的抽象化处理提供操作系统层级的虚拟化机制。通过对容器进行分解，大家将可以非常清晰地理解其中含义。

出于各种考量与需求，容器在外观上与虚拟机非常相似。举例来说，二者皆拥有专有处理空间、能够作为root执行命令、提供专有网络接口与IP地址、允许定制化路由及iptable规则，且可启动文件系统等等。

容器与虚拟机间的最大区别在于，各容器系统*共享*主机系统的内核。



容器原理示意图

以上示意图显示了容器如何对用户空间进行打包，而不像虚拟机那样同样对内核或者虚拟硬件进行打包。每套容器都拥有自己的隔离化用户空间，从而使得多套容器能够运行在同一主机系统之上。我们可以看到全部操作系统层级的架构都可实现跨容器共享。惟一需要独立构建的就是二进制文件与库。正因为如此，容器才拥有极为出色的轻量化特性。

## Docker的功能定位

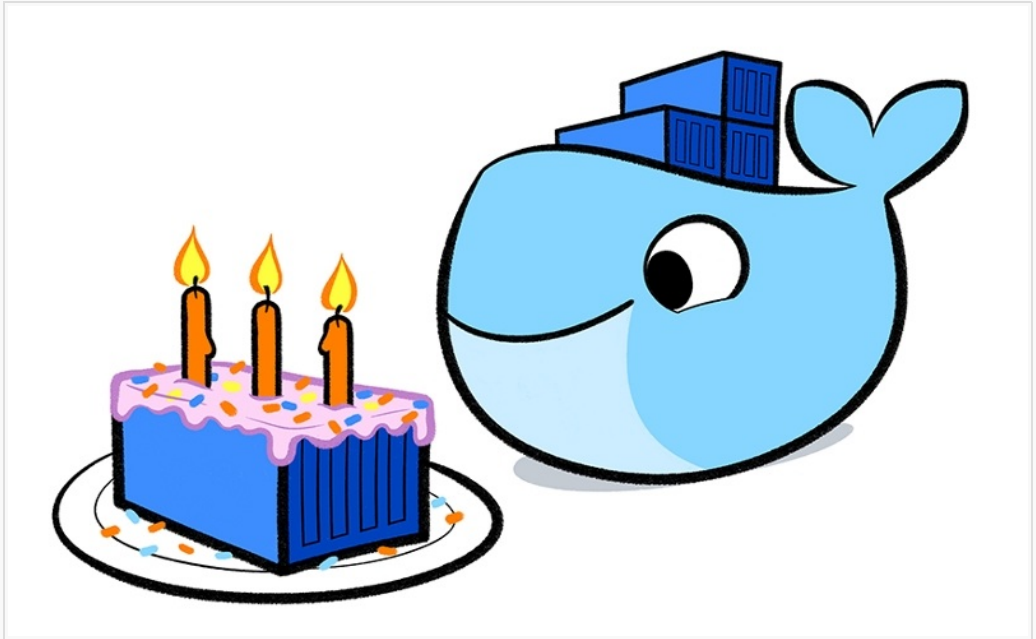
Docker为基于Linux容器的开源项目，其利用Linux内核中的各项功能——例如命名空间与控制组——以在操作系统之上创建容器。

容器概念并不是什么新鲜事物; 谷歌公司多年来一直在使用自己开发的容器技术。其它Linux容器技术方案还包括Solaris Zones、BSD jails以及LXC，且其都已经拥有多年的发展历史。

那么为什么Docker的出现会快速吸引到技术业界的注意？

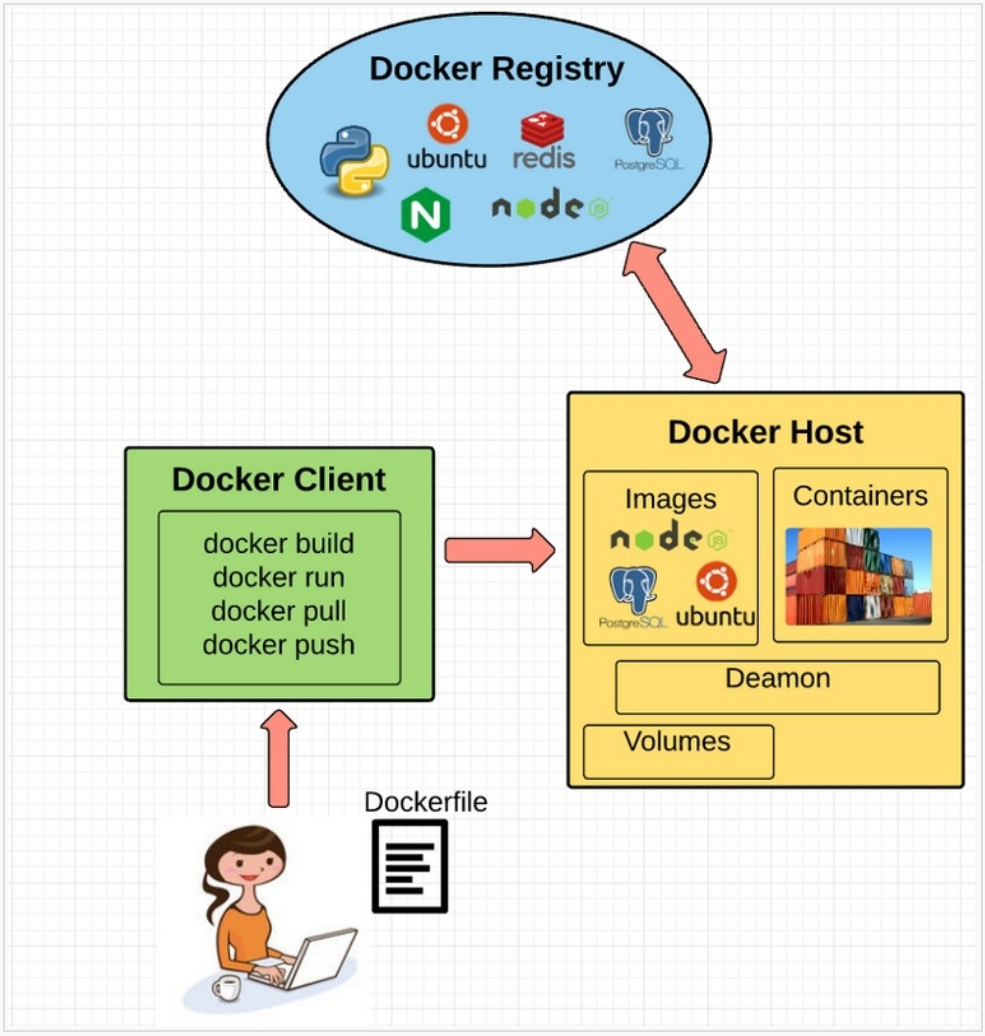
- 易用性: Docker能够为潜在受众带来出色的易用性——开发者、系统管理员以及架构师等等——从而帮助其充分利用容器技术优势以快速构建并测试可移植应用程序。每个人都可以在自己的笔记本上打包应用程序，并将其直接运行在任何公有云、私有云甚至是裸机之上。其座右铭是：一次构建，随处运行。
- 速度: Docker容器具备轻量化与高速特性。由于容器本身属于运行在内核之上的沙箱环境，因为其对资源的需求量极低。大家可以在数秒钟内完成容器的创建与运行，而虚拟机则由于需要引导完整的虚拟操作系统而耗费更多时间。
- Docker Hub: Docker用户还能够享受由Docker Hub带来的丰富生态系统支持，我们可以将其理解成“Docker镜像的应用商店”。Docker Hub提供成千上万由社区开发的公共镜像，且可立即加以使用。我们可以轻松根据需要搜索到合适的镜像，将其提取并稍加修改即加以使用。
- 模块性与可扩展性: Docker允许我们轻松将应用程序的功能拆分成多个独立容器。举例来说，我们可以将自己的Postgres数据库运行在一套容器当中，并将Redis服务器运行在另一容器内，而Node.js也拥有自己的容器系统。在Docker的帮助上，大家能够轻松将这些容器对接起来以创建完整的应用程序，这就让未来的规模伸缩或者组件更新得以通过相互独立的方式完成。

最后但同样重要的是，Docker鲸鱼实在是太惹人喜爱了 ~



# Docker基本概念

现在我们对Docker有了宏观印象，下面具体对其组件进行解读：




## Docker Engine

Docker Engine属于Docker的运行层。这是一套轻量化运行时及工具组合，负责管理容器、镜像、构建 等等。它以原生方式运行在Linux系统之上，并由以下元素构成：

Docker Daemon，运行在主机计算机之上。
Docker Client，负责与Docker Daemon通信以执行命令。
REST API，用于同Docker Daemon远程交互。

## Docker Client

Docker Client正是我们作为最终用户的通信对象。我们可以将其视为Docker的UI。

```
docker build iampeekay/someImage .
```

dockerbuild.sh hosted with ♥ by GitHub view raw

我们进行的一切操作都将直接接入Docker Client，再由其将指令传递至Docker Daemon。

## Docker Daemon

Docker Daemon直接将执行命令发送至Docker Client——例如构建、运行以及分发等等。Docker Daemon运行在主机设备之上，但作为用户，我们永远不会直接与该Daemon进行通信。
Docker Client也可以运行在主机设备上，但并非必需。它亦能够运行在另一台设备上，并与运行在目标主机上的Docker Daemon进行远程通信。

## Dockerfile

Dockerfile是我们编写指令以构建Docker镜像的载体。这些指令包括：
RUN apt-get y install some-package:安装某软件包
EXPOSE 8000: 开放端口
ENV ANT_HOME /usr/local/apache-ant：传递环境变量

类似的指令还有很多。一旦设置了Dockerfile，大家就可以利用docker build命令来构建镜像了。下面来看Dockerfile示例：

```
# Start with ubuntu 14.04
FROM ubuntu:14.04

MAINTAINER preethi kasireddy iam.preethi.k@gmail.com

# For SSH access and port redirection
ENV ROOTPASSWORD sample

# Turn off prompts during installations
ENV DEBIAN_FRONTEND noninteractive
RUN echo "debconf shared/accepted-oracle-license-v1-1 select true" | debconf-set-selections
RUN echo "debconf shared/accepted-oracle-license-v1-1 seen true" | debconf-set-selections

# Update packages
RUN apt-get -y update

# Install system tools / libraries
RUN apt-get -y install python3-software-properties \
    software-properties-common \
    bzip2 \
    ssh \
    net-tools \
    vim \
    curl \
    expect \
    git \
    nano \
    wget \
    build-essential \
    dialog \
    make \
    build-essential \
    checkinstall \
    bridge-utils \
    virt-viewer \
    python-pip \
    python-setuptools \
    python-dev

# Add oracle-jdk7 to repositories
RUN add-apt-repository ppa:webupd8team/java

# Make sure the package repository is up to date
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list

# Update apt
RUN apt-get -y update

# Install oracle-jdk7
RUN apt-get -y install oracle-java7-installer

# Export JAVA_HOME variable
ENV JAVA_HOME /usr/lib/jvm/java-7-oracle

# Run sshd
RUN apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo "root:$ROOTPASSWORD" | chpasswd
RUN sed -i 's/PermitRootLogin without-password/PermitRootLogin yes/' /etc/ssh/sshd_config

# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i /etc/pam.d/sshd

# Expose Node.js app port
EXPOSE 8000

# Create tap-to-android app directory
RUN mkdir -p /usr/src/my-app
WORKDIR /usr/src/my-app

# Install app dependencies
COPY . /usr/src/my-app
RUN npm install

# Add entrypoint
ADD entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

CMD ["npm", "start"]
```

示例Dockerfile

## Docker镜像

镜像属于只读模板，大家可以借此配合Dockerfile中的编写指令集进行容器构建。镜像定义了打包的应用程序以及其相关依赖。这些依赖就好像是其启动时需要运行的进程。

Docker镜像利用Dockerfile实现构建。Dockerfile中的每条指令都会在镜像中添加一个新的“层”，这些层则表现为镜像文件系统中的一个分区——我们可以对其进行添加或者替换。层概念正是Docker轻量化的基础。Docker利用一套Union File System建立起这套强大的结构：

## Union File System

Docker利用Union File System以构建镜像。大家可以将Union File System视为一套可堆叠文件系统，这意味着各文件系统中的文件与目录（在Docker中被称为分支）可以透明方式覆盖并构成单一文件系统。

覆盖分支内的各目录内容拥有同样的路径，并将被视为单一合并目录，这就避免了需要为每个层分别创建副本的麻烦。相反，这些目录可调用特定指针以指向同样的资源; 当特定层需要进行修改时，其会分别创建修改前与修改后的本地副本。通过这种方式，文件系统*表现*出可写入特性，但其实际上并未接受任何写入操作。（换言之，这是一套‘写入即复制’系统。）

分层系统拥有两大核心优势：

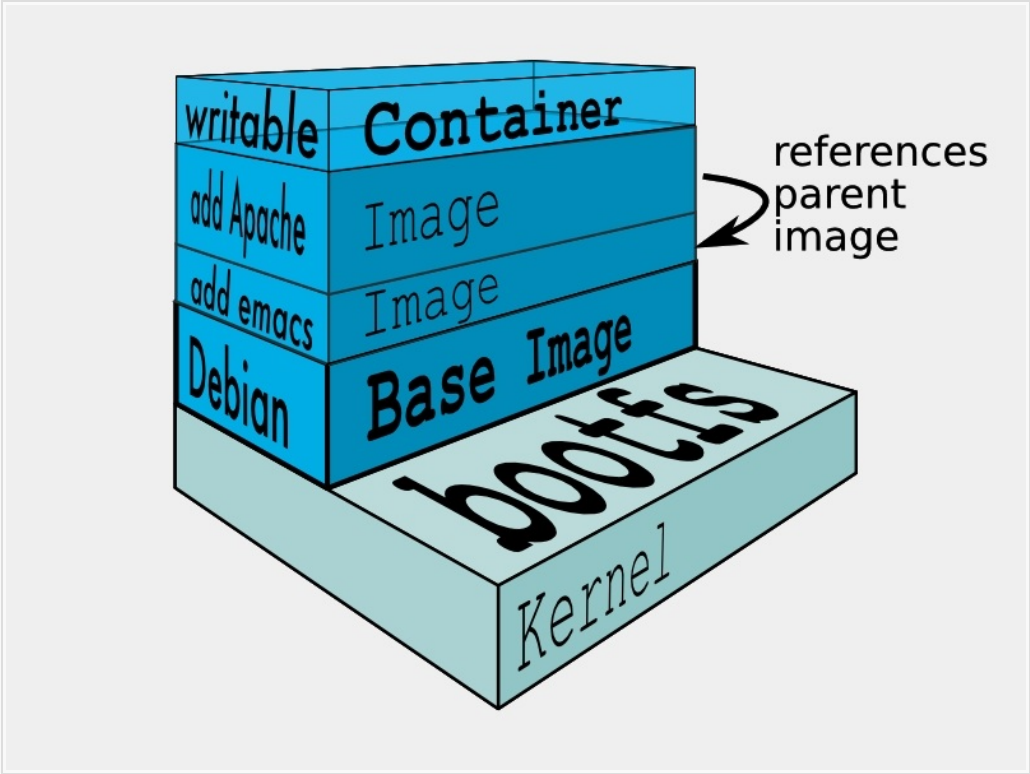
无重复数据: 分层机制使得我们在使用镜像创建并运行新容器时无需复制完整的文件集，从而保证Docker容器实例拥有良好的运行速度与低廉的资源成本。
层隔离: 变更操作速度很快——当我们对镜像进行变更时，Docker只需要对进行了变更的层进行广播。

## Volumes

Volumes属于容器中的“数据”部分，会在容器创建时进行初始化。Volumes机制允许我们持久保留并共享容器数据。数据卷 独立于默认Union File System之外，且作为普通目录及文件存在于主机文件系统当中。因此即使是在对容器进行销毁、更新或者重构时，数据卷仍然不受影响。当我们需要对某一数据卷进行更新时，直接加以变更即可。（作为另一项优势，数据卷还能够在不同容器之间进行共享与复用。）

## Docker容器

如上所述，Docker容器将一款应用程序的软件打包在单一环境当中，同时包含全部运行必需的要素。其中包括操作系统、应用程序代码、运行时、系统工具、系统库等等。Docker容器由Docker镜像构建而成。由于镜像存在只读属性，因此Docker会在镜像在只读文件系统之上添加一套读取-写入文件系统以实现容器创建。




另外，在创建容器时，Docker还会创建一套网络接口以帮助容器同本地主机通信、对接可用IP地址并执行用户在定义镜像时所执行的进程以运行应用程序。
在成功创建了一套容器之后，我们随后可以将其运行在任何环境当中，而不必再做任何变更。

## “容器”探秘

打开容器，我们会发现其中包含大量活动组件。容器的实现机制一直令我感到惊讶而好奇，特别是考虑到容器不存在任何抽象的基础设施边界。在阅读了大量材料之后，我将结合自己的理解为大家进行讲解：）

“容器”这一术语其实只是个抽象概念，用于表述多种不同的相关特性，而其与原词“container”的另一含义“集装箱”有着千丝万缕的联系。下面就来一起了解：

1) 命名空间

命名空间为容器提供对应的底层Linux系统视图，即限制容器的查看与访问范畴。当我们运行一套容器时，Docker会创建多个命名空间供特定容器使用。
Docker会在内核中使用多种不同类型的命名空间，例如：

- NET: 为容器提供独特的系统网络堆栈视图（例如自有网络设备、IP地址、IP路由表、/proc/net目录、端口编号等等）。
- PID: PID代表进程ID。如果大家曾经在命令行中运行过ps aux以检查当前系统正在运行的进程，就会发现其中一栏名为“PID”。PID命名空间为容器提供其能够查看与交互的进程范围，其中包括独立的init（PID 1），其属于“所有进程的元祖”。
- MNT: 为容器提供独特的系统“mounts”视图。这样不同mount命名空间内的进程就将拥有彼此不同的文件系统结构。
- UTS: UTS代表UNIX分时系统。它允许某一进程识别系统身份（例如主机名称或者域名等）。UTS允许容器拥有不同于其它容器以及主机系统的主机名称与NIS域名。
- IPC: IPC代表进程间通信。IPC命名空间负责对运行在每套容器内的进程进行IPC资源隔离。
- USER: 此命名空间用于对每套容器内的用户进行隔离。其允许各容器拥有不同的uid（即用户ID）与gid（组ID）视图区间，并将其与主机系统进行比对。这样一来，某一进程的uid与gid在用户命名空间之内与之外即有所不同，这也使得该进程能够在不影响容器内root权限的情况下，撤销同一用户在容器外的权限。

Docker将这些命名空间结合起来以隔离并创建容器。下面要讲的则是控制组。

2) 控制组

控制组（也被称为cgroups）属于Linux内核中的一项功能，用于对一组进程的资源使用量（包括CPU、内存、磁盘I/O以及网络等）进行隔离、排序与计数。这意味着cgroup能够确保Docker容器只使用其必需的资源——并在必要情况下设置其所能使用的资源上限。另外，cgroups还能够确保单一容器不至于占用太多资源并导致整体系统陷入瘫痪。

最后要说明的是Union文件系统：

3) 隔离化Union file system：

我们在之前的Docker镜像章节中已经解释过了：）

这就是关于Docker容器的全部内容了（当然，实现细节才是最麻烦的环节——例如如何管理不同组件间的交互）。

## Docker的未来：Docker与虚拟机将并生共存

尽管Docker已经开始逐步成为主流，但我认为它不太可能对虚拟机造成真正的威胁。容器将继续发展壮大，但虚拟机也仍然拥有适合自己的生存空间。

举例来说，如果我们需要在多台服务器上运行多款应用程序，那么最理想的办法就是使用虚拟机。在另一方面，如果我们需要运行同一应用程序的多套*副本*，那么Docker则拥有更多具体优势。

另外，尽管容器允许我们将应用程序拆分成多个功能组件，但这种分散化趋势意味着我们需要管理更多功能部件，即带来更为复杂的控制与协调任务。

安全同时也是Docker容器需要解决的一大难题——由于各容器共享同一套内核，因此不同容器之间的屏障非常薄弱。与只需要调用主机虚拟机管理程序的虚拟机方案不同，Docker容器需要向主机内核发出系统调用，而这会带来更庞大的攻击面。出于安全性的考量，很多开发人员可能更倾向于选择虚拟机——其由抽象化硬件进行隔离，从而显著提升了彼此交互的难度。

当然，安全性与管理等难题必将随着容器在生产环境下的进一步普及而得到解决。就目前来讲，关于容器与虚拟机孰优孰劣的议题已经成为开发人员与运维人员间的日常争论素材。

## 最后

到这里，希望大家已经拥有了关于Docker的必要知识，也祝愿各位早日将Docker引入自己的日常工作。

当然，如果大家在文章中发现了什么谬误，也欢迎在评论栏中做出说明。感谢大家，小数与你下次再见！

> 原文链接：https://medium.freecodecamp.com/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b#.sl2xp8tiv (https://medium.freecodecamp.com/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b#.sl2xp8tiv)

3 天前发布 (/a/1190000004681086)



### 你可能感兴趣的文章

Docker之镜像容器--我的理解 (https://segmentfault.com/a/1190000003090146) 2 收藏，435 浏览

docker 简介 (https://segmentfault.com/a/1190000003073069) 4 收藏，316 浏览

Docker怎样改造你的开发团队 (https://segmentfault.com/a/1190000003758813) 16 收藏，705 浏览

### 讨论区





本文隶属于专栏

数人云 (https://segmentfault.com/blog/shurenyun)

“数人云”是部署在公有云或者私有云（IDC）之上的云操作系统，旨在帮助用户在云端快速建立并稳定运维高性能生产环境。可以让用户像用单机电脑一样管理集群和云端应用。

数人云 (https://segmentfault.com/u/shurenyun)
作者