**第一部分：Docker镜像的基本知识**

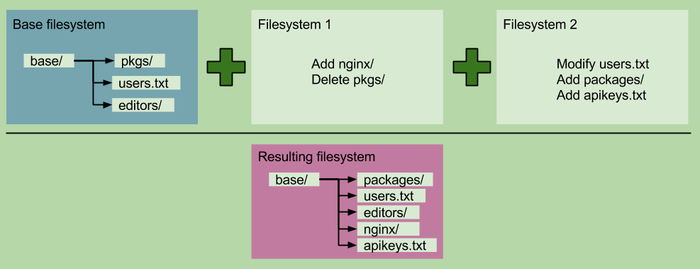
1. **Docker与虚拟机的对比**

## 底层实现

1. Docker 的基础是Linux容器（LXC）等技术。
2. 在LXC的基础上Docker进行了进一步的封装，让用户不需要去关心容器的管理，使得操作更为简便。
3. Docker使用Cgroups来提供容器隔离，而union文件系统用于保存镜像并使容器变得短暂。

Cgroups：Cgroups是Linux内核功能，它让两件事情变成可能：限制Linux进程组的资源占用（内存、CPU）；为进程组制作 PID、UTS、IPC、网络、用户及装载命名空间。

Union文件系统：在union文件系统里，文件系统可以被装载在其他文件系统之上，其结果就是一个分层的积累变化。如下图：

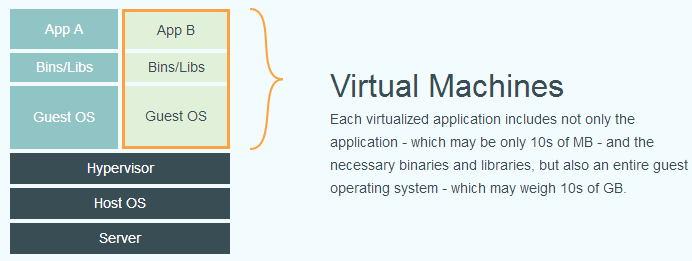


union文件系统

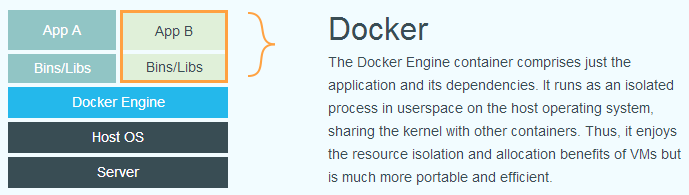
## 与虚拟机的关系

用户操作Docker的容器就像操作一个快速轻量级的虚拟机一样简单。

但是它们还是有很大的区别的，看下面两幅图：



virtualization应用



docker应用

虚拟机上的应用不仅包含了应用本身、必要的依赖（二进制和库包等），还包括整个操作系统；而docker应用不同，共享系统内核，仅仅包含应用本身和依赖。

**1.1 什么是Docker镜像**

从整体的角度来讲，一个完整的Docker镜像可以支撑一个Docker容器的运行，在 Docker容器运行过程中主要提供文件系统视角。例如一个ubuntu:14.04的镜像，提供了一个基本的ubuntu:14.04的发行版，当然此 镜像是不包含操作系统Linux内核的。

说到此，可能就需要注意一下，linux内核和ubuntu:14.04Docker镜像的区别了。传统虚拟机安装ubuntu:14.04会包含两部分，第一，某一个Linux内核的发行版本，比如Linux 3.8版本的内核；第二，第一个特定的Ubuntu发行版，这部分内容不包含Linux内核，但是包含Linux之外的软件管理方式，软件驱动，如 apt-get软件管理包等。

理解以上内容之后，就可以理解，为什么在一个Linux内核版本为3.8的ubuntu:14.04基础上，可以把Linux内核版本升级到3.18，而ubuntu的版本依然是14.04。最主要的就是：Linux内核版本与ubuntu操作系统发行版之间的区别。

Linux内核＋ubuntu操作系统发行版，组成一台工作的机器让用户体验。那么灵活替换ubuntu操作系统发行版，那是不是也可以实现呢。那么Docker很方便的利用了这一点，技术手段就是Docker镜像。

Docker的架构中，Docker镜像就是类似于“ubuntu操作系统发行版”，可 以在任何满足要求的Linux内核之上运行。简单一点有“Debian操作系统发行版”Docker镜像、“Ubuntu操作系统发行版”Docker镜 像；如果在Debian镜像中安装MySQL 5.6，那我们可以将其命名为Mysql:5.6镜像；如果在Debian镜像中安装有Golang 1.3,那我们可以将其命名为golang:1.3镜像；以此类推，大家可以根据自己安装的软件，得到任何自己想要的镜像。

那么镜像最后的作用是什么呢？很好理解，回到Linux内核上来运行，通过镜像来运行时我们常常将提供的环境称为容器。

以上内容是从宏观的角度看看Docker镜像是什么，我们再从微观的角度进一步深入 Docker镜像。刚才提到了“Debian镜像中安装MySQL 5.6，就成了mysql:5.6镜像”，其实在此时Docker镜像的层级概念就体现出来了。底层一个Debian操作系统镜像，上面叠加一个 mysql层，就完成了一个mysql镜像的构建。层级概念就不难理解，此时我们一般debian操作系统镜像称为mysql镜像层的父镜像。

层级管理的方式大大便捷了Docker镜像的分发与存储。说到分发，大家自然会联想到 Docker镜像的灵活性，传输的便捷性，以及高超的移植性。Docker Hub，作为全球的镜像仓库，作为Docker生态中的数据仓库，将全世界的Docker数据汇聚在一起，是Docker生态的命脉。

Docker有两方面的技术非常重要，第一是Linux 容器方面的技术，第二是Docker镜像的技术。从技术本身来讲，两者的可复制性很强，不存在绝对的技术难点，然而Docker Hub由于存在大量的数据的原因，导致Docker Hub的可复制性几乎不存在，这需要一个生态的营造。

**1.2 Docker镜像的内容**

大致介绍了Docker镜像是什么，我们来看看Docker镜像中有哪些内容？

介绍之前，我先分享一下，我个人在接触Docker的两年时间中，对Docker镜像内容认识的变化。

第一阶段：初步接触Docker。相信很多爱好者都会和我一样，有这样一个认识：Docker 镜像代表一个容器的文件系统内容；

第二阶段：初步接触联合文件系统。联合文件系统的概念，让我意识到镜像层级管理的技术，每一层镜像都是容器文件系统内容的一部分。

第三阶段：研究镜像与容器的关系：容器是一个动态的环境，每一层镜像中的文件属于静态内 容，然而 Dockerfile 中的 ENV、VOLUME、CMD 等内容最终都需要落实到容器的运行环境中，而这些内容均不可能直接坐落到每一层镜像所包含的文件系统内容中，那此时每一个Docker镜像还会包含 json文件记录与容器之间的关系。

因此，Docker镜像的内容主要包含两个部分：第一，镜像层文件内容；第二，镜像json文件。

**1.3 Docker镜像存储位置**

既然是说镜像存储的位置，那么应该包含：镜像层文件和镜像json文件。如一个ubuntu:14.04镜像，包含4个镜像层，在aufs存储驱动的情况下，在磁盘上的情况可以如以下图所示：

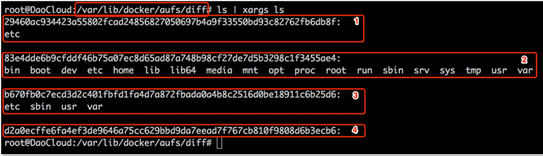
**1.3.1 查看镜像层组成：**

我们可以通过命令 docker history ubuntu:14.04 查看 ubuntu:14.04，结果如下：



**1.3.2 镜像层文件内容存储**

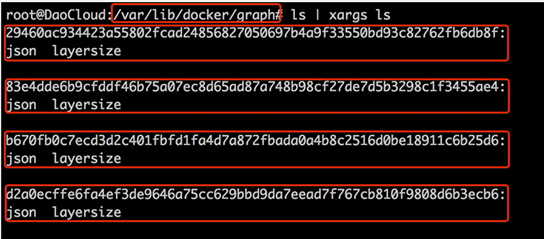
Docker 镜像层的内容一般在 Docker 根目录的 aufs 路径下，为 /var/lib/docker/aufs/diff/，具体情况如下：



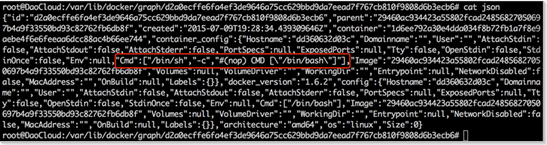
图中显示了镜像 ubuntu:14.04 的 4 个镜像层内容，以及每个镜像层内的一级目录情况。需要额外注意的是：镜像层 d2a0ecffe6fa 中没有任何内容，也就是所谓的空镜像。

**1.3.3 镜像 json 文件存储**

对于每一个镜像层，Docker 都会保存一份相应的 json 文件，json 文件的存储路径为 /var/lib/docker/graph，ubuntu:14.04 所有镜像层的 json 文件存储路径展示如下：



除了 json 文件，大家还看到每一个镜像层还包含一个 layersize 文件，该文件主要记录镜像层内部文件内容的总大小。既然谈到了镜像 json 文件，为了给下文铺垫，以下贴出 ubuntu:14.04 中空镜像层 d2a0ecffe6fa 的 json 文件：



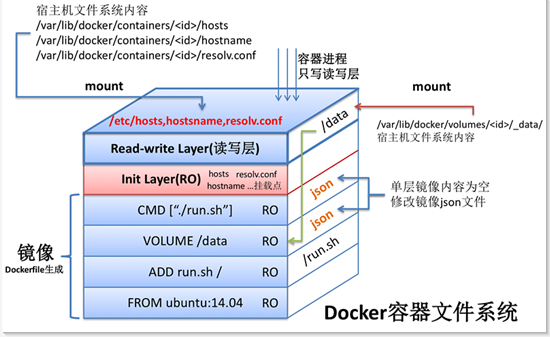
Docker镜像存储，就和大家一起先看到这。同时介绍Docker镜像的基本知识也告一段落。以下我们进入此次分享的第二部分。

**第二部分 Dockerfile、Docker镜像和Docker容器的关系**

Dockerfile 是软件的原材料，Docker 镜像是软件的交付品，而 Docker 容器则可以认为是软件的运行态。从应用软件的角度来看，Dockerfile、Docker 镜像与 Docker 容器分别代表软件的三个不同阶段，Dockerfile 面向开发，Docker 镜像成为交付标准，Docker 容器则涉及部署与运维，三者缺一不可，合力充当 Docker 体系的基石。

简单来讲，Dockerfile构建出Docker镜像，通过Docker镜像运行Docker容器。

我们可以从Docker容器的角度，来反推三者的关系。首先可以来看下图：



我们假设这个容器的镜像通过以下Dockerfile构建而得：

FROM ubuntu:14.04

ADD run.sh /

VOLUME /data

CMD ["./run.sh"]

**2.1 Dockerfile与Docker镜像**

首先，我们结合上图来看看Dockerfile与Docker镜像之间的关系。

FROM ubuntu:14.04：设置基础镜像，此时会使用基础镜像 ubuntu:14.04 的所有镜像层，为简单起见，图中将其作为一个整体展示。

ADD run.sh /：将 Dockerfile 所在目录的文件 run.sh 加至镜像的根目录，此时新一层的镜像只有一项内容，即根目录下的 run.sh。

VOLUME /data：设定镜像的 VOLUME，此 VOLUME 在容器内部的路径为 /data。需要注意的是，此时并未在新一层的镜像中添加任何文件，即构建出的磁层镜像中文件为空，但更新了镜像的 json 文件，以便通过此镜像启动容器时获取这方面的信息。

CMD ["./run.sh"]：设置镜像的默认执行入口，此命令同样不会在新建镜像中添加任何文件，仅仅在上一层镜像 json 文件的基础上更新新建镜像的 json 文件。

因此，通过以上分析，以上的Dockerfile可以构建出一个新的镜像，包含4个镜像层，每一条命令会和一个镜像层对应，镜像之间会存在父子关系。图中很清楚的表明了这些关系。

**2.2 Docker镜像与Docker容器的关系**

Docker镜像是Docker容器运行的基础，没有Docker镜像，就不可能有Docker容器，这也是Docker的设计原则之一。

可以理解的是：Docker镜像毕竟是镜像，属于静态的内容；而Docker容器就不一样了，容器属于动态的内容。动态的内容，大家很容易联想到进程，内存，CPU等之类的东西。的确，Docker容器作为动态的内容，都会包含这些。

为了便于理解，大家可以把Docker容器，理解为一个或多个运行进程，而这些运行进程将占有相应的内存，相应的CPU计算资源，相应的虚拟网络设备以及相应的文件系统资源。而Docker容器所占用的文件系统资源，则通过Docker镜像的镜像层文件来提供。

那么作为静态的镜像，如何才有能力转化为一个动态的Docker容器呢？此时，我们可以想象：第一，转化的依据是什么；第二，由谁来执行这个转化操作。

其实，转化的依据是每个镜像的json文件，Docker可以通过解析Docker镜像的json的文件，获知应该在这个镜像之上运行什么样的进程，应该为进程配置怎么样的环境变量，此时也就实现了静态向动态的转变。

谁来执行这个转化工作？答案是Docker守护进程。也许大家早就理解这样一句 话：Docker容器实质上就是一个或者多个进程，而容器的父进程就是Docker守护进程。这样的，转化工作的执行就不难理解了：Docker守护进程 手握Docker镜像的json文件，为容器配置相应的环境，并真正运行Docker镜像所指定的进程，完成Docker容器的真正创建。

Docker容器运行起来之后，Docker镜像json文件就失去作用了。此时Docker镜像的绝大部分作用就是：为Docker容器提供一个文件系统的视角，供容器内部的进程访问文件资源。

再次回到上图，我们再来看看容器和镜像之间的一些特殊关系。首先，之前已经提及Docker镜像是分层管理的，管理Docker容器的时候，Docker镜像仍然是分层管理的。由于此时动态的容器中已经存在进程，进程就会对文件系统视角内的文件进行读写操作，因此，就会涉及一个问题：容器是否会篡改Docker镜像的内容？

答案自然是不会的。统一来讲，正如上图，所有的Docker镜像层对于容器来说，都是只读的，容器对于文件的写操作绝对不会作用在镜像中。

既然如此，实现的原理就很重要，究其根本：Docker守护进程会在Docker镜像的 最上层之上，再添加一个可读写层，容器所有的写操作都会作用到这一层中。而如果Docker容器需要写底层Docker镜像中的文件，那么此时就会涉及一 个叫Copy－on－Write的机制，即aufs等联合文件系统保证：首先将此文件从Docker镜像层中拷贝至最上层的可读写层，然后容器进程再对读 写层中的副本进行写操纵。对于容器进程来讲，它只能看到最上层的文件。

Docker镜像中存储的都是一些静态文件。这些文件原则上应该和容器具体信息以及主机信息完全解藕。那么Docker容器中不存在Docker镜像中的内容主要有以下几点：

1./proc以及/sys等虚拟文件系统的内容

2.容器的hosts文件，hostname文件以及resolv.conf文件，这些事具体环境的信息，原则上的确不应该被打入镜像。

3.容器的Volume路径，这部分的视角来源于从宿主机上挂载到容器内部的路径

4.部分的设备文件