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。

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

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。