[Docker核心概念总结 1](#_Toc169085173)

[一 认识容器 1](#_Toc169085174)

[1.1 什么是容器? 2](#_Toc169085175)

[1.2 图解物理机,虚拟机与容器 2](#_Toc169085176)

[二 再来谈谈 Docker 的一些概念 2](#_Toc169085177)

[2.1 什么是 Docker? 2](#_Toc169085178)

[2.2 Docker 思想 3](#_Toc169085179)

[2.3 Docker 容器的特点 3](#_Toc169085180)

[2.4 为什么要用 Docker ? 3](#_Toc169085181)

[三 容器 VS 虚拟机 3](#_Toc169085182)

[3.1 两者对比图 4](#_Toc169085183)

[3.2 容器与虚拟机总结 4](#_Toc169085184)

[3.3 容器与虚拟机两者是可以共存的 5](#_Toc169085185)

[四 Docker 基本概念（⭐） 5](#_Toc169085186)

[4.1 镜像(Image):一个特殊的文件系统 5](#_Toc169085187)

[4.2 容器(Container):镜像运行时的实体 6](#_Toc169085188)

[4.3 仓库(Repository):集中存放镜像文件的地方 6](#_Toc169085189)

[4.4 数据卷 7](#_Toc169085190)

[4.5 Dockerfile:构建镜像 9](#_Toc169085191)

[五 常见命令 13](#_Toc169085192)

[5.1 常用命令 13](#_Toc169085193)

[5.2 镜像命令 13](#_Toc169085194)

[5.3 容器命令 14](#_Toc169085195)

[5.4 数据卷命令 15](#_Toc169085196)

[六 Build Ship and Run 15](#_Toc169085197)

[七 简单了解一下 Docker 底层原理 15](#_Toc169085198)

[7.1 虚拟化技术 15](#_Toc169085199)

[7.2 Docker 基于 LXC 虚拟容器技术 16](#_Toc169085200)

[7.2 Docker 基于 LXC 虚拟容器技术 16](#_Toc169085201)

[八 推荐阅读和参考 17](#_Toc169085202)

[九 Dockerfile镜像脚本 17](#_Toc169085203)

[Dockerfile简介 17](#_Toc169085204)

# Docker核心概念总结

https://b11et3un53m.feishu.cn/wiki/MWQIw4Zvhil0I5ktPHwcoqZdnec

## [一 认识容器](#一-认识容器)

**Docker 是世界领先的软件容器平台**，所以想要搞懂 Docker 的概念我们必须先从容器开始说起。

### [1.1 什么是容器?](#_1-1-什么是容器)

[**先来看看容器较为官方的解释**](#先来看看容器较为官方的解释)

**一句话概括容器：容器就是将软件打包成标准化单元，以用于开发、交付和部署。**

* **容器镜像是轻量的、可执行的独立软件包** ，包含软件运行所需的所有内容：代码、运行时环境、系统工具、系统库和设置。
* **容器化软件适用于基于 Linux 和 Windows 的应用，在任何环境中都能够始终如一地运行。**
* **容器赋予了软件独立性**，使其免受外在环境差异（例如，开发和预演环境的差异）的影响，从而有助于减少团队间在相同基础设施上运行不同软件时的冲突。

[**再来看看容器较为通俗的解释**](#再来看看容器较为通俗的解释)

**如果需要通俗地描述容器的话，我觉得容器就是一个存放东西的地方，就像书包可以装各种文具、衣柜可以放各种衣服、鞋架可以放各种鞋子一样。我们现在所说的容器存放的东西可能更偏向于应用比如网站、程序甚至是系统环境。**

### [1.2 图解物理机,虚拟机与容器](https://javaguide.cn/tools/docker/docker-intro.html#_1-2-%E5%9B%BE%E8%A7%A3%E7%89%A9%E7%90%86%E6%9C%BA-%E8%99%9A%E6%8B%9F%E6%9C%BA%E4%B8%8E%E5%AE%B9%E5%99%A8)

容器虚拟化的是操作系统而不是硬件，容器之间是共享同一套操作系统资源的。虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统。因此容器的隔离级别会稍低一些

## [二 再来谈谈 Docker 的一些概念](#二-再来谈谈-docker-的一些概念)

### [2.1 什么是 Docker?](#_2-1-什么是-docker)

说实话关于 Docker 是什么并太好说，下面我通过四点向你说明 Docker 到底是个什么东西。

* **Docker 是世界领先的软件容器平台。**
* **Docker** 使用 Google 公司推出的 **Go 语言** 进行开发实现，基于 **Linux 内核** 提供的 CGroup 功能和 namespace 来实现的，以及 AUFS 类的 **UnionFS** 等技术，**对进程进行封装隔离，属于操作系统层面的虚拟化技术。** 由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。
* **Docker 能够自动执行重复性任务，例如搭建和配置开发环境，从而解放了开发人员以便他们专注在真正重要的事情上：构建杰出的软件。**
* **用户可以方便地创建和使用容器，把自己的应用放入容器。容器还可以进行版本管理、复制、分享、修改，就像管理普通的代码一样。**

### [2.2 Docker 思想](#_2-2-docker-思想)

* **集装箱**
* **标准化：** ① 运输方式 ② 存储方式 ③ API 接口
* **隔离**

### [2.3 Docker 容器的特点](#_2-3-docker-容器的特点)

* **轻量** : 在一台机器上运行的多个 Docker 容器可以共享这台机器的操作系统内核；它们能够迅速启动，只需占用很少的计算和内存资源。镜像是通过文件系统层进行构造的，并共享一些公共文件。这样就能尽量降低磁盘用量，并能更快地下载镜像。
* **标准** : Docker 容器基于开放式标准，能够在所有主流 Linux 版本、Microsoft Windows 以及包括 VM、裸机服务器和云在内的任何基础设施上运行。
* **安全** : Docker 赋予应用的隔离性不仅限于彼此隔离，还独立于底层的基础设施。Docker 默认提供最强的隔离，因此应用出现问题，也只是单个容器的问题，而不会波及到整台机器。

### [2.4 为什么要用 Docker ?](#_2-4-为什么要用-docker)

* **Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 “这段代码在我机器上没问题啊” 这类问题；——一致的运行环境**
* **可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。——更快速的启动时间**
* **避免公用的服务器，资源会容易受到其他用户的影响。——隔离性**
* **善于处理集中爆发的服务器使用压力；——弹性伸缩，快速扩展**
* **可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。——迁移方便**
* **使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。——持续交付和部署**

## [三 容器 VS 虚拟机](#三-容器-vs-虚拟机)

**每当说起容器，我们不得不将其与虚拟机做一个比较。就我而言，对于两者无所谓谁会取代谁，而是两者可以和谐共存。**

简单来说：**容器和虚拟机具有相似的资源隔离和分配优势，但功能有所不同，因为容器虚拟化的是操作系统，而不是硬件，因此容器更容易移植，效率也更高。**

### [3.1 两者对比图](#_3-1-两者对比图)

传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。



### [3.2 容器与虚拟机总结](https://javaguide.cn/tools/docker/docker-intro.html#_3-2-%E5%AE%B9%E5%99%A8%E4%B8%8E%E8%99%9A%E6%8B%9F%E6%9C%BA%E6%80%BB%E7%BB%93)

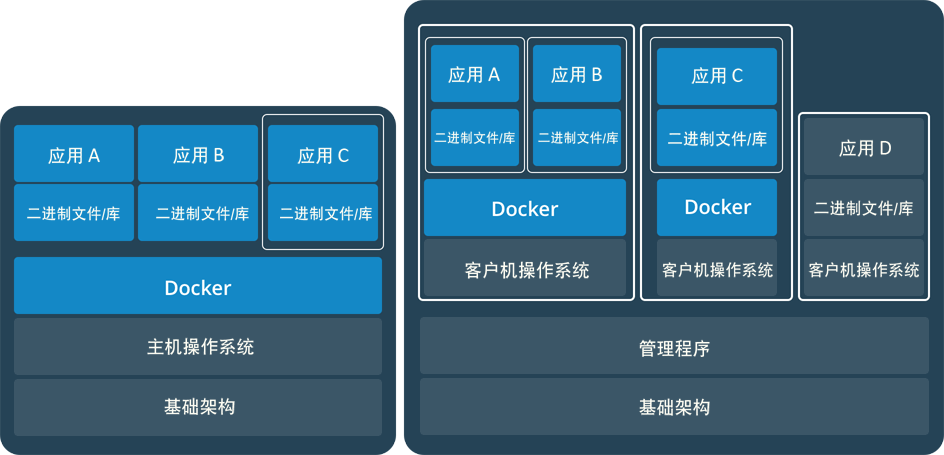


* **容器是一个应用层抽象，用于将代码和依赖资源打包在一起。** **多个容器可以在同一台机器上运行，共享操作系统内核，但各自作为独立的进程在用户空间中运行** 。与虚拟机相比， **容器占用的空间较少**（容器镜像大小通常只有几十兆），**瞬间就能完成启动** 。
* **虚拟机 (VM) 是一个物理硬件层抽象，用于将一台服务器变成多台服务器。** 管理程序允许多个 VM 在一台机器上运行。每个 VM 都包含一整套操作系统、一个或多个应用、必要的二进制文件和库资源，因此 **占用大量空间** 。而且 VM **启动也十分缓慢** 。

通过 Docker 官网，我们知道了这么多 Docker 的优势，但是大家也没有必要完全否定虚拟机技术，因为两者有不同的使用场景。**虚拟机更擅长于彻底隔离整个运行环境**。例如，云服务提供商通常采用虚拟机技术隔离不同的用户。而 **Docker 通常用于隔离不同的应用**，例如前端，后端以及数据库。

### [3.3 容器与虚拟机两者是可以共存的](#_3-3-容器与虚拟机两者是可以共存的)

对于两者无所谓谁会取代谁，而是两者可以和谐共存。



## [四 Docker 基本概念](#四-docker-基本概念)（⭐）

**Docker 中有非常重要的三个基本概念，理解了这三个概念，就理解了 Docker 的整个生命周期。**

* **镜像（Image）**
* **容器（Container）**
* **仓库（Repository）**

理解了这三个概念，就理解了 Docker 的整个生命周期

### [4.1 镜像(Image):一个特殊的文件系统](#_4-1-镜像-image-一个特殊的文件系统)

**操作系统分为内核和用户空间**。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。而 Docker 镜像（Image），就相当于是一个 root 文件系统。

**Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。** 镜像不包含任何动态数据，其内容在构建之后也不会被改变。

Docker 设计时，就充分利用 **Union FS** 的技术，将其设计为**分层存储的架构** 。镜像实际是由多层文件系统联合组成。

**镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。** 比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

### [4.2 容器(Container):镜像运行时的实体](#_4-2-容器-container-镜像运行时的实体)

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的 类 和 实例 一样，镜像是静态的定义，**容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等** 。

**容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的 命名空间。前面讲过镜像使用的是分层存储，容器也是如此。**

**容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。**

按照 Docker 最佳实践的要求，**容器不应该向其存储层内写入任何数据** ，容器存储层要保持无状态化。**所有的文件写入操作，都应该使用数据卷（Volume）、或者绑定宿主目录**，在这些位置的读写会跳过容器存储层，直接对宿主(或网络存储)发生读写，其性能和稳定性更高。数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此， **使用数据卷后，容器可以随意删除、重新 run ，数据却不会丢失。**

### [4.3 仓库(Repository):集中存放镜像文件的地方](#_4-3-仓库-repository-集中存放镜像文件的地方)

镜像构建完成后，可以很容易的在当前宿主上运行，但是， **如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。**

一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。所以说：**镜像仓库是 Docker 用来集中存放镜像文件的地方类似于我们之前常用的代码仓库。**

通常，**一个仓库会包含同一个软件不同版本的镜像**，而**标签就常用于对应该软件的各个版本** 。我们可以通过<仓库名>:<标签>的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 latest 作为默认标签.。

**这里补充一下 Docker Registry 公开服务和私有 Docker Registry 的概念：**

**Docker Registry 公开服务** 是开放给用户使用、允许用户管理镜像的 Registry 服务。一般这类公开服务允许用户免费上传、下载公开的镜像，并可能提供收费服务供用户管理私有镜像。

最常使用的 Registry 公开服务是官方的 **Docker Hub** ，这也是默认的 Registry，并拥有大量的高质量的官方镜像，网址为：[https://hub.docker.com/open in new window](https://hub.docker.com/) 。

在 Docker Hub 的搜索结果中，有几项关键的信息有助于我们选择合适的镜像：

* **OFFICIAL Image**：代表镜像为 Docker 官方提供和维护，相对来说稳定性和安全性较高。
* **Stars**：和点赞差不多的意思，类似 GitHub 的 Star。
* **Downloads**：代表镜像被拉取的次数，基本上能够表示镜像被使用的频度。

当然，除了直接通过 Docker Hub 网站搜索镜像这种方式外，我们还可以通过 docker search 这个命令搜索 Docker Hub 中的镜像，搜索的结果是一致的。

除了使用公开服务外，用户还可以在 **本地搭建私有 Docker Registry**。Docker 官方提供了 Docker Registry 镜像，可以直接使用做为私有 Registry 服务。开源的 Docker Registry 镜像只提供了 Docker Registry API 的服务端实现，足以支持 docker 命令，不影响使用。但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。

### [4.4 数据卷](#_4-3-仓库-repository-集中存放镜像文件的地方)

**数据卷（volume）**是一个虚拟目录，是**容器内目录**与**宿主机目录**之间映射的桥梁。

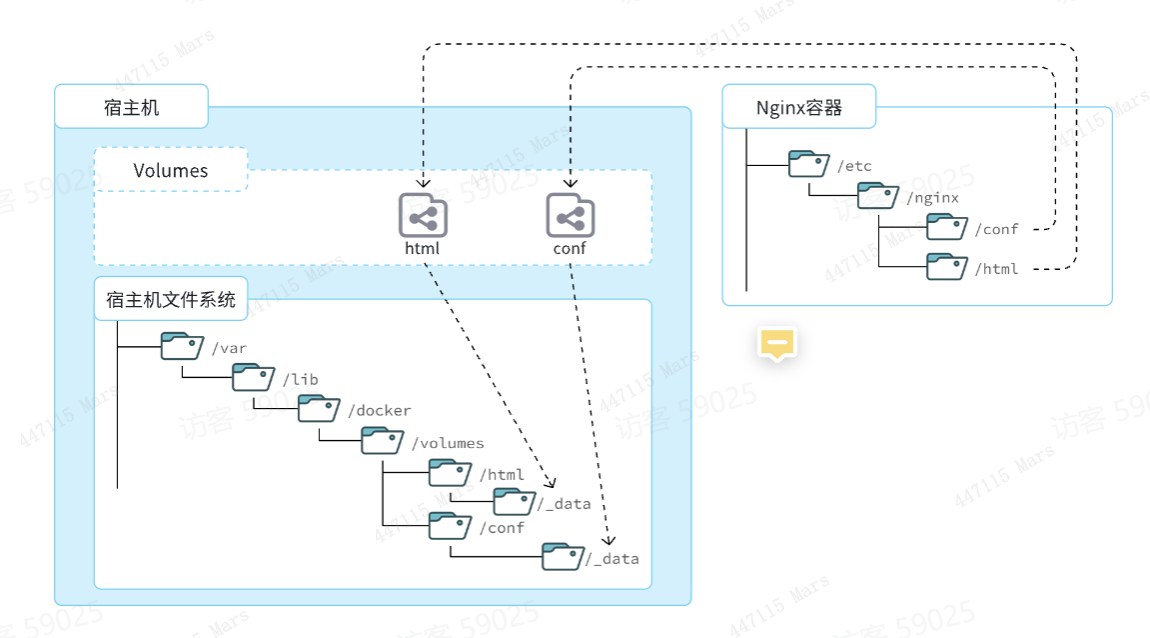
以Nginx为例，我们知道Nginx中有两个关键的目录：

- html：放置一些静态资源

- conf：放置配置文件

如果我们要让Nginx代理我们的静态资源，最好是放到html目录；如果我们要修改Nginx的配置，最好是找到conf下的nginx.conf文件。

但遗憾的是，容器运行的Nginx所有的文件都在容器内部。所以我们必须利用数据卷将两个目录与宿主机目录关联，方便我们操作。如图：



在上图中：

* 我们创建了两个数据卷：conf、html
* Nginx容器内部的conf目录和html目录分别与两个数据卷关联。
* 而数据卷conf和html分别指向了宿主机的/var/lib/docker/volumes/conf/\_data目录和/var/lib/docker/volumes/html/\_data目录

这样以来，容器内的conf和html目录就 与宿主机的conf和html目录关联起来，我们称为**挂载**。此时，我们操作宿主机的/var/lib/docker/volumes/html/\_data就是在操作容器内的/usr/share/nginx/html/\_data目录。只要我们将静态资源放入宿主机对应目录，就可以被Nginx代理了。

**/var/lib/docker/volumes这个目录就是默认的存放所有容器数据卷的目录，其下再根据数据卷名称创建新目录，格式为/数据卷名/\_data。**

**为什么不让容器目录直接指向宿主机目录呢**？

* 因为直接指向宿主机目录就与宿主机强耦合了，如果切换了环境，宿主机目录就可能发生改变了。由于容器一旦创建，目录挂载就无法修改，这样容器就无法正常工作了。
* 但是容器指向数据卷，一个逻辑名称，而数据卷再指向宿主机目录，就不存在强耦合。如果宿主机目录发生改变，只要改变数据卷与宿主机目录之间的映射关系即可。

可以发现，数据卷的目录结构较深，如果我们去操作数据卷目录会不太方便。在很多情况下，我们会直接将容器目录与宿主机指定目录挂载。挂载语法与数据卷类似：

# 挂载本地目录

-v 本地目录:容器内目录

# 挂载本地文件

-v 本地文件:容器内文件

**注意**：本地目录或文件必须以 / 或 ./开头，如果直接以名字开头，会被识别为数据卷名而非本地目录名。

例如：

-v mysql:/var/lib/mysql # 会被识别为一个数据卷叫mysql，运行时会自动创建这个数据卷

-v ./mysql:/var/lib/mysql # 会被识别为当前目录下的mysql目录，运行时如果不存在会创建目录

### [4.5 Dockerfile:构建镜像](#_4-1-镜像-image-一个特殊的文件系统)

要想自己构建镜像，必须先了解镜像的结构。

之前我们说过，镜像之所以能让我们快速跨操作系统部署应用而忽略其运行环境、配置，就是因为镜像中包含了程序运行需要的系统函数库、环境、配置、依赖。

因此，自定义镜像本质就是依次准备好程序运行的基础环境、依赖、应用本身、运行配置等文件，并且打包而成。

举个例子，我们要从0部署一个Java应用，大概流程是这样：

* 准备一个linux服务（CentOS或者Ubuntu均可）
* 安装并配置JDK
* 上传Jar包
* 运行jar包

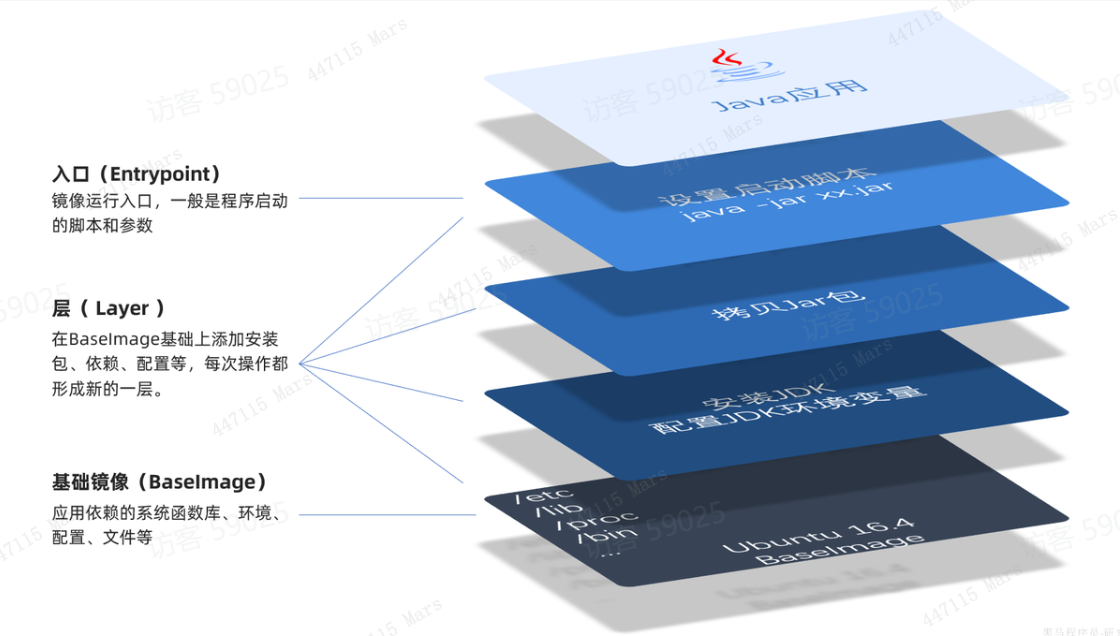
那因此，我们打包镜像也是分成这么几步：

* 准备Linux运行环境（java项目并不需要完整的操作系统，仅仅是基础运行环境即可）
* 安装并配置JDK
* 拷贝jar包
* 配置启动脚本

上述步骤中的每一次操作其实都是在生产一些文件（系统运行环境、函数库、配置最终都是磁盘文件），所以**镜像就是一堆文件的集合**。

但需要注意的是，镜像文件不是随意堆放的，而是按照操作的步骤分层叠加而成，每一层形成的文件都会单独打包并标记一个唯一id，称为**Layer**（**层**）。这样，如果我们构建时用到的某些层其他人已经制作过，就可以直接拷贝使用这些层，而不用重复制作。

例如，第一步中需要的Linux运行环境，通用性就很强，所以Docker官方就制作了这样的只包含Linux运行环境的镜像。我们在制作java镜像时，就无需重复制作，直接使用Docker官方提供的CentOS或Ubuntu镜像作为基础镜像。然后再搭建其它层即可，这样逐层搭建，最终整个Java项目的镜像结构如图所示：



由于制作镜像的过程中，需要逐层处理和打包，比较复杂，所以Docker就提供了自动打包镜像的功能。我们只需要将打包的过程，每一层要做的事情用固定的语法写下来，交给Docker去执行即可。

而这种记录镜像结构的文件就称为**Dockerfile**，其对应的语法可以参考官方文档：

<https://docs.docker.com/engine/reference/builder/>

其中的语法比较多，比较常用的有：

| **指令** | **说明** | **示例** |
| --- | --- | --- |
| **FROM** | 指定基础镜像 | FROM centos:6 |
| **ENV** | 设置环境变量，可在后面指令使用 | ENV key value |
| **COPY** | 拷贝本地文件到镜像的指定目录 | COPY ./xx.jar /tmp/app.jar |
| **RUN** | 执行Linux的shell命令，一般是安装过程的命令 | RUN yum install gcc |
| **EXPOSE** | 指定容器运行时监听的端口，是给镜像使用者看的 | EXPOSE 8080 |
| **ENTRYPOINT** | 镜像中应用的启动命令，容器运行时调用 | ENTRYPOINT java -jar xx.jar |

例如，要基于Ubuntu镜像来构建一个Java应用，其Dockerfile内容如下：



同学们思考一下：以后我们会有很多很多java项目需要打包为镜像，他们都需要Linux系统环境、JDK环境这两层，只有上面的3层不同（因为jar包不同）。如果每次制作java镜像都重复制作前两层镜像，是不是很麻烦。

所以，就有人提供了基础的系统加JDK环境，我们在此基础上制作java镜像，就可以省去JDK的配置了：



首先，我们将课前资料提供的docker-demo.jar包以及Dockerfile拷贝到虚拟机的/root/demo目录：

然后，执行命令，构建镜像：

命令说明：

* docker build : 就是构建一个docker镜像
* -t docker-demo:1.0 ：-t参数是指定镜像的名称（repository和tag）
* . : 最后的点是指构建时Dockerfile所在路径，由于我们进入了demo目录，所以指定的是.代表当前目录，也可以直接指定Dockerfile目录：

## [五 常见命令](#五-常见命令)

### [5.1 常用命令](#_5-1-基本命令)

docker version # 查看docker版本

docker info # 查看docker信息

docker build #由dockerfile构建自定义对象

docker save # 将镜像从本地镜像下载到本地内存

docker load # 将镜像从本地内存上传到本地镜像

### [5.2 镜像](#_5-2-拉取镜像)命令

**docker search 镜像名称 搜索镜像**

//docker search mysql 查看mysql相关镜像

**docker pull 镜像名称 拉取镜像**

// docker pull mysql:5.7 # 拉取mysql镜像

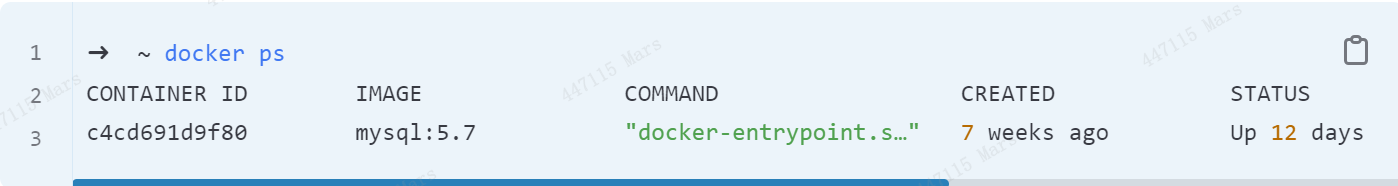
**docker push 镜像名称 上传镜像**

**docker images 查看本地所有镜像**

//查看所有已下载镜像，等价于：docker image ls 命令

**docker rmi -f 镜像名称 删除镜像**

//通过 docker rmi [image] （等价于docker image rm [image]）删除镜像之前首先要确保这个镜像没有被容器引用（可以通过标签名称或者镜像 ID 删除）。通过我们前面讲的docker ps命令即可查看。



可以看到 mysql 正在被 id 为 c4cd691d9f80 的容器引用，我们需要首先通过 docker stop c4cd691d9f80 或者 docker stop mysql暂停这个容器。

然后查看 mysql 镜像的 id



通过 IMAGE ID 或者 REPOSITORY 名字即可删除

docker rmi f6509bac4980 *# 或者 docker rmi mysql*

**docker image prune** # 清理临时的、没有被使用的镜像文件。-a, --all: 删除所有没有用的镜像，而不仅仅是临时文件；

### [5.3 容器命令](#_5-3-删除镜像)

docker container ls # 查看所有容器

docker ps 查看运行的容器

docker ps -a 查询所有容器

docker logs 查看容器日志

docker run -i 镜像名称:标签 运行容器（默认是前台运行）

// docker run -i nginx:版本号（最新版可以不加：版本号）

常用的参数：

-i：运行容器用什么镜像

-d：后台守方式运行（守护式）

// docker run -di nginx

--name：给容器添加名称

-p：端口映射，公开容器端口给当前宿主机

// docker run -di -p 90:80 nginx 公开原有端口号到90端口

-e key = value：环境变量

-v：挂载目录

docker exec -it 容器ID/容器名称 /bin/bash 进入容器内部

docker start/stop/restart 容器名称/ID 启动/停止/重启容器

docker rm -f 容器名称/ID 删除容器

//-f为强制删除，一般不加

### [5.4 数据卷命令](#_5-3-删除镜像)



## [六 Build Ship and Run](#六-build-ship-and-run)

**Docker 的概念以及常见命令基本上已经讲完，我们再来谈谈：Build, Ship, and Run。**

如果你搜索 Docker 官网，会发现如下的字样：**“Docker - Build, Ship, and Run Any App, Anywhere”**。那么 Build, Ship, and Run 到底是在干什么呢？

* **Build（构建镜像）**：镜像就像是集装箱包括文件以及运行环境等等资源。
* **Ship（运输镜像）**：主机和仓库间运输，这里的仓库就像是超级码头一样。
* **Run （运行镜像）**：运行的镜像就是一个容器，容器就是运行程序的地方。

**Docker 运行过程也就是去仓库把镜像拉到本地，然后用一条命令把镜像运行起来变成容器。所以，我们也常常将 Docker 称为码头工人或码头装卸工，这和 Docker 的中文翻译搬运工人如出一辙。**

## [七 简单了解一下 Docker 底层原理](#七-简单了解一下-docker-底层原理)

### [7.1 虚拟化技术](#_7-1-虚拟化技术)

首先，Docker **容器虚拟化**技术为基础的软件，那么什么是虚拟化技术呢？

简单点来说，虚拟化技术可以这样定义：

虚拟化技术是一种资源管理技术，是将计算机的各种[实体资源open in new window](https://zh.wikipedia.org/wiki/%E8%A8%88%E7%AE%97%E6%A9%9F%E7%A7%91%E5%AD%B8))（[CPUopen in new window](https://zh.wikipedia.org/wiki/CPU)、[内存open in new window](https://zh.wikipedia.org/wiki/%E5%86%85%E5%AD%98)、[磁盘空间open in new window](https://zh.wikipedia.org/wiki/%E7%A3%81%E7%9B%98%E7%A9%BA%E9%97%B4)、[网络适配器open in new window](https://zh.wikipedia.org/wiki/%E7%B6%B2%E8%B7%AF%E9%81%A9%E9%85%8D%E5%99%A8)等），予以抽象、转换后呈现出来并可供分割、组合为一个或多个电脑配置环境。由此，打破实体结构间的不可切割的障碍，使用户可以比原本的配置更好的方式来应用这些电脑硬件资源。这些资源的新虚拟部分是不受现有资源的架设方式，地域或物理配置所限制。一般所指的虚拟化资源包括计算能力和数据存储。

### [7.2 Docker 基于 LXC 虚拟容器技术](#_7-2-docker-基于-lxc-虚拟容器技术)

Docker 技术是基于 LXC（Linux container- Linux 容器）虚拟容器技术的。

虚拟化技术是一种资源管理技术，是将计算机的各种[实体资源open in new window](https://zh.wikipedia.org/wiki/%E8%A8%88%E7%AE%97%E6%A9%9F%E7%A7%91%E5%AD%B8))（[CPUopen in new window](https://zh.wikipedia.org/wiki/CPU)、[内存open in new window](https://zh.wikipedia.org/wiki/%E5%86%85%E5%AD%98)、[磁盘空间open in new window](https://zh.wikipedia.org/wiki/%E7%A3%81%E7%9B%98%E7%A9%BA%E9%97%B4)、[网络适配器open in new window](https://zh.wikipedia.org/wiki/%E7%B6%B2%E8%B7%AF%E9%81%A9%E9%85%8D%E5%99%A8)等），予以抽象、转换后呈现出来并可供分割、组合为一个或多个电脑配置环境。由此，打破实体结构间的不可切割的障碍，使用户可以比原本的配置更好的方式来应用这些电脑硬件资源。这些资源的新虚拟部分是不受现有资源的架设方式，地域或物理配置所限制。一般所指的虚拟化资源包括计算能力和数据存储。

### [7.2 Docker 基于 LXC 虚拟容器技术](#_7-2-docker-基于-lxc-虚拟容器技术)

Docker 技术是基于 LXC（Linux container- Linux 容器）虚拟容器技术的。

LXC，其名称来自 Linux 软件容器（Linux Containers）的缩写，一种操作系统层虚拟化（Operating system–level virtualization）技术，为 Linux 内核容器功能的一个用户空间接口。它将应用软件系统打包成一个软件容器（Container），内含应用软件本身的代码，以及所需要的操作系统核心和库。通过统一的名字空间和共用 API 来分配不同软件容器的可用硬件资源，创造出应用程序的独立沙箱运行环境，使得 Linux 用户可以容易的创建和管理系统或应用容器。

LXC 技术主要是借助 Linux 内核中提供的 CGroup 功能和 namespace 来实现的，通过 LXC 可以为软件提供一个独立的操作系统运行环境。

**cgroup 和 namespace 介绍：**

 **namespace 是 Linux 内核用来隔离内核资源的方式。** 通过 namespace 可以让一些进程只能看到与自己相关的一部分资源，而另外一些进程也只能看到与它们自己相关的资源，这两拨进程根本就感觉不到对方的存在。具体的实现方式是把一个或多个进程的相关资源指定在同一个 namespace 中。Linux namespaces 是对全局系统资源的一种封装隔离，使得处于不同 namespace 的进程拥有独立的全局系统资源，改变一个 namespace 中的系统资源只会影响当前 namespace 里的进程，对其他 namespace 中的进程没有影响。

（以上关于 namespace 介绍内容来自[https://www.cnblogs.com/sparkdev/p/9365405.htmlopen in new window](https://www.cnblogs.com/sparkdev/p/9365405.html) ，更多关于 namespace 的内容可以查看这篇文章 ）。

 **CGroup 是 Control Groups 的缩写，是 Linux 内核提供的一种可以限制、记录、隔离进程组 (process groups) 所使用的物理资源 (如 cpu memory i/o 等等) 的机制。**

（以上关于 CGroup 介绍内容来自 [https://www.ibm.com/developerworks/cn/linux/1506\_cgroup/index.htmlopen in new window](https://www.ibm.com/developerworks/cn/linux/1506_cgroup/index.html) ，更多关于 CGroup 的内容可以查看这篇文章 ）。

**cgroup 和 namespace 两者对比：**

两者都是将进程进行分组，但是两者的作用还是有本质区别。namespace 是为了隔离进程组之间的资源，而 cgroup 是为了对一组进程进行统一的资源监控和限制。

## [八 推荐阅读和参考](#九-推荐阅读)

* [10 分钟看懂 Docker 和 K8Sopen in new window](https://zhuanlan.zhihu.com/p/53260098)
* [从零开始入门 K8s：详解 K8s 容器基本概念open in new window](https://www.infoq.cn/article/te70FlSyxhltL1Cr7gzM)
* [Linux Namespace 和 Cgroupopen in new window](https://segmentfault.com/a/1190000009732550)
* [LXC vs Docker: Why Docker is Betteropen in new window](https://www.upguard.com/articles/docker-vs-lxc)
* [CGroup 介绍、应用实例及原理描述](https://www.ibm.com/developerworks/cn/linux/1506_cgroup/index.html)

## [九 Dockerfile镜像脚本](#九-推荐阅读)

### Dockerfile简介

Dockerfile其实就是我们用来构建Docker镜像的源码，当然这不是所谓的编程源码，而是一些命令的组合，只要理解它的逻辑和语法格式，就可以编写Dockerfile了。

简单点说，Dockerfile的作用：它可以让用户个性化定制Docker镜像。因为工作环境中的需求各式各样，网络上的镜像很难满足实际的需求。Dockerfile常见命令

