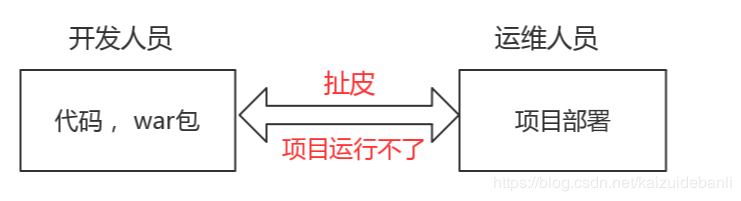
# docker

## 什么是docker

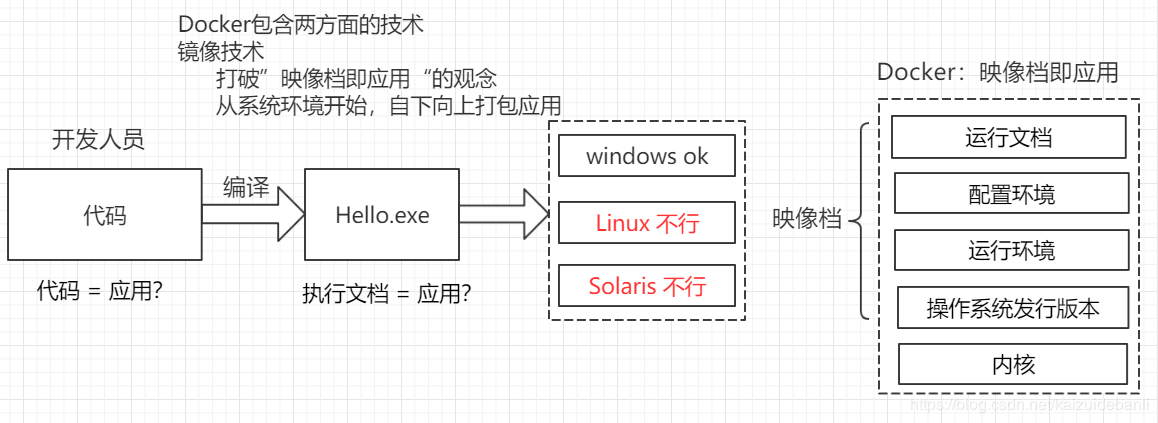
### docker引入？

    一款产品从开发到上线，从操作系统，到运行环境，再到应用配置。作为开发+运维之间的协作我们需要关心很多东西，这也是很多互联网公司都不得不面对的问题，特别是各种版本的迭代之后，**不同版本环境的兼容**，对运维人员都是考验。



   Docker之所以发展如此迅速，也是因为它对此给出了一个标准化的解决方案。

   环境配置如此麻烦，换一台机器，就要重来一次，费力费时。很多人想到，能不能从根本上解决问题，软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。开发人员利用 Docker 可以消除协作编码时“在我的机器上可正常工作”的问题。



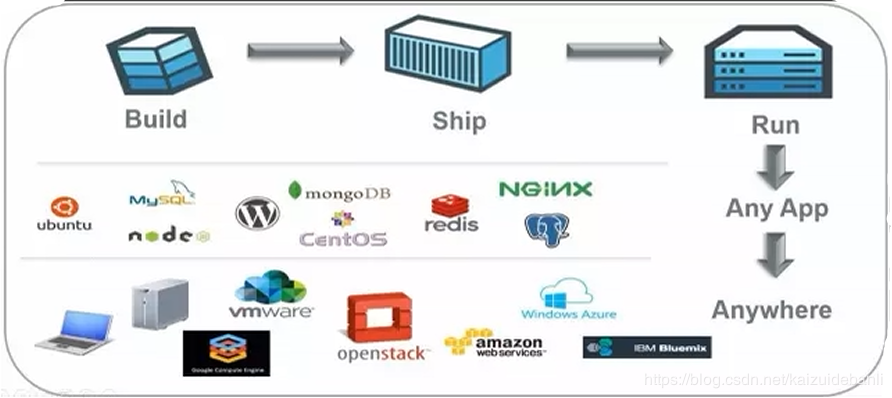
之前在服务器配置一个应用的运行环境，要安装各种软件，就拿尚硅谷电商项目的环境来说吧，Java/Tomcat/MySQL/JDBC驱动包等。安装和配置这些东西有多麻烦就不说了，它还不能跨平台。假如我们是在 Windows 上安装的这些环境，到了 Linux 又得重新装。况且就算不跨操作系统，换另一台同样操作系统的服务器，要移植应用也是非常麻烦的。

    传统上认为，软件编码开发/测试结束后，所产出的成果即是程序或是能够编译执行的二进制字节码等(java为例)。而为了让这些程序可以顺利执行，开发团队也得准备完整的部署文件，让维运团队得以部署应用程式，开发需要清楚的告诉运维部署团队，用的全部配置文件+所有软件环境。不过，即便如此，仍然常常发生部署失败的状况。Docker镜像的设计，使得Docker得以打破过去「程序即应用」的观念。透过镜像(images)将作业系统核心除外，运作应用程式所需要的系统环境，由下而上打包，达到应用程式跨平台间的无缝接轨运作。

### Docker的理念

Docker是基于Go语言实现的云开源项目。

Docker的主要目标是“Build，Ship and Run Any App,Anywhere”，也就是通过对应用组件的封装、分发、部署、运行等生命周期的管理，使用户的APP（可以是一个WEB应用或数据库应用等等）及其运行环境能够做到“一次封装，到处运行”。



  Linux 容器技术的出现就解决了这样一个问题，而 Docker 就是在它的基础上发展过来的。将应用运行在 Docker 容器上面，而 Docker 容器在任何操作系统上都是一致的，这就实现了跨平台、跨服务器。只需要一次配置好环境，换到别的机子上就可以一键部署好，大大简化了操作。

  例如，一个应用需要redis, mongoDB, MySQL三个环境，开发人员肯定是已经配置好了这个环境，那么运维人员不需要再进行安装。只需要将这三个环境分别装进三个集装箱构建出景像文件，这个镜像文件中就有三个集装箱，每个集装箱都是本地正确运行软件的一份克隆。运维人员只需要安装Docker环境就可以，把四个集装箱放在Docker上运行就可以。就好像docker图标一样，金鱼背上的集装箱。运维人员只需要将镜像文件（三个集装箱），放在docker上运行就可以了。

### 一句话概括Docker

  程序员不再是只把代码交给运维人员，运维也不需要要再10台机器上安装10次mysql, redis, hadoop当环境。现在变了，我们帮助运维人员，把本地的运行环境自下而上打包的镜像交给运维人员,镜像文件包含了各种运行所需的软件，每个软件都是一个独立的集装箱，然后放在Docker上（金鱼背上）运行即可。而运维人员，只需要安装一个Docker。

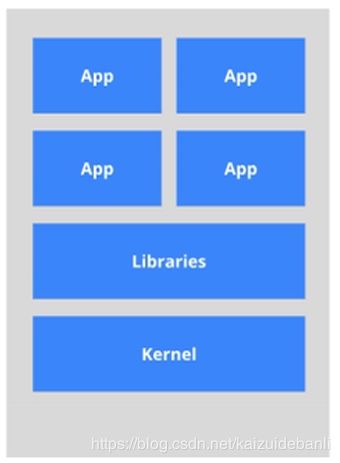
  一句话概括：Docker解决了运行环境和配置问题软件容器，方便做持续集成并有助于整体发布的容器虚拟化技术。

## docker的功能

### 之前的虚拟机技术

虚拟机（virtual machine）就是带环境安装的一种解决方案。

它可以在一种操作系统里面运行另一种操作系统，比如在Windows 系统里面运行Linux 系统。应用程序对此毫无感知，因为虚拟机看上去跟真实系统一模一样，而对于底层系统来说，虚拟机就是一个普通文件，不需要了就删掉，对其他部分毫无影响。这类虚拟机完美的运行了另一套系统，能够使应用程序，操作系统和硬件三者之间的逻辑不变。



虚拟机的缺点：

资源占用多

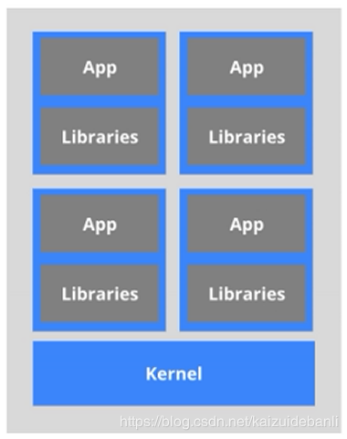
冗余步骤多

启动慢（分钟，Docker秒级）

### 容器虚拟化技术

由于前面虚拟机存在这些缺点，Linux 发展出了另一种虚拟化技术：Linux 容器（Linux Containers，缩写为 LXC）。

Linux 容器不是模拟一个完整的操作系统，而是对进程进行隔离。有了容器，就可以将软件运行所需的所有资源打包到一个隔离的容器中。容器与虚拟机不同，不需要捆绑一整套操作系统，只需要软件工作所需的库资源和设置。系统因此而变得高效轻量并保证部署在任何环境中的软件都能始终如一地运行。



比较了 Docker 和传统虚拟化方式的不同之处：

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。

每个容器之间互相隔离，每个容器有自己的文件系统 ，容器之间进程不会相互影响，能区分计算资源。

### 开发/运维（DevOps）：一次构建，导出运行

更快速的应用交付和部署

传统的应用开发完成后，需要提供一堆安装程序和配置说明文档，安装部署后需根据配置文档进行繁杂的配置才能正常运行。Docker化之后只需要交付少量容器镜像文件，在正式生产环境加载镜像并运行即可，应用安装配置在镜像里已经内置好，大大节省部署配置和测试验证时间。

更便捷的升级和扩缩容

随着微服务架构和Docker的发展，大量的应用会通过微服务方式架构，应用的开发构建将变成搭乐高积木一样，每个Docker容器将变成一块“积木”，应用的升级将变得非常容易。当现有的容器不足以支撑业务处理时，可通过镜像运行新的容器进行快速扩容，使应用系统的扩容从原先的天级变成分钟级甚至秒级。

更简单的系统运维

应用容器化运行后，生产环境运行的应用可与开发、测试环境的应用高度一致，容器会将应用程序相关的环境和状态完全封装起来，不会因为底层基础架构和操作系统的不一致性给应用带来影响，产生新的BUG。当出现程序异常时，也可以通过测试环境的相同容器进行快速定位和修复。

更高效的计算资源利用

Docker是内核级虚拟化，其不像传统的虚拟化技术一样需要额外的Hypervisor支持，所以在一台物理机上可以运行很多个容器实例，可大大提升物理服务器的CPU和内存的利用率。

## Docker仓库

### 概述

镜像构建完成后，可以很容易的在当前宿主机上运行，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。

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。

以 Ubuntu 镜像 为例，ubuntu 是仓库的名字，其内包含有不同的版本标签，如，14.04, 16.04。我们可以通过 ubuntu:14.04，或者 ubuntu:16.04 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 ubuntu，那将视为 ubuntu:latest。

仓库名经常以 两段式路径 形式出现，比如 jwilder/nginx-proxy，前者往往意味着 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对，取决于所使用的具体 Docker Registry 的软件或服务。

### 公有 Docker Registry

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。除此以外，还有 CoreOS 的 Quay.io，CoreOS 相关的镜像存储在这里；Google 的 Google Container Registry，Kubernetes 的镜像使用的就是这个服务。

由于某些原因，在国内访问这些服务可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务（Registry Mirror），这些镜像服务被称为加速器。常见的有 阿里云加速器、DaoCloud 加速器 等。使用加速器会直接从国内的地址下载 Docker Hub 的镜像，比直接从 Docker Hub 下载速度会提高很多。

国内也有一些云服务商提供类似于 Docker Hub 的公开服务。比如 时速云镜像仓库、网易云镜像服务、DaoCloud 镜像市场、阿里云镜像库 等。

### 私有 Docker Registry

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。在官方的商业化版本 Docker Trusted Registry 中，提供了这些高级功能。

除了官方的 Docker Registry 外，还有第三方软件实现了 Docker Registry API，甚至提供了用户界面以及一些高级功能。比如，VMWare Harbor 和 Sonatype Nexus。

## Ubuntu 安装 Docker

### 系统要求

Docker CE 支持以下版本的 Ubuntu 操作系统：

Artful 17.10 (Docker CE 17.11 Edge +)

Xenial 16.04 (LTS)

Trusty 14.04 (LTS)

Docker CE 可以安装在 64 位的 x86 平台或 ARM 平台上。Ubuntu 发行版中，LTS（Long-Term-Support）长期支持版本，会获得 5 年的升级维护支持，这样的版本会更稳定，因此在生产环境中推荐使用 LTS 版本,当前最新的 LTS 版本为 Ubuntu 16.04。

### 卸载旧版本

旧版本的 Docker 称为 docker 或者 docker-engine，使用以下命令卸载旧版本：

$ sudo apt-get remove docker \

docker-engine \

docker.io

### 使用脚本自动安装

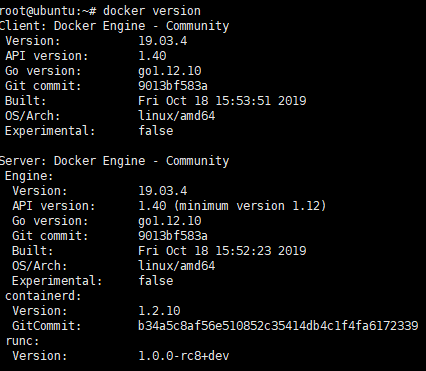
在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，Ubuntu 系统上可以使用这套脚本安装：

curl -fsSL get.docker.com -o get-docker.sh

sudo sh get-docker.sh --mirror Aliyun

安装完后检测：

docker version



### 启动docker

centos7

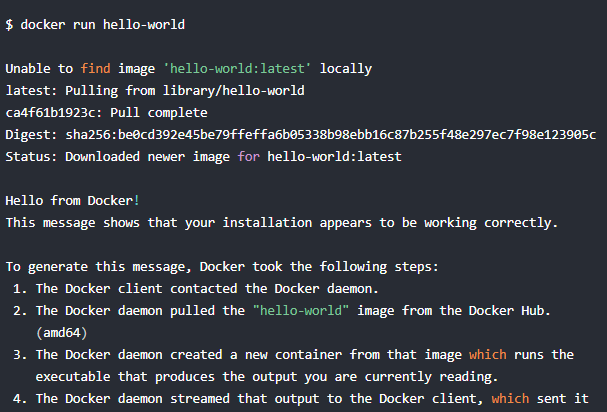
systemctl enable docker

systemctl start dockesr

Ubuntu 14.04 请使用以下命令启动：

sudo service docker start

### 测试 Docker 是否安装正确



### 配置镜像加速器

国内从 Docker Hub 拉取镜像有时会遇到困难，此时可以配置镜像加速器。Docker 官方和国内很多云服务商都提供了国内加速器服务，例如：

Docker 官方提供的中国 registry mirror

阿里云加速器

DaoCloud 加速器

Ubuntu 16.04+、Debian 8+、CentOS 7 版本

对于使用 systemd 的系统，请在 /etc/docker/daemon.json 中写入如下内容（如果文件不存在请新建该文件）

|  |
| --- |
| {  "registry-mirrors": [  "https://ceq5zz4j.mirror.aliyuncs.com"  ]  } |

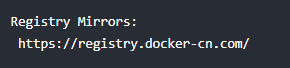
重新启动服务:

systemctl daemon-reload

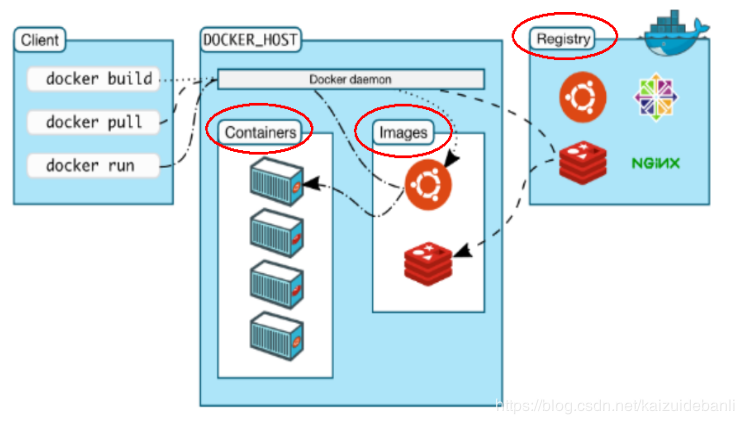
systemctl restart docker

检查加速器是否生效

配置加速器之后，如果拉取镜像仍然十分缓慢，请手动检查加速器配置是否生效，在命令行执行 docker info，如果从结果中看到了如下内容，说明配置成功。



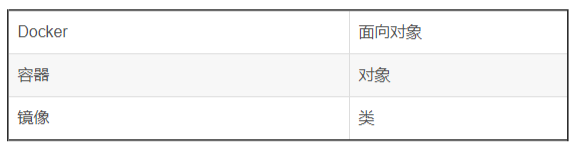
## Docker的架构图



### 镜像（image）

Docker 镜像（Image）就是一个**只读**的模板。[镜像](https://www.baidu.com/s?wd=%E9%95%9C%E5%83%8F&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)可以用来创建 Docker 容器，一个镜像可以创建很多容器。

容器与镜像的关系类似于面向对象编程中的对象与类：



|  |
| --- |
| Person person1 = new Person();  Person person2 = new Person();  Person person3 = new Person(); |

其中，person1， person2，person3相当于三个不同的容器，Person类就相当于是镜像。

### 容器（container）

Docker 利用容器（Container）独立运行的一个或一组应用。容器是用镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。

### 仓库（repository）

仓库（Repository）是集中存放镜像文件的场所。

仓库(Repository)和仓库注册服务器（Registry）是有区别的。仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是 Docker Hub(<https://hub.docker.com/>)，

存放了数量庞大的镜像供用户下载。国内的公开仓库包括阿里云 、网易云 等

### 总结

## Docker镜像

### 概述

我们都知道，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。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu 16.04 最小系统的 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。

### 分层存储

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 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。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，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。

关于镜像构建，将会在后续相关章节中做进一步的讲解。

### 使用 Docker 镜像

在之前的介绍中，我们知道镜像是 Docker 的三大组件之一。

Docker 运行容器前需要本地存在对应的镜像，如果本地不存在该镜像，Docker 会从镜像仓库下载该镜像。

### 获取镜像

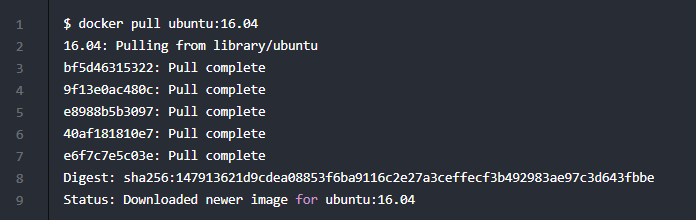
从 Docker 镜像仓库获取镜像的命令是 docker pull。其命令格式为：

docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]

例如： docker pull ip:port/tomcat:jre-9 //docker默认走官网地址下载。

Docker 镜像仓库地址：地址的格式一般是 <域名/IP>[:端口号]。默认地址是 Docker Hub

仓库名：如之前所说，这里的仓库名是两段式名称，即 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。



上面的命令中没有给出 Docker 镜像仓库地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 ubuntu:16.04，因此将会获取官方镜像 library/ubuntu 仓库中标签为 16.04 的镜像

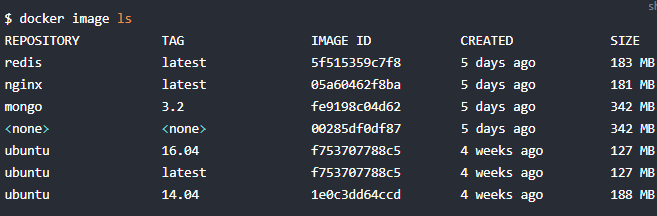
从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的 ID 的前 12 位。并且下载结束后，给出该镜像完整的 sha256 的摘要，以确保下载一致性。

在使用上面命令的时候，你可能会发现，你所看到的层 ID 以及 sha256 的摘要和这里的不一样。这是因为官方镜像是一直在维护的，有任何新的 bug，或者版本更新，都会进行修复再以原来的标签发布，这样可以确保任何使用这个标签的用户可以获得更安全、更稳定的镜像。

如果从 Docker Hub 下载镜像非常缓慢，可以参照 镜像加速器 一节配置加速器。

### 列出镜像

要想列出已经下载下来的镜像，可以使用 docker image ls 命令。



列表包含了 仓库名、标签、镜像 ID、创建时间 以及 所占用的空间。

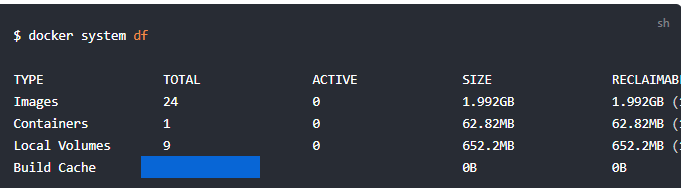
其中仓库名、标签在之前的基础概念章节已经介绍过了。镜像 ID 则是镜像的唯一标识，一个镜像可以对应多个标签。因此，在上面的例子中，我们可以看到 ubuntu:16.04 和 ubuntu:latest 拥有相同的 ID，因为它们对应的是同一个镜像。docker容器

### 镜像体积

如果仔细观察，会注意到，这里标识的所占用空间和在 Docker Hub 上看到的镜像大小不同。比如，ubuntu:16.04 镜像大小，在这里是 127 MB，但是在 Docker Hub 显示的却是 50 MB。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中镜像是保持着压缩状态的，因此 Docker Hub 所显示的大小是网络传输中更关心的流量大小。而 docker image ls 显示的是镜像下载到本地后，展开的大小，准确说，是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

另外一个需要注意的问题是，docker image ls 列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。

你可以通过以下命令来便捷的查看镜像、容器、数据卷所占用的空间。

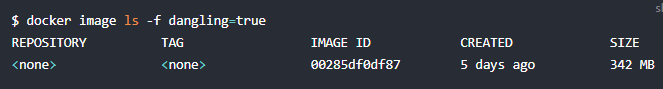


### 虚悬镜像

上面的镜像列表中，还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为 <none>。



这个镜像原本是有镜像名和标签的，原来为 mongo:3.2，随着官方镜像维护，发布了新版本后，重新 docker pull mongo:3.2 时，mongo:3.2 这个镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消，从而成为了 <none>。除了 docker pull 可能导致这种情况，docker build 也同样可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 <none> 的镜像。这类无标签镜像也被称为 虚悬镜像(dangling image) ，可以用下面的命令专门显示这类镜像：



一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的，可以用下面的命令删除。



### 中间层镜像

为了加速镜像构建、重复利用资源，Docker 会利用 中间层镜像。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 docker image ls 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加 -a 参数。



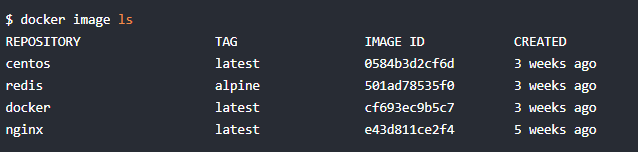
这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们。只要删除那些依赖它们的镜像后，这些依赖的中间层镜像也会被连带删除

### 镜像的删除

如果要删除本地的镜像，可以使用 docker image rm 命令，其格式为：

格式:docker image rm [选项] <镜像1> [<镜像2> ...]

用 ID、镜像名、摘要删除镜像其中，<镜像> 可以是 镜像短 ID、镜像长 ID、镜像名 或者 镜像摘要。



1、根据id

我们可以用镜像的完整 ID，也称为 长 ID，来删除镜像。使用脚本的时候可能会用长 ID，但是人工输入就太累了，所以更多的时候是用 短 ID 来删除镜像。docker image ls 默认列出的就已经是短 ID 了，一般取前3个字符以上，只要足够区分于别的镜像就可以了。

docker image rm 501

2、镜像名字

docker image rm centos

3、镜像摘要



# Docker容器

## docker 容器

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。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。也因为这种隔离的特性，很多人初学 Docker 时常常会混淆容器和虚拟机

前面讲过镜像使用的是分层存储，容器也是如此。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为容器存储层。

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，容器删除或者重新运行之后，数据却不会丢失。

### 运行

有了镜像后，我们就能够以这个镜像为基础启动并运行一个容器。以上面的 ubuntu:16.04 为例，如果我们打算启动里面的 bash 并且进行交互式操作的话，可以执行下面的命令。



docker run 就是运行容器的命令，我们这里简要的说明一下上面用到的参数。

-it：这是两个参数，一个是 -i：交互式操作，一个是 -t 终端。我们这里打算进入 bash 执行一些命令并查看返回结果，因此我们需要交互式终端。

--rm：这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 docker rm。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 --rm 可以避免浪费空间。

ubuntu:16.04：这是指用 ubuntu:16.04 镜像为基础来启动容器。

-p 宿主机端口:容器端口 指定容器的端口映射到宿主机的端口

--name tomcat tomcat 定义容器的名字，以tomcat作为镜像

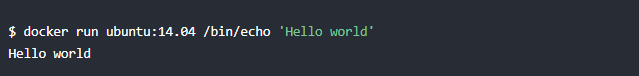
bash：放在镜像名后的是命令，这里我们希望有个交互式 Shell，因此用的是 bash

进入容器后，我们可以在 Shell 下操作，执行任何所需的命令。这里，我们执行了 cat /etc/os-release，这是 Linux 常用的查看当前系统版本的命令，从返回的结果可以看到容器内是 Ubuntu 16.04.4 LTS 系统。

最后我们通过 exit 退出了这个容器。

### 新建并启动

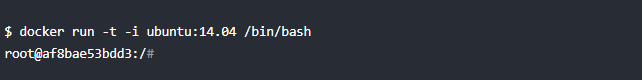
所需要的命令主要为 docker run



例如，下面的命令输出一个 “Hello World”，之后终止容器。

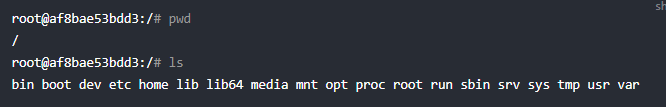
这跟在本地直接执行 /bin/echo 'hello world' 几乎感觉不出任何区别。

下面的命令则启动一个 bash 终端，允许用户进行交互。



其中，-t 选项让Docker分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上， -i 则让容器的标准输入保持打开。

在交互模式下，用户可以通过所创建的终端来输入命令，例如



当利用 docker run 来创建容器时，Docker 在后台运行的标准操作包括：

检查本地是否存在指定的镜像，不存在就从公有仓库下载

利用镜像创建并启动一个容器

分配一个文件系统，并在只读的镜像层外面挂载一层可读写层

从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去

从地址池配置一个 ip 地址给容器

执行用户指定的应用程序

执行完毕后容器被终止

### 启动已终止容器

可以利用 docker container start 命令，直接将一个已经终止的容器启动运行。

容器的核心为所执行的应用程序，所需要的资源都是应用程序运行所必需的。除此之外，并没有其它的资源。可以在伪终端中利用 ps 或 top 来查看进程信息。

### 进入容器

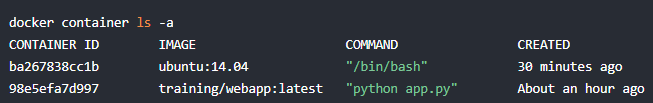
在使用 -d 参数时，容器启动后会进入后台。某些时候需要进入容器进行操作，包括使用 docker attach 命令或 docker exec 命令，推荐大家使用 docker exec 命令，原因会在下面说明。

### 终止容器

可以使用 docker container stop 来终止一个运行中的容器。此外，当 Docker 容器中指定的应用终结时，容器也自动终止。

例如对于上一章节中只启动了一个终端的容器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 docker container ls -a 命令看到。例如



处于终止状态的容器，可以通过 docker container start 命令来重新启动。

此外，docker container restart 命令会将一个运行态的容器终止，然后再重新启动它。

## 守护态运行

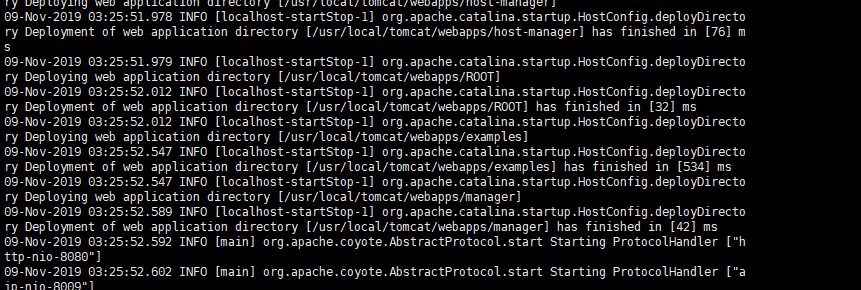
### 概述

更多的时候，需要让 Docker 在后台运行而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加 -d 参数来实现

### 例子演示

如果不使用 -d 参数运行容器。

执行命令：docker run -p 8080:8080 --name tomcat tomcat



容器会把输出的结果 (STDOUT) 打印到宿主机上面

用-d 参数

docker run -p 8080:8080 --name tomcat -d tomcat



此时容器会在后台运行并不会把输出的结果 (STDOUT) 打印到宿主机上面(输出结果可以用 docker logs 查看)。

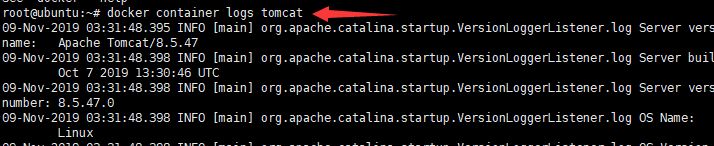
注： 容器是否会长久运行，是和 docker run 指定的命令有关，和 -d 参数无关。

### 查看日志信息

要获取容器的输出信息，可以通过 docker container logs 命令。

命令：docker container logs [container ID or NAMES

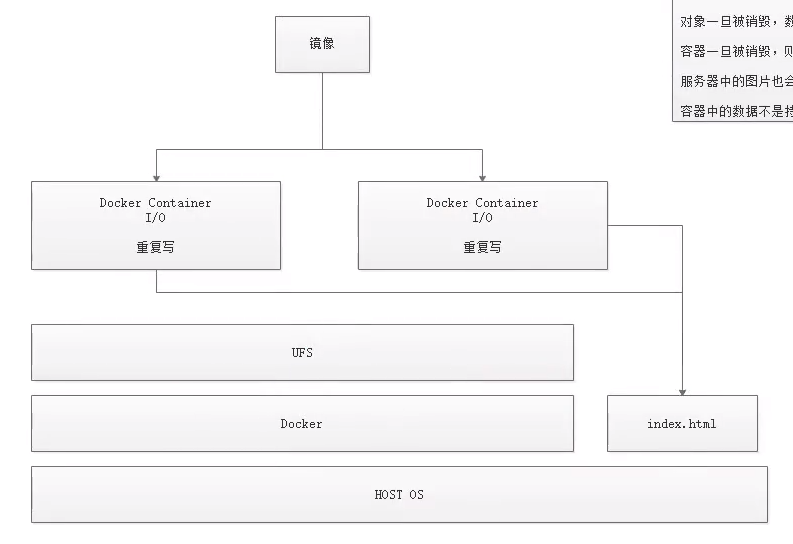
例如：



## docker 数据卷

### 数据卷的引入

对象一旦被销毁,数据就不存在了，容器一旦被销毁,则容器内的数据将一并被删除，服务器中的图片也会一并被删除，容器中的数据不是持久化状态的。



### 什么是数据卷

数据卷 是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

1、数据卷 可以在容器之间共享和重用

2、对 数据卷 的修改会立马生效

3、对 数据卷 的更新，不会影响镜像

4、数据卷 默认会一直存在，即使容器被删除

注意：数据卷 的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的 数据卷。

### 以数据卷你的方法挂载文件

将当前文件夹的ROOT目录挂载到tomcat的ROOT目录

docker run -p 8080:8080 --name tomcat -d -v /usr/local/docker/tomcat/ROOT:/usr/local/tomcat/webapps/ROOT tomcat

# dockerFile

## Dockerfile 定制镜像

从刚才的 docker commit 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。

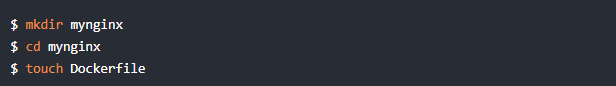
Dockerfile 是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

## dockerFile helloworld

### 创建镜像

还以之前定制 nginx 镜像为例，这次我们使用 Dockerfile 来定制

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：



Dockerfile中的内容

|  |
| --- |
| FROM tomcat  RUN echo "hello ddocker" >/usr/local/tomcat/webapps/ROOT/index.html |

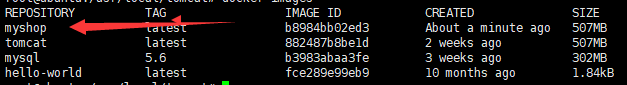
这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。

### 构建镜像

docker build -t myshop . 指定Dockerfile为当前目录下。

### 查看创建的镜像

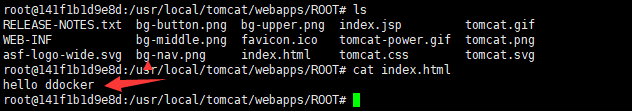
docker images



### 查看myshop中添加的内容

进入容器中 docker run -it myshop bash

在dockerFile中添加的内容



### 删除webroot目录下的所有文件

1、进入root目录删除所有的文件夹

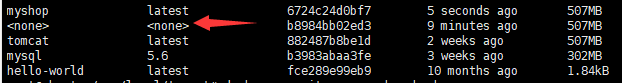
|  |
| --- |
| FROM tomcat  WORKDIR //usr/local/tomcat/webapps/ROOT  RUN rm -rf \*  RUN echo "hello 1111" >/usr/local/tomcat/webapps/ROOT/index.html |

2、当再次进入镜像中时，会在Dockerfile进入的目录中。(层级关系)



3、构建成功，上次的镜像会变为虚悬镜像。

执行 docker image prune 删除



## DockerFile指令

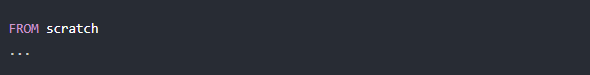
### FROM

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 FROM 就是指定基础镜像，因此一个 Dockerfile 中 FROM 是必备的指令，并且必须是第一条指令

在 Docker Store 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 nginx、redis、mongo、mysql、httpd、php、tomcat 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 node、openjdk、python、ruby、golang 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 ubuntu、debian、centos、fedora、alpine 等，这些操作系统的软件库为我们提供了更广阔的扩展空间

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。



如果你以 scratch 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如 swarm、coreos/etcd。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM scratch 会让镜像体积更加小巧。使用 Go 语言 开发的应用很多会使用这种方式来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因之一。

### RUN 执行命令

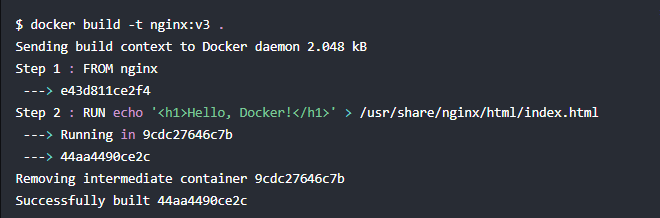
RUN 指令是用来执行命令行命令的。由于命令行的强大能力，RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

### 构建镜像

好了，让我们再回到之前定制的 nginx 镜像的 Dockerfile 来。现在我们明白了这个 Dockerfile 的内容，那么让我们来构建这个镜像吧。

在 Dockerfile 文件所在目录执行：



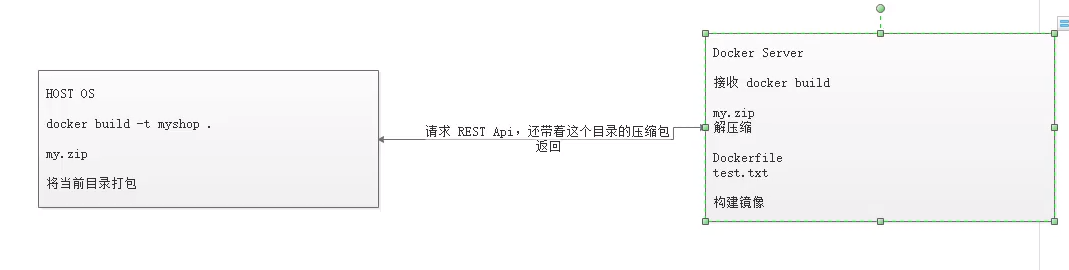
从命令的输出结果中，我们可以清晰的看到镜像的构建过程。在 Step 2 中，如同我们之前所说的那样，RUN 指令启动了一个容器 9cdc27646c7b，执行了所要求的命令，并最后提交了这一层 44aa4490ce2c，随后删除了所用到的这个容器 9cdc27646c7b

这里我们使用了 docker build 命令进行镜像构建。其格式为：

docker build [选项] <上下文路径/URL/->

在这里我们指定了最终镜像的名称 -t nginx:v3，构建成功后，我们可以像之前运行 nginx:v2 那样来运行这个镜像，其结果会和 nginx:v2 一样。

### 镜像构建上下文（Context）



如果注意，会看到 docker build 命令最后有一个 .。. 表示当前目录，而 Dockerfile 就在当前目录，因此不少初学者以为这个路径是在指定 Dockerfile 所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定上下文路径。那么什么是上下文呢？

首先我们要理解 docker build 的工作原理。Docker 在运行时分为 Docker 引擎（也就是服务端守护进程）和客户端工具。Docker 的引擎提供了一组 REST API，被称为 Docker Remote API，而如 docker 命令这样的客户端工具，则是通过这组 API 与 Docker 引擎交互，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种 docker 功能，但实际上，一切都是使用的远程调用形式在服务端（Docker 引擎）完成。也因为这种 C/S 设计，让我们操作远程服务器的 Docker 引擎变得轻而易举。

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。

如果在 Dockerfile 中这么写：

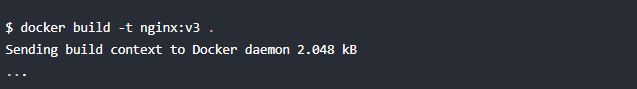
COPY ./package.json /app/

这并不是要复制执行 docker build 命令所在的目录下的 package.json，也不是复制 Dockerfile 所在目录下的 package.json，而是复制 上下文（context） 目录下的 package.json。

因此，COPY 这类指令中的源文件的路径都是相对路径。这也是初学者经常会问的为什么 COPY ../package.json /app 或者 COPY /opt/xxxx /app 无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

现在就可以理解刚才的命令 docker build -t nginx:v3 . 中的这个 .，实际上是在指定上下文的目录，docker build 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像。

如果观察 docker build 输出，我们其实已经看到了这个发送上下文的过程：



理解构建上下文对于镜像构建是很重要的，避免犯一些不应该的错误。比如有些初学者在发现 COPY /opt/xxxx /app 不工作后，于是干脆将 Dockerfile 放到了硬盘根目录去构建，结果发现 docker build 执行后，在发送一个几十 GB 的东西，极为缓慢而且很容易构建失败。那是因为这种做法是在让 docker build 打包整个硬盘，这显然是使用错误。

一般来说，应该会将 Dockerfile 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 .gitignore 一样的语法写一个 .dockerignore，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

那么为什么会有人误以为 . 是指定 Dockerfile 所在目录呢？这是因为在默认情况下，如果不额外指定 Dockerfile 的话，会将上下文目录下的名为 Dockerfile 的文件作为 Dockerfile。

这只是默认行为，实际上 Dockerfile 的文件名并不要求必须为 Dockerfile，而且并不要求必须位于上下文目录中，比如可以用 -f ../Dockerfile.php 参数指定某个文件作为 Dockerfile。

当然，一般大家习惯性的会使用默认的文件名 Dockerfile，以及会将其置于镜像构建上下文目录中。

## Dockerfile指令详解

### COPY

格式：COPY <源路径>... <目标路径>

COPY ["<源路径1>",... "<目标路径>"]

COPY 指令将从构建上下文目录中 <源路径> 的文件/目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

COPY package.json /usr/src/app/

<源路径> 可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则，如

COPY hom\* /mydir/

COPY hom?.txt /mydir/

<目标路径> 可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。

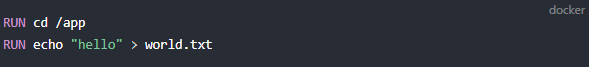
此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候

### WORKDIR

格式: WORKDIR <工作目录路径>

使用 WORKDIR 指令可以来指定工作目录（或者称为当前目录），以后各层的当前目录就被改为指定的目录，如该目录不存在，WORKDIR 会帮你建立目录。

之前提到一些初学者常犯的错误是把 Dockerfile 等同于 Shell 脚本来书写，这种错误的理解还可能会导致出现下面这样的错误：



如果将这个 Dockerfile 进行构建镜像运行后，会发现找不到 /app/world.txt 文件，或者其内容不是 hello。原因其实很简单，在 Shell 中，连续两行是同一个进程执行环境，因此前一个命令修改的内存状态，会直接影响后一个命令；而在 Dockerfile 中，这两行 RUN 命令的执行环境根本不同，是两个完全不同的容器。这就是对 Dockerfile 构建分层存储的概念不了解所导致的错误。

之前说过每一个 RUN 都是启动一个容器、执行命令、然后提交存储层文件变更。第一层 RUN cd /app 的执行仅仅是当前进程的工作目录变更，一个内存上的变化而已，其结果不会造成任何文件变更。而到第二层的时候，启动的是一个全新的容器，跟第一层的容器更完全没关系，自然不可能继承前一层构建过程中的内存变化。

因此如果需要改变以后各层的工作目录的位置，那么应该使用 WORKDIR 指令。

例如：

RUN 容器1 cd /app 只是在容器一种操作

RUN 容器2 echo “hellow” > hellow.txt 此时的工目录还是在镜像的工作沐浴露

镜像的工作目录:

例如 tomcat：docker run -it –-rm tomcat bash

进入的目录是:/usr/local/docker/tomcat

|  |
| --- |
| FROM tomcat  # 切换工作目录  WORKDIR /usr/local/tomcat/webapps/ROOT  # 删除该目录下的所有内容  RUN rm -rf \*  # 复制文件  COPY index.zip .  # 解压文件  RUN unzip index.zip  # 删除该文件  RUN rm -rf index.zip  # 切换回工作目录  WORKDIR /usr/local/tomcat |

### ADD

add:

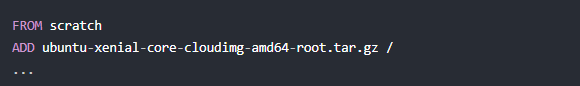
1、自动解压缩tar包。

ADD 指令和 COPY 的格式和性质基本一致。但是在 COPY 基础上增加了一些功能。

比如 <源路径> 可以是一个 URL，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 <目标路径> 去。下载后的文件权限自动设置为 600，如果这并不是想要的权限，那么还需要增加额外的一层 RUN 进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或者 curl 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果 <源路径> 为一个 tar 压缩文件的话，压缩格式为 gzip, bzip2 以及 xz 的情况下，ADD 指令将会自动解压缩这个压缩文件到 <目标路径> 去。

在某些情况下，这个自动解压缩的功能非常有用，比如官方镜像 ubuntu 中：



但在某些情况下，如果我们真的是希望复制个压缩文件进去，而不解压缩，这时就不可以使用 ADD 命令了。

在 Docker 官方的 Dockerfile 最佳实践文档 中要求，尽可能的使用 COPY，因为 COPY 的语义很明确，就是复制文件而已，而 ADD 则包含了更复杂的功能，其行为也不一定很清晰。最适合使用 ADD 的场合，就是所提及的需要自动解压缩的场合。

另外需要注意的是，ADD 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。

因此在 COPY 和 ADD 指令中选择的时候，可以遵循这样的原则，所有的文件复制均使用 COPY 指令，仅在需要自动解压缩的场合使用 ADD。

### EXPOSE

式为 EXPOSE <端口1> [<端口2>...]

EXPOSE 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 docker run -P 时，会自动随机映射 EXPOSE 的端口。

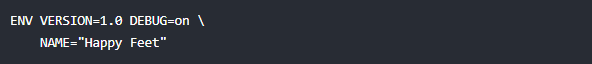
此外，在早期 Docker 版本中还有一个特殊的用处。以前所有容器都运行于默认桥接网络中，因此所有容器互相之间都可以直接访问，这样存在一定的安全性问题。于是有了一个 Docker 引擎参数 --icc=false，当指定该参数后，容器间将默认无法互访，除非互相间使用了 --links 参数的容器才可以互通，并且只有镜像中 EXPOSE 所声明的端口才可以被访问。这个 --icc=false 的用法，在引入了 docker network 后已经基本不用了，通过自定义网络可以很轻松的实现容器间的互联与隔离

要将 EXPOSE 和在运行时使用 -p <宿主端口>:<容器端口> 区分开来。-p，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

### ENV

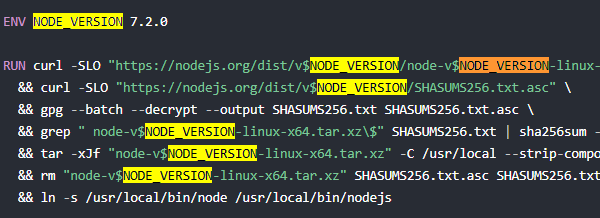
式 ENV <key> <value>

这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如 RUN，还是运行时的应用，都可以直接使用这里定义的环境变量。



这个例子中演示了如何换行，以及对含有空格的值用双引号括起来的办法，这和 Shell 下的行为是一致的。

定义了环境变量，那么在后续的指令中，就可以使用这个环境变量。比如在官方 node 镜像 Dockerfile 中，就有类似这样的代码：



在这里先定义了环境变量 NODE\_VERSION，其后的 RUN 这层里，多次使用 $NODE\_VERSION 来进行操作定制。可以看到，将来升级镜像构建版本的时候，只需要更新 7.2.0 即可，Dockerfile 构建维护变得更轻松了。

下列指令可以支持环境变量展开： ADD、COPY、ENV、EXPOSE、LABEL、USER、WORKDIR、VOLUME、STOPSIGNAL、ONBUILD。

可以从这个指令列表里感觉到，环境变量可以使用的地方很多，很强大。通过环境变量，我们可以让一份 Dockerfile 制作更多的镜像，只需使用不同的环境变量即可。

# Docker 部署

## Docker 部署mysql

1、docker hub 上面查找mysql的镜像

docker search mysql



2、从docker hub上（阿里云加速器）拉取mysql镜像到本地标签为5.6

docker pull mysql:5.6

3、

|  |
| --- |
| docker run --name mysql-utt-8 -p 3303:3306  -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7  --character-set-server=utf8mb4 --collation-server=utf8mb4\_unicode\_ci |

### mysql乱码和时间问题

1、乱码(在创建mysql的时候指定编码)

|  |
| --- |
| docker run --name mysql-utt-8 -p 3303:3306  -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7  --character-set-server=utf8mb4 --collation-server=utf8mb4\_unicode\_ci |

2、时区问题

进入docker的mysql中,在/etc/mysql/conf.d 文件夹下有个配置文件docker.cnf

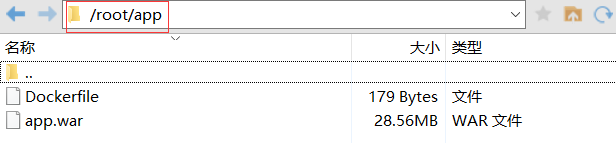
在文件夹中添加

|  |
| --- |
|  |

## docker运行war包

### app文件夹

创建app文件夹：包含Dockerfile文件和项目的war包



### Dockerfile

|  |
| --- |
| FROM tomcat  ENV WORKPATH=/usr/local/tomcat/webapps  WORKDIR ${WORKPATH}  RUN rm -rf \*  ADD ./app.war ${WORKPATH}/app.war  RUN unzip app.war -d ./ROOT  CMD ["catalina.sh","run"] |

### 构建并运行

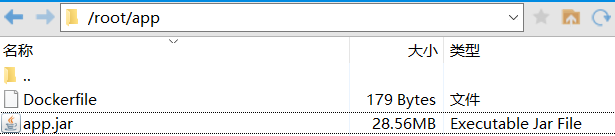
构建：docker build 文件夹 -t ”镜像名称:版本号”

运行：docker run --name 应用名称 -p 8083:8080 -d ”镜像名称:版本号”

## docker运行jar包

### app文件夹

创建app文件夹：包含Dockerfile文件和项目jar包



### Dockerfile

|  |
| --- |
| FROM openjdk  ENV WORKPATH=/usr/src/myapp  COPY . ${WORKPATH}  WORKDIR ${WORKPATH}  CMD ["java","-jar","app.jar"] |

### 构建并运行

构建：docker build 文件夹 -t ”镜像名称:版本号”

运行：docker run --name 应用名称 -p 8083:8080 -d ”镜像名称:版本号”

## solr7安装

### 安装solr

下面这个镜像非官方

|  |
| --- |
| docker run --name ego-mq -p 8161:8161 -p 61616:61616 -d webcenter/activemq |

### solr4和solr7的区别

配置有所变化：

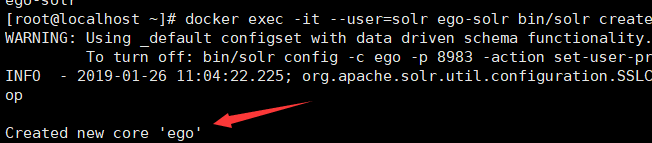
solr4:在solr-home 里面新一个sxt的文件夹，在该文件夹里面包含一个conf，

Conf 必须有schema.xml

solr7：在/opt/solr/server/solr

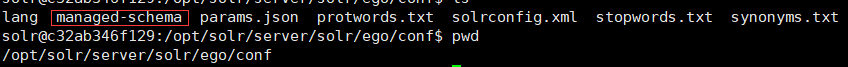
执行命令：创建ego容器

|  |
| --- |
| docker exec -it --user=solr 容器名称 bin/solr create\_core -c core的名称 |
| docker exec -it --user=solr ego-solr bin/solr create\_core -c ego |





配置文件的位置：



### 修改manager-schema 文件

1、将docker容器中的ego-solr中的managed-schema复制出来进行修改。

docker cp ego-solr:/opt/solr/server/solr/ego/conf/managed-schema managed-schema

添加以下字段

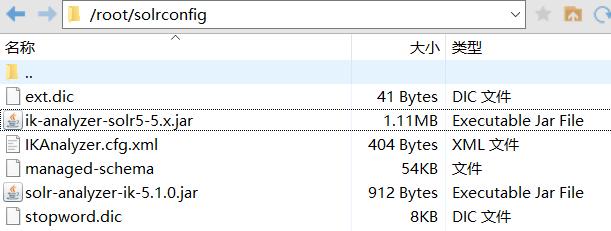
|  |
| --- |
| <field name="id" type="string" indexed="true" stored="true" required="true" multiValued="false" />  <field name="goods\_name" type="text\_cn" indexed="true" stored="true" required="true" multiValued="false" />  <field name="goods\_price" type="pdouble" indexed="true" stored="true" required="true" multiValued="false" />  <field name="goods\_img" type="string" indexed="false" stored="true" required="true" multiValued="false" />  <field name="goods\_comment\_count" type="plong" indexed="true" stored="true" required="true" multiValued="false" /> |

加字段类型

|  |
| --- |
| <fieldType name="text\_cn" class="solr.TextField">  <analyzer > //添加ar包  <tokenizer class="org.apache.lucene.analysis.ik.IKTokenizerFactory" useSmart="true"/>  </analyzer>  </fieldType> |

### 创建solrconfg文件夹

将所需要的文件复制docker中



solr的分词配置复制到该文件夹下

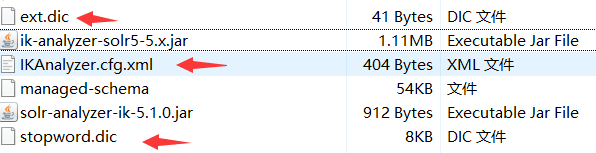
docker cp managed-schema ego-solr:/opt/solr/server/solr/ego/conf/

solr分词jar 需要复制到：/opt/solr/server/solr-webapp/webapp/WEB-INF/lib

docker cp ik-analyzer-solr5-5.x.jar ego-solr:/opt/solr/server/solr-webapp/webapp/WEB-INF/lib

docker cp solr-analyzer-ik-5.1.0.jar ego-solr:/opt/solr/server/solr-webapp/webapp/WEB-INF/lib

Solr分词扩展可以使得分词更准确：可以将



复制到/opt/solr/server/solr-webapp/webapp/WEB-INF/classes

classes：因为这个文件夹并不存在，首先要创建这个文件夹。



将三个文件分支到这个文件夹中

docker cp IKAnalyzer.cfg.xml ego-solr:/opt/solr/server/solr-webapp/webapp/WEB-INF/classes

docker cp stopword.dic ego-solr:/opt/solr/server/solr-webapp/webapp/WEB-INF/classes

docker cp ext.dic ego-solr:/opt/solr/server/solr-webapp/webapp/WEB-INF/classes

### 重启容器

# Docker compose

## 什么是 Docker Compose

Docker Compose 是 Docker 官方编排（Orchestration）项目之一，负责快速的部署分布式应用。

Compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排。从功能上看，跟 OpenStack 中的 Heat 十分类似。

Compose 定位是 「定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）」，其前身是开源项目 Fig。

通过第一部分中的介绍，我们知道使用一个 Dockerfile 模板文件，可以让用户很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。

Compose 恰好满足了这样的需求。它允许用户通过一个单独的 docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

Compose 中有两个重要的概念：

服务 (service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。

项目 (project)：由一组关联的应用容器组成的一个完整业务单元，在 docker-compose.yml 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。因此，只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

## Docker Compose 安装与卸载

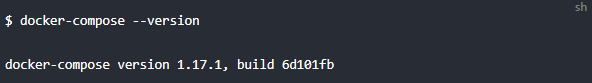
### 安装介绍

Compose 支持 Linux、macOS、Windows 10 三大平台。

Compose 可以通过 Python 的包管理工具 pip 进行安装，也可以直接下载编译好的二进制文件使用，甚至能够直接在 Docker 容器中运行。

前两种方式是传统方式，适合本地环境下安装使用；最后一种方式则不破坏系统环境，更适合云计算场景。

Docker for Mac 、Docker for Windows 自带 docker-compose 二进制文件，安装 Docker 之后可以直接使用。



### 二进制包下载

1、安装docker

curl -fsSL get.docker.com -o get-docker.sh

sudo sh get-docker.sh --mirror AzureChinaCloud

2、修改加速器

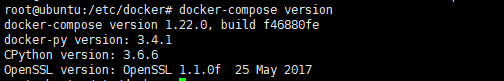
3、安装docker-compose

在 Linux 上的也安装十分简单，从 官方 GitHub Release 处直接下载编译好的二进制文件即可。

例如，在 Linux 64 位系统上直接下载对应的二进制包

|  |
| --- |
| $ sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose  $ sudo chmod +x /usr/local/bin/docker-compose //修改执行权限 |

4、检测是否安装成功



5、卸载

如果是二进制包方式安装的，删除二进制文件即可。

sudo rm /usr/local/bin/docker-compose

## docker-compose的使用

### 术语

首先介绍几个术语。

服务 (service)：一个应用容器，实际上可以运行多个相同镜像的实例。

项目 (project)：由一组关联的应用容器组成的一个完整业务单元。

可见，一个项目可以由多个服务（容器）关联而成，Compose 面向项目进行管理。

### docker-compose 安装tomcat

1、在/usr/local/docker/tomcat创建docker-compose.yml 配置文件

2、tomcat的yml

|  |
| --- |
| version: '3' #docker版本  services:  tomcat:  restart: always  image: tomcat  container\_name: tomcat  ports:  - 8080:8080 |

3、在当前的文件夹下执行

docker-compose up 命令

4、非当前文件夹下

docker-compose –f 文件路径 up

5、守护台运行

docker-compose –f 文件路径 up -d

6、看日志

docker-compose logs tomcat 查看日志

docker-compose logs -f tomcat 监听日志

### docker-compose部署项目

1、配置yml文件

|  |
| --- |
| version: '3'  services:  web:  restart: always  image: tomcat  container\_name: tomcat  ports:  - 8080:8080  # 数据卷  volumes:  - /usr/local/docker/myshop/ROOT:/usr/local/tomcat/webapps/ROOT  mysql:  restart: always  image: mysql:5.7.22  container\_name: mysql  ports:  - 3306:3306  environment:  TZ: Asia/Shanghai  MYSQL\_ROOT\_PASSWORD: 123456  command:  --character-set-server=utf8mb4  --collation-server=utf8mb4\_general\_ci  --explicit\_defaults\_for\_timestamp=true  --lower\_case\_table\_names=1  --max\_allowed\_packet=128M  --sql-mode="STRICT\_TRANS\_TABLES,NO\_AUTO\_CREATE\_USER,NO\_ENGINE\_SUBSTITUTION,NO\_ZERO\_DATE,NO\_ZERO\_IN\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO"  volumes:  - mysql-data:/var/lib/mysql  # 统一数据卷的管理方式  volumes:  mysql-data: |

2、启动

docker-compose up

3、将数据库中的数据导入到该数据库中

4、将项目解压到对应的tomcat的数据卷中。

5、重新启动容器

docker-compose down

docker-compose up

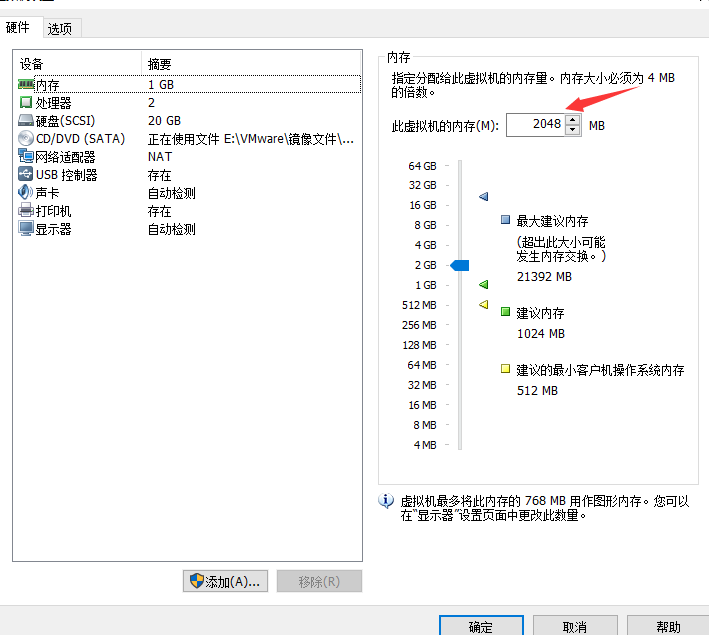
# 安装gitlab

## gitlab

### gitlab介绍

GitLab 是利用 Ruby on Rails 一个开源的版本管理系统，实现一个自托管的 Git 项目仓库，可通过 Web 界面进行访问公开的或者私人项目。它拥有与 Github 类似的功能，能够浏览源代码，管理缺陷和注释。可以管理团队对仓库的访问，它非常易于浏览提交过的版本并提供一个文件历史库。团队成员可以利用内置的简单聊天程序 (Wall) 进行交流。它还提供一个代码片段收集功能可以轻松实现代码复用，便于日后有需要的时候进行查找。

### 配置要求



## 基于 Docker 安装 GitLab

### gitlab下载

docker pull twang2218/gitlab-ce-zh

### 创建docker-compose.yml

我们使用 Docker 来安装和运行 GitLab 中文版，由于新版本问题较多，这里我们使用目前相对稳定的 10.5 版本，docker-compose.yml 配置如下：

|  |
| --- |
| version: '3'  services:  web:  image: 'twang2218/gitlab-ce-zh'  restart: always  hostname: '192.168.10.138'  environment:  TZ: 'Asia/Shanghai'  GITLAB\_OMNIBUS\_CONFIG: |  external\_url 'http://192.168.10.138'  gitlab\_rails['gitlab\_shell\_ssh\_port'] = 2222  unicorn['port'] = 8888  nginx['listen\_port'] = 80  ports:  - '80:80'  - '8443:443'  - '2222:22'  volumes:  - /usr/local/docker/gitlab/config:/etc/gitlab  - /usr/local/docker/gitlab/data:/var/opt/gitlab  - /usr/local/docker/gitlab/logs:/var/log/gitlab |

### 启动

docker-compose up

### 登录访问

1、访问 <http://192.168.10.138>

2、修改管理员登录密码



登录



## 免密登录

# Nexus

## Nexus

### Nexus概述

Nexus 是一个强大的仓库管理器，极大地简化了内部仓库的维护和外部仓库的访问。

2016 年 4 月 6 日 Nexus 3.0 版本发布，相较 2.x 版本有了很大的改变：

对低层代码进行了大规模重构，提升性能，增加可扩展性以及改善用户体验。

升级界面，极大的简化了用户界面的操作和管理。

提供新的安装包，让部署更加简单。

增加对 Docker, NeGet, npm, Bower 的支持。

提供新的管理接口，以及增强对自动任务的管理。

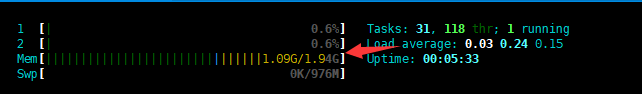
## 基于 Docker 安装 Nexus

### 下载Nexus

docker pull sonatype/nexus3

### 配置

至少两个G内存



### yml配置文件

|  |
| --- |
| version: '3.1'  services:  nexus:  restart: always  image: sonatype/nexus3  container\_name: nexus  ports:  - 8081:8081  volumes:  - /usr/local/docker/nexus/data:/nexus-data |

### 将data创建一个权限

chmod 777 data/

### 启动容器

docker-compose up

### 访问

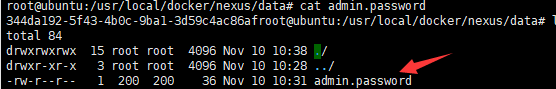
<http://192.168.10.140:8081/>

## nexus使用

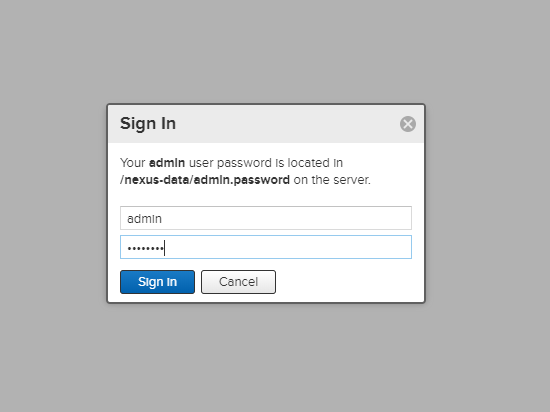
### 登录

默认账号：admin

默认密码: data/admin.password 文件中



修改密码



## 项目中使用私服

### 配置认证信息

在 Maven settings.xml 中添加 Nexus 认证信息(servers 节点下)：

|  |
| --- |
| <server>  <id>nexus-releases</id>  <username>admin</username>  <password>admin123</password>  </server>  <server>  <id>nexus-snapshots</id>  <username>admin</username>  <password>admin123</password>  </server> |

### 配置自动化部署

在 pom.xml 中添加如下代码：

|  |
| --- |
| <distributionManagement>  <repository>  <id>nexus-releases</id>  <name>Nexus Release Repository</name>  <url>http://127.0.0.1:8081/repository/maven-releases/</url>  </repository>  <snapshotRepository>  <id>nexus-snapshots</id>  <name>Nexus Snapshot Repository</name>  <url>http://127.0.0.1:8081/repository/maven-snapshots/</url>  </snapshotRepository>  </distributionManagement> |

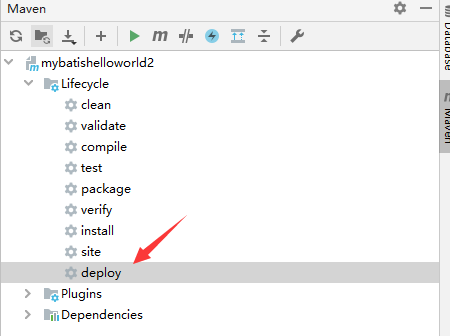
注：

ID 名称必须要与 settings.xml 中 Servers 配置的 ID 名称保持一致。

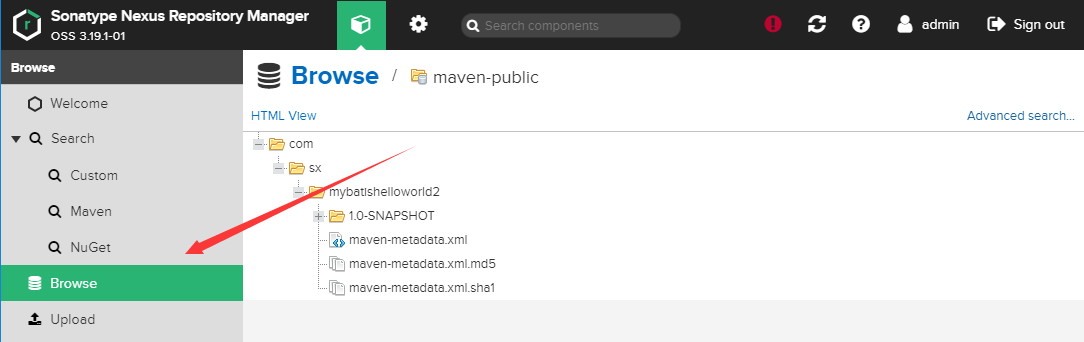
项目版本号中有 SNAPSHOT 标识的，会发布到 Nexus Snapshots Repository, 否则发布到 Nexus Release Repository，并根据 ID 去匹配授权账号。

### 部署到仓库

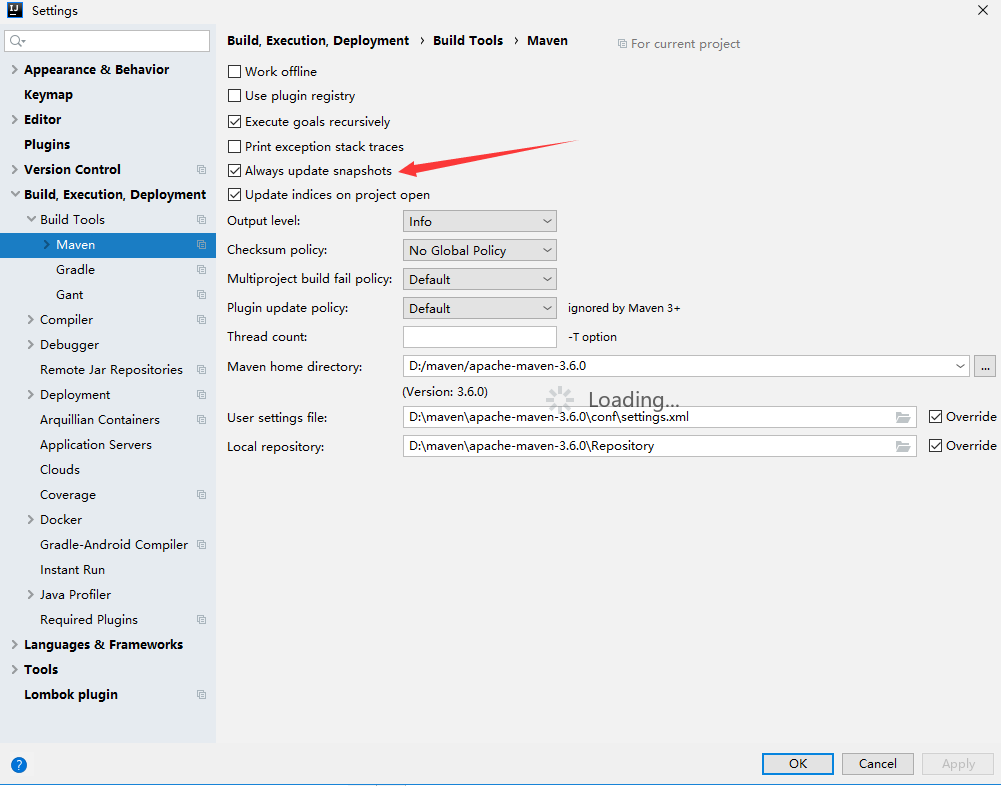
命令:mvn deploy



查看的退从的jar包



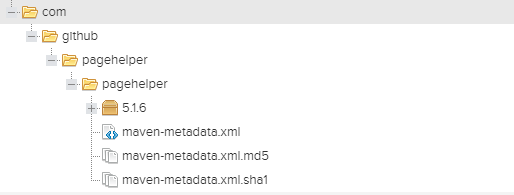
配置maven，每次引入最新的快照



## nexus上传第三方依赖

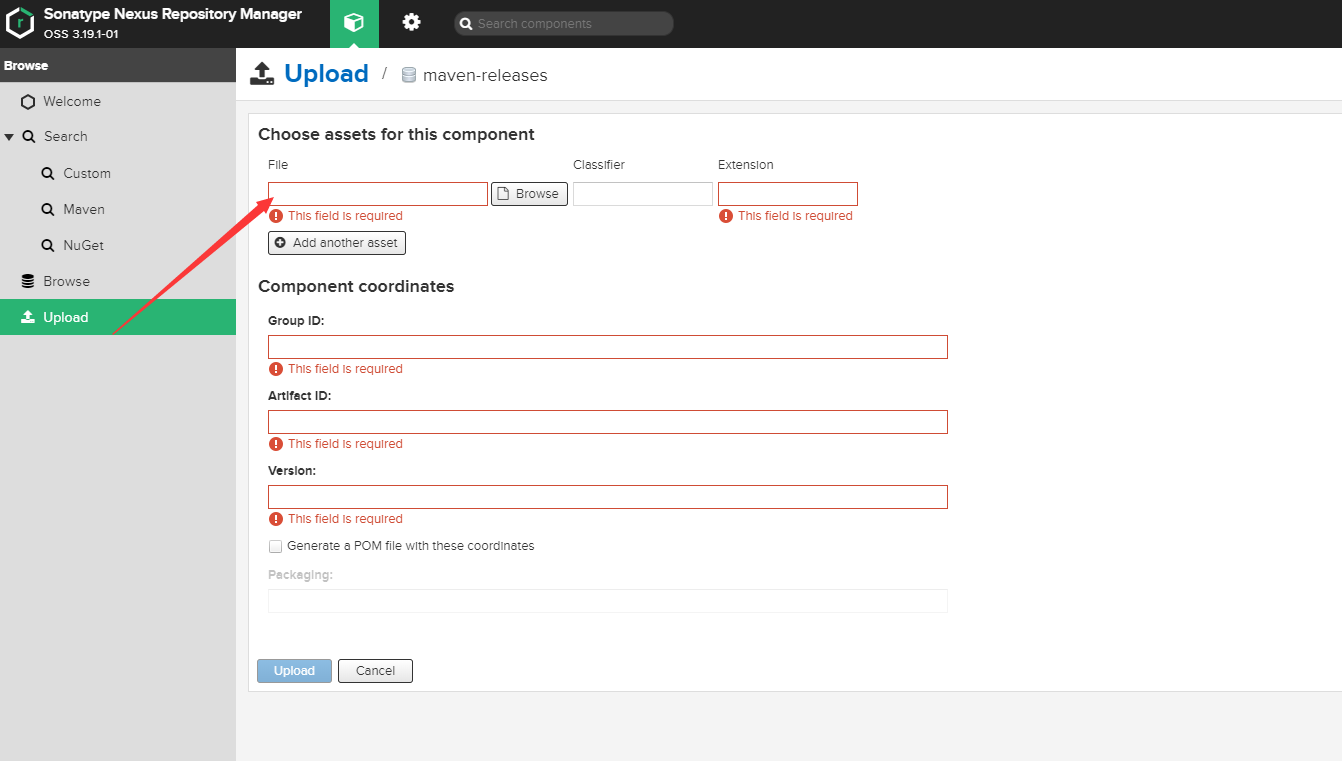
|  |
| --- |
| # 如第三方JAR包：aliyun-sdk-oss-2.2.3.jar  mvn deploy:deploy-file  -DgroupId=com.aliyun.oss  -DartifactId=aliyun-sdk-oss  -Dversion=2.2.3  -Dpackaging=jar  -Dfile=D:\aliyun-sdk-oss-2.2.3.jar  -Durl=http://127.0.0.1:8081/repository/maven-3rd/  -DrepositoryId=nexus-releases |

### 手动上传



|  |
| --- |
| mvn deploy:deploy-file -DgroupId=com.github.pagehelper -DartifactId=pagehelper -Dversion=5.1.6 -Dpackaging=jar -Dfile=D://pagehelper-5.1.6.jar -Durl=http://192.168.10.140:8081/repository/maven-releases/ -DrepositoryId=nexus-releases |

### 使用网页手动上传



## 配置代理仓库

|  |
| --- |
| <**repositories**>  <**repository**>  *<!--jar包依赖仓库-->* <**id**>nexus</**id**>  <**name**>Nexus Repository</**name**>  <**url**>http://192.168.10.140:8081/repository/maven-public/</**url**>  *<!--是否依赖快照和发行-->* <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  </**repository**> </**repositories**> <**pluginRepositories**>  <**pluginRepository**>  <**id**>nexus</**id**>  *<!--插件仓库 -->* <**name**>Nexus Plugin Repository</**name**>  <**url**>http://192.168.10.140:8081/repository/maven-public/</**url**>  *<!--是否依赖快照和发行-->* <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  </**pluginRepository**> </**pluginRepositories**> |

# Docker Registry

## Docker Registry概述

官方的 Docker Hub 是一个用于管理公共镜像的地方，我们可以在上面找到我们想要的镜像，也可以把我们自己的镜像推送上去。但是，有时候我们的服务器无法访问互联网，或者你不希望将自己的镜像放到公网当中，那么你就需要 Docker Registry，它可以用来存储和管理自己的镜像。

## 安装

在之前的 Docker 私有仓库 章节中已经提到过如何配置和使用容器运行私有仓库，这里我们使用 docker-compose 来安装，配置如下：

|  |
| --- |
| version: '3.1'  services:  registry:  image: registry  restart: always  container\_name: registry  ports:  - 5000:5000  volumes:  - /usr/local/docker/registry/data:/var/lib/registry |

## 配置register

### 配置

我们的教学案例使用的是 Ubuntu Server 16.04 LTS 版本，属于 systemd 系统，需要在 /etc/docker/daemon.json 中增加如下内容（如果文件不存在请新建该文件）

|  |
| --- |
| {  "registry-mirrors": [  "https://registry.docker-cn.com"  ],  "insecure-registries": [  "ip:5000"  ]  } |

之后重新启动服务。

$ sudo systemctl daemon-reload

$ sudo systemctl restart docker

### 测试

检查客户端配置是否生效

使用 docker info 命令手动检查，如果从配置中看到如下内容，说明配置成功（192.168.75.133 为教学案例 IP）

Insecure Registries:

192.168.75.133:5000

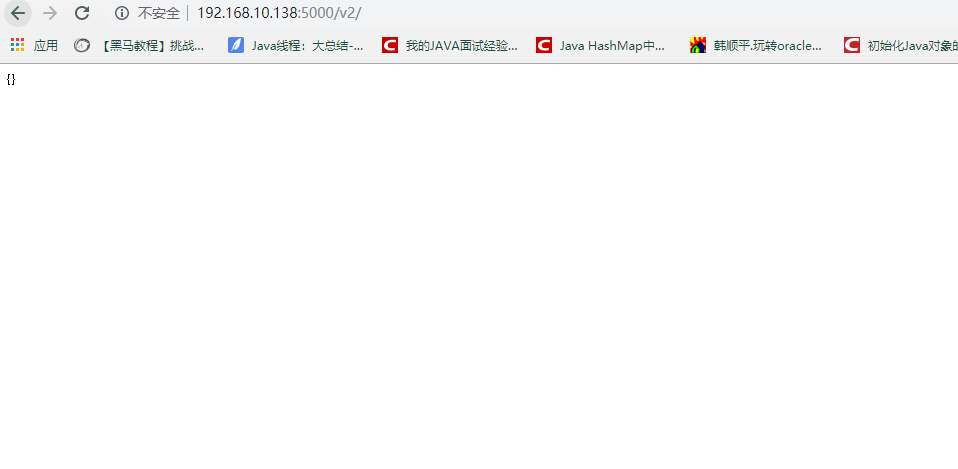
127.0.0.0/8

## 测试

启动成功后需要测试服务端是否能够正常提供服务，有两种方式

### 浏览器端访问

http://192.168.10.138:5000/v2/



## 配置客户端

重新启动一台机，上面用的是regiser 服务器，这里用的是客户端，用来上传镜像到仓库中。

### 概述

我们的教学案例使用的是 Ubuntu Server 16.04 LTS 版本，属于 systemd 系统，需要在 /etc/docker/daemon.json 中增加如下内容（如果文件不存在请新建该文件）

|  |
| --- |
| {  "registry-mirrors": [  "https://registry.docker-cn.com"  ],  "insecure-registries": [  "ip:5000"  ]  } |

之后重新启动服务

$ sudo systemctl daemon-reload

$ sudo systemctl restart docker

### 检查客户端配置是否生效

使用 docker info 命令手动检查，如果从配置中看到如下内容，说明配置成功（192.168.75.133 为教学案例 IP）

|  |
| --- |
| Insecure Registries:  192.168.75.133:5000  127.0.0.0/8 |

## register-frontend

### 部署 Docker Registry WebUI

私服安装成功后就可以使用 docker 命令行工具对 registry 做各种操作了。然而不太方便的地方是不能直观的查看 registry 中的资源情况。如果可以使用 UI 工具管理镜像就更好了。这里介绍两个 Docker Registry WebUI 工具

docker-registry-frontend

docker-registry-web

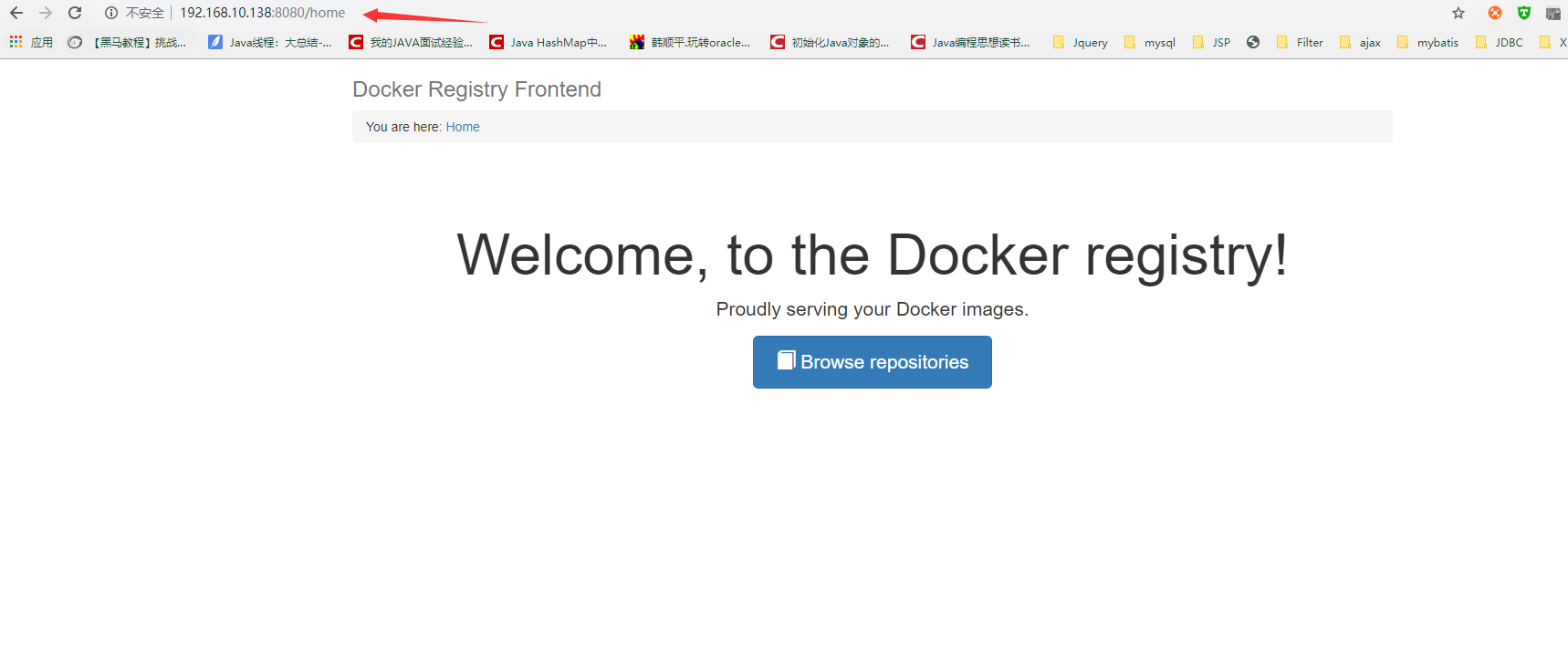
两个工具的功能和界面都差不多，我们以 docker-registry-fontend 为例讲解

### docker-registry-frontend安装

我们使用 docker-compose 来安装和运行，docker-compose.yml 配置如下：

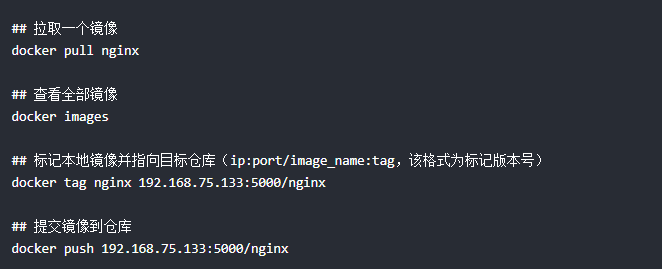
|  |
| --- |
| version: '3.1'  services:  frontend:  image: konradkleine/docker-registry-frontend:v2  ports:  - 8080:80  volumes:  - ./certs/frontend.crt:/etc/apache2/server.crt:ro  - ./certs/frontend.key:/etc/apache2/server.key:ro  environment:  - ENV\_DOCKER\_REGISTRY\_HOST=192.168.10.138  - ENV\_DOCKER\_REGISTRY\_PORT=5000 |

### 测试

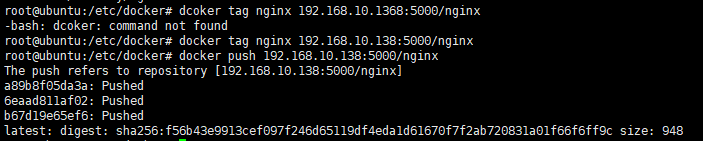


## 测试镜像上传

我们以 Nginx 为例测试镜像上传功能



测试



### 查看上传的镜像



# 项目的部署

## linux安装git

1、linux免密登录git