
容器入门

Course 101

Docker 简介

容器化应用是非常好的软件创意
容器提升了软件的交付效率

Docker 的诞生

Docker 带来的希望

Docker 式工作流的好处

Docker 不是什么

Docker 的演进与变化

Docker 的诞生

2013年3月15日 在加利福尼亚举办的 Python 开发者大会上, Docker 首次作为一种新型的 Linux 容器封装技术向外界披露。

Docker 源码很快托管到 Github 中, 任何人都可以下载或为此项目做贡献。

由于 Docker 为软件的构建、交付和运行提供了革命性的新方式, 一经发布, 迅即大规模流行起来..

Docker 并没有发明容器, 却是真正催熟容器技术的解决方案..

Docker 带来的希望

传统应用开发模式，团队成员之间很难沟通，处理问题的方式也难统一，即便小团队也是如此。Docker 是一种操作系统级别的虚拟化技术，由于关注于应用层面，所以它可以解决传统 workflow 方面众多棘手的问题。

Docker 能帮助企业把不同的团队的业务隔离开，降低开发与运维人员的沟通成本。除了能解决沟通成本问题，Docker 还鼓励使用独特的微服务方式处理软件的架构，使用微服务方式能构建更强健的软件。

微服务架构的核心应用服务的容器化，将解决依赖之前版本的构建问题。Docker 也非常契合应用微服务架构和 DevOps 构建方法论。

Docker 式工作流的好处

Docker 让软件的架构决策变得更简单，具体有如下好处：

- ❑ 使用开发者已掌握的技能打包软件
- ❑ 使用标准的镜像格式打包软件应用和所需的操作系统环境
- ❑ 在运行任何系统的任何环境中测试和部署同一个打包好的镜像
- ❑ 把软件应用与硬件剥离，不滥用资源

Docker 发布之前，Linux 容器技术已经存在很多年了，而且 Docker 使用的很多其他技术也不是全新的。

但是 Docker 采用独特的架构和工作流把这些技术融合在一起之后，终于让已经存在超过十年的 Linux 容器技术走进了普通技术人员的生活之中。

Docker 不是什么

Docker 能解决众多的问题，很多问题也是其他类的传统工具一直致力于解决的，可是广度即意味着深度浅。Docker 由于关注应用层级的封装问题，所以并不能完全取代如下的传统工具，通过与这些工具的结合使用，能得到更好的效果。

- ❑ 企业级虚拟化平台(VMware/KVM)
- ❑ 云平台(Openstack)
- ❑ 配置管理工具(Ansible)
- ❑ 部署框架(Fabric)
- ❑ 容器集群管理工具(Kubernetes)
- ❑ 开发环境(Vagrant)

Docker 的演化与变化

Docker 最初基于 LXC 技术开发, 逐步用 GO 语言重写。

Linux基金会于2015年6月成立OCI(Open Container Initiative)组织,旨在围绕容器格式和运行时制定一个开放的工业化标准。

由于 OCI 标准的确立, 从 Docker 1.11 开始, Docker容器运行已经不是简单的通过 Docker daemon来启动, 而是集成了containerd、runC等多个组件。Docker服务启动之后, 我们也可以看见系统上启动了dockerd、docker-containerd等进程。

在2017年DockerCon17上, Docker发布了两个新的开源项目LinuxKit和Moby。而原来在Github上托管的docker也随着PR #32691的合入正式变为Moby。Docker 正式由一个项目转变成 Docker 公司的商业产品, 项目名变更为Moby。

Docker 安装

软件源安装(YUM/APT)
二进制安装

RedHat 系操作系统安装

Ubuntu 系操作系统安装

Suse 操作系统安装

MacOS 操作系统安装

Windows 操作系统安装

RedHat 系操作系统安装

Redhat 安装文档:

<https://docs.docker.com/engine/installation/linux/rhel/>

由于Redhat 需要企业授权 <DOCKER-EE-URL>

可以使用开源版本的CentOS

<https://docs.docker.com/engine/installation/linux/centos/>

配置软件源, 使用Yum工具完成 Docker CE 安装...

Ubuntu 系操作系统安装

Ubuntu 安装 Docker CE 分两个步骤完成安装, 文档地址:

<https://docs.docker.com/engine/installation/linux/ubuntu/>

第一步: 需要配置 Docker CE 软件仓库, 并配置验证..

第二步: 使用 APT 工具完成 Docker CE 安装..

Suse 操作系统安装

Suse 是与Redhat 一样历史悠久的 Linux 发行版, 安装文档:

<https://docs.docker.com/engine/installation/linux/suse/>

可以使用软件包管理工具 zypper 完成 Docker 引擎安装。

MacOS 操作系统安装

Mac 是一款非常优秀的 Unix 操作系统, Docker 公司特别为它打造了一键部署安装包, 用户只要下载到本地, 一路Next 即可完成 Docker 的安装。文档地址:

<https://docs.docker.com/docker-for-mac/install/#install-and-run-docker-for-mac>

Mac 是理想的 Docker 运行平台, 由于是基于Unix开发, 所以有大量的 Linux 命令可直接在Mac上运行。建议大家使用 Mac 做为 Docker 的学习环境。

Windows 操作系统安装

由于微软的重金投入，Docker 公司也单独为 Windows 提供了部署安装包，与Mac版本一样，只要下载到本机，一步步NEXT 即可完成 Docker 的安装。文档地址：

<https://docs.docker.com/docker-for-windows/install/#download-docker-for-windows>

由于Windows 控制台对传统的Linux命令不太友好，所以使用Windows版本的Docker 容器化体验不是很好。

不过Windows的优点是可以同时提供Windows和Linux 容器同时运行。

镜像 VS 容器

镜像是容器的交付模板
容器是镜像的运行实例

查找镜像 & 运行容器

查看镜像分层

查看容器配置

查看容器日志

删除容器 vs 删除镜像

查找镜像 & 运行容器

运行镜像：

```
docker run -d busybox sh -c "while true; do echo hello world; sleep 1; done"
```

```
➡ ~ docker search busybox
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
busybox	Busybox base image.	991	[OK]	
progrum/busybox		65		[OK]
radial/busyboxplus	Full-chain, ▶ ~ docker run -d busybox sh -c "while true; do echo hello w	12		[OK]
container4armhf/armhf-busybox	Automated b ▶ ~ docker run -d busybox sh -c "while true; do echo hello w			[OK]
odise/busybox-python	b0255427da331f5e24c099133385d117ed1b44beb81e38590f00b640db68			
multiarch/busybox	multiarch p ▶ ~ docker logs -f b025			
ofayau/busybox-jvm	Prepare bus; hello world			
azukiapp/busybox	This image; hello world			

```
▶ ~ docker run -d busybox sh -c "while true; do echo hello world; sleep 1; done"
```

b0255427da331f5e24c099133385d117ed1b44beb81e38590f00b640db686046

```
➡ ~ docker logs -f b025
```

```
hello world
```

```
hello world
```

```
hello world
```

hello world

```
hello world
```

hello world

```
hello world
```

hello world

```
hello world
```

hello world

hello world

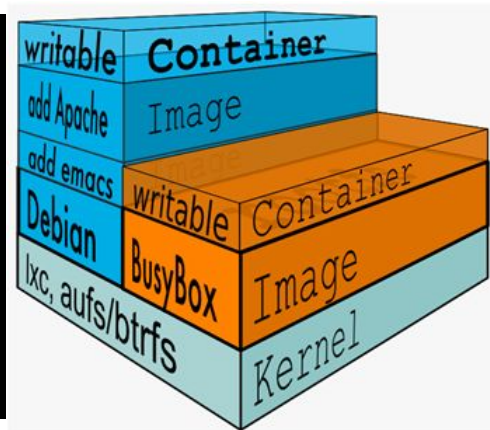
hello world

查看镜像分层

镜像代表了容器的文件系统里的内容，是容器的基础，镜像一般是通过 Dockerfile 生成的。

Docker 的镜像是分层的，所有的镜像(除了基础镜像)都是在之前镜像的基础上加上自己这层的内容生成的。如下图 Layers 所示：

```
"RootFS": {  
  "Type": "layers",  
  "Layers": [  
    "sha256:d17d48b2382adda1fd94284c51d725f0226bf20b07f4d29ce09596788bed7e8e",  
    "sha256:7b4b54c742414142cfde9f89506e9f09f8c47136daf18d33fbc693cae0dd87dd",  
    "sha256:100396c462210b6d7682e75c295477b151e8c2bccb23f39d2601d6b49d05772a",  
    "sha256:41ef8cc0bccbea4c3411d5dd16a4843e34dc7d61cd27bb963c2dec21b9f689cf",  
    "sha256:725b4ef7ffce99b9d5a9667b8e668549d1b4dae3b9ae3061ce73b7f8114c75b7",  
    "sha256:8c92a215240eaeb47642c23a85ef4fd91e97dafdf5f009b3423c4bd6a370e170",  
    "sha256:2f2b8a63dcc66bc26dd19d96a9dff0ddc12029d7a3d451be35b6f749b8c420ea"  
  ]  
}
```



查看容器配置

```
➤ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
b0255427da33       busybox            "sh -c 'while true..." 17 minutes ago     Up 17 minutes      elated_morse

➤ ~ docker inspect b0
[
  {
    "Id": "b0255427da331f5e24c099133385d117ed1b44beb81e38590f00b640db686046",
    "Created": "2017-04-28T08:18:55.845428107Z",
    "Path": "sh",
    "Args": [
      "-c",
      "while true; do echo hello world; sleep 1; done"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 5052,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2017-04-28T08:18:56.607691368Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image": "sha256:00f017a8c2a6e1fe2ffd05c281f27d069d2a99323a8cd514dd35f228ba26d2ff",
    "ResolvConfPath": "/var/lib/docker/containers/b0255427da331f5e24c099133385d117ed1b44beb81e38590f00b640db686046/resolv.conf",
```

查看容器日志

通过 **docker ps** 查看运行的容器信息, 获取容器ID..

通过 **docker logs** [容器ID] 可以看运行容器的内部应用日志。

通过 **docker exec -ti [容器ID] sh** 可以交互式的与容器进行交互, 这是一种常用的交互方式。

```
➔ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
b0255427da33      busybox            "sh -c 'while true..." 23 minutes ago     Up 23 minutes      elated_morse

➔ ~ docker exec -ti b0 sh
/ # cat /etc/resolv.conf
# Generated by dhcpcd from eth0.dhcp
# /etc/resolv.conf.head can replace this line
nameserver 192.168.65.1
# /etc/resolv.conf.tail can replace this line
/ #
```

删除容器 & 删除镜像

```
→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
b0255427da33       busybox            "sh -c 'while true..." 24 minutes ago     Up 24 minutes                      elated_morse

→ ~ docker rm -f b0
b0

→ ~ docker rmi busybox
Untagged: busybox:latest
Untagged: busybox@sha256:32f093055929dbc23dec4d03e09dfe971f5973a9ca5cf059cbfb644c206aa83f
Deleted: sha256:00f017a8c2a6e1fe2ffd05c281f27d069d2a99323a8cd514dd35f228ba26d2ff
Deleted: sha256:c0de73ac99683640bc8f8de5cda9e0e2fc97fe53d78c9fd60ea69b31303efbc9

→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
```

docker rm 是删除容器的命令，加 -f 参数表示强制删除，后面加容器的ID。

docker rmi 是删除镜像的命令，后面加镜像的名称或镜像的ID。

如果基于镜像的容器存在，删除镜像会报错，需要先将所有引用镜像的容器删除，才能继续删除镜像。

镜像的工作流

镜像是Docker的交付载体
镜像版本 vs 应用版本

容器化应用的工作流-Workflow

Dockerfile 指令介绍

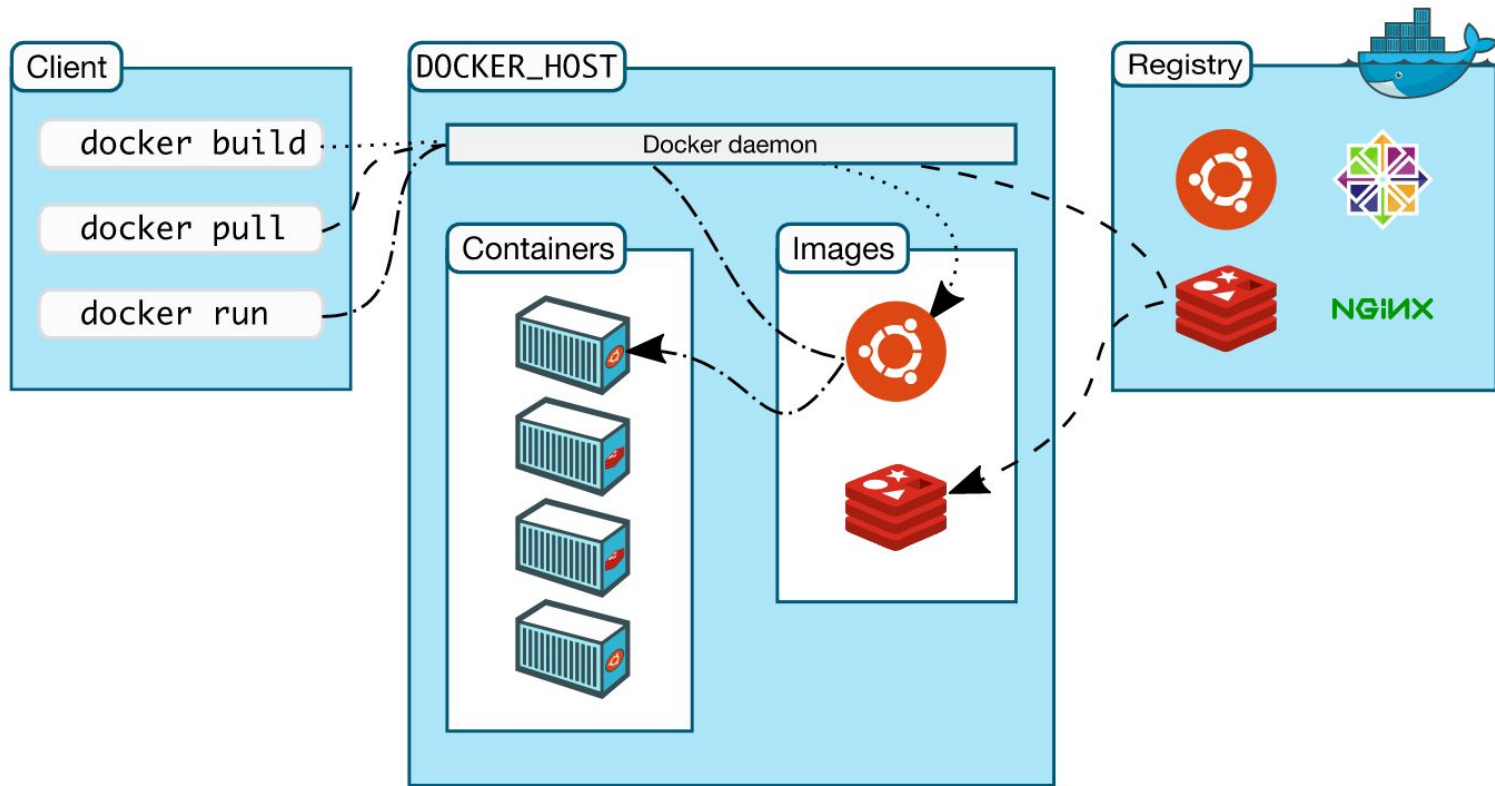
Dockerfile 最佳操作实践

创建您自己的基础镜像

镜像的交付与管理-Registry

配置Harbor镜像仓库

容器化应用的工作流 - WorkFlow



Dockerfile 指令介绍

Dockerfile 是 Docker 用来构建镜像的文本文件，包含自定义的指令和格式，可以通过 `docker build` 命令构建镜像。

Dockerfile 的指令根据作用可以分为两种，构建指令和设置指令：

构建指令：用于构建image,其指定的操作不会在运行image的容器上执行。

设置指令：用于设置image的属性，其指定的操作将在运行image的容器中执行。

Dockerfile 的指令忽略大小写，建议使用大写，使用 `#` 作为注释，每一行只支持一条指令，每条指令可以携带多个参数。

Dockerfile 指令介绍 - 概览

Dockerfile 自定义的指令只有13个，很多指令都是不言自明的，指令格式如下

:#Comment

INSTRUCTION arguments

FROM
MAINTAINER
ENV
USER
WORKDIR
ADD
COPY

RUN
CMD
ENTRYPOINT
VOLUME
EXPOSE
ONBUILD

Dockerfile 指令介绍 - FROM

构建指令 格式: FROM <image> 或 FROM <image>:<tag>

FROM 指令的功能是为后面的指令提供基础镜像, 因此一个有效的Dockerfile必须以FROM指令作为第一条非注释指令。

若FROM指令中参数<tag>是空, 则tag默认是latest。

<image>首选检测本地是否存在, 如果不存在则会从公共仓库下载(当然也可以使用私有仓库的格式)。

Dockerfile 指令介绍 - MAINTAINER

构建指令 格式: MAINTAINER <author>

使用 MAINTAINER 指令来为生成的镜像署名作者

样例:

MAINTAINER xiaolong@caicloud.io

建议每个 Dockerfile 应该写上作者信息, 方便使用 Dockerfile 的用户联系到你。

Dockerfile 指令介绍 - ENV

构建指令 格式: ENV <key> <value> 或 ENV <key>=<value> ...

此指令可以为镜像创建出来的容器声明环境变量。

设置了环境变量, 后续的RUN命令都可以使用, 当运行生成的镜像时这些环境变量依然有效, 如果需要在运行时更改这些环境变量可以在运行docker run时添加-env <key>=<value>参数来修改。

在变量前面添加斜杠\可以转义, 如: \ \$foo 将会被转义为 \$foo。

Dockerfile 指令介绍 - USER

设置指令 格式: USER <user>

为运行镜像时或者任何接下来的RUN指令指定运行用户名或UID, 设置运行容器的用户, 默认是 root 用户。

样例:

```
ENTRYPOINT ["memcached"]
```

```
USER daemon
```

或

```
ENTRYPOINT ["memcached", "-u", "daemon"]
```

Dockerfile 指令介绍 - WORKDIR

构建指令 格式: WORKDIR <path>

WORKDIR 指令用于设置 Dockerfile 中的RUN、CMD和ENTRYPOINT指令执行命令的工作目录(默认为/目录)。

该指令在Dockerfile文件中可以出现多次, 如果使用相对路径则为相对于WORKDIR上一次的值, 例如: WORKDIR /a, WORKDIR b, RUN pwd最终输出的当前目录是/a/b。

注: RUN cd /a/b RUN pwd是得不到/a/b

Dockerfile 指令介绍 - ADD

构建指令 格式: `ADD <src> <dest>`

将文件 `<src>` 拷贝到container的文件系统对应的路径 `<dest>`下。

`<src>` 可以是文件、文件夹、URL, 对于文件和文件夹 `<src>` 必须是在Dockerfile的相对路径下(build context path), 即只能是相对路径且不能包含`../path/`。

`<dest>`只能是容器中的绝对路径。如果路径不存在则会自动级联创建, 根据你的需要是`<dest>`里是否需要反斜杠`/`, 习惯使用`/`结尾从而避免被当成文件。

ADD 只有在 build 镜像的时候运行一次, 后面运行container的时候不会再重新加载, 也就是你不能在运行时通过这种方式向容器中传送文件, `-v`选项映射本地到容器的目录。

Dockerfile 指令介绍 - COPY

构建指令 格式: COPY <src> <dist>

COPY 的语法与功能和ADD 相同, 不支持<src>远程URL、自动解压这两个特性, 但是Best Practices for Writing Dockerfiles建议尽量使用COPY, 并使用RUN与COPY的组合来代替ADD。

这是因为虽然 COPY 只支持本地文件拷贝到容器, 但它的处理比ADD更加透明, 建议只在复制tar文件时使用ADD, 如ADD trusty-core-amd64.tar.gz /。

Dockerfile 指令介绍 - RUN

构建指令 格式：

`RUN <command> (shell 格式)`

`RUN ["executable", "param1", "param2"] (exec格式 - 推荐格式)`

RUN指令会在当前镜像的顶层执行任何命令，并commit成新的(中间)镜像，提交的镜像会在后面继续用到。

RUN 指令的两种格式表示命令在容器中的两种运行方式。当使用 shell 格式时，命令通过 `sh -c` 运行；当使用exec 格式时，命令是直接运行的，容器不调用 shell 程序，即容器中没有 shell 程序。

Dockerfile 指令介绍 - CMD

设置指令 格式:

CMD <command> (shell格式)

CMD ["executable", "param1", "param2"] (exec格式-推荐)

CMD ["param1", "param2"] (为 ENTRYPOINT 指令提供参数)

一个Dockerfile里只能有一个CMD, 如果有多个, 只有最后一个生效。

RUN 是在 build 成镜像时就运行的, 先于 CMD 和 ENTRYPOINT。

CMD 会在每次启动容器的时候运行, 而RUN 只在创建镜像时执行一次, 固化在 image 中。

Dockerfile 指令介绍 - ENTRYPOINT

设置指令 格式:

```
ENTRYPOINT <command>
```

```
ENTRYPOINT ["executable", "param1", "param2"]
```

ENTRYPOINT 与 CMD指令类似, 都可以让容器在每次启动时执行相同的命令, 但它们之间又有不同。

一个Dockerfile 中可以有多条 ENTRYPOINT 指令, 但只有最后一条ENTRYPOINT指令有效。

Dockerfile 指令介绍 - VOLUME

设置指令 格式: `VOLUME ["path"]`

设置指令, 使容器中的一个目录具有持久化存储数据的功能, 该目录可以被容器本身使用, 也可以共享给其他容器使用。

我们知道容器使用的是AUFS文件系统, 这种文件系统不能持久化数据, 当容器关闭后, 所有的更改都会丢失, 当容器中的应用有持久化数据的需求时可以在Dockerfile中使用该指令。

VOLUME指令用来设置一个挂载点, 可以用来让其他容器挂载以实现数据共享或对容器数据的备份、恢复或迁移。

Dockerfile 指令介绍 - EXPOSE

设置指令 格式: EXPOSE <port> [...]

EXPOSE指令用来告诉Docker这个容器在运行时会监听哪些端口, Docker在连接不同的容器(使用-link参数)时使用这些信息。当需要访问容器的时候, 可以不是用容器的IP地址而是使用宿主机的IP地址和映射后的端口。

要完成整个操作需要两个步骤:

首先在Dockerfile中使用EXPOSE设置需要映射的容器端口, 然后在运行容器的时候指定-p选项加上EXPOSE设置的端口, 这样EXPOSE设置的端口号会被随机映射成宿主主机中的一个端口号。

Dockerfile 指令介绍 - ONBUILD

格式: ONBUILD [INSTRUCTION]

ONBUILD 指令的功能是添加一个将来执行的触发器指令到镜像中。

当该镜像作为FROM指令的参数时, 这些触发器指令就会在FROM指令执行时加入到构建过程中。当需要制作一个基础镜像来构建其他镜像时, ONBUILD是很有用的。

使用ONBUILD指令的Dockerfile构建的镜像应用有特殊标签: 如 python:1.0-onbuild。在ONBUILD指令中添加ADD 或COPY指令时要额外注意。

Dockerfile 最佳操作实践

- ❑ 使用标签
- ❑ 谨慎选择基础镜像
- ❑ 充会利用缓存
- ❑ 正确使用 ADD 与 COPY 指令
- ❑ RUN指令
- ❑ 不要在 Dockerfile 中做端口映射
- ❑ 使用 Dockerfile 共享 Docker 镜像

创建您自己的基础镜像

虽然可以从 Hub.docker.com 下载很多的操作系统基础镜像，但我们一定很好奇这些基础镜像是怎么生成出来的，能否自己也做一个基础镜像呢？

事实上官方的源码仓库里已经有一些常用的系统镜像生成模板。

<https://github.com/moby/moby/blob/master/contrib/mkimage-busybox.sh>

例如生成一个 busybox 的基础镜像

```
tar --numeric-owner -cf- . | docker import - busybox
```

Setup1. 编译软件二进制包

```
//克隆测试代码
git clone https://github.com/docker-library/hello-world
//编译源码
cd hello-world
docker run --rm -it -v $PWD:/build ubuntu:16.04
#apt-get update && apt-get install build-essential
#cd /build
#gcc -o hello -static -nostartfiles hello.c
```

Setup2. 编写Dockerfile

```
FROM scratch
ADD hello /
CMD ["/hello"]
```

Setup3. 构建自己的基础镜像

```
docker build -t hello-world:build .
```

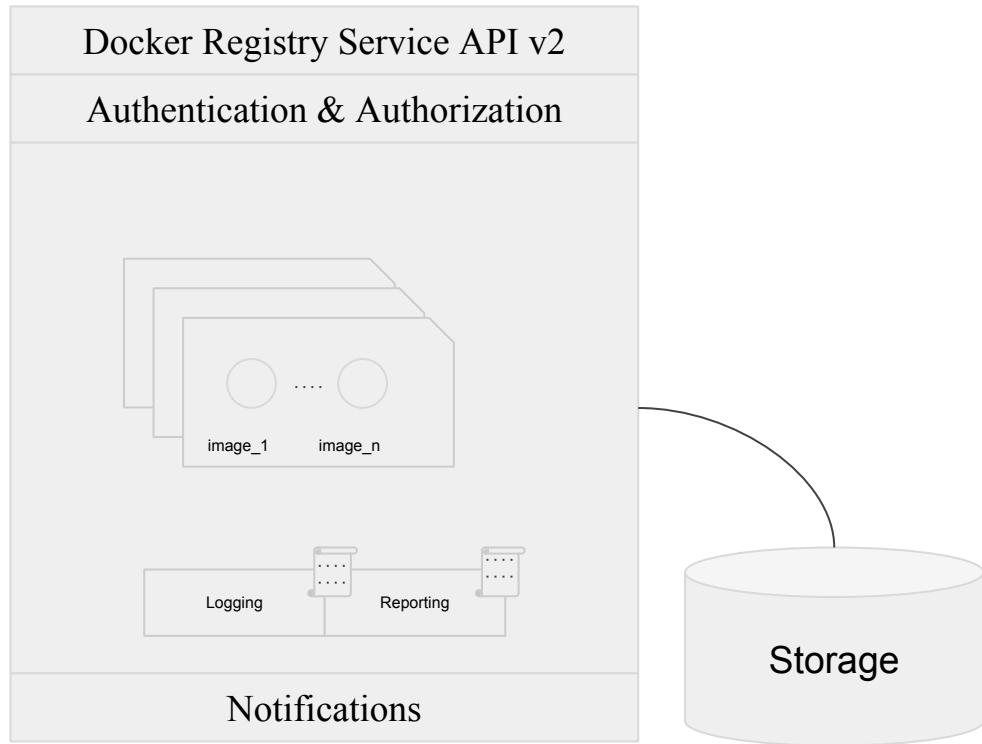
镜像的交付与管理 - Registry

Docker Registry是构建仓库的核心，用于实现开源Docker镜像的分发， Docker Registry源码发布在：

<https://github.com/docker/distribution>

右图是Registry的内部结构图，它实现了镜像的创建、存储、分发和更新等功能。

Docker先后发布了两个版本的Registry，v1是用python编写的，v2用go进行了重写，在安全性和性能上做了很多优化，并重新设计了镜像的存储格式。



镜像的交付与管理 - Registry

部署私有仓库可以节省带宽资源, 运行命令如下:

```
docker pull registry
```

```
docker run -d --name registry-v2 -v `pwd`/registry:/var/lib/registry -p 5000:5000 registry
```


上面命令会在本机运行一个名为registry-v2的服务, 命令中的-v选项会把本地宿主机某个目录mount到容器内的存储目录, 方便我们查看和管理本地镜像数据, -p选项会把容器内的5000端口映射到宿主机的5000端口, 方便演示。

```
docker tag redis localhost:5000/caicloud/redis
```

```
docker push localhost:5000/caicloud/redis
```


这样宿主机上的/root/registry目录里的数据就是刚刚PUSH好的Redis镜像

配置Harbor镜像仓库

 HARBOR™

中文

搜索项目和镜像资源


 登录


登录


注册

[忘记密码](#)

这很容易上手...


匿名访问公开镜像仓库


基于项目的镜像管理


用户角色访问控制

为什么要使用Harbor?

Harbor是可靠的企业级Registry服务器。企业用户可使用Harbor搭建私有容器Registry服务，提高生产效率和安全性，既可应用于生产环境，也可以在开发环境中使用。

[▶ 显示全部...](#)

热门镜像仓库

镜像仓库名	下载次数
google_containers/kubernetes-dashboard-amd64	89
public/nginx	83
google_containers/pause-amd64	37
google_containers/kube-proxy-amd64	34
google_containers/kube-controller-manager-amd64	23

Docker 存储基础

了解镜像、容器和存储插件
配置存储插件

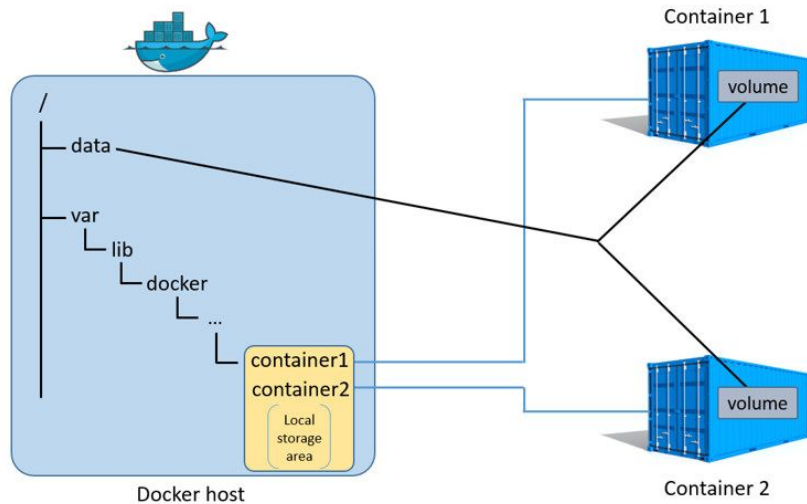
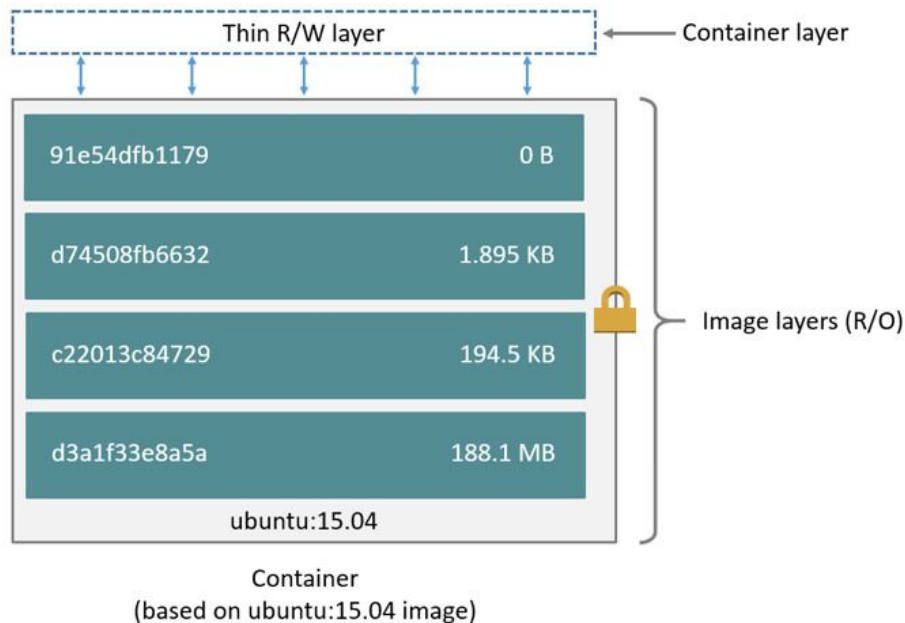
镜像分层的存储支持

查看Docker存储信息

Docker 默认存储插件介绍

Docker存储插件配置

镜像分层的存储支持



查看 Docker 存储信息

```
→ ~ docker info
Containers: 13
  Running: 3
  Paused: 0
  Stopped: 10
Images: 206
Server Version: 17.03.1-ce
Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 4ab9917febca54791c5f071a9d1f404867857fcc
runc version: 54296cf40ad8143b62dbcaa1d90e520a2136ddfe
init version: N/A (expected: 949e6facb77383876aeff8a6944dde66b3089574)
```

Docker 默认存储插件介绍

AUFS	stable	production-ready	good memory use	smooth Docker experience	high write activity	PaaS-type work
Devicemapper (loop)	stable	in mainline kernel	smooth Docker experience	production	performance	lab testing
Devicemapper (direct-lvm)	stable	production-ready	in mainline kernel	smooth Docker experience	PaaS-type work	
Btrfs	in mainline kernel	high write activity	container churn	build pools		
Overlay	stable	good memory use	in mainline kernel	container churn	lab testing	
ZFS native (ZoL)	PaaS-type work					
ZFS FUSE	stable	lab testing	production			

Key

Has attribute attribute

If good for use case use case

If bad for use case use case

Docker 存储插件配置

默认 CentOS7 下 Docker 使用的 Device Mapper 设备使用的是 loop-lvm 默认模式，它使用OS层面离散的文件来构建精简池(thin pool)。

该模式主要是设计出来让Docker能够简单的被"开箱即用(out-of-the-box)"而无需额外的配置。

但如果是在生产环境的部署Docker，官方明文不推荐使用该模式。生产部署的首选配置是direct-lvm，此模式使用块设备创建精简池来存放镜像和容器的数据。

`--storage-driver=devicemapper`

Docker 网络基础

了解容器网络

容器网络介绍

默认桥接网络

容器网络介绍

Docker 内置三个网络, 分别是: bridge、host、none。

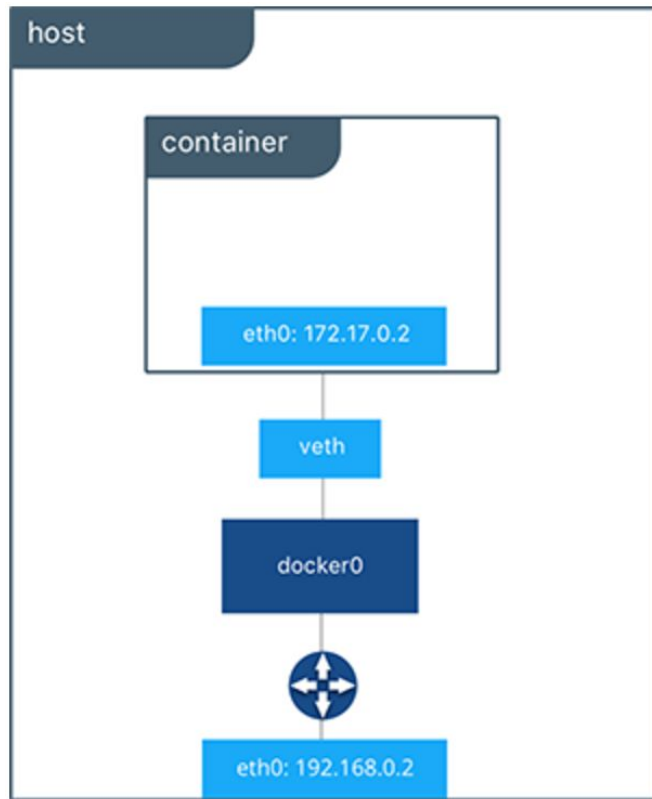
运行容器时, 可以使用 `--network` 标志指定容器应该连接到哪些网络。

还有一种网络模式是 `container`, 此模式是 `host` 模式的扩展。

默认桥接网络

bridge: 指的就是 docker0 网桥, 同一宿主机上所有的容器都会通过 docker0 连接在一起。可以通过主机上使用 ifconfig 命令, 将看到此桥作为主机网络堆栈的一部分。

作为最常见的模式, bridge 模式已经满足 Docker 容器最基本的使用需求, 然而其与外界通信使用 NAT 协议, 增加了通信的复杂性, 在复杂场景下使用会有诸多限制。



Docker 存储练习

数据持久化练习

宿主机目录挂载到容器

添加容器卷&定位容器卷 - Volume

创建和安装数据卷容器

备份、还原或迁移数据卷

移除数据卷 & 使用共享卷的提示

宿主机目录挂载到容器

挂载数据卷的语法：

```
docker run -v /Users/<path>:/<container path> ...
```

样例：

```
docker run -d -P --name web -v `pwd`/webapp:/webapp training/webapp  
python app.py
```

添加容器卷 & 定位数据卷 - Volume

```
→ ~ docker run -d -P --name web -v `pwd`/webapp training/webapp python app.py  
687f17d0c65f760925b126cc0057147ecdcd53dfa7b36563d2edd44d9f4978
```

```
→ ~ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
687f17d0c65f	training/webapp	"python app.py"	3 seconds ago	Up 2 seconds	0.0.0.0:32768->5000/tcp	web

```
"Mounts": [  
  {  
    "Type": "volume",  
    "Name": "0dd5e684a69c4838519ed1d16ca4146b2f4ba53a8fe72a59c72050584e554449",  
    "Source": "/var/lib/docker/volumes/0dd5e684a69c4838519ed1d16ca4146b2f4ba53a8fe72a59c72050584e554449/_data",  
    "Destination": "/Users/dxwsker/webapp",  
    "Driver": "local",  
    "Mode": "",  
    "RW": true,  
    "Propagation": ""  
  }  
],
```

创建和安装数据卷容器

```
➔ ~ docker volume ls
DRIVER          VOLUME NAME
local           0dd5e684a69c4838519ed1d16ca4146b2f4ba53a8fe72a59c72050584e554449
➔ ~ docker volume create my-volume
my-volume
➔ ~ docker run -d -P \
-v my-volume:/webapp \
--name web training/webapp python app.py
fd0714d4299c406f9b2967ddab483753b1d1ac92a9a0f574995662c423795923
➔ ~ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
fd0714d4299c   training/webapp "python app.py"         3 seconds ago Up 2 seconds   0.0.0.0:32771->5000/tcp            web
➔ ~ docker volume ls
DRIVER          VOLUME NAME
local           0dd5e684a69c4838519ed1d16ca4146b2f4ba53a8fe72a59c72050584e554449
local           my-volume
```

备份、还原或迁移数据卷

`docker run -v /dbdata --name dbstore -ti centos:7.2.1511 bash`

`docker run --rm --volumes-from dbstore -v $(pwd)/backup:/backup centos:7.2.1511 bash -c "cd /dbdata && tar -cvf /backup/backup.tar ."`

```
➔ docker101 docker run -v /dbdata --name dbstore -ti centos:7.2.1511 bash
[root@4e6103b7db67 /]# pwd
/
[root@4e6103b7db67 /]# cd /dbdata/
[root@4e6103b7db67 dbdata]# ls
[root@4e6103b7db67 dbdata]# vi readme.txt
[root@4e6103b7db67 dbdata]# cat readme.txt
hello world
[root@4e6103b7db67 dbdata]#

X ..lab/docker101 (zsh)
➔ docker101 docker run --rm --volumes-from dbstore -v $(pwd)/backup:/backup centos:7.2.1511 bash -c "cd /dbdata && tar -cvf /backup/backup.tar ."
./
./readme.txt
```

移除数据卷 & 使用共享卷的提示

```
➔ docker101 docker volume ls
DRIVER          VOLUME NAME
local           0dd5e684a69c4838519ed1d16ca4146b2f4ba53a8fe72a59c72050584e554449
local           8ed794acde0fee63758df22235fb64450113c1cea9c37424c43e166e64bd2c75
local           a8728e61c59c912cdf2a9563cea10a0b8fced1f3b8c5597601dbd773bc8e1b6c
local           my-volume
```

```
➔ docker101 docker volume --help
```

unknown flag: --help

See 'docker volume --help'.

Usage: docker volume COMMAND

Manage volumes

Options:

--help Print usage

Commands:

create Create a volume

inspect Display detailed information on one or more volumes

ls List volumes

prune Remove all unused volumes

rm Remove one or more volumes

Run 'docker volume COMMAND --help' for more information on a command.

Docker 网络练习

网络命令练习

查看容器默认网络

创建您自己的桥接网络

将容器加入网络

查看容器默认网络

查看docker的默认网络

docker network ls

```
➔ ~ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
d826a94afa3d	bridge	bridge	local
d0107d090164	host	host	local
750bdd1ebd48	none	null	local

创建您自己的桥接网络

```
docker run -itd --name=networktest ubuntu
```

```
docker network inspect bridge
```

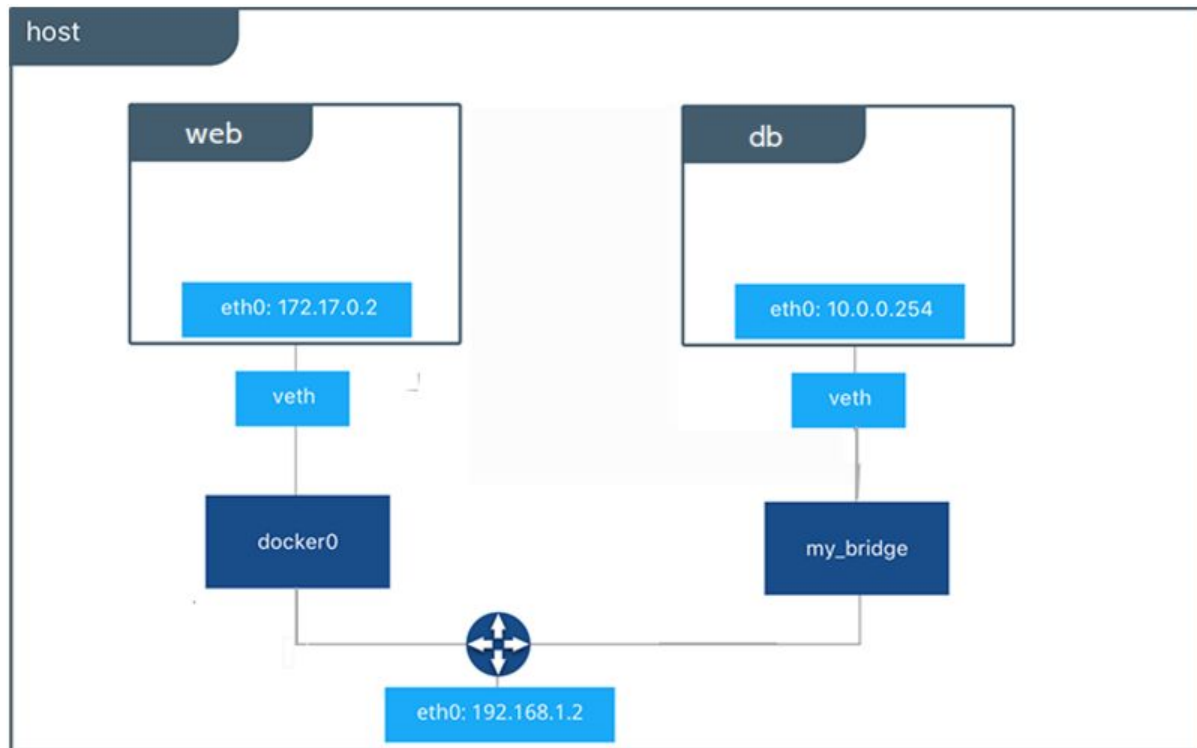
```
docker network disconnect bridge networktest
```

```
docker network create -d bridge my_bridge
```

```
docker network ls
```

```
docker network inspect my_bridge
```

将容器加入网络



课程总结:

在本套 Docker 101 课程中主要学习了Docker的基础概念, Dockerfile的编写, 基础的网络和存储, 大家应该掌握如下知识:

- ❑ 了解Docker的历史沿革和演化
- ❑ 学会常用的Docker操作指令
- ❑ 深入理解基于镜像的Docker工作流
- ❑ 学会Dockerfile的编写及性能优化及镜像仓库的搭建
- ❑ 了解Docker存储支持及默认存储插件的区别
- ❑ 了解Docker基础的网络环境

谢谢聆听

