Docker 开源书

周立 http://www.itmuch.com

干货小程序

干货公众号

Table of Contents

简介 1.1
01-Docker简介 1.2
02-Docker安装 1.3
03-配置镜像加速器 1.4
04-镜像常用命令 1.5
05-容器常用命令 1.6
06-实战：修改Nginx首页 1.7
07-Dockerfile指令详解 1.8
08-实战：使用Dockerfile修改Nginx首页 1.9
09-实战：巩固-阅读常用软件的Dockerfile 1.10
10-使用Docker Hub管理镜像 1.11
11-使用Docker Registry管理Docker镜像 1.12
12-使用Nexus管理Docker镜像 1.13
13-Docker可视化管理工具 1.14
14-Docker数据持久化 1.15
15-端口映射 1.16
16-遗留网络 1.17
17-Docker网络 1.18
18-network命令 1.19
19-默认bridge网络中配置DNS 1.20
20-用户定义网络中的内嵌DNS服务器 1.21
21-Docker Compose简介 1.22
22-安装Docker Compose 1.23
23-Docker Compose快速入门 1.24
24-docker-compose.yml常用命令 1.25
25-docker-compose常用命令 1.26
26-Docker Compose网络设置 1.27
27-实战：使用Docker Compose编排WordPress博客 1.28
28-控制服务启动顺序 1.29
29-在生产环境中使用Docker Compose 1.30
30-实战：使用Docker Compose运行ELK 1.31
31-使用Docker Compose伸缩应用 1.32

Docker开源书

Docker开源书，旨在帮助大家熟悉Docker、使用Docker。

> - GitHub地址：https://github.com/itmuch/docker-book
> - Gitee地址：https://gitee.com/itmuch/docker-book
>
> 欢迎star、fork，一起讨论！
>
> QQ群：731548893
>
> 微信群：加jumping_me，注明加群。

内容主要包括：

- 入门
- Dockerfile详解
- 镜像管理
- 工具
- 持久化
- 网络
- Docker Compose

七大主题，涵盖Docker常用命令、Dockerfile常用命令、网络、存储、Docker Compose等常见知识点，知识体系应该是比较完备的。如果学习完，你应该具备如下能力：

- 常用的命令信手拈来
- Dockerfile编写无压力
- 能用Docker Compose快速构建容器器环境
- 理解Docker网络、存储等知识点是怎么回事。

Docker简介

1.1 Docker简介

Docker是一个开源的**容器引擎**，它可以帮助我们更快地交付应用。Docker可将应用程序和基础设施层隔离，并且能将基础设施当作程序一样进行管理。使用Docker，可更快地打包、测试以及部署应用程序，并可**减少从编写到部署运行代码的周期**。

TIPS

(1) Docker官方网站：https://www.docker.com/

(2) Docker GitHub：https://github.com/docker/docker

1.2 版本与迭代计划

近日，Docker发布了Docker 17.06。进入Docker 17时代后，Docker分成了两个版本：Docker EE和Docker CE，即：企业版(EE)和社区版(CE)。

1.2.1 版本区别

Docker EE（企业版）

Docker EE由公司支持，可在经过认证的操作系统和云提供商中使用，并可运行来自Docker Store的、经过认证的容器和插件。

Docker EE提供三个服务层次：

服务层级	功能
Basic	包含用于认证基础设施的Docker平台 Docker公司的支持 经过 认证的、来自Docker Store的容器与插件
Standard	添加高级镜像与容器管理 LDAP/AD用户集成 基于角色的访问控制(Docker Datacenter)
Advanced	添加Docker安全扫描 连续漏洞监控

大家可在该页查看各个服务层次的价目：https://www.docker.com/pricing 。

Docker CE

Docker CE是免费的Docker产品的新名称，Docker CE包含了完整的Docker平台，非常适合开发人员和运维团队构建容器APP。事实上，Docker CE 17.03，可理解为Docker 1.13.1的Bug修复版本。因此，从Docker 1.13升级到Docker CE 17.03风险相对是较小的。

大家可前往Docker的RELEASE log查看详情https://github.com/docker/docker/releases 。

Docker公司认为，Docker CE和EE版本的推出为Docker的生命周期、可维护性以及可升级性带来了巨大的改进。

1.2.2 版本迭代计划

Docker从17.03开始，转向基于时间的 `YY.MM` 形式的版本控制方案，类似于Canonical为Ubuntu所使用的版本控制方案。

Docker CE有两种版本：

edge版本每月发布一次，主要面向那些喜欢尝试新功能的用户。

stable版本每季度发布一次，适用于希望更加容易维护的用户（稳定版）。

edge版本只能在当前月份获得安全和错误修复。而stable版本在初始发布后四个月内接收关键错误修复和安全问题的修补程序。这样，Docker CE用户就有一个月的窗口期来切换版本到更新的版本。举个例子，Docker CE 17.03会维护到17年07月；而Docker CE 17.03的下个稳定版本是CE 17.06，这样，6-7月这个时间窗口，用户就可以用来切换版本了。

Docker EE和stable版本的版本号保持一致，每个Docker EE版本都享受**为期一年**的支持与维护期，在此期间接受安全与关键修正。

1.2.3 参考文档

ANNOUNCING DOCKER ENTERPRISE EDITION：https://blog.docker.com/2017/03/docker-enterprise-edition/

1.3 Docker的发展历程

- 发展历程

Docker版本	Docker基于{}实现
Docker 0.7之前	基于LXC
Docker0.9后	改用libcontainer
Docker 1.11后	**改用runC和containerd**

- 表格名词对应官网
    - LXC：https://linuxcontainers.org/lxc/introduction/
    - libcontainer：https://github.com/docker/libcontainer
    - runC：https://github.com/opencontainers/runc
    - containerd：https://github.com/containerd/containerd
- 各名词之间的关系
    - OCI：定义了容器运行的标准，该标准目前由libcontainer和appc的项目负责人（maintainer）进行维护和制定，其规范文档作为一个项目在GitHub上维护。

 - runC（标准化容器执行引擎）：根据根据OCI规范编写的，生成和运行容器的CLI工具，是按照开放容器格式标准（OCF, Open Container Format）制定的一种具体实现。由libcontainer中迁移而来的，实现了容器启停、资源隔离等功能。
 - containerd：用于控制runC的守护进程，构建在OCI规范和runC之上。目前內建在Docker Engine中，参考文档：https://blog.docker.com/2015/12/containerd-daemon-to-control-runc/ ，译文：http://dockone.io/article/914
- 浅谈发展历程
 - 时序：Docker大受欢迎 - 与CoreOS相爱相杀 - rkt诞生 - 各大厂商不爽 - OCI制定（2015-06） - 成立CNCF（2015-07-21） - Kubernetes 1.0发布；
 - CNCF：云原生计算基金会，由谷歌联合发起，现隶属于Linux基金会。
- 拓展阅读
 - Docker背后的标准化容器执行引擎——runC：http://www.infoq.com/cn/articles/docker-standard-container-execution-engine-runc
 - Docker、Containerd、RunC...：你应该知道的所有:http://www.infoq.com/cn/news/2017/02/Docker-Containerd-RunC
 - Google宣布成立CNCF基金会，Kubernetes 1.0正式发布：http://dockone.io/article/518

1.4 Docker快速入门

执行如下命令，即可启动一个Nginx容器

```
docker run -d -p 91:80 nginx
```

1.5 Docker架构

我们来看一下来自Docker官方文档的架构图，如图12-1所示。

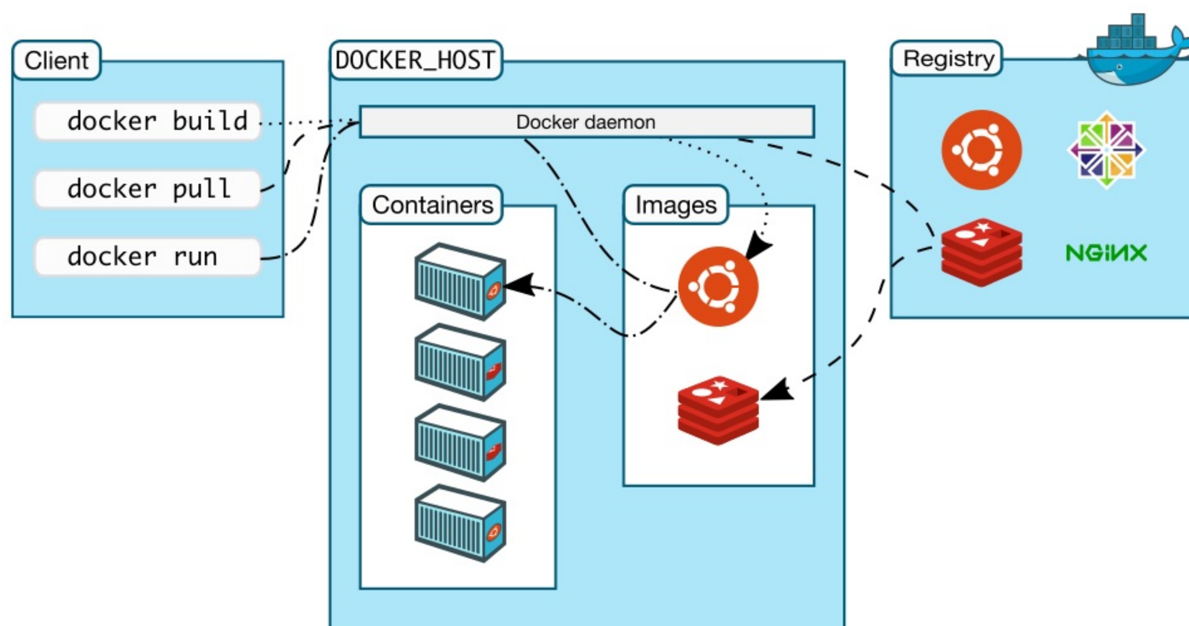


图12-1 Docker架构图

我们来讲解图中包含的组件。

(1) Docker daemon（Docker守护进程）

Docker daemon是一个运行在宿主机（DOCKER_HOST）的后台进程。我们可通过Docker客户端与之通信。

(2) Client（Docker客户端）

Docker客户端是Docker的用户界面，它可以接受用户命令和配置标识，并与Docker daemon通信。图中，docker build等都是Docker的相关命令。

(3) Images（Docker镜像）

Docker镜像是一个只读模板，它包含创建Docker容器的说明。它和系统安装光盘有点像——我们使用系统安装光盘安装系统，同理，我们使用Docker镜像运行Docker镜像中的程序。

(4) Container（容器）

容器是镜像的可运行实例。镜像和容器的关系有点类似于面向对象中，类和对象的关系。我们可通过Docker API或者CLI命令来启停、移动、删除容器。

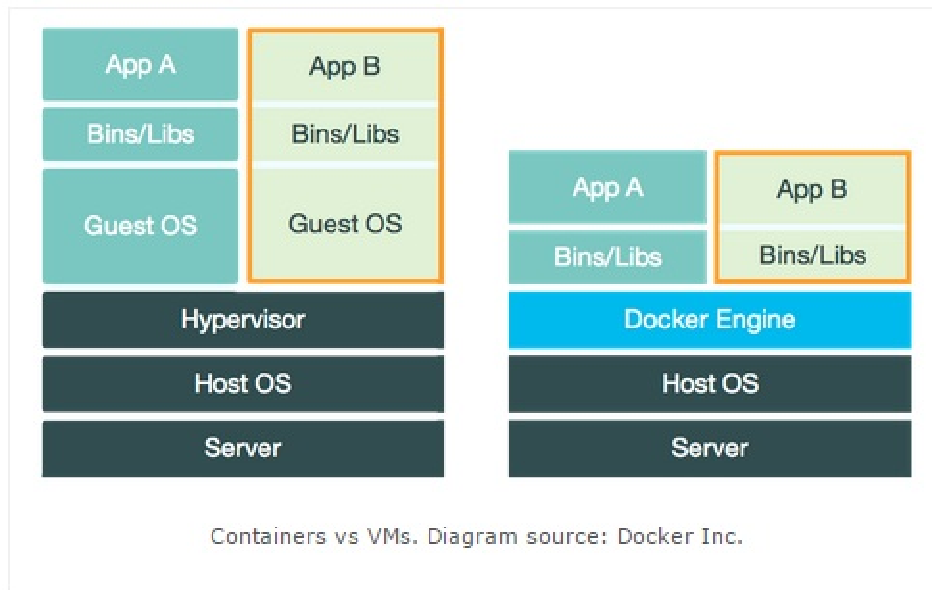
(5) Registry

Docker Registry是一个集中存储与分发镜像的服务。我们构建完Docker镜像后，就可在当前宿主机上运行。但如果想要在其他机器上运行这个镜像，我们就需要手动拷贝。此时，我们可借助Docker Registry来避免镜像的手动拷贝。

一个Docker Registry可包含多个Docker仓库；每个仓库可包含多个镜像标签；每个标签对应一个Docker镜像。这跟Maven的仓库有点类似，如果把Docker Registry比作Maven仓库的话，那么Docker仓库就可理解为某jar包的路径，而镜像标签则可理解为jar包的版本号。

Docker Registry可分为公有Docker Registry和私有Docker Registry。最常用的Docker Registry莫过于官方的Docker Hub，这也是默认的Docker Registry。Docker Hub上存放着大量优秀的镜像，我们可使用Docker命令下载并使用。

1.6 Docker与虚拟机



Containers vs VMs. Diagram source: Docker Inc.

- Hypervisor层被Docker Engine取代。
  - Hypervisor：https://baike.baidu.com/item/hypervisor/3353492
- 虚拟化粒度不同
  - 虚拟机利用Hypervisor虚拟化CPU、内存、IO设备等实现的，然后在其上运行完整的操作系统，再在该系统上运行所需的应用。资源隔离级别：OS级别
  - 运行在Docker容器中的应用直接运行于宿主机的内核，容器共享宿主机的内核，容器内部运行的是Linux副本，没有自己的内核，直接使用物理机的硬件资源，因此CPU/内存利用率上有一定优势。资源隔离级别：利用Linux内核本身支持的容器方式实现资源和环境隔离。
- 拓展阅读
  - 《Docker、LXC、Cgroup的结构关系》：http://speakingbaicai.blog.51cto.com/5667326/1359825/

1.7 Docker应用场景

- 八个Docker的真实应用场景：http://dockone.io/article/126

Docker安装

2.1 CentOS

2.1.1 系统要求

- **CentOS 7**或更高版本
- `centos-extras` 仓库必须处于启用状态，该仓库默认启用，但如果您禁用了该仓库，请按照https://wiki.centos.org/AdditionalResources/Repositories 中的描述重新启用。
- 建议使用 `overlay2` 存储驱动

2.1.2 yum安装

2.1.2.1 卸载老版本的Docker

在CentOS中，老版本Docker名称是 `docker` 或 `docker-engine` ，而Docker CE的软件包名称是 `docker-ce` 。因此，如已安装过老版本的Docker，需使用如下命令卸载。

```
sudo yum remove docker \
                  docker-common \
                  docker-selinux \
                  docker-engine
```

需要注意的是，执行该命令只会卸载Docker本身，而不会删除Docker存储的文件，例如镜像、容器、卷以及网络文件等。这些文件保存在 `/var/lib/docker` 目录中，需要手动删除。

2.1.2.2 安装仓库

1. 执行以下命令，安装Docker所需的包。其中， `yum-utils` 提供了 `yum-config-manager` 工具； `device-mapper-persistent-data` 及 `lvm2` 则是 `devicemapper` 存储驱动所需的包。

   ```
   sudo yum install -y yum-utils device-mapper-persistent-data lvm2
   ```

2. 执行如下命令，安装 `stable` 仓库。必须安装 `stable` 仓库，即使你想安装 `edge` 或 `test` 仓库中的Docker构建版本。

   ```
   sudo yum-config-manager \
       --add-repo \
       https://download.docker.com/linux/centos/docker-ce.repo
   ```

3. **[可选]** 执行如下命令，启用 `edge` 及 `test` 仓库。edge/test仓库其实也包含在了 `docker.repo` 文件中，但默认是禁用的，可使用以下命令来启用。

```
sudo yum-config-manager --enable docker-ce-edge    # 启用edge仓库
sudo yum-config-manager --enable docker-ce-test    # 启用test仓库
```

如需再次禁用，可加上 `--disable` 标签。例如，执行如下命令即可禁用edge仓库。

```
sudo yum-config-manager --disable docker-ce-edge
```

TIPS：从Docker 17.06起，stable版本也会发布到edge以及test仓库中。

2.1.2.3 安装Docker CE

1. 执行以下命令，更新 `yum` 的包索引

```
sudo yum makecache fast
```

2. 执行如下命令即可安装最新版本的Docker CE

```
sudo yum install docker-ce
```

3. 在生产环境中，可能需要指定想要安装的版本，此时可使用如下命令列出当前可用的Docker版本。

```
yum list docker-ce.x86_64  --showduplicates | sort -r
```

这样，列出版本后，可使用如下命令，安装想要安装的Docker CE版本。

```
sudo yum install docker-ce-<VERSION>
```

4. 启动Docker

```
sudo systemctl start docker
```

5. 验证安装是否正确。

```
sudo docker run hello-world
```

这样，Docker将会下载测试镜像，并使用该镜像启动一个容器。如能够看到类似如下的输出，则说明安装成功。

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b04784fba78d: Pull complete
Digest: sha256:f3b3b28a45160805bb16542c9531888519430e9e6d6ffc09d72261b0d26ff74f
```

```
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

2.1.2.4 升级Docker CE

如需升级Docker CE，只需执行如下命令：

```
sudo yum makecache fast
```

然后按照安装Docker的步骤，即可升级Docker。

2.1.2.5 参考文档

CentOS 7安装Docker官方文档：https://docs.docker.com/engine/installation/linux/docker-ce/centos/ ，文档中还讲解了在CentOS 7中安装Docker CE的其他方式，本文不作赘述。

2.1.3 shell一键安装

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

搞定一切。

2.2 Ubuntu

2.2.1 系统要求

- Docker支持以下版本的Ubuntu，要求64位。
  - Zesty 17.04
  - Xenial 16.04 (LTS)
  - Trusty 14.04 (LTS)
- 支持运行的平台： `x86_64` 、 `armhf` 、 `s390x(IBM Z)` 。其中，如选择IBM Z，那么只支持Ubuntu Xenial以及Zesty。
- 本文使用**Ubuntu 16.04 LTS**，下载地址：http://cn.ubuntu.com/download/

2.2.2 安装步骤

2.2.2.1 卸载老版本Docker

在Ubuntu中，老版本的软件包名称是 `docker` 或者 `docker-engine` ，而Docker CE的软件包名称是 `docker-ce` 。因此，如已安装过老版本的Docker，需要先卸载掉。执行以下命令，即可卸载老版本的Docker及其依赖。

```
sudo apt-get remove docker docker-engine docker.io
```

需要注意的是，执行该命令只会卸载Docker本身，而不会删除Docker内容，例如镜像、容器、卷以及网络。这些文件保存在 `/var/lib/docker` 目录中，需要手动删除。

2.2.2.2 Ubuntu Trusty 14.04 额外建议安装的包

除非你有不得已的苦衷，否则强烈建议安装 `linux-image-extra-*` 软件包，以便于Docker使用 `aufs` 存储驱动。执行如下命令，即可安装 `linux-image-extra-*` 。

```
sudo apt-get update

sudo apt-get install \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

对于Ubuntu 16.04或更高版本，Linux内核包含了对OverlayFS的支持，Docker CE默认会使用 `overlay2` 存储驱动。

2.2.2.3 安装仓库

1. 执行如下命令，更新 `apt` 的包索引。

   ```
   sudo apt-get update
   ```

2. 执行如下命令，从而允许 `apt` 使用HTTPS仓库。

   ```
   sudo apt-get install \
       apt-transport-https \
   ```

```
    ca-certificates \
    curl \
    software-properties-common
```

3. 添加Docker官方的GPG key

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

确认指纹是 `9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88` 。

```
sudo apt-key fingerprint 0EBFCD88
```

4. 执行如下命令，安装 `stable` 仓库。无论如何都必须安装 `stable` 仓库，即使你想安装 `edge` 或 `test` 仓库中的Docker构建。如需添加 `edge` 或 `test` 仓库，可在如下命令中的“stable" 后，添加 `edge` 或 `test` 或两者。请视自己Ubuntu所运行的平台来执行如下命令。 **NOTE**：如下命令中的 `lsb_release -cs` 子命令用于返回您Ubuntu的发行版名称，例如 `xenial` 。有时，在例如Linux Mint这样的发行版中，您可能需要将如下命令中的 `$(lsb_release -cs)` 更改为系统的父级Ubuntu发行版。例如，如果您使用的是Linux Mint Rafaela，则可以使用 `trusty` 。
amd64:

```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

armhf:

```
$ sudo add-apt-repository \
   "deb [arch=armhf] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

s390x:

```
$ sudo add-apt-repository \
   "deb [arch=s390x] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

NOTE：从Docker 17.06起，stable版本也会发布到edge以及test仓库中。

2.2.2.4 安装Docker CE

1. 执行如下命令，更新 `apt` 包索引。

```
sudo apt-get update
```

2. 执行如下命令，即可安装最新版本的Docker CE。任何已存在的Docker将会被覆盖安装。

   ```
   sudo apt-get install docker-ce
   ```

   WARNING：如启用了多个Docker仓库，使用命令apt-get install 或apt-get update 命令安装或升级时，如未指定版本，那么将会安装最新的版本。这可能不适合您的稳定性要求。

3. 在生产环境中，我们可能需要指定想要安装的版本，此时可使用如下命令列出当前可用的Docker版本。

   ```
   apt-cache madison docker-ce
   ```

   这样，列出版本后，可使用如下命令，安装想要安装的Docker CE版本。

   ```
   sudo apt-get install docker-ce=<VERSION>
   ```

   Docker daemon会自动启动。

4. 验证安装是否正确。

   ```
   sudo docker run hello-world
   ```

2.2.2.5 升级Docker CE

如需升级Docker CE，只需执行如下命令：

```
sudo apt-get update
```

然后按照安装Docker的步骤，即可升级Docker。

2.2.2.6 参考文档

Ubuntu安装Docker官方文档：https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/ ，文档还讲解了在Ubuntu中安装Docker CE的其他方式，本文不作赘述。

2.3 macOS

2.3.1 系统要求

macOS Yosemite 10.10.3或更高版本

2.3.2 安装步骤

- 前往https://store.docker.com/editions/community/docker-ce-desktop-mac ，点击页面右侧的“Get Docker”按钮，下载安装包；
- 双击即可安装。

2.4 Windows(docker for windows)

2.4.1 系统要求

Windows 10 Professional 或 Windows 10 Enterprise X64

对于Win 7，可使用Docker Toolbox（不建议使用）

2.4.2 安装步骤

- 前往https://store.docker.com/editions/community/docker-ce-desktop-windows ，点击页面右侧的“Get Docker”按钮，下载安装包；
- 双击即可安装。

2.5 其他系统

详见官方文档：https://docs.docker.com/engine/installation/

2.6 加速安装

注册阿里云，参考该页面的内容安装即可：https://cr.console.aliyun.com/#/accelerator

配置镜像加速器

国内访问Docker Hub的速度很不稳定，有时甚至出现连接不上的情况。本节我们来为Docker配置镜像加速器，从而解决这个问题。目前国内很多云服务商都提供了镜像加速的服务。

常用的镜像加速器有：阿里云加速器、DaoCloud加速器等。各厂商镜像加速器的使用方式大致类似，笔者以阿里云加速器为例进行讲解。

1. 注册阿里云账号后，即可在阿里云控制台（https://cr.console.aliyun.com/#/accelerator）看到类似如下的页面。

2. 按照图中的说明，即可配置镜像加速器。

Docker镜像常用命令

我们首先来讨论Docker镜像的常用命令。

搜索镜像

可使用 `docker search` 命令搜索存放在Docker Hub中的镜像。

命令格式：

```
docker search [OPTIONS] TERM
```

参数：

Name, shorthand	Default	Description
`--automated`	`false`	只列出自动构建的镜像
`--filter, -f`		根据指定条件过滤结果
`--limit`	`25`	搜索结果的最大条数
`--no-trunc`	`false`	不截断输出，显示完整的输出
`--stars, -s`	`0`	只展示Star不低于该数值的结果

示例1：

```
docker search java
```

执行该命令后，Docker就会在Docker Hub中搜索含有“java”这个关键词的镜像仓库。执行该命令后，可看到类似于如下的表格：

```
NAME                     DESCRIPTION                     STARS     OFFICIAL   AUTOMATED
java                     Java is a concurrent, ...        1281      [OK]
anapsix/alpine-java      Oracle Java 8 (and 7) ...        190                  [OK]
isuper/java-oracle       This repository conta ...        48                   [OK]
lwieske/java-8           Oracle Java 8 Contain ...        32                   [OK]
nimmis/java-centos       This is docker images ...        23                   [OK]
...
```

该表格包含五列，含义如下：

① NAME：镜像仓库名称。

② DESCRIPTION：镜像仓库描述。

③ STARS：镜像仓库收藏数，表示该镜像仓库的受欢迎程度，类似于GitHub的Stars。

④ OFFICAL：表示是否为官方仓库，该列标记为[OK]的镜像均由各软件的官方项目组创建和维护。由结果可知，java这个镜像仓库是官方仓库，而其他的仓库都不是镜像仓库。

⑤ AUTOMATED：表示是否是自动构建的镜像仓库。

示例2：

```
docker search -s 10 java
```

下载镜像[重要]

使用命令 `docker pull` 命令即可从Docker Registry上下载镜像。

命令格式：

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

参数：

Name, shorthand	Default	Description
`--all-tags, -a`	`false`	下载所有标签的镜像
`--disable-content-trust`	`true`	忽略镜像的校验

示例1：

```
docker pull java
```

执行该命令后，Docker会从Docker Hub中的java仓库下载最新版本的Java镜像。

示例2：

该命令还可指定想要下载的镜像标签以及Docker Registry地址，例如：

```
docker pull reg.itmuch.com/java:7
```

这样就可以从指定的Docker Registry中下载标签为7的Java镜像。

列出镜像[重要]

使用 `docker images` 命令即可列出已下载的镜像。

执行该命令后，将会看到类似于如下的表格：

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZ
E
java                latest              861e95c114d6        4 weeks ago         643
```

```
.1 MB
hello-world         latest              c54a2cc56cbb        5 months ago        1.8
48 kB
```

该表格包含了5列，含义如下：

① REPOSITORY：镜像所属仓库名称。

② TAG：镜像标签。默认是latest，表示最新。

③ IMAGE ID：镜像ID，表示镜像唯一标识。

④ CREATED：镜像创建时间。

⑤ SIZE：镜像大小。

命令格式：

```
docker images [OPTIONS] [REPOSITORY[:TAG]]
```

参数：

Name, shorthand	Default	Description
`--all, -a`	`false`	列出本地所有的镜像（含中间映像层，默认情况下，过滤掉中间映像层）
`--digests`	`false`	显示摘要信息
`--filter, -f`		显示满足条件的镜像
`--format`		通过Go语言模板文件展示镜像
`--no-trunc`	`false`	不截断输出，显示完整的镜像信息
`--quiet, -q`	`false`	只显示镜像ID

示例：

```
docker images
docker images java
docker images java:8
docker images --digests
docker images --filter "dangling=true"   # 展示虚悬镜像
```

删除本地镜像[重要]

使用 `docker rmi` 命令即可删除指定镜像。

命令格式：

```
docker rmi [OPTIONS] IMAGE [IMAGE...]
```

```
```

参数：

Name, shorthand	Default	Description
`--force, -f`	`false`	强制删除
`--no-prune`	`false`	不移除该镜像的过程镜像，默认移除

例1：删除指定名称的镜像。

```
docker rmi hello-world
```

表示删除hello-world这个镜像。

例2：删除所有镜像。

```
docker rmi -f $(docker images)
```

-f参数表示强制删除。

保存镜像

使用 `docker save` 即可保存镜像。

命令格式：

```
docker save [OPTIONS] IMAGE [IMAGE...]
```

参数：

Name, shorthand	Default	Description
`--output, -o`		Write to a file, instead of STDOUT

例1：

```
docker save busybox > busybox.tar
docker save --output busybox.tar busybox
```

加载镜像

使用 `docker load` 命令即可加载镜像。

命令格式：

```
docker load [OPTIONS]
```

参数：

Name, shorthand	Default	Description
`--input, -i`		从文件加载而非STDIN
`--quiet, -q`	`false`	静默加载

例1：

```
docker load < busybox.tar.gz
docker load --input fedora.tar
```

构建镜像[重要]

通过Dockerfile构建镜像。

命令格式：

```
docker build [OPTIONS] PATH | URL | -
```

参数：

Name, shorthand	Default	Description
`--add-host`		添加自定义从host到IP的映射，格式为（host:ip）
`--build-arg`		设置构建时的变量
`--cache-from`		作为缓存源的镜像
`--cgroup-parent`		容器可选的父cgroup
`--compress`	`false`	使用gzip压缩构建上下文
`--cpu-period`	`0`	限制CPU CFS (Completely Fair Scheduler) 周期
`--cpu-quota`	`0`	限制CPU CFS (Completely Fair Scheduler) 配额
`--cpu-shares, -c`	`0`	CPU使用权重（相对权重）
`--cpuset-cpus`		指定允许执行的CPU
`--cpuset-mems`		指定允许执行的内存
`--disable-content-trust`	`true`	忽略校验
`--file, -f`		指定Dockerfile的名称，默认是‘PATH/Dockerfile’
`--force-rm`	`false`	删除中间容器
`--iidfile`		将镜像ID写到文件中
`--isolation`		容器隔离技术
`--label`		设置镜像使用的元数据

`--memory, -m`	`0`	设置内存限制
`--memory-swap`	`0`	设置Swap的最大值为内存+swap，如果设置为-1表示不限swap
`--network`	`default`	在构建期间设置RUN指令的网络模式
`--no-cache`	`false`	构建镜像过程中不使用缓存
`--pull`	`false`	总是尝试去更新镜像的新版本
`--quiet, -q`	`false`	静默模式，构建成功后只输出镜像ID
`--rm`	`true`	构建成功后立即删除中间容器
`--security-opt`		安全选项
`--shm-size`	`0`	指定 `/dev/shm` 目录的大小
`--squash`	`false`	将构建的层压缩成一个新的层
`--tag, -t`		设置标签，格式：name:tag，tag可选
`--target`		设置构建时的目标构建阶段
`--ulimit`		Ulimit 选项

拓展阅读

Docker命令：https://docs.docker.com/engine/reference/commandline/docker/

容器常用命令

本节我们来讨论Docker容器的常用命令。

新建并启动容器[重要]

使用以下 `docker run` 命令即可新建并启动一个容器。该命令是我们最常用的命令了，它有很多选项，下面笔者列举一些常用的选项。

① -d选项：表示后台运行

② -P选项：随机端口映射

③ -p选项：指定端口映射，有以下四种格式。

ip:hostPort:containerPort

ip::containerPort

hostPort:containerPort

containerPort

④ --network选项：指定网络模式，该选项有以下可选参数：

--network=bridge： 默认选项，表示连接到默认的网桥。

--network=host：容器使用宿主机的网络。

--network=container:NAME_or_ID：告诉Docker让新建的容器使用已有容器的网络配置。

--network=none：不配置该容器的网络，用户可自定义网络配置。

示例1：

```
docker run java /bin/echo 'Hello World'
```

这样终端会打印Hello World的字样，跟在本地直接执行 `/bin/echo 'Hello World'` 一样。

示例2：

```
docker run -d -p 91:80 nginx
```

这样就能启动一个Nginx容器。在本例中，我们为docker run添加了两个参数，含义如下：

```
-d                            # 后台运行
-p 宿主机端口:容器端口          # 开放容器端口到宿主机端口
```

访问http://Docker宿主机IP:91/ ，将会看到如图12-3的界面：

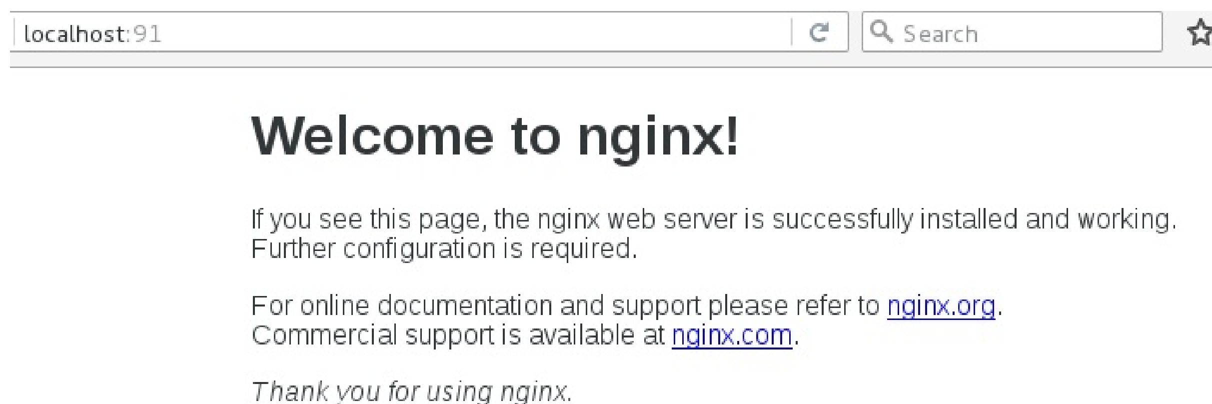


图12-3 Nginx首页

TIPS

需要注意的是，使用docker run命令创建容器时，会先检查本地是否存在指定镜像。如果本地不存在该名称的镜像，Docker就会自动从Docker Hub下载镜像并启动一个Docker容器。

列出容器[重要]

使用 `docker ps` 命令即可列出运行中的容器。执行该命令后，可看到类似于如下的表格。

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
  STATUS              PORTS                         NAMES
784fd3b294d7        nginx               "nginx -g 'daemon off"   20 minutes ago
  Up 2 seconds        443/tcp, 0.0.0.0:91->80/tcp   backstabbing_archimedes
```

如需列出所有容器（包括已停止的容器），可使用-a参数。

该表格包含了七列，含义如下：

① CONTAINER_ID：表示容器ID。

② IMAGE：表示镜像名称。

③ COMMAND：表示启动容器时运行的命令。

④ CREATED：表示容器的创建时间。

⑤ STATUS：表示容器运行的状态。Up表示运行中，Exited表示已停止。

⑥ PORTS：表示容器对外的端口号。

⑦ NAMES：表示容器名称。该名称默认由Docker自动生成，也可使用docker run命令的--name选项自行指定。

命令格式：

```
docker ps [OPTIONS]
```

参数：

Name, shorthand	Default	Description
`--all, -a`	`false`	列出所有容器，包括未运行的容器，默认只展示运行的容器
`--filter, -f`		根据条件过滤显示内容
`--format`		通过Go语言模板文件展示镜像
`--last, -n`	`-1`	显示最近创建n个容器（包含所有状态）
`--latest, -l`	`false`	显示最近创建的容器（包含所有状态）
`--no-trunc`	`false`	不截断输出
`--quiet, -q`	`false`	静默模式，只展示容器的编号
`--size, -s`	`false`	显示总文件大小

示例：

```
docker ps -n 5
docker ps -a -q
```

停止容器[重要]

使用 `docker stop` 命令，即可停止容器。

命令格式：

```
docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

参数：

Name, shorthand	Default	Description
`--time, -t`	`10`	强制杀死容器前等待的时间，单位是秒

示例：

```
docker stop 784fd3b294d7
```

其中 `784fd3b294d7` 是容器ID，当然也可使用 `docker stop 容器名称` 来停止指定容器。

强制停止容器[重要]

可使用 `docker kill` 命令停止一个或更多运行着的容器。

命令格式：

```
docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

参数：

Name, shorthand	Default	Description
`--signal, -s`	`KILL`	向容器发送一个信号

例如：

```
docker kill 784fd3b294d7
```

启动已停止的容器[重要]

使用 `docker run` 命令，即可新建并启动一个容器。对于已停止的容器，可使用 `docker start` 命令来启动。

命令格式：

```
docker start [OPTIONS] CONTAINER [CONTAINER...]
```

参数：

Name, shorthand	Default	Description
`--attach, -a`	`false`	连接STDOUT/STDERR并转发信号
`--checkpoint`		从该检查点还原
`--checkpoint-dir`		使用自定义的检查点存储目录
`--detach-keys`		覆盖断开容器的关键顺序
`--interactive, -i`	`false`	连接容器的STDIN

例如：

```
docker start 784fd3b294d7
```

重启容器[重要]

可使用 `docker restart` 命令来重启容器。该命令实际上是先执行了 `docker stop` 命令，然后执行了 `docker start` 命令。

命令格式：

```
docker restart [OPTIONS] CONTAINER [CONTAINER...]
```

参数：

Name, shorthand	Default	Description
`--time, -t`	`10`	关闭容器前等待的时间，单位是秒

进入容器[重要]

某场景下，我们可能需要进入运行中的容器。

① 使用 `docker attach` 命令进入容器。

例如：

```
docker attach 784fd3b294d7
```

很多场景下，使用 `docker attach` 命令并不方便。当多个窗口同时attach到同一个容器时，所有窗口都会同步显示。同理，如果某个窗口发生阻塞，其他窗口也无法执行操作。

② 使用 `nsenter` 进入容器

nsenter工具包含在util-linux 2.23或更高版本中。为了连接到容器，我们需要找到容器第一个进程的PID，可通过以下命令获取：

```
docker inspect --format "{{.State.Pid}}" $CONTAINER_ID
```

获得PID后，就可使用nsenter命令进入容器了：

```
nsenter --target "$PID" --mount --uts --ipc --net --pid
```

下面给出一个完整的例子：

```
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
  STATUS              PORTS                                NAMES
784fd3b294d7        nginx               "nginx -g 'daemon off"   55 minutes ago
  Up 3 minutes        443/tcp, 0.0.0.0:91->80/tcp   backstabbing_archimedes
[root@localhost ~]# docker inspect --format "{{.State.Pid}}" 784fd3b294d7
95492
[root@localhost ~]# nsenter --target 95492 --mount --uts --ipc --net --pid
root@784fd3b294d7:/#
```

读者也可将以上两条命令封装成一个Shell，从而简化进入容器的过程。

③ docker exec

```
docker exec -it 容器id /bin/bash
```

删除容器[重要]

使用 `docker rm` 命令即可删除指定容器。

命令格式

```
docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

参数：

Name, shorthand	Default	Description
`--force, -f`	`false`	通过SIGKILL信号强制删除正在运行中的容器
`--link, -l`	`false`	删除容器间的网络连接
`--volumes, -v`	`false`	删除与容器关联的卷

例1：删除指定容器。

```
docker rm 784fd3b294d7
```

该命令只能删除已停止的容器，如需删除正在运行的容器，可使用-f参数。

例2：删除所有的容器。

```
docker rm -f $(docker ps -a -q)
```

导出容器

将容器导出成一个压缩包文件。

命令格式：

```
docker export [OPTIONS] CONTAINER
```

参数：

Name, shorthand	Default	Description
`--output, -o`		将内容写到文件而非STDOUT

示例：

```
docker export red_panda > latest.tar
docker export --output="latest.tar" red_panda
```

导入容器

使用 `docker import` 命令即可从归档文件导入内容并创建镜像。

命令格式：

```
docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
```

参数：

Name, shorthand	Default	Description
`--change, -c`		将Dockerfile指令应用到创建的镜像
`--message, -m`		为导入的镜像设置提交信息

示例：

```
docker import nginx2.tar nginx
```

拓展阅读

- Docker的网络：https://docs.docker.com/engine/userguide/networking/
- Docker命令：https://docs.docker.com/engine/reference/commandline/docker/

实战：修改Nginx首页

6.1 需求

- 启动一个Nginx容器。
- 将Nginx容器的首页改为 `Welcome to 51CTO docker class` 。
- 将容器保存下来。

6.2 提示

- Nginx默认首页目录在： `/usr/share/nginx/html/index.html`

答案

```
docker exec -it nginx容器ID /bin/bash   # 进入容器
```

执行如下命令，修改/usr/share/nginx/html/index.html

```
tee index.html <<-'EOF'
Welcome to 51CTO docker class
EOF
```

Dockerfile指令详解

在前面的例子中，我们提到了FROM、RUN指令。事实上，Dockerfile有十多个指令。本节我们来系统讲解这些指令，指令的一般格式为 `指令名称 参数` 。

ADD 复制文件

ADD指令用于复制文件，格式为：

- `ADD <src>... <dest>`
- `ADD ["<src>",... "<dest>"]`

从src目录复制文件到容器的dest。其中src可以是Dockerfile所在目录的相对路径，也可以是一个URL，还可以是一个压缩包

注意：

① src必须在构建的上下文内，不能使用例如： `ADD ../somethine /something` 这样的命令，因为 `docker build` 命令首先会将上下文路径和其子目录发送到docker daemon。

② 如果src是一个URL，同时dest不以斜杠结尾，dest将会被视为文件，src对应内容文件将会被下载到dest。

③ 如果src是一个URL，同时dest以斜杠结尾，dest将被视为目录，src对应内容将会被下载到dest目录。

④ 如果src是一个目录，那么整个目录下的内容将会被拷贝，包括文件系统元数据。

⑤ 如果文件是可识别的压缩包格式，则docker会自动解压。

示例：

```
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
```

ARG 设置构建参数

ARG指令用于设置构建参数，类似于ENV。和ARG不同的是，ARG设置的是构建时的环境变量，在容器运行时是不会存在这些变量的。

格式为：

- `ARG <name>[=<default value>]`

示例：

```
ARG user1=someuser
```

详细介绍文档：https://www.centos.bz/2016/12/dockerfile-arg-instruction/

CMD 容器启动命令

CMD指令用于为执行容器提供默认值。每个Dockerfile只有一个CMD命令，如果指定了多个CMD命令，那么只有最后一条会被执行，如果启动容器的时候指定了运行的命令，则会覆盖掉CMD指定的命令。

支持三种格式：

`CMD ["executable","param1","param2"]` (推荐使用)

`CMD ["param1","param2"]` (为ENTRYPOINT指令提供预设参数)

`CMD command param1 param2` (在shell中执行)

示例：

```
CMD echo "This is a test." | wc -
```

COPY 复制文件

复制文件，格式为：

- `COPY <src>... <dest>`
- `COPY ["<src>",... "<dest>"]`

复制本地端的src到容器的dest。COPY指令和ADD指令类似，COPY不支持URL和压缩包。

ENTRYPOINT 入口点

格式为：

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

ENTRYPOINT和CMD指令的目的一样，都是指定Docker容器启动时执行的命令，可多次设置，但只有最后一个有效。ENTRYPOINT不可被重写覆盖。

ENTRYPOINT、CMD区别：http://blog.csdn.net/newjueqi/article/details/51355510

ENV 设置环境变量

ENV指令用于设置环境变量，格式为：

- `ENV <key> <value>`
- `ENV <key>=<value> ...`

示例：

```
ENV JAVA_HOME /path/to/java
```

EXPOSE 声明暴露的端口

EXPOSE指令用于声明在运行时容器提供服务的端口，格式为：

- `EXPOSE <port> [<port>...]`

需要注意的是，这只是一个声明，运行时并不会因为该声明就打开相应端口。该指令的作用主要是帮助镜像使用者理解该镜像服务的守护端口；其次是当运行时使用随机映射时，会自动映射EXPOSE的端口。示例：

```
# 声明暴露一个端口示例
EXPOSE port1
# 相应的运行容器使用的命令
docker run -p port1 image
# 也可使用-P选项启动
docker run -P image

# 声明暴露多个端口示例
EXPOSE port1 port2 port3
# 相应的运行容器使用的命令
docker run -p port1 -p port2 -p port3 image
# 也可指定需要映射到宿主机器上的端口号
docker run -p host_port1:port1 -p host_port2:port2 -p host_port3:port3 image
```

FROM 指定基础镜像

使用FROM指令指定基础镜像，FROM指令有点像Java里面的“extends”关键字。需要注意的是，FROM指令必须指定且需要写在其他指令之前。FROM指令后的所有指令都依赖于该指令所指定的镜像。

支持三种格式：

- `FROM <image>`
- `FROM <image>:<tag>`
- `FROM <image>@<digest>`

LABEL 为镜像添加元数据

LABEL指令用于为镜像添加元数据。

格式为：

- `LABEL <key>=<value> <key>=<value> <key>=<value> ...`

使用 ”"“和”\“转换命令行，示例：

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

MAINTAINER 指定维护者的信息（已过时）

MAINTAINER指令用于指定维护者的信息，用于为Dockerfile署名。

格式为：

- `MAINTAINER <name>`

示例：

```
MAINTAINER 周立<eacdy0000@126.com>
```

注：该指令已过时，建议使用如下形式：

```
LABEL maintainer="SvenDowideit@home.org.au"
```

RUN 执行命令

该指令支持两种格式：

- `RUN <command>`
- `RUN ["executable", "param1", "param2"]`

`RUN <command>` 在shell终端中运行，在Linux中默认是 `/bin/sh -c` ，在Windows中是 `cmd /s /c` ，使用这种格式，就像直接在命令行中输入命令一样。 `RUN ["executable", "param1", "param2"]` 使用exec执行，这种方式类似于函数调用。指定其他终端可以通过该方式操作，例如： `RUN ["/bin/bash", "-c", "echo hello"]` ，该方式必须使用双引号["]而不能使用单引号['],因为该方式会被转换成一个JSON 数组。

USER 设置用户

该指令用于设置启动镜像时的用户或者UID，写在该指令后的RUN、CMD以及ENTRYPOINT指令都将使用该用户执行命令。

格式为：

- `USER 用户名`

示例：

```
USER daemon
```

VOLUME 指定挂载点

该指令使容器中的一个目录具有持久化存储的功能，该目录可被容器本身使用，也可共享给其他容器。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。格式为：

- `VOLUME ["/data"]`

示例：

```
VOLUME /data
```

使用示例：

```
FROM nginx
VOLUME /tmp
```

当该Dockerfile被构建成镜像后，/tmp目录中的数据即使容器关闭也依然存在。如果另一个容器也有持久化的需求，并且想使用以上容器/tmp目录中的内容，则可使用如下命令启动容器：

```
docker run -volume-from 容器ID 镜像名称  # 容器ID是di一个容器的ID，镜像是第二个容器所使用的镜像。
```

WORKDIR 指定工作目录

格式为：

- `WORKDIR /path/to/workdir`

切换目录指令，类似于cd命令，写在该指令后的 `RUN` ， `CMD` 以及 `ENTRYPOINT` 指令都将该目录作为当前目录，并执行相应的命令。

其他

Dockerfile还有一些其他的指令，例如STOPSINGAL、HEALTHCHECK、SHELL等。由于并不是很常用，本书不作赘述。有兴趣的读者可前往https://docs.docker.com/engine/reference/builder/ 扩展阅读。

CMD/ENTRYPOINT/RUN区别

参考：https://segmentfault.com/q/1010000000417103

拓展阅读

- Dockerfile官方文档：https://docs.docker.com/engine/reference/builder/#dockerfile-reference
- Dockerfile最佳实践：https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#build-cache

实战：使用Dockerfile修改Nginx首页

创建一个Dockerfile，内容如下：

```
FROM nginx
RUN echo '<h1>Spring Cloud与Docker微服务实战</h1>' > /usr/share/nginx/html/index.html
```

巩固-阅读常用软件的Dockerfile

- Nginx：https://github.com/nginxinc/docker-nginx/blob/849fed0093112cd9f55491fccd2f861eb9fad5f9/stable/alpine/Dockerfile
- Tomcat：https://github.com/docker-library/tomcat/blob/0e9a915bf893faa9160ab1a144c7ba5049a4fe27/7/jre7-alpine/Dockerfile
- 关于Alpine Linux：http://www.cnblogs.com/zhangmingcheng/p/7122386.html

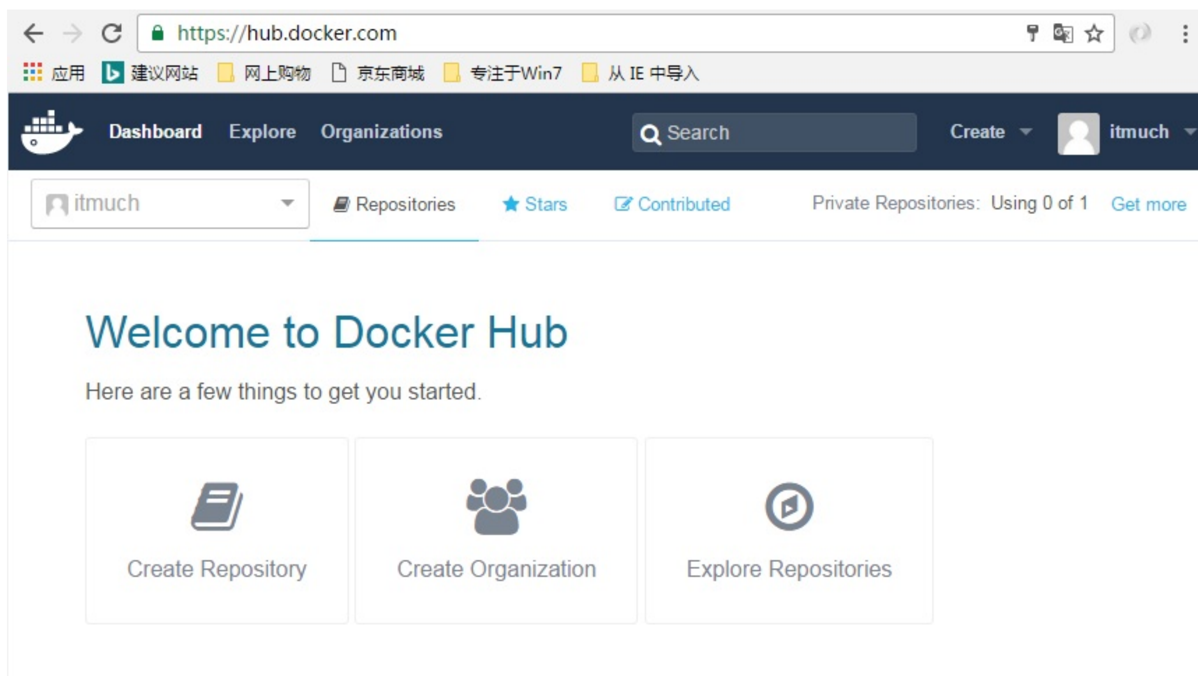
使用Docker Hub管理镜像

Docker Hub是Docker官方维护的Docker Registry，上面存放着很多优秀的镜像。不仅如此，Docker Hub还提供认证、工作组结构、工作流工具、构建触发器等工具来简化我们的工作。

前文已经讲过，我们可使用 `docker search` 命令搜索存放在Docker Hub中的镜像。本节我们来详细探讨Docker Hub的使用。

注册与登录

Docker Hub的使用非常简单，只需注册一个Docker Hub账号，就可正常使用了。登录后，我们可看到Docker Hub的主页，如图所示。

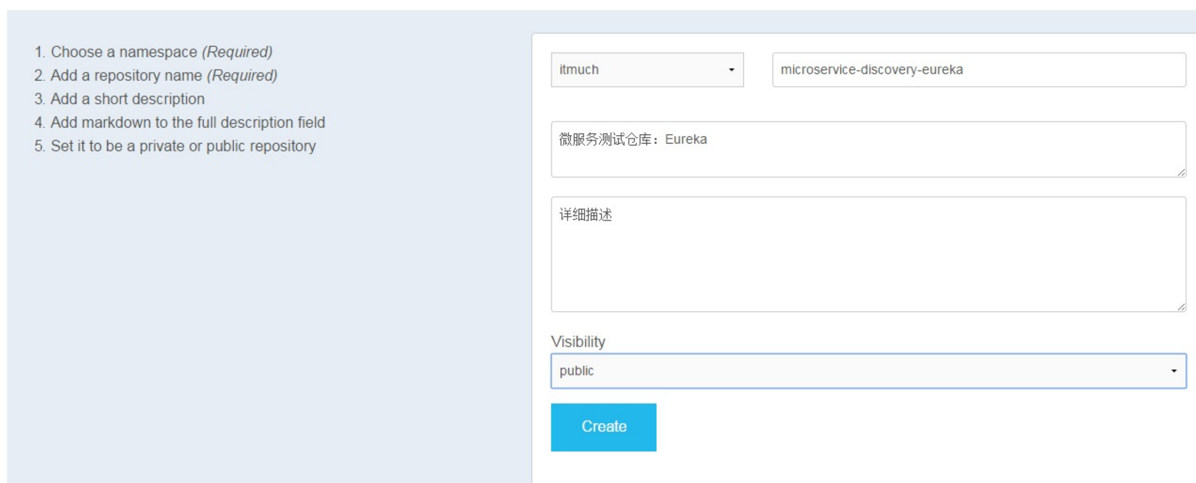


我们也可使用 `docker login` 命令登录Docker Hub。输入该命令并按照提示输入账号和密码，即可完成登录。例如：

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't hav
e a Docker ID, head over to https://hub.docker.com to create one.
Username: itmuch
Password:
Login Succeeded
```

创建仓库

点击Docker Hub主页上的"Create Repository"按钮，按照提示填入信息即可创建一个仓库。



如图，我们只需填入相关信息，并点击Create按钮，就可创建一个名为microservice-discovery-eureka的公共仓库。

推送镜像

下面我们来将前文构建的镜像推送到Docker Hub。使用以下命令即可，例如：

```
docker push itmuch/microservice-discovery-eureka:0.0.1
```

经过一段时间的等待，就可推送成功。这样，我们就可在Docker Hub查看已推送的镜像。

使用Docker Registry管理镜像

很多场景下，我们需使用私有仓库管理Docker镜像。相比Docker Hub，私有仓库有以下优势：

(1) 节省带宽，对于私有仓库中已有的镜像，无需从Docker Hub下载，只需从私有仓库中下载即可；

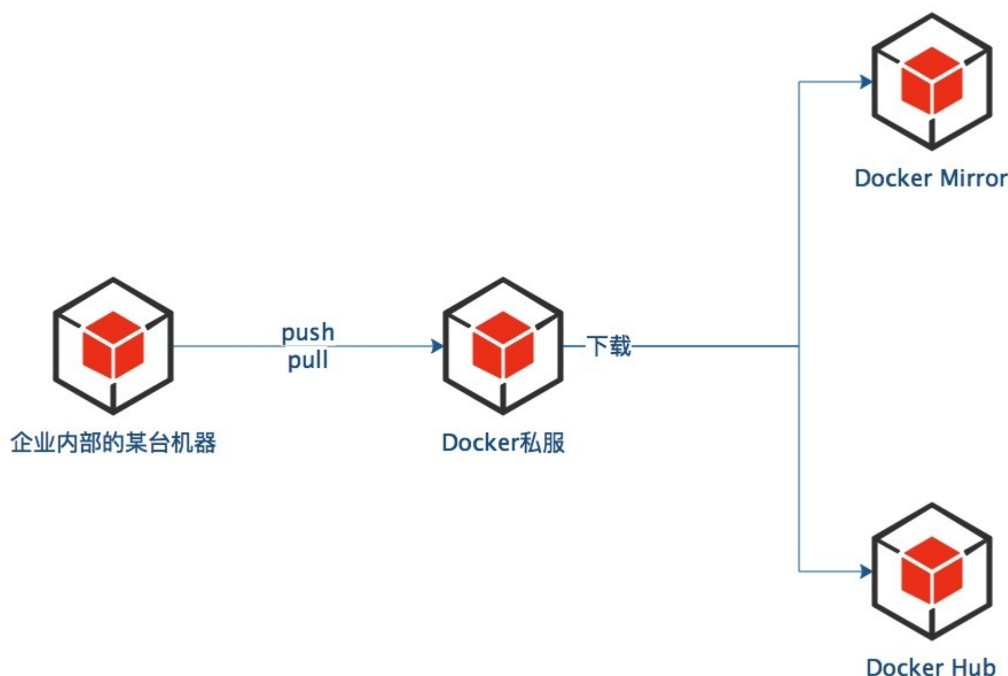
(2) 更加安全；

(3) 便于内部镜像的统一管理。

本节我们来探讨如何搭建、使用私有仓库。可使用docker-registry项目或者Docker Registry 2.0来搭建私有仓库，但docker-registry已被官方标记为过时，并且已有2年不维护了，不建议使用。

我们先用Docker Registry 2.0搭建一个私有仓库，然后将Docker镜像推送到私有仓库。

原理图



搭建Docker Registry 2.0

Docker Registry 2.0的搭建非常简单，只需执行以下命令即可新建并启动一个Docker Registry 2.0。

```
docker run -d -p 5000:5000 --restart=always --name registry2 registry:2
```

将镜像推送到Docker Registry 2.0

前文我们使用 `docker push` 命令将镜像推送到了Docker Hub，现在我们将前文构建的itmuch/microservice-discovery-eureka:0.0.1推送到私有仓库。

只需指定私有仓库的地址，即可将镜像推送到私有仓库。

```
docker push localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```

执行以上命令，我们发现推送并没有成功，且提示以下内容：

```
The push refers to a repository [localhost:5000/itmuch/microservice-discovery-eurek
a]
An image does not exist locally with the tag: localhost:5000/itmuch/microservice-di
scovery-eureka
```

我们知道，Docker Hub是默认的Docker Registry，因此，itmuch/microservice-discovery-eureka:0.0.1相当于docker.io/itmuch/microservice-discovery-eureka:0.0.1。因此，要想将镜像推送到私有仓库，需要修改镜像标签，命令如下：

```
docker tag itmuch/microservice-discovery-eureka:0.0.1 localhost:5000/itmuch/microse
rvice-discovery-eureka:0.0.1
```

修改镜像标签后，再次执行以下命令，即可将镜像推送到私有仓库。

```
docker push localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```

TIPS

(1) docker-registry的GitHub：https://github.com/docker/docker-registry

(2) Docker Registry 2.0的GitHub：https://github.com/docker/distribution

(3) 本节中“私有仓库”表示私有Docker Registry，并非Docker中仓库的概念。

(4) Docker Registry 2.0需要Docker版本高于1.6.0。

(5) 我们还可为私有仓库配置域名、SSL登录、认证等。限于篇幅，本书不作赘述。有兴趣的读者可参考笔者的开源书：https://gitee.com/itmuch/spring-cloud-book/blob/master/3%20%E4%BD%BF%E7%94%A8Docker%E6%9E%84%E5%BB%BA%E5%BE%AE%E6%9C%8D%E5%8A%A1/3.5%20Docker%E7%A7%81%E6%9C%89%E4%BB%93%E5%BA%93%E7%9A%84%E6%90%AD%E5%BB%BA%E4%B8%8E%E4%BD%BF%E7%94%A8.md。

(6) Docker Registry 2.0能够满足我们大部分场景下的需求，但它不包含界面、用户管理、权限控制等功能。如果想要使用这些功能，可使用Docker Trusted Registry。

使用Nexus管理Docker镜像

Nexus简介

Nexus是一个多功能的仓库管理器，是企业常用的私有仓库服务器软件。目前常被用来作为Maven私服、Docker私服。本文基于 `Nexus 3.5.2-01` 版本进行讲解。

Nexus下载

前往：https://www.sonatype.com/download-oss-sonatype ，根据操作系统，下载对应操作系统下的安装包即可。

安装

Nexus在不同系统中安装略有区别，但总体一致。下面以在Linux系统中的安装为例说明：

- 创建一个Linux用户，例如：nexus

  ```
  useradd nexus
  ```

- 解压Nexus安装包，为将解压后的文件设置权限，并修改属主为nexus用户

  ```
  chmod -R 755 *
  chown -R nexus:nexus *
  ```

- 将目录切换到 `$NEXUS_HOME/nexus-3.5.2-01/bin` 目录
- 需改 `nexus.rc` 文件，将其内容改为：

  ```
  run_as_user="nexus"
  ```

  表示使用nexus用户启动Nexus。

- 如提示文件限制，可参考博文：http://www.cnblogs.com/zengkefu/p/5649407.html 进行修改。
- 执行如下命令，查看Nexus为我们提供哪些命令。

  ```
  ./nexus --help
  ```

  可显示类似如下的内容：

  ```
  Usage: ./nexus {start|stop|run|run-redirect|status|restart|force-reload}
  ```

- 指定如下命令，即可启动Nexus

  ```
  ./nexus start
  ```

  稍等片刻，Nexus即可成功启动。

账户

Nexus提供了默认的管理员账户，账号密码分别是admin/admin123。用户可自行修改该默认账号密码。

创建Docker仓库

- 访问http://localhost:8081 并登录
- 点击“Create repository”按钮，创建仓库。Nexus支持多种仓库类型，例如：maven、npm、docker等。本文创建一个docker仓库。一般来说，对于特定的仓库类型（例如docker），细分了三类，分别是proxy、hosted、group，含义如下：
  - hosted，本地代理仓库，通常我们会部署自己的构件到这一类型的仓库，可以push和pull。
  - proxy，代理的远程仓库，它们被用来代理远程的公共仓库，如maven中央仓库，只能pull。
  - group，仓库组，用来合并多个hosted/proxy仓库，通常我们配置maven依赖仓库组，只能pull。
- 本文创建一个**hosted**类型的仓库
- 配置仓库，如图，填入如下结果：

Name: A unique identifier for this repository

docker-repo

Online: ☑ If checked, the repository accepts incoming requests

Repository Connectors

Connectors allow Docker clients to connect directly to hosted registries, but are not always required. Consult our documentation for which connector is appropriate for your use case.

HTTP:

Create an HTTP connector at specified port. Normally used if the server is behind a secure proxy.

☑ 8082

HTTPS:

Create an HTTPS connector at specified port. Normally used if the server is configured for https.

☐

Docker Registry API Support

Enable Docker V1 API:

☑ Allow clients to use the V1 API to interact with this Repository.

Storage

Blob store:

Blob store used to store asset contents

default

Strict Content Type Validation:

☑ Validate that all content uploaded to this repository is of a MIME type appropriate for the repository format

- 这样，仓库就创建完毕了。

Docker配置

下面，我们需要为Docker指定使用Nexus仓库。

- 修改 `/etc/docker/daemon.json` ，在其中添加类似如下的内容。

```
{
  "insecure-registries" : [
    "192.168.1.101:8082"
  ]
  ...
}
```

- 重启Docker

登录私有仓库

```
docker login 192.168.1.101:8082
```

即可登录私有仓库。然后，我们就可进行pull、push操作了。

容器启动Nexus

地址：https://store.docker.com/community/images/sonatype/nexus3

```
docker run -d -p 8081:8081  --name nexus sonatype/nexus3
```

为启动的容器映射端口：http://blog.csdn.net/github_29237033/article/details/46632647

Docker可视化管理工具

DockerUI(ui for Docker)

官方GitHub：https://github.com/kevana/ui-for-docker

项目已废弃，现在转投Portainer项目。

Portainer

简介：Portainer是一个轻量级的管理界面，可以让您轻松地管理不同的Docker环境（Docker主机或Swarm集群）。Portainer提供状态显示面板、应用模板快速部署、容器镜像网络数据卷的基本操作、事件日志显示、容器控制台操作、Swarm集群和服务等集中管理和操作、登录用户管理和控制等功能。功能十分全面，基本能满足中小型单位对容器管理的全部需求。

官方GitHub：https://github.com/portainer/portainer

使用：

```
docker run -d --privileged -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.soc
k -v /opt/portainer:/data portainer/portainer
```

如开启了SELinux，可执行如下命令启动：

```
docker run -d --privileged -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.soc
k -v /opt/portainer:/data portainer/portainer
```

官方文档：https://portainer.readthedocs.io/en/latest/deployment.html

Kitematic

简介：Kitematic是一个Docker GUI。

官方GitHub：https://github.com/docker/kitematic

使用：演示

Shipyard

简介：Shipyard 是一个基于 Web 的 Docker 管理工具，支持多 host，可以把多个 Docker host 上的 containers 统一管理；可以查看 images，甚至 build images；并提供 RESTful API 等。

官方GitHub：https://github.com/shipyard/shipyard

安装：

```
curl -s https://shipyard-project.com/deploy | bash -s
```

展示所有参数：

```
curl -s https://shipyard-project.com/deploy | bash -s -- -h
```

使用：访问http://localhost:8080，输入账号/密码：admin/shipyard即可访问Shipyard。

官方文档：https://shipyard-project.com/

各种可视化界面的比较

参考：http://m.blog.csdn.net/qq273681448/article/details/75007828

Docker数据持久化

容器中数据持久化主要有两种方式：

- 数据卷（Data Volumes）
- 数据卷容器（Data Volumes Dontainers）

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，可以绕过UFS（Unix File System）。

- 数据卷可以在容器之间共享和重用
- 对数据卷的修改会立马生效
- 对数据卷的更新，不会影响镜像
- 数据卷默认会一直存在，即使容器被删除
- 一个容器可以挂载多个数据卷

注意：数据卷的使用，类似于 Linux 下对目录或文件进行 mount。

创建数据卷

示例：

```
docker run --name nginx-data -v /mydir nginx
```

执行如下命令即可查看容器构造的详情：

```
docker inspect 容器ID
```

由测试可知：

- Docker会自动生成一个目录作为挂载的目录。
- **即使容器被删除，宿主机中的目录也不会被删除。**

删除数据卷

数据卷是被设计来持久化数据的，因此，删除容器并不会删除数据卷。如果想要在删除容器时同时删除数据卷，可使用如下命令：

```
docker rm -v 容器ID
```

这样既可在删除容器的同时也将数据卷删除。

挂载宿主机目录作为数据卷

```
docker run --name nginx-data2 -v /host-dir:/container-dir nginx
```

这样既可将宿主机的/host-dir路径加载到容器的/container-dir中。

需要注意的是：

- 宿主机路径**尽量设置**绝对路径——如果使用相对路径会怎样？
  - 测试给答案
- 如果宿主机路径不存在，Docker会自动创建

TIPS

Dockerfile暂时不支持这种形式。

挂载宿主机文件作为数据卷

```
docker run --name nginx-data3 -v /文件路径:/container路径 nginx
```

指定权限

默认情况下，挂载的权限是读写权限。也可使用 `:ro` 参数指定只读权限。

示例：

```
docker run --name nginx-data4 -v /host-dir:/container-dir:ro nginx
```

这样，在容器中就只能读取/container-dir中的文件，而不能修改了。

数据卷容器

如果有数据需要在多个容器之间共享，此时可考虑使用数据卷容器。

创建数据卷容器：

```
docker run --name nginx-volume -v /data nginx
```

在其他容器中使用 `-volumes-from` 来挂载nginx-volume容器中的数据卷。

```
docker run --name v1 --volumes-from nginx-volume nginx
docker run --name v2 --volumes-from nginx-volume nginx
```

这样：

- v1、v2两个容器即可共享nginx-volume这个容器中的文件。
- 即使nginx-volume停止，也不会有任何影响。

端口映射

在前面的内容中，我们很多地方使用了-p参数来实现端口映射。本节我们来详细讲解Docker中的端口映射。

随机映射：-P

当启动时使用-P参数时，即可让Docker随机映射一个端口到容器内部开放的端口

```
docker run -P nginx
```

指定端口映射：-p

指定端口映射有如下几种格式：

- ip:hostPort:containerPort：映射到指定IP的指定端口
- ip::containerPort：映射到指定IP的随机端口
- hostPort:containerPort：映射到宿主机所有IP的指定端口
- containerPort：映射到宿主机所有IP的随机端口

查看端口映射

有多种方式可以查看端口映射的详情。

- 方法1：

  ```
  docker ps
  ```

- 方法2：

  ```
  docker port 容器ID
  ```

遗留特性：容器互联

端口映射实现了外部与容器的网络通信，下面我们来探讨容器之间如何通信。使用 `--link` 参数即可实现容器之间的互联。该参数的格式为： `--link name:alias` ，其中，name是容器的名称，alias则是这个连接的别名。

```
docker run --link nginx:nginx eureka  # 使用eureka镜像启动容器，并将其连接上nginx这个容器
。
```

注：该特性未来可能被删除，不做赘述。相关文档：https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/

Docker容器网络

本节概述了Docker默认的网络行为，包括默认情况下创建的网络类型以及如何创建用户自定义网络。本文也描述了在单个主机或集群上创建网络所需的资源。

有关Docker如何在Linux主机上与 `iptables` 进行交互的详细信息，请参阅Docker和 `iptables` 。

默认网络

当您安装Docker时，它会自动创建三个网络，可使用 `docker network ls` 命令列出这些网络：

```
$ docker network ls

NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
```

Docker内置如上三个网络。 运行容器时，可使用 `--network` 标志来指定容器应连接到哪些网络。

`bridge` 网络代表所有Docker安装中存在的 `docker0` 网络。 除非您使用 `docker run --network=<NETWORK>` 选项，否则Docker守护程序默认将容器连接到此网络。 可使用 `ip addr show` 命令（或简写形式， `ip a` ），xia显示该网桥的信息。 （ `ifconfig` 命令已被弃用，根据系统的不同，还可能会 `command not found` 错误。）

```
$ ip addr show

docker0   Link encap:Ethernet  HWaddr 02:42:47:bc:3a:eb
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:47ff:febc:3aeb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:17 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1100 (1.1 KB)  TX bytes:648 (648.0 B)
```

> **如何在Docker for Mac或Docker for Windows上运行?**
>
> 如果您使用Docker for Mac（或在Docker for Windows上运行Linux容器）， `docker network ls` 命令将按照上述方式工作，但 `ip addr show` 和 `ifconfig` 命令可能会展示结果，但会给你本地主机的IP地址信息，而不是Docker容器网络。 这是因为Docker使用虚拟机中运行的网卡，而并非在宿主机的网卡。

> 要使用 `ip addr show` 或 `ifconfig` 命令浏览Docker网络，请前往Docker Machine 查看相关文档；如您使用的是云提供商，如AWS上的Docker Machine或Digital Ocean上的Docker Machine 。可使用 `docker-machine ssh <machine-name>` 登录到本地或云托管的机器，也可根据云提供商站点上的描述，直接 `ssh` 。

`none` 网络将容器添加到容器特定的网络，该容器缺少网卡。Attach到一个网络为 `none` 模式的容器，将会看到类似如下的内容：

```
$ docker attach nonenetcontainer

root@0cb243cd1293:/# cat /etc/hosts
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
root@0cb243cd1293:/# ifconfig
lo         Link encap:Local Loopback
           inet addr:127.0.0.1  Mask:255.0.0.0
           inet6 addr: ::1/128 Scope:Host
           UP LOOPBACK RUNNING  MTU:65536  Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@0cb243cd1293:/#
```

> **注意**： 可使用 `CTRL-p CTRL-q` 断开容器连接并离开。

`host` 网络模式将容器添加到在宿主机的网络栈上。就网络而言，宿主机和容器之间没有隔离。例如，如果您使用 `host` 网络运行在80端口上运行一个Web服务器容器，则该容器可在宿主机的80端口上使用。

在Docker中， `none` 和 `host` 网络模式不能直接配置。 但是，您可以配置默认的 `bridge` 网络，以及用户自定义的网桥。

默认网桥

所有Docker主机上都有默认的 `bridge` 网络。 如不指定网络，容器将自动连接到默认的 `bridge` 网络。

`docker network inspect` 命令返回有关网络的信息：

```
$ docker network inspect bridge

[
```

```
    {
        "Name": "bridge",
        "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.1/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "9001"
        },
        "Labels": {}
    }
]
```

运行以下两个命令启动两个 `busybox` 容器，两个容器都连接到默认的 `bridge` 网络。

```
$ docker run -itd --name=container1 busybox

3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c

$ docker run -itd --name=container2 busybox

94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c
```

启动两个容器后再检查 `bridge` 网络。 这两个 `busybox` 容器都连接到网络。 可看到类似如下的结果：

```
$ docker network inspect bridge

{[
    {
        "Name": "bridge",
        "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
```

```
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.1/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {
            "3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c": {
                "EndpointID": "647c12443e91faf0fd508b6edfe59c30b642abb60dfab890b4bd
ccee38750bc1",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c": {
                "EndpointID": "b047d090f446ac49747d3c37d63e4307be745876db7f0ceef7b3
11cbba615f48",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "9001"
        },
        "Labels": {}
    }
]
```

连接到默认 `bridge` 网络的容器可通过IP地址进行通信。 **Docker不支持在默认网桥上自动发现服务。如果您希望容器能够通过容器名称来解析IP地址，那么可使用用户自定义网络** 。您可以使用遗留的 `docker run --link` 选项将两个容器连接在一起，但在大多数情况下不推荐使用。

您可以 `attach` 到正在运行的容器，查看容器内部的IP是什么。

```
$ docker attach container1

root@3386a527aa08:/# ifconfig

eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
```

```
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1296 (1.2 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

从容器内部，使用 `ping` 命令测试与其他容器的网络连接。

```
root@3386a527aa08:/# ping -w3 172.17.0.3

PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.096 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.074 ms

--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.074/0.083/0.096 ms
```

使用 `cat` 命令查看容器上的 `/etc/hosts` 文件。 该命令显示容器识别的主机名和IP地址。

```
root@3386a527aa08:/# cat /etc/hosts

172.17.0.2    3386a527aa08
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
```

要从 `container1` 容器离开，并保持容器的运行，请依次使用**CTRL-p CTRL-q** 。 如果你愿意，也可attch到 `container2` ，并重复上面的命令。

默认的 `docker0` 桥接网络支持使用端口映射和 `docker run --link` ，以便在 `docker0` 网络中的容器之间进行通信。 不推荐这种方法。 如果可以，请使用用户定义的桥接网络。

用户自定义的网络

建议使用用户自定义网桥来控制哪些容器可以相互通信，这样也可启用自动DNS去解析容器名称到IP地址。 Docker提供了创建这些网络的默认**网络驱动程序**。您可以创建一个新的**桥接网络**， **覆盖网络**或**MACVLAN网络** 。 您还可以创建一个**网络插件**或**远程网络**进行完整的自定义和控制。

您可以根据需要创建任意数量的网络，并且可在任意时间将容器连接到这些网络中的零个或多个。 此外，您可以将运行着的容器连接或断开网络，而无需重启容器。当容器连接到多个网络时，**其外部连接通过第一个非内部网络以词汇顺序提供**。

接下来的几节将详细介绍Docker的内置网络驱动程序。

网桥网络

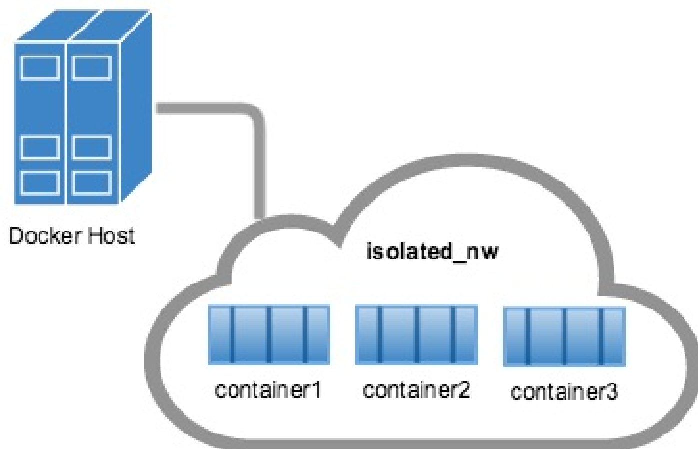
`bridge` 网络是Docker中最常见的网络类型。 桥接网络类似于默认的 `bridge` 网络，但添加一些新功能并删除一些旧的能力。 以下示例创建了桥接网络，并对这些网络上的容器执行一些实验。

```
$ docker network create --driver bridge isolated_nw

1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b

$ docker network inspect isolated_nw

[
    {
        "Name": "isolated_nw",
        "Id": "1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]

$ docker network ls

NETWORK ID          NAME                DRIVER
9f904ee27bf5        none                null
cf03ee007fb4        host                host
7fca4eb8c647        bridge              bridge
c5ee82f76de3        isolated_nw         bridge
```

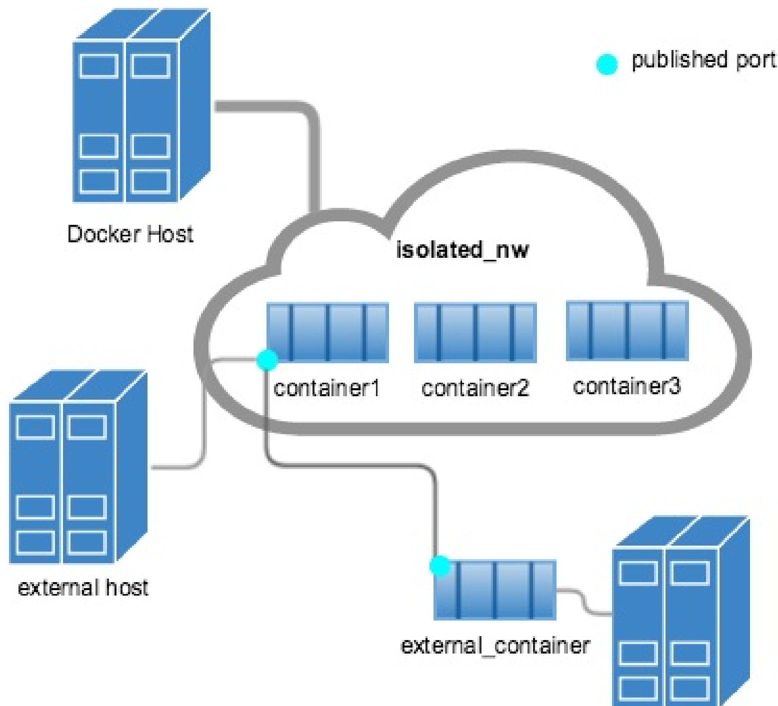
创建网络后，您可以使用 `docker run --network=<NETWORK>` 选项启动容器。

```
$ docker run --network=isolated_nw -itd --name=container3 busybox

8c1a0a5be480921d669a073393ade66a3fc49933f08bcc5515b37b8144f6d47c

$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {}
            ]
        },
        "Containers": {
            "8c1a0a5be480921d669a073393ade66a3fc49933f08bcc5515b37b8144f6d47c": {
                "EndpointID": "93b2db4a9b9a997beb912d28bcfc117f7b0eb924ff91d48cfa25
1d473e6a9b08",
                "MacAddress": "02:42:ac:15:00:02",
                "IPv4Address": "172.21.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

您启动到此网络的容器必须驻留在同一个Docker主机上。网络中的每个容器可以立即与其他容器通信。 **虽然网络本身将容器与外部网络隔离开来。**



在用户定义的桥接网络中，不支持链接（link）。 您可以在此网络中的容器上暴露和发布容器端口 。如果您希望使一部分 `bridge` 网络可用于外部网络，这将非常有用。



如果您希望在单个主机上运行相对较小的网络，桥接网络将非常有用。 但是，您可以通过创建 `overlay` 网络来创建更大的网络。

`docker_gwbridge` 网络

`docker_gwbridge` 是由Docker在两种不同情况下自动创建的本地桥接网络：

- 当您初始化或加入swarm时，Docker会创建 `docker_gwbridge` 网络，并将其用于不同主机上swarm节点之间的通信。
- 当容器网络不能提供外部连接时，除了容器的其他网络之外，Docker将容器连接到 `docker_gwbridge` 网络，以便容器可以连接到外部网络或其他swarm节点。

如果您需要自定义配置，您可以提前创建 `docker_gwbridge` 网络，否则Docker会根据需要创建它。以下示例使用一些自定义选项创建 `docker_gwbridge` 网络。

```
$ docker network create --subnet 172.30.0.0/16 \
                        --opt com.docker.network.bridge.name=docker_gwbridge \
            --opt com.docker.network.bridge.enable_icc=false \
            docker_gwbridge
```

当您使用 `overlay` 网络时， `docker_gwbridge` 网络始终存在。

swarm模式下的覆盖网络

当Docker在swarm模式下运行时，您可以在管理节点上创建覆盖网络，而无需外部key-value存储。swarm使覆盖网络仅可用于需要服务的swarm节点。 当您创建使用覆盖网络的服务时，管理节点会自动将覆盖网络扩展到运行服务任务的节点。

要了解有关在swarm模式下运行Docker Engine的更多信息，请参阅Swarm模式概述 。

下面的示例显示了如何创建网络并将其用于来自swarm管理节点的服务：

```
$ docker network create \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  my-multi-host-network

400g6bwzd68jizzdx5pgyoe95

$ docker service create --replicas 2 --network my-multi-host-network --name my-web
nginx

716thylsndqma81j6kkkb5aus
```

只有swarm服务可以连接到覆盖网络，而不是独立的容器。 有关群集的更多信息，请参阅Docker swarm模式覆盖网络安全模型 以及 将服务附加到覆盖网络 。

非swarm模式下的覆盖网络

如果您不是在swarm模式下使用Docker Engine，那么 `overlay` 网络需要有效的key-value存储。 支持的key-value存储包括Consul，Etcd和ZooKeeper（分布式存储）。 在以这种方式创建网络之前，您必须安装并配置您所选择的key-value存储服务。 网络中的Docker宿主机、服务必须能够进行通信。

> **注意**： 以swarm模式运行的Docker Engine与使用外部key-value存储的网络不兼容。

对于大多数Docker用户，不推荐这种使用覆盖网络的方法。它可以与独立的swarm一起使用，可能对在Docker顶部构建解决方案的系统开发人员有用。 将来可能会被弃用。 如果您认为可能需要以这种方式使用覆盖网络，请参阅本指南 。

自定义网络插件

如果任何上述网络机制无法满足您的需求，您可以使用Docker的插件基础架构编写自己的网络驱动插件。 该插件将在运行Docker deamon的主机上作为单独的进程运行。 使用网络插件是一个高级主题。

网络插件遵循与其他插件相同的限制和安装规则。 所有插件都使用插件API，并具有包含了安装，启动，停止和激活的生命周期。

创建并安装自定义网络驱动后，您可以使用 `--driver` 标志创建一个使用该驱动的网络。

```
$ docker network create --driver weave mynet
```

您可以检查该网络、让容器连接或断开该网络，删除该网络。 特定的插件为特定的需求而生。 检查插件文档的具体信息。 有关编写插件的更多信息，请参阅扩展Docker 以及 编写网络驱动程序插件 。

内嵌DNS服务器

Docker daemon运行一个嵌入式的DNS服务器，从而为连接到同一用户自定义网络的容器之间提供DNS解析——这样，这些容器即可将容器名称解析为IP地址。 如果内嵌DNS服务器无法解析请求，它将被转发到为容器配置的任意外部DNS服务器。 **为了方便，当容器创建时，只有 `127.0.0.11` 可访问的内嵌DNS服务器会列在容器的 `resolv.conf` 文件中。** 有关在用户自定义网络的内嵌DNS服务器的更多信息，请参阅用户定义网络中的内嵌DNS服务器

暴露和发布端口

在Docker网络中，有两种不同的机制可以直接涉及网络端口：暴露端口和发布端口。 这适用于默认网桥和用户定义的网桥。

- 您使用 `Dockerfile` 中的 `EXPOSE` 关键字或 `docker run` 命令中的 `--expose` 标志来暴露端口。 **暴露端口是记录使用哪些端口，但实际上并不映射或打开任何端口的一种方式。** 暴露端口是可选的。
- 您可以使用 `Dockerfile` 中的 `PUBLISH` 关键字或 `docker run` 命令中的 `--publish` 标志来发布端口。 这告诉Docker在容器的网络接口上打开哪些端口。当端口发布时，它将映射到宿主机上可用的高阶端口（高于 `30000` ），除非您在运行时指定要映射到宿主机的哪个端口。 您不能在Dockerfile中指定要映射的端口，因为无法保证端口在运行image的宿主机上可用。

  此示例将容器中的端口80发布到宿主机上的随机高阶端口（在这种情况下为 `32768` ）。 `-d` 标志使容器在后台运行，因此您可以发出 `docker ps` 命令。

  ```
  $ docker run -it -d -p 80 nginx

  $ docker ps

  64879472feea        nginx               "nginx -g 'daemon ..."   43 hours ago
        Up About a minute   443/tcp, 0.0.0.0:32768->80/tcp   blissful_mclean
  ```

  下一个示例指定80端口应映射到宿主机上的8080端口。 如果端口8080不可用，将失败。

  ```
  $ docker run -it -d -p 8080:80 nginx

  $ docker ps

  b9788c7adca3        nginx               "nginx -g 'daemon ..."   43 hours ago
        Up 3 seconds        80/tcp, 443/tcp, 0.0.0.0:8080->80/tcp   goofy_brahmag
  upta
  ```

容器与代理服务器

如果您的容器需要使用HTTP、HTTPS或者FTP代理，你可以使用如下两种方式进行配置：

- 对于Docker 17.07或更高版本，你可以配置Docker客户端从而将代理信息自动传递给容器。
- 对于Docker 17.06或更低版本，你必须在容器内设置环境变量。你可以在构建镜像（这样不太好移植）或启动容器时执行此操作。

配置Docker客户端

> **仅限Edge版本**： 此选项仅适用于Docker CE Edge版本。 请参阅Docker CE Edge 。

1. 在Docker客户端上，在启动容器所使用的用户的主目录中创建或编辑 `~/.config.json` 文件。在其中添加如类似下所示的JSON，如果需要，使用 `httpsproxy` 或 `ftpproxy` 替换代理类型，然后替换代理服务器的地址和端口。 您可以同时配置多个代理服务器。

   您可以通过将 `noProxy` 键设置为一个或多个逗号分隔的IP地址或主机名来选择将指定主机或指定范围排除使用代理服务器。 支持使用 `*` 字符作为通配符，如此示例所示。

   ```
   {
     "proxies":
     {
       "httpProxy": "http://127.0.0.1:3001",
       "noProxy": "*.test.example.com,.example2.com"
     }
   }
   ```

   保存文件。

2. 当您创建或启动新容器器时，环境变量将在容器内自动设置。

手动设置环境变量

在构建映像时，或在创建或运行容器时使用 `--env` 标志，可将下表中的一个或多个变量设置为适当的值。 这种方法使镜像不太可移植，因此如果您使用Docker 17.07或更高版本，则应该配置Docker客户端。

变量	Dockerfile示例	`docker run` 示例
`HTTP_PROXY`	`ENV HTTP_PROXY "http://127.0.0.1:3001"`	`--env HTTP_PROXY "http://127.0.0.1:3001"`
`HTTPS_PROXY`	`ENV HTTPS_PROXY "https://127.0.0.1:3001"`	`--env HTTPS_PROXY "https://127.0.0.1:3001"`
`FTP_PROXY`	`ENV FTP_PROXY "ftp://127.0.0.1:3001"`	`--env FTP_PROXY "ftp://127.0.0.1:3001"`
`NO_PROXY`	`ENV NO_PROXY "*.test.example.com,.example2.com"	`-env NO_PROXY“*.test.example.com, .example2.com”`

链接

在Docker包含“用户自定义网络”功能之前，您可以使用Docker `--link` 功能来允许容器将另一个容器的名称解析为IP地址，还可以访问你所链接的容器的环境变量。 如果可以，您应该避免使用 `--link` 标志。

当您创建连接时，当您使用默认 `bridge` 或用户自定义网桥时，它们的行为会有所不同。 有关详细信息，请参阅默认 `bridge` 链接功能的遗留链接以及在用户自定义网络中链接容器的链接容器 。

Docker和iptables

Linux主机使用内核模块 `iptables` 来管理对网络设备的访问，包括路由，端口转发，网络地址转换（NAT）等问题。 Docker会在启动或停止发布端口的容器、创建或修改网络、attach到容器或其他与网络相关的操作时修改 `iptables` 规则。

对 `iptables` 全面讨论超出了本主题的范围。 要查看哪个 `iptables` 规则在任何时间生效，可以使用 `iptables -L` 。 如存在多个表，例如 `nat` ， `prerouting` 或 `postrouting` ，您可以使用诸如 `iptables -t nat -L` 类的命令列出特定的表。 有关 `iptables` 的完整文档，请参阅netfilter / iptables 。

通常， `iptables` 规则由初始化脚本或守护进程创建，例如 `firewalld` 。 规则在系统重新启动时不会持久存在，因此脚本或程序必须在系统引导时执行，通常在运行级别3或直接在网络初始化之后运行。请参阅您的Linux发行版的网络相关的文档，了解如何使 `iptables` 规则持续存在。

Docker动态管理Docker daemon、容器，服务和网络的 `iptables` 规则。 在Docker 17.06及更高版本中，您可以向名为 `DOCKER-USER` 的新表添加规则，这些规则会在Docker自动创建任何规则之前加载。 如果您需要在Docker运行之前预先设置需要使用的 `iptables` 规则，这将非常有用。

相关信息

- Work with network commands
- Get started with multi-host networking
- Managing Data in Containers
- Docker Machine overview
- Docker Swarm overview
- Investigate the LibNetwork project

原文

https://docs.docker.com/engine/userguide/networking/

network命令

本文提供可用于与Docker网络及与网络中容器进行交互的network子命令的示例。这些命令可通过Docker Engine CLI获得。 这些命令是：

- `docker network create`
- `docker network connect`
- `docker network ls`
- `docker network rm`
- `docker network disconnect`
- `docker network inspect`

虽然不是必需的，但在尝试本节中的示例之前，先阅读 了解Docker网络 更佳。 示例使用默认 `bridge` 网络以便您可以立即尝试。要实验 `overlay` 网络，请参阅 多主机网络入门指南 。

创建网络

Docker Engine在安装时自动创建 `bridge` 网络。 该网络对应于Engine传统依赖的 `docker0` 网桥。除该网络外，也可创建自己的 `bridge` 或 `overlay` 网络。

`bridge` 网络驻留在运行Docker Engine实例的单个主机上。 `overlay` 网络可跨越运行Docker Engine的多个主机。 如果您运行 `docker network create` 并仅提供网络名称，它将为您创建一个桥接网络。

```
$ docker network create simple-network

69568e6336d8c96bbf57869030919f7c69524f71183b44d80948bd3927c87f6a

$ docker network inspect simple-network
[
    {
        "Name": "simple-network",
        "Id": "69568e6336d8c96bbf57869030919f7c69524f71183b44d80948bd3927c87f6a",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.22.0.0/16",
                    "Gateway": "172.22.0.1"
                }
            ]
        },
        "Containers": {},
        "Options": {},
```

```
        "Labels": {}
    }
]
```

与 `bridge` 网络不同， `overlay` 网络需要一些预制条件才能创建——

- 访问key-value存储。 引擎支持Consul，Etcd和ZooKeeper（分布式存储）key-value存储。
- 与key-value存储连接的主机集群。
- 在swarm中的每个主机上正确配置的 `Docker daemon` 。

支持 `overlay` 网络的 `dockerd` 选项有：

- `--cluster-store`
- `--cluster-store-opt`
- `--cluster-advertise`

在创建网络时，Docker引擎默认会为网络创建一个不重叠的子网。 您可以覆盖此默认值，并使用 `--subnet` 选项直接指定子网。 对于 `bridge` 网络，只可指定一个子网。 `overlay` 网络支持多个子网。

> **注意**： 强烈建议在创建网络时使用 `--subnet` 选项。 如果未指定 `--subnet` 则Docker daemon会自动为网络选择并分配子网，这可能会导致与您基础结构中的另一个子网（该子网不受 `--subnet` 管理）重叠。 当容器连接到该网络时，这种重叠可能导致连接问题或故障。

除 `--subnet` 选项以外，您还可以指定 `--gateway` ， `--ip-range` `--gateway` `--ip-range` 和 `--aux-address` 选项。

```
$ docker network create -d overlay \
  --subnet=192.168.0.0/16 \
  --subnet=192.170.0.0/16 \
  --gateway=192.168.0.100 \
  --gateway=192.170.0.100 \
  --ip-range=192.168.1.0/24 \
  --aux-address="my-router=192.168.1.5" --aux-address="my-switch=192.168.1.6" \
  --aux-address="my-printer=192.170.1.5" --aux-address="my-nas=192.170.1.6" \
  my-multihost-network
```

确保您的子网不重叠。 如果重叠，那么网络将会创建失败，Docker Engine返回错误。

创建自定义网络时，您可以向驱动传递其他选项。 `bridge` 驱动程序接受以下选项：

Option	Equivalent	Description
`com.docker.network.bridge.name`	-	创建Linux网桥时要使用的网桥名称
`com.docker.network.bridge.enable_ip_masquerade`	`--ip-masq`	启用IP伪装
`com.docker.network.bridge.enable_icc`	`--icc`	启用或禁用跨容器连接

`com.docker.network.bridge.host_binding_ipv4`	`--ip`	绑定容器端口时的默认IP
`com.docker.network.driver.mtu`	`--mtu`	设置容器网络MTU

`overlay` 驱动也支持 `com.docker.network.driver.mtu` 选项。

以下参数可以传递给任何网络驱动的 `docker network create` 。

Argument	Equivalent	Description
`--internal`	-	限制对网络的外部访问
`--ipv6`	`--ipv6`	启用IPv6网络

以下示例使用 `-o` 选项，在绑定端口时绑定到指定的IP地址，然后使用 `docker network inspect` 来检查网络，最后将新容器attach到新网络。

```
$ docker network create -o "com.docker.network.bridge.host_binding_ipv4"="172.23.0.
1" my-network

b1a086897963e6a2e7fc6868962e55e746bee8ad0c97b54a5831054b5f62672a

$ docker network inspect my-network

[
    {
        "Name": "my-network",
        "Id": "b1a086897963e6a2e7fc6868962e55e746bee8ad0c97b54a5831054b5f62672a",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.23.0.0/16",
                    "Gateway": "172.23.0.1"
                }
            ]
        },
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.host_binding_ipv4": "172.23.0.1"
        },
        "Labels": {}
    }
]

$ docker run -d -P --name redis --network my-network redis

bafb0c808c53104b2c90346f284bda33a69beadcab4fc83ab8f2c5a4410cd129
```

```
$ docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED
  STATUS              PORTS                        NAMES
bafb0c808c53        redis               "/entrypoint.sh redis"   4 seconds ago
  Up 3 seconds        172.23.0.1:32770->6379/tcp   redis
```

连接容器

您可以将一个现有容器连接到一个或多个网络。 容器可连接到使用不同网络驱动的网络。 一旦连接，容器即可使用另一个容器的IP地址或名称进行通信。

对于支持多主机连接的 `overlay` 网络或自定义插件，不同主机上的容器，只要连接到同一multi-host network多主机网络，也可以这种方式进行通信。

此示例使用六个容器，并指示您根据需要创建它们。

基本容器网络示例

1. 首先，创建并运行两个容器， `container1` 和 `container2` ：

```
$ docker run -itd --name=container1 busybox

18c062ef45ac0c026ee48a83afa39d25635ee5f02b58de4abc8f467bcaa28731

$ docker run -itd --name=container2 busybox

498eaaaf328e1018042c04b2de04036fc04719a6e39a097a4f4866043a2c2152
```

2. 创建一个隔离的 `bridge` 网络进行测试。

```
$ docker network create -d bridge --subnet 172.25.0.0/16 isolated_nw

06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8
```

3. 将 `container2` 连接到网络，然后 `inspect` 网络以验证连接：

```
$ docker network connect isolated_nw container2

$ docker network inspect isolated_nw

[
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8
",
```

```
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.25.0.0/16",
                    "Gateway": "172.25.0.1/16"
                }
            ]
        },
        "Containers": {
            "90e1f3ec71caf82ae776a827e0712a68a110a3f175954e5bd4222fd142ac9428":
 {
                "Name": "container2",
                "EndpointID": "11cedac1810e864d6b1589d92da12af66203879ab89f4ccd
8c8fdaa9b1c48b1d",
                "MacAddress": "02:42:ac:19:00:02",
                "IPv4Address": "172.25.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

请注意， `container2` 自动分配了一个IP地址。 因为在创建网络时指定了 `--subnet` 选项，所以IP地址会从该子网选择。

作为提醒， `container1` 仅连接到默认 `bridge` 。

4. 启动第三个容器，但这次使用 `--ip` 标志分配一个IP地址，并使用 `docker run` 命令的 `--network` 选项将其连接到 `--isolated_nw` 网络：

```
$ docker run --network=isolated_nw --ip=172.25.3.3 -itd --name=container3 busyb
ox

467a7863c3f0277ef8e661b38427737f28099b61fa55622d6c30fb288d88c551
```

只要您为容器指定的IP地址是如上子网的一部分，那就可使用 `--ip` 或 `--ip6` 标志将IPv4或IPv6地址分配给容器，将其连接到以上网络。 当您在使用用户自定义的网络时以这种方式指定IP地址时，配置将作为容器配置的一部分进行保留，并在容器重新加载时进行应用。 使用非用户自定义网络时，分配的IP地址将被保留，因为不保证Docker daemon重启时容器的子网不会改变，除非您使用用户定义的网络。【这一段官方文档是不是有问题？？？】

5. 检查 `container3` 所使用的网络资源。 简洁起见，截断以下输出。

```
$ docker inspect --format=''  container3
```

```
{"isolated_nw":
  {"IPAMConfig":
    {
      "IPv4Address":"172.25.3.3"},
      "NetworkID":"1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509
679b",
      "EndpointID":"dffc7ec2915af58cc827d995e6ebdc897342be0420123277103c40ae355
79103",
      "Gateway":"172.25.0.1",
      "IPAddress":"172.25.3.3",
      "IPPrefixLen":16,
      "IPv6Gateway":"",
      "GlobalIPv6Address":"",
      "GlobalIPv6PrefixLen":0,
      "MacAddress":"02:42:ac:19:03:03"}
    }
  }
}
```

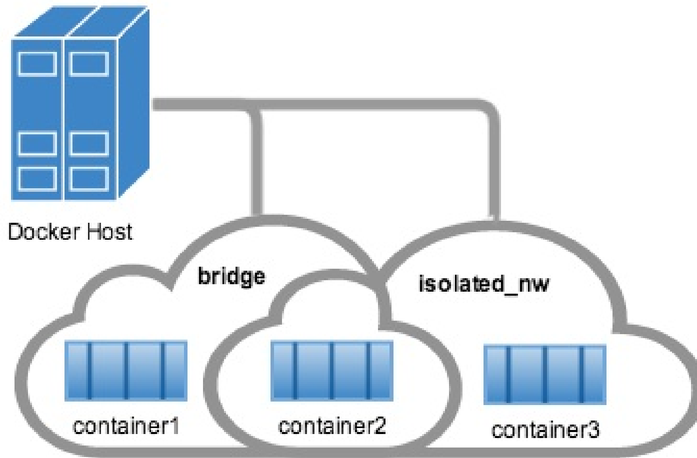
因为在启动时将 `container3` 连接到 `isolated_nw` ，所以它根本没有连接到默认的 `bridge` 网络。

6. 检查 `container2` 所使用的网络。 如果你安装了Python，你可以打印输出格式化。

```
$ docker inspect --format=''  container2 | python -m json.tool

{
    "bridge": {
        "NetworkID":"7ea29fc1412292a2d7bba362f9253545fecdfa8ce9a6e37dd10ba8bee7
129812",
        "EndpointID": "0099f9efb5a3727f6a554f176b1e96fca34cae773da68b3b6a26d046
c12cb365",
        "Gateway": "172.17.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAMConfig": null,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:03"
    },
    "isolated_nw": {
        "NetworkID":"1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d5
09679b",
        "EndpointID": "11cedac1810e864d6b1589d92da12af66203879ab89f4ccd8c8fdaa9
b1c48b1d",
        "Gateway": "172.25.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAMConfig": null,
```

```
            "IPAddress": "172.25.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:19:00:02"
        }
    }
```

请注意， `container2` 属于两个网络。 当您启动它时，它加入了默认 `bridge` 网络，并在步骤3中将其连接到 `isolated_nw` 。



eth0 Link encap:Ethernet HWaddr 02:42:AC:11:00:03

eth1 Link encap:Ethernet HWaddr 02:42:AC:15:00:02

7. 使用 `docker attach` 命令连接到正在运行的 `container2` 并检查它的网络堆栈：

```
$ docker attach container2
```

使用 `ifconfig` 命令检查容器的网络堆栈。 您应该看到两个以太网卡，一个用于默认 `bridge` ，另一个用于 `isolated_nw` 网络。

```
$ sudo ifconfig -a

eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:15:00:02
          inet addr:172.25.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe19:2/64 Scope:Link
```

```
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

8. Docker内嵌DNS服务器可使用容器名称解析连接到给定网络的容器。 这意味着网络内的容器可以通过容器名称ping在同一网络中的另一个容器。 例如，从 `container2` 可以按名称ping `container3` 。

```
/ # ping -w 4 container3
PING container3 (172.25.3.3): 56 data bytes
64 bytes from 172.25.3.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.3.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.3.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.3.3: seq=3 ttl=64 time=0.097 ms

--- container3 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

此功能不适用于默认 `bridge` 网络。 `container1` 和 `container2` 都连接到默认的 `bridge` 网络，但是并不能使用容器名称从 `container2` ping `container1` 。

```
/ # ping -w 4 container1
ping: bad address 'container1'
```

但依然可直接ping IP地址：

```
/ # ping -w 4 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.075 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.072 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.101 ms

--- 172.17.0.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.072/0.085/0.101 ms
```

离开 `container2` 容器，并使用 `CTRL-p CTRL-q` 保持容器运行。

9. 当前， `container2` 连接到默认 `bridge` 网络和 `isolated_nw` 网络，因此， `container2` 可与 `container1` 以及 `container3` 进行通信。 但是， `container3` 和 `container1` 没有任何共同的网络，所以它们不能通信。 要验证这一点，请附加到 `container3` 并尝试通过IP地址ping `container1` 。

```
$ docker attach container3

$ ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
^C

--- 172.17.0.2 ping statistics ---
10 packets transmitted, 0 packets received, 100% packet loss
```

离开 `container3` 容器，并使用 `CTRL-p CTRL-q` 保持容器运行。

> 即使容器未运行，也可以将容器连接到网络。 但是， `docker network inspect` 仅显示运行容器的信息。

链接容器而不使用用户定义的网络

完成基本容器网络示例中的步骤后， `container2` 可以自动解析 `container3` 的名称，因为两个容器都连接到 `isolated_nw` 网络。 但是，连接到默认 `bridge` 的容器无法解析彼此的容器名称。 如果您需要容器能够通过 `bridge` 网络进行通信，则需要使用遗留的连接功能。 这是唯一的建议使用 `--link` 的情况。 您应该强烈地考虑使用用户定义的网络。

使用遗留的 `link` 标志为可为默认的 `bridge` 网络添加以下功能进行通信：

- 将容器名称解析为IP地址的能力
- 使用 `--link=CONTAINER-NAME:ALIAS` 定义一个网络别名去连接容器的能力
- 安全的容器连接（通过 `--icc=false` 隔离）
- 环境变量注入

需要重申的是，当您使用用户自定义网络时，默认情况下提供所有这些功能，无需额外的配置。 **此外，您可以动态attach到多个网络，也可动态从多个网络中离开。**

- 使用DNS进行自动名称解析
- 支持 `--link` 选项为链接的容器提供名称别名
- 网络中容器的自动安全隔离环境
- 环境变量注入

以下示例简要介绍如何使用 `--link` 。

1. 继续上面的例子，创建一个新的容器 `container4` ，并将其连接到网络 `isolated_nw` 。 另外，使用 `--link` 标志链接到容器 `container5` （不存在！）！

```
$ docker run --network=isolated_nw -itd --name=container4 --link container5:c5
busybox

01b5df970834b77a9eadbaff39051f237957bd35c4c56f11193e0594cfd5117c
```

这有点棘手，因为 `container5` 还不存在。 当 `container5` 被创建时， `container4` 将能够将名称 `c5` 解析为 `container5` 的IP地址。

> **注意**： 使用遗留的link功能创建的容器之间的任何链接本质上都是静态的，并且通过别名强制绑定容器。 它无法容忍链接的容器重新启动。 用户自定义网络中的新链接功能支持容器之间的动态链接，并且允许链接容器中的重新启动和IP地址更改。

由于您尚未创建容器 `container5` 尝试ping它将导致错误。 attach到 `container4` 并尝试ping任何 `container5` 或 `c5` ：

```
$ docker attach container4

$ ping container5

ping: bad address 'container5'

$ ping c5

ping: bad address 'c5'
```

从 `container4` 离开，并使用 `CTRL-p CTRL-q` 使其保持运行。

2. 创建一个容器，名为 `container5` ，并使用别名 `c4` 将其链接到 `container4` 。

```
$ docker run --network=isolated_nw -itd --name=container5 --link container4:c4
busybox

72eccf2208336f31e9e33ba327734125af00d1e1d2657878e2ee8154fbb23c7a
```

现在attach到 `container4` ，尝试ping `c5` 和 `container5` 。

```
$ docker attach container4

/ # ping -w 4 c5
PING c5 (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

--- c5 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
```

```
round-trip min/avg/max = 0.070/0.081/0.097 ms

/ # ping -w 4 container5
PING container5 (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

--- container5 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

从 `container4` 分离，并使用 `CTRL-p CTRL-q` 使其保持运行。

3. 最后，附加到 `container5` ，验证你可以ping `container4` 。

```
$ docker attach container5

/ # ping -w 4 c4
PING c4 (172.25.0.4): 56 data bytes
64 bytes from 172.25.0.4: seq=0 ttl=64 time=0.065 ms
64 bytes from 172.25.0.4: seq=1 ttl=64 time=0.070 ms
64 bytes from 172.25.0.4: seq=2 ttl=64 time=0.067 ms
64 bytes from 172.25.0.4: seq=3 ttl=64 time=0.082 ms

--- c4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.065/0.070/0.082 ms

/ # ping -w 4 container4
PING container4 (172.25.0.4): 56 data bytes
64 bytes from 172.25.0.4: seq=0 ttl=64 time=0.065 ms
64 bytes from 172.25.0.4: seq=1 ttl=64 time=0.070 ms
64 bytes from 172.25.0.4: seq=2 ttl=64 time=0.067 ms
64 bytes from 172.25.0.4: seq=3 ttl=64 time=0.082 ms

--- container4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.065/0.070/0.082 ms
```

从 `container5` 离开，并使用 `CTRL-p CTRL-q` 使其保持运行。

网络范围的别名示例

链接容器时，无论是使用遗留的 `link` 方法还是使用用户自定义网络，您指定的任何别名只对指定的容器有意义，并且不能在默认 `bridge` 上的其他容器上运行。

另外，如果容器属于多个网络，则给定的链接别名与给定的网络范围一致。 因此，容器可以链接到不同网络中的不同别名，并且别名将不适用于不在同一网络上的容器。

以下示例说明了这些要点。

1. 创建另一个名为 `local_alias` 网络：

   ```
   $ docker network create -d bridge --subnet 172.26.0.0/24 local_alias
   76b7dc932e037589e6553f59f76008e5b76fa069638cd39776b890607f567aaa
   ```

2. 接下来，使用别名 `foo` 和 `bar` 将 `container4` 和 `container5` 连接到新的网络 `local_alias` ：

   ```
   $ docker network connect --link container5:foo local_alias container4
   $ docker network connect --link container4:bar local_alias container5
   ```

3. attach到 `container4` 并尝试使用别名 `foo` ping `container4` （是的，同一个），然后尝试使用别名 `c5` ping容器 `container5` ：

   ```
    $ docker attach container4

    / # ping -w 4 foo
    PING foo (172.26.0.3): 56 data bytes
    64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.070 ms
    64 bytes from 172.26.0.3: seq=1 ttl=64 time=0.080 ms
    64 bytes from 172.26.0.3: seq=2 ttl=64 time=0.080 ms
    64 bytes from 172.26.0.3: seq=3 ttl=64 time=0.097 ms

    --- foo ping statistics ---
    4 packets transmitted, 4 packets received, 0% packet loss
    round-trip min/avg/max = 0.070/0.081/0.097 ms

    / # ping -w 4 c5
    PING c5 (172.25.0.5): 56 data bytes
    64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
    64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
    64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
    64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

    --- c5 ping statistics ---
    4 packets transmitted, 4 packets received, 0% packet loss
    round-trip min/avg/max = 0.070/0.081/0.097 ms
   ```

   两个ping都成功了，但子网不同，这意味着网络不同。

   离开 `container4` ，并使用 `CTRL-p CTRL-q` 使其保持运行。

4. 从 `isolated_nw` 网络断开 `container5` 。附加到 `container4` 并尝试ping `c5` 和 `foo` 。

```
$ docker network disconnect isolated_nw container5

$ docker attach container4

/ # ping -w 4 c5
ping: bad address 'c5'

/ # ping -w 4 foo
PING foo (172.26.0.3): 56 data bytes
64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=3 ttl=64 time=0.097 ms

--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

您不能再从 `container5` 收到 `isolated_nw` 网络上的 `container5` 。但是，您仍然可以使用别名 `foo` 到达 `container4` （从 `container4` ）。

离开 `container4` ，并使用 `CTRL-p CTRL-q` 使其保持运行。

`docker network` 限制

虽然 `docker network` 是控制您的容器使用的网络的推荐方法，但它确实有一些限制。

环境变量注入

环境变量注入是静态的，环境变量在容器启动后无法更改。 遗留的 `--link` 标志将所有环境变量共享到链接的容器，但 `docker network` 命令没有等效选项。 当您使用 `docker network` 将容器连接到网络时，不能在容器之间动态共享环境变量。

使用网络范围的别名

遗留的link提供传出名称解析，隔离在配置别名的容器内。 网络范围的别名不允许这种单向隔离，而是为网络的所有成员提供别名。

以下示例说明了此限制。

1. 在网络 `isolated_nw` 创建另一个容器 `container6` ，并给它网络别名 `app` 。

```
$ docker run --network=isolated_nw -itd --name=container6 --network-alias app b
usybox

8ebe6767c1e0361f27433090060b33200aac054a68476c3be87ef4005eb1df17
```

2. attach到 `container4` 。尝试通过名称（ `container6` ）和网络别名（ `app` ）ping容器。请注意，IP地址是一样的。

```
$ docker attach container4

/ # ping -w 4 app
PING app (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- app ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

/ # ping -w 4 container6
PING container5 (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- container6 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

从 `container4` 离开，并使用 `CTRL-p CTRL-q` 使其保持运行。

3. 将 `container6` 连接到 `local_alias` 网络，并为其赋予网络范围的别名 `scoped-app` 。

```
$ docker network connect --alias scoped-app local_alias container6
```

现在 `container6` 在网络 `isolated_nw` 中的别名为 `app` ，在网络 `local_alias` 中别名为 `scoped-app` 。

4. 尝试从 `container4` （连接到这两个网络）和 `container5` （仅连接到 `isolated_nw` ）连接到这些别名。

```
$ docker attach container4

/ # ping -w 4 scoped-app
PING foo (172.26.0.5): 56 data bytes
64 bytes from 172.26.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.5: seq=3 ttl=64 time=0.097 ms
```

```
--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

离开 `container4` ，并使用 `CTRL-p CTRL-q` 使其保持运行。

```
$ docker attach container5

/ # ping -w 4 scoped-app
ping: bad address 'scoped-app'
```

离开 `container5` ，并使用 `CTRL-p CTRL-q` 使其保持运行。

这表明将别名仅在定义它的网络上生效，只有连接到该网络的容器才能访问该别名。

将多个容器解析为一个别名

多个容器可在同一网络内共享相同的网络范围别名。 这提供了一种DNS轮询（round-robbin）高可用性。 当使用诸如Nginx这样的软件时，这可能不可靠，Nginx通过IP地址来缓存客户端。

以下示例说明了如何设置和使用网络别名。

> **注意**： 使用网络别名进行DNS轮询高可用的用户应考虑使用swarm服务。 Swarm服务提供了开箱即用的、类似的负载均衡功能。 如果连接到任何节点，即使是不参与服务的节点。 Docker将请求发送到正在参与服务的随机节点，并管理所有的通信。

1. 在 `isolated_nw` 中启动 `container7` ，别名与 `container6` 相同，即 `app` 。

   ```
   $ docker run --network=isolated_nw -itd --name=container7 --network-alias app busybox

   3138c678c123b8799f4c7cc6a0cecc595acbdfa8bf81f621834103cd4f504554
   ```

   当多个容器共享相同的别名时，其中一个容器将解析为别名。 如果该容器不可用，则另一个具有别名的容器将被解析。 这提供了群集中的高可用性。

   > **注意**： **在IP地址解析时，所选择的容器是不完全可预测的。 因此，在下面的练习中，您可能会在一些步骤中获得不同的结果。** 如果步骤假定返回的结果是 `container6` 但是您收到 `container7` ，这就是为什么。

2. 从 `container4` 开始连续ping到 `app` 别名。

   ```
   $ docker attach container4

   $ ping app
   PING app (172.25.0.6): 56 data bytes
   64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
   64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
   64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
   ```

```
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms
...
```

返回的IP地址属于 `container6` 。

3. 在另一个终端，停止 `container6` 。

```
 $ docker stop container6
```

在连接到 `container4` 的终端，观察 `ping` 输出。当 `container6` 关闭时，它将暂停，因为 `ping` 命令在首次调用时查找IP，并且发现该IP不再可用。但是， `ping` 命令在默认情况下具有非常长的超时时间，因此不会发生错误。

4. 使用 `CTRL+C` 退出 `ping` 命令并再次运行。

```
$ ping app

PING app (172.25.0.7): 56 data bytes
64 bytes from 172.25.0.7: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.25.0.7: seq=1 ttl=64 time=0.075 ms
64 bytes from 172.25.0.7: seq=2 ttl=64 time=0.072 ms
64 bytes from 172.25.0.7: seq=3 ttl=64 time=0.101 ms
...
```

`app` 别名现在解析为 `container7` 的IP地址。

5. 最后一次测试，重新启动 `container6` 。

```
$ docker start container6
```

在连接到 `container4` 的终端，再次运行 `ping` 命令。现在可能会再次解决 `container6` 。如果您几次启动和停止 `ping` ，您将看到每个容器的响应。

```
$ docker attach container4

$ ping app
PING app (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms
...
```

用 `CTRL+C` 停止ping。从 `container4` 离开，并使用 `CTRL-p CTRL-q` 使其保持运行。

断开容器

您可以随时使用 `docker network disconnect` 命令断开容器与网络的连接。

1. 从 `isolated_nw` 网络断开 `container2` ，然后检查 `container2` 和 `isolated_nw` 网络。

```
$ docker network disconnect isolated_nw container2

$ docker inspect --format=''  container2 | python -m json.tool

{
    "bridge": {
        "NetworkID":"7ea29fc1412292a2d7bba362f9253545fecdfa8ce9a6e37dd10ba8bee7
129812",
        "EndpointID": "9e4575f7f61c0f9d69317b7a4b92eefc133347836dd83ef65deffa16
b9985dc0",
        "Gateway": "172.17.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:03"
    }
}
$ docker network inspect isolated_nw

[
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8
",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {
            "467a7863c3f0277ef8e661b38427737f28099b61fa55622d6c30fb288d88c551":
 {
                "Name": "container3",
                "EndpointID": "dffc7ec2915af58cc827d995e6ebdc897342be0420123277
103c40ae35579103",
                "MacAddress": "02:42:ac:19:03:03",
                "IPv4Address": "172.25.3.3/16",
                "IPv6Address": ""
```

```
            }
        },
        "Options": {}
    }
]
```

2. 当容器与网络断开连接时，它不能再与连接到该网络的其他容器进行通信，除非它与其他容器具有g共用他网络。 验证 `container2` 不能再到达 `isolated_nw` 上的 `container3` 。

```
$ docker attach container2

/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping container3
PING container3 (172.25.3.3): 56 data bytes
^C
--- container3 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

3. 验证 `container2` 是否仍具有与默认 `bridge` 完全连接。

```
/ # ping container1
PING container1 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.119 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.174 ms
^C
--- container1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.119/0.146/0.174 ms
/ #
```

4. 移除 `container4` ， `container5` ， `container6` 和 `container7` 。

   ```
   $ docker stop container4 container5 container6 container7

   $ docker rm container4 container5 container6 container7
   ```

处理过时的网络端点

在某些情况下，例如在多主机网络中以非优雅的方式重新启动Docker daemon，Docker daemon将无法清除过时的连接端点。 如果新的容器连接到具有与过期端点相同的名称的网络，则此类过时的端点可能会导致错误：

```
ERROR: Cannot start container bc0b19c089978f7845633027aa3435624ca3d12dd4f4f764b61ea
c4c0610f32e: container already connected to network multihost
```

要清理这些过时的端点，可移除容器并强制将其与网络断开（ `docker network disconnect -f` ）。 这样，您就可将容器成功连接到网络。

```
$ docker run -d --name redis_db --network multihost redis

ERROR: Cannot start container bc0b19c089978f7845633027aa3435624ca3d12dd4f4f764b61ea
c4c0610f32e: container already connected to network multihost

$ docker rm -f redis_db

$ docker network disconnect -f multihost redis_db

$ docker run -d --name redis_db --network multihost redis

7d986da974aeea5e9f7aca7e510bdb216d58682faa83a9040c2f2adc0544795a
```

删除网络

当网络中的所有容器都已停止或断开连接时，您可以删除网络。 如果网络连接了端点，则会发生错误。

1. 断开 `container3` 与 `isolated_nw` 连接。

   ```
   $ docker network disconnect isolated_nw container3
   ```

1. 检查 `isolated_nw` 以验证没有其他端点连接到它。

   ```
   $ docker network inspect isolated_nw

   [
   ```

```
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8
",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {}
    }
]
```

2. 删除 `isolated_nw` 网络。

```
$ docker network rm isolated_nw
```

3. 列出所有网络以验证 `isolated_nw` 不再存在：

```
$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
4bb8c9bf4292        bridge              bridge              local
43575911a2bd        host                host                local
76b7dc932e03        local_alias         bridge              local
b1a086897963        my-network          bridge              local
3eb020e70bfd        none                null                local
69568e6336d8        simple-network      bridge              local
```

相关信息

- network create
- network inspect
- network connect
- network disconnect
- network ls
- network rm

原文

https://docs.docker.com/engine/userguide/networking/work-with-networks/

默认bridge网络中配置DNS

本节描述如何在Docker默认网桥中配置容器DNS。 当您安装Docker时，就会自动创建一个名为 `bridge` 的桥接网络。

> **注意**： Docker网络功能 允许您创建除默认网桥之外的用户自定义网络。 有关用户自定义网络中DNS配置的更多信息，请参阅Docker嵌入式DNS 部分。

Docker如何为每个容器提供主机名和DNS配置，而无需在构建自定义Docker镜像时在内部写入主机名？它的诀窍是利用可以写入新信息的虚拟文件，在容器内覆盖三个关键的 `/etc` 文件。 你可以通过在一个容器中运行 `mount` 来看到这一点：

```
root@f38c87f2a42d:/# mount

...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/resolv.conf type ext4 ...
...
```

这样一来，Docker可以让宿主机在稍后通过DHCP接收到新的配置后，使所有容器中的 `resolv.conf` 保持最新状态。 Docker在容器中维护这些文件的具体细节可能会可能会随着Docker版本的演进而改变，因此您不该自己管理/etc文件，而应该用以下Docker选项。

四个不同的选项会影响容器域名服务。

参数	描述
`-h HOSTNAME` or `--hostname=HOSTNAME`	设置容器的主机名。 该设置的值将会被写入 `/etc/hostname` ；写入 `/etc/hosts` **作为容器的面向主机IP地址的名称（笔者按：在/etc/hosts里添加一条记录，IP是宿主机可以访问的IP，host就是你设置的host）**，并且是容器内部 `/bin/bash` 在其提示符下显示的名称。 但主机名不容易从容器外面看到。 它不会出现在 `docker ps` 或任何其他容器的 `/etc/hosts` 文件中。
`--link=CONTAINER_NAME` or `ID:ALIAS`	在 `run` 容器时使用此选项为新容器的 `/etc/hosts` 添加了一个名为 `ALIAS` 的额外条目，指向由 `CONTAINER_NAME_or_ID` 标识的 `CONTAINER_NAME_or_ID` 的IP地址。这使得新容器内的进程可以连接到主机名 `ALIAS` 而不必知道其IP。 `--link=` 选项将在下面进行更详细的讨论。 因为Docker可以在重新启动时为链接的容器分配不同的IP地址，Docker会更新收件人容器的 `/etc/hosts` 文件中的 `ALIAS` 条目。
`--dns=IP_ADDRESS...`	在容器的 `/etc/resolv.conf` 文件添加 `nameserver` 行，IP地址为指定IP。 容器中的进程在如果需要访问 `/etc/hosts` 里的主机名，就会连接到这些IP地址的53端口，寻找名称解析服务。
`--dns-search=DOMAIN...`	通过在容器的 `/etc/resolv.conf` 写入 `search` 行，在容器内使用裸不合格的主机名时搜索的域名。 当容器进程尝试访问 `host` 并且搜索域 `example.com` 被设置时，例如，DNS逻辑不仅将查找 `host` ，还将查找 `host.example.com` 。使用 `--dns-`

	search=. 如果您不想设置搜索域。
`--dns-opt=OPTION...`	通过将 `options` 行写入容器的 `/etc/resolv.conf` 设置DNS解析器使用的选项。有关有效选项的列表，请参阅 `resolv.conf` 文档

在没有 `--dns=IP_ADDRESS...` ， `--dns-search=DOMAIN...` 或 `--dns-opt=OPTION...` 选项的情况下，**Docker使每个容器的 `/etc/resolv.conf` 看起来像宿主机的 `/etc/resolv.conf`** 。当创建容器的 `/etc/resolv.conf` ，Docker daemon会从主机的原始文件中过滤掉所有localhost IP地址 `nameserver` 条目。

过滤是必要的，因为主机上的所有localhost地址都不可从容器的网络中访问。 过滤之后，如果容器的 `/etc/resolv.conf` 文件中没有更多的 `nameserver` 条目，Docker daemon会将Google DNS名称服务器（8.8.8.8和8.8.4.4）添加到容器的DNS配置中。 如果守护进程启用了IPv6，则也会添加公共IPv6 Google DNS名称服务器（2001:4860:4860::8888 和 2001:4860:4860::8844）。

> **注意**： 如果您需要访问主机的localhost解析器，则必须在主机上修改DNS服务，以便侦听从容器内可访问的non-localhost地址。

您可能会想知道宿主机的 `/etc/resolv.conf` 文件发生了什么变化。 `docker daemon` 有一个文件更改通知程序，它将监视主机DNS配置的更改。

> **注意**： 文件更改通知程序依赖于Linux内核的inotify功能。由于此功能目前与overlay文件系统驱动不兼容，因此使用“overlay”的Docker daemon将无法利用 `/etc/resolv.conf` 自动更新的功能。

当宿主机文件更改时，所有 `resolv.conf` 与主机匹配的**停止的容器将立即更新到最新的主机配置**。当宿主机配置更改时，**运行的容器将需要停止并开始接收主机更改**，这是由于缺少设备，以确保在容器运行时对 `resolv.conf` 文件的原子写入。 如果容器修改了默认的 `resolv.conf` 文件，则不会替换该文件，因为如果替换，将会覆盖容器执行的更改。 如果选项（ `--dns` ， `--dns-search` 或 `--dns-opt` ）已被用于修改默认的主机配置，则更换主机的 `/etc/resolv.conf` 也不会发生。

> **注意**： 对于在Docker 1.5.0中实现 `/etc/resolv.conf` 更新功能之前创建的容器：当主机 `resolv.conf` 文件更改时，这些容器将**不会**收到更新。 只有使用Docker 1.5.0及以上版本创建的容器才能使用此自动更新功能。

原文

https://docs.docker.com/engine/userguide/networking/default_network/configure-dns/

拓展阅读

Docker存储驱动的选择：https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#docker-ce

用户定义网络中的内嵌DNS服务器

本节中的信息涵盖用户自定义网络中的容器的内嵌DNS服务器操作。连接到用户自定义网络的容器的DNS lookup与连接到默认 `bridge` 网络的容器的工作机制不同。

> **注意**：为了保持向后兼容性，默认 `bridge` 网络的DNS配置保持不变，有关默认网桥中DNS配置的详细信息，请参阅默认网桥中的DNS 。

从Docker 1.10开始，Docker daemon实现了一个内嵌的DNS服务器，它为任何使用有效 `name` 、 `net-alias` 或使用 `link` 别名所创建的容器提供内置的服务发现能力。 Docker如何管理容器内DNS配置的具体细节可随着Docker版本的改变而改变。 所以你不应该自己管理容器内的 `/etc/hosts` 、 `/etc/resolv.conf` 等文件，而是使用以下的Docker选项。

影响容器域名服务的各种容器选项。

`--name=CONTAINER-NAME`	使用 `--name` 配置的容器名称用于发现用户自定义网络中的容器。 内嵌DNS服务器维护容器名称及其IP地址（在容器连接的网络上）之间的映射。
`--network-alias=ALIAS`	除如上所述的 `--name` 以外，容器可使用用户自定义网络中的一个或多个 `--network-alias` （或 `docker network connect` 命令中的 `--alias` 选项）发现。 内嵌DNS服务器维护特定用户自定义网络中所有容器别名及IP之间的映射。 通过在 `docker network connect` 命令中使用 `--alias` 选项，容器可在不同的网络中具有不同的别名。
`--link=CONTAINER_NAME:ALIAS`	在 `run` 容器时使用此选项为嵌入式DNS提供了一个名为 `ALIAS` 的额外条目，指向由 `CONTAINER_NAME` 标识的IP地址。 当使用 `--link` 时，嵌入式DNS将确保只在使用了 `--link` 选项的容器上进行本地化查找。 这允许新容器内的进程连接到容器，而不必知道其名称或IP。
`--dns=[IP_ADDRESS...]`	如果嵌入式DNS服务器无法从容器中解析名称、解析请求，嵌入式DNS服务器将使用 `--dns` 选项传递的IP地址转发DNS查询。 这些 `--dns` IP地址由嵌入式DNS服务器管理，不会在容器的 `/etc/resolv.conf` 文件中更新。
`--dns-search=DOMAIN...`	当容器内使用主机名不合格时所设置的域名。这些 `--dns-search` 选项由嵌入式DNS服务器管理，不会在容器的 `/etc/resolv.conf` 文件中更新。当容器进程尝试访问 `host` 并且搜索域 `example.com` 被设置时，例如，DNS逻辑不仅将查找 `host` ，还将查找 `host.example.com` 。
`--dns-opt=OPTION...`	设置DNS解析器使用的选项。 这些选项由嵌入式DNS服务器管理，不会在容器的 `/etc/resolv.conf` 文件中更新。有关有效选项的列表，请参阅 `resolv.conf` 文档。

在没有 `--dns=IP_ADDRESS...` ， `--dns-search=DOMAIN...` 或 `--dns-opt=OPTION...` 选项的情况下，**Docker使用宿主机的 `/etc/resolv.conf`** （ `docker daemon` 运行的地方）。 在执行此操作时，damon会从宿主机的原始文件中过滤出所有localhost IP地址 `nameserver` 条目。

过滤是必要的，因为宿主机上的所有localhost地址都不可从容器的网络中访问。过滤之后，如果容器的 `/etc/resolv.conf` 文件中没有更多的 `nameserver` 条目，daemon会将公共Google DNS名称服务器（8.8.8.8和8.8.4.4）添加到容器的DNS配置中。 如果daemon启用了IPv6，则也会添加公共IPv6 Google DNS名称服务器（2001:4860:4860::8888 以及 2001:4860:4860::8844）。

> **注意**： 如果您需要访问宿主机的localhost解析器，则必须修改宿主机上的DNS服务，以便侦听从容器内可访问的non-localhost地址。
>
> **注意**： DNS服务器始终为 `127.0.0.11` 。

原文

https://docs.docker.com/engine/userguide/networking/configure-dns/

参考&拓展阅读

Docker内置DNS：https://jimmysong.io/blogs/docker-embedded-dns/

Dns: http://blog.csdn.net/waltonwang/article/details/54098592

Docker Compose简介

经过前文讲解，我们可使用Dockerfile（或Maven）构建镜像，然后使用docker命令操作容器，例如docker run、docker kill等。

然而，使用分布式应用一般包含若干个服务，每个服务一般都会部署多个实例。如果每个服务都要手动启停，那么效率之低、维护量之大可想而知。

本章我们来讨论如何使用Docker Compose来轻松、高效地管理容器。为了简单起见，本章将Docker Compose简称为Compose。

Compose是一个用于定义和运行多容器Docker应用程序的工具，前身是Fig。它非常适合用在开发、测试、构建CI工作流等场景。本书所使用的Compose版本是1.10.0。

TIPS

Compose的GitHub：https://github.com/docker/compose

安装Docker Compose

本节我们来讨论如何安装Compose。

安装Compose

Compose有多种安装方式，例如通过Shell、pip以及将Compose作为容器安装等。本书讲解通过Shell来安装的方式，其他安装方式可详见官方文档：https://docs.docker.com/compose/install/

(1) 通过以下命令自动下载并安装适应系统版本的Compose

```
sudo curl -L https://github.com/docker/compose/releases/download/1.16.1/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

(2) 为安装脚本添加执行权限

```
chmod +x /usr/local/bin/docker-compose
```

这样，Compose就安装完成了。

可使用以下命令测试安装结果。

```
docker-compose --version
```

可输出类似于如下的内容。

```
docker-compose version 1.16.1, build 1719ceb
```

说明Compose已成功安装。

安装Compose命令补全工具

我们已成功安装Compose，然而，当我们输入 `docker-compose` 并按下Tab键时，Compose并没有为我们补全命令。要想使用Compose的命令补全，我们需要安装命令补全工具。

命令补全工具在Bash和Zsh下的安装方式不同，本书演示的是Bash下的安装。其他Shell以及其他操作系统上的安装，可详见Docker的官方文档：https://docs.docker.com/compose/completion/ ，笔者不作赘述。

- 执行以下命令，即可安装命令补全工具。

```
curl -L https://raw.githubusercontent.com/docker/compose/$(docker-compose version --short)/contrib/completion/bash/docker-compose -o /etc/bash_completion.d/docker-compose
```

这样，在重新登录后，输入 `docker-compose` 并按下Tab键，Compose就可自动补全命令了。

官方文档

Docker Compose安装：https://docs.docker.com/compose/install/

命令补全工具安装：https://docs.docker.com/compose/completion/

Docker Compose快速入门

本节我们来探讨Compose使用的基本步骤，并编写一个简单示例快速入门。

基本步骤

使用Compose大致有三个步骤：

- 使用Dockerfile（或其他方式）定义应用程序环境，以便在任何地方重现该环境。
- 在docker-compose.yml文件中定义组成应用程序的服务，以便各个服务在一个隔离的环境中一起运行。
- 运行docker-compose up命令，启动并运行整个应用程序。

入门示例

下面笔者以之前课上用到的Eureka为例讲解Compose的基本步骤。

- 在 `microservice-discovery-eureka-0.0.1-SNAPSHOT.jar` 所在路径（默认是项目的target目录）创建Dockerfile文件，并在其中添加如下内容。

```
FROM java:8
VOLUME /tmp
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 9000
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

- 在 `microservice-discovery-eureka-0.0.1-SNAPSHOT.jar` 所在路径创建文件docker-compose.yml，在其中添加如下内容。

```
version: '2'            # 表示该docker-compose.yml文件使用的是Version 2 file format

services:
  eureka:                # 指定服务名称
    build: .            # 指定Dockerfile所在路径
    ports:
      - "8761:8761"        # 指定端口映射，类似docker run的-p选项，注意使用字符串形式
```

- 在 `docker-compose.yml` 所在路径执行以下命令。

```
docker-compose up
```

Compose就会自动构建镜像并使用镜像启动容器。我们也可使用 `docker-compose up -d` 后台启动并运行这些容器。

- 访问： `http://宿主机IP:8761/` ，即可访问Eureka Server首页。

工程、服务、容器

Docker Compose将所管理的容器分为三层，分别是工程（project），服务（service）以及容器（container）。Docker Compose运行目录下的所有文件（docker-compose.yml, extends文件或环境变量文件等）组成一个工程（默认为docker-compose.yml所在目录的目录名称）。一个工程可包含多个服务；每个服务中定义了容器运行的镜像、参数和依赖，一个服务可包括多个容器实例。

对应《入门示例》一节，工程名称是docker-compose.yml所在的目录名。该工程包含了1个服务，服务名称是eureka；执行docker-compose up时，启动了eureka服务的1个容器实例。

docker-compose.yml常用命令

docker-compose.yml是Compose的默认模板文件。该文件有多种写法，例如Version 1 file format、Version 2 file format、Version 2.1 file format、Version 3 file format等。其中，Version 1 file format将逐步被被弃用；Version 2.x及Version 3.x基本兼容，是未来的趋势。考虑到目前业界的使用情况，本节只讨论Version 2 file format下的常用命令。

(1) build

配置构建时的选项，Compose会利用它自动构建镜像。build的值可以是一个路径，例如：

```
build: ./dir
```

也可以是一个对象，用于指定Dockerfile和参数，例如：

```
build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

(2) command

覆盖容器启动后默认执行的命令。示例：

```
command: bundle exec thin -p 3000
```

也可以是一个list，类似于Dockerfile中的CMD指令，格式如下：

```
command: [bundle, exec, thin, -p, 3000]
```

(3) dns

配置dns服务器。可以是一个值，也可以是一个列表。示例：

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

(4) dns_search

配置DNS的搜索域名，可以是一个值，也可以是一个列表。示例：

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

(5) environment

环境变量设置，可使用数组或字典两种方式。示例：

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

(6) env_file

从文件中获取环境变量，可指定一个文件路径或路径列表。如果通过 `docker-compose -f FILE` 指定了Compose文件，那么env_file中的路径是Compose文件所在目录的相对路径。使用environment指定的环境变量会覆盖env_file指定的环境变量。示例：

```
env_file: .env

env_file:
  - ./common.env   # 共用
  - ./apps/web.env # web用
  - /opt/secrets.env # 密码用
```

(7) expose

暴露端口，只将端口暴露给连接的服务，而不暴露给宿主机。示例：

```
expose:
 - "3000"
 - "8000"
```

(8) external_links

连接到docker-compose.yml外部的容器，甚至并非Compose管理的容器，特别是提供共享或公共服务的容器。格式跟links类似，例如：

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

(9) image

指定镜像名称或镜像id，如果本地不存在该镜像，Compose会尝试下载该镜像。

示例：

```
image: java
```

(10) links

连接到其他服务的容器。可以指定服务名称和服务别名（ `SERVICE:ALIAS` ），也可只指定服务名称。例如：

```
web:
  links:
   - db
   - db:database
   - redis
```

(11) networks

详见本书《Docker Compose网络设置》一节。

(12) network_mode

设置网络模式。示例：

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

(13) ports

暴露端口信息，可使用 `HOST:CONTAINER` 的格式，也可只指定容器端口（此时宿主机将会随机选择端口），类似于 `docker run -p` 。

需要注意的是，当使用 `HOST:CONTAINER` 格式映射端口时，容器端口小于60将会得到错误的接口，因为yaml会把 `xx:yy` 的数字解析为60进制。因此，建议使用字符串的形式。示例：

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
```

(14) volumes

卷挂载路径设置。可以设置宿主机路径 （ `HOST:CONTAINER` ） ，也可指定访问模式 （ `HOST:CONTAINER:ro` ） 。示例：

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

(15) volumes_from

从另一个服务或容器挂载卷。可指定只读（ro）或读写（rw），默认是读写（rw）。示例：

```
volumes_from:
 - service_name
 - service_name:ro
 - container:container_name
 - container:container_name:rw
```

TIPS

(1) docker-compose.yml还有很多其他命令，比如depends_on、pid、devices等。限于篇幅，笔者仅挑选常用的命令进行讲解，其他命令不作赘述。感兴趣的读者们可参考官方文档：https://docs.docker.com/compose/compose-file/ 。

docker-compose常用命令

和docker命令一样，docker-compose命令也有很多选项。下面我们来详细探讨docker-compose的常用命令。

build

构建或重新构建服务。服务被构建后将会以 `project_service` 的形式标记，例如： `composetest_db` 。

help

查看指定命令的帮助文档，该命令非常实用。docker-compose所有命令的帮助文档都可通过该命令查看。

```
docker-compose help COMMAND
```

示例：

```
docker-compose help build        # 查看docker-compose build的帮助
```

kill

通过发送 `SIGKILL` 信号停止指定服务的容器。示例：

```
docker-compose kill eureka
```

该命令也支持通过参数来指定发送的信号，例如：

```
docker-compose kill -s SIGINT
```

logs

查看服务的日志输出。

port

打印绑定的公共端口。示例：

```
docker-compose port eureka 8761
```

这样就可输出eureka服务8761端口所绑定的公共端口。

ps

列出所有容器。示例：

```
docker-compose ps
```

也可列出指定服务的容器，示例：

```
docker-compose ps eureka
```

pull

下载服务镜像。

rm

删除指定服务的容器。示例：

```
docker-compose rm eureka
```

run

在一个服务上执行一个命令。示例：

```
docker-compose run web bash
```

这样即可启动一个web服务，同时执行bash命令。

scale

设置指定服务运行容器的个数，以service=num的形式指定。示例：

```
docker-compose scale user=3 movie=3
```

start

启动指定服务已存在的容器。示例：

```
docker-compose start eureka
```

stop

停止已运行的容器。示例：

```
docker-compose stop eureka
```

停止后，可使用 `docker-compose start` 再次启动这些容器。

up

构建、创建、重新创建、启动，连接服务的相关容器。所有连接的服务都会启动，除非它们已经运行。

`docker-compose up` 命令会聚合所有容器的输出，当命令退出时，所有容器都会停止。

使用 `docker-compose up -d` 可在后台启动并运行所有容器。

TIPS

(1) 本节仅讨论常用的docker-compose命令，其他命令可详见Docker官方文档：https://docs.docker.com/compose/reference/overview/ 。

Docker Compose网络设置

本节我们来详细探讨Compose的网络设置。本节介绍的网络特性仅适用于Version 2 file format，Version 1 file format不支持该特性。

基本概念

默认情况下，Compose会为我们的应用创建一个网络，服务的每个容器都会加入该网络中。这样，容器就可被该网络中的其他容器访问，不仅如此，该容器还能以服务名称作为hostname被其他容器访问。

默认情况下，应用程序的网络名称基于Compose的工程名称，而项目名称基于docker-compose.yml所在目录的名称。如需修改工程名称，可使用--project-name标识或COMPOSE_PORJECT_NAME环境变量。

举个例子，假如一个应用程序在名为myapp的目录中，并且docker-compose.yml如下所示：

```
version: '2'
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
```

当我们运行docker-compose up时，将会执行以下几步：

- 创建一个名为myapp_default的网络；
- 使用web服务的配置创建容器，它以“web”这个名称加入网络myapp_default；
- 使用db服务的配置创建容器，它以“db”这个名称加入网络myapp_default。

容器间可使用服务名称（web或db）作为hostname相互访问。例如，web这个服务可使用 `postgres://db:5432` 访问db容器。

更新容器

当服务的配置发生更改时，可使用docker-compose up命令更新配置。

此时，Compose会删除旧容器并创建新容器。新容器会以不同的IP地址加入网络，名称保持不变。任何指向旧容器的连接都会被关闭，容器会重新找到新容器并连接上去。

links

前文讲过，默认情况下，服务之间可使用服务名称相互访问。links允许我们定义一个别名，从而使用该别名访问其他服务。举个例子：

```
version: '2'
services:
  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```

这样web服务就可使用db或database作为hostname访问db服务了。

指定自定义网络

一些场景下，默认的网络配置满足不了我们的需求，此时我们可使用networks命令自定义网络。networks命令允许我们创建更加复杂的网络拓扑并指定自定义网络驱动和选项。不仅如此，我们还可使用networks将服务连接到不是由Compose管理的、外部创建的网络。

如下，我们在其中定义了两个自定义网络。

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - front
  app:
    build: ./app
    networks:
      - front
      - back
  db:
    image: postgres
    networks:
      - back

networks:
  front:
    # Use a custom driver
    driver: custom-driver-1
  back:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
```

```
      bar: "2"
```

其中，proxy服务与db服务隔离，两者分别使用自己的网络；app服务可与两者通信。

由本例不难发现，使用networks命令，即可方便实现服务间的网络隔离与连接。

配置默认网络

除自定义网络外，我们也可为默认网络自定义配置。

```
version: '2'

services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres

networks:
  default:
    # Use a custom driver
    driver: custom-driver-1
```

这样，就可为该应用指定自定义的网络驱动。

使用已存在的网络

一些场景下，我们并不需要创建新的网络，而只需加入已存在的网络，此时可使用external选项。示例：

```
networks:
  default:
    external:
      name: my-pre-existing-network
```

小练习：使用Docker Compose编排WordPress博客

```
version: '2'
services:
  mysql:
    image: mysql:5.7
    expose:
      - "3306"
    environment:
      - MYSQL_ROOT_PASSWORD=123456
  wordpress:
    image: wordpress
    ports:
      - "80:80"
    environment:
      - WORDPRESS_DB_HOST=mysql
      - WORDPRESS_DB_USER=root
      - WORDPRESS_DB_PASSWORD=123456
```

WARNING

这里，MySQL镜像只能用5.x的镜像，不能使用8.x的镜像。否则WordPress无法正常连接到MySQL。原因是：目前PHP 7.x（例如7.1.4）所使用的字符集与MySQL 8.x所使用的默认字符集不同：https://bugs.php.net/bug.php?id=74461

控制服务启动顺序

在生产中，往往有严格控制服务启动顺序的需求。然而Docker Compose自身并不具备该能力。要想实现启动顺序的控制，Docker Compose建议我们使用：

- wait-for-it
- dockerize
- wait-for

本文演示如何使用wait-for-it 来控制服务的启动顺序。

还用前面编排WordPress博客的例子，现在我们想让MySQL先启动，启动完成后再启动WordPress。

分析

分析：找到WordPress的Dockerfile：https://github.com/docker-library/wordpress/blob/666c5c06d7bc9d02c71fd48a74911248be6f5a5b/php5.6/apache/Dockerfile

可看到类似如下的内容：

```
COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["apache2-foreground"]
```

也就是说，这个WordPress的Dockerfile执行了命令： `docker-entrypoint.sh apache2-foreground` 。

> 顺便再复习一下，ENTRYPOINT与CMD的区别——ENTRYPOINT指令是不会被覆盖的，CMD指令会覆盖。详见博客：https://segmentfault.com/q/1010000000417103 。

wait-for-it

在wait-for-it的官方GiTHub中，有详细的例子：

要想使用wait-for-it，只需使用如下形式即可：

```
wait-for-it.sh 想等的地址:端口 -- 原本想执行的命令
```

答案

分析到这里，答案就很简单了：只需在WordPress的容器中添加wait-for-it.sh，然后将原本的命令用wait-for-it包裹，即可实现控制启动顺序的目标。

```
version: '2'
services:
```

```
  mysql:
    image: mysql:5.7
    expose:
      - "3306"
    environment:
      - MYSQL_ROOT_PASSWORD=123456
  wordpress:
    image: wordpress
    ports:
      - "80:80"
    volumes:
      - ./wait-for-it.sh:/wait-for-it.sh
    environment:
      - WORDPRESS_DB_HOST=mysql
      - WORDPRESS_DB_USER=root
      - WORDPRESS_DB_PASSWORD=123456
    entrypoint: "sh /wait-for-it.sh mysql:3306 -- docker-entrypoint.sh"
    command: ["apache2-foreground"]
```

参考文档

- 《Controlling startup order in Compose》：https://docs.docker.com/compose/startup-order/ 。

在生产环境中使用Docker Compose

在development中使用Compose定义应用程序时，可使用此定义，在不同环境（如CI，staging和production）中运行应用程序。

部署应用最简单的方法是在单机服务器上运行，类似于运行development环境的方式。如果要对应用程序扩容，可在Swarm集群上运行Compose应用程序。

Modify your Compose file for production（为生产环境修改您的Compose文件）

您几乎肯定会对您的应用配置进行更改，从而使这些配置更适合线上环境。 这些更改可能包括：

- 删除任何绑定到应用程序代码的Volume，以便代码保持在容器内，不能从外部更改
- 绑定到主机上的不同端口
- 设置不同的环境变量（例如，减少日志的冗长程度或启用email发送）
  - DEBUG INFO WARN ERROR FETAL
- 指定重启策略（例如， `restart: always` ），从而避免停机
- 添加额外服务（例如，日志聚合器）

因此，您可能需要定义一个额外的Compose文件，比如 `production.yml` ，它指定了适用于生产的配置。此配置文件只需包含从原始Compose文件的修改。该附加Compose文件，可在原始的 `docker-compose.yml` 基础上被应用，从而创建新的配置。

一旦获得了第二个配置文件，可使用 `-f` 选项告诉Compose：

```
docker-compose -f docker-compose.yml -f production.yml up -d
```

请参阅 Using multiple compose files 获取更完整的示例。

Deploying changes（部署修改）

当您更改应用代码时，您需要重新构建镜像并重新创建容器。例如，重新部署名为 `web` 的服务，可使用：

```
$ docker-compose build web
$ docker-compose up --no-deps -d web
```

这将会先重新构建 `web` 的镜像，然后停止、销毁、重新创建 `web` 服务。 `--no-deps` 标志可防止Compose重新创建任何 `web` 依赖的服务。

Running Compose on a single server（单机服务器上运行Compose）

通过适当地设置 `DOCKER_HOST` 、 `DOCKER_TLS_VERIFY` 和 `DOCKER_CERT_PATH` 等环境变量，可使用Compose将应用程序部署到远程的Docker主机。 对于像这样的任务，Docker Machine 可使本地/远程Docker主机管理变得非常简单，即使您没有远程部署也推荐使用Docker Machine。

一旦您设置了如上环境变量，所有正常的 `docker-compose` 命令将无需进一步的配置。

Running Compose on a Swarm cluster（在Swarm集群上运行Compose）

Docker Swarm ，是一款Docker原生的集群系统，它暴露了与单个Docker主机相同的API，这意味着您可在Swarm实例上使用Compose，并在多个主机上运行应用程序。

阅读更多关于集成指Compose/Swarm整合的内容，请详见 integration guide 。

原文

https://docs.docker.com/compose/production/

实战：使用Docker Compose运行ELK

- ElasticSearch【存储】
- Logtash【日志聚合器】
- Kibana【界面】

答案：

```
version: '2'
services:
 elasticsearch:
  image: elasticsearch
  # command: elasticsearch
  ports:
   - "9200:9200"    # REST API端口
   - "9300:9300"    # RPC端口
 logstash:
  image: logstash
  command: logstash -f /etc/logstash/conf.d/logstash.conf
  volumes:
   - ./config:/etc/logstash/conf.d
   - /opt/build:/opt/build
  ports:
   - "5000:5000"
 kibana:
  image: kibana
  environment:
   - ELASTICSEARCH_URL=http://elasticsearch:9200
  ports:
   - "5601:5601"
```

`logstash.conf` 参考示例：

```
input {
  file {
    codec => json
    path => "/opt/build/*.json"
  }
}
filter {
  grok {
    match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
  }
}
output {
```

```
  elasticsearch {
    hosts => "elasticsearch:9200"
  }
}
```

参考文档

https://docs.docker.com/compose/samples-for-compose/#samples-tailored-to-demo-compose

使用Docker Compose伸缩应用

执行：

`docker-compose scale 服务名称=服务实例个数` 即可。