# Docker

# # 1 综述

## ## 概述

Docker是一个用于开发，交付和运行应用程序的开放平台。Docker使您能够将应用程序与基础架构分开，从而可以快速交付软件。借助Docker，您可以以与管理应用程序相同的方式来管理基础架构。通过利用Docker的方法来快速交付，测试和部署代码，您可以大大减少编写代码和在生产环境中运行代码之间的延迟。

## ## 我可以将Docker用于什么？

### 快速，一致地交付您的应用程序

Docker允许开发人员使用提供您的应用程序和服务的本地容器在标准化环境中工作，从而简化了开发生命周期。容器非常适合持续集成和持续交付（CI / CD）工作流程。

请考虑以下示例方案：

您的开发人员在本地编写代码，并使用Docker容器与同事共享他们的工作。

他们使用Docker将其应用程序推送到测试环境中，并执行自动和手动测试。

当开发人员发现错误时，他们可以在开发环境中对其进行修复，然后将其重新部署到测试环境中以进行测试和验证。

测试完成后，将修补程序推送给生产环境就像将更新的镜像推送到生产环境一样简单。

### 响应式部署和扩展

Docker基于容器的平台允许高度可移植的工作负载。Docker容器可以在开发人员的本地笔记本电脑上，数据中心中的物理或虚拟机上，云提供商上或混合环境中运行。

Docker的可移植性和轻量级的特性还使您可以轻松地动态管理工作负载，并根据业务需求指示实时扩展或拆除应用程序和服务。

### 在同一硬件上运行更多工作负载

Docker轻巧快速。它为基于虚拟机管理程序的虚拟机提供了可行，经济高效的替代方案，因此您可以利用更多的计算能力来实现业务目标。Docker非常适合于高密度环境以及中小型部署，而您需要用更少的资源做更多的事情。

## ## 优势

容器之所以受欢迎，是因为它们提供了额外的好处，例如：

* 敏捷的应用程序创建和部署：与使用VM镜像相比，容器镜像创建的简便性和效率更高。
* 持续的开发，集成和部署：通过快速简单的回滚（由于镜像不可更改），提供可靠且频繁的容器镜像构建和部署。
* 开发和运营的关注点分离：在构建/发布时而不是在部署时创建应用程序容器镜像，从而将应用程序与基础架构分离。
* 可观察性不仅可以显示操作系统级别的信息和指标，还可以显示应用程序的运行状况和其他信号。
* 跨开发，测试和生产的环境一致性：在便携式计算机上与在云中相同地运行。
* 云和操作系统分发的可移植性：可在Ubuntu，RHEL，CoreOS，本地，Google Kubernetes Engine和其他任何地方运行。
* 以应用程序为中心的管理：提高抽象级别，从在虚拟硬件上运行OS到使用逻辑资源在OS上运行应用程序。
* 松散耦合，分布式，弹性，解放的微服务：应用程序被分解成较小的独立部分，并且可以动态部署和管理–而不是在一台大型单机上运行的整体堆栈。
* 资源隔离：可预测的应用程序性能。
* 资源利用：高效率和高密度。

# # 2 架构

## ## Docker引擎

Docker Engine是具有以下主要组件的客户端-服务器应用程序：

服务器是一种长期运行的程序，称为守护程序进程（ dockerd命令）。

REST API，它指定程序可以用来与守护程序进行通信并指示其操作的接口。

命令行界面（CLI）客户端（docker命令）。

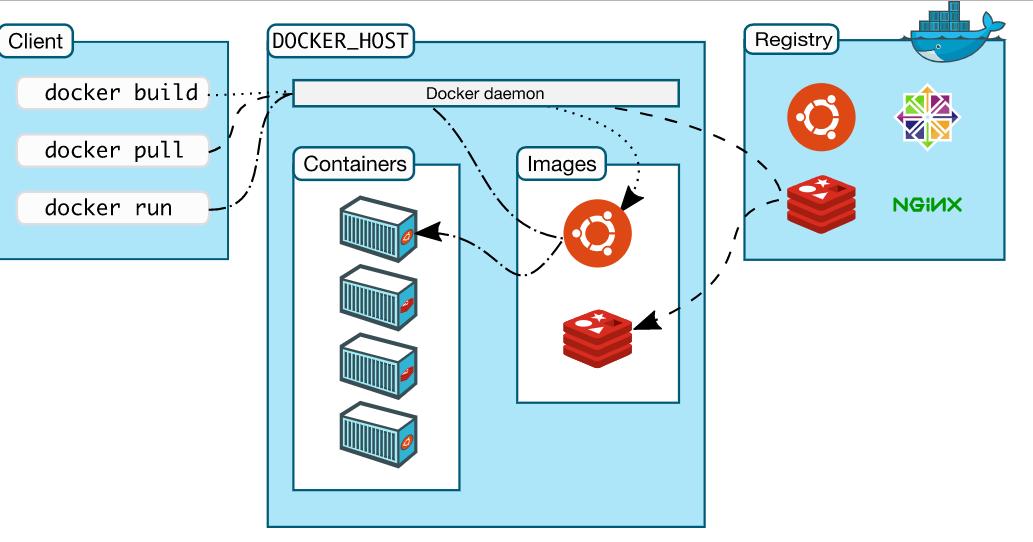


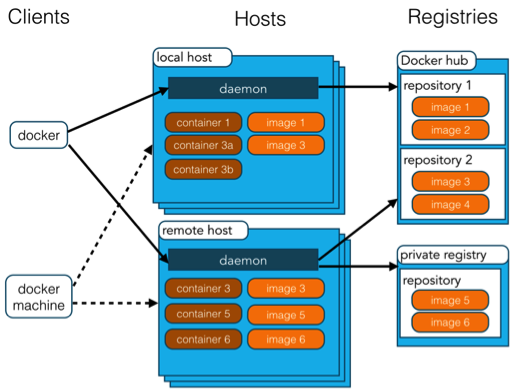
CLI使用Docker REST API通过脚本或直接CLI命令来控制Docker守护程序或与Docker守护程序进行交互。许多其他Docker应用程序都使用基础API和CLI。

守护程序创建和管理Docker 对象，例如镜像，容器，网络和卷。

## ## Docker架构

Docker使用客户端-服务器架构。Docker 客户端与Docker 守护进程进行对话，该守护进程完成了构建，运行和分发Docker容器的繁重工作。Docker客户端和守护程序可以 在同一系统上运行，也可以将Docker客户端连接到远程Docker守护程序。Docker客户端和守护程序在UNIX套接字或网络接口上使用REST API进行通信。





### Docker守护程序

Docker守护程序（dockerd）侦听Docker API请求并管理Docker对象，例如镜像，容器，网络和卷。守护程序还可以与其他守护程序通信以管理Docker服务。

### Docker客户端

Docker客户端（docker）是许多Docker用户与Docker交互的主要方式。当您使用诸如之类的命令时docker run，客户端会将这些命令发送到dockerd，以执行它们。该docker命令使用Docker API。Docker客户端可以与多个守护程序通信。

### Docker注册表

Docker 注册表存储Docker镜像。Docker Hub是任何人都可以使用的公共注册表，并且默认情况下，Docker已配置为在Docker Hub上查找镜像。您甚至可以运行自己的私人注册表。如果使用Docker数据中心（DDC），则其中包括Docker可信注册表（DTR）。

使用docker pull或docker run命令时，所需的镜像将从配置的注册表中提取。使用该docker push命令时，会将镜像推送到配置的注册表。

### Docker对象

使用Docker时，您正在创建和使用镜像，容器，网络，卷，插件和其他对象。本节是其中一些对象的简要概述。

#### 镜像

一个镜像是用于创建一个码头工人容器指令的只读模板。通常，一个镜像基于另一个镜像，并进行一些其他自定义。例如，您可以基于该ubuntu 镜像构建镜像，但是安装Apache Web服务器和您的应用程序，以及运行该应用程序所需的配置详细信息。

您可以创建自己的镜像，也可以仅使用其他人创建并在注册表中发布的镜像。要构建自己的镜像，您可以 使用简单的语法创建一个Dockerfile，以定义创建镜像并运行它所需的步骤。Dockerfile中的每个指令都会在镜像中创建一个层。当您更改Dockerfile并重建镜像时，仅重建那些已更改的层。与其他虚拟化技术相比，这是使镜像如此轻巧，小型和快速的部分原因。

#### 容器

容器是镜像的可运行实例。您可以使用Docker API或CLI创建，启动，停止，移动或删除容器。您可以将容器连接到一个或多个网络，将存储连接到它，甚至根据其当前状态创建新镜像。

默认情况下，容器与其他容器及其主机之间的隔离程度相对较高。您可以控制容器的网络，存储或其他基础子系统与其他容器或与主机的隔离程度。

容器由其镜像以及在创建或启动时为其提供的任何配置选项定义。删除容器后，未存储在持久性存储中的状态更改将消失。

#### 服务

服务允许你扩展在多个码头工人守护进程的容器，这是所有工作一起作为一个群有多个管理人员和工人。群的每个成员都是Docker守护程序，并且所有守护程序都使用Docker API进行通信。服务允许您定义所需的状态，例如在任何给定时间必须可用的服务副本的数量。默认情况下，该服务在所有工作节点之间是负载平衡的。对于消费者而言，Docker服务似乎是一个单独的应用程序。Docker Engine在Docker 1.12及更高版本中支持集群模式。

## ## 底层技术

Docker用Go编写，并利用Linux内核的多个功能来交付其功能。

### 命名空间

Docker使用一种称为namespaces提供容器的隔离工作区的技术。运行容器时，Docker会为该容器创建一组 命名空间。

这些名称空间提供了一层隔离。容器的每个方面都在单独的名称空间中运行，并且对其的访问仅限于该名称空间。

Docker Engine在Linux上使用以下名称空间：

的pid命名空间：进程隔离（PID：进程ID）。

该net命名空间：管理网络接口（NET：网络）。

该ipc命名空间：管理访问IPC资源（IPC：进程间通信）。

该mnt命名空间：管理文件系统挂载点（MNT：摩）。

该uts命名空间：隔离内核和版本标识符。（UTS：Unix时间共享系统）。

### 控制组

Linux上的Docker引擎还依赖于另一种称为控制组 （cgroups）的技术。cgroup将应用程序限制为一组特定的资源。控制组允许Docker Engine将可用的硬件资源共享给容器，并有选择地实施限制和约束。例如，您可以限制特定容器可用的内存。

### 联合文件系统

联合文件系统或UnionFS是通过创建图层进行操作的文件系统，使其非常轻便且快速。Docker Engine使用UnionFS为容器提供构建模块。Docker Engine可以使用多个UnionFS变体，包括AUFS，btrfs，vfs和DeviceMapper。

### 容器格式

Docker Engine将名称空间，控制组和UnionFS组合到一个称为容器格式的包装器中。默认容器格式为libcontainer。将来，Docker可能会通过与BSD Jails或Solaris Zones等技术集成来支持其他容器格式。

# # 3 系统级命令

## 启动

systemctl enable docker #设置docker开机自启

systemctl start docker #开启docker

## 验证是否正确安装

sudo docker run hello-world

## 搜索镜像

docker search <xxxx>

## Docke 镜像加速

请在 /etc/docker/daemon.json 中写入如下内容（如果文件不存在请新建该文件）：

{"registry-mirrors":["https://registry.docker-cn.com"]}

## 容器

### 运行容器

docker run ubuntu:15.10

#### 运行交互式的容器

docker run -i -t ubuntu:15.10 /bin/bash

#### 启动容器（后台模式）

docker run -d ubuntu:15.10 /bin/sh

#### 常用选项说明

-d, --detach=false， 指定容器运行于前台还是后台，默认为false

-i, --interactive=false， 打开STDIN，用于控制台交互

-t, --tty=false， 分配tty设备，该可以支持终端登录，默认为false

-u, --user=""， 指定容器的用户

-a, --attach=[]， 登录容器（必须是以docker run -d启动的容器）

-w, --workdir=""， 指定容器的工作目录

-c, --cpu-shares=0， 设置容器CPU权重，在CPU共享场景使用

-e, --env=[]， 指定环境变量，容器中可以使用该环境变量

-m, --memory=""， 指定容器的内存上限

-P, --publish-all=false， 指定容器暴露的端口

-p, --publish=[]， 指定容器暴露的端口

-h, --hostname=""， 指定容器的主机名

-v, --volume=[]， 给容器挂载存储卷，挂载到容器的某个目录

--volumes-from=[]， 给容器挂载其他容器上的卷，挂载到容器的某个目录

--cap-add=[]， 添加权限，权限清单详见：http://linux.die.net/man/7/capabilities

--cap-drop=[]， 删除权限，权限清单详见：http://linux.die.net/man/7/capabilities

--cidfile=""， 运行容器后，在指定文件中写入容器PID值，一种典型的监控系统用法

--cpuset=""， 设置容器可以使用哪些CPU，此参数可以用来容器独占CPU

--device=[]， 添加主机设备给容器，相当于设备直通

--dns=[]， 指定容器的dns服务器

--dns-search=[]， 指定容器的dns搜索域名，写入到容器的/etc/resolv.conf文件

--entrypoint=""， 覆盖image的入口点

--env-file=[]， 指定环境变量文件，文件格式为每行一个环境变量

--expose=[]， 指定容器暴露的端口，即修改镜像的暴露端口

--link=[]， 指定容器间的关联，使用其他容器的IP、env等信息

--lxc-conf=[]， 指定容器的配置文件，只有在指定--exec-driver=lxc时使用

--name=""， 指定容器名字，后续可以通过名字进行容器管理，links特性需要使用名字

--net="bridge"， 容器网络设置:

bridge 使用docker daemon指定的网桥

host //容器使用主机的网络

container:NAME\_or\_ID >//使用其他容器的网路，共享IP和PORT等网络资源

none 容器使用自己的网络（类似--net=bridge），但是不进行配置

--privileged=false， 指定容器是否为特权容器，特权容器拥有所有的capabilities

--restart="no"， 指定容器停止后的重启策略:

no：容器退出时不重启

on-failure：容器故障退出（返回值非零）时重启

always：容器退出时总是重启

--rm=false， 指定容器停止后自动删除容器(不支持以docker run -d启动的容器)

--sig-proxy=true， 设置由代理接受并处理信号，但是SIGCHLD、SIGSTOP和SIGKILL不能被代理

#### 重启容器

docker restart <container\_id 或者 container\_name>

### 查看容器

docker ps

#### 查看所有容器

docker ps –a

### 停止运行容器

docker stop <container\_id 或者 container\_name>

### 杀掉一个运行中的容器

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

docker kill -s KILL mynginx

### 删除容器

docker rm <container\_id 或者 container\_name>

#### #删除所有未运行的容器

（已经运行的删除不了，未运行的就一起被删除了）

sudo docker rm $(sudo docker ps -a -q)

docker ps

-a, --all Show all containers (default shows just running)

-q, --quiet Only display numeric IDs

## 镜像

### 列出本地主机上的镜像列表

docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

REPOSITORY：表示镜像的仓库源

TAG：镜像的标签

IMAGE ID：镜像ID

CREATED：镜像创建时间

SIZE：镜像大小

若只使用 ubuntu，docker 将默认使用 ubuntu:latest 镜像。

### 获取一个新的镜像

docker pull ubuntu:13.10

### 查找镜像

docker search httpd

NAME # 镜像仓库

DESCRIPTION # 镜像描述信息

STARS # 镜像收藏数

OFFICIAL # 是否为docker官方发布的镜像

AUTOMATED # 是否为自动化构建的镜像

### 更新镜像

1. 创建一个容器，并进入容器

docker run -t -i ubuntu:15.10 /bin/bash

1. 修改已存在的镜像为自己需要的新镜像，比如

在运行的容器内使用 apt-get update 命令进行更新

1. 最后，docker commit来提交容器副本

docker commit -m="has update" -a="runoob" e218edb10161 runoob/ubuntu:v2

-m:提交的描述信息

-a:指定镜像作者

e218edb10161：容器ID

runoob/ubuntu:v2:指定要创建的目标镜像名

1. docker images 命令来查看我们的新镜像

### 构建镜像

1. 创建一个 Dockerfile 文件，比如
2. FROM centos:6.7
3. MAINTAINER Fisher "fisher@sudops.com"
4. RUN /bin/echo 'root:123456' |chpasswd
5. RUN useradd runoob
6. RUN /bin/echo 'runoob:123456' |chpasswd
7. RUN /bin/echo -e "LANG=\"en\_US.UTF-8\"" >/etc/default/local
8. EXPOSE 22
9. EXPOSE 80
10. CMD /usr/sbin/sshd -D

每一个指令都会在镜像上创建一个新的层，每一个指令的前缀都必须是大写的。

第一条FROM，指定使用哪个镜像源

RUN 指令告诉docker 在镜像内执行命令，安装了什么。。。

2. 使用 Dockerfile 文件

docker build -t runoob/centos:6.7 /path/to/dockerfile

-t ：指定要创建的目标镜像名

/path/to/dockerfile ：Dockerfile 文件所在目录，可以指定Dockerfile 的绝对路径

### 设置镜像标签

docker tag 860c279d2fec runoob/centos:dev

docker tag 镜像ID，这里是 860c279d2fec ,用户名称、镜像源名(repository name)和新的标签名(tag)。

### 导出 export 保存save 镜像

sudo docker export 容器CONTAINER ID > 导出地址文件名

示例：

sudo docker export 234wer2323dfdfdsfq > /home/export.tar

docker export [OPTIONS] CONTAINER

示例：

docker export -o mysql-`date +%Y%m%d`.tar a404c6c174a2

### 导入 import 加载镜像

若导入的镜像不可以run，查看此链接（主要是command 参数没有设定以及 -it）

<https://www.cnblogs.com/wish123/p/6573899.html>

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

docker images runoob/ubuntu:v4

## dockerfile

### 基本结构

Dockerfile 一般分为四部分：基础镜像信息、维护者信息、镜像操作指令和容器启动时执行指令，’#’ 为 Dockerfile 中的注释。

### 文件说明

Docker以从上到下的顺序运行Dockerfile的指令。为了指定基本镜像，第一条指令必须是FROM。一个声明以＃字符开头则被视为注释。可以在Docker文件中使用RUN，CMD，FROM，EXPOSE，ENV等指令。

#### FROM：指定基础镜像，必须为第一个命令

格式：

　　FROM <image>

　　FROM <image>:<tag>

　　FROM <image>@<digest>

示例：

　　FROM mysql:5.6

注：

　　tag或digest是可选的，如果不使用这两个值时，会使用latest版本的基础镜像

#### MAINTAINER: 维护者信息

格式：

MAINTAINER <name>

示例：

MAINTAINER Jasper Xu

MAINTAINER sorex@163.com

MAINTAINER Jasper Xu sorex@163.com

#### RUN：构建镜像时执行的命令

RUN用于在镜像容器中执行命令，其有以下两种命令执行方式：

shell执行

格式：

RUN <command>

exec执行

格式：

RUN ["executable", "param1", "param2"]

示例：

RUN ["executable", "param1", "param2"]

RUN apk update

RUN ["/etc/execfile", "arg1", "arg1"]

注：

RUN指令创建的中间镜像会被缓存，并会在下次构建中使用。如果不想使用这些缓存镜像，可以在构建时指定--no-cache参数，如：docker build --no-cache

#### ADD：

将本地文件添加到容器中，tar类型文件会自动解压(网络压缩资源不会被解压)，可以访问网络资源，类似wget

格式：

ADD <src>... <dest>

ADD ["<src>",... "<dest>"] 用于支持包含空格的路径

示例：

ADD hom\* /mydir/ # 添加所有以"hom"开头的文件

ADD hom?.txt /mydir/ # ? 替代一个单字符,例如："home.txt"

ADD test relativeDir/ # 添加 "test" 到 `WORKDIR`/relativeDir/

ADD test /absoluteDir/ # 添加 "test" 到 /absoluteDir/

#### COPY：

功能类似ADD，但是是不会自动解压文件，也不能访问网络资源

#### CMD：

构建容器后调用，也就是在容器启动时才进行调用。

格式：

CMD ["executable","param1","param2"] (执行可执行文件，优先)

CMD ["param1","param2"] (设置了ENTRYPOINT，则直接调用ENTRYPOINT添加参数)

CMD command param1 param2 (执行shell内部命令)

示例：

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

CMD ["/usr/bin/wc","--help"]

注：

　　CMD不同于RUN，CMD用于指定在容器启动时所要执行的命令，而RUN用于指定镜像构建时所要执行的命令。

#### ENTRYPOINT：

配置容器，使其可执行化。配合CMD可省去"application"，只使用参数。

格式：

ENTRYPOINT ["executable", "param1", "param2"] (可执行文件, 优先)

ENTRYPOINT command param1 param2 (shell内部命令)

示例：

FROM ubuntu

ENTRYPOINT ["top", "-b"]

CMD ["-c"]

注：

　　　ENTRYPOINT与CMD非常类似，不同的是通过docker run执行的命令不会覆盖ENTRYPOINT，而docker run命令中指定的任何参数，都会被当做参数再次传递给ENTRYPOINT。Dockerfile中只允许有一个ENTRYPOINT命令，多指定时会覆盖前面的设置，而只执行最后的ENTRYPOINT指令。

而docker run执行的命令会覆盖CMD 命令。例如

FROM centos:centos7.1.1503

CMD ["/bin/echo","This is test cmd"]

docker run -it logan/test:0.1

将会打印：This is test cmd

但是

docker run -it logan/test:0.1 /bin/bash

将不再打印CMD设定的打印信息，因为启动容器时/bin/bash覆盖了/bin/echo命令。

#### LABEL：

用于为镜像添加元数据

格式：

LABEL <key>=<value> <key>=<value> <key>=<value> ...

示例：

　　LABEL version="1.0" description="这是一个Web服务器" by="IT笔录"

注：

使用LABEL指定元数据时，一条LABEL指定可以指定一或多条元数据，指定多条元数据时不同元数据之间通过空格分隔。推荐将所有的元数据通过一条LABEL指令指定，以免生成过多的中间镜像。

#### ENV：

设置环境变量

格式：

ENV <key> <value> # <key>之后的所有内容均会被视为其<value>的组成部分，因此，一次只能设置一个变量

ENV <key>=<value> ... # 可以设置多个变量，每个变量为一个"<key>=<value>"的键值对，如果<key>中包含空格，可以使用\来进行转义，也可以通过""来进行标示；另外，反斜线也可以用于续行

示例：

ENV myName John Doe

ENV myDog Rex The Dog

ENV myCat=fluffy

#### EXPOSE：

指定于外界交互的端口

格式：

EXPOSE <port> [<port>...]

示例：

EXPOSE 80 443

EXPOSE 8080

EXPOSE 11211/tcp 11211/udp

注：

　　EXPOSE并不会让容器的端口访问到主机。要使其可访问，需要在docker run运行容器时通过-p来发布这些端口，或通过-P参数来发布EXPOSE导出的所有端口

#### VOLUME：

用于指定持久化目录

格式：

VOLUME ["/path/to/dir"]

示例：

VOLUME ["/data"]

VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"

注：

　　一个卷可以存在于一个或多个容器的指定目录，该目录可以绕过联合文件系统，并具有以下功能：

1 卷可以容器间共享和重用

2 容器并不一定要和其它容器共享卷

3 修改卷后会立即生效

4 对卷的修改不会对镜像产生影响

5 卷会一直存在，直到没有任何容器在使用它

#### WORKDIR：

工作目录，类似于cd命令

格式：

WORKDIR /path/to/workdir

示例：

WORKDIR /a (这时工作目录为/a)

WORKDIR b (这时工作目录为/a/b)

WORKDIR c (这时工作目录为/a/b/c)

注：

　　通过WORKDIR设置工作目录后，Dockerfile中其后的命令RUN、CMD、ENTRYPOINT、ADD、COPY等命令都会在该目录下执行。在使用docker run运行容器时，可以通过-w参数覆盖构建时所设置的工作目录。

复制代码

#### USER:

指定运行容器时的用户名或 UID，后续的 RUN 也会使用指定用户。使用USER指定用户时，可以使用用户名、UID或GID，或是两者的组合。当服务不需要管理员权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户

格式:

　　USER user

　　USER user:group

　　USER uid

　　USER uid:gid

　　USER user:gid

　　USER uid:group

示例：

　　USER www

注：

　　使用USER指定用户后，Dockerfile中其后的命令RUN、CMD、ENTRYPOINT都将使用该用户。镜像构建完成后，通过docker run运行容器时，可以通过-u参数来覆盖所指定的用户。

#### ARG：

用于指定传递给构建运行时的变量

格式：

ARG <name>[=<default value>]

示例：

ARG site

ARG build\_user=www

#### ONBUILD：

用于设置镜像触发器

格式：

　　ONBUILD [INSTRUCTION]

示例：

　　ONBUILD ADD . /app/src

　　ONBUILD RUN /usr/local/bin/python-build --dir /app/src

注：

　　当所构建的镜像被用做其它镜像的基础镜像，该镜像中的触发器将会被钥触发

### dockerfile示例：

#### 示例1

# This my first nginx Dockerfile

# Version 1.0

# Base images 基础镜像

FROM centos

#MAINTAINER 维护者信息

MAINTAINER tianfeiyu

#ENV 设置环境变量

ENV PATH /usr/local/nginx/sbin:$PATH

#ADD 文件放在当前目录下，拷过去会自动解压

ADD nginx-1.8.0.tar.gz /usr/local/

ADD epel-release-latest-7.noarch.rpm /usr/local/

#RUN 执行以下命令

RUN rpm -ivh /usr/local/epel-release-latest-7.noarch.rpm

RUN yum install -y wget lftp gcc gcc-c++ make openssl-devel pcre-devel pcre && yum clean all

RUN useradd -s /sbin/nologin -M www

#WORKDIR 相当于cd

WORKDIR /usr/local/nginx-1.8.0

RUN ./configure --prefix=/usr/local/nginx --user=www --group=www --with-http\_ssl\_module --with-pcre && make && make install

RUN echo "daemon off;" >> /etc/nginx.conf

#EXPOSE 映射端口

EXPOSE 80

#CMD 运行以下命令

CMD ["nginx"]

#### 示例2

# Name: eureka-server

# Time: 2018

FROM java:8-jre

MAINTAINER GuoliangDi <digl@yiche.com>

RUN mkdir /app

WORKDIR /app

COPY ./target/eureka-server.jar /app

CMD ["java", "-Xmx200m", "-jar", "/app/eureka-server.jar"]

EXPOSE 30081

## Docker-compose

### 概述

每个微服务一般都会部署多个实例，如果每个微服务都要手动启停，那么效率之低，维护量之大可想而知

使用 Docker Compose 可以轻松、高效的管理容器，它是一个用于定义和运行多容器 Docker 的应用程序工具

### 安装

安装 Docker Compose 可以通过下面命令自动下载适应版本的 Compose，并为安装脚本添加执行权限

sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose

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

查看安装是否成功

docker-compose -v

### 快速入门

1. 打包项目，获得 jar 包 docker-demo-0.0.1-SNAPSHOT.jar

mvn clean package

1. 在 jar 包所在路径创建 Dockerfile 文件，添加以下内容

FROM java:8

VOLUME /tmp

ADD docker-demo-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"]

1. 在 jar 包所在路径创建文件 docker-compose.yml，添加以下内容

version: '2' # 表示该 Docker-Compose 文件使用的是 Version 2 file

services:

docker-demo: # 指定服务名称

build: . # 指定 Dockerfile 所在路径

ports: # 指定端口映射

- "9000:8761"

1. 在 docker-compose.yml 所在路径下执行该命令 Compose 就会自动构建镜像并使用镜像启动容器

docker-compose -f docker-compose.yml up

docker-compose -f docker-compose.yml up -d // 后台启动并运行容器

1. 访问 <http://localhost:9000/hello> 即可访问微服务接口

### docker-compose.yml具体示例

version: "2.2"

services:

# ELK

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:5.2.2

ports:

- ${ES\_SERVER\_PORT}:${ES\_SERVER\_PORT}

networks:

- xproject

env\_file:

- bootstrap.env

environment:

- xpack.security.enabled=false

- ES\_JAVA\_OPTS=${ES\_JAVA\_OPTS}

healthcheck:

test: curl -f http://localhost:${ES\_SERVER\_PORT}

interval: ${INTERVAL}

timeout: ${TIMEOUT}

retries: 10

kibana:

image: docker.elastic.co/kibana/kibana:5.2.2

depends\_on:

elasticsearch:

condition: service\_healthy

ports:

- ${KIBANA\_SERVER\_PORT}:${KIBANA\_SERVER\_PORT}

networks:

- xproject

env\_file:

- bootstrap.env

environment:

- xpack.security.enabled=false

healthcheck:

test: curl -f http://localhost:${KIBANA\_SERVER\_PORT}

interval: ${INTERVAL}

timeout: ${TIMEOUT}

retries: 10

logstash:

image: docker.elastic.co/logstash/logstash:5.2.2

depends\_on:

kibana:

condition: service\_healthy

command: logstash -f /elk-config/logstash.config

ports:

- ${LOGSTASH\_SERVER\_PORT}:${LOGSTASH\_SERVER\_PORT}

- ${LOGSTASH\_UDP\_SERVER\_PORT}:${LOGSTASH\_UDP\_SERVER\_PORT}

networks:

- xproject

volumes:

- $PWD/config/elk-config:/elk-config

env\_file:

- bootstrap.env

environment:

- JAVA\_TOOL\_OPTIONS=${JAVA\_TOOL\_OPTIONS}

healthcheck:

test: curl -f http://localhost:${LOGSTASH\_SERVER\_PORT}

interval: ${INTERVAL}

timeout: ${TIMEOUT}

retries: 10

# Redis

redis-server:

image: redis:3-alpine

ports:

- ${REDIS\_SERVER\_PORT}:${REDIS\_SERVER\_PORT}

networks:

- xproject

env\_file:

- bootstrap.env

healthcheck:

test: nc -v -w 5 -z localhost ${REDIS\_SERVER\_PORT}

interval: ${INTERVAL}

timeout: ${TIMEOUT}

retries: 10

auth-mysql:

image: mysql:5.6

ports:

- ${AUTH\_MYSQL\_PORT}:${AUTH\_MYSQL\_PORT}

networks:

- xproject

volumes:

- "data-auth-mysql:/var/lib/mysql"

env\_file:

- bootstrap.env

environment:

- SPRING\_PROFILES\_ACTIVE=docker

- SPRING\_CLOUD\_CONFIG\_PROFILE=docker

- MYSQL\_ROOT\_PASSWORD=${MYSQL\_ROOT\_PASSWORD}

- MYSQL\_USER=${AUTH\_MYSQL\_USER}

- MYSQL\_PASSWORD=${AUTH\_MYSQL\_PASSWORD}

- MYSQL\_DATABASE=${AUTH\_MYSQL\_DB}

healthcheck:

test: mysqladmin ping -u${AUTH\_MYSQL\_USER} -p${AUTH\_MYSQL\_PASSWORD}

interval: ${INTERVAL}

timeout: ${TIMEOUT}

retries: 10

networks:

xproject:

volumes:

data-rabbit:

data-zipkin-mysql:

data-auth-mysql:

data-logs:

#######################Useful Reference#########################

#https://docs.docker.com/compose/compose-file/compose-file-v2/

#volumes:

# db-data:

#extra\_hosts:

# - "somehost:162.242.195.82"

# - "otherhost:50.31.209.229"

#logging:

# driver: syslog

# options:

# syslog-address: "tcp://192.168.0.42:123"

#ulimits:

# nproc: 65535

# nofile:

# soft: 20000

# hard: 40000

#volumes:

# - "/var/run/docker.sock:/var/run/docker.sock"

# - db-data:/var/lib/postgresql/data

#test: ["CMD", "nc", "-v", "-w", "5", "-z", "localhost", "${CONFIG\_SERVER\_PORT}"]

#test: ["CMD-SHELL", "nc -v -w 5 -z localhost ${CONFIG\_SERVER\_PORT}"]

#restart: on-failure/none/always

更多详情请查看： <https://github.com/junneyang/xxproject>

#### docker-compose.yml 属性

version：指定 docker-compose.yml 文件的写法格式

services：多个容器集合

build：配置构建时，Compose 会利用它自动构建镜像，该值可以是一个路径，也可以是一个对象，用于指定 Dockerfile 参数

build: ./dir

---------------

build:

context: ./dir

dockerfile: Dockerfile

args:

buildno: 1

command：覆盖容器启动后默认执行的命令

command: bundle exec thin -p 3000

----------------------------------

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

dns：配置 dns 服务器，可以是一个值或列表

dns: 8.8.8.8

------------

dns:

- 8.8.8.8

- 9.9.9.9

dns\_search：配置 DNS 搜索域，可以是一个值或列表

dns\_search: example.com

------------------------

dns\_search:

- dc1.example.com

- dc2.example.com

environment：环境变量配置，可以用数组或字典两种方式

environment:

RACK\_ENV: development

SHOW: 'ture'

-------------------------

environment:

- RACK\_ENV=development

- SHOW=ture

env\_file：从文件中获取环境变量，可以指定一个文件路径或路径列表，其优先级低于 environment 指定的环境变量

env\_file: .env

---------------

env\_file:

- ./common.env

expose：暴露端口，只将端口暴露给连接的服务，而不暴露给主机

expose:

- "3000"

- "8000"

image：指定服务所使用的镜像

image: java

network\_mode：设置网络模式

network\_mode: "bridge"

network\_mode: "host"

network\_mode: "none"

network\_mode: "service:[service name]"

network\_mode: "container:[container name/id]"

ports：对外暴露的端口定义，和 expose 对应

ports: # 暴露端口信息 - "宿主机端口:容器暴露端口"

- "8763:8763"

- "8763:8763"

links：将指定容器连接到当前连接，可以设置别名，避免ip方式导致的容器重启动态改变的无法连接情况

links: # 指定服务名称:别名

- docker-compose-eureka-server:compose-eureka

volumes：卷挂载路径

volumes:

- /lib

- /var

logs：日志输出信息

--no-color 单色输出，不显示其他颜.

-f, --follow 跟踪日志输出，就是可以实时查看日志

-t, --timestamps 显示时间戳

--tail 从日志的结尾显示，--tail=200

# # 4 与容器交互命令

## 端口映射、绑定

将容器端口5000 映射到实际主机5008端口

docker run -d -p 5008:5000 training/webapp python app.py

多个端口

docker run -d -p 5008:5000 -p 8081:8080 -p 6379:6379 training/webapp python app.py

## 指定容器绑定的网络地址

docker run -d -p 127.0.0.1:5008:5000 training/webapp python app.py

例如，绑定 127.0.0.15008

这样我们就可以通过访问 127.0.0.1:5008 来访问容器的 5000 端口。

如果要绑定 UDP 端口，可以在端口后面加上 /udp。

docker run -d -p 127.0.0.1:5008:5000/udp training/webapp python app.py

## 查看端口的绑定情况

docker port <container\_id 或者 container\_name> 5000

回显：127.0.0.1:5008

## 文件夹路径映射

原因：

1. 容器被删除后通过镜像重新启动容器后，之前容器中发生的所有改变全部都会消失，

2. 当你需要修改一下配置时，需要登录容器，然后修改，然后重启，然后通过镜像重启后你会惊喜的发现，之前的修改全没了。。。

3. 通过将宿主机上的代码目录挂载到容器中，修改宿主机上的代码，容器中的代码也会自动更新。

-v 用来将宿主机目录挂载到容器中，`:` 前面是宿主机目录，`:` 后面是容器目录

docker run -itd -p 80:80 -v /opt /html:/usr/local/nginx/html --name=nginx nginx

## Logs-查看容器内部的标准输出

docker logs -f <container\_id 或者 container\_name>

-f: 让 docker logs 像使用 tail -f 一样来输出容器内部的标准输出。

## 查看应用程序容器的进程

docker top <container\_id 或者 container\_name>

## 查看 Docker 的底层信息

docker inspect <container\_id 或者 container\_name>

## 在运行的容器中执行命令

docker exec -it mysqlTest bash

mysql -uroot -p

## 传入参数

docker run --name=mysql -it -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=root -d docker.io/mysql:5.7

-e MYSQL\_ROOT\_PASSWORD=root

## Docker容器连接

端口映射并不是唯一把 docker 连接到另一个容器的方法。

docker 有一个连接系统允许将多个容器连接在一起，共享连接信息。

docker 连接会创建一个父子关系，其中父容器可以看到子容器的信息。

要建立容器之间的链接,必须在启动时通过参数--name $container\_name给容器命名。

命令： 容器直接建立连接要使用--link选项

--link <name or id>:alias

1. 首先创建mysql容器（源容器）：

docker run --name=db\_mysql -it -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=root -d docker.io/mysql:5.7

1. 创建一个web容器（接收容器）,并链接到db容器

docker run -d -P --name web --link db\_mysql: db\_mysql training/webapp python app.py

1. 可能需要更改jdbc连接

由 url:jdbc:mysql://127.0.0.1:3306/~~~~

改为 url:jdbc:mysql://db\_mysql:3306/~~~~

连接多个容器

docker run -d -P --name web --link db\_mysql: db\_mysql --link redis:redis training/webapp python app.py

要点：

接收容器能够访问源容器的信息，Docker提供了两种方式：

1. 环境变量
2. /etc/hosts文件

其中：

Docker在连接容器的时候，会根据--link提供的参数自动的在接收者容器中创建一些环境变量，这些变量包括源容器的Dockerfile中使用ENV命令设置的环境变量和源容器启动时(docker run)，使用-e或者--env， --env-file参数指定的环境变量。

除了环境变量之外，Docker也在接收容器的/etc/hosts文件中更新了hosts信息。

与环境变量不同的是，如果源容器重启了，接收容器中/etc/hosts中的信息会自动更新。

## Docker修改容器参数

1. 查看容器的全id

docker ps --no-trunc

1. 然后停止容器

docker stop <container\_id 或者 container\_name>

1. 修改宿主主机 /var/lib/docker/containers/容器ID

docker容器的大多数参数配置都在这几个文件里，找到相应的配置去改动就可以了

注意此种使用方法（临时性）：

容器不应该是长久性的东西，要保持容器的可抛弃性，有问题就应该rm掉，数据保存在容器外，然后直接run新的容器。参数修改是那个时候进行的。

## 宿主主机与docker文件传输

### 1 直接在主机上拷贝

从主机复制到容器 sudo docker cp host\_path containerID:container\_path

从容器复制到主机 sudo docker cp containerID:container\_path host\_path

### 2 文件挂载方式

运行容器时，添加挂载目录

docker run -itd -p 80:80 -v /opt /html: /mnt --name=nginx nginx

进入容器内部，从挂载目录复制

cp /mnt/sourcefile /path/to/destfile

### 3 输入输出方式

容器启动时

docker run -i ubuntu /bin/bash -c 'cat > /path/to/container/file' < /path/to/host/file/

或者，在容器启动后进入容器执行命令的方式

docker exec -it <container\_id> bash -c 'cat > /path/to/container/file' < /path/to/host/file/

# # 5 Docker跨多主机容器网络

## 五种方案：

一、利用OpenVSwitch

二、利用Weave

三、Docker在1.9之后支持的Overlay network（官方的做法）

Docker 1.9 Overlay Network实现跨主机网络互通

四、将多个物理机的容器组到一个物理网络来

1.创建自己的网桥br0

2.将docker默认网桥绑定到br0

多台物理主机之间的容器互联

五、修改主机docker默认的虚拟网段，然后在各自主机上分别把对方的docker网段加入到路由表中，配合iptables即可实现docker容器跨主机通信

参考-<https://blog.csdn.net/JackLiu16/article/details/80906525>

## 原生的跨多主机容器网络

参考-<https://blog.csdn.net/ztsinghua/article/details/51545561>

### 前提条件：

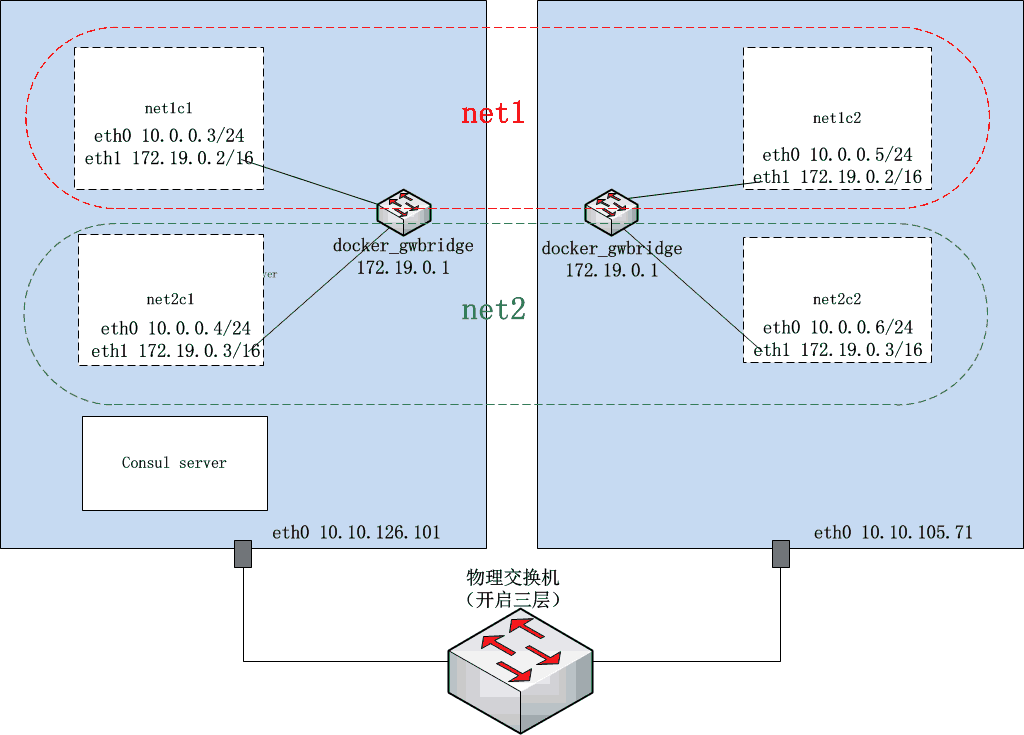
1、Linux Kernel版本 >= 3.16；

2、需要一个外部Key-value Store（官方例子中使用的是consul）；

3、各物理主机上的Docker Daemon需要一些特定的启动参数；

4、物理主机允许某些特定TCP/UDP端口可用。

### 架构



### 跨多主机容器网络搭建

#### 1创建consul 服务

考虑到kv store在本文并非关键，仅作跨多主机容器网络创建启动的前提条件之用，因此仅用包含一个server节点的”cluster”。

$./consul -d agent -server -bootstrap-expect 1 -data-dir ./data -node=master -bind=10.10.126.101 -client=0.0.0.0 &

#### Consul(题外)

参考--<https://tonybai.com/2015/07/06/implement-distributed-services-registery-and-discovery-by-consul/>

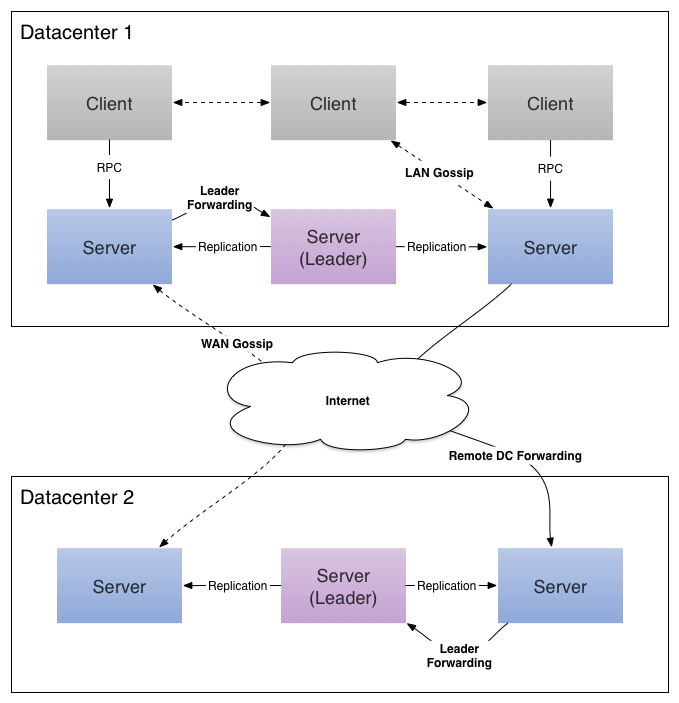
Consul的方案更“一站式”， 内置了服务注册与发现框 架、分布一致性协议实现、健康检查、Key/Value存储、多数据中心方案，不再需要依赖其他工具（比如ZooKeeper等）

Consul用Golang实现，因此具有天然可移植性(支持Linux、windows和Mac OS X)

安装包仅包含一个可执行文件，方便部署，与Docker等轻量级容器可无缝配合。

##### 一、Consul Cluster

要想利用Consul提供的服务实现服务的注册与发现，我们需要建立Consul Cluster。在Consul方案中，每个提供服务的节点上都要部署和运行Consul的agent，所有运行Consul agent节点的集合构成Consul Cluster。Consul agent有两种运行模式：Server和Client。这里的Server和Client只是Consul集群层面的区分，与搭建在Cluster之上 的应用服务无关。以Server模式运行的Consul agent节点用于维护Consul集群的状态，官方建议每个Consul Cluster至少有3个或以上的运行在Server mode的Agent，Client节点不限。



每个数据中心的Consul Cluster都会在运行于server模式下的agent节点中选出一个Leader节点，这个选举过程通过Consul实现的raft协议保证，多个 server节点上的Consul数据信息是强一致的。处于client mode的Consul agent节点比较简单，无状态，仅仅负责将请求转发给Server agent节点。

##### 安装consul

下载安装包

<https://www.consul.io/downloads.html>

在文件位置，解压安装包，并移动至当前用户的bin目录（或者添加环境变量）

unzip consul\_1.6.2\_linux\_amd64.zip

mv consul ~/bin

环境变量添加方法

先查看

#查看PATH：

echo $PATH

方法一：

export PATH=/usr/local/bin:$PATH

#配置完后可以通过echo $PATH查看配置结果。

#生效方法：立即生效

#有效期限：临时改变，只能在当前的终端窗口中有效，当前窗口关闭后就会恢#复原有的path配置

#用户局限：仅对当前用户

方法二：

#通过修改.bashrc文件:

vim ~/.bashrc

#在最后一行添上：

export PATH=/usr/local/bin:$PATH

#生效方法：（有以下两种）

#1、关闭当前终端窗口，重新打开一个新终端窗口就能生效

#2、输入“source ~/.bashrc”命令，立即生效

#有效期限：永久有效

#用户局限：仅对当前用户

方法三：

#通过修改profile文件:

vim /etc/profile

export PATH=/usr/local/bin:$PATH

#生效方法：系统重启

#有效期限：永久有效

#用户局限：对所有用户

方法四：

#通过修改environment文件:

vim /etc/environment

在PATH="/usr/local/sbin:/usr/sbin:/usr/bin:/sbin:/bin"中加入

":/usr/local/bin"

#生效方法：系统重启

#有效期限：永久有效

#用户局限：对所有用户

测试安装结果

终端中输入 consul 查看是否可以运行

##### 搭建

实验环境和节点角色如下：

n1(Ubuntu 14.04 x86\_64): 10.10.105.71 server mode

n2(Ubuntu 12.04 x86\_64): 10.10.126.101 server mode with Consul Web UI

n3(Ubuntu 9.04 i386): 10.10.126.187 client mode

在三台主机上分别下载和安装Consul包，安装包很简单，只是包含一个可执行文件consul。在n2主机上还要下载一份Consul Web UI包，支持图形化展示Consul cluster中的节点状态和服务状态。

###### 节点1

consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=n1 -bind=10.10.105.71 -dc=dc1

# -node：节点的名称

# -bind：绑定的一个地址，用于节点之间通信的地址，可以是内外网，必须是可以访问到的地址

# -server：这个就是表示这个节点是个SERVER

# -bootstrap-expect：这个就是表示期望提供的SERVER节点数目，数目一达到，它就会被激活，然后就是LEADER了

# -dc：指明数据中心的名字

# -client 0.0.0.0 -ui：启动UI（为了方便后续的UI访问）

###### 节点2

consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=n2 -bind=10.10.126.101 -ui-dir ./dist -dc=dc1

从两个server agent的启动日志可以看出，n1、n2启动后并不知道集群其他节点的存在。以n1为例，通过consul members和consul info查看当前agent状态：

可以看出，n1上的agent当前状态是Follower，bootstrap = false；n2同样也是这个情况。整个Cluster并未完成Bootstrap过程。

我们用consul join命令触发Cluster bootstrap过程，我们在n1上执行如下命令：

$ consul join 10.10.126.101

Successfully joined cluster by contacting 1 nodes.

我们通过consul join子命令将当前节点加入包含成员10.10.126.101（也就是n2)的集群中去。命令执行结果通过n1和n2的日志可以观察到：

join后，两台主机互相知道了对方，并进行了leader election过程，n2被选举为Leader。

在n2主机上通过consul info确认一下n2 agent的状态：

$consul info

可以看到n2的state已经为Leader了，n1的state依旧是Follower。

到这里，n1和n2就成为了dc1这个数据中心Consul Cluster的两个节点，而且是用来维护集群状态的Server node。n2被选举为Leader，n1是Folllower。

如果作为Leader的n2退出集群，我们来看看集群状态会发生怎样变化。在n2上，我们通过consul leave命令告诉n2上的agent离开集群并退出：

$ consul leave

这个时候我们在n1上通过consul info查看，n1的状态依旧是Follower，也就是说在双server节点的集群下，一个server退出，将产生无Leader状态。在三 server节点集群里，Leader退出，其余两个会再协商选出一个新Leader，但一旦再退出一个节点，同样集群就不会再有Leader了。 当然，如果是单节点bootstrap的集群( -bootstrap-expect 1 )，集群只有一个server节点，那这个server节点自然当选Leader。

现在我们在n1上通过consul members查看集群状态：

执行结果显示：n2是Left状态。我们重新启动n2，再来看看集群的状态变化。

n2启动后，并未自动加入之前的cluster，而是依旧如第一次启动那样，看不到peers，孤立运行。

我们再来在n1上join一下：consul join 10.10.126.101

n1和n2无法再选出Leader，通过info命令看，两个节点都变成了Follower，集群仍然处于无Leader状态。

这个问题在consul的github repositroy issues中被多人多次提及，但作者似乎不将此作为bug。产生这个问题的原因是当n2退出时，consul会将/tmp/consul/raft /peers.json的内容由：

["10.10.105.71:8300", "10.10.126.101:8300"]

改为

null

n2重启后，该文件并未改变，依旧为null，n2启动就不会重新自动join到n1的cluster中。

关于这个问题的cluster恢复方法，官方在Outage Recovery一文中有明确说明。我们来测试一下：

我们打开n1和n2的/tmp/consul/raft/peers.json，将其内容统一修改为：

["10.10.126.101:8300","10.10.105.71:8300"]

然后重启n2，但加上-rejoin命令：

$ consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=n2 -bind=10.10.126.101 -ui-dir ./dist -dc=dc1 -rejoin

…. …

2015/07/03 14:56:02 [WARN] raft: Election timeout reached, restarting

这回集群的Leader重新选举成功，集群状态恢复。

###### 节点3

consul agent -data-dir /tmp/consul -node=n3 -bind=10.10.126.187 -dc=dc1

处于client mode的agent可以自由退出和启动，不会出现server mode下agent的问题。

#### 2、修改Docker Daemon DOCKER\_OPTS参数

通过Docker 1.9创建跨多主机容器网络需要重新配置每个主机节点上的Docker Daemon的启动参数：

ubuntu系统这个配置在/etc/default/docker下：

DOCKER\_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-advertise eth0:2375 --cluster-store consul://10.10.126.101:8500/network --storage-driver=devicemapper"

说明：

-H(或–host)配置的是Docker client(包括本地和远程的client)与Docker Daemon的通信媒介，也是Docker REST api的服务端口。默认是/var/run/docker.sock（仅用于本地），当然也可以通过tcp协议通信以方便远程Client访问，就像上面 配置的那样。非加密网通信采用2375端口，而TLS加密连接则用2376端口。这两个端口已经申请在IANA注册并获批，变成了知名端口。-H可以配置多个，就像上面配置的那样。 unix socket便于本地docker client访问本地docker daemon；tcp端口则用于远程client访问。这样一来：docker pull ubuntu，走docker.sock；而docker -H 10.10.126.101:2375 pull ubuntu则走tcp socket。

–cluster-advertise 配置的是本Docker Daemon实例在cluster中的地址；

–cluster-store配置的是Cluster的分布式KV store的访问地址；

如果你之前手工修改过iptables的规则，建议重启Docker Daemon之前清理一下iptables规则：sudo iptables -t nat -F, sudo iptables -t filter -F等。

#### 3、启动各节点上的Docker Daemon

以10.10.126.101为例：

$ sudo service docker start

$ ps -ef|grep docker

root 2069 1 0 Feb02 ? 00:01:41 /usr/bin/docker -d --dns 8.8.8.8 --dns 8.8.4.4 --storage-driver=devicemapper -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-advertise eth0:2375 --cluster-store consul://10.10.126.101:8500/network

启动后iptables的nat, filter规则与单机Docker网络初始情况并无二致。

101节点上初始网络driver类型：

$docker network ls

NETWORK ID NAME DRIVER

47e57d6fdfe8 bridge bridge

7c5715710e34 none null

19cc2d0d76f7 host host

#### 4、创建overlay网络net1和net2

在101节点上，创建net1：

$ sudo docker network create -d overlay net1

在71节点上，创建net2:

$ sudo docker network create -d overlay net2

之后无论在71节点还是101节点，我们查看当前网络以及驱动类型都是如下结果：

$ docker network ls

NETWORK ID NAME DRIVER

283b96845cbe net2 overlay

da3d1b5fcb8e net1 overlay

00733ecf5065 bridge bridge

71f3634bf562 none null

7ff8b1007c09 host host

此时，iptables规则也并无变化。

#### 5、启动两个overlay net下的containers

我们分别在net1和net2下面启动两个container，每个节点上各种net1和net2的container各一个：

101:

sudo docker run -itd --name net1c1 --net net1 ubuntu:14.04

sudo docker run -itd --name net2c1 --net net2 ubuntu:14.04

71:

sudo docker run -itd --name net1c2 --net net1 ubuntu:14.04

sudo docker run -itd --name net2c2 --net net2 ubuntu:14.04

启动后，我们就得到如下网络信息（容器的ip地址可能与前面拓扑图中的不一致，每次容器启动ip地址都可能变化）：

net1:

net1c1 - 10.0.0.7

net1c2 - 10.0.0.5

net2:

net2c1 - 10.0.0.4

net2c2 - 10.0.0.6

#### 6、容器连通性

在net1c1中，我们来看看其到net1和net2的连通性：

root@021f14bf3924:/# ping net1c2

PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.

64 bytes from 10.0.0.5: icmp\_seq=1 ttl=64 time=0.670 ms

64 bytes from 10.0.0.5: icmp\_seq=2 ttl=64 time=0.387 ms

^C

--- 10.0.0.5 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 999ms

rtt min/avg/max/mdev = 0.387/0.528/0.670/0.143 ms

root@021f14bf3924:/# ping 10.0.0.4

PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.

^C

--- 10.0.0.4 ping statistics ---

2 packets transmitted, 0 received, 100% packet loss, time 1008ms

可见，net1中的容器是互通的，但net1和net2这两个overlay net之间是隔离的。

# # 6 常用镜像

## ## Mysql（官方）

Docker hub 网址--<https://hub.docker.com/_/mysql>

### 启动mysql服务器实例

启动一个MySQL实例很简单：

$ docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag

$ docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag

此处some-mysql是您要分配给容器的名称，my-secret-pw是要为MySQL根用户设置的密码，并且tag是指定所需MySQL版本的标记。有关相关标签，可以是（8.0.18，8.0，8，latest，5.7.28，5.7，5，5.6.46， 5.6）中任意一个选项。

例如：

docker run --name hellomysql -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7

### 使用MySQL命令行客户端连接到MySQL

以下命令启动另一个mysql容器实例(即MySQL命令行客户端)，针对原始刚刚运行的mysql容器运行命令行客户端工具，从而允许您针对数据库实例执行SQL语句：

$ docker run -it --network some-network --rm mysql mysql -hsome-mysql -uexample-user -p

some-mysql原始mysql容器的名称在哪里（连接到some-networkDocker网络）。

例如：

docker run -it --network 127.0.0.1 --rm mysql mysql -hhellomysql -uroot –p123456

该镜像也可用作非Docker或远程实例的客户端：

$ docker run -it --rm mysql mysql -hsome.mysql.host -usome-mysql-user -p

### 通过docker stack deploy或docker-compose

实施例子stack.yml为mysql：

# Use root/example as user/password credentials

version: '3.1'

services:

db:

image: mysql

command: --default-authentication-plugin=mysql\_native\_password

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: example

adminer:

image: adminer

restart: always

ports:

- 8080:8080

运行docker stack deploy -c stack.yml mysql（或docker-compose -f stack.yml up），等它完全初始化，并且访问http://swarm-ip:8080，http://localhost:8080或http://host-ip:8080（如适用）。

### 容器shell访问和查看MySQL日志

该docker exec命令允许您在Docker容器中运行命令。以下命令行将在mysql容器内为您提供一个bash shell ：

$ docker exec -it some-mysql bash

该日志可通过Docker的容器日志获取：

$ docker logs some-mysql

### 使用自定义MySQL配置文件

MySQL的默认配置可以在中找到/etc/mysql/my.cnf，其中可以包含!includedir其他目录，例如/etc/mysql/conf.d或/etc/mysql/mysql.conf.d。请检查mysql镜像本身中的相关文件和目录以获取更多详细信息。

如果/my/custom/config-file.cnf是自定义配置文件的路径和名称，则可以mysql像这样启动容器（请注意，此命令仅使用自定义配置文件的目录路径，而忽略默认的my.cnf文件）：

$ docker run --name some-mysql -v /my/custom:/etc/mysql/conf.d -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag

这将启动一个新的容器some-mysql，其中MySQL实例从采用混合启动设置/etc/mysql/my.cnf和/etc/mysql/conf.d/config-file.cnf，并且使用后者路径中的config-file.cnf文件，也即后者路径文件优先（覆盖前者文件）。

#### 无需cnf文件进行配置

可以将许多配置选项作为标志传递给mysqld。这将使您可以灵活地自定义容器，而无需cnf文件。例如，如果要更改所有表的默认编码和排序规则以使用UTF-8（utf8mb4），请运行以下命令：

$ docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag --character-set-server=utf8mb4 --collation-server=utf8mb4\_unicode\_ci

如果您想查看可用选项的完整列表，请运行：

$ docker run -it --rm mysql:tag --verbose --help

### 环境变量

启动mysql镜像时，可以通过在docker run命令行上传递一个或多个环境变量来调整MySQL实例的配置。请注意，如果使用已包含数据库的数据目录启动容器（即已经通过指定路径下的自定义配置文件进行数据库配置，则命令行传入的参数无效），则以下任何变量都将无效：容器启动时，任何现有数据库都将保持不变。

#### MYSQL\_ROOT\_PASSWORD

此变量是必需变量，它指定将为MySQL root超级用户帐户设置的密码。在上面的示例中，将其设置为my-secret-pw。

#### MYSQL\_DATABASE

此变量是可选的，允许您指定在镜像启动时要创建的数据库的名称。如果提供了用户名/密码（请参阅下文），则将授予该用户对该数据库的超级用户访问权限（与相对应GRANT ALL）。

#### MYSQL\_USER， MYSQL\_PASSWORD

这些变量是可选的，与创建新用户和设置该用户的密码一起使用。该MYSQL\_DATABASE变量将为该用户授予超级用户权限（请参见上文）。这两个变量都是创建用户所必需的。

请注意，不需要使用此机制来创建根超级用户，该用户默认情况下是使用MYSQL\_ROOT\_PASSWORD变量指定的密码创建的。

#### MYSQL\_ALLOW\_EMPTY\_PASSWORD

这是一个可选变量。设置为yes允许以root用户的空白密码启动容器。注意：yes除非您真的知道自己在做什么，否则不建议将此变量设置为yes，因为这将使您的MySQL实例完全不受保护，从而使任何人都可以获得完全的超级用户访问权限。

#### MYSQL\_RANDOM\_ROOT\_PASSWORD

这是一个可选变量。设置yes时将为root用户生成一个随机的初始密码（使用pwgen）。生成的root密码将被打印到stdout（GENERATED ROOT PASSWORD: .....）。

#### MYSQL\_ONETIME\_PASSWORD

初始化完成后，将root用户（不是MYSQL\_USER！中指定的用户）设置为过期用户，从而在首次登录时强制更改密码。注意：仅MySQL 5.6+支持此功能。在MySQL 5.5上使用此选项将在初始化期间引发适当的错误。

### Docker的密码变量

作为通过环境变量传递敏感信息的替代方法，\_FILE可以将其附加到先前列出的环境变量中，从而使初始化脚本从容器中存在的文件中加载这些变量的值。特别是，这可用于从/run/secrets/<secret\_name>文件中存储的Docker机密加载密码。例如：

$ docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD\_FILE=/run/secrets/mysql-root -d mysql:tag

目前，这仅支持MYSQL\_ROOT\_PASSWORD，MYSQL\_ROOT\_HOST，MYSQL\_DATABASE，MYSQL\_USER，和MYSQL\_PASSWORD这几个变量。

### 初始化一个新实例（导入数据）

首次启动容器时，将创建一个具有指定名称的新数据库，并使用提供的配置变量对其进行初始化。

此外，它会执行文件（可扩展）.sh，.sql文件并且.sql.gz文件是在发现/docker-entrypoint-initdb.d。文件将按字母顺序执行。您可以mysql通过将SQL转储装载到该目录中并为自定义镜像提供需要导入的数据来轻松初始化数据库服务。

默认情况下，SQL文件将导入到MYSQL\_DATABASE变量指定的数据库中。

### 注意事项

#### 数据存储位置

重要说明：有几种方法可以存储在Docker容器中运行的应用程序使用的数据。我们鼓励mysql镜像用户熟悉可用的选项，包括：

1. 让Docker 通过使用自己的内部卷管理将数据库文件写入主机系统上的磁盘，从而管理数据库数据的存储。这是默认设置，对用户来说是简单且相当透明的。不利之处在于，对于直接在主机系统（即外部容器）上运行的工具和应用程序而言，文件可能难以定位。

2. 在主机系统上（容器外部）创建一个数据目录，并将其安装到从容器内部可见的目录中。这会将数据库文件放置在主机系统上的已知位置，并使主机系统上的工具和应用程序易于访问文件。缺点是用户需要确保目录存在，并且例如必须正确设置主机系统上的目录权限和其他安全机制。

Docker文档是了解不同存储选项和变体的一个很好的起点，并且有多个博客和论坛文章讨论了这一领域并提供了建议。我们将在此处仅显示上述后一种选择的基本过程：

1. 在主机系统上的适当卷上创建数据目录，例如/my/own/datadir。
2. mysql像这样启动容器：

$ docker run --name some-mysql -v /my/own/datadir:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag

该-v /my/own/datadir:/var/lib/mysql命令的一部分/my/own/datadir从宿主主机系统/var/lib/mysql在容器内部安装目录，默认情况下，MySQL将在该目录中写入其数据文件。

#### 在MySQL初始化完成之前没有连接

如果在容器启动时没有初始化数据库，则将创建一个默认数据库。尽管这是预期的行为，但这意味着在初始化完成之前它将不接受传入的连接。使用自动化工具（例如）docker-compose同时启动多个容器时，这可能会引起问题。

如果您尝试连接到MySQL的应用程序无法处理MySQL停机时间或等待MySQL正常启动，则可能有必要在服务启动之前放置connect-retry循环。有关官方镜像中此类实现的示例，请参阅[WordPress](https://github.com/docker-library/wordpress/blob/1b48b4bccd7adb0f7ea1431c7b470a40e186f3da/docker-entrypoint.sh#L195-L235)或[Bonita](https://github.com/docker-library/docs/blob/9660a0cccb87d8db842f33bc0578d769caaf3ba9/bonita/stack.yml#L28-L44)。

#### 针对现有数据库的用法

如果mysql使用已经包含数据库的数据目录（特别是mysql子目录）启动容器实例，$MYSQL\_ROOT\_PASSWORD则应在运行命令行中省略该变量；在任何情况下都将忽略它，并且不会以任何方式更改现有数据库。

#### 以任意用户身份运行

如果您知道目录的权限已经适当设置（例如，如上所述，针对现有数据库运行），或者需要mysqld使用特定的UID / GID 运行，则可以将此镜像--user设置为任意值（root/ 0除外）以实现所需的访问/配置：

$ mkdir data

$ ls -lnd data

drwxr-xr-x 2 1000 1000 4096 Aug 27 15:54 data

$ docker run -v "$PWD/data":/var/lib/mysql --user 1000:1000 --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag

#### 创建数据库转储

大多数普通工具都可以使用，尽管在某些情况下它们的使用可能会有些麻烦，以确保它们可以访问mysqld服务器。确保这一点的一种简单方法是docker exec在同一容器中使用和运行该工具，类似于以下内容：

$ docker exec some-mysql sh -c 'exec mysqldump --all-databases -uroot -p"$MYSQL\_ROOT\_PASSWORD"' > /some/path/on/your/host/all-databases.sql

#### 从转储文件还原数据

用于还原数据。您可以使用docker exec带-i标志的命令，类似于以下内容：

$ docker exec -i some-mysql sh -c 'exec mysql -uroot -p"$MYSQL\_ROOT\_PASSWORD"' < /some/path/on/your/host/all-databases.sql

## ## 构建mysql并配置

### 文件准备

#### Dockerfile

FROM mysql:5.7

ENV MYSQL\_ALLOW\_EMPTY\_PASSWORD yes

COPY setup.sh /Data/setup.sh

COPY sqlData.sql /Data/sqlData.sql

COPY changeUser.sql /Data/changeUser.sql

CMD ["sh", "/Data/setup.sh"]

#### setup.sh

(shell脚本 内容为容器启动时执行的命令，且脚本已复制到容器/Data/文件夹下 )

#!/bin/bash

echo `service mysql status`

echo '1 Service Start.....'

service mysql start

sleep 3

echo `service mysql status`

echo '2.In data....'

mysql< /Data/schema.sql

sleep 3

echo '2.In user docker 123456....'

mysql< /Data/changeUser.sql

echo '3 ok.....'

sleep 3

echo `service mysql status`

echo '4 ok'

tail -f /dev/null

注意点：

sleep 3   脚本执行延迟三秒。

echo ' '  可将' '之间内容打印输出  echo ``  可将` ` 之间命令的结果返回（注意' ' 和 `` 的不同 ）。

service mysql start 启动容器中mysql服务。

mysql< /Data/schema.sql     命令是调用mysql命令去执行sql文件

#### schema.sql

(此文件用来初始化数据库，包括建库，建表插入数据)

create database `mydocker` default character set utf8 collate utf8\_general\_ci;

use docker;

#### changeUser.sql

use mysql;

select host, user from user;

create user myself identified by '123123';

grant all on mydocker.\* to myself@'%' identified by '123123' with grant option;

flush privileges;

为了初始化数据，我们之前设置了，免密登陆数据可，所以这个sql用来为sql设置一个myself用户 密码123123 并且将之前建立的mydocker数据库权限给myself用户。

### 创建镜像

docker build -t my/docker-mysql /path/to/dockerfile

### 启动容器

docker run -i -t -p 3306:3306 my/docker-mysql

### 注

#### Mysql授权相关

##### 授权

grant all on mydocker.\* to myself@'%' identified by '123123' with grant option;

flush privileges;

解释说明：

grant 权限1,权限2,…权限n on 数据库名称.表名称 to 用户名@用户地址 identified by ‘连接口令’;

权限1,权限2,…权限n代表select,insert,update,delete,create,drop,index,alter,grant,references,reload,shutdown,process,file等14个权限。

当权限1,权限2,…权限n被all privileges或者all代替，表示赋予用户全部权限。

当数据库名称.表名称被\*.\*代替，表示赋予用户操作服务器上所有数据库所有表的权限。

用户地址可以是localhost，也可以是ip地址、机器名字、域名。也可以用’%'表示从任何地址连接。

‘连接口令’不能为空，连接口令也就是该用户密码，否则创建失败。

该语句将会给指定用户授权，如若该用户还未在数据库用户中创建，则自动创建该用户并授权给该用户指定权限。

参考例子：

grant select,insert,update,delete,create,drop on vtdc.employee to joe@10.163.225.87 identified by ‘123′;

给来自10.163.225.87的用户joe分配可对数据库vtdc的employee表进行select,insert,update,delete,create,drop等操作的权限，并设定口令为123。

#### 手动创建新用户

需要有mysql里的root权限的

执行以下语句

insert into mysql.user(Host,User,Password) values("localhost"," testuser1 ",password("1234"));

flush privileges;

这样就创建了一个名为：testuser1 密码为：1234 的用户

#### 修改指定用户密码

需要有mysql里的root权限的

执行以下语句

update mysql.user set password=password('123456789') where User="testuser1 " and Host="localhost";

flush privileges;

#### 删除用户

需要有mysql里的root权限的

执行以下语句

DELETE FROM user WHERE User=" testuser1" and Host="localhost";

flush privileges;

## ## nginx

### ### 概念

Nginx（发音为“ engine-x”）是用于HTTP，HTTPS，SMTP，POP3和IMAP协议的开源反向代理服务器，以及负载平衡器，HTTP缓存和Web服务器（原始服务器）。 Nginx项目一开始就非常注重高并发，高性能和低内存使用。 它获得了两节式BSD许可，并在Linux，BSD变体，Mac OS X，Solaris，AIX，HP-UX以及其他\* nix版本上运行。 它还具有用于Microsoft Windows的概念证明端口。

### ### 使用

#### 方式1

$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx

#### 方式2

或者，可以使用简单的Dockerfile生成包含必要内容的新镜像（与上面的绑定安装相比，这是一种更干净的解决方案）：

FROM nginx

COPY static-html-directory /usr/share/nginx/html

将此文件放置在与目录相同的目录（“ static-html-directory”）中，运行docker build -t some-content-nginx，然后启动容器：

$ docker run --name some-nginx -d some-content-nginx

### ### 暴露外部端口

$ docker run --name some-nginx -d -p 8080:80 some-content-nginx

然后便可以通过浏览器输入http://localhost:8080 or http://host-ip:8080 进行访问。

### ### 进行自定义配置

$ docker run --name my-custom-nginx-container -v /host/path/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx

有关nginx配置文件的语法的信息，请参见官方文档（特别是《入门指南》）。

<http://nginx.org/en/docs/>

如果您希望采用默认配置，请使用以下类似内容从运行的nginx容器中复制它：

$ docker run --name tmp-nginx-container -d nginx

$ docker cp tmp-nginx-container:/etc/nginx/nginx.conf /host/path/nginx.conf

$ docker rm -f tmp-nginx-container

也可以使用简单的Dockerfile（在/ host / path /中）更干净地完成此操作：

FROM nginx

COPY nginx.conf /etc/nginx/nginx.conf

如果您在Dockerfile中添加了自定义CMD，请确保包括-g守护程序off； 为了使nginx停留在前台，以便Docker可以正确跟踪进程（否则，您的容器将在启动后立即停止）！

然后使用docker build -t custom-nginx构建镜像。 并如下运行：

$ docker run --name my-custom-nginx-container -d custom-nginx

#### 在Nginx配置中使用环境变量

现成的nginx不支持大多数配置块中的环境变量。但是，envsubst如果您需要在nginx启动之前动态生成nginx配置，则可以将其用作解决方法。

这是使用docker-compose.yml的示例：

web:

image: nginx

volumes:

- ./mysite.template:/etc/nginx/conf.d/mysite.template

ports:

- "8080:80"

environment:

- NGINX\_HOST=foobar.com

- NGINX\_PORT=80

command: /bin/bash -c "envsubst < /etc/nginx/conf.d/mysite.template > /etc/nginx/conf.d/default.conf && exec nginx -g 'daemon off;'"

mysite.template然后，该文件可能包含如下变量引用：

listen ${NGINX\_PORT};

#### 在只读模式下运行Nginx

要以只读模式运行nginx，您需要将Docker卷安装到nginx写入信息的每个位置。默认的nginx配置需要对/var/cache和的写入权限/var/run。可以通过如下运行nginx轻松完成此操作：

$ docker run -d -p 80:80 --read-only -v $(pwd)/nginx-cache:/var/cache/nginx -v $(pwd)/nginx-pid:/var/run nginx

如果您有一个更高级的配置，需要nginx写入其他位置，只需将更多的卷安装添加到这些位置。

#### 在调试模式下运行Nginx

自1.9.8版起的镜像带有nginx-debug二进制文件，当使用较高的日志级别时，该文件会产生详细的输出。它可以与简单的CMD替换一起使用：

$ docker run --name my-nginx -v /host/path/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx nginx-debug -g 'daemon off;'

docker-compose.yml中的类似配置如下所示：

web:

image: nginx

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf:ro

command: [nginx-debug, '-g', 'daemon off;']

#### 用户和组ID

从1.17.0开始，基于高山和基于debian的镜像变体都使用相同的用户名和组ID来删除工作进程的特权：

$ id

uid=101(nginx) gid=101(nginx) groups=101(nginx)

#### 以非root用户身份运行nginx

可以将镜像作为特权较低的任意UID / GID运行。但是，这需要修改nginx配置，以使用该特定UID / GID对可写的目录：

$ docker run -d -v $PWD/nginx.conf:/etc/nginx/nginx.conf nginx

当前目录中的nginx.conf应该重新定义以下指令：

pid /tmp/nginx.pid;

并在http上下文中：

http {

client\_body\_temp\_path /tmp/client\_temp;

proxy\_temp\_path /tmp/proxy\_temp\_path;

fastcgi\_temp\_path /tmp/fastcgi\_temp;

uwsgi\_temp\_path /tmp/uwsgi\_temp;

scgi\_temp\_path /tmp/scgi\_temp;

...

}

#### 使用Amplify监控Nginx

Amplify是一个免费的监视工具，可用于监视基于nginx的微服务架构。Amplify由Nginx软件背后的公司开发和维护。

借助Amplify，可以跨容器收集和汇总指标，并提供关键绩效数据（如活动连接或每秒请求）的连贯可视化集。快速检查任何性能下降，流量异常，并更深入地了解Nginx配置也很容易。

### ### 镜像版本及变体

该nginx镜像有许多种，每一个设计用于特定的使用情况。

nginx:<version>

这是事实上的镜像。如果不确定自己的需求，则可能要使用这一需求。它既可以用作一次性容器（安装源代码并启动容器以启动应用程序），也可以用作构建其他镜像的基础。

nginx:<version>-alpine

该镜像是基于流行的高山Linux项目，可用的alpine官方镜像。Alpine Linux比大多数分发基础镜像（〜5MB）小得多，因此通常导致更苗条的镜像。

当希望最终镜像尺寸尽可能小时，强烈建议使用此变体。需要注意的主要警告是，它确实使用了musl libc而不是glibc和friends，因此某些软件可能会遇到问题，具体取决于其libc要求的深度。但是，大多数软件对此没有问题，因此，此变体通常是非常安全的选择。

#### 版本

1.17.6, mainline, 1, 1.17, latest

1.17.6-perl, mainline-perl, 1-perl, 1.17-perl, perl

1.17.6-alpine, mainline-alpine, 1-alpine, 1.17-alpine, alpine

1.17.6-alpine-perl, mainline-alpine-perl, 1-alpine-perl, 1.17-alpine-perl, alpine-perl

1.16.1, stable, 1.16

1.16.1-perl, stable-perl, 1.16-perl

1.16.1-alpine, stable-alpine, 1.16-alpine

1.16.1-alpine-perl, stable-alpine-perl, 1.16-alpine-perl

# # 7 Dockerfile

Docker可以通过阅读Docker的指令来自动构建镜像 Dockerfile。A Dockerfile是一个文本文档，其中包含用户可以在命令行上调用以组装镜像的所有命令。使用docker build 用户可以创建自动构建，该构建连续执行多个命令行指令。

## ## 用法

该docker build命令根据Dockerfile和上下文构建镜像。构建的上下文是指定位置PATH或的文件集URL。这PATH是本地文件系统上的目录。该URL是一个Git仓库的位置。

上下文是递归处理的。正由于递归处理的原因，请不要将Dockerfile放在服务器根目录之下执行，否则将会递归处理整个服务器的所有文件位置。 因此，a PATH包括任何子目录，并且a 包括URL存储库及其子模块。此示例显示了一个使用当前目录作为上下文的构建命令：

$ docker build .

Sending build context to Docker daemon 6.51 MB

...

构建是由Docker守护程序而不是CLI运行的。构建过程要做的第一件事是将整个上下文（递归）发送到守护程序。在大多数情况下，最好以空目录作为上下文，并将Dockerfile保留在该目录中。仅添加构建Dockerfile所需的文件。

警告：不要用你的根目录下，/作为PATH因为它会导致生成到您的硬盘驱动器的全部内容传输到码头工人守护进程。

传统上，Dockerfile称为Dockerfile并位于上下文的根中。您可以使用-f标志with docker build指向文件系统中任何位置的Dockerfile。

$ docker build -f /path/to/a/Dockerfile .

如果构建成功，则可以指定一个存储库和标记，用于在其中存储新镜像：

$ docker build -t shykes/myapp .

要在构建后将镜像标记到多个存储库中，请在-t运行build命令时添加多个参数：

$ docker build -t shykes/myapp:1.0.2 -t shykes/myapp:latest .

在Docker守护程序运行中的指令之前Dockerfile，它会对进行初步验证，Dockerfile如果语法不正确，则返回错误：

$ docker build -t test/myapp .

Sending build context to Docker daemon 2.048 kB

Error response from daemon: Unknown instruction: RUNCMD

Docker守护程序以Dockerfile一对一的方式运行指令，如有必要，将每个指令的结果提交到新镜像，然后最终输出新镜像的ID。Docker守护程序将自动清理您发送的上下文。

请注意，每条指令都是独立运行的，并会导致创建新镜像-因此RUN cd /tmp对下一条指令不会有任何影响。

Docker将尽可能重用中间镜像（缓存），以docker build显着加速该过程。这由Using cache控制台输出中的消息指示。

构建缓存仅用于具有本地父链的镜像。这意味着这些镜像是由以前的版本创建的，或者整个镜像链都已加载docker load。如果您希望使用特定镜像的构建缓存，则可以使用--cache-from选项指定它。指定带有的镜像 --cache-from不需要具有父链，并且可以从其他注册表中提取。

完成构建后，就可以考虑将存储库推送到其注册表了。

## ## 构建工具

从版本18.09开始，Docker支持由moby / buildkit 项目提供的用于执行构建的新后端。与旧的实现相比，BuildKit后端提供了许多好处。例如，BuildKit可以：

检测并跳过执行未使用的构建阶段

并行构建独立构建阶段

两次构建之间仅增量传输构建上下文中的更改文件

在构建上下文中检测并跳过传输未使用的文件

使用具有许多新功能的外部Dockerfile实现

避免其他API的副作用（中间镜像和容器）

优先考虑构建缓存以进行自动修剪

要使用BuildKit后端，您需要DOCKER\_BUILDKIT=1在CLI上设置环境变量 ，然后再调用docker build。

## ## 格式

这是的格式Dockerfile：

# Comment

INSTRUCTION arguments

该指令不区分大小写。但是，约定是大写，以便更轻松地将它们与参数区分开。

Docker Dockerfile按顺序运行指令。一个Dockerfile 必须用`FROM`指令开始。这可能在解析器指令，注释和全局范围的 ARG之后。该FROM指令指定要从中构建父镜像。FROM 只能在一个或多个ARG指令之前，这些指令声明在中的FROM行中使用的参数Dockerfile。

该码头工人对待线开始以#作为注释，除非该行是一个有效的解析器指令。#一行中其他任何地方的标记都被视为参数。这允许如下语句：

# Comment

RUN echo 'we are running some # of cool things'

注释中不支持换行符。

## ## 解析器指令

解析器指令是可选的，并且会影响Dockerfile处理a 中后续行的方式。解析器指令不会在构建中添加图层，也不会显示为构建步骤。解析器指令以形式写为特殊类型的注释# directive=value。单个指令只能使用一次。

处理完注释，空行或生成器指令后，Docker不再寻找解析器指令。而是将格式化为解析器指令的任何内容都视为注释，并且不会尝试验证它是否可能是解析器指令。因此，所有解析器指令必须位于的顶部Dockerfile。

解析器指令不区分大小写。但是，约定是小写的。约定还应在任何解析器指令之后包含一个空白行。解析器伪指令不支持行继续字符。

由于这些规则，以下示例均无效：

由于行继续而无效：

# direc \

tive=value

由于出现两次而无效：

# directive=value1

# directive=value2

FROM ImageName

由于出现在构建器指令之后而被视为注释：

FROM ImageName

# directive=value

由于出现在不是解析器指令的注释之后，因此被视为注释：

# About my dockerfile

# directive=value

FROM ImageName

由于未被识别，未知指令被视为注释。另外，由于在非语法分析器指令的注释之后出现，因此已知指令被视为注释。

# unknowndirective=value

# knowndirective=value

解析器指令中允许非换行空格。因此，以下各行都被相同地对待：

#directive=value

# directive =value

# directive= value

# directive = value

# dIrEcTiVe=value

支持以下解析器指令：

syntax

escape

### 句法

# syntax=[remote image reference]

例如：

# syntax=docker/dockerfile

# syntax=docker/dockerfile:1.0

# syntax=docker.io/docker/dockerfile:1

# syntax=docker/dockerfile:1.0.0-experimental

# syntax=example.com/user/repo:tag@sha256:abcdef...

仅当使用BuildKit后端时才启用此功能。

语法指令定义用于构建当前Dockerfile的Dockerfile构建器的位置。BuildKit后端允许无缝使用构建器的外部实现，这些构建器以Docker镜像的形式分发并在容器沙箱环境中执行。

自定义Dockerfile实现使您能够：

自动获取错误修正，而无需更新守护程序

确保所有用户都使用相同的实现来构建您的Dockerfile

使用最新功能而不更新守护程序

试用新的实验性或第三方功能

#### 正式发布

Docker分发了镜像的正式版本，这些镜像可用于docker/dockerfile在Docker Hub上的存储库下构建Dockerfile 。有两个发布新镜像的渠道：稳定版和实验版。

稳定的通道遵循语义版本控制。例如：

docker / dockerfile：1.0.0-仅允许不可变版本1.0.0

docker / dockerfile：1.0-允许版本1.0。\*

docker / dockerfile：1-允许版本1 。

docker / dockerfile：latest-稳定通道上的最新版本

在发布之时，实验频道使用稳定版本中主要和次要组件的增量版本控制。例如：

docker / dockerfile：1.0.1-experimental-仅允许不可变版本1.0.1-experimental

docker / dockerfile：1.0-experimental-1.0之后的最新实验版本

docker / dockerfile：experimental-实验频道上的最新版本

您应该选择最适合自己需求的渠道。如果您只想修正错误，则应使用docker/dockerfile:1.0。

### 逃逸

# escape=\ (backslash)

要么

# escape=` (backtick)

该escape指令设置用来逃避的字符的字符 Dockerfile。如果未指定，则默认转义字符为\。

转义字符用于转义一行中的字符和转义换行符。这允许一条Dockerfile指令跨越多行。请注意，无论escapeparser指令是否包含在Dockerfile，都不会在RUN命令中执行转义，除非在行末。

将转义字符设置`为尤其有用 Windows，其中\目录路径分隔符为。`与Windows PowerShell一致。

考虑以下示例，该示例将以非显而易见的方式失败 Windows。第二\，在第二行的端部将被解释为用于换行的逃逸，而不是从第一逸出的目标\。类似地，\假设第三行的末尾实际上被当作一条指令来处理，则将其视为行的继续。该dockerfile的结果是第二和第三行被视为一条指令：

FROM microsoft/nanoserver

COPY testfile.txt c:\\

RUN dir c:\

PS C:\John> docker build -t cmd .

Sending build context to Docker daemon 3.072 kB

Step 1/2 : FROM microsoft/nanoserver

---> 22738ff49c6d

Step 2/2 : COPY testfile.txt c:\RUN dir c:

GetFileAttributesEx c:RUN: The system cannot find the file specified.

PS C:\John>

上述解决方案之一是将 和和/用作目标。但是，此语法充其量是令人困惑的，因为它对上的路径而言并不自然，而在最坏的情况下，由于并非所有支持的命令都作为路径分隔符，因此容易出错 。COPYdirWindowsWindows/

通过添加escape解析器指令，通过对以下Dockerfile文件路径使用自然平台语义，以下操作可以按预期成功Windows：

# escape=`

FROM microsoft/nanoserver

COPY testfile.txt c:\

RUN dir c:\

PS C:\John> docker build -t succeeds --no-cache=true .

Sending build context to Docker daemon 3.072 kB

Step 1/3 : FROM microsoft/nanoserver

---> 22738ff49c6d

Step 2/3 : COPY testfile.txt c:\

---> 96655de338de

Removing intermediate container 4db9acbb1682

Step 3/3 : RUN dir c:\

---> Running in a2c157f842f5

## ## 更换环境

环境变量（与声明的ENV声明），也可以在特定指令作为变量用来被解释 Dockerfile。转义也可以通过在字面上将类似变量的语法包含到语句中来处理。

环境变量Dockerfile用 $variable\_name或表示${variable\_name}。它们被同等对待，并且大括号语法通常用于解决变量名不带空格的问题，例如${foo}\_bar。

该${variable\_name}语法还支持bash 以下指定的一些标准修饰符：

${variable:-word}表示如果variable设置，则结果将是该值。如果variable未设置，则为word结果。

${variable:+word}指示如果variable设置了if，则将为word结果，否则结果为空字符串。

在所有情况下，word都可以是任何字符串，包括其他环境变量。

可以通过\在变量前添加a来进行转义：例如，\$foo或\${foo}将分别转换为$foo和${foo}文字。

示例（解析的表示形式显示在之后#）：

FROM busybox

ENV foo /bar

WORKDIR ${foo} # WORKDIR /bar

ADD . $foo # ADD . /bar

COPY \$foo /quux # COPY $foo /quux

以下变量中的指令列表支持环境变量Dockerfile：

* ADD
* COPY
* ENV
* EXPOSE
* FROM
* LABEL
* STOPSIGNAL
* USER
* VOLUME
* WORKDIR

以及：

ONBUILD （当与以上支持的说明之一结合使用时）

注：1.4之前，ONBUILD说明没有不支持的环境变量，即使与任何上面列出的指令结合。

在整个指令中，环境变量替换将对每个变量使用相同的值。换句话说，在此示例中：

ENV abc=hello

ENV abc=bye def=$abc

ENV ghi=$abc

将导致def值为hello，而不是bye。但是， ghi将具有的值，bye因为它不是设置abc为的同一指令的一部分bye。

## ## .dockerignore文件

在docker CLI将上下文发送到docker守护程序之前，它会.dockerignore在上下文的根目录中查找名为的文件。如果此文件存在，则CLI会修改上下文以排除与其中的模式匹配的文件和目录。这有助于避免不必要地将大型文件或敏感文件和目录发送到守护程序，并避免使用ADD或将它们添加到镜像中COPY。

CLI将.dockerignore文件解释为以换行符分隔的模式列表，类似于Unix shell的文件组。为了匹配，上下文的根被认为是工作目录和根目录。例如，图案 /foo/bar和foo/bar两个排除命名的文件或目录bar 的foo子目录PATH或位于Git仓库的根URL。两者都不排除其他任何东西。

如果.dockerignore文件中的一行以第#1列开头，则该行将被视为注释，并在CLI解释之前将其忽略。

这是一个示例.dockerignore文件：

# comment

\*/temp\*

\*/\*/temp\*

temp?

此文件导致以下生成行为：

| **规则** | **行为** |
| --- | --- |
| # comment | 忽略了。 |
| \*/temp\* | 排除名称以temp根的任何直接子目录开头的文件和目录。例如，/somedir/temporary.txt排除纯文件，排除目录/somedir/temp。 |
| \*/\*/temp\* | 排除temp从根以下两个级别的任何子目录开始的文件和目录。例如，/somedir/subdir/temporary.txt被排除。 |
| temp? | 排除根目录中名称为的一个字符扩展名的文件和目录temp。例如，/tempa和/tempb被排除。 |

使用Go的filepath.Match规则进行匹配 。预处理步骤使用Go的filepath.Clean删除开头和结尾的空格，并消除.和..元素 。预处理后空白的行将被忽略。

除了Go的filepath.Match规则外，Docker还支持一个特殊的通配符字符串\*\*，该字符串匹配任意数量的目录（包括零个）。例如，\*\*/\*.go将排除.go 在所有目录（包括构建上下文的根目录）中找到的所有以结尾结尾的文件。

以!（感叹号）开头的行可用于排除例外。以下是.dockerignore使用此机制的示例文件：

\*.md

!README.md

除 README.md上下文外，所有降价文件都排除在外。

!异常规则的位置会影响行为：.dockerignore与特定文件匹配的的最后一行确定是包含还是排除该文件。考虑以下示例：

\*.md

!README\*.md

README-secret.md

除了以外的自述文件，上下文中没有markdown文件 README-secret.md。

现在考虑以下示例：

\*.md

README-secret.md

!README\*.md

包括所有自述文件。中线没有作用，因为 !README\*.md匹配README-secret.md并排在最后。

您甚至可以使用该.dockerignore文件排除Dockerfile 和.dockerignore文件。这些文件仍被发送到守护程序，因为它需要它们来完成其工作。但是ADD和COPY指令不会将它们复制到镜像中。

最 后，您可能要指定要包含在上下文中的文件，而不是要排除的文件。为此，请指定\*为第一个模式，然后指定一个或多个!异常模式。

注意：由于历史原因，该模式将.被忽略。

## ## FROM

FROM <image> [AS <name>]

要么

FROM <image>[:<tag>] [AS <name>]

要么

FROM <image>[@<digest>] [AS <name>]

该FROM指令初始化一个新的构建阶段，并为后续指令设置 基本镜像。因此，有效的Dockerfile必须从FROM指令开始。该镜像可以是任何有效的镜像– 从公共存储库中拉出镜像特别容易。

ARG是先于仅指示FROM在Dockerfile。请参阅了解ARG和FROM之间的相互作用。

FROM可以一次出现多次Dockerfile以创建多个镜像，也可以将一个构建阶段作为对另一个构建阶段的依赖。只需在每个新FROM指令之前记录一次提交输出的最后一个镜像ID 。每个FROM指令清除由先前指令创建的任何状态。

可选的名称可以通过添加给予一个新的构建阶段AS name的 FROM指令。该名称可用于后续版本FROM和 COPY --from=<name|index>说明中，以引用此阶段中构建的镜像。

该tag或digest值是可选的。如果您忽略其中任何一个latest，那么缺省情况下构建器将采用标签。如果构建器找不到该tag值，则返回错误。

### 了解ARG和FROM之间的交互方式

FROM指令支持由ARG 第一指令之前的任何指令声明的变量FROM。

ARG CODE\_VERSION=latest

FROM base:${CODE\_VERSION}

CMD /code/run-app

FROM extras:${CODE\_VERSION}

CMD /code/run-extras

ARG在a之前的声明FROM位于构建阶段之外，因此不能在以后的任何指令中使用它FROM。要ARG在第一次FROM使用声明之前使用默认值，请在ARG构建阶段使用不带值的指令：

ARG VERSION=latest

FROM busybox:$VERSION

ARG VERSION

RUN echo $VERSION > image\_version

## ## RUN

RUN有2种形式：

RUN <command>（shell形式，命令在shell中运行，默认情况下/bin/sh -c在Linux或cmd /S /CWindows 上运行）

RUN ["executable", "param1", "param2"]（执行表格）

该RUN指令将在当前镜像顶部的新层中执行所有命令，并提交结果。生成的提交镜像将用于中的下一步Dockerfile。

分层RUN指令和生成提交符合Docker的核心概念，在Docker上，提交是廉价的，并且可以从镜像历史记录的任何位置创建容器，就像源代码控制一样。

在EXEC形式使得能够避免壳串改写（munging），并RUN 使用不包含指定壳可执行基本镜像的命令。

可以使用以下 命令更改外壳程序表单的默认外壳程序SHELL。

在shell形式中，您可以使用\（反斜杠）将一条RUN指令继续到下一行。例如，考虑以下两行：

RUN /bin/bash -c 'source $HOME/.bashrc; \

echo $HOME'

它们在一起等效于以下这一行：

RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'

注意：要使用'/ bin / sh'以外的其他shell，请使用exec形式传入所需的shell。例如， RUN ["/bin/bash", "-c", "echo hello"]

注意：exec表单被解析为JSON数组，这意味着您必须在单词而非单引号（'）周围使用双引号（“）。

注意：与shell表单不同，exec表单不会调用命令shell。这意味着正常的外壳处理不会发生。例如， RUN [ "echo", "$HOME" ]将不会对进行变量替换$HOME。如果要进行shell处理，则可以使用shell形式或直接执行shell，例如：RUN [ "sh", "-c", "echo $HOME" ]。当使用exec表单并直接执行shell时（例如在shell表单中），是由shell进行环境变量扩展，而不是docker。

注意：在JSON格式中，必须转义反斜杠。在Windows中，反斜杠是路径分隔符，这一点尤其重要。否则，由于无效的JSON，以下行将被视为shell形式，并以意外的方式失败： RUN ["c:\windows\system32\tasklist.exe"] 此示例的正确语法是： RUN ["c:\\windows\\system32\\tasklist.exe"]

RUN下一次构建期间，指令缓存不会自动失效。类似指令的缓存 RUN apt-get dist-upgrade -y将在下一次构建中重用。RUN指令的缓存可以通过使用--no-cache 标志来使无效，例如docker build --no-cache。

有关更多信息，请参见Dockerfile最佳实践指南。

RUN指令的高速缓存可以被ADD指令无效。有关详情，请参见 下文。

## ## CMD

该CMD指令具有三种形式：

CMD ["executable","param1","param2"]（exec形式，这是首选形式）

CMD ["param1","param2"]（作为ENTRYPOINT的默认参数）

CMD command param1 param2（外壳形式）

CMD指令中只能有一条指令Dockerfile。如果您列出多个，CMD 则只有最后一个CMD才会生效。

一个的主要目的CMD是为执行中的容器提供默认值。这些默认值可以包含可执行文件，也可以省略可执行文件，在这种情况下，您还必须指定一条ENTRYPOINT 指令。

注意：如果CMD用于提供ENTRYPOINT 指令的默认参数，则CMD和ENTRYPOINT指令均应使用JSON数组格式指定。

注意：exec表单被解析为JSON数组，这意味着您必须在单词而非单引号（'）周围使用双引号（“）。

注意：与shell表单不同，exec表单不会调用命令shell。这意味着正常的外壳处理不会发生。例如， CMD [ "echo", "$HOME" ]将不会对进行变量替换$HOME。如果要进行shell处理，则可以使用shell形式或直接执行shell，例如：CMD [ "sh", "-c", "echo $HOME" ]。当使用exec表单并直接执行shell时（例如在shell表单中），是由shell进行环境变量扩展，而不是docker。

当用于shell或exec格式时，该CMD指令设置运行镜像时要执行的命令。

如果您使用的shell形式CMD，则将在中<command>执行 /bin/sh -c：

FROM ubuntu

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

如果要在 <command> 没有外壳的情况下运行，则必须将命令表示为JSON数组，并提供可执行文件的完整路径。 此数组形式是的首选格式CMD。任何其他参数必须在数组中分别表示为字符串：

FROM ubuntu

CMD ["/usr/bin/wc","--help"]

如果您希望容器每次都运行相同的可执行文件，则应考虑ENTRYPOINT与结合使用CMD。请参阅 ENTRYPOINT。

如果用户指定的参数，docker run则它们将覆盖中指定的默认参数CMD。

注意：请勿RUN与混淆CMD。RUN实际上运行命令并提交结果；CMD在生成时不执行任何操作，但指定镜像的预期命令。

## ## LABEL

LABEL <key>=<value> <key>=<value> <key>=<value> ...

该LABEL指令将元数据添加到镜像。A LABEL是键值对。要在LABEL值中包含空格，请像在命令行分析中一样使用引号和反斜杠。一些用法示例：

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."

一幅镜像可以有多个标签。您可以在一行上指定多个标签。在Docker 1.10之前的版本中，这减小了最终镜像的大小，但是情况不再如此。您仍然可以选择通过以下两种方式之一在一条指令中指定多个标签：

LABEL multi.label1="value1" multi.label2="value2" other="value3"

LABEL multi.label1="value1" \

multi.label2="value2" \

other="value3"

基本或父镜像（该FROM行中的镜像）中包含的标签由您的镜像继承。如果标签已经存在但具有不同的值，则最近应用的值将覆盖任何先前设置的值。

要查看镜像的标签，请使用docker inspect命令。

"Labels": {

"com.example.vendor": "ACME Incorporated"

"com.example.label-with-value": "foo",

"version": "1.0",

"description": "This text illustrates that label-values can span multiple lines.",

"multi.label1": "value1",

"multi.label2": "value2",

"other": "value3"

},

## ## MAINTAINER（已弃用）

MAINTAINER <name>

该MAINTAINER指令设置生成镜像的作者字段。该LABEL指令是此版本的灵活得多，您应该改用它，因为它可以设置所需的任何元数据，并且可以轻松查看，例如使用docker inspect。要设置与该MAINTAINER字段相对应的标签， 可以使用：

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

这样docker inspect，其他标签就可以显示出来了。

## ## EXPOSE

EXPOSE <port> [<port>/<protocol>...]

该EXPOSE指令通知Docker容器在运行时监听指定的网络端口。您可以指定端口是侦听TCP还是UDP，如果未指定协议，则默认值为TCP。

该EXPOSE指令实际上并未发布端口。它充当构建镜像的人员和运行容器的人员之间的一种文档类型，有关打算发布哪些端口的信息。要在运行容器时实际发布端口，请使用-p标记on docker run 来发布和映射一个或多个端口，或者使用-P标记来发布所有公开的端口并将它们映射到高阶端口。

默认情况下，EXPOSE假定为TCP。您还可以指定UDP：

EXPOSE 80/udp

要同时在TCP和UDP上公开，请包括以下两行：

EXPOSE 80/tcp

EXPOSE 80/udp

在这种情况下，如果-P与配合使用docker run，则该端口仅对TCP公开一次，对于UDP公开一次。请记住，-P该端口在主机上使用临时的高阶主机端口，因此该端口对于TCP和UDP将是不同的。

无论EXPOSE设置如何，都可以在运行时使用该-p标志覆盖它们。例如

docker run -p 80:80/tcp -p 80:80/udp ...

要在主机系统上设置端口重定向，请参阅使用-P标志。该docker network命令支持创建网络以在容器之间进行通信，而无需暴露或发布特定端口，因为连接到网络的容器可以通过任何端口相互通信。有关详细信息，请参阅 此功能概述。

## ## ENV

ENV <key> <value>

ENV <key>=<value> ...

该ENV指令将环境变量<key>设置为value <value>。此值将在构建阶段中所有后续指令的环境中使用，并且在许多情况下也可以内联替换。

该ENV指令有两种形式。第一种形式ENV <key> <value>会将一个变量设置为一个值。第一个空格之后的整个字符串将被视为<value>-包括空格字符。该值将为其他环境变量解释，因此如果不对引号字符进行转义，则将其删除。

第二种形式ENV <key>=<value> ...允许一次设置多个变量。请注意，第二种形式在语法中使用等号（=），而第一种形式则不使用等号（=）。与命令行解析一样，引号和反斜杠可用于在值中包含空格。

例如：

ENV myName="John Doe" myDog=Rex\ The\ Dog \

myCat=fluffy

和

ENV myName John Doe

ENV myDog Rex The Dog

ENV myCat fluffy

将在最终镜像中产生相同的净结果。

ENV从结果镜像运行容器时，使用设置的环境变量将保留。您可以使用查看值docker inspect，并使用更改它们docker run --env <key>=<value>。

注意：环境持久性可能导致意外的副作用。例如，设置ENV DEBIAN\_FRONTEND noninteractive可能会使基于Debian的镜像上的apt-get用户感到困惑。要为单个命令设置值，请使用 RUN <key>=<value> <command>。

## ## ADD

ADD有两种形式：

ADD [--chown=<user>:<group>] <src>... <dest>

ADD [--chown=<user>:<group>] ["<src>",... "<dest>"] （此格式对于包含空格的路径是必需的）

注意：此--chown功能仅在用于构建Linux容器的Dockerfiles上受支持，而在Windows容器上不起作用。由于用户和组所有权概念不会在Linux和Windows之间转换，因此使用/etc/passwd和/etc/group将用户名和组名转换为ID的使用限制了此功能仅对基于Linux OS的容器可用。

该ADD指令从中复制新文件，目录或远程文件URL <src> ，并将它们添加到该路径的镜像文件系统中<dest>。

<src>可以指定多个资源，但是如果它们是文件或目录，则将其路径解释为相对于构建上下文源的路径。

每个都<src>可能包含通配符，并且匹配将使用Go的 filepath.Match规则完成。例如：

ADD hom\* /mydir/ # adds all files starting with "hom"

ADD hom?.txt /mydir/ # ? is replaced with any single character, e.g., "home.txt"

的<dest>是一个绝对路径，或相对于一个路径WORKDIR，到其中的源将在目标容器内进行复制。

ADD test relativeDir/ # adds "test" to `WORKDIR`/relativeDir/

ADD test /absoluteDir/ # adds "test" to /absoluteDir/

添加包含特殊字符（例如[ 和]）的文件或目录时，您需要按照Golang规则转义那些路径，以防止将它们视为匹配模式。例如，要添加名为的文件arr[0].txt，请使用以下命令；

ADD arr[[]0].txt /mydir/ # copy a file named "arr[0].txt" to /mydir/

除非可选--chown标志指定给定的用户名，组名或UID / GID组合以请求对所添加内容的特定所有权，否则所有新文件和目录的UID和GID均为0 。--chown标志的格式允许用户名和组名字符串或直接整数UID和GID任意组合。提供不带组名的用户名或不带GID的UID将使用与GID相同的数字UID。如果提供了用户名或组名，则将使用容器的根文件系统 /etc/passwd和/etc/group文件将名称分别转换为整数UID或GID。以下示例显示了该--chown标志的有效定义：

ADD --chown=55:mygroup files\* /somedir/

ADD --chown=bin files\* /somedir/

ADD --chown=1 files\* /somedir/

ADD --chown=10:11 files\* /somedir/

如果容器根文件系统不包含/etc/passwd或 /etc/group文件，并且在--chown 标志中使用了用户名或组名，则该构建将在该ADD操作上失败。使用数字ID不需要查找，并且不依赖于容器根文件系统内容。

如果<src>是远程文件URL，则目标将具有600的权限。如果要检索的远程文件具有HTTP Last-Modified标头，则该标头中的时间戳将用于设置mtime目标文件上的时间戳。但是，就像在期间处理的任何其他文件一样，在确定文件是否已更改以及是否应更新缓存时ADD，mtime将不包括该文件。

注意：如果通过传递DockerfileSTDIN（docker build - < somefile）进行构建，则没有构建上下文，因此Dockerfile 只能包含基于URL的ADD指令。您也可以通过STDIN：（docker build - < archive.tar.gz）传递压缩的归档文件，归档文件Dockerfile根目录中的，归档文件的其余部分将用作构建的上下文。

注意：如果您的URL文件受身份验证保护，则您将需要使用RUN wget，RUN curl或从容器中使用其他工具，因为该ADD指令不支持身份验证。

注意：ADD如果Dockerfile中的内容<src>已更改，则第一个遇到的指令将使Dockerfile中所有后续指令的缓存无效。这包括使高速缓存中的RUN指令无效。有关更多信息，请参见Dockerfile最佳实践指南。

ADD 遵守以下规则：

该<src>路径必须是内部语境的构建; 您不能这样做ADD ../something /something，因为a的第一步 docker build是将上下文目录（和子目录）发送到docker守护程序。

如果<src>是URL，<dest>并且不以斜杠结尾，则从URL下载文件并将其复制到<dest>。

如果<src>是URL并<dest>以斜杠结尾，则从URL推断文件名，然后将文件下载到 <dest>/<filename>。例如，ADD http://example.com/foobar /将创建文件/foobar。该URL必须具有非平凡的路径，以便在这种情况下可以找到适当的文件名（http://example.com 将不起作用）。

如果<src>是目录，则将复制目录的整个内容，包括文件系统元数据。

注意：目录本身不被复制，仅被复制其内容。

如果<src>是以公认的压缩格式（身份，gzip，bzip2或xz）作为本地 tar归档文件，则将其解压缩为目录。来自远程 URL的资源不会被解压缩。复制或解压缩目录时，其行为与相同tar -x，结果是以下各项的并集：

1. 目标路径上存在的任何内容和
2. 源树的内容，以逐个文件为基础，以“ 2.”为基础解决了冲突。

注意：文件是否被识别为公认的压缩格式仅根据文件的内容而不是文件名来完成。例如，如果一个空文件恰好以.tar.gz该文件结尾，则不会被识别为压缩文件，并且不会生成任何类型的解压缩错误消息，而是会将文件简单地复制到目标位置。

如果<src>是任何其他类型的文件，则将其及其元数据一起单独复制。在这种情况下，如果<dest>以斜杠结尾/，则将其视为目录，并将其内容<src>写入<dest>/base(<src>)。

如果<src>直接或由于使用通配符而指定了多个资源，则该资源<dest>必须是目录，并且必须以斜杠结尾/。

如果<dest>不以斜杠结尾，则将其视为常规文件，并将其内容<src>写入<dest>。

如果<dest>不存在，它将与路径中所有缺少的目录一起创建。

## ## COPY

COPY有两种形式：

COPY [--chown=<user>:<group>] <src>... <dest>

COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] （此格式对于包含空格的路径是必需的）

注意：此--chown功能仅在用于构建Linux容器的Dockerfiles上受支持，而在Windows容器上不起作用。由于用户和组所有权概念不会在Linux和Windows之间转换，因此使用/etc/passwd和/etc/group将用户名和组名转换为ID的使用限制了此功能仅对基于Linux OS的容器可用。

该COPY指令从中复制新文件或目录<src> ，并将它们添加到路径中容器的文件系统中<dest>。

<src>可以指定多个资源，但是文件和目录的路径将被解释为相对于构建上下文的源。

每个都<src>可能包含通配符，并且匹配将使用Go的 filepath.Match规则完成。例如：

COPY hom\* /mydir/ # adds all files starting with "hom"

COPY hom?.txt /mydir/ # ? is replaced with any single character, e.g., "home.txt"

的<dest>是一个绝对路径，或相对于一个路径WORKDIR，到其中的源将在目标容器内进行复制。

COPY test relativeDir/ # adds "test" to `WORKDIR`/relativeDir/

COPY test /absoluteDir/ # adds "test" to /absoluteDir/

复制包含特殊字符（例如[ 和]）的文件或目录时，您需要按照Golang规则转义那些路径，以防止将它们视为匹配模式。例如，要复制名为的文件arr[0].txt，请使用以下命令；

COPY arr[[]0].txt /mydir/ # copy a file named "arr[0].txt" to /mydir/

除非可选--chown标志指定给定的用户名，组名或UID / GID组合，以请求对复制内容的特定所有权，否则所有新文件和目录均使用UID和GID为0创建。--chown标志的格式允许用户名和组名字符串或直接整数UID和GID任意组合。提供不带组名的用户名或不带GID的UID将使用与GID相同的数字UID。如果提供了用户名或组名，则将使用容器的根文件系统 /etc/passwd和/etc/group文件将名称分别转换为整数UID或GID。以下示例显示了该--chown标志的有效定义：

COPY --chown=55:mygroup files\* /somedir/

COPY --chown=bin files\* /somedir/

COPY --chown=1 files\* /somedir/

COPY --chown=10:11 files\* /somedir/

果容器根文件系统不包含/etc/passwd或 /etc/group文件，并且在--chown 标志中使用了用户名或组名，则该构建将在该COPY操作上失败。使用数字ID不需要查找，并且不依赖于容器根文件系统内容。

注意：如果使用STDIN（docker build - < somefile）进行构建，则没有构建上下文，因此COPY无法使用。

（可选）COPY接受一个标志--from=<name|index>，该标志可用于将源位置设置为将使用的先前构建阶段（使用创建FROM .. AS <name>），而不是用户发送的构建上下文。该标志还接受为以FROM指令开头的所有先前构建阶段分配的数字索引 。如果找不到具有指定名称的构建阶段，则尝试使用具有相同名称的镜像代替。

COPY 遵守以下规则：

该<src>路径必须是内部语境的构建; 您不能这样做COPY ../something /something，因为a的第一步 docker build是将上下文目录（和子目录）发送到docker守护程序。

如果<src>是目录，则将复制目录的整个内容，包括文件系统元数据。

注意：目录本身不被复制，仅被复制其内容。

如果<src>是任何其他类型的文件，则将其及其元数据一起单独复制。在这种情况下，如果<dest>以斜杠结尾/，则将其视为目录，并将其内容<src>写入<dest>/base(<src>)。

如果<src>直接或由于使用通配符而指定了多个资源，则该资源<dest>必须是目录，并且必须以斜杠结尾/。

如果<dest>不以斜杠结尾，则将其视为常规文件，并将其内容<src>写入<dest>。

如果<dest>不存在，它将与路径中所有缺少的目录一起创建。

## ## ENTRYPOINT

ENTRYPOINT有两种形式：

ENTRYPOINT ["executable", "param1", "param2"] （执行表格，首选）

ENTRYPOINT command param1 param2 （外壳形式）

An ENTRYPOINT允许您配置将作为可执行文件运行的容器。

例如，以下将使用其默认内容启动nginx，并监听端口80：

docker run -i -t --rm -p 80:80 nginx

命令行参数to docker run <image>将以exec形式附加在所有元素之后ENTRYPOINT，并将覆盖使用指定的所有元素CMD。这允许将参数传递给入口点，即，docker run <image> -d 将-d参数传递给入口点。您可以ENTRYPOINT使用该docker run --entrypoint 标志覆盖指令。

所述壳形式防止任何CMD或run被使用命令行参数，但具有的缺点是你ENTRYPOINT将开始作为的子命令/bin/sh -c，其不通过信号。这意味着可执行文件将不是容器的PID 1-并且将不会接收Unix信号-因此您的可执行文件将不会SIGTERM从接收 到docker stop <container>。

只有中的最后一条ENTRYPOINT指令Dockerfile才会生效。

### ### 执行表单ENTRYPOINT示例

您可以使用exec形式的ENTRYPOINT来设置相当稳定的默认命令和参数，然后使用这两种形式的CMD来设置更可能被更改的其他默认值。

FROM ubuntu

ENTRYPOINT ["top", "-b"]

CMD ["-c"]

运行容器时，可以看到这top是唯一的过程：

$ docker run -it --rm --name test top -H

top - 08:25:00 up 7:27, 0 users, load average: 0.00, 0.01, 0.05

Threads: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.1 us, 0.1 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem: 2056668 total, 1616832 used, 439836 free, 99352 buffers

KiB Swap: 1441840 total, 0 used, 1441840 free. 1324440 cached Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1 root 20 0 19744 2336 2080 R 0.0 0.1 0:00.04 top

要进一步检查结果，可以使用docker exec：

$ docker exec -it test ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 2.6 0.1 19752 2352 ? Ss+ 08:24 0:00 top -b -H

root 7 0.0 0.1 15572 2164 ? R+ 08:25 0:00 ps aux

您也可以使用优雅地请求top关闭docker stop test。

下面Dockerfile显示了使用ENTRYPOINT来在前台运行Apache（即as PID 1）：

FROM debian:stable

RUN apt-get update && apt-get install -y --force-yes apache2

EXPOSE 80 443

VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

如果需要为单个可执行文件编写启动脚本，则可以使用exec和gosu 命令确保最终的可执行文件接收Unix信号：

#!/usr/bin/env bash

set -e

if [ "$1" = 'postgres' ]; then

chown -R postgres "$PGDATA"

if [ -z "$(ls -A "$PGDATA")" ]; then

gosu postgres initdb

fi

exec gosu postgres "$@"

fi

exec "$@"

最后，如果您需要在关闭时进行一些额外的清理（或与其他容器通信），或者协调多个可执行文件，则可能需要确保ENTRYPOINT脚本接收Unix信号，将其传递，然后执行一些更多的工作：

#!/bin/sh

# Note: I've written this using sh so it works in the busybox container too

# USE the trap if you need to also do manual cleanup after the service is stopped,

# or need to start multiple services in the one container

trap "echo TRAPed signal" HUP INT QUIT TERM

# start service in background here

/usr/sbin/apachectl start

echo "[hit enter key to exit] or run 'docker stop <container>'"

read

# stop service and clean up here

echo "stopping apache"

/usr/sbin/apachectl stop

echo "exited $0"

如果使用来运行该镜像docker run -it --rm -p 80:80 --name test apache，则可以使用docker exec或来检查容器的进程docker top，然后要求脚本停止Apache：

$ docker exec -it test ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.1 0.0 4448 692 ? Ss+ 00:42 0:00 /bin/sh /run.sh 123 cmd cmd2

root 19 0.0 0.2 71304 4440 ? Ss 00:42 0:00 /usr/sbin/apache2 -k start

www-data 20 0.2 0.2 360468 6004 ? Sl 00:42 0:00 /usr/sbin/apache2 -k start

www-data 21 0.2 0.2 360468 6000 ? Sl 00:42 0:00 /usr/sbin/apache2 -k start

root 81 0.0 0.1 15572 2140 ? R+ 00:44 0:00 ps aux

$ docker top test

PID USER COMMAND

10035 root {run.sh} /bin/sh /run.sh 123 cmd cmd2

10054 root /usr/sbin/apache2 -k start

10055 33 /usr/sbin/apache2 -k start

10056 33 /usr/sbin/apache2 -k start

$ /usr/bin/time docker stop test

test

real 0m 0.27s

user 0m 0.03s

sys 0m 0.03s

注意：您可以使用覆盖ENTRYPOINT设置--entrypoint，但这只能将二进制文件设置为exec（sh -c将不使用）。

注意：exec表单被解析为JSON数组，这意味着您必须在单词而非单引号（'）周围使用双引号（“）。

注意：与shell表单不同，exec表单不会调用命令shell。这意味着正常的外壳处理不会发生。例如， ENTRYPOINT [ "echo", "$HOME" ]将不会对进行变量替换$HOME。如果要进行shell处理，则可以使用shell形式或直接执行shell，例如：ENTRYPOINT [ "sh", "-c", "echo $HOME" ]。当使用exec表单并直接执行shell时（例如在shell表单中），是由shell进行环境变量扩展，而不是docker。

### ### Shell形式ENTRYPOINT示例

您可以为指定一个纯字符串ENTRYPOINT，它将在中执行/bin/sh -c。这种形式将使用外壳处理来替代外壳环境变量，并且将忽略任何CMD或docker run命令行参数。为了确保能够正确docker stop发出任何长期运行的ENTRYPOINT可执行文件信号，您需要记住以以下命令启动它exec：

FROM ubuntu

ENTRYPOINT exec top -b

运行此镜像时，您将看到单个PID 1过程：

$ docker run -it --rm --name test top

Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached

CPU: 5% usr 0% sys 0% nic 94% idle 0% io 0% irq 0% sirq

Load average: 0.08 0.03 0.05 2/98 6

PID PPID USER STAT VSZ %VSZ %CPU COMMAND

1 0 root R 3164 0% 0% top -b

它将在docker stop以下位置干净地退出：

$ /usr/bin/time docker stop test

test

real 0m 0.20s

user 0m 0.02s

sys 0m 0.04s

如果您忘记添加exec到您的开头ENTRYPOINT：

FROM ubuntu

ENTRYPOINT top -b

CMD --ignored-param1

然后，您可以运行它（为下一步命名）：

$ docker run -it --name test top --ignored-param2

Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238337K cached

CPU: 9% usr 2% sys 0% nic 88% idle 0% io 0% irq 0% sirq

Load average: 0.01 0.02 0.05 2/101 7

PID PPID USER STAT VSZ %VSZ %CPU COMMAND

1 0 root S 3168 0% 0% /bin/sh -c top -b cmd cmd2

7 1 root R 3164 0% 0% top -b

您可以从输出中top看到，指定ENTRYPOINT的不是PID 1。

如果随后运行docker stop test，容器将不会干净退出- 超时后将stop强制命令发送a SIGKILL：

$ docker exec -it test ps aux

PID USER COMMAND

1 root /bin/sh -c top -b cmd cmd2

7 root top -b

8 root ps aux

$ /usr/bin/time docker stop test

test

real 0m 10.19s

user 0m 0.04s

sys 0m 0.03s

### ### 了解CMD和ENTRYPOINT如何相互作用

无论CMD和ENTRYPOINT指令定义运行的容器中时什么命令得到执行。很少有规则描述他们的合作。

Dockerfile应该至少指定CMD或ENTRYPOINT命令之一。

ENTRYPOINT 使用容器作为可执行文件时应定义。

CMD应该用作ENTRYPOINT在容器中定义命令或执行临时命令的默认参数的方式。

CMD 使用替代参数运行容器时，将被覆盖。

下表显示了针对不同ENTRYPOINT/ CMD组合执行的命令：

|  | **没有入口点** | **ENTRYPOINT exec\_entry p1\_entry** | **ENTRYPOINT [“ exec\_entry”，“ p1\_entry”]** |
| --- | --- | --- | --- |
| **没有CMD** | *错误，不允许* | / bin / sh -c exec\_entry p1\_entry | exec\_entry p1\_entry |
| **CMD [“ exec\_cmd”，“ p1\_cmd”]** | exec\_cmd p1\_cmd | / bin / sh -c exec\_entry p1\_entry | exec\_entry p1\_entry exec\_cmd p1\_cmd |
| **CMD [“ p1\_cmd”，“ p2\_cmd”]** | p1\_cmd p2\_cmd | / bin / sh -c exec\_entry p1\_entry | exec\_entry p1\_entry p1\_cmd p2\_cmd |
| **CMD exec\_cmd p1\_cmd** | / bin / sh -c exec\_cmd p1\_cmd | / bin / sh -c exec\_entry p1\_entry | exec\_entry p1\_entry / bin / sh -c exec\_cmd p1\_cmd |

注意：如果CMD是从基础镜像定义的，则设置ENTRYPOINT将重置CMD为空值。在这种情况下，CMD必须在当前镜像中定义一个值。

## ## VOLUME

VOLUME ["/data"]

该VOLUME指令创建具有指定名称的安装点，并将其标记为保存来自本地主机或其他容器的外部安装的卷。该值可以是JSON数组，也可以是VOLUME ["/var/log/"]具有多个参数的纯字符串，例如VOLUME /var/log或VOLUME /var/log /var/db。有关通过Docker客户端的更多信息/示例和安装说明，请参阅 通过Volumes共享目录 。

该docker run命令使用基础镜像内指定位置上存在的任何数据初始化新创建的卷。例如，考虑以下Dockerfile片段：

FROM ubuntu

RUN mkdir /myvol

RUN echo "hello world" > /myvol/greeting

VOLUME /myvol

该Dockerfile生成一个镜像，该镜像导致docker run在处创建一个新的挂载点/myvol并将该greeting文件复制 到新创建的卷中。

### ### 有关指定卷的注意事项

关于卷，请注意以下几点Dockerfile。

基于Windows的容器上的卷：使用基于Windows的容器时，容器内的卷的目的地必须是以下之一：

不存在或空目录

除以下驱动器 C:

从Dockerfile中更改卷：如果在声明了卷之后有任何构建步骤更改了卷中的数据，则这些更改将被丢弃。

JSON格式：列表被解析为JSON数组。您必须用双引号（"）而不是单引号（'）括住单词。

主机目录是在容器运行时声明的：主机目录（挂载点）从本质上说是依赖于主机的。这是为了保留镜像的可移植性，因为不能保证给定的主机目录在所有主机上都可用。因此，您无法从Dockerfile内挂载主机目录。该VOLUME指令不支持指定host-dir 参数。创建或运行容器时，必须指定安装点。

## ## USER

USER <user>[:<group>] or

USER <UID>[:<GID>]

该USER指令设置运行镜像时以及用于任何镜像时使用的用户名（或UID）以及可选的用户组（或GID）RUN，CMD以及在 ENTRYPOINT后面跟随的指令Dockerfile。

警告：当用户没有主要组时，该镜像（或后续说明）将与该root组一起运行。

在Windows上，如果不是内置帐户，则必须首先创建用户。这可以通过net user作为Dockerfile的一部分调用的命令来完成。

FROM microsoft/windowsservercore

# Create Windows user in the container

RUN net user /add patrick

# Set it for subsequent commands

USER patrick

## ## WORKDIR

WORKDIR /path/to/workdir

该WORKDIR指令集的工作目录对任何RUN，CMD， ENTRYPOINT，COPY和ADD它后面的说明Dockerfile。如果WORKDIR不存在，那么即使以后的任何Dockerfile指令中都没有使用它，也将创建它。

该WORKDIR指令可以在中多次使用Dockerfile。如果提供了相对路径，则它将相对于上一条WORKDIR指令的路径 。例如：

WORKDIR /a

WORKDIR b

WORKDIR c

RUN pwd

最终的输出pwd命令这Dockerfile将是 /a/b/c。

该WORKDIR指令可以解析先前使用设置的环境变量 ENV。您只能使用在中显式设置的环境变量Dockerfile。例如：

ENV DIRPATH /path

WORKDIR $DIRPATH/$DIRNAME

RUN pwd

最终pwd命令的输出Dockerfile是 /path/$DIRNAME

## ## ARG

ARG <name>[=<default value>]

该ARG指令定义了一个变量，用户可以在构建时docker build使用带有--build-arg <varname>=<value> 标志的命令将其传递给构建器。如果用户指定了未在Dockerfile中定义的构建参数，则构建会输出警告。

[Warning] One or more build-args [foo] were not consumed.

Dockerfile可能包含一个或多个ARG指令。例如，以下是有效的Dockerfile：

FROM busybox

ARG user1

ARG buildno

...

警告：不建议使用构建时变量来传递诸如github密钥，用户凭据等机密。构建时变量值对于使用该docker history命令的镜像的任何用户都是可见的。

### ### 默认值

的ARG指令可以可选地包括一个默认值：

FROM busybox

ARG user1=someuser

ARG buildno=1

...

如果ARG指令具有缺省值，并且在构建时未传递任何值，那么构建器将使用缺省值。

### ### 范围

一个ARG变量定义进入从在其上在限定的线效果Dockerfile不从参数对命令行或其他地方使用。例如，考虑以下Dockerfile：

1 FROM busybox

2 USER ${user:-some\_user}

3 ARG user

4 USER $user

...

用户通过调用以下命令来构建此文件：

$ docker build --build-arg user=what\_user .

USER在第2行，at some\_user在user变量3上定义为。USER在第4 行，at 定义为what\_useras user，并且what\_user在命令行上传递了值。在通过ARG指令定义变量之前 ，任何对变量的使用都会导致一个空字符串。

的ARG指令在它被定义的构建阶段结束推移的范围进行。要在多个阶段使用arg，每个阶段都必须包含ARG指令。

FROM busybox

ARG SETTINGS

RUN ./run/setup $SETTINGS

FROM busybox

ARG SETTINGS

RUN ./run/other $SETTINGS

### ### 使用ARG变量

您可以使用ARG或ENV指令来指定该RUN指令可用的变量。使用ENV指令定义的环境变量 始终会覆盖ARG同名指令。考虑此Dockerfile和ENVand ARG指令。

1 FROM ubuntu

2 ARG CONT\_IMG\_VER

3 ENV CONT\_IMG\_VER v1.0.0

4 RUN echo $CONT\_IMG\_VER

然后，假定此镜像是使用以下命令构建的：

$ docker build --build-arg CONT\_IMG\_VER=v2.0.1 .

在这种情况下，该RUN指令将使用v1.0.0而不是ARG用户传递的设置：v2.0.1此行为类似于Shell脚本，其中局部作用域的变量从其定义的角度覆盖作为参数传递或从环境继承的变量。

使用上面的示例，但使用不同的ENV规范，可以在ARG和ENV指令之间创建更有用的交互：

1 FROM ubuntu

2 ARG CONT\_IMG\_VER

3 ENV CONT\_IMG\_VER ${CONT\_IMG\_VER:-v1.0.0}

4 RUN echo $CONT\_IMG\_VER

与ARG指令不同，ENV值始终保留在生成的镜像中。考虑不带--build-arg标志的Docker构建：

$ docker build .

使用此Dockerfile示例，CONT\_IMG\_VER它仍然保留在镜像中，但其值将是指令v1.0.0第3行中的默认设置ENV。

在此示例中，变量扩展技术使您可以从命令行传递参数，并利用ENV指令将其保留在最终镜像中 。仅有限的一组Dockerfile指令支持变量扩展。

### ### 预定义的ARG

Docker具有一组预定义ARG变量，您可以在不使用ARGDockerfile中相应指令的情况下使用它们。

* HTTP\_PROXY
* http\_proxy
* HTTPS\_PROXY
* https\_proxy
* FTP\_PROXY
* ftp\_proxy
* NO\_PROXY
* no\_proxy

要使用这些，只需使用以下标志在命令行中传递它们：

--build-arg <varname>=<value>

默认情况下，这些预定义变量从的输出中排除 docker history。排除它们可以降低意外泄露HTTP\_PROXY变量中的敏感身份验证信息的风险。

例如，考虑使用以下命令构建以下Dockerfile

--build-arg HTTP\_PROXY=http://user:pass@proxy.lon.example.com

FROM ubuntu

RUN echo "Hello World"

在这种情况下，HTTP\_PROXY变量的值在中不可用， docker history也不被缓存。如果要更改位置，并且代理服务器已更改为http://user:pass@proxy.sfo.example.com，则后续的构建不会导致高速缓存未命中。

如果您需要覆盖此行为，则可以通过ARG 在Dockerfile中添加如下语句来做到这一点：

FROM ubuntu

ARG HTTP\_PROXY

RUN echo "Hello World"

构建此Dockerfile时，将HTTP\_PROXY保留在中 docker history，并且更改其值会使构建缓存无效。

### ### 全局范围内的自动平台ARG

仅当使用BuildKit后端时，此功能才可用。

Docker ARG在执行构建的节点的平台（构建平台）和结果镜像的平台（目标平台）上用信息预定义了一组变量。可以使用--platform标志on 来指定目标平台docker build。

以下ARG变量是自动设置的：

* TARGETPLATFORM-构建结果的平台。例如linux/amd64，linux/arm/v7，windows/amd64。
* TARGETOS -TARGETPLATFORM的OS组件
* TARGETARCH -TARGETPLATFORM的体系结构组件
* TARGETVARIANT -TARGETPLATFORM的变体组件
* BUILDPLATFORM -执行构建的节点的平台。
* BUILDOS -BUILDPLATFORM的OS组件
* BUILDARCH -BUILDPLATFORM的体系结构组件
* BUILDVARIANT -BUILDPLATFORM的变体组件
* 这些参数是在全局范围内定义的，因此不会在构建阶段或您的RUN命令中自动提供。为了在构建阶段公开这些参数之一，请重新定义它而没有价值。
* 例如：

FROM alpine

ARG TARGETPLATFORM

RUN echo "I'm building for $TARGETPLATFORM"

### ### 对构建缓存的影响

ARG变量不会像ENV变量那样持久化到生成的镜像中。但是，ARG变量确实以类似方式影响构建缓存。如果Dockerfile定义了一个ARG其值不同于先前构建的变量，则首次使用时会发生“缓存未命中”，而不是其定义。特别是，RUN一条指令之后的所有指令都 隐式地ARG使用该ARG变量（作为环境变量），因此可能导致高速缓存未命中。ARG除非。中有匹配的ARG语句，否则所有预定义变量均免于缓存Dockerfile。

例如，考虑以下两个Dockerfile：

1 FROM ubuntu

2 ARG CONT\_IMG\_VER

3 RUN echo $CONT\_IMG\_VER

1 FROM ubuntu

2 ARG CONT\_IMG\_VER

3 RUN echo hello

如果--build-arg CONT\_IMG\_VER=<value>在命令行上指定，则在两种情况下，第2行上的指定都不会导致高速缓存未命中。第3行确实会导致缓存未命中。ARG CONT\_IMG\_VER导致RUN行被标识为与正在运行的CONT\_IMG\_VER=<value>echo hello 相同，因此，如果进行了<value> 更改，则将导致缓存未命中。

考虑同一命令行下的另一个示例：

1 FROM ubuntu

2 ARG CONT\_IMG\_VER

3 ENV CONT\_IMG\_VER $CONT\_IMG\_VER

4 RUN echo $CONT\_IMG\_VER

在此示例中，高速缓存未命中发生在第3行。之所以发生未命中，是因为变量中的变量值ENV引用了该ARG变量，并且该变量是通过命令行更改的。在此示例中，该ENV 命令使镜像包含该值。

如果一条ENV指令覆盖ARG了同名的一条指令，例如Dockerfile：

1 FROM ubuntu

2 ARG CONT\_IMG\_VER

3 ENV CONT\_IMG\_VER hello

4 RUN echo $CONT\_IMG\_VER

第3行不会导致缓存未命中，因为的值CONT\_IMG\_VER是一个常量（hello）。结果，RUN（第4行）使用的环境变量和值在两次构建之间不会改变。

## ## ONBUILD

ONBUILD [INSTRUCTION]

该ONBUILD指令将镜像用作另一个构建的基础时，将在以后的时间向该镜像添加触发指令。触发器将在下游构建的上下文中执行，就像它已FROM在下游指令之后立即插入一样 Dockerfile。

任何构建指令都可以注册为触发器。

如果要构建的镜像将用作构建其他镜像的基础，例如应用程序构建环境或可以使用用户特定配置自定义的守护程序，则此功能很有用。

例如，如果您的镜像是可重用的Python应用程序构建器，则将需要在特定目录中添加应用程序源代码，并且此后可能需要调用构建脚本 。你不能只是打电话ADD和RUN现在，因为你还没有访问应用程序的源代码，这将是为每个应用程序生成不同的。您可以简单地为应用程序开发人员提供Dockerfile可复制粘贴到其应用程序中的样板，但这效率低下，容易出错且难以更新，因为它与特定于应用程序的代码混合在一起。

解决方案是使用ONBUILD注册预先的指令，以便在下一个构建阶段中稍后运行。

运作方式如下：

当遇到ONBUILD指令时，构建器将触发器添加到正在构建的镜像的元数据中。该指令不会影响当前版本。

在构建结束时，所有触发器的列表都存储在镜像清单的key下OnBuild。可以使用docker inspect命令检查它们。

稍后，可以使用该FROM指令将该镜像用作新版本的基础 。作为处理FROM指令的一部分，下游构建器将查找ONBUILD触发器，并以注册时的顺序执行它们。如果任何触发器失败，则该FROM指令将中止，从而导致构建失败。如果所有触发器都成功，则FROM指令将完成，并且构建将照常继续。

执行完触发器后，将从最终镜像中清除触发器。换句话说，它们不是“孙子代”版本所继承的。

例如，您可以添加以下内容：

[...]

ONBUILD ADD . /app/src

ONBUILD RUN /usr/local/bin/python-build --dir /app/src

[...]

警告：不允许ONBUILD使用链接说明ONBUILD ONBUILD。

警告：该ONBUILD指令可能不会触发FROM或执行MAINTAINER。

## ## STOPSIGNAL

STOPSIGNAL signal

该STOPSIGNAL指令设置将被发送到容器退出的系统调用信号。该信号可以是与内核syscall表中的位置匹配的有效无符号数字（例如9），也可以是格式为SIGNAME的信号名称（例如SIGKILL）。

## ## HEALTHCHECK

该HEALTHCHECK指令有两种形式：

HEALTHCHECK [OPTIONS] CMD command （通过在容器内部运行命令来检查容器的运行状况）

HEALTHCHECK NONE （禁用从基本镜像继承的任何运行状况检查）

该HEALTHCHECK指令告诉Docker如何测试容器以检查其是否仍在工作。这样可以检测到诸如Web服务器陷入无限循环并且无法处理新连接的情况，即使服务器进程仍在运行。

当指定了运行状况检查的容器时，除了其正常状态外，它还具有运行状况。此状态最初为starting。只要运行状况检查通过，它将变为healthy（以前处于任何状态）。在一定数量的连续失败之后，它变为unhealthy。

之前可能出现的选项CMD是：

--interval=DURATION（默认值：30s）

--timeout=DURATION（默认值：30s）

--start-period=DURATION（默认值：0s）

--retries=N（默认值：3）

运行状况检查将首先在容器启动后的间隔秒数内运行，然后在之前每次检查完成后的间隔秒数内运行。

如果单次检查花费的时间超过超时秒数，则认为检查失败。

对于要考虑的容器，需要重试连续进行的运行状况检查失败unhealthy。

开始时间段为需要时间进行引导的容器提供了初始化时间。在此期间内的探针故障将不计入最大重试次数。但是，如果运行状况检查在启动期间成功，则认为该容器已启动，并且所有连续失败将计入最大重试次数。

HEALTHCHECKDockerfile中只能有一条指令。如果您列出多个，则只有最后一个HEALTHCHECK才会生效。

CMD关键字后面的命令可以是shell命令（例如HEALTHCHECK CMD /bin/check-running）或exec数组（与其他Dockerfile命令一样；ENTRYPOINT有关详细信息，请参见例如）。

该命令的退出状态指示容器的健康状态。可能的值为：

0：成功-容器健康且可以使用

1：不健康-容器无法正常工作

2：保留-请勿使用此退出代码

例如，要每五分钟检查一次，以便网络服务器能够在三秒钟内为站点的主页提供服务：

HEALTHCHECK --interval=5m --timeout=3s \

CMD curl -f http://localhost/ || exit 1

为了帮助调试失败的探针，命令在stdout或stderr上写入的任何输出文本（UTF-8编码）都将存储在运行状况中，并可以通过查询 docker inspect。此类输出应保持简短（当前仅存储前4096个字节）。

容器的健康状态发生更改时，将health\_status生成具有新状态的事件。

该HEALTHCHECK功能已在Docker 1.12中添加。

## ## SHELL

SHELL ["executable", "parameters"]

该SHELL指令允许覆盖用于命令的shell形式的默认shell 。在Linux上["/bin/sh", "-c"]，默认的shell是，在Windows 上，默认的shell 是["cmd", "/S", "/C"]。该SHELL指令必须以JSON格式编写在Dockerfile中。

该SHELL指令在Windows上特别有用，在Windows上有两个常用且完全不同的本机shell：cmd和powershell，以及可用的替代shell包括sh。

该SHELL说明可以出现多次。每个SHELL指令将覆盖所有先前的SHELL指令，并影响所有后续的指令。例如：

FROM microsoft/windowsservercore

# Executed as cmd /S /C echo default

RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default

RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello

SHELL ["powershell", "-command"]

RUN Write-Host hello

# Executed as cmd /S /C echo hello

SHELL ["cmd", "/S", "/C"]

RUN echo hello

以下说明可以通过影响SHELL指令时， 壳他们的形式在一个Dockerfile使用：RUN，CMD和ENTRYPOINT。

以下示例是Windows上常见的模式，可通过使用SHELL指令进行精简：

...

RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"

...

docker调用的命令将是：

cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"

这效率低下有两个原因。首先，有一个不必要的cmd.exe命令处理器（又名Shell）被调用。其次，shell 形式的每条RUN指令都需要在命令前加上前缀。powershell -command

为了使其更有效，可以采用两种机制之一。一种是使用RUN命令的JSON形式，例如：

...

RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]

...

尽管JSON形式是明确的，并且不使用不必要的cmd.exe，但它确实需要通过双引号和转义来实现更多的详细信息。另一种机制是使用SHELL指令和外壳程序形式，使Windows用户的语法更自然，尤其是与escapeparser指令结合使用时：

# escape=`

FROM microsoft/nanoserver

SHELL ["powershell","-command"]

RUN New-Item -ItemType Directory C:\Example

ADD Execute-MyCmdlet.ps1 c:\example\

RUN c:\example\Execute-MyCmdlet -sample 'hello world'

导致：

PS E:\docker\build\shell> docker build -t shell .

Sending build context to Docker daemon 4.096 kB

Step 1/5 : FROM microsoft/nanoserver

---> 22738ff49c6d

Step 2/5 : SHELL powershell -command

---> Running in 6fcdb6855ae2

---> 6331462d4300

Removing intermediate container 6fcdb6855ae2

Step 3/5 : RUN New-Item -ItemType Directory C:\Example

---> Running in d0eef8386e97

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 10/28/2016 11:26 AM Example

---> 3f2fbf1395d9

Removing intermediate container d0eef8386e97

Step 4/5 : ADD Execute-MyCmdlet.ps1 c:\example\

---> a955b2621c31

Removing intermediate container b825593d39fc

Step 5/5 : RUN c:\example\Execute-MyCmdlet 'hello world'

---> Running in be6d8e63fe75

hello world

---> 8e559e9bf424

Removing intermediate container be6d8e63fe75

Successfully built 8e559e9bf424

PS E:\docker\build\shell>

该SHELL指令还可用于修改外壳的操作方式。例如，SHELL cmd /S /C /V:ON|OFF在Windows上使用，可以修改延迟的环境变量扩展语义。

如果SHELL需要备用shell，例如，和其他zsh，该指令也可以在Linux上使用。cshtcsh

该SHELL功能已在Docker 1.12中添加。

## ## 外部实施功能

仅当使用BuildKit后端时，此功能才可用 。

Docker构建支持实验性功能，例如缓存安装，构建机密和ssh转发，这些功能通过使用带有语法指令的构建器的外部实现启用。

## ## Dockerfile示例

在下面，您可以看到Dockerfile语法的一些示例。如果您对更现实的东西感兴趣，请查看Dockerization示例列表。

# Nginx

#

# VERSION 0.0.1

FROM ubuntu

LABEL Description="This image is used to start the foobar executable" Vendor="ACME Products" Version="1.0"

RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server

# Firefox over VNC

#

# VERSION 0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox

RUN apt-get update && apt-get install -y x11vnc xvfb firefox

RUN mkdir ~/.vnc

# Setup a password

RUN x11vnc -storepasswd 1234 ~/.vnc/passwd

# Autostart firefox (might not be the best way, but it does the trick)

RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900

CMD ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example

#

# VERSION 0.1

FROM ubuntu

RUN echo foo > bar

# Will output something like ===> 907ad6c2736f

FROM ubuntu

RUN echo moo > oink

# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with

# /oink.

## ## 编写Dockerfile的最佳实践

Docker通过从一个Dockerfile文本文件中读取指令来自动构建镜像，该 文本文件按顺序包含构建给定镜像所需的所有命令。

Docker镜像由只读层组成，每个只读层代表一个Dockerfile指令。这些层是堆叠的，每个层都是上一层的变化的增量。考虑一下Dockerfile：

FROM ubuntu:18.04

COPY . /app

RUN make /app

CMD python /app/app.py

每条指令创建一层：

FROM从ubuntu:18.04Docker镜像创建一个图层。

COPY 从Docker客户端的当前目录添加文件。

RUN使用构建您的应用程序make。

CMD 指定在容器中运行什么命令。

运行镜像并生成容器时，可以 在基础层之上添加一个新的可写层（“容器层”）。对运行中的容器所做的所有更改（例如写入新文件，修改现有文件和删除文件）都将写入此薄可写容器层。

### ### 一般准则和建议

#### #### 创建临时容器

您定义的镜像Dockerfile应生成尽可能短暂的容器。“短暂”是指可以停止并销毁容器，然后对其进行重建和替换，并采用绝对的最低限度的设置和配置。

#### #### 了解构建环境

发出docker build命令时，当前的工作目录称为构建上下文。默认情况下，假定Dockerfile位于此处，但是您可以使用文件标志（-f）指定其他位置。无论Dockerfile实际位于何处，当前目录中文件和目录的所有递归内容都将作为构建上下文发送到Docker守护程序。

##### 构建上下文示例

为构建上下文创建一个目录并cd进入该目录。将“ hello”写入名为的文本文件，hello然后创建一个cat在其上运行的Dockerfile 。从构建上下文（.）中构建镜像：

mkdir myproject && cd myproject

echo "hello" > hello

echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile

docker build -t helloapp:v1 .

移动Dockerfile并hello进入单独的目录并构建镜像的第二个版本（不依赖于上次构建的缓存）。使用-f 以指向Dockerfile并指定构建上下文的目录：

mkdir -p dockerfiles context

mv Dockerfile dockerfiles && mv hello context

docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context

疏忽地包含了构建镜像所不需要的文件会导致较大的构建上下文和较大的镜像大小。这会增加生成镜像的时间，拉动和推动镜像的时间以及容器运行时的大小。要查看您的构建上下文有多大，请在构建您的时查找如下消息Dockerfile：

Sending build context to Docker daemon 187.8MB

#### #### 通过管道传递Dockerfile stdin

多克尔具有通过管道的能力来构建镜像Dockerfile通过stdin 与本地或远程构建上下文。管道中的Dockerfile通过stdin 可以执行一次性构建无需编写Dockerfile到磁盘上，或者在情况下是有用Dockerfile的产生，并且不应该事后持续。

为了方便起见，本节中的示例使用此处的文档，但是可以使用提供Dockerfileon的任何方法stdin。

例如，以下命令是等效的：

echo -e 'FROM busybox\nRUN echo "hello world"' | docker build -

docker build -<<EOF

FROM busybox

RUN echo "hello world"

EOF

您可以使用首选方法或最适合您的用例的方法替换示例。

##### 使用来自STDIN的DOCKERFILE构建镜像，而无需发送构建上下文

使用此语法可使用Dockerfilefrom 来构建镜像stdin，而无需发送其他文件作为构建上下文。连字符（-）占据的位置PATH，并指示Docker从而不是目录中读取构建上下文（仅包含Dockerfile）stdin：

docker build [OPTIONS] -

以下示例使用Dockerfile传递的来 构建镜像stdin。没有文件作为构建上下文发送到守护程序。

docker build -t myimage:latest -<<EOF

FROM busybox

RUN echo "hello world"

EOF

在您Dockerfile 不需要将文件复制到镜像中的情况下，省略构建上下文可能很有用，并且由于没有文件发送到守护程序，因此可以提高构建速度。

如果要通过从构建上下文中排除某些文件来提高构建速度，请使用.dockerignore进行排除。

注意：如果使用此语法，尝试构建使用COPY或ADD将失败的Dockerfile 。以下示例说明了这一点：

# create a directory to work in

mkdir example

cd example

# create an example file

touch somefile.txt

docker build -t myimage:latest -<<EOF

FROM busybox

COPY somefile.txt .

RUN cat /somefile.txt

EOF

# observe that the build fails

...

Step 2/3 : COPY somefile.txt .

COPY failed: stat /var/lib/docker/tmp/docker-builder249218248/somefile.txt: no such file or directory

##### 使用STDIN中的DOCKERFILE从本地构建上下文进行构建

使用此语法可使用本地文件系统上的文件，但使用Dockerfilefrom 来构建镜像stdin。该语法使用-f（或--file）选项来指定Dockerfile使用，使用连字符（-）作为文件名可指示多克尔读取Dockerfile从stdin：

docker build [OPTIONS] -f- PATH

下面的示例使用当前目录（.）作为构建上下文，并构建用的镜像Dockerfile，其通过传递stdin使用这里文档。

# create a directory to work in

mkdir example

cd example

# create an example file

touch somefile.txt

# build an image using the current directory as context, and a Dockerfile passed through stdin

docker build -t myimage:latest -f- . <<EOF

FROM busybox

COPY somefile.txt .

RUN cat /somefile.txt

EOF

##### 使用STDIN中的DOCKERFILE从远程构建上下文进行构建

使用此格式可以从远程文件来构建一个镜像git库，使用Dockerfile从stdin。该语法使用-f（或--file）选项来指定Dockerfile使用，使用连字符（-）作为文件名可指示多克尔读取Dockerfile从stdin：

docker build [OPTIONS] -f- PATH

如果您要从不包含的存储库中构建镜像Dockerfile，或者想要使用custom进行构建 Dockerfile而又不维护自己的存储库分支，则此语法很有用。

下面的示例使用Dockerfilefrom 构建一个镜像stdin，并hello.c从GitHub上的“ hello-world” Git存储库添加文件。

docker build -t myimage:latest -f- https://github.com/docker-library/hello-world.git <<EOF

FROM busybox

COPY hello.c .

EOF

引擎盖下

当使用远程Git存储库作为构建上下文构建镜像时，Docker git clone在本地计算机上执行一个存储库，并将这些文件作为构建上下文发送到守护程序。需要git在运行docker build命令的主机上安装此功能。

#### #### 使用多阶段构建

多阶段构建使您可以大幅度减小最终镜像的大小，而不必努力减少中间层和文件的数量。

由于镜像是在生成过程的最后阶段生成的，因此可以利用生成缓存来最小化镜像层。

例如，如果您的构建包含多个层，则可以将它们从更改频率较低（以确保生成缓存可重用）到更改频率较高的顺序排序：

安装构建应用程序所需的工具

安装或更新库依赖项

生成您的申请

Go应用程序的Dockerfile可能类似于：

FROM golang:1.11-alpine AS build

# Install tools required for project

# Run `docker build --no-cache .` to update dependencies

RUN apk add --no-cache git

RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock

# These layers are only re-built when Gopkg files are updated

COPY Gopkg.lock Gopkg.toml /go/src/project/

WORKDIR /go/src/project/

# Install library dependencies

RUN dep ensure -vendor-only

# Copy the entire project and build it

# This layer is rebuilt when a file changes in the project directory

COPY . /go/src/project/

RUN go build -o /bin/project

# This results in a single layer image

FROM scratch

COPY --from=build /bin/project /bin/project

ENTRYPOINT ["/bin/project"]

CMD ["--help"]

#### #### 不要安装不必要的软件包

为了降低复杂性，依赖性，文件大小和构建时间，请避免仅仅因为它们“很容易安装”而安装多余或不必要的软件包。例如，您不需要在数据库镜像中包括文本编辑器。

#### #### 解耦应用

每个容器应该只有一个方面。将应用程序解耦到多个容器中，可以更轻松地水平缩放和重复使用容器。例如，一个Web应用程序堆栈可能由三个单独的容器组成，每个容器都有自己的唯一镜像，以分离的方式管理Web应用程序，数据库和内存中的缓存。

将每个容器限制为一个进程是一个很好的经验法则，但这并不是一成不变的规则。例如，不仅可以使用初始化进程来生成容器 ，而且某些程序还可以自行生成其他进程。例如，Celery可以产生多个工作进程，而Apache可以为每个请求创建一个进程。

根据您的最佳判断，使容器尽可能保持清洁和模块化。如果容器相互依赖，则可以使用Docker容器网络 来确保这些容器可以通信。

#### #### 减少层数

在较旧的Docker版本中，务必最小化镜像中的层数以确保其性能。添加了以下功能来减少此限制：

只有说明RUN，COPY，ADD创建图层。其他说明创建临时的中间镜像，并且不会增加构建的大小。

尽可能使用多阶段构建，并且仅将所需的工件复制到最终镜像中。这使您可以在中间构建阶段中包含工具和调试信息，而无需增加最终镜像的大小。

#### #### 排序多行参数

尽可能通过字母数字排序多行参数来简化以后的更改。这有助于避免软件包重复，并使列表更易于更新。这也使PR易于阅读和查看。在反斜杠（\）之前添加空格也有帮助。

下面是来自一个示例buildpack-deps镜像：

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion

#### #### 利用构建缓存

在构建镜像时，Docker将逐步Dockerfile执行您的指令， 并按指定的顺序执行每个指令。在检查每条指令时，Docker会在其缓存中查找可重用的现有镜像，而不是创建新的（重复的）镜像。

如果根本不想使用缓存，则可以使用命令--no-cache=true 上的选项docker build。但是，如果您确实允许Docker使用其缓存，那么了解何时可以找到匹配的镜像非常重要。Docker遵循的基本规则概述如下：

从已在缓存中的父镜像开始，将下一条指令与从该基本镜像派生的所有子镜像进行比较，以查看是否其中一个是使用完全相同的指令构建的。如果不是，则高速缓存无效。

在大多数情况下，只需将中的指令Dockerfile与子镜像之一进行比较就足够了。但是，某些说明需要更多的检查和解释。

对于ADD和COPY指令，将检查镜像中文件的内容，并为每个文件计算一个校验和。在这些校验和中不考虑文件的最后修改时间和最后访问时间。在高速缓存查找期间，将校验和与现有镜像中的校验和进行比较。如果文件中的任何内容（例如内容和元数据）发生了更改，则缓存将无效。

除了ADD和COPY命令之外，缓存检查不会查看容器中的文件来确定缓存是否匹配。例如，在处理RUN apt-get -y update命令时，不检查容器中更新的文件以确定是否存在缓存命中。在这种情况下，仅使用命令字符串本身来查找匹配项。

一旦缓存无效，所有后续Dockerfile命令都会生成新镜像，并且不使用缓存。

### ### Dockerfile说明

这些建议旨在帮助您创建高效且可维护的Dockerfile。

##### FROM

Dockerfile的FROM指令参考

尽可能使用当前的官方镜像作为镜像的基础。我们建议使用Alpine镜像，因为它受到严格控制且尺寸较小（当前小于5 MB），同时仍是完整的Linux发行版。

##### LABEL

了解对象标签

您可以在镜像上添加标签，以帮助按项目组织镜像，记录许可信息，帮助自动化或其他原因。对于每个标签，添加一行LABEL并以一个或多个键值对开头。以下示例显示了不同的可接受格式。内嵌包含解释性注释。

带空格的字符串必须用引号引起来，否则必须转义。内引号（"）也必须转义。

# Set one or more individual labels

LABEL com.example.version="0.0.1-beta"

LABEL vendor1="ACME Incorporated"

LABEL vendor2=ZENITH\ Incorporated

LABEL com.example.release-date="2015-02-12"

LABEL com.example.version.is-production=""

一幅镜像可以有多个标签。在Docker 1.10之前，建议将所有标签合并为一条LABEL指令，以防止创建额外的层。不再需要此操作，但仍支持组合标签。

# Set multiple labels on one line

LABEL com.example.version="0.0.1-beta" com.example.release-date="2015-02-12"

上面也可以写成：

# Set multiple labels at once, using line-continuation characters to break long lines

LABEL vendor=ACME\ Incorporated \

com.example.is-beta= \

com.example.is-production="" \

com.example.version="0.0.1-beta" \

com.example.release-date="2015-02-12"

请参阅了解对象标签 以获取有关可接受的标签键和值的准则。有关查询标​​签的信息，请参阅“ 管理对象上的标签”中与过滤有关的项目。另请参阅 Dockerfile参考中的LABEL。

##### RUN

RUN指令的Dockerfile参考

将多行长或复杂的RUN语句分割成多行，并用反斜杠分隔，以使您Dockerfile更具可读性，可理解性和可维护性。

适当的

可能最常见的用例RUN是的应用apt-get。因为它安装了软件包，所以该RUN apt-get命令需要注意一些陷阱。

避免RUN apt-get upgrade和dist-upgrade，因为许多从父镜像的“基本”套餐的不能内部升级 特权的容器。如果父镜像中包含的软件包已过期，请联系其维护者。如果您知道有foo需要更新的特定软件包，请使用 apt-get install -y foo来自动更新。

始终在同 一条语句中结合RUN apt-get update使用。例如：apt-get installRUN

RUN apt-get update && apt-get install -y \

package-bar \

package-baz \

package-foo

apt-get update在RUN语句中单独使用会导致缓存问题，并且后续apt-get install指令会失败。例如，假设您有一个Dockerfile：

FROM ubuntu:18.04

RUN apt-get update

RUN apt-get install -y curl

构建镜像后，所有层都在Docker缓存中。假设您以后apt-get install通过添加额外的程序包进行修改：

FROM ubuntu:18.04

RUN apt-get update

RUN apt-get install -y curl nginx

Docker将初始指令和修改后的指令视为相同，并重复使用先前步骤中的缓存。其结果是，apt-get update在不执行，因为编译使用缓存的版本。由于apt-get update未运行，因此您的构建可能会获得curl和 nginx包的过时版本。

使用RUN apt-get update && apt-get install -y确保您的Dockerfile安装了最新的软件包版本，而无需进一步的编码或手动干预。这种技术称为“缓存清除”。您还可以通过指定软件包版本来实现缓存清除。这称为版本固定，例如：

RUN apt-get update && apt-get install -y \

package-bar \

package-baz \

package-foo=1.3.\*

版本固定会强制构建检索特定版本，而不管缓存中的内容是什么。该技术还可以减少由于所需包装中的意外更改而导致的故障。

以下是格式正确的RUN说明，其中演示了所有apt-get 建议。

RUN apt-get update && apt-get install -y \

aufs-tools \

automake \

build-essential \

curl \

dpkg-sig \

libcap-dev \

libsqlite3-dev \

mercurial \

reprepro \

ruby1.9.1 \

ruby1.9.1-dev \

s3cmd=1.1.\* \

&& rm -rf /var/lib/apt/lists/\*

该s3cmd参数指定版本1.1.\*。如果镜像先前使用的是旧版本，则指定新版本会导致缓存崩溃，apt-get update并确保安装新版本。在每行上列出软件包还可以防止软件包重复中的错误。

另外，/var/lib/apt/lists由于通过将apt缓存未存储在图层中来清理它，从而减小了镜像大小。由于该 RUN语句开头为apt-get update，因此包缓存始终在之前刷新apt-get install。

官方Debian和Ubuntu镜像会自动运行apt-get clean，因此不需要显式调用。

##### 使用管道

某些RUN命令取决于使用管道字符（|）将一个命令的输出管道传输到另一个命令的能力，如以下示例所示：

RUN wget -O - https://some.site | wc -l > /number

Docker使用/bin/sh -c解释器执行这些命令，该解释器仅评估管道中最后一个操作的退出代码以确定成功。在上面的示例中，只要wc -l命令成功，即使该wget命令失败，该构建步骤也会成功并生成一个新镜像。

如果您希望由于管道中的任何阶段的错误而导致命令失败，请添加前缀set -o pipefail &&以确保意外错误可以防止构建意外成功。例如：

RUN set -o pipefail && wget -O - https://some.site | wc -l > /number

并非所有的外壳程序都支持该-o pipefail选项。

在诸如dash基于Debian的镜像上的shell之类的情况下，请考虑使用exec形式的RUN显式选择确实支持该pipefail选项的shell 。例如：

RUN ["/bin/bash", "-c", "set -o pipefail && wget -O - https://some.site | wc -l > /number"]

##### CMD

CMD指令的Dockerfile参考

该CMD说明应用于运行镜像所包含的软件以及所有参数。CMD应该几乎总是以的形式使用CMD ["executable", "param1", "param2"…]。因此，如果镜像用于服务（例如Apache和Rails），则将运行CMD ["apache2","-DFOREGROUND"]。实际上，对于任何基于服务的镜像都建议使用这种形式的指令。

在大多数其他情况下，CMD应使用交互式外壳程序，例如bash，python和perl。例如，CMD ["perl", "-de0"]，CMD ["python"]，或CMD ["php", "-a"]。使用这种形式意味着执行诸如之类的东西时 docker run -it python，您将被放入一个可用的shell中，可以开始使用了。 除非您和您的预期用户已经非常熟悉其 工作CMD方式，否则应该很少CMD ["param", "param"]与结合使用。ENTRYPOINTENTRYPOINT

##### EXPOSE

Dockerfile的EXPOSE指令参考

该EXPOSE指令指示容器在其上侦听连接的端口。因此，应为应用程序使用通用的传统端口。例如，包含Apache Web服务器EXPOSE 80的镜像将使用，而包含MongoDB的镜像将使用EXPOSE 27017等等。

对于外部访问，您的用户可以执行docker run一个标志，指示如何将指定端口映射到他们选择的端口。对于容器链接，Docker为从接收者容器到源容器（即MYSQL\_PORT\_3306\_TCP）的路径提供了环境变量。

##### ENV

ENV指令的Dockerfile参考

为了使新软件更易于运行，可以使用ENV更新PATH容器所安装软件的 环境变量。例如，ENV PATH /usr/local/nginx/bin:$PATH确保其CMD ["nginx"] 正常工作。

该ENV指令对于提供特定于您要容器化的服务的必需环境变量（例如Postgres的）也很有用 PGDATA。

最后，ENV还可以用于设置常用的版本号，以便更容易维护版本凹凸，如以下示例所示：

ENV PG\_MAJOR 9.3

ENV PG\_VERSION 9.3.4

RUN curl -SL http://example.com/postgres-$PG\_VERSION.tar.xz | tar -xJC /usr/src/postgress && …

ENV PATH /usr/local/postgres-$PG\_MAJOR/bin:$PATH

类似于在程序中具有恒定变量（与硬编码值相反），此方法使您可以更改一条ENV指令以自动神奇地更改容器中软件的版本。

每ENV行都创建一个新的中间层，就像RUN命令一样。这意味着即使您在以后的层中取消设置环境变量，它也仍将保留在该层中，并且其值可以转储。您可以通过创建如下所示的Dockerfile，然后对其进行构建来进行测试。

FROM alpine

ENV ADMIN\_USER="mark"

RUN echo $ADMIN\_USER > ./mark

RUN unset ADMIN\_USER

$ docker run --rm test sh -c 'echo $ADMIN\_USER'

mark

为避免这种情况，并真正取消设置环境变量，请在RUN命令行中使用带有shell命令的命令来设置，使用和取消设置环境变量。您可以使用;或分隔命令&&。如果您使用第二种方法，并且其中一个命令失败，则命令docker build也将失败。这通常是个好主意。使用\作为Linux的Dockerfiles续行符提高可读性。您也可以将所有命令放入一个Shell脚本中，并让该RUN命令仅运行该Shell脚本。

FROM alpine

RUN export ADMIN\_USER="mark" \

&& echo $ADMIN\_USER > ./mark \

&& unset ADMIN\_USER

CMD sh

$ docker run --rm test sh -c 'echo $ADMIN\_USER'

##### ADD or COPY

用于ADD指令的Dockerfile参考

COPY指令的Dockerfile参考

尽管ADD和COPY在功能上相似，但是一般来说COPY 是优选的。那是因为它比透明ADD。COPY仅支持将本地文件基本复制到容器中，而ADD具有一些功能（例如，仅本地tar提取和远程URL支持）并不立即显而易见。因此，最好的用途ADD是将本地tar文件自动提取到镜像中，如中所示ADD rootfs.tar.xz /。

如果您有多个Dockerfile步骤使用了上下文中的不同文件，COPY则应单独执行而不是一次执行。这样可以确保仅在特别需要的文件发生更改的情况下，才使每个步骤的构建缓存无效（强制重新运行该步骤）。

例如：

COPY requirements.txt /tmp/

RUN pip install --requirement /tmp/requirements.txt

COPY . /tmp/

与在RUN步骤COPY . /tmp/之前放置缓存相比，该步骤 导致的缓存无效化更少。

由于镜像大小很重要，ADD因此强烈建议不要使用从远程URL获取软件包的方法。您应该使用curl或wget代替。这样，您可以在提取文件后删除不再需要的文件，而不必在镜像中添加另一层。例如，您应该避免做以下事情：

ADD http://example.com/big.tar.xz /usr/src/things/

RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things

RUN make -C /usr/src/things all

相反，请执行以下操作：

RUN mkdir -p /usr/src/things \

&& curl -SL http://example.com/big.tar.xz \

| tar -xJC /usr/src/things \

&& make -C /usr/src/things all

对于不需要ADDtar自动提取功能的其他项目（文件，目录），应始终使用COPY。

##### ENTRYPOINT

ENTRYPOINT指令的Dockerfile参考

最好的用法ENTRYPOINT是设置镜像的主命令，从而使该镜像像该命令一样运行（然后CMD用作默认标志）。

让我们从命令行工具的镜像示例开始s3cmd：

ENTRYPOINT ["s3cmd"]

CMD ["--help"]

现在可以像这样运行镜像以显示命令的帮助：

$ docker run s3cmd

或使用正确的参数执行命令：

$ docker run s3cmd ls s3://mybucket

这很有用，因为镜像名称可以用作对二进制文件的引用，如上面的命令所示。

该ENTRYPOINT指令也可以与辅助脚本结合使用，即使启动该工具可能需要一个以上的步骤，也可以使其以与上述命令类似的方式起作用。

例如，Postgres Official Image 使用以下脚本作为其脚本ENTRYPOINT：

#!/bin/bash

set -e

if [ "$1" = 'postgres' ]; then

chown -R postgres "$PGDATA"

if [ -z "$(ls -A "$PGDATA")" ]; then

gosu postgres initdb

fi

exec gosu postgres "$@"

fi

exec "$@"

将应用程序配置为PID 1

此脚本使用的execbash命令 ，以使最终运行的应用程序成为容器的PID 1.这允许应用程序接收发送到所述容器任何Unix信号。有关更多信息，请参见ENTRYPOINT参考资料。

将帮助程序脚本复制到容器中，并通过ENTRYPOINT在容器启动时运行：

COPY ./docker-entrypoint.sh /

ENTRYPOINT ["/docker-entrypoint.sh"]

CMD ["postgres"]

该脚本允许用户以多种方式与Postgres进行交互。

它可以简单地启动Postgres：

$ docker run postgres

或者，它可以用于运行Postgres并将参数传递给服务器：

$ docker run postgres postgres --help

最后，它也可以用于启动一个完全不同的工具，例如Bash：

$ docker run --rm -it postgres bash

##### VOLUME

VOLUME指令的Dockerfile参考

该VOLUME指令应用于公开由Docker容器创建的任何数据库存储区，配置存储或文件/文件夹。强烈建议VOLUME您将镜像用于任何可变和/或用户可维修的部分。

##### USER

USER指令的Dockerfile参考

如果服务可以在没有特权的情况下运行，请使用USER更改为非root用户。通过创建在用户和组开始Dockerfile喜欢的东西

RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres

考虑一个明确的UID / GID

为镜像中的用户和组分配了不确定的UID / GID，因为无论镜像重建如何，都会分配“下一个” UID / GID。因此，如果很关键，则应分配一个明确的UID / GID。

由于Go存档/ tar软件包处理稀疏文件中的一个未解决的错误，尝试在Docker容器内创建具有非常大的UID的用户可能会导致磁盘耗尽，因为/var/log/faillog在容器层中填充了NULL（\ 0）字符。一种解决方法是将--no-log-init标志传递给useradd。Debian / Ubuntu adduser包装器不支持此标志。

避免安装或使用sudo它具有不可预测的TTY和信号转发行为，这可能会导致问题。如果您绝对需要类似的功能sudo（例如将守护程序初始化为，root但以非运行方式运行）root，请考虑使用“ gosu”。

最后，为了减少层次和复杂性，请避免USER频繁地来回切换。

##### WORKDIR

WORKDIR指令的Dockerfile参考

为了清楚和可靠起见，您应始终为使用绝对路径 WORKDIR。另外，您应该使用WORKDIR而不是像那样RUN cd … && do-something繁琐的说明，这些说明难以阅读，排除故障和维护。

##### ONBUILD

Dockerfile参考ONBUILD指令

一个ONBUILD命令将当前执行后Dockerfile构建完成。 ONBUILD在派生FROM当前镜像的任何子镜像中执行。将该ONBUILD命令视为父母Dockerfile对孩子的指示Dockerfile。

Docker构建ONBUILD在子级中的任何命令之前先执行命令 Dockerfile。

ONBUILD对于将要构建FROM给定镜像的镜像很有用。例如，您将使用ONBUILD一个语言堆栈镜像，该镜像构建Dockerfile在Ruby中ONBUILD以该语言编写的任意用户软件 ，正如您在Ruby的变体中看到的那样。

使用生成的镜像ONBUILD应获得一个单独的标签，例如： ruby:1.9-onbuild或ruby:2.0-onbuild。

放入ADD或COPY放入时要小心ONBUILD。如果新构建的上下文缺少要添加的资源，则“ onbuild”镜像将灾难性地失败。如上所述，添加一个单独的标签可以允许Dockerfile作者做出选择，从而缓解这种情况。

# # 8 配置docker网络

Docker容器和服务如此强大的原因之一是您可以将它们连接在一起，或将它们连接到非Docker工作负载。Docker容器和服务甚至不需要知道它们已部署在Docker上，也不必知道它们的对等对象是否也是Docker工作负载。无论您的Docker主机运行Linux，Windows还是两者结合，您都可以使用Docker以与平台无关的方式管理它们。

## ## 本主题范围

本主题并**没有**进入操作系统的具体细节如何泊坞网络的工作，所以你不会找到有关信息如何泊坞操纵iptables 在Linux或规则，它是如何操纵在Windows服务器上的路由规则，你不会找到码头工人如何形成的详细信息并封装数据包或处理加密。请参阅[Docker和iptables](https://docs.docker.com/network/iptables/) 和 [Docker参考架构：设计可扩展的可移植Docker容器网络](http://success.docker.com/article/networking) ，以获取更深入的技术细节。

此外，本主题未提供有关如何创建，管理和使用Docker网络的任何教程。每个部分均包含指向相关教程和命令参考的链接。

## ## 网络驱动

Docker的网络子系统可使用驱动程序插入。默认情况下，有几个驱动程序，它们提供核心联网功能：

* bridge：默认的网络驱动程序。如果未指定驱动程序，则这是您正在创建的网络类型。**当您的应用程序在需要通信的独立容器中运行时，通常会使用网桥网络。**请参阅 [网桥网络](https://docs.docker.com/network/bridge/)。
* host：对于独立容器，请删除容器与Docker主机之间的网络隔离，然后直接使用主机的网络。host 仅可用于Docker 17.06及更高版本上的集群服务。请参阅 [使用主机网络](https://docs.docker.com/network/host/)。
* overlay：覆盖网络将多个Docker守护程序连接在一起，并使群集服务能够相互通信。您还可以使用覆盖网络来促进群集服务和独立容器之间或不同Docker守护程序上的两个独立容器之间的通信。这种策略消除了在这些容器之间进行操作系统级路由的需要。请参阅[叠加网络](https://docs.docker.com/network/overlay/)。
* macvlan：Macvlan网络允许您将MAC地址分配给容器，使其在网络上显示为物理设备。Docker守护程序通过其MAC地址将流量路由到容器。macvlan 在处理希望直接连接到物理网络而不是通过Docker主机的网络堆栈进行路由的旧应用程序时，使用驱动程序有时是最佳选择。请参阅 [Macvlan网络](https://docs.docker.com/network/macvlan/)。
* none：对于此容器，请禁用所有联网。通常与自定义网络驱动程序一起使用。none不适用于群体服务。请参阅 [禁用容器联网](https://docs.docker.com/network/none/)。
* [网络插件](https://docs.docker.com/engine/extend/plugins_services/)：您可以在Docker上安装和使用第三方网络插件。这些插件可从 [Docker Hub](https://hub.docker.com/search?category=network&q=&type=plugin) 或第三方供应商处获得。有关安装和使用给定网络插件的信息，请参阅供应商的文档。

络驱动程序摘要

* 当您需要多个容器在同一Docker主机上进行通信时，最好**使用用户定义的网桥网络**。
* 当网络堆栈不应与Docker主机隔离时，但您希望容器的其他方面隔离时，**主机网络**是最佳选择。
* 当您需要在不同Docker主机上运行的容器进行通信时，或者当多个应用程序使用集群服务一起工作时，**覆盖网络**是最好的。
* 从VM设置迁移或需要容器看起来像网络上的物理主机（每个主机都有唯一的MAC地址）时，**Macvlan网络**是最好的。
* **第三方网络插件**使您可以将Docker与专用网络堆栈集成。

## ## 使用网桥网络

就网络而言，桥接网络是在网段之间转发流量的链路层设备。桥可以是在主机内核中运行的硬件设备或软件设备。

就Docker而言，网桥网络使用软件网桥，该软件网桥允许连接到同一网桥网络的容器进行通信，同时与未连接到该网桥网络的容器隔离。Docker网桥驱动程序会自动在主机中安装规则，以使不同网桥网络上的容器无法直接相互通信。

桥接网络适用于在**同一** Docker守护程序主机上运行的容器。为了在不同Docker守护程序主机上运行的容器之间进行通信，您可以在OS级别管理路由，也可以使用[覆盖网络](https://docs.docker.com/network/overlay/)。

启动Docker时，会自动创建一个[默认的桥接网络](https://docs.docker.com/network/bridge/#use-the-default-bridge-network)（也称为bridge），除非另有说明，否则新启动的容器将连接到它。您还可以创建用户定义的自定义网桥网络。**用户定义的网桥网络优于默认bridge 网络。**

### ### 用户定义的网桥和默认网桥之间的区别

* **用户定义的桥可在容器化应用程序之间提供更好的隔离和互操作性**。

连接到同一用户定义的网桥网络的容器会自动将**所有端口**彼此公开，而**不会**向外界公开**任何端口**。这使容器化的应用程序可以轻松地彼此通信，而不会意外打开对外界的访问。

想象一个具有Web前端和数据库后端的应用程序。外界需要访问Web前端（也许在端口80上），但是只有后端本身需要访问数据库主机和端口。使用用户定义的网桥，只需打开Web端口，并且数据库应用程序不需要打开任何端口，因为Web前端可以通过用户定义的网桥到达它。

如果在默认网桥网络上运行相同的应用程序堆栈，则需要使用-p或--publish 标志分别打开Web端口和数据库端口。这意味着Docker主机需要通过其他方式阻止对数据库端口的访问。

* **用户定义的网桥可在容器之间提供自动DNS解析**。

默认桥接网络上的容器只能通过IP地址相互访问，除非您使用被认为是传统的[--link选项](https://docs.docker.com/network/links/)。在用户定义的网桥网络上，容器可以通过名称或别名相互解析。

想象一下与上一点相同的应用程序，它具有一个Web前端和一个数据库后端。如果调用容器web和db，则db无论应用程序堆栈在哪个Docker主机上运行，Web容器都可以在处连接到db容器。

如果在默认网桥网络上运行相同的应用程序堆栈，则需要在容器之间手动创建链接（使用旧式--link 标志）。这些链接需要双向创建，因此您可以看到，要进行通信的容器超过两个，这将变得很复杂。另外，您可以操纵/etc/hosts容器中的文件，但这会产生难以调试的问题。

* **容器可以随时随地从用户定义的网络连接和分离**。

在容器的生命周期内，您可以即时将其与用户定义的网络连接或断开连接。要从默认桥接网络中删除容器，您需要停止容器并使用其他网络选项重新创建它。

* **每个用户定义的网络都会创建一个可配置的网桥**。

如果您的容器使用默认桥接网络，则可以对其进行配置，但是所有容器都使用相同的设置，例如MTU和iptables规则。另外，配置默认桥接网络发生在Docker本身之外，并且需要重新启动Docker。

用户定义的桥接网络是使用创建和配置的 docker network create。如果不同的应用程序组具有不同的网络要求，则可以在创建时分别配置每个用户定义的网桥。

* **默认网桥网络上的链接容器共享环境变量**。

最初，在两个容器之间共享环境变量的唯一方法是使用[--link](https://docs.docker.com/network/links/)链接它们。用户定义的网络无法进行这种类型的变量共享。但是，存在共享环境变量的高级方法。一些想法：

多个容器可以使用Docker卷挂载包含共享信息的文件或目录。

使用可以一起启动多个容器docker-compose，并且compose文件可以定义共享变量。

您可以使用群体服务代替独立容器，并利用共享的[机密](https://docs.docker.com/engine/swarm/secrets/)和 [配置](https://docs.docker.com/engine/swarm/configs/)。

连接到同一用户定义网桥网络的容器可以有效地将所有端口彼此公开。为了使容器或不同网络上的非Docker主机可以访问该端口，必须使用或 标志*发布*该端口。-p--publish

### ### 管理用户定义的网桥

使用该docker network create命令创建用户定义的网桥网络。

$ docker network create my-net

您可以指定子网，IP地址范围，网关和其他选项。有关详细信息，请参见 [docker网络创建](https://docs.docker.com/engine/reference/commandline/network_create/#specify-advanced-options) 参考或输出docker network create --help。

使用该docker network rm命令删除用户定义的网桥网络。如果容器当前已连接到网络， [请先断开它们的连接](https://docs.docker.com/network/bridge/#disconnect-a-container-from-a-user-defined-bridge) 。

$ docker network rm my-net

到底发生了什么事？

当您创建或删除用户定义的网桥或将容器与用户定义的网桥连接或断开连接时，Docker使用特定于操作系统的工具来管理基础网络基础架构（例如iptables在Linux上添加或删除网桥设备或配置规则） ）。这些细节应视为实施细节。让Docker为您管理用户定义的网络。

### ### 将容器连接到用户定义的网桥

创建新容器时，可以指定一个或多个--network标志。本示例将Nginx容器连接到my-net网络。它还将容器中的端口80发布到Docker主机上的端口8080，以便外部客户端可以访问该端口。连接到my-net 网络的任何其他容器都可以访问该my-nginx容器上的所有端口，反之亦然。

$ docker create --name my-nginx \

--network my-net \

--publish 8080:80 \

nginx:latest

要将**运行中的**容器连接到现有的用户定义的网桥，请使用以下 docker network connect命令。以下命令将一个已经在运行的my-nginx容器连接 到一个已经存在的my-net网络：

$ docker network connect my-net my-nginx

### ### 断开容器与用户定义的网桥的连接

要将运行中的容器与用户定义的网桥断开连接，请使用以下docker network disconnect命令。以下命令将my-nginx 容器与my-net网络断开连接。

$ docker network disconnect my-net my-nginx

### ### 使用IPv6

如果需要对Docker容器的IPv6支持，则需要 在创建任何IPv6网络或分配容器IPv6地址之前，在Docker守护程序上启用该选项并重新加载其配置。

创建网络时，可以指定--ipv6标志以启用IPv6。您不能在默认bridge网络上有选择地禁用IPv6支持。

### ### 启用从Docker容器到外界的转发

默认情况下，来自连接到默认网桥网络的容器的流量 不会转发到外界。要启用转发，您需要更改两个设置。这些不是Docker命令，它们会影响Docker主机的内核。

1.配置Linux内核以允许IP转发。

$ sysctl net.ipv4.conf.all.forwarding=1

2. 将策略的iptables FORWARD策略从更改DROP为 ACCEPT。

$ sudo iptables -P FORWARD ACCEPT

这些设置不会在重新启动后持续存在，因此您可能需要将它们添加到启动脚本中。

### ### 使用默认的桥接网络

默认bridge网络被认为是Docker的遗留细节，不建议用于生产环境。对其进行配置是手动操作，并且存在 技术缺陷。

### ### 将容器连接到默认网桥网络

如果未使用该--network标志指定网络，而是指定了网络驱动程序，则默认情况下，容器将连接到默认bridge网络。连接到默认bridge网络的容器只能通过IP地址进行通信，除非它们使用legacy --link进行链接 。

### ### 配置默认网桥网络

要配置默认bridge网络，请在中指定选项daemon.json。这是daemon.json指定了几个选项的示例。仅指定您需要自定义的设置。

{

"bip": "192.168.1.5/24",

"fixed-cidr": "192.168.1.5/25",

"fixed-cidr-v6": "2001:db8::/64",

"mtu": 1500,

"default-gateway": "10.20.1.1",

"default-gateway-v6": "2001:db8:abcd::89",

"dns": ["10.20.1.2","10.20.1.3"]

}

重新启动Docker以使更改生效。

### ### 将IPv6与默认桥接网络一起使用

如果将Docker配置为支持IPv6（请参阅使用IPv6），则还会自动为IPv6配置默认的桥接网络。与用户定义的网桥不同，您不能在默认网桥上有选择地禁用IPv6。

## ## 使用覆盖网络

该overlay网络驱动程序会创建多个码头工人守护主机之间的分布式网络。该网络位于特定于主机的网络之上（（覆盖）特定主机的网络），从而允许与其连接的容器（包括群集服务容器）进行安全通信。Docker透明地处理每个数据包与正确的Docker守护程序主机和正确的目标容器之间的路由。

初始化群集或将Docker主机加入现有群集时，将在该Docker主机上创建两个新网络：

* 一个称为的覆盖网络ingress，用于处理与群体服务有关的控制和数据流量。创建群集服务并且不将其连接到用户定义的覆盖网络时，ingress 默认情况下它将连接到网络。
* 一个名为的桥接网络docker\_gwbridge，它将各个Docker守护程序连接到该集群中的其他守护程序。

您可以overlay使用来创建用户定义的网络docker network create，就像创建用户定义的bridge网络一样。服务或容器可以一次连接到多个网络。服务或容器只能在它们各自连接的网络之间进行通信。

尽管您可以将群集服务和独立容器都连接到覆盖网络，但是默认行为和配置问题有所不同。因此，本主题的其余部分分为适用于所有覆盖网络的操作，适用于群集服务网络的操作以及适用于独立容器使用的覆盖网络的操作。

### ### 所有覆盖网络的操作

#### 创建覆盖网络

**先决条件：**

* 使用覆盖网络的Docker守护程序的防火墙规则

您需要打开以下端口，以往返于参与覆盖网络的每个Docker主机的流量：

* + 用于群集管理通信的TCP端口2377
  + TCP和UDP端口7946，用于节点之间的通信
  + UDP端口4789，用于覆盖网络流量
* 在创建覆盖网络之前，您需要使用初始化Docker守护程序作为swarm管理器，docker swarm init或者使用将该Docker守护程序加入现有的swarm docker swarm join。这两个都将创建默认的ingress覆盖网络，默认情况下，群集服务会使用该 覆盖网络。即使您从未计划使用群体服务，也需要这样做。之后，您可以创建其他用户定义的覆盖网络。

要创建用于群服务的覆盖网络，请使用以下命令：

$ docker network create -d overlay my-overlay

要创建覆盖网络，群集服务**或** 独立容器可以使用该覆盖网络来与在其他Docker守护程序上运行的其他独立容器进行通信，请添加以下--attachable标志：

$ docker network create -d overlay --attachable my-attachable-overlay

您可以指定IP地址范围，子网，网关和其他选项。有关docker network create --help详细信息，请参见 。

#### 加密覆盖网络上的流量

默认情况下，在GCM模式下使用[AES算法](https://en.wikipedia.org/wiki/Galois/Counter_Mode)对所有群集服务管理流量进行加密 。群中的管理器节点每12小时旋转一次用于加密八卦数据的密钥。

要同时加密应用程序数据，请--opt encrypted在创建覆盖网络时添加。这样可以在vxlan级别启用IPSEC加密。这种加密对性能的影响是不可忽略的，因此在生产中使用此选项之前，应先对其进行测试。

启用覆盖加密后，Docker会在计划为覆盖网络上附加的服务安排任务的所有节点之间创建IPSEC隧道。这些隧道还在GCM模式下使用AES算法，并且管理器节点每12小时自动旋转一次密钥。

**不要将Windows节点附加到加密的覆盖网络。**

Windows不支持覆盖网络加密。如果Windows节点尝试连接到加密的覆盖网络，则不会检测到错误，但是该节点无法通信。

群模式覆盖网络和独立容器

您可以同时使用覆盖网络功能，--opt encrypted --attachable 并将非托管容器附加到该网络：

$ docker network create --opt encrypted --driver overlay --attachable my-attachable-multi-host-network

#### 自定义默认入口网络

大多数用户从不需要配置ingress网络，但是Docker 17.05及更高版本允许您进行配置。如果自动选择的子网与网络上已经存在的子网发生冲突，或者您需要自定义其他低级网络设置（例如MTU），这将很有用。

定制ingress网络涉及删除并重新创建它。这通常在您在集群中创建任何服务之前完成。如果您具有发布端口的现有服务，则需要先删除这些服务，然后才能删除ingress网络。

在没有ingress网络的时间内，未发布端口的现有服务将继续运行，但负载不平衡。这会影响发布端口的服务，例如发布端口80的WordPress服务。

使用检查ingress网络docker network inspect ingress，并删除所有与容器连接的服务。这些是发布端口的服务，例如发布端口80的WordPress服务。如果未停止所有此类服务，则下一步将失败。

删除现有ingress网络：

$ docker network rm ingress

WARNING! Before removing the routing-mesh network, make sure all the nodes

in your swarm run the same docker engine version. Otherwise, removal may not

be effective and functionality of newly created ingress networks will be

impaired.

Are you sure you want to continue? [y/N]

使用该--ingress标志以及要设置的自定义选项创建一个新的覆盖网络。本示例将MTU设置为1200，将子网设置为10.11.0.0/16，并将网关设置为10.11.0.2。

$ docker network create \

--driver overlay \

--ingress \

--subnet=10.11.0.0/16 \

--gateway=10.11.0.2 \

--opt com.docker.network.driver.mtu=1200 \

my-ingress

注意：您可以为ingress网络命名ingress，而不是 ，但是只能有一个。尝试创建第二个失败。

重新启动您在第一步中停止的服务。

#### 自定义docker\_gwbridge接口

这docker\_gwbridge是一个虚拟网桥，用于将覆盖网络（包括ingress网络）连接到单个Docker守护程序的物理网络。当您初始化集群或将Docker主机加入集群时，Docker会自动创建它，但它不是Docker设备。它存在于Docker主机的内核中。如果需要自定义其设置，则必须在将Docker主机加入群集之前或从群集中暂时删除主机之后进行。

停止Docker。

删除现有docker\_gwbridge接口。

$ sudo ip link set docker\_gwbridge down

$ sudo ip link del dev docker\_gwbridge

启动Docker。不要加入或初始化群体。

docker\_gwbridge使用docker network create命令使用自定义设置手动创建或重新创建网桥。本示例使用子网10.11.0.0/16。有关可自定义选项的完整列表，请参阅网桥驱动程序选项。

$ docker network create \

--subnet 10.11.0.0/16 \

--opt com.docker.network.bridge.name=docker\_gwbridge \

--opt com.docker.network.bridge.enable\_icc=false \

--opt com.docker.network.bridge.enable\_ip\_masquerade=true \

docker\_gwbridge

初始化或加入群。由于该桥已经存在，因此Docker不会使用自动设置来创建它。

### ### 群服务的操作

#### 在覆盖网络上发布端口

连接到同一覆盖网络的群集服务有效地将所有端口彼此公开。对于一个端口是服务的访问之外，该端口必须公布使用-p或--publish标志上docker service create或docker service update。支持旧的冒号分隔的语法和较新的逗号分隔的值语法。首选较长的语法，因为它有些自我记录。

| **标志值** | **描述** |
| --- | --- |
| -p 8080：80或 -p发布= 8080，目标= 80 | 将服务上的TCP端口80映射到路由网格上的端口8080。 |
| -p 8080：80 / udp或 -p Published = 8080，target = 80，protocol = udp | 将服务上的UDP端口80映射到路由网格上的端口8080。 |
| -p 8080：80 / tcp -p 8080：80 / udp或 -p Published = 8080，target = 80，protocol = tcp -p Published = 8080，target = 80，protocol = udp | 将服务上的TCP端口80映射到路由网格上的TCP端口8080，并将服务上的UDP端口80映射到路由网格上的UDP端口8080。 |

#### 绕过路由网格以实现群集服务

默认情况下，发布端口的群集服务会使用路由网格进行发布。当您连接到任何群集节点上的已发布端口时（无论它是否正在运行给定服务），您都将透明地重定向到正在运行该服务的工作器。实际上，Docker充当了集群服务的负载平衡器。使用路由网格的服务以虚拟IP（VIP）模式运行。甚至在每个节点上运行的服务（通过--mode global 标志）也使用路由网格。使用路由网格时，无法保证客户端请求哪些Docker节点服务。

要绕过路由网格，可以通过将标志设置为来使用DNS循环（DNSRR）模式启动服务。您必须在服务之前运行自己的负载平衡器。在Docker主机上对服务名称进行DNS查询会返回运行该服务的节点的IP地址列表。配置您的负载平衡器以使用此列表并平衡节点之间的流量。--endpoint-modednsrr

#### 控制和数据流量分开

默认情况下，尽管群集控制通信已加密，但与群集管理和进出您的应用程序的通信有关的控制通信在同一网络上运行。您可以将Docker配置为使用单独的网络接口来处理两种不同类型的流量。初始化或加入群集时，请分别指定--advertise-addr和--datapath-addr。您必须对加入群集的每个节点执行此操作。

### ### 覆盖网络上独立容器的操作

#### 将独立容器连接到覆盖网络

该ingress网络没有建立--attachable标志，这意味着只有群服务可以使用它，而不是独立的容器。您可以将独立容器连接到使用该--attachable标志创建的用户定义的覆盖网络。这使运行在不同Docker守护程序上的独立容器能够进行通信，而无需在各个Docker守护程序主机上设置路由。

#### 发布端口

| **标志值** | **描述** |
| --- | --- |
| -p 8080:80 | 将容器中的TCP端口80映射到覆盖网络上的端口8080。 |
| -p 8080:80/udp | 将容器中的UDP端口80映射到覆盖网络上的端口8080。 |
| -p 8080:80/sctp | 将容器中的SCTP端口80映射到覆盖网络上的端口8080。 |
| -p 8080:80/tcp -p 8080:80/udp | 将容器中的TCP端口80映射到覆盖网络上的TCP端口8080，并将容器中的UDP端口80映射到覆盖网络上的UDP端口8080。 |

#### 容器发现

在大多数情况下，您应该连接到服务名称，该名称是负载均衡的，并由支持该服务的所有容器（“任务”）处理。要获取支持该服务的所有任务的列表，请执行DNS查找tasks.<service-name>.

## ## 使用主机网络

如果host对容器使用网络模式，则该容器的网络堆栈不会与Docker主机隔离（该容器共享主机的网络名称空间），并且该容器不会分配自己的IP地址。例如，如果您运行绑定到端口80 host 的容器并使用网络，则该容器的应用程序可在主机IP地址上的端口80上使用。

**注：由于使用时容器不拥有自己的IP地址 host模式的网络，**[**端口映射**](https://docs.docker.com/network/overlay/#publish-ports)**不生效，并且-p，--publish，-P，和--publish-all选项都将被忽略，产生一个警告而不是：**

WARNING: Published ports are discarded when using host network mode

主机模式网络对于优化性能以及在容器需要处理大量端口的情况下很有用，因为它不需要网络地址转换（NAT），并且不会为每个端口创建“ userland-proxy”。

主机网络驱动程序仅在Linux主机上工作，而Mac的Docker桌面，Windows的Docker桌面或Windows Server的Docker EE不支持该主机网络驱动程序。

host通过传递--network host 给docker service create命令，您还可以将网络用于群集服务。在这种情况下，控制流量（与管理群集和服务有关的流量）仍会通过覆盖网络发送，但是单个群集服务容器会使用Docker守护程序的主机网络和端口发送数据。这带来了一些额外的限制。例如，如果服务容器绑定到端口80，则在给定的群集节点上只能运行一个服务容器。

## ## 使用macvlan网络

某些应用程序，尤其是旧版应用程序或监视网络流量的应用程序，期望直接连接到物理网络。在这种情况下，可以使用macvlan网络驱动程序为每个容器的虚拟网络接口分配MAC地址，使其看起来像是直接连接到物理网络的物理网络接口。在这种情况下，您需要在Docker主机上指定用于的物理接口macvlan，以及的子网和网关macvlan。您甚至可以macvlan使用不同的物理网络接口隔离网络。请记住以下几点：

* 由于IP地址耗尽或“ VLAN传播”，很容易无意间损坏您的网络，在这种情况下，您的网络中有大量不正确的唯一MAC地址。
* 您的网络设备需要能够处理“混杂模式”，在该模式下，可以为一个物理接口分配多个MAC地址。
* 如果您的应用程序可以使用网桥（在单个Docker主机上）或覆盖（跨多个Docker主机进行通信）工作，那么从长远来看，这些解决方案可能会更好。

### ### 创建一个macvlan网络

创建macvlan网络时，它可以处于桥接模式或802.1q中继桥接模式。

* 在桥接模式下，macvlan流量通过主机上的物理设备。
* 在802.1q中继桥接模式下，流量通过Docker动态创建的802.1q子接口。这使您可以更精细地控制路由和过滤。

#### 桥接模式

要创建macvlan与给定物理网络接口桥接的网络，请--driver macvlan与docker network create命令一起使用。您还需要指定parent，这是流量将在Docker主机上实际通过的接口。

$ docker network create -d macvlan \

--subnet=172.16.86.0/24 \

--gateway=172.16.86.1 \

-o parent=eth0 pub\_net

如果您需要排除IP地址在macvlan网络中的使用，例如已使用给定IP地址时，请使用--aux-addresses：

$ docker network create -d macvlan \

--subnet=192.168.32.0/24 \

--ip-range=192.168.32.128/25 \

--gateway=192.168.32.254 \

--aux-address="my-router=192.168.32.129" \

-o parent=eth0 macnet32

#### 802.1q中继桥接模式

如果您指定parent带有点的接口名称，例如eth0.50，则Docker会将其解释为的子接口，eth0并自动创建该子接口。

$ docker network create -d macvlan \

--subnet=192.168.50.0/24 \

--gateway=192.168.50.1 \

-o parent=eth0.50 macvlan50

#### 使用ipvlan而不是macvlan

在上面的示例中，您仍在使用L3桥。您可以改用ipvlan L2桥接器。指定-o ipvlan\_mode=l2。

$ docker network create -d ipvlan \

--subnet=192.168.210.0/24 \

--subnet=192.168.212.0/24 \

--gateway=192.168.210.254 \

--gateway=192.168.212.254 \

-o ipvlan\_mode=l2 ipvlan210

### ### 使用IPv6

如果已将Docker守护程序配置为允许IPv6，则可以使用双栈IPv4 / IPv6 macvlan网络。

$ docker network create -d macvlan \

--subnet=192.168.216.0/24 --subnet=192.168.218.0/24 \

--gateway=192.168.216.1 --gateway=192.168.218.1 \

--subnet=2001:db8:abc8::/64 --gateway=2001:db8:abc8::10 \

-o parent=eth0.218 \

-o macvlan\_mode=bridge macvlan216

## ## 禁用容器联网

如果要完全禁用容器上的网络堆栈，可以--network none在启动容器时使用该标志。在容器内，仅创建回送设备。以下示例说明了这一点。

### ### 1 创建容器。

$ docker run --rm -dit \

--network none \

--name no-net-alpine \

alpine:latest \

ash

### ### 2 通过在容器内执行一些常见的联网命令来检查容器的网络堆栈。

请注意，没有eth0创建。

$ docker exec no-net-alpine ip link show

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1

link/ipip 0.0.0.0 brd 0.0.0.0

3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN qlen 1

link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00 brd 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00

$ docker exec no-net-alpine ip route

第二个命令返回空，因为没有路由表。

### ### 3 停止容器。

它是使用--rm标记创建的，因此会自动删除。

$ docker container rm no-net-alpine

## ## 桥接网络--与独立容器联网

本系列教程探讨了独立Docker容器的网络连接。有关使用群集服务进行联网，请参阅 [使用群集服务进行联网](https://docs.docker.com/network/network-tutorial-overlay/)。如果您需要总体上了解有关Docker网络的更多信息，请参阅[概述](https://docs.docker.com/network/)。

本主题包括三个不同的教程。您可以在Linux，Windows或Mac上运行它们中的每一个，但对于最后两个，则需要在其他位置运行第二个Docker主机。

* [使用默认桥接网络](https://docs.docker.com/network/network-tutorial-standalone/#use-the-default-bridge-network)演示了如何使用bridgeDocker自动为您设置的默认网络。该网络不是生产系统的最佳选择。
* [使用用户定义的桥接网络](https://docs.docker.com/network/network-tutorial-standalone/#use-user-defined-bridge-networks)显示了如何创建和使用自己的自定义桥接网络，以连接在同一Docker主机上运行的容器。建议将其用于生产中运行的独立容器。

尽管[覆盖网络](https://docs.docker.com/network/overlay/)通常用于群体服务，但Docker 17.06及更高版本允许您将覆盖网络用于独立容器。这是[使用叠加网络](https://docs.docker.com/network/network-tutorial-overlay/#use-an-overlay-network-for-standalone-containers)的[教程的](https://docs.docker.com/network/network-tutorial-overlay/#use-an-overlay-network-for-standalone-containers)一部分 。

### ### 使用默认的桥接网络

在此示例中，您alpine在同一Docker主机上启动了两个不同的容器，并进行了一些测试以了解它们如何相互通信。您需要安装并运行Docker。

1 打开一个终端窗口。在执行其他任何操作之前，请先列出当前网络。如果您从未在此Docker守护程序上添加网络或初始化过集群，则应该看到以下内容。您可能会看到不同的网络，但至少应该看到以下内容（网络ID会有所不同）：

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

17e324f45964 bridge bridge local

6ed54d316334 host host local

7092879f2cc8 none null local

bridge列出了默认网络，以及host和none。后两个不是完全成熟的网络，但是用于启动直接连接到Docker守护程序主机的网络堆栈的容器，或者用于启动不具有网络设备的容器。本教程将把两个容器连接到bridge网络。

2 启动两个alpine运行的容器ash，这是Alpine的默认外壳程序，而不是bash。这些-dit标志意味着以分离的方式（在后台），交互的（具有输入能力）和TTY（以便您可以看到输入和输出）启动容器。由于您是分开启动的，因此您不会立即连接到容器。而是将打印容器的ID。由于未指定任何 --network标志，因此容器将连接到默认bridge网络。

$ docker run -dit --name alpine1 alpine ash

$ docker run -dit --name alpine2 alpine ash

检查两个容器是否确实已启动：

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

602dbf1edc81 alpine "ash" 4 seconds ago Up 3 seconds alpine2

da33b7aa74b0 alpine "ash" 17 seconds ago Up 16 seconds alpine1

3 检查bridge网络以查看连接了哪些容器。

$ docker network inspect bridge

[

{

"Name": "bridge",

"Id": "17e324f459648a9baaea32b248d3884da102dde19396c25b30ec800068ce6b10",

"Created": "2017-06-22T20:27:43.826654485Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Containers": {

"602dbf1edc81813304b6cf0a647e65333dc6fe6ee6ed572dc0f686a3307c6a2c": {

"Name": "alpine2",

"EndpointID": "03b6aafb7ca4d7e531e292901b43719c0e34cc7eef565b38a6bf84acf50f38cd",

"MacAddress": "02:42:ac:11:00:03",

"IPv4Address": "172.17.0.3/16",

"IPv6Address": ""

},

"da33b7aa74b0bf3bda3ebd502d404320ca112a268aafe05b4851d1e3312ed168": {

"Name": "alpine1",

"EndpointID": "46c044a645d6afc42ddd7857d19e9dcfb89ad790afb5c239a35ac0af5e8a5bc5",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"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": "1500"

},

"Labels": {}

}

]

在顶部附近，bridge列出了有关网络的信息，包括Docker主机和bridge 网络之间的网关的IP地址（172.17.0.1）。在该Containers键下，列出了每个已连接的容器以及有关其IP地址的信息（172.17.0.2for alpine1和172.17.0.3for alpine2）。

4 容器在后台运行。使用docker attach 命令连接到alpine1。

$ docker attach alpine1

/ #

提示更改为#指示您是root容器中的用户。使用ip addr show命令显示网络接口alpine1在容器中的外观：

# ip addr show

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

27: eth0@if28: <BROADCAST,MULTICAST,UP,LOWER\_UP,M-DOWN> mtu 1500 qdisc noqueue state UP

link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff

inet 172.17.0.2/16 scope global eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::42:acff:fe11:2/64 scope link

valid\_lft forever preferred\_lft forever

第一个接口是回送设备。现在忽略它。请注意，第二个接口具有IP地址172.17.0.2，该地址alpine1与上一步中显示的地址相同。

5 从内部alpine1，确保可以通过ping连接到Internet google.com。该-c 2标志将命令限制为两次ping 尝试。

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes

64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.841 ms

64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.897 ms

--- google.com ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 9.841/9.869/9.897 ms

6 现在尝试ping第二个容器。首先，通过其IP地址ping它 172.17.0.3：

# ping -c 2 172.17.0.3

PING 172.17.0.3 (172.17.0.3): 56 data bytes

64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.086 ms

64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.094 ms

--- 172.17.0.3 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.086/0.090/0.094 ms

这样成功了。接下来，尝试alpine2按容器名称ping 容器。这将失败。

# ping -c 2 alpine2

ping: bad address 'alpine2'

7 分离从alpine1没有通过使用分离的序列，停止它 CTRL+ p CTRL+ q（按住CTRL和类型p，随后q）。如果你愿意，重视alpine2并重复步骤4，5和6出现，取代alpine1了alpine2。

8 停止并卸下两个容器。

$ docker container stop alpine1 alpine2

$ docker container rm alpine1 alpine2

请记住，bridge不建议将默认网络用于生产。要了解用户定义的桥接网络，请继续 阅读下一个教程。

### ### 使用用户定义的网桥网络

在此示例中，我们再次启动两个alpine容器，但是将它们附加到alpine-net我们已经创建的用户定义的网络上。这些容器根本没有连接到默认bridge网络。然后，我们启动alpine连接到bridge网络但未连接到的第三个容器alpine-net，以及alpine连接到两个网络的第四个容器。

1 创建alpine-net网络。--driver bridge因为它是默认标志，所以您不需要该标志，但是此示例显示了如何指定它。

$ docker network create --driver bridge alpine-net

2 列出Docker的网络：

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

e9261a8c9a19 alpine-net bridge local

17e324f45964 bridge bridge local

6ed54d316334 host host local

7092879f2cc8 none null local

检查alpine-net网络。这显示了它的IP地址以及没有容器连接到它的事实：

$ docker network inspect alpine-net

[

{

"Name": "alpine-net",

"Id": "e9261a8c9a19eabf2bf1488bf5f208b99b1608f330cff585c273d39481c9b0ec",

"Created": "2017-09-25T21:38:12.620046142Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Containers": {},

"Options": {},

"Labels": {}

}

]

请注意，172.18.0.1与默认网桥网络相反，该网络的网关为，网关为172.17.0.1。您系统上的确切IP地址可能不同。

3 创建四个容器。注意--network标志。您只能在docker run命令期间连接到一个网络，因此您docker network connect以后也需要使用它 来连接alpine4到bridge 网络。

$ docker run -dit --name alpine1 --network alpine-net alpine ash

$ docker run -dit --name alpine2 --network alpine-net alpine ash

$ docker run -dit --name alpine3 alpine ash

$ docker run -dit --name alpine4 --network alpine-net alpine ash

$ docker network connect bridge alpine4

验证所有容器都在运行：

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

156849ccd902 alpine "ash" 41 seconds ago Up 41 seconds alpine4

fa1340b8d83e alpine "ash" 51 seconds ago Up 51 seconds alpine3

a535d969081e alpine "ash" About a minute ago Up About a minute alpine2

0a02c449a6e9 alpine "ash" About a minute ago Up About a minute alpine1

4 检查bridge网络并alpine-net再次检查网络：

$ docker network inspect bridge

[

{

"Name": "bridge",

"Id": "17e324f459648a9baaea32b248d3884da102dde19396c25b30ec800068ce6b10",

"Created": "2017-06-22T20:27:43.826654485Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Containers": {

"156849ccd902b812b7d17f05d2d81532ccebe5bf788c9a79de63e12bb92fc621": {

"Name": "alpine4",

"EndpointID": "7277c5183f0da5148b33d05f329371fce7befc5282d2619cfb23690b2adf467d",

"MacAddress": "02:42:ac:11:00:03",

"IPv4Address": "172.17.0.3/16",

"IPv6Address": ""

},

"fa1340b8d83eef5497166951184ad3691eb48678a3664608ec448a687b047c53": {

"Name": "alpine3",

"EndpointID": "5ae767367dcbebc712c02d49556285e888819d4da6b69d88cd1b0d52a83af95f",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"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": "1500"

},

"Labels": {}

}

]

容器alpine3和alpine4连接到bridge网络。

$ docker network inspect alpine-net

[

{

"Name": "alpine-net",

"Id": "e9261a8c9a19eabf2bf1488bf5f208b99b1608f330cff585c273d39481c9b0ec",

"Created": "2017-09-25T21:38:12.620046142Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Containers": {

"0a02c449a6e9a15113c51ab2681d72749548fb9f78fae4493e3b2e4e74199c4a": {

"Name": "alpine1",

"EndpointID": "c83621678eff9628f4e2d52baf82c49f974c36c05cba152db4c131e8e7a64673",

"MacAddress": "02:42:ac:12:00:02",

"IPv4Address": "172.18.0.2/16",

"IPv6Address": ""

},

"156849ccd902b812b7d17f05d2d81532ccebe5bf788c9a79de63e12bb92fc621": {

"Name": "alpine4",

"EndpointID": "058bc6a5e9272b532ef9a6ea6d7f3db4c37527ae2625d1cd1421580fd0731954",

"MacAddress": "02:42:ac:12:00:04",

"IPv4Address": "172.18.0.4/16",

"IPv6Address": ""

},

"a535d969081e003a149be8917631215616d9401edcb4d35d53f00e75ea1db653": {

"Name": "alpine2",

"EndpointID": "198f3141ccf2e7dba67bce358d7b71a07c5488e3867d8b7ad55a4c695ebb8740",

"MacAddress": "02:42:ac:12:00:03",

"IPv4Address": "172.18.0.3/16",

"IPv6Address": ""

}

},

"Options": {},

"Labels": {}

}

]

容器alpine1，alpine2和alpine4连接到 alpine-net网络。

5 在诸如的用户定义网络上alpine-net，容器不仅可以通过IP地址进行通信，还可以将容器名称解析为IP地址。此功能称为自动服务发现。让我们连接alpine1并测试一下。alpine1应该能够解析 alpine2和alpine4（以及alpine1本身）为IP地址。

$ docker container attach alpine1

# ping -c 2 alpine2

PING alpine2 (172.18.0.3): 56 data bytes

64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.085 ms

64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.090 ms

--- alpine2 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.085/0.087/0.090 ms

# ping -c 2 alpine4

PING alpine4 (172.18.0.4): 56 data bytes

64 bytes from 172.18.0.4: seq=0 ttl=64 time=0.076 ms

64 bytes from 172.18.0.4: seq=1 ttl=64 time=0.091 ms

--- alpine4 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.076/0.083/0.091 ms

# ping -c 2 alpine1

PING alpine1 (172.18.0.2): 56 data bytes

64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.026 ms

64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.054 ms

--- alpine1 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.026/0.040/0.054 ms

6 从中alpine1，您根本无法连接alpine3，因为它不在alpine-net网络上。

# ping -c 2 alpine3

ping: bad address 'alpine3'

不仅如此，您也无法通过其IP地址alpine3从alpine1进行连接。回顾网络的docker network inspect输出 bridge，找到alpine3的IP地址：172.17.0.2尝试ping通它。

ping -c 2 172.17.0.2

PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---

2 packets transmitted, 0 packets received, 100% packet loss

分离从alpine1用分离序列， CTRL+ p CTRL+ q（压紧CTRL和类型p，随后q）。

7 请记住，这alpine4已连接到默认bridge网络和alpine-net。它应该能够到达所有其他容器。但是，您将需要alpine3按其IP地址进行寻址。附加到它并运行测试。

$ docker container attach alpine4

# ping -c 2 alpine1

PING alpine1 (172.18.0.2): 56 data bytes

64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.074 ms

64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.082 ms

--- alpine1 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.074/0.078/0.082 ms

# ping -c 2 alpine2

PING alpine2 (172.18.0.3): 56 data bytes

64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.075 ms

64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.080 ms

--- alpine2 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.075/0.077/0.080 ms

# ping -c 2 alpine3

ping: bad address 'alpine3'

# ping -c 2 172.17.0.2

PING 172.17.0.2 (172.17.0.2): 56 data bytes

64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.089 ms

64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.075 ms

--- 172.17.0.2 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.075/0.082/0.089 ms

# ping -c 2 alpine4

PING alpine4 (172.18.0.4): 56 data bytes

64 bytes from 172.18.0.4: seq=0 ttl=64 time=0.033 ms

64 bytes from 172.18.0.4: seq=1 ttl=64 time=0.064 ms

--- alpine4 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.033/0.048/0.064 ms

8 作为最终测试，请通过ping确保您的容器都可以连接到互联网google.com。您已经很迷恋，alpine4因此请从那里开始尝试。接下来，断开alpine4并连接到alpine3 （仅连接到bridge网络）并重试。最后，连接到alpine1（仅连接到alpine-net网络），然后重试。

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes

64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.778 ms

64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.634 ms

--- google.com ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 9.634/9.706/9.778 ms

CTRL+p CTRL+q

$ docker container attach alpine3

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes

64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.706 ms

64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.851 ms

--- google.com ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 9.706/9.778/9.851 ms

CTRL+p CTRL+q

$ docker container attach alpine1

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes

64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.606 ms

64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.603 ms

--- google.com ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 9.603/9.604/9.606 ms

CTRL+p CTRL+q

9 停止并卸下所有容器和alpine-net网络。

$ docker container stop alpine1 alpine2 alpine3 alpine4

$ docker container rm alpine1 alpine2 alpine3 alpine4

$ docker network rm alpine-net

## ## 使用主机网络联网

### ### 目标

本教程的目标是启动一个nginx容器，该容器直接绑定到Docker主机上的端口80。从网络的角度来看，这与nginx进程直接在Docker主机而不是在容器中运行的隔离级别相同。但是，通过所有其他方式（例如存储，进程名称空间和用户名称空间），nginx进程与主机是隔离的。

### ### 先决条件

此过程要求端口80在Docker主机上可用。要使Nginx在其他端口上侦听，请参阅该镜像的 文档nginx

host网络驱动程序仅适用于Linux主机，而Mac的Docker桌面，Windows的Docker桌面或Windows Server的Docker EE不支持该网络驱动程序。

### ### 程序

1 创建并启动容器作为一个独立的过程。该--rm选项意味着一旦容器退出/停止就将其移除。该-d标志意味着要启动分离的容器（在后台）。

docker run --rm -d --network host --name my\_nginx nginx

2 通过浏览到http：// localhost：80 /来访问Nginx 。

3 使用以下命令检查您的网络堆栈：

检查所有网络接口，并确认未创建新的接口。

ip addr show

使用netstat命令验证哪个进程绑定到端口80 。您需要使用sudo该进程，因为该进程归Docker守护程序用户所有，否则您将无法看到其名称或PID。

sudo netstat -tulpn | grep :80

4 停止容器。使用--rm选项启动时，它将自动删除。

docker container stop my\_nginx

## ## 与覆盖网络联网

本主题包括四个不同的教程。您可以在Linux，Windows或Mac上运行它们中的每一个，但对于最后两个，则需要在其他位置运行第二个Docker主机。

使用默认覆盖网络演示了如何在初始化或加入群集时使用Docker自动为您设置的默认覆盖网络。该网络不是生产系统的最佳选择。

使用用户定义的覆盖网络显示了如何创建和使用自己的自定义覆盖网络来连接服务。建议将其用于生产中运行的服务。

将覆盖网络用于独立容器 显示了如何使用覆盖网络在不同Docker守护程序上的独立容器之间进行通信。

容器与群集服务 之间的通信使用可连接的覆盖网络在独立容器与群集服务之间建立通信。Docker 17.06及更高版本支持此功能。

### ### 先决条件

这些要求您至少有一个单节点群集，这意味着您已启动Docker并docker swarm init在主机上运行。您也可以在多节点群集上运行示例。

最后一个示例需要Docker 17.06或更高版本。

### ### 使用默认的覆盖网络

在此示例中，您将启动alpine服务并从各个服务容器的角度检查网络的特征。

本教程不涉及有关如何实现重叠网络的特定于操作系统的详细信息，而是从服务的角度着重介绍重叠的功能。

#### 先决条件

本教程需要三台物理或虚拟Docker主机，它们可以相互通信，并且都运行Docker 17.03或更高版本的新安装。本教程假定这三台主机在同一网络上运行，并且不涉及防火墙。

这些主机将被称为manager，worker-1和worker-2。该 manager主机将作为既是经理和工人，这意味着它可以运行服务任务和管理群。worker-1并且worker-2将作为唯一的工人，

如果您没有三台主机，一个简单的解决方案是在云提供商（例如Amazon EC2）上设置三台Ubuntu主机，它们全部位于同一网络上，并允许与该网络上的所有主机进行所有通信（使用诸如EC2安全组），然后按照Ubuntu上Docker Engine-Community的 安装说明进行操作。

#### 演练

创建群

在此过程结束时，所有三个Docker主机都将加入集群，并使用称为的覆盖网络将其连接在一起ingress。

1 在manager。初始化群。如果主机只有一个网络接口，则该--advertise-addr标志是可选的。

$ docker swarm init --advertise-addr=<IP-ADDRESS-OF-MANAGER>

记下所打印的文本，因为其中包含将用于加入worker-1和加入worker-2群体的令牌。将令牌存储在密码管理器中是个好主意。

2 在上worker-1，加入群体。如果主机只有一个网络接口，则该--advertise-addr标志是可选的。

$ docker swarm join --token <TOKEN> \

--advertise-addr <IP-ADDRESS-OF-WORKER-1> \

<IP-ADDRESS-OF-MANAGER>:2377

3 在上worker-2，加入群体。如果主机只有一个网络接口，则该--advertise-addr标志是可选的。

$ docker swarm join --token <TOKEN> \

--advertise-addr <IP-ADDRESS-OF-WORKER-2> \

<IP-ADDRESS-OF-MANAGER>:2377

4 在上manager，列出所有节点。此命令只能由管理员执行。

$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

d68ace5iraw6whp7llvgjpu48 \* ip-172-31-34-146 Ready Active Leader

nvp5rwavvb8lhdggo8fcf7plg ip-172-31-35-151 Ready Active

ouvx2l7qfcxisoyms8mtkgahw ip-172-31-36-89 Ready Active

您还可以使用该--filter标志按角色进行过滤：

$ docker node ls --filter role=manager

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

d68ace5iraw6whp7llvgjpu48 \* ip-172-31-34-146 Ready Active Leader

$ docker node ls --filter role=worker

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

nvp5rwavvb8lhdggo8fcf7plg ip-172-31-35-151 Ready Active

ouvx2l7qfcxisoyms8mtkgahw ip-172-31-36-89 Ready Active

5 名单上的多克尔网络manager，worker-1以及worker-2和通知他们每个人现在有所谓的覆盖网络ingress，并称为桥接网络docker\_gwbridge。这里只显示清单manager：

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

495c570066be bridge bridge local

961c6cae9945 docker\_gwbridge bridge local

ff35ceda3643 host host local

trtnl4tqnc3n ingress overlay swarm

c8357deec9cb none null local

在docker\_gwbridge该连接ingress网络到主机多克尔的网络接口，以便通信可以流入和流出群经理和工人。如果创建群集服务但未指定网络，则它们将连接到ingress网络。建议对将一起使用的每个应用程序或一组应用程序使用单独的覆盖网络。在下一个过程中，您将创建两个覆盖网络并将服务连接到每个覆盖网络。

创建服务

1 在上manager，创建一个新的覆盖网络，称为nginx-net：

$ docker network create -d overlay nginx-net

您无需在其他节点上创建覆盖网络，因为当其中一个节点开始运行需要该服务的服务任务时，该覆盖网络将自动创建。

2 在上manager，创建一个连接到的5副本Nginx服务nginx-net。该服务将向外部世界发布80端口。所有服务任务容器都可以彼此通信，而无需打开任何端口。

注意：服务只能在管理器上创建。

$ docker service create \

--name my-nginx \

--publish target=80,published=80 \

--replicas=5 \

--network nginx-net \

nginx

默认发布的模式ingress，这是用来当你没有指定mode的--publish标志，这意味着，如果你浏览到端口80上manager，worker-1或者worker-2，你会被连接到端口80上的5项服务任务之一，即使没有任务当前正在您浏览到的节点上运行。如果要使用host模式发布端口 ，则可以将其添加mode=host到--publish输出中。但是，在这种情况下，也应该使用--mode global代替--replicas=5，因为只有一个服务任务可以绑定给定节点上的给定端口。

3 运行docker service ls以监视服务启动的进度，这可能需要几秒钟。

4 检查nginx-net网络manager，worker-1和worker-2。请记住，您不需要手动创建它worker-1， worker-2因为Docker是为您创建的。输出将很长，但请注意Containers和Peers部分。Containers列出了从该主机连接到覆盖网络的所有服务任务（或独立容器）。

5 从manager，使用检查服务，docker service inspect my-nginx 并注意有关服务使用的端口和端点的信息。

6 创建一个新网络nginx-net-2，然后更新服务以使用该网络，而不是nginx-net：

$ docker network create -d overlay nginx-net-2

7 运行docker service ls以验证服务已更新，并且所有任务都已重新部署。运行docker network inspect nginx-net以验证没有任何容器连接到它。运行相同的命令 nginx-net-2，请注意所有服务任务容器都已连接到该命令。

注意：即使需要根据需要在swarm worker节点上自动创建覆盖网络，也不会自动删除它们。

8 清理服务和网络。从manager，运行以下命令。管理者将指示工作人员自动删除网络。

$ docker service rm my-nginx

$ docker network rm nginx-net nginx-net-2

### ### 使用用户定义的覆盖网络

#### 先决条件

本教程假定已经建立了集群，并且您正在管理中。

#### 演练

1 创建用户定义的覆盖网络。

$ docker network create -d overlay my-overlay

2 使用覆盖网络启动服务，并将端口80发布到Docker主机上的端口8080。

$ docker service create \

--name my-nginx \

--network my-overlay \

--replicas 1 \

--publish published=8080,target=80 \

nginx:latest

3 通过查看部分，运行docker network inspect my-overlay并验证my-nginx服务任务已连接到该任务Containers。

4 删除服务和网络。

$ docker service rm my-nginx

$ docker network rm my-overlay

### ### 对独立容器使用覆盖网络

此示例演示了DNS容器发现-具体地说，是如何使用覆盖网络在不同Docker守护程序上的独立容器之间进行通信。步骤如下：

在上host1，将节点初始化为群集（管理器）。

在上host2，将节点加入群集（工作人员）。

在上host1，创建一个可附加的覆盖网络（test-net）。

在上host1运行一个交互式高山容器（alpine1）test-net。

在上host2运行，并运行一个独立的交互式高山容器（alpine2）test-net。

在ping host1会话中进行。alpine1alpine2

#### 先决条件

对于此测试，您需要两个可以相互通信的不同Docker主机。每个主机必须具有Docker 17.06或更高版本，并且在两个Docker主机之间打开以下端口：

TCP端口2377

TCP和UDP端口7946

UDP端口4789

一种简单的设置方法是拥有两个VM（本地或在AWS等云提供商上），每个VM均已安装并运行Docker。如果您使用的是AWS或类似的云计算平台，最简单的配置是使用一个安全组，该安全组打开两个主机之间的所有传入端口以及来自客户端IP地址的SSH端口。

本示例将群中的两个节点称为host1和host2。此示例还使用Linux主机，但是相同的命令在Windows上也可以使用。

#### 演练

1 设置群。

一种。在上host1，初始化群集（如果出现提示，则用于--advertise-addr 为与群集中其他主机通信的接口指定IP地址，例如AWS上的私有IP地址）：

$ docker swarm init

Swarm initialized: current node (vz1mm9am11qcmo979tlrlox42) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join --token SWMTKN-1-5g90q48weqrtqryq4kj6ow0e8xm9wmv9o6vgqc5j320ymybd5c-8ex8j0bc40s6hgvy5ui5gl4gy 172.31.47.252:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

b。在上host2，按照上述说明加入群组：

$ docker swarm join --token <your\_token> <your\_ip\_address>:2377

This node joined a swarm as a worker.

如果节点无法加入群集，则docker swarm join命令超时。要解决此问题，请在docker swarm leave --force上运行host2，验证您的网络和防火墙设置，然后重试。

2 在上host1，创建一个名为的可连接覆盖网络test-net：

$ docker network create --driver=overlay --attachable test-net

uqsof8phj3ak0rq9k86zta6ht

请注意，返回的NETWORK ID-当您从连接时会再次看到它host2。

3 在上host1，启动一个交互式（-it）容器（alpine1），该容器连接到test-net：

$ docker run -it --name alpine1 --network test-net alpine

/ #

4 在上host2，列出可用的网络-注意test-net尚不存在：

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

ec299350b504 bridge bridge local

66e77d0d0e9a docker\_gwbridge bridge local

9f6ae26ccb82 host host local

omvdxqrda80z ingress overlay swarm

b65c952a4b2b none null local

5 在上host2，启动一个独立的（-d）和交互式（-it）容器（alpine2），该容器连接到test-net：

$ docker run -dit --name alpine2 --network test-net alpine

fb635f5ece59563e7b8b99556f816d24e6949a5f6a5b1fbd92ca244db17a4342

自动DNS容器发现仅适用于唯一的容器名称。

6 上host2，验证test-net被创建（和具有相同的网络ID为test-net上host1）：

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

...

uqsof8phj3ak test-net overlay swarm

7 在上host1，alpine2在以下交互式终端中ping alpine1：

/ # ping -c 2 alpine2

PING alpine2 (10.0.0.5): 56 data bytes

64 bytes from 10.0.0.5: seq=0 ttl=64 time=0.600 ms

64 bytes from 10.0.0.5: seq=1 ttl=64 time=0.555 ms

--- alpine2 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.555/0.577/0.600 ms

两个容器与连接两个主机的覆盖网络通信。如果您在另一个未分离的高山容器上运行host2，则可以从ping （在这里，我们添加了 remove选项以自动清除容器）：alpine1host2

$ docker run -it --rm --name alpine3 --network test-net alpine

/ # ping -c 2 alpine1

/ # exit

8 在上host1，关闭alpine1会话（这也会停止容器）：

/ # exit

9 清理您的容器和网络：

您必须在每个主机上独立停止和删除容器，因为Docker守护程序是独立运行的，并且它们是独立的容器。您只需要删除网络，host1因为当您停止 alpine2时host2，会test-net消失。

一种。在上host2，停止alpine2，检查是否test-net已删除，然后删除alpine2：

$ docker container stop alpine2

$ docker network ls

$ docker container rm alpine2

一种。在上host1，删除alpine1并test-net：

$ docker container rm alpine1

$ docker network rm test-net

### ### 在容器和集群服务之间进行通信

#### 先决条件

对于此示例，您需要Docker 17.06或更高版本。

#### 演练

在此示例中，您alpine在同一Docker主机上启动了两个不同的容器，并进行了一些测试以了解它们如何相互通信。您需要安装并运行Docker。

1 打开一个终端窗口。在执行其他任何操作之前，请先列出当前网络。如果您从未在此Docker守护程序上添加网络或初始化过集群，则应该看到以下内容。您可能会看到不同的网络，但至少应该看到以下内容（网络ID会有所不同）：

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

17e324f45964 bridge bridge local

6ed54d316334 host host local

7092879f2cc8 none null local

bridge列出了默认网络，以及host和none。后两个不是完全成熟的网络，但是用于启动直接连接到Docker守护程序主机的网络堆栈的容器，或者用于启动不具有网络设备的容器。本教程将把两个容器连接到bridge网络。

2 启动两个alpine运行的容器ash，这是Alpine的默认外壳程序，而不是bash。这些-dit标志意味着以分离的方式（在后台），交互的（具有输入能力）和TTY（以便您可以看到输入和输出）启动容器。由于您是分开启动的，因此您不会立即连接到容器。而是将打印容器的ID。由于未指定任何 --network标志，因此容器将连接到默认bridge网络。

$ docker run -dit --name alpine1 alpine ash

$ docker run -dit --name alpine2 alpine ash

检查两个容器是否确实已启动：

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

602dbf1edc81 alpine "ash" 4 seconds ago Up 3 seconds alpine2

da33b7aa74b0 alpine "ash" 17 seconds ago Up 16 seconds alpine1

3 检查bridge网络以查看连接了哪些容器。

$ docker network inspect bridge

[

{

"Name": "bridge",

"Id": "17e324f459648a9baaea32b248d3884da102dde19396c25b30ec800068ce6b10",

"Created": "2017-06-22T20:27:43.826654485Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Containers": {

"602dbf1edc81813304b6cf0a647e65333dc6fe6ee6ed572dc0f686a3307c6a2c": {

"Name": "alpine2",

"EndpointID": "03b6aafb7ca4d7e531e292901b43719c0e34cc7eef565b38a6bf84acf50f38cd",

"MacAddress": "02:42:ac:11:00:03",

"IPv4Address": "172.17.0.3/16",

"IPv6Address": ""

},

"da33b7aa74b0bf3bda3ebd502d404320ca112a268aafe05b4851d1e3312ed168": {

"Name": "alpine1",

"EndpointID": "46c044a645d6afc42ddd7857d19e9dcfb89ad790afb5c239a35ac0af5e8a5bc5",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"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": "1500"

},

"Labels": {}

}

]

在顶部附近，bridge列出了有关网络的信息，包括Docker主机和bridge 网络之间的网关的IP地址（172.17.0.1）。在该Containers键下，列出了每个已连接的容器以及有关其IP地址的信息（172.17.0.2for alpine1和172.17.0.3for alpine2）。

4 容器在后台运行。使用docker attach 命令连接到alpine1。

$ docker attach alpine1

/ #

提示更改为#指示您是root容器中的用户。使用ip addr show命令显示网络接口alpine1在容器中的外观：

# ip addr show

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

27: eth0@if28: <BROADCAST,MULTICAST,UP,LOWER\_UP,M-DOWN> mtu 1500 qdisc noqueue state UP

link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff

inet 172.17.0.2/16 scope global eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::42:acff:fe11:2/64 scope link

valid\_lft forever preferred\_lft forever

第一个接口是回送设备。现在忽略它。请注意，第二个接口具有IP地址172.17.0.2，该地址alpine1与上一步中显示的地址相同。

5 从内部alpine1，确保可以通过ping连接到Internet google.com。该-c 2标志将命令限制两次两次ping 。

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes

64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.841 ms

64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.897 ms

--- google.com ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 9.841/9.869/9.897 ms

6 现在尝试ping第二个容器。首先，通过其IP地址ping它 172.17.0.3：

# ping -c 2 172.17.0.3

PING 172.17.0.3 (172.17.0.3): 56 data bytes

64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.086 ms

64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.094 ms

--- 172.17.0.3 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.086/0.090/0.094 ms

这样成功了。接下来，尝试alpine2按容器名称ping 容器。这将失败。

# ping -c 2 alpine2

ping: bad address 'alpine2'

7 分离从alpine1没有通过使用分离的序列，停止它 CTRL+ p CTRL+ q（按住CTRL和类型p，随后q）。如果你愿意，重视alpine2并重复步骤4，5和6出现，取代alpine1了alpine2。

8 停止并卸下两个容器。

$ docker container stop alpine1 alpine2

$ docker container rm alpine1 alpine2

请记住，bridge不建议将默认网络用于生产。

## ## 使用Macvlan网络联网

本系列教程讨论了连接到macvlan网络的联网独立容器。在这种类型的网络中，Docker主机在其IP地址上接受对多个MAC地址的请求，并将这些请求路由到适当的容器。有关其他联网主题，请参见 概述。

### ### 目标

这些教程的目标是建立桥接macvlan网络并将容器连接到该网络，然后建立802.1q中继macvlan网络并将容器连接到该网络。

### ### 先决条件

大多数云提供商阻止macvlan网络。您可能需要物理访问网络设备。

macvlan网络驱动程序仅适用于Linux主机，而Mac的Docker桌面，Windows的Docker桌面或Windows Server的Docker EE不支持该网络驱动程序。

您至少需要3.9版的Linux内核，并建议使用4.0版或更高版本。

这些示例假定您的以太网接口是eth0。如果您的设备使用其他名称，请改用该名称。

### ### 桥梁的例子

在简单的网桥示例中，您的流量通过eth0，Docker使用其MAC地址将流量路由到您的容器。对于网络上的设备，您的容器似乎已物理连接到网络。

1 创建一个macvlan名为的网络my-macvlan-net。将subnet，gateway和parent值修改为您的环境中有意义的值。

$ docker network create -d macvlan \

--subnet=172.16.86.0/24 \

--gateway=172.16.86.1 \

-o parent=eth0 \

my-macvlan-net

您可以使用docker network ls和docker network inspect my-macvlan-net 命令来验证网络是否存在并且是macvlan网络。

2 启动一个alpine容器并将其附加到my-macvlan-net网络。这些 -dit标志在后台启动容器，但允许您附加到它。该--rm标志表示容器在停止时被移除。

$ docker run --rm -dit \

--network my-macvlan-net \

--name my-macvlan-alpine \

alpine:latest \

ash

3 检查my-macvlan-alpine容器并注意MacAddress钥匙内的Networks钥匙：

$ docker container inspect my-macvlan-alpine

...truncated...

"Networks": {

"my-macvlan-net": {

"IPAMConfig": null,

"Links": null,

"Aliases": [

"bec64291cd4c"

],

"NetworkID": "5e3ec79625d388dbcc03dcf4a6dc4548644eb99d58864cf8eee2252dcfc0cc9f",

"EndpointID": "8caf93c862b22f379b60515975acf96f7b54b7cf0ba0fb4a33cf18ae9e5c1d89",

"Gateway": "172.16.86.1",

"IPAddress": "172.16.86.2",

"IPPrefixLen": 24,

"IPv6Gateway": "",

"GlobalIPv6Address": "",

"GlobalIPv6PrefixLen": 0,

"MacAddress": "02:42:ac:10:56:02",

"DriverOpts": null

}

}

...truncated

4 通过运行几个docker exec命令来检查容器如何看到自己的网络接口。

$ docker exec my-macvlan-alpine ip addr show eth0

9: eth0@tunl0: <BROADCAST,MULTICAST,UP,LOWER\_UP,M-DOWN> mtu 1500 qdisc noqueue state UP

link/ether 02:42:ac:10:56:02 brd ff:ff:ff:ff:ff:ff

inet 172.16.86.2/24 brd 172.16.86.255 scope global eth0

valid\_lft forever preferred\_lft forever

$ docker exec my-macvlan-alpine ip route

default via 172.16.86.1 dev eth0

172.16.86.0/24 dev eth0 scope link src 172.16.86.2

5 停止容器（Docker会由于该--rm标志而将其删除），然后删除网络。

$ docker container stop my-macvlan-alpine

$ docker network rm my-macvlan-net

### ### 802.1q中继网桥示例

在802.1q中继网桥示例中，您的流量通过eth0（称为eth0.10）的子接口流动，并且Docker使用其MAC地址将流量路由到您的容器。对于网络上的设备，您的容器似乎已物理连接到网络。

1 创建一个macvlan名为的网络my-8021q-macvlan-net。将subnet，gateway和parent值修改为 您的环境中有意义的值。

$ docker network create -d macvlan \

--subnet=172.16.86.0/24 \

--gateway=172.16.86.1 \

-o parent=eth0.10 \

my-8021q-macvlan-net

您可以使用docker network ls和docker network inspect my-8021q-macvlan-net 命令来验证网络是否存在，是否为macvlan网络以及是否具有parent eth0.10。您可以ip addr show在Docker主机上使用来验证接口是否eth0.10存在并具有单独的IP地址

2 启动一个alpine容器并将其附加到my-8021q-macvlan-net 网络。这些-dit标志在后台启动容器，但允许您附加到它。该--rm标志表示容器在停止时被移除。

$ docker run --rm -itd \

--network my-8021q-macvlan-net \

--name my-second-macvlan-alpine \

alpine:latest \

ash

3 检查my-second-macvlan-alpine容器并注意MacAddress 钥匙内的Networks钥匙：

$ docker container inspect my-second-macvlan-alpine

...truncated...

"Networks": {

"my-8021q-macvlan-net": {

"IPAMConfig": null,

"Links": null,

"Aliases": [

"12f5c3c9ba5c"

],

"NetworkID": "c6203997842e654dd5086abb1133b7e6df627784fec063afcbee5893b2bb64db",

"EndpointID": "aa08d9aa2353c68e8d2ae0bf0e11ed426ea31ed0dd71c868d22ed0dcf9fc8ae6",

"Gateway": "172.16.86.1",

"IPAddress": "172.16.86.2",

"IPPrefixLen": 24,

"IPv6Gateway": "",

"GlobalIPv6Address": "",

"GlobalIPv6PrefixLen": 0,

"MacAddress": "02:42:ac:10:56:02",

"DriverOpts": null

}

}

...truncated

4 通过运行几个docker exec命令来检查容器如何看到自己的网络接口。

$ docker exec my-second-macvlan-alpine ip addr show eth0

11: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER\_UP,M-DOWN> mtu 1500 qdisc noqueue state UP

link/ether 02:42:ac:10:56:02 brd ff:ff:ff:ff:ff:ff

inet 172.16.86.2/24 brd 172.16.86.255 scope global eth0

valid\_lft forever preferred\_lft forever

$ docker exec my-second-macvlan-alpine ip route

default via 172.16.86.1 dev eth0

172.16.86.0/24 dev eth0 scope link src 172.16.86.2

5 停止容器（Docker会由于该--rm标志而将其删除），然后删除网络。

$ docker container stop my-second-macvlan-alpine

$ docker network rm my-8021q-macvlan-net

## ## 守护--启用IPv6支持

必须先在Docker守护程序中启用IPv6支持，然后才能在Docker容器或群集服务中使用IPv6。之后，您可以选择对任何容器，服务或网络使用IPv4或IPv6（或两者）。

注意：仅在Linux主机上运行的Docker守护程序支持IPv6网络。

1 编辑密钥/etc/docker/daemon.json并将其设置ipv6为true。

{

"ipv6": true

}

保存文件。

2 重新加载Docker配置文件。

$ systemctl reload docker

现在，您可以使用该--ipv6标志创建网络，并使用该--ip6标志分配容器IPv6地址。

## ## 守护--Docker和iptables

在Linux上，Docker操纵iptables规则以提供网络隔离。这是一个实现细节，您不应修改Docker插入iptables策略中的规则。

### ### 在Docker的规则之前添加iptables策略

Docker的所有iptables规则都已添加到DOCKER链中。不要手动操作该表。如果您需要添加在Docker规则之前加载的规则，请将它们添加到DOCKER-USER链中。在Docker自动创建任何规则之前，将加载这些规则。

### ### 限制与Docker守护程序的连接

默认情况下，允许所有外部源IP连接到Docker守护程序。要仅允许特定的IP或网络访问容器，请在DOCKER过滤器链的顶部插入一个否定的规则。例如，以下规则将外部访问限制为除192.168.1.1之外的所有IP地址：

$ iptables -I DOCKER-USER -i ext\_if ! -s 192.168.1.1 -j DROP

请注意，您将需要更改ext\_if以与主机的实际外部接口相对应。您可以改为允许来自源子网的连接。以下规则仅允许从子网192.168.1.0/24访问：

$ iptables -I DOCKER-USER -i ext\_if ! -s 192.168.1.0/24 -j DROP

最后，您可以指定要接受的IP地址范围--src-range （请记住-m iprange在使用--src-range或时也要添加--dst-range）：

$ iptables -I DOCKER-USER -m iprange -i ext\_if ! --src-range 192.168.1.1-192.168.1.3 -j DROP

您可以结合-s或--src-range与-d或--dst-range一起控制源和目标。例如，如果Docker守护程序同时监听192.168.1.99和10.1.2.3，则可以制定特定于10.1.2.3并保持 192.168.1.99开放的规则。

iptables是复杂的，更复杂的规则不在此主题范围内。有关 更多信息，请参见Netfilter.org HOWTO。

### ### 防止Docker操作iptables

可以在Docker引擎的配置文件中将iptables密钥设置false为/etc/docker/daemon.json，但是此选项不适用于大多数用户。完全阻止Docker创建iptables规则是不可能的，事后创建规则非常复杂，超出了这些说明的范围。设置iptables为false很有可能会破坏Docker引擎的容器网络。

## ## 守护—容器网络

容器使用的网络类型，无论是网桥， 覆盖，macvlan网络还是自定义网络插件，在容器内部都是透明的。从容器的角度来看，它具有一个带有IP地址，网关，路由表，DNS服务和其他网络详细信息的网络接口（假定容器未使用none网络驱动程序）。从容器的角度来看，本主题与网络相关。

### ### 发布的端口

默认情况下，创建容器时，它不会将其任何端口发布到外界。要使端口可用于Docker外部的服务或未连接到容器网络的Docker容器，请使用 --publish或-p标志。这将创建一个防火墙规则，该规则将容器端口映射到Docker主机上的端口。这里有些例子。

| **标志值** | **描述** |
| --- | --- |
| -p 8080:80 | 将容器中的TCP端口80映射到Docker主机上的端口8080。 |
| -p 192.168.1.100:8080:80 | 将容器中的TCP端口80映射到Docker主机上的端口8080，以连接到主机IP 192.168.1.100。 |
| -p 8080:80/udp | 将容器中的UDP端口80映射到Docker主机上的端口8080。 |
| -p 8080:80/tcp -p 8080:80/udp | 将容器中的TCP端口80映射到Docker主机上的TCP端口8080，并将容器中的UDP端口80映射到Docker主机上的UDP端口8080。 |

### ### IP地址和主机名

默认情况下，为容器连接到的每个Docker网络分配一个IP地址。IP地址是从分配给网络的池中分配的，因此Docker守护程序实际上充当了每个容器的DHCP服务器。每个网络还具有默认的子网掩码和网关。

容器启动时，只能使用将该容器连接到单个网络 --network。但是，您可以使用将运行中的容器连接到多个网络docker network connect。使用--network标志启动容器时 ，可以使用--ip或--ip6标志指定分配给该网络上的容器的IP地址。

使用将现有容器连接到其他网络时 docker network connect，可以使用该命令上的--ip或--ip6标志来指定其他网络上容器的IP地址。

同样，容器的主机名默认为Docker中容器的ID。您可以使用覆盖主机名--hostname。当使用来连接到现有网络时docker network connect，可以使用该--alias 标志为该网络上的容器指定其他网络别名。

### ### DNS服务

默认情况下，容器会继承Docker守护程序的DNS设置，包括/etc/hosts和/etc/resolv.conf。您可以在每个容器的基础上覆盖这些设置。

| **旗** | **描述** |
| --- | --- |
| --dns | DNS服务器的IP地址。要指定多个DNS服务器，请使用多个--dns标志。如果容器无法访问您指定的任何IP地址，8.8.8.8则会添加Google的公共DNS服务器，以便您的容器可以解析Internet域。 |
| --dns-search | 一个DNS搜索域，用于搜索非完全限定的主机名。要指定多个DNS搜索前缀，请使用多个--dns-search标志。 |
| --dns-opt | 代表DNS选项及其值的键/值对。有关resolv.conf有效选项，请参阅操作系统的文档。 |
| --hostname | 容器本身使用的主机名。如果未指定，则默认为容器的ID。 |

## ## 守护--配置Docker以使用代理服务器

如果您的容器需要使用HTTP，HTTPS或FTP代理服务器，则可以通过不同的方式对其进行配置：

在Docker 17.07及更高版本中，您可以 配置Docker客户端以将代理信息自动传递到容器。

在Docker 17.06及更低版本中，您必须 在容器内设置适当的环境变量。您可以在构建镜像时执行此操作（这会降低镜像的可移植性），或者在创建或运行容器时执行此操作。

### ### 配置Docker客户端

在Docker客户端上，~/.docker/config.json在启动容器的用户的主目录中创建或编辑文件。添加如下所示的JSON，用httpsProxy或替换代理的类型，并ftpProxy在必要时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。

通过将noProxy密钥设置为一个或多个逗号分隔的IP地址或主机，您可以选择排除主机或范围通过代理服务器。\*如本示例所示，支持将字符用作通配符。

{

"proxies":

{

"default":

{

"httpProxy": "http://127.0.0.1:3001",

"httpsProxy": "http://127.0.0.1:3001",

"noProxy": "\*.test.example.com,.example2.com"

}

}

}

保存文件。

创建或启动新容器时，环境变量将在容器内自动设置。

### ### 使用环境变量

手动设置环境变量

在生成镜像时，或者--env在创建或运行容器时使用标志时，可以将以下一个或多个变量设置为适当的值。此方法使镜像的可移植性降低，因此，如果您具有Docker 17.07或更高版本，则应配置Docker客户端 。

| **变量** | **变量** | **Dockerfile示例** | **docker run 例** |
| --- | --- | --- | --- |
| HTTP\_PROXY | HTTP\_PROXY | ENV HTTP\_PROXY "http://127.0.0.1:3001" | --env HTTP\_PROXY="http://127.0.0.1:3001" |
| HTTPS\_PROXY | HTTPS\_PROXY | ENV HTTPS\_PROXY "https://127.0.0.1:3001" | --env HTTPS\_PROXY="https://127.0.0.1:3001" |
| FTP\_PROXY | FTP\_PROXY | ENV FTP\_PROXY "ftp://127.0.0.1:3001" | --env FTP\_PROXY="ftp://127.0.0.1:3001" |
| NO\_PROXY | NO\_PROXY | ENV NO\_PROXY "\*.test.example.com,.example2.com" | --env NO\_PROXY="\*.test.example.com,.e |

## ## Swarm Classic--独立群集

本文仅适用于需要在Docker上使用独立群而不是群模式的用户。独立群集（有时称为Swarm Classic）依赖于外部键值存储来存储网络信息。Docker群集模式将网络信息存储在群集管理器上的Raft日志中。如果使用群集模式，请参阅 [群集模式网络](https://docs.docker.com/engine/swarm/networking/)而不是本文。

Universal Control Plane的用户**确实**使用了外部键值存储，但是UCP会为您管理它，并且您无需手动干预。如果您在键值存储方面遇到问题，请参阅 [对etcd键值存储进行故障排除](https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-configurations/#troubleshoot-the-etcd-key-value-store)

如果您使用的是独立集群而不是UCP，则本文可能对您有用。本文使用一个示例来说明使用独立群集和overlay网络驱动程序创建多主机网络的基础。与bridge网络不同，覆盖网络在创建一个网络之前需要先存在一些条件：

### ### 使用外部键值存储覆盖网络

要将Docker与外部键值存储一起使用，您需要满足以下条件：

* 访问键值存储。Docker支持Consul，Etcd和ZooKeeper（分布式存储）键值存储。本示例使用Consul。
* 与键值存储连接的主机集群。
* Docker在集群中的每个主机上运行。
* 集群中的主机必须具有唯一的主机名，因为键值存储使用主机名来标识集群成员。

并非必须使用Docker Machine和Docker Swarm才能通过键值存储体验Docker多主机网络。但是，此示例使用它们来说明它们是如何集成的。您可以使用Machine通过独立的群集来创建键值存储服务器和主机群集。

**注意**：这些示例与以群体模式运行的Docker不相关，并且在这种配置下不起作用。

### ### 先决条件

在开始之前，请确保您的网络上已经安装了最新版本的Docker和Docker Machine。该示例还依赖于VirtualBox。如果您使用Docker Toolbox安装在Mac或Windows上，则已经安装了所有这些工具。

如果尚未执行此操作，请确保将Docker和Docker Machine升级到最新版本。

### ### 设置键值存储

覆盖网络需要键值存储。键值存储保存有关网络状态的信息，包括发现，网络，端点，IP地址等。Docker支持Consul，Etcd和ZooKeeper键值存储。本示例使用Consul。

1 登录到已安装Docker和Docker Machine的系统。

2 预配一个名为VirtualBox的计算机mh-keystore。

$ docker-machine create -d virtualbox mh-keystore

当您配置新计算机时，该过程会将Docker添加到主机。这意味着您可以使用Docker Hub中的consul镜像来创建实例，而不是手动安装Consul 。您可以在下一步中执行此操作。

3 将本地环境设置为mh-keystore计算机。

$ eval "$(docker-machine env mh-keystore)"

4 启动consul在mh-keystoreDocker计算机上运行的容器。

$ docker run -d \

--name consul \

-p "8500:8500" \

-h "consul" \

consul agent -server -bootstrap -client "0.0.0.0"

客户端启动consul在mh-keystoreDocker计算机上运行的镜像 。服务器被调用consul并在port上监听8500。

5 运行docker ps命令以查看consul容器。

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a47492d6c4d1 consul "docker-entrypoint..." 2 seconds ago Up 1 second 8300-8302/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp consul

保持终端处于打开状态，然后继续 创建“群集群”。

### ### 创建集群

在此步骤中，您将docker-machine用于为网络配置主机。您实际上尚未创建网络。您在VirtualBox中创建了几台Docker计算机。其中一台机器充当群组管理器，然后您首先创建它。创建每个主机时，需要在overlay网络驱动程序所需的该机器选项上传递Docker守护程序。

注意：这将创建一个独立的群集集群，而不是在群集模式下使用Docker。这些示例与以群体模式运行的Docker不相关，并且在这种配置下不起作用。

1 创建一个群管理器。

$ docker-machine create \

-d virtualbox \

--swarm --swarm-master \

--swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \

--engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore):8500" \

--engine-opt="cluster-advertise=eth1:2376" \

mhs-demo0

在创建时，您将为Docker守护程序提供该--cluster-store 选项。此选项告诉引擎overlay网络的键值存储的位置。bash扩展$(docker-machine ip mh-keystore) 解析为您在“ STEP 1”中创建的Consul服务器的IP地址。该 --cluster-advertise选件在网络上发布计算机。

2 创建另一个主机并将其添加到群集。

$ docker-machine create -d virtualbox \

--swarm \

--swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \

--engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore):8500" \

--engine-opt="cluster-advertise=eth1:2376" \

mhs-demo1

3 列出您的Docker机器以确认它们都已启动并正在运行。

$ docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM

default - virtualbox Running tcp://192.168.99.100:2376

mh-keystore \* virtualbox Running tcp://192.168.99.103:2376

mhs-demo0 - virtualbox Running tcp://192.168.99.104:2376 mhs-demo0 (master)

mhs-demo1 - virtualbox Running tcp://192.168.99.105:2376 mhs-demo0

此时，您的网络上已运行了一组主机。您已准备好使用这些主机为容器创建多主机网络。

让您的终端保持打开状态，然后继续 创建覆盖网络。

### ### 创建覆盖网络

创建覆盖网络：

1 将docker环境设置为swarm管理器。

$ eval $(docker-machine env --swarm mhs-demo0)

使用--swarm带有标志将docker-machine限制docker 命令仅聚集信息。

2 使用docker info命令查看群。

$ docker info

Containers: 3

Images: 2

Role: primary

Strategy: spread

Filters: affinity, health, constraint, port, dependency

Nodes: 2

mhs-demo0: 192.168.99.104:2376

└ Containers: 2

└ Reserved CPUs: 0 / 1

└ Reserved Memory: 0 B / 1.021 GiB

└ Labels: executiondriver=native-0.2, kernelversion=4.1.10-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 4187d2c - Wed Oct 14 14:00:28 UTC 2015, provider=virtualbox, storagedriver=aufs

mhs-demo1: 192.168.99.105:2376

└ Containers: 1

└ Reserved CPUs: 0 / 1

└ Reserved Memory: 0 B / 1.021 GiB

└ Labels: executiondriver=native-0.2, kernelversion=4.1.10-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 4187d2c - Wed Oct 14 14:00:28 UTC 2015, provider=virtualbox, storagedriver=aufs

CPUs: 2

Total Memory: 2.043 GiB

Name: 30438ece0915

此输出表明您正在管理器上运行三个容器和两个镜像。

3 创建您的overlay网络。

$ docker network creat

您只需要在群集中的单个主机上创建网络。在这种情况下，您使用了群集管理器，但是可以轻松地在群集中的任何主机上运行它。

注意：强烈建议--subnet在创建网络时使用该选项。如果--subnet未指定，则Docker会自动为网络选择并分配一个子网，并且该子网可能与基础架构中不受Docker管理的另一个子网重叠。当容器连接到该网络时，此类重叠可能导致连接问题或故障。

4 检查网络是否存在：

$ docker network ls

NETWORK ID NAME DRIVER

412c2496d0eb mhs-demo1/host host

dd51763e6dd2 mhs-demo0/bridge bridge

6b07d0be843f my-net overlay

b4234109bd9b mhs-demo0/none null

1aeead6dd890 mhs-demo0/host host

d0bb78cbe7bd mhs-demo1/bridge bridge

1c0eb8f69ebb mhs-demo1/none null

由于您处于swarm管理器环境中，因此您将看到所有swarm参与者上的所有网络：每个Docker守护程序上的默认网络和单个覆盖网络。每个网络都有一个唯一的ID和一个命名空间的名称。

5 依次切换到每个群集代理并列出网络。

$ eval $(docker-machine env mhs-demo0)

$ docker network ls

NETWORK ID NAME DRIVER

6b07d0be843f my-net overlay

dd51763e6dd2 bridge bridge

b4234109bd9b none null

1aeead6dd890 host host

$ eval $(docker-machine env mhs-demo1)

$ docker network ls

NETWORK ID NAME DRIVER

d0bb78cbe7bd bridge bridge

1c0eb8f69ebb none null

412c2496d0eb host host

6b07d0be843f my-net overlay

两个代理都报告他们具有ID 的my-net网络6b07d0be843f。现在，您正在运行一个多主机容器网络！

### ### 在网络上运行应用程序

创建网络后，您可以在任何主机上启动一个容器，它自动成为网络的一部分。

1 将环境设置为群集管理器。

$ eval $(docker-machine env --swarm mhs-demo0)

2 在mhs-demo0实例上启动Nginx Web服务器。

$ docker run -itd \

--name=web \

--network=my-net \

--env="constraint:node==mhs-demo0" \

nginx:alpine

3 在该busybox实例上运行一个实例，mhs-demo1并获取Nginx服务器主页的内容。

$ docker run -it --rm \

--network=my-net \

--env="constraint:node==mhs-demo1" \

busybox wget -O- http://web

Unable to find image 'busybox:latest' locally

latest: Pulling from library/busybox

ab2b8a86ca6c: Pull complete

2c5ac3f849df: Pull complete

Digest: sha256:5551dbdfc48d66734d0f01cafee0952cb6e8eeecd1e2492240bf2fd9640c2279

Status: Downloaded newer image for busybox:latest

Connecting to web (10.0.9.2:80)

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

<style>

body {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</style>

</head>

<body>

<h1>Welcome to nginx!</h1>

<p>If you see this page, the nginx web server is successfully installed and

working. Further configuration is required.</p>

<p>For online documentation and support, refer to

<a href="http://nginx.org/">nginx.org</a>.<br/>

Commercial support is available at

<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>

</body>

</html>

- 100% |\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*| 612 0:00:00 ETA

### ### 检查外部连接

如您所见，Docker的内置覆盖网络驱动程序提供了同一网络内多个主机上的容器之间的开箱即用的连接性。此外，连接到多主机网络的容器会自动连接到docker\_gwbridge网络。该网络允许容器在其群集之外具有外部连接。

1 将环境更改为群集代理。

$ eval $(docker-machine env mhs-demo1)

2 docker\_gwbridge通过列出网络来查看网络。

$ docker network ls

NETWORK ID NAME DRIVER

6b07d0be843f my-net overlay

dd51763e6dd2 bridge bridge

b4234109bd9b none null

1aeead6dd890 host host

e1dbd5dff8be docker\_gwbridge bridge

3 在群管理器上重复步骤1和2。

$ eval $(docker-machine env mhs-demo0)

$ docker network ls

NETWORK ID NAME DRIVER

6b07d0be843f my-net overlay

d0bb78cbe7bd bridge bridge

1c0eb8f69ebb none null

412c2496d0eb host host

97102a22e8d2 docker\_gwbridge bridge

4 检查Nginx容器的网络接口。

$ docker container exec web ip addr

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

22: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 qdisc noqueue state UP group default

link/ether 02:42:0a:00:09:03 brd ff:ff:ff:ff:ff:ff

inet 10.0.9.2/24 scope global eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::42:aff:fe00:903/64 scope link

valid\_lft forever preferred\_lft forever

24: eth1: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default

link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff

inet 172.18.0.2/16 scope global eth1

valid\_lft forever preferred\_lft forever

inet6 fe80::42:acff:fe12:2/64 scope link

valid\_lft forever preferred\_lft forever

该eth0接口表示连接到my-net覆盖网络的容器接口。虽然eth1接口代表连接到所述容器的接口docker\_gwbridge网络。

# # 9 管理应用数据

## ## 存储概述

### ### 在Docker中管理数据

默认情况下，在容器内创建的所有文件都存储在可写容器层上。这意味着：

当该容器不再存在时，数据将不会持久保存，并且如果另一个进程需要它，则可能很难从容器中取出数据。

容器的可写层与运行容器的主机紧密耦合。您不能轻易地将数据移动到其他地方。

写入容器的可写层需要 存储驱动程序来管理文件系统。存储驱动程序使用Linux内核提供联合文件系统。与使用直接写入主机文件系统的数据卷相比，这种额外的抽象降低了性能 。

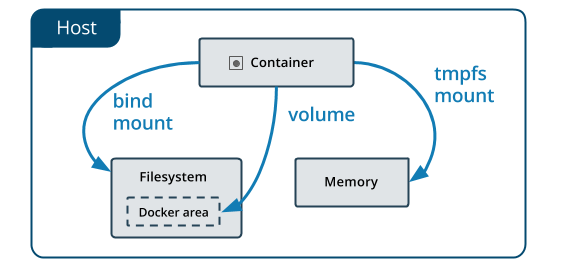
Docker为容器提供了两个选项来将文件存储在主机中，以便即使容器停止后文件也可以持久存储：卷和 绑定安装。如果您在Linux上运行Docker，则还可以使用tmpfs挂载。如果您在Windows上运行Docker，则还可以使用命名管道。

继续阅读有关这两种持久数据方式的更多信息。

#### 选择正确的安装类型

无论您选择使用哪种类型的安装，容器中的数据看起来都是相同的。它在容器的文件系统中显示为目录或单个文件。

可视化卷，绑定挂载和tmpfs 挂载之间差异的一种简单方法是考虑数据在Docker主机上的位置。



1 卷存储在主机文件系统的一部分中，该文件系统由Docker管理（/var/lib/docker/volumes/在Linux上）。非Docker进程不应修改文件系统的这一部分。卷是在Docker中持久保存数据的最佳方法。

2 绑定挂载可以存储在主机系统上的任何位置。它们甚至可能是重要的系统文件或目录。Docker主机或Docker容器上的非Docker进程可以随时对其进行修改。

3 tmpfs挂载仅存储在主机系统的内存中，并且永远不会写入主机系统的文件系统中。

### ### 有关安装类型的更多详细信息

1 卷：由Docker创建和管理。您可以使用docker volume create命令显式创建卷，或者Docker可以在容器或服务创建期间创建卷。

创建卷时，它存储在Docker主机上的目录中。将卷装入容器时，此目录就是装入容器的目录。这类似于绑定安装的工作方式，除了卷由Docker管理并且与主机的核心功能隔离。

给定的volume可以同时安装到多个容器中。当没有正在运行的容器使用卷时，该卷仍可用于Docker，并且不会自动删除。您可以使用删除未使用的卷docker volume prune。

挂载卷时，它可以命名为或匿名。匿名卷首次安装到容器中时，不会为其指定明确的名称，因此Docker为它们提供一个随机名称，该名称在给定的Docker主机中保证是唯一的。除了名称之外，命名卷和匿名卷的行为也相同。

卷还支持使用卷驱动程序，该驱动程序使您可以将数据存储在远程主机或云提供商上，以及其他可能性。

2 绑定挂载：自Docker早期以来可用。与卷相比，绑定安装的功能有限。使用绑定安装时，主机上的文件或目录将安装到容器中。文件或目录由主机上的完整路径引用。该文件或目录不需要在Docker主机上已经存在。如果尚不存在，则按需创建。绑定挂载性能非常好，但是它们依赖于具有特定目录结构的主机文件系统。如果要开发新的Docker应用程序，请考虑使用命名卷。您不能使用Docker CLI命令直接管理绑定安装。

绑定安装允许访问敏感文件

使用绑定挂载的好与坏的一个副作用是，您可以通过容器中运行的进程来更改主机文件系统 ，包括创建，修改或删除重要的系统文件或目录。这是一项强大的功能，可能会带来安全隐患，包括影响主机系统上的非Docker进程。

3 tmpfs mounts：tmpfs挂载不会持久化在磁盘上，无论是在Docker主机上还是在容器内。容器在其生存期内可以使用它来存储非持久状态或敏感信息。例如，在内部，群集服务使用tmpfs安装将机密安装到服务的容器中。

4 命名管道：npipe 挂载可用于Docker主机与容器之间的通信。常见用例是在容器内运行第三方工具，并使用命名管道连接到Docker Engine API。

绑定安装和卷都可以使用-v或 --volume标志安装到容器中，但是两者的语法略有不同。对于tmpfs 安装，您可以使用该--tmpfs标志。但是，在Docker 17.06及更高版本中，建议将--mount标志用于容器和服务，用于绑定安装，卷或tmpfs安装，因为语法更清晰。

### ### 卷的好用例

卷是将数据持久保存在Docker容器和服务中的首选方法。卷的一些用例包括：

1 在多个运行中的容器之间共享数据。如果未显式创建卷，则在首次将卷安装到容器中时将创建该卷。当该容器停止或卸下时，该卷仍然存在。多个容器可以同时装载相同的卷（可读写或只读）。仅在显式删除卷时才将它们删除。

2 不保证Docker主机具有给定的目录或文件结构时。卷可帮助您将Docker主机的配置与容器运行时解耦。

3 当您要将容器的数据存储在远程主机或云提供商上时，而不是在本地。

4 当您需要将数据从一个Docker主机备份，还原或迁移到另一个Docker主机时，卷是一个更好的选择。您可以停止使用该卷的容器，然后备份该卷的目录（例如/var/lib/docker/volumes/<volume-name>）。

### ### 绑定安装的好用例

通常，应尽可能使用卷。绑定安装适用于以下类型的用例：

1 将配置文件从主机共享到容器。默认情况下，这就是Docker通过/etc/resolv.conf从主机安装到每个容器的方式为容器提供DNS解析的方式 。

2 在Docker主机上的开发环境和容器之间共享源代码或构建工件。例如，您可以将Maven target/ 目录安装到容器中，并且每次在Docker主机上构建Maven项目时，容器都可以访问重建的工件。

如果您以这种方式使用Docker进行开发，那么您的生产Dockerfile会将生产就绪的工件直接复制到镜像中，而不是依赖于绑定安装。

3 当确保Docker主机的文件或目录结构与容器所需的绑定挂载一致时。

### ### tmpfs挂载的好用例

tmpfs当您不希望数据在主机上或容器内持久存在时，最好使用mount。当您的应用程序需要写入大量非持久状态数据时，这可能是出于安全原因或为了保护容器的性能。

使用绑定安装或卷的提示

如果使用绑定安装或卷，请牢记以下几点：

如果将空卷装入存在文件或目录的容器中的目录中，则这些文件或目录将传播（复制）到该卷中。同样，如果启动一个容器并指定一个尚不存在的卷，则会为您创建一个空卷。这是预填充另一个容器所需数据的好方法。

如果将绑定安装或非空卷安装到存在某些文件或目录的容器中的目录中，则这些文件或目录会被安装遮盖，就像您将文件保存到/mntLinux主机上然后再安装USB驱动器插入/mnt。的内容/mnt将由USB驱动器，直至USB驱动器的内容被遮蔽被卸载。被遮盖的文件不会被删除或更改，但是在安装绑定安装或卷时将无法访问。

## ## 卷数

### ### 使用量

卷是用于持久化由Docker容器生成和使用的数据的首选机制。尽管绑定挂载取决于主机的目录结构，但是卷完全由Docker管理。与绑定安装相比，卷具有几个优点：

与绑定安装相比，卷更易于备份或迁移。

您可以使用Docker CLI命令或Docker API管理卷。

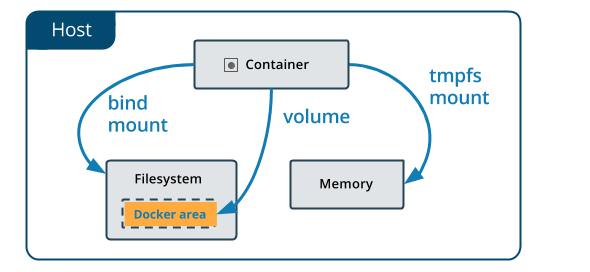
卷在Linux和Windows容器上均可工作。

可以在多个容器之间更安全地共享卷。

卷驱动程序使您可以将卷存储在远程主机或云提供商上，以加密卷内容或添加其他功能。

新卷的内容可以由容器预先填充。

此外，与将数据持久保存在容器的可写层中相比，卷通常是更好的选择，因为卷不会增加使用卷的容器的大小，并且卷的内容存在于给定容器的生命周期之外。



如果您的容器生成非持久状态数据，请考虑使用 tmpfs挂载以避免将数据永久存储在任何地方，并通过避免写入容器的可写层来提高容器的性能。

卷使用rprivate绑定传播，并且绑定传播无法为卷配置。

### ### 选择-v或--mount标志

最初，-vor --volume标志用于独立容器，而--mount标志用于集群服务。但是，从Docker 17.06开始，您还可以使用--mount独立容器。总的来说， --mount是更明确和冗长的。最大的区别是该-v 语法将所有选项合并在一个字段中，而--mount 语法将它们分开。这是每个标志的语法比较。

新用户应该尝试--mount比--volume语法更简单的语法。

如果需要指定volume驱动程序选项，则必须使用--mount。

1 -v或--volume：由三个字段组成，以冒号（:）分隔。这些字段必须以正确的顺序排列，并且每个字段的含义不是立即显而易见的。

对于命名卷，第一个字段是卷的名称，在给定的主机上是唯一的。对于匿名卷，将省略第一个字段。

第二个字段是文件或目录在容器中的安装路径。

第三个字段是可选的，并且是逗号分隔的选项列表，例如ro。这些选项将在下面讨论。

2 --mount：包含多个键值对，以逗号分隔，每个键值对都由一个<key>=<value>元组组成。该--mount语法是更详细的比-v或--volume，但按键的顺序并不显著，并且标志的价值更容易理解。

该type安装件，其可以是bind，volume，或 tmpfs。本主题讨论卷，因此类型始终为 volume。

该source的安装。对于命名卷，这是卷的名称。对于匿名卷，将省略此字段。可以指定为source 或src。

destination作为其值，其中的文件或目录被安装在容器的路径。可以指定为destination，dst或target。

该readonly选项（如果存在）会使绑定安装以只读方式安装到容器中。

volume-opt可以多次指定的选项采用键值对，该键值对由选项名称及其值组成。

从外部CSV解析器转义值

如果您的卷驱动程序接受逗号分隔的列表作为选项，则必须从外部CSV解析器中转义该值。要对a进行转义volume-opt，请用双引号（"）包围，并用单引号（'）包围整个mount参数。

例如，local驱动程序接受挂载选项作为o参数中逗号分隔的列表。此示例显示了转义列表的正确方法。

$ docker service create \

--mount 'type=volume,src=<VOLUME-NAME>,dst=<CONTAINER-PATH>,volume-driver=local,volume-opt=type=nfs,volume-opt=device=<nfs-server>:<nfs-path>,"volume-opt=o=addr=<nfs-address>,vers=4,soft,timeo=180,bg,tcp,rw"'

--name myservice \

<IMAGE>

下面的示例在可能的情况下同时显示--mount和-v语法，并 --mount首先展示。

##### -v和--mount行为之间的差异

与绑定安装相反，所有卷选项都可用于 --mount和-v标志。

将卷与服务一起使用时，仅--mount支持。

### ### 创建和管理卷

与绑定安装不同，您可以在任何容器范围之外创建和管理卷。

1 创建一个卷：

$ docker volume create my-vol

2 清单数量：

$ docker volume ls

local my-vol

3 检查volume：

$ docker volume inspect my-vol

[

{

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/my-vol/\_data",

"Name": "my-vol",

"Options": {},

"Scope": "local"

}

]

4 删除卷：

$ docker volume rm my-vol

### ### 启动具有一定volume的容器

如果您使用尚不存在的卷启动容器，则Docker将为您创建该卷。以下示例将卷myvol2装入 /app/容器中。

下面的-v和--mount示例产生相同的结果。您不能同时运行它们，除非在运行第一个devtest容器和myvol2卷之后将其删除。

--mount

$ docker run -d \

--name devtest \

--mount source=myvol2,target=/app \

nginx:latest

-v

$ docker run -d \

--name devtest \

-v myvol2:/app \

nginx:latest

使用docker inspect devtest验证创建卷并安装正确。查找Mounts部分：

"Mounts": [

{

"Type": "volume",

"Name": "myvol2",

"Source": "/var/lib/docker/volumes/myvol2/\_data",

"Destination": "/app",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

这表明安装是一个卷，它显示了正确的源和目标，并且该安装是可读写的。

停止容器并取出volume。volume清除是一个单独的步骤。

$ docker container stop devtest

$ docker container rm devtest

$ docker volume rm myvol2

### ### 卷启动服务

启动服务并定义卷时，每个服务容器都使用其自己的本地卷。如果您使用local 卷驱动程序，则没有一个容器可以共享此数据，但是某些卷驱动程序确实支持共享存储。适用于AWS的Docker和适用于Azure的Docker均使用Cloudstor插件支持持久存储。

以下示例nginx以四个副本启动服务，每个副本使用一个称为的本地卷myvol2。

$ docker service create -d \

--replicas=4 \

--name devtest-service \

--mount source=myvol2,target=/app \

nginx:latest

使用docker service ps devtest-service验证服务正在运行：

$ docker service ps devtest-service

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

4d7oz1j85wwn devtest-service.1 nginx:latest moby Running Running 14 seconds ago

删除该服务，这将停止其所有任务：

$ docker service rm devtest-service

删除服务不会删除该服务创建的任何卷。卷删除是一个单独的步骤。

服务的语法差异

该docker service create命令不支持-vor --volume标志。将卷装入服务的容器时，必须使用该--mount 标志。

### ### 使用容器填充volume

如果如上所述启动一个容器来创建新的卷，并且该容器在要挂载的目录（例如/app/上面）中具有文件或目录，则目录的内容将复制到该卷中。然后，该容器将安装并使用该卷，并且使用该卷的其他容器也可以访问预填充的内容。

为了说明这一点，此示例启动一个nginx容器，并nginx-vol使用容器/usr/share/nginx/html目录的内容填充新卷，Nginx 在该 目录中存储其默认HTML内容。

在--mount和-v实施例具有相同的最终结果。

--mount

$ docker run -d \

--name=nginxtest \

--mount source=nginx-vol,destination=/usr/share/nginx/html \

nginx:latest

--v

$ docker run -d \

--name=nginxtest \

-v nginx-vol:/usr/share/nginx/html \

nginx:latest

运行这些示例中的任何一个之后，请运行以下命令以清理容器和卷。volume清除是一个单独的步骤。

$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol

### ### 使用只读卷

对于某些开发应用程序，容器需要写入绑定安装，以便将更改传播回Docker主机。在其他时候，容器仅需要对数据的读取访问权限。请记住，多个容器可以挂载相同的卷，并且可以同时对其中一些容器进行读写安装，而对其他容器则同时进行只读安装。

此示例修改了上面的示例，但通过ro在容器中的安装点之后添加到（默认为空）选项列表中，将目录作为只读卷安装。如果存在多个选项，请用逗号分隔。

在--mount和-v实例有同样的结果。

--mount

$ docker run -d \

--name=nginxtest \

--mount source=nginx-vol,destination=/usr/share/nginx/html,readonly \

nginx:latest

-v

$ docker run -d \

--name=nginxtest \

-v nginx-vol:/usr/share/nginx/html:ro \

nginx:latest

使用docker inspect nginxtest验证只读mount正确创建。查找Mounts部分：

"Mounts": [

{

"Type": "volume",

"Name": "nginx-vol",

"Source": "/var/lib/docker/volumes/nginx-vol/\_data",

"Destination": "/usr/share/nginx/html",

"Driver": "local",

"Mode": "",

"RW": false,

"Propagation": ""

}

],

停止并取出容器，然后取出volume。卷删除是一个单独的步骤。

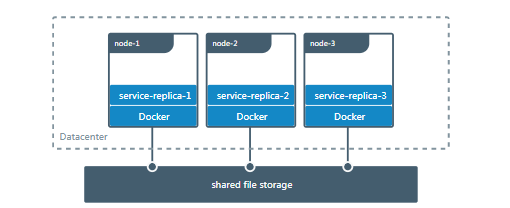
$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol

### ### 在机器之间共享数据

在构建容错应用程序时，您可能需要配置同一服务的多个副本才能访问相同的文件。

开发应用程序时，有几种方法可以实现此目的。一种是向您的应用程序添加逻辑，以将文件存储在像Amazon S3这样的云对象存储系统上。另一个是使用支持将文件写入外部存储系统（例如NFS或Amazon S3）的驱动程序来创建卷。

卷驱动程序使您可以从应用程序逻辑中抽象底层存储系统。例如，如果您的服务使用带有NFS驱动程序的卷，则可以更新服务以使用其他驱动程序（例如，将数据存储在云中），而无需更改应用程序逻辑。

### ### 使用volume驱动器

当使用创建卷时docker volume create，或者启动使用尚未创建的卷的容器时，可以指定卷驱动器。以下示例vieux/sshfs首先在创建独立卷时使用卷驱动程序，然后在启动创建新卷的容器时使用卷驱动程序。

#### 初始设置

本示例假定您有两个节点，其中第一个是Docker主机，并且可以使用SSH连接到第二个。

在Docker主机上，安装vieux/sshfs插件：

$ docker plugin install --grant-all-permissions vieux/sshfs

#### 使用卷驱动程序创建卷

此示例指定了SSH密码，但是如果两个主机都配置了共享密钥，则可以省略该密码。每个卷驱动程序可能具有零个或多个可配置选项，每个选项都使用一个-o标志指定。

$ docker volume create --driver vieux/sshfs \

-o sshcmd=test@node2:/home/test \

-o password=testpassword \

sshvolume

#### 启动一个使用卷驱动程序创建卷的容器

此示例指定了SSH密码，但是如果两个主机都配置了共享密钥，则可以省略该密码。每个卷驱动程序可能具有零个或多个可配置选项。如果卷驱动程序要求您传递选项，则必须使用该--mount标志而不是来安装卷-v。

$ docker run -d \

--name sshfs-container \

--volume-driver vieux/sshfs \

--mount src=sshvolume,target=/app,volume-opt=sshcmd=test@node2:/home/test,volume-opt=password=testpassword \

nginx:latest

#### 创建一个创建NFS卷的服务

本示例说明了创建服务时如何创建NFS卷。本示例10.0.0.10用作NFS服务器和NFS服务器上/var/docker-nfs的导出目录。请注意，指定的volume驱动程序为local。

##### NFSV3

$ docker service create -d \

--name nfs-service \

--mount 'type=volume,source=nfsvolume,target=/app,volume-driver=local,volume-opt=type=nfs,volume-opt=device=:/var/docker-nfs,volume-opt=o=addr=10.0.0.10' \

nginx:latest

##### NFSV4

docker service create -d \

--name nfs-service \

--mount 'type=volume,source=nfsvolume,target=/app,volume-driver=local,volume-opt=type=nfs,volume-opt=device=:/,"volume-opt=o=10.0.0.10,rw,nfsvers=4,async"' \

nginx:latest

### ### 备份，还原或迁移数据卷

卷对于备份，还原和迁移很有用。使用该 --volumes-from标志创建一个安装该卷的新容器。

#### 备份容器

例如，创建一个名为的新容器dbstore：

$ docker run -v /dbdata --name dbstore ubuntu /bin/bash

然后在下一个命令中，我们：

启动一个新容器并从该dbstore容器装入卷

将本地主机目录挂载为 /backup

将命令将dbdata卷的内容放到目录中的backup.tar文件中/backup。

$ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

当命令完成并且容器停止时，我们将获得dbdata卷的备份。

#### 从备份还原容器

使用刚刚创建的备份，您可以将其还原到同一容器或在其他位置创建的另一个容器。

例如，创建一个名为的新容器dbstore2：

$ docker run -v /dbdata --name dbstore2 ubuntu /bin/bash

然后将备份文件解压缩到新容器的数据卷中：

$ docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

您可以使用首选工具使用上述技术来自动执行备份，迁移和还原测试。

### ### 删除卷

删除容器后，Docker数据卷仍然存在。有两种类型的卷需要考虑：

命名卷具有来自容器外部的特定来源，例如awesome:/bar。

匿名卷没有特定来源，因此在删除容器时，请指示Docker Engine守护程序将其删除。

#### 删除匿名卷

要自动删除匿名卷，请使用该--rm选项。例如，此命令创建一个匿名/foo卷。删除容器后，Docker Engine会删除该/foo卷，但不会删除该awesome卷。

$ docker run --rm -v /foo -v awesome:/bar busybox top

#### 删除所有卷

要删除所有未使用的卷并释放空间：

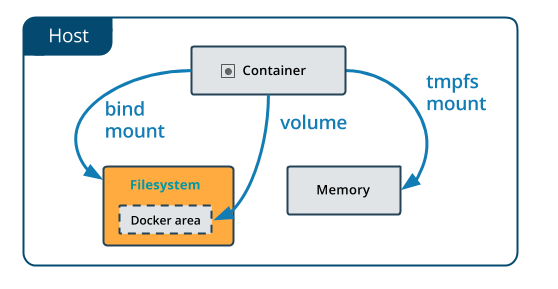
$ docker volume prune

## ## 绑定坐骑

### ### 使用绑定安装

自Docker诞生以来，绑定挂载就已经存在。与卷相比，绑定安装的功能有限。使用绑定安装时，主机上的文件或目录将安装到容器中。文件或目录由主机上的完整或相对路径引用。相比之下，当您使用卷时，将在主机上的Docker的存储目录中创建一个新目录，并且Docker管理该目录的内容。

该文件或目录不需要在Docker主机上已经存在。如果尚不存在，则按需创建。绑定挂载性能非常好，但是它们依赖于具有特定目录结构的主机文件系统。如果要开发新的Docker应用程序，请考虑使用命名卷。您不能使用Docker CLI命令直接管理绑定安装。



### ### 选择-v或--mount标志

最初，-vor --volume标志用于独立容器，而--mount标志用于集群服务。但是，从Docker 17.06开始，您还可以使用--mount独立容器。总的来说， --mount是更明确和冗长的。最大的区别是该-v 语法将所有选项合并在一个字段中，而--mount 语法将它们分开。这是每个标志的语法比较。

提示：新用户应使用--mount语法。有经验的用户可能更熟悉-v或--volume语法，但建议使用--mount，因为研究表明它更易于使用。

1 -v或--volume：由三个字段组成，以冒号（:）分隔。这些字段必须以正确的顺序排列，并且每个字段的含义不是立即显而易见的。

对于绑定安装，第一个字段是主机上文件或目录的路径。

第二个字段是文件或目录在容器中的安装路径。

第三个字段是可选的，并且是用逗号分隔的选项，诸如列表ro，consistent，delegated，cached，z，和Z。这些选项将在下面讨论。

2 --mount：包含多个键值对，以逗号分隔，每个键值对都由一个<key>=<value>元组组成。该--mount语法是更详细的比-v或--volume，但按键的顺序并不显著，并且标志的价值更容易理解。

该type安装件，其可以是bind，volume，或tmpfs。本主题讨论绑定安装，因此类型始终为bind。

该source的安装。对于绑定安装，这是Docker守护程序主机上文件或目录的路径。可以指定为source或 src。

的destination作为其值，其中的文件或目录被安装在容器的路径。可以指定为destination，dst或target。

该readonly选项（如果存在）会使绑定安装以只读方式安装到容器中。

该bind-propagation选项（如果存在）将更改 绑定传播。可以是一个rprivate， private，rshared，shared，rslave，slave。

consistency选项，如果存在，可以是一种consistent，delegated或cached。此设置仅适用于Mac的Docker桌面，在所有其他平台上将被忽略。

该--mount标志不支持z或没有Z用于修改selinux标签的选项。

下面的示例在可能的情况下同时显示--mount和-v语法，并 --mount首先展示。

#### -v和--mount行为之间的差异

由于-vand --volume标志已经很长时间成为Docker的一部分，因此它们的行为无法更改。这意味着和之间存在一种不同的行为。-v--mount

如果您使用-v或--volume绑定安装Docker主机上尚不存在的文件或目录，请-v为您创建端点。始终将其创建为目录。

如果您使用--mount绑定贴装尚不泊坞窗主机上存在的文件或目录，码头工人也不会自动为您创建它，但会产生一个错误。

### ### 使用绑定安装启动容器

考虑以下情况：您有一个目录source，并且在构建源代码时，工件会保存到另一个目录中source/target/。您希望这些工件可用于的容器/app/，并且希望每次在开发主机上构建源代码时，容器都可以访问新的构建。使用以下命令将target/ 目录绑定安装到您的容器中，位于/app/。从source目录中运行命令 。该$(pwd)子命令将扩展到Linux或macOS主机上的当前工作目录。

下面的--mount和-v示例产生相同的结果。您不能同时运行它们，除非devtest在运行第一个容器后删除容器。

--mount

$ docker run -d \

-it \

--name devtest \

--mount type=bind,source="$(pwd)"/target,target=/app \

nginx:latest

-v

$ docker run -d \

-it \

--name devtest \

-v "$(pwd)"/target:/app \

nginx:latest

使用docker inspect devtest验证绑定安装正确创建。查找Mounts部分：

"Mounts": [

{

"Type": "bind",

"Source": "/tmp/source/target",

"Destination": "/app",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

],

这表明该坐骑是一个bind坐骑，它显示了正确的源和目标，表明该坐骑是可读写的，并且传播设置为rprivate。

停止容器：

$ docker container stop devtest

$ docker container rm devtest

#### 挂载到容器上的非空目录中

如果将绑定安装到容器上的非空目录中，则该目录的现有内容将被绑定安装遮盖。这可能是有益的，例如，当您要测试应用程序的新版本而不构建新镜像时。但是，这也可能令人惊讶，并且此行为不同于docker volume的行为。

该示例被认为是极端的，但是用主机上的/usr/目录替换了容器目录的内容/tmp/。在大多数情况下，这将导致容器无法正常工作。

在--mount和-v实施例具有相同的最终结果。

--mount

$ docker run -d \

-it \

--name broken-container \

--mount type=bind,source=/tmp,target=/usr \

nginx:latest

docker: Error response from daemon: oci runtime error: container\_linux.go:262:

starting container process caused "exec: \"nginx\": executable file not found in $PATH".

-v

$ docker run -d \

-it \

--name broken-container \

-v /tmp:/usr \

nginx:latest

docker: Error response from daemon: oci runtime error: container\_linux.go:262:

starting container process caused "exec: \"nginx\": executable file not found in $PATH".

容器已创建但未启动。去掉它：

$ docker container rm broken-container

### ### 使用只读绑定安装

对于某些开发应用程序，容器需要写入绑定安装，因此更改将传播回Docker主机。在其他时间，容器仅需要读取访问权限。

此示例修改了上面的示例，但ro通过在容器中的安装点之后添加到（默认为空）选项列表中，将目录作为只读绑定安装进行安装。如果存在多个选项，请用逗号分隔。

在--mount和-v实例有同样的结果。

--mount

$ docker run -d \

-it \

--name devtest \

--mount type=bind,source="$(pwd)"/target,target=/app,readonly \

nginx:latest

-v

$ docker run -d \

-it \

--name devtest \

-v "$(pwd)"/target:/app:ro \

nginx:latest

使用docker inspect devtest验证绑定安装正确创建。查找Mounts部分：

"Mounts": [

{

"Type": "bind",

"Source": "/tmp/source/target",

"Destination": "/app",

"Mode": "ro",

"RW": false,

"Propagation": "rprivate"

}

],

停止容器：

$ docker container stop devtest

$ docker container rm devtest

### ### 配置绑定传播

rprivate对于绑定安装和卷，绑定传播默认为。它仅可用于绑定安装，并且只能在Linux主机上配置。绑定传播是一个高级主题，许多用户不需要配置它。

绑定传播是指是否可以将在给定绑定安装或命名卷中创建的安装传播到该安装的副本。考虑一个安装点/mnt，它也安装在上/tmp。传播设置控制是否/tmp/a可以在上安装底座/mnt/a。每个传播设置都有一个递归对点。在递归的情况下，请考虑将/tmp/a其也安装为/foo。传播设置控制是否/mnt/a和/或/tmp/a将存在。

| **传播设置** | **描述** |
| --- | --- |
| shared | 原始安装的子安装暴露于副本安装，副本安装的子安装也传播到原始安装。 |
| slave | 与共享安装类似，但仅在一个方向上。如果原始安装公开了子安装，则副本安装可以看到它。但是，如果副本安装公开了子安装，则原始安装看不到它。 |
| private | 坐骑是私人的。其中的子安装不暴露于副本安装，副本安装的子安装也不暴露于原始安装。 |
| rshared | 与共享相同，但传播也扩展到嵌套在任何原始或副本安装点中的安装点以及从这些安装点扩展。 |
| rslave | 与从属服务器相同，但是传播也扩展到嵌套在任何原始或副本安装点中的安装点以及从这些安装点扩展。 |
| rprivate | 默认值。与专用相同，这意味着原始或副本安装点内的任何安装点都不会向任一方向传播。 |

在可以在安装点上设置绑定传播之前，主机文件系统需要已经支持绑定传播。

有关绑定传播的更多信息，请参见 Linux内核文档中的共享子树。

以下示例将target/目录两次安装到容器中，第二次安装同时设置了ro选项和rslave绑定传播选项。

在--mount和-v实例有同样的结果。

--mount

$ docker run -d \

-it \

--name devtest \

--mount type=bind,source="$(pwd)"/target,target=/app \

--mount type=bind,source="$(pwd)"/target,target=/app2,readonly,bind-propagation=rslave \

nginx:latest

-v

$ docker run -d \

-it \

--name devtest \

-v "$(pwd)"/target:/app \

-v "$(pwd)"/target:/app2:ro,rslave \

nginx:latest

现在，如果您创建/app/foo/，它/app2/foo/也存在。

### ### 配置selinux标签

如果使用selinux，则可以添加z或Z选项来修改要装入容器的主机文件或目录的selinux标签。这会影响主机本身上的文件或目录，并可能导致超出Docker范围的后果。

该z选项指示绑定安装内容在多个容器之间共享。

该Z选项指示绑定安装内容是私有的且未共享。

这些选项请格外小心。绑定安装系统目录（例如/home或/usr带有该Z选项）会使主机无法操作，并且您可能需要手动重新标记主机文件。

重要说明：将绑定安装与服务一起使用时，selinux标签（:Z和:z）以及将:ro被忽略。有关详细信息，请参见 moby / moby＃32579。

本示例设置z选项以指定多个容器可以共享绑定安装的内容：

无法使用该--mount标志修改selinux标签。

$ docker run -d \

-it \

--name devtest \

-v "$(pwd)"/target:/app:z \

nginx:latest

### ### 配置macOS的安装一致性

用于Mac的Docker桌面用于osxfs将从macOS共享的目录和文件传播到Linux VM。这种传播使这些目录和文件可用于在Mac上运行Docker Desktop的Docker容器。

默认情况下，这些共享是完全一致的，这意味着每次在macOS主机上或通过容器中的挂载进行写操作时，所做的更改都会刷新到磁盘上，以便共享中的所有参与者都具有完全一致的视图。在某些情况下，完全一致性可能会严重影响性能。Docker 17.05及更高版本引入了一些选项，可以针对每个容器，每个容器来调整一致性设置。提供以下选项：

consistent或default：如上所述，具有完全一致性的默认设置。

delegated：容器运行时的挂载视图具有权威性。在主机上看到容器中所做的更新之前可能会有所延迟。

cached：macOS主机的挂载视图具有权威性。在容器中可以看到主机上所做的更新，这可能会有所延迟。

这些选项在除macOS之外的所有主机操作系统上均被完全忽略。

在--mount和-v实例有同样的结果。

--mount

$ docker run -d \

-it \

--name devtest \

--mount type=bind,source="$(pwd)"/target,destination=/app,consistency=cached \

nginx:latest

-v

$ docker run -d \

-it \

--name devtest \

-v "$(pwd)"/target:/app:cached \

nginx:latest

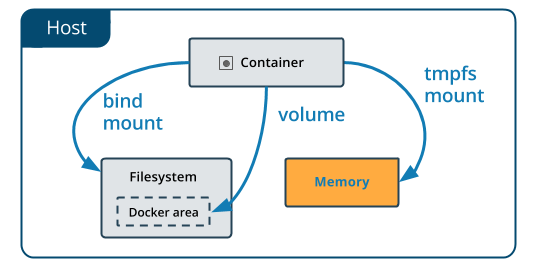
## ## tmpfs坐骑

### ### 使用tmpfs挂载

卷和绑定安装使您可以在主机和容器之间共享文件，以便即使容器停止后也可以保留数据。

如果您在Linux上运行Docker，则还有第三种选择：tmpfs挂载。当创建带有tmpfs安装架的容器时，该容器可以在该容器的可写层之外创建文件。

与卷和绑定挂载相反，tmpfs挂载是临时的，并且仅保留在主机内存中。当容器停止时，tmpfs挂载将被删除，并且在其中写入的文件将不会保留。



### ### tmpfs安装的局限性

与卷和绑定安装不同，您不能tmpfs在容器之间共享安装。

仅当您在Linux上运行Docker时，此功能才可用。

### ### 选择--tmpfs或--mount标志

最初，该--tmpfs标志用于独立容器，而该--mount标志用于集群服务。但是，从Docker 17.06开始，您还可以使用--mount独立容器。总的来说， --mount是更明确和冗长的。最大的区别是该 --tmpfs标志不支持任何可配置的选项。

--tmpfs：挂载tmpfs挂载而不允许您指定任何可配置的选项，并且只能与独立容器一起使用。

--mount：包含多个键值对，以逗号分隔，每个键值对都由一个<key>=<value>元组组成。该--mount语法比更详细--tmpfs：

该type安装件，其可以是bind，volume，或 tmpfs。本主题讨论tmpfs，因此类型始终为 tmpfs。

destination作为其值，其中的路径tmpfs安装被安装在容器中。可以指定为destination，dst或target。

在tmpfs-size和tmpfs-mode选项。请参阅 tmpfs选项。

下面的示例在可能的情况下同时显示--mount和--tmpfs语法，并--mount首先展示。

#### --tmpfs和--mount行为之间的差异

该--tmpfs标志不允许您指定任何可配置的选项。

该--tmpfs标志不能与群集服务一起使用。您必须使用--mount。

### ### 在容器中使用tmpfs挂载

要tmpfs在容器中使用安装座，请使用--tmpfs标志，或将 --mount标志与type=tmpfs和destination选项一起使用。没有 source了tmpfs坐骑。以下示例在Nginx容器中的上创建一个tmpfs安装 /app。第一个示例使用--mount标志，第二个示例使用--tmpfs标志。

--mount

$ docker run -d \

-it \

--name tmptest \

--mount type=tmpfs,destination=/app \

nginx:latest

--tmpfs

$ docker run -d \

-it \

--name tmptest \

--tmpfs /app \

nginx:latest

tmpfs通过运行docker container inspect tmptest并查找以下Mounts部分来验证安装是否为安装：

"Tmpfs": {

"/app": ""

},

卸下容器：

$ docker container stop tmptest

$ docker container rm tmptest

#### 指定tmpfs选项

tmpfs安装允许两个配置选项，都不是必需的。如果需要指定这些选项，则必须使用该--mount标志，因为该--tmpfs标志不支持它们。

| **选项** | **描述** |
| --- | --- |
| tmpfs-size | tmpfs安装的大小（以字节为单位）。默认情况下不受限制。 |
| tmpfs-mode | tmpfs的文件模式（八进制）。例如700或0770。默认为1777或世界可写。 |

以下示例将设置tmpfs-mode为1770，以使其在容器内不被世界范围内读取。

docker run -d \

-it \

--name tmptest \

--mount type=tmpfs,destination=/app,tmpfs-mode=1770 \

nginx:latest

## ## 解决卷数问题

### ### 解决卷数错误

本主题讨论使用Docker卷或绑定安装时可能发生的错误。

Error: Unable to remove filesystem

一些基于容器的实用程序（例如Google cAdvisor）将Docker系统目录（例如）安装/var/lib/docker/到容器中。例如，用于cadvisor指导您的文档的文档cadvisor如下：

$ sudo docker run \

--volume=/:/rootfs:ro \

--volume=/var/run:/var/run:rw \

--volume=/sys:/sys:ro \

--volume=/var/lib/docker/:/var/lib/docker:ro \

--publish=8080:8080 \

--detach=true \

--name=cadvisor \

google/cadvisor:latest

当您进行bind-mount时/var/lib/docker/，这有效地将所有其他正在运行的容器的所有资源作为mount的容器内的文件系统装入 /var/lib/docker/。当您尝试删除这些容器中的任何一个时，删除尝试可能会失败，并显示如下错误：

Error: Unable to remove filesystem for

74bef250361c7817bee19349c93139621b272bc8f654ae112dd4eb9652af9515:

remove /var/lib/docker/containers/74bef250361c7817bee19349c93139621b272bc8f654ae112dd4eb9652af9515/shm:

Device or resource busy

如果绑定安装的容器在文件系统中/var/lib/docker/ 使用statfs或fstatfs在文件系统上进行处理/var/lib/docker/ ，而不关闭它们，则会发生问题。

通常，我们建议不要/var/lib/docker以这种方式进行绑定安装。但是，cAdvisor需要此绑定安装才能实现核心功能。

如果不确定哪个进程导致错误中提到的路径繁忙并阻止其删除，则可以使用lsof命令查找其进程。例如，对于上面的错误：

$ sudo lsof /var/lib/docker/containers/74bef250361c7817bee19349c93139621b272bc8f654ae112dd4eb9652af9515/shm

要变通解决此问题，停止绑定安装的容器， /var/lib/docker然后再次尝试删除另一个容器。

## ## 在容器内存储数据

### ### 关于存储驱动程序

为了有效地使用存储驱动程序，了解Docker如何构建和存储镜像以及容器如何使用这些镜像非常重要。您可以使用此信息做出有关从应用程序持久保存数据并避免性能问题的最佳方法的明智选择。

存储驱动程序使您可以在容器的可写层中创建数据。删除容器后，这些文件将不会保留，并且读写速度都低于本机文件系统性能。

注意：已知有问题的操作包括写密集型数据库存储，尤其是在只写层中存在预先存在的数据时。本文档中提供了更多详细信息。

了解如何使用卷来保留数据并提高性能。

#### 镜像和图层

Docker镜像由一系列层组成。每层代表镜像的Dockerfile中的一条指令。除最后一层外的每一层都是只读的。考虑以下Dockerfile：

FROM ubuntu:18.04

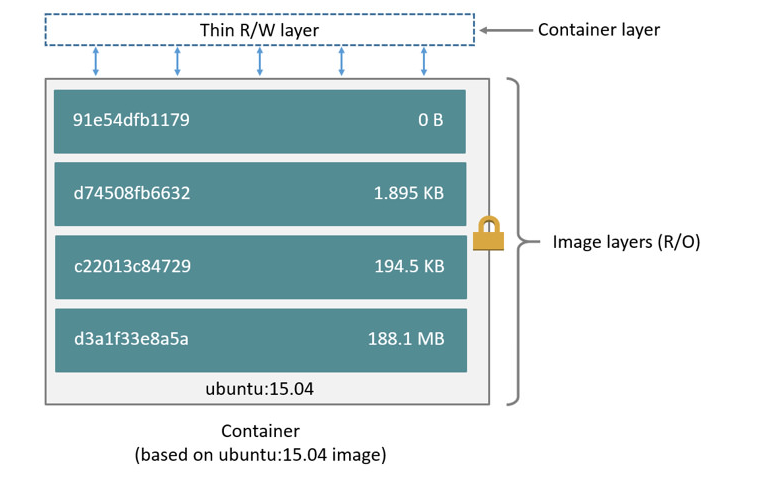
COPY . /app

RUN make /app

CMD python /app/app.py

该Dockerfile包含四个命令，每个命令创建一个层。该 FROM语句首先从ubuntu:18.04镜像创建图层。该COPY命令从Docker客户端的当前目录添加一些文件。该RUN命令使用命令构建您的应用程序make。最后，最后一层指定在容器中运行什么命令。

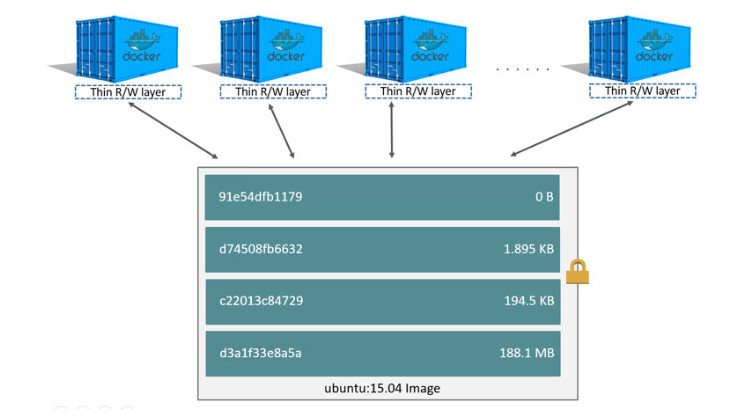
每一层只是与上一层不同的一组。这些层彼此堆叠。创建新容器时，可以在基础层之上添加新的可写层。该层通常称为“容器层”。对运行中的容器所做的所有更改（例如写入新文件，修改现有文件和删除文件）都将写入此薄可写容器层。下图显示了基于Ubuntu 18.04镜像的容器。

一个存储驱动程序处理的细节有关的方式互相交流这些层。提供了不同的存储驱动程序，它们在不同情况下各有利弊。

#### 容器和层数

容器和镜像之间的主要区别是可写顶层。在容器中添加新数据或修改现有数据的所有写操作都存储在此可写层中。删除容器后，可写层也会被删除。基础镜像保持不变。

因为每个容器都有其自己的可写容器层，并且所有更改都存储在该容器层中，所以多个容器可以共享对同一基础镜像的访问，但具有自己的数据状态。下图显示了多个共享同一Ubuntu 18.04镜像的容器。



注意：如果您需要多个镜像来共享对完全相同的数据的访问权限，请将此数据存储在Docker卷中并将其安装到您的容器中。

Docker使用存储驱动程序来管理镜像层和可写容器层的内容。每个存储驱动程序对实现的处理方式不同，但是所有驱动程序都使用可堆叠的镜像层和写时复制（CoW）策略。

#### 磁盘上的容器大小

要查看正在运行的容器的大致大小，可以使用以下docker ps -s 命令。有两个不同的列与大小有关。

size：用于每个容器的可写层的数据量（在磁盘上）。

virtual size：容器使用的只读镜像数据加上容器的可写层使用的数据量size。多个容器可以共享一些或所有只读镜像数据。从同一镜像开始的两个容器共享100％的只读数据，而具有不同镜像的两个容器（具有相同的层）共享这些公共层。因此，您不能只计算虚拟大小。这高估了总磁盘使用量，可能是一笔不小的数目。

磁盘上所有正在运行的容器使用的磁盘总空间是每个容器size及其virtual size值的某种组合。如果多个容器从相同的精确镜像开始，则这些容器在磁盘上的总大小为SUM（size容器）加上一个镜像大小（virtual size- size）。

这也不包括容器可以占用磁盘空间的以下其他方式：

如果使用json-file日志记录驱动程序，则用于日志文件的磁盘空间。如果您的容器生成大量的日志数据并且未配置日志轮转，那么这可能并非易事。

容器使用的卷和绑定挂载。

用于容器的配置文件的磁盘空间，通常较小。

内存写入磁盘（如果启用了交换）。

检查点（如果您正在使用实验性检查点/还原功能）。

#### 写入时复制（CoW）策略

写时复制是一种共享和复制文件的策略，可最大程度地提高效率。如果文件或目录位于镜像的较低层中，而另一层（包括可写层）需要对其进行读取访问，则它仅使用现有文件。另一层第一次需要修改文件时（在构建镜像或运行容器时），将文件复制到该层并进行修改。这样可以将I / O和每个后续层的大小最小化。这些优点将在下面更深入地说明。

##### 分享可以提升较小的镜像

当您用于docker pull从存储库中下拉镜像时，或者当您从本地尚不存在的镜像创建容器时，每个层都会被分别下拉，并存储在Docker的本地存储区域中，该区域通常/var/lib/docker/在Linux主机上。您可以在此示例中看到这些图层被拉出：

$ docker pull ubuntu:18.04

18.04: Pulling from library/ubuntu

f476d66f5408: Pull complete

8882c27f669e: Pull complete

d9af21273955: Pull complete

f5029279ec12: Pull complete

Digest: sha256:ab6cb8de3ad7bb33e2534677f865008535427390b117d7939193f8d1a6613e34

Status: Downloaded newer image for ubuntu:18.04

这些层中的每一层都存储在Docker主机本地存储区域内的自己的目录中。要检查文件系统上的层，请列出的内容/var/lib/docker/<storage-driver>。本示例使用overlay2 存储驱动程序：

$ ls /var/lib/docker/overlay2

16802227a96c24dcbeab5b37821e2b67a9f921749cd9a2e386d5a6d5bc6fc6d3

377d73dbb466e0bc7c9ee23166771b35ebdbe02ef17753d79fd3571d4ce659d7

3f02d96212b03e3383160d31d7c6aeca750d2d8a1879965b89fe8146594c453d

ec1ec45792908e90484f7e629330666e7eee599f08729c93890a7205a6ba35f5

l

目录名称与层ID不对应（从Docker 1.10开始就是如此）。

现在，假设您有两个不同的Dockerfile。您使用第一个创建名为的镜像acme/my-base-image:1.0。

FROM ubuntu:18.04

COPY . /app

第二acme/my-base-image:1.0层基于，但具有一些附加层：

FROM acme/my-base-image:1.0

CMD /app/hello.sh

第二个镜像包含第一个镜像的所有层，再加上带有CMD指令的新层，以及一个可读写容器层。Docker已经具有第一个镜像中的所有层，因此不需要再次将其拉出。这两个镜像共享它们共有的任何图层。

如果从两个Dockerfile构建镜像，则可以使用docker image ls和 docker history命令来验证共享层的密码ID是否相同。

1 创建一个新目录cow-test/并切换到该目录。

2 在中cow-test/，创建一个hello.sh具有以下内容的新文件：

#!/bin/sh

echo "Hello world"

保存文件，并使其可执行：

chmod +x hello.sh

3 将上面第一个Dockerfile的内容复制到一个名为的新文件中 Dockerfile.base。

4 将上面第二个Dockerfile的内容复制到一个名为的新文件中 Dockerfile。

5 在cow-test/目录中，构建第一个镜像。不要忘记.在命令中包含final 。设置了PATH，它告诉Docker在哪里寻找需要添加到镜像的任何文件。

$ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .

Sending build context to Docker daemon 812.4MB

Step 1/2 : FROM ubuntu:18.04

---> d131e0fa2585

Step 2/2 : COPY . /app

---> Using cache

---> bd09118bcef6

Successfully built bd09118bcef6

Successfully tagged acme/my-base-image:1.0

6 建立第二张镜像。

$ docker build -t acme/my-final-image:1.0 -f Dockerfile .

Sending build context to Docker daemon 4.096kB

Step 1/2 : FROM acme/my-base-image:1.0

---> bd09118bcef6

Step 2/2 : CMD /app/hello.sh

---> Running in a07b694759ba

---> dbf995fc07ff

Removing intermediate container a07b694759ba

Successfully built dbf995fc07ff

Successfully tagged acme/my-final-image:1.0

7 检查镜像的大小：

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

acme/my-final-image 1.0 dbf995fc07ff 58 seconds ago 103MB

acme/my-base-image 1.0 bd09118bcef6 3 minutes ago 103MB

8 检查组成每个镜像的图层：

$ docker history bd09118bcef6

IMAGE CREATED CREATED BY SIZE COMMENT

bd09118bcef6 4 minutes ago /bin/sh -c #(nop) COPY dir:35a7eb158c1504e... 100B

d131e0fa2585 3 months ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0B

<missing> 3 months ago /bin/sh -c mkdir -p /run/systemd && echo '... 7B

<missing> 3 months ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\... 2.78kB

<missing> 3 months ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0B

<missing> 3 months ago /bin/sh -c set -xe && echo '#!/bin/sh' >... 745B

<missing> 3 months ago /bin/sh -c #(nop) ADD file:eef57983bd66e3a... 103MB

$ docker history dbf995fc07ff

IMAGE CREATED CREATED BY SIZE COMMENT

dbf995fc07ff 3 minutes ago /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "/a... 0B

bd09118bcef6 5 minutes ago /bin/sh -c #(nop) COPY dir:35a7eb158c1504e... 100B

d131e0fa2585 3 months ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0B

<missing> 3 months ago /bin/sh -c mkdir -p /run/systemd && echo '... 7B

<missing> 3 months ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\... 2.78kB

<missing> 3 months ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0B

<missing> 3 months ago /bin/sh -c set -xe && echo '#!/bin/sh' >... 745B

<missing> 3 months ago /bin/sh -c #(nop) ADD file:eef57983bd66e3a... 103MB

请注意，除了第二个镜像的顶层以外，所有层都是相同的。所有其他层在两个镜像之间共享，并且仅在中存储一次/var/lib/docker/。实际上，新层根本不占用任何空间，因为它不更改任何文件，而仅运行命令。

注意：输出中的<missing>行docker history表示这些层是在另一个系统上构建的，并且在本地不可用。这可以忽略。

#### 复制使容器高效

启动容器时，会在其他层之上添加一个薄的可写容器层。容器对文件系统所做的任何更改都存储在这里。容器未更改的任何文件都不会复制到此可写层。这意味着可写层尽可能小。

修改容器中的现有文件后，存储驱动程序将执行写时复制操作。涉及的具体步骤取决于特定的存储驱动程序。对于aufs，overlay和overlay2驱动程序时，写入时复制操作遵循以下顺序粗糙：

在镜像层中搜索要更新的文件。该过程从最新层开始，一次向下一层到基础层。找到结果后，会将它们添加到缓存中以加快将来的操作。

copy\_up对找到的文件的第一个副本执行操作，以将文件复制到容器的可写层。

对该文件的副本进行了任何修改，并且容器看不到存在于较低层中的文件的只读副本。

Btrfs，ZFS和其他驱动程序以不同方式处理写时复制。您可以在稍后的详细描述中阅读有关这些驱动程序方法的更多信息。

写入大量数据的容器比不写入数据的容器消耗更多的空间。这是因为大多数写操作会占用容器的可写薄顶层中的新空间。

注意：对于繁重的应用程序，不应将数据存储在容器中。取而代之的是使用Docker卷，它们独立于正在运行的容器，并且旨在提高I / O效率。此外，卷可以在容器之间共享，并且不会增加容器可写层的大小。

一项copy\_up操作可能会导致明显的性能开销。该开销因所使用的存储驱动程序而异。大文件，许多层和深层目录树可以使影响更加明显。每个copy\_up操作仅在第一次修改给定文件时才发生，这可以缓解这种情况。

为了验证写时复制的工作方式，以下过程根据acme/my-final-image:1.0我们之前构建的镜像启动了5个容器，并检查它们占用了多少空间。

注意：此过程不适用于Mac的Docker桌面或Windows的Docker桌面。

1 在Docker主机上的终端上，运行以下docker run命令。最后的字符串是每个容器的ID。

$ docker run -dit --name my\_container\_1 acme/my-final-image:1.0 bash \

&& docker run -dit --name my\_container\_2 acme/my-final-image:1.0 bash \

&& docker run -dit --name my\_container\_3 acme/my-final-image:1.0 bash \

&& docker run -dit --name my\_container\_4 acme/my-final-image:1.0 bash \

&& docker run -dit --name my\_container\_5 acme/my-final-image:1.0 bash

c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409

dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513

1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c

38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544

1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765

2 运行docker ps命令以验证5个容器正在运行。

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1a174fc216cc acme/my-final-image:1.0 "bash" About a minute ago Up About a minute my\_container\_5

38fa94212a41 acme/my-final-image:1.0 "bash" About a minute ago Up About a minute my\_container\_4

1e7264576d78 acme/my-final-image:1.0 "bash" About a minute ago Up About a minute my\_container\_3

dcad7101795e acme/my-final-image:1.0 "bash" About a minute ago Up About a minute my\_container\_2

c36785c423ec acme/my-final-image:1.0 "bash" About a minute ago Up About a minute my\_container\_1

3 列出本地存储区的内容。

$ sudo ls /var/lib/docker/containers

1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765

1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c

38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544

c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409

dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513

4 现在查看它们的大小：

$ sudo du -sh /var/lib/docker/containers/\*

32K /var/lib/docker/containers/1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765

32K /var/lib/docker/containers/1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c

32K /var/lib/docker/containers/38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544

32K /var/lib/docker/containers/c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409

32K /var/lib/docker/containers/dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513

这些容器中的每个仅占用文件系统上32k的空间。

写时复制不仅可以节省空间，还可以缩短启动时间。当启动一个容器（或同一镜像中的多个容器）时，Docker只需要创建可写的容器薄层。

如果Docker每次启动新容器都必须制作基础镜像堆栈的完整副本，则容器启动时间和使用的磁盘空间将大大增加。这将类似于虚拟机的工作方式，每个虚拟机具有一个或多个虚拟磁盘。

### ### Docker存储驱动程序

#### Docker存储驱动程序

理想情况下，很少有数据被写入容器的可写层，并且您使用Docker卷来写入数据。但是，某些工作负载要求您能够写入容器的可写层。这是存储驱动程序进入的地方。

Docker使用可插拔架构支持几种不同的存储驱动程序。存储驱动程序控制如何在Docker主机上存储和管理镜像和容器。

阅读存储驱动程序概述之后，下一步就是为您的工作负载选择最佳的存储驱动程序。在做出此决定时，需要考虑三个高层因素：

如果内核中支持多个存储驱动程序，则在未明确配置任何存储驱动程序的情况下，Docker会优先列出要使用的存储驱动程序，前提是该存储驱动程序满足先决条件。

在大多数情况下，请使用具有最佳整体性能和稳定性的存储驱动程序。

Docker支持以下存储驱动程序：

overlay2 是当前所有受支持的Linux发行版的首选存储驱动程序，不需要任何额外的配置。

aufs是在内核3.13上不支持的Ubuntu 14.04上运行时，Docker 18.06和更早版本的首选存储驱动程序overlay2。

devicemapper支持，但direct-lvm对于生产环境是必需的，因为loopback-lvm零配置虽然性能很差。devicemapper是CentOS和RHEL的推荐存储驱动程序，因为它们的内核版本不支持overlay2。但是，当前版本的CentOS和RHEL现在支持overlay2，这是推荐的驱动程序。

如果btrfs和zfs驱动程序是后备文件系统（安装了Docker的主机的文件系统），则使用它们。这些文件系统允许使用高级选项，例如创建“快照”，但需要更多的维护和设置。这些中的每一个都依赖于正确配置的后备文件系统。

该vfs存储驱动程序的目的是为了进行测试，并在那里不能使用任何写入时复制文件系统的情况。此存储驱动程序的性能很差，通常不建议在生产中使用。

Docker的源代码定义了选择顺序。您可以在Docker Engine-Community 19.03的源代码上看到订单

如果运行其他版本的Docker，则可以使用文件查看器顶部的分支选择器来选择其他分支。

一些存储驱动程序要求您为备份文件系统使用特定格式。如果您有使用特定备份文件系统的外部要求，则可能会限制您的选择。请参阅支持的后备文件系统。

在确定了可以选择的存储驱动程序之后，您的选择取决于工作负载的特征和所需的稳定性。请参阅其他注意事项， 以帮助您做出最终决定。

注意：您的选择可能会受到Docker版本，操作系统和发行版的限制。例如，aufs仅在Ubuntu和Debian上受支持，并且可能需要安装额外的软件包，而btrfs仅在SLES受支持，SLES仅受Docker Enterprise支持。有关 更多信息，请参见按Linux发行版支持存储驱动程序。

#### 每个Linux发行版支持的存储驱动程序

在较高级别上，可以使用的存储驱动程序部分取决于所使用的Docker版本。

此外，Docker不建议您使用任何配置来禁用操作系统的安全功能，例如，selinux如果您在CentOS上使用overlay或overlay2驱动程序，则需要禁用 该配置。

##### Docker Engine-企业版和Docker企业版

对于Docker Engine-Enterprise和Docker Enterprise，支持存储驱动程序的权威资源是 产品兼容性列表。要从Docker获得商业支持，您必须使用受支持的配置。

##### Docker Engine-社区

对于Docker Engine-Community，仅测试了某些配置，并且您操作系统的内核可能不支持每个存储驱动程序。通常，以下配置适用于最新版本的Linux发行版：

| **Linux发行版** | **推荐的存储驱动程序** | **替代驱动** |
| --- | --- | --- |
| Docker Engine-Ubuntu上的社区 | overlay2或aufs（对于在内核3.13上运行的Ubuntu 14.04） | overlay¹，devicemapper² zfs，vfs |
| Docker Engine-Debian上的社区 | overlay2（Debian Stretch）aufs或devicemapper（旧版本） | overlay¹， vfs |
| Docker Engine-CentOS上的社区 | overlay2 | overlay¹，devicemapper² zfs，vfs |
| Docker Engine-Fedora上的社区 | overlay2 | overlay¹，devicemapper² zfs，vfs |

¹）overlay存储驱动程序已在Docker Engine-Enterprise 18.09中弃用，并将在以后的版本中删除。建议将overlay存储驱动程序的用户迁移到overlay2。

²）devicemapper存储驱动程序已在Docker Engine 18.09中弃用，并将在以后的版本中删除。建议将devicemapper存储驱动程序的用户迁移到overlay2。

如果可能，overlay2建议使用存储驱动程序。首次安装Docker时，overlay2默认情况下使用。以前，aufs默认情况下会在可用时使用它，但情况不再如此。如果要aufs在以后的新安装中使用，则需要对其进行显式配置，并且可能需要安装其他软件包，例如linux-image-extra。参见aufs。

在使用的现有安装中aufs，它仍然被使用。

如有疑问，最好的全面配置是使用具有支持overlay2存储驱动程序的内核的现代Linux发行版，并使用Docker卷处理繁重的工作负载，而不是依赖于将数据写入容器的可写层。

该vfs存储驱动程序通常不是最好的选择。使用vfs 存储驱动程序之前，请务必阅读 其性能以及存储特性和限制。

注意：对不推荐的存储驱动程序的期望：Docker Engine-Community不提供商业支持，您可以从技术上使用适用于您平台的任何存储驱动程序。例如btrfs，即使不建议在Docker Engine-Community的任何平台上使用它，也可以将其与Docker Engine-Community一起使用，但后果自负。

上表中的建议基于自动回归测试和已知适用于大量用户的配置。如果使用推荐的配置并发现可重现的问题，则很可能会很快解决该问题。如果根据该表不推荐您使用的驱动程序，则需要您自担风险。您可以并且仍然应该报告遇到的任何问题。但是，此类问题的优先级低于使用推荐配置时遇到的问题。

##### Mac的Docker桌面和Windows的Docker桌面

适用于Mac的Docker Desktop和适用于Windows的Docker Desktop用于开发而非生产。无法在这些平台上修改存储驱动程序。

#### 支持的后备文件系统

对于Docker，支持文件系统是所在的文件系统 /var/lib/docker/。一些存储驱动程序仅适用于特定的后备文件系统。

| **存储驱动** | **支持的后备文件系统** |
| --- | --- |
| overlay2， overlay | xfs 在ftype = 1的情况下， ext4 |
| aufs | xfs， ext4 |
| devicemapper | direct-lvm |
| btrfs | btrfs |
| zfs | zfs |
| vfs | 任何文件系统 |

#### 其他注意事项

##### 适合您的工作量

除其他事项外，每个存储驱动程序都有其自己的性能特征，这使其或多或少地适合于不同的工作负载。考虑以下概括：

overlay2，，aufs和overlay全部在文件级别而不是块级别运行。这样可以更有效地使用内存，但是在写繁重的工作负载中，容器的可写层可能会变得非常大。

块级存储驱动器，例如devicemapper，btrfs和zfs更好地为写繁重的工作负担（虽然不是还有泊坞窗卷）执行。

对于许多小型写入或具有多层或深文件系统的容器，其性能 overlay可能比更好overlay2，但会消耗更多的inode，这可能导致inode耗尽。

btrfs并zfs需要大量内存。

zfs 对于高密度工作负载（例如PaaS）是一个不错的选择。

每个存储驱动程序的文档中提供了有关性能，适用性和最佳实践的更多信息。

##### 共享存储系统和存储驱动程序

如果您的企业使用SAN，NAS，硬件RAID或其他共享存储系统，则它们可以提供高可用性，增强的性能，自动精简配置，重复数据删除和压缩。在许多情况下，Docker可以在这些存储系统上运行，但是Docker并未与其紧密集成。

每个Docker存储驱动程序均基于Linux文件系统或卷管理器。确保在共享存储系统之上遵循用于操作存储驱动程序（文件系统或卷管理器）的现有最佳实践。例如，如果在共享存储系统上使用ZFS存储驱动程序，请确保遵循在该特定共享存储系统上操作ZFS文件系统的最佳实践。

##### 稳定性

对于某些用户而言，稳定性比性能更重要。尽管Docker认为这里提到的所有存储驱动程序都是稳定的，但其中一些是较新的并且仍在积极开发中。在一般情况下，overlay2，aufs，overlay，和devicemapper与稳定性最高的选择。

##### 用您自己的工作负载进行测试

您可以在不同的存储驱动程序上运行自己的工作负载时测试Docker的性能。确保使用等效的硬件和工作负载来匹配生产条件，以便您可以看到哪个存储驱动程序提供最佳的整体性能。

#### 检查您当前的存储驱动程序

每个单独的存储驱动程序的详细文档详细说明了使用给定存储驱动程序的所有设置步骤。

要查看Docker当前正在使用什么存储驱动程序，请使用docker info并查找以下Storage Driver行：

$ docker info

Containers: 0

Images: 0

Storage Driver: overlay2

Backing Filesystem: xfs

<output truncated>

要更改存储驱动程序，请参阅新存储驱动程序的特定说明。一些驱动程序需要其他配置，包括对Docker主机上的物理或逻辑磁盘的配置。

重要提示：更改存储驱动程序时，所有现有的镜像和容器都将无法访问。这是因为新存储驱动程序无法使用其图层。如果还原更改，则可以再次访问旧的镜像和容器，但是使用新驱动程序提取或创建的任何镜像和容器将无法访问。

### ### 使用AUFS存储驱动程序

AUFS是联合文件系统。该aufs存储驱动程序以前是默认存储驱动程序，用于在Docker上管理Ubuntu和Ubuntu之前的Debian版本上的镜像和层。如果您的Linux内核是版本4.0或更高版本，并且您使用的是Docker Engine-Community，请考虑使用较新的 overlay2，它比aufs存储驱动程序具有潜在的性能优势。

注意：在某些发行版和Docker版本中不支持AUFS。有关支持的平台的更多信息，请参阅先决条件 >， 另请参阅 存储驱动程序的首选项顺序。

#### 先决条件

对于Docker Engine-Community，在Ubuntu和Stretch之前的Debian版本中支持AUFS。

对于Docker EE，Ubuntu支持AUFS。

如果使用Ubuntu，则需要 安装额外的软件包 才能将AUFS模块添加到内核。如果不安装这些软件包，则需要devicemapper在Ubuntu 14.04（不建议使用）或overlay2Ubuntu 16.04及更高版本上使用（也受支持）。

AUFS不能使用以下支持的文件系统：aufs，btrfs，或 ecryptfs。这意味着包含的文件系统 /var/lib/docker/aufs不能是这些文件系统类型之一。

#### 使用aufs存储驱动程序配置Docker

如果在启动Docker时将AUFS驱动程序加载到内核中，并且未配置其他存储驱动程序，则Docker默认使用它。

1 使用以下命令来验证您的内核是否支持AUFS。

$ grep aufs /proc/filesystems

nodev aufs

2 检查Docker使用哪个存储驱动程序。

$ docker info

<truncated output>

Storage Driver: aufs

Root Dir: /var/lib/docker/aufs

Backing Filesystem: extfs

Dirs: 0

Dirperm1 Supported: true

<truncated output>

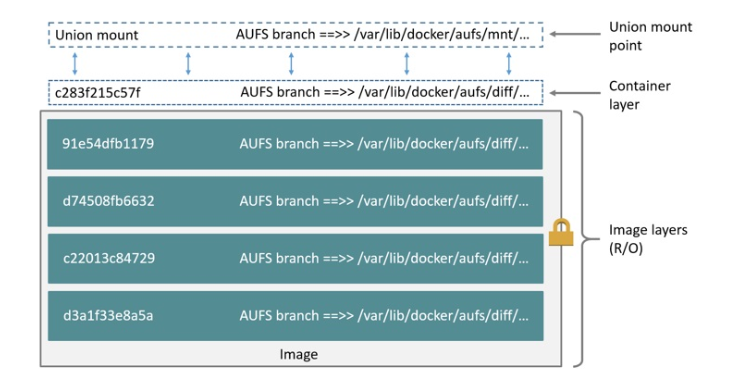
3 如果您使用其他存储驱动程序，则内核中不包含AUFS（在这种情况下，将使用其他默认驱动程序），或者已将Docker明确配置为使用其他驱动程序。检查 /etc/docker/daemon.json或的输出ps auxw | grep dockerd以查看Docker是否已使用该--storage-driver标志启动。

#### aufs存储驱动程序如何工作

AUFS是一个联合文件系统，这意味着它将在一个Linux主机上分层多个目录，并将它们显示为一个目录。这些目录在AUFS术语中称为分支，在Docker术语中称为层。

统一过程称为联合安装。

下图显示了基于该ubuntu:latest镜像的Docker容器。



每个镜像层和容器层在Docker主机上均表示为中的子目录/var/lib/docker/。联合安装提供了所有层的统一视图。目录名称并不直接对应于图层本身的ID。

AUFS使用写时复制（CoW）策略来最大化存储效率并最小化开销。

##### 示例：镜像和容器磁盘构造

以下docker pull命令显示了一个Docker主机下载一个包含五层的Docker镜像。

$ docker pull ubuntu

Using default tag: latest

latest: Pulling from library/ubuntu

b6f892c0043b: Pull complete

55010f332b04: Pull complete

2955fb827c94: Pull complete

3deef3fcbd30: Pull complete

cf9722e506aa: Pull complete

Digest: sha256:382452f82a8bbd34443b2c727650af46aced0f94a44463c62a9848133ecb1aa8

Status: Downloaded newer image for ubuntu:latest

###### 镜像层

警告：请勿直接操作其中的任何文件或目录 /var/lib/docker/。这些文件和目录由Docker管理。

有关镜像和容器层的所有信息都存储在的子目录中/var/lib/docker/aufs/。

diff/：每层的内容，每层的内容存储在单独的子目录中

layers/：有关如何堆叠镜像层的元数据。该目录为Docker主机上的每个镜像或容器层包含一个文件。每个文件都包含堆栈中其下所有层（其父级）的ID。

mnt/：安装点，每个镜像或容器层一个，用于组装和安装容器的统一文件系统。对于只读镜像，这些目录始终为空。

###### 容器层

如果容器正在运行，则/var/lib/docker/aufs/更改内容的方式如下：

diff/：可写容器层中引入的差异，例如新文件或修改过的文件。

layers/：有关可写容器层的父层的元数据。

mnt/：每个正在运行的容器的统一文件系统的安装点，与从容器内部出现的安装点完全相同。

#### 容器如何读写 aufs

##### 读取文件

考虑三种情况，其中容器打开文件以通过aufs进行读取访问。

该文件在容器层中不存在：如果容器打开文件进行读取访问，并且该文件在容器层中尚不存在，则存储驱动程序将在镜像层中搜索文件，从镜像层正下方的层开始容器层。从找到它的层读取它。

该文件仅存在于容器层中：如果容器打开文件以进行读取访问，并且该文件存在于容器层中，则从那里进行读取。

该文件存在于容器层和镜像层中：如果容器打开文件以进行读取访问，并且该文件存在于容器层和一个或多个镜像层中，则从容器层读取文件。容器层中的文件会使镜像层中具有相同名称的文件模糊。

##### 修改文件或目录

考虑在某些情况下修改了容器中的文件。

1 第一次写入文件：容器第一次写入现有文件时，该文件在容器（upperdir）中不存在。的aufs驾驶员进行copy\_up操作将文件从那里它存在到可写的容器层中的镜像层复制。然后，容器将更改写入容器层中文件的新副本。

但是，AUFS在文件级别而不是块级别工作。这意味着所有copy\_up操作都将复制整个文件，即使该文件非常大且只有一小部分正在被修改。这会对容器写入性能产生明显影响。在具有多层的镜像中搜索文件时，AUFS可能会出现明显的延迟。但是，值得注意的是，copy\_up操作仅在第一次写入给定文件时发生。随后对同一文件的写入将对已经复制到容器的文件副本进行操作。

2 删除文件和目录：

当一个文件是一个容器内删除，一个白斑在容器层中创建的文件。不会删除镜像层中文件的版本（因为镜像层是只读的）。但是，白化文件会阻止容器使用它。

当在容器内删除目录时，将在容器层中创建一个不透明文件。这与停电文件的工作方式相同，并且即使该目录仍然存在于镜像层中，也可以有效地防止该目录被访问。

3 重命名目录：rename(2)AUFS不完全支持调用目录。EXDEV即使源路径和目标路径都在同一AUFS层上，它也会返回（“不允许跨设备链接”），除非目录中没有子目录。您的应用程序需要设计为处理EXDEV并退回到“复制和取消链接”策略。

#### AUFS和Docker性能

总结一些已经提到的与性能相关的方面：

1 AUFS存储驱动程序的性能不如该overlay2驱动程序，但是对于容器密度很重要的PaaS和其他类似用例来说，它是一个不错的选择。这是因为AUFS可以在多个运行中的容器之间有效地共享镜像，从而缩短了容器的启动时间，并减少了磁盘空间的使用。

2 AUFS如何在镜像层和容器之间共享文件的基本机制非常有效地使用了页面缓存。

3 AUFS存储驱动程序可以将明显的延迟引入容器写入性能。这是因为容器第一次写入任何文件时，都需要找到该文件并将其复制到容器的顶层可写层中。当这些文件存在于许多镜像层下面并且文件本身很大时，这些延迟会增加并且变得更加复杂。

##### 绩效最佳实践

以下通用性能最佳做法也适用于AUFS。

1 固态设备（SSD）的读取和写入速度比旋转磁盘快。

2 将卷用于繁重的写工作负载：卷可为繁重的写工作负载提供最佳和最可预测的性能。这是因为它们绕过了存储驱动程序，并且不会产生任何精简配置和写入时复制所带来的潜在开销。卷还有其他好处，例如，允许您在容器之间共享数据，并且即使没有正在运行的容器正在使用它们也可以持久存储。

### ### 使用BTRFS存储驱动程序

Btrfs是下一代写时复制文件系统，支持许多高级存储技术，使其非常适合Docker。Btrfs包含在主线Linux内核中。

Docker的btrfs存储驱动程序利用许多Btrfs功能进行镜像和容器管理。这些功能包括块级操作，自动精简配置，写时复制快照和易于管理。您可以轻松地将多个物理块设备组合到一个Btrfs文件系统中。

本文将Docker的Btrfs存储驱动程序称为btrfsBtrfs，将整个Btrfs文件系统称为Btrfs。

注意：btrfs仅Docker Engine-Community在Ubuntu或Debian上支持存储驱动程序，而SLES上的Docker EE / CS Engine支持。

#### 先决条件

btrfs 如果满足以下先决条件，则支持：

Docker Engine-Community：btrfs仅在Ubuntu或Debian上建议使用Docker Engine-Community 。

Docker EE：对于Docker EE和CS-Engine，btrfs仅在SLES上受支持。有关 商业支持的Docker的所有受支持配置，请参阅 产品兼容性列表。

更改存储驱动程序会使本地系统上无法访问您已经创建的任何容器。使用docker save保存的容器，并推动现有镜像多克尔集线器或私人仓库，让你不必再后来创建它们。

btrfs需要专用的块存储设备，例如物理磁盘。此块设备必须格式化为Btrfs并安装到中/var/lib/docker/。下面的配置说明将指导您完成此过程。默认情况下，SLES /文件系统是使用BTRFS格式化的，因此对于SLES，您不需要使用单独的块设备，但是出于性能考虑，您可以选择这样做。

btrfs支持必须存在于您的内核中。要检查这一点，请运行以下命令：

$ sudo cat /proc/filesystems | grep btrfs

btrfs

要在操作系统级别管理BTRFS文件系统，您需要 btrfs命令。如果没有此命令，请安装btrfsprogs 软件包（SLES）或btrfs-tools软件包（Ubuntu）。

#### 配置Docker以使用btrfs存储驱动程序

在SLES和Ubuntu上，此过程本质上是相同的。

1 停止Docker。

2 将的内容复制/var/lib/docker/到备份位置，然后清空的内容/var/lib/docker/：

$ sudo cp -au /var/lib/docker /var/lib/docker.bk

$ sudo rm -rf /var/lib/docker/\*

3 将一个或多个专用块设备格式化为Btrfs文件系统。本示例假定您正在使用两个称为/dev/xvdf和的 块设备/dev/xvdg。仔细检查块设备名称，因为这是破坏性操作。

$ sudo mkfs.btrfs -f /dev/xvdf /dev/xvdg

Btrfs还有更多选项，包括条带化和RAID。请参阅 Btrfs文档。

4 在/var/lib/docker/安装点上安装新的Btrfs文件系统。您可以指定用于创建Btrfs文件系统的任何块设备。

$ sudo mount -t btrfs /dev/xvdf /var/lib/docker

不要忘记通过将条目添加到重新启动来使更改永久生效/etc/fstab。

5 将的内容复制/var/lib/docker.bk到/var/lib/docker/。

$ sudo cp -au /var/lib/docker.bk/\* /var/lib/docker/

6 配置Docker以使用btrfs存储驱动程序。即使/var/lib/docker/现在正在使用Btrfs文件系统，这也是必需的。编辑或创建文件/etc/docker/daemon.json。如果是新文件，请添加以下内容。如果它是现有文件，则仅添加键和值，如果不是结束大括号（}）之前的最后一行，请小心以逗号结束。

{

"storage-driver": "btrfs"

}

查看每个存储驱动程序的所有存储选项：

* [稳定](https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)
* [边缘](https://docs.docker.com/edge/engine/reference/commandline/dockerd/#storage-driver-options)

7 启动Docker。运行后，请确认它btrfs已被用作存储驱动程序。

$ docker info

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 17.03.1-ce

Storage Driver: btrfs

Build Version: Btrfs v4.4

Library Version: 101

<output truncated>

8 准备好后，删除/var/lib/docker.bk目录。

#### 管理Btrfs卷

Btrfs的好处之一是易于管理Btrfs文件系统，而无需卸载文件系统或重新启动Docker。

当空间变小时，Btrfs会自动将卷扩展为大约1 GB的块。

要将块设备添加到Btrfs卷，请使用btrfs device add和 btrfs filesystem balance命令。

$ sudo btrfs device add /dev/svdh /var/lib/docker

$ sudo btrfs filesystem balance /var/lib/docker

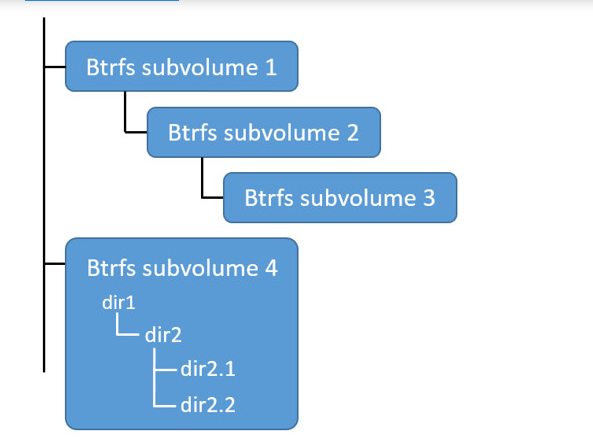
注意：虽然您可以在Docker运行时执行这些操作，但性能会受到影响。最好计划一个中断窗口以平衡Btrfs文件系统。

#### btrfs存储驱动程序如何工作

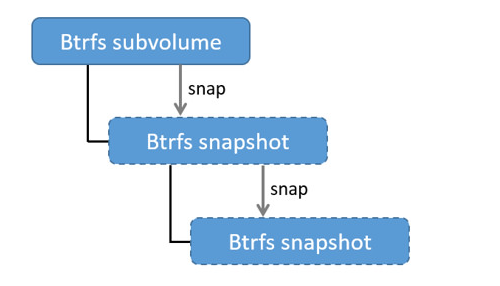
该btrfs存储驱动器的工作方式不同devicemapper或在您的整个其他存储驱动程序/var/lib/docker/目录存储在增加了Btrfs卷上。

#### 磁盘上的镜像和容器层

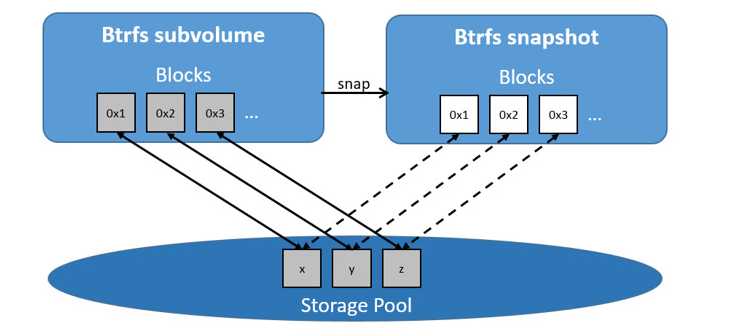
有关镜像层和可写容器层的信息存储在中 /var/lib/docker/btrfs/subvolumes/。此子目录在每个镜像或容器层包含一个目录，该统一目录由一个层及其所有父层构建而成。子卷本来就是写时复制的，并从基础存储池按需分配了空间。它们也可以嵌套和快照。下图显示了4个子卷。“子卷2”和“子卷3”是嵌套的，而“子卷4”则显示其自己的内部目录树。



仅镜像的基础层存储为真实的子volume。所有其他层都存储为快照，其中仅包含该层中引入的差异。您可以如下图所示创建快照的快照。

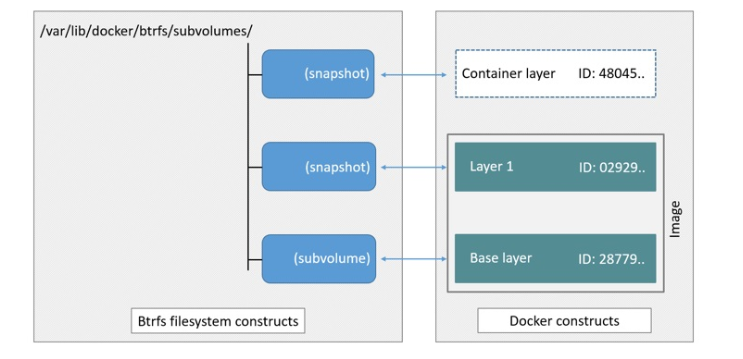


在磁盘上，快照的外观和感觉就像子卷一样，但实际上它们更小且更节省空间。写入时复制用于最大化存储效率和最小化层大小，并且在块级别管理容器可写层中的写入。下图显示了一个子卷及其快照共享数据。



为了获得最大效率，当一个容器需要更多空间时，将以大约1 GB的块为单位进行分配 。

Docker的btrfs存储驱动程序将每个镜像层和容器存储在其自己的Btrfs子卷或快照中。镜像的基础层存储为子卷，而子镜像层和容器存储为快照。如下图所示。



在运行btrfs驱动程序的Docker主机上创建镜像和容器的高级过程如下：

镜像的基本层被存储在一增加了Btrfs 子volume下 /var/lib/docker/btrfs/subvolumes。

随后的镜像层被存储为增加了Btrfs 快照父层的子volume或快照的，但与此层中引入的变化。这些差异存储在块级别。

容器的可写层是最终镜像层的Btrfs快照，其中差异由运行中的容器引入。这些差异存储在块级别。

#### 容器如何读写 btrfs

##### 读取文件

容器是镜像的节省空间的快照。快照中的元数据指向存储池中的实际数据块。这与子卷相同。因此，对快照执行的读取与对子卷执行的读取基本相同。

##### 写文件

编写新文件：将新文件写入容器会调用按需分配操作，以将新数据块分配给容器的快照。然后将文件写入此新空间。按需分配操作是使用Btrfs进行的所有写入的本机操作，与将新数据写入子卷相同。结果，将新文件写入容器的快照以本机Btrfs速度运行。

修改现有文件：更新容器中的现有文件是写时复制操作（写时重定向是Btrfs术语）。从文件当前所在的层读取原始数据，并且仅将修改后的块写入容器的可写层。接下来，Btrfs驱动程序更新快照中的文件系统元数据以指向此新数据。此行为导致很少的开销。

删除文件或目录：如果容器删除了下层中存在的文件或目录，则Btrfs会掩盖下层中文件或目录的存在。如果容器创建了一个文件然后将其删除，则此操作在Btrfs文件系统本身中执行，并回收空间。

使用Btrfs，写入和更新许多小文件可能会导致性能降低。

#### Btrfs和Docker性能

在btrfs 存储驱动程序下，有几个因素会影响Docker的性能。

注意：通过将Docker卷用于繁重的写工作负载，而不是依赖于将数据存储在容器的可写层中，可以缓解许多因素。然而，在增加了Btrfs的情况下，多克尔卷仍从这些画中后卫吃亏，除非/var/lib/docker/volumes/是不通过增加了Btrfs支持。

1 页面缓存。Btrfs不支持页面缓存共享。这意味着每个访问同一文件的进程都会将该文件复制到Docker主机的内存中。结果，btrfs驱动程序可能不是诸如PaaS之类的高密度用例的最佳选择。

2 小写。容器执行大量小写操作（此使用模式与您在短时间内启动和停止许多容器时发生的情况相匹配）也会导致Btrfs块的使用不佳。这可能会过早填充Btrfs文件系统，并导致Docker主机空间不足。使用btrfs filesys show您必须密切监控增加了Btrfs设备上的可用空间量。

3 顺序写入。Btrfs在写入磁盘时使用日记技术。这可能会影响顺序写入的性能，从而使性能降低多达50％。

4 碎片化。碎片是Btrfs等写时复制文件系统的自然副产品。许多小的随机写入可能会使此问题复杂化。使用SSD时，碎片可能表现为CPU峰值，而使用旋转磁盘时，则表现为磁头抖动。这些问题中的任何一个都可能损害性能。

如果您的Linux内核版本是3.9或更高版本，则autodefrag 在安装Btrfs卷时可以启用该功能。在将该功能部署到生产中之前，请先对您的工作负载进行测试，因为某些测试显示了对性能的负面影响。

5 SSD性能：Btrfs包括针对SSD介质的本机优化。要启用这些功能，请使用-o ssdmount选项安装Btrfs文件系统。这些优化通过避免优化（例如不适用于固态介质的寻道优化）来增强SSD的写入性能。

6 经常平衡Btrfs文件系统：cron在非高峰时段，使用操作系统实用程序（例如 作业）定期平衡Btrfs文件系统。这样可以回收未分配的块，并有助于防止文件系统不必要地填满。除非您向文件系统添加其他物理块设备，否则您无法重新平衡完全完整的Btrfs文件系统。请参阅 BTRFS Wiki。

7 使用快速存储：固态驱动器（SSD）提供比旋转磁盘更快的读写速度。

8 将卷用于繁重的写工作负载：卷可为繁重的写工作负载提供最佳和最可预测的性能。这是因为它们绕过了存储驱动程序，并且不会产生任何精简配置和写入时复制所带来的潜在开销。卷还有其他好处，例如，允许您在容器之间共享数据，并且即使没有正在运行的容器正在使用它们也可以持久存储。

### ### 使用设备映射器存储驱动程序

Device Mapper是基于内核的框架，是Linux上许多高级卷管理技术的基础。Docker的devicemapper存储驱动程序利用此框架的精简配置和快照功能进行镜像和容器管理。本文将Device Mapper存储驱动程序称为devicemapper，将内核框架称为Device Mapper。

对于受devicemapper支持的系统，Linux内核中包含支持。但是，将其与Docker结合使用需要特定的配置。

该devicemapper驱动程序使用专用于Docker的块设备，并在块级别而非文件级别运行。可以通过向Docker主机添加物理存储来扩展这些设备，它们的性能要比在操作系统（OS）级别上使用文件系统更好。

#### 先决条件

devicemapper存储驱动程序是许多操作系统发行版上受支持的Docker EE存储驱动程序。有关详细信息，请参见 产品兼容性列表。

devicemapper Docker引擎-在CentOS，Fedora，Ubuntu或Debian上运行的社区也受支持。

更改存储驱动程序会使本地系统上无法访问您已经创建的任何容器。使用docker save保存的容器，并推动现有镜像多克尔集线器或私人仓库，这样你就不会需要稍后重新创建。

#### 使用devicemapper存储驱动程序配置Docker

在执行这些步骤之前，您必须首先满足所有 先决条件。

##### 配置loop-lvm测试模式

此配置仅适用于测试。该loop-lvm模式利用一种“回送”机制，该机制可以读取和写入本地磁盘上的文件，就像它们是实际的物理磁盘或块设备一样。但是，添加回送机制以及与OS文件系统层的交互意味着IO操作可能很慢且占用大量资源。使用环回设备也会引入竞争条件。但是，设置loop-lvm模式可以在尝试启用direct-lvm模式所需的更复杂的设置之前，帮助您识别基本问题（例如缺少用户空间软件包，内核驱动程序等）。loop-lvm因此，模式仅应在配置之前用于执行基本测试 direct-lvm。

对于生产系统，请参阅为生产 配置直接lvm​​模式。

1 停止Docker。

$ sudo systemctl stop docker

2 编辑/etc/docker/daemon.json。如果尚不存在，请创建它。假设文件为空，请添加以下内容。

{

"storage-driver": "devicemapper"

}

查看每个存储驱动程序的所有存储选项：

* [稳定](https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)
* [边缘](https://docs.docker.com/edge/engine/reference/commandline/dockerd/#storage-driver-options)

如果daemon.json文件包含格式错误的JSON，则Docker无法启动。

3 启动Docker。

$ sudo systemctl start docker

4 证守护程序正在使用devicemapper存储驱动程序。使用 docker info命令并查找Storage Driver。

$ docker info

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 17.03.1-ce

Storage Driver: devicemapper

Pool Name: docker-202:1-8413957-pool

Pool Blocksize: 65.54 kB

Base Device Size: 10.74 GB

Backing Filesystem: xfs

Data file: /dev/loop0

Metadata file: /dev/loop1

Data Space Used: 11.8 MB

Data Space Total: 107.4 GB

Data Space Available: 7.44 GB

Metadata Space Used: 581.6 KB

Metadata Space Total: 2.147 GB

Metadata Space Available: 2.147 GB

Thin Pool Minimum Free Space: 10.74 GB

Udev Sync Supported: true

Deferred Removal Enabled: false

Deferred Deletion Enabled: false

Deferred Deleted Device Count: 0

Data loop file: /var/lib/docker/devicemapper/data

Metadata loop file: /var/lib/docker/devicemapper/metadata

Library Version: 1.02.135-RHEL7 (2016-11-16)

<output truncated>

该主机以loop-lvm模式运行，生产系统不支持该模式。这可以通过以下事实来表明：Data loop file 和Metadata loop file均位于下方的文件中 /var/lib/docker/devicemapper。这些是环回安装的稀疏文件。对于生产系统，请参阅为生产 配置直接lvm​​模式。

##### 配置Direct-LVM模式进行生产

使用devicemapper存储驱动程序的生产主机必须使用direct-lvm 模式。此模式使用块设备创建精简池。这比使用环回设备更快，更有效地使用系统资源，并且块设备可以根据需要增长。但是，与loop-lvm 模式相比，需要更多的设置。

满足前提条件后，请按照以下步骤将Docker配置为devicemapper以direct-lvm模式使用存储驱动程序 。

警告：更改存储驱动程序会使您已经创建的所有容器在本地系统上均不可访问。使用docker save保存的容器，并推动现有镜像多克尔集线器或私人仓库，这样你就不会需要稍后重新创建。

###### 允许DOCKER配置DIRECT-LVM模式

使用Docker 17.06及更高版本，Docker可以为您管理块设备，从而简化direct-lvm模式配置。这仅适用于新的Docker设置。您只能使用单个块设备。如果需要使用多个块设备，请手动配置direct-lvm模式。添加了以下新的配置选项：

| **选项** | **描述** | **需要？** | **默认** | **例** |
| --- | --- | --- | --- | --- |
| dm.directlvm\_device | 要为其配置的块设备的路径direct-lvm。 | 是 |  | dm.directlvm\_device="/dev/xvdf" |
| dm.thinp\_percent | 传入的块设备中用于存储的空间百分比。 | 没有 | 95 | dm.thinp\_percent=95 |
| dm.thinp\_metapercent | 来自传入的块设备的用于元数据存储的空间百分比。 | 没有 | 1个 | dm.thinp\_metapercent=1 |
| dm.thinp\_autoextend\_threshold | lvm应该何时自动扩展精简池的阈值占总存储空间的百分比。 | 没有 | 80 | dm.thinp\_autoextend\_threshold=80 |
| dm.thinp\_autoextend\_percent | 触发自动扩展时增加精简池的百分比。 | 没有 | 20 | dm.thinp\_autoextend\_percent=20 |
| dm.directlvm\_device\_force | 即使文件系统已经存在，是否格式化块设备。如果设置为，false并且存在文件系统，则会记录错误，并且文件系统将保持不变。 | 没有 | 假 | dm.directlvm\_device\_force=true |

编辑daemon.json文件并设置适当的选项，然后重新启动Docker以使更改生效。以下daemon.json配置设置了上表中的所有选项。

{

"storage-driver": "devicemapper",

"storage-opts": [

"dm.directlvm\_device=/dev/xdf",

"dm.thinp\_percent=95",

"dm.thinp\_metapercent=1",

"dm.thinp\_autoextend\_threshold=80",

"dm.thinp\_autoextend\_percent=20",

"dm.directlvm\_device\_force=false"

]

}

查看每个存储驱动程序的所有存储选项：

稳定

边缘

重新启动Docker以使更改生效。Docker调用命令为您配置块设备。

警告：不支持在Docker为您准备好块设备后更改这些值，并且会导致错误。

您仍然需要执行定期维护任务。

###### 手动配置DIRECT-LVM模式

以下过程创建一个配置为精简池的逻辑卷，以用作存储池的后备。假定您有一个备用块设备，该设备/dev/xvdf具有足够的可用空间来完成任务。设备标识符和卷大小在您的环境中可能会有所不同，因此您应该在整个过程中替换您自己的值。该过程还假定Docker守护程序处于stopped状态中。

1 标识要使用的块设备。该设备位于/dev/（例如/dev/xvdf）下方 ，并且需要足够的可用空间来存储镜像和容器层，以供主机运行的工作负载使用。固态驱动器是理想的。

2 停止Docker。

$ sudo systemctl stop docker

3 安装以下软件包：

RHEL / CentOS的：device-mapper-persistent-data，lvm2，和所有的依赖

Ubuntu的/ Debian的：thin-provisioning-tools，lvm2，和所有的依赖

4 使用pvcreate命令在步骤1的块设备上创建物理卷 。将您的设备名称替换为/dev/xvdf。

警告：接下来的几个步骤具有破坏性，因此请确保您指定了正确的设备！

$ sudo pvcreate /dev/xvdf

Physical volume "/dev/xvdf" successfully created.

5 docker使用vgcreate 命令在同一设备上创建卷组。

$ sudo vgcreate docker /dev/xvdf

Volume group "docker" successfully created

6 使用 命令创建两个名为thinpool和的逻辑卷。最后一个参数指定可用空间量，以在空间不足时允许自动扩展数据或元数据，这是一个临时的权宜之计。这些是推荐值。thinpoolmetalvcreate

$ sudo lvcreate --wipesignatures y -n thinpool docker -l 95%VG

Logical volume "thinpool" created.

$ sudo lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG

Logical volume "thinpoolmeta" created.

7 使用命令将卷转换为精简池和精简池元数据的存储位置lvconvert。

$ sudo lvconvert -y \

--zero n \

-c 512K \

--thinpool docker/thinpool \

--poolmetadata docker/thinpoolmeta

WARNING: Converting logical volume docker/thinpool and docker/thinpoolmeta to

thin pool's data and metadata volumes with metadata wiping.

THIS WILL DESTROY CONTENT OF LOGICAL VOLUME (filesystem etc.)

Converted docker/thinpool to thin pool.

8 通过lvm配置文件配置精简池的自动扩展。

$ sudo vi /etc/lvm/profile/docker-thinpool.profile

9 指定thin\_pool\_autoextend\_threshold和thin\_pool\_autoextend\_percent 值。

thin\_pool\_autoextend\_threshold是lvm 尝试自动扩展可用空间之前使用的空间百分比（100 =禁用，不推荐）。

thin\_pool\_autoextend\_percent 是自动扩展时要添加到设备的空间量（0 =禁用）。

当磁盘使用率达到80％时，下面的示例将容量增加20％。

activation {

thin\_pool\_autoextend\_threshold=80

thin\_pool\_autoextend\_percent=20

}

保存文件。

10 使用lvchange命令应用LVM配置文件。

$ sudo lvchange --metadataprofile docker-thinpool docker/thinpool

Logical volume docker/thinpool changed.

11 确保已启用对逻辑卷的监视。

$ sudo lvs -o+seg\_monitor

LV VG Attr LSize Pool Origin Data% Meta% Move Log Cpy%Sync Convert Monitor

thinpool docker twi-a-t--- 95.00g 0.00 0.01 not monitored

如果该Monitor列中的输出如上所述报告该卷为 not monitored，则需要显式启用监视。如果没有此步骤，则无论应用的配置文件中的任何设置如何，逻辑卷的自动扩展都不会发生。

$ sudo lvchange --monitor y docker/thinpool

通过sudo lvs -o+seg\_monitor再次运行该命令，再次检查是否已启用监视 。该Monitor列现在应报告逻辑卷正在monitored。

12 如果您曾经在此主机上运行过Docker（如果已/var/lib/docker/ 存在），或者将其移开，那么Doc​​ker可以使用新的LVM池来存储镜像和容器的内容。

$ sudo su -

# mkdir /var/lib/docker.bk

# mv /var/lib/docker/\* /var/lib/docker.bk

# exit

如果以下任何步骤失败，而您需要还原，则可以将/var/lib/docker其删除 并替换为/var/lib/docker.bk。

13 编辑/etc/docker/daemon.json并配置devicemapper存储驱动程序所需的选项 。如果该文件以前为空，则现在应包含以下内容：

14 启动Docker。

systemd：

$ sudo systemctl start docker

服务内容：

$ sudo service docker start

15 使用验证Docker是否正在使用新配置docker info。

$ docker info

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 17.03.1-ce

Storage Driver: devicemapper

Pool Name: docker-thinpool

Pool Blocksize: 524.3 kB

Base Device Size: 10.74 GB

Backing Filesystem: xfs

Data file:

Metadata file:

Data Space Used: 19.92 MB

Data Space Total: 102 GB

Data Space Available: 102 GB

Metadata Space Used: 147.5 kB

Metadata Space Total: 1.07 GB

Metadata Space Available: 1.069 GB

Thin Pool Minimum Free Space: 10.2 GB

Udev Sync Supported: true

Deferred Removal Enabled: true

Deferred Deletion Enabled: true

Deferred Deleted Device Count: 0

Library Version: 1.02.135-RHEL7 (2016-11-16)

<output truncated>

如果Docker配置正确，则Data fileand Metadata file为空白，池名称为docker-thinpool。

16 确认配置正确后，可以删除/var/lib/docker.bk包含先前配置的 目录。

$ sudo rm -rf /var/lib/docker.bk

#### 管理devicemapper

##### 监控精简池

不要仅依靠LVM自动扩展。卷组自动扩展，但是该卷仍然可以填满。您可以使用lvs或监视卷上的可用空间lvs -a。考虑在操作系统级别使用监视工具，例如Nagios。

要查看LVM日志，可以使用journalctl：

$ sudo journalctl -fu dm-event.service

如果精简池反复出现问题，则可以在中将存储选项设置 dm.min\_free\_space为一个值（代表百分比） /etc/docker/daemon.json。例如，将其设置为10确保当可用空间为10％或接近10％时，操作将失败并发出警告。请参阅引擎守护程序参考中的 存储驱动程序选项。

##### 增加正在运行的设备上的容量

您可以在运行的瘦池设备上增加池的容量。如果数据的逻辑卷已满并且卷组已满，这将很有用。具体过程取决于您使用的是 loop-lvm精简池还是 Direct-lvm精简池。

###### 调整LOOP-LVM精简池的大小

调整loop-lvm精简池大小的最简单方法是 使用device\_tool实用程序，但您可以改用操作系统实用程序 。

###### 使用device\_tool实用程序

Moby / moby Github存储库中device\_tool.go提供了 一个由社区提供的脚本，称为。您可以使用此工具来调整精简池的大小，从而避免了上述漫长的过程。该工具不能保证能正常工作，但您只能在非生产系统上使用。loop-lvmloop-lvm

如果不想使用device\_tool，则可以手动调整精简池的大小。

1 要使用该工具，请克隆Github存储库，更改为 contrib/docker-device-tool，然后按照中的说明README.md 编译该工具。

2 使用工具。以下示例将精简池的大小调整为200GB。

$ ./device\_tool resize 200GB

使用操作系统实用程序

如果不想使用device-tool实用程序，则可以loop-lvm使用以下过程手动调整精简池的大小。

在loop-lvm模式下，回送设备用于存储数据，另一个用于存储元数据。loop-lvm仅支持测试模式，因为它具有明显的性能和稳定性缺点。

如果您使用loop-lvm模式，则输出的docker info显示为Data loop file和的文件路径Metadata loop file：

$ docker info |grep 'loop file'

Data loop file: /var/lib/docker/devicemapper/data

Metadata loop file: /var/lib/docker/devicemapper/metadata

请按照以下步骤增加精简池的大小。在此示例中，精简池为100 GB，并增加到200 GB。

1 列出设备的大小。

$ sudo ls -lh /var/lib/docker/devicemapper/

total 1175492

-rw------- 1 root root 100G Mar 30 05:22 data

-rw------- 1 root root 2.0G Mar 31 11:17 metadata

2 data使用truncate命令将文件的大小增加到200 G，该命令用于增加或减小文件的大小。请注意，减小大小是一种破坏性操作。

$ sudo truncate -s 200G /var/lib/docker/devicemapper/data

3 验证文件大小已更改。

$ sudo ls -lh /var/lib/docker/devicemapper/

total 1.2G

-rw------- 1 root root 200G Apr 14 08:47 data

-rw------- 1 root root 2.0G Apr 19 13:27 metadata

4 回送文件已在磁盘上更改，但在内存中未更改。以GB为单位列出内存中回送设备的大小。重新加载，然后再次列出尺寸。重新加载后，大小为200 GB。

$ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]

100

$ sudo losetup -c /dev/loop0

$ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]

200

5 重新加载devicemapper精简池。

a首先获取池名称。池名称是第一个字段，以`：`分隔。此命令将其提取。

$ sudo dmsetup status | grep ' thin-pool ' | awk -F ': ' {'print $1'}

docker-8:1-123141-pool

b 转储瘦池的设备映射器表。

$ sudo dmsetup table docker-8:1-123141-pool

0 209715200 thin-pool 7:1 7:0 128 32768 1 skip\_block\_zeroing

C。使用输出的第二个字段计算精简池的总扇区数。该数字以512 k扇区表示。100G文件具有209715200 512-k扇区。如果将此数字加倍到200G，则会得到419430400 512-k扇区。

d。使用以下三个dmsetup 命令，用新的扇区号重新加载精简池。

$ sudo dmsetup suspend docker-8:1-123141-pool

$ sudo dmsetup reload docker-8:1-123141-pool --table '0 419430400 thin-pool 7:1 7:0 128 32768 1 skip\_block\_zeroing'

$ sudo dmsetup resume docker-8:1-123141-pool

调整直接LVM​​精简池的大小

要扩展direct-lvm精简池，您需要首先将新的块设备连接到Docker主机，并记下内核分配给它的名称。在此示例中，新的块设备为/dev/xvdg。

按照此过程扩展direct-lvm精简池，替换块设备和其他参数以适合您的情况。

1 收集有关您的卷组的信息。

使用该pvdisplay命令查找您的精简池当前正在使用的物理块设备以及卷组的名称。

$ sudo pvdisplay |grep 'VG Name'

PV Name /dev/xvdf

VG Name docker

在以下步骤中，根据需要替换您的块设备或卷组名称。

2 使用vgextend带有VG Name 上一步中的命令和新块设备的名称来扩展卷组。

$ sudo vgextend docker /dev/xvdg

Physical volume "/dev/xvdg" successfully created.

Volume group "docker" successfully extended

3 扩展docker/thinpool逻辑卷。此命令立即使用100％的卷，而不会自动扩展。要扩展元数据瘦池，请使用docker/thinpool\_tmeta。

$ sudo lvextend -l+100%FREE -n docker/thinpool

Size of logical volume docker/thinpool\_tdata changed from 95.00 GiB (24319 extents) to 198.00 GiB (50688 extents).

Logical volume docker/thinpool\_tdata successfully resized.

4 使用Data Space Available的输出中的字段验证新的精简池大小docker info。如果docker/thinpool\_tmeta改为扩展逻辑卷，请查找Metadata Space Available。

Storage Driver: devicemapper

Pool Name: docker-thinpool

Pool Blocksize: 524.3 kB

Base Device Size: 10.74 GB

Backing Filesystem: xfs

Data file:

Metadata file:

Data Space Used: 212.3 MB

Data Space Total: 212.6 GB

Data Space Available: 212.4 GB

Metadata Space Used: 286.7 kB

Metadata Space Total: 1.07 GB

Metadata Space Available: 1.069 GB

<output truncated>

##### devicemapper重新启动后激活

如果重新引导主机，然后发现docker服务无法启动，请查找错误“非现有设备”。您需要使用以下命令重新激活逻辑卷：

sudo lvchange -ay docker/thinpool

#### devicemapper存储驱动程序如何工作

警告：请勿直接操作其中的任何文件或目录 /var/lib/docker/。这些文件和目录由Docker管理。

lsblk从操作系统的角度，使用命令查看设备及其池：

$ sudo lsblk

NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT

xvda 202:0 0 8G 0 disk

└─xvda1 202:1 0 8G 0 part /

xvdf 202:80 0 100G 0 disk

├─docker-thinpool\_tmeta 253:0 0 1020M 0 lvm

│ └─docker-thinpool 253:2 0 95G 0 lvm

└─docker-thinpool\_tdata 253:1 0 95G 0 lvm

└─docker-thinpool 253:2 0 95G 0 lvm

使用mount命令查看Docker正在使用的安装点：

$ mount |grep devicemapper

/dev/xvda1 on /var/lib/docker/devicemapper type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

使用时devicemapper，Docker将镜像和图层内容存储在thinpool中，并通过将它们安装在的子目录下将它们公开给容器/var/lib/docker/devicemapper/。

##### 磁盘上的镜像和容器层

该/var/lib/docker/devicemapper/metadata/目录包含有关Devicemapper配置本身以及每个存在的镜像和容器层的元数据。该devicemapper存储驱动程序使用快照，这个元数据包含有关这些快照的信息。这些文件为JSON格式。

该/var/lib/docker/devicemapper/mnt/目录包含每个存在的镜像和容器层的安装点。镜像层安装点为空，但是容器的安装点显示容器中从容器内部出现的文件系统。

##### 镜像分层和共享

该devicemapper存储驱动程序使用专用的块设备，而不是格式化的文件系统，并在写入时复制（全体）操作上以获得最大性能的块级的文件进行操作。

###### 快照

devicemapper快照的另一个功能是使用快照（有时也称为 精简设备或虚拟设备），它们将在每一层中引入的差异存储为非常小的，轻量级的精简池。快照具有许多优点：

1 容器之间共有的共享层除非可写，否则仅在磁盘上存储一次。例如，如果您有10个基于的不同镜像，则alpine该alpine镜像及其所有父镜像仅在磁盘上存储一次。

2 快照是写时复制（CoW）策略的实现。这意味着仅当给定文件或目录被该容器修改或删除时，该文件或目录才被复制到该容器的可写层。

3 由于devicemapper在块级别进行操作，因此可同时修改可写层中的多个块。

4 可以使用标准的操作系统级备份实用程序来备份快照。只需复制一份即可/var/lib/docker/devicemapper/。

###### DEVICEMAPPER工作流程

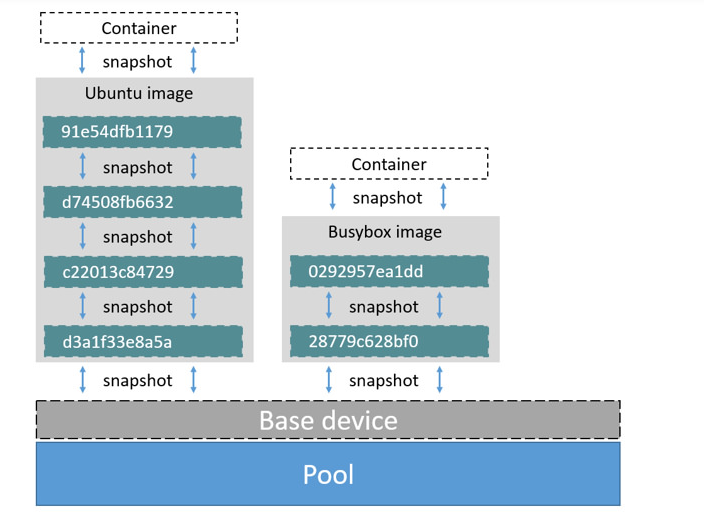
当您使用devicemapper存储驱动程序启动Docker时，与镜像和容器层相关的所有对象都存储在中 /var/lib/docker/devicemapper/，该对象由一个或多个块级设备（回送设备（仅测试）或物理磁盘）支持。

1所述基部设备是最低级的对象。这就是瘦池本身。您可以使用检查它docker info。它包含一个文件系统。此基本设备是每个镜像和容器层的起点。基本设备是Device Mapper实施细节，而不是Docker层。

2 有关基本设备以及每个镜像或容器层的元数据/var/lib/docker/devicemapper/metadata/以JSON格式存储 。这些层是写时复制快照，这意味着它们是空的，直到它们脱离其父层。

3 每个容器的可写层都安装在中的安装点上 /var/lib/docker/devicemapper/mnt/。每个只读镜像层和每个停止的容器都存在一个空目录。

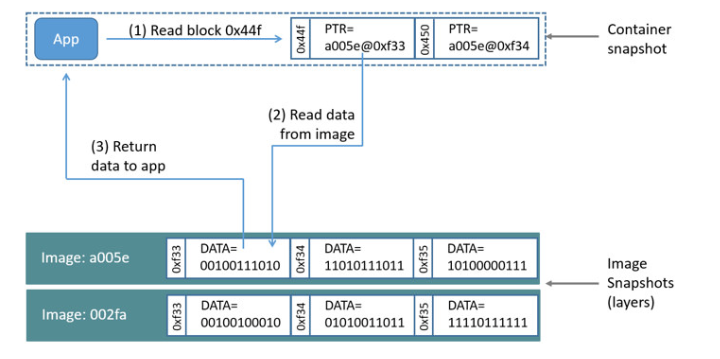
每个镜像层都是其下一层的快照。每个镜像的最低层是池中存在的基本设备的快照。运行容器时，它是容器所基于的镜像的快照。以下示例显示了具有两个正在运行的容器的Docker主机。第一个是ubuntu 容器，第二个是busybox容器。



#### 容器如何读写 devicemapper

##### 读取文件

使用devicemapper，读取发生在块级别。下图显示了0x44f在示例容器中读取单个块（）的高级过程。



应用程序0x44f对容器中的块发出读取请求。因为容器是镜像的精简快照，所以它没有块，但是它有一个指向存在它的最近父镜像上的块的指针，并从那里读取该块。现在，该块存在于容器的内存中。

##### 写文件

写入新文件：使用devicemapper驱动程序，通过按需分配操作可以将新数据写入容器。新文件的每个块都分配在容器的可写层中，并将该块写入那里。

更新现有文件：从文件所在的最近层读取文件的相关块。当容器写入文件时，仅将修改后的块写入容器的可写层。

删除文件或目录：在容器的可写层中删除文件或目录时，或者在镜像层删除其父层中存在的文件时，devicemapper存储驱动程序会拦截对该文件或目录的进一步读取尝试，并响应文件或目录不存在。

写 入然后删除文件：如果容器写入文件，随后又删除了文件，则所有这些操作都在容器的可写层中进行。在这种情况下，如果使用direct-lvm，则释放块。如果使用loop-lvm，则可能无法释放这些块。这是不用于loop-lvm生产的另一个原因 。

#### 设备映射器和Docker性能

allocate-on demand性能影响：

在devicemapper存储驱动程序使用的allocate-on-demand操作以从薄池的新块分配到容器中的可写层。每个块为64KB，因此这是用于写入的最小空间量。

写时复制对性能的影响：容器第一次修改特定块时，该块将被写入容器的可写层。由于这些写操作发生在块级别而不是文件级别，因此对性能的影响最小。但是，写入大量块仍可能对性能产生负面影响，devicemapper在这种情况下，存储驱动程序的性能可能实际上比其他存储驱动程序差。对于繁重的工作负载，应使用数据卷，该数据卷会完全绕过存储驱动程序。

#### 绩效最佳实践

使用devicemapper 存储驱动程序时，请记住这些事项以最大化性能。

1 使用direct-lvm：该loop-lvm模式性能不佳，切勿在生产中使用。

2 使用快速存储：固态驱动器（SSD）提供比旋转磁盘更快的读写速度。

3 内存使用率：devicemapper比其他一些存储驱动程序使用更多的内存。每个启动的容器都将其文件的一个或多个副本加载到内存中，具体取决于同时修改同一文件的多少块。由于内存压力，devicemapper对于高密度用例中的某些工作负载，存储驱动程序可能不是正确的选择。

4 将卷用于繁重的写工作负载：卷可为繁重的写工作负载提供最佳和最可预测的性能。这是因为它们绕过了存储驱动程序，并且不会产生任何精简配置和写入时复制所带来的潜在开销。卷还有其他好处，例如，允许您在容器之间共享数据，并且即使没有正在运行的容器正在使用它们也可以持久存储。

5 注意：使用devicemapper和json-file日志驱动程序时，默认情况下，由容器生成的日志文件仍存储在Docker的dataroot目录中/var/lib/docker。如果您的容器生成大量日志消息，则可能导致磁盘使用率增加或由于磁盘已满而无法管理系统。您可以配置 日志驱动程序以在外部存储容器日志。

### ### 使用OverlayFS存储驱动程序

OverlayFS是一种现代的联合文件系统，与AUFS类似，但速度更快且实现更简单。Docker为OverlayFS提供了两个存储驱动程序：原始的overlay，更新的和更稳定的overlay2。

本主题将Linux内核驱动OverlayFS程序称为overlay或将Docker存储驱动器称为或overlay2。

注意：如果使用OverlayFS，请使用overlay2驱动程序而不是 overlay驱动程序，因为它在inode利用率方面更为有效。要使用新的驱动程序，您需要Linux内核的版本4.0或更高版本，或者使用3.10.0-514及更高版本的RHEL或CentOS。

有关overlayvs 之间差异的更多信息overlay2，请检查 Docker存储驱动程序。

#### 先决条件

如果满足以下先决条件，则支持OverlayFS：

1 该overlay2驱动程序受Docker Engine-Community和Docker EE 17.06.02-ee5及更高版本支持，并且是推荐的存储驱动程序。

2 Linux内核的版本4.0或更高版本，或使用内核的版本3.10.0-514或更高版本的RHEL或CentOS。如果使用较旧的内核，overlay则不建议使用驱动程序。

3 在overlay和overlay2驱动程序支持xfs支持的文件系统，但只d\_type=true启用。

使用xfs\_info验证ftype选项设置为1。要xfs正确格式化 文件系统，请使用标志-n ftype=1。

警告：在不支持d\_type的XFS上运行现在会导致Docker跳过使用overlay或overlay2驱动程序的尝试。现有安装将继续运行，但会产生错误。这是为了允许用户迁移其数据。在将来的版本中，这将是一个致命错误，它将阻止Docker启动。

4 更改存储驱动程序将使现有容器和镜像在本地系统上不可访问。使用docker save保存你已经建立的任何镜像或改变存储驱动程序之前，他们推到码头工人集线器或私人注册，这样就不需要再后来创建它们。

#### 使用overlay或overlay2存储驱动程序配置Docker

强烈建议您overlay2尽可能使用驱动程序，而不要使用overlay驱动程序。Docker EE 不支持该overlay驱动程序。

要将Docker配置为使用overlay存储驱动程序，您的Docker主机必须运行Linux内核的3.18版本（最好是更新的）并加载了覆盖内核模块。对于overlay2驱动程序，您的内核版本必须为4.0或更高版本。

在执行此过程之前，您必须首先满足所有 先决条件。

以下步骤概述了如何配置overlay2存储驱动程序。如果需要使用旧版overlay驱动程序，请改为指定它。

1 停止Docker。

$ sudo systemctl stop docker

2 将的内容复制/var/lib/docker到一个临时位置。

$ cp -au /var/lib/docker /var/lib/docker.bk

3 如果要使用与使用的备份文件系统不同的备份文件系统 /var/lib/，请格式化该文件系统并将其装入/var/lib/docker。确保添加此安装座/etc/fstab以使其永久。

4 编辑/etc/docker/daemon.json。如果尚不存在，请创建它。假设文件为空，请添加以下内容。

{

"storage-driver": "overlay2"

}

如果daemon.json文件包含格式错误的JSON，则Docker无法启动。

5 启动Docker。

$ sudo systemctl start docker

6 验证守护程序正在使用overlay2存储驱动程序。使用docker info命令并查找Storage Driver和 Backing filesystem。

$ docker info

Containers: 0

Images: 0

Storage Driver: overlay2

Backing Filesystem: xfs

Supports d\_type: true

Native Overlay Diff: true

<output truncated>

码头工人正在使用的overlay2存储驱动程序，并自动创建覆盖所要求的安装lowerdir，upperdir，merged，和workdir结构。

继续阅读有关OverlayFS在您的Docker容器中如何工作的详细信息，以及性能建议以及有关其与不同后备文件系统兼容性的限制的信息。

#### 如何overlay2驱动器的工作原理

如果您仍在使用overlay驱动程序而不是overlay2，请参阅 覆盖驱动程序的工作方式。

OverlayFS在单个Linux主机上分层两个目录，并将它们显示为单个目录。这些目录称为“ 层”，统一过程称为“ 联合安装”。OverlayFS指的是下层目录as lowerdir和上层目录a upperdir。统一视图通过其自己的目录公开merged。

该overlay2驱动程序本机最多支持128个较低的OverlayFS层。此功能为与图层相关的Docker命令（例如docker build和）提供了更好的性能docker commit，并且在备份文件系统上消耗了更少的inode。

##### 磁盘上的镜像和容器层

使用下载五层镜像后docker pull ubuntu，您可以在下看到六个目录/var/lib/docker/overlay2。

警告：请勿直接操作其中的任何文件或目录 /var/lib/docker/。这些文件和目录由Docker管理。

$ ls -l /var/lib/docker/overlay2

total 24

drwx------ 5 root root 4096 Jun 20 07:36 223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7

drwx------ 3 root root 4096 Jun 20 07:36 3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b

drwx------ 5 root root 4096 Jun 20 07:36 4e9fa83caff3e8f4cc83693fa407a4a9fac9573deaf481506c102d484dd1e6a1

drwx------ 5 root root 4096 Jun 20 07:36 e8876a226237217ec61c4baf238a32992291d059fdac95ed6303bdff3f59cff5

drwx------ 5 root root 4096 Jun 20 07:36 eca1e4e1694283e001f200a667bb3cb40853cf2d1b12c29feda7422fed78afed

drwx------ 2 root root 4096 Jun 20 07:36 l

新的l（小写L）目录包含缩短的层标识符作为符号链接。这些标识符用于避免达到mount命令参数的页面大小限制。

$ ls -l /var/lib/docker/overlay2/l

total 20

lrwxrwxrwx 1 root root 72 Jun 20 07:36 6Y5IM2XC7TSNIJZZFLJCS6I4I4 -> ../3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/diff

lrwxrwxrwx 1 root root 72 Jun 20 07:36 B3WWEFKBG3PLLV737KZFIASSW7 -> ../4e9fa83caff3e8f4cc83693fa407a4a9fac9573deaf481506c102d484dd1e6a1/diff

lrwxrwxrwx 1 root root 72 Jun 20 07:36 JEYMODZYFCZFYSDABYXD5MF6YO -> ../eca1e4e1694283e001f200a667bb3cb40853cf2d1b12c29feda7422fed78afed/diff

lrwxrwxrwx 1 root root 72 Jun 20 07:36 NFYKDW6APBCCUCTOUSYDH4DXAT -> ../223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7/diff

lrwxrwxrwx 1 root root 72 Jun 20 07:36 UL2MW33MSE3Q5VYIKBRN4ZAGQP -> ../e8876a226237217ec61c4baf238a32992291d059fdac95ed6303bdff3f59cff5/diff

最低层包含一个名为的文件link，其中包含缩短的标识符的名称，以及一个diff包含该层的内容的目录。

$ ls /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/

diff link

$ cat /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/link

6Y5IM2XC7TSNIJZZFLJCS6I4I4

$ ls /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/diff

bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

最低的第二层以及每个较高的层包含一个名为的文件lower，该文件表示其父目录，以及一个diff包含其内容的目录。它还包含一个merged目录，该目录包含其父层及其自身的统一内容，以及一个work目录，供OverlayFS在内部使用。

$ ls /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7

diff link lower merged work

$ cat /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7/lower

l/6Y5IM2XC7TSNIJZZFLJCS6I4I4

$ ls /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7/diff/

etc sbin usr var

要查看将overlay存储驱动程序与Docker一起使用时存在的安装，请使用mount命令。为了易于阅读，下面的输出被截断了。

$ mount | grep overlay

overlay on /var/lib/docker/overlay2/9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/merged

type overlay (rw,relatime,

lowerdir=l/DJA75GUWHWG7EWICFYX54FIOVT:l/B3WWEFKBG3PLLV737KZFIASSW7:l/JEYMODZYFCZFYSDABYXD5MF6YO:l/UL2MW33MSE3Q5VYIKBRN4ZAGQP:l/NFYKDW6APBCCUCTOUSYDH4DXAT:l/6Y5IM2XC7TSNIJZZFLJCS6I4I4,

upperdir=9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/diff,

workdir=9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/work)

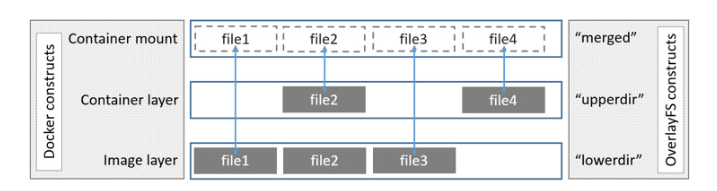
的rw在第二行示出了overlay安装件是可读写的。

#### 如何overlay驱动器的工作原理

此内容仅适用于overlay驱动程序。Docker建议使用overlay2不同的 驱动程序。见 该overlay2司机如何工作 的overlay2。

OverlayFS在单个Linux主机上分层两个目录，并将它们显示为单个目录。这些目录称为“ 层”，统一过程称为“ 联合安装”。OverlayFS指的是下层目录as lowerdir和上层目录a upperdir。统一视图通过其自己的目录公开merged。

下图显示了如何将Docker镜像和Docker容器分层。镜像层是lowerdir，容器层是upperdir。统一视图通过名为的目录公开，该目录merged实际上是容器的安装点。该图显示了Docker构造如何映射到OverlayFS构造。



在镜像层和容器层包含相同文件的情况下，容器层“获胜”并掩盖了镜像层中相同文件的存在。

该overlay驱动程序仅适用于两层。这意味着不能将多层镜像实现为多个OverlayFS层。而是将每个镜像层实现为自己的目录/var/lib/docker/overlay。然后，将硬链接用作节省空间的方式，以引用与较低层共享的数据。硬链接的使用会导致索引节点的过度使用，这是对传统overlay存储驱动程序的已知限制，并且可能需要对备份文件系统进行其他配置。有关详细信息，请参考overlayFS和Docker性能。

为了创建一个容器，overlay驱动程序将代表镜像顶层的目录与该容器的新目录结合在一起。镜像的顶层lowerdir位于叠加层中，并且是只读的。容器的新目录是upperdir且可写。

##### 磁盘上的镜像和容器层

以下docker pull命令显示了一个Docker主机下载一个包含五层的Docker镜像。

$ docker pull ubuntu

Using default tag: latest

latest: Pulling from library/ubuntu

5ba4f30e5bea: Pull complete

9d7d19c9dc56: Pull complete

ac6ad7efd0f9: Pull complete

e7491a747824: Pull complete

a3ed95caeb02: Pull complete

Digest: sha256:46fb5d001b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4

Status: Downloaded newer image for ubuntu:latest

###### 镜像层

每个镜像层在中都有自己的目录/var/lib/docker/overlay/，其中包含其内容，如下所示。镜像层ID与目录ID不对应。

警告：请勿直接操作其中的任何文件或目录 /var/lib/docker/。这些文件和目录由Docker管理。

$ ls -l /var/lib/docker/overlay/

total 20

drwx------ 3 root root 4096 Jun 20 16:11 38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8

drwx------ 3 root root 4096 Jun 20 16:11 55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358

drwx------ 3 root root 4096 Jun 20 16:11 824c8a961a4f5e8fe4f4243dab57c5be798e7fd195f6d88ab06aea92ba931654

drwx------ 3 root root 4096 Jun 20 16:11 ad0fe55125ebf599da124da175174a4b8c1878afe6907bf7c78570341f308461

drwx------ 3 root root 4096 Jun 20 16:11 edab9b5e5bf73f2997524eebeac1de4cf9c8b904fa8ad3ec43b3504196aa3801

镜像层目录包含该层独有的文件以及与较低层共享的数据的硬链接。这样可以有效利用磁盘空间。

$ ls -i /var/lib/docker/overlay/38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8/root/bin/ls

19793696 /var/lib/docker/overlay/38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8/root/bin/ls

$ ls -i /var/lib/docker/overlay/55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358/root/bin/ls

19793696 /var/lib/docker/overlay/55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358/root/bin/ls

###### 容器层

容器也存在于Docker主机文件系统下的磁盘上 /var/lib/docker/overlay/。如果使用ls -l命令列出正在运行的容器的子目录，那么将存在三个目录和一个文件：

$ ls -l /var/lib/docker/overlay/<directory-of-running-container>

total 16

-rw-r--r-- 1 root root 64 Jun 20 16:39 lower-id

drwxr-xr-x 1 root root 4096 Jun 20 16:39 merged

drwxr-xr-x 4 root root 4096 Jun 20 16:39 upper

drwx------ 3 root root 4096 Jun 20 16:39 work

该lower-id文件包含容器所基于的镜像顶层的ID，即Ove​​rlayFS lowerdir。

$ cat /var/lib/docker/overlay/ec444863a55a9f1ca2df72223d459c5d940a721b2288ff86a3f27be28b53be6c/lower-id

55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358

该upper目录包含容器的读写层的内容，该层对应于OverlayFS upperdir。

该merged目录是联合安装的lowerdir和upperdir，该方法包括从正在运行的容器内的文件系统的图。

该work目录在OverlayFS内部。

要查看将overlay存储驱动程序与Docker一起使用时存在的安装，请使用mount命令。为了易于阅读，下面的输出被截断了。

$ mount | grep overlay

overlay on /var/lib/docker/overlay/ec444863a55a.../merged

type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay/55f1e14c361b.../root,

upperdir=/var/lib/docker/overlay/ec444863a55a.../upper,

workdir=/var/lib/docker/overlay/ec444863a55a.../work)

的rw在第二行示出了overlay安装件是可读写的。

#### 容器如何使用overlay或进行读取和写入overlay2

##### 读取文件

考虑三种情况，其中容器打开文件以进行覆盖访问。

该文件在容器层中不存在：如果容器打开文件以进行读取访问，并且该文件在容器（upperdir）中尚不存在，则将从镜像中读取该文件（lowerdir)。这将导致很少的性能开销。

该文件仅存在于容器层中：如果容器打开文件进行读取访问，并且该文件存在于容器（upperdir）中，而不存在于镜像（lowerdir）中，则直接从容器中读取文件。

该文件在容器层和镜像层中都存在：如果容器打开文件以供读取访问，并且文件在镜像层和容器层中存在，则将读取容器层中文件的版本。容器层（upperdir）中的文件会使镜像层（lowerdir）中具有相同名称的文件模糊。

##### 修改文件或目录

考虑在某些情况下修改了容器中的文件。

1 第一次写入文件：容器第一次写入现有文件时，该文件在容器（upperdir）中不存在。的overlay/ overlay2驱动器执行一个copy\_up操作将文件从镜像（复制lowerdir）到所述容器（upperdir）。然后，容器将更改写入容器层中文件的新副本。

但是，OverlayFS在文件级别而不是块级别工作。这意味着所有OverlayFS copy\_up操作都会复制整个文件，即使该文件非常大且只有一小部分被修改。这会对容器写入性能产生明显影响。但是，有两点值得注意：

copy\_up操作仅在第一次写入给定文件时发生。随后对同一文件的写入将对已经复制到容器的文件副本进行操作。

OverlayFS仅适用于两层。这意味着性能应该比AUFS更好，后者在多层镜像中搜索文件时可能会出现明显的延迟。这一优点适用于 overlay和overlay2驱动程序。与初始读取相比，它的overlayfs2性能稍差overlayfs，因为它必须遍历更多的层，但是会缓存结果，因此这只是一个小小的代价。

2 删除文件和目录：

当一个文件是一个容器内删除，一个白斑在容器中创建文件（upperdir）。lowerdir不会删除镜像层（）中文件的版本（因为lowerdir只读）。但是，白化文件会阻止容器使用它。

当一个目录由容器内删除，一个不透明目录在容器内产生（upperdir）。这与中断文件的工作方式相同，并且即使该目录仍然存在于镜像（lowerdir）中，也可以有效地防止该目录被访问。

3 重命名目录：rename(2)仅当源路径和目标路径都位于顶层时，才允许调用目录。否则，它将返回EXDEV错误（“不允许跨设备链接”）。您的应用程序需要设计为处理EXDEV并退回到“复制和取消链接”策略。

#### OverlayFS和Docker性能

这两个overlay2和overlay驱动程序比更高性能aufs和 devicemapper。在某些情况下，性能overlay2可能会更好 btrfs。但是，请注意以下详细信息。

1 页面缓存。OverlayFS支持页面缓存共享。访问同一文件的多个容器共享该文件的单个页面缓存条目。这使得overlayand overlay2驱动程序可以有效地利用内存，并且是高密度用例（例如PaaS）的不错选择。

2 copy\_up。与AUFS一样，每当容器第一次写入文件时，OverlayFS都会执行复制操作。这会增加写入操作的延迟，尤其是对于大文件。但是，一旦文件被复制，对该文件的所有后续写入都将在上层进行，而无需进行进一步的复制操作。

OverlayFS copy\_up操作比使用AUFS进行的操作要快，这是因为AUFS支持的层数比OverlayFS还要多，并且如果在许多AUFS层中进行搜索，可能会产生更大的延迟。overlay2同样支持多层，但是可以减轻缓存对性能的影响。

3 索引节点限制。使用旧版overlay存储驱动程序可能会导致过多的inode消耗。在Docker主机上存在大量镜像和容器的情况下尤其如此。增加文件系统可用的索引节点数量的唯一方法是对其进行重新格式化。为避免遇到此问题，强烈建议您overlay2尽可能使用。

#### 绩效最佳实践

以下通用性能最佳实践也适用于OverlayFS。

使用快速存储：固态驱动器（SSD）提供比旋转磁盘更快的读写速度。

将卷用于繁重的写工作负载：卷可为繁重的写工作负载提供最佳和最可预测的性能。这是因为它们绕过了存储驱动程序，并且不会产生任何精简配置和写入时复制所带来的潜在开销。卷还有其他好处，例如，即使没有运行中的容器正在使用它们，它也允许您在容器之间共享数据并保留数据。

#### OverlayFS兼容性限制

总结OverlayFS与其他文件系统不兼容的方面：

open（2）：OverlayFS仅实现POSIX标准的子集。这可能导致某些OverlayFS操作违反POSIX标准。一种这样的操作是复制操作。假设您的应用程序调用 fd1=open("foo", O\_RDONLY)，然后fd2=open("foo", O\_RDWR)。在这种情况下，您的应用程序期望fd1并fd2引用相同的文件。但是，由于在第二次调用之后发生了复制操作open(2)，因此描述符引用了不同的文件。的fd1继续引用该文件的镜像（在lowerdir），并且fd2该文件引用在容器（upperdir）。一种解决方法是对touch导致复制操作发生的文件进行处理。所有后续open(2)无论只读还是读写访问方式，操作都引用容器中的文件（upperdir）。

yum除非yum-plugin-ovl安装了软件包，否则已知会受到影响。如果yum-plugin-ovl软件包在您的发行版（例如6.8或7.2之前的RHEL / CentOS）中不可用，则可能需要先运行，touch /var/lib/rpm/\* 然后再运行yum install。该软件包实现了touch上面针对引用的解决方法yum。

named （2）：OverlayFS不完全支持rename(2)系统调用。您的应用程序需要检测到它的故障并退回到“复制和取消链接”策略。

### ### 使用ZFS存储驱动程序

ZFS是下一代文件系统，它支持许多高级存储技术，例如卷管理，快照，校验和，压缩和重复数据删除，复制等。

它由Sun Microsystems（现为Oracle Corporation）创建，并以CDDL许可证开源。由于CDDL和GPL之间的许可不兼容，因此ZFS不能作为主线Linux内核的一部分提供。但是，Linux上的ZFS（ZoL）项目提供了树外内核模块和用户空间工具，可以分别安装它们。

Linux（ZoL）端口上的ZFS正常且成熟。但是，目前不建议将zfsDocker存储驱动程序用于生产，除非您对Linux上的ZFS有丰富的经验。

注意：Linux平台上也有ZFS的FUSE实现。不建议这样做。本地ZFS驱动程序（ZoL）经过了更多测试，性能更高且得到了更广泛的使用。本文档的其余部分指的是本机ZoL端口。

#### 先决条件

1 ZFS需要一个或多个专用块设备，最好是固态驱动器（SSD）。

2 ZFS仅支持泊坞窗引擎-社区与Ubuntu 14.04或更高版本，与zfs 包（16.04及更高版本）或zfs-native与ubuntu-zfs已安装的软件包（14.04）。

2.1 对于Ubuntu 14.04，需要先启用补充软件包存储库， ppa:zfs-native/stable然后才能安装软件包。有关 说明，请参见 https://launchpad.net/~zfs-native/+archive/ubuntu/stable。

3 Docker EE或CS-Engine或任何其他Linux平台不支持ZFS。

4 该/var/lib/docker/目录必须安装在ZFS格式的文件系统上。

5 更改存储驱动程序会使本地系统上无法访问您已经创建的任何容器。使用docker save保存的容器，并推动现有镜像多克尔集线器或私人仓库，这样就不需要再后来创建它们。

注意：不需要MountFlags=slave与Docker Engine 18.09或更高版本一起使用，因为dockerd和containerd处于不同的安装命名空间中。

#### 使用zfs存储驱动程序配置Docker

1 停止Docker。

2 复制内容/var/lib/docker/到/var/lib/docker.bk并删除的内容/var/lib/docker/。

$ sudo cp -au /var/lib/docker /var/lib/docker.bk

$ sudo rm -rf /var/lib/docker/\*

3 在一个zpool或多个专用块设备上创建一个新文件，然后将其安装到中/var/lib/docker/。确保指定了正确的设备，因为这是破坏性操作。本示例将两个设备添加到池中。

$ sudo zpool create -f zpool-docker -m /var/lib/docker /dev/xvdf /dev/xvdg

该命令将创建zpool和命名zpool-docker。该名称仅用于显示目的，您可以使用其他名称。使用检查创建和正确安装了池zfs list。

$ sudo zfs list

NAME USED AVAIL REFER MOUNTPOINT

zpool-docker 55K 96.4G 19K /var/lib/docker

4 配置Docker以使用zfs。编辑/etc/docker/daemon.json并将设置 storage-driver为zfs。如果文件以前为空，则现在应如下所示：

{

"storage-driver": "zfs"

}

保存并关闭文件。

5 启动Docker。使用docker info验证存储驱动程序zfs。

$ sudo docker info

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 17.03.1-ce

Storage Driver: zfs

Zpool: zpool-docker

Zpool Health: ONLINE

Parent Dataset: zpool-docker

Space Used By Parent: 249856

Space Available: 103498395648

Parent Quota: no

Compression: off

<output truncated>

#### 管理 zfs

##### 增加正在运行的设备上的容量

要增加的大小zpool，您需要向Docker主机添加专用的块设备，然后zpool使用以下zpool add命令将其添加到：

$ sudo zpool add zpool-docker /dev/xvdh

##### 限制容器的可写存储配额

如果要基于每个镜像/数据集实施配额，则可以设置 size存储选项以限制单个容器可用于其可写层的空间量。

编辑/etc/docker/daemon.json并添加以下内容：

{

"storage-driver": "zfs",

"storage-opts": ["size=256M"]

}

查看每个存储驱动程序的所有存储选项：

稳定

边缘

保存并关闭文件，然后重新启动Docker。

#### zfs存储驱动程序如何工作

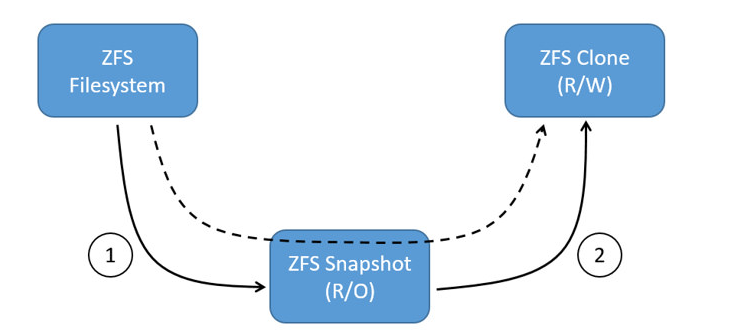
ZFS使用以下对象：

1 文件系统：精简配置，并zpool根据需要分配空间。

2 快照：文件系统的只读节省空间的时间点副本。

3 clones：快照的读写副本。用于存储与上一层的差异。

创建克隆的过程：



1 从文件系统创建一个只读快照。

2 从快照创建可写克隆。这包含与父层的任何差异。

文件系统，快照和克隆都从底层分配空间 zpool。

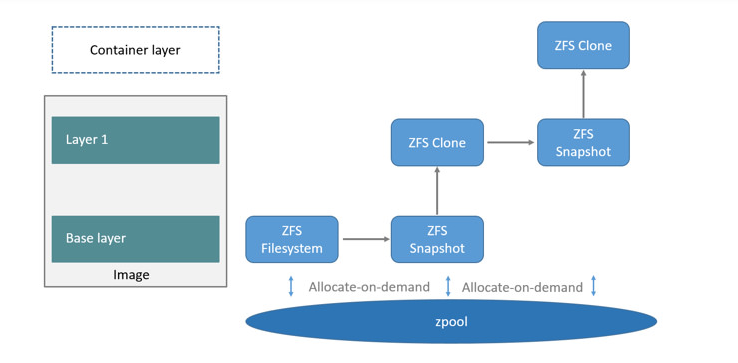
##### 磁盘上的镜像和容器层

每个正在运行的容器的统一文件系统都安装在中的安装点上 /var/lib/docker/zfs/graph/。继续阅读有关统一文件系统组成的说明。

##### 镜像分层和共享

镜像的基础层是ZFS文件系统。每个子层都是一个基于其下一层的ZFS快照的ZFS克隆。容器是一个ZFS克隆，它基于从其创建镜像的顶层的ZFS快照。

下图显示了如何将其与基于两层镜像的运行容器一起放置。



启动容器时，将按顺序执行以下步骤：

1 镜像的基础层作为ZFS文件系统存在于Docker主机上。

2 其他镜像层是数据集的克隆，这些数据集在其正下方托管着镜像层。

在该图中，通过获取基础层的ZFS快照，然后从该快照创建克隆来添加“第1层”。该克隆是可写的，并从zpool按需消耗空间。快照是只读的，将基本层保留为不变的对象。

3 启动容器后，可写层将添加到镜像上方。

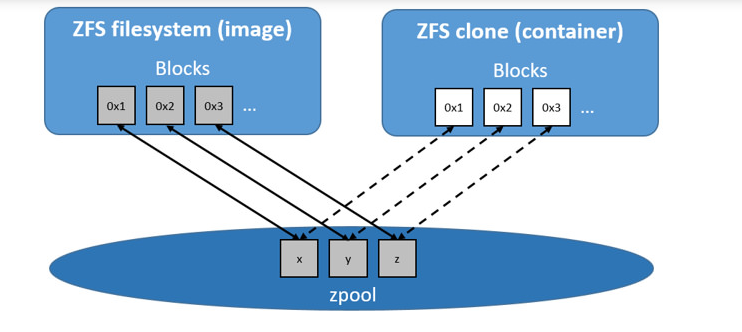
在该图中，通过对镜像顶层（第1层）进行快照并从该快照创建克隆来创建容器的读写层。

4 当容器修改其可写层的内容时，将为已更改的块分配空间。默认情况下，这些块为128k。

#### 容器如何读写 zfs

##### 读取文件

每个容器的可写层都是ZFS克隆，它与从其创建的数据集（其父层的快照）共享所有数据。即使正在读取的数据来自较深的一层，读取操作也是最容易的。下图说明了块共享的工作方式：



##### 写文件

编写一个新文件：根据需要从底层分配空间，zpool 并将这些块直接写入容器的可写层。

修改现有文件：仅为更改的块分配空间，然后使用写时复制（CoW）策略将这些块写入容器的可写层。这样可以最小化层的大小并提高写入性能。

删除文件或目录：

删除较低层中存在的文件或目录时，即使文件或目录仍存在于较低的只读层中，ZFS驱动程序也会在容器的可写层中屏蔽该文件或目录的存在。

如果在容器的可写层中创建然后删除文件或目录，则这些块将被回收zpool。

#### ZFS和Docker性能

有几个因素会影响使用zfs存储驱动程序的Docker性能 。

1 内存：内存对ZFS性能有重大影响。ZFS最初是为具有大量内存的大型企业级服务器设计的。

2 ZFS功能：ZFS包含重复数据删除功能。使用此功能可以节省磁盘空间，但会占用大量内存。建议zpool您在与Docker一起使用时禁用此功能，除非您正在使用SAN，NAS或其他硬件RAID技术。

3 ZFS缓存：ZFS将磁盘块缓存在称为自适应替换缓存（ARC）的内存结构中。ZFS 的Single Copy ARC功能允许块的多个克隆共享一个块的单个缓存副本。使用此功能，多个运行中的容器可以共享一个缓存块的单个副本。此功能使ZFS成为PaaS和其他高密度用例的不错选择。

4碎片：碎片是ZFS等写时复制文件系统的自然副产品。ZFS通过使用128k的小块大小来减轻这种情况。ZFS意向日志（ZIL）和写入（延迟写入）的合并也有助于减少碎片。您可以使用监视碎片 zpool status。但是，如果不重新格式化和还原文件系统，就无法对ZFS进行碎片整理。

对Linux使用本地ZFS驱动程序：由于性能不佳，不建议使用ZFS FUSE实现。

#### 绩效最佳实践

使用快速存储：固态驱动器（SSD）提供比旋转磁盘更快的读写速度。

将卷用于繁重的写工作负载：卷可为繁重的写工作负载提供最佳和最可预测的性能。这是因为它们绕过了存储驱动程序，并且不会产生任何精简配置和写入时复制所带来的潜在开销。卷还有其他好处，例如，允许您在容器之间共享数据，并且即使没有正在运行的容器正在使用它们也可以持久存储。

### ### 使用VFS存储驱动程序

VFS存储驱动程序不是联合文件系统。相反，每一层都是磁盘上的目录，并且不支持写时复制。要创建新层，需要对上一层进行“深层复制”。与其他存储驱动程序相比，这导致较低的性能和更多的磁盘空间使用。但是，它是健壮，稳定的，并且可以在每种环境下工作。在测试环境中，它还可以用作验证其他存储后端所依据的机制。

#### 使用vfs存储驱动程序配置Docker

1 停止Docker。

$ sudo systemctl stop docker

2 编辑/etc/docker/daemon.json。如果尚不存在，请创建它。假设文件为空，请添加以下内容。

{

"storage-driver": "vfs"

}

如果要设置配额以控制VFS存储驱动程序可以使用的最大大小size，请在storage-opts密钥上设置选项。仅在Docker 17.12及更高版本中支持配额。

{

"storage-driver": "vfs",

"storage-opts": ["size=256M"]

}

如果daemon.json文件包含格式错误的JSON，则Docker无法启动。

3 启动Docker。

$ sudo systemctl start docker

4 验证守护程序正在使用vfs存储驱动程序。使用docker info命令并查找Storage Driver。

$ docker info

Storage Driver: vfs

...

Docker现在正在使用vfs存储驱动程序。Docker自动创建了/var/lib/docker/vfs/目录，其中包含运行容器所使用的所有层。

#### vfs存储驱动程序如何工作

VFS不是联合文件系统。相反，每个镜像层和可写容器层在Docker主机上都表示为中的子目录 /var/lib/docker/。联合安装提供了所有层的统一视图。目录名称并不直接对应于图层本身的ID。

VFS不支持写时复制（COW），因此每次创建新层时，它都是其父层的深层副本。这些图层都位于下 /var/lib/docker/vfs/dir/。

##### 示例：镜像和容器磁盘构造

以下docker pull命令显示了一个Docker主机下载一个包含五层的Docker镜像。

$ docker pull ubuntu

Using default tag: latest

latest: Pulling from library/ubuntu

e0a742c2abfd: Pull complete

486cb8339a27: Pull complete

dc6f0d824617: Pull complete

4f7a5649a30e: Pull complete

672363445ad2: Pull complete

Digest: sha256:84c334414e2bfdcae99509a6add166bbb4fa4041dc3fa6af08046a66fed3005f

Status: Downloaded newer image for ubuntu:latest

拉动后，这些层中的每一个都表示为的子目录 /var/lib/docker/vfs/dir/。目录名称与docker pull命令中显示的镜像层ID不相关。要查看每个层在磁盘上占用的大小，可以使用du -sh命令，该命令将大小作为人类可读的值给出。

$ ls -l /var/lib/docker/vfs/dir/

total 0

drwxr-xr-x. 2 root root 19 Aug 2 18:19 3262dfbe53dac3e1ab7dcc8ad5d8c4d586a11d2ac3c4234892e34bff7f6b821e

drwxr-xr-x. 21 root root 224 Aug 2 18:23 6af21814449345f55d88c403e66564faad965d6afa84b294ae6e740c9ded2561

drwxr-xr-x. 21 root root 224 Aug 2 18:23 6d3be4585ba32f9f5cbff0110e8d07aea5f5b9fbb1439677c27e7dfee263171c

drwxr-xr-x. 21 root root 224 Aug 2 18:23 9ecd2d88ca177413ab89f987e1507325285a7418fc76d0dcb4bc021447ba2bab

drwxr-xr-x. 21 root root 224 Aug 2 18:23 a292ac6341a65bf3a5da7b7c251e19de1294bd2ec32828de621d41c7ad31f895

drwxr-xr-x. 21 root root 224 Aug 2 18:23 e92be7a4a4e3ccbb7dd87695bca1a0ea373d4f673f455491b1342b33ed91446b

$ du -sh /var/lib/docker/vfs/dir/\*

4.0K /var/lib/docker/vfs/dir/3262dfbe53dac3e1ab7dcc8ad5d8c4d586a11d2ac3c4234892e34bff7f6b821e

125M /var/lib/docker/vfs/dir/6af21814449345f55d88c403e66564faad965d6afa84b294ae6e740c9ded2561

104M /var/lib/docker/vfs/dir/6d3be4585ba32f9f5cbff0110e8d07aea5f5b9fbb1439677c27e7dfee263171c

125M /var/lib/docker/vfs/dir/9ecd2d88ca177413ab89f987e1507325285a7418fc76d0dcb4bc021447ba2bab

104M /var/lib/docker/vfs/dir/a292ac6341a65bf3a5da7b7c251e19de1294bd2ec32828de621d41c7ad31f895

104M /var/lib/docker/vfs/dir/e92be7a4a4e3ccbb7dd87695bca1a0ea373d4f673f455491b1342b33ed91446b

上面的输出显示三层各占104M，两层占125M。这些目录彼此之间只有很小的差异，但是占用的磁盘空间几乎相同。这是使用vfs存储驱动程序的缺点之一 。

# ＃　SDK　API

## ## 概述

Docker提供了一个用于与Docker守护进程进行交互的API（称为Docker Engine API）以及Go和Python的SDK。使用SDK，您可以快速轻松地构建和扩展Docker应用程序和解决方案。如果Go或Python不适用于您，则可以直接使用Docker Engine API。

Docker Engine API是一种RESTful API，可通过HTTP客户端（例如wget或 curl）或HTTP库（大多数现代编程语言的一部分）进行访问。

## ## 起步-官方

### 非官方库

有许多社区支持的库可用于其他语言。它们尚未经过Docker的测试，因此，如果遇到任何问题，请向库维护者归档。

|  |  |
| --- | --- |
| 爪哇 | [码头客户](https://github.com/spotify/docker-client) |
| 爪哇 | [码头工人](https://github.com/docker-java/docker-java) |
| 爪哇 | [docker-java-api](https://github.com/amihaiemil/docker-java-api) |

### 安装SDK

使用以下命令安装Go或Python SDK。这两个SDK可以安装并共存。

#### go到SDK

go get github.com/docker/docker/client

客户端需要Go的最新版本。运行go version并确保您至少运行Go的1.9.4版本

#### Python SDK

推荐：跑pip install docker。

#### 版本化的API和SDK

您应使用的Docker Engine API的版本取决于Docker守护程序和Docker客户端的版本。

给定版本的Docker Engine SDK支持特定版本的Docker Engine API，以及所有早期版本。如果发生重大变化，则会在显眼位置进行记录。

#### 守护程序和客户端API不匹配

Docker守护程序和客户端不必始终保持相同版本。但是，请记住以下几点。

1 如果守护程序比客户端新，则客户端不知道守护程序中的新功能或API端点已弃用。

2 如果客户端比守护程序新，则客户端可以请求守护程序不知道的API端点。

添加新功能后，将发布API的新版本。Docker API是向后兼容的，因此除非需要利用新功能，否则您不需要更新使用该API的代码。

#### 您可以通过以下方式之一指定要使用的API版本：

使用SDK时，请使用可以使用的最新版本，但至少要使用将API版本与所需功能结合在一起的版本。

curl直接使用时，将版本指定为URL的第一部分。例如，如果端点为/containers/，则可以使用 /v1.27/containers/。

要强制Docker CLI或Docker Engine SDK使用比API报告的版本更旧的API版本docker version，请将环境变量DOCKER\_API\_VERSION设置为正确的版本。这适用于Linux，Windows或macOS客户端。

DOCKER\_API\_VERSION='1.27'

设置环境变量后，即使Docker守护程序支持较新的版本，也会使用该版本的API。

对于SDK，您还可以通过编程方式指定API版本，作为client对象的参数。

请参阅 [Go构造函数](https://github.com/moby/moby/blob/master/client/client.go#L136) 或的 [Python SDK文档client](https://docker-py.readthedocs.io/en/stable/client.html)。

### SDK和API快速入门

#### 极简示例

例如，docker run可以直接使用Docker API或使用Python或Go SDK轻松实现该命令。

Python

import docker

client = docker.from\_env()

print client.containers.run("alpine", ["echo", "hello", "world"])

#### 入门示例

##### 运行一个容器

第一个示例显示了如何使用Docker API运行容器。在命令行上，您将使用该docker run命令，但是从您自己的应用程序执行此操作也很容易。

这等效docker run alpine echo hello world于在命令提示符下键入：

Python API

import docker

client = docker.from\_env()

print client.containers.run("alpine", ["echo", "hello", "world"])

GO

package main

import (

"os"

"io"

"github.com/docker/docker/api/types"

"github.com/docker/docker/api/types/container"

"github.com/docker/docker/pkg/stdcopy"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

reader, err := cli.ImagePull(ctx, "docker.io/library/alpine", types.ImagePullOptions{})

if err != nil {

panic(err)

}

io.Copy(os.Stdout, reader)

resp, err := cli.ContainerCreate(ctx, &container.Config{

Image: "alpine",

Cmd: []string{"echo", "hello world"},

Tty: true,

}, nil, nil, "")

if err != nil {

panic(err)

}

if err := cli.ContainerStart(ctx, resp.ID, types.ContainerStartOptions{}); err != nil {

panic(err)

}

statusCh, errCh := cli.ContainerWait(ctx, resp.ID, container.WaitConditionNotRunning)

select {

case err := <-errCh:

if err != nil {

panic(err)

}

case <-statusCh:

}

out, err := cli.ContainerLogs(ctx, resp.ID, types.ContainerLogsOptions{ShowStdout: true})

if err != nil {

panic(err)

}

stdcopy.StdCopy(os.Stdout, os.Stderr, out)

}

HTTP

$ curl --unix-socket /var/run/docker.sock -H "Content-Type: application/json" \

-d '{"Image": "alpine", "Cmd": ["echo", "hello world"]}' \

-X POST http:/v1.24/containers/create

{"Id":"1c6594faf5","Warnings":null}

$ curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/1c6594faf5/start

$ curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/1c6594faf5/wait

{"StatusCode":0}

$ curl --unix-socket /var/run/docker.sock "http:/v1.24/containers/1c6594faf5/logs?stdout=1"

hello world

##### 在后台运行容器

您也可以在后台运行容器，相当于键入 docker run -d bfirsh/reticulate-splines：

Python API

import docker

client = docker.from\_env()

container = client.containers.get('f1064a8a4c82')

print container.logs()

GO

package main

import (

"fmt"

"io"

"os"

"github.com/docker/docker/api/types"

"github.com/docker/docker/api/types/container"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

imageName := "bfirsh/reticulate-splines"

out, err := cli.ImagePull(ctx, imageName, types.ImagePullOptions{})

if err != nil {

panic(err)

}

io.Copy(os.Stdout, out)

resp, err := cli.ContainerCreate(ctx, &container.Config{

Image: imageName,

}, nil, nil, "")

if err != nil {

panic(err)

}

if err := cli.ContainerStart(ctx, resp.ID, types.ContainerStartOptions{}); err != nil {

panic(err)

}

fmt.Println(resp.ID)

}

HTTP

$ curl --unix-socket /var/run/docker.sock http:/v1.24/containers/json

[{

"Id":"ae63e8b89a26f01f6b4b2c9a7817c31a1b6196acf560f66586fbc8809ffcd772",

"Names":["/tender\_wing"],

"Image":"bfirsh/reticulate-splines",

...

}]

$ curl --unix-socket /var/run/docker.sock \

-X POST http:/v1.24/containers/ae63e8b89a26/stop

##### 打印特定容器的日志

您还可以对单个容器执行操作。此示例在给定其ID的情况下打印容器的日志。您需要在运行代码之前对其进行修改，以更改要打印其日志的容器的硬编码ID。

Python API

import docker

client = docker.from\_env()

container = client.containers.get('f1064a8a4c82')

print container.logs()

GO

package main

import (

"fmt"

"github.com/docker/docker/api/types"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

containers, err := cli.ContainerList(ctx, types.ContainerListOptions{})

if err != nil {

panic(err)

}

for \_, container := range containers {

fmt.Print("Stopping container ", container.ID[:10], "... ")

if err := cli.ContainerStop(ctx, container.ID, nil); err != nil {

panic(err)

}

fmt.Println("Success")

}

}

HTTP

$ curl --unix-socket /var/run/docker.sock "http:/v1.24/containers/ca5f55cdb/logs?stdout=1"

Reticulating spline 1...

Reticulating spline 2...

Reticulating spline 3...

Reticulating spline 4...

Reticulating spline 5...

#### 列出所有镜像

列出您引擎上的图像，类似于docker image ls：

Python API

import docker

client = docker.from\_env()

for image in client.images.list():

print image.id

GO

package main

import (

"io"

"os"

"github.com/docker/docker/api/types"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

options := types.ContainerLogsOptions{ShowStdout: true}

// Replace this ID with a container that really exists

out, err := cli.ContainerLogs(ctx, "f1064a8a4c82", options)

if err != nil {

panic(err)

}

io.Copy(os.Stdout, out)

}

HTTP

$ curl --unix-socket /var/run/docker.sock http:/v1.24/images/json

[{

"Id":"sha256:31d9a31e1dd803470c5a151b8919ef1988ac3efd44281ac59d43ad623f275dcd",

"ParentId":"sha256:ee4603260daafe1a8c2f3b78fd760922918ab2441cbb2853ed5c439e59c52f96",

...

}]

#### 拉图像

拉图像，例如docker pull：

Python API

import docker

client = docker.from\_env()

image = client.images.pull("alpine")

print image.id

GO

package main

import (

"fmt"

"github.com/docker/docker/api/types"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

images, err := cli.ImageList(context.Background(), types.ImageListOptions{})

if err != nil {

panic(err)

}

for \_, image := range images {

fmt.Println(image.ID)

}

}

HTTP

$ curl --unix-socket /var/run/docker.sock \

-X POST "http:/v1.24/images/create?fromImage=alpine"

{"status":"Pulling from library/alpine","id":"3.1"}

{"status":"Pulling fs layer","progressDetail":{},"id":"8f13703509f7"}

{"status":"Downloading","progressDetail":{"current":32768,"total":2244027},"progress":"[\u003e ] 32.77 kB/2.244 MB","id":"8f13703509f7"}

...

#### 通过身份验证提取图像

docker pull通过身份验证来提取图像，例如：

注意：凭据以明文形式发送。Docker的官方注册表使用HTTPS。私有注册表也应配置为使用HTTPS。

Python API

Python SDK从[凭证存储](https://docs.docker.com/engine/reference/commandline/login/#credentials-store)文件中检索身份验证信息，并与[凭证帮助](https://github.com/docker/docker-credential-helpers)程序集成。可以覆盖这些凭据，但这不在本入门指南的范围之内。使用后docker login，Python SDK会自动使用这些凭据。

import docker

client = docker.from\_env()

image = client.images.pull("alpine")

print image.id

GO

package main

import (

"encoding/base64"

"encoding/json"

"io"

"os"

"github.com/docker/docker/api/types"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

authConfig := types.AuthConfig{

Username: "username",

Password: "password",

}

encodedJSON, err := json.Marshal(authConfig)

if err != nil {

panic(err)

}

authStr := base64.URLEncoding.EncodeToString(encodedJSON)

out, err := cli.ImagePull(ctx, "alpine", types.ImagePullOptions{RegistryAuth: authStr})

if err != nil {

panic(err)

}

defer out.Close()

io.Copy(os.Stdout, out)

}

HTTP

package main

import (

"encoding/base64"

"encoding/json"

"io"

"os"

"github.com/docker/docker/api/types"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

authConfig := types.AuthConfig{

Username: "username",

Password: "password",

}

encodedJSON, err := json.Marshal(authConfig)

if err != nil {

panic(err)

}

authStr := base64.URLEncoding.EncodeToString(encodedJSON)

out, err := cli.ImagePull(ctx, "alpine", types.ImagePullOptions{RegistryAuth: authStr})

if err != nil {

panic(err)

}

defer out.Close()

io.Copy(os.Stdout, out)

}

#### 提交容器

提交容器以根据其内容创建图像：

Python API

import docker

client = docker.from\_env()

container = client.containers.run("alpine", ["touch", "/helloworld"], detach=True)

container.wait()

image = container.commit("helloworld")

print image.id

GO

package main

import (

"fmt"

"github.com/docker/docker/api/types"

"github.com/docker/docker/api/types/container"

"github.com/docker/docker/client"

"golang.org/x/net/context"

)

func main() {

ctx := context.Background()

cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())

if err != nil {

panic(err)

}

createResp, err := cli.ContainerCreate(ctx, &container.Config{

Image: "alpine",

Cmd: []string{"touch", "/helloworld"},

}, nil, nil, "")

if err != nil {

panic(err)

}

if err := cli.ContainerStart(ctx, createResp.ID, types.ContainerStartOptions{}); err != nil {

panic(err)

}

statusCh, errCh := cli.ContainerWait(ctx, createResp.ID, container.WaitConditionNotRunning)

select {

case err := <-errCh:

if err != nil {

panic(err)

}

case <-statusCh:

}

commitResp, err := cli.ContainerCommit(ctx, createResp.ID, types.ContainerCommitOptions{Reference: "helloworld"})

if err != nil {

panic(err)

}

fmt.Println(commitResp.ID)

}

HTTP

$ docker run -d alpine touch /helloworld

0888269a9d584f0fa8fc96b3c0d8d57969ceea3a64acf47cd34eebb4744dbc52

$ curl --unix-socket /var/run/docker.sock\

-X POST "http:/v1.24/commit?container=0888269a9d&repo=helloworld"

{"Id":"sha256:6c86a5cd4b87f2771648ce619e319f3e508394b5bfc2cdbd2d60f59d52acda6c"}

Python API

GO

HTTP

## ## java spotify/docker-client

Github地址： https://github.com/spotify/docker-client

### 起步

#### 引入依赖

<dependency>

<groupId>com.spotify</groupId>

<artifactId>docker-client</artifactId>

<version>LATEST-VERSION</version>

</dependency>

#### Usage Example

// Create a client based on DOCKER\_HOST and DOCKER\_CERT\_PATH env vars

final DockerClient docker = DefaultDockerClient.fromEnv().build();

// Pull an image

docker.pull("busybox");

// Bind container ports to host ports

final String[] ports = {"80", "22"};

final Map<String, List<PortBinding>> portBindings = new HashMap<>();

for (String port : ports) {

List<PortBinding> hostPorts = new ArrayList<>();

hostPorts.add(PortBinding.of("0.0.0.0", port));

portBindings.put(port, hostPorts);

}

// Bind container port 443 to an automatically allocated available host port.

List<PortBinding> randomPort = new ArrayList<>();

randomPort.add(PortBinding.randomPort("0.0.0.0"));

portBindings.put("443", randomPort);

final HostConfig hostConfig = HostConfig.builder().portBindings(portBindings).build();

// Create container with exposed ports

final ContainerConfig containerConfig = ContainerConfig.builder()

.hostConfig(hostConfig)

.image("busybox").exposedPorts(ports)

.cmd("sh", "-c", "while :; do sleep 1; done")

.build();

final ContainerCreation creation = docker.createContainer(containerConfig);

final String id = creation.id();

// Inspect container

final ContainerInfo info = docker.inspectContainer(id);

// Start container

docker.startContainer(id);

// Exec command inside running container with attached STDOUT and STDERR

final String[] command = {"sh", "-c", "ls"};

final ExecCreation execCreation = docker.execCreate(

id, command, DockerClient.ExecCreateParam.attachStdout(),

DockerClient.ExecCreateParam.attachStderr());

final LogStream output = docker.execStart(execCreation.id());

final String execOutput = output.readFully();

// Kill container

docker.killContainer(id);

// Remove container

docker.removeContainer(id);

// Close the docker client

docker.close();

更多请移步github参看使用文档。

# 问题：

## unix:///var/run/docker.sock

docker 客户端通过unix:///var/run/docker.sock与docker daemon通信，unix:///var/run/docker.sock需要管理员权限才能访问，所以要么运行sudo docker run hello-world，要么将当前用户添加在docker用户组中  
sudo usermod -aG docker $USER