# Docker入门到精通

Docker 是一个开源项目，诞生于 2013 年初，最初是 dotCloud 公司内部的一个业余项目。它基于 Google 公司推出的 Go 语言实现。 项目后来加入了 Linux 基金会，遵从了 Apache 2.0 协议，项目代码在 [GitHub](https://github.com/docker/docker" \t "http://www.dockerinfo.net/_blank) 上进行维护。Docker 自开源后受到广泛的关注和讨论，以至于 dotCloud 公司后来都改名为 Docker Inc。Redhat 已经在其 RHEL6.5 中集中支持 Docker；Google 也在其 PaaS 产品中广泛应用。Docker 项目的目标是实现轻量级的操作系统虚拟化解决方案。 Docker 的基础是 Linux 容器（LXC）等技术。在 LXC 的基础上 Docker 进行了进一步的封装，让用户不需要去关心容器的管理，使得操作更为简便。用户操作 Docker 的容器就像操作一个快速轻量级的虚拟机一样简单。Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低。

## 为什么要使用Docker

### Docker容器虚拟化的好处

在云时代，开发者创建的应用必须要能很方便地在网络上传播，也就是说应用必须脱离底层物理硬件的限制；同时必须满足“任何时间任何地点”可获取可使用的特点。因此，开发者们需要一种新型的创建分布式应用程序的方式，快速分发部署，而这正是Docker所能够提供的最大优势。Docker提供了一种更为聪明的方式，通过容器来打包应用、解耦应用和运行平台。这意味着迁移的时候，只需要在新的服务器上启动需要的容器就可以了，无论新旧服务器是否是同一类别的平台。这无疑帮助我们节约了大量的宝贵时间，并降低部署过程出现问题的风险。

### Docker在开发和运维中的优势

对于开发和运维人员来说，最梦寐以求的效果可能就是一次创建和配置，之后可以在任意地方、任意时间让应用正常运行，而Docker恰恰可以实现这一中级目标。具体来说，在开发和运维过程中，Docker具有以下几个方面的优势：

* **更快的交付和部署：**使用Docker，开发人员可以使用镜像来快速构建一套标准的开发环境；开发完之后，测试和运维人员可以直接使用完全相同的环境来部署代码。只要是开发测试过的代码，就可以确保在生产环境无缝运行。Docker可以快速创建和删除容器，实现快速迭代，节约开发、测试及部署的时间。
* **更高效的利用资源：**运行Docker容器不需要额外的虚拟化管理程序的支持，Docker是内核级的虚拟化，可以实现更高的性能，同时对资源的额外需求很低，与传统的虚拟机方式相比，Docker的性能要提高1 ~ 2个数量级。
* **更轻松的迁移和扩展：**Docker容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑等等，同时支持主流的操作系统发行版本。这种兼容性能让用户可以在不同的平台之间轻松的迁移应用。
* **更轻松的管理和更新：**使用Dockerfile，只需要小小的配置修改，就可以替代以往大量的更新工作。所有的修改都以增量的方式被分发和更新，从而实现自动化并且高效的容器管理。

### Docker与虚拟机的比较

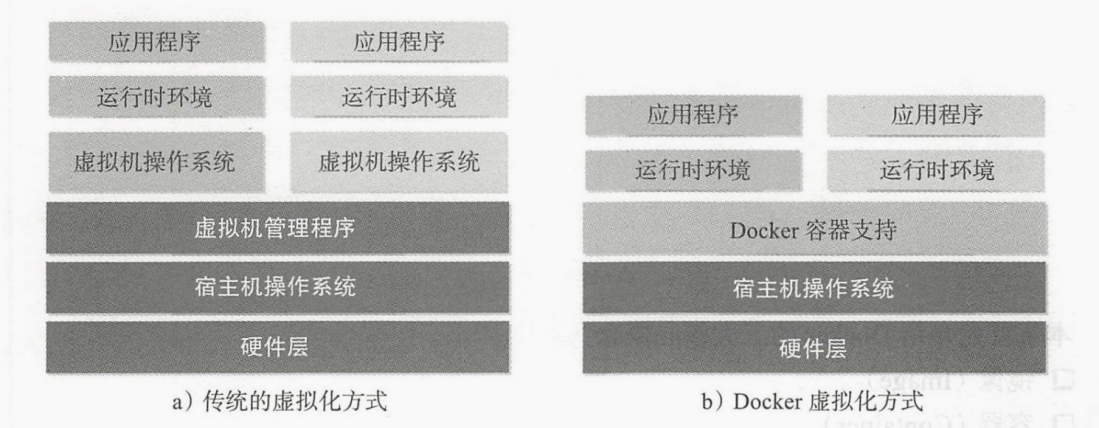
作为一种轻量级的虚拟化方式，Docker在运行应用上跟传统的虚拟机的方式相比具有如下显著优势：

* Docker容器启动很快，启动和停止可以实现秒级，相比传统的虚拟机方式（分钟级）要快速很多。
* Docker容器对系统资源需求很少，一台主机上可以同时运行数千个Docker容器。
* Docker通过类似git设计理念的操作来方便用户获取、分发和更新应用镜像，存储复用，增量更新。
* Docker通过Dockerfile支持灵活的自动化创建和部署机制，可以提高工作效率，并标准化流程。

|  |  |  |
| --- | --- | --- |
| **特性** | **容器** | **虚拟机** |
| 启动速度 | 秒级 | 分钟级 |
| 性能 | 接近原生 | 较好 |
| 内存 | MB级 | GB级 |
| 硬盘适应 | MB级 | GB级 |
| 运行密度 | 单台主机支持上千个 | 单台主机支持几个 |
| 隔离性 | 安全隔离 | 完全隔离 |
| 迁移 | 优秀 | 一般 |

## Docker与虚拟化

Docker以及其他容器技术，都属于操作系统虚拟化范畴，操作系统细腻化最大的特点就是不需要额外的supervisor支持。Docker虚拟化方式之所以有众多优势，跟操作系统虚拟化技术自身的设计和实现分不开。



传统方式是在硬件层面实现虚拟化，需要有额外的虚拟机管理应用和虚拟机操作系统层。Docker容器时在操作系统层面实现虚拟化，直接复用本地主机的操作系统，因此更加轻量级。

## Docker概念和使用

Docker中有三个核心概念：镜像、容器和仓库。因此，准确把握这三大概念对于掌握Docker技术尤为重要。

1. 镜像（Image）

Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统。

1. 容器（Container）

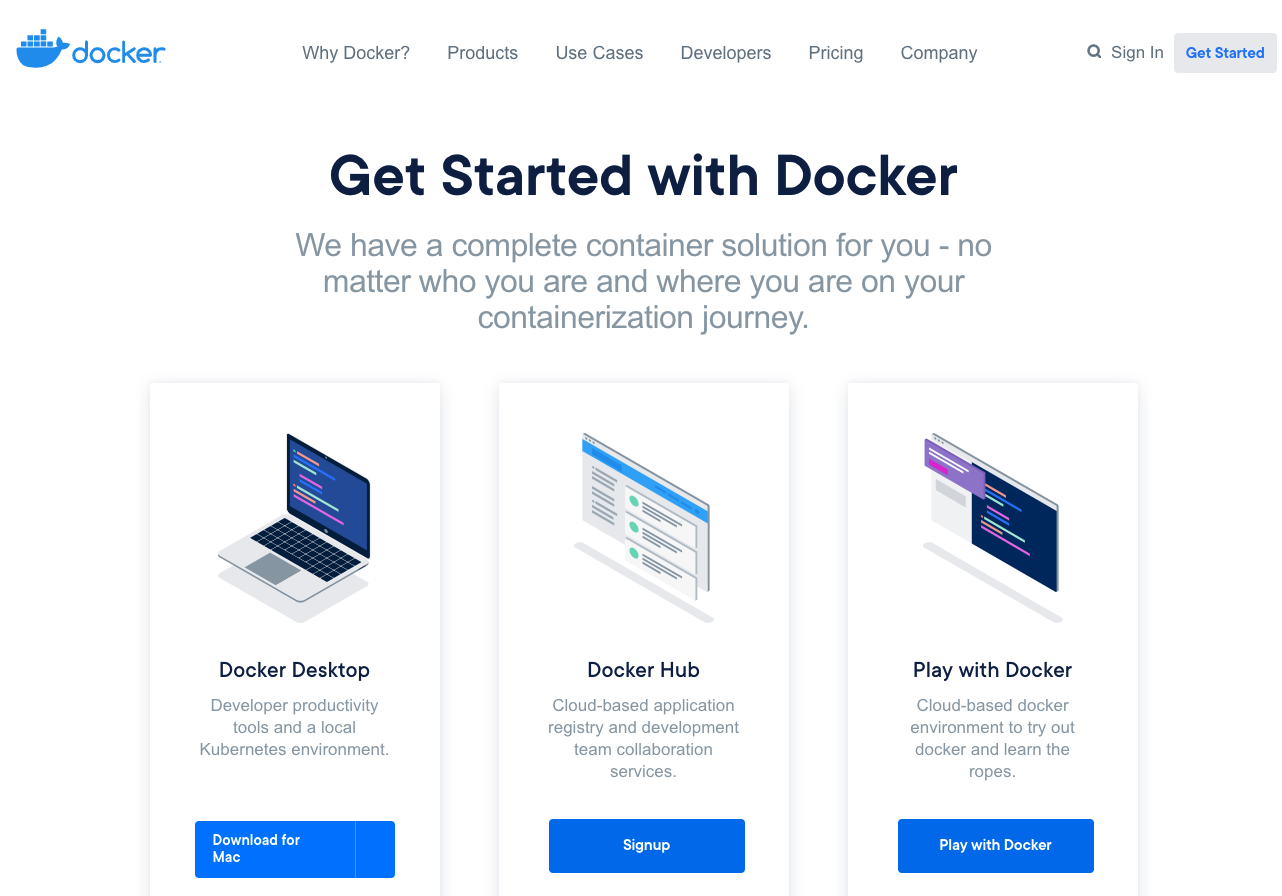
镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

1. 仓库（Repository）

用来保存镜像的仓库。当我们构建好自己的镜像之后，需要存放在仓库中，当我们需要启动一个镜像时，可 以在仓库中下载下来。

### Docker安装

Docker引擎是使用Docker容器的核心组件，可以在主流的操作系统和云平台上使用，包括Linux操作系统、MacOSS和Windows系统上。用户可以访问Docker官网（https://www.docker.com/get-started）去获取Docker安装包。



#### 在CentOS系统上安装Docker

Docker目前支持CentOS 7及以后版本。系统的要求跟Ubuntu情况类似，64位操作系统，内核至少是3.10以上。

##### 更换系统yum源

# 查看系统内核版本

[root@alvin-test-os ~]# uname -a

Linux alvin-test-os 3.10.0-1127.el7.x86\_64 #1 SMP Tue Mar 31 23:36:51 UTC 2020 x86\_64 x86\_64 x86\_64 GNU/Linux

# 备份原来YUN源

cp /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.bak

# 更换YUM源

curl -o /etc/yum.repos.d/CentOS-Base.repo https://mirrors.aliyun.com/repo/Centos-7.repo

# 刷新yum源缓存

yum makecache

##### 首先，为了方便我们需要更新一下系统：

[root@alvin-test-os ~]# yum update -y --exclud=kernel\*

已加载插件：fastestmirror

Loading mirror speeds from cached hostfile

\* base: mirrors.aliyun.com

\* extras: mirrors.aliyun.com

\* updates: mirrors.aliyun.com

...

##### 安装所需的基础软件

[root@alvin-test-os yum.repos.d]# yum install -y yum-utils device-mapper-persistent-data lvm2

##### 安装yum源

[root@alvin-test-os yum.repos.d]# yum-config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

已加载插件：fastestmirror

adding repo from: https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

grabbing file https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo to /etc/yum.repos.d/docker-ce.repo

repo saved to /etc/yum.repos.d/docker-ce.repo

1.5、 更新并安装Docker-CE

[root@alvin-test-os yum.repos.d]# yum makecache fast

已加载插件：fastestmirror

Loading mirror speeds from cached hostfile

\* base: mirrors.aliyun.com

\* extras: mirrors.aliyun.com

\* updates: mirrors.aliyun.com

base | 3.6 kB 00:00:00

docker-ce-stable | 3.5 kB 00:00:00

extras | 2.9 kB 00:00:00

updates | 2.9 kB 00:00:00

元数据缓存已建立

[root@alvin-test-os ~]# yum -y install docker-ce

已加载插件：fastestmirror

Loading mirror speeds from cached hostfile

\* base: mirrors.aliyun.com

\* extras: mirrors.aliyun.com

\* updates: mirrors.aliyun.com

##### 启动并设置开机自启动

[root@alvin-test-os ~]# systemctl enable --now docker

##### 测试启动

[root@alvin-test-os ~]# docker info

Client:

Debug Mode: false

Server:

Containers: 1

Running: 1

Paused: 0

Stopped: 0

Images: 1

Server Version: 19.03.13

Storage Driver: overlay2

Backing Filesystem: xfs

Supports d\_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: cgroupfs

Plugins:

Volume: local

Network: bridge host ipvlan macvlan null overlay

Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: 8fba4e9a7d01810a393d5d25a3621dc101981175

runc version: dc9208a3303feef5b3839f4323d9beb36df0a9dd

init version: fec3683

Security Options:

seccomp

Profile: default

Kernel Version: 3.10.0-1127.el7.x86\_64

Operating System: CentOS Linux 7 (Core)

OSType: linux

Architecture: x86\_64

CPUs: 8

Total Memory: 15.67GiB

Name: alvin-test-os

ID: HR6P:IIZP:6CKH:N2L5:IQ2T:AJFP:CCAT:S2ZG:MWC4:DZBQ:XUZM:UKLC

Docker Root Dir: /var/lib/docker

Debug Mode: false

Registry: https://index.docker.io/v1/

Labels:

Experimental: false

Insecure Registries:

127.0.0.0/8

Live Restore Enabled: false

#### 在Ubuntu环境上安装Docker

Ubuntu操作系统对Docker的支持十分成熟，可以支持包括x86\_64、armhf等系统架构，只要是64位即可。

Docker目前支持最低Ubuntu版本为14.04LTS，但实际上从稳定性上考虑，推荐使用16.04LTS 或18.04LTS版本，并且系统内核越新越好，以支持Docker最新的特性。

##### 查看版本信息

# 第一种方式

root@alvin-test-os:~# uname -a

Linux alvin-test-os 4.15.0-117-generic #118-Ubuntu SMP Fri Sep 4 20:02:41 UTC 2020 x86\_64 x86\_64 x86\_64 GNU/Linux

# 第二种方式

root@alvin-test-os:~# cat /proc/version

Linux version 4.15.0-117-generic (buildd@lcy01-amd64-008) (gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)) #118-Ubuntu SMP Fri Sep 4 20:02:41 UTC 2020

##### 更新系统

root@alvin-test-os:~# apt-get update

Get:1 http://mirrors.cloud.aliyuncs.com/ubuntu bionic InRelease [242 kB]

Get:2 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-updates InRelease [88.7 kB]

Get:3 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-security InRelease [88.7 kB]

Get:4 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/universe Sources [9,051 kB]

Get:5 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/main Sources [829 kB]

Get:6 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/main amd64 Packages [1,019 kB]

Get:7 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/main i386 Packages [1,007 kB]

Get:8 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/main Translation-en [516 kB]

Get:9 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/universe amd64 Packages [8,570 kB]

Get:10 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/universe i386 Packages [8,531 kB]

Get:11 http://mirrors.cloud.aliyuncs.com/ubuntu bionic/universe Translation-en [4,941 kB]

Get:12 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-updates/main Sources [497 kB]

... 此处略去许多行

##### 安装基础软件

root@alvin-test-os:~# apt-get -y install apt-transport-https ca-certificates curl software-properties-common

Reading package lists... Done

Building dependency tree

Reading state information... Done

curl is already the newest version (7.58.0-2ubuntu3.10).

The following additional packages will be installed:

python3-software-properties unattended-upgrades

Suggested packages:

。。。此处略去很多行

##### 安装GPG证书

root@alvin-test-os:~# curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -

OK

##### 写入软件源信息

root@alvin-test-os:~# sudo add-apt-repository "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb\_release -cs) stable"

Hit:1 http://mirrors.cloud.aliyuncs.com/ubuntu bionic InRelease

Hit:2 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-updates InRelease

Hit:3 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-security InRelease

Get:4 https://mirrors.aliyun.com/docker-ce/linux/ubuntu bionic InRelease [64.4 kB]

Get:5 https://mirrors.aliyun.com/docker-ce/linux/ubuntu bionic/stable amd64 Packages [13.0 kB]

Fetched 77.4 kB in 0s (307 kB/s)

Reading package lists... Done

##### 更新并安装Docker-CE

root@alvin-test-os:~# sudo apt-get -y update

Hit:1 http://mirrors.cloud.aliyuncs.com/ubuntu bionic InRelease

Hit:2 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-updates InRelease

Hit:3 http://mirrors.cloud.aliyuncs.com/ubuntu bionic-security InRelease

Hit:4 https://mirrors.aliyun.com/docker-ce/linux/ubuntu bionic InRelease

Reading package lists... Done

root@alvin-test-os:~# sudo apt-get -y install docker-ce

Reading package lists... Done

Building dependency tree

Reading state information... Done

The following additional packages will be installed:

aufs-tools cgroupfs-mount containerd.io docker-ce-cli git git-man libcurl3-gnutls liberror-perl pigz

Suggested packages:

git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn

The following NEW packages will be installed:

aufs-tools cgroupfs-mount containerd.io docker-ce docker-ce-cli git git-man libcurl3-gnutls liberror-perl pigz

0 upgraded, 10 newly installed, 0 to remove and 86 not upgraded.

##### 启动并检验

root@alvin-test-os:~# docker version

Client: Docker Engine - Community

Version: 19.03.13

API version: 1.40

Go version: go1.13.15

Git commit: 4484c46d9d

Built: Wed Sep 16 17:02:36 2020

OS/Arch: linux/amd64

Experimental: false

Server: Docker Engine - Community

Engine:

Version: 19.03.13

API version: 1.40 (minimum version 1.12)

Go version: go1.13.15

Git commit: 4484c46d9d

Built: Wed Sep 16 17:01:06 2020

OS/Arch: linux/amd64

Experimental: false

containerd:

Version: 1.3.7

GitCommit: 8fba4e9a7d01810a393d5d25a3621dc101981175

runc:

Version: 1.0.0-rc10

GitCommit: dc9208a3303feef5b3839f4323d9beb36df0a9dd

docker-init:

Version: 0.18.0

#### 第一个Docker实例

root@alvin-test-os:~# docker run -d --rm --name nginx -p 80:80 nginx

Unable to find image 'nginx:latest' locally

latest: Pulling from library/nginx

bb79b6b2107f: Pull complete

5a9f1c0027a7: Downloading [==============> ] 7.457MB/26.49MB

5a9f1c0027a7: Downloading [======================> ] 12.17MB/26.49MB

166a2418f7e8: Download complete

1966ea362d23: Download complete

### 使用Docker镜像

docker镜像是docker中三大概念之一，其主要作用是作为启动容器的模板。

#### 获取镜像

镜像是运行容器的模板，官方Docker Hub 仓库已经提供了许多镜像共开发者使用。如果我们需要获取某个镜像则可以去docker仓库下载所需的镜像。

下载镜像的格式：docker pull [仓库的URL]/[名称空间]/[仓库的名称]:[版本号]

下载镜像实例:

[root@alvin-test-os ~]# docker pull nginx:1.17

1.17: Pulling from library/nginx

afb6ec6fdc1c: Pull complete

b90c53a0b692: Pull complete

11fa52a0fdc0: Pull complete

Digest: sha256:6fff55753e3b34e36e24e37039ee9eae1fe38a6420d8ae16ef37c92d1eb26699

Status: Downloaded newer image for nginx:1.17

docker.io/library/nginx:1.17

下载的时候，我们可以看到有若干层组成，像afb6ec6fdc1c这样的字符串是层的唯一ID（实际上，完整的ID包括256比特， 64个十六进制字符组成）。使用docker pull 命令下载中会获取并输出镜像的各层信息。当不同的镜像包括相同的层的时候，本地仅存一份内容，减小存储空间。

#### 查看镜像信息

镜像主要包括镜像文件、镜像tag以及镜像详细信息等等。

##### 镜像列表

使用docker images 或 docker images ls命令可以列举本地主机上已有镜像的基本信息。

[root@alvin-test-os ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

registry.cn-hangzhou.aliyuncs.com/alvinos/teach 1000-teach-2020-10-28-13-40-27 22fdec3d9a6d 3 weeks ago 1.02GB

nginx 1.17 9beeba249f3e 6 months ago 127MB

列举信息中，可以看到几个字段信息：

* 镜像来源：来自哪个仓库，默认来自：hub.docker.com
* 镜像标签：比喻1.17、1000-teach-2020-10-28-13-40-27
* 镜像ID：例如22fdec3d9a6d
* 镜像创建时间：例如：3 weeks ago
* 镜像大小：127MB

其中镜像的ID信息十分重要，它唯一标识了镜像。在使用镜像ID的时候，一般可以使用该ID的前若干个字符组成的可区分串来替代完整的ID。

TAG信息用于标记来自同一个仓库的不同镜像。TAG在同一个仓库中是唯一的。

镜像大小信息只是表示了该镜像的逻辑体积大小，实际上由于相同的镜像层本地只会存储一份，物理上占用的存储空间会小于各镜像逻辑体积之和。

image子命令主要支持如下选项：

* -a : 列出所有（包括临时文件）镜像文件

alvin@AlvindeMacBook-Pro: docker images -a

REPOSITORY TAG IMAGE ID CREATED SIZE

registry.cn-hangzhou.aliyuncs.com/alvinos/teach 1000-teach-2020-10-22-19-03-16 d83921684085 3 weeks ago 1.05GB

test v1 2047b4806990 3 weeks ago 1.05GB

<none> <none> 85b0590cd234 3 weeks ago 1.05GB

<none> <none> 611b48e08ec2 3 weeks ago 1.05GB

<none> <none> 9fbe4fba95d2 3 weeks ago 1.05GB

<none> <none> 31b4192b2e41 3 weeks ago 1GB

<none> <none> 2c8ba419a512 3 weeks ago 1GB

* --digests=true|false：列出镜像的数字摘要值

[root@alvin-test-os ~]# docker images --digests

REPOSITORY TAG DIGEST IMAGE ID CREATED SIZE

registry.cn-hangzhou.aliyuncs.com/alvinos/teach 1000-teach-2020-10-28-13-40-27 sha256:1dd7db35cc9f3c2f0349806d05f93cbd18f1f583290c5bf2d513f942929bf068 22fdec3d9a6d 3 weeks ago 1.02GB

nginx 1.17 sha256:6fff55753e3b34e36e24e37039ee9eae1fe38a6420d8ae16ef37c92d1eb26699 9beeba249f3e 6 months ago 127MB

* -q : 仅显示ID信息

[root@alvin-test-os ~]# docker images -q

74d107221092

22fdec3d9a6d

9beeba249f3e

##### 为镜像添加tag

为了方便后续工作中使用特定的镜像，还可以使用docker tag命令来为本地的镜像添加标签。

[root@alvin-test-os ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

redis latest 74d107221092 19 hours ago 104MB

[root@alvin-test-os ~]# docker tag redis:latest myredis:latest

[root@alvin-test-os ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myredis latest 74d107221092 19 hours ago 104MB

redis latest 74d107221092 19 hours ago 104MB

##### 使用inspect命令查看详细信息

使用docker inspect 命令获取镜像的详细信息，包括PID、作者、架构等等。

[root@alvin-test-os ~]# docker inspect redis

[

{

"Id": "sha256:74d107221092875724ddb06821416295773bee553bbaf8d888ababe9be7b947f",

"RepoTags": [

"myredis:latest",

"redis:latest"

],

"RepoDigests": [

"redis@sha256:5b98e32b58cdbf9f6b6f77072c4915d5ebec43912114031f37fa5fa25b032489"

],

... 此处略去千百行

}

]

##### 使用history命令查看镜像历史

[root@alvin-test-os ~]# docker history redis

IMAGE CREATED CREATED BY SIZE COMMENT

74d107221092 19 hours ago /bin/sh -c #(nop) CMD ["redis-server"] 0B

<missing> 19 hours ago /bin/sh -c #(nop) EXPOSE 6379 0B

<missing> 19 hours ago /bin/sh -c #(nop) ENTRYPOINT ["docker-entry… 0B

<missing> 19 hours ago /bin/sh -c #(nop) COPY file:df205a0ef6e6df89… 374B

<missing> 19 hours ago /bin/sh -c #(nop) WORKDIR /data 0B

<missing> 19 hours ago /bin/sh -c #(nop) VOLUME [/data] 0B

<missing> 19 hours ago /bin/sh -c mkdir /data && chown redis:redis … 0B

<missing> 19 hours ago /bin/sh -c set -eux; savedAptMark="$(apt-m… 30.5MB

<missing> 19 hours ago /bin/sh -c #(nop) ENV REDIS\_DOWNLOAD\_SHA=dc… 0B

<missing> 19 hours ago /bin/sh -c #(nop) ENV REDIS\_DOWNLOAD\_URL=ht… 0B

<missing> 19 hours ago /bin/sh -c #(nop) ENV REDIS\_VERSION=6.0.9 0B

<missing> 19 hours ago /bin/sh -c set -eux; savedAptMark="$(apt-ma… 4.15MB

<missing> 19 hours ago /bin/sh -c #(nop) ENV GOSU\_VERSION=1.12 0B

<missing> 19 hours ago /bin/sh -c groupadd -r -g 999 redis && usera… 329kB

<missing> 31 hours ago /bin/sh -c #(nop) CMD ["bash"] 0B

<missing> 31 hours ago /bin/sh -c #(nop) ADD file:d2abb0e4e7ac17737… 69.2MB

注意：有些构建信息过长，可以使用--no-trunc选项来输出完整信息。

##### 搜索镜像

在docker中搜索镜像主要使用Search子命令，默认只搜索Docker Hub官方镜像仓库中的镜像。其语法为docker search [option] keyword。支持的命令选项主要包括：

* -f : 过滤输出内容

# 搜索官方提供的带有Redis关键字的镜像

[root@alvin-test-os ~]# docker search -f is-official=true redis

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

redis Redis is an open source key-value store that… 8779 [OK]

# 搜索被收藏超过8000个的并且关键词包括Redis的镜像

[root@alvin-test-os ~]# docker search -f stars=8000 redis

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

redis Redis is an open source key-value store that… 8779 [OK]

* --limit: 限制输出结果

[root@alvin-test-os ~]# docker search -f stars=8 --limit 3 redis

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

redis Redis is an open source key-value store that… 8779 [OK]

bitnami/redis Bitnami Redis Docker Image 167 [OK]

rediscommander/redis-commander Alpine image for redis-commander - Redis man… 47 [OK]

* --no-trunc: 不截断输出结果

[root@alvin-test-os ~]# docker search -f stars=8 --limit 3 --no-trunc redis

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

redis Redis is an open source key-value store that functions as a data structure server. 8779 [OK]

bitnami/redis Bitnami Redis Docker Image 167 [OK]

rediscommander/redis-commander Alpine image for redis-commander - Redis management tool. 47 [OK]

###### 输出参数释义

* NAME: 镜像仓库源的名称
* DESCRIPTION: 镜像的描述
* OFFICIAL: 是否 docker 官方发布
* stars: 类似 Github 里面的 star，表示点赞、喜欢的意思。
* AUTOMATED: 自动构建。

##### 删除和清理镜像

在docker中，删除镜像主要使用rmi子命令，清理镜像主要使用prune子命令。

###### 使用标签删除镜像

使用docker rmi 或docker image rm命令可以删除镜像，命令格式为docker rmi image

[root@alvin-test-os ~]# docker rmi redis

Untagged: redis:latest

Untagged: redis@sha256:5b98e32b58cdbf9f6b6f77072c4915d5ebec43912114031f37fa5fa25b032489

参数：

* -f : 强制删除镜像。

[root@alvin-test-os ~]# docker rmi -f redis

Untagged: redis:latest

Untagged: redis@sha256:5b98e32b58cdbf9f6b6f77072c4915d5ebec43912114031f37fa5fa25b032489

###### 清理镜像

使用一段时间之后，docker会产生很多临时文件，以及一些没有被使用的镜像， 我们可以通过docker image prune命令来进行清理。

参数：

* -a ：删除所有无用的镜像，不光是临时镜像。

# 查看原来的镜像

[root@alvin-test-os ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myredis latest 74d107221092 19 hours ago 104MB

redis latest 74d107221092 19 hours ago 104MB

registry.cn-hangzhou.aliyuncs.com/alvinos/teach 1000-teach-2020-10-28-13-40-27 22fdec3d9a6d 3 weeks ago 1.02GB

Nginx

# 删除所有的未被使用的镜像

[root@alvin-test-os ~]# docker image prune -a

WARNING! This will remove all images without at least one container associated to them.

Are you sure you want to continue? [y/N] y

Deleted Images:

untagged: nginx:1.17

untagged: nginx@sha256:6fff55753e3b34e36e24e37039ee9eae1fe38a6420d8ae16ef37c92d1eb26699

deleted: sha256:9beeba249f3ee158d3e495a6ac25c5667ae2de8a43ac2a8bfd2bf687a58c06c9

deleted: sha256:8fb6373b4cca3383756d7fd7843dd92f95827e5f2913609e09a9621dcddb3752

deleted: sha256:8b09841626797a03a9fe5e73aa38aeacf9ff0ce85a3004236ff35234eec3b35c

deleted: sha256:ffc9b21953f4cd7956cdf532a5db04ff0a2daa7475ad796f1bad58cfbaf77a07

untagged: myredis:latest

deleted: sha256:74d107221092875724ddb06821416295773bee553bbaf8d888ababe9be7b947f

deleted: sha256:d951b383737320b4e1ac7f9bb63f3919bcf25363ccb59fbb52a41e45ba70ffdd

deleted: sha256:d3b2581a1c92973ee9a41fc00e5628047ce7e644a66240fb859b38831bd525b4

deleted: sha256:a447231da503a58432b4d7409980139206fdf398fbde189d8a7229dd0663f472

deleted: sha256:f786204ca260bcaef3d47ecad10821878028239072c65ceb2a1f212f275f9367

deleted: sha256:b68afce5f52461f79be59806be00e43ea95152a0358b8dc5de9ac3f486a70d7e

deleted: sha256:f5600c6330da7bb112776ba067a32a9c20842d6ecc8ee3289f1a713b644092f8

Total reclaimed space: 231MB

* -f ：强制删除镜像，而不进行提示。

alvin@AlvindeMacBook-Pro： docker image prune -a -f

Deleted Images:

untagged: registry.cn-hangzhou.aliyuncs.com/alvinos/swoole:latest

untagged: registry.cn-hangzhou.aliyuncs.com/alvinos/swoole@sha256:9f39e18ba29c4e5d064d316663d09ac125104d379cb2e7e4e6d3ee7ad4470584

deleted: sha256:d48372a060549a465a0eb23adfbbd59506bc809fc1bf37125c3ce180d07cd455

deleted: sha256:c251abf00a09541780fd972d284bf9c5213a68890b922d2ff2e7ff36009db67e

deleted: sha256:b340ab85cc561d2dd2776be0197b0847f4b3228cd73eeaaebac7d17e03ba8388

deleted: sha256:a78a173ce6a255e9e941a9668eee7ae88a64faa19eb79f76c19ff12623c59d1d

deleted: sha256:26f2d6d97c083ecff570e3b4b945395b1062bcf65f9eaec0afb6e90ef9b7b671

deleted: sha256:a84b2d4c28c5eec8c3401015ab5fffb759304a1b7b6aca09d9eb77ebede634ac

deleted: sha256:2e215f5f1db2ad5093c4f3879dc6b9c4f7de494b3654e09572ee51738a7f559f

deleted: sha256:aad5610214ba2382858fd8d436dfd34b86e39978ca89aefff61b2781018ab019

deleted: sha256:da7b0b2c986dcf127345bafad35ae18a66fd1ba2e99f8302fc3e73822c2d9959

deleted: sha256:9c7c0fade080ec64e220d78e6579d5a7fd37f64fd322d8b529148b8ca9d25e8f

deleted: sha256:9b57ee5d38e3c490237848c2a37720f7fe7eb8cf35b4143dda2f921fd331d640

deleted: sha256:89c094c94208c4a064195ff1daf5db1b00b2584318a0e5fb28c31268a8086787

deleted: sha256:63fcc276289672a37ac9bb918e1a9839a48674057aaec5693181764e6945a756

deleted: sha256:3168d43cb9db5a2f62f4fb6ddf208891eec6f1ee03ceb1a44aa6df39d9d1cc51

deleted: sha256:fe7e89f5270da83dfbfe2df62622262cb2555c9701daececda29906b03136f80

deleted: sha256:a2af60f6d03fe6b181e0c1c3d8b008d598acd8c1098a1f267887a2e9de7c3998

deleted: sha256:47860335fff22fac8885ba0e7a22c9287a2c076aed25d06319fefeee1704cc1f

untagged: nginx:latest

##### 构建镜像

构建镜像一般有三种情况，基于容器导入、基于本地模板导入、基于Dockerfile创建，本章主讲基于容器保存镜像和本地保存镜像导入。

###### 基于容器保存

主要格式：docker [container id] commit

主要参数：

|  |  |
| --- | --- |
| -a | 作者信息 |
| -m | 提交信息 |
| -p | 提交时，暂停容器运行 |

**启动一个容器、修改并保存**

[root@alvin-test-os ~]# docker run -d -it centos /bin/bash

Unable to find image 'centos:latest' locally

latest: Pulling from library/centos

3c72a8ed6814: Pull complete

Digest: sha256:76d24f3ba3317fa945743bb3746fbaf3a0b752f10b10376960de01da70685fbd

Status: Downloaded newer image for centos:latest

820a17fe3935938b0d4208797f165db22e205e2a15d6dd053b4aeedbcba6cb12

[root@alvin-test-os ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

820a17fe3935 centos "/bin/bash" About a minute ago Up About a minute happy\_kare

[root@alvin-test-os ~]# docker exec 820a17fe3935 touch test

[root@alvin-test-os ~]# docker commit -m "Add a file" -a "Alvin" 820a17fe3935 centos:v1

sha256:407b93005a8f27cd7eba5a10077994cf27c7a978e542fcf5b85a9caed0003bc9

[root@alvin-test-os ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

centos v1 407b93005a8f 4 seconds ago 215MB

###### 基于本地模板导入

用户也可以将保存的镜像传到另一台机器上，另一台机器就可以直接导入。

[root@alvin-test-os ~]# cat ubuntu-18.04-x86\_64-minimal.tar.gz | docker import - ubuntu:18.04

##### 保存镜像

有时我们需要将一台电脑上的镜像复制到另一台电脑上使用，除了可以借助仓库外，还可以直接将镜像保存成一个文件，再拷贝到另一台电脑上导入使用。

对于镜像的导出和导入，Docker 提供了两种方案，下面分别进行介绍。

###### 使用 export 和 import

export 和 import的针对点是容器，将本机的容器导出为镜像包。

使用export保存容器为镜像

[root@instance-gvpb80ao docs]# docker export daf9c3656be3 > nginx.tar

[root@instance-gvpb80ao docs]# ll | grep nginx.tar

-rw-r--r-- 1 root root 135117824 9月 24 20:51 nginx.tar

使用import导入包为镜像

[root@instance-gvpb80ao docs]# docker import nginx.tar test/nginx:v1

sha256:02107323de1b074c5d2034b01eff855fec5922b45776c2721882d100ba6dd15b

[root@instance-gvpb80ao docs]# docker images | grep test

test/nginx v1 02107323de1b 22 seconds ago 131MB

###### 使用 save 和 load

save 和 load的针对的点是镜像，将本机的镜像导入、导出为镜像包。

使用save保存镜像

[root@instance-gvpb80ao docs]# docker save 6858809bf669 > busybox.tar

[root@instance-gvpb80ao docs]# ll | grep busy

-rw-r--r-- 1 root root 1458176 9月 24 21:01 busybox.tar

使用save保存多个镜像

[root@instance-gvpb80ao docs]# docker save -o test.tar busybox nginx:1.18.0

[root@instance-gvpb80ao docs]# docker load < test.tar

Loaded image: busybox:latest

Loaded image: nginx:1.18.0

使用load导入镜像

[root@instance-gvpb80ao docs]# docker load < busybox.tar

[root@instance-gvpb80ao docs]# docker load -i busybox.tar

Loaded image ID: sha256:6858809bf669cc5da7cb6af83d0fae838284d12e1be0182f92f6bd96559873e3

[root@instance-gvpb80ao docs]# docker images | grep 685880

busybox latest 6858809bf669 2 weeks ago 1.23MB

###### 两种方案的差别

文件大小不同

export导出的镜像文件体积小于 save 保存的镜像

是否可以对镜像重命名

docker import可以为镜像指定新名称，docker load不能对载入的镜像重命名。

是否可以同时将多个镜像打包到一个文件中

docker export不支持，docker save支持。

是否包含镜像历史

export 导出（import 导入）是根据容器拿到的镜像，再导入时会丢失镜像所有的历史记录和元数据信息（即仅保存容器当时的快照状态），所以无法进行回滚操作。

save 保存（load 加载）的镜像，没有丢失镜像的历史，可以回滚到之前的层（layer）。

[root@instance-gvpb80ao docs]# docker history 6858809bf669cc5da7cb6af83d0fae838284d12e1be0182f92f6bd96559873e3

IMAGE CREATED CREATED BY SIZE COMMENT

6858809bf669 2 weeks ago /bin/sh -c #(nop) CMD ["sh"] 0B

<missing> 2 weeks ago /bin/sh -c #(nop) ADD file:72be520892d0a903d… 1.23MB

应用场景不同

docker export的应用场景：主要用来制作基础镜像，比如我们从一个 ubuntu 镜像启动一个容器，然后安装一些软件和进行一些设置后，使用 docker export保存为一个基础镜像。然后，把这个镜像分发给其他人使用，比如作为基础的开发环境。

docker save的应用场景：如果我们的应用是使用docker-compose.yml编排的多个镜像组合，但我们要部署的客户服务器并不能连外网。这时就可以使用docker save将用到的镜像打个包，然后拷贝到客户服务器上使用docker load载入。

### 使用Docker容器

#### 创建容器

在Docker中，真正对外提供服务的还是容器，本小节咱们来创建一个容器。

创建容器的格式：docker run [option] image [cmd]

[root@alvin-test-os ~]# docker run -d --name nginx -p 80:80 nginx

Unable to find image 'nginx:latest' locally

latest: Pulling from library/nginx

852e50cd189d: Pull complete

a29b129f4109: Pull complete

b3ddf1fa5595: Pull complete

c5df295936d3: Pull complete

232bf38931fc: Pull complete

Digest: sha256:c3a1592d2b6d275bef4087573355827b200b00ffc2d9849890a4f3aa2128c4ae

Status: Downloaded newer image for nginx:latest

6381d29d6e0ec3f6b01cf1aabb58b799ee88acf1a722e251807c9cb44e73a3e0

##### 容器启动参数

|  |  |
| --- | --- |
| -d | 以守护进程方式运行 |
| -p | 指定映射端口 |
| -P | 随机映射端口 |
| -i | 保持标准输入打开 |
| -t | 分配一个伪终端 |
| -v | 设置挂载文件到主机上 |
| --rm | 当容器关闭时自动删除 |
| --name | 为启动的容器设置一个名字 |
| --network | 指定使用哪个网络 |
| -e | 设置容器中的环境变量 |
| --link | 链接到另一个容器 |
| -h | 指定容器内的主机名 |

#### 停止容器

docker终止容器是首先向容器发送SIGTERM信号，等待一段时间超时后（默认10秒），再发送SIGKILL信号来终止容器。

[root@alvin-test-os ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6381d29d6e0e nginx "/docker-entrypoint.…" 26 minutes ago Up 25 minutes 0.0.0.0:80->80/tcp nginx

820a17fe3935 centos "/bin/bash" 4 hours ago Up 4 hours happy\_kare

[root@alvin-test-os ~]# docker stop nginx

nginx

[root@alvin-test-os ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

820a17fe3935 centos "/bin/bash" 4 hours ago Up 4 hours happy\_kare

可见，NGINX容器已经被我们终止了。这个时候我们也可以重新启动这个容器。

[root@alvin-test-os ~]# docker start nginx

nginx

[root@alvin-test-os ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6381d29d6e0e nginx "/docker-entrypoint.…" 34 minutes ago Up 1 second 0.0.0.0:80->80/tcp nginx

#### 进入容器

在使用容器的过程中，我们难免需要进入容器进行排查问题。下面我们就来介绍进入容器的集中方式。

##### attach

attach是最早docker官方推出的进入容器的命令了，不过使用该命令有一个问题。当多个窗口同时使用该命令进入该容器时，所有的窗口都会同步显示。如果有一个窗口阻塞了，那么其他窗口也无法再进行操作，当所有窗口退出时，容器结束。

[root@instance-gvpb80ao docs]# docker attach nginx

127.0.0.1 - "GET / HTTP/1.1" 308 171 "-" "curl/7.59.0" 0.000 - .

##### exec

既attach之后，exec是官方推出的有一个新的进入容器的命令，这个命令相当于在容器中执行一个命令。

[root@instance-gvpb80ao docs]# docker exec -it nginx /bin/bash

nginx [ / ]$

nginx [ / ]$

##### nsenter

需要配合docker inspect来使用（早期没有exec命令时，企业当中最长用的方式之一）,Docker是用golang语言开发，所以它也支持go语言的摸版语法。

[root@instance-gvpb80ao docs]# nsenter --target $( docker inspect -f {{.State.Pid}} nginxv1 ) --mount --uts --ipc --net --pid

mesg: ttyname failed: No such device

root@6f99ae8757f7:/#

##### ssh

在生产环境中排除了使用docker attach命令进入容器之后，相信大家第一个想到的就是ssh。在镜像（或容器）中安装SSH Server，这样就能保证多人进入容器且相互之间不受干扰了，相信大家在当前的生产环境中（没有使用Docker的情况）也是这样做的。但是使用了Docker容器之后不建议使用ssh进入到Docker容器内。

##### 总结

进入docker container中一般情况下有4种方式，最常用的是exec和nsenter这两种。

Nsenter和exec之间的区别？

* Exec是docker自带的命令，Nsenter是Linux提供的命令。
* Exec相当于在容器内执行一个命令，而Nsenter是仅仅进入容器之中而已。

#### 删除容器

可以使用 docker rm 命令来删除处于终止或退出状态的容器，命令格式为：docker rm container。

[root@alvin-test-os ~]# docker rm nginx

##### 强制删除

强制删除一个正在运行的容器。

[root@alvin-test-os ~]# docker rm -f nginx

nginx

#### 导入与导出容器

某些时候，需要将容器从一个系统迁移到另外一个系统，此时可以使用 Docker 的导人和导出功能，这也是 Docker 自身提供的一个重要特性。

##### 导出容器

导出容器是指，导出一个已经创建的容器到一个文件，不管此时这个容器是否处于运行状态 可以使用 docker [container] export 令，该命令格式为：

[root@instance-gvpb80ao docs]# docker export daf9c3656be3 > nginx.tar

[root@instance-gvpb80ao docs]# ll | grep nginx.tar

##### 导入容器

导出的文件又可以使用 docker [ container] import 命令导人变成镜像，该命令格式为：

[root@instance-gvpb80ao docs]# docker import nginx.tar test/nginx:v1

sha256:02107323de1b074c5d2034b01eff855fec5922b45776c2721882d100ba6dd15b

[root@instance-gvpb80ao docs]# docker images | grep test

test/nginx v1 02107323de1b 22 seconds ago 131MB

实际上，既可以使用 docker load 命令来导入镜像存储文件到本地镜像库，也可以使docker [container] import 命令来导入一个容器快照到本地镜像库 这两者的区别在于 容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积更大 此外，从容器快照文件导人时可以重新指定标签等元数据信息。

#### 查看容器

查看容器详情可以使用 docker container inspect [OPTIONS] CONTAINER [CONTAINER . .. ］子命令。

[root@alvin-test-os ~]# docker inspect k8s

[

{

"Id": "726b695a337c63b2ba617ed282171dff82a3d51cc0bdc43194ce63beaf656c38",

"Created": "2020-11-16T04:05:53.271281122Z",

"Path": "docker-entrypoint.sh",

"Args": [

"./start"

],

... 此处略去千百行

}

]

#### 容器命令详解

##### 复制命令

复制命令类似于Linux系统中的scp命令，是将宿主主机上的内容上传到容器中，也可能是将容器中的文件下载到宿主主机中。

# 将容器中的内容复制到宿主主机

[root@alvin-test-os ~]# docker cp 726b695a337c:/opt/start .

[root@alvin-test-os ~]# ls | grep start

Start

#将宿主主机中的文件复制到容器中

[root@alvin-test-os ~]# docker cp start 726b695a337c:/root

[root@alvin-test-os ~]# docker exec 726b695a337c ls /root

start

### Docker网络

Docker本身的技术依赖于Linux内核虚拟化技术的发展。所以Docker对Linux内核的特性有很强的依赖。本章主要介绍Docker所使用的Linux网络技术。

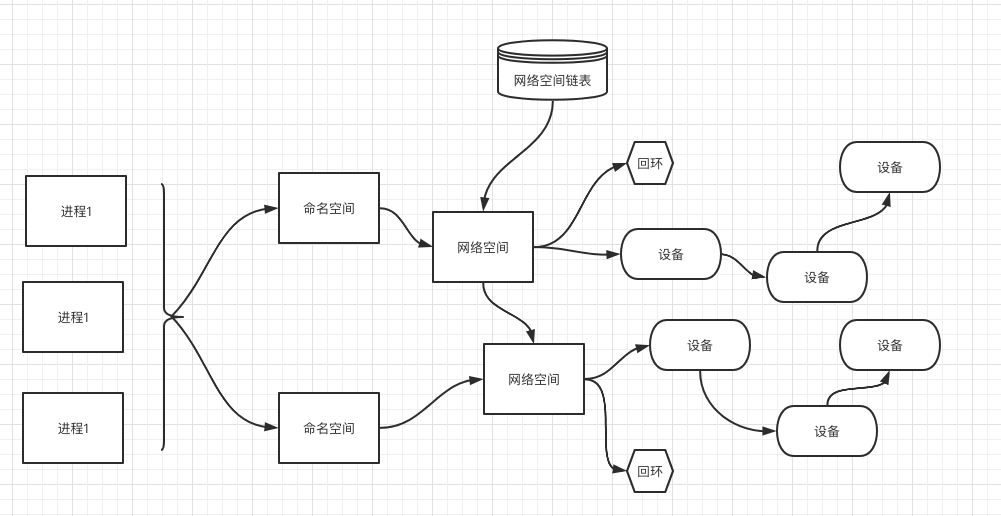
#### 网络基础

其中Docker使用到的与Linux网络有关的技术分别有：网络名称空间、Veth、Iptables、网桥、路由。

##### 网络名称空间

为了支持网络协议栈的多个实例，Linux在网络协议栈中引入了网络名称空间(Network Namespace)，这些独立的协议栈被隔离到不同的命名空间中。处于不同的命名空间的网络协议栈是完全隔离的，彼此之间无法进行网络通信，就好像两个“平行宇宙”。通过这种对网络资源的隔离，就能在一个宿主机上虚拟多个不同的网络环境，而Docker正是利用这种网络名称空间的特性，实现了不同容器之间的网络隔离。在Linux的网络命名空间内可以有自己独立的Iptables来转发、NAT及IP包过滤等功能。

Linux的网络协议栈是十分复杂的，为了支持独立的协议栈，相关的这些全局变量都必须修改为协议栈私有。最好的办法就是让这些全局变量成为一个Net Namespace变量的成员，然后为了协议栈的函数调用加入一个Namespace参数。这就是Linux网络名称空间的核心。所以的网络设备都只能属于一个网络名称空间。当然，通常的物理网络设备只能关联到root这个命名空间中。虚拟网络设备则可以被创建并关联到一个给定的命名空间中，而且可以在这些名称空间之间移动。



##### 创建一个命名空间

[root@alvin-test-os ~]# ip netns add test01

[root@alvin-test-os ~]# ip netns add test02

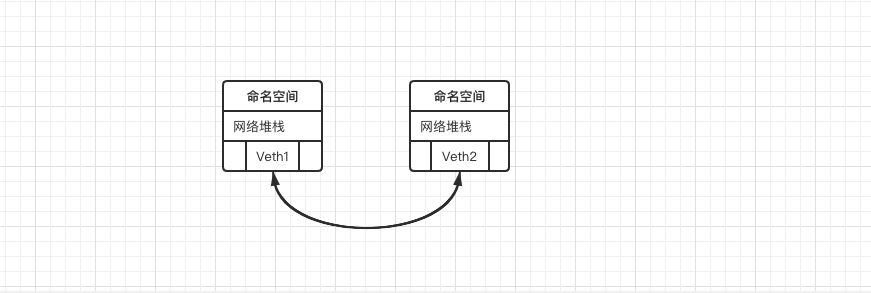
[root@alvin-test-os ~]# ip netns list

test02

test01

###### Veth设备

引入Veth设备对是为了在不同的网络名称空间之间进行通信，利用它可以直接将两个网络名称空间链接起来。由于要连接的两个网络命名空间，所以Veth设备是成对出现的，很像一对以太网卡，并且中间有一根直连的网线。既然是一对网卡，那么我们将其中一端称为另一端的peer。在Veth设备的一端发送数据时，它会将数据直接发送到另一端，并触发另一端的接收操作。



###### Veth设备操作

创建Veth设备对

[root@alvin-test-os ~]# ip link add veth type veth peer name veth001

[root@alvin-test-os ~]# ip link show

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000

link/ether 00:0c:29:e2:5b:2d brd ff:ff:ff:ff:ff:ff

3: docker0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default

link/ether 02:42:ad:75:28:7e brd ff:ff:ff:ff:ff:ff

7: veth4cb1ab3@if6: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default

link/ether ce:90:13:71:7f:8f brd ff:ff:ff:ff:ff:ff link-netnsid 0

9: vethbbf7148@if8: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default

link/ether 7e:6c:9f:d5:5b:26 brd ff:ff:ff:ff:ff:ff link-netnsid 1

**14: veth001@veth: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000**

**link/ether 96:f1:a2:1d:1d:10 brd ff:ff:ff:ff:ff:ff**

**15: veth@veth001: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000**

**link/ether 66:af:ad:f0:5f:6d brd ff:ff:ff:ff:ff:ff**

生成了两个veth设备， 互为对方的peer。

绑定命名空间

[root@alvin-test-os ~]# ip link set veth001 netns test01

[root@alvin-test-os ~]# ip link show | grep veth

7: veth4cb1ab3@if6: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default

9: vethbbf7148@if8: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default

15: veth@if14: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000

已经查看不到veth001，当我们进入test01命名空间之后，就可以查看到

[root@alvin-test-os ~]# ip netns exec test01 bash

[root@alvin-test-os ~]# ip link show

1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

14: veth001@if15: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000

link/ether 96:f1:a2:1d:1d:10 brd ff:ff:ff:ff:ff:ff link-netnsid 0

将Veth分配IP

# 设置IP

[root@alvin-test-os ~]# ip netns exec test01 ip addr add 172.16.0.111/20 dev veth001

# 绑定

[root@alvin-test-os ~]# ip netns exec test01 ip link set dev veth001 up

# 查看

[root@alvin-test-os ~]# ip netns exec test01 ip a

1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

14: veth001@if15: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state LOWERLAYERDOWN group default qlen 1000

link/ether 96:f1:a2:1d:1d:10 brd ff:ff:ff:ff:ff:ff link-netnsid 0

inet 172.16.0.111/20 scope global veth001

valid\_lft forever preferred\_lft forever

这个时候双方就通了。

查看对端Veth设备

[root@alvin-test-os ~]# ip netns exec test01 ethtool -S veth001

NIC statistics:

peer\_ifindex: 15

[root@alvin-test-os ~]# ip a | grep 15

14: veth001@if15: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state LOWERLAYERDOWN group default qlen 1000

为对端Veth设备设置IP

[root@alvin-test-os ~]# ip addr add 172.16.0.112/20 dev veth

[root@alvin-test-os ~]# ip link set dev veth down

[root@alvin-test-os ~]# ip link set dev veth up

[root@alvin-test-os ~]# ping 172.16.0.111

PING 172.16.0.111 (172.16.0.111) 56(84) bytes of data.

64 bytes from 172.16.0.111: icmp\_seq=1 ttl=64 time=0.126 ms

64 bytes from 172.16.0.111: icmp\_seq=2 ttl=64 time=0.081 ms

^C

--- 172.16.0.111 ping statistics ---

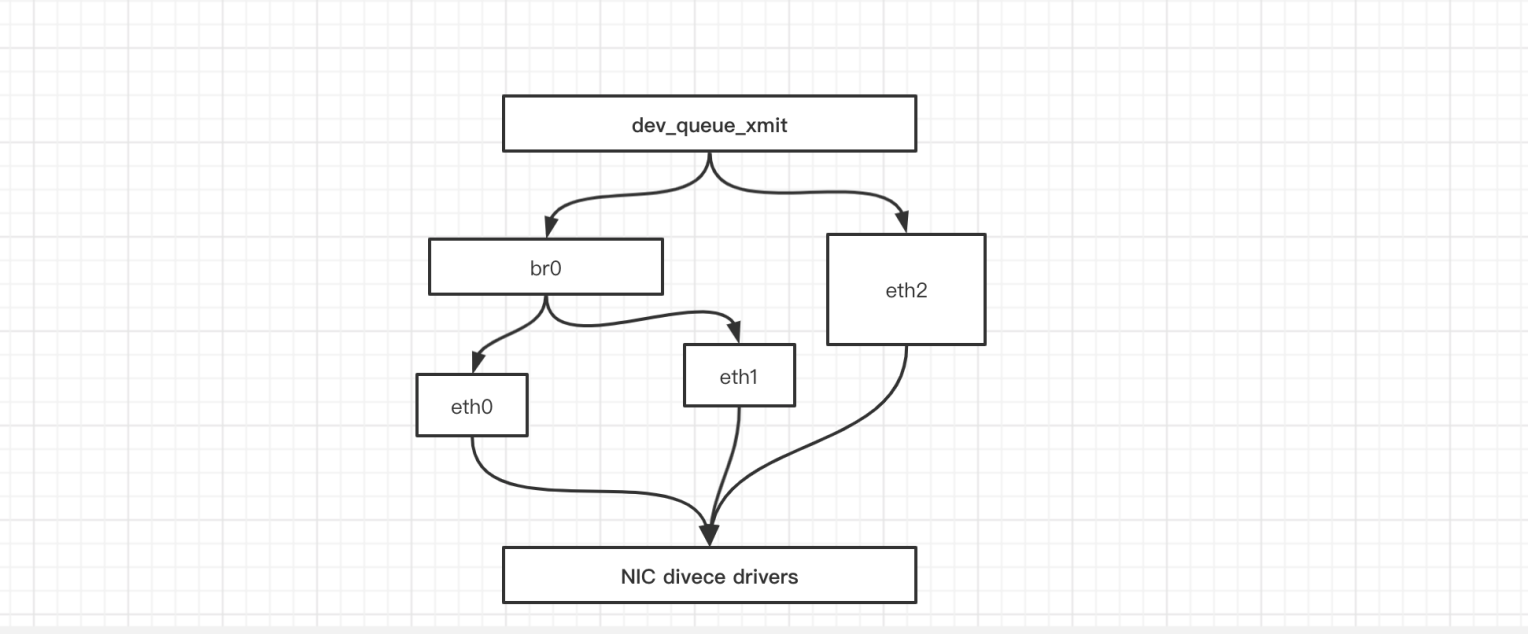
2 packets transmitted, 2 received, 0% packet loss, time 999ms

rtt min/avg/max/mdev = 0.081/0.103/0.126/0.024 ms

##### 网桥

Linux 可以支持多个不同的网络，它们之间能够相互通信，就需要一个网桥。 网桥是二层的虚拟网络设备，它是把若干个网络接口“连接”起来，从而报文能够互相转发。网桥能够解析收发的报文，读取目标 MAC 地址的信息，和自己记录的 MAC 表结合，来决定报文的转发目标网口。

网桥设备 brO 绑定了 eth0、 eth1 。对于网络协议械的上层来说，只看得到 brO 。因为桥接是在数据链路层实现的 ，上层不需要关心桥接的细节，于是协议枝上层需要发送的报文被送到 brO ，网桥设备的处理代码判断报文该被转发到 ethO 还是 ethl ，或者两者皆转发。反过来，从 ethO 或从 ethl 接收到的报文被提交给网桥的处理代码，在这里会判断报文应该被转发、丢弃还是提交到协议枝上层。 而有时 ethl 也可能会作为报文的源地址或目的地址 直接参与报文的发送与接收，从而绕过网桥。



##### Iptables

我们知道， Linux 络协议樵非常高效，同时比较复杂 如果我们希望在数据的处理过程中对关心的数据进行一些操作该怎么做呢？ Linux 提供了一套机制来为用户实现自定义的数据包处理过程。

在Linux网络协议棋中有一组回调函数挂接点，通过这些挂接点挂接的钩子函数可以在Linux 网络棋处理数据包的过程中对数据包进行 些操作，例如过滤、修改、丢弃等 整个挂接点技术叫作 Netfilter lptables

Netfilter 负责在内核中执行各种挂接的规则，运行在内核模式中：而 lptables 是在用户模式下运行的进程，负责协助维护内核中 Netfilter 的各种规则表 通过 者的配合来实现整个 Linux网络协议战中灵活的数据包处理机制。

##### 总结

|  |  |
| --- | --- |
| 设备 | 作用总结 |
| network  namespace | 主要提供了关于网络资源的隔离，包括网络设备、IPv4和IPv6协议栈、IP路由表、防火墙、/proc/net目录、/sys/class/net目录、端口（socket）等。 |
| linux Bridge | 功能相当于物理交换机，为连在其上的设备（容器）转发数据帧。如docker0网桥。 |
| iptables | 主要为容器提供NAT以及容器网络安全。 |
| veth pair | 两个虚拟网卡组成的数据通道。在Docker中，用于连接Docker容器和Linux Bridge。一端在容器中作为eth0网卡，另一端在Linux Bridge中作为网桥的一个端口。 |

#### Docker网络模式

Docker使用Linux桥接的方式，在宿主机虚拟一个Docker容器网桥(docker0)，Docker启动一个容器时会根据Docker网桥的网段分配给容器一个IP地址，称为Container-IP，同时Docker网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能够通过容器的Container-IP直接通信。

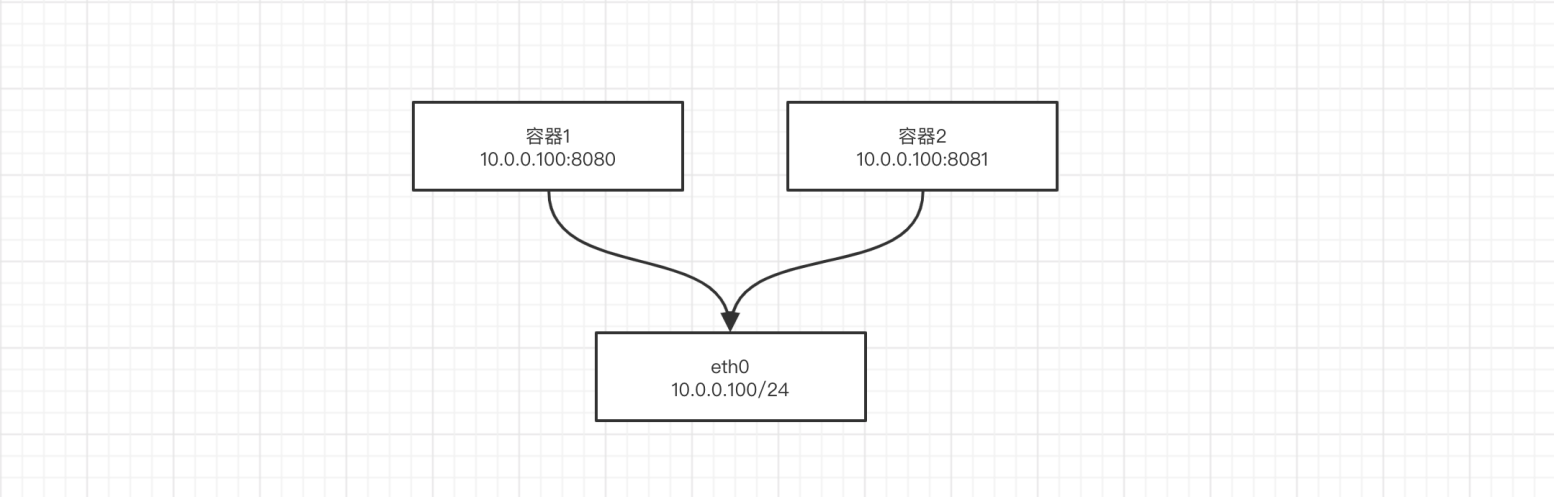
Docker网桥是宿主机虚拟出来的，并不是真实存在的网络设备，外部网络是无法寻址到的，这也意味着外部网络无法通过直接Container-IP访问到容器。如果容器希望外部访问能够访问到，可以通过映射容器端口到宿主主机（端口映射），即docker run创建容器时候通过 -p 或 -P 参数来启用，访问容器的时候就通过[宿主机IP]:[容器端口]访问容器。

|  |  |  |
| --- | --- | --- |
| **Docker网络模型** | **配置** | **说明** |
| host模式 | –-network=host | 容器和宿主机共享Network namespace。 |
| containe模式 | --network=container:ID | 容器和另外一个容器共享Network namespace。 kubernetes中的pod就是多个容器共享一个Network namespace。 |
| none模式 | --network=none | 容器有独立的Network namespace，但并没有对其进行任何网络设置，如分配veth pair 和网桥连接，配置IP等。 |
| bridge模式 | --network=bridge | 当Docker进程启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的Docker容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。（默认为该模式） |

##### HOST模式

如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

使用host模式的容器可以直接使用宿主机的IP地址与外界通信，容器内部的服务端口也可以使用宿主机的端口，不需要进行NAT，host最大的优势就是网络性能比较好，但是docker host上已经使用的端口就不能再用了，网络的隔离性不好。



[root@instance-gvpb80ao ~]# docker run -d --name my-web --network host nginx

Unable to find image 'nginx:latest' locally

latest: Pulling from library/nginx

d121f8d1c412: Pull complete

ebd81fc8c071: Pull complete

655316c160af: Pull complete

d15953c0e0f8: Pull complete

2ee525c5c3cc: Pull complete

Digest: sha256:c628b67d21744fce822d22fdcc0389f6bd763daac23a6b77147d0712ea7102d0

Status: Downloaded newer image for nginx:latest

06941559a3f7e0c53cf228302dedc2040c10f2eb0b6e3d0f962c065b0e0419ce

[root@instance-gvpb80ao ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

06941559a3f7 nginx "/docker-entrypoint.…" 6 minutes ago Up 6 minutes my-web

[root@instance-gvpb80ao ~]# curl 127.0.0.1:80

<!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 please 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>

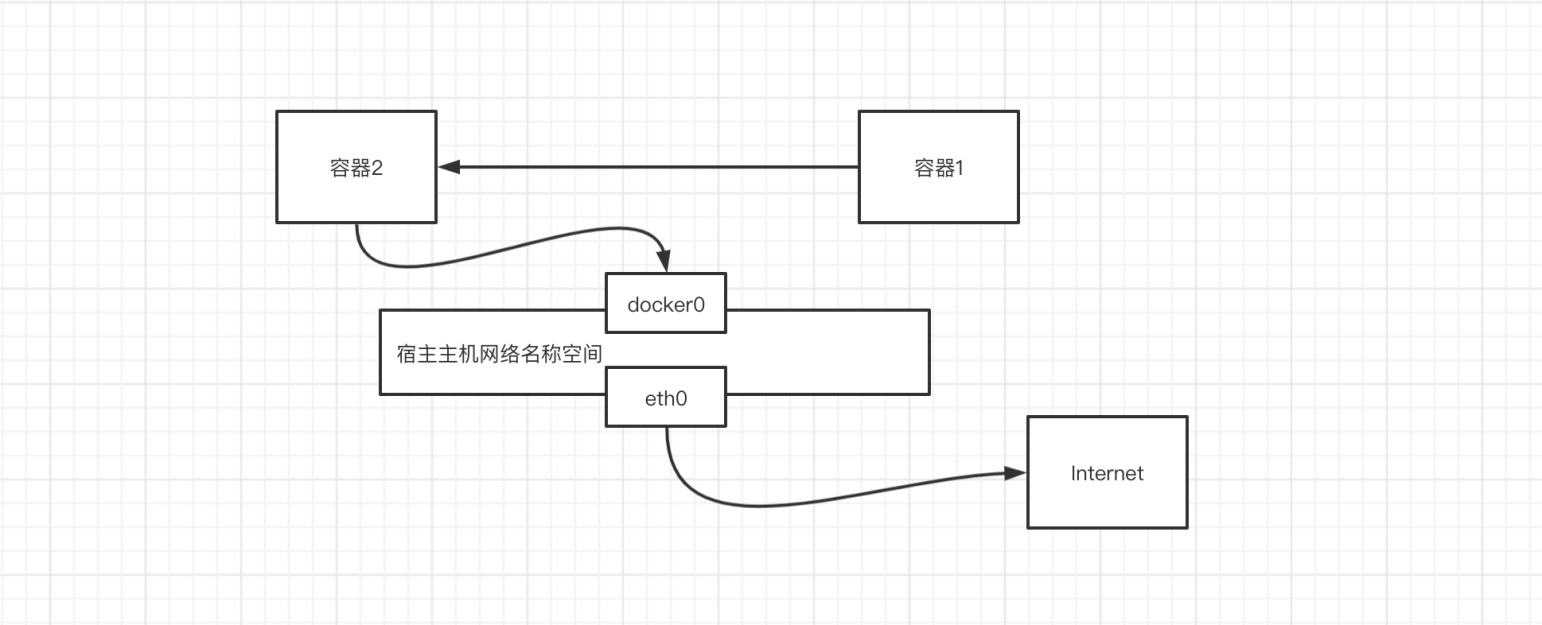
<p><em>Thank you for using nginx.</em></p>

</body>

</html>

##### Containe模式

这个模式指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。



[root@instance-gvpb80ao ~]# docker run -itd --name test01 busybox

Unable to find image 'busybox:latest' locally

latest: Pulling from library/busybox

df8698476c65: Pull complete

Digest: sha256:d366a4665ab44f0648d7a00ae3fae139d55e32f9712c67accd604bb55df9d05a

Status: Downloaded newer image for busybox:latest

dd99687d303d61d649364084ad353ec38b4e07149b0328301ec9691f18669951

[root@instance-gvpb80ao ~]# docker run -itd --name test02 --network "container:test01" busybox

72adb0dcdb93b6adc7a62e1ad1ac274f64b8cec941d5f89da43fcd0757f99fa3

[root@instance-gvpb80ao ~]# docker exec -it test02 sh

/ # ifconfig

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

inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:28 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:2423 (2.3 KiB) TX bytes:0 (0.0 B)

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

/ # exit

[root@instance-gvpb80ao ~]# docker exec -it test01 sh

/ # ifconfig

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

inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:29 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:2493 (2.4 KiB) TX bytes:0 (0.0 B)

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

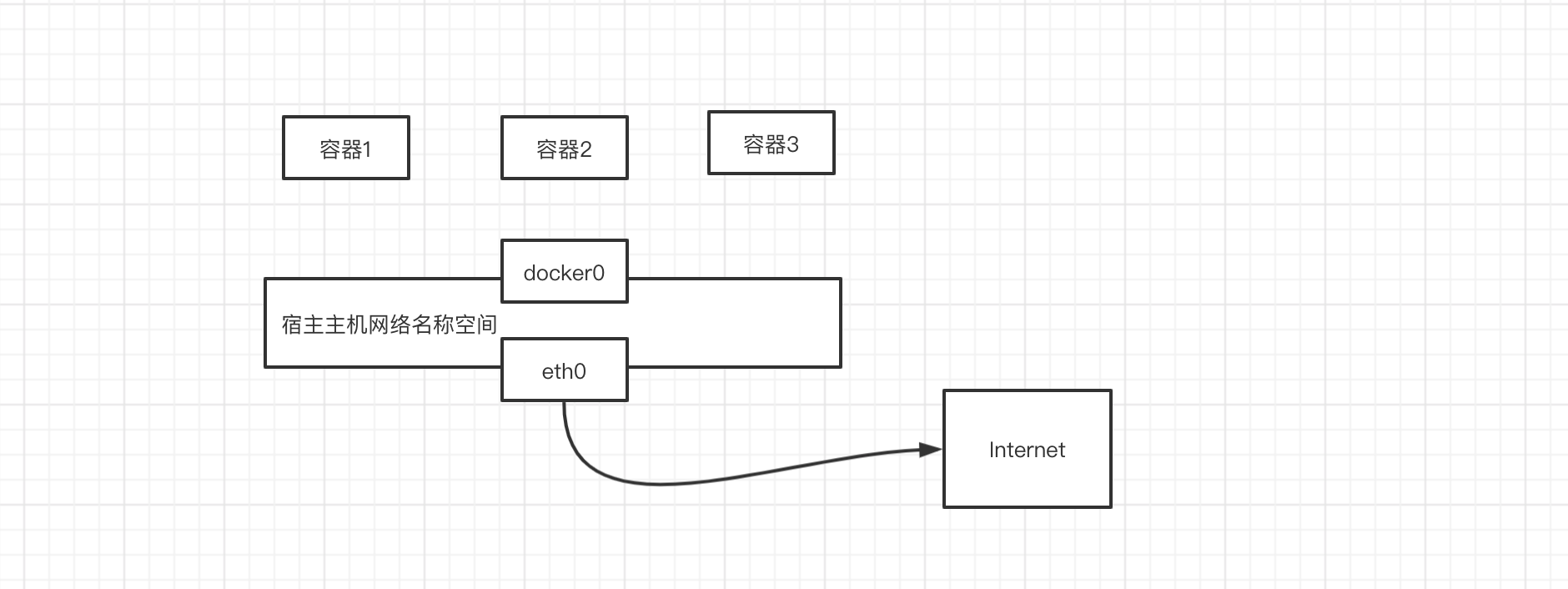
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

/ #

##### none模式

使用none模式，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等。

这种网络模式下容器只有lo回环网络，没有其他网卡。none模式可以在容器创建时通过--network=none来指定。这种类型的网络没有办法联网，封闭的网络能很好的保证容器的安全性。



[root@instance-gvpb80ao ~]# docker run -itd --name test03 --network none busybox

b9dde79754bc110314be4aecde1251dcdd6ce28fdf43102ae184b79c6e7414bc

[root@instance-gvpb80ao ~]# docker exec -it test03 sh

/ # ifconfig

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

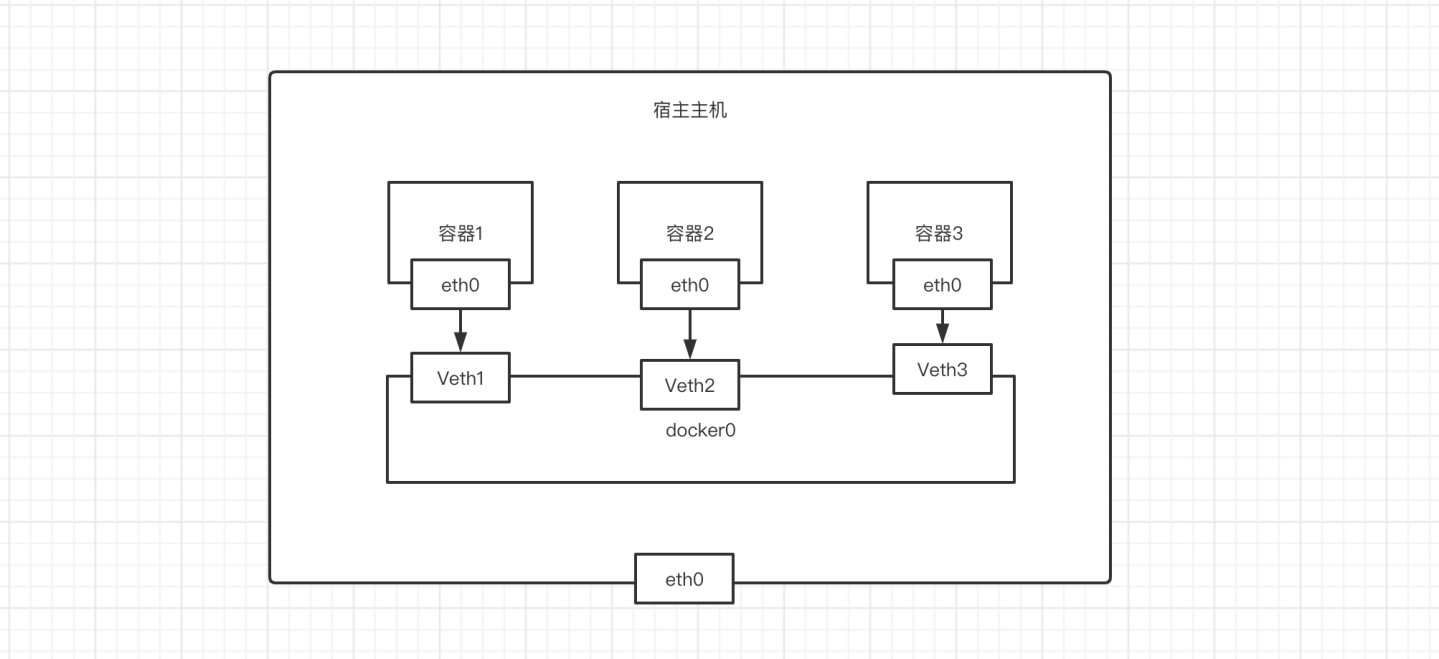
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

##### bridge模式

当Docker进程启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的Docker容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。

从docker0子网中分配一个IP给容器使用，并设置docker0的IP地址为容器的默认网关。在主机上创建一对虚拟网卡veth pair设备，Docker将veth pair设备的一端放在新创建的容器中，并命名为eth0（容器的网卡），另一端放在主机中，以vethxxx这样类似的名字命名，并将这个网络设备加入到docker0网桥中。可以通过brctl show命令查看。

bridge模式是docker的默认网络模式，不写--net参数，就是bridge模式。使用docker run -p时，docker实际是在iptables做了DNAT规则，实现端口转发功能。可以使用iptables -t nat -vnL查看。



[root@instance-gvpb80ao ~]# docker run -itd --name test04 busybox

dd17d863b2957a29df44b2552365eb0cfc01552fd4ec6b8b63e28dfd0d61472e

[root@instance-gvpb80ao ~]# ip a

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000

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

2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP group default qlen 1000

link/ether fa:16:3e:4a:9c:c7 brd ff:ff:ff:ff:ff:ff

inet 172.16.0.4/20 brd 172.16.15.255 scope global noprefixroute eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::f816:3eff:fe4a:9cc7/64 scope link noprefixroute

valid\_lft forever preferred\_lft forever

3: docker0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default

link/ether 02:42:ab:42:6a:8a brd ff:ff:ff:ff:ff:ff

inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0

valid\_lft forever preferred\_lft forever

inet6 fe80::42:abff:fe42:6a8a/64 scope link

valid\_lft forever preferred\_lft forever

5: veth870d996@if4: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue master docker0 state UP group default

link/ether ae:36:24:dc:11:6d brd ff:ff:ff:ff:ff:ff link-netnsid 0

inet6 fe80::ac36:24ff:fedc:116d/64 scope link

valid\_lft forever preferred\_lft forever

7: veth6acb18a@if6: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue master docker0 state UP group default

link/ether 4e:03:b1:5e:40:91 brd ff:ff:ff:ff:ff:ff link-netnsid 1

inet6 fe80::4c03:b1ff:fe5e:4091/64 scope link

valid\_lft forever preferred\_lft forever

[root@instance-gvpb80ao ~]# docker exec -it test04 bash

/ # ping 172.16.0.4

PING 172.16.0.4 (172.16.0.4): 56 data bytes

64 bytes from 172.16.0.4: seq=0 ttl=64 time=0.113 ms

64 bytes from 172.16.0.4: seq=1 ttl=64 time=0.101 ms

^C

--- 172.16.0.4 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.101/0.107/0.113 ms

### Dockerfile构建镜像

Dockerfile 由一行行命令语句组成， 并且支持以＃开头的注释行。一般而言， Dockerfile 主体内容分为四部分：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。

Docker以从上到下的顺序运行Dockerfile的指令。为了指定基本映像，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)

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

指令：COPY

功能描述：复制文件到镜像中

语法：COPY < src>… < dest>|[“< src>”,… “< dest>”]

提示：指令逻辑和ADD十分相似，同样Docker Daemon会从编译目录寻找文件或目录，dest为镜像中的绝对路径或者相对于WORKDIR的路径

#### 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。

#### 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。

#### 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构建镜像案例

FROM centos7

MAINTAINER Alvin alvincy@qq.com

RUN mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup

RUN curl -o /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.repo

RUN yum makecache

RUN yum update -y

RUN yum install python3 -y

RUN pip3 install django

COPY docker /root/docker

WORKDIR /root/docker

EXPOSE 8080

CMD ["python3", "manage.py", "runserver", "0.0.0.0:8080"]