Docker 基础

学习笔记

创作者：暮毅

联系方式：QQ24980609

发布日期：2023 年 1 月 1 日

目　录

一、docker 基本概念

Docker 是基于 Go 语言实现的开源容器项目。它诞生于 2013 年，最初的发起者是 dotCloud 公司。2013 年底 dotCloud 公司 Docker Inc。目前已成为全球最大的 Docker 容器服务提供商。

官网为：docker.com

docker 项目已加入 linux 基金会，遵循 Apache 2.0 协议，鼓励代码共享和尊重原作者的著作权，同样允许代码修改，再发布（作为开源或商业软件）。全部源代码在：http://githup.com/docker 维护。

Docker 扶持各大主流 linux 版本、MAC OS、windows.。

可以将 docker 理解为一个轻量级的沙盒。每个容器内运行着一个应用，不同容器相互隔离，容器之间也可以通过网络通信。且对系统资源的需求远低于虚拟化。

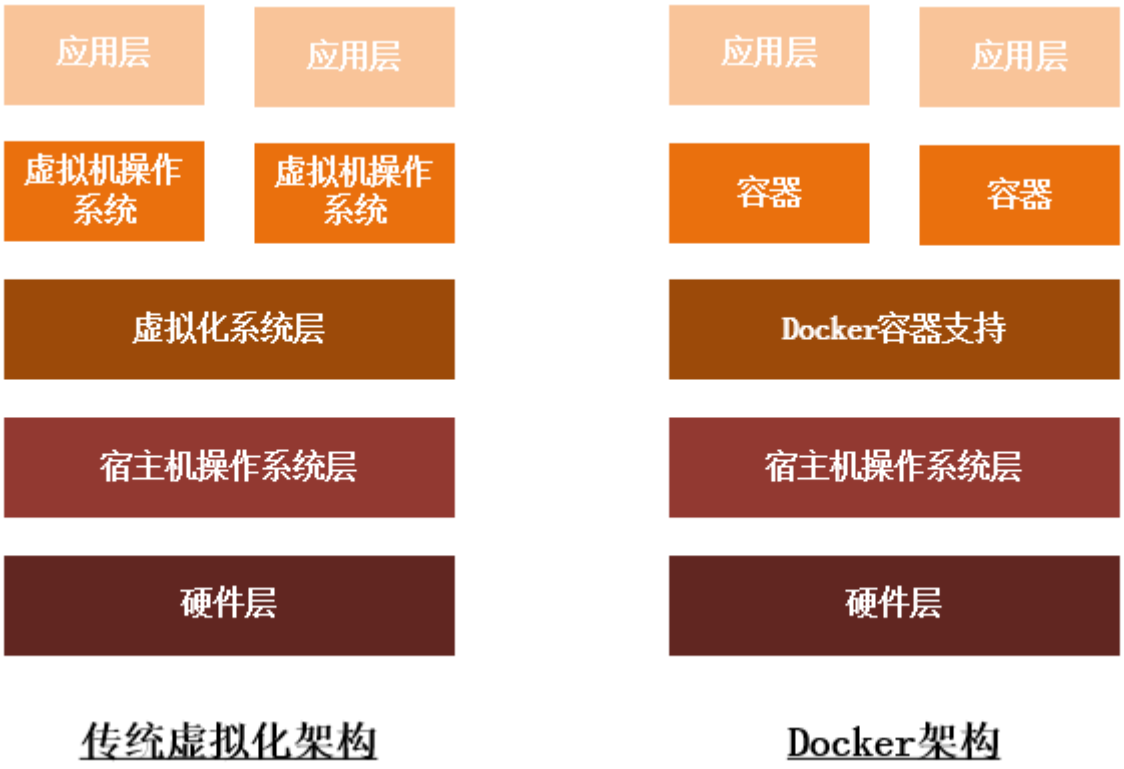
跟传统虚拟化相比，docker 最大的优势在于，除了运行在其中的应用外，本身几乎不消耗系统资源。

Docker 和传统虚拟之比较：

性能指标	Docker	虚拟化
启动速度	秒级	分钟级
性能	接近原生	相关较弱
内存代价	很少	较多
硬盘代价	一般为 MB 级别	一般为 GB 级别
运行密度	单机支持上千个容器	一般最多几十个
隔离性	完全隔离	完全隔离
迁移性	优秀	一般

Docker 属于操作系统虚拟化，即内核通过创建多个虚拟的操作实例来隔离不同的进程。

传统虚拟化和 Docker 的架构：



传统方式是在硬件层面实现虚拟化，需要有额外的虚拟化管理程序和虚拟机操作系统；

Docker 是在操作系统层面实现虚拟化，直接复用宿主机的内核，更加轻量。

二、docker 的安装

2.1 docker 三大核心概念

1）docker 镜像

- ✧ 类似于虚拟机的镜像文件，一个只读模版；
- ✧ 一个镜像包包含了一个操作系统镜像（不是操作系统），如安装了某个应用程序；
- ✧ 镜像是 docker 容器的基础。

2）docker 容器

- ✧ 类似于一个轻量级的沙盒，docker 使用容器来运行和隔离应用；
- ✧ 容器是从镜像创建的应用运行实例，容器之间相互隔离；
- ✧ 可以把容器看作是一个简易的 Linux 系统环境（不是操作系统），以及运行在其上的应用程序打包而成的盒子；
- ✧ 镜像是只读的。容器从镜像启动的时候，会在镜像的最上层创建一个可写层。

3）docker 仓库和注册服务器

- ✧ 集中存放镜像文件的场所；
- ✧ 最大的仓库是官方提供的 docker hub;
- ✧ 也可以自己创建自己的私有仓库；
- ✧ 用户可以用 pull 和 push 命令来拉取或推送镜像。

2.2 centos 7 环境安装 docker

系统版本：CentOS Linux release 7.9.2009

内核版本：3.10.0-1160.el7.x86_64

第一步：安装程序 yum 工具包集合 yum-utils

yum install yum-utils

第二步：添加 docker yum 源

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

在/etc/yum.repos.d/会出现一个 docker-ce.repo 的文件，就是 docker yum 源了。

```
[root@docker197 yum.repos.d]# ll
total 44
-rw-r--r--. 1 root root 1664 Oct 23  2020 CentOS-Base.repo
-rw-r--r--. 1 root root 1309 Oct 23  2020 CentOS-CR.repo
-rw-r--r--. 1 root root  649 Oct 23  2020 CentOS-Debuginfo.repo
-rw-r--r--. 1 root root  314 Oct 23  2020 CentOS-fasttrack.repo
-rw-r--r--. 1 root root  630 Oct 23  2020 CentOS-Media.repo
-rw-r--r--. 1 root root 1331 Oct 23  2020 CentOS-Sources.repo
-rw-r--r--. 1 root root 8515 Oct 23  2020 CentOS-Vault.repo
-rw-r--r--. 1 root root  616 Oct 23  2020 CentOS-x86_64-kernel.repo
-rw-r--r--. 1 root root 1919 Jan  5 15:22 docker-ce.repo
```

每三步：重新生成 yum 缓存

yum makecache

第四步：安装 docker 社区版

yum install docker-ce

```
Dependencies Resolved

================================================================================
 Package                    Arch          Version                    Repository                Size
================================================================================
Installing:
 docker-ce                  x86_64        3:20.10.12-3.el7           docker-ce-stable          23 M
Installing for dependencies:
 container-selinux          noarch        2:2.119.2-1.911c772.el7_8  extras                    40 k
 containerd.io              x86_64        1.4.12-3.1.el7             docker-ce-stable          28 M
 docker-ce-cli              x86_64        1:20.10.12-3.el7           docker-ce-stable          30 M
 docker-ce-rootless-extras  x86_64        20.10.12-3.el7             docker-ce-stable         8.0 M
 docker-scan-plugin         x86_64        0.12.0-3.el7               docker-ce-stable         3.7 M
 fuse-overlayfs             x86_64        0.7.2-6.el7_8              extras                    54 k
 fuse3-libs                 x86_64        3.6.1-4.el7                extras                    82 k
 slirp4netns                x86_64        0.4.3-4.el7_8              extras                    81 k

Transaction Summary
================================================================================
```

第五步：开启 docker 服务，并将 docker 服务设为自启动

systemctl start docker

systemctl start docker

查看 docker 服务状态,确保服务状态正常

systemctl status docker

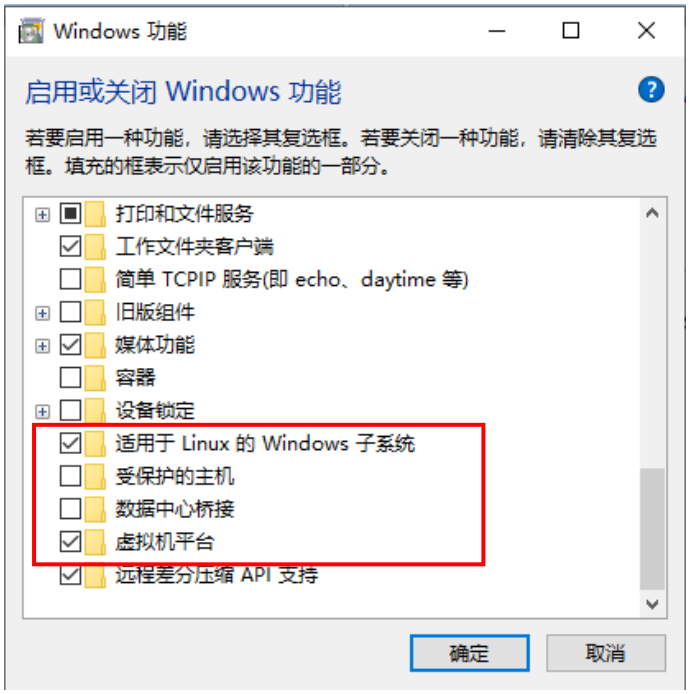
```
[root@docker197 yum.repos.d]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Wed 2022-01-12 14:27:10 CST; 1min 31s ago
     Docs: https://docs.docker.com
 Main PID: 3676 (dockerd)
   CGroup: /system.slice/docker.service
           └─3676 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Jan 12 14:27:09 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:09.252499737+08:00" level=info msg="ccResolverWrapper: s...e=grpc
Jan 12 14:27:09 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:09.252509129+08:00" level=info msg="ClientConn switching...e=grpc
Jan 12 14:27:09 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:09.302534584+08:00" level=info msg="Loading containers: start."
Jan 12 14:27:10 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:10.292804599+08:00" level=info msg="Default bridge (dock...dress"
Jan 12 14:27:10 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:10.467321727+08:00" level=info msg="Firewalld: interface...rning"
Jan 12 14:27:10 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:10.591170926+08:00" level=info msg="Loading containers: done."
Jan 12 14:27:10 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:10.632456262+08:00" level=info msg="Docker daemon" commi....10.12
Jan 12 14:27:10 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:10.632693244+08:00" level=info msg="Daemon has completed...ation"
Jan 12 14:27:10 docker197.localdomain systemd[1]: Started Docker Application Container Engine.
Jan 12 14:27:10 docker197.localdomain dockerd[3676]: time="2022-01-12T14:27:10.673502981+08:00" level=info msg="API listen on /var/r....sock"
Hint: Some lines were ellipsized, use -l to show in full.
```

2.3 window 环境下安装 docker

step 1:安装 wsl

docker destktop 依赖于 wsl（Windows Subsystem for Linux），是一个在 Windows 10 上能够原生运行 Linux 二进制可执行文件的兼容层。

打开控制面板→程序和功能→启用或关闭 windows 功能



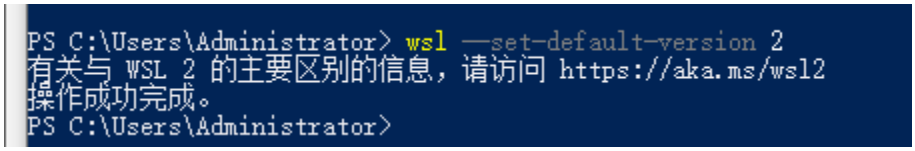
勾引上图红框中的两个选项。

step 2: 下载并安装 Linux kernel update package

https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi

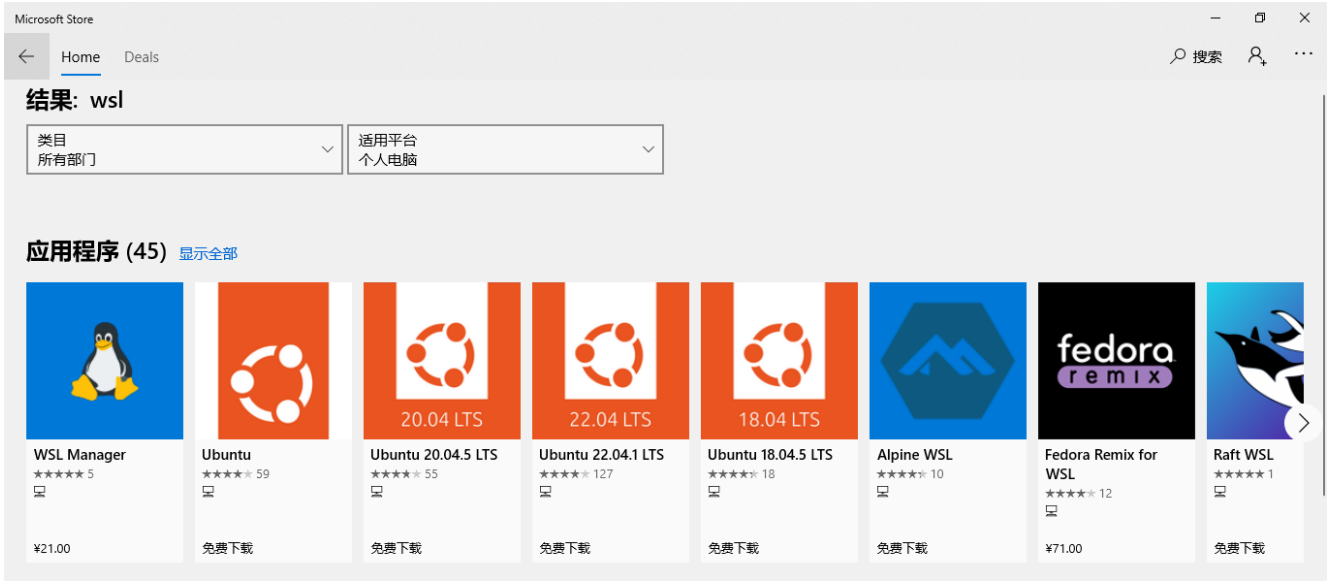
step 3: 将 WSL2 设定为默认版本

在 power shell 中执行：wsl --set-default-version 2



step 4:安装一个 linux 系统

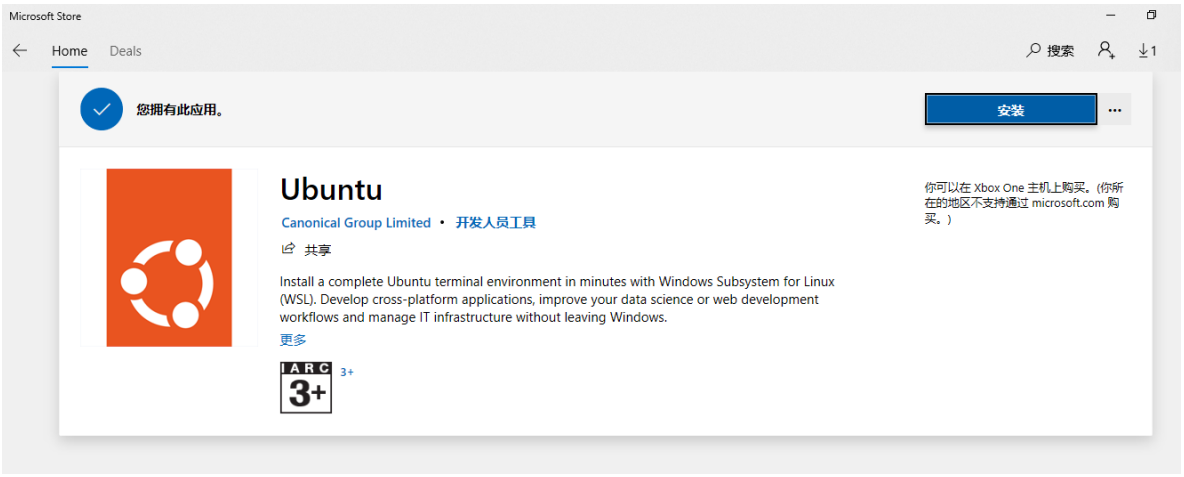
1） 打开 microsoft store,并搜索 wsl



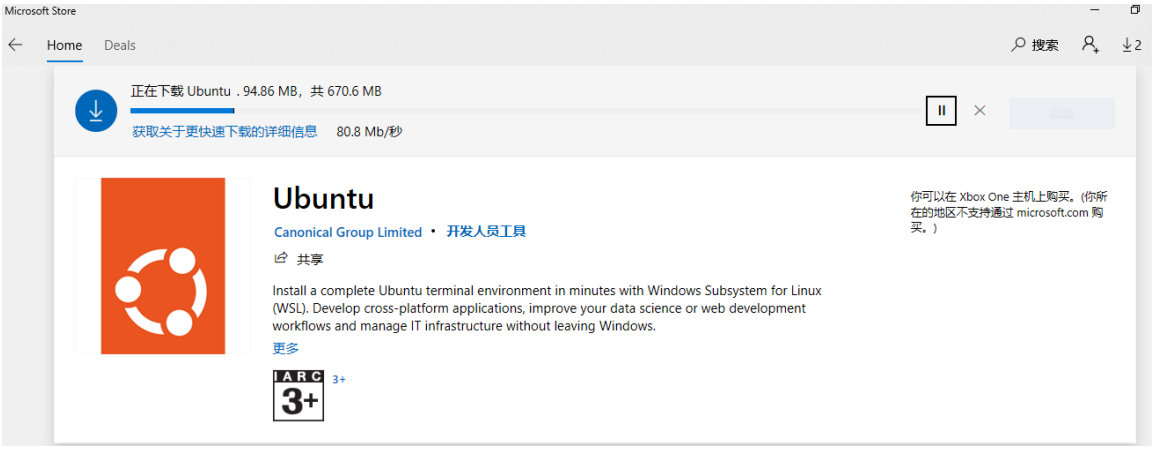
这里我们可以看到，有不同版本的 linux 系统可用，这里我们选择安装 ubuntu

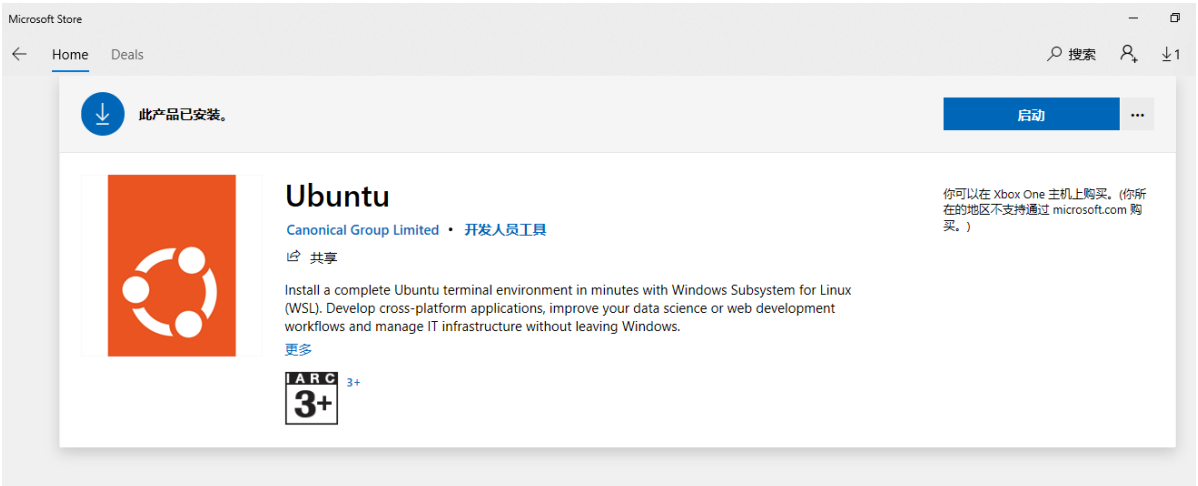
2)安装 ubuntu

点击上图中的 ubuntu 后，就进入到下图的界面。

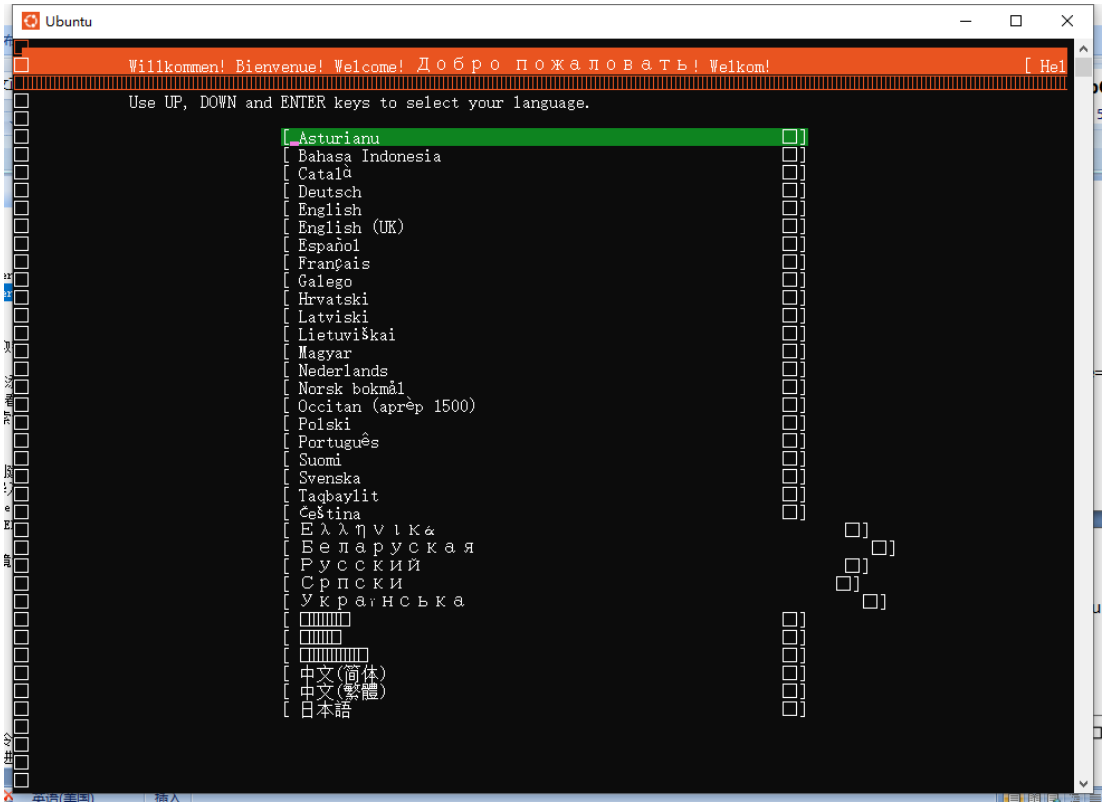


点击安装，就进入了安装界面。

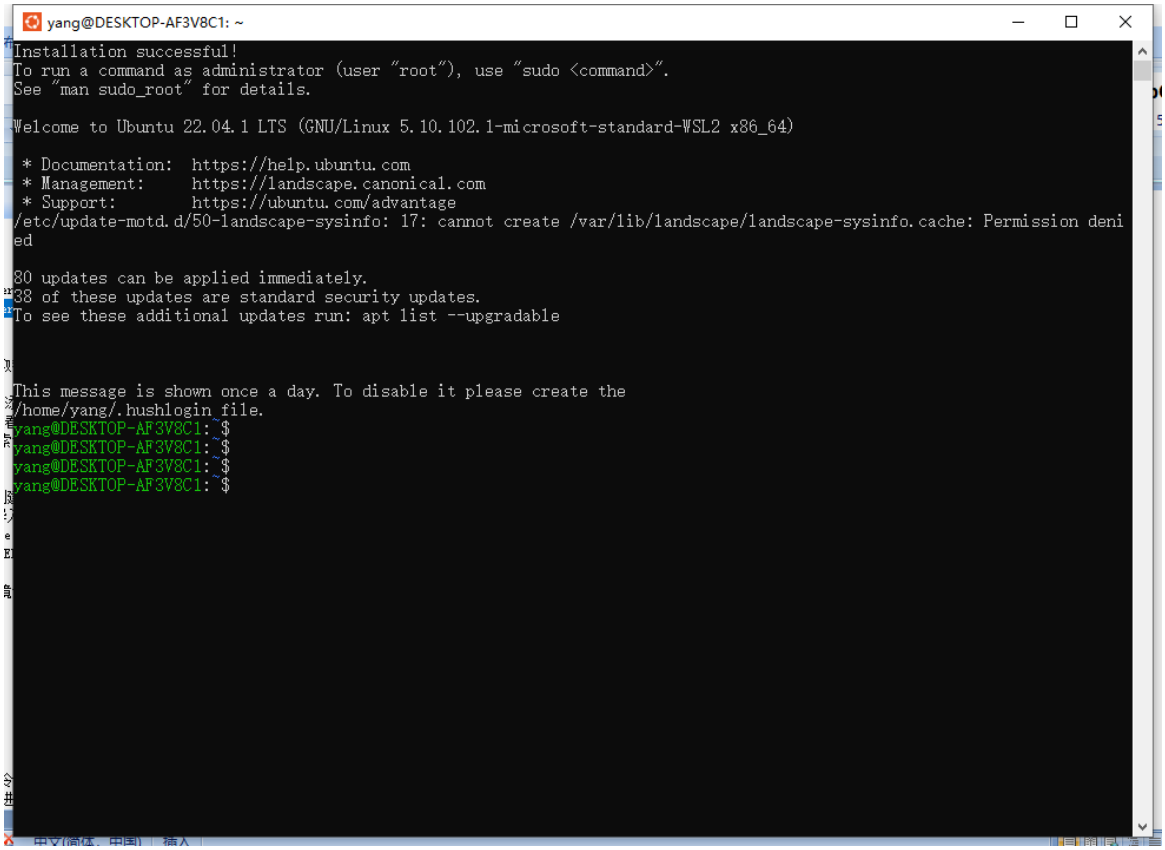




这里点击启动（或者在开始菜单里也会生成 ubuntu 图标），就可以进入 ubuntu 界面。



然后会让你做一些配置，根据提示来一步步往下就可以了。



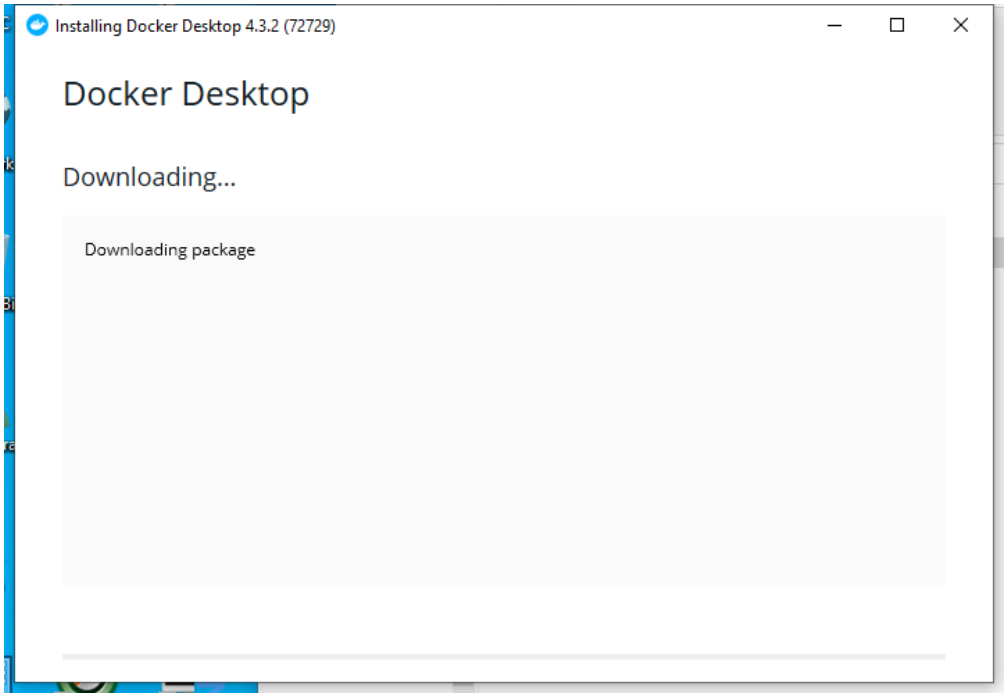
unbuntu 安装成功

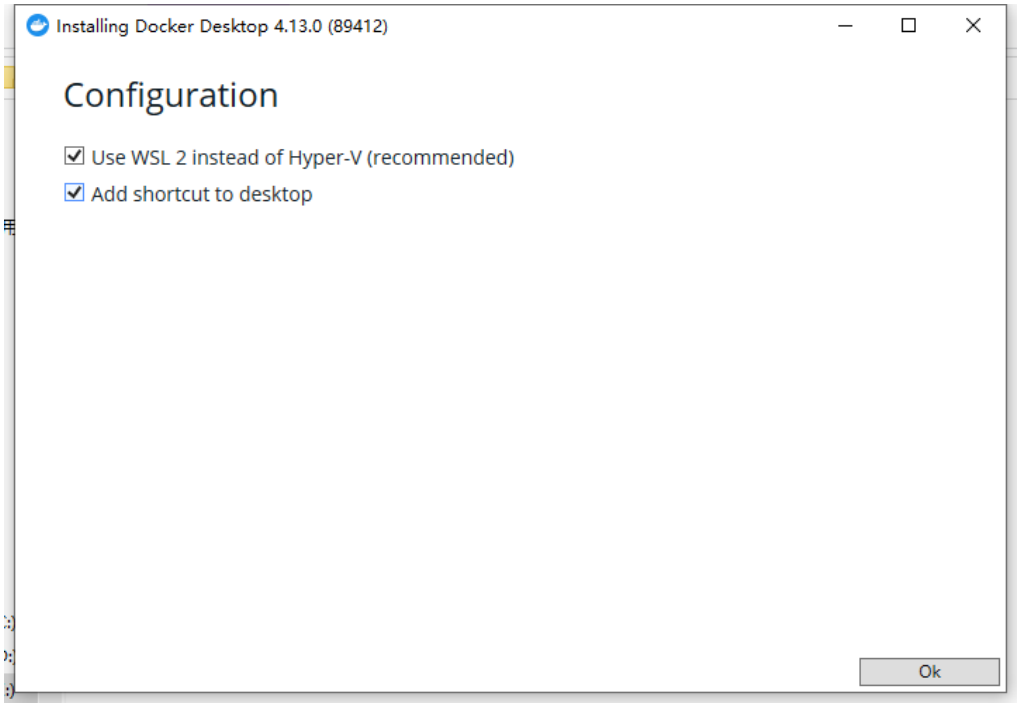
step 3 下载并安装 docker desktop

下载链接：

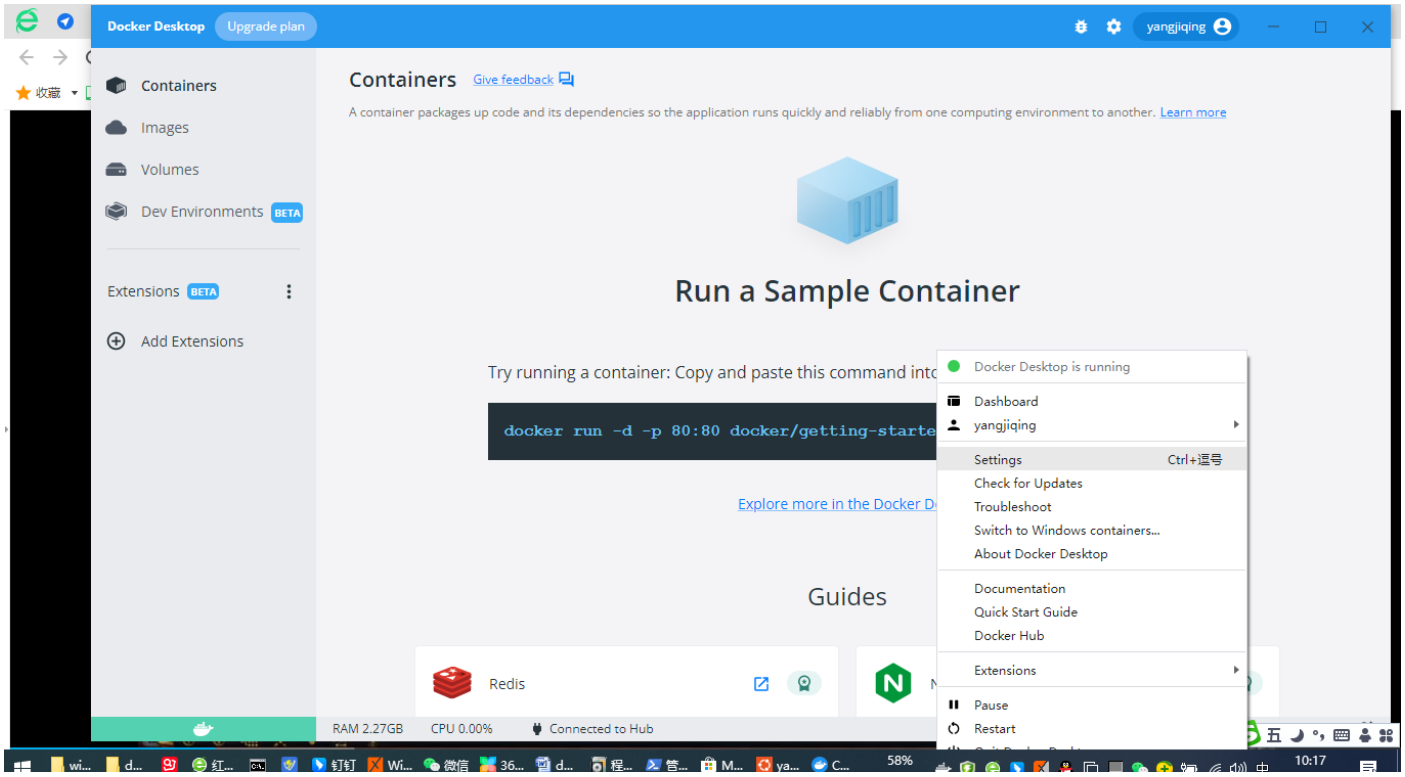
https://desktop.docker.com/win/main/amd64/Docker%20Desktop%20Installer.exe?utm_source=docker&utm_medium=webreferral&utm_campaign=dd-smartbutton&utm_location=module

点击安装：

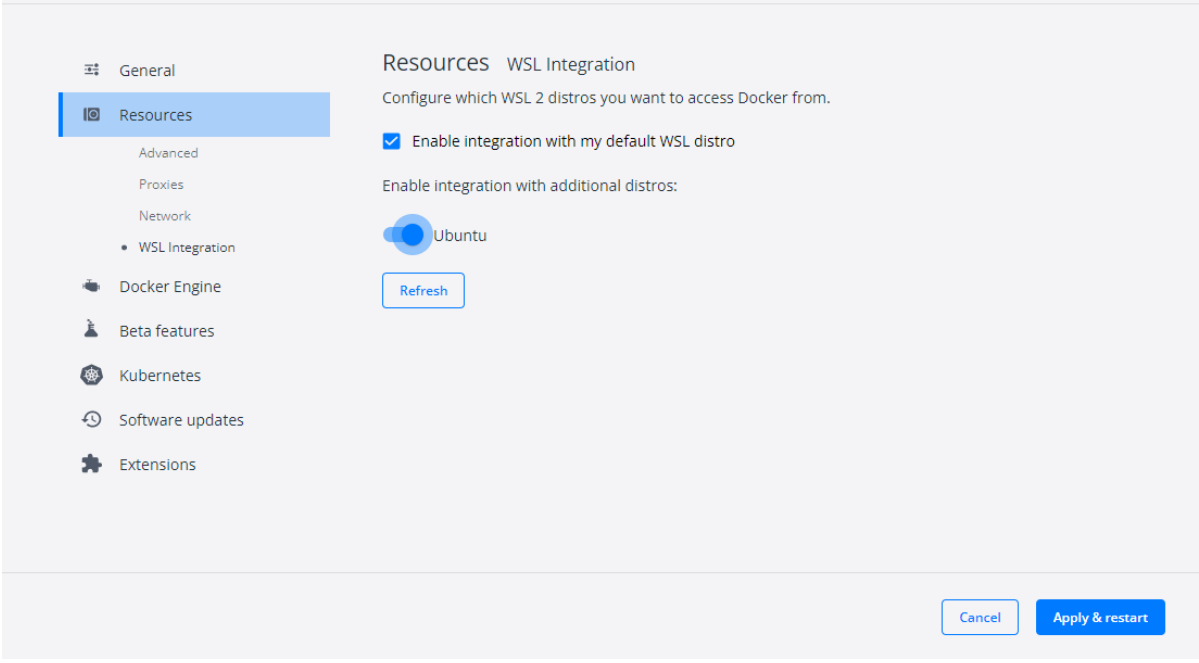




step 4 打开 docker desktop 并配置 wsl



右击任务栏上的 docker 图标，然后点 settings

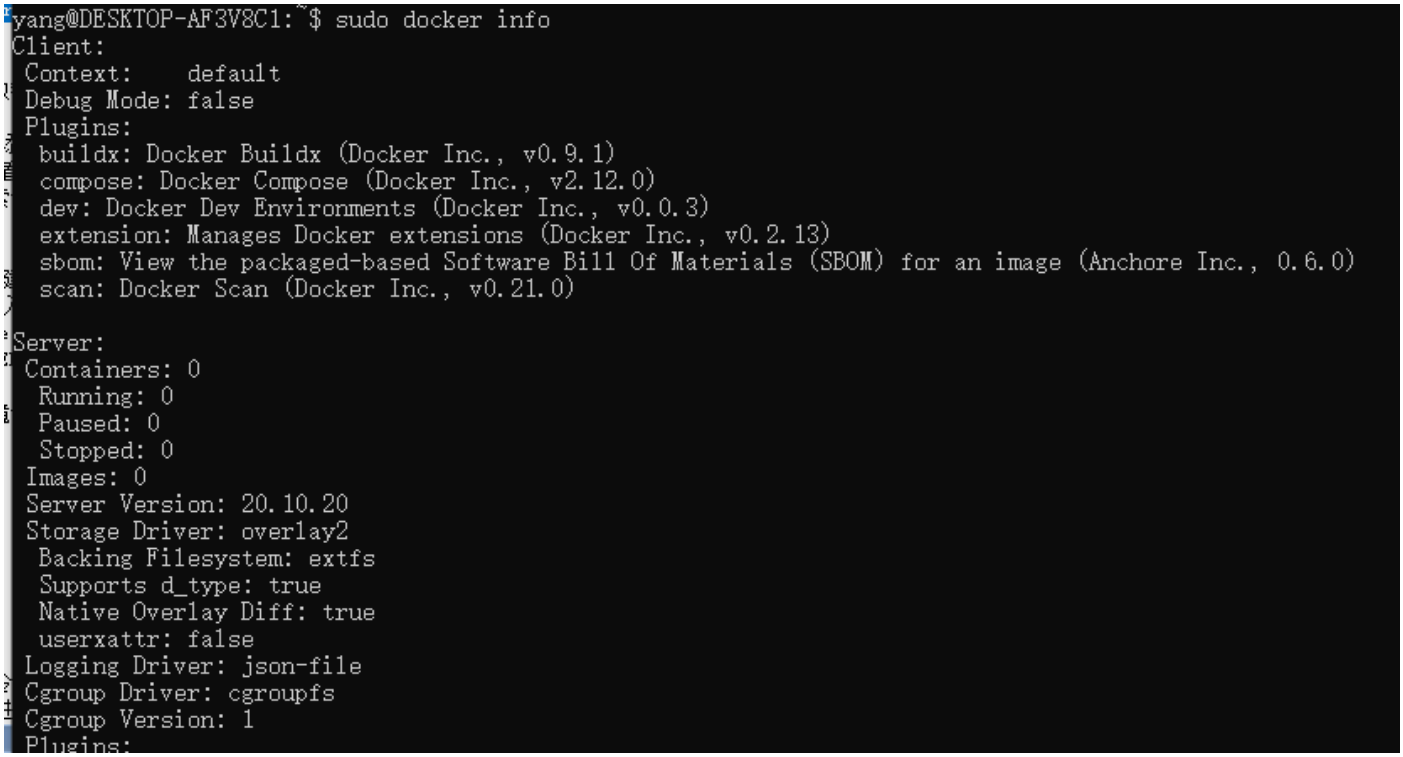


选择 Resources→WSL Integration

这里把 ubuntu（就是我们刚刚那的个 ubuntu）选上.

点击或下角的 apply&restart

这样我们就可以在这个 ubuntu 里（或在 windows 的 cmd 里）使用 docker 了。



三、docker 的基本配置

3.1docker 镜像的操作

Docker 容器运行前，需要本地存在对应的镜像，如果本地镜像不存在，docker 会尝试从默认镜像仓库下载（docker hub 公共注册服务器的仓库）。用户也可以创建自己的镜像仓库。

3.1.1 使用 pull 命令来获取镜像

docker pull <image> NAME:TAG

- ✧ image:镜像名称
- ✧ NAME:镜像仓库名称（如不指定，默认使用 docker hub 公共注册服务器的仓库）
- ✧ TAG:镜像标签（如不指定，默认使用最新版本 latest）

实验 1：从 docker hub 公共注册服务器的仓库获取一个 httpd 镜像的最新版本

docker pull httpd

完整格式：docker pull registry.hub.docker.com/httpd:latest

实验 2：查看本地镜像信息

docker image ls

```
[root@docker197 yum.repos.d]# docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
httpd        latest    a8ea074f4566   7 days ago    144MB
```

实验 3：启动第一个容器

docker run -it httpd bash

-i:保持标准输入

-t:分配一个 tty

```
[root@docker197 yum.repos.d]# docker run -it httpd bash
root@b727517090b4:/usr/local/apache2#
```

3.1.2 查看镜像信息

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

OPTIONS:选项

--all/-a:显示所有镜像，包括临时文件

----digests：显示哈稀摘要

-q:只显示 ID

-f/--filter:按条件过虑

dangling=true|false　只显示 tag 为空的镜像，默认为 false

before=[image[:tag]]显示此镜像创建之前就存在的镜像

since=[image[:tag]]显示此镜像创建之后存在的镜像

label=　根据标签进行过滤，，其中 lable 的值，是 docker 在编译的时候配置的或者在 Dockerfile 中配置的

reference=　：添加正则进行匹配（使用通配符进行匹配），如 reference=h*:latest,表示 h 开头，TAG 是 lattest 的镜像文件。

--format ：按照指定的格式显示

如：　--format "{{.Repository}}　{{.ID}}　{{.Tag}} {{.CreatedSince}}" ,则依次显示 Repostitory、ID、Tag、CreatedSince 这四个值

```
[root@docker197 ~]# docker images --format "{{.Repository}}  {{.ID}}  {{.Tag}} {{.CreatedSince}}"
httpd  a8ea074f4566  latest 8 days ago
```

显示镜像详细信息

docker image inspect IMAGE[:TAG]

```
[root@docker197 ~]# docker image inspect httpd:latest
[
    {
        "Id": "sha256:a8ea074f4566addcd01f9745397f32be471df4a4abf200f0f10c885ed14b1d28",
        "RepoTags": [
            "httpd:latest",
            "myhttpd:1.0"
        ],
        "RepoDigests": [
            "httpd@sha256:5cc947a200524a822883dc6ce6456d852d7c5629ab177dfbf7e38c1b4a647705"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2022-01-26T08:38:51.175633696Z",
        "Container": "7557c24c2cd3e5568d6a4f03288b6ab0dc5dd68829c7b7f5044f0531448ecafd",
        "ContainerConfig": {
            "Hostname": "7557c24c2cd3",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
```

3.1.3 使用 tag 命令给镜像添加标签

docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]

例如：docker tag httpd:latest myhttpd:1.0

给 httpd:latest 这个镜像添加一个名为 myhttpd:1.0 的标签

```
[root@docker197 ~]# docker tag httpd:latest myhttpd:1.0
[root@docker197 ~]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
httpd        latest    a8ea074f4566   8 days ago    144MB
myhttpd      1.0       a8ea074f4566   8 days ago    144MB
```

这两个标签的 ID 是相同的，也就是说这两个标签指向的镜像实际上是同一下。

3.1.4 使用 history 命令查看镜像文件的创建历史

```
[root@docker197 ~]# docker history httpd:latest
IMAGE          CREATED      CREATED BY                                      SIZE      COMMENT
a8ea074f4566   8 days ago   /bin/sh -c #(nop)  CMD ["httpd-foreground"]     0B
<missing>      8 days ago   /bin/sh -c #(nop)  EXPOSE 80                    0B
<missing>      8 days ago   /bin/sh -c #(nop) COPY file:c432ff61c4993ecd…   138B
<missing>      8 days ago   /bin/sh -c #(nop)  STOPSIGNAL SIGWINCH          0B
<missing>      8 days ago   /bin/sh -c set -eux;   savedAptMark="$(apt-m…   60.5MB
<missing>      8 days ago   /bin/sh -c #(nop)  ENV HTTPD_PATCHES=           0B
<missing>      8 days ago   /bin/sh -c #(nop)  ENV HTTPD_SHA256=0127f7dc…   0B
<missing>      8 days ago   /bin/sh -c #(nop)  ENV HTTPD_VERSION=2.4.52     0B
<missing>      8 days ago   /bin/sh -c set -eux;  apt-get update;  apt-g…   2.63MB
<missing>      8 days ago   /bin/sh -c #(nop) WORKDIR /usr/local/apache2    0B
<missing>      8 days ago   /bin/sh -c mkdir -p "$HTTPD_PREFIX"  && chow…   0B
<missing>      8 days ago   /bin/sh -c #(nop)  ENV PATH=/usr/local/apach…   0B
<missing>      8 days ago   /bin/sh -c #(nop)  ENV HTTPD_PREFIX=/usr/loc…   0B
<missing>      8 days ago   /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      8 days ago   /bin/sh -c #(nop) ADD file:90495c24c897ec479…   80.4MB
```

3.1.5 使用 search 命令搜索仓库中的镜像

docker search [OPTIONS] TERM

TERM:关键字

如：docker search http

在仓库中搜索带 http 关键字的镜像

```
[root@docker197 ~]# docker search http
NAME                              DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
httpd                             The Apache HTTP Server Project                  3860      [OK]
haproxy                           HAProxy - The Reliable, High Performance TCP…   1691      [OK]
sentry                            DEPRECATED; https://develop.sentry.dev/self-…   610       [OK]
caddy                             Caddy 2 is a powerful, enterprise-ready, ope…   340       [OK]
adoptopenjdk                      DEPRECATED; use https://hub.docker.com/_/ecl…   337       [OK]
steveltn/https-portal             A fully automated HTTPS server powered by Ng…   117                 [OK]
kennethreitz/httpbin              A simple HTTP service.                          90                  [OK]
hashicorp/http-echo               http-echo is an in-memory web server that ec…   45
centos/httpd-24-centos7           Platform for running Apache httpd 2.4 or bui…   41
mendhak/http-https-echo           Listens on http/https, echoes details about …   28                  [OK]
geldim/https-redirect             Very small http to https redirector based on…   20                  [OK]
citizenstig/httpbin               Docker container for httpbin: HTTP Request &…   16                  [OK]
bretfisher/httping                Ping with HTTP requests, build directly from…   16                  [OK]
alpine/httpie                     Auto-trigger docker build for `httpie` when …   16                  [OK]
clue/httpie                       HTTPie is a cURL-like tool for humans. Usefu…   14                  [OK]
kennship/http-echo                Echoes HTTP request info as JSON. Useful for…   3                   [OK]
aequitas/http-api-resource        Concourse resource to allow interaction with…   2                   [OK]
camptocamp/https-redirect         Docker image that just redirect http to https   2                   [OK]
lorands/http-put-resource         Concourse HTTP PUT resource                     1                   [OK]
shreddedbacon/http-directory-index Custom concourse resource type to check stat…  1                   [OK]
manageiq/httpd                    Container with httpd, built on CentOS for Ma…   1                   [OK]
scrapinghub/httpbin               HTTP Request & Response Service                 1                   [OK]
articulate/http-to-https          Simple container which just listens on port …   1                   [OK]
aequitas/http-resource            Concourse resource for fetching files from v…   0                   [OK]
cfje/http-resource                Http Concourse Resource                         0
```

OPTIONS:选项

-f/--filter:按条件过虑

is-automated=是否自动创建

is-official=是否官方发布

stars=星数

--format ：按照指定的格式显示

如： --format "{{.Description}} {{.StarCount }} {{.IsOfficial}} {{.IsAutomated}}" ,则依次显示镜像描述、星数、是否官方、是否自动创建这四个值

3.1.6 删除镜像

docker image rm/docker rmi [OPTION] IMAGE[:TAG]

-f/--force:强制删除，即使此镜像正在被某容器使用；

--no-prune：不删除无 TAG 的镜像

当一个镜像有多个标签的时候，删除任意一个 IMAGE[:TAG]，都不会把镜像真正删除。但一个镜像只有一个标签时，删除 IMAGE[:TAG]，就会把镜像删除。

3.1.7 清理临时镜像和未使用镜像

docker image prune[OPTIONS]

-a/--all：删除所有无用镜像

--filter:只清理符合过滤条件的镜像

-f/--force:强制清理

3.1.7 创建镜像

3.1.7.1 基于已有容器创建镜像

docker container commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

-a/--author:作者信息

-m,--message:提交一个信息

-p:创建时暂停容器运行

CONTAINER:容器 ID

实验一：基于已有容器 myhttpd:1.0,创建一个新的镜像

step 1 启动容器：docker run myhttpd:1.0 bash

step 2 在容器中新增一个文件

touch 1

```
root@39d1d270fe70:/usr/local/apache2# ls
1  bin  build  cgi-bin  conf  error  htdocs  icons  include  logs  modules
```

step 3 退出容器

exit

step4 创建一个新镜像 myhttpd:2.0

docker container commit -a "muyi" -m "add a new image" e5eedc31d230 myhttpd:2.0

```
[root@docker197 ~]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
myhttpd      2.0       b49de57f5610   15 seconds ago   144MB
myhttpd      1.0       a8ea074f4566   8 days ago       144MB
```

3.1.7.2 基于本地模板导入创建镜像

step 1 下载模版

https://wiki.openvz.org/Download/template/precreated

wget http://download.openvz.org/template/precreated/centos-7-x86_64.tar.gz

```
[root@docker197 templates]# wget http://download.openvz.org/template/precreated/centos-7-x86_64.tar.gz
--2022-02-04 11:23:31--  http://download.openvz.org/template/precreated/centos-7-x86_64.tar.gz
Resolving download.openvz.org (download.openvz.org)... 185.231.241.69
Connecting to download.openvz.org (download.openvz.org)|185.231.241.69|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 221597800 (211M) [application/x-gzip]
Saving to: 'centos-7-x86_64.tar.gz'

 8% [=======>                                                                ] 19,106,926   545KB/s  eta 3m 36s
```

step 2 从模版创建镜像

cat centos-7-x86_64.tar.gz | docker import - centos:1.0

```
[root@docker197 templates]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
centos       1.0       945cc60fc585   26 seconds ago   589MB
myhttpd      2.0       b49de57f5610   30 minutes ago   144MB
myhttpd      1.0       a8ea074f4566   8 days ago       144MB
[root@docker197 templates]#
```

3.1.7.3 基于 dockerfile 创建镜像

dockerfile 指令集

分类	指令	说明
	AGR	定义创建过程中使用的变量
	FROM	指定所创建镜像的基础镜像
	LABEL	为生成的镜像添加元数据标签信息
	EXPOSE	声明镜像内服务监听的端口
	ENV	指定环境变量
	ENTRYPOINT	指定镜像的默认入口命令
配置指令	VOLUME	创建一个数据卷挂载点
	USER	指定容器运行时的用户名或 UID
	WORKDIR	配置工作目录
	ONBUILD	创建子镜像时指定自动执行的操作指令
	STOPSIGNAL	指定退出的信号值
	HEALTHCHECK	配置所启动容器如何进行健康检查
	SHELL	指定默认 shell 类型
	RUN	运行指定命令
操作指令	CMD	启动容器时默认执行的命令
	ADD	添加内容到镜像
	COPY	复制内容到镜像

通过 docker build 命令创建镜像

docker build [OPTIONS] PATH | URL | -

options:

选项	说明
--add-host HOST:IP	添加一个自定义的主机到 IP 的映射
--build-arg list	添加创建时的变量
--cache-from strings	使用指定镜像做为缓存源
--cgroup-parent string	继承的上层 cgroup
--compress	使用 gzip 压缩创建时的上下文数据
--cpu-period int	CFS 调度器时长限制

--cpu-quota int　　CFS 调度器份额限制

-c, --cpu-shares int　　CPU 权重

--cpuset-cpus string　　CPU 数量(0-3, 0,1)

--cpuset-mems string　　多 CPU 允许使用的内存数量

--disable-content-trust　　忽略镜像验证，默认为需要验证

-f, --file string　　Dockerfile 文件路径和名称，默认为当前目录下的./dockerfile

--force-rm　　总是删除中间过程的容器

--iidfile string　　将镜像 ID 写入到文件

--isolation string　　容器隔离机制

--label list　　镜像元数据

-m, --memory bytes　　内存使用限制

--memory-swap bytes　　内存和交换区总限制

--network string　　指定使用 run 命令启动容器时的网络模式，默认为 default

--no-cache　　创建镜像时不使用缓存

--pull　　总试尝试获取镜像的最新版本

-q, --quiet　　不打印创建过程中的日志信息

--rm　　创建成功后删除中间容器，默认为 true

--security-opt strings　　安全选项

--shm-size bytes　　/dev/shm 的大小

-t, --tag list　　指定镜像的标签列表，name:tag

--target string　　指定创建的目标阶段

--ulimit ulimit　　Ulimit 操作

实验一：使用 dockerfile 创建一个运行 nginx 的容器

step 1　创建 dockerfile

vim dockerfile　　#在当前目录下创建一个文本文件，命名为 dockerfile

以下是文件的内容

```
FROM centos:7.0                    #指定一个基础镜像：centos:7.0
LABEL nginx 1.0                    #这边可以自定义一些版本信息，作者信息等
EXPOSE 80                          #说明容器开放的端口号
WORKDIR /root                      #指定容器的工作目录
RUN    yum install -y epel-release.noarch \        #安装 yum epel 源
```

&&yum install -y nginx \ #安装 nginx

&& rm -rf /var/cache/yum/* #删除 yum 缓存

CMD ["nginx","-g","daemon off;"] #在前台启动 nginx,且不让它成为 daemon(守护程序)

step 2 使用刚刚创建的 dockerfile 来创建镜像

docker build –t nginx:1.0 -f ./dockerfile .

```
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Updating   : httpd-tools-2.4.6-97.el7.centos.4.x86_64                    1/4
  Updating   : httpd-2.4.6-97.el7.centos.4.x86_64                          2/4
  Cleanup    : httpd-2.4.6-40.el7.centos.4.x86_64                          3/4
  Cleanup    : httpd-tools-2.4.6-40.el7.centos.4.x86_64                    4/4
  Verifying  : httpd-tools-2.4.6-97.el7.centos.4.x86_64                    1/4
  Verifying  : httpd-2.4.6-97.el7.centos.4.x86_64                          2/4
  Verifying  : httpd-tools-2.4.6-40.el7.centos.4.x86_64                    3/4
  Verifying  : httpd-2.4.6-40.el7.centos.4.x86_64                          4/4

Updated:
  httpd.x86_64 0:2.4.6-97.el7.centos.4

Dependency Updated:
  httpd-tools.x86_64 0:2.4.6-97.el7.centos.4

Complete!
Removing intermediate container 1835e17de5f5
 ---> aa822482f129
Step 7/7 : CMD "httpd -DFOREGROUND"
 ---> Running in b77f64182f40
Removing intermediate container b77f64182f40
 ---> 68aea2ef43ff
Successfully built 68aea2ef43ff
Successfully tagged yangjiqing/httpd:1.0
```

创建成功

```
^C[root@docker197 nginx]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
nginx        1.0       bc15b7737ffc   2 hours ago    627MB
centos       7.0       a9729bb0e1d6   4 hours ago    589MB
```

镜像文件已经生成

step 3 在后台启动容器，并将宿主机的 8080 端口映射到容器的 80 端口。

docker run -d --name=nginx –p 8080:80 nginx:1.0

查看容器信息：

docker inspect nginx

```
[root@docker197 nginx]# docker inspect nginx
[
    {
        "Id": "0d564397054afa5c6bee6b3de22d3a262c43afec3866ce1a21beb95a3320dbe6",
        "Created": "2022-10-30T03:41:25.688891867Z",
        "Path": "nginx",
        "Args": [
            "-g",
            "daemon off;"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 10766,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2022-10-30T03:41:31.442528586Z",
            "FinishedAt": "2022-10-30T03:41:27.728631204Z"
        },
```

成功启动。

进入容器看一下

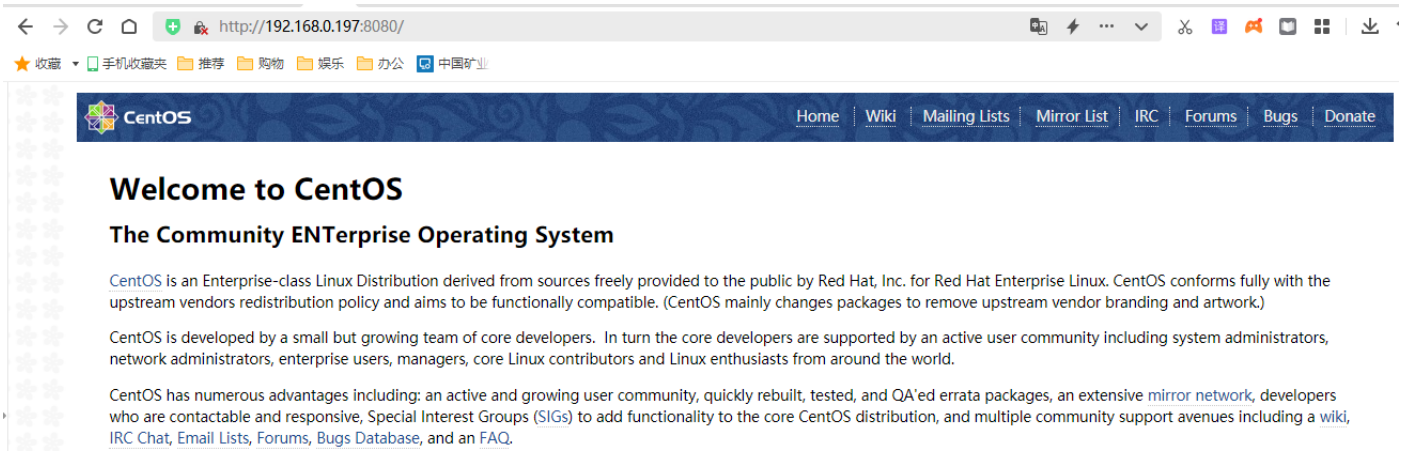
docker exec –it nginx bash

```
[root@0d564397054a ~]# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.3  39252  3908 ?        Ss   Oct29   0:00 nginx: master process nginx -g daemon off;
nginx         6  0.0  0.1  39640  1796 ?        S    Oct29   0:00 nginx: worker process
root         24  2.5  0.1  11772  1804 pts/0    Ss   02:00   0:00 bash
root         40  0.0  0.1  47420  1676 pts/0    R+   02:00   0:00 ps aux
```

nginx 服务已经进来了。

```
[root@httpd ~]# ll /var/www/html/
total 4
-rw-r--r--. 1 root root 10 Feb 18 03:51 index.html
[root@httpd ~]#
```

看看能否打开网页



是可以打开的，这样这个镜像就算是创建成功了。

2.1.7.4 dockerfile 中 ENTRYPONT 和 CMD 命令的区别

ENTRYPONT:

指定镜像的默认入口命令，该入口命令会在启动容器时作为根命令执行，所有传入值作为该命令的参数。

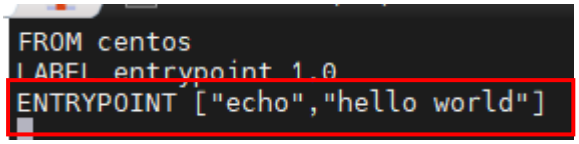
支持两种格式：

- ENTRYPOINT["executable"，"param1"，"param2"]：exec 调用执行；
- ENTRYPOINT command param1 param2：shell 中执行。

每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个生效。

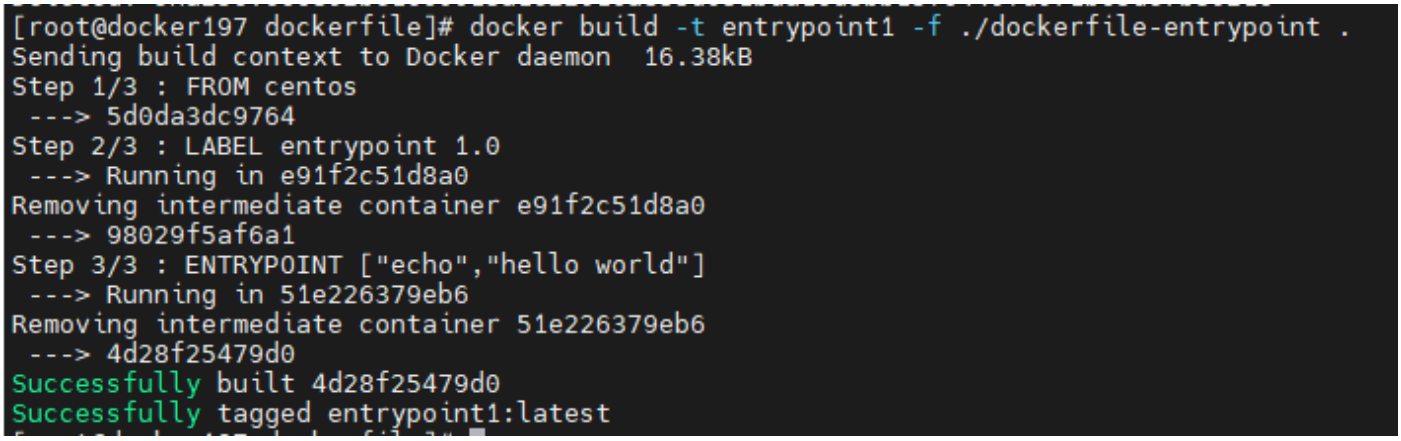
在运行时，可以被--entrypoint 参数覆盖掉，如 docker run --entrypoint。

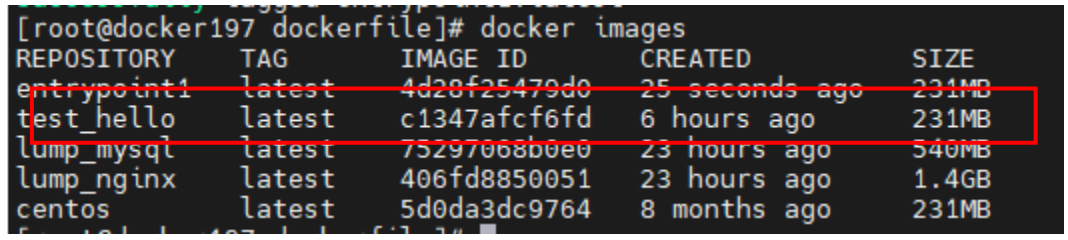
实验 1:



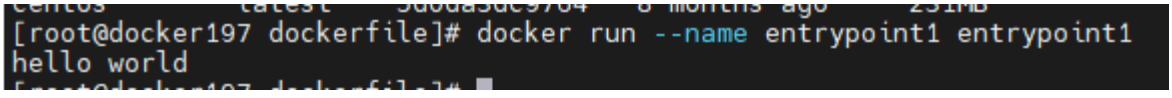
这里指定了 entrypoint 的命令是 echo hello world

Build 一下





我们用这个镜像创建一个容器看看效果



这里会显示 hello world,说明 entrypoint 指令运行正常.

我们再看一下另一种格式

```
FROM centos
LABEL entrypoint 1.0
ENTRYPOINT echo hello world
```

```
[root@docker197 dockerfile]# docker run --name entrypoint2 entrypoint2
hello world
```

发现效果是一样的.

CMD 命令

CMD 指令用来指定启动容器时默认执行的命令。

支持三种格式：

- CMD["executable"，"param1"，"param2"]：相当于执行 executable param1 param2，推荐方式；
- CMD command param1 param2：在默认的 Shell 中执行，提供给需要交互的应用；
- CMD["param1"，"param2"]：提供给 ENTRYPOINT 的默认参数。

每个 Dockerfile 只能有一条 CMD 命令。如果指定了多条命令，只有最后一条会被执行。

如果用户启动容器时候手动指定了运行的命令(作为 run 命令的参数),则会覆盖掉 CMD 指定的命令。

第一和第二种格式效果是一样的,我们就用第二种格式来做实验:

实验 2:

```
FROM centos
LABEL entrypoint 1.0
CMD echo hello world
```

```
[root@docker197 dockerfile]# docker run --name cmd1 cmd1
hello world
```

这个效果跟 entrypoint 是一样的

实验 3:ENTRYPOINT 和 CMD 命令同时存在

```
FROM centos
LABEL entrypoint+cmd 1.0
ENTRYPOINT echo hello entrypoint
CMD echo cmd
```

再看看效果

```
[root@docker197 dockerfile]# docker run --name entrypointcmd1 entrypointcmd1
hello entrypoint
```

这里发现只执行了 ENTRYPOINT 指令,CMD 指令不会被执行.

实验 4:将 CMD 指令的参数提供给 ENTRYPOINT 使用

```
FROM centos
LABEL entrypoint+cmd 1.0
ENTRYPOINT ["echo"]
CMD ["hello world"]
```

这里 CMD 的参数”hello world”会被传递给 ENTRYPOINT.

```
[root@docker197 dockerfile]# docker run --name entrypointcmd2 entrypointcmd2
hello world
```

这个效果跟实验 1 是样的.

3.1.8 导出导入镜像

导出镜像

docker save [OPTIONS]IMAGE[:TAG]

-o：导出的文件路径和文件名

例：将 ftp:1.0 这个镜像导出为 ftp.tar 文件

docker save -o ftp.tar ftp:1.0

这样就可以将这个文件分享给别人使用了。

导入镜像

docker load -i filepath

-i:导入的文件路径和文件名

例：将 ftp.tar 导入到本机

docker load –i ftp.tar

```
[root@docker197 templates]# docker load -i ftp.tar
Loaded image: ftp:1.0
[root@docker197 templates]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED              SIZE
ftp          1.0       a297bef48210   12 minutes ago       750MB
centos       1.0       945cc60fc585   31 minutes ago       589MB
myhttpd      2.0       b49de57f5610   About an hour ago    144MB
myhttpd      1.0       a8ea074f4566   8 days ago           144MB
```

3.1.9 使用 push 命令上传镜像

step 1 注册帐号

到这个网站 https://hub.docker.com/注册账号

step 2 登入

docker login -u USERNAME -p PASSWORD

```
[root@docker197 templates]# docker login -u yangjiqing -p
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

step 2 上传镜像

上传的镜像名称必须将你的用户加上

docker tag ftp:1.0 yangjiqing/ftp:1.0

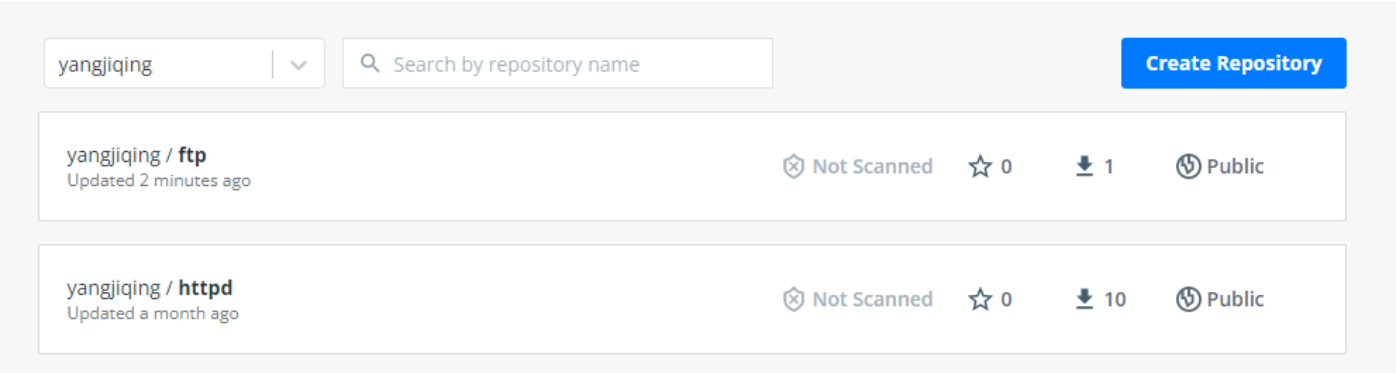
将 ftp:1.0 这个镜像加一个标签，变成 username/ftp:1.0 。这样才可以。

```
REPOSITORY          TAG       IMAGE ID       CREATED             SIZE
yangjiqing/ftp      1.0       a297bef48210   About an hour ago   750MB
ftp                 1.0       a297bef48210   About an hour ago   750MB
centos              1.0       945cc60fc585   About an hour ago   589MB
myhttpd             2.0       b49de57f5610   2 hours ago         144MB
myhttpd             1.0       a8ea074f4566   8 days ago          144MB
```

docker push [OPTIONS] NAME[:TAG]

```
[root@docker197 templates]# docker push yangjiqing/ftp:1.0
The push refers to repository [docker.io/yangjiqing/ftp]
efdee619b87c: Pushed
25710d453e5e: Pushed
1.0: digest: sha256:a676e145621af7c9d5a952dade94b8302233f030f4a928adf99e006b6a5096cd size: 742
```

完成上传，到这个网站上你可看到自己上传的资源。



上传完以后，就可以通过 3.1.1 小节的方式，用 pull 命令来拉取这个镜像了。

3.2docker 容器的操作

3.2.1 创建容器

docker container create [OPTIONS] IMAGE [COMMAND] [ARG...]

Options	说明
与容器运行模式相关的 options	
-a,--attach stderr|stdin|stdout	是否绑定到标准错误、标准输入、标准输出
--entrypoint=”“	镜像存在入口命令时，覆盖为新的命令
--expose=[ ]	指定容器暴露出来的端口，或端口范围
--group-add[ ]	运行容器的用户组
-i,--interactive=true|false	保持标准输入打开，默认为 false
--ipc=”“	容器 IPC 命名空间，可以为其它容器或主机
--isolation=”default”	容器使用的隔离机制
--log-driver=””	指定容器的日志驱动类型，可以为 json-file、syslog、journald、gelf、fluentd、awslogs、splunk、etwlogs、gcplogs、none
--log-opt=[ ]	传递给日志驱动的选项
--net=””	指定容器的网络模式，bridge、none、从其它容器内网络、host 的网络、某个现有网络
--net-alias=[ ]	容器在网络中的别名
-P，--publish-all=true|false	通过 NAT 机制将容器标记暴露的端口自动映射到本地主机的临时端口
-p,--publish=[ ]	指定如何映射到本地主机端口，例如：-p 1111-2222:21111-22222
--pid=host	容器的 PID 命名空间
--userns=”“	启用 userns-remap 时配置用户命名空间的模式
--uts=host	容器的 UTS 命名空间
--restart=”no”	容器的重启策略，包括 no,failure[:max-retry],always,unless-stoped 等
--rm=true|false	容器退出后是否自动删除，不能跟-d 同时使用

-t,--tty=true|false	是否分配一个伪终端，默认为 false
--tmpfs=[ ]	挂载临时文件系统到容器
-v,--volume[=[[HOST-DIR:]CONTAINER-DIR[:OPTIONS]]]	挂载主机上的文件系统到容器
--volume-driver=”“	挂载文件卷的驱动类型
--volumes-from=[ ]	从其它容器挂载卷
-w,--workdir=”“	容器内的默认工作目录
与容器环境和配置相关的 option	
--add-host=[ ]	在容器内添加一个主机名到 IP 地址的映射关系，通过/etc/hosts 文件。
--device=[ ]	映射物理机上的设备到容器内。
--dns-search=[ ]	DNS 搜索域
--dns-opt=[ ]	自定义的 DNS 选项
--dns=[ ]	自定义的 DNS 服务器
-e,--env=[ ]	指定容器内的环境变量
--env-file=[ ]	从文件中读取环境变量到容器内
-h,--hostname=”“	指定容器内的主机名
--ip=”“	指定容器的 ipv4 地址
--ip6=”“	指定容器的 ipv6 地址
--link=[<name or id>:alias]	链接到其它容器
--link-local-ip=[ ]	容器的本地链接地址列表
--mac-address=”“	指定容器的 MAC 地址
--name=”“	指定容器的别名
与容器资源限制和安全保护相关的 option	
--blkio-weight=10-1000	容器读写块设备的 I/O 性能权重，默认为 0
--blkio-weight-device=[DEVICE_NAME:WEIGHT]	指定各个块设备的 I/O 性能权重
--cpu-shares=[ ]	允许容器使用 CPU 资源的相对权重，默认一个容器能用满一个核的 CPU
--cap-add=[ ]	增加容器的 linux 指定安全能力
--cap-drop=[ ]	移除容器的 linux 指定安全能力
--cgroup-parent=”“	容器 cgroup 限制的创建路径

--cidfile=”“	指定容器的进程 ID 写入到文件
--cpu-period=0	限制容器在 CFS 调度器下的 CPU 占用时间片
--cpuset-cpus=”“	限制容器能使用哪些 CPU 核
--cpuset-mems=”“	NUMA 架构下使用哪些核心的内存
--cpu-qouta=0	限制容器在 CFS 调度器下的 CPU 配额
--device-read-bps=[ ]	挂载设备的读吞吐率（以 bps 为单位）限制
--device-write-bps=[ ]	挂载设备的写吞吐率（以 bps 为单位）限制
--device-read-iops=[ ]	挂载设备的读吞吐率（以每秒 i/o 次数为单位）限制
--device-write-iops=[ ]	挂载设备的写吞吐率（以每秒 i/o 次数为单位）限制
--health-cmd=”“	指定检查容器健康状态的命令
--health-interval=0s	执行健康检查的间隔时间，单位可以为 ms,s,m 或 h
--health-retries=int	健康检查失败重度次数，超过则认为不健康
--health-start-period=0s	容器启动后进行健康检查的等待时间，单位可以为 ms,s,m 或 h
--health-timout=0s	健康检查的执行超时，单位可以为 ms,s,m 或 h
--no-healthcheck=true|false	是否禁用健康检查
--init	在容器中执行一个 init 进程，来负责响应信号和处理僵尸状态子进程
--kernel-memory=”“	限制容器使用内核内存的大小，单位可以是 b,k,m,g
-m，--memory=”“	限制容器内应用使用的内存，单位可以是 b,k,m,g
--memory-reservation=”“	当系统中内存过低时，容器会被强制限制内存到给定值，默认情况下为内存限定值
--memory-swap=”LIMIT”	限制容器内存和交换区的总大小
--oom-kill-disable=true|false	内存耗尽时是否杀死容器
--oom-score-adj=”“	调整容器的内存耗尽参数
--pids-limit=”“	限制容器的 PID 个数
--privileged=true|false	是否给容器高权限，意味着容器内应用不受权限的限制，一般不推荐
--read-only=true|false	是否让容器内的文件系统只读
--security-opt=[ ]	指定一些安全参数，包括权限、安全能力、apparmor 等
--stop-signal=SIGTERM	指定停止容器的系统信号

--shm-size=”“	/etc/shm 的大小
--sig-proxy=true|false	是否代理收到的信号给应用，默认为 true,不能代理 SIGCHLD、SIGSTOP、SIGKILL 信号
--memory-swappinness=”0~100”	调整容器的内存交换区参数
-u,--user=”“	指定容器内执行命令的用户信息
--userns=”“	指定用户命名空间
--ulimit=[ ]	通过 ulimit 来限制最大文件数、最大进程数等
其它	
-l,--lable=[ ]	以键值对方式指定容器的标签信息
--lable-file=[ ]	从文件中读取标签信息

实验 1：使用 centos:7.0 这个镜像，创建一个别名为 centos 的容器。并且让执行一个”sleep 10”的命令，这个命令的意思是让当前的 shell 休眠 10 秒

docker container create --name centos1 centos:7.0 sleep 10

查看容器状态，已成功创建了一个容器。

docker container ps –a

```
[root@docker197 ~]# docker create --name centos1 centos:7.0 sleep 10
4114a1b40028690d485e5810088f91bac1e57616e397979490f73ef3bb6880a8
[root@docker197 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[root@docker197 ~]# docker ps -a
CONTAINER ID   IMAGE         COMMAND       CREATED          STATUS    PORTS     NAMES
4114a1b40028   centos:7.0    "sleep 10"    14 seconds ago   Created             centos1
```

3.2.2 启动容器

docker container start [OPTIONS] CONTAINER [CONTAINER...]

options:

-a,--attach 绑定到标准错误、标准输出和转发信号

-i,--interactive 绑定容器的标准输入（可对容器进行操作）

实验 1：启动刚刚创建的容器 centos，并观察容器的状态

docker container start centos

```
[root@docker197 ~]# docker ps
CONTAINER ID   IMAGE          COMMAND       CREATED              STATUS              PORTS     NAMES
4114a1b40028   centos:7.0     "sleep 10"    About a minute ago   Up 1 second                   centos1
```

此时，容器状态已变成“up”了。

重启容器

docker container restart [OPTIONS] CONTAINER [CONTAINER...]

options:

-t,--time 一段时间以后重启容器

3.2.3 创建并启动容器

相当于将创建和启动容器合并为一个步骤了。

docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]

实验 1：利用镜像 centos 创建并启动一个容器，并允许交互。

docker run -it --name=centos centos:1.0 bash

```
CONTAINER ID   IMAGE          COMMAND                  CREATED              STATUS                          PORTS     NAMES
3a0fd6dd6b9f   centos:1.0     "bash"                   11 seconds ago       Exited (0) 4 seconds ago                  centos
7d200a583a14   centos:1.0     "bash --name=centos"     29 seconds ago       Exited (2) 29 seconds ago                 condescending_fermi
b52d3e31e37b   centos:1.0     "bash"                   About a minute ago   Exited (0) About a minute ago             silly_hermann
a21c029fd32f   ftp:1.0        "bash"                   6 days ago           Exited (0) 6 days ago                     compassionate_davinci
e181671d5910   centos:1.0     "bash"                   6 days ago           Exited (0) 6 days ago                     recursing_wiles
18e4e4155d16   myhttpd:2.0    "bash"                   6 days ago           Exited (0) 6 days ago                     nice_shamir
e5eedc31d230   myhttpd:1.0    "bash"                   6 days ago           Exited (0) 6 days ago                     optimistic_turing
39d1d270fe70   myhttpd:1.0    "bash"                   6 days ago           Exited (0) 6 days ago                     musing_davinci
b1fe7bfde18b   myhttpd:1.0    "bash"                   6 days ago           Exited (0) 6 days ago                     upbeat_visvesvaraya
39507b651a9a   myhttpd:1.0    "bash"                   6 days ago           Exited (127) 6 days ago                   zealous_yonath
09f0b1a2c7b1   myhttpd:1.0    "httpd-foreground"       6 days ago           Exited (0) 6 days ago                     strange_pare
a0f8de463979   a8ea074f4566   "bash"                   7 days ago           Exited (129) 7 days ago                   xenodochial_northcutt
68eafa4b8426   a8ea074f4566   "bash"                   7 days ago           Exited (0) 7 days ago                     ecstatic_austin
b727517090b4   a8ea074f4566   "bash"                   7 days ago           Exited (127) 7 days ago                   admiring_
d9f9883c9c13   a8ea074f4566   "bash"                   7 days ago           Exited (0) 7 days ago                     silly_proskuriakova
```

这时如果按 Ctrl+d 或 exit 退出容器后，容器会自动关闭。这是因为，当容器内的应用退出后，容器的使命已经完成，没有继续运行的必要了。

实验 2 ：让容器在保持在后台运行

docker run -d --name=centos-5sleep 9999999

加-d 参数的意义是让容器保持在后台运行

```
[root@docker197 ~]# docker ps -a
CONTAINER ID   IMAGE          COMMAND              CREATED          STATUS          PORTS     NAMES
8f9a9797e6c5   centos:1.0     "sleep 999999"       5 seconds ago    Up 4 seconds              centos-5
```

发现容器已经在后台运行

或者也可以，进入容器后，按 ctrl+p+q 来“退出“容器（而不是用 ctrl+z 或 exit 命令）

do

```
[root@docker197 ~]# docker run -it --name=centos-7 centos:1.0 bash
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]#
[root@467876eae467 /]# [root@docker197 ~]#
[root@docker197 ~]# docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                      PORTS     NAMES
467876eae467   centos:1.0     "bash"                   17 seconds ago   Up 16 seconds                         centos-7
```

3.2.4 查看容器输出

docker container logs [OPTIONS] CONTAINER

--details 查看详细信息

-f,--follow 保持持续输出

--since string 输出从某个时间点开始的信息

-t, --timestamps 显示时间戳信息

-n, --tail string 显示最近的信息

--until string 输出某个时间点之前的信息

3.2.5 暂停和恢复容器

暂停容器

docker container pause CONTAINER [CONTAINER...]

```
[root@docker197 ~]# docker ps -a | grep centos-7
467876eae467   centos:1.0     "bash"                   24 hours ago   Up 34 seconds (Paused)              centos-7
[root@docker197 ~]#
```

恢复暂停的容器

docker container unpause CONTAINER [CONTAINER...]

```
[root@docker197 ~]# docker container unpause centos-7
centos-7
[root@docker197 ~]# docker ps -a | grep centos-7
467876eae467   centos:1.0     "bash"                   24 hours ago   Up About a minute                   centos-7
[root@docker197 ~]#
```

3.2.6 终止容器

docker container stop [OPTIONS] CONTAINER [CONTAINER...]

options:

-t, --time 等待一段时间后终容器

```
[root@docker197 ~]# docker container stop -t 10 centos-7
centos-7
[root@docker197 ~]# docker ps -a | grep centos-7
467876eae467   centos:1.0     "bash"                24 hours ago   Exited (137) 10 seconds ago            centos-7
[root@docker197 ~]#
```

强行终止容器

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

3.2.7 删除容器

docker container rm [OPTIONS] CONTAINER [CONTAINER...]

options:

-f, --force=ture|false 是否强制终止容器，相当于使用 sigkill

删除已停止的容器

docker container prune [OPTIONS]

```
[root@docker197 ~]# docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
e3a30e952f152837697cf3cd55e42c943177ff68c43e0c7c4fd3e4e073be799c
0363a55b069f2e78f4363b50433ac5c615850c78db07894a349321d55789c8ab
633d6de74f3f64e36b832c72084469b1713244bccff24439da0b507977bde8b1
```

```
[root@docker197 ~]# docker ps -a
CONTAINER ID   IMAGE         COMMAND   CREATED         STATUS              PORTS     NAMES
467876eae467   centos:1.0    "bash"    25 hours ago    Up About a minute             centos-7
[root@docker197 ~]#
```

处于停止状态的容器都被删除了

3.2.8 进入容器

3.2.8.1 使用 attach 命令进入容器

docker container attach [OPTIONS] CONTAINER

options:

--detach-keys[=[]]指定退出 attach 模式的快捷键，默认是 Ctrl+p+q

--no-stdin=true|false 是否关闭标准输入，默认为 true

--sig-proxy=true|false 是否代理收到的系统信号给应用进程，默认为 true

当多个窗口同时 atttach 到同一个容器时，多个窗口会同步显示，一个窗口操作死机，其它窗口也都会死机。

3.2.8.2 使用 exec 命令进入容器

docker container exec [OPTIONS] CONTAINER COMMAND [ARG...]

option:

-d, --detach 在容器中后台执行命令

--detach-keys=""指定将容器切回后台的快捷键

-e, --env=[ ] 指定环境变量列表

-i, --interactive=true|false 打开标准输入接受用户输入命令，默认为 false

--privileged=true|false 是否给给执行命令以最高权限，默认为 false

-t, --tty=true|false 分配一个伪终端，默认为 false

-u, --user=""执行命令的用户或 ID

```
[root@docker197 ~]# docker container exec -it centos-7 bash
[root@467876eae467 /]#
[root@467876eae467 /]#
```

exec 不同于 attach,其有多个窗口运行同一个容器时，是相互独立的。

3.2.9 导入导出容器

3.2.9.1 导出容器

将一个容器导出为一个文件

docker container export [OPTIONS] CONTAINER

options:

-o,--output string 指定导出的文件名

实验一：

将 centos-7 这个容器导出到 centos-7.tar

```
[root@docker197 dockerlab]# docker container export -o centos-7.tar centos-7
[root@docker197 dockerlab]# ll
total 593692
-rw-------. 1 root root 607939072 Feb 13 15:41 centos-7.tar
dr-xr-xr-x. 3 root root       127 Feb  4 12:02 templates
[root@docker197 dockerlab]#
```

3.2.9.2 导入容器

将容器导出文件，导入成一个容器镜像文件

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

实验 1：将 3.2.9.1 导出的容器 tar 文件，导入到另一台主机

step 1:将容器导出文件拷到另一台主机上

```
[root@docker197 dockerlab]# scp ./centos-7.tar root@192.168.0.198:/root/dockerlab/
The authenticity of host '192.168.0.198 (192.168.0.198)' can't be established.
ECDSA key fingerprint is SHA256:hOiLnJ0nlfgwTSVQNlZh/K5RgivLAE9pZ2mfdl1/AkY.
ECDSA key fingerprint is MD5:bc:7b:52:7f:bb:f3:d3:b4:35:f8:cd:6b:6b:66:52:5b.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.198' (ECDSA) to the list of known hosts.
root@192.168.0.198's password:
centos-7.tar                                  23%  136MB  20.8MB/s   00:21 ETA
```

step 2:在另一台主机上查看文件是否已经成功拷贝过来

```
[root@docker198 dockerlab]# ll
total 593692
-rw-------. 1 root root 607939072 Feb 13 15:47 centos-7.tar
dr-xr-xr-x. 3 root root       127 Feb  4 12:02 templates
```

step 3:在另一台主机上导入容器

docker import centos-7.tar centos-7:1.0

```
[root@docker198 dockerlab]# docker import centos-7.tar centos-7:1.0
sha256:4b6099d23605effebeb52d4ad3315077170f86ce83b7da321d1ea7016e946f7b
[root@docker198 dockerlab]# docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[root@docker198 dockerlab]# docker images
REPOSITORY          TAG       IMAGE ID       CREATED              SIZE
centos-7            1.0       4b6099d23605   About a minute ago   589MB
<none>              <none>    107c61074092   7 minutes ago        589MB
ftp                 1.0       a297bef48210   9 days ago           750MB
yangjiqing/ftp      1.0       a297bef48210   9 days ago           750MB
centos              1.0       945cc60fc585   9 days ago           589MB
myhttpd             2.0       b49de57f5610   9 days ago           144MB
myhttpd             1.0       a8ea074f4566   2 weeks ago          144MB
[root@docker198 dockerlab]#
```

跟 3.1.8 导入镜像的功能差不多。

3.2.10 查看容器

3.2.10.1 查看容器详细信息

docker container inspect [OPTIONS] CONTAINER [CONTAINER...]

```
[root@docker197 dockerlab]# docker container inspect centos-7
[
    {
        "Id": "467876eae467a2b1a6b448053e77dd7e3d08457ceaf48ff758ac7dddd4dfbe59",
        "Created": "2022-02-11T02:20:21.382034198Z",
        "Path": "bash",
        "Args": [],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 137,
            "Error": "",
            "StartedAt": "2022-02-13T04:17:07.306584699Z",
            "FinishedAt": "2022-02-13T04:25:13.451768348Z"
        },
        "Image": "sha256:945cc60fc5855e683f26a3f754ee4cd7c9a452ab96583709f3092a737cba8c52",
        "ResolvConfPath": "/var/lib/docker/containers/467876eae467a2b1a6b448053e77dd7e3d08457ceaf48ff758ac7dddd4dfbe59/resolv.conf",
        "HostnamePath": "/var/lib/docker/containers/467876eae467a2b1a6b448053e77dd7e3d08457ceaf48ff758ac7dddd4dfbe59/hostname",
        "HostsPath": "/var/lib/docker/containers/467876eae467a2b1a6b448053e77dd7e3d08457ceaf48ff758ac7dddd4dfbe59/hosts",
        "LogPath": "/var/lib/docker/containers/467876eae467a2b1a6b448053e77dd7e3d08457ceaf48ff758ac7dddd4dfbe59/467876eae467a2b1a6b448053e77d
d7e3d08457ceaf48ff758ac7dddd4dfbe59-json.log",
```

3.2.10.2 查看容器的进程信息

docker container top CONTAINER [ps OPTIONS]

```
[root@docker197 dockerlab]# docker container top centos-7
UID                 PID                 PPID                C                   STIME               TTY                 TIME                C
MD
root                3067                3049                3                   16:12               pts/0               00:00:00            b
ash
```

3.2.10.3 查看容器状态

docker container stats [OPTIONS] [CONTAINER...]

options:

-a, --all 显示所有容器的状态

--format string 格式化输出信息

--no-stream 不持续输出

--no-trunc 不截断输出信息

```
CONTAINER ID   NAME       CPU %     MEM USAGE / LIMIT     MEM %     NET I/O       BLOCK I/O     PIDS
467876eae467   centos-7   0.00%     6.168MiB / 972.3MiB   0.63%     2.46kB / 0B   21.3MB / 0B   1
```

3.2.11 容器的其它操作

3.2.11.1 在主机和容器之间复制文件

docker container cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH

docker cp [OPTIONS] SRC_PATH CONTAINER:DEST_PATH

options:

-a, --archive:打包模式，会将源文件的 UID、GID 信息一起复制过来

-L, --follow-link：如果源文件是一个链接文件的话，将链接的文件复制过来

3.2.11.2 查看容器中的变更内容

docker container diff CONTAINER

```
[root@docker197 dockerlab]# docker container diff centos-7
C /root
A /root/1
A /root/2
A /root/.bash_history
```

3.2.11.3 查看容器的端口映射

docker container port CONTAINER [PRIVATE_PORT[/PROTO]]

3.2.11.4 更新容器配置

docker container update [OPTIONS] CONTAINER [CONTAINER...]

options:

--blkio-weight uint16 更新块 IO 限制，10~1000，默认值为 0，代表无限制

--cpu-period int 限制 CPU 调度器 CFS（Completely Fair Scheduler）使用时间，单位为微秒，最小为 1000

--cpu-quota int 限制 CPU 调度器配额，单位为微秒，最小为 1000

--cpu-rt-period int 限制 CPU 调度器 CFS 的实时周期，单位为微秒

--cpu-rt-runtime int 限制 CPU 调度器 CFS 的实时运行时，单位为微秒

--cpu-shares int 限制 CPU 使用份额

--cpus decimal 限制 CPU 使用个数

--cpuset-cpus string 允许使用的 CPU 核

--cpuset-mems string 允许使用的内存块

--kernel-memory bytes 限制使用的内核内存

-m, --memory bytes 限制使用的内存

--memory-reservation bytes 内存软限制

--memory-swap bytes 内存及交换区的限制，-1 为对交换区无限制

--pids-limit int 容器 PID 数量的限制

--restart string 容器退出后的重启策略

3.3、docker 数据管理

docker 数据管理的两种方式

1）数据卷方式：容器内数据映射到宿主机环境；

2）数据卷容器：使用特定容器维护数据卷。

3.3.1 容器内数据映射到宿主机环境

数据卷（date volumes）：可供容器使用的特殊目录，它将宿主机的目录直接映射至容器内。

数据卷特性：

1） 数据卷可在容器之间共享或重用；

2） 对数据卷内的数据操作会马上生效，不管在宿主机上操作，还是在容器内操作；

3） 对数据卷的更新不会影响应用，解耦开数据和应用；

4） 数据卷会一直存在，直到没有容器使用它，才可以卸载。

3.3.1 创建数据卷

```
docker volume create [OPTIONS] [VOLUME]
```

options:

-d, --driver string,指定数据卷驱动器的名称，默认为 local

实验 1：创建一个数据卷 test

```
docker volume create test
```

查看数据卷：

```
docker volume ls
```

```
[root@docker197 ~]# docker volume ls
DRIVER    VOLUME NAME
local     657f601a8ead1321b43b18aaae793cbf4505e92ccc672432c7e72df296dd7098
local     a9cb36ad484a94b54bbd9534e1b807d954d230d6e963b915a99e721d7238f079
local     test
```

这个目录实际上是在/var/lib/docker/volumes/

```
[root@docker197 ~]# ll /var/lib/docker/volumes/
total 24
drwx-----x. 3 root root     19 Feb 14 10:18 657f601a8ead1321b43b18aaae793cbf4505e92ccc672432c7e72df296dd7098
drwx-----x. 3 root root     19 Feb 14 10:19 a9cb36ad484a94b54bbd9534e1b807d954d230d6e963b915a99e721d7238f079
brw-------. 1 root root 253, 0 Feb 14 09:18 backingFsBlockDev
-rw-------. 1 root root  32768 Feb 14 14:07 metadata.db
drwx-----x. 3 root root     19 Feb 14 14:07 test
```

3.3.2 查看数据卷详细信息

```
docker volume inspect [OPTIONS] VOLUME [VOLUME...]
```

```
[root@docker197 ~]# docker volume inspect test
[
    {
        "CreatedAt": "2022-02-14T14:07:25+08:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/test/_data",
        "Name": "test",
        "Options": {},
        "Scope": "local"
    }
]
```

3.3.3 删除未使用的数据卷

```
docker volume prune [OPTIONS]
```

options:

-f, --force 强制删除

3.3.4 删除指定数据卷

```
docker volume rm [OPTIONS] VOLUME [VOLUME...]
```

options:

-f, --force 强制删除

3.3.5 绑定数据卷

```
docker run --mount IMAGE [COMMAND] [ARG...]
```

--mount 支持三种类型的数据卷

1）volume:普通数据卷，就是 4.1.1 小节中创建的数据卷，在/var/lib/docker/volumes/；

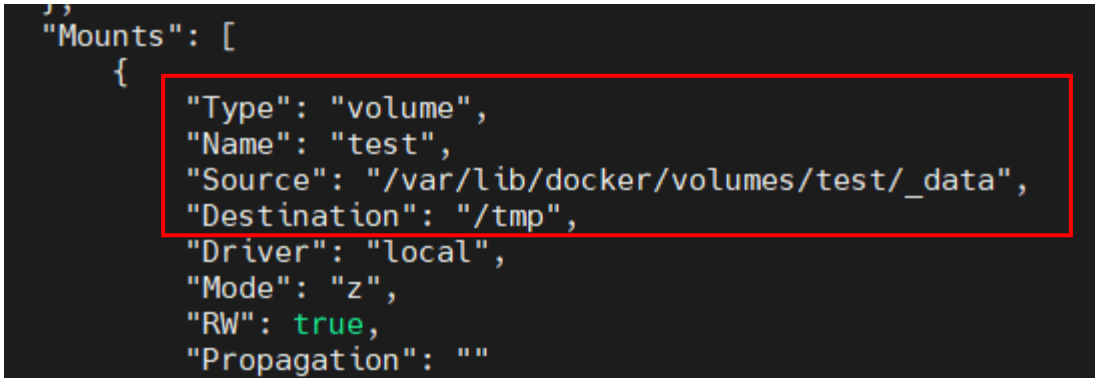
2）bind:绑定数据卷，映射到宿主机的任意指定目录下；

3）tmpfs:临时数据卷，只存于内存中。

实验 1：将刚刚创建的那个数据卷 test，挂载到容器 centos1 的/tmp

```
docker run -it --name centos1 --mount type=volume,source=test,destination=/tmp centos:7.0
bash
```

docker inspect centos1



实验 2：将宿主机中/root/ dockerlab/volume1,这个目录，映射到容器 centos 的/tmp

```
docker run -it –name centos --mount
```

type=bind,source=/root/dockerlab/volume1,destination=/tmp centos:1.0 bash

些命令等同于：

docker run -it –name centos –v /root/dockerlab/ volume1:/tmp centos:1.0 bash

```
[root@docker197 dockerlab]# docker run -it --name=centos --mount type=bind,source=/root/dockerlab/volume1,destination=/tmp centos:1.0 bash
[root@99f534bb2386 /]# ll /tmp/
total 0
-rw-r--r--. 1 root root 0 Feb 14 01:24 1
[root@99f534bb2386 /]#
```

/tmp 目录里的内容，就是宿主机/root/dockerlab/volume1 的内容。

docker run -it –name centos –v /root/dockerlab/volume1:/tmp:ro centos:1.0 bash

目录后面加个:ro,表示只读挂载

```
[root@ae596d8626ee /]# cd /tmp
[root@ae596d8626ee tmp]# touch 2
touch: cannot touch '2': Read-only file system
[root@ae596d8626ee tmp]#
```

此时是无法往目录里写东西的。

3.3.6 数据卷容器

数据卷容器也是一个容器，它的目的是专门为其它容器提供数据卷挂载。

实验 1：

step 1 创建一个容器 volume 做为数据卷容器，并将宿主机的/root/dockerlab/volume1 挂载到容器的 /volume1。

docker run -it –name volume -v /root/dockerlab/volume1:/volume1 centos:1.0 bash

```
[root@docker197 dockerlab]# docker run -it --name=volume -v /root/dockerlab/volume1:/volume1 centos:1.0 bash
[root@de1aefc0399e /]#
[root@de1aefc0399e /]#
[root@de1aefc0399e /]# ls /volume1/
1  2
```

目录已挂载成功。

step 2 创建一个容器 centos1,挂载数据卷容器中的目录

docker run -it --name=centos1 --volumes-from volume centos:1.0 bash

进入容器，发现目录已正常挂载

```
[root@dd367c249bb7 /]# ls ./volume1/
1  2
[root@dd367c249bb7 /]#
```

step 3 再来创建一个容器 centos2，挂载数据卷容器中的目录

```
docker run -it --name=centos2 --volume-from volume centos:1.0 bash
```

进入容器，发现也能挂载同一个目录，且两边的内容是一致的。

任意在哪一个容器内，对目录的数据进行操作，另一个容器内的内容也会同步改变。

```
[root@docker197 dockerlab]# docker run -it --name=centos2 --volumes-from volume centos:1.0 bash
[root@de839e7423d6 /]# ll ./volume1/
ls: cannot access ./volume1/: No such file or directory
[root@de839e7423d6 /]# ls ./volume1/
1  2
[root@de839e7423d6 /]#
```

--volumes-from 后指定的容器不一定要处于开启状态。

3.4 备份和恢复数据卷

3.4.1 备份数据卷容器的数据

```
docker run --volumes-from volume -v $(pwd):/backup --name=backup centos:1.0 tar -cvf /backup/volumebak.tar /volume1
```

原理：

1） 创建一个名为 backup 的容器，并挂载数据卷容器 volume 中的数据卷（--volumes-from volume），同时将宿主机的当前目录挂载到 backup 容器的/backup（-v $(pwd):/backup）；

2）启动容器后，将数据卷容器 volume 中的数据卷”volume1”目录打包到容器的/backup/volumebak.tar（tar -cvf /backup/volumebak.tar /volume1）

3）容器 backup 的/backup 目录，就是挂载的宿主机上的当前目录，因此，在宿主机的当前目录下，会出现一个 volumebak.tar 的备份文件。

```
[root@docker197 ~]# docker run --volumes-from volume -v $(pwd):/backup --name=backup centos:1.0 tar -cvf /backup/volumebak.tar /volume1
tar: Removing leading `/' from member names
/volume1/
/volume1/1
/volume1/2
/volume1/3
[root@docker197 ~]# ll
total 20
-rw-------. 1 root root  1590 Jan 14 22:44 anaconda-ks.cfg
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Desktop
drwxr-xr-x. 4 root root    76 Feb 14 14:23 dockerlab
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Documents
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Downloads
-rw-r--r--. 1 root root  1638 Jan 14 22:47 initial-setup-ks.cfg
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Music
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Pictures
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Public
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Templates
drwxr-xr-x. 2 root root     6 Jan 14 22:54 Videos
-rw-r--r--. 1 root root 10240 Feb 15 09:38 volumebak.tar
```

3.4.2 恢复数据卷容器的数据

先将数据卷容器的数据删除（模拟数据丢失）。

```
[root@docker197 dockerlab]# docker exec volume ls /volume1/
[root@docker197 dockerlab]#
```

数据卷中的数据已经没有了。

```
docker run --name=restory --volumes-from volume -v $(pwd):/restory centos:1.0 tar -xvf /restory/volumebak.tar -C /
```

原理：

1） 创建一个名为 restory 的容器（--name=restory），并挂载 volume 中的数据卷（--volumes-from volume），同时将宿主机的当前目录，挂载到容器 restory 的/restory；

2）将备份文件 volumebak.tar 解压到根目录下（tar -xvf /restory/volumebak.tar -C /）。

此时，文件已经恢复。

```
[root@docker197 dockerlab]# docker exec volume ls /volume1/
1
2
3
```

3.5 容器端口映射与容器互连

3.5.1 将宿主机的网络端口映射到容器的网络端口

```
docker run –P/-p  IMAGE [COMMAND] [ARG...]
```

options:

-P 将宿主机 49000~49900 端口随机映射到容器的开放端口

-p 指定宿主机和容器的端口来进行映射

实验 1：

```
docker run -d -P --name=nginx nginx:1.0
```

启动一个容 nginx,这个容器的默认配置会开放一个 80 端口

使用-P 参数的话，会从宿主机的的 49000~49900 端口，随机映射到容器的 80 端口，本实验中，宿主机的 49161 端口被映射到了容器的 8443 端口。

```
[root@docker197 ~]# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS         PORTS
     NAMES
455c23463824   nginx:1.0   "nginx -g 'daemon of…"   11 seconds ago   Up 2 seconds   0.0.0.0:49154->80/tcp, :::49154->80/t
cp   nginx
```

此时我们在外部网络，访问宿主机的 49154 端口，这个访问数据会被重写向到容器的 8443 端口,并且可以打开网页。

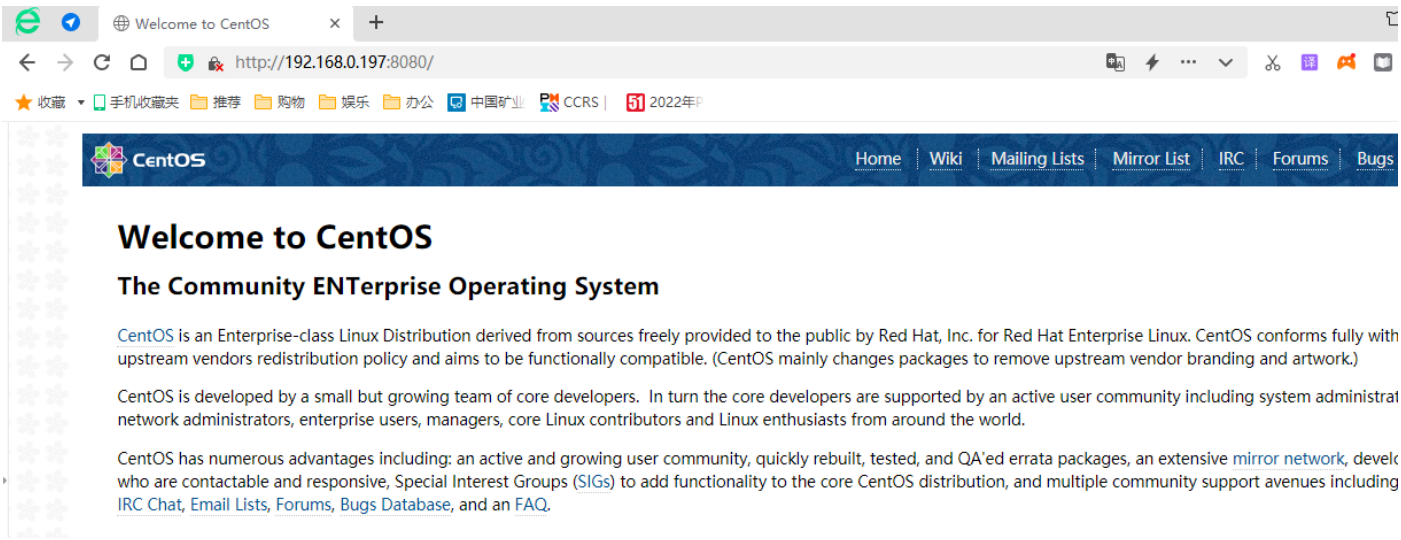
实验 2：

docker run -d -p 8080:80 --name nginx nginx:1.0

启动一个容器 nginx,这个容器的默认配置会开放一个 80 端口

使用-p 参数的话，可以指定端口来进行映射，本实验中，指定宿主机的 8080 端口映射到容器的 80 端口。

```
[root@docker197 ~]# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS         PORTS
   NAMES
86bb8838ec8f   nginx:1.0   "nginx -g 'daemon of…"   13 seconds ago   Up 2 seconds   0.0.0.0:8080->80/tcp, :::8080->80/tcp
```



这个 192.168.0.197 是宿主机的 IP 地址，8080 端口则被映射到了容器的 80 端口。

docker run -d -p 192.168.0.197:8080:172.17.0.3:80 --name nginx nginx:1.0

在端口前面加 IP 地址，则是指定被映射端口的 IP 地址。如果不加 IP 地址，则所有的 IP 的此端口都会被映射。

3.5.2 容器互联

容器互联（linking）是一种让多个容器的应用进行交互的机制。它会在源容器和接收容器之间创建连接关系。接收容器可以通过容器名快速访问到源容器，而不用指定具体的 IP 地址。

docker run --link string [SOURCE CONTAINER] container

实验 1：

step 1 创建一个新容器 nginx，做为源容器

docker run -d --name=nginx nginx:1.0

step 2 创建一个新容器 centos,连接到源容器

docker run -P -it --name=centos1 --link httpd centos:1.0 bash

查看容器 centos1 的/etc/hosts 文件

docker exec -it centos cat /etc/hosts

```
[root@6f53aeca72d3 /]# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      nginx 455c23463824
172.17.0.4      6f53aeca72d3
```

会发现这个文件中已经把 nginx 这个容器名和它的 IP 地址绑定了。

在容器内直接 ping nginx，发现是可以 ping 通的

```
[root@6f53aeca72d3 /]# ping nginx
PING nginx (172.17.0.2) 56(84) bytes of data.
64 bytes from nginx (172.17.0.2): icmp_seq=1 ttl=64 time=0.091 ms
64 bytes from nginx (172.17.0.2): icmp_seq=2 ttl=64 time=0.173 ms
64 bytes from nginx (172.17.0.2): icmp_seq=3 ttl=64 time=0.308 ms
```

3.6 Docker 仓库操作

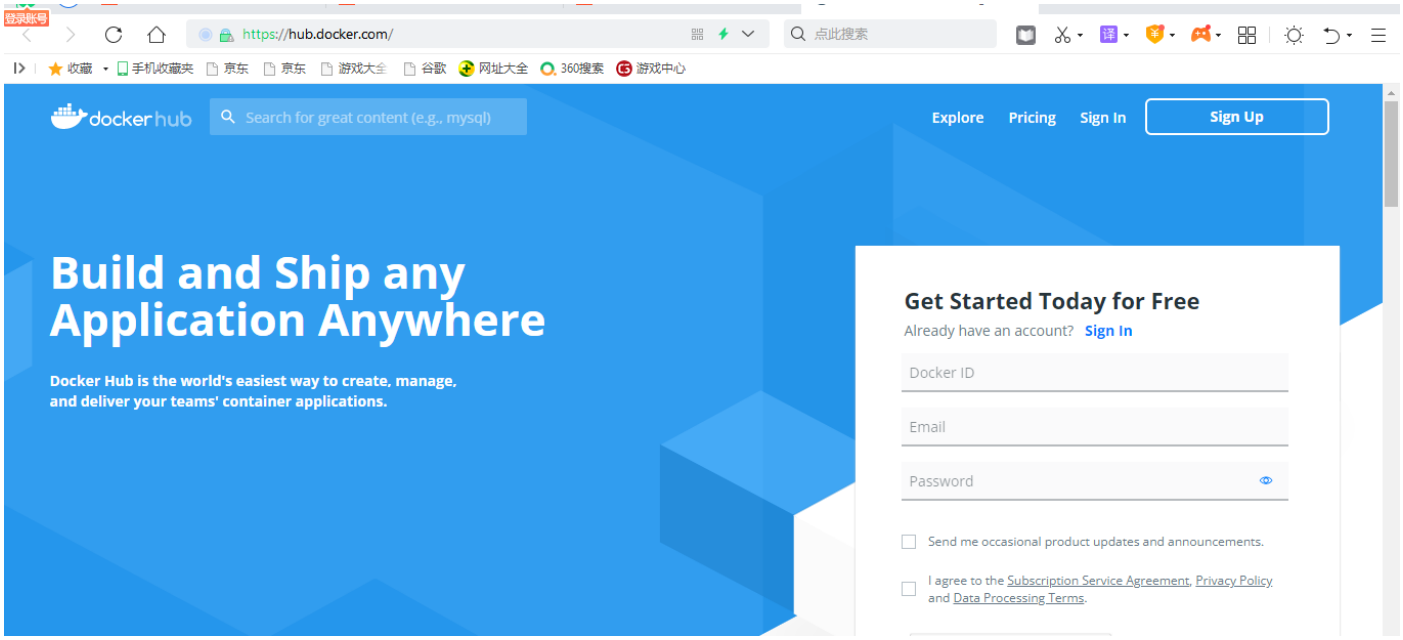
仓库(repository)是集中存放镜像的地方，分为公共仓库和私有仓库。

注册服务器（registry）是存放仓库的具体服务器，一个注册服务器上可以有多个仓库。

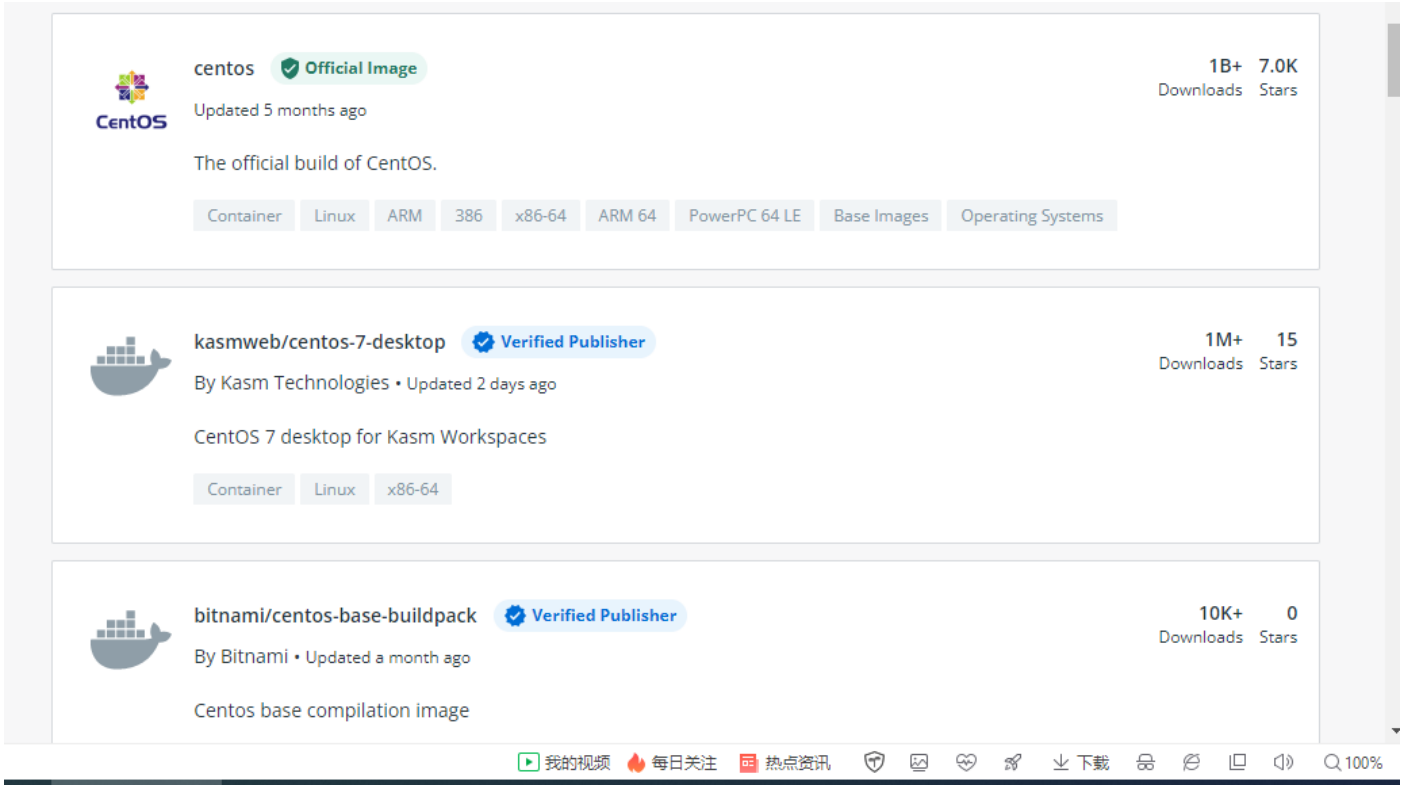
3.6.1 docker hub 公共镜像仓库

docker hub 是全球最大的公共镜像仓库

地址：https://hub.docker.com



需要去注册一个账号才可以登入。



可以搜索到各种镜像。

也可以使用 docker search“关键字”来搜索镜像文件

```
[root@docker197 ~]# docker search centos
NAME                              DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
centos                            The official build of CentOS.                   7016      [OK]
ansible/centos7-ansible           Ansible on Centos7                              135                  [OK]
consol/centos-xfce-vnc            Centos container with "headless" VNC session…   135                  [OK]
jdeathe/centos-ssh                OpenSSH / Supervisor / EPEL/IUS/SCL Repos - …   121                  [OK]
centos/systemd                    systemd enabled base container.                 105                  [OK]
imagine10255/centos6-lnmp-php56   centos6-lnmp-php56                              58                   [OK]
tutum/centos                      Simple CentOS docker image with SSH access      48
centos/postgresql-96-centos7      PostgreSQL is an advanced Object-Relational …   45
centos/httpd-24-centos7           Platform for running Apache httpd 2.4 or bui…   42
jdeathe/centos-ssh-apache-php     Apache PHP - CentOS.                            31                   [OK]
kinogmt/centos-ssh                CentOS with SSH                                 29                   [OK]
guyton/centos6                    From official centos6 container with full up…   10                   [OK]
nathonfowlie/centos-jre           Latest CentOS image with the JRE pre-install…   8                    [OK]
centos/tools                      Docker image that has systems administration…   7                    [OK]
drecom/centos-ruby                centos ruby                                     6                    [OK]
roboxes/centos8                   A generic CentOS 8 base image.                  5
darksheer/centos                  Base Centos Image -- Updated hourly             3                    [OK]
amd64/centos                      The official build of CentOS.                   2
miko2u/centos6                    CentOS6 日本語環境                                2                    [OK]
dokken/centos-7                   CentOS 7 image for kitchen-dokken               2
blacklabelops/centos              CentOS Base Image! Built and Updates Daily!     1                    [OK]
mcnaughton/centos-base            centos base image                               1                    [OK]
jelastic/centosvps                An image of the CentOS Elastic VPS maintaine…   0
starlabio/centos-native-build     Our CentOS image for native builds              0                    [OK]
smartentry/centos                 centos with smartentry                          0                    [OK]
[root@docker197 ~]# ^C
```

然后通过 docker image pull [OPTIONS] NAME[:TAG|@DIGEST],来拉取镜像。

pull 命令详细使用方法，见 3.1.1 小节

3.6.2 搭建本地私有仓库

step 1 拉取一个 registry 镜像

docker image pull registry:2

```
[root@docker197 ~]# docker image pull registry
Using default tag: latest
latest: Pulling from library/registry
59bf1c3509f3: Pull complete
666ba61612fd: Pull complete
a4642f78634a: Pull complete
9ab650d99063: Pull complete
91dceb018e81: Pull complete
Digest: sha256:c26590bcf53822a542e78fab5c88e1dfbcdee91c1882f4656b7db7b542d91d97
Status: Downloaded newer image for registry:latest
docker.io/library/registry:latest
[root@docker197 ~]# docker images
REPOSITORY          TAG       IMAGE ID       CREATED         SIZE
registry            latest    9c97225e83c8   6 days ago      24.2MB
ftp                 1.0       a297bef48210   9 days ago      750MB
yangjiqing/ftp      1.0       a297bef48210   9 days ago      750MB
centos              1.0       945cc60fc585   9 days ago      589MB
myhttpd             2.0       b49de57f5610   9 days ago      144MB
myhttpd             1.0       a8ea074f4566   2 weeks ago     144MB
[root@docker197 ~]#
```

step 2 创建并启动 registry 容器

docker run -d --name registry -v /root/dockerlab/registry/:/var/lib/registry -p 5000:5000 --restart=always registry:2

命令注解：

-d 容器在后台运行

--name registry 将容器命名为 registry

-v /root/dockerlab/registry/:/var/lib/registry 将宿主机的/root/dockerlab/registry 映射到容器的/var/lib/registry 目录

-p 5000:5000 将宿主机的 5000 端口映射到容器的 5000 端口

--restart=always 让容器可以自动重启

registry:2 镜像文件名

```
[root@docker198 dockerlab]# docker run \
> -d \
> --name registry \
> -v /root/dockerlab/registry/:/var/lib/registry \
> -p 5000:5000 \
> --restart=always \
> registry:2
```

```
[root@docker198 dockerlab]# docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS         PORTS                                       NAMES
92c33f233973   registry:2   "/entrypoint.sh /etc…"   3 minutes ago   Up 3 minutes   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp   registry
```

```
[root@docker197 ~]# netstat -tupln | grep docker
tcp        0      0 0.0.0.0:5000            0.0.0.0:*               LISTEN      4400/docker-proxy
tcp6       0      0 :::5000                 :::*                    LISTEN      4404/docker-proxy
```

docker-proxy 进程会监听 5000 端口

查看容器的进程

docker exec registry netstat –tupln

```
[root@docker197 ~]# docker exec registry netstat -tupln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 :::5000                 :::*                    LISTEN      1/registry
```

容器中有一个 PID 号为 1 的进程 registry,并监听在 5000 端口上。

step 3 放通宿主机的防火墙 5000/TCP 端口，并测试网络

firewall-cmd --add-port=5000/tcp

firewall-cmd --add-port=5000/tcp –permanent

在另外一台主机上，测试网络和服务的联通性

curl http://192.168.0.197:5000/v2/

如果会出现“{}”这个符号，则说明网络和服务是正常的。

```
[root@docker197 dockerlab]# curl http://192.168.0.198:5000/v2/
{}[root@docker197 dockerlab]#
```

step 4 上传镜像

在另外一台 docker 主机上将镜像上传到私有仓库。

先修改或创建 vim /etc/docker/daemon.json 文件

```
{
  "insecure-registries": [
    "192.168.0.197:5000"
  ]
}
```

这里的地址就是仓库所有主机的 IP 和端口号。

重启 docker 服务

systemctl daemon-reload

systemctl restart docker

修改下镜像文件的标签，**一定要以 192.168.0.197:5000 为前缀**

docker tag centos 192.168.0.197:5000/centos:1.0

```
[root@docker198 ~]# docker tag centos 192.168.0.197:5000/centos:1.0
[root@docker198 ~]# docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
192.168.0.197:5000/centos     1.0       5d0da3dc9764   5 months ago    231MB
centos                        latest    5d0da3dc9764   5 months ago    231MB
[root@docker198 ~]#
```

上传镜像：

docker push 192.168.0.198:5000/centos:1.0

```
[root@docker197 dockerlab]# docker push 192.168.0.198:5000/centos:1.0
The push refers to repository [192.168.0.198:5000/centos]
74ddd0ec08fa: Pushing [==================================>                ]    151MB/231.3MB
```

验证是否上传成功：

curl http://192.168.0.197:5000/v2/_catalog

```
[root@docker197 dockerlab]# curl http://192.168.0.198:5000/v2/_catalog
{"repositories":["centos","lump","mysql"]}
```

出现了这个镜像名，表示上传成功。

镜像文件默认放在以下路径下：

/var/lib/registry/docker/registry/v2/repositories/

```
[root@docker197 ~]# docker exec registry ls /var/lib/registry/docker/registry/v2/repositories/
centos
```

step 5 下载镜像

docker pull 192.168.0.198:5000/centos:1.0

```
[root@docker197 dockerlab]# docker pull 192.168.0.198:5000/centos:1.0
1.0: Pulling from centos
Digest: sha256:a1801b843b1bfaf77c501e7a6d3f709401a1e0c83863037fa3aab063a7fdb9dc
Status: Downloaded newer image for 192.168.0.198:5000/centos:1.0
192.168.0.198:5000/centos:1.0
[root@docker197 dockerlab]# docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
registry                      2         2e200967d166   10 days ago     24.2MB
192.168.0.198:5000/lump       1.0       38c19e45e1ad   11 days ago     1.4GB
lump                          latest    38c19e45e1ad   11 days ago     1.4GB
192.168.0.198:5000/mysql      1.0       b91913db5402   3 weeks ago     540MB
mysql                         1.0       b91913db5402   3 weeks ago     540MB
192.168.0.198:5000/centos     1.0       5d0da3dc9764   7 months ago    231MB
centos                        latest    5d0da3dc9764   7 months ago    231MB
```

下载成功。

四、docker 系统实战

4.1 操作系统

4.1.1busybox

busybox 是一个集成了 100 多个最常用 linux 命令(如 cat、echo、grep、mount、telnet 等）的精简工具箱，只有不到 2M。被誉为“linux 系统的军刀”。busybox 可运行于多款 POSIX 环境的操作系统中，如 linux、android、hurd、freeBSD 等。

busybox 最初是基于希望在一张软件上创建一个操作系统而设计的，所以容量非常小。现在，busybox 在嵌入式系统中应用广泛。

busyboxt 算不上是一个操作系统，只是一个工具的压缩包，它需要运行在 linux 系统上。

step 1 获取官方镜像

docker pull busybox

```
[root@docker197 ~]# docker image ls | grep busybox
busybox                 latest    ec3f0931a6e6   2 weeks ago    1.24MB
```

只有 1.24M。

step 2 启动容器

docker run –it busybox

进入容器后发现，可以使用大部分的常用命令。

```
/ # ls /bin/
[               deluser         gunzip          losetup         partprobe       setfont         true
[[              depmod          gzip            lpd             passwd          setkeycodes     truncate
acpid           devmem          halt            lpq             paste           setlogcons      ts
add-shell       df              hd              lpr             patch           setpriv         tty
addgroup        dhcprelay       hdparm          ls              pgrep           setserial       ttysize
adduser         diff            head            lsattr          pidof           setsid          tunctl
adjtimex        dirname         hexdump         lsmod           ping            setuidgid       ubiattach
ar              dmesg           hexedit         lsof            ping6           sh              ubidetach
arch            dnsd            hostid          lspci           pipe_progress   sha1sum         ubimkvol
arp             dnsdomainname   hostname        lsscsi          pivot_root      sha256sum       ubirename
arping          dos2unix        httpd           lsusb           pkill           sha3sum         ubirmvol
ascii           dpkg            hush            lzcat           pmap            sha512sum       ubirsvol
ash             dpkg-deb        hwclock         lzma            popmaildir      showkey         ubiupdatevol
awk             du              i2cdetect       lzop            poweroff        shred           udhcpc
base32          dumpkmap        i2cdump         makedevs        powertop        shuf            udhcpc6
base64          dumpleases      i2cget          makemime        printenv        slattach        udhcpd
basename        echo            i2cset          man             printf          sleep           udpsvd
bc              ed              i2ctransfer     md5sum          ps              smemcap         uevent
beep            egrep           id              mdev            pscan           softlimit       umount
blkdiscard      eject           ifconfig        mesg            pstree          sort            uname
blkid           env             ifdown          microcom        pwd             split           unexpand
blockdev        envdir          ifenslave       mim             pwdx            ssl_client      uniq
bootchartd      envuidgid       ifplugd         mkdir           raidautorun     start-stop-daemon unix2dos
brctl           ether-wake      ifup            mkdosfs         rdate           stat            unlink
bunzip2         expand          inetd           mke2fs          rdev            strings         unlzma
busybox         expr            init            mkfifo          readahead       stty            unshare
bzcat           factor          insmod          mkfs.ext2       readlink        su              unxz
bzip2           fakeidentd      install         mkfs.minix      readprofile     sulogin         unzip
cal             fallocate       ionice          mkfs.vfat       realpath        sum             uptime
cat             false           iostat          mknod           reboot          sv              users
chat            fatattr         ip              mkpasswd        reformime       svc             usleep
chattr          fbset           ipaddr          mkswap          remove-shell    svlogd          uudecode
chgrp           fbsplash        ipcalc          mktemp          renice          svok            uuencode
chmod           fdflush         ipcrm           modinfo         reset           swapoff         vconfig
chown           fdformat        ipcs            modprobe        resize          swapon          vi
chpasswd        fdisk           iplink          more            resume          switch_root     vlock
chpst           fgconsole       ipneigh         mount           rev             sync            volname
chroot          fgrep           iproute         mountpoint      rm              sysctl          w
chrt            find            iprule          mpstat          rmdir           syslogd         wall
chvt            findfs          iptunnel        mt              rmmod           tac             watch
cksum           flock           kbd_mode        mv              route           tail            watchdog
clear           fold            kill            nameif          rpm             tar             wc
cmp             free            killall         nanddump        rpm2cpio        taskset         wget
comm            freeramdisk     killall5        nandwrite       rtcwake         tc              which
conspy          fsck            klogd           nbd-client      run-init        tcpsvd          who
cp              fsck.minix      last            nc              run-parts       tee             whoami
cpio            fsfreeze        less            netstat         runlevel        telnet          whois
crc32           fstrim          link            nice            runsv           telnetd         xargs
crond           fsync           linux32         nl              runsvdir        test            xxd
crontab         ftpd            linux64         nmeter          rx              tftp            xz
cryptpw         ftpget          linuxrc         nohup           script          tftpd           xzcat
cttyhack        ftpput          ln              nologin         scriptreplay    time            yes
cut             fuser           loadfont        nproc           sed             timeout         zcat
date            getconf         loadkmap        nsenter         sendmail        top             zcip
dc              getopt          logger          nslookup        seq             touch
dd              getty           login           ntpd            setarch         tr
deallocvt       grep            logname         od              setconsole      traceroute
delgroup        groups          logread         openvt          setfattr        traceroute6
```

相关资源：

busybox 官网: https://busybox.net

busyboxs 官方仓库：https://git.busybox.net/busybox

busybox 官方镜像：https://hup.docker.com/_/busybox

busybox 官方镜像仓库：https://github.com/docker-library/busybox

4.1.2 alpine

alpine 是一个面向安全的轻型 linux 系统，关注安全、性能和资源。

alpine 采用了 musl libc（Musl 是一个轻量级的 C 标准库，设计作为 GNU C library (glibc)、 uClibc 或 Android Bionic 的替代用于嵌入式操作系统和移动设备。)和 busybox 以减少系统的体积和资源消耗。此外还提供了包管理工具 apk 查询和安装软件包。

step 1 获取 alpine 镜像

docker pull alpine

```
[root@docker197 ~]# docker images | grep alpine
alpine              latest    c059bfaa849c   2 months ago   5.59MB
```

只有 5.59M

step 2 启动容器

docker run -it --name=alpine --hostname=alpine alpine

```
[root@docker197 ~]# docker run -it alpine
/ #
/ #
/ #
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp    usr    var
/ #
```

查看一下/bin 下，发现只有一个可执行文件 busybox，其它的文件（也就是系统命令）全部都是链接到 busybox 这个可执行文件的。而且命令也少了很多。

```
/ # ls -l /bin/
total 808
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 arch -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 ash -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 base64 -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 bbconfig -> /bin/busybox
-rwxr-xr-x    1 root     root        824984 Nov 23 00:57 busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 cat -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 chgrp -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 chmod -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 chown -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 cp -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 date -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 dd -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 df -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 dmesg -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 dnsdomainname -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 dumpkmap -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 echo -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 ed -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 egrep -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 false -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 fatattr -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 fdflush -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 fgrep -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 fsync -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 getopt -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 grep -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 gunzip -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Nov 24 09:20 gzip -> /bin/busybox
```

可以说 alpine 是一个阉割版的 linux 操作系统.

alpine 下可以使用 apk 安装软件

apk add SOFEWARE-NAME

```
/ # apk add curl
fetch http://dl-cdn.alpinelinux.org/alpine/v3.15/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.15/community/x86_64/APKINDEX.tar.gz
(1/5) Installing ca-certificates (20211220-r0)
(2/5) Installing brotli-libs (1.0.9-r5)
(3/5) Installing nghttp2-libs (1.46.0-r0)
(4/5) Installing libcurl (7.80.0-r0)
(5/5) Installing curl (7.80.0-r0)
Executing busybox-1.34.1-r3.trigger
Executing ca-certificates-20211220-r0.trigger
OK: 8 MiB in 19 packages
```

```
/ # apk info -a
musl
busybox
alpine-baselayout
alpine-keys
ca-certificates-bundle
libcrypto1.1
libssl1.1
libretls
ssl_client
zlib
apk-tools
scanelf
musl-utils
libc-utils
ca-certificates
brotli-libs
nghttp2-libs
libcurl
curl
mini_httpd
```

alpine 相关资源

alpine 官网：http://alpinelinux.org

alpine 官方仓库：https://github.com/alpinelinux

alpine 官方镜像：https://hub.docker.com/_/alpine

alpine 官方镜像：https://github.com/gliderlabs/docker-alpine

5.1.3 centos

step 1 获取镜像

docker pull centos

```
[root@docker197 ~]# docker pull centos
Using default tag: latest
latest: Pulling from library/centos
a1d0c7532777: Pull complete
Digest: sha256:a27fd8080b517143cbbbab9dfb7c8571c40d67d534bbdee55bd6c473f432b177
Status: Downloaded newer image for centos:latest
docker.io/library/centos:latest
[root@docker197 ~]# docker images
REPOSITORY                 TAG       IMAGE ID       CREATED         SIZE
yangjiqing/httpd           1.0       2849b22e751d   46 hours ago    629MB
registry                   latest    9c97225e83c8   13 days ago     24.2MB
busybox                    latest    ec3f0931a6e6   2 weeks ago     1.24MB
yangjiqing/ftp             1.0       a297bef48210   2 weeks ago     750MB
192.168.0.197:5000/ftp     1.0       a297bef48210   2 weeks ago     750MB
ftp                        1.0       a297bef48210   2 weeks ago     750MB
centos                     1.0       945cc60fc585   2 weeks ago     589MB
alpine                     latest    c059bfaa849c   2 months ago    5.59MB
centos                     latest    5d0da3dc9764   5 months ago    231MB
centos/httpd-24-centos7    latest    5ae4c76f6a3e   6 months ago    348MB
centos/systemd             latest    05d3c1e2d0c1   3 years ago     202MB
```

这个就比较大了，231M

step 2 启动容器

docker run -it centos:latest bash

```
[root@docker197 ~]# docker run -it centos:latest bash
[root@4c5c34f7a70f /]#
[root@4c5c34f7a70f /]#
[root@4c5c34f7a70f /]#
[root@4c5c34f7a70f /]#
[root@4c5c34f7a70f /]# ls
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
[root@4c5c34f7a70f /]#
```

进到容器以后，就跟普通的 centos 一样操作了。

CentOS 的相关资源

CentOS 官网：https://https://www.centos.org/

CentOS 官方镜像：https://hub.docker.com/_/centos/

5.1.4 debian/ubuntu

这两个跟 centos 的操作是一样的。

相关资源：

Debian 官网：https://www.debian.org/

Debian 官方镜像：https://hub.docker.com/_/debian/

Ubuntu 的相关资源如下：

Ubuntu 官网：http://www.ubuntu.org.cn/global

Ubuntu 官方镜像：https://hub.docker.com/_/ubuntu/

4.2 为镜像添加 SSH 服务

4.2.1 基于 commit 命令添加 SSH 服务

step 1 用 centos:latest 这个镜像创建一个容器

```
[root@docker197 ~]# docker run -it --name=centos --hostname=centos centos:latest bash
[root@centos /]#
```

```
[root@centos /]# ip address
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
22: eth0@if23: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

容器会从宿主机那边获得一个 IP 地址,我这边容器的 IP 是 172.17.0.2/16

如果你的宿主机可以上外网，默认情况下，容器也是可以上网的。

```
[root@centos /]# ping baidu.com
PING baidu.com (220.181.38.148) 56(84) bytes of data.
64 bytes from 220.181.38.148 (220.181.38.148): icmp_seq=1 ttl=52 time=26.5 ms
64 bytes from 220.181.38.148 (220.181.38.148): icmp_seq=2 ttl=52 time=30.6 ms
64 bytes from 220.181.38.148 (220.181.38.148): icmp_seq=3 ttl=52 time=27.3 ms
64 bytes from 220.181.38.148 (220.181.38.148): icmp_seq=4 ttl=52 time=28.1 ms
```

果然是可以 ping 通的。

step 2 配置下 yum 源

我这个镜像是 centos8 的，镜像里默认的 yum 源，似乎都不能使用了，要替换成我这个源。

https://mirrors.aliyun.com/repo/Centos-vault-8.5.2111.repo

把原来/etc/yum.repos.d/里的 yum 配置文件全部删掉，替换成上面链接里的这个源。

清除下 yum 缓存

yum makecache

这样就可以了

step 3 安装 openssh-server

yum install openssh-server–y

step 4 安装下网络工具

yum install net-tools #可以不装，这里我们为了便于做实验时的验证。

step5 生成一下 SSH 密钥

/usr/bin/ssh-keygen -A

```
[root@centos /]# /usr/bin/ssh-keygen -A
ssh-keygen: generating new host keys: RSA DSA ECDSA ED25519
```

step 6 启动 SSHD 服务

/sbin/sshd -D & # “&”这个符号代表让服务在后台启动

```
[root@centos /]# ps aux | grep sshd
root         168  0.0  0.3  76520  3884 pts/4    S    09:22   0:00 /sbin/sshd -D
```

服务已经起来了。

netstat –tupln #用来查看系统监听端口

```
[root@centos /]# netstat -tupln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      168/sshd
tcp6       0      0 :::22                   :::*                    LISTEN      168/sshd
```

SSHD 的服务端口是 22，已经起来了。

step 7 在宿主机中通过 ssh 来连接容器

ssh 172.17.0.2

```
[root@docker197 ~]# ssh 172.17.0.2
The authenticity of host '172.17.0.2 (172.17.0.2)' can't be established.
ECDSA key fingerprint is SHA256:BfE4Gqj43SjlMHuygqXeeYRDrwze0C7gKZlJpx4UaoU.
ECDSA key fingerprint is MD5:05:91:8b:66:ad:5c:f4:f6:f0:db:c3:82:42:78:a8:25.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.17.0.2' (ECDSA) to the list of known hosts.
root@172.17.0.2's password:
Permission denied, please try again.
root@172.17.0.2's password:
Permission denied, please try again.
root@172.17.0.2's password:
```

这里让我们输入 root 密码，但我们容器的 root 没有密码，所以登入不了。

这里我们在容器中安装一下 passwd

yum install passwd –y

这样就可以修改 root 密码了。

```
[root@centos /]# passwd
Changing password for user root.
New password:
BAD PASSWORD: The password fails the dictionary check - it is based on a dictionary word
Retype new password:
passwd: all authentication tokens updated successfully.
```

```
[root@docker197 ~]# ssh 172.17.0.2
root@172.17.0.2's password:
"System is booting up. Unprivileged users are not permitted to log in yet. Please
"
Last failed login: Mon Feb 21 09:31:20 UTC 2022 from 172.17.0.1 on ssh:notty
There were 2 failed login attempts since the last successful login.
[root@centos ~]#
[root@centos ~]#
[root@centos ~]#
[root@centos ~]#
```

可以登入了。

step 8 退出容器，保存镜像

docker stop centos

```
[root@docker197 ~]# docker stop centos
centos
[root@docker197 ~]# docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED       STATUS                     PORTS     NAMES
cfed685d9656   centos:latest  "bash"                   2 hours ago   Exited (0) 5 seconds ago             centos
```

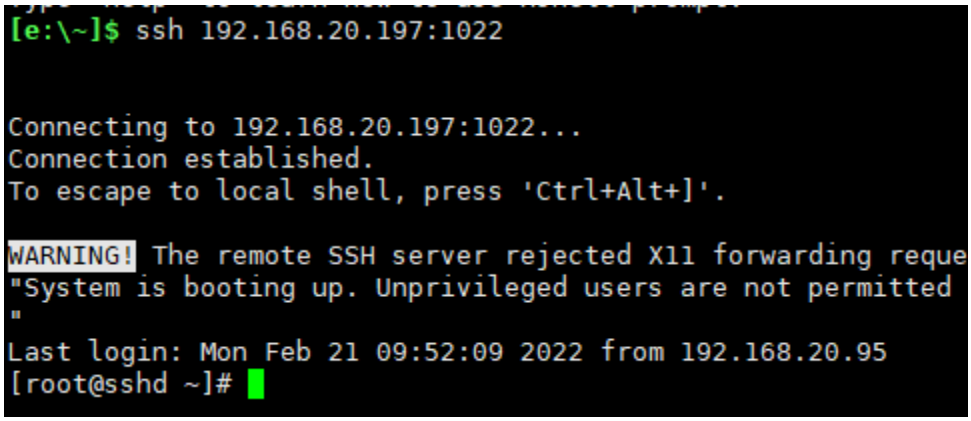
docker commit centos sshd:centos#将容器保存为一个名为 sshd:centos 的镜像。

```
[root@docker197 ~]# docker commit centos sshd:centos
sha256:b94344d3cc091a94d3c32790567404cb1fcee8143ce5b978fee3d43f50862b99
[root@docker197 ~]# docker images
REPOSITORY                 TAG       IMAGE ID       CREATED              SIZE
sshd                       centos    b94344d3cc09   About a minute ago   279MB
```

step 9 使用 sshd:centos 这个镜像来创建一个容器，并将宿主机的 1022 端口映射到容器的 22 端口。

docker run -d -p 1022:22 --name=sshd --hostname=sshd sshd:centos /sbin/sshd -D

这样在外网上就可以通过宿主机 IP:1022 来 ssh 到容器了。



4.2.2 基于 dockerfile 文件添加 SSH 服务

step 1 准备一个国内的 yum 源配置文件备用

wget https://mirrors.aliyun.com/repo/Centos-vault-8.5.2111.repo #centos8 阿里云 yum 源配置文件

step 2 创建 dockerfile 文件

vim dockerfile-sshd

文件内容如下：

FROM centos:latest#指定基础镜像为 centos:latest

LABEL auto build sshd#指定一些版本信息

EXPOSE 22#说明容器开放的端口号

COPY ./Centos-vault-8.5.2111.repo /root/Centos-vault-8.5.2111.repo#将国内 yum 源拷到镜像中,这边的源路径必须为 dockerfile 所在目录的相对路径

RUN rm -rf /etc/yum.repos.d/*#删除容器内原有 yum 源

RUN mv /root/Centos-vault-8.5.2111.repo /etc/yum.repos.d/# 将国内 yum 源配置文件移入 /etc/yum.repos/

RUN yum makecache #创建 yum 缓存

RUN yum install openssh-server –y #安装 openssh-server

RUN yum install passwd –y#安装 passwd 工具

RUN rm -rf /var/cache/yum/*#删除 yum 缓存数据

RUN echo "123456" | passwd --stdin root#将镜像内 root 账号密码设为 123456

RUN /usr/bin/ssh-keygen -A #镜像内生成 SSH 密钥对

CMD /sbin/sshd –D #启动 sshd 服务

step 3 创建镜像文件

docker build -t sshd1:centos -f ./dockerfile-sshd . #注意最后面那个“.”，代表当前目录

```
[root@docker197 dockerfile]# docker images | grep sshd1
sshd1                    centos    22f1fdf9db49   14 minutes ago      317MB
```

这边镜像文件已经成功创建。

step 4 使用 sshd1:centos 这个镜像来创建一个容器，并将宿主机的 2022 端口映射到容器的 22 端口。

docker run -d -p 2022:22 --name=sshd1 --hostname=sshd1 sshd:centos

这样在外网上就可以通过宿主机 IP:2022 来 ssh 到容器了。

```
Connecting to 192.168.20.197:2022...
Connection established.
To escape to local shell, press 'Ctrl+Alt+]'.

WARNING! The remote SSH server rejected X11 forwarding req
"System is booting up. Unprivileged users are not permitte
"
Last login: Tue Feb 22 09:19:01 2022 from 192.168.20.95
[root@sshd1 ~]#
```

4.3 基于 dockerfile 文件添加 MySQL 服务

step 1 配置 centos8 yum 源

wgethttps://mirrors.aliyun.com/repo/Centos-vault-8.5.2111.repo

step 2 创建容器自启动脚本

vim mysql_entrypoint.sh #容器启动的时候自动运行些脚本，对 mysql 做一些必要的配置

#!/bin/bash

chown -R mysql:mysql /var/lib/mysql/ #将 mysql 数据目录的所有权属性配置为 mysql 用户

mysql 用户是在安装 mysql 的时候自动创建的。

chown mysql:mysql /usr/sbin/mysqld #将 mysql 的启动文件的所有权属性配置为 mysql 用户

echo user=mysql>> /etc/my.cnf.d/mysql-server.cnf #在 mysql 的配置文件中加入一行 “user=mysql”,表示使用 mysql 这个用户，来启动 mysql 服务

```
IFNULL=`ls /var/lib/mysql/ | wc -l`                #列出 mysql 数据目录下的文件数，并将值赋予 IFNULL 这个变量
if [ "$IFNULL" = 0 ]; then                         #如果 IFNULL 的值为 0（数据目录下没有文件），就执行 mysql 的初始化工作。
    mysqld --initialize                            #初始化 mysql 系统
    mysqld -D                                      #在后台启动 mysqld 服务
    DATABASE_PASSWORD_TEMP=`tail /var/log/mysql/mysqld.log | grep password |awk '{print $NF}'`
     #mysqld 8 在初始化的时候，会在/var/log/mysql/mysqld.log 这个文件中生成一个临时密码，这条命令的作用是将这个临时密码取出来，并将其赋予 DATABASE_PASSWORD_TEMP 这个变量
     mysql -u root -p$DATABASE_PASSWORD_TEMP -e "alter user user( ) identified by '$ $DATABASE_PASSWORD_ROOT';" --connect-expired-password
     #使用临时密码来更改 mysql 的 root 用户密码，新密码使用 DATABASE_PASSWORD_ROOT 这个变量，这个变量的值，在启动容器的时候通过 docker run -e 这个命令带进来
     mysql -u root -p$ $DATABASE_PASSWORD_ROOT -e "update mysql.user set host='%' where user='root'"
     #将 mysql 的 root 用户的 host 值改为‘%’，代表可以从何意主机连进来。
     mysql -u root -p$ $DATABASE_PASSWORD_ROOT -e "flush privileges;"
     #刷新数据库，让配置的内容立即生效。
     PID=`ps aux | grep mysqld | grep -v grep | awk '{print $2}'`
     kill -9 $PID
fi
mysqld          #在前台启动 mysql 服务
```

step3 创建 dockfile 文件

```
vim dockerfile_mysql
FROM centos:latest                          #指定基础镜像文件
LABEL auto build mysql                      #自定义标签信息
EXPOSE 3306                                 #描述端口信息
COPY ./Centos-vault-8.5.2111.repo /root/Centos-vault-8.5.2111.repo     #将 yum 源文件拷贝到容器
COPY ./mysql_entrypoint.sh /mysql_entrypoint.sh                        #将启动脚本拷贝到容器
RUN chmod 755 /mysql_entrypoint.sh \                                   #将启动脚本配置为可执行
```

&& ln -s /mysql_entrypoint.sh /usr/bin/mysql_entrypoint.sh \ #创建启动脚本软件软链接

&& rm -rf /etc/yum.repos.d/* \ #清除容器内原有的 yum 源文件

&& mv /root/Centos-vault-8.5.2111.repo /etc/yum.repos.d/ \ #将 yum 源文件拷贝到 yum 目录

&& yum makecache #创建 yum 缓存

RUN yum install mysql mysql-server -y \ #安装 mysql

&& rm -rf /var/cache/yum/* #清除 yum 缓存文件

CMD mysql_entrypoint.sh #配置容器自启动脚本为 mysql_entrypoint.sh 就是上一步我们创建的那个脚本文件

step 4 创建镜像

通过 dockerfile-mysql 这个 dockerfile，创建一个名为 mysql:1.0 的镜像。

docker build -t mysql:1.0 -f ./dockerfile-mysql . #注意最后面那个“.”，代表当前目录

```
[root@docker197 dockerfile]# docker images | grep mysql
mysql        1.0       6f4a6e44a7a0   2 days ago     540MB
```

step 5 启动容器

先在宿主机上创建一个 mysql 的数据目录。（容器中只跑应用，数据目录直接挂载宿主机中的目录）

mkdir mysqldata

启动容器

docker run -d --name=mysql --hostname=mysql -v /root/dockerlab/mysqldata:/var/lib/mysql -e DATABASE_PASSWORD_ROOT='xxxxxxxx' -p 3306:3306 mysql:1.0

命令注解：

-d :让容器在后台运行

--name=mysql 容器命名为 mysql

--hostname=容器的主机名为 mysql

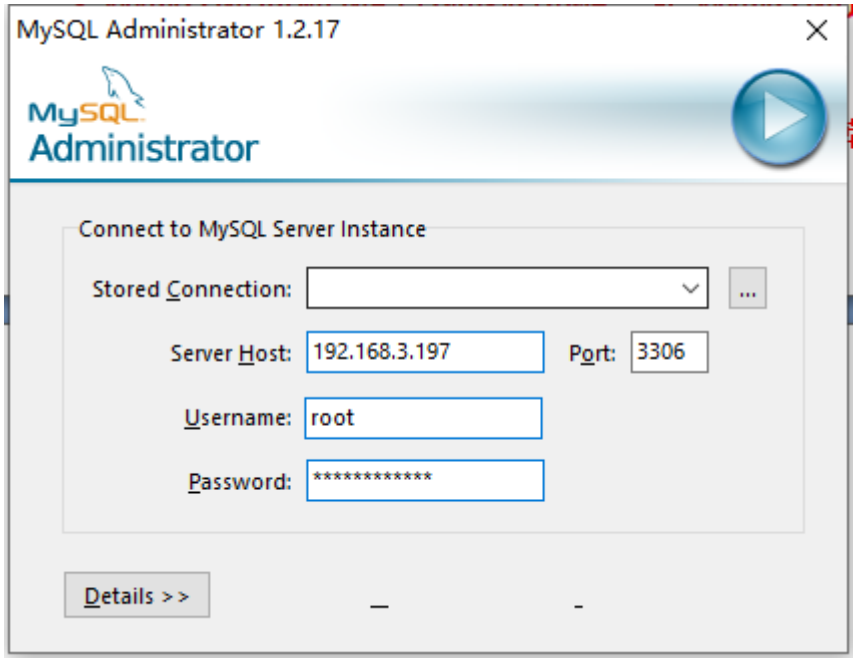
-v /root/dockerlab/mysqldata:/var/lib/mysql 将宿主机的/root/dockerlab/mysqldata 这个目录挂载到容器的/var/lib/mysql 目录（此目录为 mysql 默认的数据目录）

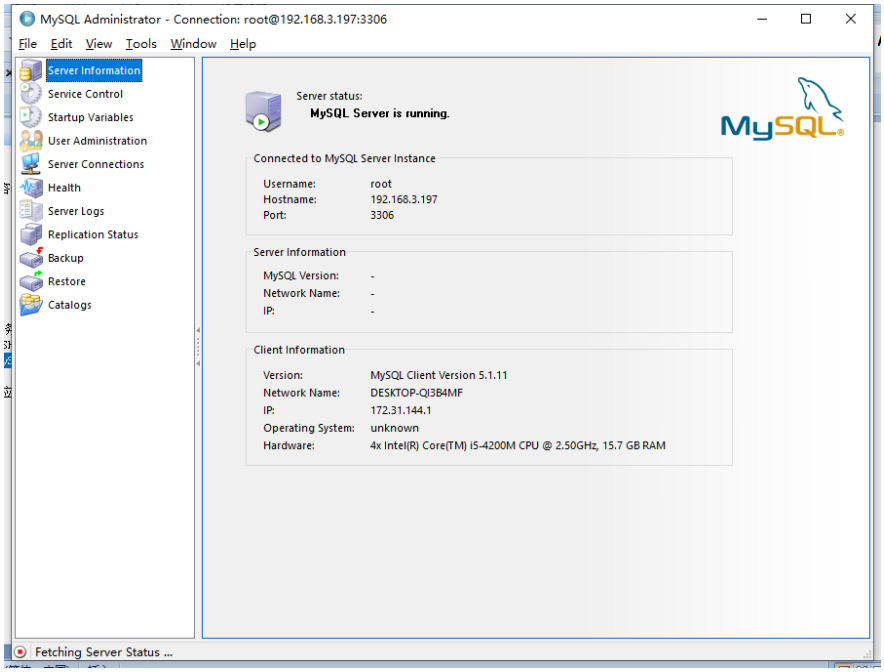
-e DATABASE_PASSWORD_ROOT='xxxxxxxxx'将 MYPASSWORDNEW 带入到容器中，它的值就是设置容器中 mysql 系统的 root 密码）

-p 3306:3306 将宿主机的 3306 端口映射到容器的 3306 端口（3306 为 mysql 的默认服务端口）

step6 测试容器中 mysql 是否启动成功

使用 mysql 连接工具去连接宿主机的 3306 端口





发现是可以连接的。

这样就说明我们的 mysql 已经成功在容器中运行起来了。

查看一下宿主机的 mysqldata 目录：

```
[root@docker197 dockerfile]# ls ../mysqldata/
auto.cnf        ca-key.pem        #ib_16384_0.dblwr  ib_logfile0   mysql           mysqlx.sock         public_key.pem   undo_001
binlog.000001   ca.pem            #ib_16384_1.dblwr  ib_logfile1   mysql.ibd       mysqlx.sock.lock    server-cert.pem  undo_002
binlog.000002   client-cert.pem   ib_buffer_pool     ibtmp1        mysql.sock      performance_schema  server-key.pem
binlog.index    client-key.pem    ibdata1            #innodb_temp  mysql.sock.lock private_key.pem     sys
```

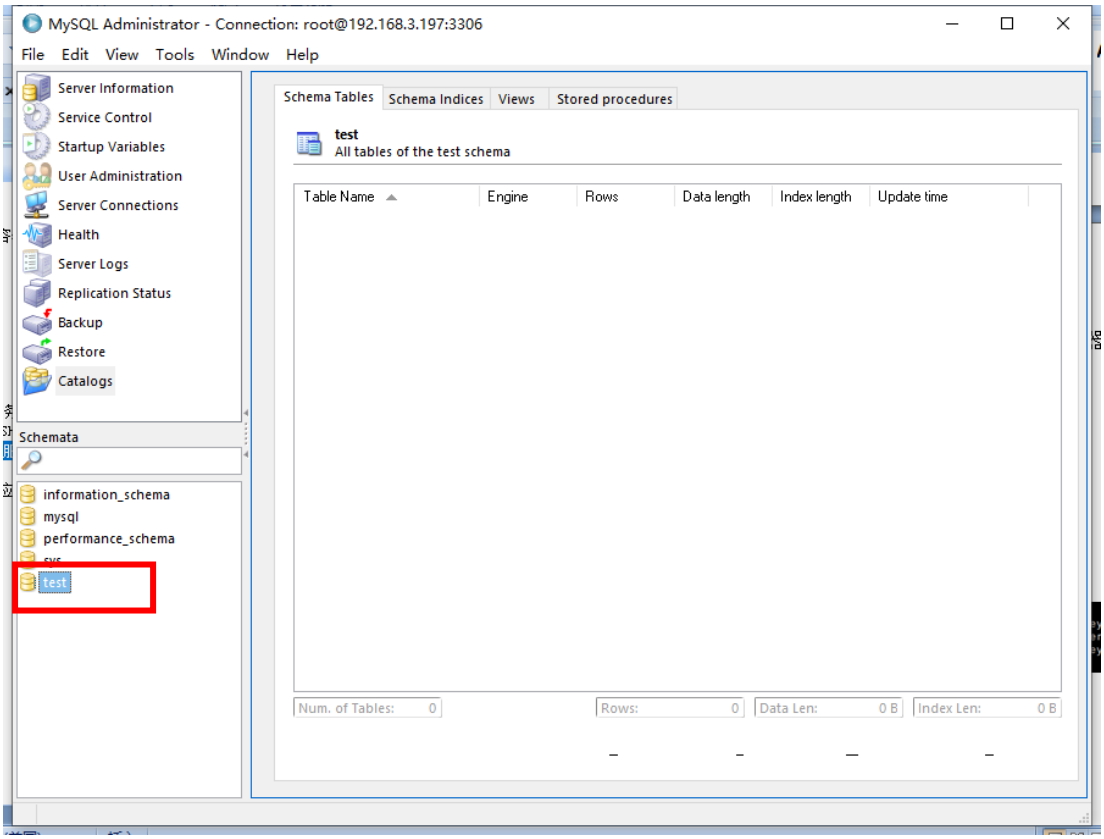
里面已经有数据了，这些数据就是容器中的 mysql 产生的数据库文件。

step 7 模拟容器崩溃后的数据恢复

由于容器中只跑了 mysql 的应用，mysql 的数据目录其实是在宿主机的，所以就算容器崩溃了，

数据目录其实还是存在的，也很容易恢复。

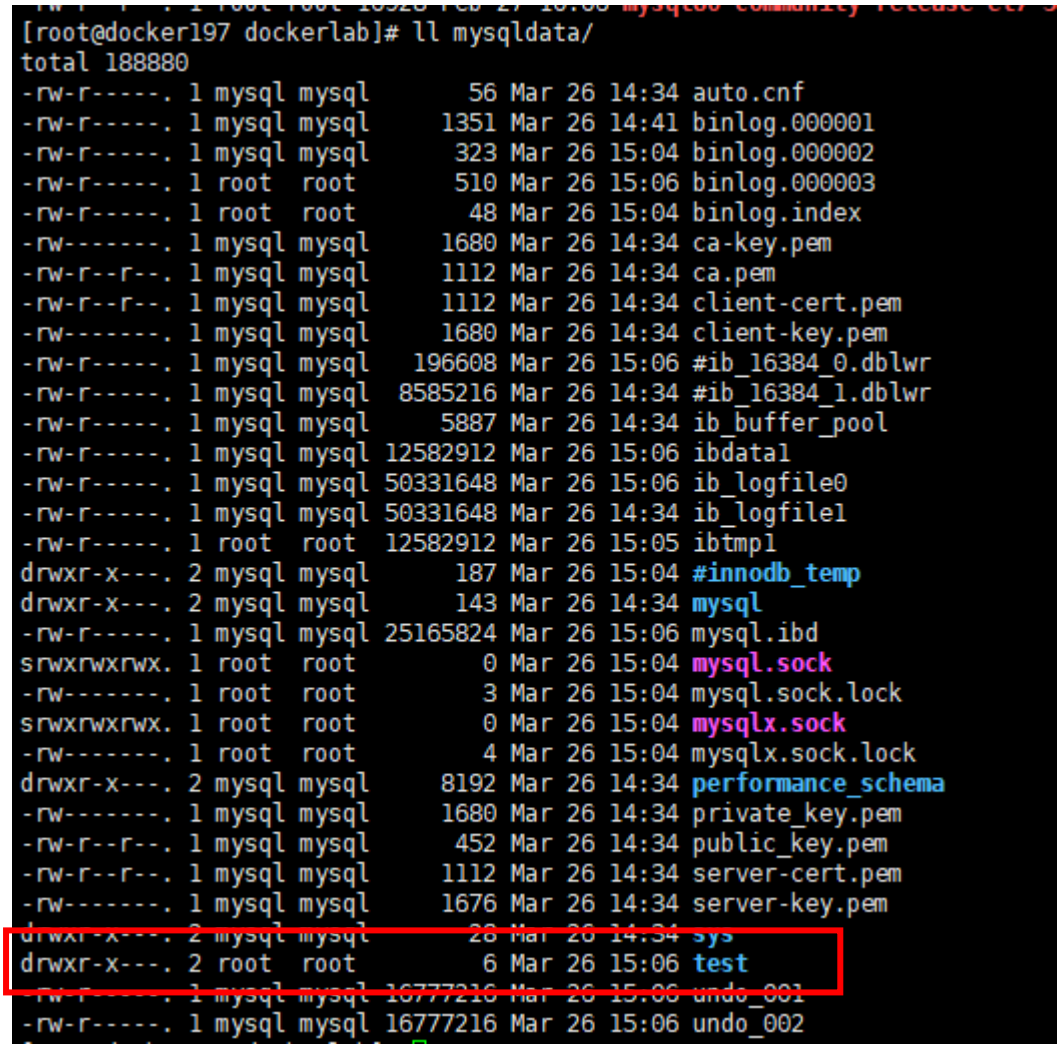
1） 使用 mysql 工具，在数据库中创建一个新的数据库，命名为 test



2） 我们先把刚创建的容器删掉；

```
[root@docker197 dockerfile]# docker rm -f mysql
mysql
```

3） 看一下宿主机中的 mysql 数据目录

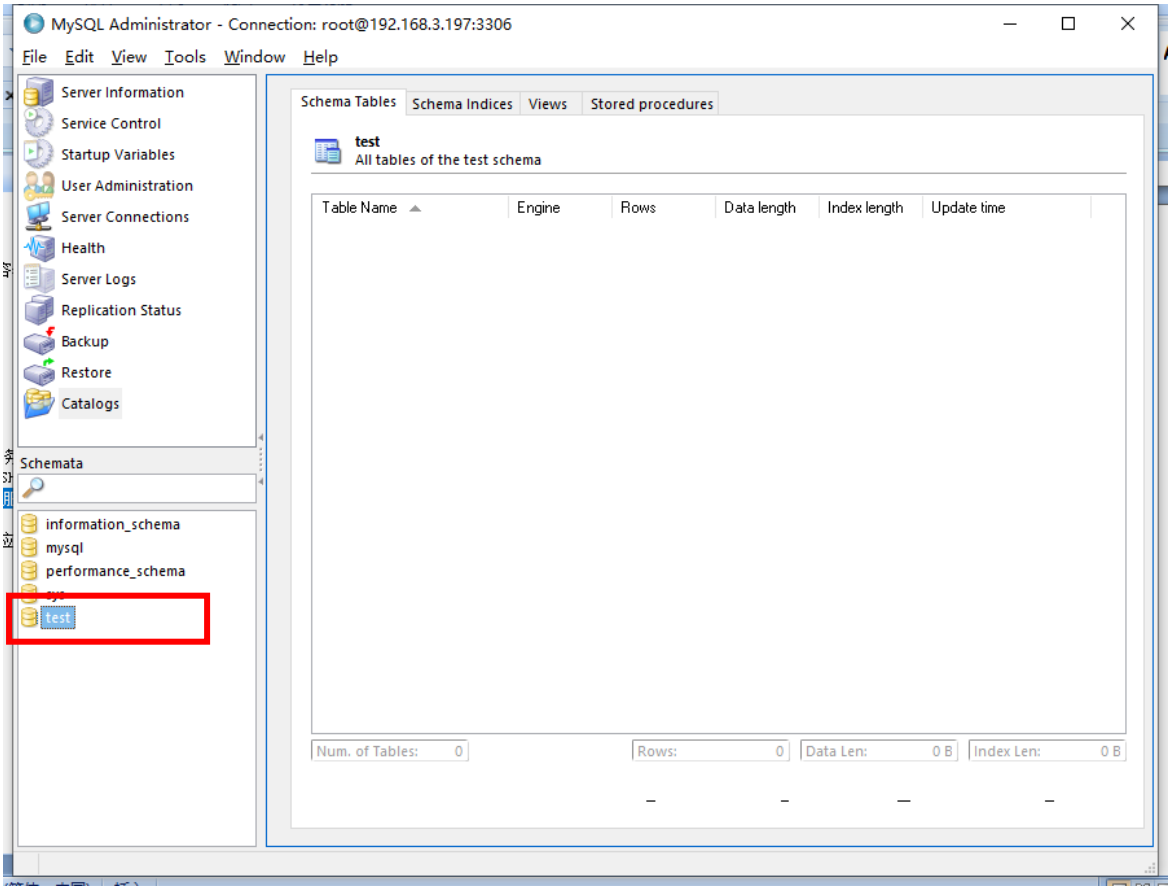


虽然容器已被删除，但其实数据文件都还在,上图红框内的就是我们刚刚新创建的那个数据库 test.

4)重新创建并启动容器

docker run -d --name=mysql --hostname=mysql -v /root/dockerlab/mysqldata:/var/lib/mysql -e DATABASE_PASSWORD_ROOT='xxxxxxxx' -p 3306:3306 mysql:1.0

5)再次用 mysql 连接工具，验证数据



这里我们发现,我们之前新建的那个 test 数据库，还是存在的。

4.4 综合实验：基于 docker 的 LNMP WEB 服务及应用

4.4.1 LNMP 概念

LNMP 动态网站部署架构是一套由 Linux + Nginx + MySQL + PHP 组成的动态网站系统解决方案（其各自的 Logo 见图 20-1）。LNMP 中的字母 L 是 Linux 系统的意思，不仅可以是 RHEL、CentOS、Fedora，还可以是 Debian、Ubuntu 等系统。

4.4.1 部署的思路

架构：创建两个容器，一个容器跑 nginx+PHP,一个容器跑 mysql，网站数据和数据库数据都存放在宿主机上。在 nginx 上跑一个 wordpress，用于博客网站的架设。

安装方式

mysql 就使用我们上一节中创建好的镜像来实现；

nginx 使用 yum 安装；

PHP 通过使用源码包生成 rpm 安装包，再进行安装；

wordpress 只需要将安装包拷贝到 nginx 目录下即可。

4.4.2 需要准备的资源

1）Centos 8 dnf 源

wget https://mirrors.aliyun.com/repo/Centos-vault-8.5.2111.repo

2） PHP 源码包

wget https://www.php.net/distributions/php-7.4.28.tar.gz

3)wordpress 安装包

wget https://cn.wordpress.org/latest-zh_CN.tar.gz

3） 编辑一个 nginx 配置文件 nginx.conf

```
        server {
            listen       80 default_server;
            listen       [::]:80 default_server;
            server_name  _;
            root         /usr/share/nginx/html;

            # Load configuration files for the default server block.
            include /etc/nginx/default.d/*.conf;
    #以下部分是定义根目录，和 index 文件
    location / {
            root html;
            index  index.php index.html index.htm;
            }
    #以下部分红色字体是用于 nginx 和 php 模块的连接
            location ~ \.php$ {
            root           html;
            fastcgi_pass   127.0.0.1:9000;
            fastcgi_index  index.php;
            fastcgi_param  SCRIPT_FILENAME  /usr/share/nginx/html$fastcgi_script_name;
            include        fastcgi_params;
            }

            location = /favicon.ico {
            log_not_found off;
            access_log off;
            }
            error_page 404 /404.html;
                location = /40x.html {
```

```
        }

        error_page 500 502 503 504 /50x.html;
            location = /50x.html {
        }
    }
```

4.4.3 编写 dockerfile

vim dockerfile-lnmp

```
FROM centos:latest#定义镜像来源基础包
LABEL auto build lnmp   #定义一个镜像标签
EXPOSE 80#定义开放的端口号
COPY ./Centos-vault-8.5.2111.repo /tmp/Centos-vault-8.5.2111.repo#将国内 yum 源文件拷入镜像
COPY ./lnmp_entrypoint.sh /root/lnmp_entrypoint.sh#将容器启动脚本拷入镜像
COPY ./nginx.conf /tmp/nginx.conf #将 nginx 配置文件拷入镜像
COPY ./php7.4/php-7.4.28.tar.gz /tmp/php-7.4.28.tar.gz#将 PHP 源码包拷入镜像
COPY ./wordpress/latest-zh_CN.tar.gz /tmp/latest-zh_CN.tar.gz#将 wordpress 源码包拷入镜像
RUN chmod 755 /root/lnmp_entrypoint.sh \#将启动脚本设为可执行
&& ln -s /root/lnmp_entrypoint.sh /usr/bin/lnmp_entrypoint.sh \#将启动脚本软链接到系统目录
&& rm -rf /etc/yum.repos.d/* \#删除原有 yum 源
&& mv /tmp/Centos-vault-8.5.2111.repo /etc/yum.repos.d/ \#将国内 yum 源文件拷到 yum 配置目录
&& dnf makecache \ #重建 yum 缓存
&& dnf install 'dnf-command(config-manager)' -y\ #安装 dnf 插件
&& dnf config-manager --set-enabled PowerTools -y\#开启 PowerTools
&& dnf makecache #重建 yum 缓存
RUN dnf install gcc libxml2-devel sqlite-devel libcurl-devel oniguruma-devel make -y \#安装依赖包
&& dnf install nginx -y \#安装 nginx
&& rm -rf /var/cache/yum/* #清除 yum 缓存
RUN cd /tmp \
&& tar -xzf php-7.4.28.tar.gz \#解压缩 php 源码包
&& cd /tmp/php-7.4.28 \
&& ./configure --prefix=/usr/local/php --enable-fpm --with-mysqli --with-curl --with-pdo_mysql
--with-pdo_sqlite --enable-mysqlnd --enable-mbstring \ #配置 PHP 编译环境
```

&& make install \#编译及安装 PHP

&& ln -s /usr/local/php/sbin/php-fpm /usr/sbin/php-fpm \#软链接 php 启动程序到系统目录

&& cp /tmp/php-7.4.28/php.ini-development /usr/local/php/lib/php.ini \#生成 php 配置文件 1：php.ini

&& mv /usr/local/php/etc/php-fpm.conf.default /usr/local/php/etc/php-fpm.conf \ #生成 php 配置文件 2：php-fpm.conf

&& mv /usr/local/php/etc/php-fpm.d/www.conf.default /usr/local/php/etc/php-fpm.d/www.conf#生成 php 配置文件 3：www.conf

RUN rm -f /etc/nginx/nginx.conf \#删除原有 nginx 配置文件

&& mv /tmp/nginx.conf /etc/nginx/ \#将配置好的 nginx 文件拷贝到 nginx 配置目录

&& rm -rf /usr/share/nginx/html/* \ #删除原有 nginx html 文件

&& cd /tmp \

&& tar -xzf latest-zh_CN.tar.gz \#解压缩 wordpress 源码包

&& rm -r latest-zh_CN.tar.gz #删除 wordpress 源码包

CMD lnmp_entrypoint.sh;sleep infinity#执行启动脚本，并让启动脚本常驻内存。

4.4.4 编写 lnmp_entrypoint 文件

vim lnmp_entrypoint

#!/bin/bash

mv /tmp/wordpress/* /usr/share/nginx/html/#将 wordpress 文件移入 nginx html 目录

cp /usr/share/nginx/html/wp-config-sample.php /usr/share/nginx/html/wp-config.php

#生成 wordpress 配置文件：wp-config.php

chown -R nginx:nginx /usr/share/nginx #设置 nginx 目录的拥有者为 nginx 用户

sed -i "s/database_name_here/$DATABASE_NAME/g" /usr/share/nginx/html/wp-config.php

#配置 wp-config.php 中的数据库名称为变量$DATABASE_NAME 的值，此变量的值在创建容器时由用户自定义。

sed -i "s/username_here/$DATABASE_USER/g" /usr/share/nginx/html/wp-config.php

#配置 wp-config.php 中的数据库用户名为变量$DATABASE_USER 的值，此变量的值在创建容器时由用户自定义。

sed -i "s/password_here/$DATABASE_PASSWORD/g" /usr/share/nginx/html/wp-config.php

#配置 wp-config.php 中的数据库密码为变量$DATABASE_PASSWORD 的值，此变量的值在创建容器时由用户自定义。

sed -i "s/localhost/$DATABASE_ADDRESS/g" /usr/share/nginx/html/wp-config.php

#配置 wp-config.php 中的数据库地址为变量$DATABASE_ADDRESS 的值，此变量的值在创建容器时由用户自定义。

php-fpm#启动 php 服务

nginx #启动 nginx 服务

4.4.5 创建 lnmp 镜像

在 dockerfile 文件的目录中执行以下命令：

docker build -t lnmp -f dockerfile-lnmp .#使用 dockerfile-lnmp 这个 dockerfile 来创建一个名为 lnmp 的镜像

```
Successfully built 38c19e45e1ad
Successfully tagged lump:latest
```

```
[root@docker197 lump]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
lump         latest    38c19e45e1ad   13 hours ago    1.4GB
```

4.4.6 创建数据库

在 4.3 小节里创建好的 mysql 容器中，创建一个名为 wordpress 的数据库。

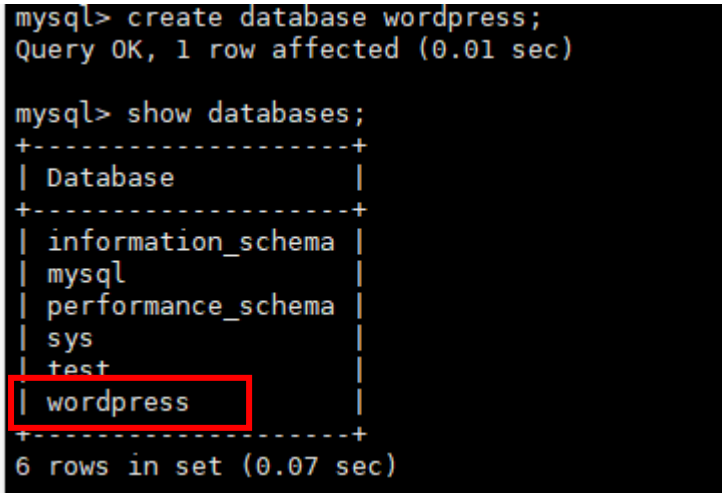
mysql -h 172.17.0.5 -u root –p　　#进入 mysql 系统，命令的中 IP 地址，是 mysql 容器的地址。

```
[root@docker197 lump]# mysql -h 172.17.0.5 -u root -p
Enter password:
ERROR 1045 (28000): Access denied for user 'root'@'_gateway' (using password: YES)
[root@docker197 lump]# mysql -h 172.17.0.5 -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 336
Server version: 8.0.26 Source distribution

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

创建一个数据库

create database wordpress;

使用“show databases;”命令查看数据库是否已创建成功；



4.4.7 创建并启动容器

docker run -d \

> --name lnmp \ #定义容器名称

> --hostname lnmp \#定义容器系统主机名

> -p 80:80 \#将宿主机的 80 端口映射到容器的 80 端口

> -e DATABASE_NAME='wordpress' \#定义 DATABASE_NAME 变量，此变量将被带入容器，并通过 lnmp_entrypoint.sh 启动脚本写入 wordpress 的配置文件中，作为 wordpress 使用的数据库名称。

> -e DATABASE_USER='root' \#定义 DATABASE_USER 变量，此变量将被带入容器，并通过 lnmp_entrypoint.sh 启动脚本写入 wordpress 的配置文件中，作为 wordpress 使用的数据库用户名。

> -e DATABASE_PASSWORD='123qweASD!@#' \#定义 DATABASE_PASSWORD 变量，此变量将被带入容器，并通过 lnmp_entrypoint.sh 启动脚本写入 wordpress 的配置文件中，作为 wordpress 使用的数据库密码。

> -e DATABASE_ADDRESS='172.17.0.5' \#定义 DATABASE_ADDRESS 变量，此变量将被带入容器，并通过 lnmp_entrypoint.sh 启动脚本写入 wordpress 的配置文件中，作为 wordpress 使用的数据库地址。

> -v /root/dockerlab/lnmp/html:/usr/share/nginx/html \#将主机中的一个目录映射到容器中的 nginx html 目录，作为 wordpress 的数据目录

> lnmp:latest#使用的镜像名称（就是 4.4.5 中创建的镜像）

```
[root@docker197 lump]# docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS         PORTS                                              NAMES
44bccad53443   lump:latest   "/bin/sh -c 'lump_en…"   7 seconds ago   Up 6 seconds   0.0.0.0:80->80/tcp, :::80->80/tcp                  lump
054545fea90f   mysql:1.0     "/bin/sh -c 'mysql_e…"   9 days ago      Up 37 hours    0.0.0.0:3306->3306/tcp, :::3306->3306/tcp          mysql
```

容器创建成功，并已正常运行。

4.4.8 配置并测试 wordpress

通过 web 浏览器连接打开 wordpress 的配置页面，进行配置

http://192.168.0.197/wp-admin/setup-config.php #这边的 ip 地址是宿主机的 ip

输入一些基本信息后，点击安装”wordpress”

安装成功，登录系统。

这样我们就成功地在容器中安装好了 wordpress。

wordpress 的具体配置和应用，不在我们的教程范围之内，大家有兴趣的话，可以去参考相关的资料。

五、Docker 网络配置

5.1 Docker 网络工作原理

Docker 服务启动时候会创建一个名为 docker0 的网桥，在启动容器的时候，会在宿主机和容器内各生成一个虚拟网卡。宿主机和容器内的虚拟网卡默认是连接到 docker0 这个网桥上的。所以容器

可以和宿主机或其它容器网络通信。

Docker0 的默认 IP 为 172.17.0.1

```
[root@docker197 ~]# ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:32:d9:65:6b  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

启动一个容器再观察：

```
[root@docker197 ~]# docker run -it --name centos centos bash
[root@e2cc86b9b40e /]# ifconfig
bash: ifconfig: command not found
[root@e2cc86b9b40e /]# ip address
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
9: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
[root@e2cc86b9b40e /]#
```

发现容器的 ip 是 172.17.0.2

```
[root@e2cc86b9b40e /]# ping 172.17.0.1
PING 172.17.0.1 (172.17.0.1) 56(84) bytes of data.
64 bytes from 172.17.0.1: icmp_seq=1 ttl=64 time=0.149 ms
64 bytes from 172.17.0.1: icmp_seq=2 ttl=64 time=0.155 ms
64 bytes from 172.17.0.1: icmp_seq=3 ttl=64 time=0.211 ms
64 bytes from 172.17.0.1: icmp_seq=4 ttl=64 time=0.205 ms
64 bytes from 172.17.0.1: icmp_seq=5 ttl=64 time=0.222 ms
^C64 bytes from 172.17.0.1: icmp_seq=6 ttl=64 time=0.244 ms
```

ping 宿主机，发现是可以 ping 通的

在宿主机上查看桥接信息

brctl show

```
[root@docker197 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
br-393380282e12         8000.0242bd98a375       no
docker0         8000.024232d9656b       no              veth12901b2
virbr0          8000.525400286ec0       yes             virbr0-nic
```

发现这个 docker0 上连接了一块虚拟网卡，这块虚拟网卡就是用来连接容器的。

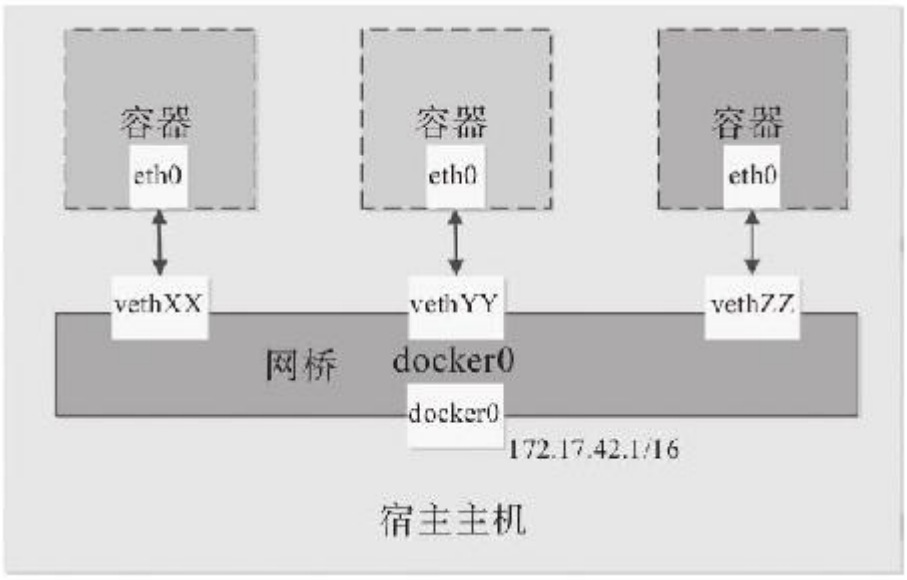
我们再开一个容器

```
[root@docker197 ~]# docker start mysql
mysql
[root@docker197 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
br-393380282e12         8000.0242bd98a375       no
docker0         8000.024232d9656b       no              veth12901b2
                                                        vetha0772a3
virbr0          8000.52540026fec0       yes             virbr0-nic
[root@docker197 ~]#
```

发现这个 docker0 上连接了两块虚拟网卡，分别用于连接两个容器

用 ip link show 这个命令也能看到这两块网卡。

```
[root@docker197 ~]# 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: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:6d:e3:5c brd ff:ff:ff:ff:ff:ff
3: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default qlen 1000
    link/ether 52:54:00:26:fe:c0 brd ff:ff:ff:ff:ff:ff
4: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 state DOWN mode DEFAULT group default qlen 1000
    link/ether 52:54:00:26:fe:c0 brd ff:ff:ff:ff:ff:ff
5: br-393380282e12: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:bd:98:a3:75 brd ff:ff:ff:ff:ff:ff
6: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:32:d9:65:6b brd ff:ff:ff:ff:ff:ff
10: veth12901b2@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 92:ea:fd:c4:16:72 brd ff:ff:ff:ff:ff:ff link-netnsid 0
12: vetha0772a3@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether c6:aa:57:4b:a5:d5 brd ff:ff:ff:ff:ff:ff link-netnsid 1
[root@docker197 ~]#
```



docker 容器工作示意图。。

那么，docker 又是如何可以连通宿主机以外的网络，外网又是如何可以访问到容器呢？

很简单，就是通过 iptables 的 nat 功能。

查看 iptables 的 nat 配置：

iptables -t nat –L

```
[root@docker197 ~]# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
PREROUTING_direct  all  --  anywhere             anywhere
PREROUTING_ZONES_SOURCE  all  --  anywhere             anywhere
PREROUTING_ZONES  all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
OUTPUT_direct  all  --  anywhere             anywhere
DOCKER     all  --  anywhere            !loopback/8           ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        anywhere
MASQUERADE  all  --  172.22.16.0/24       anywhere
RETURN     all  --  192.168.122.0/24     base-address.mcast.net/24
RETURN     all  --  192.168.122.0/24     255.255.255.255
MASQUERADE  tcp  --  192.168.122.0/24     !192.168.122.0/24     masq ports: 1024-65535
MASQUERADE  udp  --  192.168.122.0/24     !192.168.122.0/24     masq ports: 1024-65535
MASQUERADE  all  --  192.168.122.0/24     !192.168.122.0/24
POSTROUTING_direct  all  --  anywhere             anywhere
POSTROUTING_ZONES_SOURCE  all  --  anywhere             anywhere
POSTROUTING_ZONES  all  --  anywhere             anywhere
MASQUERADE  tcp  --  172.17.0.3           172.17.0.3           tcp dpt:mysql

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere
DNAT       tcp  --  anywhere             anywhere             tcp dpt:mysql to:172.17.0.3:3306
```

第一个红框内是源地址转换，意思是容器访问外网的时候，源地址转换为宿主机外网网卡的地址。

第二个红框内是目的地址转换，意思是外网访问宿主机 3306 端口时，iptables 外将目的地址转换为容器的 IP 地址。

我们之前学习过的 3.5.1 小节“将宿主机的端口映射到容器”这一章讲过的，在创建容器时加-p SPT:DPT 这个参数，期实就是容器启动时在 iptables 中添加了一条目的地址转换的策略。

5.2 Docker 网络的三种模式

bridge:桥接模式，容器网卡桥接到宿主机的桥上。（默认模式）

host:容器使用宿主机的网络配置。

none:容器不配置网卡

Docker 启动后，会自动创建三个网络，分别对应以上三种模式。

docker network ls

```
[root@docker197 ~]# docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
2ec91eb35c07   bridge    bridge    local
dbddcaffe8fd   host      host      local
70bdcf9cb289   none      null      local
```

host 模式：

创建一个容器，并指定使用 host 网络

docker run -it --name centos1 --hostname centos1 **--network host** centos bash

--network bridge|host|none　#这里可以指定使用哪个网络，默认为 bridge.

```
    inet 127.0.0.1/8 scope host lo
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: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:6d:e3:5c brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.197/24 brd 192.168.0.255 scope global noprefixroute ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::53ec:c8fa:78c2:b0cd/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default qlen 1000
    link/ether 52:54:00:26:fe:c0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.1/24 brd 192.168.122.255 scope global virbr0
       valid_lft forever preferred_lft forever
4: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 state DOWN group default qlen 1000
    link/ether 52:54:00:26:fe:c0 brd ff:ff:ff:ff:ff:ff
6: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:32:d9:65:6b 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::42:32ff:fed9:656b/64 scope link
       valid_lft forever preferred_lft forever
10: veth12901b2@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether 92:ea:fd:c4:16:72 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::90ea:fdff:fec4:1672/64 scope link
       valid_lft forever preferred_lft forever
12: vetha0772a3@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether c6:aa:57:4b:a5:d5 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::c4aa:57ff:fe4b:a5d5/64 scope link
       valid_lft forever preferred_lft forever
[root@centos1 /]# ^C
[root@centos1 /]#
```

可以看到，容器内的网络参数和宿主机是一样的。

NONE 模式：

docker run -it --name centos2 --hostname centos2 **--network none** centos bash

```
[root@docker197 ~]# docker run -it --name centos2 --hostname centos2 --network none centos bash
[root@centos2 /]# ip address
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
```

可以看到，容器内没有网卡。

5.3 新建 docker 网络

默认的网络：bridge，使用 docker0 做为网桥，默认使用 172.17.0.1/16，且似乎无法修改，且容器的 IP 地址只能由宿主机自动分配，没办法指定固定的 IP。

如果 docker0 的 IP 网段和我们宿主机的网段有冲突的话，那我们就没办法使用 docker 了。

因此我们需要新建一个 docker 网络。

docker network create mynet --subnet 192.168.100.0/24 --gateway 192.168.100.1

--subnet 192.168.100.0/24 将 mynet 的网段设为 192.168.100.0/24

--gateway 192.168.100.1 将 mynet 的网关设为 192.168.100.1（宿主机会取得这个地址）

```
[root@docker197 etc]# docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
2ec91eb35c07   bridge    bridge    local
dbddcaffe8fd   host      host      local
ea62d3eddd8a   mynet     bridge    local
70bdcf9cb289   none      null      local
```

这里会出现刚刚创建的那个网络。

```
[root@docker197 etc]# docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id": "ea62d3eddd8a095d48c7a63c5e2f3fcc9983c811f07914a1d62866c63baa6c23",
        "Created": "2022-04-18T14:46:36.517193447+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "192.168.100.0/24",
                    "Gateway": "192.168.100.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

```
[root@87f8ce1d545f /]# [root@docker197 etc]# ifconfig
br-ea62d3eddd8a: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.1  netmask 255.255.255.0  broadcast 192.168.100.255
        inet6 fe80::42:f4ff:feb4:e9ac  prefixlen 64  scopeid 0x20<link>
        ether 02:42:f4:b4:e9:ac  txqueuelen 0  (Ethernet)
        RX packets 20980  bytes 4178292 (3.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6365  bytes 936196 (914.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

宿主机上会多出一个网桥，并且 IP 地址是 192.168.100.1

5.4 自定义容器网络

现在新建一个容器，并将它连接到上一小节创建的网络 mynet 上。

docker run -it --name centos1 **--network mynet** centos bash

```
[root@docker197 etc]# docker run -it --name centos1 --network mynet centos bash
[root@87f8ce1d545f /]# ip address
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
20: eth0@if21: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:64:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.2/24 brd 192.168.100.255 scope global eth0
       valid_lft forever preferred_lft forever
```

发现容器的 IP 是 192.168.100.2

```
[root@87f8ce1d545f /]# ping 192.168.100.1
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=0.096 ms
64 bytes from 192.168.100.1: icmp_seq=2 ttl=64 time=0.087 ms
```

是可以 ping 通宿主机上，我们刚建的那个网桥的。

在宿主机上 brctl show，桥接已经建起来了。

```
[root@docker197 etc]# brctl show
bridge name     bridge id               STP enabled     interfaces
br-ea62d3eddd8a         8000.0242f4b4e9ac       no              vethdd35441
```

我们也可以更改已创建完成的容器的网络

docker network connect bridge centos1

以上命令，将我们刚刚创建的容器（已连接到 mynet，这个网络的），连接到默认的 bridge 上。

```
[root@docker197 etc]# docker exec -it centos1 bash
[root@87f8ce1d545f /]# ip address
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
20: eth0@if21: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:64:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.2/24 brd 192.168.100.255 scope global eth0
       valid_lft forever preferred_lft forever
22: eth1@if23: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.4/16 brd 172.17.255.255 scope global eth1
       valid_lft forever preferred_lft forever
[root@87f8ce1d545f /]#
```

容器的 IP 地址，又变回到默认 docker0 的网段了。

我们也可指定容器使用固定 IP 地址

docker run -it --name centos3 --network mynet --ip 192.168.100.100 centos bash

--network mynet：指定连接到 mynet 这个网络

--ip 192.168.100.100：指定容器的 IP 地址为 192.168.100.100.

注：当容器使用默认网络连接的时候，是不能指定容器的固定 IP 的；

指定的 IP 地址，必须和网桥在同一个网段。

```
[root@docker197 etc]# docker run -it --name centos3 --network mynet --ip 192.168.100.100 centos bash
[root@4765d5d24e3e /]# ip address
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
24: eth0@if25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:64:64 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.100/24 brd 192.168.100.255 scope global eth0
       valid_lft forever preferred_lft forever
```

5.5 使用 openvswitch 网桥

Docker 默认使用的是 Linux 自带的网桥实现，可以替换为使用功能更强大的 Openv-Switch 虚拟交换机实现。

Step 1 安装 openvswitch

1) 配置 yum 源

Openvswitch 的 yum 源，在系统默认的 yum 源里是没有的，需要安装一个 openstack yum 源

yum -y install centos-release-openstack-queens

yum makecache

2) 使用 yum 安装 openswitch

yum -y install openvswitch

3) 启动 openvswitch 服务，并设置为开机自启动

systemctl start openvswitch

systemctl enable openvswitch

4) 查看 openvswitch 状态

systemctl status openvswitch

```
[root@docker197 yum.repos.d]# systemctl status openvswitch
● openvswitch.service - Open vSwitch
   Loaded: loaded (/usr/lib/systemd/system/openvswitch.service; enabled; vendor preset: disabled)
   Active: active (exited) since Thu 2022-05-26 19:53:48 CST; 1min 59s ago
 Main PID: 2265 (code=exited, status=0/SUCCESS)

May 26 19:53:48 docker197.localdomain systemd[1]: Starting Open vSwitch...
May 26 19:53:48 docker197.localdomain systemd[1]: Started Open vSwitch.
```

Step 2 创建一个名为 ovsbr0 的网桥

ovs-vsctl add-br ovsbr0

查看一下创建好的网桥

```
[root@docker197 yum.repos.d]# ovs-vsctl show
e84797b9-7c1e-407a-8b60-74796b5d3a6e
    Bridge "ovsbr0"
        Port "ovsbr0"
            Interface "ovsbr0"
                type: internal
    ovs_version: "2.11.0"
```

创建成功

Step 3 使用特权模式创建并运行一下无网络的容器

docker run -it --name centos1 --privileged=true --network none centos bash

--privileged=true:使用特权模式，容器可以获得宿主机的 root 权限

--network none :设备容器为无网络模式

此时容器内只有一个回环网卡

```
[root@3c8317749e54 /]# 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
[root@3c8317749e54 /]#
```

Step 4 下载 OpenvSwitch 项目提供的支持 Docker 容器的辅助脚本 ovs-docker

wget https://github.com/openvswitch/ovs/raw/master/utilities/ovs-docker

给这个脚本加一下可执行权限

Chmod a+x ovs-docker

Step 5 给容器添加一块网卡 eth0,并桥接到 openvswitch 创建的桥 ovsrb0,并配置 IP 地址

./ovs-docker add-port ovsbr0 eth0 centos1 --ipaddress=172.16.0.2/24

到容器内看一下网络配置

docker exec -it centos1 bash

ip address

```
[root@3c8317749e54 /]# ip address
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
9: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether ca:c9:ba:6e:5f:19 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.16.0.2/24 scope global eth0
       valid_lft forever preferred_lft forever
```

这里可以看到，容器内已添加了一块网卡，并且有了 IP 地址

Step 6 在宿主机上配置一下 ovsbr0 的地址

ifconfig ovsbr0 172.16.0.1/24

```
[root@docker197 ovs-docker]# ifconfig ovsbr0
ovsbr0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.16.0.1  netmask 255.255.255.0  broadcast 172.16.0.255
        inet6 fe80::90c6:a9ff:fe5a:14d  prefixlen 64  scopeid 0x20<link>
        ether 92:c6:a9:5a:01:4d  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19  bytes 3512 (3.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

测试一下容器网络：

```
[root@docker197 ovs-docker]# ping 172.16.0.2
PING 172.16.0.2 (172.16.0.2) 56(84) bytes of data.
64 bytes from 172.16.0.2: icmp_seq=1 ttl=64 time=0.972 ms
64 bytes from 172.16.0.2: icmp_seq=2 ttl=64 time=1.35 ms
64 bytes from 172.16.0.2: icmp_seq=3 ttl=64 time=0.070 ms
^C
```

网络是通的

5.5 自定义 DOCKER_OPTS

可以使用 DOCKER_OPTS 对 docker 做更精细的管理。

Step1 :

修改 docker.sevice 文件

vim /usr/lib/systemd/system/docker.service

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
EnvironmentFile=-/etc/default/docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock $DOCKER_OPTS
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

以上红框内是需要加入的内容。

Step 2 创建/etc/default/docker 文件

```
[root@docker197 ~]# vim /etc/default/docker

DOCKER_OPTS="--bip 10.1.0.1/24"
~
```

这边加入一个 OPTS “--bip 10.1.0.1/24”

实验 1：修改 docker0 的默认 IP

系统默认的 docker0 的 IP 地址是 172.17.0.1/16，这里我们把这个默认 IP 改为 10.1.0.1/24

1）按照 step 2 中的操作，增加一个 OPTS “--bip 10.1.0.1/24”

2）重载服务配置文件，重启 docker 服务

systemctl daemon-reload

systemctl restart docker

3）查看 docker0 的 IP 地址

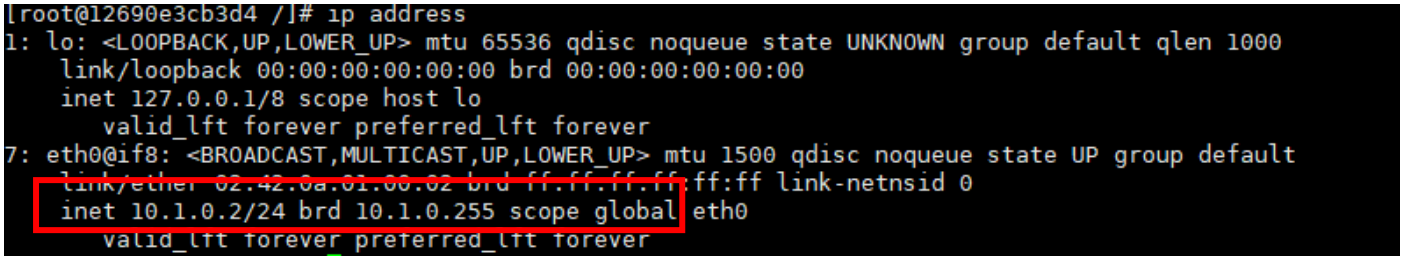
ifconfig docker0

```
[root@docker197 ~]# ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.1.0.1  netmask 255.255.255.0  broadcast 10.1.0.255
        ether 02:42:1c:b4:62:f1  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

发现 docker0 的 IP 地址已经变了。

4） 创建一个名为 centos3 的容器

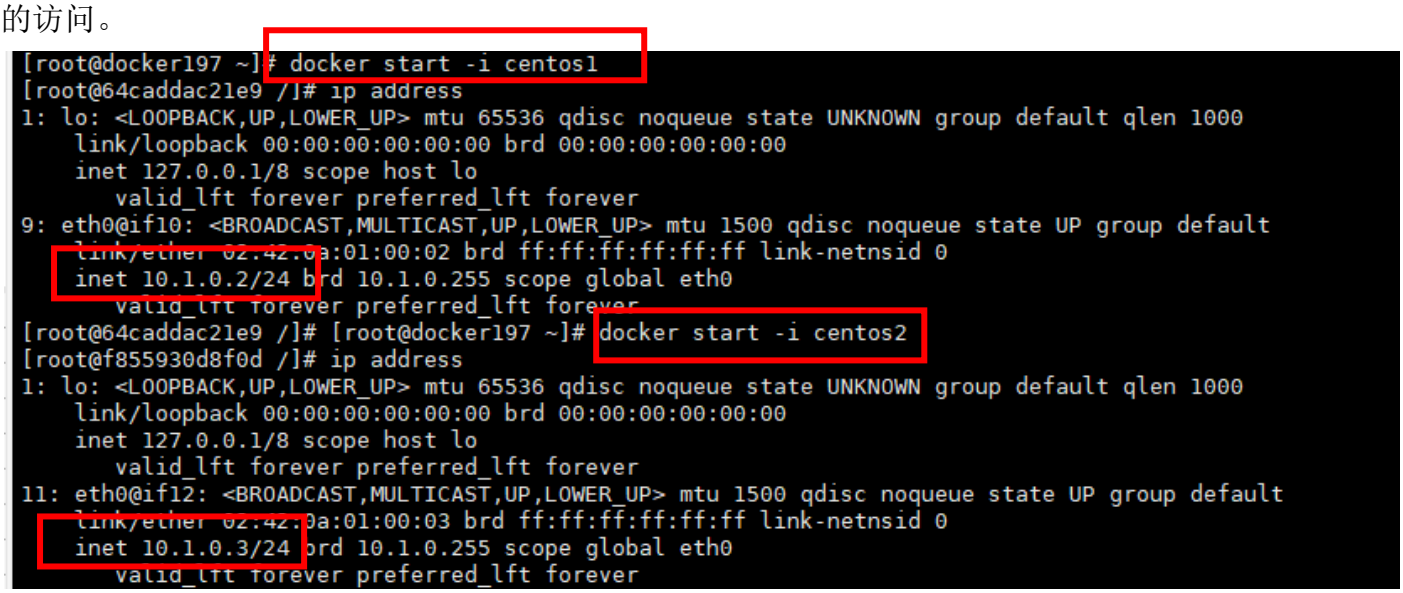
docker run -it --name centos3 centos bash



发现容器的 IP 地址变成 10.1.0.0 这个网段了

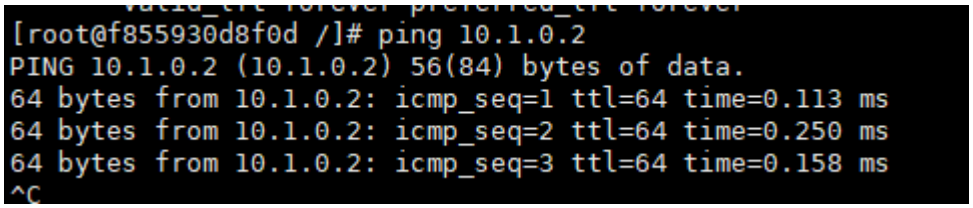
实验 2：让容器之间不能互通

默认情况下，容器之间是可以通过网络互相访问的，但有些情况下，为了安全起见，需要限制容器间的访问。



在配置前，我们先开两个容器，centos1 (IP:10.1.0.2),centos2(IP:10.1.0.3)

进入容器 centos2,去 ping centos1 的 IP



发现是可以通的。

然后我们在/etc/default/docker 这个文件里加入一个 OPTS “--icc=false”

清空 iptables

iptables -F

重载服务配置文件，重启 docker 服务

systemctl daemon-reload

systemctl restart docker

再次开启容器 centos1 和 centos2,并进入 centos2 ping centos1 ,看效果:

```
root@64caddac21e9 /]# [root@docker197 ~]# docker start -i centos2
root@f855930d8f0d /]# ping 10.1.0.2
ING 10.1.0.2 (10.1.0.2) 56(84) bytes of data.
```

发现已经 ping 不通了。

这个之所以 ping 不通了，是因为加了这个 OPTS 后，会在 iptables 中生成一条规则。

我们查看下 iptables 规则：

iptables -nF

```
Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER-USER  all  --  0.0.0.0/0            0.0.0.0/0
DOCKER-ISOLATION-STAGE-1  all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           ctstate RELATED,ESTABLISHED
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           ctstate RELATED,ESTABLISHED
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

iptables-save

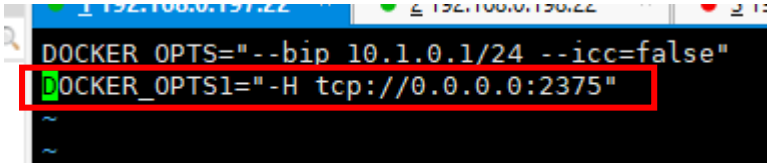
```
:OUTPUT_direct - [0:0]
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -o br-ea62d3eddd8a -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o br-ea62d3eddd8a -j DOCKER
-A FORWARD -i br-ea62d3eddd8a ! -o br-ea62d3eddd8a -j ACCEPT
-A FORWARD -i br-ea62d3eddd8a -o br-ea62d3eddd8a -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 -o docker0 -j DROP
```

5.6 配置 docker 远程服务

我们使用 docker 命令进行操作的时候，默认情况下是操作本地的 docker 服务，也可以通过配置，让 docker 监听一个 tcp 端口，以便让其它主机通过网络来进行操作。

5.6.1 通过 tcp:2375 端口实现 docker 远程服务

在/etc/default/docker 这个文件中，加入一个参数 DOCKER_OPTS="-H tcp://0.0.0.0:2375"



修改 docker.service

vim /usr/lib/systemd/system/docker.service

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
EnvironmentFile=-/etc/default/docker
ExecStart=/usr/bin/dockerd -H fd://  --containerd=/run/containerd/containerd.sock $DOCKER_OPTS1
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

增加红框内的参数。

在防火墙中放通 2375 端口

firewall-cmd --add-port=2375/tcp

firewall-cmd --add-port=2375/tcp --permanent

重载 daemon 参数，并重启 docker 服务

systemctl daemon-reload

systemctl restart docker

查看 docker 状态

systemctl status docker

```
[root@docker197 ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Mon 2022-04-25 10:10:02 CST; 12s ago
     Docs: https://docs.docker.com
 Main PID: 3236 (dockerd)
    Tasks: 9
   Memory: 36.8M
   CGroup: /system.slice/docker.service
           └─3236 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock -H tcp://0.0.0.0:2375

Apr 25 10:10:01 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:01.433463638+08:00" level=info msg="Firewalld: interface...rning"
Apr 25 10:10:01 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:01.474620379+08:00" level=info msg="Firewalld: interface...rning"
Apr 25 10:10:01 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:01.942413057+08:00" level=info msg="Default bridge (dock...dress"
Apr 25 10:10:02 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:02.113961361+08:00" level=info msg="Firewalld: interface...rning"
Apr 25 10:10:02 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:02.310676282+08:00" level=info msg="Loading containers: done."
Apr 25 10:10:02 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:02.350215002+08:00" level=info msg="Docker daemon" commi....10.12
Apr 25 10:10:02 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:02.351159212+08:00" level=info msg="Daemon has completed...ation"
Apr 25 10:10:02 docker197.localdomain systemd[1]: Started Docker Application Container Engine.
Apr 25 10:10:02 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:02.428961587+08:00" level=info msg="API listen on [::]:2375"
Apr 25 10:10:02 docker197.localdomain dockerd[3236]: time="2022-04-25T10:10:02.430647105+08:00" level=info msg="API listen on /var/r....sock"
Hint: Some lines were ellipsized, use -l to show in full.
```

红框内的这个参数，就是我们刚刚配置的参数。

查看监听端口

系统会监听 2375 端口

netstat –tupn

```
[root@docker197 ~]# netstat -tupln | grep dockerd
tcp6       0      0 :::2375                 :::*                    LISTEN      3236/dockerd
```

在时候在另一台主机上，就可以使用 docker 命令来管理此主机了。

```
[root@machine198 ~]# docker -H tcp://192.168.0.197 images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
registry                      2         2e200967d166   2 weeks ago     24.2MB
192.168.0.198:5000/lump       1.0       38c19e45e1ad   2 weeks ago     1.4GB
lump                          latest    38c19e45e1ad   2 weeks ago     1.4GB
192.168.0.198:5000/mysql      1.0       b91913db5402   4 weeks ago     540MB
mysql                         1.0       b91913db5402   4 weeks ago     540MB
192.168.0.198:5000/centos     1.0       5d0da3dc9764   7 months ago    231MB
centos                        latest    5d0da3dc9764   7 months ago    231MB
```

已经可以管理了。

提示：这种配置法是极其危险的，因为远程管理不需要任何的身份验证，所以只用作实验环境，生产环境绝对不可使用。

5.6.2 通过 TCP:2376 端口+TLS 方式实现 docker 远程服务

上一小节中，我们演示了，通过 2375 端口实现 docker 远程服务，但是这种方式风险较大，不建议使用。

这一小节中，我们介绍 TCP:2376+TLS 方式来实验安全地 docker 远程服务。

实验环境：

Server 端：192.168.0.197

Client 端：192.168.0.198

step 1 修改/etc/default/docker 文件

vim /etc/default/docker

在文件里输入以下内容：

DOCKER_OPTS="-H 0.0.0.0:2376 --tlsverify --tlscacert=/etc/docker/ca.pem --tlscert=/etc/docker/server-cert.pem --tlskey=/etc/docker/server-key.pem"

Step 2 修改/usr/lib/systemd/system/docker.service

vim /usr/lib/systemd/system/docker.service

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
EnvironmentFile=-/etc/default/docker
ExecStart=/usr/bin/dockerd -H fd:// -containerd=/run/containerd/containerd.sock $DOCKER_OPTS
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

增加红框中的内容:

第一个红框是指定环境变量文件为/etc/default/docker(就是我们在上一步中修改的文件)

第二个红框是引用上一步中定义的那个 OPTS

step 3 生成 TLS 相关证书和密钥:

此时虽然 2376 端口已经起来,但我们在 client 端,是无法访问 docker 的.

```
[root@machine198 ~]# docker -H 192.168.0.197:2376 images
Error response from daemon: Client sent an HTTP request to an HTTPS server.
```

我们先要生存 TLS 的相关证书和密钥:

进入/etc/docker 目录

cd /etc/docker

openssl genrsa -aes256 -out ca-key.pem 4096 #创建 CA 私钥

```
[root@machine198 docker]#  openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................
.....................................................................
..++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
[root@machine198 docker]# ll
total 4
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
```

第一个红框内需要我们输入一个密码.

这个命令执行完以后会生成一个 ca-key.pem 文件,就是 CA 私钥

openssl req -new -x509 -days 1000 -key ca-key.pem -sha256 -subj "/CN=*" -out ca.pem #用 CA 私钥创建一个 CA 证书

```
[root@machine198 docker]# openssl req -new -x509 -days 1000 -key ca-key.pem -sha256 -subj "/CN=*" -out ca.pem
Enter pass phrase for ca-key.pem:
[root@machine198 docker]# ll
total 8
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 19:22 ca.pem
```

第一个红框处需要输入 CA 的密码(就是我们上一步中输入的那个密码)

这里会生成一个 ca.pem 文件 ,就是 CA 证书.

openssl genrsa -out server-key.pem 4096　　#生成 server 证书私钥

```
[root@machine198 docker]# openssl genrsa -out server-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....
.....
.....
.....
e is 65537 (0x10001)
[root@machine198 docker]# ll
total 12
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 19:22 ca.pem
-rw-r--r--. 1 root root 3247 May 25 19:28 server-key.pem
```

openssl req -subj "/CN=*" -sha256 -new -key server-key.pem -out server.csr　#用 server 证书私钥生成一个证书请求.

```
[root@machine198 docker]#  openssl req -subj "/CN=*" -sha256 -new -key server-key.pem -out server.csr
[root@machine198 docker]# ll
total 16
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 19:22 ca.pem
-rw-r--r--. 1 root root 1574 May 25 19:30 server.csr
-rw-r--r--. 1 root root 3247 May 25 19:28 server-key.pem
```

openssl x509 -req -days 1000 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem #用 CA 密钥、CA 证书、server 证书请求生成 server 自签证书

```
[root@machine198 docker]# openssl x509 -req -days 1000 -sha256 -in server.csr -CA ca.pem  -CAkey ca-key.pem -CAcreateserial -out server-cert.pem
Signature ok
subject=/CN=*
Getting CA Private Key
Enter pass phrase for ca-key.pem:
[root@machine198 docker]# ll
total 24
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 19:22 ca.pem
-rw-r--r--. 1 root root   17 May 25 19:35 ca.srl
-rw-r--r--. 1 root root 1647 May 25 19:35 server-cert.pem
-rw-r--r--. 1 root root 1574 May 25 19:30 server.csr
-rw-r--r--. 1 root root 3247 May 25 19:28 server-key.pem
```

第一个红框处需要输入 CA 密钥的密码。

openssl genrsa -out key.pem 4096　　#生成 client 端证书私钥

```
[root@machine198 docker]#  openssl genrsa -out key.pem 4096
Generating RSA private key, 4096 bit long modulus
...............................++
.....................................................++
e is 65537 (0x10001)
[root@machine198 docker]# ll
total 28
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 19:22 ca.pem
-rw-r--r--. 1 root root   17 May 25 19:35 ca.srl
-rw-r--r--. 1 root root 3243 May 25 19:38 key.pem
-rw-r--r--. 1 root root 1647 May 25 19:35 server-cert.pem
-rw-r--r--. 1 root root 1574 May 25 19:30 server.csr
-rw-r--r--. 1 root root 3247 May 25 19:28 server-key.pem
```

openssl req -subj "/CN=client" -new -key key.pem -out client.csr #生成 client 端证书请求

```
[root@machine198 docker]#  openssl req -subj "/CN=client" -new -key key.pem -out client.csr
[root@machine198 docker]# ll
total 32
-rw-r--r--. 1 root root 3326 May 25 19:19 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 19:22 ca.pem
-rw-r--r--. 1 root root   17 May 25 19:35 ca.srl
-rw-r--r--. 1 root root 1582 May 25 19:40 client.csr
-rw-r--r--. 1 root root 3243 May 25 19:38 key.pem
-rw-r--r--. 1 root root 1647 May 25 19:35 server-cert.pem
-rw-r--r--. 1 root root 1574 May 25 19:30 server.csr
-rw-r--r--. 1 root root 3247 May 25 19:28 server-key.pem
[root@machine198 docker]#
```

echo extendedKeyUsage=clientAuth > extfile.cnf #创建配置文件，告知服务端，服务器要开 TLS 验证，为了产生客户端自签证书'

openssl x509 -req -days 1000-sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem

-extfile extfile.cnf #生成客户端自签证书

```
[root@docker197 docker]# openssl x509 -req -days 1000 -sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem -
extfile extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
[root@docker197 docker]# ll
total 40
-rw-r--r--. 1 root root 3326 May 25 20:04 ca-key.pem
-rw-r--r--. 1 root root 1765 May 25 20:05 ca.pem
-rw-r--r--. 1 root root   17 May 25 20:08 ca.srl
-rw-r--r--. 1 root root 1696 May 25 20:08 cert.pem
-rw-r--r--. 1 root root 1582 May 25 20:08 client.csr
-rw-r--r--. 1 root root   28 May 25 20:06 extfile.cnf
-rw-r--r--. 1 root root 3243 May 25 20:06 key.pem
-rw-r--r--. 1 root root 1647 May 25 20:06 server-cert.pem
-rw-r--r--. 1 root root 1574 May 25 20:06 server.csr
-rw-r--r--. 1 root root 3243 May 25 20:05 server-key.pem
```

第一个红框处需要输入 CA 密钥的密码。

这些命令执行完以后，会在/etc/docker 这个目录下生成几个 CA 文件

```
[root@docker197 ca]# ll -t /etc/docker/
total 40
-r--r--r--. 1 root root 1168 Apr 27 11:47 cert.pem
-rw-r--r--. 1 root root   17 Apr 27 11:47 ca.srl
-rw-r--r--. 1 root root   28 Apr 27 11:47 extfile.cnf
-r--------. 1 root root 1679 Apr 27 11:47 key.pem
-r--r--r--. 1 root root 1131 Apr 27 11:47 server-cert.pem
-r--------. 1 root root 1679 Apr 27 11:47 server-key.pem
-r--r--r--. 1 root root 1407 Apr 27 11:47 ca.pem
-r--------. 1 root root 1766 Apr 27 11:47 ca-key.pem
```

cert.pem:client 自签证书

key.pem:client 证书私钥

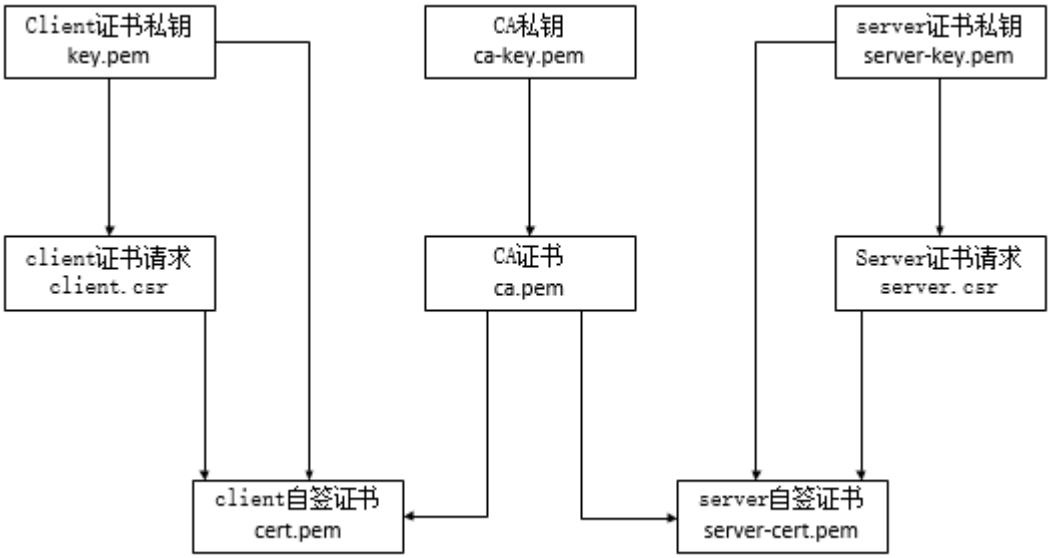
server-cert.pem:server 自签证书

server-key.pem:server 证书私钥

ca.pem:CA 证书

ca-key.pem:CA 私钥

以下是各个 CA 文件的依赖关系：



Step 4 重载服务配置,并重新启动 docker 服务

systemctl daemon-reload

systemctl restart docker

step 5 在防火墙中放通 2376/tcp 端口

firewall-cmd --add-port=2376 --permanent

firewall-cmd --add-port=2376

Step7:配置 client 端

在 client 端主机的/root 下面创建一个名为“.docker”的目录

mkdir /root/.docker　　#注意：目录名的第一个字段是一个”.”

将 server 端刚刚生成的的,ca.pem,cert.pem,key.pem 这三个文件，拷贝到 client 端的/root/.docker 下。

scp root@192.168.0.197:/etc/docker/xxxx　/root/.docker

```
[root@machine198 .docker]# ll
total 12
-rw-r--r--. 1 root root 1765 May 25 20:13 ca.pem
-rw-r--r--. 1 root root 1696 May 25 20:13 cert.pem
-rw-r--r--. 1 root root 3243 May 25 20:13 key.pem
[root@machine198 .docker]#
```

Step 8　在 client 端测试连接

docker -H 192.168.0.197:2376 --tls ps -a #执行 192.168.0.197 这个主机上的 docker ps –a　命令

```
[root@machine198 .docker]# docker -H 192.168.0.197:2376 --tls ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS                       PORTS     NAMES
12690e3cb3d4   centos         "bash"                   5 weeks ago     Exited (0) 5 weeks ago               centos3
f855930d8f0d   centos         "bash"                   5 weeks ago     Exited (1) 4 weeks ago               centos2
64caddac21e9   centos         "bash"                   5 weeks ago     Exited (0) 4 weeks ago               centos1
e2cc86b9b40e   centos         "bash"                   5 weeks ago     Exited (0) 5 weeks ago               centos
44bccad53443   lump:latest    "/bin/sh -c 'lump_en…"   7 weeks ago     Exited (137) 5 weeks ago             lump
054545fea90f   mysql:1.0      "/bin/sh -c 'mysql_e…"   2 months ago    Exited (137) 4 weeks ago             mysql
[root@machine198 .docker]# ^C
```

执行成功，说明配置成功。

六、Docker 相关开源项目

6.1 Etcd—高可用的键值数据库

6.1.1 etcd 概述

Etcd 是 CoreOS 团队于 2013 年 6 月发起的开源项目，它的目标是构建一个高可用的分布式键值（key-value）仓库，遵循 Apache v2 许可，基于 Go 语言实现。

Etcd 专门为集群环境设计，采用了更为简洁的 Raft 共识算法，同样可以实现数据强一致。

通常情况下，用户使用 Etcd 可以在多个节点上启动多个实例，并将它们添加为一个集群。同一个集群中的 Etcd 实例将会自动保持彼此信息的一致性，这意味着分布在各个节点上的应用也将获取到一致的信息。

6.1.2 安装并启动 etcd

yum install etcd -y

启动 etcd,并指定数据目录为/root/dockerlab/etcd-data

etcd --data-dir /root/dockerlab/etcd-data &

注：后面加一个&，是让它在后台运行

启动以后，会监听两个端口

2379：客户端请求端口

2380：其他节点连接端口

```
[root@docker197 ~]# netstat -tupln | grep etcd
tcp        0      0 127.0.0.1:2379          0.0.0.0:*               LISTEN      4837/etcd
tcp        0      0 127.0.0.1:2380          0.0.0.0:*               LISTEN      4837/etcd
```

查看群康健康状态

etcdctl cluster-health

```
[root@docker197 ~]# etcdctl cluster-health
member 8e9e05c52164694d is healthy: got healthy result from http://localhost:2379
cluster is healthy
```

以上状态说明是正常的。

6.1.3 etcd 基本操作

1）设置键值对

etcdctl set myetcd "hello word"

设置键值对 myetcd 的内容为 hello word

如在此命令后加：-ttl *time*,则表示键值的生存时间，单位为秒。

2）获取键值对内容

etcdctl get myetcd

```
[root@docker197 ~]# etcdctl get myetcd
hello word
```

会把键值对的内容打印出来

也可以用 curl 来获取键值对的内容

curl http://127.0.0.1:2379/v2/keys/myetcd

```
[root@docker197 ~]# curl http://127.0.0.1:2379/v2/keys/myetcd
{"action":"get","node":{"key":"/myetcd","value":"hello word","modifiedIndex":5,"createdIndex":5}}
```

3）更新键值的内容

etcdctl update myetcd world

将 myetcd 的内容更新为 world

```
[root@docker197 ~]# etcdctl update myetcd world
world
[root@docker197 ~]# etcdctl get myetcd
world
```

4）如果给定的键不存在，则创建一个新的键值,如果该键值存在，则报错。

etcdctl mk

```
[root@docker197 ~]# etcdctl mk myetcd1 "hello world"
hello world
[root@docker197 ~]# etcdctl mk myetcd1 "hello "
Error:  105: Key already exists (/myetcd1) [8]
```

5）删除键值

etcdctl rm

```
[root@docker197 ~]# etcdctl rm myetcd1
PrevNode.Value: hello world
[root@docker197 ~]# etcdctl get myetcd1
Error:  100: Key not found (/myetcd1) [9]
```

6）监测一个键值的变化，一旦键值发生更新，就会输出最新的值并退出。

etcdctl watch

```
[root@docker197 ~]# etcdctl watch myetcd
22222
```

etcdctl watch –f #一直监测，不退出。

```
[root@docker197 ~]# etcdctl watch -f myetcd
11111
22222
33333
```

7）监测一个键值的变化，一旦键值发生更新，自动执行一个命令。

etcdctl exec –watch *key* -- ‘command’

```
[root@docker197 ~]# etcdctl exec-watch myetcd -- 'ls'
1.txe                    Centos-8.repo    dockerlab                    GRANT         mysql      testt
1.txt                    Centos-8.repo.1  Documents                    initial-setup-ks.cfg  Pictures   Videos
alpine-standard-3.15.0-x86_64.iso  default.etcd     Downloads                    lump                    Public
anaconda-ks.cfg            Desktop        etcd-v3.3.1-linux-amd64.tar.gz  Music         Templates
1.txe                    Centos-8.repo    dockerlab                    GRANT         mysql      testt
1.txt                    Centos-8.repo.1  Documents                    initial-setup-ks.cfg  Pictures   Videos
alpine-standard-3.15.0-x86_64.iso  default.etcd     Downloads                    lump                    Public
anaconda-ks.cfg            Desktop        etcd-v3.3.1-linux-amd64.tar.gz  Music         Templates
```

以上命令：etcdctl　extc-watch myetcd　--'ls'的意思是当 myetcd 这个键值改变时，就执行 ls 命令。

8）列出所有键值和目录

etcdctl ls

```
[root@docker197 ~]# etcdctl ls
/myetcd
/myetcd1
```

etcdctl ls –r #同时显示子目录下的内容

```
[root@docker197 ~]# etcdctl ls -r
/myetcd1
/test
/test/111
/myetcd
```

9）创建一个目录

etcdctl mkdir *Directory*

```
[root@docker197 ~]# etcdctl mkdir test
[root@docker197 ~]# etcdctl ls
/myetcd
/myetcd1
/test
```

10）删除一个空目录,如果目录为非空，则会报错

etcdctl rmdir *Directory*

```
[root@docker197 ~]# etcdctl rmdir test
Error:  108: Directory not empty (/test) [19]
```

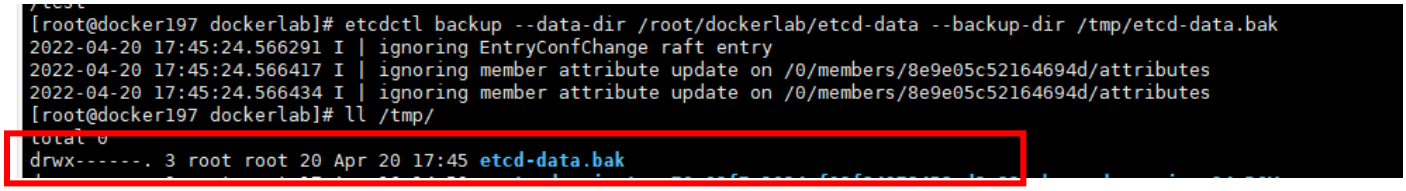
创建一个键目录，无论存在与否。

etcdctl setdir *Directory*

11）备份 etcd

etcdctl backup --data-dir /root/dockerlab/etcd-data --backup-dir /tmp/etcd-data.bak

将当前目录下的 etcd 数据库备份到/tmp 下面



6.1.4 用户管理

etcdctl user command [command options] [arguments...]

command:

add: 添加用户

list: 查看用户

remove: 删除用户

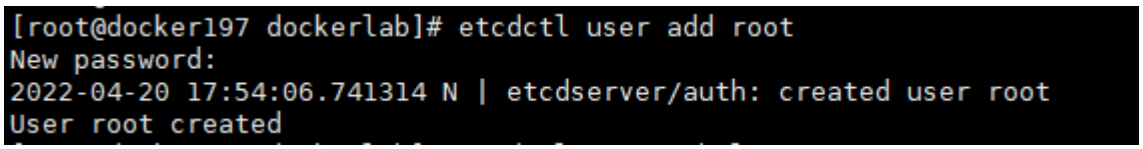
grant：添加用户到角色；

revoke：删除用户的角色；

passwd：修改用户的密码。

默认情况下，需要先创建（启用）root 用户作为 etcd 集群的最高权限管理员：

etcdctl user add root



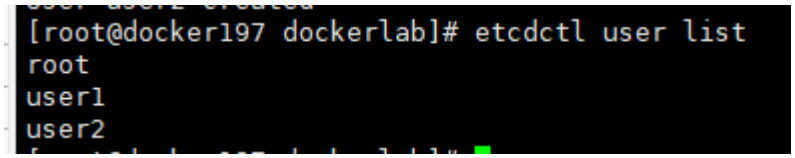
再创建两个普通用户：

etcdctl user add user1

etcdctl user add user2

查看用户：

etcdctl user list



将 user2 删除：

etcdctl user remove user2

```
[root@docker197 dockerlab]# etcdctl user remove user2
2022-04-20 18:00:36.626204 N | etcdserver/auth: deleted user user2
User user2 removed
[root@docker197 dockerlab]# etcdctl user list
root
user1
```

6.1.5 用户角色 role

etcdctl role command [command options] [arguments...]

- add：添加一个角色；
- get：查询角色详细信息；
- list：列出所有用户角色；
- remove：删除用户角色；
- grant：添加内容到角色控制，可以为 read、write 或者 readwrite；
- revoke：删除某路径的用户角色信息。

查看用户角色，默认只有 root 一个角色

etcdctl role list

```
[root@docker197 dockerlab]# etcdctl role list
root
```

查询 root 角色详细信息

etcdctl role get root

```
[root@docker197 dockerlab]# etcdctl role get root
Role: root
KV Read:
        /*
KV Write:
        /*
```

我们看到，root 这个角色对所有内容都有读和写的权限。

添加一个角色 guest,并设置此角色可以所有内容具有只读权限。

etcdctl role add guest

查看一下该角色的详细信息

```
[root@docker197 dockerlab]# etcdctl role get guest
Role: guest
KV Read:
KV Write:
```

它是没有任何权限的

etcdctl role grant guest -path　'/*' --read

加上权限后，再来看：

```
[root@docker197 dockerlab]# etcdctl role get guest
Role: guest
KV Read:
        /*
KV Write:
```

它已经对根目录下所有内容，具有只读权限，但没有写的权限。

将 user1 这个用户赋予 guest 角色

etcdctl user grant user1 --roles guest

```
[root@docker197 dockerlab]# etcdctl user get user1
User: user1
Roles:  guest
```

我们发现这个 user1 的角色已经是 guest 了。

启用访问验证：

etcdctl auth enable

```
[root@docker197 dockerlab]# etcdctl set test1 'yang'
2022-04-20 18:17:47.965880 W | etcdserver/api/v2http: auth: invalid access for unauthenticated user on resource /test1.
Error:  110: The request requires user authentication (Insufficient credentials) [0]
```

此时，已经不能添加键值了。

需要加入--username *user* 这个参数，比如我们用 root 用户来操作

etcdctl --username root set test1 'yang'

```
[root@docker197 dockerlab]# etcdctl --username root set test1 'yang'
Password:
```

这里会让我们输入密码

```
[root@docker197 dockerlab]# etcdctl ls
/test
/test1
```

这样才能创建成功。

6.1.6 使用 systemd 方式启动并管理 etcd

启动 etcd

systemctl start etcd

将 etcd 设为自启动

systemctl enable etcd

查看 etcd 服务的状态

systemctl status etcd

```
[root@etcd_node1 etcd]# systemctl status etcd
● etcd.service - Etcd Server
   Loaded: loaded (/usr/lib/systemd/system/etcd.service; enabled; vendor preset: disabled)
   Active: active (running) since Fri 2022-04-22 16:44:42 CST; 3min 10s ago
 Main PID: 3682 (etcd)
   CGroup: /system.slice/etcd.service
           └─3682 /usr/bin/etcd --name=default --data-dir=/var/lib/etcd/default.etcd --listen-client-urls=http://localhost:2379

Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: 8e9e05c52164694d received MsgVoteResp from 8e9e05c52164694d at term 2
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: 8e9e05c52164694d became leader at term 2
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: raft.node: 8e9e05c52164694d elected leader 8e9e05c52164694d at term 2
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: published {Name:default ClientURLs:[http://localhost:2379]} to cluster cdf818194e3a8c32
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: setting up the initial cluster version to 3.3
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: ready to serve client requests
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: serving insecure client requests on 127.0.0.1:2379, this is strongly discouraged!
Apr 22 16:44:42 etcd_node1.localdomain systemd[1]: Started Etcd Server.
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: set the initial cluster version to 3.3
Apr 22 16:44:42 etcd_node1.localdomain etcd[3682]: enabled capabilities for version 3.3
```

etcd 的配置文件

/etc/etcd/etcd.conf

#[Member]

#ETCD_CORS=""

ETCD_DATA_DIR="/var/lib/etcd/default.etcd"#默认是将数据库放在这个目录下

#ETCD_WAL_DIR=""

#ETCD_LISTEN_PEER_URLS="http://localhost:2380"

ETCD_LISTEN_CLIENT_URLS="http://localhost:2379"

#ETCD_MAX_SNAPSHOTS="5"

#ETCD_MAX_WALS="5"

ETCD_NAME="default"

#ETCD_SNAPSHOT_COUNT="100000"

#ETCD_HEARTBEAT_INTERVAL="100"

#ETCD_ELECTION_TIMEOUT="1000"

#ETCD_QUOTA_BACKEND_BYTES="0"

#ETCD_MAX_REQUEST_BYTES="1572864"

#ETCD_GRPC_KEEPALIVE_MIN_TIME="5s"

#ETCD_GRPC_KEEPALIVE_INTERVAL="2h0m0s"

#ETCD_GRPC_KEEPALIVE_TIMEOUT="20s"

6.1.7 etcd 集群

Etcd 的集群也采用了典型的“主-从”模型，通过 Raft 协议来保证在一段时间内有一个节点为主节点，其他节点为从节点。一旦主节点发生故障，其他节点可以自动再重新选举出新的主节点。

与其他分布式系统类似，集群中节点个数推荐为奇数个，最少为 3 个，此时 quorum 为 2，越多节点个数自然能提供更多的冗余性，但同时会带来写数据性能的下降。

Etcd 支持两种模式来构建集群：静态配置和动态发现。

6.1.7.1 静态方式构建集群

Step1:准备三个 NODE

etcd_node1:192.168.0.191

etcd_node2:192.168.0.192

etcd_node3:192.168.0.193

并将三个 NODE 的 hosts 文件配置好，保证网络互通。

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.0.191 etcd_node1
192.168.0.192 etcd_node2
192.168.0.193 etcd_node3
```

```
[root@etcd_node3 ~]# ping etcd_node2
PING etcd_node2 (192.168.0.192) 56(84) bytes of data.
64 bytes from etcd_node2 (192.168.0.192): icmp_seq=1 ttl=63 time=2.47 ms
64 bytes from etcd_node2 (192.168.0.192): icmp_seq=2 ttl=63 time=2.67 ms
```

step 2:在三个 NODE 上启动 etcd

配置/etc/etcd/etcd.conf 文件

vim /etc/etcd/etcd.conf

node1:

[Member]

ETCD_DATA_DIR="/var/lib/etcd/default.etcd" #定义数据库目录

ETCD_LISTEN_PEER_URLS=http://localhost:2380,http://192.168.0.191:2380#监听端口（给集群内其它成员的）

ETCD_LISTEN_CLIENT_URLS=http://localhost:2379,http://192.168.0.191:2379#监听端口（给客户端的）

ETCD_NAME="n1"#集群成员名字

[Clustering]

ETCD_INITIAL_ADVERTISE_PEER_URLS=http://etcd_node1:2380#给集群内其它成员访问的 URL

ETCD_ADVERTISE_CLIENT_URLS=http://etcd_node1:2379#给客户端访问的 URL

ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"#定义集群标记（所有 node 需一至）

ETCD_INITIAL_CLUSTER_STATE="new"#定义集群状态（new|existing）

ETCD_INITIAL_CLUSTER="n1=http://etcd_node1:2380,n2=http://etcd_node2:2380,n3=http://etcd_node3:2380"#定义群成员

node2:

[Member]

ETCD_DATA_DIR="/var/lib/etcd/default.etcd" #定义数据库目录

ETCD_LISTEN_PEER_URLS=http://localhost:2380,http://192.168.0.192:2380#监听端口（给集群内其它成员的）

ETCD_LISTEN_CLIENT_URLS=http://localhost:2379,http://192.168.0.192:2379#监听端口（给客户端的）

ETCD_NAME="n2"#集群成员名字

[Clustering]

ETCD_INITIAL_ADVERTISE_PEER_URLS=http://etcd_node2:2380#给集群内其它成员访问的 URL

ETCD_ADVERTISE_CLIENT_URLS=http://etcd_node2:2379#给客户端访问的 URL

ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"#定义集群标记（所有 node 需一至）

ETCD_INITIAL_CLUSTER_STATE="new"#定义集群状态（new|existing）

ETCD_INITIAL_CLUSTER="n1=http://etcd_node1:2380,n2=http://etcd_node2:2380,n3=http://etcd_node3:2380"#定义群成员

node3:

[Member]

ETCD_DATA_DIR="/var/lib/etcd/default.etcd" #定义数据库目录

ETCD_LISTEN_PEER_URLS=http://localhost:2380,http://192.168.0.193:2380#监听端口（给集群内其它成员的）

ETCD_LISTEN_CLIENT_URLS=http://localhost:2379,http://192.168.0.193:2379#监听端口（给客户端的）

ETCD_NAME="n2"#集群成员名字

[Clustering]

ETCD_INITIAL_ADVERTISE_PEER_URLS=http://etcd_node3:2380#给集群内其它成员访问的 URL

ETCD_ADVERTISE_CLIENT_URLS=http://etcd_node3:2379#给客户端访问的 URL

ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"#定义集群标记（所有 node 需一至）

ETCD_INITIAL_CLUSTER_STATE="new"#定义集群状态（new|existing）

ETCD_INITIAL_CLUSTER="n1=http://etcd_node1:2380,n2=http://etcd_node2:2380,n3=http://etcd_node3:2380"#定义群成员

放通三个 NODE 上的 2379/tcp 和 2380/tcp 端口

firewall-cmd --add-port 2379/tcp

firewall-cmd --add-port 2379/tcp –permanent

firewall-cmd --add-port 2380/tcp

firewall-cmd --add-port 2380/tcp –permanent

在三个 NODE 上启动 etcd

systemctl start etcd

检查集群的康健度：

etcdctl cluster-health

```
[root@etcd_node1 etcd]# etcdctl cluster-health
member 88a842acc16154b is healthy: got healthy result from http://etcd_node2:2379
member 6b850397db315739 is healthy: got healthy result from http://etcd_node3:2379
member b8a078188bd6783c is healthy: got healthy result from http://etcd_node1:2379
cluster is healthy
```

会出现三个 NODE，说明配置正确

在任意一个 NODE 上查看 member

etcdctl member list

```
[root@etcd_node1 etcd]# etcdctl member list
88a842acc16154b: name=n2 peerURLs=http://etcd_node2:2380 clientURLs=http://etcd_node2:2379 isLeader=false
6b850397db315739: name=n3 peerURLs=http://etcd_node3:2380 clientURLs=http://etcd_node3:2379 isLeader=true
b8a078188bd6783c: name=n1 peerURLs=http://etcd_node1:2380 clientURLs=http://etcd_node1:2379 isLeader=false
```

会出现三个 member,说明配置正确

在 node1 设置一个键值：

etcdctl set test “11111”

```
[root@etcd_node1 etcd]# etcdctl set test1 "111111"
111111
```

我们会发现在任何一个其它 NODE 上，都能看到这个键值

```
[root@etcd_node2 etcd]# etcdctl ls
/test1
```

说明集群已经建立起来了。

6.1.7.2 动态发现方式构建集群

静态配置的方法虽然简单，但是如果节点信息需要变动的时候，就需要手动进行修改。

很自然想到，可以通过动态发现的方法，让集群自动更新节点信息。要实现动态发现，首先需要一套支持动态发现的服务。

CoreOS 提供了一个公开的 Etcd 发现服务，地址在 https://discovery.etcd.io。

在公共 etcd 公共服务上获取一个 UUID

curl https://discovery.etcd.io/new?size=3

```
[root@etcd_node1 etcd]# curl https://discovery.etcd.io/new?size=3
https://discovery.etcd.io/6d2710590d93977d8a79c8390a42573e[root@etcd_node1 etcd]#
```

修改各 NODE 的 etcd 配置文件 /etc/etcd/etcd.conf

vim /etc/etcd/etcd.conf

[Member]

ETCD_DATA_DIR="/var/lib/etcd/default.etcd"

ETCD_LISTEN_PEER_URLS="http://localhost:2380,http://192.168.0.191:2380"

ETCD_LISTEN_CLIENT_URLS="http://localhost:2379,http://192.168.0.191:2379"

ETCD_NAME="n1"

[Clustering]

ETCD_INITIAL_ADVERTISE_PEER_URLS="http://etcd_node1:2380"

ETCD_ADVERTISE_CLIENT_URLS="http://etcd_node1:2379"

ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"

ETCD_INITIAL_CLUSTER_STATE="new"

ETCD_DISCOVERY=" https://discovery.etcd.io/e51ad532d2b8fa893bad7e7508b0c6c5"

只需要将 6.1.6.1 小节中的配置文件的最后一行删除，然后新增一行，像红框里一样。

ETCD_DISCOVERY 这个值的内容，就是上面我们通过 curl https://discovery.etcd.io/new?size=3 获取到的 UUID。

```
[root@etcd_node1 etcd]# etcdctl cluster-health
member 43539d885e16a96c is healthy: got healthy result from http://etcd_node3:2379
member b42828a23184ef90 is healthy: got healthy result from http://etcd_node2:2379
member c4c679dfa938aaa0 is healthy: got healthy result from http://etcd_node1:2379
cluster is healthy
```

集群创建成功。

6.1.7.3 集群参数配置

1）时钟同步

对于分布式集群来说，各个节点上的同步时钟十分重要，Etcd 集群需要各个节点时钟差异不超过 1s，否则可能会导致 Raft 协议的异常。

在各节点安装 NTP 服务。

yum install ntp

配置 NTP

vim /etc/ntpd.conf

For more information about this file, see the man pages

ntp.conf(5), ntp_acc(5), ntp_auth(5), ntp_clock(5), ntp_misc(5), ntp_mon(5).

driftfile /var/lib/ntp/drift

Permit time synchronization with our time source, but do not

permit the source to query or modify the service on this system.

restrict default nomodify notrap nopeer noquery

```
# Permit all access over the loopback interface.  This could
# be tightened as well, but to do so would effect some of
# the administrative functions.
restrict 127.0.0.1
restrict ::1

# Hosts on local network are less restricted.
#restrict 192.168.1.0 mask 255.255.255.0 nomodify notrap

# Use public servers from the pool.ntp.org project.
# Please consider joining the pool (http://www.pool.ntp.org/join.html).
server 0.centos.pool.ntp.org iburst
server 1.centos.pool.ntp.org iburst
server 2.centos.pool.ntp.org iburst
server 3.centos.pool.ntp.org iburst
```

红框的就是 ntp 服务器，可以选用网上的公共 ntp 服务器，也可以自己架设 ntp 服务器。

架设 ntp 服务器，不在本项目讲解范围之同，大家可以自己去参考相关资料。

总之，是要让群集内的 NODE 的时间保持一致。

启动，并将 ntp 服务设为自动启动

```
systemctl start ntpd
systemctl enable ntpd
```

2） 配置心跳时间

心跳消息时间间隔：主节点多久发一次心跳消息

选举时间间隔：从节点多久没收到心跳消息后发起重新选举

这两个参数可以在/etc/etcd/etcd.conf 里面配置

```
[Member]
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://localhost:2380,http://192.168.0.191:2380"
ETCD_LISTEN_CLIENT_URLS="http://localhost:2379,http://192.168.0.191:2379"
ETCD_NAME="n1"
ETCD_HEARTBEAT_INTERVAL="100"
ETCD_ELECTION_TIMEOUT="1000"

[Clustering]
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://etcd_node1:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://etcd_node1:2379"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_DISCOVERY="https://discovery.etcd.io/8bd738db9f21a7c6b59e1a217bdf0f27"
~
```

红框里的两个参数就是心跳时间间隔和选举时间间隔。通过选举时间间隔为心跳时间间隔的 5 倍以上。

3） 修改 snapshot 频率

Etcd 会定期将数据的修改存储为 snapshot，默认情况下每 10 000 次修改才会存一个 snapshot。

```
[Member]
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://localhost:2380,http://192.168.0.191:2380"
ETCD_LISTEN_CLIENT_URLS="http://localhost:2379,http://192.168.0.191:2379"
ETCD_NAME="n1"
ETCD_HEARTBEAT_INTERVAL="100"
ETCD_ELECTION_TIMEOUT="1000"
ETCD_SNAPSHOT_COUNT="100000"

[Clustering]
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://etcd_node1:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://etcd_node1:2379"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_DISCOVERY="https://discovery.etcd.io/8bd738db9f21a7c6b59e1a217bdf0f27"
~
~
```

在存储为 snap 的时候会有大量数据进行写入，影响 Etcd 的性能。建议将这个值调整的小一些，例如将这个数字改为 2000。

6.1.8 Docker 主机连接 etcd

我们准备两台 docker 主机，去连接 etcd.

Docker 主机 1：192.168.0.195

Docker 主机 2：192.168.0.196

Etcd 服务器：192.168.0.191

Step 1 按照之前的操作步骤，部署好 etcd 主机

防火墙放通 2379/tcp 端口

firewall-cmd --add-port=2379/tcp --permanent

firewall-cmd --add-port=2379/tcp

Step 2 修改 docker 的 DOCKER_OPTS 文件

Vim /etc/default/docker

```
DOCKER_OPTS="--cluster-store=etcd://192.168.0.191:2379 --cluster-advertise=ens33:2375"
```

--cluster-store=etcd://192.168.0.191:2379 #指向 ETCD 服务器

--cluster-advertise=ens33:2375 #指定一个宣告地址（宿主机的物理网卡）

Step 3 重载服务配置，并重启 docker

systemctl daemon-reload

systemctl restart docker

```
[root@docker197 ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Mon 2022-05-30 14:51:22 CST; 5min ago
     Docs: https://docs.docker.com
 Main PID: 5748 (dockerd)
    Tasks: 9
   Memory: 36.7M
   CGroup: /system.slice/docker.service
           └─5748 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --cluster-store=etcd://192.168.0.191:2379 --c...

May 30 14:51:21 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:21.548403109+08:00" level=info msg="Firewalld: inter...rning"
May 30 14:51:21 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:21.636624365+08:00" level=warning msg="Multi-Host ov...uster"
May 30 14:51:21 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:21.637597779+08:00" level=error msg="initializing serf ins...
May 30 14:51:21 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:21.840256806+08:00" level=info msg="Default bridge (...dress"
May 30 14:51:21 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:21.960182916+08:00" level=info msg="Firewalld: inter...rning"
May 30 14:51:22 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:22.088307367+08:00" level=info msg="Loading containe...done."
May 30 14:51:22 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:22.111361338+08:00" level=info msg="Docker daemon" c....10.16
May 30 14:51:22 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:22.112031595+08:00" level=info msg="Daemon has compl...ation"
May 30 14:51:22 docker197.localdomain systemd[1]: Started Docker Application Container Engine.
May 30 14:51:22 docker197.localdomain dockerd[5748]: time="2022-05-30T14:51:22.147092783+08:00" level=info msg="API listen on /v....sock"
Hint: Some lines were ellipsized, use -l to show in full.
[root@docker197 ~]# systemctl status docker --no
```

红框内，就是我们刚刚配的 docker opts

Step 4 查看 etcd 服务器上数据

etcdctl ls -r

```
[root@etcd_node1 ~]# etcdctl ls -r
/docker
/docker/nodes
/docker/nodes/192.168.0.195:2375
/docker/nodes/192.168.0.196:2375
/docker/network
/docker/network/v1.0
/docker/network/v1.0/network
/docker/network/v1.0/endpoint_count
/docker/network/v1.0/endpoint
/docker/network/v1.0/endpoint/801c6db1e2422ca0824f4136d5df0d3065916db669dfca6c4f1415935115fae5
/docker/network/v1.0/endpoint/8a0e97be0470c67a5bc1feddc8e603b0656401266b35f16ebd2c8c372298a591
/docker/network/v1.0/endpoint/669aa6540de3dcad1df913f1619b0d2fc80393eec00795cc1b182aa8aa4fe5a6
/docker/network/v1.0/endpoint/b45cb684455461bf35daed37a6c539afc77b8c1d92c7b7642d6ee81221b37018
/docker/network/v1.0/endpoint/1f7426580fd1533f2f0ff67fe35f7fe75a065581d4b9efcc5b54c1dca4f1e4fb
/docker/network/v1.0/endpoint/31dd7b4ef1c480cf4f7a976bca6b749e15a758e606660bb5032adc77b6e06e50
/docker/network/v1.0/endpoint/651e36b62aa9fce7ff3c5088bafb4c01b25f0c148406cadcb931f91843ed2f47
```

可以看到，etcd 上已有生成了很多 docker 的变量数据，并且已经有了两个节点的网络信息。

查看一下端口信息：

netstat –tupln

```
[root@docker195 ~]# netstat -tupln | grep docker
tcp        0      0 192.168.0.195:7946      0.0.0.0:*               LISTEN      1628/dockerd
udp        0      0 192.168.0.195:7946      0.0.0.0:*                           1628/dockerd
```

docker 服务监听了两个端口 7946/tcp 和 7946/udp，这两个端口是用来同步 docker 服务器信息用的端口，在防火墙中把这两个端口放通。

firewall-cmd --add-port=7946/tcp –permanent

firewall-cmd --add-port=7946/tcp

firewall-cmd --add-port=7946/udp –permanent

firewall-cmd --add-port=7946/udp

我们在 195 这台 docker 主机上创建一个 overlay 网络

docker network create -d overlay mynet

在两台 docker 主机上都会出现刚创建的这个网络

```
[root@docker195 ~]# docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
c7ba0f6612c4   bridge            bridge    local
8a0e97be0470   docker_gwbridge   bridge    local
31dd7b4ef1c4   host              host      local
87543a99a29f   mynet             overlay   global
1f7426580fd1   none              null      local
```

```
[root@docker196 ~]# docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
c685e30bb4cb   bridge            bridge    local
651e36b62aa9   docker_gwbridge   bridge    local
31dd7b4ef1c4   host              host      local
87543a99a29f   mynet             overlay   global
1f7426580fd1   none              null      local
```

这个信息就是通过 etcd 服务器来同步的。

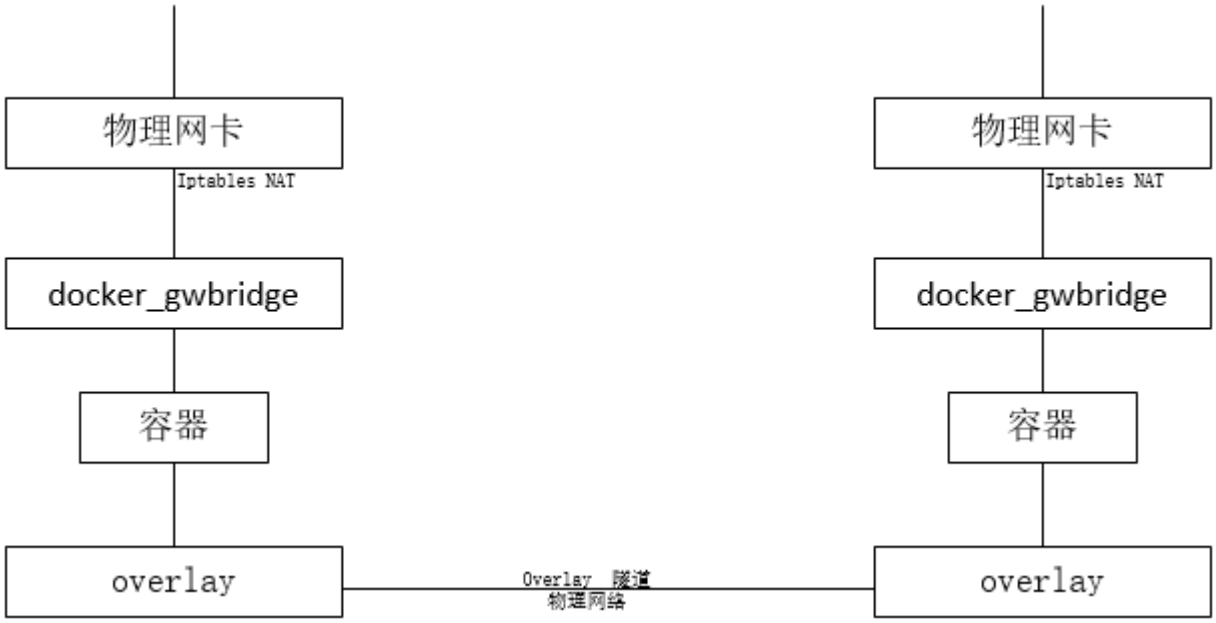
在 etcd 服务器上会出现相应的键值

etcd ls -r

```
[root@etcd_node1 ~]# etcdctl ls -r
/docker
/docker/network
/docker/network/v1.0
/docker/network/v1.0/network
/docker/network/v1.0/network/87543a99a29f5b6401506111a7b68c999e5c08093582d116ceb0c236fde946e3
/docker/network/v1.0/endpoint_count
/docker/network/v1.0/endpoint_count/87543a99a29f5b6401506111a7b68c999e5c08093582d116ceb0c236fde946e3
/docker/network/v1.0/endpoint
/docker/network/v1.0/endpoint/b45cb684455461bf35daed37a6c539afc77b8c1d92c7b7642d6ee81221b37018
/docker/network/v1.0/endpoint/1f7426580fd1533f2f0ff67fe35f7fe75a065581d4b9efcc5b54c1dca4f1e4fb
/docker/network/v1.0/endpoint/31dd7b4ef1c480cf4f7a976bca6b749e15a758e606660bb5032adc77b6e06e50
/docker/network/v1.0/endpoint/651e36b62aa9fce7ff3c5088bafb4c01b25f0c148406cadcb931f91843ed2f47
/docker/network/v1.0/endpoint/801c6db1e2422ca0824f4136d5df0d3065916db669dfca6c4f1415935115fae5
/docker/network/v1.0/endpoint/8a0e97be0470c67a5bc1feddc8e603b0656401266b35f16ebd2c8c372298a591
/docker/network/v1.0/endpoint/669aa6540de3dcad1df913f1619b0d2fc80393eec00795cc1b182aa8aa4fe5a6
/docker/network/v1.0/ipam
/docker/network/v1.0/ipam/default
/docker/network/v1.0/ipam/default/config
/docker/network/v1.0/ipam/default/config/GlobalDefault
/docker/network/v1.0/ipam/default/data
/docker/network/v1.0/ipam/default/data/GlobalDefault
/docker/network/v1.0/ipam/default/data/GlobalDefault/10.0.0.0
/docker/network/v1.0/ipam/default/data/GlobalDefault/10.0.0.0/24
/docker/network/v1.0/idm
/docker/network/v1.0/idm/vxlan-id
/docker/network/v1.0/overlay
/docker/network/v1.0/overlay/network
/docker/network/v1.0/overlay/network/87543a99a29f5b6401506111a7b68c999e5c08093582d116ceb0c236fde946e3
/docker/nodes
/docker/nodes/192.168.0.195:2375
/docker/nodes/192.168.0.196:2375
```

6.1.9 利用 etcd+docker overlay 网络实现容器跨主机通讯

overlay（覆盖）式网络会在多个 docker 守护进程所在的主机之间创建一个分布式的网络。这个网络会覆盖宿主机特有的网络，并允许容器连接它（包括集群服务中的容器）来安全通信。docker 会处理 docker 守护进程源容器和目标容器之间的数据报的路由。



创建好一个 overlay 网络以后，系统会自动创建一个 docker_gwbridge 网桥，这个网桥是用来处理容

器对外通讯的，默认 ip 地址为 172.18.0.1/16。

overlay 网络是一个虚拟网络，覆盖在各台 docker 主机之间，用于不同 docker 主机中容器的互通，默认 IP 地址是：10.0.0.1/24

在上一小节中，我们已经创建了一个名为 mynet 的 overlay 网络，现在我们在 192.168.0.195 这台主机上创建一个容器，名为 centos1,并将容器的网络设置为 mynet.

docker run -it --name centos1 --network mynet centos bash

这里的--network mynet 就是指定网络。

我们看一下容器的 ip 地址：

```
[root@f60d6fa6786e /]# 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
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.2/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
9: eth1@if10: <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 link-netnsid 1
    inet 172.18.0.2/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
```

这个容器有两个 ip 地址，分别为 10.0.0.2 和 172.18.0.2.

10.0.0.2 这个地址就是连接 overlay 网络的，用于容器间的互通。

172.18.0.2 这个地址是连接 docker_gwbridge 网桥，用于容器的对外通讯。

然后我们在 192.168.0.196 这台主机上也创建一个容器，名为 centos2, 并将容器的网络设置为 mynet.

docker run -it --name centos2 --network mynet centos bash

```
[root@4b019d196637 /]# 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
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.3/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
9: eth1@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 172.19.0.2/16 brd 172.19.255.255 scope global eth1
       valid_lft forever preferred_lft forever
```

这个容器有两个 ip 地址，分别为 10.0.0.3 和 172.19.0.2.

10.0.0.3 这个地址就是连接 overlay 网络的，用于容器间的互通。

172.19.0.2 这个地址是连接 docker_gwbridge 网桥，用于容器的对外通讯。

我们再看一下端口信息：

```
[root@docker196 ~]# netstat -tupln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1074/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      1234/master
tcp        0      0 192.168.0.196:7946      0.0.0.0:*               LISTEN      1633/dockerd
tcp6       0      0 :::22                   :::*                    LISTEN      1074/sshd
tcp6       0      0 ::1:25                  :::*                    LISTEN      1234/master
udp        0      0 0.0.0.0:4789            0.0.0.0:*                           -
udp        0      0 192.168.0.196:7946      0.0.0.0:*                           1633/dockerd
udp        0      0 127.0.0.1:323           0.0.0.0:*                           692/chronyd
udp6       0      0 ::1:323                 :::*                                692/chronyd
```

发现又开了一个端口 4789/udp,这个端口就是 overlay 网络的数据传输端口，需要在防火墙中把它放通。

firewall-cmd --add-port=4789/udp –permanent

firewall-cmd --add-port=4789/udp

这样我们在两台主机上的容器就可以互通了。

```
[root@f60d6fa6786e /]# ping centos2
PING centos2 (10.0.0.3) 56(84) bytes of data.
64 bytes from centos2.mynet (10.0.0.3): icmp_seq=1 ttl=64 time=1.13 ms
64 bytes from centos2.mynet (10.0.0.3): icmp_seq=2 ttl=64 time=0.852 ms
64 bytes from centos2.mynet (10.0.0.3): icmp_seq=3 ttl=64 time=1.20 ms
```

这边，我们在 centos1 这个容器里直接 ping centos2，发现 IP 被解析成 10.0.0.3（overlay 的网段，并且是可以通的。

6.1.10overlay 网络探究

通过上面的实验，我们可以看到，overlay 网络的网段是 10.0.0.0/24，但是我们的宿主机上是没有这个网段的。

```
[root@f60d6fa6786e /]# [root@docker195 netns]# 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: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:98:f1:ca brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.195/24 brd 192.168.0.255 scope global noprefixroute ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::6d8d:ad48:9430:7d00/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:ae:67:50:bb 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
4: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:41:ee:6b:7f brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global docker_gwbridge
       valid_lft forever preferred_lft forever
    inet6 fe80::42:41ff:feee:6b7f/64 scope link
       valid_lft forever preferred_lft forever
10: veth0b04708@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group defau
lt
    link/ether c2:66:18:2d:e5:b2 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::c066:18ff:fe2d:e5b2/64 scope link
       valid_lft forever preferred_lft forever
[root@docker195 netns]#
```

可以看到，在宿主机上并没有这个网段，那么这个网络在哪里呢？

其实它是存在于另外一个命名空间,我们进到/run/docker/netns 这个目录下，会发现这个目录下有三个文件。

```
[root@docker195 netns]# ll
total 0
-r--r--r-- 1 root root 0 Dec 15 21:05 1-87543a99a2
-r--r--r-- 1 root root 0 Dec 15 20:38 34a0a014bfa1
-r--r--r-- 1 root root 0 Dec 15 21:05 4761d9a9dc06
```

这个就是三个 docker 网络命名空间，我们可以用 nsenter 这个命令来看下它们的详细信息。

```
[root@docker195 netns]# nsenter --net=1-87543a99a2 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
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:4e:90:b7:87:60 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global br0
       valid_lft forever preferred_lft forever
6: vxlan0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UNKNOWN group default
    link/ether 02:4e:90:b7:87:60 brd ff:ff:ff:ff:ff:ff link-netnsid 0
8: veth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP group default
    link/ether 86:c6:b5:a8:f5:f0 brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

可以看到，这里有个 br0，IP 是 10.0.0.1,其实这个正是我们刚创建的 overlay 网络。

```
[root@docker195 netns]# docker network ls
NETWORK ID          NAME                DRIVER    SCOPE
c7ba0f6612c4        bridge              bridge    local
8a0e97be0470        docker_gwbridge     bridge    local
31dd7b4ef1c4        host                host      local
87543a99a29f        mynet               overlay   global
1f7426580fd1        none                null      local
```

注意上图中红框内 mynet 这个网络的 ID 号，跟/run/docker/netns 目录下的第一个文件名是差不多的。

再看一下另一个网络：

```
[root@docker195 netns]# nsenter --net=4761d9a9dc06 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
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.2/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
9: eth1@if10: <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 link-netnsid 1
    inet 172.18.0.2/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
```

这个网络里的 IP 地址信息和容器中的一模一样，其实这个网络就是给容器使用的，这个网络称为 sandbox(沙盒)。

6.2 Docker 三剑客之 Machine

Machine 项目是 Docker 官方的开源项目，负责实现对 Docker 运行环境进行安装和管理，特别在管理多个 Docker 环境时，使用 Machine 要比手动管理高效得多。

Machine 的定位是“在本地或者云环境中创建 Docker 主机”。

6.2.1 安装 machine

获取安装包

wget https://github.com/docker/machine/releases/download/v0.13.0/docker-machine-linux-x86_64

将此文件赋予可执行权限

chmod a+x docker-machine-linux-x86_64

创建软链接到/usr/sbin 下

ln -s /root/dockerlab/machine/docker-machine-linux-x86_64 /usr/sbin/docker-machine

这样，我们就可以不管在任何路径下，都可以执行 docker-machine 命令了。

6.2.2 生成第一台 machine

首先确保本地主机和目标主机网络互通，并且本地主机可以免密登入到目标主机。

实验环境：

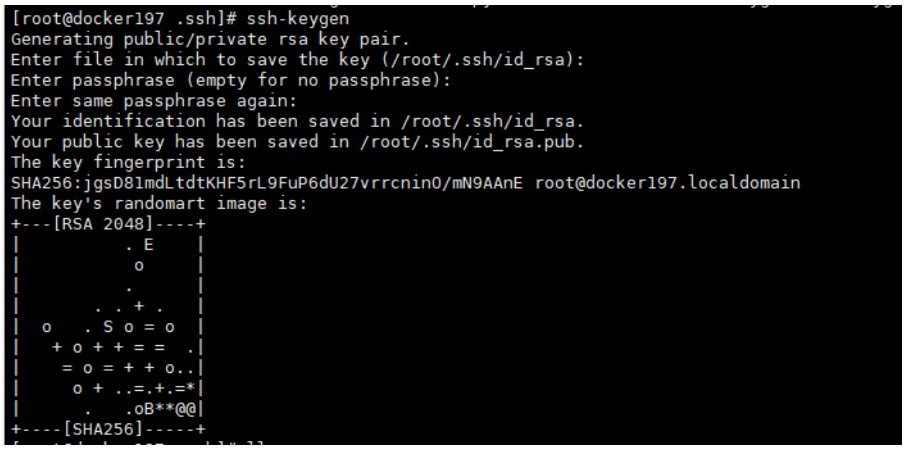
本地主机：192.168.0.197（就是安装了 machine 的那台主机）

目标主机 1：192.168.0.198

目标主机 2：192.168.0.199

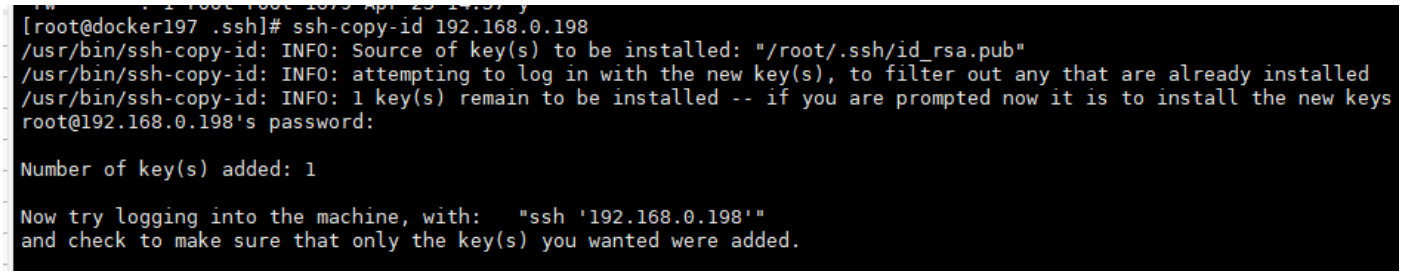
step1:在本地主机上生成 ssh 密钥对,并将公钥复制到目标主机

ssh-keygen



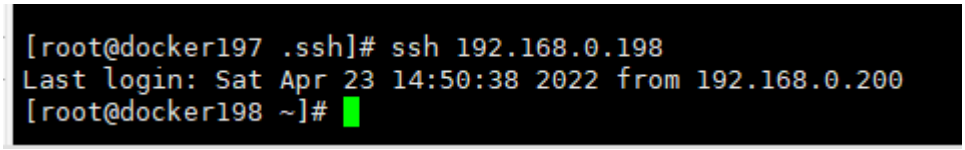
这边所有的选项都不管，直接打回车就可以。

ssh-copy-id <dest server ip>



这边执行过程中会让你输入目标主机的用户密码（我们这边使用 root 用户）

这样，我们就可以在本地主机上免密码登入到目标主机了。



这里不用输入密码了。

Step2 在目标主机上放通 2376/tcp 端口

firewall-cmd --add-port=2376/tcp --permanent

firewall-cmd --add-port=2376/tcp

step3 部署目标主机

docker-machine create -d generic --generic-ip-address=192.168.0.198 --generic-ssh-user=root machine198

命令注解：

create:创建一个 docker 主机

-d:驱动类型（generic|virtualbox）

Generic:普通类型（通过 SSH 创建主机）

Virtualbox:启动一个 virtualbox 虚拟机，并部署 docker

--generic-ip-adddress=:目标主机的 ip

--generic-ssh-user=:目标主机的用户名

```
[root@docker197 .ssh]# docker-machine create -d generic --generic-ip-address=192.168.0.198 --generic-ssh-user=root machine198
Running pre-create checks...
Creating machine...
(machine198) No SSH key specified. Assuming an existing key at the default location.
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with centos...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine env machine198
[root@docker197 .ssh]#
```

到目标主机上去看，发现 docker 服务已经起来了。

```
[root@docker198 ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
  Drop-In: /etc/systemd/system/docker.service.d
           └─10-machine.conf
   Active: active (running) since Sat 2022-04-23 19:04:44 CST; 1min 34s ago
     Docs: https://docs.docker.com
 Main PID: 56389 (dockerd)
    Tasks: 8
   Memory: 27.9M
   CGroup: /system.slice/docker.service
           └─56389 /usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock --storage-driver devicemapper --tlsverify --tlsca...

Apr 23 19:04:43 machine198 dockerd[56389]: time="2022-04-23T19:04:43.191740953+08:00" level=info msg="Firewalld: interface docker0 ...urning"
Apr 23 19:04:43 machine198 dockerd[56389]: time="2022-04-23T19:04:43.247067671+08:00" level=info msg="Firewalld: interface docker0 ...urning"
Apr 23 19:04:43 machine198 dockerd[56389]: time="2022-04-23T19:04:43.615555754+08:00" level=info msg="Default bridge (docker0) is a...ddress"
Apr 23 19:04:43 machine198 dockerd[56389]: time="2022-04-23T19:04:43.800308199+08:00" level=info msg="Firewalld: interface docker0 ...urning"
Apr 23 19:04:44 machine198 dockerd[56389]: time="2022-04-23T19:04:44.005249232+08:00" level=info msg="Loading containers: done."
Apr 23 19:04:44 machine198 dockerd[56389]: time="2022-04-23T19:04:44.031049123+08:00" level=info msg="Docker daemon" commit=87a90dc...0.10.14
Apr 23 19:04:44 machine198 dockerd[56389]: time="2022-04-23T19:04:44.031142181+08:00" level=info msg="Daemon has completed initialization"
Apr 23 19:04:44 machine198 systemd[1]: Started Docker Application Container Engine.
Apr 23 19:04:44 machine198 dockerd[56389]: time="2022-04-23T19:04:44.109705047+08:00" level=info msg="API listen on [::]:2376"
Apr 23 19:04:44 machine198 dockerd[56389]: time="2022-04-23T19:04:44.126055020+08:00" level=info msg="API listen on /var/run/docker.sock"
Hint: Some lines were ellipsized, use -l to show in full.
```

注意红框中的内容，目标主机上的 docker 已开启了 2376+tls，我们可以通过 tls 远程访问（详见 5.6.2 小节）

6.2.3 docker-machine 常用命令

1）docker –machine ls　　#列出所有 docker 主机

```
[root@docker197 machine]# docker-machine ls
NAME         ACTIVE   DRIVER    STATE     URL                         SWARM   DOCKER      ERRORS
machine198   -        generic   Running   tcp://192.168.0.198:2376            v20.10.16
machine199   -        generic   Running   tcp://192.168.0.199:2376            v20.10.16
```

这两台主机就是我们刚刚用 docker-machine create 创建的两台主机

2）docker-machine active　　#列出处于 active 的 docker 主机

激活状态意味着当前的 DOCKER_HOST 环境变量指向该主机。

```
[root@docker197 machine]# docker-machine active
No active host found
```

目前还没有主机处于 active 状态

我们将 machine198 激活，实际上就是将 docker-machine 的一些环境变量设成 machine198 的内容

eval "$(docker-machine env machine198)"

```
[root@docker197 machine]# docker-machine active
machine198
```

```
[root@docker197 machine]# docker-machine ls
NAME         ACTIVE   DRIVER    STATE     URL                         SWARM   DOCKER      ERRORS
machine198   *        generic   Running   tcp://192.168.0.198:2376            v20.10.16
machine199   -        generic   Running   tcp://192.168.0.199:2376            v20.10.16
```

可以看到，machine198 已经处于 active 状态

这时候我们在 docker-machine 这台主机上操作 docker，就相当于是在 machine198 这台主机的操作。

我们来验证一下：

我们在 197（安装 docker-machine）的主机上拉取一个 busybox 的镜像

```
[root@docker197 machine]# docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
Digest: sha256:de56395ae0788e364797f0c60464d4693c43c33cc04ec26fc3b0931b2e7c9d7d
Status: Image is up to date for busybox:latest
docker.io/library/busybox:latest
[root@docker197 machine]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
busybox      latest    560956ac186f   14 hours ago    1.24MB
```

然后到 machine198 这台主机上云看一下

```
[root@docker198 ~]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
busybox      latest    560956ac186f   14 hours ago    1.24MB
```

发现是一样的。

4） docker-machine config <host>　　#显示 docker 主机的配置信息

docker-machine config machine198

```
[root@docker197 machine]# docker-machine config machine198
--tlsverify
--tlscacert="/root/.docker/machine/machines/machine198/ca.pem"
--tlscert="/root/.docker/machine/machines/machine198/cert.pem"
--tlskey="/root/.docker/machine/machines/machine198/key.pem"
-H=tcp://192.168.0.198:2376
```

5) docker-machine config <host>　　#显示连接到某个主机需要的环境变量。

```
[root@docker197 machine]# docker-machine env machine198
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.0.198:2376"
export DOCKER_CERT_PATH="/root/.docker/machine/machines/machine198"
export DOCKER_MACHINE_NAME="machine198"
# Run this command to configure your shell:
# eval $(docker-machine env machine198)
```

注意红框内的提示，意思是用 eval "$(docker-machine env machine198)"这个命令来配置你的 shell，就是我们上面第二部分，讲 docker-machine active 时候用的这个命令。

这个命令相当于是执行了以下几条命令：

export DOCKER_TLS_VERIFY="1"

export DOCKER_HOST="tcp://192.168.0.198:2376"

export DOCKER_CERT_PATH="/root/.docker/machine/machines/machine198"

export DOCKER_MACHINE_NAME="machine198"

这个 eval,相当于是把 docker-machine env machine198 这条命令获取的输出，转换成了命令。

这时候，我们看一下环境变量：

```
[root@docker197 machine]# env | grep DOCKER
DOCKER_HOST=tcp://192.168.0.198:2376
DOCKER_MACHINE_NAME=machine198
DOCKER_TLS_VERIFY=1
DOCKER_CERT_PATH=/root/.docker/machine/machines/machine198
```

环境变量都已成变成 machine198 的参数了，把我们执行 docker 命令就相当是在 machine198 那台主机上执行是一样的。

6）docker-machine inspect <host>　　#输出 docker 主机的详细信息

```
[root@docker197 ~]# docker-machine inspect machine198
{
    "ConfigVersion": 3,
    "Driver": {
        "IPAddress": "192.168.0.198",
        "MachineName": "machine198",
        "SSHUser": "root",
        "SSHPort": 22,
        "SSHKeyPath": "",
        "StorePath": "/root/.docker/machine",
        "SwarmMaster": false,
        "SwarmHost": "",
        "SwarmDiscovery": "",
        "EnginePort": 2376,
        "SSHKey": ""
```

7）docker-machine ip <host>获取 docker 主机的 IP

```
[root@docker197 machine]# docker-machine ip machine198
192.168.0.198
```

8）docker-machine kill<host>#杀死 docker 主机

```
[root@docker197 machine]# docker-machine kill machine198
Killing "machine198"...
generic driver does not support kill
```

Genric 模式不支持，只有在虚拟模式下才支持此功能。

9）docker-machine restart <host>　　#重启 docker 主机

```
[root@docker197 machine]# docker-machine restart machine198
Restarting "machine198"...
ssh command error:
command : sudo shutdown -r now
err     : exit status 255
output  : Connection to 192.168.0.198 closed by remote host.
```

相当于执行了一下 shutdown –r now 这个命令

10）docker-machine　ssh　<host>　　#通过 ssh 登入 docker 主机

```
[root@docker197 machine]# docker-machine ssh machine198
Last login: Fri Jun  3 15:16:59 2022 from 192.168.0.197
[root@machine198 ~]#
```

11）docker-machine status　<host>　　#获取 docker 主机的状态

```
[root@docker197 machine]# docker-machine status machine198
Running
```

12）docker -H <ip>:2376 --tls <docker command>　　#远程执行 docker 命令

```
[root@docker197 machine]# docker -H 192.168.0.198:2376 --tls images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
busybox       latest    560956ac186f   15 hours ago    1.24MB
```

这个功能和 5.6.1 小节中介绍的 TLS 方式 docker 远程管理，是一样的。

6.3 Docker 三剑客之 compose

Compose 项目是 Docker 官方的开源项目，负责实现对基于 Docker 容器的多应用服务的快速编排。Compose 定位是“定义和运行多个 Docker 容器的应用”。

Compose 中的几个重要的概念：

1) 任务（task）：一个容器被称为一个任务。任务拥有独一无二的 ID，在同一个服务中的多个任务序号依次递增。
2) 服务（service）：某个相同应用镜像的容器副本集合，一个服务可以横向扩展为多个容器实例。
3) 服务栈（stack）：由多个服务组成，相互配合完成特定业务，如 Web 应用服务、数据库服务共同构成 Web 服务栈，一般由一个 docker-compose.yml 文件定义。

6.3.1 安装 compose

yum install docker-compose

查看一下版本号：

docker-compose version

```
[root@docker197 ~]# docker-compose version
docker-compose version 1.18.0, build 8dd22a9
docker-py version: 2.6.1
CPython version: 3.6.8
OpenSSL version: OpenSSL 1.0.2k-fips  26 Jan 2017
```

6.3.2 compose 模版文件

默认的模板文件名称为 docker-compose.yml，格式为 YAML 格式，目前最新的版本为 v3。

模版常用命令

6.3.2.1 version: ""命令

#指定版号,现在我们使用的版本号是 3

6.3.2.2 services 命令

#代表以下的内容是服务

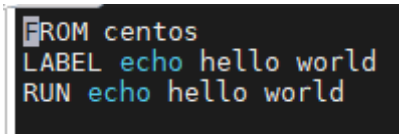
6.3.2.3 build 命令

#指定 Dockerfile 所在文件的路径（可以是绝对路径，或者相对 docker-compose.yml 文件的路径）,并创建一个镜像.

实验 1:使用 build 创建一个镜像和容器

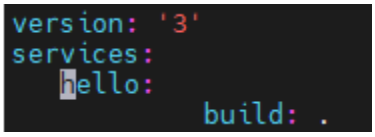
Step1 创建一个 dockerfile

vim dockerfile



Step2 创建 docker-compose.yml

vim docker-compose.yml



version: '3' #指定版本号为 3

services: #表示以下内容为定义服务

hello: #定义一个名为 hello 的服务

build: . #定义 dockerfile 在当前目录下

注:命令和参数之间需要加空格

然后使用命令拉起

docker-compose up #需在 docker-compose.yml 文件所在的目录下执行

```
[root@docker197 test]# docker-compose up
Building hello
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello world
 ---> Running in f3bbf51955e2
Removing intermediate container f3bbf51955e2
 ---> 05d50469c197
Step 3/3 : RUN echo hello world
 ---> Running in 9535952b2a32
hello world
Removing intermediate container 9535952b2a32
 ---> 9a699ffbc30e
Successfully built 9a699ffbc30e
Successfully tagged test_hello:latest
WARNING: Image for service hello was built beca
 `docker-compose up --build`.
Creating test_hello_1 ... done
Attaching to test_hello_1
test_hello_1 exited with code 0
```

会创建一个镜像:

```
[root@docker197 test]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
test_hello    latest    9a699ffbc30e   21 seconds ago   231MB
lump_mysql    latest    7529706800e0   16 hours ago     540MB
lump_nginx    latest    406fd8850051   16 hours ago     1.4GB
centos        latest    5d0da3dc9764   8 months ago     231MB
```

红框中中 test_hello 就是我们刚创建的镜像,其中 test 是 docker-compose.yml 所在的目录名,hello 是服务名.

同时也会创建一个容器:

```
[root@docker197 test]# docker ps -a
CONTAINER ID   IMAGE         COMMAND       CREATED          STATUS                      PORTS     NAMES
83b7e000fc3d   test_hello    "/bin/bash"   21 minutes ago   Exited (0) 20 minutes ago             test_hello_1
2396b3ed2398   centos        "bash"        5 days ago       Exited (0) 5 days ago                 centos1
[root@docker197 test]# docker start test_hello_1
test_hello_1
```

也可以指定一些详细的参数:

```
version: '3'
services:
   hello:
         build:
               context: /root/dockerlab/compose/test
               dockerfile: dockerfile
               args:
                     args1: hello
                     args2: world
```

Context: /root/dockerlab/compose/test #指定 dockerfile 文件所在的目录,该目录也是发送到

Docker 守护程序构建镜像的上下文。

dockerfile: dockerfile　　#指定 dockerfile 的文件名

args:　　#定义两个变量

修改下 dockerfile

```
FROM centos
LABEL echo hello world
ARG args1
ARG args2
RUN echo "$args1 $args2"
```

ARG arges1　　#定义一个名为 ares1 的变量(这个变量的值,将会由上面的 docker-compose.yml 文件中传入

```
[root@docker197 test]# docker-compose up
Building hello
Step 1/5 : FROM centos
 ---> 5d0da3dc9764
Step 2/5 : LABEL echo hello world
 ---> Running in 43b42dda5e80
Removing intermediate container 43b42dda5e80
 ---> cecff4633259
Step 3/5 : ARG args1
 ---> Running in e743e8569605
Removing intermediate container e743e8569605
 ---> 3d382b1c9f93
Step 4/5 : ARG args2
 ---> Running in 28b36ffa7963
Removing intermediate container 28b36ffa7963
 ---> c5e6ab93ff26
Step 5/5 : RUN echo "$args1 $args2"
 ---> Running in 6c7e5703a9c5
hello world
Removing intermediate container 6c7e5703a9c5
 ---> 6f3140d3a81d
Successfully built 6f3140d3a81d
Successfully tagged test_hello:latest
WARNING: Image for service hello was built because
 `docker-compose up --build`.
Creating test_hello_1 ... done
Attaching to test_hello_1
test_hello_1 exited with code 0
```

注意红框的内容,说明这两个变量已成功传递.

6.3.2.4 image 命令

#指定一个镜像文件

实验 1:指定一个名为 centos 的镜像文件,并创建一个容器

vim docker-compose.yml

```
version: '3'
services:
   hello:
        image: centos
```

Docker-compose up

```
[root@docker197 test]# docker-compose up
Creating test_hello_1 ... done
Attaching to test_hello_1
test_hello_1 exited with code 0
[root@docker197 test]# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS                      PORTS     NAMES
6a9fa5cc51f2   centos    "/bin/bash"   7 seconds ago    Exited (0) 6 seconds ago              test_hello_1
2396b3ed2398   centos    "bash"        5 days ago       Up 3 hours                            centos1
```

6.3.2.5 cap_add,cap_drop 命令

#添加/删除容器的权限

```
version: '3'
services:
   hello:
        image: centos
        cap_add:
             - NET_ADMIN
```

这里是增加了一个 NET_ADMIN 的权限

具体容器权限的含义请参考以下文档:

https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities

6.3.2.6 command 命令

#覆盖容器启动后默认执行的命令

```
version: '3'
services:
   hello:
        image: centos
        command: bash -c "touch /tmp/1  && ls /tmp/"
```

红框中的内容意思是:在容器启动的时候执行 bash –c “touch /tmp/1 && ls /tmp/这条命令,也就是先在 /tmp 下创建一个名为 1 的文件,再用 ls 看一下/tmp/中的内容

```
[root@docker197 test]# docker-compose up
Creating test_hello_1 ... done
Attaching to test_hello_1
hello_1  | 1
hello_1  | ks-script-4luisyla
hello_1  | ks-script-o23i7rc2
hello_1  | ks-script-x6ei4wuu
test_hello_1 exited with code 0
```

这个文件已经创建好了.

这里证实了 command 命令已经生效

那么，如果 dockerfile 里指定了 ENTRYPOINT 或 CMD（需要回顾一下 2.1.7.4）,最终容器会执行哪个呢？

让我们来再做一个实验：

```
FROM centos
LABEL echo hello dockerfile
CMD echo dockerfile
```

Dockerfile 中指定了一个 CMD 命令，echo dockerfile

```
version: '3'
services:
   hello:
        build:
             - context: .
             - dockerfile: dockerfile
        command: bash -c "echo hello docker-compose"
```

模版文件中指定了一个 command 命令， echo hello docker-compose

我们来观察一下，最终生成的容器，会执行哪一条命令

```
[root@docker197 test]# docker-compose up
Creating test_hello_1 ... done
Attaching to test_hello_1
hello_1  | hello docker-compose
```

这里我们看到，只会执行 command 的命令，而 dockerfile 中的 CMD 命令不会被执行。

6.3.2.7 container_name 命令

#指定容器名称。默认将会使用“项目名称_服务名称_序号”这样的格式。目前不支持在 Swarm 模式中使用。

```
version: '3'
services:
   hello:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        command: bash -c "echo hello docker-compose"
        container_name: hello_container
```

```
[root@docker197 test]# docker-compose ps
     Name                    Command               State    Ports
------------------------------------------------------------------
hello_container   /bin/sh -c mkdir /1 bash - ...   Exit 0
```

6.3.2.8entrypoint 命令

覆盖容器中默认的入口命令。注意，也会取消掉镜像中指定的入口命令和默认启动命令。

这个是我们学过的第 4 个容器入口命令了。

我们来回顾一下

1） dockerfile 中的 ENTRYPOINT

2） dockerfile 中的 CMD

3） docker-compose 中的 command

4） docker-compose 中的 entrypoint

dockerfile 中的两个入口命令的区别，我们在 2.1.7.4 小节中已经讲过了。

而在 6.3.2.6 小节中，我们又验证了 command 命令的优先级比前两个要高。

那么我们再来看下 command 和 entrypoint 的优先级哪下更高

```
version: '3'
services:
   hello:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        command: bash -c "echo hello command"
        entrypoint: bash -c "echo hello entrpoint"
        container_name: hello_container
```

在模版文件中这两个命令都指定一下，我们看看，最终容器会执行哪个？

```
[root@docker197 test]# docker-compose up
Recreating hello_container ... done
Attaching to hello_container
hello_container | hello entrpoint
hello_container exited with code 0
```

这里看到，最终执行的是 entrypoint 命令。

3.2.2.9environment 命令

设置环境变量

这里制定了两个环境变量，env1 和 env2,并在开启的时候显示环境变量。

```
version: '3'
services:
   hello:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        environment:
                - env1="I am env1"
                - env2="I am env2"

        entrypoint: bash -c "env"
        container_name: hello_container
```

```
[root@docker197 test]# docker-compose up
Creating hello_container ... done
Attaching to hello_container
hello_container | HOSTNAME=0ed15338be92
hello_container | env2="I am env2"
hello_container | env1="I am env1"
hello_container | PWD=/
hello_container | HOME=/root
hello_container | SHLVL=1
hello_container | PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
hello_container | _=/usr/bin/env
hello_container exited with code 0
```

3.2.2.10extends 命令

基于其他模板文件进行扩展。

比如我们已经有一了个 compose1.xml 模版文件，内容如下：

```
version: '2'
services:
   hello:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        environment:
                - env1="I am env1"
                - env2="I am env2"

        entrypoint: bash -c "env"
        container_name: hello_container
```

这里注意，v3 格式不支持，extends，所以我们改成 v2 格式。

然后我们在 docker-compose.xml 里可以调用这个文件里的内容

```
version: '2'
services:
  hello1:
    extends:
      file: compose1.yml
      service: hello
```

file: compose1.xml　　#调用 compose1 中的内容

service: hello　　#调用 hello 这个服务

```
[root@docker197 test]# docker-compose ps
     Name            Command     State    Ports
-------------------------------------------------
hello_container   bash -c env   Exit 0
```

是可以执行成功的。

3.2.2.11 devices 命令

指定设备映射关系，不支持 Swarm 模式。

```
version: '3'
services:
   hello:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        environment:
                - env1="I am env1"
                - env2="I am env2"

        entrypoint: bash -c "ls /dev/"
        container_name: hello_container
        devices:
             - "/dev/sdc:/dev/sdc"
```

这里是把宿主机的/dev/sdc 这个设备映射到容器中的/dev/sdc

```
WARNING: Image for service hello was bui
  `docker-compose up --build`.
Creating hello_container ... done
Attaching to hello_container
hello_container | core
hello_container | fd
hello_container | full
hello_container | mqueue
hello_container | null
hello_container | ptmx
hello_container | pts
hello_container | random
hello_container | sdc
hello_container | shm
hello_container | stderr
hello_container | stdin
hello_container | stdout
hello_container | tty
hello_container | urandom
hello_container | zero
hello_container exited with code 0
```

在容中内，会出现这个 sdc 的设备。

3.2.2.12 depends_on 命令

#指定多个服务之间的依赖关系。启动时，会先启动被依赖服务。

```
version: '3'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile

        entrypoint: bash -c "echo hello1"
        depends_on:
             - hello2
   hello2:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        entrypoint: bash -c "echo hello2"
~
```

以上模版文件中，定义了两个服务：hello1,hello2.

并定义了，hello1 依赖于 hello2.

我们来执行一下看看：

```
[root@docker197 test]# docker-compose up hello1
Starting test_hello2_1 ... done
Recreating test_hello1_1 ... done
Attaching to test_hello1_1
hello1_1  | I am hello1
test_hello1_1 exited with code 0
```

这里发现在启动 hello1 这个服务之前，会先启动 hello2 这个服务。

3.2.2.13 external_links 命令

#链接到其它容器

创建一个名为 centos1 的容器，并将此容器的网络指定为 test_default（用 docker-compose 创建的容器，默认在这个网络下）

docker run -it --name centos1 --hostname centos1 --network test_default centos bash

```
version: '3'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        external_links:
                  - centos1
        entrypoint: bash -c "sleep infinity"
~
```

这里链接到 centos1 这个容器

最后一行容器启动时执行的命令 sleep infinity 的意思是让当前的 shell 永久处理睡眠模式，目的是让容器不要退出。

```
[root@docker197 test]# docker-compose up -d
Starting test_hello1_1 ... done
```

这个 docker-compose up 后面的-d，是代表让容器在后台运行

```
[root@docker197 test]# docker-compose exec hello1 ping centos1
PING centos1 (172.19.0.3) 56(84) bytes of data.
64 bytes from centos1.test_default (172.19.0.3): icmp_seq=1 ttl=64 time=0.162 ms
64 bytes from centos1.test_default (172.19.0.3): icmp_seq=2 ttl=64 time=0.325 ms
64 bytes from centos1.test_default (172.19.0.3): icmp_seq=3 ttl=64 time=0.169 ms
```

这里我们可以 ping 通被链接的那个容器名，说明链接成功。

这里被链接的容器要求和本容器处于一个网络下。

3.2.2.14 links 命令

#链接到其他服务中的容器。

```
version: '3.1'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        links:
             - hello2
        entrypoint: bash -c "sleep infinity"
   hello2:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        entrypoint: bash -c "sleep infinity"
~
```

这里定义了两个服务，hello1 这个服务 link 到了 hello2

这样在 hello1 上就可以直接 ping 通 hello2 这个服务名了

```
[root@docker197 test]# docker-compose exec hello1 ping hello2
PING hello2 (172.19.0.4) 56(84) bytes of data.
64 bytes from test_hello2_1.test_default (172.19.0.4): icmp_seq=1 ttl=64 time=0.464 ms
64 bytes from test_hello2_1.test_default (172.19.0.4): icmp_seq=2 ttl=64 time=0.111 ms
^C
```

3.2.2.15extra_hosts 命令

指定额外的 host 名称映射信息。

```
version: '3'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        extra_hosts:
             - "googledns:8.8.8.8"
        entrypoint: bash -c "sleep infinity"
~
```

这里指定一个主机名为：googledns，IP 为 8.8.8.8

注意：需要将参数用“”引起来。

```
[root@docker197 test]# docker-compose exec hello1 ping googledns
PING googledns (8.8.8.8) 56(84) bytes of data.
64 bytes from googledns (8.8.8.8): icmp_seq=1 ttl=110 time=79.1 ms
64 bytes from googledns (8.8.8.8): icmp_seq=2 ttl=110 time=77.3 ms
64 bytes from googledns (8.8.8.8): icmp_seq=3 ttl=110 time=76.7 ms
64 bytes from googledns (8.8.8.8): icmp_seq=4 ttl=110 time=74.10 ms
```

这里发现是可能 ping 通这个 googledns 的主机名的。

```
[root@docker197 test]# docker-compose exec hello1 cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
8.8.8.8 googledns
172.19.0.2      a867a9862576
```

其实就是在容器的/etc/hosts 文件里增加了“8.8.8.8 googledns”这一行

3.2.2.16 healthcheck 命令

#指定检测应用健康状态的机制，包括检测方法（test）、间隔（interval）、超时（timeout）、重试次数（retries）、启动等待时间（start_period）等。

```
version: '3.4'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        extra_hosts:
             - "googledns:8.8.8.8"
        healthcheck:
             test: ["CMD","crul","-f","http://localhost:8080"]
             interval: 10s
             timeout:  5s
             retries:  3
             start_period: 5s
        entrypoint: bash -c "sleep infinity"
~
```

注意红框内，这里要指定几个选项：

test 检测方式（有 CMD 和 CMD-SHELL 两种）

这里的意思是 crul –f http://localhost:8080 这条命令。

Interval　检测间隔时间

timeout 超时时间

retries　　重试次数

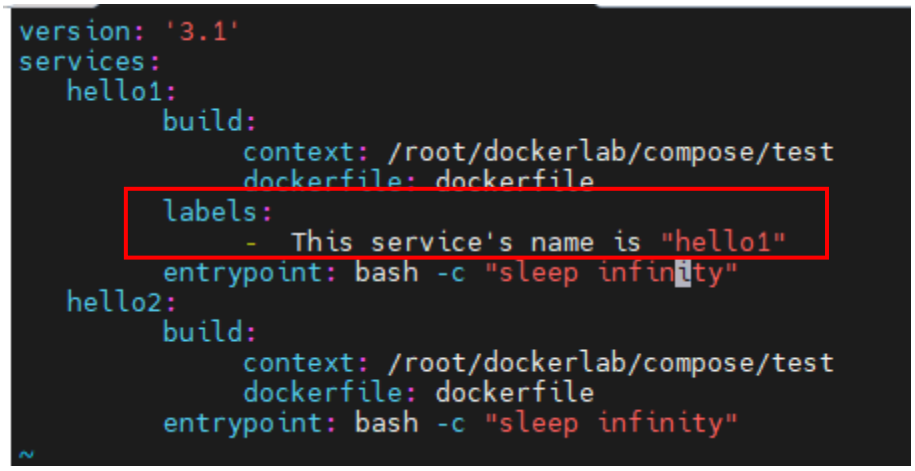
start period　启动后等待多长时间开始检测。

```
[root@docker197 test]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                            PORTS     NAMES
4c9c4af65e80   test_hello1    "bash -c 'sleep infi…"   17 seconds ago   Up 15 seconds (unhealthy)                   test_hello1_1
```

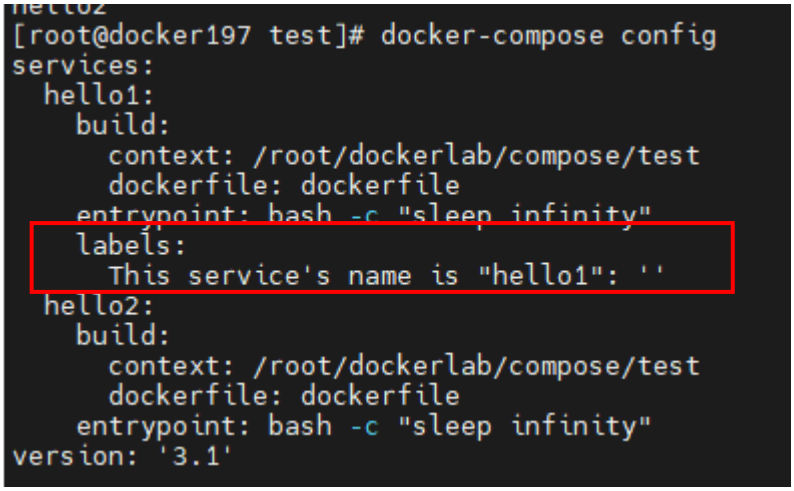
因为我们的容器没开 8080 端口，所以过一会以后，docker 的状态就会变成 unhealthy.

3.2.2.17 labels 命令

#为容器添加 Docker 元数据（metadata）信息。例如可以为容器添加辅助说明信息。



用 docker-compose config 这个命令，来看一下服务的详细信息



3.2.2.18 network_mode 命令

设置网络模式。

network_mode: "bridge"　#设置网络模式为 bridge

network_mode: "host"　　#设置网络模式为 host

network_mode: "none"　#设置网络模式为 none(禁用所有网络)

network_mode: "service:[service name]"　#共享其它服务的网络命名空间

network_mode: "container:[container name/id]"　　#共享其它容器的网络命名空间

```
version: '3.1'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        labels:
             - This service's name is "hello1"
        network_mode: "host"
        entrypoint: bash -c "sleep infinity"
   hello2:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        network_mode: "service: hello1"
        entrypoint: bash -c "sleep infinity"
```

hello1 服务的网络模式设为 host

hello2 服务的网络模式共享 hello1 的网络命令空间。

3.2.2.19 networks 命令

所加入的网络。需要在顶级的 networks 字段中定义具体的网络信息。

如果我们在模版文件中什么网络参数都不配置的话，在 up 的时候，会自动创建一个以 project_default 命名的网络。

```
[root@docker197 test1]# docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
d830b6f54881   bridge            bridge    local
b449da1d1af4   docker_gwbridge   bridge    local
dbddcaffe8fd   host              host      local
d3715506cdb6   lump_default      bridge    local
70bdcf9cb289   none              null      local
9d7523264c05   test1_default     bridge    local
df363c52596c   test_default      bridge    local
[root@docker197 test1]#
```

我这边用 docker-compose 创建了三个项目，就会产生三个网络。

```
[root@docker197 nginx]# ifconfig
br-9d7523264c05: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.21.0.1  netmask 255.255.0.0  broadcast 172.21.255.255
        inet6 fe80::42:49ff:fe7c:6f8  prefixlen 64  scopeid 0x20<link>
        ether 02:42:49:7c:06:f8  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15  bytes 1186 (1.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

br-d3715506cdb6: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:c4:ce:79:92  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

br-df363c52596c: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.19.0.1  netmask 255.255.0.0  broadcast 172.19.255.255
        inet6 fe80::42:78ff:fe16:ac25  prefixlen 64  scopeid 0x20<link>
        ether 02:42:78:16:ac:25  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.20.64.1  netmask 255.255.255.0  broadcast 172.20.64.255
        ether 02:42:a6:c1:4b:5d  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

且网段都不一样。

我们也可以在模版文件中指定容器的网络。

```
version: '3.1'
services:
   hello1:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        labels:
             -  This service's name is "hello1"
        networks: bridge
        entrypoint: bash -c "sleep infinity"
   hello2:
        build:
             context: /root/dockerlab/compose/test
             dockerfile: dockerfile
        networks: bridge
        entrypoint: bash -c "sleep infinity"
~
```

实验 1：指定一个默认的自定义网络

Step 1 创建一个网络（详见 5.3 小节）

docker network create mynet --subnet 192.168.100.0/24 --gateway 192.168.100.1

```
[root@docker197 test]# docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
d830b6f54881   bridge            bridge    local
b449da1d1af4   docker_gwbridge   bridge    local
dbddcaffe8fd   host              host      local
d3715506cdb6   lump_default      bridge    local
b7a43120d86a   mynet             bridge    local
70bdcf9cb289   none              null      local
9d7523264c05   test1_default     bridge    local
df363c52596c   test_default      bridge    local
```

这里会创建成功一个名为 mynet 的网络

Step 2 修改模版文件

```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         entrypoint: bash -c "sleep infinity"
networks:
  default:
     external:
          name: mynet
```

Step 3 确认网络配置

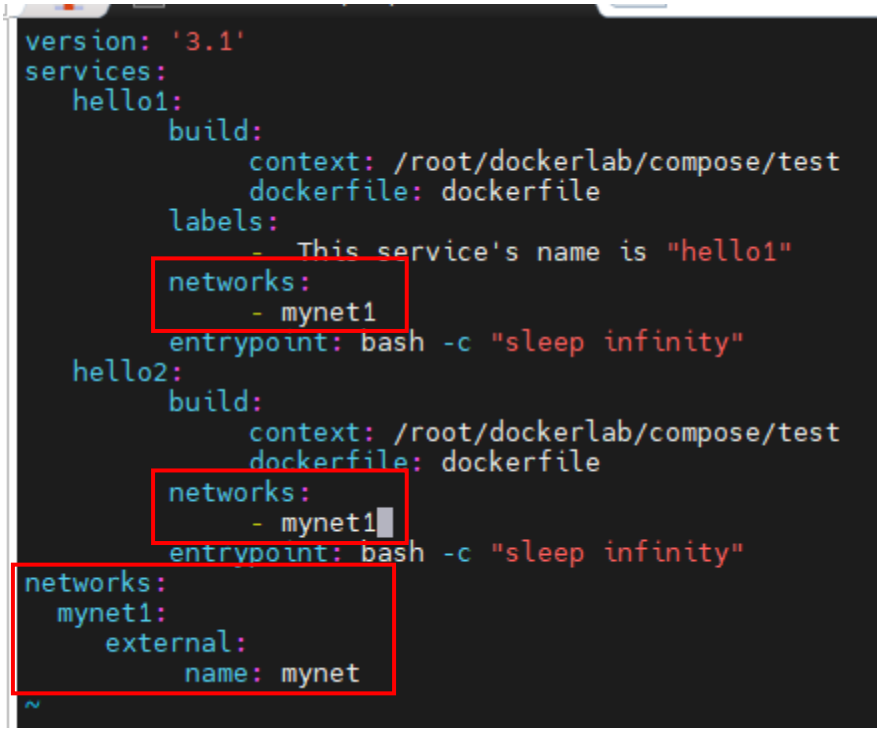
将服务 up 起来以后，观察容器的网络

docker network inspect mynet

```
    "ConfigOnly": false,
    "Containers": {
        "2ea15cfedd67a113986f28ae5c15e345f2a162af9018ad82e5d45cddf9cb9de5": {
            "Name": "test_hello2_1",
            "EndpointID": "250722d5386e85f3426da670388cd247418c57a60eb82e3722885a816d9b5bee",
            "MacAddress": "02:42:c0:a8:64:02",
            "IPv4Address": "192.168.100.2/24",
            "IPv6Address": ""
        },
        "3bcf9c2503c8287c3daef15560b7b0e1dcee1794f8c2dda7a3233f1b8a30c610": {
            "Name": "test_hello1_1",
            "EndpointID": "af2abfd73614381bdb9fa668f108adb9b70783734ebdfae901b29afaee9b6172",
            "MacAddress": "02:42:c0:a8:64:03",
            "IPv4Address": "192.168.100.3/24",
            "IPv6Address": ""
        }
```

可以看到 hello1 和 hello2 这两个容器都在 mynet 这个网络下了。

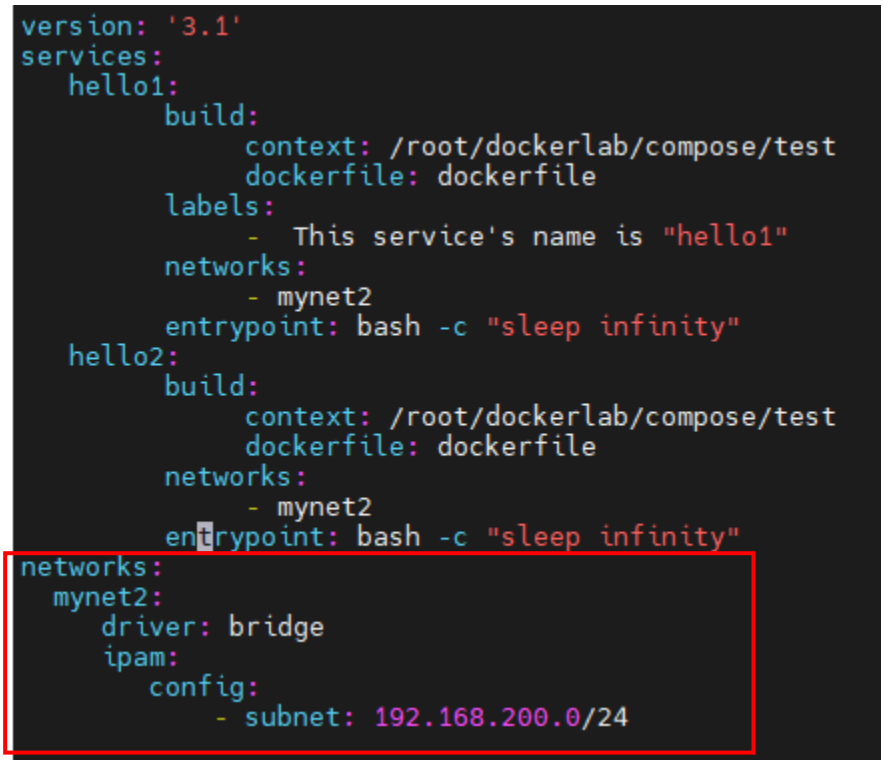
如果我们自定义的这个网络名不是 default,则需要在每个服务里指定一下。



```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         networks:
              - mynet1
         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         networks:
              - mynet1
         entrypoint: bash -c "sleep infinity"
networks:
  mynet1:
     external:
          name: mynet
~
```

注意红框中的内容，这个效果是一样的。

实验 2 利用模块文件来创建自定义网络



```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         networks:
              - mynet2
         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         networks:
              - mynet2
         entrypoint: bash -c "sleep infinity"
networks:
  mynet2:
     driver: bridge
     ipam:
        config:
            - subnet: 192.168.200.0/24
```

这里在模版文件中创建一个自定义网络，并指定其网段是 192.168.200.0/24

```
[root@docker197 test]# docker network ls
NETWORK ID     NAME               DRIVER    SCOPE
d830b6f54881   bridge             bridge    local
b449da1d1af4   docker_gwbridge    bridge    local
dbddcaffe8fd   host               host      local
d3715506cdb6   lump_default       bridge    local
b7a43120d86a   mynet              bridge    local
70bdcf9cb289   none               null      local
9d7523264c05   test1_default      bridge    local
df363c52596c   test_default       bridge    local
42477e6d07ea   test_mynet2        bridge    local
```

这里会产生一个名为 test_mynet2 的网络，系统会自动在我们定义的网络名前加上 project 名。

docker network inspect mynet


```
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "7baf6e3ba88fb9818db4426ad1f1b9ca0d6bb8279e809ee516bebad6e6978216": {
            "Name": "test_hello1_1",
            "EndpointID": "39046befee88d093f3d5b4085d5816b4229d92c56cabbc0fc767b553ecaba6b8",
            "MacAddress": "02:42:c0:a8:c8:03",
            "IPv4Address": "192.168.200.3/24",
            "IPv6Address": ""
        },
        "c7a47b0fa3b265c5e5d98cd9210c5984fa44ec267c7adc6dc62a135797dd5eea": {
            "Name": "test_hello2_1",
            "EndpointID": "87a6272765477bed8e59fe93e247a67f7690ab570a65831102c66b2cff21ef70",
            "MacAddress": "02:42:c0:a8:c8:02",
            "IPv4Address": "192.168.200.2/24",
            "IPv6Address": ""
```


两个容器都在 mynet2 这个网络下了。

实验 3 设置网络别名 aliases

```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         networks:
              - mynet2
         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         networks:
              mynet2:
                 aliases:
                       - net2
         entrypoint: bash -c "sleep infinity"
networks:
  mynet2:
     driver: bridge
     ipam:
        config:
            - subnet: 192.168.200.0/24
```

红框中的意思是，hello2 这个服务使用 mynet2 这个网络，并定义了一个别名。

这样我们在 hello1 这个服务中，可以直接使用这个别名来和 hello2 通讯。

```
[root@docker197 test]# docker-compose exec hello1 ping net2
PING net2 (192.168.200.2) 56(84) bytes of data.
64 bytes from test_hello2_1.test_mynet2 (192.168.200.2): icmp_seq=1 ttl=64 time=0.421 ms
64 bytes from test_hello2_1.test_mynet2 (192.168.200.2): icmp_seq=2 ttl=64 time=0.164 ms
64 bytes from test_hello2_1.test_mynet2 (192.168.200.2): icmp_seq=3 ttl=64 time=0.166 ms
64 bytes from test_hello2_1.test_mynet2 (192.168.200.2): icmp_seq=4 ttl=64 time=0.213 ms
```

这个功能有点和 3.2.2.14 小节那个 links 命令相似。

3.2.2.20 posts 命令

#将宿主机的映射到容器

使用宿主：容器（HOST：CONTAINER）格式，或者仅仅指定容器的端口（宿主将会随机选择端口）都可以。

ports:

- "3000"　　#宿主机随机一个端口映射到容器的 3000 端口
- "8000:8000"　　#宿主机的 8000 端口映射到容器的 8000 端口
- "49100:22"　　#宿主机的 49100 端口映射到容器的 22 端口
- "127.0.0.1:8001:8001"　　#宿主机 127.0.0.1：8001 端口映射到容器的 22 端口。

也可以像这样写

ports:

- target：80 #目标端口（容器端口）

  published：8080 #发布端口（宿主机端口）

  protocol：tcp #协议（TCP/UDP）

  mode：ingress #模式：进入

```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         networks:
              - mynet2
         ports:
              - 8080:80
         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         networks:
              mynet2:
                 aliases:
                         - net2
         entrypoint: bash -c "sleep infinity"
networks:
  mynet2:
     driver: bridge
     ipam:
        config:
            - subnet: 192.168.200.0/24
```

这里将宿主机的 8080 端口映射到了容器的 80 端口

```
[root@docker197 test]# docker-compose ps
    Name                  Command             State                    Ports
-------------------------------------------------------------------------------------------
test_hello1_1   bash -c sleep infinity   Up      0.0.0.0:8080->80/tcp,:::8080->80/tcp
test_hello2_1   bash -c sleep infinity   Up
```

可以看到已经映射成功

3.2.2.21 secrets 命令

配置应用的秘密数据。

可以指定来源秘密、挂载后名称、权限等。

Step1 创建一个文件名为 mypassword.txt 的文件

echo 123456 > mpassword.txt

```
[root@docker197 test]# echo 123456 > mypassword.txt
[root@docker197 test]# cat mypassword.txt
123456
```

Step2 修改 docker-compose.yml 文件

```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         networks:
              - mynet2
         ports:
              - "8080:80"
         secrets:
              - mypassword
         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         networks:
              mynet2:
                 aliases:
                         - net2
         entrypoint: bash -c "sleep infinity"
networks:
  mynet2:
     driver: bridge
     ipam:
        config:
            - subnet: 192.168.200.0/24

secrets:
  mypassword:
     file: ./mypassword.txt
```

第二个红框定义了一个名为 mypassword 的 secret，内容来致于./mypassword.txt 这个文件。

第一个红框是调用这个定义好的 secret

Step 3:启动服务，并查看效果

docker-compose exec hello1 cat /run/secrets/mypassword

```
[root@docker197 test]# docker-compose exec hello1 cat /run/secrets/mypassword
123456
```

在这个服务的容器中，会出现一个生成一个/run/secrets/mypassword 的文件，并且这个文件的内容，就是我们在第一步中创建的那个文件的内容。

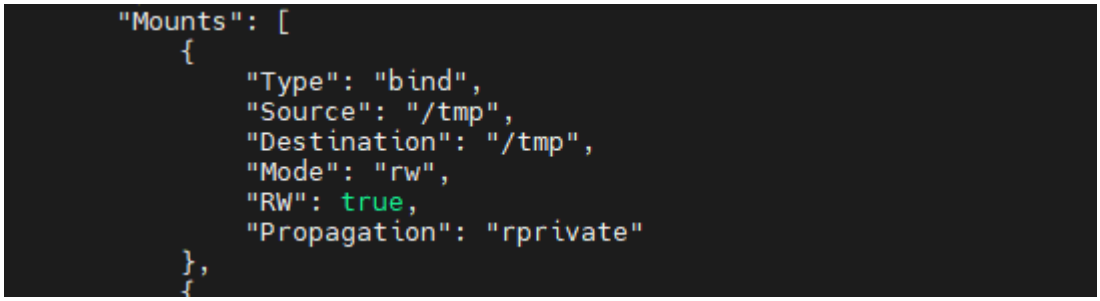
3.2.2.22 voloumes 命令

数据卷所挂载路径设置。可以设置宿主机路径（HOST：CONTAINER）或加上访问模式（HOST：CONTAINER：ro|rw）

```
version: '3.1'
services:
   hello1:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         labels:
              -  This service's name is "hello1"
         networks:
              - mynet2
         ports:
              - "8080:80"
         secrets:
              mypassword
         volumes:
              - /tmp:/tmp

         entrypoint: bash -c "sleep infinity"
   hello2:
         build:
              context: /root/dockerlab/compose/test
              dockerfile: dockerfile
         networks:
              mynet2:
                 aliases:
                       - net2
         entrypoint: bash -c "sleep infinity"
networks:
  mynet2:
     driver: bridge
     ipam:
        config:
            - subnet: 192.168.200.0/24
```

红框中的内容是将容器的/tmp 映射到宿主机的/tmp 上。

使用 docker inspect test_hello1_1 这个命令可以看到容的挂载情况



3.2.3.23 docker-compose 模版文件命令一览表

命令	功能
build	指定 dockerfile 所在文件夹的路径
cap_add,cap_drop	指定容器的内核能力（capacity）分配
command	覆盖容器启动后默认执行的命令
cgroup_parent	指定父 cgroup,意味着将继承该组的资源限制。目前不支持 swarm 模式
container_name	指定容器名称。目前不支持 swarm 模式
depends_on	指定多个服务之间的依赖关系
dns	自定义 DNS 服务器
dns_search	配置 DNS 搜索域
dockerfile	指定额外的编译镜像的 dockerfile 文件
entrypoint	覆盖容器中默认的入口命令
env_file	从文件中获取环境变量
environment	设置环境变量
expose	暴露端口，但不映射到宿主机，只被连接的服务访问
extends	基于其它模版文件进行扩展
external_links	链接到 docker-compose.yml 外部的容器
extra_hosts	指定额外的 host 名称映射信息
healthcheck	指定检测应用健康状态的机制
image	指定为镜像名称或镜像 ID
isolation	配置容器隔离的机制
labels	为容器添加 docker 元数据信息
links	链接到其它服务中的容器

logging	跟日志相关的配置
network_mode	设置网络模式
networks	所加入的网络
pid	跟主机系统共享进程命名空间
ports	暴露端口信息
secrets	配置应用的秘密数据
security_opt	指定容器模版标签（label）机制的默认属性（用户、角色、类型、级别等）
stop_grace_period	指定应用停止时，容器的优雅停止限期。过期后则通过 SIGKILL 强制退出。默认值为 10s
stop_signal	指定停止容器的信号
sysctls	配置容器内的内核参数。目前不支持 swarm 模式
ulimits	指定容器的 ulimits 限制值
userns_mode	指定用户命名空间模式。目前不支持 swarm 模式
volumes	数据卷所挂载路径设置
restart	指定重启策略
deploy	指定部署和运行时的容器相关配置。该命令只在 swarm 模式下生效，且只支持 docker stack deploy 命令部署。

6.3.3 compose 命令

docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]

-f,--file:指定使用的 Compose 模板文件，默认为 docker-compose.yml

-p,--project-name:指定项目名称，默认将使用所在目录名称作为项目名

--verbose:输出更多调试信息

-v,--version:打印版本信息

-H，--host:指定所要操作的 docker 服务的主机地址

--tls:启用 TLS，如果指定-tlsverify 则默认开启

--tlscacert ：信任的 TLS CA 的证书

--tlscert：客户端使用的 TLS 证书

--tlskey：TLS 的私钥文件路径

--tlsverify：使用 TLS 校验连接对方

--skip-hostname-check：不使用 TLS 证书校验对方的主机名

--project-directory PATH：指定工作目录，默认为 Compose 文件所在路径

6.3.3.1compose 命令一览表

命令	功能
build	构建或重新构建项目中的容器
bundle	创建一个可分发的配置包，包括整个服务栈的所有数据，它人可以利用该文件启动服务栈
config	查看 compose 文件的配置信息
down	停止服务栈，并删除相关资源,包括容器、挂载卷、网络、创建镜像等，默认情况下，只清除容器和网络资源
events	实时监控容器的事件信息
exec	在一个运行的容器中，执行一个指定的命令
help	获得一个命令的帮助
images	列出服务所创建的镜像
kill	通过发送 SIGKILL 信号来强制停止容器服务
logs	查看容器服务的输出
pause	暂停一个服务容器
port	打印某个容器端口所映射的公共端口
ps	列出项目中的所有容器
pull	拉取服务依赖的镜像
push	推送服务创建的镜像到镜像仓库
restart	重启项目中的服务
rm	删除所有（停止状态）的容器
run	在指定服务上执行一条命令
scale	设置指定服务运行容器的个数
start	启动已存在的服务容器
stop	停止已处于运行状态的服务容器，但不删除它
top	显示服务栈中正在运行的进程信息
unpause	恢复暂停的服务

up	尝试自动完成一系列的操作，包括构建镜像、创建（重建）服务，启动服务，并关联服务的相关容器等
version	打印版本信息

6.3.3.2 build 命令

构建（重新构建）项目中的服务容器(只会构建或重新构建容器的镜像文件)。

build [options] [--build-arg key=val...] [SERVICE...]

OPTIONS:

--force-rm:强制删除构建过程中的临时容器；

--no-cache：构建镜像过程中不使用,cache（这将加长构建过程）；

--pull：始终尝试通过 pull 来获取更新版本的镜像；

--m，-memory MEM：指定创建服务所使用的内存限制；

--build-arg key=val：指定服务创建时的参数。

docker-compose build

```
[root@docker197 test]# docker-compose images
Container   Repository   Tag   Image Id   Size
------------------------------------------------
[root@docker197 test]# docker-compose build
Building hello1
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello dockerfile
 ---> Using cache
 ---> 0669c0d14e91
Step 3/3 : ENTRYPOINT mkdir /1
 ---> Using cache
 ---> d01a29e31f73
Successfully built d01a29e31f73
Successfully tagged test_hello1:latest
Building hello2
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello dockerfile
 ---> Using cache
 ---> 0669c0d14e91
Step 3/3 : ENTRYPOINT mkdir /1
 ---> Using cache
 ---> d01a29e31f73
Successfully built d01a29e31f73
Successfully tagged test_hello2:latest
[root@docker197 test]#
```

创建了两个服务容器的镜像。

6.3.3.3 config 命令

打印服务栈的配置信息

docker-compose config [options]

options:

resolve-image-digests：为镜像添加对应的摘要信息；

-q，--quiet：只检验格式正确与否，不输出内容；

--services：打印出 Compose 中所有的服务信息；

--volumes：打印出 Compose 中所有的挂载卷信息；

```
[root@docker197 test]# docker-compose config
networks:
  mynet2:
    driver: bridge
    ipam:
      config:
      - subnet: 192.168.200.0/24
secrets:
  mypassword:
    file: /root/dockerlab/compose/test/mypassword.txt
services:
  hello1:
    build:
      context: /root/dockerlab/compose/test
      dockerfile: dockerfile
    entrypoint: bash -c "sleep infinity"
    labels:
      This service's name is "hello1": ''
    networks:
      mynet2: null
    ports:
    - 8080:80/tcp
  hello2:
    build:
      context: /root/dockerlab/compose/test
      dockerfile: dockerfile
    entrypoint: bash -c "sleep infinity"
    networks:
      mynet2:
        aliases:
        - net2
version: '3.1'
```

这里会将服务栈的信息打印出来。

6.3.3.4 up 命令

自动完成包括构建镜像，（重新）创建服务，启动服务，并关联服务相关容器的一系列操作。

默认情况，如果服务容器已经存在，docker-compose up 将会尝试停止容器，然后重新创建，以保证

新启动的服务匹配 docker-compose.yml 文件的最新内容。

docker-compose up [options] [--scale SERVICE=NUM...] [SERVICE...]

options:

-d：在后台运行服务容器；

--no-color：不使用颜色来区分不同的服务的控制台输出；

--no-deps：不启动服务所链接的容器；

--force-recreate：强制重新创建容器，不能与——no-recreate 同时使用；

--no-recreate：如果容器已经存在了，则不重新创建，不能与——force-recreate 同时使用；

--no-build：不自动构建缺失的服务镜像；

--abort-on-container-exit：当有容器停止时中止整个服务，与-d 选项冲突。

-t，--timeout TIMEOUT：停止容器时候的超时（默认为 10 秒），与-d 选项冲突；

--remove-orphans：删除服务中未定义的孤儿容器；

--exit-code-from SERVICE：退出时返回指定服务容器的退出符；

--scale SERVICE=NUM：扩展指定服务实例到指定数目。

```
[root@docker197 test]# docker-compose up
Building hello1
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello dockerfile
 ---> Running in 7885520d0886
Removing intermediate container 7885520d0886
 ---> 4448067f9050
Step 3/3 : ENTRYPOINT mkdir /1
 ---> Running in d35a09540650
Removing intermediate container d35a09540650
 ---> 0f963b2de8fc
Successfully built 0f963b2de8fc
Successfully tagged test_hello1:latest
WARNING: Image for service hello1 was built bec
r `docker-compose up --build`.
Building hello2
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello dockerfile
 ---> Using cache
 ---> 4448067f9050
Step 3/3 : ENTRYPOINT mkdir /1
 ---> Using cache
 ---> 0f963b2de8fc
Successfully built 0f963b2de8fc
Successfully tagged test_hello2:latest
WARNING: Image for service hello2 was built bec
r `docker-compose up --build`.
Creating test_hello1_1 ... done
Creating test_hello2_1 ... done
Attaching to test_hello1_1, test_hello2_1
```

执行 up 命令启动服务时，它干了三件事：

1、 为每个服务创建镜像（上图绿框所示）；

2、 创建容器（上图蓝框所示）；

3、 启动容器（上图红框所示）。

6.3.3.5 down 命令

停止服务栈，并删除相关资源，包括容器、挂载卷、网络、创建镜像等。

默认情况下只清除所创建的容器和网络资源。

docker-compose down [options]

options:

-rmi type：指定删除镜像的类型，包括 all（所有镜像），local（仅本地）；

-v，--volumes：删除挂载数据卷；

--remove-orphans：清除孤儿容器，即未在 Compose 服务中定义的容器；

--t，-timeout TIMEOUT：指定超时时间，默认为 10s。

```
[root@docker197 test]# docker-compose down
Stopping test_hello2_1 ... done
Stopping test_hello1_1 ... done
Removing test_hello2_1 ... done
Removing test_hello1_1 ... done
Removing network test_mynet2
```

我们看到，执行 down 命令的时候，做了两件事

1、 停止容器

2、 删除容器（但不会删除镜像）

如果使用:docker-compose down --rmi all，则会把镜像一块删除。

```
[root@docker197 test]# docker-compose down --rmi all
Stopping test_hello2_1 ... done
Stopping test_hello1_1 ... done
Removing test_hello2_1 ... done
Removing test_hello1_1 ... done
Removing network test_mynet2
Removing image test_hello1
Removing image test_hello2
```

6.3.3.6 events 命令

实时监控容器的事件信息

events [options] [SERVICE...]

options:

--json:以 Json 对象流格式输出事件信息。

6.3.3.7 exec 命令

在一个运行中的容器内执行给定命令。

docker-compose exec [options] [-e KEY=VAL...] SERVICE COMMAND [ARGS...]

options:

-d：在后台运行命令；

--privileged：以特权角色运行命令；

-u，-user USER：以给定用户身份运行命令；

-T：不分配 TTY 伪终端，默认情况下会打开；

--index=index：当服务有多个容器实例时指定容器索引，默认为第一个；

-e，-env KEY=VAL：设置环境变量。

docker-compose exec hello1 ps aux　　#在 hello1 这个服务容器中执行 ps aux 这个命令

```
[root@docker197 test]# docker-compose exec hello1 ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1  23048  1472 ?        Ss   07:07   0:00 /usr/bin/coreutils --coreutils-prog-shebang=sleep
root         27  0.0  0.3  44668  3448 pts/0    Rs+  07:14   0:00 ps aux
[root@docker197 test]#
```

6.3.3.8 images 命令

列出服务所创建的镜像。

docker-compose images [options] [SERVICE...]

options:

-q：仅显示镜像的 ID。

```
[root@docker197 test]# docker-compose images
  Container     Repository     Tag       Image Id      Size
-----------------------------------------------------------------
test_hello1_1   test_hello1   latest   c1fd9fd89241   221 MB
test_hello2_1   test_hello1   latest   c1fd9fd89241   221 MB
```

我们这个项目，有两个服务，两个镜像文件。

6.3.3.9 kill 命令

通过发送 SIGKILL 信号来强制停止服务容器。

docker-compose kill [options] [SERVICE...]

options:

-s:指定信号，默认为 SIGINT（相当于按 ctrl_c）

还有几种常用的信号：

SIGKILL：强制结束

SIGTERM：正常结束

```
[root@docker197 test]# docker-compose kill hello1
Killing test_hello1_1 ... done
[root@docker197 test]# docker-compose kill hello1
[root@docker197 test]# docker-compose ps
    Name                Command             State     Ports
-------------------------------------------------------------
test_hello1_1   bash -c sleep infinity   Exit 137
test_hello2_1   bash -c sleep infinity   Up
```

6.3.3.10 logs 命令

查看服务容器的输出。

docker-compose logs [options] [SERVICE...]

options:

-no-color：关闭彩色输出；

-f，-follow：持续跟踪输出日志消息；

-t，-timestamps：显示时间戳信息；

-tail="all"：仅显示指定行数的最新日志消息。

6.3.3.11 pause 命令和 unpause 命令

暂停一个服务容器和恢复暂停。

pause [SERVICE...]　　/ unpause [SERVICE]

```
[root@docker197 test]# docker-compose pause hello1
Pausing test_hello1_1 ... done
[root@docker197 test]# docker-compose ps
    Name                  Command             State                    Ports
-------------------------------------------------------------------------------------------
test_hello1_1   bash -c sleep infinity   Paused   0.0.0.0:8080->80/tcp,:::8080->80/tcp
test_hello2_1   bash -c sleep infinity   Up
[root@docker197 test]# docker-compose unpause hello1
Unpausing test_hello1_1 ... done
[root@docker197 test]# docker-compose ps
    Name                  Command             State                    Ports
-------------------------------------------------------------------------------------------
test_hello1_1   bash -c sleep infinity   Up       0.0.0.0:8080->80/tcp,:::8080->80/tcp
test_hello2_1   bash -c sleep infinity   Up
```

以上是先彤 pause 命令暂停 hello1 这个服务，然后再用 unpause 这个命令来恢复

6.3.3.12 port 命令

打印某个容器端口所映射的公共端口。

port [options] SERVICE PRIVATE_PORT

options:

--protocol=proto：指定端口协议，tcp（默认值）或者 udp；

--index=index：如果同一服务存在多个容器，指定命令对象容器的序号（默认为 1）。

```
[root@docker197 test]# docker-compose port hello1 80
0.0.0.0:8080
```

这里我们可以看到，服务容器的 80 端口，是映射到宿主机 8080 端口。

6.3.3.13 ps 命令

列出项目中目前的所有容器。

docker-compose ps [options] [SERVICE...]

options:

-q：只打印容器的 ID 信息。

6.3.3.14 pull 命令

为 compose 文件中定义的服务拉取镜像，但不启动容器。

docker-compose pull [options] [SERVICE...]

options:

--ignore-push-failures：忽略推送镜像过程中的错误;

--parallel:拉取相似的多个镜像；

--quiet:安静模式，拉取的时候不打印进度信息

```
version: '3'
services:
   hello1:
        image: centos:latest
        entrypoint: bash -c "sleep infinity"
~
```

红框中的内容是指我这个 hello1 服务，需要从 centos:latest 这个镜像来创建。

如果我主机本地没有这个镜像的话，使用 pull 命令，可以从镜像仓中自动拉取这个镜像。

```
[root@docker197 test1]# docker image rm -f centos
Untagged: centos:latest
Untagged: centos@sha256:a27fd8080b517143cbbbab9dfb7c8571c40d67d534bbdee55bd6c473f432b177
[root@docker197 test1]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
test_hello1   latest    c1fd9fd89241   29 hours ago    231MB
test_hello2   latest    c1fd9fd89241   29 hours ago    231MB
mysql         1.0       9667ee377686   2 months ago    540MB
```

把我本地的 centos 这个镜像先删掉。

```
[root@docker197 test1]# docker-compose pull hello1
Pulling hello1 (centos:latest)...
latest: Pulling from library/centos
Digest: sha256:a27fd8080b517143cbbbab9dfb7c8571c40d67d534bbdee55bd6c473f432b177
Status: Downloaded newer image for centos:latest
[root@docker197 test1]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
test_hello1   latest    c1fd9fd89241   29 hours ago    231MB
test_hello2   latest    c1fd9fd89241   29 hours ago    231MB
mysql         1.0       9667ee377686   2 months ago    540MB
centos        latest    5d0da3dc9764   10 months ago   231MB
```

执行 docker-compose pull hello1 后，发现自动从镜像仓中拉取了 centos 这个镜像了。

6.3.3.15 push 命令

把服务创建的镜像推送到镜像仓。

push [options] [SERVICE...]

options:

--ignore-push-failures:忽略推送镜像过程中的错误;

docker-compose push hello1　　#将 hello1 这个服务创建的镜像推送到镜像仓。

6.3.3.16 restart 命令

重启服务;

restart [options] [SERVICE...]

options:

-t, --timeout:指定重启前服务停止超时的时间，单位为秒。

```
[root@docker197 test]# docker-compose restart hello1
Restarting test_hello1_1 ... done
```

6.3.3.17 rm 命令

删除容器服务。

rm [options] [SERVICE...]

options:

-f, --force:强制删除，包括处于运行状态的容器;

-s, --stop：在删除之前停止容器;

-v:同时删除数据卷;

6.3.3.18 run 命令

在一个服务上执行一个 one-off 命令。

docker-compose run [options] [-v VOLUME...] [-p PORT...] [-e KEY=VAL...] [-l KEY=VALUE...] SERVICE [COMMAND] [ARGS...]

options:

-d:在后台运行

--name:指定容器的名称

--entrypoint:覆盖镜像中指定的入口命令；

-e KEY=VAL:设置一个环境变量，可多次使用；

-l,--label:增加或覆盖一个标签；

-u,--user:指定用户名或 UID 来运行容器；

--no-deps:不启动相关联的服务；

--rm:运行以后删除容器，后台运行模式下将被忽略；

-p,--publish:映射容器到端口到宿主机；

--service-ports:配置服务端口并映射到宿主机；

-v,--volume:挂载一个数据卷；

-T：不分配伪 TTL；

-w,--workdir：指定容器内的工作目录。

docker-compose run -d hello1 ping baidu.com

在 hello1 这个服务上执行”ping baidu.com”这个命令。

```
[root@docker197 test]# docker ps --no-trunc
CONTAINER ID                                                       IMAGE          COMMAND                                  CREATED
     STATUS          PORTS     NAMES
8cb09aae41089ea660a6aca33629da9af3614d1af6b729bb1feae91837bed966   test_hello1    "bash -c 'sleep infinity; ping baidu.com"   34 seconds a
go   Up 33 seconds             test_hello1_run_1
```

它会自动创建一个名为 test_hello1_run_1 的容器，并执行 ping baidu.com 命令。

6.3.3.19 start 命令

启动已在存在的服务容器。

docker-compose start [SERVICE...]

6.3.3.20 stop 命令

停止处于运行状态的服务容器，但是不删除它；

docker-compose　stop [options] [SERVICE...]

options:

-t,--timeout:指定一个容器停止超时，默认为 10S。

6.3.3.21 top 命令

显示服务栈中的进程信息。

docker-compose top [SERVICE...]

6.3.3.22 version 命令

打印 docker-compose 版本信息

docker-compose version [--short]

--short:只显示 compose 版本号

6.3.4 compose 环境变量

环境变量可以用来配置 compose 的行为，详见下表：

变量	功能
COMPOSE_PROJECT_NAME	设置项目名称，默认是当前的工作目录(docker-compose.yml 文件所在的目录)的名字。
COMPOSE_FILE	设置要使用的 docker-compose.yml 的路径。如果不指定，默认会先查找当前工作目录下是否存在 docker-compose.yml 文件，如果找不到，则会继续查找上层目录。
COMPOSE_API_VERSION	某些情况下，compose 发出的 docker 请求，其版本可能在服务端并不支持，可以通过指定 API 版本来临时解决这个问题。
DOCKER_HOST	设置 dockere 服务端的监听地址。默认使用 unix:///var/run/docker.sock
DOCKER_TLS_VERIFY	如果该环境变量不为空，则与 docker 服务端的所有交互都通过 TLS 协议进行加密。
DOCKER_CERT_PATH	配置 TLS 通信所需要的验证文件（包括 ca.pem、cert.pem 和 key.pem）的路径，默认是~/.docker
COMPOSE_HTTP_TIMEOUT	compose 向 docker 服务端发送请求时的超时，默认值为 60s
COMPOSE_TLS_VERSION	指定与 docker 服务进行交互的 TLS 版本，支持版本为 TLSV1(默认值)、TLSV1_1、TLSV1_2
COMPOSE_PATH_SEPARATOR	指定 COMPOSE_FILE 环境变量中的路径间隔符
COMPOSE_IGNORE_ORPHANS	是否忽略孤儿容器
COMPOSE_PARALLEL_LIMIT	设置 compose 可以执行进程的并发数
COMPOSE_INTERACTIVE_NO_CLI	尝试不使用 docker 命令来执行 run 和 exec 指令

使用这些环境参数，需要在工作目录下创建一个名为.env 的文件（注意：必须是这个文件名）

```
[root@docker197 test]# cat .env
COMPOSE_PROJECT_NAME=myfirstproject
```

这里将 COMPOSE_PROJECT_NAME 这个环境变量的值，设为 myfirstproject

```
[root@docker197 test]# docker-compose up -d
Creating network "myfirstproject_mynet2" with driver "bridge"
Building hello1
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello dockerfile
 ---> Using cache
 ---> 1ab633f73002
Step 3/3 : ENTRYPOINT mkdir /1
 ---> Using cache
 ---> 62d399b16708

Successfully built 62d399b16708
Successfully tagged myfirstproject_hello1:latest
WARNING: Image for service hello1 was built because it did not already exist. To r
r `docker-compose up --build`.
Building hello2
Step 1/3 : FROM centos
 ---> 5d0da3dc9764
Step 2/3 : LABEL echo hello dockerfile
 ---> Using cache
 ---> 1ab633f73002
Step 3/3 : ENTRYPOINT mkdir /1
 ---> Using cache
 ---> 62d399b16708

Successfully built 62d399b16708
Successfully tagged myfirstproject_hello2:latest
WARNING: Image for service hello2 was built because it did not already exist. To r
r `docker-compose up --build`.
Creating myfirstproject_hello1_1 ... done
Creating myfirstproject_hello2_1 ... done
```

可以看到，项目名称已经变成 myfirstproject 了。

6.3.5 docker-compose 综合实验

6.3.5.1 实验目标：

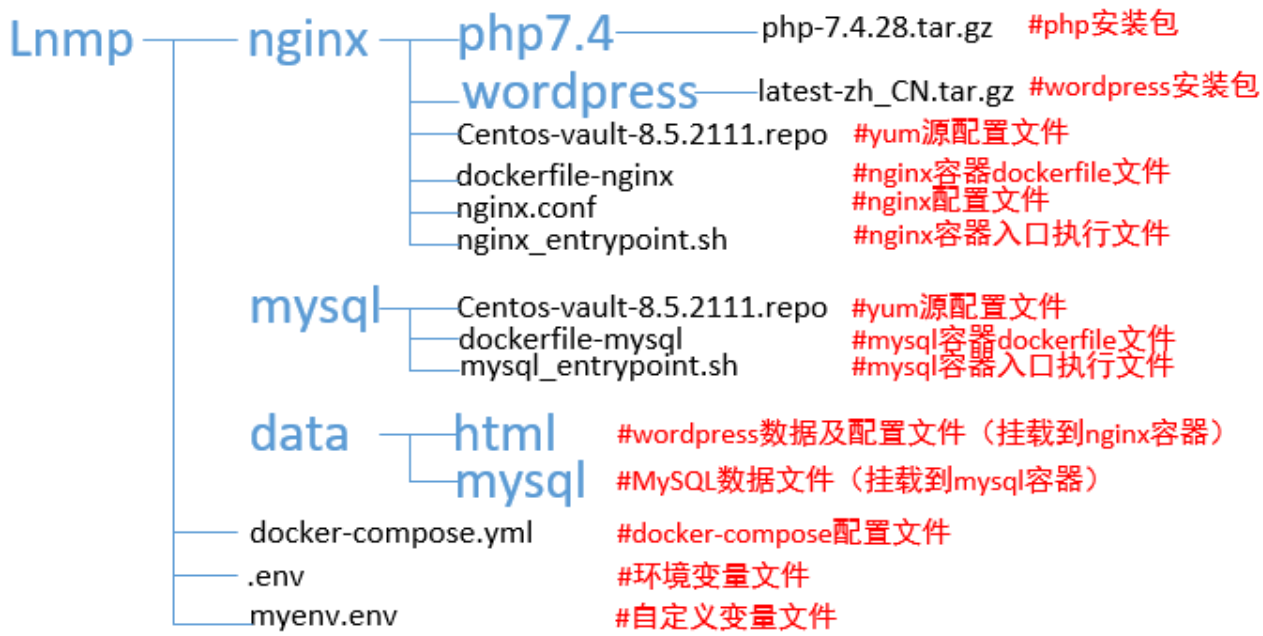
搭建一个基本于 nginx 和 MySQL 的 wordpress 博客系统

6.3.5.2 基本架构：

在同一宿主机上创建两个容器，容器 A 安装 nginx 和 wordpress 应用，容器 B 安装 mysql 应用。容器 A 依赖于容器 B。

wordpress 和 mysql 的应用数据保存在宿主机中，容器采用挂载的方式连接数据目录。

实验文档目录结构：



6.3.5.3 文件来源（内容）和用途

1）php-7.4.28.tar.gz 文件

用途：php 安装包（安装在 nginx 容器）

来源：wget https://www.php.net/distributions/php-7.4.28.tar.gz

2） latest-zh_CN.tar.gz 文件

用途：wordpress 博客系统安装包（安装在 nginx 容器）

来源：wget https://cn.wordpress.org/latest-zh_CN.tar.gz

3） Centos-vault-8.5.2111.repo 文件

用途：centos ali yum 源配置文件（安装在 nginx 和 mysql 容器）

来源：wgethttps://mirrors.aliyun.com/repo/Centos-vault-8.5.2111.repo

4） dockerfile-nginx 文件

用途：nginx 容器 dockerfile 文件

内容：

```
FROM centos:latest#指定基础镜像文件为 centos:latest
LABEL auto build nginx#自定义一个容器标签
EXPOSE 80#定义开放的端口号
COPY ./Centos-vault-8.5.2111.repo /tmp/Centos-vault-8.5.2111.repo#将 yum 源配置文件拷贝到 nginx 镜像中
COPY ./nginx_entrypoint.sh /root/nginx_entrypoint.sh#将入口执行脚本拷贝到 nginx 镜像中
COPY ./nginx.conf /tmp/nginx.conf#将 nginx 配置文件拷贝到 nginx 镜像中
COPY ./php7.4/php-7.4.28.tar.gz /tmp/php-7.4.28.tar.gz#将 php 安装包拷贝到 nginx 镜像中
COPY ./wordpress/latest-zh_CN.tar.gz /tmp/latest-zh_CN.tar.gz #将 wordpress 安装包拷贝到 nginx 镜像中
RUN chmod 755 /root/nginx_entrypoint.sh \#将入口执行脚本文件定义为可执行
&& ln -s /root/nginx_entrypoint.sh /usr/bin/nginx_entrypoint.sh \#建一个入口执行脚本文件的软链接到/usr/bin 下
&& rm -rf /etc/yum.repos.d/* \#删除基础镜像中的 yum 源配置文件
&& mv /tmp/Centos-vault-8.5.2111.repo /etc/yum.repos.d/ \ #将 ali yum 源配置文件移至/etc/um.repos.d 目录下
&& dnf makecache \#生成 yum 缓存
&& dnf install 'dnf-command(config-manager)' -y\#安装 dnf-command 工具
&& dnf config-manager --set-enabled PowerTools -y\#开启 dnf PowerTools
&& dnf makecache#生成 yum 缓存
RUN dnf install gcc libxml2-devel sqlite-devel libcurl-devel oniguruma-devel make -y \#安装编译依赖包（安装 php 时需要）
&& dnf install nginx -y \#安装 nginx
&& dnf install mysql -y \ #安装 mysql 客户端
&& rm -rf /var/cache/yum/*#清除 yum 缓存
RUN cd /tmp \#进入/tmp(php 和 wordpress 安装包位置)
&& tar -xzf php-7.4.28.tar.gz \#解压缩 php 安装包
&& cd /tmp/php-7.4.28 \#进入解压后的 php 安装包目录
&& ./configure --prefix=/usr/local/php --enable-fpm --with-mysqli --with-curl --with-pdo_mysql --with-pdo_sqlite --enable-mysqlnd
--enable-mbstring \#编译 php 安装文件
&& make install \#安装 php
&& ln -s /usr/local/php/sbin/php-fpm /usr/sbin/php-fpm \#将 php 的执行文件软链接到/usr/sbin 下
&& cp /tmp/php-7.4.28/php.ini-development /usr/local/php/lib/php.ini \#将 php 配置文件 php.ini 移动（并更名）到 php 安装目录下
&& mv /usr/local/php/etc/php-fpm.conf.default /usr/local/php/etc/php-fpm.conf \#将 php 配置文件 php-fpm.conf 移动（并更名）到 php 安
装目录下
&& mv /usr/local/php/etc/php-fpm.d/www.conf.default /usr/local/php/etc/php-fpm.d/www.conf#将 php 配置文件 www.conf 移动（并更
名）到 php 安装目录下
```

```
RUN rm -f /etc/nginx/nginx.conf \ #删除默认的 nginx 配置文件
&& mv /tmp/nginx.conf /etc/nginx/ \ #将我们定义好的 nginx 配置文件移动到/etc/nginx 目录下
&& rm -rf /usr/share/nginx/html/* \#将 nginx 默认的 html 文件删除
&& cd /tmp \ #进入/tmp 目录（wordpress 安装包在此目录下）
&& tar -xzf latest-zh_CN.tar.gz \#解压缩 wordpress 安装包
&& rm -r latest-zh_CN.tar.gz #删除 wordpress 压缩包

CMD nginx_entrypoint.sh #执行入口执行脚本。
```

5) nginx.conf 文件

用途：自定义 nginx 配置

内容：

```
server {
        listen          80 default_server;
        listen          [::]:80 default_server;
        server_name     _;
        root            /usr/share/nginx/html;
        # Load configuration files for the default server block.
        include /etc/nginx/default.d/*.conf;
#以下部分是定义根目录，和 index 文件
location / {
        root html;
        index    index.php index.html index.htm;
        }
#以下部分红色字体是用于 nginx 和 php 模块的连接
        location ~ \.php$ {
        root           html;
        fastcgi_pass   127.0.0.1:9000;
        fastcgi_index  index.php;
        fastcgi_param  SCRIPT_FILENAME  /usr/share/nginx/html$fastcgi_script_name;
        include        fastcgi_params;
        }
        location = /favicon.ico {
        log_not_found off;
        access_log off;
        }
        error_page 404 /404.html;
                location = /40x.html {
        }
        error_page 500 502 503 504 /50x.html;
                location = /50x.html {
        }
    }
```

6）nginx_entrypoint.sh 文件

文件用途：容器启动时自动执行的脚本文件（在 dockerfile 中 CMD 命令中执行）

文件内容：

```
#!/bin/bash
mysqladmin ping -u root -p$DATABASE_PASSWORD_ROOT -h $DATABASE_HOST#探测 mysql 服务是否正常启动
while [ "$?" != 0 ]#执行循环判断，如果 mysql 服务未正常启动，则发出提示
  do
      echo "The mysql server is not up!"
      sleep 1
      mysqladmin ping -u root -p$DATABASE_PASSWORD_ROOT -h $DATABASE_HOST
  done
mysql -u root -p$DATABASE_PASSWORD_ROOT -h $DATABASE_HOST -e "use $DATABASE_NAME"
if [ "$?" != 0 ]; then        #如果 DATABASE 不存在（第一次启动时），则执行创建数据库和用户
      mysql -u root -p$DATABASE_PASSWORD_ROOT -h $DATABASE_HOST -e "create user $DATABASE_USER identified by '$DATABASE_PASSWORD_USER'"#创建一个 mysql 用户名（给 wordpress 使用）
      mysql -u root -p$DATABASE_PASSWORD_ROOT -h $DATABASE_HOST -e "create database if not exists $DATABASE_NAME"#创建一个数据库（给 wordpress 使用）
      mysql -u root -p$DATABASE_PASSWORD_ROOT -h $DATABASE_HOST -e "grant all on $DATABASE_NAME.* to $DATABASE_USER"#给刚创建的 mysql 用户名赋予对刚创建的数据库赋予操作权限
fi
IFNULL=`ls /usr/share/nginx/html/ | wc -l`
if [ "$?" = 0 ]; then        #如果/usr/share/nginx/html 里没有文件（第一次启动时），则执行以下操作。
      mv /tmp/wordpress/* /usr/share/nginx/html/#将 wordpress 文件移至 nginx 目录下
      cp /usr/share/nginx/html/wp-config-sample.php /usr/share/nginx/html/wp-config.php#将 wordpress 配置文件更名
      chown -R nginx:nginx /usr/share/nginx#将 nginx 目录的拥有者改成用户 nginx
      sed -i "s/database_name_here/$DATABASE_NAME/g" /usr/share/nginx/html/wp-config.php#将自定义变量文件中定义的数据库名称添加到 wordpress 配置文件中
      sed -i "s/username_here/$DATABASE_USER/g" /usr/share/nginx/html/wp-config.php#将将自定义变量文件中定义的数据库用户名添加到 wordpress 配置文件中
      sed -i "s/password_here/$DATABASE_PASSWORD_USER/g" /usr/share/nginx/html/wp-config.php#将将自定义变量文件中定义的数据库用户名密码添加到 wordpress 配置文件中
      sed -i "s/localhost/$DATABASE_HOST/g" /usr/share/nginx/html/wp-config.php#将将自定义变量文件中定义的数据库服务器地址添加到 wordpress 配置文件中
fi
php-fpm#启动 php 服务
nginx#启动 nginx 服务
```

7）dockerfile-mysql 文件

用途：mysql 容器 dockerfile 文件

内容：

```
FROM centos:latest #指定一个基础镜像
LABEL auto build mysql#自定义一个标签
EXPOSE 3306#定义开放的端口
```

```
COPY ./Centos-vault-8.5.2111.repo /root/Centos-vault-8.5.2111.repo#将 yum 源配置文件拷贝到镜像中
COPY ./mysql_entrypoint.sh /mysql_entrypoint.sh#将入口执行脚本拷贝到镜像文件中
RUN chmod a+x /mysql_entrypoint.sh \#将入口执行脚本添加可执行属性
&& ln -s /mysql_entrypoint.sh /usr/bin/mysql_entrypoint.sh \#为入口执行脚本创建一个链接到/usr/bin 目录
&&rm -rf /etc/yum.repos.d/* \#删除默认 yum 源配置文件
&& mv /root/Centos-vault-8.5.2111.repo /etc/yum.repos.d/ \#将 yum 源文件移至/etc/yum.repos.d 下
&& yum makecache#创建 yum 缓存
RUN yum install mysql mysql-server -y \#安装 mysql 客户端和服务端
&& rm -rf /var/cache/yum/*#清除 yum 缓存
CMD mysql_entrypoint.sh #执行入口执行脚本，并让当前 shell 处理睡眠状态，以防止容器执行完脚本后自动停止。
```

8）mysql_entrypoint.sh 文件

用途：mysql 容器入口执行脚本

内容：

```
#!/bin/bash
chown -R mysql:mysql /var/lib/mysql/#将 mysql 目录的拥有者改为用户 mysql
chown mysql:mysql /usr/sbin/mysqld #将 mysql 服务启动程序的拥有者改为用户 mysql
echo user=mysql>> /etc/my.cnf.d/mysql-server.cnf#将 mysql 用户添加到 mysql 配置文件中，表示将使用 mysql 用户来执行 mysql
IFNULL=`ls /var/lib/mysql/ | wc -l` #判断 mysql 数据目录中是否有文件
if [ "$IFNULL" = 0 ]; then   #如果 mysql 目录中没有文件（第一次启动），则执行以下初始化的工作。
    mysqld –initialize#初始化 mysql
    mysqld –D#在后台启动 mysql
    ROOT_PASSWORD_TEMP=`tail /var/log/mysql/mysqld.log | grep password |awk '{print $NF}'`#从 mysqld.log 这个文件中获取 mysq 的初始密码
    mysql -u root -p$ROOT_PASSWORD_TEMP -e "alter user user( ) identified by '$DATABASE_PASSWORD_ROOT';" --connect-expired-password#修改 MySQL root 用户密码
    mysql -u root -p$DATABASE_PASSWORD_ROOT -e "update mysql.user set host='%' where user='root'"#配置任何主机都有权限从网络访问 mysql
    mysql -u root -p$DATABASE_PASSWORD_ROOT -e "flush privileges" #刷新数据库
    PID=`ps aux | grep mysqld | grep -v grep | awk '{print $2}'`    #将 mysql 的进程 ID 值赋予 PID 这个变量
     kill -9 $PID    #终止 mysql 进程
fi
mysqld #在前台启动 mysql 服务
```

9）docker-compose.yml 文件

用途：docker-compose 配置文件

内容：

```
version: '3.4' #版本号
services:
  nginx:#第一个服务，名为 nginx
    build:#构建容器
                context: ./nginx#指定 dockerfile 所在目录
                dockerfile: dockerfile-nginx#指定 dockerfile 文件名
```

```
    labels:#定义一个标签
            - nginx
    container_name: nginx#定义容器名称
    env_file:#定义环境文件
            - ./myenv.env
    depends_on:#依赖的服务
            - mysql
    links:#link 到其它容器
            - mysql
    networks:#定义使用的网络
            - lnmpnet
    ports:#将宿主机的 8080 端口映射到容器的 80 端口
            - 8080:80
    volumes:#容器挂载宿主机的目录
            - /root/dockerlab/compose/lnmp/data/html:/usr/share/nginx/html
    healthcheck:    #执行容器的健康检测
            test: ["CMD","curl","-f","http://mysql:3306"]
            interval: 5s
            timeout: 2s
            retries: 3
  mysql:#第二个服务，名为 mysql
    build:#构建容器
            context: ./mysql #指定 dockerfile 所在目录
            dockerfile: dockerfile-mysql#指定 dockerfile 文件名
    labels:#定义一个标签
            - mysql
    container_name: mysql#定义容器名称
    env_file:#定义环境文件
            - ./myenv.env
    networks:#定义使用的网络
            - lnmpnet
    ports:
            - 3306:3306#将宿主机的 3306 端口映射到容器的 3306 端口
    volumes:#容器挂载宿主机的目录
            - /root/dockerlab/compose/lnmp/data/mysql:/var/lib/mysql

networks:#创建一个网络
    lnmpnet:#定义新建网络的名称
            driver: bridge #定义新网络的类型为 bridge
            ipam: #配置新建网络的 IP 信息
                config:
                    - subnet: 192.168.200.0/24#配置将新建网络的网段
```

10）.env 文件

用途：定义项目环境变量

内容：

```
COMPOSE_PROJECT_NAME=lnmp#将项目名称定义为 lnmp
```

11）myenv.env 文件

用途：自定义变量

内容：

```
DATABASE_NAME=wordpress #定义 wordpress 使用的数据库名称
DATABASE_USER=wordpress #定义 wordpress 使用的数据库用户名
DATABASE_PASSWORD_ROOT=123456#定义 mysql 的 root 密码
DATABASE_PASSWORD_USER=123456#定义给 wordpress 使用的 mysql 用户名密码
DATABASE_HOST=mysql #定义给 wordpress 使用的数据库服务器地址
```

6.3.5.4 启动服务

创建好以上的文件以后，就可以启动服务了。

```
docker-compose up –d#在后台启动服务
```

```
[root@docker197 lump]# docker-compose ps
Name               Command               State                    Ports
-------------------------------------------------------------------------------------------
mysql   /bin/sh -c mysql_entrypoin ...   Up      0.0.0.0:3306->3306/tcp,:::3306->3306/tcp
nginx   /bin/sh -c nginx_entrypoin ...   Up      0.0.0.0:8080->80/tcp,:::8080->80/tcp
```

使用 docker-compose ps 命令可以看到两个容器都已经启动。

在浏览器中打开 wordpress 配置网页，开始进行配置和使用

http://192.168.0.197/wp-admin/setup-config.php，这里的地址就是宿主机的 IP 地址。

6.4 Docker 三剑客之 Swarm

Docker Swarm 是 Docker 官方三剑客项目之一，提供 Docker 容器集群服务，是 Docker 官方对容器云生态进行支持的核心方案。使用它，用户可以将多个 Docker 主机抽象为大规模的虚拟 Docker 服务，快速打造一套容器云平台。

6.4.1 几个概念

Swarm 集群（Cluster）： Swarm 集群（Cluster）为一组被统一管理起来的 Docker 主机。

节点：

- 管理节点（manager node）：负责响应外部对集群的操作请求，并维持集群中资源，分发任务给工作节点。同时，多个管理节点之间通过 Raft 协议构成共识。一般推荐每个集群设置 5 个或 7 个管理节点；
- 工作节点（worker node）：负责执行管理节点安排的具体任务。默认情况下，管理节点自身也同时是工作节点。每个工作节点上运行代理（agent）来汇报任务完成情况。

服务：一个服务可以由若干个任务组成，每个任务为某个具体的应用。服务还包括对应的存储、网络、端口映射、副本个数、访问配置、升级配置等附加参数。

- 复制服务（replicated services）模式：默认模式，每个任务在集群中会存在若干副本，这些副本会被管理节点按照调度策略分发到集群中的工作节点上。此模式下可以使用-replicas 参数设置副本数量；
- 全局服务（global services）模式：调度器将在每个可用节点都执行一个相同的任务。该模式适合运行节点的检查，如监控应用等。

任务：任务是 Swarm 集群中最小的调度单位，即一个指定的应用容器。

6.4.2 环境准备

准备 3 台 linux 主机（swarm191,swarm192,swarm193）；

分别配置好 ip 地址，使它全可以互通；

分配一下/etc/hosts 文件，使 3 台主机可以使用主机名通讯（比较方便一点）；

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.10.191 swarm191
192.168.10.192 swarm192
192.168.10.193 swarm193
~
```

在 swarm191 这台主机上安装好 docker

参照本文 2.2

在 swarm191 上安装 machine(其它节点使用 machine 来部署)

具体参照本文 6.2

6.4.3 创建集群

docker swarm init [OPTIONS]

Options:

--advertise-addr:指定监听的 IP 地址和端口 (format: <ip|interface>[:port])，默认为所有网口的 2377 端口。

--autolock：自动锁定管理服务的启停操作，对服务进行启动或停止都需要通过口令来解锁；

--availability string：节点的可用性，包括 active、pause、drain 三种，默认为 active；

--cert-expiry duration：根证书的过期时长，默认为 90 天；

--data-path-addr：指定数据流量使用的网络接口或地址；

--dispatcher-heartbeat duration：分配组件的心跳时长，默认为 5 秒；

--external-ca external-ca：指定使用外部的证书签名服务地址；

--force-new-cluster：强制创建新集群；

--max-snapshots uint：Raft 协议快照保留的个数；

--snapshot-interval uint：Raft 协议进行快照的间隔（单位为事务个数），默认为 10 000 个事物；

--task-history-limit int：任务历史的保留个数，默认为 5。

```
[root@swarm191 ~]# docker swarm init
Swarm initialized: current node (b4zh6lmo2egx7krg45j55iff0) is now a manager.
To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-3u6gdfuyrtln8or4dngvzt4h8mijjdymw1jftfl6l8xwbfb0h0-c1kkwn73edpqdlan3kcjj24w2 192.168.18.191:2377
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

群集创建成功，上图中红框内的一串字符需要保存下来，用于其它节点加入集群时使用。

使用 netstat –tupln 命令看一下端口：

```
[root@swarm191 ~]# netstat -tupln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1020/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      1190/master
tcp6       0      0 :::22                   :::*                    LISTEN      1020/sshd
tcp6       0      0 ::1:25                  :::*                    LISTEN      1190/master
tcp6       0      0 :::2377                 :::*                    LISTEN      1086/dockerd
tcp6       0      0 :::7946                 :::*                    LISTEN      1086/dockerd
udp        0      0 0.0.0.0:4789            0.0.0.0:*                           -
udp        0      0 127.0.0.1:323           0.0.0.0:*                           694/chronyd
udp6       0      0 :::7946                 :::*                                1086/dockerd
udp6       0      0 ::1:323                 :::*                                694/chronyd
```

红框内的三个端口，就是 swarm 开放的三个端口，我们需要在防火墙中把它们放通

firewall-cmd --add-port=2377/tcp --permanent

firewall-cmd --add-port=2377/tcp

firewall-cmd --add-port=7946/tcp --permanent

firewall-cmd --add-port=7946/tcp

firewall-cmd --add-port=4789/udp --permanent

firewall-cmd --add-port=4789/udp

查看一下集群的信息：

docker info

```
 Debug Mode: false
 Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Docker Buildx (Docker Inc., v0.9.1-docker)
  scan: Docker Scan (Docker Inc., v0.17.0)

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 20.10.18
 Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: active
  NodeID: b4zh6lmo2egx7krg45j55iff0
  Is Manager: true
  ClusterID: ebwjho7wgn3jgxw03vze27dfq
  Managers: 1
  Nodes: 1
  Default Address Pool: 10.0.0.0/8
  SubnetSize: 24
  Data Path Port: 4789
  Orchestration:
   Task History Retention Limit: 5
  Raft:
   Snapshot Interval: 10000
   Number of Old Snapshots to Retain: 0
   Heartbeat Tick: 1
   Election Tick: 10
  Dispatcher:
   Heartbeat Period: 5 seconds
  CA Configuration:
   Expiry Duration: 3 months
   Force Rotate: 0
  Autolock Managers: false
  Root Rotation In Progress: false
  Node Address: 192.168.18.191
  Manager Addresses:
   192.168.18.191:2377
 Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 9cd3357b7fd7218e4aec3eae239db1f68a5a6ec6
 runc version: v1.1.4-0-g5fd4c4d
 init version: de40ad0
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 3.10.0-1160.el7.x86_64
 Operating System: CentOS Linux 7 (Core)
 OSType: linux
 Architecture: x86_64
 CPUs: 1
 Total Memory: 972.3MiB
 Name: swarm191
 ID: UFKZ:PZPV:YNLR:NPSI:SSZI:PPFU:XO7J:ECQ4:RS2I:KGUN:TVAK:5WQE
 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

[root@swarm191 ~]#
[root@swarm191 ~]#
```

这里会把集群的基本信息列出来。

查看节点信息：

docker node ls

```
[root@swarm191 ~]# docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
b4zh6lmo2egx7krg45j55iff0 *   swarm191   Ready     Active         Leader           20.10.18
```

目前只有一个节点，也就是创建群集的本机。

6.4.4 其它节点加入集群

docker swarm join [OPTIONS] HOST:PORT

options:

--advertise-addr:指定管理节点的 IP 地址和端口 (format: <ip|interface>[:port])，默认为 2377 端口。

--availability string：节点的可用性，包括 active、pause、drain 三种，默认为 active；

--data-path-addr：指定数据流量使用的网络接口或地址；

--listen-addr node-addr 监听端口 (format: <ip|interface>[:port]) (default 0.0.0.0:2377)

--token：标识字符串（就是创建集群时产生的那串字符）

我们到另一台主机上执行：

docker swarm join --token xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ip:port

```
[root@swarm192 ~]# docker swarm join --token SWMTKN-1-3u6gdfuyrtln8or4dngvzt4h8mijjdymw1jftfl6l8xwbfb0h0-c1kkwn73edpqdlan3kcjj24w2 192.16
8.18.191
This node joined a swarm as a worker.
```

这里显示已经加入了集群。

到管理节点上看一下节点信息：

```
[root@swarm191 nginx]# docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
g72rhhrn9sbpml08kdv3xkziq *   swarm191   Ready     Active         Leader           20.10.18
tbsqe3a2w3i005aokf744hc4u     swarm192   Ready     Active                          20.10.18
ht0dmg0xa1ys4roketudru99t     swarm193   Ready     Active                          20.10.18
```

发现已经有三个节点了。swarm191 那台是 leader，也就是管理节点。

看下工作节点上的端口信息：

```
[root@swarm192 ~]# netstat -tupln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1020/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      1191/master
tcp6       0      0 :::7946                 :::*                    LISTEN      1086/dockerd
tcp6       0      0 :::22                   :::*                    LISTEN      1020/sshd
tcp6       0      0 ::1:25                  :::*                    LISTEN      1191/master
udp        0      0 127.0.0.1:323           0.0.0.0:*                           695/chronyd
udp        0      0 0.0.0.0:4789            0.0.0.0:*                           -
udp6       0      0 :::7946                 :::*                                1086/dockerd
udp6       0      0 ::1:323                 :::*                                695/chronyd
[root@swarm192 ~]#
```

工作节点会开放两个端口，我们同样也需要在防火墙中把这两个端口打开。

firewall-cmd --add-port=7946/tcp --permanent

firewall-cmd --add-port=7946/tcp

firewall-cmd --add-port=7946/udp--permanent

firewall-cmd --add-port=7946/udp

firewall-cmd --add-port=4789/udp --permanent

firewall-cmd --add-port=4789/udp

6.4.5 服务管理

docker service COMMAND

COMMAND:

create:创建一个服务

inspect:查看服务的详细信息

logs:获取服务的日志

ls:列出服务

ps:列出服务内的任务（容器）

rm:删除服务

rollback:恢复服务的配置变更

scale:扩展一个或多个被复制的服务

update:更新服务

6.4.5.1 创建建服务

docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

options:

--config config：指定暴露给服务的配置；

--constraint list：应用实例在集群中被放置时的位置限制；

-d，detach：不等待创建后对应用进行状态探测即返回；

--dns list：自定义使用的 DNS 服务器地址；

--endpoint-mode string：指定外部访问的模式，包括 vip（虚地址自动负载均衡）或 dnsrr（DNS 轮询）；

--e，-env list：环境变量列表；

--health-cmd string：进行健康检查的指令；

--l，-label list：执行服务的标签；

--mode string：服务模式，包括 replicated（默认）或 global；

--replicas uint：指定实例的复制份数；

--secret secret：向服务暴露的秘密数据；

--u，-user string：指定用户信息，UID：[GID]；

--w，-workdir string：指定容器中的工作目录位置。

-p, --publish:指定宿主机到容器的端口映射

docker service create --name nginx -p 8080:80 --replicas 2 nginx

这里我们创建一个 nginx 服务，服务名为 nginx,端口映射为宿主机的 8080 端口映射到服务的 80 端口，实例复制份数为 2 份。

```
[root@swarm191 nginx]# docker service create --name nginx -p 8080:80 --replicas 2 nginx
lgo5jjo5xd7ynkqglgwid1nht
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

6.4.5.2 列出服务

docker service ls

```
[root@swarm191 nginx]# docker service ls
ID             NAME      MODE         REPLICAS   IMAGE          PORTS
qxi00wd480tb   nginx     replicated   2/2        nginx:latest   *:8080->80/tcp
```

这里服务已经起来了。

6.4.5.3 查看服务详细信息

docker service inspect nginx

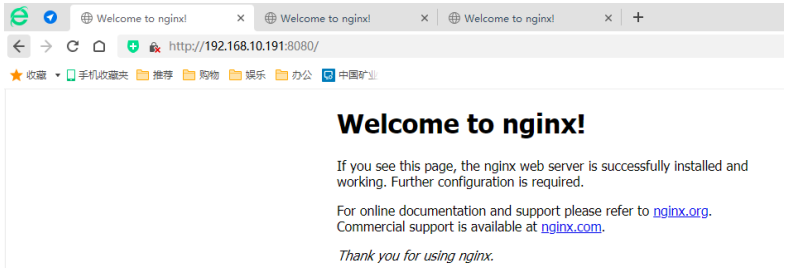
6.4.5.4 列出任务

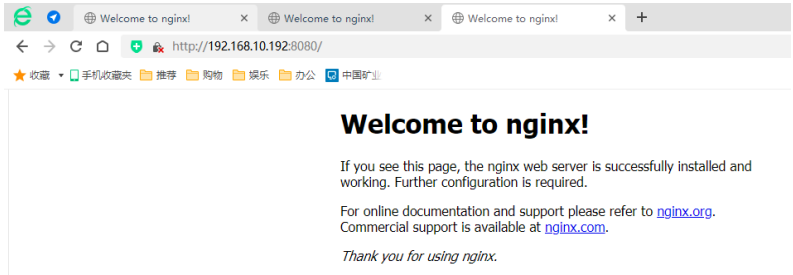
docker service ps nginx

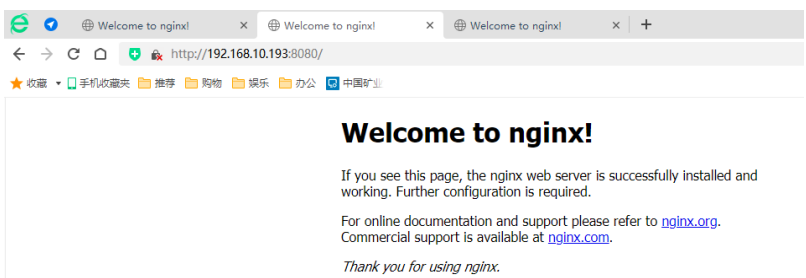
```
[root@swarm191 nginx]# docker service ps nginx
ID             NAME      IMAGE          NODE       DESIRED STATE   CURRENT STATE            ERROR     PORTS
7pyzvvv9zc6i   nginx.1   nginx:latest   swarm192   Running         Running 12 minutes ago
fu8e8w8pe34u   nginx.2   nginx:latest   swarm191   Running         Running 12 minutes ago
```

有两个任务（容器），分别运行在 swarm192 和 swarm191 这两个节点上。

我们看下，nginx 有没有正常启动。







这里我们发现，使用三外节点的 IP 都可以访问 nginx.

即使任务并没有跑在 swarm193 上，用 swarm193 的地址，也可以正常访问，这里有点神奇，下面一节我们再来研究这个。

如果我们让一个节点上的任务（容器）停止一下，发现还是可以正常访问，因为我们这个任务（容器）有两个，可以帮到冗余的效果，且 swarm 还会做负载均衡，将负载分别分配到各个任务中。

6.4.5.5 扩展服务

将服务的复制份数增加或减少。

如上例中我，我们创建 nginx 服务时，选择的复制份数是 2 份，但我们有三个节点，就可以将复制份数扩展到 3 份。

docker service scale nginx=3

```
[root@swarm191 nginx]#  docker service scale nginx=3
nginx scaled to 3
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
```

```
[root@swarm191 nginx]# docker service ps nginx
ID             NAME      IMAGE          NODE       DESIRED STATE   CURRENT STATE            ERROR     PORTS
7pyzvvv9zc6i   nginx.1   nginx:latest   swarm192   Running         Running 29 minutes ago
fu8e8w8pe34u   nginx.2   nginx:latest   swarm191   Running         Running 29 minutes ago
mcmc6xkx6o28   nginx.3   nginx:latest   swarm193   Running         Running 24 seconds ago
```

已经扩展为 3 份了

6.4.5.6 更新服务

docker service update [OPTIONS] SERVICE

Options:

--args command：服务的命令参数；

--config-add config：增加或更新一个服务的配置信息；

--config-rm list：删除一个配置文件；

--constraint-add list：增加或更新放置的限制条件；

--constraint-rm list：删除一个限制条件；

--d，-detach：执行后返回，不等待服务状态校验完整；

--dns-add list：增加或更新 DNS 服务信息；

--dns-rm list：删除 DNS 服务信息；

--endpoint-mode string：指定外部访问的模式，包括 vip（虚地址自动负载均衡）或 dnsrr（DNS 轮询）；

--entrypoint command：指定默认的入口命令；

--env-add list：添加或更新一组环境变量；

--env-rm list：删除环境变量；

--health-cmd string：进行健康检查的指令；

--label-add list：添加或更新一组标签信息；

--label-rm list：删除一组标签信息；

--no-healthcheck：不进行健康检查；

--publish-add port：添加或更新外部端口信息；

--publish-rm port：删除端口信息；

--q，-quiet：不显示进度信息；

--read-only：指定容器的文件系统为只读；

--replicas uint：指定服务实例的复制份数；

--rollback：回滚到上次配置；

--secret-add secret：添加或更新服务上的秘密数据；

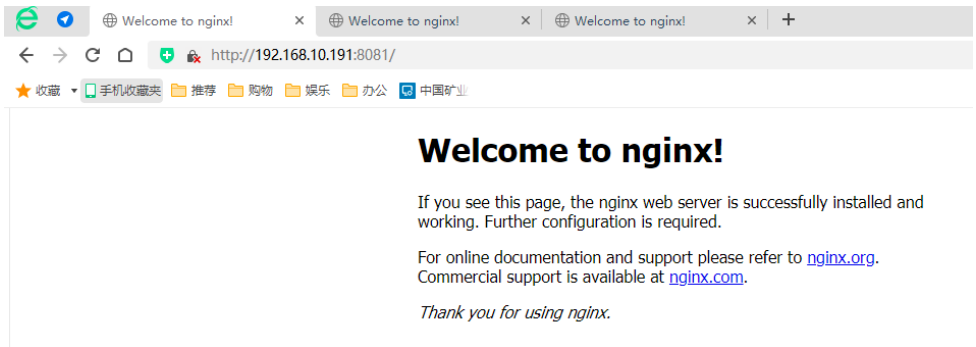
--secret-rm list：删除服务上的秘密数据；

--update-parallelism uint：更新执行的并发数；

我们给 nginx 这个服务增加一个端口映射：

docker service update --publish-add 8081:80 nginx

```
[root@swarm191 nginx]# docker service ls
ID             NAME      MODE         REPLICAS   IMAGE          PORTS
qxi00wd480tb   nginx     replicated   2/2        nginx:latest   *:8080-8081->80/tcp
```



这样，我们通过 8081 端口也可以访问 nginx 了。

6.4.6 探索 swarm 的工作原理

上面我们只是从表面上看到了 swarm 的基本功能，接下来，我们深入探索一下 swarm 的工作原理。

6.4.6.1 观测外部访问 swarm 时的效果

先做一个小实验：

```
[root@swarm191 ~]# docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
1aibcjhnbhkinr5hvwwpb3z6m *   swarm191   Ready     Active         Leader           20.10.18
p6tw9zkzmjdvgrf6guou8yyie     swarm192   Ready     Active                          20.10.18
68qfx98uhgghmjsd0vupognqq     swarm193   Ready     Active                          20.10.18
```

还是有三个节点，swarm191,swarm192,swarm193

创建一个 nginx 服务，复制份数为 2。

docker service create --name nginx -p 8080:80 --replicas 2 nginx:1.0

```
[root@swarm191 ~]# docker service ps nginx
ID             NAME      IMAGE       NODE       DESIRED STATE   CURRENT STATE            ERROR     PORTS
mw25as41jzg7   nginx.1   nginx:1.0   swarm193   Running         Running 56 seconds ago
97gihlau7tp8   nginx.2   nginx:1.0   swarm191   Running         Running 57 seconds ago
```

```
[root@swarm191 ~]# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED              STATUS              PORTS     NAMES
dba99488d555   nginx:1.0   "/bin/sh -c 'nginx;s…"   About a minute ago   Up About a minute   80/tcp    nginx.2.97gihlau7tp8s5d2alhw63498
```

```
[root@swarm193 ~]# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED         STATUS         PORTS     NAMES
a1cca632c2dc   nginx:1.0   "/bin/sh -c 'nginx;s…"   2 minutes ago   Up 2 minutes   80/tcp    nginx.1.mw25as41jzg7d8ydp3zehu2bt
```

```
[root@swarm192 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

可以看到，这个 nginx 服务运行在 swarm191 和 swarm193 上。

在三个节点上分别用 docker ps 这个命令查看一下容器：

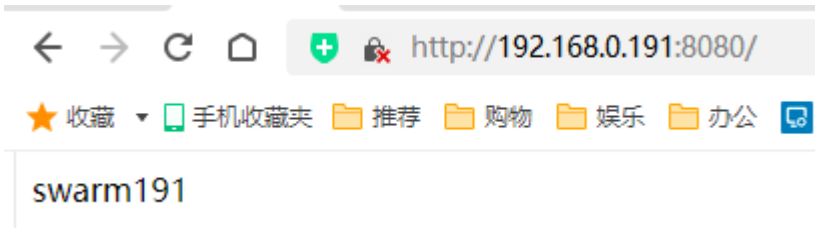
发现在 swarm191 和 swarm192 上各生成了一个容器，而 swarm193 上没有。

现在我们先用 docker exec –it *<container name>* bash 这个命令进入 swarm191 和 swarm192 上的容器，修改一下 nginx 的 index.html 文件。

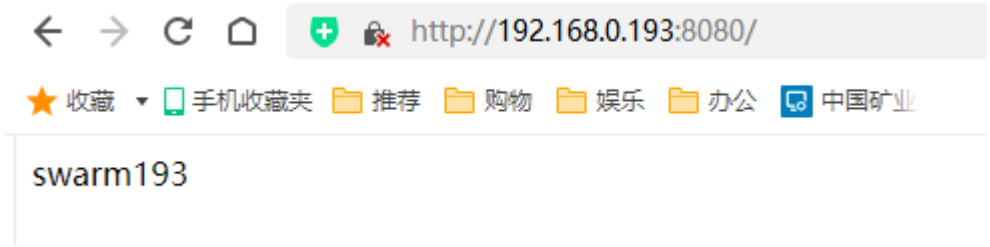
swarm191 上的容器里的/usr/share/nginx/html/index.html 文件内容改为 swarm191

swarm193 上的容器里的/usr/share/nginx/html/index.html 文件内容改为 swarm193

改完以后，访问 swarm191 节点，显示的内容是 swarm191，说明访问的是在节点 swarm191 上的那个容器。



访问 swarm193 节点，显示的内容是 swarm193，说明访问的是在节点 swarm193 上的那个容器。

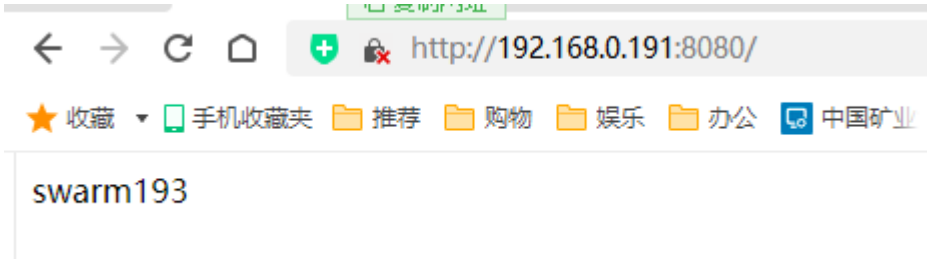


那么访问 swarm192 节点，会显示什么内容呢？它会访问哪个节点上的容器呢？

最开始，它会访问到 swarm191，你尝试不断刷新网页后，它又会访问到 swarm193。

现在，我们把 191 上的容器暂停掉，再来观测：

```
[root@swarm191 ~]# docker pause nginx.2.97gihlau7tp8s5d2alhw63498
nginx.2.97gihlau7tp8s5d2alhw63498
[root@swarm191 ~]# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS                    PORTS     NAMES
dba99488d555   nginx:1.0   "/bin/sh -c 'nginx;s…"   18 minutes ago   Up 18 minutes (Paused)    80/tcp    nginx.2.97gihlau7tp8s5d2alhw63498
```



这里我们发现，它会跳转到 swarm193 去

从这个实验我可以得出以下结论：

1、 当任务（容器）在本节点上，有外部访问时，会优先转发到本节点的容器上；

2、 当任务（容器）不在本节点上，有外部访问时，会转发到其它节点上，并且会访问其它不同的节点，也就是可以实现负载均衡的功能。

3、 当本节点上的容器故障，有外部访问时，会转发到其它节点上，也就是可以实现冗余的功能。

6.4.6.2 理解网络命名空间

要想深入探索 swarm 的工作原理，我们首先需要理解网络命名空间。

在 Linux 中，网络命名空间可以被认为是隔离的拥有单独网络栈（网卡、路由转发表、iptables）的环境。网络命名空间经常用来隔离网络设备和服务，只有拥有同样网络命名空间的设备，才能看到彼此。

我们重新开一下虚拟来做网络命名空间的实验。

1、 添加两个网络命名空间

```
ip netns add test0
ip netns add test1
```

2、 查看网络命名空间

```
ip netns ls
```

```
[root@localhost ~]# ip netns ls
test1
test0
```

这里我们可以看到刚刚创建的网络命名空间了。

3、 创建一对 veth，两个网络空间之间要通信的话，需要 veth 对。把网络空间比喻成两个水池的话，veth 就是连接水池的管道有了管道，两个水池的水才能相互流通

ip link add 第一个 veth 名称 type veth peer name 第二个 veth 名称

ip link add veth0 type veth peer name veth1

使用 ip link show 查看一下

已经生成了 veth 对

```
[root@localhost ~]# ip link add veth0 type veth peer name veth1
[root@localhost ~]# 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: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:b8:c7:e4 brd ff:ff:ff:ff:ff:ff
5: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 62:7b:10:c1:69:dc brd ff:ff:ff:ff:ff:ff
6: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 1e:52:e7:4c:54:f6 brd ff:ff:ff:ff:ff:ff
```

4、 将两个虚拟网卡分别放到不同的命名空间中

ip link set veth0 netns test0

ip link set veth1 netns test1

5、 为 veth 添加 ip 地址

ip netns exec test0 ip addr add 192.168.10.10/24 dev veth0

ip netns exec test1 ip addr add 192.168.10.11/24 dev veth1

6、 启动两块网卡

ip netns exec test0 ip link set up dev veth0

ip netns exec test1 ip link set up dev veth1

7、 查看两个命名空间的 IP 地址

ip netns exec test0 ip a

ip netns exec test1 ip a

```
[root@localhost ~]# ip netns exec test0 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
6: veth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 1e:52:e7:4c:54:f6 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 192.168.10.10/24 scope global veth0
       valid_lft forever preferred_lft forever
    inet6 fe80::1c52:e7ff:fe4c:54f6/64 scope link
       valid_lft forever preferred_lft forever
[root@localhost ~]# ip netns exec test1 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
5: veth1@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 62:7b:10:c1:69:dc brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.10.11/24 scope global veth1
       valid_lft forever preferred_lft forever
    inet6 fe80::607b:10ff:fec1:69dc/64 scope link
       valid_lft forever preferred_lft forever
```

注意红框的数字，他们交叉相等，说明这两个命空间已经成功建立了 veth 对，应该是可以互通的。

ip netns exec test0 ping 192.168.10.11

```
[root@localhost ~]# ip netns exec test0 ping 192.168.10.11
PING 192.168.10.11 (192.168.10.11) 56(84) bytes of data.
64 bytes from 192.168.10.11: icmp_seq=1 ttl=64 time=0.055 ms
64 bytes from 192.168.10.11: icmp_seq=2 ttl=64 time=0.057 ms
64 bytes from 192.168.10.11: icmp_seq=3 ttl=64 time=0.244 ms
```

我们再建一个命名空间 test3，并重新创建一块虚拟网卡，看会不会通

ip netns add test2

ip link add veth2 type veth

ip link set veth2 netns test2

ip netns exec test2 ip addr add 192.168.10.12/24 dev veth2

ip netns exec test2 ip link set up dev veth2

ip netns exec test2 ip a

```
[root@localhost ~]# ip netns exec test2 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
8: veth2@if7: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state LOWERLAYERDOWN group default qlen 1000
    link/ether a2:a6:6f:fe:5e:60 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.10.12/24 scope global veth2
       valid_lft forever preferred_lft forever
```

```
[root@localhost ~]# ip netns exec test2 ping 192.168.10.10
PING 192.168.10.10 (192.168.10.10) 56(84) bytes of data.
```

发现，是 ping 不通的。

6.4.6.2swarm 集群通讯过程探索

上面，我们只是看到了 swarm 集群工作的表面，接下来，我们来尝试探索一下 swarm 集群整个通讯的过程。

我们应用上一小节的知识，就可以很容易地搞清楚 swarm 的网络结构了。

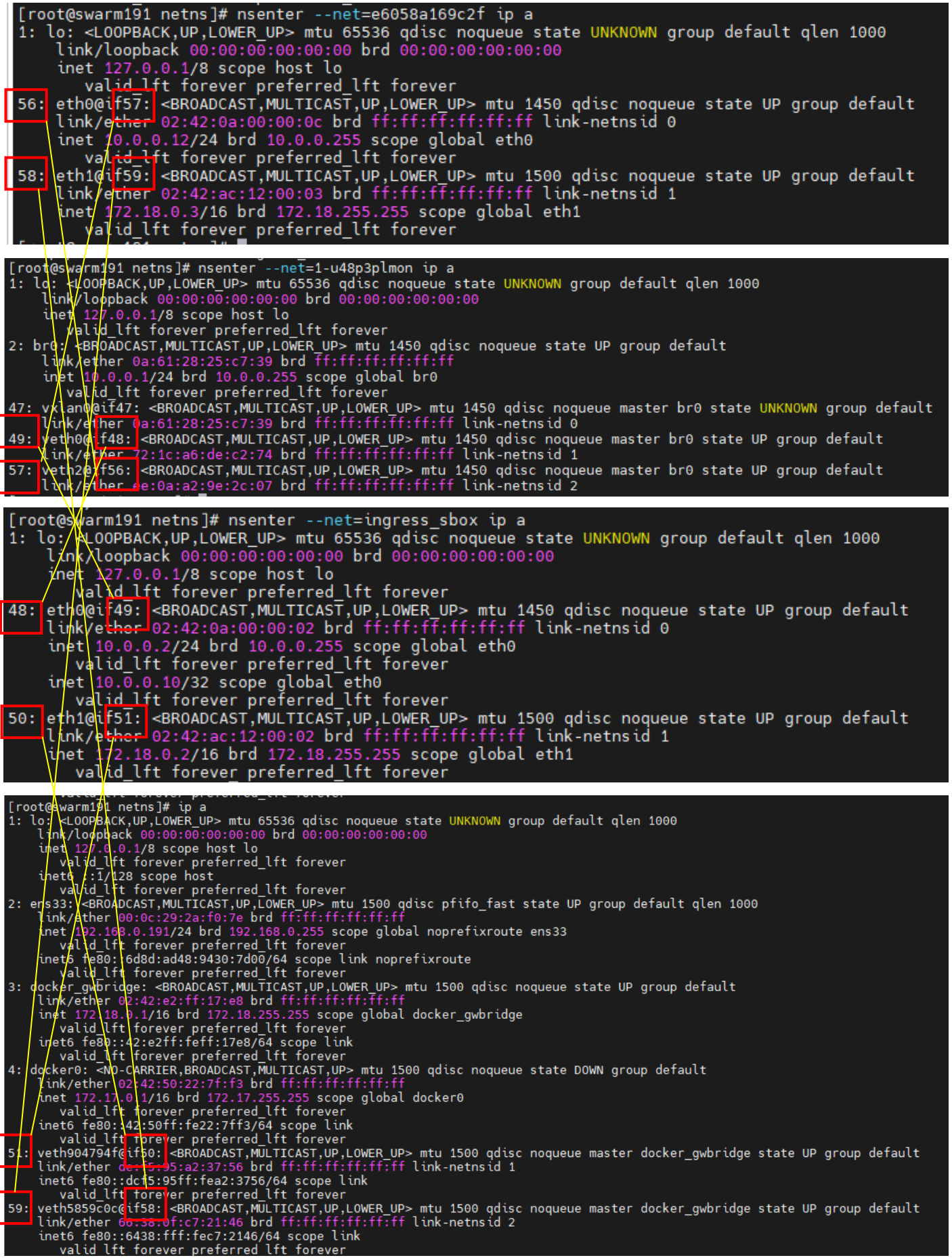
首先，docker swarm 会生成三个网络命名空间，具体位置在/run/docker/netns 这个目录下面。

```
[root@swarm191 netns]# ll
total 0
-r--r--r--. 1 root root 0 Oct  5 23:29 1-u48p3plmon
-r--r--r--. 1 root root 0 Oct  5 23:37 e6058a169c2f
-r--r--r--. 1 root root 0 Oct  5 23:29 ingress_sbox
```

但是我们无法使用 ip netns 来管理这几个网络命名空间，但可以使用 nsenter 这个工具来管理。

nsenter --net=<filename> [command]

分别看一下三个网络命名空间、宿主机容器的 IP。


[root@swarm191 netns]# nsenter --net=e6058a169c2f 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
56: eth0@if57: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
link/ether 02:42:0a:00:00:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 10.0.0.12/24 brd 10.0.0.255 scope global eth0
valid_lft forever preferred_lft forever
58: eth1@if59: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 1
inet 172.18.0.3/16 brd 172.18.255.255 scope global eth1
valid_lft forever preferred_lft forever
[root@swarm191 netns]# nsenter --net=1-u48p3plmon 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
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
link/ether 0a:61:28:25:c7:39 brd ff:ff:ff:ff:ff:ff
inet 10.0.0.1/24 brd 10.0.0.255 scope global br0
valid_lft forever preferred_lft forever
47: vxlan0@if47: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UNKNOWN group default
link/ether 0a:61:28:25:c7:39 brd ff:ff:ff:ff:ff:ff link-netnsid 0
49: veth0@if48: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP group default
link/ether 72:1c:a6:de:c2:74 brd ff:ff:ff:ff:ff:ff link-netnsid 1
57: veth2@if56: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP group default
link/ether ee:0a:a2:9e:2c:07 brd ff:ff:ff:ff:ff:ff link-netnsid 2
[root@swarm191 netns]# nsenter --net=ingress_sbox 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
48: eth0@if49: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 10.0.0.2/24 brd 10.0.0.255 scope global eth0
valid_lft forever preferred_lft forever
inet 10.0.0.10/32 scope global eth0
valid_lft forever preferred_lft forever
50: eth1@if51: <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 link-netnsid 1
inet 172.18.0.2/16 brd 172.18.255.255 scope global eth1
valid_lft forever preferred_lft forever
[root@swarm191 netns]# 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: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
link/ether 00:0c:29:2a:f0:7e brd ff:ff:ff:ff:ff:ff
inet 192.168.0.191/24 brd 192.168.0.255 scope global noprefixroute ens33
valid_lft forever preferred_lft forever
inet6 fe80::6d8d:ad48:9430:7d00/64 scope link noprefixroute
valid_lft forever preferred_lft forever
3: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether 02:42:e2:ff:17:e8 brd ff:ff:ff:ff:ff:ff
inet 172.18.0.1/16 brd 172.18.255.255 scope global docker_gwbridge
valid_lft forever preferred_lft forever
inet6 fe80::42:e2ff:feff:17e8/64 scope link
valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
link/ether 02:42:50:22:7f:f3 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::42:50ff:fe22:7ff3/64 scope link
valid_lft forever preferred_lft forever
51: veth904794f@if50: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group default
link/ether de:f5:95:a2:37:56 brd ff:ff:ff:ff:ff:ff link-netnsid 1
inet6 fe80::dcf5:95ff:fea2:3756/64 scope link
valid_lft forever preferred_lft forever
59: veth5859c0c@if58: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group default
link/ether 66:38:0f:c7:21:46 brd ff:ff:ff:ff:ff:ff link-netnsid 2
inet6 fe80::6438:fff:fec7:2146/64 scope link
valid_lft forever preferred_lft forever

看一下容器的 IP

```
[root@swarm191 netns]# docker exec nginx.2.97gihlau7tp8s5d2alhw63498 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
56: eth0@if57: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.12/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
58: eth1@if59: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
```

发现容器的 IP 和 e6058a169c2f 是一样的，可以认定，容器实际上使用的是 e6058a169c2f 这个网络命名空间。

再看一下 docker 系统中的网络，docker network ls

```
[root@swarm191 netns]# docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
9156c69785e8   bridge            bridge    local
0076b37e2293   docker_gwbridge   bridge    local
f58d3ab04e3d   host              host      local
u48p3plmonpb   ingress           overlay   swarm
eab673bf5729   none              null      local
```

这里 docker_gwbridge 和 ingress 这两个网络是 swarm 创建的。

我们先看一下 ingress 这个网络的详细信息

```
[root@swarm191 netns]# docker network inspect ingress
[
    {
        "Name": "ingress",
        "Id": "u48p3plmonpbwz7tiakm9sdz8",
        "Created": "2022-10-05T23:29:51.380357089-04:00",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
                }
            ]
```

注意看这个 ID，和这个/run/docker/netns/1-u48p3plmon 网络命名空间太像了，我认定，这个 ingress 和/run/docker/netns/1-u48p3plmon，就是同一个命名空间。

至于 docker 里的 docker_gwbridge 这个网络，就是宿主机上的那个 docker_gwbridge 网络。

```
[root@swarm191 netns]# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:50ff:fe22:7ff3  prefixlen 64  scopeid 0x20<link>
        ether 02:42:50:22:7f:f3  txqueuelen 0  (Ethernet)
        RX packets 13  bytes 8830 (8.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 1914 (1.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker_gwbridge: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.0.1  netmask 255.255.0.0  broadcast 172.18.255.255
        inet6 fe80::42:e2ff:feff:17e8  prefixlen 64  scopeid 0x20<link>
        ether 02:42:e2:ff:17:e8  txqueuelen 0  (Ethernet)
        RX packets 466782  bytes 77635396 (74.0 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 349530  bytes 44474694 (42.4 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
[root@swarm191 netns]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.024250227ff3       no
docker_gwbridge         8000.0242e2ff17e8       no              veth5859c0c
                                                                veth904794f
[root@swarm191 netns]#
```

而且这个 docker_gwbridge 又桥接了两块虚拟网卡。

```
[root@swarm191 netns]# 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: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:2a:f0:7e brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.191/24 brd 192.168.0.255 scope global noprefixroute ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::6d8d:ad48:9430:7d00/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:e2:ff:17:e8 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global docker_gwbridge
       valid_lft forever preferred_lft forever
    inet6 fe80::42:e2ff:feff:17e8/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:50:22:7f:f3 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::42:50ff:fe22:7ff3/64 scope link
       valid_lft forever preferred_lft forever
51: veth904794f@if50: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group default
    link/ether de:f5:95:a2:37:56 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::dcf5:95ff:fea2:3756/64 scope link
       valid_lft forever preferred_lft forever
59: veth5859c0c@if58: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group default
    link/ether 66:38:0f:c7:21:46 brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::6438:fff:fec7:2146/64 scope link
       valid_lft forever preferred_lft forever
```

就是这两块，veth51 和 50.

根据我们上面的连线，51 和 50 其实是连接到 ingress-box 这个命名空间的。

再看一下 iptables

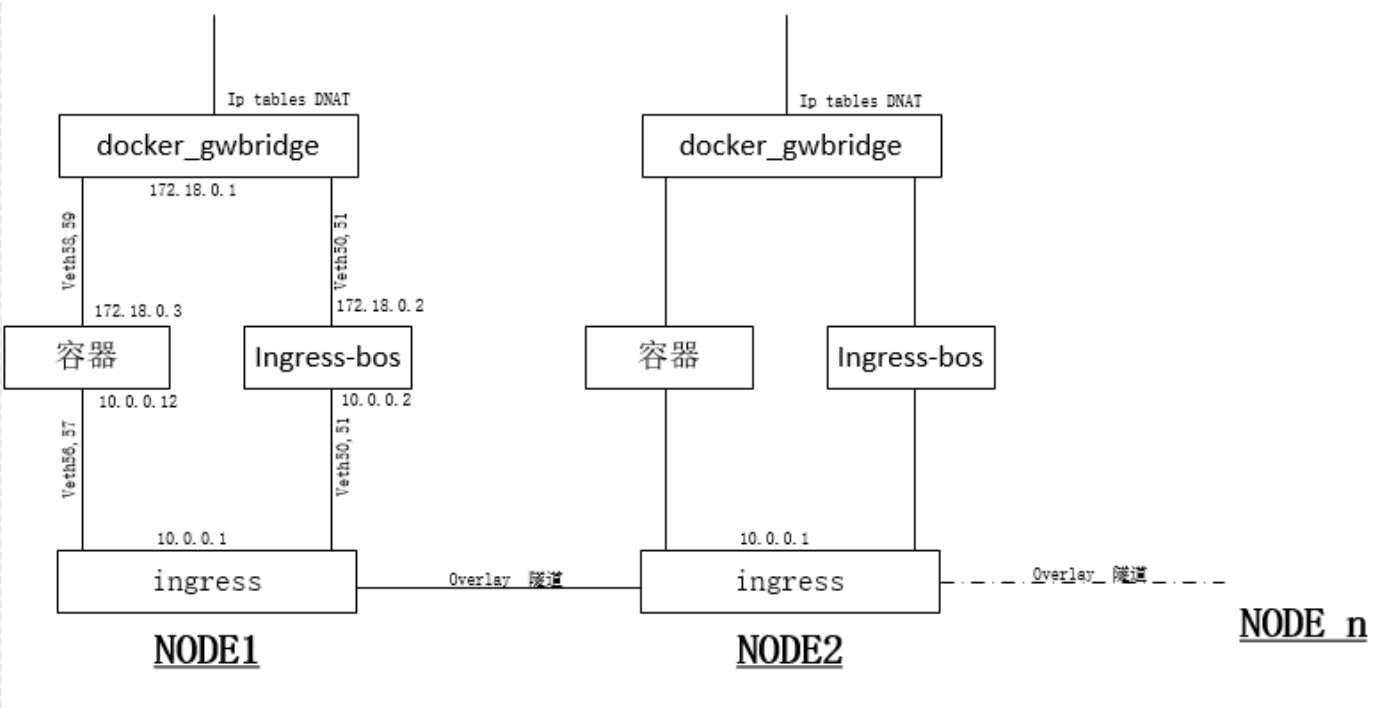
```
[root@swarm191 ~]# iptables -t nat -L | grep DNAT
DNAT       tcp  --  anywhere             anywhere             tcp dpt:webcache to:172.18.0.2:8080
```

这里有一条指向 172.18.0.2:8080 的源地址 NAT，也就是指向 Ingress-box.

根据以上分析，我可以得出以下网络拓扑图：

结论：

经过以上的探索，我们可以猜想，swarm 集群的拓扑应该如下图：



当外部访问到宿主机 IP 时，通过 iptables DNAT，将数据包丢到 ingress-box,再由 ingress-box 通过 ingress 网络选择丢到哪个容。

至于 ingress-box 如何选择哪个容器，这个不在我们本次的讨论范围之内。

6.4.7 服务栈管理

6.3 小节，我们讲过 compose 可用于服务栈的编排，但 compose 无法在 swarm 环境中部署，在 swarm 环境中部署栈，就需要使用 docker stack。

docker stack [OPTIONS] COMMAND

Options:

--orchestrator 指定集群类型，可选项：swarm|kubernetes|all

Command:

deploy:部署或更新服务栈

ls:列出已部署的服务栈

ps:列出服务栈中的任务（容器）

rm:删除服务栈

services：列出服务栈中的服务

6.4.7.1 部署服务栈

docker stack deploy [OPTIONS] STACK

Options:

-c, --compose-file:指定一个 compose 文件

--orchestrator：指定编排格式，可选项：swarm|kubernetes|all

--prune：删除没有被引用的服务，默认为 false

--resolve-image：查询注册表以解决图像摘要和支持的平台（“always”|“changed”|“never”）,默认为 alwys.

--with-registry-auth：向 Swarm 代理发送注册表认证详细信息,默认为 false

要部署服务栈，首先要创建一个 compose.yml 文件，这个文件和 6.3 小节中 docker-compose 中的 yml 文件差不多，但是不支持以下命令。

```
build                #docker stack 不支持创建镜像，所以必须先在节点上部署好镜像文件。
cgroup_parent
container_name
devices
tmpfs
external_links
links
network_mode
restart
security_opt
userns_mode
```

我们以 6.3.5 小节中，部署 lnmp 服务栈为例。

step 1：在集群的各个节点上部署好容器镜像

```
[root@swarm191 lump]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
lump_nginx   latest    e73d609294c4   6 days ago       1.47GB
lump_mysql   latest    95140c7d9fd7   6 days ago       540MB
```

这两个镜像的创建过程，详见 6.3.5 小节

step 2:创建一个 nfs 共享存储，并挂载到群集的各个节点中，挂载的目录名和路径需要一致，用于存储数据（nginx 数据和 mysql 数据）

1) 新开一台 linux 主机，作为 nfs 服务器，主机名设为 nfs195,ip 地址设为 192.168.0.195

2) 在 nfS195 上安装 nfs 服务

yum install nfs-utils

3) 创建一个共享目录。

mkdir lnmp_data

mkdir lnmp_data/html

mkdir lnmp_data/mysql

```
[root@nfs195 ~]# chown -R nfsnobody:nfsnobody lnmp_data/
[root@nfs195 ~]# ll
total 4
-rw-------. 1 root      root      1260 Jan 14  2022 anaconda-ks.cfg
drwxr-xr-x. 4 nfsnobody nfsnobody   31 Oct 19 02:05 lnmp_data
[root@nfs195 ~]# cd lnmp_data/
[root@nfs195 lnmp_data]# ll
total 0
drwxr-xr-x. 2 nfsnobody nfsnobody 6 Oct 19 02:05 html
drwxr-xr-x. 2 nfsnobody nfsnobody 6 Oct 19 02:05 mysql
[root@nfs195 lnmp_data]# ll
```

4) 配置并启动 NFS

5) vim /etc/exports

```
/lnmp_data/ *(rw,sync,no_root_squash)
```

红框中的 no_root_squash 表示，其它主机所有连到 nfs 的 root 用户，都被映射到 nfs 服务器的 root 用户

6) 重启 rpcbind 服务,并将 rpcbind 服务设置为开机自启动

systemctl restart rpcbind

systemctl enable rpcbind

7) 启动 NFS 服务，并将 NFS 服务设置为开机自启动

systemctl start nfs-server

systemctl enable nfs-server

8) 放通防火墙相关服务

firewall-cmd --add-service=rpc-bind

firewall-cmd --add-service=rpc-bind --permanent

firewall-cmd --add-service=nfs

firewall-cmd --add-service=nfs --permanent

firewall-cmd --add-service=mountd

firewall-cmd --add-service=mountd --permanent

9) 到三个节点上查看 NFS 共享信息

```
[root@swarm191 images]# showmount -e 192.168.0.195
Export list for 192.168.0.195:
/root/lnmp_data *
```

看到这个信息，说明 nfs 服务已经启动成功。

10) 在三个节点上创建用来挂载 NFS 的目录

mkdir /lnmp_data

11) 在三个节点上挂载 NFS 共享目

mount -t nfs 192.168.0.195:/lnmp_data /lnmp_data

12) 让内核重新生成挂载信息：

partprobe

13) 查看挂载情况：

df -h

```
[root@swarm191 lnmp]# df -h
df: '/mysql': Stale file handle
Filesystem               Size  Used Avail Use% Mounted on
devtmpfs                 475M     0  475M   0% /dev
tmpfs                    487M     0  487M   0% /dev/shm
tmpfs                    487M   51M  436M  11% /run
tmpfs                    487M     0  487M   0% /sys/fs/cgroup
/dev/mapper/centos-root   17G  9.0G  8.1G  53% /
/dev/sda1               1014M  138M  877M  14% /boot
tmpfs                     98M     0   98M   0% /run/user/0
192.168.0.195:/lnmp_data  17G  3.5G   14G  21% /lnmp_data
overlay                   17G  9.0G  8.1G  53% /var/lib/docker/over
/merged
```

已经挂载成功

14) 在三个节点上将 NFS 共享挂载写入分区配置文件：

vim /etc/fstab

```
#
# /etc/fstab
# Created by anaconda on Fri Jan 14 09:14:14 2022
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/centos-root /                       xfs     defaults        0 0
UUID=f5d7dce9-627d-4698-b41f-3978397c7132 /boot                   xfs     defaults        0 0
/dev/mapper/centos-swap swap                    swap    defaults        0 0
192.168.0.195:/lnmp_data        /lnmp_data      xfs     defaults        0 0
```

确保三个节点都可以访问到 nfs 的共享目录

```
[root@swarm191 lnmp_data]# ll
total 0
drwxr-xr-x. 2 root root  6 Oct 18 21:28 html
drwxr-xr-x. 2 root root 50 Oct 18 22:53 images
drwxr-xr-x. 2 root root  6 Oct 18 21:28 mysql
```

step 3:创建一个 compose 文件

```
version: '3.4'
services:
  nginx:  #第一个服务 nginx
    image: lnmp_nginx#定义创建服务所使用的镜像文件
    labels:#打一个标签
            - nginx
    env_file:#使用一个环境变量文件
            - ./myenv.env
    depends_on:#该服务依赖于 mysql 这个服务
            - mysql
    ports:#将宿主机的 8080 端口映射到服务的 80 端口
            - 8080:80
    deploy:#定义容器复本数为 3
            replicas: 3
    volumes:#将 nginx 数据目录挂载到宿主机
            - /lnmp_data/html:/usr/share/nginx/html

  mysql:#第二个服务 mysql
    image:  lnmp_mysql#定义创建服务所使用的镜像文件
    labels:#打一个标签
            - mysql
    env_file:#使用一个环境变量文件
```

```
      - ./myenv.env
  ports:#将宿主机的 3306 端口映射到服务的 3306 端口
      - 3306:3306
  deploy:#定义容器复本数为 1
     replicas: 1
  volumes:#将 mysql 数据目录映射到宿主机
      - /lnmp_data/mysql:/var/lib/mysql
```

step 4 创建一个环境变量文件

```
DATABASE_NAME=wordpress #定义 wordpress 使用的数据库名称
DATABASE_USER=wordpress#定义 wordpress 使用的数据库用户名
DATABASE_PASSWORD_ROOT=123456#定义 mysql 的 root 密码
DATABASE_PASSWORD_USER=123456#定义给 wordpress 使用的 mysql 用户名密码
DATABASE_HOST=mysql#定义给 wordpress 使用的数据库服务器地址或主机名
```

step 5 创建服务

docker stack deploy -c docker-stack.yml lnmp

这里的 docker-stack.yml 就是我们在 step3 中创建的那个 compose 文件。

```
[root@swarm191 lnmp]# docker stack deploy -c docker-stack.yml lnmp
Creating network lnmp_default
Creating service lnmp_nginx
Creating service lnmp_mysql
```

这里我们看到，创建服务时做了三件事件上：

1） 创建了一个名为 lnmp_default 的网络

2） 创建了一个名为 lnmp_nginx 的服务

3） 创建了一个名为 lnmp_mysql 的服务

然后我们看一下，我们的 lnmp 网站有没有起来：

网站已经起来了。

6.4.7.2 列出服务栈

docker stack ls [OPTIONS]

options:

--format:指定输出的显示格式

--orchestrator：--orchestrator：指定编排格式，可选项：swarm|kubernetes|all

docker stack ls

```
[root@swarm191 lnmp]# docker stack ls
NAME      SERVICES   ORCHESTRATOR
lnmp      2          Swarm
```

这里已经列出了服务栈，里面有两个服务，编排格式为 swarm.

6.4.7.3 列出服务

docker stack services [OPTIONS] STACK

options:

```
-f, --filter: 根据提供的条件筛选输出
--format: 指定输出的显示格式
--orchestrator：--orchestrator：指定编排格式，可选项：swarm|kubernetes|all
  -q, --quiet：安静模式，只显示 id
```

```
[root@swarm191 lnmp]# docker stack services lnmp
ID             NAME          MODE         REPLICAS   IMAGE                PORTS
nu9qinjtf1ib   lnmp_mysql    replicated   1/1        lnmp_mysql:latest    *:3306->3306/tcp
sm0ek48asl4m   lnmp_nginx    replicated   2/2        lnmp_nginx:latest    *:8080->80/tcp
```

这里列出了，lnmp 这个服务栈有两个服务，分别是 lnmp_mysql 和 lnmp_nginx

6.4.7.4 列出任务

docker stack ps [OPTIONS] STACK

options:

-f, --filter: 根据提供的条件筛选输出

--format: 指定输出的显示格式

--no-resolve: 不将 id 映射到名称

--no-trunc：不截断输出

--orchestrator：--orchestrator：指定编排格式，可选项：swarm| kubernetes|all

-q, --quiet：安静模式，只显示 id

```
[root@swarm191 lnmp]# docker stack ps lnmp
ID             NAME            IMAGE               NODE       DESIRED STATE   CURRENT STATE            ERROR     PORTS
xqc9ersy4utb   lnmp_mysql.1    lnmp_mysql:latest   swarm191   Running         Running 22 minutes ago
ziv9whja6of8   lnmp_nginx.1    lnmp_nginx:latest   swarm193   Running         Running 22 minutes ago
f0s95zlvn4br   lnmp_nginx.2    lnmp_nginx:latest   swarm192   Running         Running 22 minutes ago
```

这里看到,服务栈中有三个任务(容器),一个 mysql,在 swarm191 上,两个 nginx 分别在 node swarm192 和 swarm193 上。

6.4.7.5 删除服务栈

docker stack rm [OPTIONS] STACK [STACK...]

options:

--orchestrator：--orchestrator：指定编排格式，可选项：swarm|kubernetes|all

全文完！！