

环境准备

主机环境

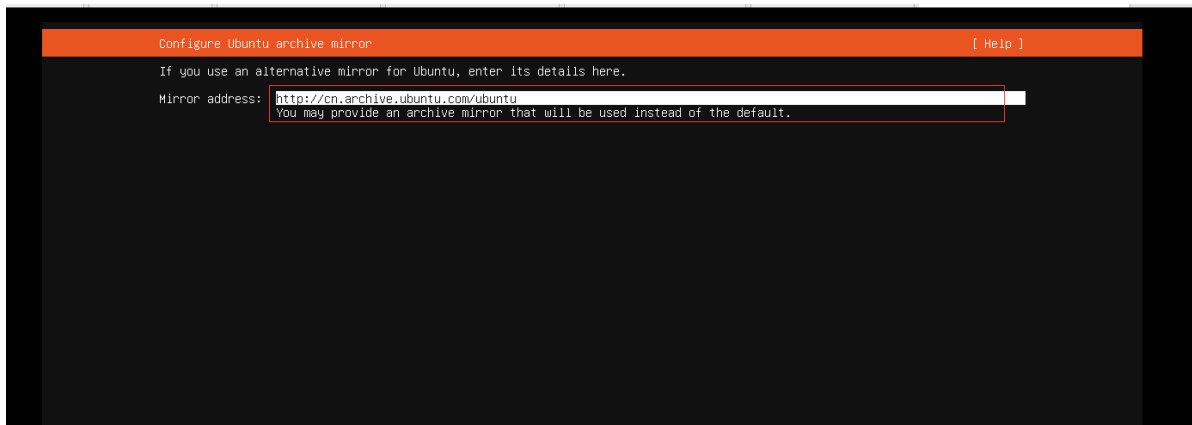
1. 宿主机环境 ubuntu-20.04.4-live-server-amd64, 下载地址:

<https://mirrors.aliyun.com/ubuntu-releases/20.04.4/ubuntu-20.04.4-live-server-amd64.iso>

2. apt 包管理器, 镜像源修改:

将 <http://cn.archive.ubuntu.com/ubuntu> 改为 <https://mirrors.tuna.tsinghua.edu.cn/ubuntu>
两种修改方式:

第一种: 在安装虚拟机时, 修改图下图:



第二种: 已经安装好系统的情况下, 修改 /etc/apt/sources.list 将对应的地址替换, 替换完成后执行 apt-get update。注意: 先备份/etc/apt/source.list文件

docker 安装

基于apt包管理器安装

1. 安装

```
sudo apt install docker.io
```

2. 卸载

```
sudo apt-get purge docker.io
sudo rm -rf /var/lib/docker
sudo rm -rf /var/lib/containerd
```

根据官方文档安装

1. 官方文档: <https://docs.docker.com/engine/install/ubuntu/>
2. 有三种安装方式: 1. 基于官方存储库安装; 2. 下载软件包安装; 3. 基于官方给出的快捷脚本安装。

使用docker官方存储库安装

1. 更新 apt 包索引，并且安装一些软件使得apt可以通过HTTPS协议访问软件库。

```
sudo apt-get update
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

2. 添加docker官方的GPG 秘钥

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

3. 设置一个标准的docker软件仓库。

```
echo \
    "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-
    archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
    /dev/null
```

4. 更新apt 包索引并查看docker 可用版本列表

```
sudo apt-get update
apt-cache madison docker-ce
```

5. 安装特定版本docker语法，例如：5:20.10.16~3-0~ubuntu-focal

```
sudo apt-get install docker-ce=<VERSION_STRING> docker-ce-cli=<VERSION_STRING>
containerd.io docker-compose-plugin
```

6. 安装docker-ce及相关软件

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

7. 检查docker engine是否正确安装

```
docker run hello-world
```

卸载

1. 卸载

```
sudo apt-get purge docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

2. 删除相关目录

```
sudo rm -rf /var/lib/docker
sudo rm -rf /var/lib/containerd
```

将用户添加到docker组

将用户添加到docker用户组后，不需要每次都输入sudo来执行docker命令了

```
//将用户从docker用户组中移除 gpasswd -d <username> docker
//将用户添加到docker 用户组
sudo addgroup -a <username> docker
sudo service docker restart
//查看用户信息
id <username>
```

退出终端，重新连接即可

docker简介

docker定义

根据官方的定义，Docker是以Docker容器为资源分割和调度的基本单位，封装整个软件运行时环境，为开发者和系统管理员设计的，用于构建、发布和运行分布式应用的平台。

引用网图：



docker解决了什么问题

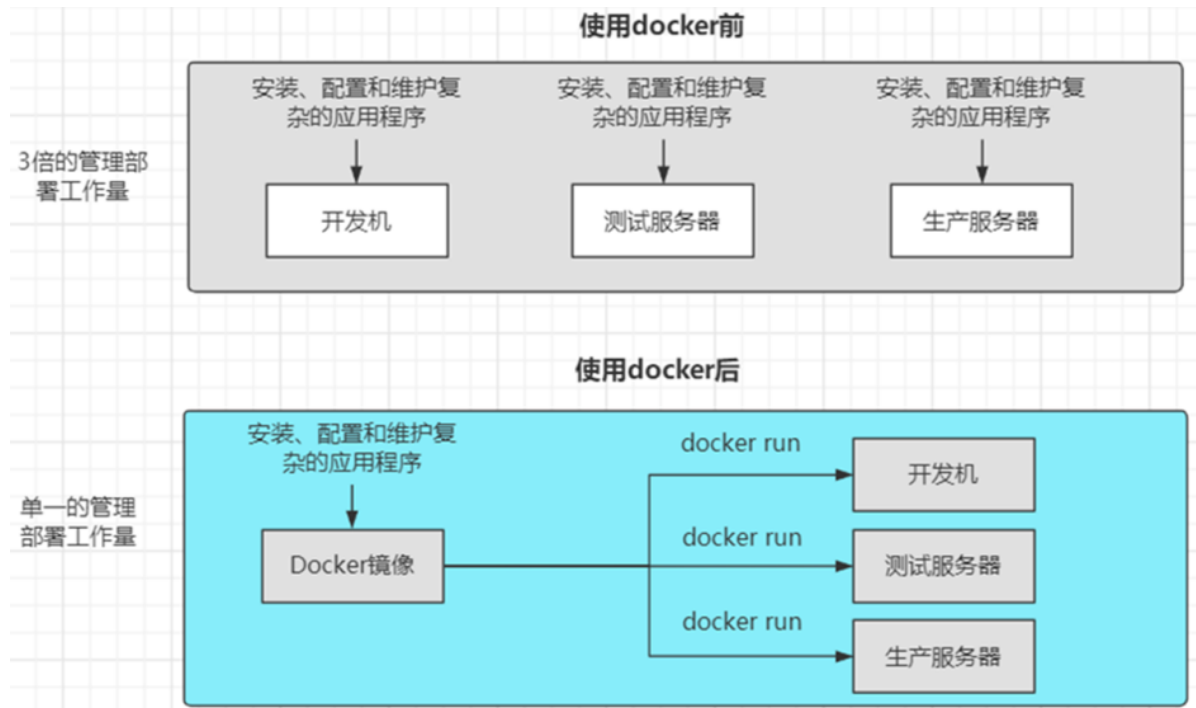
1. 解决了应用程序本地运行环境与生产运行环境不一致的问题
2. 解决了应用程序资源使用的问题，docker会一开始就为每个程序指定内存分配和CPU分配
3. 让快速扩展、弹性伸缩变得简单

docker技术边界

docker是容器化技术，针对的是应用及应用所依赖的环境做容器化。遵循单一原则，一个容器只运行一个主进程。多个进程都部署在一个容器中，弊端很多。比如更新某个进程的镜像时，其他进程也会被迫重启，如果一个进程出问题导致容器挂了，所有进程都将无法访问。再根据官网的提倡的原则而言。容器 = 应用 + 依赖的执行环境而不是像虚拟机一样，把一堆进程都部署在一起。

docker给我们带来了哪些改变

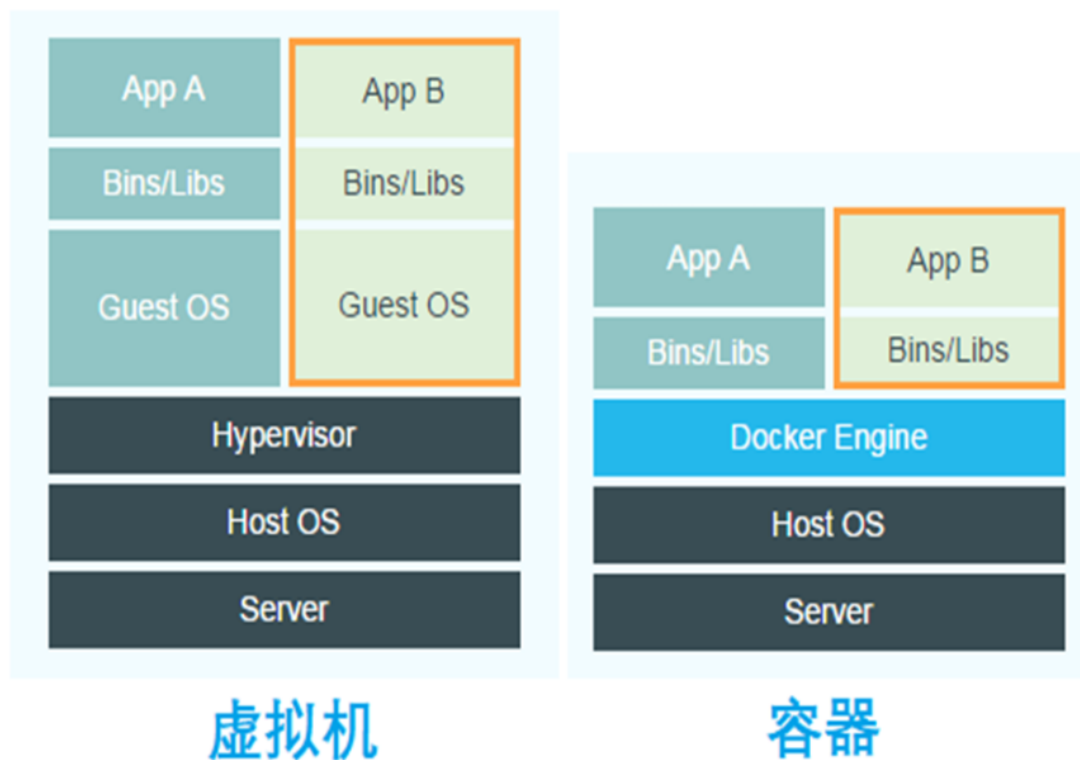
引用网图：



1. 软件交付方式发生了变化
2. 替代了虚拟机
3. 改变了我们体验软件的模式
4. 降低了企业成本
5. 促进了持续集成、持续部署的发展
6. 促进了微服务的发展

docker和虚拟机的区别

引用网图：

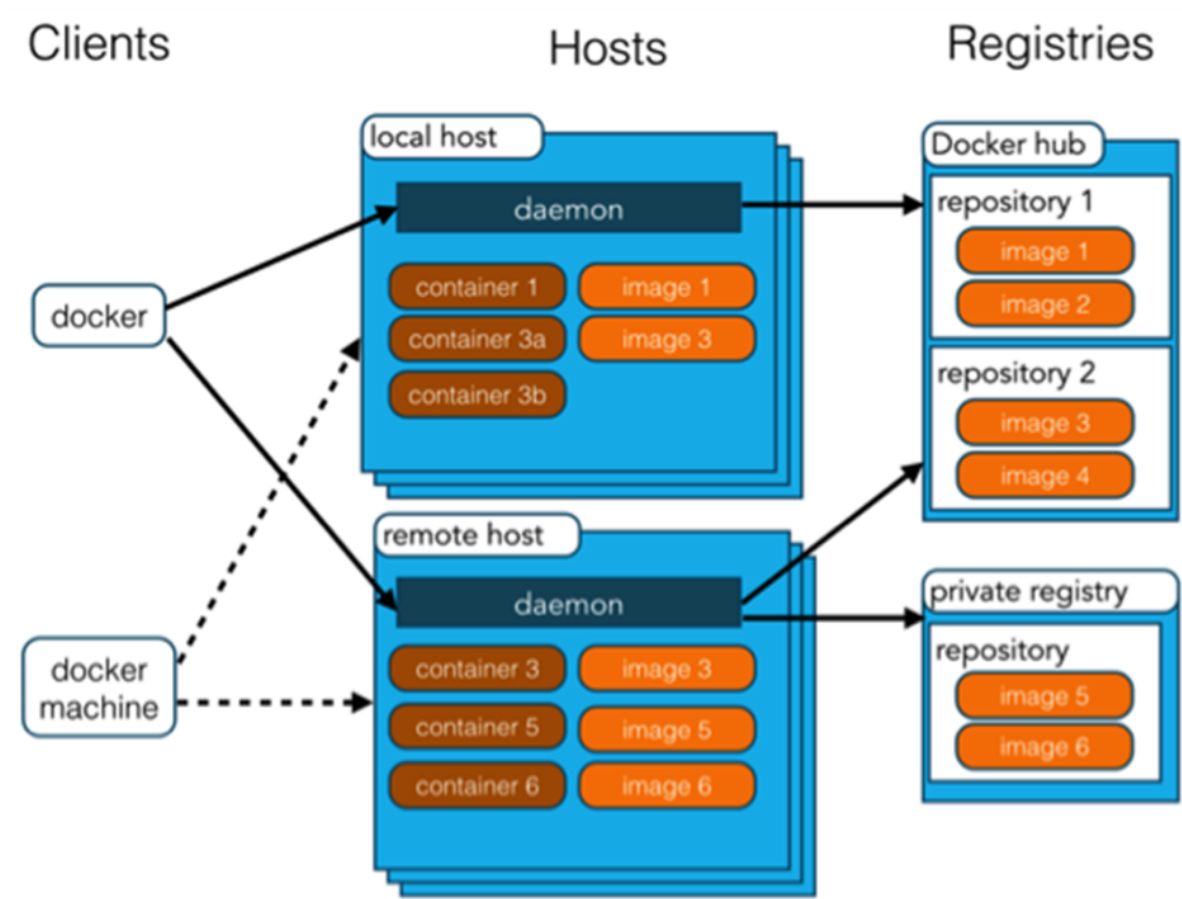


1. vm（虚拟机）与docker（容器）框架，直观上来讲vm多了一层guest OS，同时Hypervisor会对硬件资源进行虚拟化，docker直接使用硬件资源，所以资源利用率相对docker低
2. 服务器虚拟化解决的核心问题是资源调配，而容器解决的核心问题是应用开发、测试和部署。
3. 容器技术严格来说并不是虚拟化，没有客户机操作系统，是共享内核的。

docker基本架构

基本架构图

引用网图：



涉及概念

1. 镜像 (Image) : Docker 镜像是用于创建 Docker 容器的模板, 比如 Ubuntu 系统
2. 容器 (Container) : 容器是独立运行的一个或一组应用, 是镜像运行时的实体
3. 客户端 (client) : Docker 客户端通过命令行或者其他工具使用 Docker SDK (<https://docs.docker.com/develop/sdk/>) 与 Docker 的守护进程通信
4. 主机 (host) : 一个物理或者虚拟的机器用于执行 Docker 守护进程和容器
5. 注册中心 (Registry) : Docker 仓库用来保存镜像, 可以理解为代码控制中的代码仓库。Docker Hub(<https://hub.docker.com/>) 提供了庞大的镜像集合供使用。
6. Docker Machine: Docker Machine是一个简化Docker安装的命令行工具, 通过一个简单的命令行即可在相应的平台上安装Docker。

直观感受client请求server

client 通过http协议访问 host

```
sudo apt install socat
```

```
socat -v UNIX-LISTEN:/tmp/dockerapi.sock UNIX-CONNECT:/var/run/docker.sock &
```

这条命令中, -v 用于提高输出的可读性, 带有数据流的指示。UNIX-LISTEN 部分是让socat 在一个Unix套接字上进行监听, 而UNIX-CONNECT 是让socat 连接到Docker 的Unix套接字。

```
docker -H unix:///tmp/dockerapi.sock ps
```

```

nick@ubuntu20:~$ docker -H unix:///tmp/dockerapi.sock ps -a
> 2022/04/26 09:55:25.985948 length=81 from=0 to=80
HEAD /_ping HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/20.10.7 (linux)\r
\r
< 2022/04/26 09:55:25.986933 length=280 from=0 to=279
HTTP/1.1 200 OK\r
Api-Version: 1.41\r
Cache-Control: no-cache, no-store, must-revalidate\r
Content-Length: 0\r
Content-Type: text/plain; charset=utf-8\r
Docker-Experimental: false\r
Ostype: linux\r
Pragma: no-cache\r
Server: Docker/20.10.7 (linux)\r
Date: Tue, 26 Apr 2022 09:55:25 GMT\r
\r
> 2022/04/26 09:55:25.988577 length=102 from=81 to=182
GET /v1.41/containers/json?all=1 HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/20.10.7 (linux)\r
\r
< 2022/04/26 09:55:25.993967 length=8192 from=280 to=8471
HTTP/1.1 200 OK\r
Api-Version: 1.41\r
Content-Type: application/json\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/20.10.7 (linux)\r
Date: Tue, 26 Apr 2022 09:55:25 GMT\r
Transfer-Encoding: chunked\r
\r
65cb\r
[{"Id": "23a69b4b4dd893e8708f407689db22053c5991a975f4bb04c5d12b71870461a2", "Names": ["/brave_yalow"], "c7d0edeal8d", "Command": "/bin/false", "Created": 1650965183, "Ports": [], "Labels": {}, "State": "exited", "S

```

docker隔离机制

1. 在容器进程启动之前重新挂载它的整个根目录“/”，用来为容器提供隔离后的执行环境文件系统（rootfs）。
2. 通过Linux Namespace 创建隔离，决定进程能够看到和使用哪些东西。
3. 通过control groups 技术来约束进程对资源的使用

rootfs

rootfs 是Docker 容器在启动时内部进程可见的文件系统，即Docker容器的根目录。rootfs通常包含一个操作系统运行所需的文件系统，例如可能包含经典的类Unix操作系统中的目录系统，

如/dev、/proc、/bin、/etc、/lib、/usr、/tmp及运行Docker容器所需的配置文件、工具等。

在传统的linux操作系统内核启动时，首先挂载一个只读（read-only）的rootfs，当系统检测器完整性之后，再将其切换为读写（read-write）模式。而在Docker架构中，当Docker daemon 为Docker容器挂载rootfs时，沿用的Linux内核启动时的方法，即将rootfs设为只读模式。在挂载完毕之后，**利用联合挂载（union mount）技术**在已有的只读rootfs上再挂载一个读写层。这样，可读写层处于Docker容器文件系统的最顶层，其下可能联合挂载了多个只读层，只有在Docker容器运行过程中文件系统发生变化是，才会把变化的文件内容写到可读写层，并且隐藏只读层中的老版本文件。

rootfs只是一个操作系统所包含的文件、配置和目录，并不包括操作系统内核。在Linux操作系统中，这两部分是分开存放的，操作系统只有在开机启动时才会加载指定版本的内核镜像。正是由于rootfs的存在，容器才有了一个被反复宣传至今的重要特性：一致性。由于rootfs里打包的不只是应用，而是整个操作系统的文件和目录，也就意味着，应用以及它运行所需要的所有依赖，都被封装在了一起。

有了容器镜像“打包操作系统”的能力，这个最基础的依赖环境也终于变成了应用沙盒的一部分。这就赋予了容器所谓的一致性：无论在本地、云端，还是在一台任何地方的机器上，用户只需要解压打包好的容器镜像，那么这个应用运行所需要的完整的执行环境就被重现出来了

Linux Namespace

namespace 是什么

1. **namespace 是 Linux 内核用来隔离内核资源的方式。**通过 namespace 可以让一些进程只能看到与自己相关的一部分资源，而另外一些进程也只能看到与它们自己相关的资源，这两拨进程根本就感觉不到对方的存在。具体的实现方式是把一个或多个进程的相关资源指定在同一个 namespace 中。
2. Linux namespaces 是对全局系统资源的一种封装隔离，使得处于不同 namespace 的进程拥有独立的全局系统资源，改变一个 namespace 中的系统资源只会影响当前 namespace 里的进程，对其他 namespace 中的进程没有影响。

namespace 解决了什么问题

1. Linux 内核实现 namespace 的一个主要目的就是实现轻量级虚拟化(容器)服务。在同一个 namespace 下的进程可以感知彼此的变化，而对外界的进程一无所知。这样就可以让容器中的进程产生错觉，认为自己置身于一个独立的系统中，从而达到隔离的目的。也就是说 linux 内核提供的 namespace 技术为 docker 等容器技术的出现和发展提供了基础条件。

namespace 有哪些

1. **Mount Namespace隔离了一组进程所看到的文件系统挂载点的集合**，因此，在不同Mount Namespace的进程看到的文件系统层次结构也不同
2. **UTS Namespace隔离了uname()系统调用返回的两个系统标示符nodename和domainname**，在容器的上下文中，UTS Namespace允许每个容器拥有自己的hostname和NIS domain name，这对于初始化和配置脚本是很有用的，这些脚本根据这些名称来定制它们的操作
3. **IPC Namespace隔离了某些IPC资源（interprocess community，进程间通信）使划分到不同IPC Namespace的进程组通信上隔离，无法通过消息队列、共享内存、信号量方式通信**
4. **PID Namespace隔离了进程ID号空间**，不同的PID Namespace中的进程可以拥有相同的PID。PID Namespace的好处之一是，容器可以在主机之间迁移，同时容器内的进程保持相同的进程ID。PID命名空间还允许每个容器拥有自己的init（PID 1），它是 "所有进程的祖先"，负责管理各种系统初始化任务，并在子进程终止时收割孤儿进程
5. **Network Namespace提供了网络相关系统资源的隔离**，因此，每个Network Namespace都有自己的网络设备、IP地址、IP路由表、/proc/net目录、端口号等。
6. **User Namespace隔离了用户和组ID号空间**，一个进程的用户和组ID在用户命名空间内外可以是不同的，一个进程可以在用户命名空间外拥有一个正常的无权限用户ID，同时在命名空间内拥有一个（root权限）的用户ID

```
ls /proc/{PD}/ns
```

```
$ docker run -it busybox /bin/sh
/ #
```

Cgroup

什么是Cgroup

1. cgroup全称是control groups
2. control groups：控制组，被整合在了linux内核当中，把进程（tasks）放到组里面，对组设置权限，对进程进行控制。可以理解为用户和组的概念，用户会继承它所在组的权限。

3. cgroups是linux内核中的机制，这种机制可以根据特定的行为把一系列的任务，子任务整合或者分离，按照资源划分的等级的不同，从而实现资源统一控制的框架，cgroup可以控制、限制、隔离进程所需要的物理资源，包括cpu、内存、IO，为容器虚拟化提供了最基本的保证，是构建docker一系列虚拟化的管理工具

Cgroup 解决了什么问题

1. 资源控制：cgroup通过进程组对资源总额进行限制。如：程序使用内存时，要为程序设定可以使用主机的多少内存，也叫作限额
2. 优先级分配：使用硬件的权重值。当两个程序都需要进程读取cpu，哪个先哪个后，通过优先级来进行控制
3. 资源统计：可以统计硬件资源的用量，如：cpu、内存...使用了多长时间
4. 进程控制：可以对进程组实现挂起/恢复的操作，

cgroup子系统有哪些

1. cpu子系统：该子系统为每个进程组设置一个使用CPU的权重值，以此来管理进程对cpu的访问。
2. cpuset子系统：对于多核cpu，该子系统可以设置进程组只能在指定的核上运行，并且还可以设置进程组在指定的内存节点上申请内存。
3. cpuacct子系统：该子系统只用于生成当前进程组内的进程对cpu的使用报告
4. memory子系统：该子系统提供了以页面为单位对内存的访问，比如对进程组设置内存使用上限等，同时可以生成内存资源报告
5. blkio子系统：该子系统用于限制每个块设备的输入输出。首先，与CPU子系统类似，该系统通过为每个进程组设置权重来控制块设备对其的I/O时间；其次，该子系统也可以限制进程组的I/O带宽以及IOPS
6. devices子系统：通过该子系统可以限制进程组对设备的访问，即该允许或禁止进程组对某设备的访问
7. freezer子系统：该子系统可以使得进程组中的所有进程挂起
8. net_cls子系统：该子系统提供对网络带宽的访问限制，比如对发送带宽和接收带宽进程限制

```
mount -t cgroup
```

运行一个docker容器，查看容器的cgroup 和 linux namespace

```
docker pull nginx
```

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

```
cd /sys/fs/cgroup //查看cgroup 子系统变化 并找到其对应的PID  
cd /proc/{PID}/ns //查看命名空间
```

docker 常用命令

docker 命令官方文档

<https://docs.docker.com/reference/>

Docker 环境信息命令

docker info

`docker info` : 显示 Docker 系统信息，包括镜像和容器数。

`docker info [OPTIONS]`

docker version

`docker version` : 显示 Docker 版本信息。

`docker version [OPTIONS]`

OPTIONS说明:

`-f` : 指定返回值的模板文件。

系统日志信息常用命令

docker events

`docker events` : 从服务器获取实时事件

`docker events [OPTIONS]`

OPTIONS说明:

`-f` : 根据条件过滤事件;

`--since` : 从指定的时间戳后显示所有事件;

`--until` : 流水时间显示到指定的时间为止;

//第一个终端执行

`docker events`

//第二个终端操作容器

`docker start/stop/restart`

//查看第一个终端输出

docker logs

`docker logs` : 获取容器的日志

```
docker logs [OPTIONS] CONTAINER
```

OPTIONS说明:

-f : 跟踪日志输出

--since :显示某个开始时间的所有日志

-t : 显示时间戳

--tail :仅列出最新N条容器日志

```
docker logs -f mynginx
```

docker history

docker history : 查看指定镜像的创建历史。

```
docker history [OPTIONS] IMAGE
```

OPTIONS说明:

-H :以可读的格式打印镜像大小和日期，默认为**true**;

--no-trunc :显示完整的提交记录;

-q :仅列出提交记录ID

```
docker history mynginx:v1
```

容器的生命周期管理命令

docker create

docker create : 创建一个新的容器但不启动它

语法同docker run

docker run

docker run :创建一个新的容器并运行一个命令

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

OPTIONS说明:

-a stdin: 指定标准输入输出内容类型，可选 **STDIN/STDOUT/STDERR** 三项;

-d: 后台运行容器，并返回容器ID;

-i: 以交互模式运行容器，通常与 **-t** 同时使用；

-P: 随机端口映射，容器内部端口随机映射到主机的端口

-p: 指定端口映射，格式为：主机(宿主)端口:容器端口

-t: 为容器重新分配一个伪输入终端，通常与 **-i** 同时使用；

--name="nginx-lb": 为容器指定一个名称；

--dns 8.8.8.8: 指定容器使用的DNS服务器，默认和宿主一致；

--dns-search example.com: 指定容器DNS搜索域名，默认和宿主一致；

-h "mars": 指定容器的hostname；

-e username="ritchie": 设置环境变量；

--env-file=[]: 从指定文件读入环境变量；

--cpuset="0-2" or --cpuset="0,1,2": 绑定容器到指定CPU运行；

-m :设置容器使用内存最大值；

--net="bridge": 指定容器的网络连接类型，支持 **bridge/host/none/container**: 四种类型；

--link=[]: 添加链接到另一个容器；

--expose=[]: 开放一个端口或一组端口；

--volume , -v: 绑定一个卷

--restart 指定容器重启条件，默认为no

restart 参数选项：

1. no:不重启；
2. on-failure[:max-retries] : 失败时重启，可以指定重试次数；
3. always : 总是重启；
4. unless-stopped : docker守护进程停止前，如果容器是停止状态除外，其他情况总是重启
5. 可以通过docker attach 来观察到always选项的容器重启

```
docker run -p 81:80 -d --name=mynginx nginx
```

docker start/stop/restart

docker start :启动一个或多个已经被停止的容器

docker stop :停止一个运行中的容器

docker restart :重启容器

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

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

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

```
docker restart mynginx
docker stop mynginx
docker start mynginx
```

docker kill

`docker kill` :杀掉一个运行中的容器

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

OPTIONS说明:

`-s` :向容器发送一个信号

```
docker kill -s KILL mynginx
docker kill -s TERM mynginx
```

如果容器终止时的状态对用户而言很重要，用户可能会想要了解docker kill 和docker stop之间的区别。docker kill 的行为和标准的kill 命令行程序并不相同。kill 程序的默认工作方式是向指定的进程发送TERM信号（即信号值为15）。这个信号表示程序应该被终止，但是不要强迫程序终止。当这个信号被处理时，大多数程序将执行某种清理工作，但是该程序也可以执行其他操作，包括忽略该信号。而docker kill 对正在运行的程序使用的是KILL信号，这使得该进程没办法处理终止过程。这就意味着一些诸如包含运行进程ID之类的文件可能会残留在文件系统中。根据应该用程序管理状态的能力，如果再次启动容器，这可能会也可能不会造成问题。

docker stop 命令则像kill 命令那样工作，发送的是一个TERM型号。

命令	默认信号量	默认信号量的值
kill	TERM	15
docker kill	KILL	9
docker stop	TERM	15

docker rm

`docker rm` : 删除一个或多个容器。

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

OPTIONS说明:

-f :通过 SIGKILL 信号强制删除一个运行中的容器。

-l :移除容器间的网络连接，而非容器本身。

-v :删除与容器关联的卷。

docker pause/unpause

docker pause :暂停容器中所有的进程。

docker unpause :恢复容器中所有的进程。

```
docker pause CONTAINER [CONTAINER...]
```

```
docker unpause CONTAINER [CONTAINER...]
```

docker exec

docker exec : 在运行的容器中执行命令

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

OPTIONS说明:

-d :分离模式: 在后台运行

-i :即使没有附加也保持STDIN 打开

-t :分配一个伪终端

```
docker exec -i -t mynginx /bin/bash
```

// 通过exec 执行命令

```
docker exec mynginx ls
```

容器运维操作命令

docker ps

docker ps : 列出容器

```
docker ps [OPTIONS]
```

OPTIONS说明:

-a :显示所有的容器，包括未运行的。

-f :根据条件过滤显示的内容。

`--format` :指定返回值的模板文件。

`-l` :显示最近创建的容器。

`-n` :列出最近创建的n个容器。

`--no-trunc` :不截断输出。

`-q` :静默模式，只显示容器编号。

`-s` :显示总的文件大小。

`docker ps`

docker inspect

`docker inspect` : 获取容器/镜像的元数据。

`docker inspect [OPTIONS] NAME|ID [NAME|ID...]`

OPTIONS说明:

`-f` :指定返回值的模板文件。

`-s` :显示总的文件大小。

`--type` :为指定类型返回JSON

```
// 查看容器的示例id
docker inspect -f '{{.Id}}' <id>
// 检查镜像或者容器的参数，默认返回 JSON 格式
docker inspect <id>
```

docker top

`docker top` :查看容器中运行的进程信息，支持 `ps` 命令参数

`docker top [OPTIONS] CONTAINER [ps OPTIONS]`

```
# 查看所有运行容器的进程信息。
for i in `docker ps |grep Up|awk '{print $1}'`;do echo \ &&docker top $i; done
```

docker attach

`docker attach` :连接到正在运行中的容器

`docker attach [OPTIONS] CONTAINER`

```
docker attach mynginx
```

退出attach模式，默认配置 先按 Ctrl + P 然后再按 Ctrl + Q

docker exec 与 docker attach 的区别：

1. exec 在容器中执行命令，并且可以通过-i -t 创建虚拟终端的方式与容器交互
2. attach 进入容器某个正在执行的命令终端，不能交互操作。但是如果该容器的命令终端是一个可以交互的终端，那么也可以交互

docker wait

`docker wait` : 阻塞运行直到容器停止，然后打印出它的退出代码。

```
docker wait [OPTIONS] CONTAINER [CONTAINER...]
```

docker export

`docker export` : 将文件系统作为一个tar归档文件导出到STDOUT。

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

OPTIONS说明：

`-o` : 将输入内容写到文件。

```
docker export mynginx -o mynginx.tar
```

解压归档文件，查看内容

```
tar vxf mynginx.tar -c mynginx
```

docker port

`docker port` : 列出指定的容器的端口映射，或者查找将PRIVATE_PORT NAT到面向公众的端口。

```
docker port [OPTIONS] CONTAINER [PRIVATE_PORT[/PROTO]]
```

```
docker port mynginx
```

docker cp

`docker cp` : 用于容器与主机之间的数据拷贝


```
docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-
```

```
docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH
```

OPTIONS说明:

-L :保持源目标中的链接

主机/www/html目录拷贝到容器1881f0cba5bc的/www目录下

```
docker cp /www/html 1881f0cba5bc:/www/
```

将主机/www/html目录拷贝到容器1881f0cba5bc中，目录重命名为www。

```
docker cp /www/html 1881f0cba5bc:/www
```

将容器1881f0cba5bc的/www目录拷贝到主机的/tmp目录中

```
docker cp 1881f0cba5bc:/www /tmp/
```

docker diff

docker diff : 检查容器里文件结构的更改。

docker diff命令会列出 3 种容器内文件状态变化 (A - Add, D - Delete, C - Change) 的列表清单

```
docker diff [OPTIONS] CONTAINER
```

```
docker diff mynginx
```

docker rename

```
docker rename CONTAINER NEW_NAME
```

docker stats

```
docker stats [容器]
```

docker update

查看帮助，可以看出docker update 是用来修改docker run 指定的运行参数

```
docker update -h
```

```
docker update --restart always myubuntu3
```

```
docker update --restart no myubuntu3
```

镜像管理命令

docker build

`docker build` 命令用于使用 `Dockerfile` 创建镜像。

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

`--build-arg=[]` :设置镜像创建时的变量;

`--cpu-shares` :设置 `cpu` 使用权重;

`--cpu-period` :限制 `CPU CFS`周期;

`--cpu-quota` :限制 `CPU CFS`配额;

`--cpuset-cpus` :指定使用的`CPU id`;

`--cpuset-mems` :指定使用的内存 `id`;

`--disable-content-trust` :忽略校验, 默认开启;

`-f` :指定要使用的`Dockerfile`路径;

`--force-rm` :设置镜像过程中删除中间容器;

`--isolation` :使用容器隔离技术;

`--label=[]` :设置镜像使用的元数据;

`-m` :设置内存最大值;

`--memory-swap` :设置`Swap`的最大值为内存+`swap`, `"-1"`表示不限`swap`;

`--no-cache` :创建镜像的过程不使用缓存;

`--pull` :尝试去更新镜像的新版本;

`--quiet, -q` :安静模式, 成功后只输出镜像 `ID`;

`--rm` :设置镜像成功后删除中间容器;

`--shm-size` :设置`/dev/shm`的大小, 默认值是`64M`;

`--ulimit` :`Ulimit`配置。

`--squash` :将 `Dockerfile` 中所有的操作压缩为一层。

`--tag, -t`: 镜像的名字及标签, 通常 `name:tag` 或者 `name` 格式; 可以在一次构建中为一个镜像设置多个标签。

`--network`: 默认 `default`。在构建期间设置`RUN`指令的网络模式

`docker build -t nodetodo:v1.0.0 -f Dockerfile .`

docker images

`docker images` : 列出本地镜像。

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

OPTIONS说明:

`-a` : 列出本地所有的镜像 (含中间映像层, 默认情况下, 过滤掉中间映像层);

`--digests` : 显示镜像的摘要信息;

`-f` : 显示满足条件的镜像;

`--format` : 指定返回值的模板文件;

`--no-trunc` : 显示完整的镜像信息;

`-q` : 只显示镜像ID

查看所有悬空镜像

```
docker images --filter dangling=true
```

清除所有悬空镜像

```
docker image prune
```

docker rmi

`docker rmi` : 删除本地一个或多个镜像。

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

OPTIONS说明:

`-f` : 强制删除;

`--no-prune` : 不移除该镜像的过程镜像, 默认移除;

docker tag

`docker tag` : 标记本地镜像, 将其归入某一仓库。

```
docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/] [USERNAME/]NAME[:TAG]
```

```
docker tag mynginx:v1 mynginx:1.0
```

docker save

`docker save` : 将指定镜像保存成 `tar` 归档文件。

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

`-o` : 输出到的文件。

```
docker save -o mynginx.tar mynginx:v1
```

```
# 解压文件, 查看内容
tar vxf mynginx.tar -C mynginx
```

docker load

`docker load` : 导入使用 `docker save` 命令导出的镜像。

`docker load [OPTIONS]`
`OPTIONS` 说明:

`--input` , `-i` : 指定导入的文件, 代替 `STDIN`。

`--quiet` , `-q` : 精简输出信息。

```
docker load -i mynginx.tar
```

docker import

`docker import` : 从归档文件中创建镜像。

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

`-c` : 应用`docker` 指令创建镜像;

`-m` : 提交时的说明文字

```
docker import mynginx.tar mynginx:v3
```

docker commit

`docker commit` : 从容器创建一个新的镜像。

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

OPTIONS说明:

-a :提交的镜像作者;

-c :使用Dockerfile指令来创建镜像;

-m :提交时的说明文字;

-p :在commit时, 将容器暂停

```
docker commit -a "nick" -m "mynginx commit" mynginx myngintest:1.0
```

镜像仓库命令

docker login/logout

docker login : 登陆到一个Docker镜像仓库, 如果未指定镜像仓库地址, 默认为官方仓库 Docker Hub

docker logout : 登出一个Docker镜像仓库, 如果未指定镜像仓库地址, 默认为官方仓库 Docker Hub

```
docker login [OPTIONS] [SERVER]
```

```
docker logout [OPTIONS] [SERVER]
```

OPTIONS说明:

-u :登陆的用户名

-p :登陆的密码

```
docker login -u 用户名 -p 密码
```

```
docker logout
```

docker pull

docker pull : 从镜像仓库中拉取或者更新指定镜像

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

OPTIONS说明:

-a :拉取所有 tagged 镜像

--disable-content-trust :忽略镜像的校验, 默认开启

```
docker pull nginx
```

docker push

`docker push` : 将本地的镜像上传到镜像仓库,要先登录到镜像仓库

```
docker push [OPTIONS] NAME[:TAG]
```

OPTIONS说明:

`--disable-content-trust` :忽略镜像的校验,默认开启

docker search

`docker search` : 从Docker Hub查找镜像

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

OPTIONS说明:

`--automated` :只列出 `automated build`类型的镜像;

`--no-trunc` :显示完整的镜像描述;

`-f <过滤条件>`:列出收藏数不小于指定值的镜像

//从 Docker Hub 查找所有镜像名包含 `nginx`, 并且收藏数大于 `10` 的镜像

```
docker search -f stars=10 nginx
```

参数说明:

NAME: 镜像仓库源的名称

DESCRIPTION: 镜像的描述

OFFICIAL: 是否 docker 官方发布

stars: 类似 Github 里面的 star, 表示点赞、喜欢的意思。

AUTOMATED: 自动构建。

dockerfile

dockerfile是什么

Dockerfile是一个创建镜像所有命令的文本文件, 包含了一条条指令和说明, 每条指令构建一层, 通过 `docker build`命令,根据Dockerfile的内容构建镜像,因此每一条指令的内容, 就是描述该层如何构建.有了 Dockerfile, 就可以制定自己的docker镜像规则,只需要在Dockerfile上添加或者修改指令, 就可生成 docker 镜像.

dockerfile解决了什么问题

Dockerfile 包含了镜像制作的完整操作流程, 其他开发者可以通过 Dockerfile 了解并复现制作过程 Dockerfile 中的每一条指令都会创建新的镜像层, 这些镜像可以被 Docker Daemon 缓存。再次制作镜像时, Docker 会尽量复用缓存的镜像层 (using cache), 而不是重新逐层构建, 这样可以节省时间和磁盘空间

Dockerfile 的操作流程可以通过`docker image history [镜像名称]`查询, 方便开发者查看变更记录

docker build 构建流程

docker build命令会读取Dockerfile的内容，并将Dockerfile的内容发送给 Docker 引擎，最终 Docker 引擎会解析Dockerfile中的每一条指令，构建出需要的镜像。

第一步，docker build会将 context 中的文件打包传给 Docker daemon。如果 context 中有.dockerignore文件，则会从上传列表中删除满足.dockerignore规则的文件。**注意：如果上下文中有相当多的文件，可以明显感受到整个文件发送过程**

这里有个例外，如果.dockerignore文件中有.dockerignore或者Dockerfile，docker build命令在排除文件时会忽略掉这两个文件。如果指定了镜像的 tag，还会对 repository 和 tag 进行验证。

第二步，docker build命令向 Docker server 发送 HTTP 请求，请求 Docker server 构建镜像，请求中包含了需要的 context 信息。

第三步，Docker server 接收到构建请求之后，会执行以下流程来构建镜像：

1. 创建一个临时目录，并将 context 中的文件解压到该目录下。
2. 读取并解析 Dockerfile，遍历其中的指令，根据命令类型分发到不同的模块去执行。
3. Docker 构建引擎为每一条指令创建一个临时容器，在临时容器中执行指令，然后 commit 容器，生成一个新的镜像层。
4. 最后，将所有指令构建出的镜像层合并，形成 build 的最后结果。最后一次 commit 生成的镜像 ID 就是最终的镜像 ID。

为了提高构建效率，docker build默认会缓存已有的镜像层。如果构建镜像时发现某个镜像层已经被缓存，就会直接使用该缓存镜像，而不用重新构建。如果不希望使用缓存的镜像，可以在执行docker build命令时，指定--no-cache=true参数。

Docker 匹配缓存镜像的规则为：遍历缓存中的基础镜像及其子镜像，检查这些镜像的构建指令是否和当前指令完全一致，如果不一样，则说明缓存不匹配。对于ADD、COPY指令，还会根据文件的校验和（checksum）来判断添加到镜像中的文件是否相同，如果不相同，则说明缓存不匹配。

这里要注意，缓存匹配检查不会检查容器中的文件。比如，当使用RUN apt-get -y update命令更新了容器中的文件时，缓存策略并不会检查这些文件，来判断缓存是否匹配。

最后，可以通过docker history命令来查看镜像的构建历史

关键字

FROM 设置镜像使用的基础镜像
MAINTAINER 设置镜像的作者
RUN 编译镜像时运行的脚步
CMD 设置容器的启动命令
LABEL 设置镜像标签
EXPOSE 设置镜像暴露的端口
ENV 设置容器的环境变量
ADD 编译镜像时复制上下文中文件到镜像中
COPY 编译镜像时复制上下文中文件到镜像中
ENTRYPOINT 设置容器的入口程序
VOLUME 设置容器的挂载卷
USER 设置运行 RUN CMD ENTRYPOINT的用户名
WORKDIR 设置 RUN CMD ENTRYPOINT COPY ADD 指令的工作目录
ARG 设置编译镜像时加入的参数
ONBUILD 设置镜像的ONBUILD 指令
STOPSIGNAL 设置容器的退出信号量

dockerfile 实践

基本语法实践

```
mkdir example1
cd example1
```

```
# syntax=docker/dockerfile:1
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
# 仅指定镜像元数据内容
LABEL hello 1.0.0
RUN git clone https://gitee.com/nickdemo/helloworld.git
WORKDIR helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

```
docker build -t hello:1.0.0 -f Dockerfile .
```

```
docker run -p 80:80 -d --name hello hello:1.0.0
```

docker build上下文

1. 素材准备

```
# 创建一个目录，案例需要
mkdir example2
cd example2
# 下载nginx源码包，作为案例素材
curl https://nginx.org/download/nginx-1.21.6.tar.gz > ./nginx-1.21.6.tar.gz
# 下载app代码
git clone https://gitee.com/nickdemo/helloworld
```

```
# ./nginx*
# helloworld
```

2. 没有什么作用的上下文

```
# syntax=docker/dockerfile:1
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
# 仅指定镜像元数据内容
LABEL hello 1.0.0
RUN git clone https://gitee.com/nickdemo/helloworld.git
WORKDIR helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```



```
# 调整为不同的上下文，查看不同的效果
docker build -t hello:1.0.0 -f Dockerfile .
```

3. 上下文的真正作用

```
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
COPY ./helloworld /go/src/helloworld
WORKDIR /go/src/helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

```
docker build -t hello:1.0.0 -f Dockerfile .
```

操作：

1. 尝试在.dockerignore 文件中忽略不同的内容，观察发送到守护进程的数据
2. 尝试 采用不同的上下文路径来构建镜像
3. 尝试 copy上下文以外的内容来构建镜像

多阶段构建以及ADD与COPY

多阶段构建

Docker 17.05版本以后，新增了Dockerfile多阶段构建。所谓多阶段构建，实际上是允许一个Dockerfile中出现多个 FROM 指令。这样做有什么意义呢？

多个 FROM 指令的意义

多个 FROM 指令并不是为了生成多根的层关系，最后生成的镜像，仍以最后一条 FROM 为准，之前的 FROM 会被抛弃，那么之前的FROM 又有什么意义呢？

每一条 FROM 指令都是一个构建阶段，多条 FROM 就是多阶段构建，虽然最后生成的镜像只能是最后一个阶段的结果，但是，能够将前置阶段中的文件拷贝到后边的阶段中，这就是多阶段构建的最大意义。

最大的使用场景是将编译环境和运行环境分离，比如，之前我们需要构建一个Go语言程序，那么就需要用到go命令等编译环境

1. 素材准备

```
# 创建一个目录，案例需要
mkdir example3
cd example3
# 下载nginx源码包，作为案例素材
curl https://nginx.org/download/nginx-1.21.6.tar.gz > ./nginx-1.21.6.tar.gz
# 下载app代码
git clone https://gitee.com/nickdemo/helloworld
```

2. 一个最基本的dockerfile

```
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
COPY ./helloworld /go/src/helloworld
WORKDIR /go/src/helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

3. 多阶段构建dockerfile

```
FROM golang:1.18
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

4. 通过as关键词，为构建阶段指定别名，可以提高可读性

```
FROM golang:1.18 as stage0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

ADD 与 COPY

1. ADD 与 COPY 不能拷贝**上下文**以外的文件
2. COPY 命令语法格式

```
COPY <src> <dest> //将上下文中源文件，拷贝到目标文件
```

```
COPY prefix* /destDir/ //将所有prefix 开头的文件拷贝到 destDir 目录下
```

```
COPY prefix?.log /destDir/ //支持单个占位符，例如： prefix1.log、prefix2.log 等
```

3. 对于目录而言，COPY 和 ADD 命令具有相同的特点：**只复制目录中的内容而不包含目录自身**

```
COPY srcDir /destDir/ //只会将源文件夹srcDir下的文件拷贝到 destDir 目录下
```

4. COPY 区别于ADD在于Dockerfile中使用multi-stage

```
FROM golang:1.18 as stage0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

5. ADD 命令语法

```
ADD <src> <dest>
```

ADD 命令除了不能用在 multistage 的场景下，ADD 命令可以完成 COPY 命令的所有功能，并且还可以完成两类的功能：

- 解压压缩文件并把它们添加到镜像中，对于宿主机本地压缩文件，ADD命令会自动解压并添加到镜像
- 从 url 拷贝文件到镜像中，需要注意：url 所在文件如果是压缩包，ADD 命令不会自动解压缩

6. ADD 案例

```
# 下载nginx tar.gz包作为素材
curl https://nginx.org/download/nginx-1.21.6.tar.gz > ./nginx-1.21.6.tar.gz
```

```
FROM golang:1.18 as stage0
# ADD ./helloworld /go/src/helloworld/
COPY ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
```

```
MAINTAINER nick
LABEL hello 1.0.0
ADD https://nginx.org/download/nginx-1.21.6.tar.gz /soft/
COPY nginx-1.21.6.tar.gz /soft/copy/
ADD nginx-1.21.6.tar.gz /soft/add/
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["/app", "--param1=p1", "--param2=p2"]
```

```
docker build -t hello:1.0.0 -f Dockerfile .
```

```
# 运行容器
docker run -d --name hello hello:1.0.0
# 查看add和copy的文件
docker exec -it hello /bin/sh
```

注意：目标文件位置要注意路径后面是否带 "/"，带斜杠表示目录，不带斜杠表示文件名
文件名里带有空格，需要再 ADD（或COPY）指令里用双引号的形式标明：

```
ADD "space file.txt" "/tmp/space file.txt"
```

CMD 与 ENTRYPOINT

CMD

CMD 指令有三种格式

```
# shell 格式
CMD <command>
# exec格式，推荐格式
CMD ["executable","param1","param2"]
# 为ENTRYPOINT 指令提供参数
CMD ["param1","param2"]
```

CMD 指令提供容器运行时的默认值，这些默认值可以是一条指令，也可以是一些参数。一个dockerfile中可以有多条CMD指令，但只有最后一条CMD指令有效。**CMD参数格式**是在CMD指令与ENTRYPOINT指令配合时使用，CMD指令中的参数会添加到ENTRYPOINT指令中。使用**shell 和exec 格式时**，命令在容器中的运行方式与RUN 指令相同。不同在于，RUN指令在构建镜像时执行命令，并生成新的镜像。CMD指令在构建镜像时并不执行任何命令，而是在容器启动时默认将CMD指令作为第一条执行的命令。如果在命令行界面运行docker run 命令时指定命令参数，则会覆盖CMD指令中的命令。

```
# syntax=docker/dockerfile:1
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
RUN git clone https://gitee.com/nickdemo/helloworld.git
WORKDIR helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
# CMD [./app --param1=p1 --param2=p]
CMD ["/app", "--param1=p1", "--param2=p2"]
```

ENTRYPOINT

ENTRYPOINT指令有两种格式

```
# shell 格式
ENTRYPOINT <command>
# exec 格式，推荐格式
ENTRYPOINT ["executable", "param1", "param2"]
```

ENTRYPOINT指令和CMD指令类似，都可以让容器每次启动时执行相同的命令，但它们之间又有不同。一个Dockerfile中可以有多条ENTRYPOINT指令，但只有最后一条ENTRYPOINT指令有效。当使用shell格式时，ENTRYPOINT指令会忽略任何CMD指令和docker run 命令的参数，并且会运行在bin/sh -c中。推荐使用exec格式，使用此格式，docker run 传入的命令参数将会覆盖CMD指令的内容并且附加到ENTRYPOINT指令的参数中。

CMD可以是参数，也可以是指令，ENTRYPOINT只能是命令；docker run 命令提供的运行命令参数可以覆盖CMD，但不能覆盖ENTRYPOINT。

```
# syntax=docker/dockerfile:1
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
RUN git clone https://gitee.com/nickdemo/helloworld.git
WORKDIR helloworld
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
#ENTRYPOINT ["/app"]
#ENTRYPOINT ./app --param1=p11.0.3 --param2=p2
ENTRYPOINT ["/app", "--param1=p1", "--param2=p2"]
#CMD ["/app", "--param1=p1", "--param2=p2"]
#CMD ./app --param1=p1 --param2=p2
CMD ["--param1=p1", "--param2=p2"]
```

Docker 镜像

什么是Docker镜像

Docker镜像是一个只读的Docker容器模板，含有启动Docker容器所需的文件系统及其内容，因此是启动一个Docker容器的基础。Docker镜像的内容以及一些Docker容器的配置文件组成了Docker容器的静态文件运行环境——rootfs。可以这么理解，Docker镜像是Docker容器的静态视角，Docker容器是Docker镜像的运行状态。

Docker镜像的主要特点

1. **分层**，Docker镜像是采用分层的方式构建的，每个镜像都有一系列的“镜像层”组成。分层结构是Docker镜像如此轻量的重要原因，当需要修改容器镜像内的某个文件时，只对处于最上方的读写层进行变动，不覆写下层已有文件系统的内容，已有文件在只读层中的原始版本仍然存在，但会被读写层中的新版文件所隐藏。**当使用docker commit提交这个修改过的容器文件系统为一个新的镜像时，保存的内容仅为最上层读写文件系统中被更新过的文件。分层达到了在不同镜像之间共享镜像层的效果。**
2. **写时复制**，Docker镜像使用了写时复制策略，**在多个容器之间共享镜像，每个容器在启动的时候并不需要单独复制一份镜像文件，而是将所有镜像层以只读的方式挂载到一个挂载点，再在上面覆盖一个可读可写的容器层。**在未更改文件内容是，所有容器共享同一份数据，只有在Docker容器运行过程中文件系统发生变化是，才会把变化的文件内容写到可读写层，并隐藏只读层中的老版本文件。写时复制配合分层机制建好了镜像对磁盘空间的占用和容器启动时间。
3. **内容寻址**，Docker 1.10版本后，Docker镜像映入了内容寻址存储的机制，根据文件内容来索引镜像和镜像层。与之前版本对每一个镜像层随机生成一个UUID不同，新模型对镜像层的内容**计算校验和**，生成一个内容哈希值，并以此哈希值代替之前的UUID作为镜像层的唯一标志。该机制提高了镜像的安全性，并在pull、push、load和save操作后检查数据的完整性。另外，基于内容哈希来索引镜像层，在一定程度上减少了ID的冲突并且增强了镜像层的共享。对于来自不同构建的镜像层，只要拥有相同的内容哈希，也能被不同的镜像共享。
4. **联合挂载**，联合挂载技术可以在一个挂载点同时挂载多个文件系统，将挂载点的原目录与被挂载内容进行整合，使得最终课件的文件系统将会包含整合之后的各个层的文件和目录。实现这种联合挂载技术的文件系统通常被称为联合文件系统（union filesystem）。

Docker镜像关键概念

registry（注册中心）

registry 用来保存Docker镜像，其中还包括镜像层次结构和关于镜像的元数据（元数据：描述数据的数据，主要描述数据属性的信息）。可以将registry简单的想象成类似于git仓库之类的实体。用户可以在自己的数据中心搭建私有的registry，也可以使用Docker官方的公用registry服务，即Docker Hub。它是由Docker公司维护的一个公共镜像仓库，供用户下载使用。Docker Hub 中有两种类型的仓库，即用户仓库（user repository）与顶层仓库（top-level repository）。用户仓库有普通的Docker Hub用户创建，顶层仓库则由Docker公司负责运维，提供官方版本镜像。理论上顶层仓库中的镜像经过Docker公司验证，被认为是架构良好且安全的。

Repository

repository 即由具有某个功能的Docker镜像的所有迭代版本构成的镜像组。registry由一系列经过命名的Repository组成，repository 通过命名规范对用户仓库和顶层仓库进行组织。用户仓库的命名由用户名和repository 名两部分组成，中间以“/”隔开，即username/repository_name的形式，repository名通常表示镜像所具有的功能，如ansible/ubuntu14.04-ansible、xlhmzch/mysql等；而顶层仓库则只包含repository名的部分，如Ubuntu。**我们通常所说的镜像名其实就是指 repository。**repository和镜像之间是什么关系呢？事实上，repository是一个镜像的集合，其中包含了多个不同版本的镜像，使用标签进行版本区分。

manifest

主要存在于registry中作为Docker镜像的元数据文件，在pull、push、save和load中作为镜像结构和基础信息的描述文件。在镜像被pull或者load到Docker宿主机时，manifest被转化为本地的镜像配置文件config。

image和layer

Docker内部的image概念是用来存储一组镜像相关的元数据信息，主要包括镜像的架构（如amd64）、镜像默认配置信息、构建镜像的容器配置信息、包含所有镜像层信息的rootfs。Docker利用rootfs中的**diff_id** 计算出内容寻址的索引（chainID）来获取layer相关信息，进而获取每一个镜像层的文件内容。layer（镜像层）是一个Docker用来管理镜像层的中间概念，镜像是由镜像层组成的，而单个镜像侧可以被多个镜像共享，所以Docker将layer和image的概念分离。Docker镜像管理中的layer主要存放镜像的diff_id、size、cache-id和parent等内容，实际的文件内容则由存储驱动来管理，并可以通过cache-id在本地索引到。

Dockerfile

Dockerfile是在通过docker build 命令构建自己的Docker镜像时需要使用到的定义文件。它允许用户使用基本的DSL语法来定义Docker镜像，每一条指令描述了构建镜像的步骤。

常用命令

镜像管理

docker images

`docker images` : 列出本地镜像。

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

OPTIONS说明:

`-a` : 列出本地所有的镜像（含中间映像层，默认情况下，过滤掉中间映像层）；

`--digests` : 显示镜像的摘要信息；

`-f` : 显示满足条件的镜像；

`--format` : 指定返回值的模板文件；

`--no-trunc` : 显示完整的镜像信息；

`-q` : 只显示镜像ID

docker rmi

`docker rmi` : 删除本地一个或多个镜像。

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

OPTIONS说明:

-f :强制删除;

--no-prune :不移除该镜像的过程镜像, 默认移除;

docker tag

docker tag : 标记本地镜像, 将其归入某一仓库。

```
docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

```
docker tag mynginx:v1 mynginx:1.0
```

docker build

docker build 命令用于使用 **Dockerfile** 创建镜像。

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

OPTIONS说明:

--build-arg=[] :设置镜像创建时的变量;

--cpu-shares :设置 cpu 使用权重;

--cpu-period :限制 CPU CFS周期;

--cpu-quota :限制 CPU CFS配额;

--cpuset-cpus :指定使用的CPU id;

--cpuset-mems :指定使用的内存 id;

--disable-content-trust :忽略校验, 默认开启;

-f :指定要使用的**Dockerfile**路径;

--force-rm :设置镜像过程中删除中间容器;

--isolation :使用容器隔离技术;

--label=[] :设置镜像使用的元数据;

-m :设置内存最大值;

--memory-swap :设置Swap的最大值为内存+swap, "-1"表示不限swap;

--no-cache :创建镜像的过程不使用缓存;

--pull :尝试去更新镜像的新版本;

`--quiet, -q` :安静模式，成功后只输出镜像 ID；

`--rm` :设置镜像成功后删除中间容器；

`--shm-size` :设置/dev/shm的大小，默认值是64M；

`--ulimit` :Ulimit配置。

`--squash` :将 Dockerfile 中所有的操作压缩为一层。

`--tag, -t`: 镜像的名字及标签，通常 `name:tag` 或者 `name` 格式；可以在一次构建中为一个镜像设置多个标签。

`--network`: 默认 `default`。在构建期间设置RUN指令的网络模式

```
docker build -t nodetodo:v1.0.0 -f Dockerfile .
```

docker save

`docker save` : 将指定镜像保存成 `tar` 归档文件。

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

`-o` :输出到的文件。

```
docker save -o mynginx.tar mynginx:v1
```

docker load

`docker load` : 导入使用 `docker save` 命令导出的镜像。

`docker load [OPTIONS]`
OPTIONS 说明:

`--input, -i` : 指定导入的文件，代替 `STDIN`。

`--quiet, -q` : 精简输出信息。

```
docker load -i mynginx.tar
```

docker import

`docker import` : 从归档文件中创建镜像。

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

OPTIONS说明:

-c :应用docker 指令创建镜像;

-m :提交时的说明文字

```
docker import mynginx.tar mynginx:v3
```

镜像仓库操作

docker login/logout

docker login : 登陆到一个Docker镜像仓库, 如果未指定镜像仓库地址, 默认为官方仓库 Docker Hub

docker logout : 登出一个Docker镜像仓库, 如果未指定镜像仓库地址, 默认为官方仓库 Docker Hub

```
docker login [OPTIONS] [SERVER]
```

```
docker logout [OPTIONS] [SERVER]
```

OPTIONS说明:

-u :登陆的用户名

-p :登陆的密码

```
docker login -u 用户名 -p 密码
```

```
docker logout
```

docker pull

docker pull : 从镜像仓库中拉取或者更新指定镜像

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

OPTIONS说明:

-a :拉取所有 tagged 镜像

--disable-content-trust :忽略镜像的校验,默认开启

```
docker pull nginx
```

docker push

docker push : 将本地的镜像上传到镜像仓库,要先登陆到镜像仓库

```
docker push [OPTIONS] NAME[:TAG]
```

OPTIONS说明:

`--disable-content-trust` :忽略镜像的校验,默认开启

docker search

`docker search` : 从Docker Hub查找镜像

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

OPTIONS说明:

`--automated` :只列出 `automated build`类型的镜像;

`--no-trunc` :显示完整的镜像描述;

`-f <过滤条件>`:列出收藏数不小于指定值的镜像

//从 Docker Hub 查找所有镜像名包含 `nginx`, 并且收藏数大于 10 的镜像

```
docker search -f stars=10 nginx
```

参数说明:

NAME: 镜像仓库源的名称

DESCRIPTION: 镜像的描述

OFFICIAL: 是否 docker 官方发布

stars: 类似 Github 里面的 star, 表示点赞、喜欢的意思。

AUTOMATED: 自动构建。

镜像分享

tag命令的使用方式

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

```
docker tag IMAGE_ID TARGET_IMAGE[:TAG]
```

作用:

本地镜像管理, 可以用版本号来指定镜像的tag,方便镜像管理

可以用来修改本地镜像名和tag

指定目标远程仓库镜像和tag

公共仓库分享

1. 登录 <https://hub.docker.com/> 创建公有的 repository, 注意此处的repository具体到了镜像名

例如: 本例子使用的repository 为: `xlhmzch/mynginx` 表示 `xlhmzch` 这个账户下`mynginx`镜像

2. 登录远程仓库

```
//语法
//docker login [OPTIONS] [SERVER]
docker login
//打tag
docker tag mynginx:v1 x1hzmzch/mynginx:v1
//推送镜像
docker push x1hzmzch/mynginx:v1
```

3. 镜像推送成功之后，即可通过docker pull 拉取到镜像

docker save 加 docker load 分享

通过docker save 保存一个或多个镜像包到.tar文件，通过docker load 加载tar文件中的镜像

docker save : 将指定镜像保存成 tar 归档文件。

```
docker save -o images.tar mynginx:v1 mysql:5.7.30
```

docker load : 导入使用 docker save 命令导出的镜像。

```
docker load -i images.tar
```

私有注册中心搭建并分享镜像

```
docker run -d -p 5000:5000 --name registry --restart=always registry:2
docker ps -a
# curl http://localhost:5000/v2/_catalog
# {"repositories":""}
```

```
mkdir -p $HOME/registry/
mkdir -p $HOME/registry/auth
mkdir -p $HOME/registry/data
```

```
sudo apt install apache2-utils
```

```
htpasswd -Bbn testuser testpwd > /home/nick/registry/auth/htpasswd
```

```
docker run -d -p 5000:5000 --name registry --restart=always -v
$HOME/registry/data:/var/lib/registry -v $HOME/registry/auth:/auth -e
"REGISTRY_AUTH=htpasswd" -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" -e
REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd registry:2
```

```
vim /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
--insecure-registry localhost:5000
```

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

注意: --insecure-registry 指定一个受信任的注册中心, IP:Port 或 Domain:Port

```
docker login http://localhost:5000

docker tag mynginx:1.1 localhost:5000/mynginx:1.0.0
docker push localhost:5000/mynginx:1.0.0
```

数据卷

简介

Docker的镜像是有一系列的只读层组合而来，当启动一个容器时，Docker加载镜像的所有只读层，并在最上层加入一个读写层。这个设计使得Docker可以提高镜像构建、存储和分发的效率，节省了时间和存储空间，然而也存在一些问题：

1. 容器中的文件在宿主机上存在形式复杂，不能在宿主机上很方便地对容器中的文件进行访问。
2. 多个容器之间的数据无法共享
3. 当删除容器时，容器产生的数据将丢失。

为了解决这些问题，Docker引入了数据卷（volume）机制。**Volume 是存在于一个或多个容器中的特定文件或文件夹，这个目录以独立于联合文件系统的形式在宿主机中存在**，并为数据的共享和持久化提供便利。

1. volume 在容器创建时就会初始化，在容器运行时就可以使用其中的文件。
2. volume 能在不同的容器之间共享和重用
3. 对volume中数据的操作会马上生效
4. 对volume中数据的操作不会影响镜像本身
5. volume的生存周期独立于容器的生命周期，即使删除容器，volume仍然会存在，没有任何容器使用的volume也不会被Docker删除。

数据卷的使用

命令介绍

```
//Create a volume
docker volume create
//Display detailed information on one or more volumes
docker volume inspect
//List volumes
docker volume ls
//Remove all unused local volumes
docker volume prune
//Remove one or more volumes
docker volume rm
```

创建数据卷

```
//创建一个名为 vol_simple的存储卷
docker volume create --name vol_simple
```

Docker run 和 docker create 通过指定-v参数 可以为容器挂载一个数据卷

```
//没有指定volume卷，只指定了容器中的挂载点/data
docker run -d -it -v /data ubuntu /bin/bash
//查看随机创建的volume位置，查看mounts的内容
docker inspect {ID}
```

```
//创建一个指定名字的volume，并挂载到容器中的/data目录
docker run -d -v vol_simple:/data ubuntu
```

```
//查看数据卷的信息
docker volume inspect vol_simple
```

挂载数据卷

使用docker run 或者 docker create 创建新容器是，可以使用-v标签为容器添加volume。可以将自行创建或者有Docker创建的volume挂载到容器中，也可以将宿主机上的目录或者文件作为volume挂载到容器中。

将宿主机中指定目录作为volume挂载到容器中的/data目录下，文件夹必须使用绝对路径，如果宿主机中不存在指定的目录，则会创建一个空文件夹；如果宿主机文件夹已经存在，容器可以通过访问挂载点/data 从而访问宿主机文件夹中的所有内容。如果容器下原本已经存在/data文件夹，且不为空，**那么容器中文件夹下原有的内容将被隐藏，来保持与宿主机中的文件夹一致**

1. 挂载方式

```
//创建名为vol_simple的数据卷，并挂载到容器中的/data目录下
docker volume create --name vol_simple
docker run -d -v vol_simple:/data ubuntu /bin/bash

//创建一个随机的ID的volume，并将其挂载到容器中的/data目录下
docker run -d -v /data ubuntu /bin/bash

//将宿主机中指定目录作为volume挂载到容器中的/data目录下，文件夹必须使用绝对路径
//如果宿主机中不存在指定的目录，则会创建一个空文件夹；
//如果宿主机文件夹已经存在，容器可以通过访问挂载点/data 从而访问宿主机文件夹中的所有内容
//如果容器下原本已经存在/data文件夹，且不为空，那么容器中文件夹下原有的内容将被隐藏，来保持与宿主机中的文件夹一致
docker run -d -v $HOME/data:/data ubuntu /bin/bash
```

2. 只读挂载

```
# ro表示只读，rw表示读写，默认为rw
docker run -it -v $HOME/data:/data:ro ubuntu /bin/bash
# 通过修改文件验证
echo 123456abcdefg >> /data/test
```

注意，ro表示只读，rw表示读写，默认为rw

3. 同时挂载多个数据卷

```
docker run -it -v $HOME/data:/data:ro -v /data1 -v /data2 ubuntu /bin/bash
```

4. 通过dockerfile指定数据卷挂载

```
//dockerfile 添加指令
VOLUME /data
// 指定多个数据卷的挂载点
VOLUME ["/data1", "/data2"]
```

使用dockerfile VOLUME 指令，与docker run -v 不同的是，dockerfile指令不能挂载主机中指定的文件夹。这时为了保证Dockerfile的可移植性，因为不能保证所有的宿主机都有对应的文件夹。如果镜像中存才/data文件夹，这个文件夹中的内容将全部**被复制到宿主机上对应文件夹中**，并且根据容器中的文件设置合适的权限和所有者。

注意：在Dockerfile中使用VOLUME指令后的代码，如果尝试对这个volume进行修改，这些修改都不会生效。因为：**volume 是独立于rootfs的存储，镜像构建过程，每一层类似docker commit 提交临时镜像为镜像层，docker commit不会对挂载的volume进行保存。**

案例1：

VOLUME指定了挂载点之后，再对挂载点进行操作，由于docker commit 提交临时镜像不会对volume进行保存，所以该案例的data下的file并没有被保存，更不会同步到宿主机

```
FROM ubuntu
RUN useradd foo
VOLUME /data
RUN touch /data/file
RUN chown -R foo:foo /data
```

```
docker build -t test1:v1 -f Dockerfile .
docker run -d -it test1:v1 /bin/bash
//查看挂载点信息
docker inspect 容器ID
//查看宿主机存储卷目录
sudo ls 宿主机存储卷目录
```

案例二：

由于挂载volume时，/data目录已经存在，所以/data中的文件以及它们的权限和所有者设置都会被复制到volume中。

```
FROM ubuntu
RUN useradd foo
RUN mkdir /data && touch /data/file
RUN chown -R foo:foo /data
VOLUME /data
```

```
docker build -t test1:v2 -f Dockerfile1 .
docker run -d -it test1:v2 /bin/bash
//查看挂载点信息
docker inspect 容器ID
//查看宿主机存储卷目录
sudo ls 宿主机存储卷目录
```

案例三：

通过CMD和ENTRYPOINT指令，在容器启动时执行挂载点下文件的初始化

```
FROM ubuntu
RUN useradd foo
VOLUME /data
CMD touch /data/file && chown -R foo:foo /data
```

共享数据卷

在使用docker run 或docker create创建新容器时，可以使用--volumes-from 标签使得容器与已有容器共享volume。可以使用多个--volumes-from标签，使得容器与多个已有容器共享volume。

一个容器挂载了一个volume，即使这个容器停止运行，该volume人仍然存在，其他容器也可以使用--volume-from与这个容器共享volume。

作用

如果有一些数据，比如配置文件、数据文件等，要在多个容器之间共享，一种常见的做法是创建一个数据容器，其他的容器与之共享volume。

1. 创建一个带数据卷的容器

```
mkdir $HOME/data
mkdir $HOME/readOnly_data
docker volume create vol_simple
docker volume create vol_simple1
//创建一个数据容器 包含已创建数据卷，只读数据卷，私有数据卷等
docker run -d -it -v vol_simple:/vol_simple -v $HOME/data:/data -v
$HOME/readOnly_data:/readOnly_data:ro --name share_data ubuntu /bin/bash
docker run -d -it -v vol_simple1:/vol_simple1 --name share_data1 ubuntu
/bin/bash
//进入容器查看
docker exec -it 容器ID /bin/bash
```

```
// 运行新的容器通过数据容器共享数据卷 容器名: volume_from1
docker run -d -it --volumes-from share_data --volumes-from share_data1 --name
volume_from1 ubuntu /bin/bash
// 运行新的容器通过数据容器共享数据卷 容器名: volume_from2
docker run -d -it --volumes-from share_data --volumes-from share_data1 --name
volume_from2 ubuntu /bin/bash
```

1. 只读共享点不能被写入
2. 可以同时共享多个容器的数据卷
3. docker inspect 查看容器元数据

删除数据卷

如果创建容器时从容器中挂载了volume，在/var/lib/docker/volumes下会生产与volume对应的目录（可使用docker inspect 命令查看容器信息找到对应的信息）。

使用docker rm 删除容器并不会删除与volume对应的目录，这些目录会占据不必要的存储空间。

删除volume的三种方式：

1. 使用docker volume rm <volume_name> 删除数据卷
2. 使用docker rm -v <container_name> 删除容器时一并删除它挂载的数据卷
3. 在运行容器时使用docker run --rm 标签会在容器停止运行时删除容器以及容器所挂载的volume

注意：

1. 在使用第一种docker volume rm 删除时，只有当没有任何容器使用该volume的时候，才能被删除成功
2. 另外两种删除方式，只会对挂载在该容器上的未指定名称（匿名的）的volume进行删除，而会对用户指定名称的（具名的）volume进行保留
3. 如果volume 是在创建容器时从宿主机中挂载的，无论对容器进行任何操作都不会导致其在宿主机中被删除，如果不需要这些文件，只能手动删除。（从宿主机中挂载是指 docker run -v dir:dir 这种模式）

备份和迁移数据卷

volume 作为数据的载体，在很多情况下需要对其中的数据进行备份、迁移，或是从已有数据恢复。我们最容易想到一个备份还原的方式，那就是通过inspect 命令查看容器信息，找到对应的数据卷，手动打包数据；同样的还原那就是把打包好的数据解压到对应的数据卷。

除了上述原始的备份还原，我们采用--volumes-from实现备份与还原：

1. 备份：

启动另外一个临时容器，共享挂载数据容器share_data，同时挂载当前目录到容器的/backup。启动容器时执行打包命令，将/data挂载点下的数据打包到/backup/data.tar 文件中。--rm 表示该容器停止后会删除该容器和该容器的数据卷

```
docker run --rm --volumes-from share_data -v $(pwd):/backup ubuntu tar cvf /backup/data.tar /data
```

2. 恢复

```
docker run -d -it --name vol_bck -v /data ubuntu /bin/bash
```

创建临时容器，通过共享存储的方式与目标容器共享存储，同时挂载当前目录到容器的/backup。启动容器时从backup挂载点下的data.tar 解压文件到容器的根目录。

```
docker run --rm --volumes-from vol_bck -v $(pwd):/backup ubuntu tar xvf /backup/data.tar -C /
```

Docker Compose

Compose 简介

Docker Compose 是 Docker 官方编排（Orchestration）项目之一，负责快速的部署分布式应用。Compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排。从功能上看，跟 OpenStack 中的 Heat 十分类似。

其代码目前在 <https://github.com/docker/compose> 上开源。

Compose 定位是「定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）」，其前身是开源项目 Fig。

通过第一部分中的介绍，我们知道使用一个 Dockerfile 模板文件，可以让用户很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。

Compose 恰好满足了这样的需求。它允许用户通过一个单独的 docker-compose.yml 模板文件

(YAML 格式) 来定义一组相关联的应用容器为一个项目 (project)。

Compose 中有两个重要的概念：

- 服务 (service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
- 项目 (project)：由一组关联的应用容器组成的一个完整业务单元，在 `docker-compose.yml` 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。因此，只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

安装

在 Linux 上的安装十分简单，从 [官方 GitHub Release](#) 处直接下载编译好的二进制文件即可。

例如，在 Linux 64 位系统上直接下载对应的二进制包。

```
// 先把docker-compose文件dump到当前目录
wget https://github.com/docker/compose/releases/download/v2.5.0/docker-compose-
linux-x86_64
// 然后拷贝到/usr/bin/
$ sudo cp -arf docker-compose-linux-x86_64 /usr/bin/docker-compose
$ sudo chmod +x /usr/bin/docker-compose
```

卸载

如果是二进制包方式安装的，删除二进制文件即可。

```
$ sudo rm /usr/bin/docker-compose
```

实例-Web计数

术语

首先介绍几个术语。

- 服务 (service)：一个应用容器，实际上可以运行多个相同镜像的实例。
- 项目 (project)：由一组关联的应用容器组成的一个完整业务单元。

可见，一个项目可以由多个服务（容器）关联而成，`**Compose**` 面向项目进行管理。

准备

创建一个测试目录：

```
$ mkdir composetest
$ cd composetest
```

在测试目录中创建一个名为 `app.py` 的文件，并复制粘贴以下内容：

`composetest/app.py` 文件代码

```
import time
import redis
from flask import Flask
app = Flask(__name__)
```

```

cache = redis.Redis(host='redis', port=6379)
def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)
@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello world! I have been seen {} times.\n'.format(count)

```

在此示例中，redis 是应用程序网络上的 redis 容器的主机名，该主机使用的端口为 6379。在 composetest 目录中创建另一个名为 requirements.txt 的文件，内容如下：

```

flask
redis

```

创建 Dockerfile 文件

在 composetest 目录中，创建一个名为的文件 Dockerfile，内容如下：

```

FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
# RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .
CMD ["flask", "run"]

```

Dockerfile 内容解释：

- **FROM python:3.7-alpine:** 从 Python 3.7 映像开始构建镜像。
- **WORKDIR /code:** 将工作目录设置为 /code。

ENV FLASK_APP app.py

ENV FLASK_RUN_HOST 0.0.0.0

- 设置 flask 命令使用的环境变量。
- **RUN apk add --no-cache gcc musl-dev linux-headers:** 安装 gcc，以便诸如 MarkupSafe 和 SQLAlchemy 之类的 Python 包可以编译加速。这里先注释掉，下载太费时间了。

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

- 复制 requirements.txt 并安装 Python 依赖项。
- **COPY . .:** 将 . 项目中的当前目录复制到 . 镜像中的工作目录。
- **CMD ["flask", "run"]:** 容器提供默认的执行命令为：flask run。

创建 docker-compose.yml

在测试目录中创建一个名为 docker-compose.yml 的文件，然后粘贴以下内容：

```
# yaml 配置
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

该 Compose 文件定义了两个服务：web 和 redis。

- **web**：该 web 服务使用从 Dockerfile 当前目录中构建的镜像。然后，它将容器和主机绑定到暴露的端口 5000。此示例服务使用 Flask Web 服务器的默认端口 5000。
- **redis**：该 redis 服务使用 Docker Hub 的公共 Redis 映像。

使用 Compose 命令构建和运行您的应用

在测试目录中，执行以下命令来启动应用程序：

```
docker-compose up
```

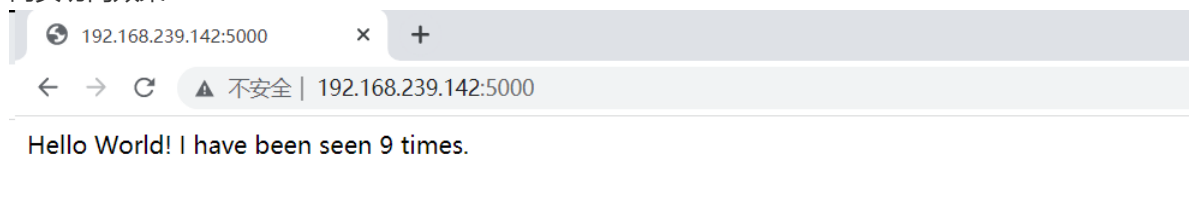
如果你想在后台执行该服务可以加上 **-d** 参数：docker-compose up -d

运行后的效果

```
# 192e93523482 Pull complete
# 7151bccd2756 Pull complete
# 683d02ead94f Pull complete
# b4ca937b9a43 Pull complete
# b4bb2d8d1296 Pull complete
[+] Building 69.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 281B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/python:3.7-alpine
=> [1/5] FROM docker.io/library/python:3.7-alpine@sha256:ad5383edd9d9109639a44d725314d1f37af256e81c86a04a0e8e077f899f63ca
=> => resolve docker.io/library/python:3.7-alpine@sha256:ad5383edd9d9109639a44d725314d1f37af256e81c86a04a0e8e077f899f63ca
=> => sha256:ad5383edd9d9109639a44d725314d1f37af256e81c86a04a0e8e077f899f63ca 1.65kB / 1.65kB
=> => sha256:634f06af32afe9926177560feebd9f2ae14aa9a70e1d3cc0a124495a5d7168ae 1.37kB / 1.37kB
=> => sha256:56dc087ee36a4664ca9d7a40f4328bd353e4c651cf64328a0761f2d9ecaadeb1 7.71kB / 7.71kB
=> => sha256:a1ef3e6b7a02c0e1d7d3ee96f273c2c5074b3f127082b4cd3f629f82097e1a8 667.03kB / 667.03kB
=> => sha256:592c06358a0be3e5bea7de17c359984f3180e3e57aa18b5afb51cbffbb6df7c71 11.19MB / 11.19MB
=> => sha256:cb59fb21dd583d1d59b6815c76fb183a76fad49e34f19cb39a7f4b2d5310f8e9 231B / 231B
=> => sha256:4304e370afec6248784d847b401b6fc02599663ee50dc37f68731241d10ceb46 2.87MB / 2.87MB
=> => extracting sha256:a1ef3e6b7a02c0e1d7d3ee96f273c2c5074b3f127082b4cd3f629f82097e1a8
=> => extracting sha256:592c06358a0be3e5bea7de17c359984f3180e3e57aa18b5afb51cbffbb6df7c71
=> => extracting sha256:cb59fb21dd583d1d59b6815c76fb183a76fad49e34f19cb39a7f4b2d5310f8e9
=> => extracting sha256:4304e370afec6248784d847b401b6fc02599663ee50dc37f68731241d10ceb46
=> [internal] load build context
=> => transferring context: 1.05kB
=> [2/5] WORKDIR /code
=> [3/5] COPY requirements.txt requirements.txt
=> [4/5] RUN pip install -r requirements.txt
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:358ec0412794a4495701eb473d794542d1c1f35a47500a72521193414113a4fc
=> => naming to docker.io/library/composetest_web
[+] Running 3/3
# Network composetest_default Created
# Container composetest-redis-1 Started
# Container composetest-web-1 Started
nick@nick:~/composetest$ docker ps
```

以上效果图现实，创建了一个网络，一个Redis 和一个web

网页访问效果：



进入redis容器，查看redis数据

```
nick@nick:~/composetest$ docker exec -it composetest-redis-1 /bin/sh
/data # ls
/data # cd ..
# # ls
bin  data  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
/ # cd bin/
/bin # ls
arch          chmod          dnsdomainname fdflush        hostname       linux32        mkdir          mv             netstat        printenv       run-parts      stty           uname
ash           chown          dumpmap        fgrep          iostat        linux64        mknod          nice           ps             sed            su             usleep
base64        cp             echo           fsync          iostat        ln             more           nice           pwd            setpriv        tar            watch
bbconfig      date           ed             getopt         ipcalc        login          mount          pidof          reformime      setserial      true          zcat
busybox       df             egrep          grep           kbd_mode     ls             mount          ping           rev            sh             sleep          touch
cat           dd             false          gunzip         kill          lzop           mountpoint     ping6          rm             sleep          stat           true
chgrp         dmesg          fatattr        gzip           link          mknod          mpstat         pipe_progress  radir          stat           true          umount

/bin # re
read          readlink       realpath       redis-benchmark redis-check-rdb redis-sentinel reformime       renice         resize         rev
readhead      readonly       reboot         redis-check-aof redis-cli       redis-server   remove-shell   reset          return
/bin # re
read          readlink       realpath       redis-benchmark redis-check-rdb redis-sentinel reformime       renice         resize         rev
readhead      readonly       reboot         redis-check-aof redis-cli       redis-server   remove-shell   reset          return
/bin # redis-cli
127.0.0.1:6379> keys *
1) "hits"
127.0.0.1:6379> ls
(error) ERR unknown command 'ls', with args beginning with:
127.0.0.1:6379> get hits
"8"
127.0.0.1:6379> get hits
"9"
127.0.0.1:6379>
```

查看docker 网络信息

```
docker network ls
docker network inspect composetest_default
```

Compose 命令说明

命令对象与格式

官方文档: <https://docs.docker.com/compose/reference/>

对于 Compose 来说, 大部分命令的对象既可以是项目本身, 也可以指定为项目中的服务或者容器。如果没有特别的说明, 命令对象将是项目, 这意味着项目中所有的服务都会受到命令影响。

执行 `docker-compose [COMMAND] --help` 或者 `docker-compose help [COMMAND]` 可以查看具体某个命令的使用格式。

`docker-compose` 命令的基本的使用格式是

```
docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
```

命令选项

- `-f, --file FILE` 指定使用的 Compose 模板文件, 默认为 `docker-compose.yml`, 可以多次指定。
- `-p, --project-name NAME` 指定项目名称, 默认将使用所在目录名称作为项目名。
- `--x-networking` 使用 Docker 的可拔插网络后端特性
- `--x-network-driver DRIVER` 指定网络后端的驱动, 默认为 `bridge`
- `--verbose` 输出更多调试信息。
- `-v, --version` 打印版本并退出。

命令使用说明

build

格式为 `docker-compose build [options] [SERVICE...]`。

构建 (重新构建) 项目中的服务容器。

服务容器一旦构建后, 将会带上一个标记名, 例如对于 web 项目中的一个 db 容器, 可能是 web_db。

可以随时在项目目录下运行 `docker-compose build` 来重新构建服务。

选项包括:

- `--force-rm` 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。

config

验证 Compose 文件格式是否正确，若正确则显示配置，若格式错误显示错误原因。

down

此命令将会停止 `up` 命令所启动的容器，并移除网络

exec

进入指定的容器。

help

获得一个命令的帮助。

images

列出 Compose 文件中包含的镜像。

kill

格式为 `docker-compose kill [options] [SERVICE...]`。

通过发送 `SIGKILL` 信号来强制停止服务容器。

支持通过 `-s` 参数来指定发送的信号，例如通过如下指令发送 `SIGINT` 信号。

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

logs

格式为 `docker-compose logs [options] [SERVICE...]`。

查看服务容器的输出。默认情况下，docker-compose 将对不同的服务输出使用不同的颜色来区分。

可以通过 `--no-color` 来关闭颜色。

该命令在调试问题的时候十分有用。

pause

格式为 `docker-compose pause [SERVICE...]`。

暂停一个服务容器。

port

格式为 `docker-compose port [options] SERVICE PRIVATE_PORT`。

打印某个容器端口所映射的公共端口。

选项：

- `--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）。

ps

格式为 `docker-compose ps [options] [SERVICE...]`。

列出项目中目前的所有容器。

选项：

- `-q` 只打印容器的 ID 信息。

pull

格式为 `docker-compose pull [options] [SERVICE...]`。

拉取服务依赖的镜像。

选项：

- `--ignore-pull-failures` 忽略拉取镜像过程中的错误。

push

推送服务依赖的镜像到 Docker 镜像仓库。

restart

格式为 `docker-compose restart [options] [SERVICE...]`。

重启项目中的服务。

选项：

- `-t, --timeout TIMEOUT` 指定重启前停止容器的超时（默认为 10 秒）。

rm

格式为 `docker-compose rm [options] [SERVICE...]`。

删除所有（停止状态的）服务容器。推荐先执行 `docker-compose stop` 命令来停止容器。

选项：

- `-f, --force` 强制直接删除，包括非停止状态的容器。一般尽量不要使用该选项。
- `-v` 删除容器所挂载的数据卷。

run

格式为 `docker-compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]`。

在指定服务上执行一个命令。

```
$ docker-compose run web ping docker.com
```

将会启动一个 ubuntu 服务容器，并执行 `ping docker.com` 命令。

默认情况下，如果存在关联，则所有关联的服务将会自动被启动，除非这些服务已经在运行中。

该命令类似启动容器后运行指定的命令，相关卷、链接等等都将会按照配置自动创建。

两个不同点：

- 给定命令将会覆盖原有的自动运行命令；
- 不会自动创建端口，以避免冲突。

如果不希望自动启动关联的容器，可以使用 `--no-deps` 选项，例如

```
//该操作作为伪操作
docker-compose run --no-deps web python manage.py shell
```

将不会启动 web 容器所关联的其它容器。

选项：

- `-d` 后台运行容器。
- `--name NAME` 为容器指定一个名字。

- `--entrypoint CMD` 覆盖默认的容器启动指令。
- `-e KEY=VAL` 设置环境变量值，可多次使用选项来设置多个环境变量。
- `-u, --user=""` 指定运行容器的用户名或者 uid。
- `--no-deps` 不自动启动关联的服务容器。
- `--rm` 运行命令后自动删除容器，`d` 模式下将忽略。
- `-p, --publish=[]` 映射容器端口到本地主机。
- `--service-ports` 配置服务端口并映射到本地主机。
- `-T` 不分配伪 tty，意味着依赖 tty 的指令将无法运行。

start

格式为 `docker-compose start [SERVICE...]`。

启动已经存在的服务容器。

stop

格式为 `docker-compose stop [options] [SERVICE...]`。

停止已经处于运行状态的容器，但不删除它。通过 `docker-compose start` 可以再次启动这些容器。

选项：

- `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

top

查看各个服务容器内运行的进程。

unpause

格式为 `docker-compose unpause [SERVICE...]`。

恢复处于暂停状态中的服务。

up

格式为 `docker-compose up [options] [SERVICE...]`。

该命令十分强大，它将尝试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。

链接的服务都将会被自动启动，除非已经处于运行状态。

可以说，大部分时候都可以直接通过该命令来启动一个项目。

默认情况，`docker-compose up` 启动的容器都在前台，控制台将会同时打印所有容器的输出信息，可以很方便进行调试。

当通过 `Ctrl-C` 停止命令时，所有容器将会停止。

如果使用 `docker-compose up -d`，将会在后台启动并运行所有的容器。一般推荐生产环境下使用该选项。

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（保持使用 `volumes-from` 挂载的卷），以保证新启动的服务匹配 `docker-compose.yml` 文件的最新内容。如果用户不希望容器被停止并重新创建，可以使用 `docker-compose up --no-recreate`。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。如果用户只想重新部署某个服务，可以使用 `docker-compose up --no-deps -d <SERVICE_NAME>` 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。

选项：

- `-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。
- `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

version

格式为 `docker-compose version`。

打印版本信息。

yaml 配置指令参考（以下内容只作为参考）

官方文档: <https://docs.docker.com/compose/compose-file/>

version

指定本 yaml 依从的 compose 哪个版本制定的

build

指定为构建镜像上下文路径：

例如 webapp 服务，指定为从上下文路径 `./dir/Dockerfile` 所构建的镜像

```
version: "3.7"
services:
  webapp:
    build: ./dir
```

或者，作为具有在上下文指定的路径的对象，以及可选的 Dockerfile 和 args：

```
version: "3.7"
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
      labels:
        - "com.example.description=Accounting webapp"
        - "com.example.department=Finance"
        - "com.example.label-with-empty-value"
      target: prod
```

- context: 上下文路径。
- dockerfile: 指定构建镜像的 Dockerfile 文件名。
- args: 添加构建参数，这是只能在构建过程中访问的环境变量。
- labels: 设置构建镜像的标签。
- target: 多层构建，可以指定构建哪一层。

command

覆盖容器启动的默认命令。

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

container_name

指定自定义容器名称，而不是生成的默认名称

```
container_name: my-web-container
```

depends_on

设置依赖关系。

- docker-compose up：以依赖性顺序启动服务。在以下示例中，先启动 db 和 redis，才会启动 web。
- docker-compose up SERVICE：自动包含 SERVICE 的依赖项。在以下示例中，docker-compose up web 还将创建并启动 db 和 redis。
- docker-compose stop：按依赖关系顺序停止服务。在以下示例中，web 在 db 和 redis 之前停止。

```
version: "3.7"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

注意：web 服务不会等待 redis db 完全启动 之后才启动

entrypoint

覆盖容器默认的 entrypoint。

```
entrypoint: /code/entrypoint.sh
```

也可以是以下格式：

```
entrypoint:
  - php
  - -d
    zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-
20100525/xdebug.so
  - -d
    memory_limit=-1
  - vendor/bin/phpunit
```

env_file

从文件添加环境变量。可以是单个值或列表的多个值。

```
env_file: .env
```

也可以是列表格式：

```
env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

environment

添加环境变量。您可以使用数组或字典、任何布尔值，布尔值需要用引号引起来，以确保 YML 解析器不会将其转换为 True 或 False。

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

expose

暴露端口，但不映射到宿主机，只被连接的服务访问。

仅可以指定内部端口为参数：

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

extra_hosts

添加主机名映射。类似 docker client --add-host。

```
extra_hosts:
  - "somehost:162.242.195.82"
  - "otherhost:50.31.209.229"
```

以上会在此服务的内部容器中 /etc/hosts 创建一个具有 ip 地址和主机名的映射关系：

```
162.242.195.82  somehost
50.31.209.229   otherhost
```

healthcheck

用于检测 docker 服务是否健康运行。

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"] # 设置检测程序
  interval: 1m30s # 设置检测间隔
  timeout: 10s # 设置检测超时时间
  retries: 3 # 设置重试次数
  start_period: 40s # 启动后, 多少秒开始启动检测程序
```

image

指定容器运行的镜像。以下格式都可以：

```
image: redis
image: ubuntu:14.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd # 镜像id
```

logging

服务的日志记录配置。

driver: 指定服务容器的日志记录驱动程序，默认值为json-file。有以下三个选项

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

仅在 json-file 驱动程序下，可以使用以下参数，限制日志得数量和大小。

```
logging:
  driver: json-file
  options:
    max-size: "200k" # 单个文件大小为200k
    max-file: "10" # 最多10个文件
```

当达到文件限制上限，会自动删除旧得文件。

syslog 驱动程序下，可以使用 syslog-address 指定日志接收地址。

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

restart

- no: 是默认的重启策略，在任何情况下都不会重启容器。
- always: 容器总是重新启动。
- on-failure: 在容器非正常退出时（退出状态非0），才会重启容器。
- unless-stopped: 在容器退出时总是重启容器，但是不考虑在Docker守护进程启动时就已经停止了的容器

```
restart: "no"
restart: always
restart: on-failure
restart: unless-stopped
```

volumes

将主机的数据卷或者文件挂载到容器里。

```
version: "3.7"
services:
  db:
    image: postgres:latest
    volumes:
      - "/localhost/postgres.sock:/var/run/postgres/postgres.sock"
      - "/localhost/data:/var/lib/postgresql/data"
```

Docker Swarm

简介

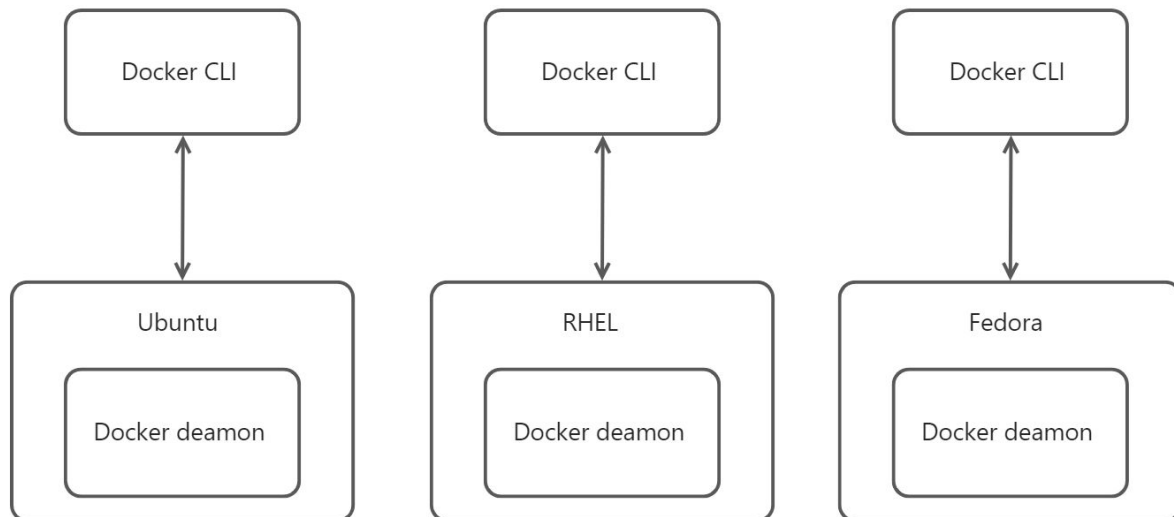
Swarm 是什么？

Docker Swarm 是Docker官方的跨节点的容器编排工具。用户只需要在单一的管理节点上操作，即可管理集群下的所有节点和容器。

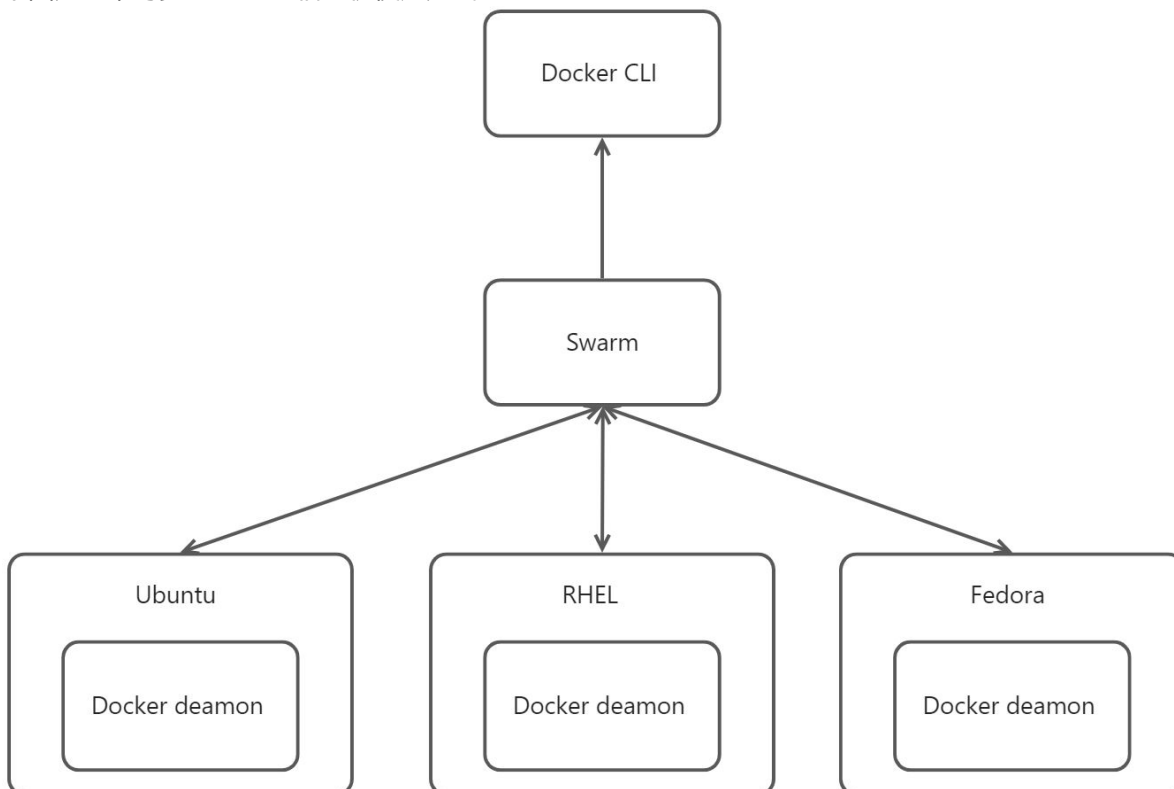
主要解决什么问题

1. 解决docker server的集群化管理和部署。
2. Swarm通过对Docker宿主机上添加的标签信息来将宿主主机资源进行细粒度分区，通过分区来帮助用户将容器部署到目标宿主主机上，同样通过分区方式还能提供更多的资源调度策略扩展。

下图为：单对单的Docker宿主机使用方式



下图为：单对多的Docker宿主机使用方式



能做什么

1. 管理节点高可用，原生支持管理节点高可用，采用raft共识算法来支撑管理节点高可用。
2. 应用程序高可用，支持服务伸缩，滚动更新和应用回滚等部署策略

涉及到哪些概念

1. docker的两种模式，单引擎模式和swarm集群模式

单引擎模式，之docker server没有加入任何集群，且自身也没有加入初始化为swarm 节点，简单的说就我我们平时所操作的孤立的docker server。

swarm模式，当docker server 加入到任意swarm集群，或者通过docker swarm init初始化swarm集群时，docker server会自动切换到swarm 集群模式。

2. swarm集群中节点分类，分为：manager（管理节点）、node（工作节点）

manager: 是Swarm Daemon工作的节点, 包含了调度器、路由、服务发现等功能, 负责接收客户端的集群管理请求以及调度Node进行具体工作。manager 本身也是一个node节点
Node: 接受manager调度, 对容器进行创建、扩容和销毁等具体操作。

3. raft共识算法, 是实现分布式共识的一种算法, 主要用来管理日志复制的一致性。

当Docker引擎在swarm模式下运行时, manager节点实现Raft一致性算法来管理全局集群状态。

Docker swarm模式之所以使用一致性算法, 是为了确保集群中负责管理和调度任务的所有manager节点都存储相同的一致状态。

在整个集群中具有相同的一致状态意味着, 如果出现故障, 任何管理器节点都可以拾取任务并将服务恢复到稳定状态。例如, 如果负责在集群中调度任务的领导管理器意外死亡, 则任何其他管理器都可以选择调度任务并重新平衡任务以匹配所需状态。

使用一致性算法在分布式系统中复制日志的系统需要特别小心。它们通过要求大多数节点在值上达成一致, 确保集群状态在出现故障时保持一致。

Raft最多可承受 $(N-1)/2$ 次故障, 需要 $(N/2) + 1$ 名成员的多数或法定人数才能就向集群提议的值达成一致。这意味着, 在运行Raft的5个管理器集群中, 如果3个节点不可用, 系统将无法处理更多的请求来安排其他任务。现有任务保持运行, 但如果管理器集不正常, 调度程序无法重新平衡任务以应对故障。

4. Swarm管理节点高可用

Swarm管理节点内置有对HA的支持, 即使有一个或多个节点发送故障, 剩余管理节点也会继续保证Swarm运转

Swarm实现了一种主从方式的多管理节点的HA, 即使有多个管理节点也只有一个节点出于活动状态, 处于活动状态的节点被称为主节点(leader), 而主节点也是唯一一个会对Swarm发送控制命令的节点, 如果一个备用管理节点接收到了Swarm命令, 则它会将其转发给主节点

5. 集群管理, 集群创建和组织。

6. 节点管理, 集群下节点角色信息变更, 节点任务情况监控。

7. 服务管理, 集群服务部署, 服务管理等。

需要注意什么

1. manager节点容错

manager数量最好设置为奇数个, 可以减少脑裂(Split-Brain)出现情况。在网络被划分成2个部分情况下, 奇数个manager节点能够较高等度的保证有投票结果的可能性

不要部署太多的manager节点(通常3到5个)。对于所有共识算法来说, 更多的参与节点意味着需要花费更多的时间来达成共识, 所以最好部署5个或三个节点

Raft最多可承受 $(N-1)/2$ 次故障, 需要 $(N/2) + 1$ 名成员的多数或法定人数才能就向集群提议的值达成一致

manager节点数	选举票数	允许manager不可用个数
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

集群管理

创建集群

1. 不包含在任何Swarm中的Docker节点，该Docker节点被称为运行于单引擎(Single-Engine)模式，一旦加入Swarm集群，则切换为Swarm模式
2. `docker swarm init` 会通知Docker来初始化一个新的Swarm，并且将自身设置为第一个管理节点。同时也会使该节点开启Swarm模式
3. `--advertise-addr IP:2377` 指定其他节点用来连接到当前管理节点的IP和端口，这一指令是可选的，当节点有多个IP时，可以指定其中一个
4. `--listen-addr` 指定用于承载Swarm流量的IP和端口，其设置通常与`--advertise-addr`相匹配，但是当节点上有多个IP的时候，可用于指定具体某个IP
5. `--autolock` 启用管理器自动锁定（需要解锁密钥才能启动已停止的管理器）
6. `--force-new-cluster` 强制从当前状态创建新群集

```
docker swarm init
```

将节点加入集群

1. 生成join-token

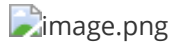
```
# 生成work节点 join-token
docker swarm join-token worker
# 生成manager节点 join-token
docker swarm join-token manager
```

2. 加入集群

```
# 此命令为 join-token 命令执行结果，在相应的节点执行该结果即可加入到集群
docker swarm join --token SWMTKN-1-
36o8i823751vozwd75mpzgsy28h4ti93zbn4o9wht8kywj35ir-6zh4id2t0ck4on4oyw95xs3cs
192.168.239.142:2377
```


查看集群状态

```
docker info
```



将节点从集群中移除

```
# 将节点从集群中移除 （只能移除worker节点）
docker swarm leave
# 将节点从集群中强制移除（包括manager节点）
docker swarm leave -f
```

更新集群

```
# 更新集群的部分参数
docker swarm update --autolock=false
```

锁定/解锁集群

1. 重启一个旧的管理节点或者进行备份恢复可能对集群造成影响，一个旧的管理节点重新接入Swarm会自动解密并获得Raft数据库中长时间序列的访问权，这会带来安全隐患。进行备份恢复可能会抹掉最新的Swarm配置
2. 为了规避上述问题，Docker提供自动锁机制来锁定Swarm，这会强制要求重启的管理节点在提供一个集群解锁码之后才有权重新接入集群（也可以防止原来的主节点宕机后重新接入集群，和当前主节点一起成为双主，双主也是一种脑裂问题）

```
# 设置为自动锁定集群
docker swarm update --autolock=true
```

```
# 当集群设置为 --autolock后，可以通过该命令查询解锁集群的密钥
# 如果该节点必须为集群有效的管理节点
docker swarm unlock-key
```

```
# 重启管理节点，集群将被自动锁定
service docker restart
```

```
# 重启后的管理节点必须提供解锁码后才能重新接入集群
docker swarm unlock
```

节点管理

```
docker node -h

# 降级节点
demote      Demote one or more nodes from manager in the swarm
# 查看节点详情
inspect     Display detailed information on one or more nodes
# 查看所有节点
```

```
ls          List nodes in the swarm
# 升级节点
promote     Promote one or more nodes to manager in the swarm
# 查看节点上运行的任务，默认当前节点
ps          List tasks running on one or more nodes, defaults to current node
# 删除节点
rm          Remove one or more nodes from the swarm
# 更新节点
update      Update a node
```

查看集群节点列表

```
docker node ls
```

升级或降级节点

```
# 降级一个或多个节点
docker node demote <NODE>
# 通过修改单个节点的role属性，来降级节点
docker node update --role worker <NODE>

# 升级一个或多个节点
docker node promote <NODE>
# 通过修改单个节点的role属性，来升级节点
docker node update --role manager <NODE>
```

更改节点状态

```
# 更改节点状态
docker node update --availability active|pause|drain <NODE>

# active: 正常
# pause: 挂起
# drain: 排除
```

1. 正常节点，可正常部署应用
2. 挂起节点，已经部署的应用不会发生变化，新应用将不会部署到该节点
3. 排除节点，已经部署在该节点的应用会被调度到其他节点

删除节点

```
# 只能删除已关闭服务的工作节点
docker node rm <node>
# 只能强制删除工作节点
docker node rm -f <node>
```

注：管理节点的删除只能先将管理节点降级为工作节点，再执行删除动作

```
# 停止docker server
sudo service docker stop
```

服务部署

素材准备

1. 拉取代码、获取tag

```
# 拉取代码
git clone https://gitee.com/nickdemo/helloworld.git

# 进入代码目录
cd helloworld

# fetch
git fetch origin
```

2. 镜像制作

```
# 切换到1.0.0 tag
git checkout 1.0.0
```

```
# 构建1.0.0镜像
docker build -f Dockerfile -t hello:1.0.0 .
```

```
# 切换到1.0.1 tag
git checkout 1.0.1
```

```
# 构建1.0.1镜像
docker build -f Dockerfile -t hello:1.0.1 .
```

3. 将镜像推送到仓库（这里使用公共仓库作为演示）


```
# 登录注册中心
docker login
```


4. 为镜像重新打tag（格式：用户名/仓库名:tag）


```
docker tag hello:1.0.0 xlmzch/hello:1.0.0
docker push xlmzch/hello:1.0.0

docker tag hello:1.0.1 xlmzch/hello:1.0.1
docker push xlmzch/hello:1.0.1
```

5. 登录<https://hub.docker.com/> 查看镜像情况

 **xlhzmzch / hello**

This repository does not have a description 

 Last pushed: a few seconds ago


Docker commands

To push a new tag to this repository,





`docker push xlhzmzch/hello:tagname`

Public View

Tags and Scans

 VULNERABILITY SCANNING - DISABLED [Enable](#)

This repository contains 2 tag(s).

TAG	OS	PULLED	PUSHED
 1.0.1		---	a few seconds ago
 1.0.0		---	5 minutes ago

[See all](#)

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions.

Upgrade to Pro

[Learn more](#)

服务管理

常用命令

```
docker service -h
```

```
# 创建一个服务
```

```
create          Create a new service
```

```
# 查看服务详细信息
```

```
inspect         Display detailed information on one or more services
```

```
# 查看服务日志
```

```
logs           Fetch the logs of a service or task
```

```
# 列出所有服务
```

```
ls             List services
```

```
# 列出一个或多个服务的任务列表
```

```
ps            List the tasks of one or more services
```

```
# 删除一个或多个服务
```

```
rm            Remove one or more services
```

```
# 回滚
```

```
rollback      Revert changes to a service's configuration
```

```
# 弹性伸缩一个或多个服务
```

```
scale         Scale one or multiple replicated services
```

```
# 更新服务
```

```
update        Update a service
```

在集群上部署应用

1. 部署一个副本数量

```
docker service create --name myhello --publish published=81,target=80 --replicas 3 xlhzmzch/hello:1.0.0
```

2. 查看服务下的任务（容器）

```
docker service ps myhello
```

3. 查看节点下的任务（容器）

```
docker node ps <NODE>
```

- 访问集群中任意节点（包括没有运行任务的节点）对应的端口号均能访问到应用程序，swarm为集群实现了负载均衡

```
curl http://localhost:81/ping
```

- 查看service详细信息

```
docker service inspect myhello
```

- 查看service日志

```
docker service logs myhello
```

- 伸缩服务

```
docker service scale myhello=5
```

- 修改节点状态，查看任务部署情况

```
docker node update --availability active|pause|drain <NODE>
```

在集群上部署一个带更新策略和回滚策略的应用

- 部署应用

```
docker service create \
--name myhello1 \
--publish published=82,target=80 \
--replicas 20 \
--update-delay 5s \
--update-parallelism 2 \
--update-failure-action continue \
--rollback-parallelism 2 \
--rollback-monitor 10s \
--rollback-max-failure-ratio 0.2 \
xlhmzch/hello:1.0.0

#--update-delay 5s : 每个容器依次更新，间隔5s

# --update-parallelism 2 : 每次允许两个服务一起更新

#--update-failure-action continue : 更新失败后的动作是继续

# --rollback-parallelism 2 : 回滚时允许两个一起

# --rollback-monitor 10s : 回滚监控时间10s

# --rollback-max-failure-ratio 0.2 : 回滚失败率20%
```

- 检查部署后的应用设置项是否都被成功设置

```
# 查看并打印友好的详细信息
docker service inspect --pretty myhello1
```

注意：查看详细信息时，详细信息中时间单位为ns(纳秒)。友好格式下，时间单位为 (s)

3. 更新未设置成功的项

```
docker service update --update-delay 5s --rollback-monitor 10s myhello1
```

4. 访问服务

```
curl http://localhost:82/ping
curl http://localhost:82/ping/v1.0.1
```

更新服务

1. 启动更新

```
docker service update --image xlhmzch/hello:1.0.1 myhello1
```

```
nick@nick:~/work/helloworld$ docker service update --image xlhmzch/hello:1.0.1 myhello1
myhello1
overall progress: 2 out of 20 tasks
1/20:
2/20:
3/20:
4/20:
5/20:
6/20:
7/20: running  [=====>]
8/20:
9/20:
10/20:
11/20:
12/20:
13/20:
14/20:
15/20:
16/20:
17/20:
18/20:
19/20: running  [=====>]
20/20:
```

2. 更新完成

```

nick@nick:~$ docker service ps myhello1
ID            NAME                IMAGE                NODE                DESIRED STATE    CURRENT STATE    ERROR
ltt9kg1k5s8f myhello1.1          xlmzch/hello:1.0.1  ubuntu20            Running           Running 22 seconds ago
1308ypv9bv9j  \_ myhello1.1       xlmzch/hello:1.0.0  ubuntu20            Shutdown          Shutdown 23 seconds ago
j6vvvs3qlx86  myhello1.2          xlmzch/hello:1.0.1  nick                Running           Running 1 second ago
ojwk9fftouxc  \_ myhello1.2       xlmzch/hello:1.0.0  nick                Shutdown          Shutdown 2 seconds ago
lik49x2eksix  myhello1.3          xlmzch/hello:1.0.0  nick                Running           Running 27 minutes ago
61rbzdh3h3ay  myhello1.4          xlmzch/hello:1.0.0  ubuntu20qhtest      Running           Running 27 minutes ago
n0tkgyzbmqnn  myhello1.5          xlmzch/hello:1.0.1  ubuntu20            Running           Running 15 seconds ago
6oqs3pxoh5ni  \_ myhello1.5       xlmzch/hello:1.0.0  ubuntu20            Shutdown          Shutdown 16 seconds ago
w3ufi0bgtd6r  myhello1.6          xlmzch/hello:1.0.1  nick                Running           Running 29 seconds ago
x2350rqfakmh  \_ myhello1.6       xlmzch/hello:1.0.0  nick                Shutdown          Shutdown 29 seconds ago
w1flj20pqb2u  myhello1.7          xlmzch/hello:1.0.0  ubuntu20qhtest      Running           Running 27 minutes ago
fci535eaqt1g  myhello1.8          xlmzch/hello:1.0.0  nick                Running           Running 27 minutes ago
ukpqj507lm8m  myhello1.9          xlmzch/hello:1.0.1  ubuntu20            Running           Running 42 seconds ago
6e7hllmu7giz  \_ myhello1.9       xlmzch/hello:1.0.0  ubuntu20qhtest      Shutdown          Shutdown 49 seconds ago
z29rn2g0wjm  myhello1.10         xlmzch/hello:1.0.0  ubuntu20qhtest      Running           Running 27 minutes ago
4vtklo9ulgiv  myhello1.11         xlmzch/hello:1.0.0  nick                Running           Running 27 minutes ago
ljbwzsojmdct  myhello1.12         xlmzch/hello:1.0.1  nick                Running           Running 8 seconds ago
b7eohlympieu  \_ myhello1.12      xlmzch/hello:1.0.0  nick                Shutdown          Shutdown 9 seconds ago
o5bpxpre3ynx3 myhello1.13         xlmzch/hello:1.0.1  nick                Running           Running 1 second ago
uk5jgojf2so  \_ myhello1.13      xlmzch/hello:1.0.0  nick                Shutdown          Shutdown 2 seconds ago
vklkfns8xsxv  myhello1.14         xlmzch/hello:1.0.0  nick                Running           Running 27 minutes ago
tjli4s3wioqs  myhello1.15         xlmzch/hello:1.0.1  ubuntu20            Running           Running 35 seconds ago
ga9vlzp7zlnk  \_ myhello1.15      xlmzch/hello:1.0.0  ubuntu20            Shutdown          Shutdown 36 seconds ago
mtow4rgf4upf  myhello1.16         xlmzch/hello:1.0.1  nick                Running           Running 1 second ago
ku5jgojf2so  \_ myhello1.16      xlmzch/hello:1.0.1  nick                Shutdown          Rejected 8 seconds ago      "No such image: xlmzch/hello:1.0.1"
65ut21orssr7  \_ myhello1.16      xlmzch/hello:1.0.0  nick                Shutdown          Shutdown 22 seconds ago
cnk885pve7kp  myhello1.17         xlmzch/hello:1.0.0  nick                Running           Running 27 minutes ago
j7bciultezk8  myhello1.18         xlmzch/hello:1.0.1  nick                Running           Running 29 seconds ago
v9gieq49wgh5  \_ myhello1.18      xlmzch/hello:1.0.0  nick                Shutdown          Shutdown 36 seconds ago
iryy926iik57  myhello1.19         xlmzch/hello:1.0.1  ubuntu20qhtest      Running           Running 42 seconds ago
c0valhztrcj0  \_ myhello1.19      xlmzch/hello:1.0.0  ubuntu20            Shutdown          Shutdown 48 seconds ago
yjhph71jfy9et myhello1.20         xlmzch/hello:1.0.0  nick                Running           Running 27 minutes ago
nick@nick:~$

```

3. 访问服务

```

curl http://localhost:82/ping
curl http://localhost:82/ping/v1.0.1

```

回滚服务

1. 启动回滚（策略是失败后回滚，目前没有失败的情况，我们手动回滚）

```

docker service update --rollback myhello1

```

```

nick@nick:~/work/helloworld$ docker service update --rollback myhello1
myhello1
rollback: manually requested rollback
overall progress: rolling back update: 7 out of 20 tasks
1/20:
2/20:
3/20:
4/20:
5/20: running  [>
6/20: running  [>
7/20:
8/20: running  [>
9/20:
10/20: pending  [=====>
11/20:
12/20:
13/20: starting [=====>
14/20: running  [>
15/20:
16/20: running  [>
17/20: running  [>
18/20: running  [>
19/20:
20/20:

```

2. 回滚成功

```
nick@nick:~$ docker service ps myhello1
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
ul3h1fk1t4xy	myhello1.1	xlhmzch/hello:1.0.0	ubuntu20	Running	Running 58 seconds ago	
ltj9kg1k5s8f	_ myhello1.1	xlhmzch/hello:1.0.1	ubuntu20	Shutdown	Shutdown 59 seconds ago	
1308ypy9bv9j	_ myhello1.1	xlhmzch/hello:1.0.0	ubuntu20	Shutdown	Shutdown 4 minutes ago	
cixq4w02z9of	myhello1.2	xlhmzch/hello:1.0.0	nick	Running	Running 54 seconds ago	
j6vvvs3qlx86	_ myhello1.2	xlhmzch/hello:1.0.1	nick	Shutdown	Shutdown 55 seconds ago	
ojwk9ftouxcs	_ myhello1.2	xlhmzch/hello:1.0.0	nick	Shutdown	Shutdown 4 minutes ago	
luvkj8zw226o	myhello1.3	xlhmzch/hello:1.0.0	ubuntu20qhtest	Running	Running 56 seconds ago	
9nhd2pdpkeyn	_ myhello1.3	xlhmzch/hello:1.0.1	nick	Shutdown	Shutdown 57 seconds ago	
lik49x2eksix	_ myhello1.3	xlhmzch/hello:1.0.0	nick	Shutdown	Shutdown 3 minutes ago	
rwln1j28nscj	myhello1.4	xlhmzch/hello:1.0.0	nick	Running	Running 56 seconds ago	
52fb056astif	_ myhello1.4	xlhmzch/hello:1.0.1	ubuntu20qhtest	Shutdown	Shutdown 57 seconds ago	
6lrbzd3h3ay	_ myhello1.4	xlhmzch/hello:1.0.0	ubuntu20qhtest	Shutdown	Shutdown 3 minutes ago	
vizohwjcnodox	myhello1.5	xlhmzch/hello:1.0.0	ubuntu20	Running	Running about a minute ago	
n0tkgyzbmqnn	_ myhello1.5	xlhmzch/hello:1.0.1	ubuntu20	Shutdown	Shutdown about a minute ago	
6oqs3pxoh5ni	_ myhello1.5	xlhmzch/hello:1.0.0	ubuntu20	Shutdown	Shutdown 4 minutes ago	
v8nds3raa17l	myhello1.6	xlhmzch/hello:1.0.0	nick	Running	Running 52 seconds ago	
w3ufi0gtd6r	_ myhello1.6	xlhmzch/hello:1.0.1	nick	Shutdown	Shutdown 53 seconds ago	
x2350rqfakmh	_ myhello1.6	xlhmzch/hello:1.0.0	nick	Shutdown	Shutdown 4 minutes ago	
bbyzej1rd8in	myhello1.7	xlhmzch/hello:1.0.0	nick	Running	Running about a minute ago	
ihpac219clys	_ myhello1.7	xlhmzch/hello:1.0.1	ubuntu20qhtest	Shutdown	Shutdown about a minute ago	
wifli20pgb2u	_ myhello1.7	xlhmzch/hello:1.0.0	ubuntu20qhtest	Shutdown	Shutdown 4 minutes ago	

结合docker-compose.yml部署

部署应用

1. 常用命令

```
# 部署或更新 stack
deploy      Deploy a new stack or update an existing stack
# 查看 stack 列表
ls          List stacks
# 查看 stack 的任务列表
ps          List the tasks in the stack
# 删除 stack
rm          Remove one or more stacks
# 查看stack 中的服务列表
services    List the services in the stack
```

2. docker-compose.yml 文件

```
version: "3.7"
services:
  myhello2:
    image: xlhmzch/hello:1.0.0
    ports:
      - "83:80"
    depends_on:
      - redis
    deploy:
      mode: replicated
      replicas: 20
      endpoint_mode: vip
      rollback_config:
        parallelism: 2
        delay: 10s
        monitor: 10s
        max_failure_ratio: 0.2
      update_config:
        parallelism: 2
        delay: 5s
        failure_action: continue
  redis:
```



```
image: redis:alpine
deploy:
  mode: replicated
  replicas: 6
  endpoint_mode: dnsrr
  labels:
    description: "This redis service label"
  resources:
    limits:
      cpus: '0.50'
      memory: 50M
    reservations:
      cpus: '0.25'
      memory: 20M
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s
```

3. 部署stack

```
docker stack deploy -c docker-compose.yml mystack
```

4. 查看服务详情

```
# 查看并打印友好的详细信息
docker service inspect --pretty mystack_myhello2
docker service inspect --pretty mystack_redis
```

语法说明

endpoint_mode: 访问集群服务的方式。

```
endpoint_mode: vip
# Docker 集群服务一个对外的虚拟 ip。所有的请求都会通过这个虚拟 ip 到达集群服务内部的机器。
endpoint_mode: dnsrr
# DNS 轮询（DNSRR）。所有的请求会自动轮询获取到集群 ip 列表中的一个 ip 地址。
```

labels: 在服务上设置标签。可以用容器上的 labels（跟 deploy 同级的配置）覆盖 deploy 下的 labels。

mode: 指定服务提供的模式。

- **replicated**: 复制服务，复制指定服务到集群的机器上。
- **global**: 全局服务，服务将部署至集群的每个节点。
- **replicas: mode** 为 replicated 时，需要使用此参数配置具体运行的节点数量。

resources: 配置服务器资源使用的限制，例如上例子，配置 redis 集群运行需要的 cpu 的百分比和内存的占用。避免占用资源过高出现异常。

restart_policy: 配置如何在退出容器时重新启动容器。

- condition: 可选 none, on-failure 或者 any（默认值: any）。
- delay: 设置多久之后重启（默认值: 0）。

- max_attempts: 尝试重新启动容器的次数, 超出次数, 则不再尝试 (默认值: 一直重试)。
- window: 设置容器重启超时时间 (默认值: 0)。

rollback_config: 配置在更新失败的情况下应如何回滚服务。

- parallelism: 一次要回滚的容器数。如果设置为0, 则所有容器将同时回滚。
- delay: 每个容器组回滚之间等待的时间 (默认为0s)。
- failure_action: 如果回滚失败, 该怎么办。其中一个 continue 或者 pause (默认pause)。
- monitor: 每个容器更新后, 持续观察是否失败的时间 (ns|us|ms|s|m|h) (默认为0s)。
- max_failure_ratio: 在回滚期间可以容忍的故障率 (默认为0)。
- order: 回滚期间的操作顺序。其中一个 stop-first (串行回滚), 或者 start-first (并行回滚) (默认 stop-first)。

update_config: 配置应如何更新服务, 对于配置滚动更新很有用。

- parallelism: 一次更新的容器数。
- delay: 在更新一组容器之间等待的时间。
- failure_action: 如果更新失败, 该怎么办。其中一个 continue, rollback 或者 pause (默认: pause)。
- monitor: 每个容器更新后, 持续观察是否失败的时间 (ns|us|ms|s|m|h) (默认为0s)。
- max_failure_ratio: 在更新过程中可以容忍的故障率。
- order: 回滚期间的操作顺序。其中一个 stop-first (串行回滚), 或者 start-first (并行回滚) (默认stop-first)。

注: 仅支持 V3.4 及更高版本