一、基本概念

Docker 包括三个基本概念

镜像(Image)

容器(Container)

仓库(Repository)

理解了这三个概念，就理解了 Docker 的整个生命周期。

1.Docker 镜像

我们都知道，操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。而 Docker 镜像(Image)，就相当于是一个 root 文件系统。比如官方镜像 ubuntu:14.04 就包含了完整的一套 Ubuntu 14.04 最小系统的 root 文件系统。

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数(如匿名卷、环境变量、用户等)。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

2.Docker 容器

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

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的 命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。也因为这种隔离的特性，很多人初学 Docker 时常常会把容器和虚拟机搞混。

前面讲过镜像使用的是分层存储，容器也是如此。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为容器存储层。

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。

按照 Docker 最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用 数据卷(Volume)、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主(或网络存储)发生读写，其性能和稳定性更高。

数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器可以随意删除、重新 run，数据却不会丢失。

3.Docker Registry

镜像构建完成后，可以很容易的在当前宿主上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。

一个 Docker Registry 中可以包含多个仓库(Repository)；每个仓库可以包含多个标签(Tag)；每个标签对应一个镜像。

通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 <仓库名>:<标签> 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 latest 作为默认标签。

以 Ubuntu 镜像 为例，ubuntu 是仓库的名字，其内包含有不同的版本标签，如，14.04, 16.04。我们可以通过 ubuntu:14.04，或者 ubuntu:16.04 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 ubuntu，那将视为 ubuntu:latest。

仓库名经常以 两段式路径 形式出现，比如 jwilder/nginx-proxy，前者往往意味着 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对，取决于所使用的具体 Docker Registry 的软件或服务。

4.Docker Registry 公开服务

Docker Registry 公开服务是开放给用户使用、允许用户管理镜像的 Registry 服务。一般这类公开服务允许用户免费上传、下载公开的镜像，并可能提供收费服务供用户管理私有镜像。

最常使用的 Registry 公开服务是官方的 Docker Hub，这也是默认的 Registry，并拥有大量的高质量的官方镜像。除此以外，还有 CoreOS 的 Quay.io，CoreOS 相关的镜像存储在这里；Google 的 Google Container Registry，Kubernetes 的镜像使用的就是这个服务。

由于某些原因，在国内访问这些服务可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务(Registry Mirror)，这些镜像服务被称为加速器。常见的有 阿里云加速器、DaoCloud 加速器、灵雀云加速器等。使用加速器会直接从国内的地址下载 Docker Hub 的镜像，比直接从官方网站下载速度会提高很多。在后面的章节中会有进一步如何配置加速器的讲解。

国内也有一些云服务商提供类似于 Docker Hub 的公开服务。比如 时速云镜像仓库、网易云镜像服务、DaoCloud 镜像市场、阿里云镜像库等。

应用案例：网络问题获取镜像失败

解决方法：修改docker镜像源：

docker默认的源为国外官方源，下载速度较慢，可改为国内，加速

Docker 官方中国区：https://registry.docker-cn.com

网易：http://hub-mirror.c.163.com

中国科技大学：https://docker.mirrors.ustc.edu.cn

阿里云：https://pee6w651.mirror.aliyuncs.com

修改或新增 /etc/docker/daemon.json

# vim /etc/docker/daemon.json

{

"registry-mirrors": ["https://pee6w651.mirror.aliyuncs.com"]

}

重启Docker。

# systemctl restart docker.service

二、CentOS 安装 Docker CE

1.卸载旧版本

旧版本的 Docker 称为 docker 或者 docker-engine，使用以下命令卸载旧版本：

# yum remove docker docker-client docker-client-latest docker-common docker-latest docker-latest-logrotate docker-logrotate docker-selinux docker-engine-selinux docker-engine

2.安装 Docker CE

更新 yum 软件源缓存，并安装 docker-ce。

# yum install -y yum-utils device-mapper-persistent-data lvm2

# yum-config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

# yum makecache fast

# yum install docker-ce

使用脚本自动安装

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，CentOS 系统上可以使用这套脚本安装，另外可以通过 --mirror 选项使用国内源进行安装：

# curl -fsSL get.docker.com -o get-docker.sh

# sh get-docker.sh --mirror Aliyun

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker CE 的稳定(stable)版本安装在系统中。

3.启动 Docker CE

# systemctl enable docker

# systemctl start docker

建立 docker 用户组

默认情况下，docker 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 root 用户和 docker 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 root 用户。因此，更好地做法是将需要使用 docker 的用户加入 docker 用户组。

4.建立 docker 组：

# groupadd docker

将当前用户加入 docker 组：# usermod -aG docker $USER

退出当前终端并重新登录，进行如下测试。

5.测试 Docker 是否安装正确

$ docker run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

d1725b59e92d: Pull complete

Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788

Status: Downloaded newer image for hello-world:latest

若能正常输出以上信息，则说明安装成功。

参考文献：

https://www.jb51.net/article/177789.htm

三、Docker常用命令详解

1.检查安装 docker info

如果 Docker 没有安装，会提示command not found，如果 Docker 已经成功安装，则会有类似如下的提示：

[root@k8s-master ~]# docker info

Containers: 42

Running: 20

Paused: 0

Stopped: 22

Images: 31

Server Version: 18.06.1-ce

Storage Driver: overlay2

Backing Filesystem: xfs

Supports d\_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: systemd

Plugins:

Volume: local

Network: bridge host macvlan null overlay

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

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: 468a545b9edcd5932818eb9de8e72413e616e86e

runc version: 69663f0bd4b60df09991c08812a60108003fa340

init version: fec3683

Security Options:

seccomp

Profile: default

Kernel Version: 3.10.0-1062.4.1.el7.x86\_64

Operating System: CentOS Linux 7 (Core)

OSType: linux

Architecture: x86\_64

CPUs: 4

Total Memory: 1.777GiB

2. 查看本地镜像 docker images或者docker image ls

镜像是Docker生命周期中的"构建"部分，可以用来创建 Docker 容器。

[root@k8s-master ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

fastop/kafka 2.2.0 b6e476335a87 2 weeks ago 368MB

nginx latest 2073e0bcb60e 2 weeks ago 127MB

[root@k8s-master ~]# docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

fastop/kafka 2.2.0 b6e476335a87 2 weeks ago 368MB

nginx latest 2073e0bcb60e 2 weeks ago 127MB

3. 下载镜像 docker pull <image name>

[root@localhost ~]# docker pull nginx

Using default tag: latest

latest: Pulling from library/nginx

afb6ec6fdc1c: Pull complete

b90c53a0b692: Pull complete

11fa52a0fdc0: Pull complete

Digest: sha256:30dfa439718a17baafefadf16c5e7c9d0a1cde97b4fd84f63b69e13513be7097

Status: Downloaded newer image for nginx:latest

docker.io/library/nginx:latest

4. 运行镜像 docker run <image name>

-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高端口

-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；

-m：设置容器使用内存最大值；

应用实例：

(1).使用docker镜像nginx:latest以后台模式启动一个容器,并将容器命名为mynginx。

docker run --name mynginx -d nginx:latest

[root@localhost ~]# docker run --name mynginx -d nginx:latest

96f9bcfcd57185ccef7dd824644ad48c31d600f9b9c51d31fa6294d5f97cc6c4

(2).使用镜像nginx:latest以交互模式启动一个容器,在容器内执行/bin/bash命令。

[root@localhost ~]# docker run -it nginx:latest /bin/bash

root@269b5399c891:/#

(3).使用镜像nginx:latest以交互模式，后台启动一个容器

[root@localhost ~]# docker run -it -d nginx:latest /bin/bash

f474fc58eea8a90f61e430d0c047de451f816c0856c077eda4b4d57675844b48

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

f474fc58eea8 nginx:latest "/bin/bash" About a minute ago Up About a minute 80/tcp thirsty\_rhodes

[root@localhost ~]# docker exec -it f474fc58eea8 /bin/bash

root@f474fc58eea8:/#

(4).使用镜像nginx:latest以后台模式启动一个容器,并将容器的80端口映射到主机随机端口。

docker run -P -d nginx:latest

[root@localhost ~]# docker run -P -d nginx:latest

254b16d716f837a2b31a0b02422dc926987ee2673b4b3d7e5ca5201c44e9365c

[root@localhost ~]# netstat -tulnp

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 966/sshd

tcp 0 0 127.0.0.1:25 0.0.0.0:\* LISTEN 1178/master

tcp6 0 0 :::22 :::\* LISTEN 966/sshd

tcp6 0 0 ::1:25 :::\* LISTEN 1178/master

tcp6 0 0 :::32768 :::\* LISTEN 15351/docker-proxy

(5).使用镜像 nginx:latest，以后台模式启动一个容器,将容器的 80 端口映射到主机的 80 端口,

主机的目录 /data 映射到容器的 /data。

docker run -p 80:80 -v /data:/data -d nginx:latest

[root@localhost ~]# docker run -p 80:80 -v /data:/data -d nginx:latest

855713508d05e994f60cee110a4e5067af86d0855059774ecb42ef034ddf80a2

[root@localhost ~]# netstat -tulnp

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 966/sshd

tcp 0 0 127.0.0.1:25 0.0.0.0:\* LISTEN 1178/master

tcp6 0 0 :::80 :::\* LISTEN 15494/docker-proxy

tcp6 0 0 :::22 :::\* LISTEN 966/sshd

tcp6 0 0 ::1:25 :::\* LISTEN 1178/master

5.删除镜像 docker rmi

如果要删除本地的镜像，可以使用 docker rmi 命令，其格式为：

docker rmi [选项] <镜像1> [<镜像2> ...]

注意 docker rm 命令是删除容器，不要混淆。

用 ID、镜像名、摘要删除镜像

其中，<镜像> 可以是 镜像短 ID、镜像长 ID、镜像名 或者 镜像摘要。

docker rmi <ID>/<name>

[[root@localhost ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

nginx latest 9beeba249f3e 12 days ago 127MB

ubuntu latest 1d622ef86b13 4 weeks ago 73.9MB

[root@localhost ~]#

[root@localhost ~]# docker rmi ubuntu

Untagged: ubuntu:latest

Untagged: ubuntu@sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7

Deleted: sha256:1d622ef86b138c7e96d4f797bf5e4baca3249f030c575b9337638594f2b63f01

Deleted: sha256:279e836b58d9996b5715e82a97b024563f2b175e86a53176846684f0717661c3

Deleted: sha256:39865913f677c50ea236b68d81560d8fefe491661ce6e668fd331b4b680b1d47

Deleted: sha256:cac81188485e011e56459f1d9fc9936625a1b62cacdb4fcd3526e5f32e280387

Deleted: sha256:7789f1a3d4e9258fbe5469a8d657deb6aba168d86967063e9b80ac3e1154333f

比如我们有这么一些镜像：

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

centos latest 0584b3d2cf6d 3 weeks ago 196.5 MB

redis alpine 501ad78535f0 3 weeks ago 21.03 MB

docker latest cf693ec9b5c7 3 weeks ago 105.1 MB

nginx latest e43d811ce2f4 5 weeks ago 181.5 MB

我们可以用镜像的完整 ID，也称为 长 ID，来删除镜像。使用脚本的时候可能会用长 ID，但是人工输入就太累了，所以更多的时候是用 短 ID 来删除镜像。docker images 默认列出的就已经是短 ID 了，一般取前3个字符以上，只要足够区分于别的镜像就可以了。

比如这里，如果我们要删除 redis:alpine 镜像，可以执行：

$ docker rmi 501

Untagged: redis:alpine

Untagged: redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeea521a86d

Deleted: sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7

Deleted: sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b

Deleted: sha256:32770d1dcf835f192cafd6b9263b7b597a1778a403a109e2cc2ee866f74adf23

Deleted: sha256:127227698ad74a5846ff5153475e03439d96d4b1c7f2a449c7a826ef74a2d2fa

Deleted: sha256:1333ecc582459bac54e1437335c0816bc17634e131ea0cc48daa27d32c75eab3

Deleted: sha256:4fc455b921edf9c4aea207c51ab39b10b06540c8b4825ba57b3feed1668fa7c7

我们也可以用镜像名，也就是 <仓库名>:<标签>，来删除镜像。

$ docker rmi centos

Untagged: centos:latest

Untagged: centos@sha256:b2f9d1c0ff5f87a4743104d099a3d561002ac500db1b9bfa02a783a46e0d366c

Deleted: sha256:0584b3d2cf6d235ee310cf14b54667d889887b838d3f3d3033acd70fc3c48b8a

Deleted: sha256:97ca462ad9eeae25941546209454496e1d66749d53dfa2ee32bf1faabd239d38

用 docker images 命令来配合

像其它可以承接多个实体的命令一样，可以使用 docker images -q 来配合使用 docker rmi，这样可以成批的删除希望删除的镜像。比如之前我们介绍过的，删除虚悬镜像的指令是：

$ docker rmi $(docker images -q -f dangling=true)

我们在“镜像列表”章节介绍过很多过滤镜像列表的方式都可以拿过来使用。

比如，我们需要删除所有仓库名为 redis 的镜像：

$ docker rmi $(docker images -q redis)

或者删除所有在 mongo:3.2 之前的镜像：

$ docker rmi $(docker images -q -f before=mongo:3.2)

充分利用你的想象力和 Linux 命令行的强大，你可以完成很多非常赞的功能

ERROR：镜像删除失败

[root@k8s-master ~]# docker rmi ccc6e87d482b

Error response from daemon: conflict: unable to delete ccc6e87d482b (cannot be forced) - image has dependent child images

原因：有另外的 image FROM 了这个 image，可以使用下面的命令列出所有在指定 image 之后创建的 image 的父 image

方案：先查询依赖

docker image inspect --format='{{.RepoTags}} {{.Id}} {{.Parent}}' $(docker image ls -q --filter since=XXX) # XXX指镜像ID

[root@k8s-master ~]# docker image inspect --format='{{.RepoTags}} {{.Id}} {{.Parent}}' $(docker image ls -q --filter since=ccc6e87d482b)

[fastop/kafka:2.2.0] sha256:b6e476335a871e749a91306b3e9a5b6d5d8a4ef88142e48947f787181f7fa001 sha256:d00343c02bc73b0ac154f82ee460458d5e8f4ea1619e80cd80018d2bb29df304

[root@k8s-master ~]# docker rmi b6e476335a

Untagged: fastop/kafka:2.2.0

Deleted: sha256:b6e476335a871e749a91306b3e9a5b6d5d8a4ef88142e48947f787181f7fa001

Deleted: sha256:f1b3e8b8dad050f8414ae35790434bbdc1c51748eeb141c04d54016d8706b960

[root@k8s-master ~]# docker rmi ccc6e87d482b

6.查看本地容器 docker ps

docker ps 查看正在运行的容器

[root@k8s-master ~]# docker ps | grep ubuntu

f02023cb856b ubuntu "/bin/bash" 3 seconds ago Up 2 seconds affectionate\_kilby

查看所有的容器docker ps -a ，-q 返回 id

[root@k8s-master ~]# docker ps -a -q

2ea47493720d

853b1422b116

1af36327ab65

8e904580b22f

5bb39d808030

f6e7a9eef7f6

7. 查询输出docker logs <ID>/<name>

查询容器的输出内容：

[root@k8s-master ~]# docker ps | grep dashboard

853b1422b116 kubernetesui/dashboard "/dashboard --insecu…" 2 hours ago Up 2 hours k8s\_kubernetes-dashboard\_kubernetes-dashboard-b65488c4-cwcq5\_kubernetes-dashboard\_44767c7f-70b0-427c-8aeb-4a3ea7397e11\_5

[root@k8s-master ~]# docker logs 853b1422b116

2020/02/20 08:03:13 Starting overwatch

2020/02/20 08:03:13 Using namespace: kubernetes-dashboard

2020/02/20 08:03:13 Using in-cluster config to connect to apiserver

2020/02/20 08:03:13 Using secret token for csrf signing

2020/02/20 08:03:13 Initializing csrf token from kubernetes-dashboard-csrf secret

2020/02/20 08:03:13 Successful initial request to the apiserver, version: v1.17.0

查看日志： docker logs <container\_id>

docker logs -f <container\_id>动态的查看日志，类似查看tomcat的日志一样

8. 启动容器docker start

docker start <ID>/<name>

[root@k8s-master ~]# docker ps -a |grep ubuntu

f02023cb856b ubuntu "/bin/bash" About an hour ago Exited (0) 14 seconds ago affectionate\_kilby

[root@k8s-master ~]# docker start f02023cb856b

f02023cb856b

9. 停止容器docker stop

可以使用 docker stop 来终止一个运行中的容器。

此外，当Docker容器中指定的应用终结时，容器也自动终止。 例如对于上一章节中只启动了一个终端的容器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 docker ps -a 命令看到。例如：

# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ba267838cc1b ubuntu:14.04 "/bin/bash" 30 minutes ago Exited (0) About a minute ago trusting\_newton

98e5efa7d997 training/webapp:latest "python app.py" About an hour ago Exited (0) 34 minutes ago backstabbing\_pike

处于终止状态的容器，可以通过 docker start 命令来重新启动。

10. 重启容器docker restart

docker restart <ID>/<name>

[root@k8s-master ~]# docker restart f02023cb856b

f02023cb856b

docker restart 命令会将一个运行态的容器终止，然后再重新启动它。

11.删除容器 docker rm

(1).先停止容器，然后通过docker rm删除

docker rm <ID>/<name>

[root@k8s-master ~]# docker stop f02023cb856b

f02023cb856b

[root@k8s-master ~]# docker rm f02023cb856b

f02023cb856b

(2).如果要删除一个运行中的容器，可以添加 -f 参数。Docker 会发送 SIGKILL 信号给容器。

[root@k8s-node1 ~]# docker run -it -d ccc6e87d482b

ecc0176ccabbfc2fccbd82a0e001562850bca5b5337eca15257d74838aab377a

[root@k8s-node1 ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ecc0176ccabb ccc6e87d482b "/bin/bash" 2 minutes ago Up 2 minutes awesome\_wing

[root@k8s-node1 ~]# docker rm -f ecc0176ccabb

ecc0176ccabb

(3).清理所有处于终止状态的容器

用 docker ps -a 命令可以查看所有已经创建的包括终止状态的容器，如果数量太多要一个个删除可能会很麻烦，用 docker rm $(docker ps -a -q) 可以全部清理掉。

注：这个命令其实会试图删除所有的包括还在运行中的容器，但正如上面提过的 docker rm 默认并不会删除运行中的容器。

12.进入容器

当我们使用 -d参数运行了一个Container的时候，有时候我们需要进入这个容器进行一些操作。例如有这样的一个情况，我们运行了一个app在一个容器里，我们想进入容器看看，这个app运行的状态，查看log。那们如何进入呢？其实有很多种方法，这里介绍两种。

a.docker attach <ID>/<name>

[root@k8s-master ~]# docker attach 39ea7fc52032

root@39ea7fc52032:/#

b.docker exec <ID>/<name> /bin/bash

[root@k8s-master ~]# docker exec -it 39ea7fc52032 /bin/bash

root@39ea7fc52032:/#

注：docker attach 和 docker exec区别

docker attach：

让用户可以进入Container查看输出等等操作，但是并不会另外启动一个进程! 如果你用exit来退出，同时这个信号会kill。 Container(默认情况)

docker exec：

会启动另外一个进程来进入Container，这里的操作是在这个进程下的。如果你用exit来退出，不会kill 原来的Container。

应用实例：

[root@k8s-node1 ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

3690802617f9 ccc6e87d482b "/bin/bash" 5 minutes ago Up 42 seconds competent\_dubinsky

[root@k8s-node1 ~]# docker attach 3690802617f9

root@3690802617f9:/# exit

exit

[root@k8s-node1 ~]# docker attach 3690802617f9

You cannot attach to a stopped container, start it first

# 说明通过docker attach命令进入container之后，通过exit退出容器会导致容器停止运行。

[root@k8s-node1 ~]# docker start 3690802617f9

3690802617f9

[root@k8s-node1 ~]# docker exec -it 3690802617f9 /bin/bash

root@3690802617f9:/# exit

exit

[root@k8s-node1 ~]# docker exec -it 3690802617f9 /bin/bash

root@3690802617f9:/# # 说明通过docker exec命令进入container之后，通过exit退出容器不会导致容器停止运行。

13.退出容器Ctrl+P+Q

进入docker容器后如果退出容器，容器就会变成Exited的状态，那么如何退出容器让容器不关闭呢？

如果要正常退出不关闭容器，请按Ctrl+P+Q进行退出容器，这一点很重要，请牢记！

[root@k8s-master ~]# docker exec -it 39ea7fc52032 /bin/bash

Ctrl+P+Q

root@39ea7fc52032:/# read escape sequence

[root@k8s-master ~]#

14.镜像的导入与导出load & save

使用 load 从 stdin 导入一个 tar 格式的镜像或者仓库，然后用 save 将 tar 镜像输出到 stdout。

docker save -o <output file> <image>

或者

docker save <image> > <output file>

docker load -i <file name>或者docker load < <file name>

这两个命令常用于多节点部署。

[root@k8s-master ~]# docker save -o ubuntu.tar ubuntu

[root@k8s-master ~]# ll

-rw------- 1 root root 66601984 Feb 21 09:51 ubuntu.tar

[root@k8s-node2 ~]# docker images |grep ubuntu

[root@k8s-node2 ~]# docker load -i ubuntu.tar

43c67172d1d1: Loading layer [==================================================>] 65.57MB/65.57MB

21ec61b65b20: Loading layer [==================================================>] 991.2kB/991.2kB

1d0dfb259f6a: Loading layer [==================================================>] 15.87kB/15.87kB

f55aa0bd26b8: Loading layer [==================================================>] 3.072kB/3.072kB

Loaded image: ubuntu:latest

[root@k8s-node2 ~]# docker images |grep ubuntu

ubuntu latest ccc6e87d482b 5 weeks ago 64.2MB

15.容器导入import

用于导入 URL / 文件，从本地导入需要 - 参数。docker import [OPTIONS] URL|- [REPOSITORY[:TAG]]、URL/-二选一。

[root@k8s-master ~]# docker import http://mirrors.ustc.edu.cn/openvz/template/precreated/suse-13.1-x86-minimal.tar.gz suse:minimal

Downloading from http://mirrors.ustc.edu.cn/openvz/template/precreated/suse-13.1-x86-minimal.tar.gz

sha256:2b56da0856b74c04608397c3b0b1e1dcda22c50b168c8a5f360a602321a0f25bB/60.56MB

#这里使用的是ustc镜像源

[root@k8s-master ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

suse minimal 2b56da0856b7 5 minutes ago 156MB

导入本地镜像：

cat suse-13.1-x86-minimal.tar.gz |docker import - suse:minmal

应用实例A：

[root@k8s-node2 ~]# docker images | grep suse

[root@k8s-node2 ~]# ll

-rw-r--r-- 1 root root 60561554 Feb 21 10:13 suse-13.1-x86-minimal.tar.gz

[root@k8s-node2 ~]# cat suse-13.1-x86-minimal.tar.gz |docker import - suse:minmal

sha256:9387cc6bdc88d6d1b9712dbe0195441d0db8c89f94ec14e1306371a3a0de24a0

[root@k8s-node2 ~]#

[root@k8s-node2 ~]# docker images | grep suse

suse minmal 9387cc6bdc88 6 seconds ago 156MB

应用实例B：

[root@k8s-node2 ~]# docker images | grep ubuntu

[root@k8s-node2 ~]# ll

-rw-r--r-- 1 root root 66601984 Feb 21 09:51 ubuntu.tar

[root@k8s-node2 ~]# cat ubuntu.tar | docker import - ubuntu

sha256:b2be15ade1f3561f39a769e6abec4e889f1d52e69ceedf2a166609107a3443ea

[root@k8s-node2 ~]# docker images | grep ubuntu

ubuntu latest b2be15ade1f3 2 minutes ago 66.6MB

16.容器从A导出export作为镜像，然后将此镜像导入到B

docker export -o suse1.tar <ContainerID>

docker export <ContainerID> > suse2.tar

和 import 相反，export 将容器导出成 tar 压缩包。

[root@k8s-master ~]# docker run -i -t -d suse:minimal /bin/bash

5c5d8dcaf22d5c66d5456814ef2f50aa9b5ba2e3629fd518978f1bedffbf7632

[root@k8s-master ~]# docker ps |grep suse

5c5d8dcaf22d suse:minimal "/bin/bash" 20 seconds ago Up 18 seconds fervent\_leavitt

[root@k8s-master ~]# docker export -o suse1.tar 5c5d8dcaf22d

或者

[root@k8s-master ~]# docker export 5c5d8dcaf22d > suse2.tar

[root@k8s-node2 ~]# docker images | grep suse

suse2 latest b48f3e943806 3 seconds ago 156MB

suse2.tar latest 7c73f60eac80 15 minutes ago 156MB

17.docker build 构建镜像

该命令用于构建镜像，与 Dockerfile 配合使用。

常用参数

-c：控制 CPU 使用

-f：指定 Dockerfile 文件的路径和名称

-m：设置构建内存上限

-q：不显示构建过程的信息

-t：为构建的镜像打上标签

(1).构建镜像

好了，让我们再回到之前定制的 nginx 镜像的 Dockerfile 来。现在我们明白了这个 Dockerfile 的内容，那么让我们来构建这个镜像吧。

还以之前定制 nginx 镜像为例，这次我们使用 Dockerfile 来定制。

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

[root@k8s-node1 ~]# vim Dockerfile

FROM nginx

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。

在 Dockerfile 文件所在目录执行：

[root@k8s-node1 ~]# docker build -t nginx:v3 .

Sending build context to Docker daemon 39.94kB

Step 1/2 : FROM nginx

---> a1523e859360

Step 2/2 : RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

---> Running in 0ddbda356e61

Removing intermediate container 0ddbda356e61

---> ced026f95462

Successfully built ced026f95462

Successfully tagged nginx:v3

从命令的输出结果中，我们可以清晰的看到镜像的构建过程。在 Step 2 中，如同我们之前所说的那样，RUN 指令启动了一个容器 0ddbda356e61，执行了所要求的命令，并最后提交了这一层 ced026f95462，随后删除了所用到的这个intermediate container(中间容器) ：0ddbda356e61。

这里我们使用了 docker build 命令进行镜像构建。其格式为：

docker build [选项] <上下文路径/URL/->

在这里我们指定了最终镜像的名称 -t nginx:v3，构建成功后，我们可以像之前运行 nginx:v2 那样来运行这个镜像，其结果会和 nginx:v2 一样。

(2).镜像构建上下文(Context)

如果注意，会看到 docker build 命令最后有一个 .。. 表示当前目录，而 Dockerfile 就在当前目录，因此不少初学者以为这个路径是在指定 Dockerfile 所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定上下文路径。那么什么是上下文呢？

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种C/S的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。

如果在 Dockerfile 中这么写：COPY ./package.json /app/

这并不是要复制执行 docker build 命令所在的目录下的 package.json，也不是复制 Dockerfile 所在目录下的 package.json，而是复制 上下文(context) 目录下的 package.json。

因此，COPY 这类指令中的源文件的路径都是相对路径。这也是初学者经常会问的为什么 COPY ../package.json /app 或者 COPY /opt/xxxx /app 无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

现在就可以理解刚才的命令 docker build -t nginx:v3 . 中的这个 .，实际上是在指定上下文的目录，docker build 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像。

如果观察 docker build 输出，我们其实已经看到了这个发送上下文的过程：

$ docker build -t nginx:v3 .

Sending build context to Docker daemon 2.048 kB

...

理解构建上下文对于镜像构建是很重要的，避免犯一些不应该的错误。比如有些初学者在发现 COPY /opt/xxxx /app 不工作后，于是干脆将 Dockerfile 放到了硬盘根目录去构建，结果发现 docker build 执行后，在发送一个几十 GB 的东西，极为缓慢而且很容易构建失败。那是因为这种做法是在让 docker build 打包整个硬盘，这显然是使用错误。

一般来说，应该会将 Dockerfile 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 .gitignore 一样的语法写一个 .dockerignore，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

那么为什么会有人误以为 . 是指定 Dockerfile 所在目录呢？这是因为在默认情况下，如果不额外指定 Dockerfile 的话，会将上下文目录(.)下的名为 Dockerfile 的文件作为 Dockerfile。

这只是默认行为，实际上 Dockerfile 的文件名并不要求必须为 Dockerfile，而且并不要求必须位于上下文目录中，比如可以用 -f ../Dockerfile.php 参数指定某个文件作为 Dockerfile。

当然，一般大家习惯性的会使用默认的文件名 Dockerfile，以及会将其置于镜像构建上下文目录中。

(3).应用实例：

①.构建一个镜像，命名为 lsu，指定当前目录为上下文目录，将Dockerfile文件放在当前目录中，暨上下文目录中：

[root@k8s-node1 ~]# vim Dockerfile

FROM ubuntu

CMD ["-l"]

ENTRYPOINT ["ls"]

# docker build -t "lsu" .

[root@k8s-node1 ~]# docker build -t "lsu" .

Sending build context to Docker daemon 35.84kB

Successfully built 5899b788ad7a

Successfully tagged lsu:latest

②.构建一个镜像，命名为 lsc，同时指定上下文目录为/usr/local/Docker/，并将Dockerfile文件放在此路径下：

[root@k8s-node1 ~]# vim /usr/local/Dockerfile

FROM centos

CMD ["-l"]

ENTRYPOINT ["ls"]

# docker build -t "lsc" /usr/local/Docker/

[root@k8s-node1 ~]# docker build -t "lsc" /usr/local/Docker/

Sending build context to Docker daemon 18.43kB

Successfully built 26443149efa2

Successfully tagged lsc:latest

③.构建一个镜像，命名为 zhuhaifeng/lsu，指定版本号为v2，上下文目录为/usr/local/Docker/，并将Dockerfile放在此路径下：

[root@k8s-node1 ~]# docker build -t zhuhaifeng/lsu:v2 /usr/local/Docker/

Sending build context to Docker daemon 18.43kB

Successfully built 749079b7b312

Successfully tagged zhuhaifeng/lsu:v2

④.构建一个镜像，命名为 free，通过-f 指定Dockerfile文件位置为/usr/local/MemDockerfile，上下文路径为/usr/local/Docker/

docker build -t "free" -f /usr/local/MemDockerfile /usr/local/Docker/

[root@k8s-node1 ~]# docker build -t "free" -f /usr/local/MemDockerfile /usr/local/Docker/

Sending build context to Docker daemon 36.91kB

Successfully built 50a41a721165

Successfully tagged free:latest

[root@k8s-node1 ~]# docker images |grep 50a41a721165

free latest 50a41a721165 2 minutes ago 64.2MB

四、什么是Dockerfile

Dockerfile实际上是由一行行命令组成的,让用户可以方便的创建自定义镜像。

镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

还以之前定制 nginx 镜像为例，这次我们使用 Dockerfile 来定制。

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

# cd mynginx

# vim Dockerfile

FROM nginx

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。

1.Dockerfile大体由四部分组成：

指明基础镜像指令FROM

维护者信息指令MAINTAINER

镜像操作指令RUN、EVN、ADD和WORKDIR等

容器启动时的执行指令CMD、ENTRYPOINT和USER等

下边就是一个Dockerfile的例子

FROM python:2.7

MAINTAINER zhuhaifeng <13956072050@163.com>

COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt

EXPOSE 5000

ENTRYPOINT ["python"]

CMD ["app.py"]

a. 从dockerhub上pull下python 2.7的基础镜像。

b. 维护者的信息

c. copy当前目录到容器中的 /app目录下 ，复制本地主机的<src>(Dockerfile所在目录的相对路径)到容器里<dest>

d. 指定工作路径为/app

e. 安装依赖

f. 暴露5000端口

g. 启动app

这个例子是启动一个python flask app的Dockerfile(flask是python的一个轻量的web框架)。

2.dockerfile里常用的指令介绍。

(1).FROM

用于指定基础的images

格式为 FROM <image> or FORM <image>:<tag>

所有的Dockerfile都以 FROM 开头，FROM 命令指明 Dockerfile 所创建的镜像文件以什么镜像为基础，FROM 以后的所有指令都会在 FROM 的基础上进行创建镜像；可以在同一个 Dockerfile 中多次使用 FROM 命令用于创建多个镜像。

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 FROM 就是指定基础镜像，因此一个 Dockerfile 中 FROM 是必备的指令，并且必须是第一条指令。

在 Docker Hub1 上有非常多的高质量的官方镜像， 有可以直接拿来使用的服务类的镜像，如 nginx、redis、mongo、mysql、httpd、php、tomcat 等； 也有一些方便开发、构建、运行各种语言应用的镜像，如 node、openjdk、python、ruby、golang 等。 可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。 如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 ubuntu、debian、centos、fedora、alpine 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

(2).MAINTAINER

格式为 MAINTAINER <name> 用于指定镜像创建者和联系方式。

MAINTAINER zhuhaifeng <13956072050@163.com>

(3).RUN

格式为 RUN <command>

用于容器内部执行命令。每个 RUN 命令相当于在原有的镜像基础上添加了一个改动层，原有的镜像不会有变化。

其格式有两种：

shell 格式：RUN <命令>，就像直接在命令行中输入的命令一样。刚才写的 Dockrfile 中的 RUN 指令就是这种格式。

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

exec 格式：RUN ["可执行文件", "参数1", "参数2"]，这更像是函数调用中的格式。

既然 RUN 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 RUN 呢？

比如这样：

FROM debian:jessie

RUN apt-get update

RUN apt-get install -y gcc libc6-dev make

RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"

RUN mkdir -p /usr/src/redis

RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1

RUN make -C /usr/src/redis

RUN make -C /usr/src/redis install

之前说过，Dockerfile 中每一个指令都会建立一层，RUN 也不例外。每一个 RUN 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，commit 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。 这是很多初学 Docker 的人常犯的一个错误。

Union FS 是有最大层数限制的，比如 AUFS，曾经是最大不得超过 42 层，现在是不得超过 127 层。

上面的 Dockerfile 正确的写法应该是这样：

FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \

&& apt-get update \

&& apt-get install -y $buildDeps \

&& wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \

&& mkdir -p /usr/src/redis \

&& tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \

&& make -C /usr/src/redis \

&& make -C /usr/src/redis install \

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

&& rm redis.tar.gz \

&& rm -r /usr/src/redis \

&& apt-get purge -y --auto-remove $buildDeps

首先，之前所有的命令只有一个目的，就是编译、安装 redis 可执行文件。因此没有必要建立很多层，这只是一层的事情。因此，这里没有使用很多个 RUN 对一一对应不同的命令，而是仅仅使用一个 RUN 指令，并使用 && 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。在撰写 Dockerfile 的时候，要经常提醒自己，这并不是在写 Shell 脚本，而是在定义每一层该如何构建。

并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 \ 的命令换行方式，以及行首 # 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 apt 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉。

很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

(4).COPY

格式：

COPY <源路径>... <目标路径>

COPY ["<源路径1>",... "<目标路径>"]

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。

COPY 指令将从构建上下文目录 <源路径>中的文件或目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

COPY package.json /usr/src/app/

<源路径> 可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则，如：

COPY hom\* /mydir/

COPY hom?.txt /mydir/

<目标路径> 可以是容器内的绝对路径，也可以是相对于工作目录的相对路径(工作目录可以用 WORKDIR 指令来指定)。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。

此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候。

(5).ADD

格式为 ADD<src><dest> 将主机文件复制到容器中

该命令将复制指定的 <src> 到容器中的 <dest>。其中 <src> 可以是Dockerfile所在目录的一个相对路径，可以是文件或目录的路径，也可以是一个URL，还可以是一个 tar 文件(自动解压为目录)。

ADD /path/to/sourcefile/in/host /path/to/targetfile/in/container

ADD 指令和 COPY 的格式和性质基本一致。但是在 COPY 基础上增加了一些功能。

比如 <源路径> 可以是一个 URL，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 <目标路径> 去。下载后的文件权限自动设置为 600，如果这并不是想要的权限，那么还需要增加额外的一层 RUN 进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或者 curl 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果 <源路径> 为一个 tar 压缩文件的话，压缩格式为 gzip, bzip2 以及 xz 的情况下，ADD 指令将会自动解压缩这个压缩文件到 <目标路径> 去。但在某些情况下，如果我们真的是希望复制个压缩文件进去，而不解压缩，这时就不可以使用 ADD 命令了。

在 Docker 官方的最佳实践文档中要求，尽可能的使用 COPY，因为 COPY 的语义很明确，就是复制文件而已，而 ADD 则包含了更复杂的功能，其行为也不一定很清晰。最适合使用 ADD 的场合，就是所提及的需要自动解压缩的场合。

另外需要注意的是，ADD 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。

因此在 COPY 和 ADD 指令中选择的时候，可以遵循这样的原则，所有的文件复制均使用 COPY 指令，仅在需要自动解压缩的场合使用 ADD。

(6).CMD

CMD 命令有三种格式：

shell 格式：CMD <命令>

exec 格式：CMD ["可执行文件", "参数1", "参数2"...]

参数列表格式：CMD ["参数1", "参数2"...]。在指定了 ENTRYPOINT 指令后，用 CMD 指定具体的参数。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如，ubuntu 镜像默认的 CMD 是 /bin/bash，如果我们直接 docker run -it ubuntu 的话，会直接进入 bash。我们也可以在运行时指定运行别的命令，如 docker run -it ubuntu cat /etc/os-release。这就是用 cat /etc/os-release 命令替换了默认的 /bin/bash 命令了，输出了系统版本信息。

在指令格式上，一般推荐使用 exec 格式，这类格式在解析时会被解析为 JSON 数组，因此一定要使用双引号 "，而不要使用单引号。

数组语法和shell语法的区别：

CMD /bin/echo

#or

CMD ["/bin/echo"]

这看起来好像没什么问题，但仔细一看其实两种方式差距很大。如果你使用第二个语法：CMD(或ENTRYPOINT)是一个数组，它执行的命令完全像你期望的那样。如果使用第一种语法，Docker会在你的命令前面加上/bin/sh -c，我记得一直都是这样。

如果你不知道Docker修改了CMD命令，在命令前加上/bin/sh -c可能会导致一些意想不到的问题以及难以理解的功能。

因此，在使用这两个指令时你应当使用数组语法，因为数组语法会确切地执行你打算执行的命令。

使用CMD和ENTRYPOINT时，请务必使用数组语法。

提到 CMD 就不得不提容器中应用在前台执行和后台执行的问题。这是初学者常出现的一个混淆。

Docker 不是虚拟机，容器中的应用都应该以前台执行，而不是像虚拟机、物理机里面那样，用 upstart/systemd 去启动后台服务，容器内没有后台服务的概念。

一些初学者将 CMD 写为：CMD service nginx start

然后发现容器执行后就立即退出了。甚至在容器内去使用 systemctl 命令结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

而使用 service nginx start 命令，则是希望 upstart 来以后台守护进程形式启动 nginx 服务。而刚才说了 CMD service nginx start 会被理解为 CMD [ "sh", "-c", "service nginx start"]，因此主进程实际上是 sh。那么当 service nginx start 命令结束后，sh 也就结束了，sh 作为主进程退出了，自然就会令容器退出。

正确的做法是直接执行 nginx 可执行文件，并且要求以前台形式运行。比如：CMD ["nginx", "-g", "daemon off;"]

a.一个 Dockerfile 中只能有一个CMD，如果有多个，则最后一个生效。

b.CMD 的 shell 形式默认调用 /bin/sh -c 执行命令。

c.CMD 命令会被 Docker 命令行传入的参数覆盖：docker run busybox /bin/echo Hello Docker 会把 CMD 里的命令覆盖

(7).ENTRYPOINT

ENTRYPOINT 命令也有两种格式：

ENTRYPOINT ["executable", "param1", "param2"] ：推荐使用的 exec 形式

ENTRYPOINT command param1 param2 ：shell 形式

ENTRYPOINT 命令的字面意思是进入点，而功能也恰如其意：他可以让你的容器表现得像一个可执行程序一样。

一个 Dockerfile 中只能有一个 ENTRYPOINT，如果有多个，则最后一个生效。

当指定了 ENTRYPOINT 后，CMD 的含义就发生了改变，不再是直接的运行其命令，而是将 CMD 的内容作为参数传给 ENTRYPOINT 指令，换句话说实际执行时，将变为：

<ENTRYPOINT> "<CMD>"

那么有了 CMD 后，为什么还要有 ENTRYPOINT 呢？这种 <ENTRYPOINT> "<CMD>" 有什么好处么？让我们来看几个场景。

应用实例1：

关于 CMD 和 ENTRYPOINT 的联系请看下面的例子

仅仅使用 ENTRYPOINT：

FROM ubuntu

ENTRYPOINT ls -l

[root@k8s-node1 ~]# docker build -t lstest .

Sending build context to Docker daemon 35.84kB

Successfully built d5f618b23aa5

执行 docker run d5f618b23aa5 /etc/fstab 和 docker run d5f618b23aa5 结果并不会有什么差别：

[root@k8s-node1 ~]# docker run d5f618b23aa5

[root@k8s-node1 ~]# docker run d5f618b23aa5 /etc/fstab

total 4

drwxr-xr-x 2 root root 4096 Jan 12 21:10 bin

drwxr-xr-x 2 root root 6 Apr 24 2018 boot

drwxr-xr-x 5 root root 340 Mar 2 07:58 dev

drwxr-xr-x 1 root root 66 Mar 2 07:58 etc

drwxr-xr-x 2 root root 6 Apr 24 2018 home

但是我们通常使用 ENTRYPOINT 作为容器的入口，使用 CMD 给 ENTRYPOINT 增加默认选项：

[root@k8s-node1 ~]# vim Dockerfile

FROM ubuntu

CMD ["-l"]

ENTRYPOINT ["ls"]

[root@k8s-node1 ~]# docker build -t lstest:v1 .

Successfully built 5899b788ad7a

Successfully tagged lstest:v1

然后执行这个容器：

不加参数便会默认有 -l参数：

[root@k8s-node1 ~]# docker run 5899b788ad7a

total 4

drwxr-xr-x 2 root root 4096 Jan 12 21:10 bin

drwxr-xr-x 2 root root 6 Apr 24 2018 boot

drwxr-xr-x 5 root root 340 Mar 2 08:05 dev

drwxr-xr-x 1 root root 66 Mar 2 08:05 etc

drwxr-xr-x 2 root root 6 Apr 24 2018 home

加了 /etc/fstab 参数便会覆盖原有的 -l 参数：

[root@k8s-node1 ~]# docker run 5899b788ad7a /etc/fstab

/etc/fstab

应用实例2：让镜像变成像命令一样使用

假设我们需要一个得知自己当前公网 IP 的镜像，那么可以先用 CMD 来实现

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

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

CMD [ "curl", "-s", "http://ip.cn" ]

假如我们使用 docker build -t myip . 来构建镜像的话，如果我们需要查询当前公网 IP，只需要执行：

# docker run myip

当前 IP：61.148.226.66 来自：北京市 联通

嗯，这么看起来好像可以直接把镜像当做命令使用了，不过命令总有参数，如果我们希望加参数呢？比如从上面的 CMD 中可以看到实质的命令是 curl，那么如果我们希望显示 HTTP 头信息，就需要加上 -i 参数。那么我们可以直接加 -i 参数给 docker run myip 么？

# docker run myip -i

docker: Error response from daemon: invalid header field value "oci runtime error: container\_linux.go:247: starting container process caused \"exec: \\\"-i\\\": executable file not found in $PATH\"\n".

我们可以看到可执行文件找不到的报错，executable file not found。之前我们说过，跟在镜像名后面的是 command，运行时会替换 CMD 的默认值。因此这里的 -i 替换了原来的 CMD，而不是添加在原来的 curl -s http://ip.cn 后面。而 -i 根本不是命令，所以自然找不到。

那么如果我们希望加入 -i 这参数，我们就必须重新完整的输入这个命令：

# docker run myip curl -s http://ip.cn -i

这显然不是很好的解决方案，而使用 ENTRYPOINT 就可以解决这个问题。现在我们重新用 ENTRYPOINT 来实现这个镜像：

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

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

ENTRYPOINT [ "curl", "-s", "http://ip.cn" ]

这次我们再来尝试直接使用 docker run myip -i：

# docker run myip

当前 IP：61.148.226.66 来自：北京市 联通

# docker run myip -i

HTTP/1.1 200 OK

Server: nginx/1.8.0

Date: Tue, 22 Nov 2016 05:12:40 GMT

Content-Type: text/html; charset=UTF-8

Vary: Accept-Encoding

X-Powered-By: PHP/5.6.24-1~dotdeb+7.1

X-Cache: MISS from cache-2

X-Cache-Lookup: MISS from cache-2:80

X-Cache: MISS from proxy-2\_6

Transfer-Encoding: chunked

Via: 1.1 cache-2:80, 1.1 proxy-2\_6:8006

Connection: keep-alive

当前 IP：61.148.226.66 来自：北京市 联通

可以看到，这次成功了。这是因为当存在 ENTRYPOINT 后，CMD 的内容将会作为参数传给 ENTRYPOINT，而这里 -i 就是新的 CMD，因此会作为参数传给 curl，从而达到了我们预期的效果。

应用实例2：应用运行前的准备工作

启动容器就是启动主进程，但有些时候，启动主进程前，需要一些准备工作。

比如 mysql 类的数据库，可能需要一些数据库配置、初始化的工作，这些工作要在最终的 mysql 服务器运行之前解决。

此外，可能希望避免使用 root 用户去启动服务，从而提高安全性，而在启动服务前还需要以 root 身份执行一些必要的准备工作，最后切换到服务用户身份启动服务。或者除了服务外，其它命令依旧可以使用 root 身份执行，方便调试等。

这些准备工作是和容器 CMD 无关的，无论 CMD 为什么，都需要事先进行一个预处理的工作。这种情况下，可以写一个脚本，然后放入 ENTRYPOINT 中去执行，而这个脚本会将接到的参数(也就是 <CMD>)作为命令，在脚本最后执行。比如官方镜像 redis 中就是这么做的：

Step1：编写Dockerfile

[root@localhost Redis-Docker]# vim Dockerfile

FROM centos:7.5.1804

ADD redis-5.0.5.tar.gz /usr/local/

COPY redis\_install.sh /usr/local/

EXPOSE 6379

RUN sh /usr/local/redis\_install.sh

ENTRYPOINT [ "/usr/local/redis/src/redis-server","/usr/local/redis/redis.conf"]

CMD []

Step2：编写redis安装脚本

[root@localhost Redis-Docker]# vim redis\_install.sh

#!/bin/bash

yum install -y gcc gcc-c++ make openssl openssl-devel

cd /usr/local/

mv redis-5.0.5 redis

cd redis

make && make install

sed -i 's/bind 127.0.0.1/bind 0.0.0.0/g' /usr/local/redis/redis.conf

sed -i 's/protected-mode yes/protected-mode no/g' /usr/local/redis/redis.conf

echo "requirepass 123456" >> /usr/local/redis/redis.conf

Step3：构建redis\_server镜像

[root@localhost ~]# docker build -t redis\_server /root/Redis-Docker/

Sending build context to Docker daemon 1.979MB

Step 1/7 : FROM centos:7.5.1804

---> cf49811e3cdb

Step 2/7 : ADD redis-5.0.5.tar.gz /usr/local/

---> 36b4eabdef18

......................................................................................................................

Removing intermediate container 20694a7d14b9

---> d2b0d0fc56c4

Step 6/7 : ENTRYPOINT [ "/usr/local/redis/src/redis-server","/usr/local/redis/redis.conf"]

---> Running in a63ea3322c6d

Removing intermediate container a63ea3322c6d

---> ae65db44ea50

Step 7/7 : CMD []

---> Running in ef5bdb0969c4

Removing intermediate container ef5bdb0969c4

---> 20377202a6b2

Successfully built 20377202a6b2

Successfully tagged redis\_server:latest

Step4：启动redis\_server容器

[root@localhost ~]# docker run -it -d -p 6379:6379 redis\_server

55960d015d28d0cf301f3861c407d235170dea01a2375e54b5a5f9448ed5888e

[root@localhost ~]# netstat -tulnp|grep 6379

tcp6 0 0 :::6379 :::\* LISTEN 18881/docker-proxy

Step5：测试Docker-redis服务

[root@localhost redis]# src/redis-cli -h 10.200.9.105 -p 6379 -a 123456

Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.

10.200.9.105:6379> select 0

OK

(8).EXPOSE

EXPOSE <port> [<port>...] 命令用来指定对外开放的端口。

例如 EXPOSE 80 3306，开放 80 和 3306 端口。

注：两个Docker的核心概念是可重复和可移植。镜像应该可以运行在任何主机上并且运行尽可能多的次数。在Dockerfile中你有能力映射私有和公有端口，但是你永远不要通过Dockerfile映射公有端口。通过映射公有端口到主机上，你将只能运行一个容器化应用程序实例。(译者注：运行多个端口不就冲突啦)

# private and public mapping

EXPOSE 80:8080

# private only

EXPOSE 80

如果镜像的使用者关心容器公有映射了哪个公有端口，他们可以在运行镜像时通过-p参数设置，否则，Docker会自动为容器分配端口。

切勿在Dockerfile映射公有端口。

(9).WORKDIR 指定工作目录

格式为 WORKDIR <工作目录路径>。

使用 WORKDIR 指令可以来指定工作目录(或者称为当前目录)，以后各层的当前目录就被改为指定的目录，如该目录不存在，WORKDIR 会帮你建立目录。

之前提到一些初学者常犯的错误是把 Dockerfile 等同于 Shell 脚本来书写，这种错误的理解会导致出现下面这样的错误：

RUN cd /app

RUN echo "hello" > world.txt

如果将这个 Dockerfile 进行构建镜像运行后，会发现找不到 /app/world.txt 文件，或者其内容不是 hello。原因其实很简单：

在 Shell 中，连续两行是同一个进程执行环境，因此前一个命令修改的内存状态，会直接影响后一个命令；

而在 Dockerfile 中，这两行 RUN 命令的执行环境根本不同，是两个完全不同的容器。这就是对 Dokerfile 构建分层存储的概念不了解所导致的错误。

之前说过每一个 RUN 都是启动一个容器、执行命令、然后提交存储层文件变更。

第一层 RUN cd /app 的执行仅仅是当前进程的工作目录变更，一个内存上的变化而已，其结果不会造成任何文件变更。

然而到第二层的时候，启动的是一个全新的容器，跟第一层的容器更完全没关系，自然不可能继承前一层构建过程中的内存变化。

因此如果需要改变以后各层的工作目录的位置，那么应该使用 WORKDIR 指令。

应用实例：

[root@k8s-node1 ~]# vim Dockerfile

FROM ubuntu

WORKDIR /etc

WORKDIR ..

WORKDIR usr

WORKDIR lib

ENTRYPOINT pwd

[root@k8s-node1 ~]# docker build -t "path" .

Sending build context to Docker daemon 36.35kB

Successfully built 581e5126f576

Successfully tagged path:latest

[root@k8s-node1 ~]# docker run 581e5126f576

/usr/lib

(10).USER

USER <UID/Username> 为容器内指定 CMD、RUN、ENTRYPOINT 命令运行时的用户名或UID。

指定当前用户

格式：USER <用户名>

USER 指令和 WORKDIR 相似，都是改变环境状态并影响以后的层。WORKDIR 是改变工作目录，USER 则是改变之后层的执行 RUN, CMD 以及 ENTRYPOINT 这类命令的身份。

当然，和 WORKDIR 一样，USER 只是帮助你切换到指定用户而已，这个用户必须是事先建立好的，否则无法切换。

RUN groupadd -r redis && useradd -r -g redis redis

USER redis

RUN [ "redis-server" ]

如果以 root 执行的脚本，在执行期间希望改变身份，比如希望以某个已经建立好的用户来运行某个服务进程，不要使用 su 或者 sudo，这些都需要比较麻烦的配置，而且在 TTY 缺失的环境下经常出错。建议使用 gosu，可以从其项目网站看到进一步的信息：https://github.com/tianon/gosu

# 建立 redis 用户，并使用 gosu 换另一个用户执行命令

RUN groupadd -r redis && useradd -r -g redis redis

# 下载 gosu

RUN wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/1.7/gosu-amd64" \

&& chmod +x /usr/local/bin/gosu \

&& gosu nobody true

# 设置 CMD，并以另外的用户执行

CMD [ "exec", "gosu", "redis", "redis-server" ]

(11).VLOUME： 定义匿名卷

格式为：

VOLUME ["<路径1>", "<路径2>"...]

VOLUME <路径>

之前我们说过，容器运行时应该尽量保持容器存储层不发生写操作，对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中，后面的章节我们会进一步介绍 Docker 卷的概念。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 Dockerfile 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

VOLUME /data

这里的 /data 目录就会在运行时自动挂载为匿名卷，任何向 /data 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。比如：

docker run -d -v mydata:/data xxxx

在这行命令中，就使用了 mydata 这个命名卷挂载到了 /data 这个位置，替代了 Dockerfile 中定义的匿名卷的挂载配置。

那么Dockerfile中的VOLUME指令实际使用中是不是就是跟docker run中的-v参数一样是将宿主机的一个目录绑定到容器中的目录以达到共享目录的作用呢？并不然，其实VOLUME指令只是起到了声明了容器中的目录作为匿名卷，但是并没有将匿名卷绑定到宿主机指定目录的功能。但是当我们生成镜像的Dockerfile中以Volume声明了匿名卷，并且我们以这个镜像run了一个容器的时，docker会在安装目录下的指定目录下面生成一个目录来绑定容器的匿名卷(这个指定目录不同版本的docker会有所不同)。如我使用的版本如下：

[root@localhost ~]# docker --version

Docker version 19.03.9, build 9d988398e7

[root@localhost ~]# cd /var/lib/docker/volumes/

[root@localhost volumes]# ll -h

total 32K

drwxr-xr-x. 3 root root 19 Jul 27 16:42 53c144446ccaee65791f81cbba91b83f5d07a0b0dba334af2a88c332ca483557

//其中一个容器的共享目录文件夹

drwxr-xr-x. 3 root root 19 Jul 27 16:30 62451022eaab8cc8e1c4caa9b043f0dd5051762d244d39a9daa8a866cf7d3381

//其中一个容器的共享目录文件夹

[root@localhost volumes]# pwd

/var/lib/docker/volumes //默认绑定容器的匿名卷的目录

就是说当Dockerfile中声明了匿名卷但是run的时候没有使用 -v绑定匿名卷的话那么docker就会在/var/lib/docker/volumes这个目录下创建一个目录来绑定匿名卷。

所以真正使用的时候我们在Dockerfile构建镜像的时候如：

[root@localhost ~]# vim Dockerfile

FROM centos:latest

RUN groupadd -r redis && useradd -r -g redis redis

RUN yum -y update && yum -y install epel-release && yum -y install redis && yum -y install net-tools

RUN mkdir -p /config && chown -R redis:redis /config

VOLUME /redis/data　　　　　　#声明容器中/redis/data为匿名卷

EXPOSE 6379

那么使用该Dockerfile构建镜像的为

[root@localhost ~]# docker build -t image-redis　 //构建镜像image-redis

[root@localhost ~]# docker run -it -d -name redis1 -v /data:/redis/data image-redis

//运行一个容器并且将当前机器的/data目录绑定到容器的匿名卷中

[root@localhost ~]#docker run -itd -name redis2 image-redis

//运行一个容器但是不绑定目录到容器的匿名卷，这时候在/var/lib/docker/volumes中就会创建一个目录绑定匿名卷

(12).HEALTHCHECK：健康检查

格式：

HEALTHCHECK [选项] CMD <命令>：设置检查容器健康状况的命令

HEALTHCHECK NONE：如果基础镜像有健康检查指令，使用这行可以屏蔽掉其健康检查指令

HEALTHCHECK 指令是告诉 Docker 应该如何进行判断容器的状态是否正常，这是 Docker 1.12 引入的新指令。

在没有 HEALTHCHECK 指令前，Docker 引擎只可以通过容器内主进程是否退出来判断容器是否状态异常。很多情况下这没问题，但是如果程序进入死锁状态，或者死循环状态，应用进程并不退出，但是该容器已经无法提供服务了。在 1.12 以前，Docker 不会检测到容器的这种状态，从而不会重新调度，导致可能会有部分容器已经无法提供服务了却还在接受用户请求。

而自 1.12 之后，Docker 提供了 HEALTHCHECK 指令，通过该指令指定一行命令，用这行命令来判断容器主进程的服务状态是否还正常，从而比较真实的反应容器实际状态。

当在一个镜像指定了 HEALTHCHECK 指令后，用其启动容器，初始状态会为 starting，在 HEALTHCHECK 指令检查成功后变为 healthy，如果连续一定次数失败，则会变为 unhealthy。

HEALTHCHECK 支持下列选项：

--interval=<间隔>：两次健康检查的间隔，默认为 30 秒；

--timeout=<时长>：健康检查命令运行超时时间，如果超过这个时间，本次健康检查就被视为失败，默认 30 秒；

--retries=<次数>：当连续失败指定次数后，则将容器状态视为 unhealthy，默认 3 次。

和 CMD, ENTRYPOINT 一样，HEALTHCHECK 只可以出现一次，如果写了多个，只有最后一个生效。

在 HEALTHCHECK [选项] CMD 后面的命令，格式和 ENTRYPOINT 一样，分为 shell 格式，和 exec 格式。命令的返回值决定了该次健康检查的成功与否：0：成功；1：失败；2：保留，不要使用这个值。

假设我们有个镜像是个最简单的 Web 服务，我们希望增加健康检查来判断其 Web 服务是否在正常工作，我们可以用 curl 来帮助判断，其 Dockerfile 的 HEALTHCHECK 可以这么写：

[root@k8s-node1 ~]# vim Dockerfile

FROM nginx

RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/\*

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

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

这里我们设置了每 5 秒检查一次(这里为了试验所以间隔非常短，实际应该相对较长)，如果健康检查命令超过 3 秒没响应就视为失败，并且使用 curl -fs http://localhost/ || exit 1 作为健康检查命令。

使用 docker build 来构建这个镜像：

[root@k8s-node1 ~]# docker build -t myweb:v1 .

构建好了后，我们启动一个容器：

[root@k8s-node1 ~]# docker run -d --name web -p 80:80 myweb:v1

当运行该镜像后，可以通过 docker ps 看到最初的状态为 (health: starting)：

在等待几秒钟后，再次 docker ps，就会看到健康状态变化为了 (healthy)：

[root@k8s-node1 ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

f4cee52a1252 myweb:v1 "nginx -g 'daemon of…" 9 seconds ago Up 8 seconds (healthy) 0.0.0.0:80->80/tcp web

如果健康检查连续失败超过了重试次数，状态就会变为 (unhealthy)。

为了帮助排障，健康检查命令的输出(包括 stdout 以及 stderr)都会被存储于健康状态里，可以用 docker inspect 来查看。

[root@k8s-node1 ~]# docker inspect --format '{{json .State.Health}}' web | python -m json.tool

{

"FailingStreak": 0,

"Log": [

{

"End": "2020-03-04T17:35:34.241256752+08:00",

"ExitCode": 0,

"Output": "<!DOCTYPE html>\n<html>\n<head>\n<title>Welcome to nginx!</title>\n<style>\n body {\n width: 35em;\n margin: 0 auto;\n font-family: Tahoma, Verdana, Arial, sans-serif;\n }\n</style>\n</head>\n<body>\n<h1>Welcome to nginx!</h1>\n<p>If you see this page, the nginx web server is successfully installed and\nworking. Further configuration is required.</p>\n\n<p>For online documentation and support please refer to\n<a href=\"http://nginx.org/\">nginx.org</a>.<br/>\nCommercial support is available at\n<a href=\"http://nginx.com/\">nginx.com</a>.</p>\n\n<p><em>Thank you for using nginx.</em></p>\n</body>\n</html>\n",

"Start": "2020-03-04T17:35:34.091237515+08:00"

}

],

"Status": "healthy"

}

应用实例：构建dockerfile

[root@k8s-node1 ~]# vim Dockerfile

FROM nginx

MAINTAINER zhuhaifeng "13956072050@163.com"

ENV REFRESHED\_AT 2020-03-02

RUN echo '<h1>hello,<a href="http://www.imooc.com">HaiFeng</a>!</h1>' > /usr/share/nginx/html/index.html

[root@k8s-node1 ~]# docker build -t zhuhaifeng/nginx\_web:v1 .

Sending build context to Docker daemon 36.35kB

Successfully built 6e0ae8405c4c

Successfully tagged zhuhaifeng/nginx\_web:v1

[root@k8s-node1 ~]# docker run --name nginx\_web -d -p 82:80 zhuhaifeng/nginx\_web:v1

1ef8e1e934ce402fc460fb03f6d42c1a06c9bf477c96086b643df3732e526858

curl访问192.168.70.129:82

[root@k8s-node1 ~]# curl 192.168.70.129:82

<h1>hello,<a href="http://www.imooc.com">HaiFeng</a>!</h1>

五、docker搭建私有仓库

Docker提供了开放的中央仓库dockerhub，同时也允许我们使用registry搭建本地私有仓库。搭建私有仓库有如下的优点：

节省网络带宽，提升Docker部署速度，不用每个镜像从DockerHub上去下载，只需从私有仓库下载就可；

私有镜像，包含公司敏感信息，不方便公开对外，只在公司内部使用。

1.搭建私有仓库

(1). 下载镜像registry

# docker pull registry

[root@k8s-node1 ~]# docker pull registry

Using default tag: latest

latest: Pulling from library/registry

486039affc0a: Pull complete

ba51a3b098e6: Pull complete

8bb4c43d6c8e: Pull complete

6f5f453e5f2d: Pull complete

42bc10b72f42: Pull complete

Digest: sha256:7d081088e4bfd632a88e3f3bcd9e007ef44a796fddfe3261407a3f9f04abe1e7

Status: Downloaded newer image for registry:latest

(2) 启动容器

# mkdir -p /mydata/dockerRegistry

# docker run -d -p 5000:5000 --restart=always --name=registry-srv -v /mydata/dockerRegistry:/var/lib/registry registry

解释一下:

-d：后台运行

-p：将容器的5000端口映射到宿主机的5000端口

--restart：docker服务重启后总是重启此容器

--name：容器的名称

-v：将容器内的/var/lib/registry映射到宿主机的/mydata/dockerRegistry目录

[root@k8s-node1 ~]# mkdir -p /mydata/dockerRegistry

[root@k8s-node1 ~]# docker run -d -p 5000:5000 --restart=always --name=registry-srv -v /mydata/dockerRegistry:/var/lib/registry registry

60f44394a32d1f3739ceb0cb0aee52cc961e985b056bfb50dc2da864866e9f10

2.搭建WEB服务

私有仓库搭建好了，怎么查看仓库里的镜像，搭建一个web服务，查看修改image比较方便。

(1) 下载镜像

# docker pull hyper/docker-registry-web

[root@k8s-node1 ~]# docker pull hyper/docker-registry-web

Using default tag: latest

latest: Pulling from hyper/docker-registry-web

Digest: sha256:723ffa29aed2c51417d8bd32ac93a1cd0e7ef857a0099c1e1d7593c09f7910ae

Status: Downloaded newer image for hyper/docker-registry-web:latest

(2) 启动容器

# docker run -it -d -p 8080:8080 --restart=always --name registry-web --link registry-srv -e REGISTRY\_URL=http://registry-srv:5000/v2 -e REGISTRY\_NAME=localhost:5000 hyper/docker-registry-web

解释一下：

-it: 以交互模式运行

--link：链接其它容器(registry-srv)，在此容器中，使用registry-srv等同于registry-srv容器的局域网地址

-e：设置环境变量

[root@k8s-node1 ~]# docker run -it -d -p 8080:8080 --restart=always --name registry-web --link registry-srv -e REGISTRY\_URL=http://registry-srv:5000/v2 -e REGISTRY\_NAME=localhost:5000 hyper/docker-registry-web

f6b1305a6d7235c846133b712d23f16bdcfb013066113127434173a12d9b30fc

3.上传本地镜像到私有仓库

例如，将本地的hcharts上传到仓库

[root@k8s-node1 ~]# docker images | grep myweb

myweb v4 48367ccdc186 18 hours ago 135MB

(1) 修改镜像tag

[root@k8s-node1 ~]# docker tag myweb:v4 192.168.70.129:5000/myweb:v4

(2). 上传tag镜像到仓库

上传本地镜像到私有仓库

# docker push 192.168.70.129:5000/myweb:v4

[root@k8s-node1 ~]# docker push 192.168.70.129:5000/myweb:v4

The push refers to repository [192.168.70.129:5000/myweb]

66506abdf2e2: Pushed

318be7aea8fc: Pushed

fe08d5d042ab: Pushed

f2cb0ecef392: Pushed

v4: digest: sha256:039f86b231930471dc01297af20c5aafec7a9e90d45553cad913c733598743c4 size: 1159

(3). 会出现的push失败

上面命令执行后可能会报"http: server gave HTTP response to HTTPS client"错误

[root@k8s-node1 ~]# docker push 192.168.70.129:5000/myweb:v4

The push refers to repository [192.168.70.129:5000/myweb]

Get https://192.168.70.129:5000/v2/: http: server gave HTTP response to HTTPS client

这是因为 Docker从1.3.X 之后，与 docker registry 交互默认使用的是 https，然而我们前面搭建的私有仓库只提供 http 服务，为了解决这个问题需要在启动 docker server 时增加启动参数为默认使用 http 访问。

# vim /etc/docker/daemon.json

{ "insecure-registries":["10.164.9.50:5000"] }

最后执行如下两个命令重启 docker：

# systemctl daemon-reload

# systemctl restart docker

(4).查看镜像

我们访问 http://IP:5000/v2/\_catalog 即可看到当前仓库里的所有镜像：

http://192.168.70.129:5000/v2/\_catalog

{"repositories":["myweb"]}

(5). 下载镜像

在另外的客户机同样使用 docker pull 从本地 registry 下载镜像，除了镜像名称长一些，使用方式同 Docker Hub 下载镜像完全一样：

# docker pull 192.168.70.129:5000/myweb:v4

[root@k8s-node2 ~]# docker pull 192.168.70.129:5000/myweb:v4

Error response from daemon: Get https://192.168.70.129:5000/v2/: http: server gave HTTP response to HTTPS client

[root@k8s-node2 ~]# vim /usr/lib/systemd/system/docker.service

找到 ExecStart，按下 i 键在其末尾添加如下内容(IP 则为 registry 主机的 IP)：--insecure-registry 192.168.70.129:5000

最后执行如下两个命令重启 docker：

# systemctl daemon-reload

# systemctl restart docker

[root@k8s-node2 ~]# systemctl daemon-reload

[root@k8s-node2 ~]# systemctl restart docker

[root@k8s-node2 ~]# docker pull 192.168.70.129:5000/myweb:v4

v4: Pulling from myweb

68ced04f60ab: Pull complete

c4039fd85dcc: Pull complete

c16ce02d3d61: Pull complete

1e058f7ba8ce: Pull complete

Digest: sha256:039f86b231930471dc01297af20c5aafec7a9e90d45553cad913c733598743c4

Status: Downloaded newer image for 192.168.70.129:5000/myweb:v4

192.168.70.129:5000/myweb:v4

[root@k8s-node2 ~]# docker images |grep "myweb"

192.168.70.129:5000/myweb v4 48367ccdc186 18 hours ago 135MB

六、Docker 数据管理

1.数据卷

数据卷是一个可供容器使用的特殊目录，有如下特性：

数据卷可以在容器之间共享和重用

数据卷修改会立即生效

数据卷的更新不会影响镜像

如果有容器使用数据卷，该卷会一直存在

需求：现在我有一个CentOS镜像，希望centos镜像运行起来的时候其containerdata目录与宿主机的centosdata两个目录之间能够进行数据共享；

命令：docker run -it -v 宿主机中的共享目录(centosdata)路径 : 容器中的共享目录(containerdata)的路径 容器名

示例：比如centosdata在宿主机的根目录下(/centosdata)，containerdata在容器的根目录下(/containerdata)

(1).首先进行目录绑定：

docker run -it -d -v /centosdata:/containerdata centos

[root@localhost ~]# docker run -it -d -v /centosdata:/containerdata centos

02e1084d981bac741d1d77a62466f991e5ac12fc1bcbfd5cfc0071407ecaee85

(2).查看宿主机和容器的绑定关系

docker inspect 容器ID

[root@localhost ~]# docker inspect 02e1084d981bac

"Mounts": [

{

"Type": "bind",

"Source": "/centosdata",

"Destination": "/containerdata",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

]

(3).查看宿主机和容器中是否分别有centosdata和containerdata文件夹

[root@localhost /]# ls -lh | grep centosdata

drwxr-xr-x. 2 root root 6 Jul 28 15:27 centosdata

[root@localhost ~]# docker exec -it 02e1084d981b /bin/bash

[root@02e1084d981b /]# ls -lh | grep containerdata

drwxr-xr-x. 2 root root 6 Jul 28 07:27 containerdata

现在在centosdata目录中创建一个文档：

[root@localhost centosdata]# touch AnHui

[root@localhost centosdata]# ls -lh

total 0

-rw-r--r--. 1 root root 0 Jul 28 15:36 AnHui

既然centosdata目录和centos的容器中的containerdata目录进行了数据共享，那么现在containerdata目录下应该也有一个AnHui文档。

查看containerdata目录：

[root@02e1084d981b containerdata]# ls -l -h

total 0

-rw-r--r--. 1 root root 0 Jul 28 07:36 AnHui

发现containerdata目录下确实有AnHui文档，现在通过容器编辑该文档：

[root@02e1084d981b containerdata]# cat AnHui

HeFei is the center of AnHui

在该文档中输入一段话，按照数据共享的原理，在宿主机的centosdata目录下的AnHui文档中应该也会有这样一句话,

此时再次查看宿主机的centosdata下的AnHui：

[root@localhost centosdata]# cat AnHui

HeFei is the center of AnHui

到这里就实现了宿主机与容器之间的数据共享~

注意：

通过上面的绑定操作之后，即使我们的容器关闭运行了，但是如果我们在宿主机的centosdata目录下进行的操作依然会被同步到与其绑定的centos镜像中的containerdata目录下：

示例：

关闭centos的运行并退出：

[root@localhost ~]# docker stop 02e1084d981b

02e1084d981b

查看此时运行的镜像文件：

确认上面的centos镜像文件已经退出运行了~

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

此时在宿主机的centosdata目录下创建一个FuJian文件，并添加内容：FuZhou is the center of FuJian

[root@localhost centosdata]# touch FuJian

[root@localhost centosdata]# cat FuJian

FuZhou is the center of FuJian

此时再运行centos容器，到containerdata目录下查看：

[root@localhost ~]# docker start 02e1084d981b

02e1084d981b

查看containerdata目录下是否有新增的FuJian文档：

[root@localhost ~]# docker exec -it 02e1084d981b /bin/bash

[root@02e1084d981b /]# cd containerdata/

[root@02e1084d981b containerdata]# ls -lh

total 8.0K

-rw-r--r--. 1 root root 29 Jul 28 07:40 AnHui

-rw-r--r--. 1 root root 31 Jul 28 07:51 FuJian

[root@02e1084d981b containerdata]# cat FuJian

FuZhou is the center of FuJian

发现我们在宿主机上新建的FuJian文档以及里面的数据同步到了centos容器相应的绑定位置；

(4).宿主机与容器数据共享的时候添加权限

宿主机与容器进行数据共享的目录绑定命令：

# docker run -it -v /宿主机共享文件的绝对路径:/容器共享文件的绝对路径 :ro 镜像名

注：ro就是read only(只读权限)

通过以上命令进行共享数据目录绑定之后，我们可以从宿主机上写数据(增删改),而容器上只能读数据。

注：归纳总结

①数据卷目录与容器内目录有映射关系，所以不管是在容器内部修改数据卷还是在外部修改数据卷，相对应的数据卷都会发生改变。

②在容器内修改文件数据后,exit退出容器,进入本机路径查看文件,会发现文件也发生了变化.

③只读权限是无法写入数据的

2.数据卷容器

数据卷容器用于用户需要在容器间共享一些持续更新的数据，数据卷容器专门提供数据卷供其它容器挂载使用。

(1).创建数据卷容器

Example:

创建数据卷容器db1

# docker run -d --name db1 -v /tmp/dbdata/ -it centos bash

创建容器db2与db1共享dbdata的数据

# docker run -d --name db2 --volumes-from db1 -ti centos bash

创建数据卷容器db1

[root@k8s-node1 ~]# docker run -d --name db1 -v /tmp/dbdata/ -it centos bash

23cfe3e68fa4459792514ea076c53de6a04cb1d48b029c5d7258d655d30c0363

创建数据卷容器db2,使用db1的数据卷

[root@k8s-node1 ~]# docker run -d --name db2 --volumes-from db1 -it centos bash

70a70e2daf6943275716e25526baa4209efb1000c429e5f8d1f3cf22fa61f90a

在容器db1和容器db2任意一个容器修改/tmp/dbdata/的内容，在两个容器内均生效。

[root@k8s-node1 ~]# docker exec -it db2 /bin/bash

[root@70a70e2daf69 /]# cd /tmp/dbdata/

[root@70a70e2daf69 dbdata]# echo "databases" > DBS

[root@70a70e2daf69 dbdata]# ls

DBS

[root@k8s-node1 ~]# docker exec -it db1 /bin/bash

[root@23cfe3e68fa4 /]# cd /tmp/dbdata/

[root@23cfe3e68fa4 dbdata]# ls

DBS

(2).数据卷容器的删除

如果删除了挂载的容器，数据卷并不会被自动删除；

如果要删除一个数据卷，必须在删除最后一个还挂载它的容器时显示使用docker rm -v 命令指定同时删除关联的容器；

可以利用数据卷容器对其中的数据卷进行备份、恢复，以实现数据的迁移。

(3).数据卷容器的备份

使用下面的命令来备份dbdata数据卷容器内的数据卷：

# docker run --volumes-from db1 -d -it -v /tmp/webapp:/backup --name db3 centos \tar -apcv -f /backup/backup.tar.gz /tmp/dbdata

说明：

利用centos镜像创建一个容器db3。

使用--volumes-from /tmp/dbdata来让db3容器挂载dbdata的数据卷；

db3启动后，使用tar命令将/tmp/dbdata下的内容备份为容器内的/backup/路径下,名称为backup.tar.gz；

使用/tmp/webapp:/backup来挂载本地目录/tmp/webapp到db3容器的/backup目录。

[root@k8s-node1 ~]# docker run --volumes-from db1 -d -it -v /tmp/webapp:/backup --name db3 centos \tar -apcv -f /backup/backup.tar.gz /tmp/dbdata

tar: Removing leading `/' from member names

/tmp/dbdata/

/tmp/dbdata/DBS

[root@k8s-node1 ~]# ll /tmp/webapp/

total 4

-rw-r--r-- 1 root root 162 Mar 5 18:02 backup.tar.gz

[root@k8s-node1 ~]# cd /tmp/webapp/

[root@k8s-node1 webapp]# ll

total 4

-rw-r--r-- 1 root root 162 Mar 5 18:02 backup.tar.gz

[root@k8s-node1 webapp]# tar -apxv -f backup.tar.gz

tmp/dbdata/

tmp/dbdata/DBS

[root@k8s-node1 webapp]# ll tmp/dbdata/DBS

-rw-r--r-- 1 root root 10 Mar 5 17:57 tmp/dbdata/DBS

DBS数据文件已经完成了备份。

(4).数据卷容器的恢复

如果恢复数据到一个容器，可以参照下面的操作。首先创建一个带有数据卷的容器db4

# docker run -d -it -v /tmp/dbdata --name db4 centos /bin/bash

[root@k8s-node1 ~]# docker run -d -it -v /tmp/dbdata --name db4 centos /bin/bash

e2bbac101dba17ea2365d0e7905e5a91e964436c07293636cf338a8877f366a0

然后创建另一个新的容器db5，挂载到db4，并使用tar命令解压备份文件到挂载的容器卷中即可

# docker run --volumes-from db4 -d -it -v /tmp/webapp/:/backup --name db5 centos tar -apxv -f /backup/backup.tar.gz

[root@k8s-node1 ~]# docker run --volumes-from db4 -d -it -v /tmp/webapp/:/backup --name db5 centos tar -apxv -f /backup/backup.tar.gz

tmp/dbdata/

tmp/dbdata/DBS

此时发现db4的数据卷(/dbdata)的内容与db1数据卷(/dbdata)的内容的数据是同步的了, 暨实现了数据卷数据的恢复。

[root@k8s-node1 ~]# docker exec -it db4 /bin/bash

[root@e2bbac101dba /]# cd /tmp/dbdata/

[root@e2bbac101dba dbdata]# ls -l

total 4

-rw-r--r-- 1 root root 10 Mar 5 09:57 DBS

[root@k8s-node1 ~]# docker exec -it db1 /bin/bash

[root@23cfe3e68fa4 /]# cd /tmp/dbdata/

[root@23cfe3e68fa4 dbdata]# ls -l

total 4

-rw-r--r-- 1 root root 10 Mar 5 09:57 DBS

七、匿名卷(匿名挂载)和命名卷(具名挂载)

1.什么是匿名挂载和具名挂载？

匿名挂载(匿名卷)：即在进行数据卷挂载的时候不指定宿主机的数据卷目录，-v命令之后直接跟上容器内数据卷所在的路径。

具名挂载(命名卷)：即在进行数据卷挂载的时候既指定宿主机数据卷所在路径，又指定容器数据卷所在路径。

#匿名挂载(匿名卷)

docker run -it -d -p 88:88 --name CentOS1 -v /data centos

[root@localhost ~]#

[root@localhost ~]# docker run -it -d -p 88:88 --name CentOS1 -v /data centos

2be83d47a972dfbbb83b84a2cff39d59788507c1443b9ad2a1d69ff01ccfc8b9

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2be83d47a972 centos "/bin/bash" 3 seconds ago Up 2 seconds 0.0.0.0:88->88/tcp CentOS1

#具名挂载(命名卷) -v 宿主机数据卷所在路径：容器数据卷所在路径

docker run -it -d -p 99:99 --name CentOS2 -v /data:/data centos

[root@localhost ~]# docker run -it -d -p 99:99 --name CentOS2 -v /data:/data centos

6e51572396b02b9ca70adf2e33eecf824eac73eb135cd801f2c8c2bb8eeb1a48

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6e51572396b0 centos "/bin/bash" 3 seconds ago Up 2 seconds 0.0.0.0:99->99/tcp CentOS2

2be83d47a972 centos "/bin/bash" 2 minutes ago Up 2 minutes 0.0.0.0:88->88/tcp CentOS1

除此种方式之外，我们也可以在dockerfile构建docker镜像的时候使用VOLUME保留字来对数据卷进行挂载，此种挂载方式是匿名挂载的，我们可以指定一个或多个数据卷，这样只要启动了该自定义容器镜像，则会自动进行数据挂载，不会出现忘记挂载导致数据不安全的情况。

VOLUME ["容器内数据卷路径1","容器内数据卷路径2"......]

由于匿名挂载的时候只是指定了容器内数据卷的路径，至于该容器内数据卷的路径到底和宿主机中的哪个文件进行数据挂载，可以使用下面命令进行查看：

#查看当前正在运行CentOS1的镜像容器id

[root@localhost ~]# docker ps | grep CentOS1

2be83d47a972 centos "/bin/bash" 7 minutes ago Up 7 minutes 0.0.0.0:88->88/tcp CentOS1

#使用 inspect 查看镜像信息

[root@localhost ~]# docker inspect 2be83d47a972(这是容器id)

#在弹出来的信息中找到下面的数据：

"Mounts": [

{

"Type": "volume",

"Name": "7d47ea9889c08df06aa56df5db8d65ea1904dd9dce7bb423a33178058b734ce5",

"Source": "/var/lib/docker/volumes/7d47ea9889c08df06aa56df5db8d65ea1904dd9dce7bb423a33178058b734ce5/\_data",

"Destination": "/data",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

]

#查看当前正在运行CentOS2的镜像容器id

[root@localhost ~]# docker ps | grep CentOS2

6e51572396b0 centos "/bin/bash" 10 minutes ago Up 10 minutes 0.0.0.0:99->99/tcp CentOS2

#使用 inspect 查看镜像信息

[root@localhost ~]# docker inspect 6e51572396b0

"Mounts": [

{

"Type": "bind",

"Source": "/data",

"Destination": "/data",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

]

2.匿名卷和命名卷的区别

命名卷在用过一次之后以后挂载容器的时候还是可以继续使用，所以一般在需要保存数据的时候使用命名卷的方式，下面例子用以展示命名卷对于数据的保存方式：

#1.查看目前的镜像

[root@localhost ~]# docker images | grep centos

centos latest 831691599b88 6 weeks ago 215MB

centos 7.5.1804 cf49811e3cdb 16 months ago 200MB

#2.启动centos镜像并且使用命名挂载的方式进行数据卷挂载(此centos是官网下载的官方镜像包)

[root@localhost ~]# docker run -it -d --name CentOS3 -v /data:/data 831691599b88

193f127b532350e7e9e3b1031e5d20a615cf95e2dd5f1aad762962739d279196

#3.查看刚刚启动的容器id

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

193f127b5323 831691599b88 "/bin/bash" 5 seconds ago Up 4 seconds CentOS3

#4.进入容器内部

[root@localhost ~]# docker exec -it 193f127b5323 /bin/bash

#5.查看/usr/local/,发现多了data文件夹

[root@193f127b5323 /]# ls -l

total 0

lrwxrwxrwx. 1 root root 7 May 11 2019 bin -> usr/bin

drwxr-xr-x. 3 root root 24 Jul 29 02:15 data

drwxr-xr-x. 5 root root 360 Jul 29 02:39 dev

drwxr-xr-x. 1 root root 66 Jul 29 02:39 etc

drwxr-xr-x. 2 root root 6 May 11 2019 home

lrwxrwxrwx. 1 root root 7 May 11 2019 lib -> usr/lib

lrwxrwxrwx. 1 root root 9 May 11 2019 lib64 -> usr/lib64

drwx------. 2 root root 6 Jun 11 02:35 lost+found

drwxr-xr-x. 2 root root 6 May 11 2019 media

drwxr-xr-x. 2 root root 6 May 11 2019 mnt

drwxr-xr-x. 2 root root 6 May 11 2019 opt

dr-xr-xr-x. 132 root root 0 Jul 29 02:39 proc

dr-xr-x---. 2 root root 162 Jun 11 02:35 root

drwxr-xr-x. 11 root root 163 Jun 11 02:35 run

lrwxrwxrwx. 1 root root 8 May 11 2019 sbin -> usr/sbin

drwxr-xr-x. 2 root root 6 May 11 2019 srv

dr-xr-xr-x. 13 root root 0 Jul 28 06:47 sys

drwxrwxrwt. 7 root root 145 Jun 11 02:35 tmp

drwxr-xr-x. 12 root root 144 Jun 11 02:35 usr

drwxr-xr-x. 20 root root 262 Jun 11 02:35 var

#6.创建一个AnHui文件

[root@193f127b5323 /]# cd /data/

[root@193f127b5323 data]# touch AnHui

[root@193f127b5323 data]# vi AnHui

[root@193f127b5323 data]# cat AnHui

HeFei WuHu AnQing FuYang

#7.使用 ctrl+p+Q 不停止容器退出

#8.进行宿主机 容器卷 所在路径

[root@localhost ~]# cd /data/

#9.发现多了一个AnHui文件，说明数据挂载成功

[root@localhost data]# ll

total 4

-rw-r--r--. 1 root root 25 Jul 29 10:44 AnHui

[root@localhost data]# cat AnHui

HeFei WuHu AnQing FuYang

#10.此时将启动的容器停掉：

[root@localhost data]# docker stop CentOS3

CentOS3

#11.然后重新进入该容器

[root@localhost ~]# docker run -it -d --name CentOS4 -v /data:/data 831691599b88

660c6fc790767e368affeffafdc7461eec235bf60724dfdbac55b483ff627bba

[root@localhost ~]#

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

660c6fc79076 831691599b88 "/bin/bash" 32 seconds ago Up 31 seconds CentOS4

#12.直接到 /data路径

[root@localhost ~]# docker exec -it CentOS4 /bin/bash

[root@660c6fc79076 /]# cd /data/

#13.查看该AnHui文件

[root@660c6fc79076 data]# ls

AnHui

[root@660c6fc79076 data]# cat AnHui

HeFei WuHu AnQing FuYang

匿名卷则是随着容器的建立而建立，随着容器的关闭而消亡。匿名卷一般用来存储无关痛痒的数据。

#1.查看镜像

[root@localhost ~]# docker images |grep centos

centos latest 831691599b88 6 weeks ago 215MB

centos 7.5.1804 cf49811e3cdb 16 months ago 200MB

#2.使用匿名挂载的方式只指定容器内数据卷所在路径启动镜像

[root@localhost ~]# docker run -it -d --name CentOS5 -v /data 831691599b88

99b30d162402486db917ec74f197b6cc12160fb3e28db532f0090d5697d47830

#3.进入 容器内数据卷所在路径

[root@localhost ~]# docker run -it -d --name CentOS5 -v /data 831691599b88

99b30d162402486db917ec74f197b6cc12160fb3e28db532f0090d5697d47830

[root@localhost ~]# docker exec -it CentOS5 /bin/bash

[root@99b30d162402 /]# cd /data/

#4.编辑一个文件

[root@99b30d162402 data]# vi JiangSu

[root@99b30d162402 data]# cat JiangSu

NanJing SuZhou YangZhou XuZhou

#5.使用ctrl+p+Q命令不停止并退出容器

#6.查看当前容器数据卷挂载在宿主机的文件路径

[root@localhost ~]# docker ps | grep CentOS5

99b30d162402 831691599b88 "/bin/bash" 4 minutes ago Up 4 minutes CentOS5

[root@localhost ~]# docker inspect 99b30d162402

#7.从显示的信息中找到Mounts

"Mounts": [

{

"Type": "volume",

"Name": "a8772d94db3e22f7abe09a05b554a5ac3a55e4ed170d9b110c5dcf1b5a570599",

"Source": "/var/lib/docker/volumes/a8772d94db3e22f7abe09a05b554a5ac3a55e4ed170d9b110c5dcf1b5a570599/\_data",

"Destination": "/data",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

]

#8.进入宿主机挂载的路径目录中

[root@localhost ~]# cd /var/lib/docker/volumes/a8772d94db3e22f7abe09a05b554a5ac3a55e4ed170d9b110c5dcf1b5a570599/\_data

#9.发现该目录下也生成了JiangSu文件

[root@localhost \_data]# ll -h

total 4.0K

-rw-r--r--. 1 root root 31 Jul 29 11:02 JiangSu

[root@localhost \_data]# cat JiangSu

NanJing SuZhou YangZhou XuZhou

#10.关闭当前容器并重新使用匿名方式启动

[root@localhost ~]# docker run -it -d --name CentOS6 -v /data 831691599b88

1b551fafd3444afc4d42d35e086fbc74bc7e40f0d4901c5946a23510de1d7a58

#11.进入容器

[root@localhost ~]# docker exec -it CentOS6 /bin/bash

#12.发现重新启动之后，该数据卷之下的数据没有了，但是命名卷在重启之后数据就还在

[root@1b551fafd344 /]# cd /data/

[root@1b551fafd344 data]# ls -l

total 0

八、外部访问容器(Docker端口映射)

Docker端口映射

docker容器在启动的时候，如果不指定端口映射参数，在容器外部是无法通过网络来访问容器内的网络应用和服务的。

亦可使用Dockerfile文件中的EXPOSE指令来配置。

端口映射可使用-p、-P来实现：

-p指定要映射的端口，一个指定端口上只可以绑定一个容器

-P将容器内部开放的网络端口随机映射到宿主机的一个端口上

端口映射支持的格式：

ip:hostport:containerport #指定ip、指定宿主机port、指定容器port

ip::containerport #指定ip、未指定宿主机port(随机)、指定容器port

hostport:containerport #未指定ip、指定宿主机port、指定容器port

端口的映射有以下3种方法：

1.将容器暴露的所有端口，都随机映射到宿主机上。(不推荐使用)

应用实例：

# docker run -it -d -P nginx

[root@k8s-node1 ~]# docker run -it -d -P -p nginx

9df989372b6a98b68765cb4e2f23aa0306d51e5fe5f982218bd860ba1fc5282a

[root@k8s-node1 ~]# docker ps |grep nginx

9df989372b6a nginx "nginx -g 'daemon of…" 11 seconds ago Up 9 seconds 0.0.0.0:32769->80/tcp affectionate\_lamport

2.将容器指定端口映射到宿主机的指定端口上。

应用实例：将容器的80端口映射到宿主机的8000端口上

# docker run -it -d -p 8000:80 nginx

[root@k8s-node1 ~]# docker run -it -d -p 8000:80 nginx

ef8f82a16d2cecc6cd5420215330bf464fd4dacaa2aa10758a1e1349f5fe5b61

[root@k8s-node1 ~]# docker ps |grep nginx

ef8f82a16d2c nginx "nginx -g 'daemon of…" 4 seconds ago Up 2 seconds 0.0.0.0:8000->80/tcp trusting\_turing

3.将容器的ip和端口，指定映射到宿主机上

应用实例：将容器的ip192.168.70.129和80端口，映射到宿主机的9000端口

# docker run -p 192.168.70.129:9000:80 -it -d nginx

[root@k8s-node1 ~]# docker run -p 192.168.70.129:9000:80 -it -d nginx

0095f89706223233e326ae314afcafa058fdca958b02665a9e33e1b326b44d84

[root@k8s-node1 ~]# docker ps |grep nginx

0095f8970622 nginx "nginx -g 'daemon of…" 14 seconds ago Up 13 seconds 192.168.70.129:9000->80/tcp musing\_borg

[root@k8s-node1 ~]# netstat -tulnp|grep 9000

tcp 0 0 192.168.70.129:9000 0.0.0.0:\* LISTEN 9767/docker-proxy

参考文献：

https://segmentfault.com/a/1190000008557309

http://book.ashan.org/docker\_practice/image/list.html

https://www.cnblogs.com/sanduzxcvbnm/p/11390126.html

https://www.hangge.com/blog/cache/detail\_2410.html

http://www.kancloud.cn:8080/woshigrey/docker/935037

https://www.runoob.com/docker/docker-build-command.html