**云服务器免费领取**

1. 访问https://developer.aliyun.com/plan/student，进行学生认证，免费领取1个月使用权限。
2. 创建实例时，选择系统版本CentOS7.9
3. 创建实例后，使用VNC方式登录（先修改密码再登录，详情参考https://help.aliyun.com/document\_detail/25433.html）
4. 使用VNC方式登录后，修改系统配置文件内容（参考https://help.aliyun.com/document\_detail/469713.html?spm=a2c4g.442434.0.0.49187fe6Fmv4KE）
5. 设置实例密码，设置后可使用密码通过SSH方式（如Xshell等）远程登录云服务器实例



**一、Docker概述**

**1. Docker为什么出现**

1.避免环境差异：传统的软件开发中，应用程序和它所依赖的库和运行环境通常需要在目标环境中进行安装和配置才能运行。这种方式存在很多问题，比如在不同的环境中部署和运行同一个软件包可能会遇到依赖性和兼容性的问题。Docker的出现解决了这个问题，使用容器技术将应用程序和它所依赖的库和运行环境打包成一个独立的容器，这个容器可以在任何支持Docker的环境中运行，而不需要进行额外的安装和配置。

2.提高开发效率：Docker可以实现快速创建和销毁容器，开发人员可以在不同的容器中进行应用程序的开发、测试和部署，从而提高开发效率。

3.提高系统资源利用率：Docker容器是轻量级的，可以在同一台主机上运行多个相互独立的应用程序，从而提高系统资源的利用率。

4.便于管理：Docker提供了一系列工具和服务，使得容器的构建、部署和管理变得更加容易和自动化。

5.支持跨平台：Docker可以在不同的操作系统和云平台上运行，从而支持跨平台的应用程序部署。

**Tips：**

（1）在容器技术出来之前，我们都是使用虚拟机技术！

（2）虚拟机：在window中装一个Vmware，通过这个软件我们可以虚拟出来一台或者多台电脑！（很笨重）

（3）虚拟机也是属于虚拟化技术，Docker容器技术，也是一种虚拟化技术！

**（4）虚拟机和Docker：**

1. vm：隔离，需要开启多个虚拟机！linux centos原生镜像（一个电脑！）需要几个G内存空间，开启需要几分钟！
2. docker：隔离，镜像（最核心的环境4m内存）十分的小巧，运行镜像就可以了！小巧！几个M或KB的内存空间，秒级启动！

**（5）聊聊Docker**

Docker是基于Go语言开发的！开源项目！

官网：[https://www.docker.com/](https://www.docker.com/" \t "_blank)

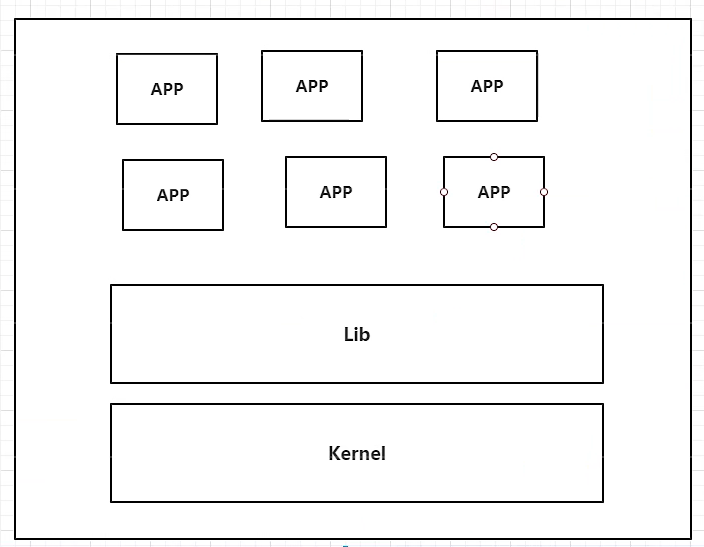
文档地址：[https://docs.docker.com/](https://docs.docker.com/" \t "_blank)

Doker镜像仓库地址：[https://hub.docker.com/](https://hub.docker.com/" \t "_blank)

视频教程：<https://www.bilibili.com/video/BV1og4y1q7M4>

**2. Docker能做什么**

**（1）之前的虚拟机技术:**



2.虚拟机技术缺点：

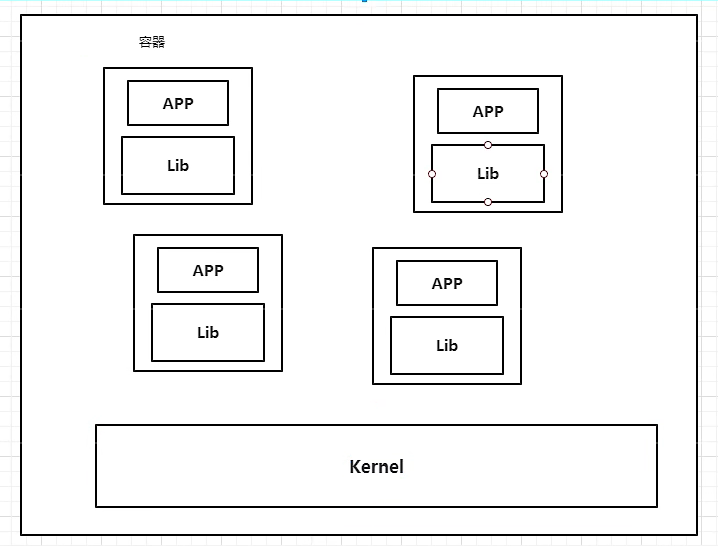
(1)资源占用十分多

(2)冗余步骤多

(3)启动很慢！

**（2）容器化技术**

**容器化技术不是模拟的一个完整的操作系统**



**（3）比较Docker和虚拟机技术的不同：**

* 传统虚拟机，虚拟出一条硬件，运行一个完整的操作系统，然后在这个系统上安装和运行软件。
* 容器内的应用直接运行在宿主机的内核中，容器是没有自己的内核的，也没有虚拟我们的硬件，所以就轻便了。
* 每个容器间是互相隔离，每个容器内都有一个属于自己的文件系统，互不影响。

**（4）Devops（开发、运维）**

**应用更快速的交付和部署**

传统：一堆帮助文档，安装程序!

Docker：打包镜像发布测试，一键运行!

**更便捷的升级和扩容**

使用了Docker之后，我们部署应用就和搭积木一样！

项目打包为一个镜像，扩展服务器A！服务器B!

**更简单的系统运维**

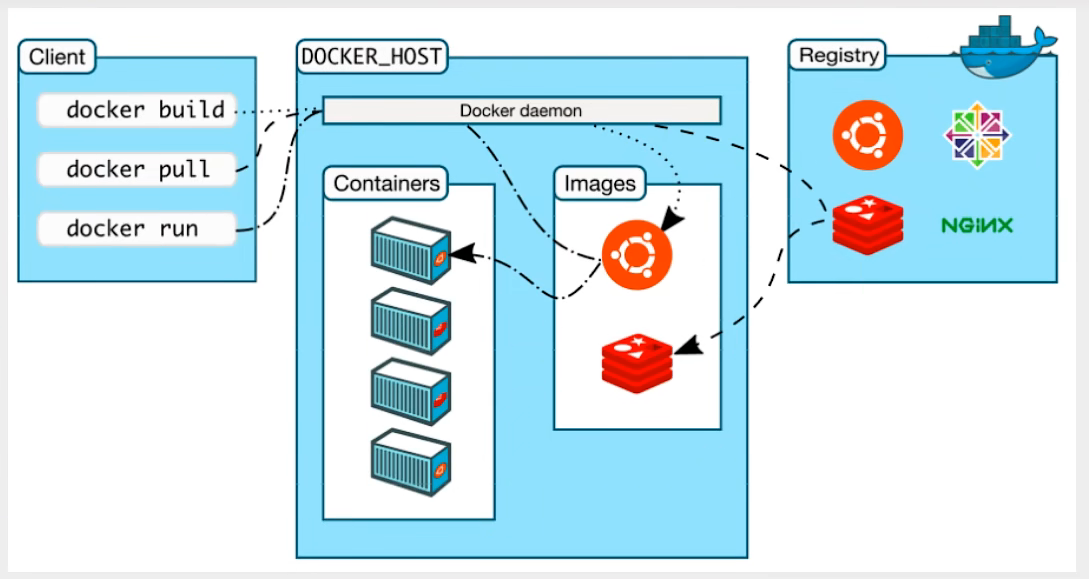
在容器化之后，我们的开发，测试环境都是高度一致的。

**更高效的计算资源利用**

Docker是内核级别的虚拟化，可以在一个物理机上可以运行很多的容器实例！

**二、Docker安装**

**1. Docker的基本组成**



**（1）镜像（image）：**

docker镜像就好比是一个模板，可以通过这个模板来创建容器服务，tomcat镜像===>run==>tomcat01容器（提供服务器），通过这个镜像可以创建多个容器（最终服务运行或者项目运行就是在容器中的）。

**（2）容器（container）：**

Docker利用容器技术，独立运行一个或者一个组应用，通过镜像来创建的。

启动，停止，删除，基本命令！

目前就可以把这个容器理解为就是一个简易的linux系统

**（3）仓库（repository）：**

仓库就是存放镜像的地方！

仓库分为公有仓库和私有仓库！

Docker Hub（默认是国外的）阿里云.…都有容器服务器（配置镜像加速！）

**2. 安装Docker**

**（1）环境准备**

1. CentOS7
2. 我们使用Xshell连接远程服务器进行操作

**（2）环境查看**

1. # 系统内核是 3.10 以上的
2. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# uname -r
3. 3.10.0-1160.66.1.el7.x86\_64
4. # 查看系统版本
5. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# cat /etc/os-release
6. NAME="CentOS Linux"
7. VERSION="7 (Core)"
8. ID="centos"
9. ID\_LIKE="rhel fedora"
10. VERSION\_ID="7"
11. PRETTY\_NAME="CentOS Linux 7 (Core)"
12. ANSI\_COLOR="0;31"
13. CPE\_NAME="cpe:/o:centos:centos:7"
14. HOME\_URL="https://www.centos.org/"
15. BUG\_REPORT\_URL="https://bugs.centos.org/"
16. CENTOS\_MANTISBT\_PROJECT="CentOS-7"
17. CENTOS\_MANTISBT\_PROJECT\_VERSION="7"
18. REDHAT\_SUPPORT\_PRODUCT="centos"
19. REDHAT\_SUPPORT\_PRODUCT\_VERSION="7"

**（3）安装**

查看文档：[https://docs.docker.com/](https://docs.docker.com/" \t "_blank) （[https://docs.docker.com/engine/install/centos/）](https://docs.docker.com/engine/install/centos/%EF%BC%89" \t "_blank)

1. # 1.卸载旧的版本

yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

9.# 2.需要的安装包

10.yum install -y yum-utils

11.# 3.设置镜像的仓库

yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo # 默认是从国外的。

15.

yum-config-manager \

--add-repo \

http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo # 推荐使用阿里云的。

19.

20.

21.# 安装容器之前，更新yum软件包索引。

yum makecache fast

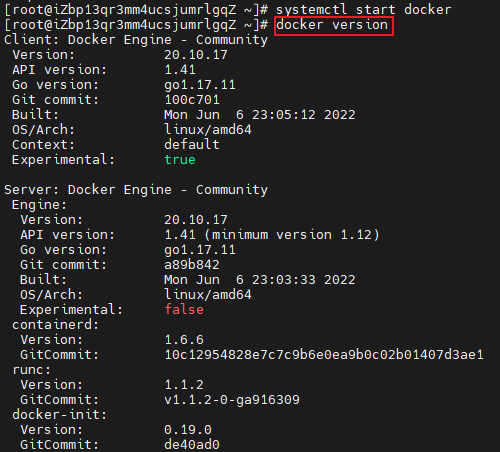
23.# 4.安装容器相关的。docker-ce（社区版）docker-ee（企业版）

yum install docker-ce docker-ce-cli containerd.io (如果jenkins有问题，先执行sudo yum-config-manager --disable jenkins)

26.# 5.启动docker

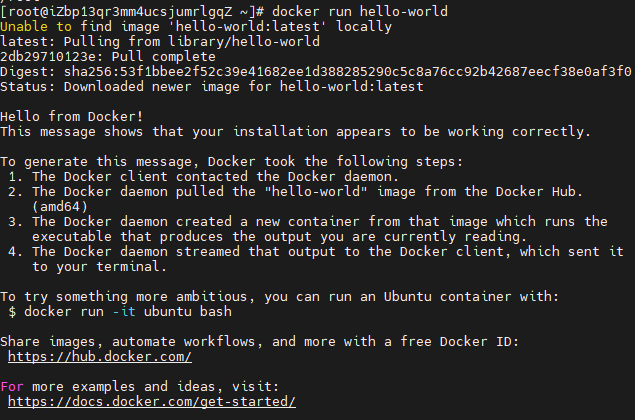
systemctl start docker

28.# 6.使用docker version查看是否安装成功



# 7.测试hello-world

docker run hello-world



# 8.查看一下下载的这个hello-world镜像

**（4）了解：卸载docker**

# 1.卸载依赖

yum remove docker-ce docker-ce-cli containerd.io

# 2.删除资源

rm -rf /var/lib/docker

rm -rf /var/lib/containerd

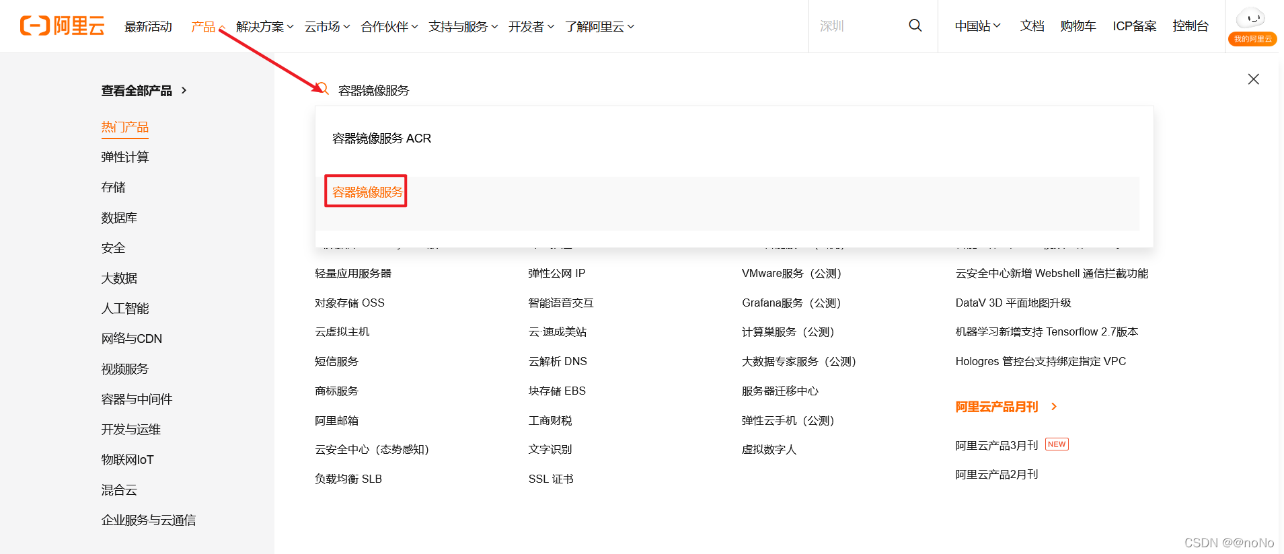
# /var/lib/docker docker的默认工作路径

1. **阿里云镜像加速（不稳定，目前已失效）**

替代方案：

无需加速（需支持访问外网）或国内能用的镜像源（如https://docker.m.daocloud.io/，参考：[https://github.com/DaoCloud/public-image-mirror）](https://github.com/DaoCloud/public-image-mirror))

1、登录阿里云（地址：https://www.aliyun.com），产品标签栏搜索“容器镜像服务”

2、进入容器镜像服务的“管理控制台”

3、选择“镜像工具”-“镜像加速器”

4、复制加速地址

5、配置

sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://xxx.xxx.xxx.com"]

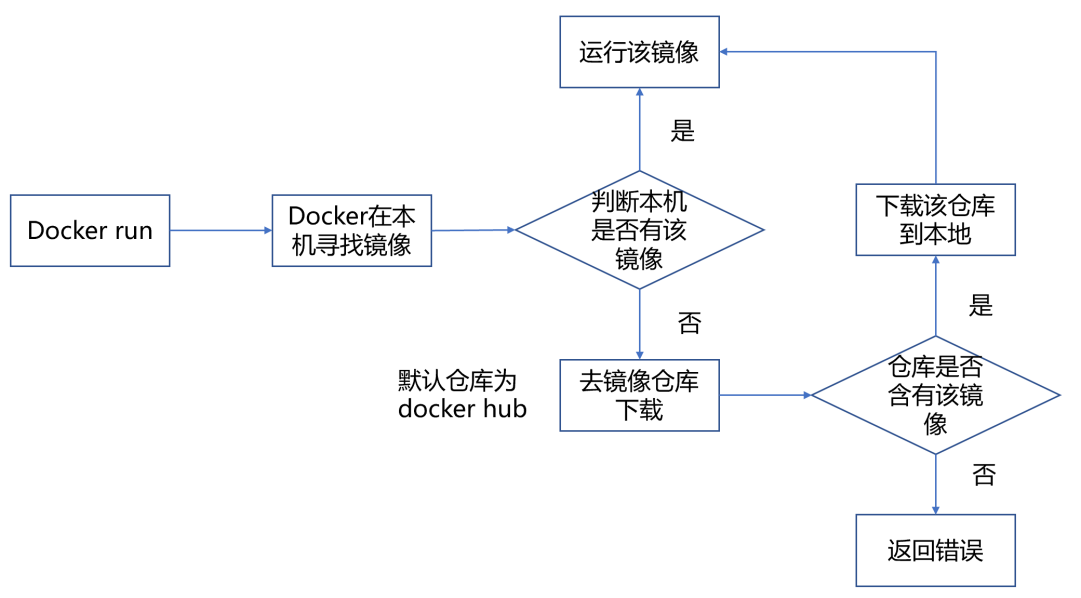
}

EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

**4. 回顾hello-world流程**

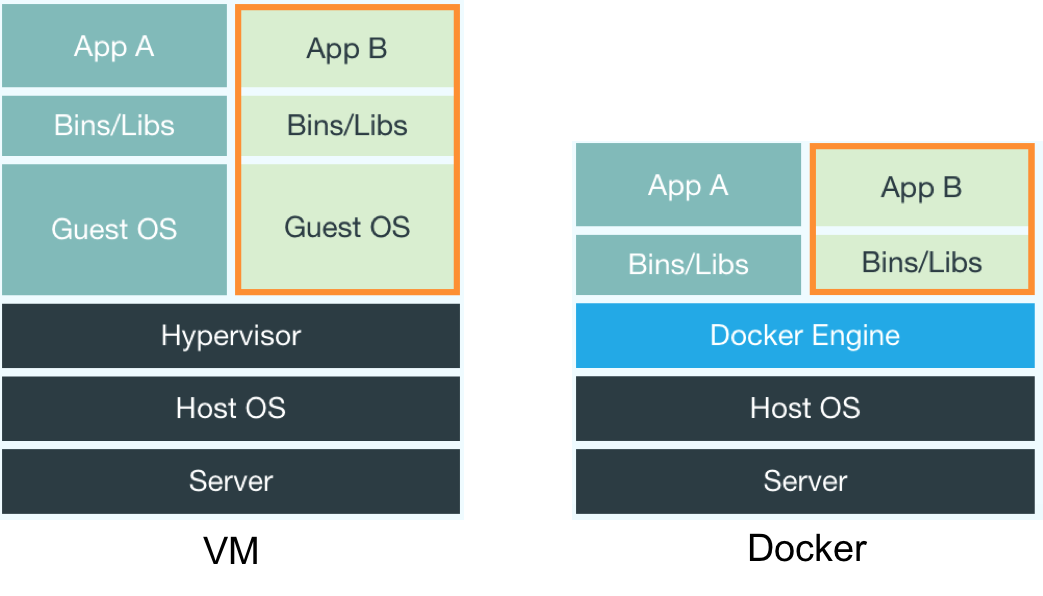


**5. 底层原理**

**Docker为什么比VM快？**

1、Docker有着比虚拟机更少的抽象层。

2、docker利用的是宿主机的内核，vm需要是Guest OS。



所以说，新建一个容器的时候，docker不需要像虚拟机一样重新加载一个操作系统内核，避免引导。

虚拟机是加载GuestOS，分钟级别的，而docker是利用宿主机的操作系统，省略了这个复杂的过程，秒级！

**三、Docker的常用命令**

**1. 帮助命令**

docker version # 显示docker的版本信息

docker info # 显示docker的系统信息，包括镜像和容器的数量

docker 命令 --help # 帮助命令

帮助文档的地址：[https://docs.docker.com/reference/](https://docs.docker.com/reference/" \t "_blank)

**2. 镜像命令**

**（1）dokcer images**

查看所有本地的主机上的镜像

[root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

hello-world latest feb5d9fea6a5 10 months ago 13.3kB

# 解释

REPOSITORY 镜像的仓库源

TAG 镜像的标签

IMAGE ID 镜像的id

CREATED 镜像的创建时间

SIZE 镜像的大小

# 命令参数可选项

-a, --all # 显示所有镜像 (docker images -a)

-q, --quiet # 仅显示镜像id (docker images -q)

**（2）docker search**

搜索镜像

[root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker search mysql

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

mysql MySQL is a widely used, open-source relation… 12940 [OK]

mariadb MariaDB Server is a high performing open sou… 4957 [OK]

phpmyadmin phpMyAdmin - A web interface for MySQL and M… 587 [OK]

percona Percona Server is a fork of the MySQL relati… 582 [OK]

# 解释

# 命令参数可选项 (通过搜索来过滤)

--filter=STARS=3000 # 搜索出来的镜像就是stars大于3000的

[root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker search mysql --filter=STARS=3000

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

mysql MySQL is a widely used, open-source relation… 12941 [OK]

mariadb MariaDB Server is a high performing open sou… 4957 [OK]

**（3）docker pull**

下载镜像

1. # 下载镜像：docker pull 镜像名[:tag]
2. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker pull mysql
3. Using default tag: latest # 如果不写tag，默认就是latest，最新的版本
4. latest: Pulling from library/mysql
5. 72a69066d2fe: Pull complete # 分层下载，docker image的核心，联合文件下载
6. 93619dbc5b36: Pull complete
7. 99da31dd6142: Pull complete
8. 626033c43d70: Pull complete
9. 37d5d7efb64e: Pull complete
10. ac563158d721: Pull complete
11. d2ba16033dad: Pull complete
12. 688ba7d5c01a: Pull complete
13. 00e060b6d11d: Pull complete
14. 1c04857f594f: Pull complete
15. 4d7cfa90e6ea: Pull complete
16. e0431212d27d: Pull complete
17. Digest: sha256:e9027fe4d91c0153429607251656806cc784e914937271037f7738bd5b8e7709 #签名
18. Status: Downloaded newer image for mysql:latest
19. docker.io/library/mysql:latest # 真实地址
20. # 两个命令是等价的
21. docker pull mysql
22. docker pull docker.io/library/mysql:latest
23. # 指定版本下载
24. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker pull mysql:5.7
25. 5.7: Pulling from library/mysql
26. 72a69066d2fe: Already exists # 联合文件下载，已经存在的资源可以共用
27. 93619dbc5b36: Already exists
28. 99da31dd6142: Already exists
29. 626033c43d70: Already exists
30. 37d5d7efb64e: Already exists
31. ac563158d721: Already exists
32. d2ba16033dad: Already exists
33. 0ceb82207cd7: Pull complete
34. 37f2405cae96: Pull complete
35. e2482e017e53: Pull complete
36. 70deed891d42: Pull complete
37. Digest: sha256:f2ad209efe9c67104167fc609cca6973c8422939491c9345270175a300419f94
38. Status: Downloaded newer image for mysql:5.7
39. docker.io/library/mysql:5.7

**（4）docker rmi**

删除镜像

1. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker rmi -f 镜像id # 删除指定的镜像
2. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker rmi -f 镜像id 镜像id 镜像id # 删除多个镜像（空格分隔）
3. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker rmi -f $(docker images -aq) # 删除全部的镜像

**3. 容器命令**

说明：我们有了镜像才可以创建容器，linux，下载一个centos 镜像来测试学习。

1. docker pull centos

**（1）新建容器并启动**

1. docker run [可选参数] image
2. # 参数说明
3. --name="name" 容器名字：用来区分容器
4. -d 后台方式运行：相当于nohup
5. -it 使用交互式运行：进入容器查看内容
6. -p 指定容器的端口（四种方式）小写字母p
7. -p ip:主机端口：容器端口
8. -p 主机端口：容器端口
9. -p 容器端口
10. -P 随机指定端口（大写字母P）
11. # 测试：启动并进入容器
12. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker run -it centos /bin/bash
13. [root@526c31d2c298 /]# ls # 查看容器内的centos（基础版本，很多命令都是不完善的）
14. bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var
15. # 从容器中退回到主机
16. [root@526c31d2c298 /]# exit
17. exit
18. [root@iZbp13qr3mm4ucsjumrlgqZ /]# ls
19. bin boot dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var www

**（2）列出所有运行的容器**

1. docker ps # 列出当前正在运行的容器
2. # 命令参数可选项
3. -a # 列出当前正在运行的容器+历史运行过的容器
4. -n=? # 显示最近创建的容器（可以指定显示几条，比如-n=1）
5. -q # 只显示容器的编号
6. [root@iZbp13qr3mm4ucsjumrlgqZ /]# docker ps
7. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8. [root@iZbp13qr3mm4ucsjumrlgqZ /]# docker ps -a
9. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
10. 526c31d2c298 centos "/bin/bash" 4 minutes ago Exited (0) 2 minutes ago optimistic\_allen
11. ce0eb11fbf8a feb5d9fea6a5 "/hello" 4 hours ago Exited (0) 4 hours ago keen\_ellis
12. [root@iZbp13qr3mm4ucsjumrlgqZ /]# docker ps -a -n=1
13. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
14. 526c31d2c298 centos "/bin/bash" 5 minutes ago Exited (0) 3 minutes ago optimistic\_allen

**（3）退出容器**

1. exit # 容器直接停止，并退出
2. ctrl+P+Q # 容器不停止，退出
3. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker run -it centos /bin/bash
4. [root@c5d61aa9d7df /]# [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker ps
5. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
6. c5d61aa9d7df centos "/bin/bash" 56 seconds ago Up 55 seconds
7. kind\_clarke
8. [root@iZbp13qr3mm4ucsjumrlgqZ ~]#

**（4）删除容器**

1. docker rm 容器id # 删除容器（不能删除正在运行的容器）如果要强制删除：docker rm -f 容器id
2. docker rm -f $(docker ps -aq) # 删除全部容器
3. docker ps -a -q|xargs docker rm # 删除所有容器

**（5）启动和停止容器的操作**

1. docker start 容器id # 启动容器
2. docker restart 容器id # 重启容器
3. docker stop 容器id # 停止当前正在运行的容器
4. docker kill 容器id # 强制停止当前容器

**4. 常用其他命令**

**（1）后台启动容器**

1. # 命令docker run -d 镜像名
2. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker run -d centos
3. # 问题：docker ps发现centos停止了
4. # 常见的坑：docker容器使用后台运行，就必须要有要一个前台进程，docker发现没有应用，就会自动停止。

**（2）查看日志**

1. docker logs -tf --tail 容器id
2. # 自己编写一段shell脚本

docker run -d centos /bin/sh -c "while true;do echo myshell;sleep 1;done"

1. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker logs -tf 容器id
2. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker logs -tf --tail 10 容器id
3. # 显示日志
4. -tf # 显示日志
5. --tail number # 要显示的日志条数

**（3）查看容器中进程的信息**

1. # 命令 docker top 容器id
2. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker top 88d23bcbe1f2
3. UID PID PPID C STIME TTY TIME CMD
4. root 21212 21193 0 16:23 ? 00:00:00
5. root 21600 21212 0 16:29 ? 00:00:00

**（4）查看镜像的元数据**

1. # 命令docker inspect 容器id
2. [
3. {
4. "Id": "88d23bcbe1f28e1f8ae5d2b63fa8d57d2abcbdacf193db05852f4f74a95b9ffe",
5. "Created": "2022-07-29T08:23:56.862239223Z",
6. "Path": "/bin/sh",
7. "Args": [
8. "-c",
9. "while true;do echo kuangshen;sleep 1;done"
10. ],
11. "State": {
12. "Status": "running",
13. "Running": true,
14. "Paused": false,
15. "Restarting": false,
16. "OOMKilled": false,
17. "Dead": false,
18. "Pid": 21212,
19. "ExitCode": 0,
20. "Error": "",
21. "StartedAt": "2022-07-29T08:23:57.109766809Z",
22. "FinishedAt": "0001-01-01T00:00:00Z"
23. },
24. "Image": "sha256:5d0da3dc976460b72c77d94c8a1ad043720b0416bfc16c52c45d4847e53fadb6",
25. "ResolvConfPath": "/var/lib/docker/containers/88d23bcbe1f28e1f8ae5d2b63fa8d57d2abcbdacf193db05852f4f74a95b9ffe/resolv.conf",
26. "HostnamePath": "/var/lib/docker/containers/88d23bcbe1f28e1f8ae5d2b63fa8d57d2abcbdacf193db05852f4f74a95b9ffe/hostname",
27. "HostsPath": "/var/lib/docker/containers/88d23bcbe1f28e1f8ae5d2b63fa8d57d2abcbdacf193db05852f4f74a95b9ffe/hosts",
28. "LogPath": "/var/lib/docker/containers/88d23bcbe1f28e1f8ae5d2b63fa8d57d2abcbdacf193db05852f4f74a95b9ffe/88d23bcbe1f28e1f8ae5d2b63fa8d57d2abcbdacf193db05852f4f74a95b9ffe-json.log",
29. "Name": "/silly\_lichterman",
30. "RestartCount": 0,
31. "Driver": "overlay2",
32. "Platform": "linux",
33. "MountLabel": "",
34. "ProcessLabel": "",
35. "AppArmorProfile": "",
36. "ExecIDs": null,
37. "HostConfig": {
38. "Binds": null,
39. "ContainerIDFile": "",
40. "LogConfig": {
41. "Type": "json-file",
42. "Config": {}
43. },
44. "NetworkMode": "default",
45. "PortBindings": {},
46. "RestartPolicy": {
47. "Name": "no",
48. "MaximumRetryCount": 0
49. },
50. "AutoRemove": false,
51. "VolumeDriver": "",
52. "VolumesFrom": null,
53. "CapAdd": null,
54. "CapDrop": null,
55. "CgroupnsMode": "host",
56. "Dns": [],
57. "DnsOptions": [],
58. "DnsSearch": [],
59. "ExtraHosts": null,
60. "GroupAdd": null,
61. "IpcMode": "private",
62. "Cgroup": "",
63. "Links": null,
64. "OomScoreAdj": 0,
65. "PidMode": "",
66. "Privileged": false,
67. "PublishAllPorts": false,
68. "ReadonlyRootfs": false,
69. "SecurityOpt": null,
70. "UTSMode": "",
71. "UsernsMode": "",
72. "ShmSize": 67108864,
73. "Runtime": "runc",
74. "ConsoleSize": [
75. 0,
76. 0
77. ],
78. "Isolation": "",
79. "CpuShares": 0,
80. "Memory": 0,
81. "NanoCpus": 0,
82. "CgroupParent": "",
83. "BlkioWeight": 0,
84. "BlkioWeightDevice": [],
85. "BlkioDeviceReadBps": null,
86. "BlkioDeviceWriteBps": null,
87. "BlkioDeviceReadIOps": null,
88. "BlkioDeviceWriteIOps": null,
89. "CpuPeriod": 0,
90. "CpuQuota": 0,
91. "CpuRealtimePeriod": 0,
92. "CpuRealtimeRuntime": 0,
93. "CpusetCpus": "",
94. "CpusetMems": "",
95. "Devices": [],
96. "DeviceCgroupRules": null,
97. "DeviceRequests": null,
98. "KernelMemory": 0,
99. "KernelMemoryTCP": 0,
100. "MemoryReservation": 0,
101. "MemorySwap": 0,
102. "MemorySwappiness": null,
103. "OomKillDisable": false,
104. "PidsLimit": null,
105. "Ulimits": null,
106. "CpuCount": 0,
107. "CpuPercent": 0,
108. "IOMaximumIOps": 0,
109. "IOMaximumBandwidth": 0,
110. "MaskedPaths": [
111. "/proc/asound",
112. "/proc/acpi",
113. "/proc/kcore",
114. "/proc/keys",
115. "/proc/latency\_stats",
116. "/proc/timer\_list",
117. "/proc/timer\_stats",
118. "/proc/sched\_debug",
119. "/proc/scsi",
120. "/sys/firmware"
121. ],
122. "ReadonlyPaths": [
123. "/proc/bus",
124. "/proc/fs",
125. "/proc/irq",
126. "/proc/sys",
127. "/proc/sysrq-trigger"
128. ]
129. },
130. "GraphDriver": {
131. "Data": {
132. "LowerDir": "/var/lib/docker/overlay2/59b088d44f6e67f4ed336de44d19a5784c1a13fa856760bd2b166c4a4d421e2b-init/diff:/var/lib/docker/overlay2/7fc43e24b63e4656ab7e7718d3e4ef5297fe82509452be305a01605a7cdc3b97/diff",
133. "MergedDir": "/var/lib/docker/overlay2/59b088d44f6e67f4ed336de44d19a5784c1a13fa856760bd2b166c4a4d421e2b/merged",
134. "UpperDir": "/var/lib/docker/overlay2/59b088d44f6e67f4ed336de44d19a5784c1a13fa856760bd2b166c4a4d421e2b/diff",
135. "WorkDir": "/var/lib/docker/overlay2/59b088d44f6e67f4ed336de44d19a5784c1a13fa856760bd2b166c4a4d421e2b/work"
136. },
137. "Name": "overlay2"
138. },
139. "Mounts": [],
140. "Config": {
141. "Hostname": "88d23bcbe1f2",
142. "Domainname": "",
143. "User": "",
144. "AttachStdin": false,
145. "AttachStdout": false,
146. "AttachStderr": false,
147. "Tty": false,
148. "OpenStdin": false,
149. "StdinOnce": false,
150. "Env": [
151. "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
152. ],
153. "Cmd": [
154. "/bin/sh",
155. "-c",
156. "while true;do echo kuangshen;sleep 1;done"
157. ],
158. "Image": "centos",
159. "Volumes": null,
160. "WorkingDir": "",
161. "Entrypoint": null,
162. "OnBuild": null,
163. "Labels": {
164. "org.label-schema.build-date": "20210915",
165. "org.label-schema.license": "GPLv2",
166. "org.label-schema.name": "CentOS Base Image",
167. "org.label-schema.schema-version": "1.0",
168. "org.label-schema.vendor": "CentOS"
169. }
170. },
171. "NetworkSettings": {
172. "Bridge": "",
173. "SandboxID": "bfc28d2ba671adfe5c93325173b93c335d50d8b017eed4fdce2ab75410d6ac2e",
174. "HairpinMode": false,
175. "LinkLocalIPv6Address": "",
176. "LinkLocalIPv6PrefixLen": 0,
177. "Ports": {},
178. "SandboxKey": "/var/run/docker/netns/bfc28d2ba671",
179. "SecondaryIPAddresses": null,
180. "SecondaryIPv6Addresses": null,
181. "EndpointID": "ad0252d454e751e2c5ecef24e64c32b0884c53242925bd66e9e5dbf5542af179",
182. "Gateway": "172.17.0.1",
183. "GlobalIPv6Address": "",
184. "GlobalIPv6PrefixLen": 0,
185. "IPAddress": "172.17.0.2",
186. "IPPrefixLen": 16,
187. "IPv6Gateway": "",
188. "MacAddress": "02:42:ac:11:00:02",
189. "Networks": {
190. "bridge": {
191. "IPAMConfig": null,
192. "Links": null,
193. "Aliases": null,
194. "NetworkID": "7254ffccbdf53e0c72c1d19252980f04fa65153ea3c7ca72af55cfac504cbe3f",
195. "EndpointID": "ad0252d454e751e2c5ecef24e64c32b0884c53242925bd66e9e5dbf5542af179",
196. "Gateway": "172.17.0.1",
197. "IPAddress": "172.17.0.2",
198. "IPPrefixLen": 16,
199. "IPv6Gateway": "",
200. "GlobalIPv6Address": "",
201. "GlobalIPv6PrefixLen": 0,
202. "MacAddress": "02:42:ac:11:00:02",
203. "DriverOpts": null
204. }
205. }
206. }
207. }
208. ]

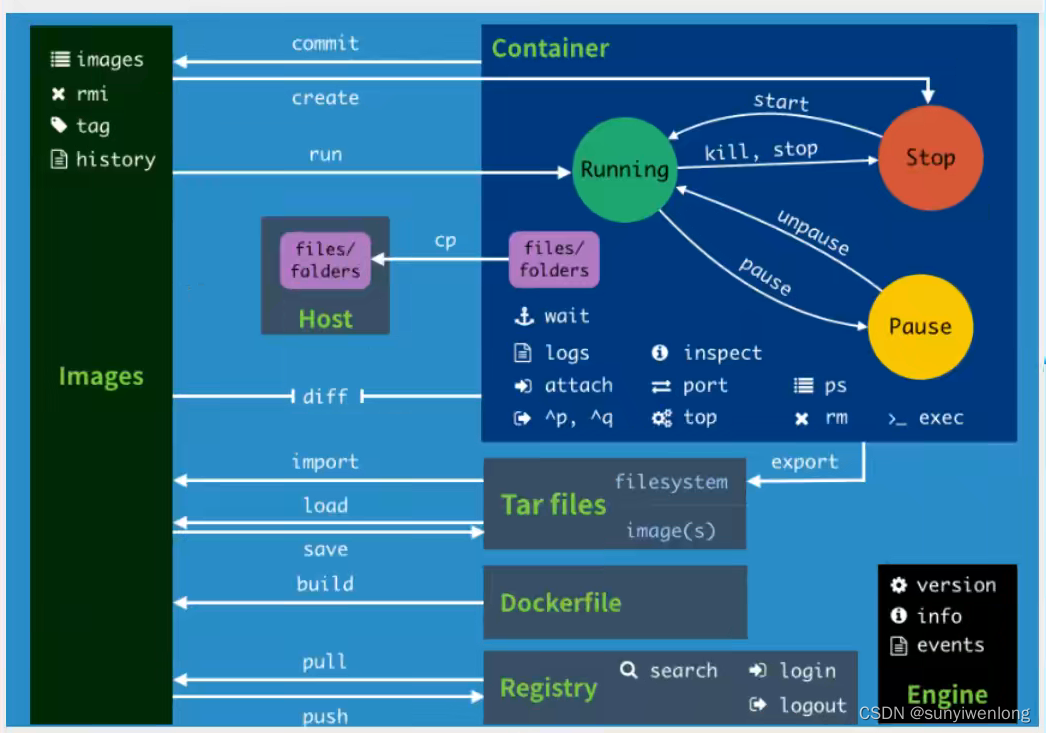
**（5）进入当前正在运行的容器**

1. # 我们通常容器都是使用后台方式运行的，需要进入容器，修改一些配置
2. # 命令
3. docker exec -it 容器id /bin/bash
4. # 测试
5. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker ps
6. CONTAINER ID IMAGE COMMAND CREATED STATUS
7. PORTS NAMES
8. 88d23bcbe1f2 centos "/bin/sh -c 'while t…" 13 minutes ago Up 13 minutes silly\_lichterman
9. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker exec -it 88d23bcbe1f2 /bin/bash
10. [root@88d23bcbe1f2 /]# ps -ef
11. UID PID PPID C STIME TTY TIME CMD
12. root 1 0 0 08:23 ? 00:00:00 /bin/sh -c while true;do echo kuangshen;sleep 1;done
13. root 841 0 0 08:37 pts/0 00:00:00 /bin/bash
14. root 858 1 0 08:37 ? 00:00:00 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1
15. root 859 841 0 08:37 pts/0 00:00:00 ps -ef
16. # 方式二
17. docker attach 容器id
18. # 测试
19. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker attach 88d23bcbe1f2
20. 正在执行当前的代码...
21. # docker exec # 进入容器后开启一个新的终端，可以再里面操作（常用）
22. # docker attach # 进入容器正在执行的终端，不会启动新的进程。

**（6）从容器内拷贝文件到主机上**

1. docker cp 容器id:容器内路径 目的主机的路径
2. [root@iZbp13qr3mm4ucsjumrlgqZ home]# ll
3. total 0
4. [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker ps
5. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
6. [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker run -it centos /bin/bash
7. [root@6eda31ad7987 /]# [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker ps
8. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9. 6eda31ad7987 centos "/bin/bash" 17 seconds ago Up 16 seconds stoic\_kepler
10. # 进入到容器内部
11. [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker attach 6eda31ad7987
12. [root@6eda31ad7987 /]# ls
13. bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var
14. [root@6eda31ad7987 /]# cd /home/
15. [root@6eda31ad7987 home]# ls
16. # 在容器的/home路径下创建test.java文件
17. [root@6eda31ad7987 home]# touch test.java
18. [root@6eda31ad7987 home]# ls
19. test.java
20. [root@6eda31ad7987 home]# exit
21. exit
22. [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker ps
23. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
24. [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker ps -a
25. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
26. 6eda31ad7987 centos "/bin/bash" About a minute ago Exited (0) 28 seconds ago stoic\_kepler
27. # 将文件拷贝出来到主机上（在主机上执行该命令）
28. [root@iZbp13qr3mm4ucsjumrlgqZ home]# docker cp 6eda31ad7987:/home/test.java /home
29. [root@iZbp13qr3mm4ucsjumrlgqZ home]# ls
30. test.java
31. # 拷贝是一个手动过程，未来我们使用 -v 卷的技术，可以实现，自动同步（容器内的/home路径和主机上的/home路径打通）

**5. 小结**

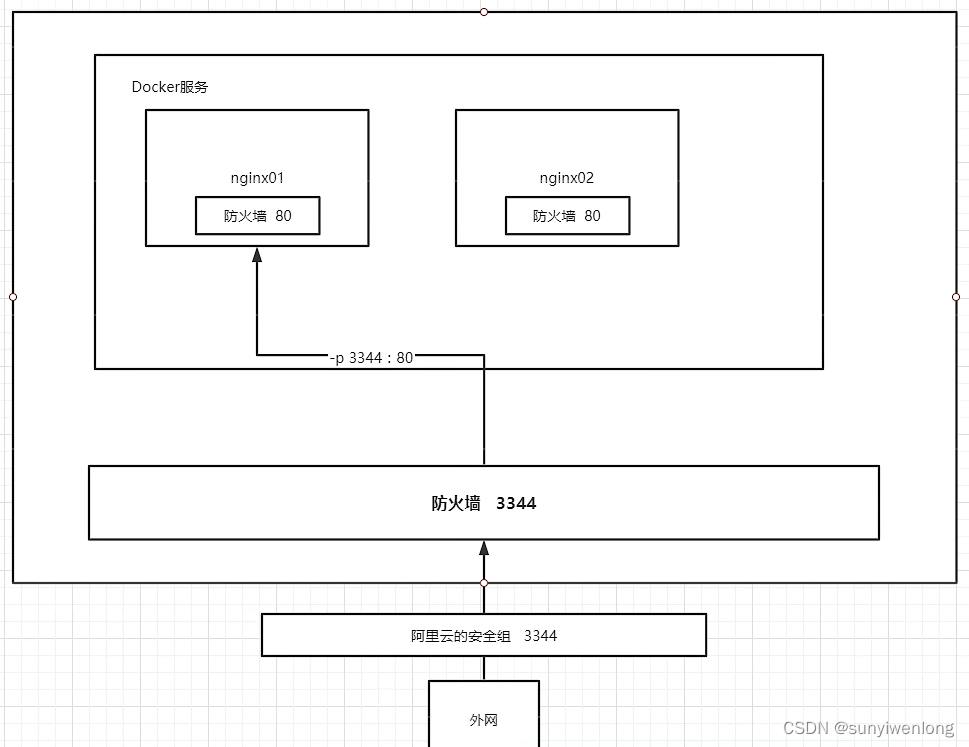


**6. 实验练习**

**（1）docker安装nginx**

1. 搜索镜像：docker search nginx (建议去dockerHub上去搜索)
2. 下载镜像：docker pull nginx
3. 启动nginx：
   1. # -d 后台运行
   2. # --name="nginx01" 给容器命名
   3. # -p 宿主机端口:容器内部端口
   4. [root[@iZbp13qr3mm4ucsjumrlgqZ](https://github.com/iZbp13qr3mm4ucsjumrlgqZ" \t "_blank" \o "@iZbp13qr3mm4ucsjumrlgqZ) ~]# docker run -d --name="nginx-1" -p 3344:80 nginx
   5. 6e02190a50bc8d79653ffa88f6b5c143d79c5ac3257d5d5ed6a01247980fb48a
   6. [root[@iZbp13qr3mm4ucsjumrlgqZ](https://github.com/iZbp13qr3mm4ucsjumrlgqZ" \t "_blank" \o "@iZbp13qr3mm4ucsjumrlgqZ) ~]# docker ps
   7. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
   8. 6e02190a50bc nginx "/docker-entrypoint.…" 18 seconds ago Up 17
   9. seconds 0.0.0.0:3344->80/tcp, :::3344->80/tcp nginx-1
   10. # 进入容器
   11. [root[@iZbp13qr3mm4ucsjumrlgqZ](https://github.com/iZbp13qr3mm4ucsjumrlgqZ" \t "_blank" \o "@iZbp13qr3mm4ucsjumrlgqZ) ~]# docker exec -it a1e130aa184d /bin/bash
   12. root[@a1e130aa184d](https://github.com/a1e130aa184d" \t "_blank" \o "@a1e130aa184d):/# whereis nginx
   13. nginx: /usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx
   14. root[@a1e130aa184d](https://github.com/a1e130aa184d" \t "_blank" \o "@a1e130aa184d):/# cd /etc/nginx/
   15. root[@a1e130aa184d](https://github.com/a1e130aa184d" \t "_blank" \o "@a1e130aa184d):/etc/nginx# ls
   16. conf.d fastcgi\_params mime.types modules nginx.conf scgi\_params uwsgi\_params
4. 本机测试
   1. [root[@iZbp13qr3mm4ucsjumrlgqZ](https://github.com/iZbp13qr3mm4ucsjumrlgqZ" \t "_blank" \o "@iZbp13qr3mm4ucsjumrlgqZ) ~]# curl localhost:3344
   2. <!DOCTYPE html>
   3. <html>
   4. <head>
   5. <title>Welcome to nginx!</title>
   6. <style>
   7. html { color-scheme: light dark; }
   8. body { width: 35em; margin: 0 auto;
   9. font-family: Tahoma, Verdana, Arial, sans-serif; }
   10. </style>
   11. </head>
   12. <body>
   13. <h1>Welcome to nginx!</h1>
   14. <p>If you see this page, the nginx web server is successfully installed and
   15. working. Further configuration is required.</p>
   16. <p>For online documentation and support please refer to
   17. <a href="http://nginx.org/">nginx.org</a>.<br/>
   18. Commercial support is available at
   19. <a href="http://nginx.com/">nginx.com</a>.</p>
   20. <p><em>Thank you for using nginx.</em></p>
   21. </body>
   22. </html>

端口暴露的概念



思考问题：我们每次改动nginx配置文件，都需要进入容器内部？十分的麻烦，我要是可以在容器外部提供一个映射路径，达到在容器外修改文件名，容器内部就可以自动修改？-v 数据卷

**（2）docker安装tomcat**

1. # 下载
2. docker pull tomcat
3. # 启动运行
4. docker run -d --name="tomcat01" -p 3355:8080 tomcat
5. # 测试访问，提示找不到资源
6. curl localhost:3355
7. # 进入容器，有一个webapps文件夹和webapps.dist文件夹
8. docker exec -it tomcat01 /bin/bash
9. # webapps文件夹下没有资源,所以直接访问404，资源都在webapp.dist文件夹下
10. root@35eb825661e0:/usr/local/tomcat# ls
11. BUILDING.txt CONTRIBUTING.md LICENSE NOTICE README.md RELEASE-NOTES RUNNING.txt bin conf lib logs native-jni-lib temp webapps webapps.dist work
12. # 发现问题：（阿里云镜像的原因：默认是最小的镜像，所有不必要的都剔除掉）保证最小可运行的环境
13. # 没有webapps文件夹，发现有一个webapps.dist文件夹，资源在webapps.dist文件夹下；
14. # 把webapps.dist文件夹下的文件复制到webapps文件夹下，就可以访问成功。
15. root@35eb825661e0:/usr/local/tomcat# cp -r webapps.dist/\* webapps
16. # 退出容器，重新访问curl localhost:3355，看到欢迎页面

**四、Docker镜像讲解**

**1. 镜像是什么**

镜像是一种轻量级、可执行的独立软件包，用来打包软件运行环境和基于运行环境开发的软件，它包含运行某个软件所需的所有内容，包括代码、运行时、库、环境变量和配置文件。

所有的应用，直接打包docker镜像，就可以直接跑起来！

如何得到镜像：

* 从远程仓库下载
* 自己制作一个镜像DockerFile

**2. Docker镜像加速原理**

**（1）UnionFS（联合文件系统）**

Union文件系统（UnionFS）是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下（unite several directories into a single virtual filesystem）。Union 文件系统是Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

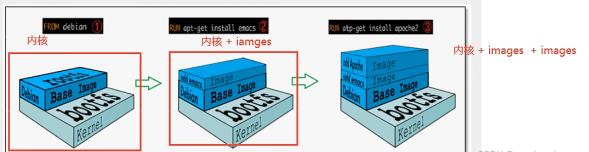
**（2）Docker镜像加载原型**

docker的镜像实际上由一层一层的文件系统组成，这种层级的文件系统UnionFS。

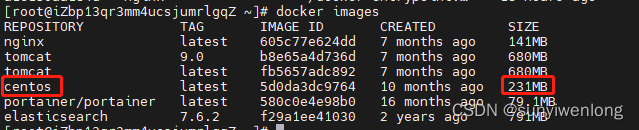
bootfs（boot file system）主要包含bootloader和kernel，bootloader主要是引导加载kernel，Linux刚启动时会加载bootfs文件系统，在Docker镜像的最底层是bootfs。这一层与我们典型的Linux/Unix系统是一样的，包含boot加载器和内核。当boot加载完成之后整个内核就都在内存中了，此时内存的使用权已由bootfs转交给内核，此时系统也会卸载bootfs。

rootfs（root file system），在bootfs之上。包含的就是典型Linux系统中的/dev，/proc，/bin，/etc等标准目录和文件。

rootfs就是各种不同的操作系统发行版，比如Ubuntu，Centos等等。



平时我们安装进虚拟机的CentOS都是好几个G，为什么Docker这里才200M？



对于一个精简的OS，rootfs可以很小，只需要包含最基本的命令，工具和程序库就可以了，因为底层直接用Host的kernel，自己只需要提供rootfs就可以了。

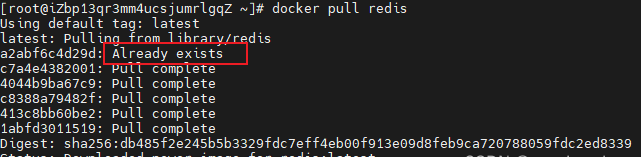
由此可见对于不同的linux发行版，bootfs基本是一致的，rootfs会有差别，因此不同的发行版可以公用bootfs。

（虚拟机是分钟级，容器是秒级）

**3. 分层理解**

**分层的镜像**

我们可以去下载一个镜像，注意观察下载的日志输出，可以看到是一层一层的在下载！



思考：为什么Docker镜像要采用这种分层的结构呢？

最大的好处，我觉得莫过于是资源共享了。比如有多个镜像都从相同的Base镜像构建而来，那么宿主机只需在磁盘上保留一份base镜像，同时内存中也只需要加载一份base镜像，这样就可以为所有的容器服务了，而且镜像的每一层都可以被共享。

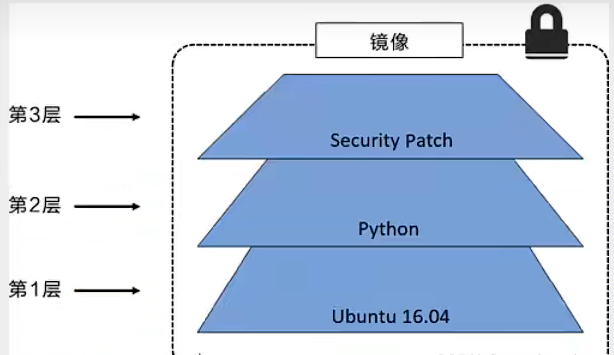
查看镜像分层的方式可以通过 docker image inspect命令！

**理解：**

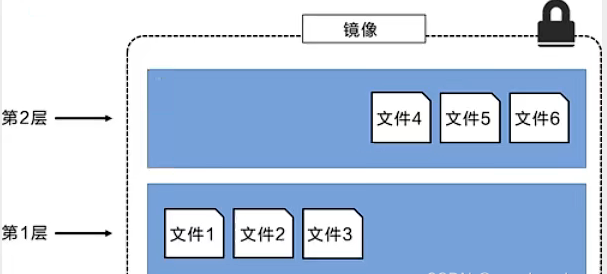
所有的Docker 镜像都起始于一个基础镜像层，当进行修改或增加新的内容时，就会在当前镜像层之上，创建新的镜像层。

举一个简单的例子，假如基于Ubuntu Linux 16.04创建一个新的镜像，这就是新镜像的第一层；如果在该镜像中添加Python包，就会在基础镜像层之上创建第二个镜像层；如果继续添加一个安全补丁，就会创建第三个镜像层。

该镜像当前已经包含3个镜像层，如下图所示。

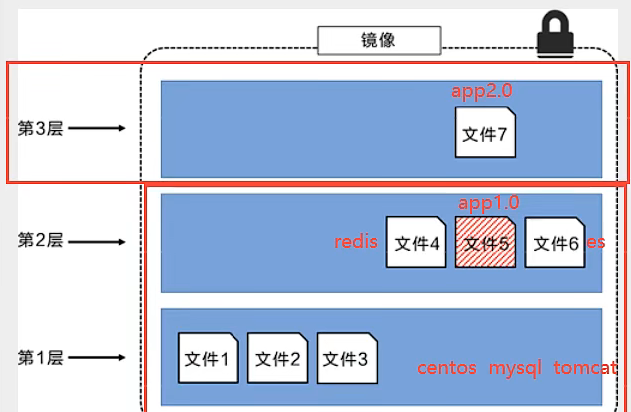


在添加额外的镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要。下图中举了一个简单的例子，每个镜像层包含3个文件，而镜像包含了来自两个镜像层的6个文件。



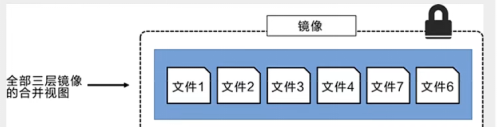
上图中的镜像层跟之前图中的略有区别，主要目的是便于展示文件。

下图中展示了一个稍微复杂的三层镜像，在外部看来整个镜像只有6个文件，这是因为最上层中的文件7是文件5的一个更新版本。



这种情况下，上层镜像层中的文件覆盖了底层镜像层中的文件。这样就使得文件的更新版本作为一个新镜像层添加到镜像当中。

下图展示了与系统显示相同的三层镜像。所有镜像层堆叠并合并，对外提供统一的视图。

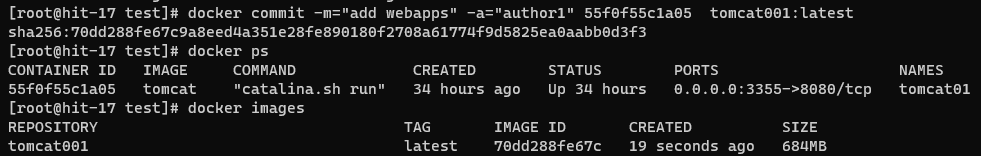


**4. commit镜像**

1. # 提交容器成为一个新的副本
2. docker commit
3. # 命令和git原理类似
4. docker commit -m="提交的描述信息" -a="作者" 容器id 目标镜像名:[TAG]

**实战测试**

1. # 将我们操作过的tomcat容器通过commit提交为一个镜像！我们以后就使用我们修改过的镜像即可，这就是我们自己的一个修改的镜像。



**五、容器数据卷**

**1. 什么是容器数据卷**

**docker的理念回顾**

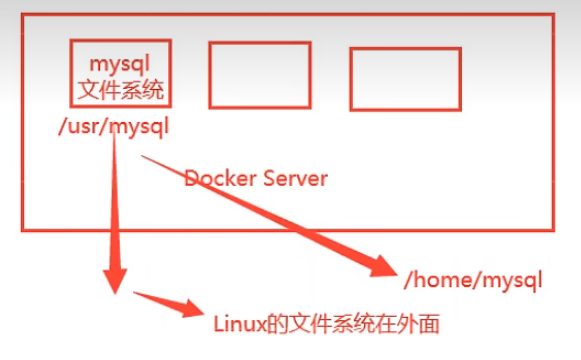
将应用和环境打包成一个镜像！

数据？如果数据都在容器中，那么我们容器删除，数据就会丢失！（需求：数据可以持久化）

MySQL，容器删了，删库跑路。（需求：MySQL数据可以存储在本地）

容器之间可以有一个数据共享的技术。Docker容器中产生的数据，同步到本地。

这就是卷技术。目录的挂载，将Linux主机的目录挂载到容器目录上，容器删除后，数据依然存在于主机目录上。



总结一句话：容器的持久化和同步操作！容器间也是可以数据共享的！

**2. 使用数据卷**

**直接使用命令来挂载：-v**

1. docker run -it -v 主机目录:容器内目录 镜像名 /bin/bash
2. # 测试，查看容器信息
3. docker inspect 容器id



测试文件的同步：

在容器的/home文件夹下，新建test.java文件，会同步到主机的/home/ceshi文件夹下。

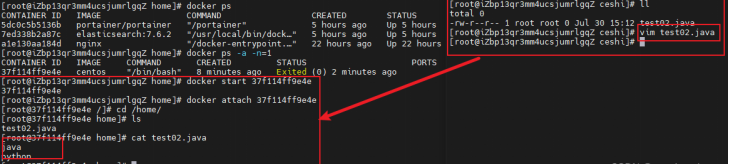
删除操作也是同步的；双向绑定，保证两边文件夹下的数据始终是一直的。



再来测试：

停止容器后，在主机的/home/ceshi文件夹下，修改文件或新增文件，启动容器，查看容器的/home文件夹，发现容器内的数据依旧是同步的

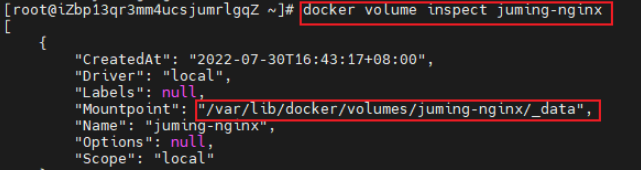
1. 停止容器。
2. 宿主机上修改文件。
3. 启动容器。
4. 容器内的数据依旧是同步的。



好处：我们以后修改只需要在本地修改即可，容器内会自动同步。

**3. 匿名和具名挂载**

1. # 匿名挂载
2. docker run -d -p --name nginx01 -v /etc/nginx nginx
3. # 查看所有的volume的情况
4. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker volume ls
5. DRIVER VOLUME NAME
6. local 964b8e505f12f65fb23fd21f05cfa9ecd6c2c6b2ca89c0e44f168bb017dfabd6
7. # 这种就是匿名挂载：我们在-v挂载目录时，只写了容器内的路径，没有写容器外的路径。
8. # 具名挂载
9. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker volume create juming-nginx [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker run -d -p 3346:80 --name nginx02 -v juming-nginx:/etc/nginx nginx
10. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker volume ls
11. DRIVER VOLUME NAME
12. local 964b8e505f12f65fb23fd21f05cfa9ecd6c2c6b2ca89c0e44f168bb017dfabd6
13. local juming-nginx
14. # 通过 -v 卷名:容器内的路径（具名挂载）
15. # 查看一下这个卷



所有的docker容器内的卷，没有指定目录的情况下都是在/var/lib/docker/volumes/xxxx/\_data"（xxxx是卷名）

我们通过具名挂载可以方便的找到我们的一个卷，大多数情况在使用的具名挂载和指定路径挂载

1. # 如何确定是具名挂载，还是匿名挂载，还是指定路径挂载
2. -v 容器内的路径 # 匿名挂载
3. -v 卷名:容器内的路径 # 具名挂载
4. -v /宿主机路径:容器内路径 # 指定路径挂载

**六、DockerFile**

**1. DockerFile介绍**

dockerfile是用来构建docker镜像的文件！命令参数脚本！

构建步骤：

1、编写一个dockerfile文件

2、docker build 构建成为一个镜像

3、docker run运行镜像

4、docker push发布镜像（DockerHub、阿里云镜像仓库）

**2. DockerFile的构建过程**

**基础知识：**

1、每个保留关键字（指令）都是必须是大写字母

2、执行从上到下顺序执行

3、# 表示注释

4、每一个指令都会创建提交一个新的镜像层，并提交！

Dockerfile是面向开发的，我们以后要发布项目，做镜像，就需要编写dockerfile文件。

DockerFile：构建文件，定义了一切的步骤，源代码。

Dockerlmages：通过DockerFile构建生成的镜像，最终发布和运行的产品。

Docker容器：容器就是镜像运行起来提供服务的。

**3. DockerFile的命令**

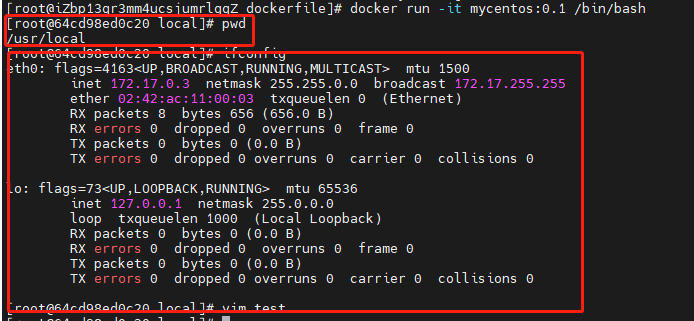
1. FROM # 基础镜像，一切从这里开始构建
2. MAINTAINER # 镜像是谁写的：姓名+邮箱
3. RUN # 镜像构建的时候需要运行的命令
4. ADD # 步骤：tomcat镜像，这个tomcat压缩包。添加内容
5. WORKDIR # 镜像的工作目录
6. VOLUME # 挂载的目录
7. EXPOSE # 暴露端口配置
8. CMD # 指定这个容器启动的时候要运行的命令，只有最后一个会生效，可被替代
9. ENTRYPOINT # 指定这个容器启动的时候要运行的命令，可以追加命令
10. ONBUILD # 当构建一个被继承DockerFile这个时候就会运行ONBUILD的指令。
11. COPY # 类似ADD，将我们文件拷贝到镜像中
12. ENV # 构建的时候设置环境变量

**4. 实战测试**

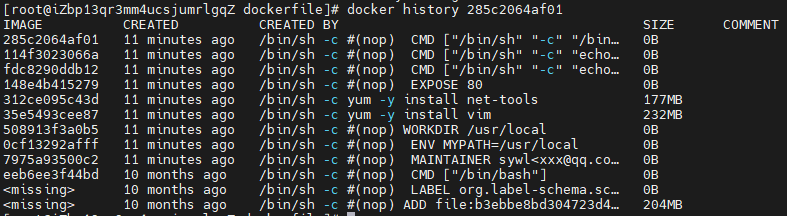
Docker Hub中99%镜像都是从这个基础镜像过来的FROM scratch，然后配置需要的软件和配置来进行的构建。

1. # 1. 创建dockerfile的文件
2. touch Dockerfile
3. # 2. 编写dockerfile的文件
4. FROM centos:7
5. MAINTAINER sywl<xxx@qq.com>
6. ENV MYPATH /usr/local
7. WORKDIR $MYPATH
8. RUN yum -y install vim
9. RUN yum -y install net-tools
10. EXPOSE 80
11. CMD echo $MYPATH
12. CMD echo "-----end-----"
13. CMD /bin/bash
14. # 2. 通过这个文件构建镜像
15. # 命令：docker build -f dockerfile文件路径 -t 镜像名:[tag]
16. docker build -f Dockerfile -t mycentos:0.1 .
17. Successfully built 285c2064af01
18. Successfully tagged mycentos:0.1
19. # 3. 测试运行

查看工作目录



我们可以列出本地进行的变更历史：docker history 镜像名+版本号

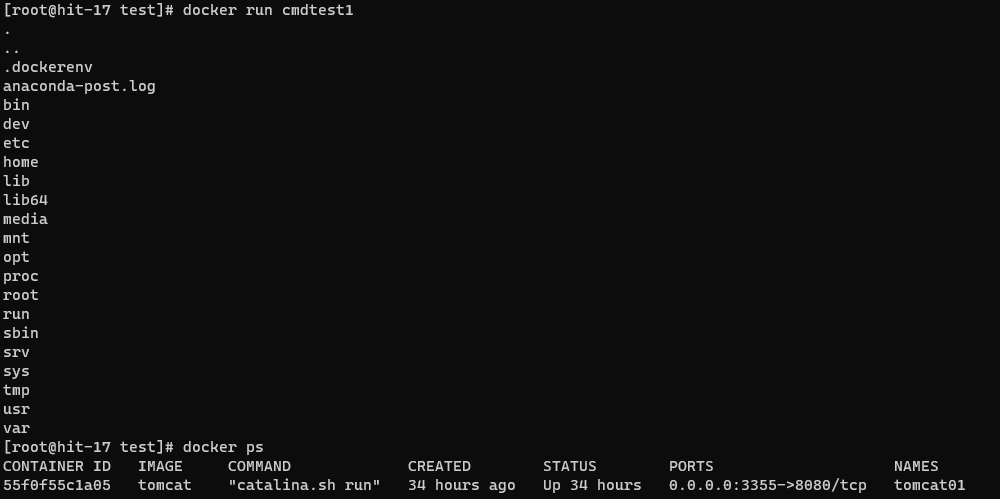


CMD和ENTRYPOINT区别

1. CMD # 指定这个容器启动的时候要运行的命令，只有最后一个会生效，可被替代
2. ENTRYPOINT # 指定这个容器启动的时候要运行的命令，可以追加命令

测试CMD

1. # 1. 编写dockerfile文件
2. [root@hit-17 test]# touch mydockerfile-cmd-test
3. [root@hit-17 test]# vim mydockerfile-cmd-test
4. FROM centos:7
5. CMD ["ls","-a"]
6. #保存并退出
7. # 2. 构建镜像
8. [root@hit-17 test]# docker build -f mydockerfile-cmd-test -t cmdtest1 .
9. # 3. run运行，发现我们的"ls -a"命令生效、执行



# 4. 然而，由于没有给容器映射任何端口或者提供其他的运行参数，"ls -a"命令很快就会执行完毕，容器也会立即退出。

# 5. 我们先追加一个命令"l",构成"ls -al"命令，发现报错

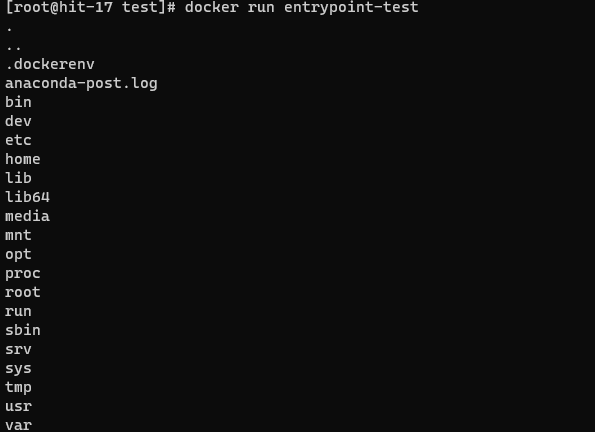
[root@hit-17 test]# docker run cmdtest1 -l

docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable to start container process: exec: "-l": executable file not found in $PATH: unknown.

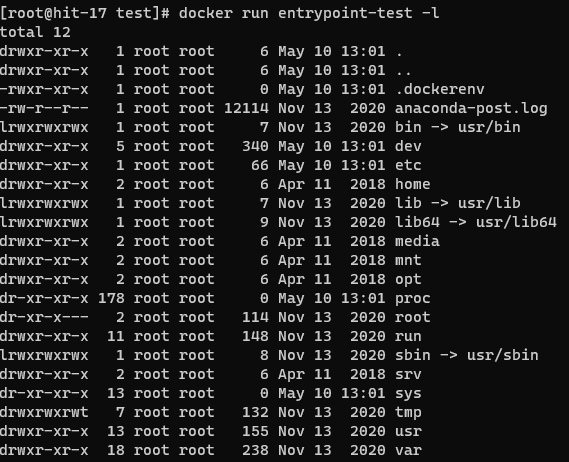
# 原因：CMD命令的情况下，"-l"替换了CMD["1s"，"-a"]命令，因为"-l"不是命令，所以报错！

测试ENTRYPOINT

1. # 1. 编写dockerfile文件
2. [root@hit-17 test]# touch mydockerfile-entrypoint-test
3. [root@hit-17 test]# vim mydockerfile-entrypoint-test
4. FROM centos:7
5. ENTRYPOINT ["ls","-a"]
6. # 2. 构建镜像
7. [root@hit-17 test]# docker build -f mydockerfile-entrypoint-test -t entrypoint-test .
8. # 3. run运行，发现我们的"ls -a"命令生效、执行



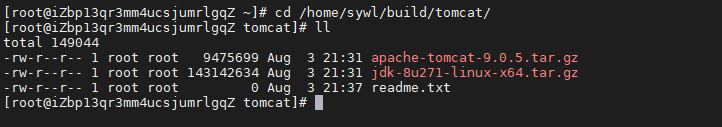
1. # 4. 我们先追加一个命令"l",构成"ls -al"命令，发现命令生效、执行



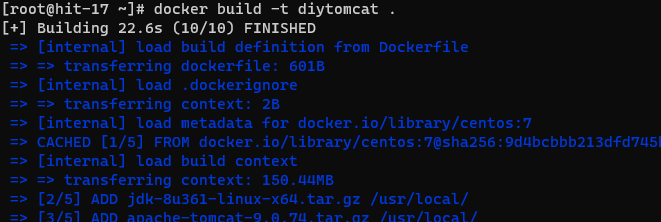
1. # 原因：ENTRYPOINT命令的情况下，"-l"追加在ENTRYPOINT ["1s"，"-a"]命令后面，得到"ls -al"的命令，所以命令正常执行！
2. # （我们的追加命令，是直接拼接在我们的ENTRYPOINT命令的后面）

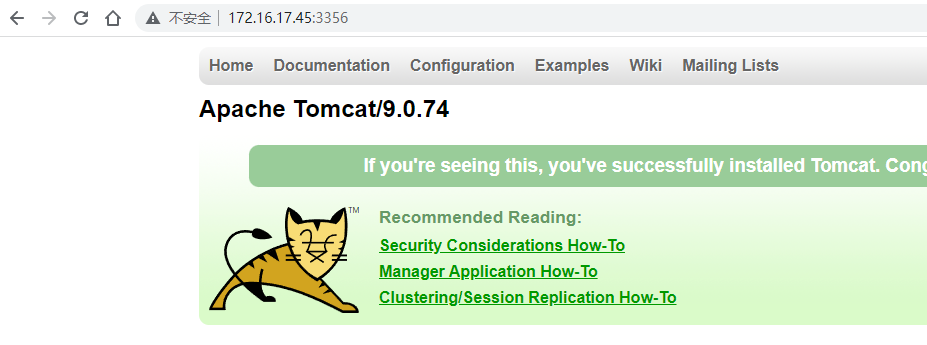
**5. 制作tomcat镜像**

1. 准备镜像文件：tomcat压缩包，jdk的压缩包！



1. 编写dockerfile文件，官方命名Dockerfile，build会自动寻找这个文件，就不需要-f指定文件名了。
   1. FROM centos:7
   2. ADD jdk-8u361-linux-x64.tar.gz /usr/local/
   3. ADD apache-tomcat-9.0.74.tar.gz /usr/local/
   4. RUN yum -y install vim
   5. RUN yum -y install iproute
   6. ENV MYPATH /usr/local
   7. WORKDIR $MYPATH
   8. ENV JAVA\_HOME /usr/local/jdk1.8.0\_361
   9. ENV CLASS\_PATH $JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar
   10. ENV CATALINA\_HOME /usr/local/apache-tomcat-9.0.74
   11. ENV CATALINA\_BASH /usr/local/apache-tomcat-9.0.74
   12. ENV PATH $PATH:$JAVA\_HOME/bin:$CATALINA\_HOME/lib:$CATALINA\_HOME/bin
   13. EXPOSE 8080
   14. CMD /usr/local/apache-tomcat-9.0.74/bin/startup.sh && tail -F /usr/local/apache-tomcat-9.0.74/bin/logs/catalina.out
2. 构建镜像
   1. docker build -t diytomcat .



1. 启动镜像
   1. [root@hit-17 ~]# docker run -d -p 3356:8080 --name mydiytomcat -v /root/test/tomcat/test:/usr/local/apache-tomcat-9.0.74/webapps/test -v /root/test/tomcat/tomcatlog:/usr/local/apache-tomcat-9.0.74/logs diytomcat
2. 访问测试
3. 
4. 发布项目（由于做了卷挂载，我们直接在本地test文件夹下编写项目就可以发布了！）
   1. # /root/test/tomcat/test/WEB-INF/web.xml文件
   2. <?xml version="1.0" encoding="UTF-8"?>
   3. <web-app xmlns="http://java.sun.com/xml/ns/javaee"
   4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"
   6. version="2.5">
   7. </web-app>
   8. # /root/test/tomcat/test/index.jsp文件
   9. <%@ page language="java" contentType="text/html; charset=UTF-8"
   10. pageEncoding="UTF-8"%>
   11. <!DOCTYPE html>
   12. <html>
   13. <head>
   14. <meta charset="utf-8">
   15. <title>hello,sywl</title>
   16. </head>
   17. <body>
   18. Hello World!<br/>
   19. <%
   20. System.out.println("----my web test----");
   21. %>
   22. </body>
   23. </html>

发现：项目部署成功，可以直接访问ok！（ip地址:3356/test）



我们以后开发的步骤：需要掌握Dokcerfile的编写！我们之后的一切都是使用docker镜像来发布运行！

**6. 发布自己的镜像**

**~~（1）发布到dockerhub上~~**

1.[https://hub.docker.com/](https://hub.docker.com/" \t "_blank) 注册自己的账号

2.确定这个账号可以登录

3.在我们服务器上提交自己的镜像

4.登录完毕后就可以提交镜像了，就是一步： docker push

. # 登录命令

i.[root@hit-17 test]# docker login -u xingyongchao

ii.Password:

iii.WARNING! Your password will be stored unencrypted in /root/.docker/config.json.

iv.Configure a credential helper to remove this warning. See

v.https://docs.docker.com/engine/reference/commandline/login/#credentials-store

vi.

vii.Login Succeeded

viii. # 给镜像增加一个tag

ix.[root@hit-17 test]# docker tag diytomcat:latest xingyongchao/diytomcat:latest

x. # docker push即可

xi. [root@hit-17 test]# docker push xingyongchao/diytomcat:latest

xii.The push refers to repository [docker.io/xingyongchao/diytomcat]

xiii.5f70bf18a086: Pushed

xiv.8828381bf6cc: Pushed

xv.85566e67147a: Pushed

xvi.904df27a01b9: Pushed

xvii.174f56854903: Pushed

xviii.latest: digest: sha256:e649659eb58baadb3ed9a62671acc0af2052f19eabdd934cd76cd51199e7ff2a size: 1372

**（2）Harbor安装**

这里采用原生的方式安装Harbor。

* 下载Harbor安装包：<https://github.com/goharbor/harbor/releases/download/v2.3.4/harbor-offline-installer-v2.3.4.tgz>
* 拖拽到Linux并解压：

tar -zxvf harbor-offline-installer-v2.3.4.tgz -C /usr/local/

* 修改Harbor配置文件：

首先复制一份harbor.yml配置

cp harbor.yml.tmpl harbor.yml

编辑harbor.yml配置文件

| * 配置Harbor文件 |
| --- |
| fig: |

* 启动Harbor

./install.sh

| * 查看日志 |
| --- |
| f3ff8609e38488a66d96830b46814c2 |

* 登录Harbor

| 登录Harbor (Harbor地址:端口 默认为80) |
| --- |
| fig: |

| * 首页信息 |
| --- |
| fig: |

* 根据说明编写推送指令



修改docker配置文件，将harbor的ip:port设为insecure-registries，允许发送http请求而不是https请求

sudo vi /etc/docker/daemon.json

添加：

"insecure-registries": ["172.16.17.45:8085"]

保存后重启docker

sudo systemctl restart docker

登录Harbor

docker login 172.16.17.45:8085

Username: admin

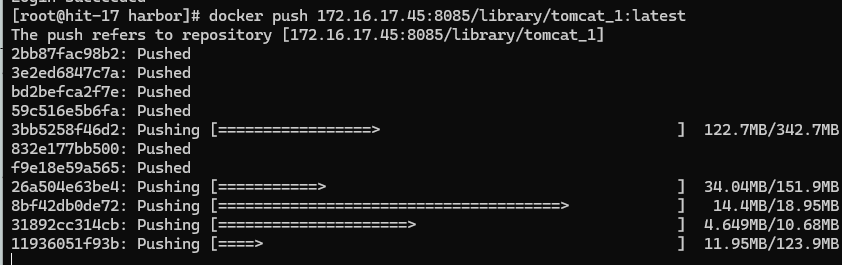
Password: Harbor12345(默认)

# 给镜像增加一个tag

[root@hit-17 test]# docker tag diytomcat:latest 172.16.17.45:8085/library/diytomcat:latest

* 推送

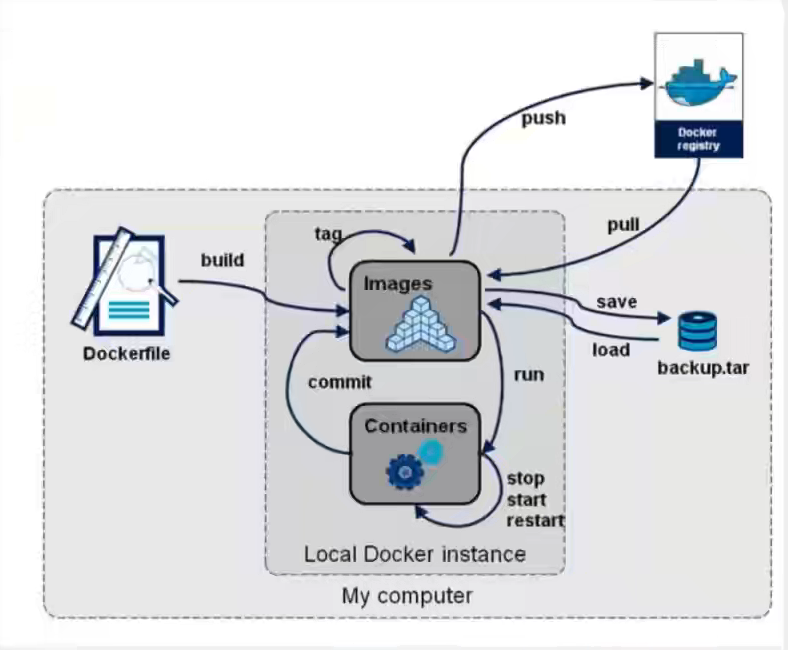
docker push 172.16.17.45:8085/library/diytomcat:latest



* 刷新harbor页面



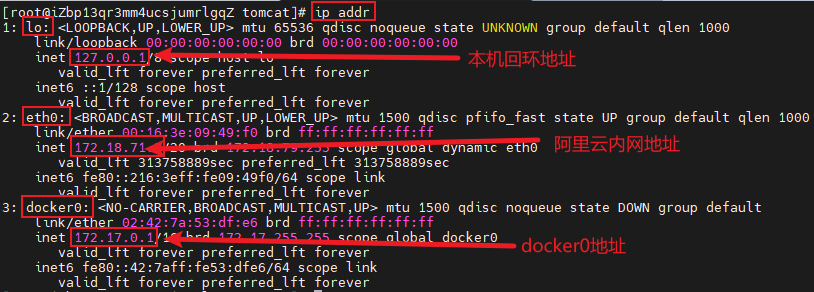
**7. 小结**



**七、Docker网络**

**1. 理解docker0**

**（1）测试**



有三个网络

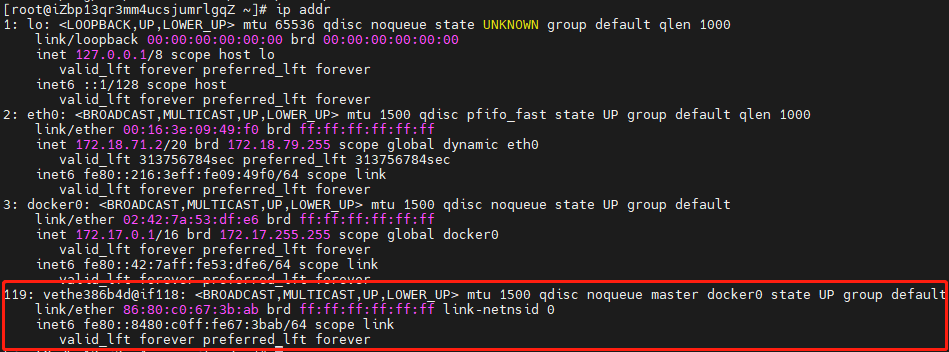
1. # 问题：docker是如何处理容器网络访问的？



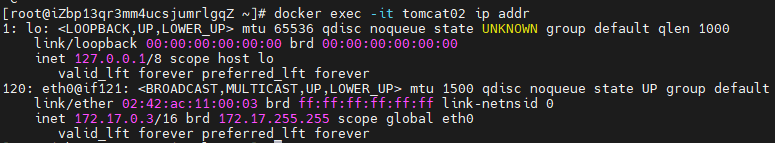
1. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker run -d -P --name tomcat01 tomcat
2. # 查看容器的内部网络地址ip addr，发现容器启动的时候会得到一个eth0@if119的ip地址（docker分配的）
3. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# docker exec -it tomcat01 ip addr
4. **yum install iproute**
5. 1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
6. link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
7. inet 127.0.0.1/8 scope host lo
8. valid\_lft forever preferred\_lft forever
9. 118: eth0@if119: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default
10. link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
11. inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
12. valid\_lft forever preferred\_lft forever
13. # 思考：liunx能不能ping通容器内部？（linux可以ping通容器内部）
14. [root@iZbp13qr3mm4ucsjumrlgqZ ~]# ping 172.17.0.2
15. PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
16. 64 bytes from 172.17.0.2: icmp\_seq=1 ttl=64 time=0.059 ms
17. 64 bytes from 172.17.0.2: icmp\_seq=2 ttl=64 time=0.046 ms
18. 64 bytes from 172.17.0.2: icmp\_seq=3 ttl=64 time=0.057 ms
19. 64 bytes from 172.17.0.2: icmp\_seq=4 ttl=64 time=0.045 ms

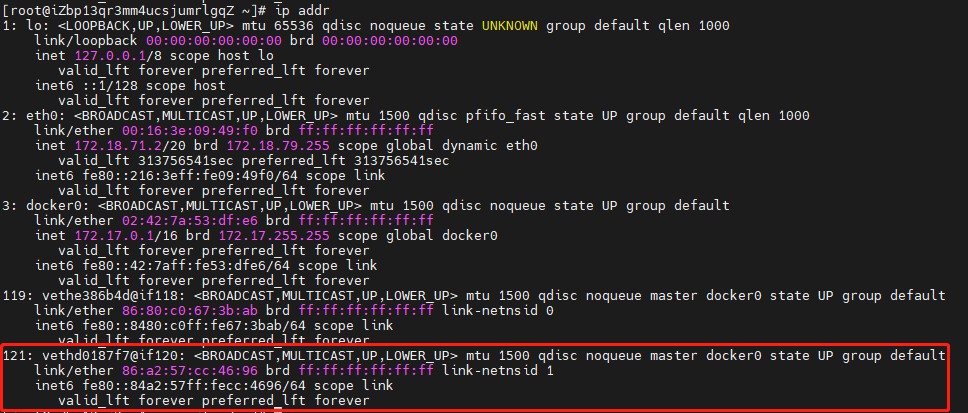
**（2）原理**

1. 我们每启动一个docker容器，docker就会给docker容器分配一个ip，我们只要安装了docker，就会有一个网卡docker0
2. （桥接模式，使用的技术是veth-pair技术）
3. 再次测试：



1. 在启动一个容器测试，发现又多了一对网卡。

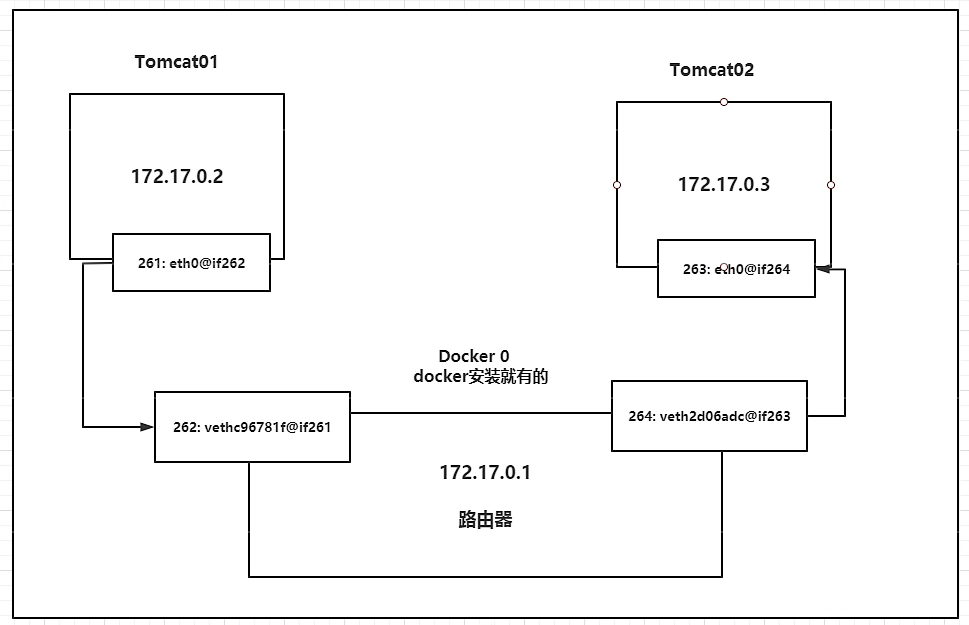




* 1. # 我们发现这个容器带来网卡，都是一对对的。
  2. # veth-pair 就是一对的虚拟设备接口，他们都是成对出现的，一端连着协议，一端彼此相连。
  3. # 正因为有这个特性，evth-pair充当一个桥梁，连接各种虚拟网络设备的。
  4. # openstack，Docker容器之间的连接，OVS的连接，都是使用veth-pair技术。

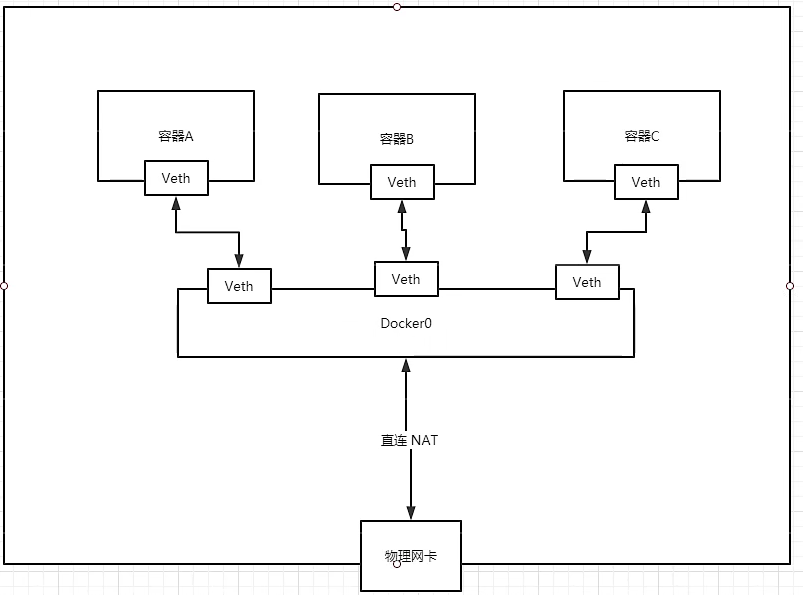
1. 我们来测试下tomcat01和tomcat02是否可以ping通！（可以ping通）
   1. # 由于docker pull tomcat 拉取的镜像不包括iputils-ping包， 所以无法在tomcat容器中使用ping，我们自己制作一个可以使用ping的tomcat镜像。
   2. # 先删除通过tomcat镜像启动的容器
   3. docker rm -f 容器id
   4. # 先前制作的diytomcat镜像可以使用ping，所以基于该镜像启动tomcat01和tomcat02
   5. docker run -d -p 3357:8080 --name tomcat01 diytomcat
   6. docker run -d -p 3358:8080 --name tomcat02 mydiytomcat
   7. [root[@iZbp13qr3mm4ucsjumrlgqZ](https://github.com/iZbp13qr3mm4ucsjumrlgqZ" \t "_blank" \o "@iZbp13qr3mm4ucsjumrlgqZ) ~]# docker exec -it tomcat02 ping 172.17.0.2
   8. PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
   9. 64 bytes from 172.17.0.2: icmp\_seq=1 ttl=64 time=0.094 ms
   10. 64 bytes from 172.17.0.2: icmp\_seq=2 ttl=64 time=0.055 ms
   11. # 结论：容器和容器之间是可以互相ping通的！

绘制一个网络模型图



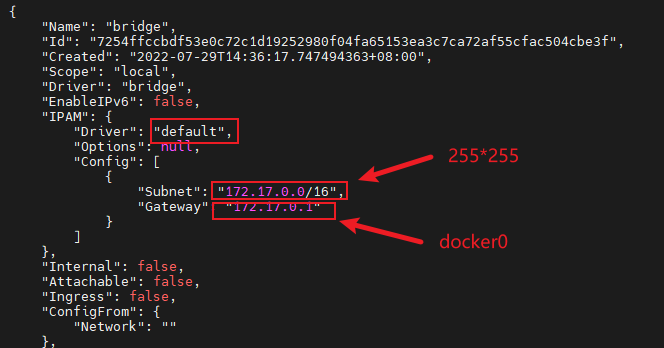
结论：tomcat01和tomcat02是公用的一个路由器，dockero。

所有的容器不指定网络的情况下，都是docker0路由的，docker会给我们的容器分配一个默认的可用IP



Docker中的所有的网络接口都是虚拟的。虚拟的转发效率高！（内网传递文件！）

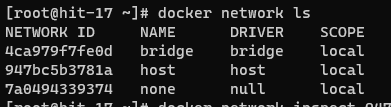
只要容器删除，对应网桥一对就没了！



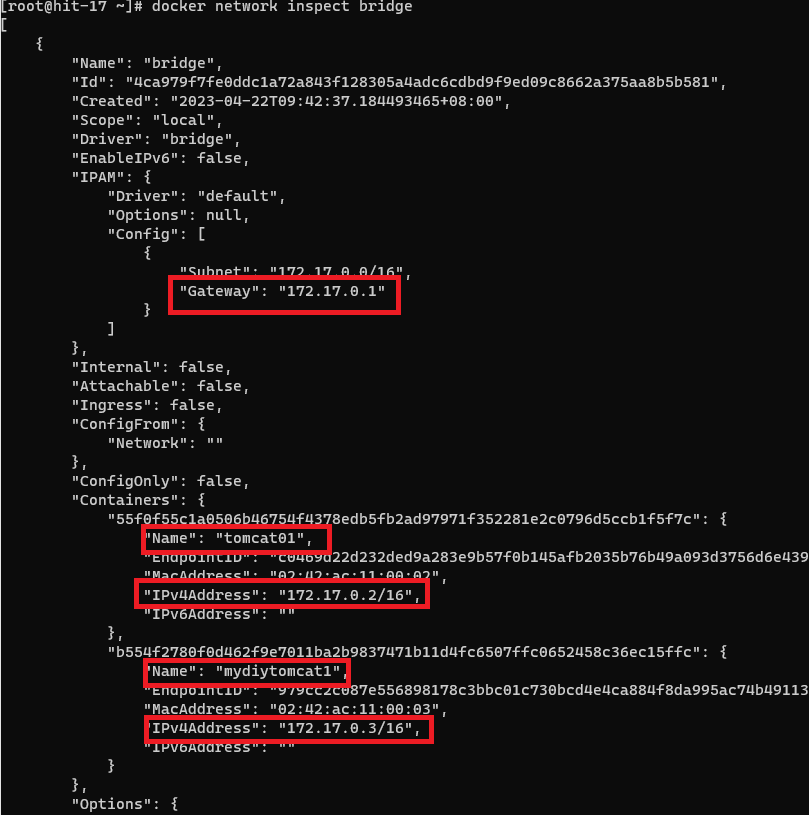
**2. —link**

1. # 通过服务名ping不通；如何解决？
2. [root@hit-17 test]# docker exec -it mydiytomcat ping tomcat01
3. ping: tomcat01: Name or service not known
4. # 通过--link可以解决网络连接问题。
5. [root@hit-17 ~]# docker run -d -p 3356:8080 --name mydiytomcat1 --link tomcat01 diytomcat
6. b554f2780f0d462f9e7011ba2b9837471b11d4fc6507ffc0652458c36ec15ffc
7. [root@hit-17 ~]# docker exec -it mydiytomcat1 ping tomcat01
8. PING tomcat01 (172.17.0.2) 56(84) bytes of data.
9. 64 bytes from tomcat01 (172.17.0.2): icmp\_seq=1 ttl=64 time=0.096 ms
10. 64 bytes from tomcat01 (172.17.0.2): icmp\_seq=2 ttl=64 time=0.085 ms
11. # 反向可以ping通吗？（不可以）
12. [root@hit-17 ~]# docker exec -it tomcat01 ping mydiytomcat1
13. ping: mydiytomcat1: Name or service not known

docker network ls可以查看networkID



探究：docker network inspect networkID (docker network ls可以查看networkID)



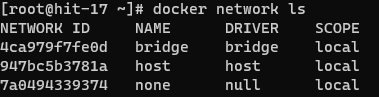
其实mydiytomcat1的host文件中已经配置了tomcat01和其ip的映射，所以mydiytomcat1可以ping通tomcat01。tomcat01的host的文件中没有配置mydiytomcat1的映射，所以tomcat无法ping通mydiytomcat1。

1. # 查看
2. [root@hit-17 ~]# docker exec -it mydiytomcat1 cat /etc/hosts
3. 127.0.0.1 localhost
4. ::1 localhost ip6-localhost ip6-loopback
5. fe00::0 ip6-localnet
6. ff00::0 ip6-mcastprefix
7. ff02::1 ip6-allnodes
8. ff02::2 ip6-allrouters
9. 172.17.0.2 tomcat01 55f0f55c1a05
10. 172.17.0.3 b554f2780f0d
11. [root@hit-17 ~]# docker exec -it tomcat01 cat /etc/hosts
12. 127.0.0.1 localhost
13. ::1 localhost ip6-localhost ip6-loopback
14. fe00::0 ip6-localnet
15. ff00::0 ip6-mcastprefix
16. ff02::1 ip6-allnodes
17. ff02::2 ip6-allrouters
18. 172.17.0.2 55f0f55c1a05

本质探究：—link 就是我们在hosts配置中增加了一个”172.17.0.2 tomcat01 55f0f55c1a05”

**3. 自定义网络**

**（1）查看所有的docker网络**

=

**（2）网络模式**

bridge：桥接 docker（默认，自己创建也使用bridge桥接模式）

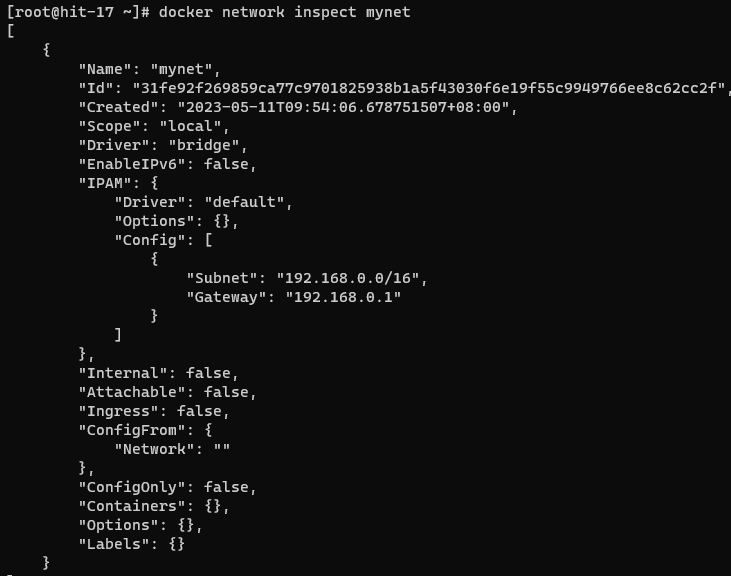
none：不配置网络

host：和主机共享网络

**（3）测试**

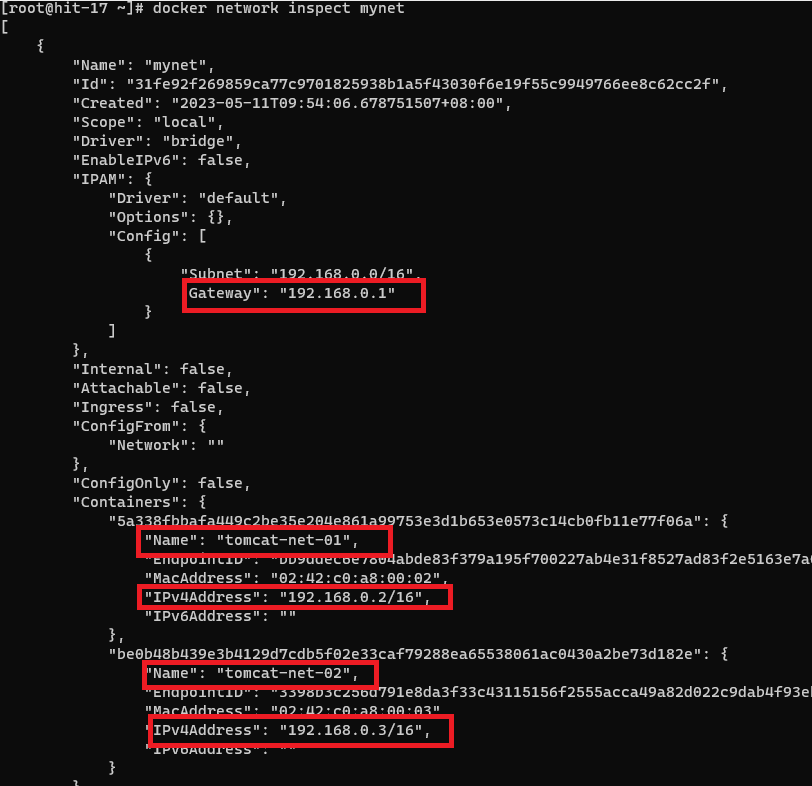
1. # 我们直接启动的命令--net bridge（这个就是我们的docker0）；默认带上这个参数的，以下两种启动方式效果一致。
2. docker run -d -P --name tomcat01 tomcat
3. docker run -d -P --name tomcato1 --net bridge tomcat
4. # docker0特点：默认，域名不能访问，--1ink可以打通连接！
5. # 我们可以自定义一个网络！
6. [root@hit-17 ~]# docker network create --driver bridge --subnet 192.168.0.0/16 --gateway 192.168.0.1 mynet
7. 31fe92f269859ca77c9701825938b1a5f43030f6e19f55c9949766ee8c62cc2f
8. [root@hit-17 ~]# docker network ls
9. NETWORK ID NAME DRIVER SCOPE
10. 4ca979f7fe0d bridge bridge local
11. 947bc5b3781a host host local
12. 31fe92f26985 mynet bridge local
13. 7a0494339374 none null local

我们自己的网络就创建好了



启动两个容器测试：

1. [root@hit-17 ~]# docker run -d -p 3380:8080 --name tomcat-net-01 --net mynet diytomcat
2. 5a338fbbafa449c2be35e204e861a99753e3d1b653e0573c14cb0fb11e77f06a[root@hit-17 ~]# docker run -d -p 3381:8080 --name tomcat-net-02 --net mynet diytomcat
3. be0b48b439e3b4129d7cdb5f02e33caf79288ea65538061ac0430a2be73d182e



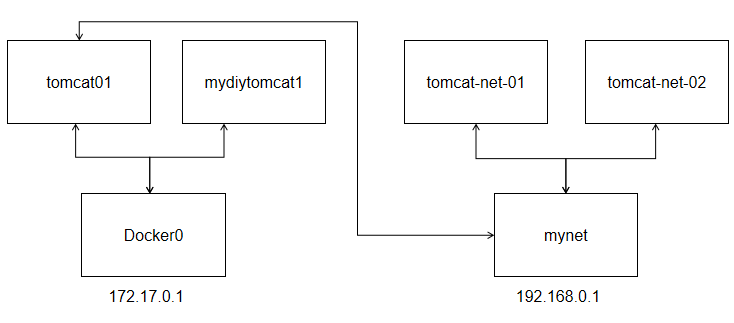
1. # 不使用--link，ping名字也可以ping通。tomcat-net-01 ping tomcat-net-02可以ping通
2. [root@hit-17 ~]# docker exec -it tomcat-net-01 ping tomcat-net-02
3. PING tomcat-net-02 (192.168.0.3) 56(84) bytes of data.
4. 64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp\_seq=1 ttl=64 time=0.066 ms
5. 64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp\_seq=2 ttl=64 time=0.089 ms
6. 64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp\_seq=3 ttl=64 time=0.098 ms
7. # 不使用--link，ping名字也可以ping通。tomcat-net-02 ping tomcat-net-01可以ping通
8. [root@hit-17 ~]# docker exec -it tomcat-net-02 ping tomcat-net-01
9. PING tomcat-net-01 (192.168.0.2) 56(84) bytes of data.
10. 64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp\_seq=1 ttl=64 time=0.054 ms
11. 64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp\_seq=2 ttl=64 time=0.079 ms
12. 64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp\_seq=3 ttl=64 time=0.082 ms

自定义的网络docker都已经帮我们维护好了对应的关系。

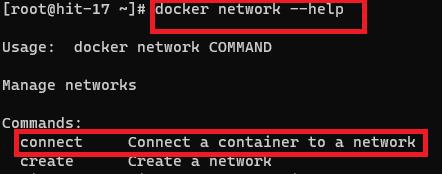
**4. 网络连通**

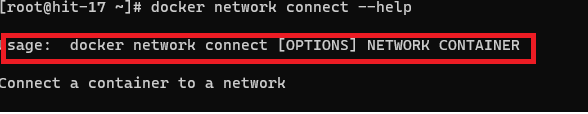
1. # tomcat01在docker0网络下，tomcat-net-01在mynet网络下；
2. # tomcat01 ping tomcat-net-01是ping不通的
3. [root@hit-17 ~]# docker exec -it tomcat01 ping tomcat-net-01
4. OCI runtime exec failed: exec failed: unable to start container process: exec: "ping": executable file not found in $PATH: unknown

**容器和mynet网络需要打通**



打通命令





1. # 测试：打通tomcat01连接mynet
2. docker network connect mynet tomcat01
3. # 连通之后就是将tomcat01放到了mynet网络下
4. # 一个容器两个ip地址！



1. # 连接ok
2. [root@hit-17 test]# docker exec -it tomcat01 ping tomcat-net-01
3. PING tomcat-net-01 (192.168.0.2) 56(84) bytes of data.
4. 64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp\_seq=1 ttl=64 time=0.086 ms
5. 64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp\_seq=2 ttl=64 time=0.086 ms
6. 64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp\_seq=3 ttl=64 time=0.090 ms