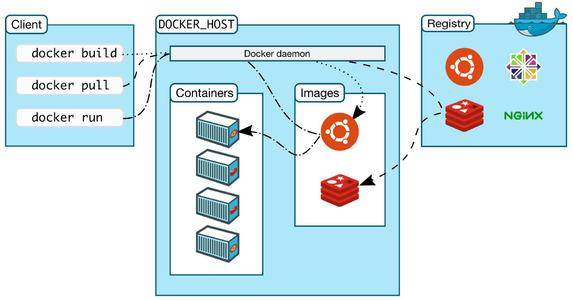
**Docke基本组成**



**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）

**容器(container):**

Docker利用容器技术，独立运行一个或一组应用，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

**仓库(repository)：**

仓库就是存放镜像的地方！

仓库分为公有和私有仓库！

Docker Hub（默认是国外的）

阿里云。。。。都有容器服务器(配置镜像加速！)

安装Docker

环境准备

1.需要linux基础

2.Centos7

3.我们使用xshell连接远程服务器进行操作！

环境查看

系统内核
[root@localhost /]# uname -r
3.10.0-1127.el7.x86\_64

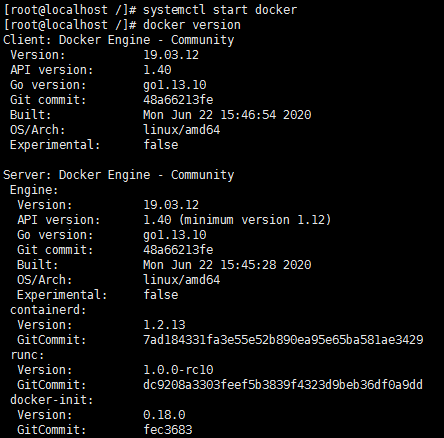
系统版本
[root@localhost /]# cat /etc/os-release
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID\_LIKE="rhel fedora"
VERSION\_ID="7"
PRETTY\_NAME="CentOS Linux 7 (Core)"
ANSI\_COLOR="0;31"
CPE\_NAME="cpe:/o:centos:centos:7"
HOME\_URL="https://www.centos.org/"
BUG\_REPORT\_URL="https://bugs.centos.org/"
CENTOS\_MANTISBT\_PROJECT="CentOS-7"
CENTOS\_MANTISBT\_PROJECT\_VERSION="7"
REDHAT\_SUPPORT\_PRODUCT="centos"
REDHAT\_SUPPORT\_PRODUCT\_VERSION="7"

安装

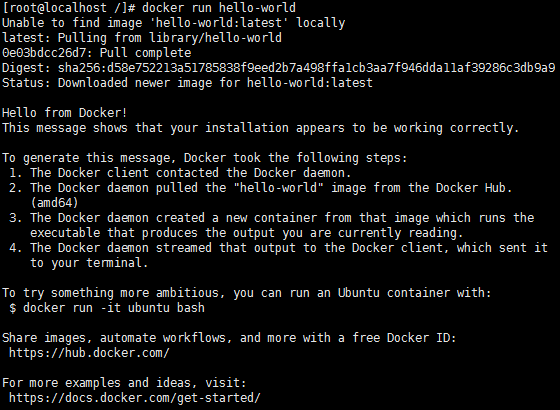
帮助文档：

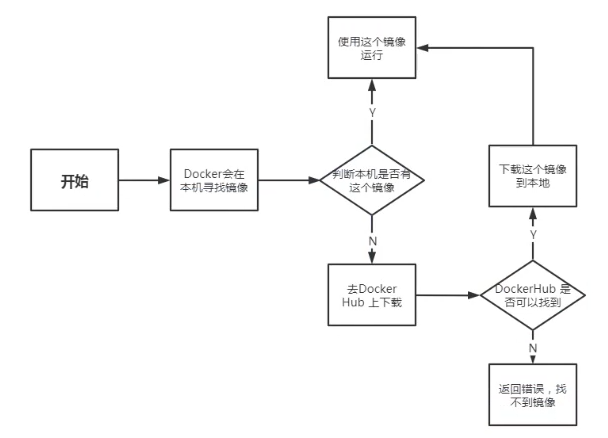
<https://docs.docker.com/engine/install/centos/>

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
2.需要安装包
yum install -y yum-utils
3.设置镜像的仓库
yum-config-manager \
--add-repo \
https://download.docker.com/linux/centos/docker-ce.repo （默认是国外的）
yum-config-manager \
--add-repo \
http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo （推荐使用阿里云）
4.更新yum软件包索引
yum makecache fast
5.安装docker相关的镜像 docker-ce时区版本 ee-企业版本
yum install docker-ce docker-ce-cli containerd.io （最新版本）
yum install docker-ce-<VERSION\_STRING> docker-ce-cli-<VERSION\_STRING> containerd.io（安装指定版本）
6.启动docker容器
systemctl start docker
7.使用docker version 是否安装成功



7.测试docker hello-world
docker run hello-world





8.查看一下下载这个hello-world 镜像
[root@localhost /]# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
hello-world latest bf756fb1ae65 5 months ago 13.3kB

了解卸载docke

1.卸载依赖
yum remove docker-ce docker-ce-cli containerd.io
2.删除资源
rm -rf /var/lib/docke
/var/lib/docke docke默认的工作路径

**阿里云镜像加速（有阿里云可配置）**

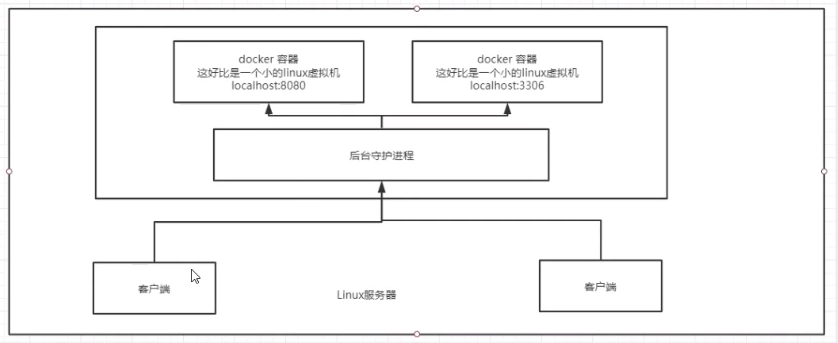
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
"registry-mirrors": ["https://7qx4mgv1.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker

**底层原理**

**Docker是怎么工作的**

**Docker是一个Client-Srver结构的系统，Docker的守护进程运行在主机上，通过Socket从客户端访问！**

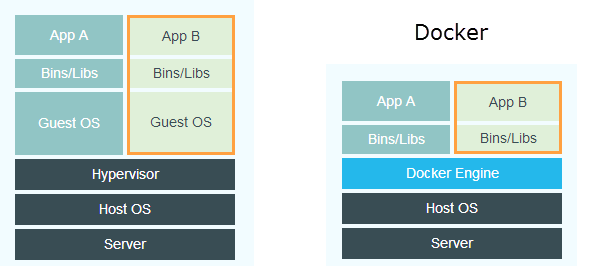
**DockerServer接受到DockerClient的指令，就会执行这个命令。**



**Docker为什么比虚拟机（VM）快？**

**1、docker有着比虚拟机更少的抽象层。由于docker不需要Hypervisor实现硬件资源虚拟化,运行在docker容器上的程序直接使用的都是实际物理机的硬件资源。因此在CPU、内存利用率上docker将会在效率上有明显优势。**

**2、docker利用的是宿主机的内核,而不需要Guest OS。**



**所以说，新建一个容器的时候，docker不需要像虚拟机一样加载一个操作系统内核，避免引导性操作，虚拟机加载Guest oS，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，秒级别！**

学习完毕所有的命令，再回头看这段理论非常清晰。

**Docker常用命令**

**帮助命令**

docker version 显示docker版本信息
docker info 显示docker系统信息，包括镜像和容器数量
docker 命令 --help 帮助命令
service docker restart 重新启动docker

**帮助文档地址**

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

**镜像命令**

**docker images 查看所有本地的主机上的镜像**

[root@localhost /]# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
hello-world latest bf756fb1ae65 5 months ago 13.3kB
解释
REPOSITORY 镜像的仓库源
TAG 镜像的标签
IMAGE ID 镜像ID
CREATED 镜像的创建时间
SIZE 镜像大小
可选项
-a, --all 列出所有镜像
-q, --quiet 只显示镜像id

**docker search 搜索镜像**

**.**[root@localhost /]# docker search mysql
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
mysql MySQL is a widely used, open-source relation… 9673 [OK]
mariadb MariaDB is a community-developed fork of MyS… 3518 [OK]
可选项 通过搜藏来过滤
docker search mysql --filter=stars=3000 搜索出来的mysql镜像就是大于3000的
[root@localhost /]# docker search mysql --filter=stars=3000
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
mysql MySQL is a widely used, open-source relation… 9673 [OK]
mariadb MariaDB is a community-developed fork of MyS… 3518 [OK]

**docker pull下载镜像**

x下载镜像 docker pull 镜像名[:tag]
[root@localhost /]# docker pull mysql
Using default tag: latest 如果不写tag 默认就是latest
latest: Pulling from library/mysql
8559a31e96f4: Pull complete 分层下载 docker image的核心 联合文件系统
d51ce1c2e575: Pull complete
c2344adc4858: Pull complete
fcf3ceff18fc: Pull complete
16da0c38dc5b: Pull complete
b905d1797e97: Pull complete
4b50d1c6b05c: Pull complete
c75914a65ca2: Pull complete
1ae8042bdd09: Pull complete
453ac13c00a3: Pull complete
9e680cd72f08: Pull complete
a6b5dc864b6c: Pull complete
Digest: sha256:8b7b328a7ff6de46ef96bcf83af048cb00a1c86282bfca0cb119c84568b4caf6 签名
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest 真实地址
等价于
docker pull mysql
docker pull docker.io/library/mysql:latest
指定版本
[root@localhost docker]# docker pull mysql:5.7
5.7: Pulling from library/mysql
8559a31e96f4: Already exists 共用上个mysql下载的层不需要重新下载
d51ce1c2e575: Already exists
c2344adc4858: Already exists
fcf3ceff18fc: Already exists
16da0c38dc5b: Already exists
b905d1797e97: Already exists
4b50d1c6b05c: Already exists
d85174a87144: Pull complete
a4ad33703fa8: Pull complete
f7a5433ce20d: Pull complete
3dcd2a278b4a: Pull complete
Digest: sha256:32f9d9a069f7a735e28fd44ea944d53c61f990ba71460c5c183e610854ca4854
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7

**docker rmi 删除镜像**

docker rmi -f 容器id 删除指定的容器
docker rmi -f 容器id 容器id 容器id 删除多个容器id
docker rmi -f $(docker images -aq) 删除全部容器

**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镜像来学习**

docker pull centos 下载最新的centos容器

**新建容器并启动**

docker run 【可选参数】image
参数说明
--name="Name" 容器名字 tomcat01 tomcat02，用来区分容器
-d 后台方式运行
-it 使用交互方式运行，进入容器查看内容
-p 指定容器端口 -p 8080：8080
-p ip：主机端口：容器端口
-p 主机端口：容器端口（常用）
-p 容器端口
容器端口
-p 随机指定端口
测试 启动并进入容器
[root@localhost /]# docker run -it centos /bin/bash
[root@9420d463b591 /]#
[root@9420d463b591 /]# ls 查看容器内的centos，基础版本，很多命令都是不完善的
bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var
从容器返回主机
[root@9420d463b591 /]# exit
exit
[root@localhost /]# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

**列出所有运行容器**

docker ps
列出当前正在运行的容器
-a 列出当前正在运行的容器+带出历史运行过的容器
-n 显示最近创建的容器
-q 只显示容器的编号
[root@localhost /]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
[root@localhost /]# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9420d463b591 centos "/bin/bash" 4 minutes ago Exited (0) About a minute ago quizzical\_khorana
56b52eeb17fb bf756fb1ae65 "/hello" 17 hours ago Exited (0) 17 hours ago agitated\_haibt
f1159d910dc6 be0dbf01a0f3 "docker-entrypoint.s…" 23 hours ago Exited (1) 23 hours ago heuristic\_bardeen
0d86b9f75c93 bf756fb1ae65 "/hello" 25 hours ago Exited (0) 25 hours ago zealous\_shamir

**退出容器**

exit 直接容器停止并退出
ctrl+p+q 容器不停止退出

**删除容器**

docker rm 容器id 删除指定容器
docker rm -f $(docker ps -aq) 删除所有容器
docker ps -a -q|xargs docker rm 删除所有的容器

**启动和停止容器操作**

docker start 容器id 启动容器
docker restart 容器id 重启容器
docker stop 容器id 停止当前运行容器
docker kill 容器id 强制停止当前容器

**常用其它命令**

命令docker run -d 镜像名！
[root@localhost /]# docker run -d centos
问题docker ps 发现centos停止了
常见的坑 docker 容器使用后台运行，就必须要有一个前台进程，docker发现没有应用，就会自动停止
Nginx，容器启动后，发现自己没有提供服务，就会立刻停止，就是没有程序了

**查看日志**

docker logs -f -t --tail 容器,没有日志
自己写一个脚shell本
[root@localhost /]# docker run -d centos bin/sh -c "while true;do echo wang;slepp 1; done"
查看启动镜像
[root@localhost /]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ae1fd5e514c5 centos "bin/sh -c 'while tr…" 2 minutes ago Up 2 minutes happy\_grothendieck
显示日志
-tf 显示日志信息
--tail number 要显示的日志条数
[root@localhost /]# docker logs -tf --tail 10 ae1fd5e514c5

**查看容器中的进程信息**

命令 docker top 容器id
[root@localhost /]# docker top ae1fd5e514c5
UID PID PPID C STIME TTY TIME CMD
root 12942 12925 66 13:55 ? 00:03:48 bin/sh -c while true;do echo wang;slepp 1; done

**查看镜像元数据**

命令docker inspect 容器id
[root@localhost /]# docker inspect ae1fd5e514c5
[
{
"Id": "ae1fd5e514c5bf55544fd6cd475e0dfcea8e61b760e8777241971a308c6d9fa7",
"Created": "2020-06-26T05:55:25.437540713Z",
"Path": "bin/sh",
"Args": [
"-c",
"while true;do echo wang;slepp 1; done"
],
"State": {
"Status": "running",
"Running": true,
"Paused": false,
"Restarting": false,
"OOMKilled": false,
"Dead": false,
"Pid": 12942,
"ExitCode": 0,
"Error": "",
"StartedAt": "2020-06-26T05:55:26.436532985Z",
"FinishedAt": "0001-01-01T00:00:00Z"
},
"Image": "sha256:831691599b88ad6cc2a4abbd0e89661a121aff14cfa289ad840fd3946f274f1f",
"ResolvConfPath": "/var/lib/docker/containers/ae1fd5e514c5bf55544fd6cd475e0dfcea8e61b760e8777241971a308c6d9fa7/resolv.conf",
"HostnamePath": "/var/lib/docker/containers/ae1fd5e514c5bf55544fd6cd475e0dfcea8e61b760e8777241971a308c6d9fa7/hostname",
"HostsPath": "/var/lib/docker/containers/ae1fd5e514c5bf55544fd6cd475e0dfcea8e61b760e8777241971a308c6d9fa7/hosts",
"LogPath": "/var/lib/docker/containers/ae1fd5e514c5bf55544fd6cd475e0dfcea8e61b760e8777241971a308c6d9fa7/ae1fd5e514c5bf55544fd6cd475e0dfcea8e61b760e8777241971a308c6d9fa7-json.log",
"Name": "/happy\_grothendieck",
"RestartCount": 0,
"Driver": "overlay2",
"Platform": "linux",
"MountLabel": "",
"ProcessLabel": "",
"AppArmorProfile": "",
"ExecIDs": null,
"HostConfig": {
"Binds": null,
"ContainerIDFile": "",
"LogConfig": {
"Type": "json-file",
"Config": {}
},
"NetworkMode": "default",
"PortBindings": {},
"RestartPolicy": {
"Name": "no",
"MaximumRetryCount": 0
},
"AutoRemove": false,
"VolumeDriver": "",
"VolumesFrom": null,
"CapAdd": null,
"CapDrop": null,
"Capabilities": null,
"Dns": [],
"DnsOptions": [],
"DnsSearch": [],
"ExtraHosts": null,
"GroupAdd": null,
"IpcMode": "private",
"Cgroup": "",
"Links": null,
"OomScoreAdj": 0,
"PidMode": "",
"Privileged": false,
"PublishAllPorts": false,
"ReadonlyRootfs": false,
"SecurityOpt": null,
"UTSMode": "",
"UsernsMode": "",
"ShmSize": 67108864,
"Runtime": "runc",
"ConsoleSize": [
0,
0
],
"Isolation": "",
"CpuShares": 0,
"Memory": 0,
"NanoCpus": 0,
"CgroupParent": "",
"BlkioWeight": 0,
"BlkioWeightDevice": [],
"BlkioDeviceReadBps": null,
"BlkioDeviceWriteBps": null,
"BlkioDeviceReadIOps": null,
"BlkioDeviceWriteIOps": null,
"CpuPeriod": 0,
"CpuQuota": 0,
"CpuRealtimePeriod": 0,
"CpuRealtimeRuntime": 0,
"CpusetCpus": "",
"CpusetMems": "",
"Devices": [],
"DeviceCgroupRules": null,
"DeviceRequests": null,
"KernelMemory": 0,
"KernelMemoryTCP": 0,
"MemoryReservation": 0,
"MemorySwap": 0,
"MemorySwappiness": null,
"OomKillDisable": false,
"PidsLimit": null,
"Ulimits": null,
"CpuCount": 0,
"CpuPercent": 0,
"IOMaximumIOps": 0,
"IOMaximumBandwidth": 0,
"MaskedPaths": [
"/proc/asound",
"/proc/acpi",
"/proc/kcore",
"/proc/keys",
"/proc/latency\_stats",
"/proc/timer\_list",
"/proc/timer\_stats",
"/proc/sched\_debug",
"/proc/scsi",
"/sys/firmware"
],
"ReadonlyPaths": [
"/proc/bus",
"/proc/fs",
"/proc/irq",
"/proc/sys",
"/proc/sysrq-trigger"
]
},
"GraphDriver": {
"Data": {
"LowerDir": "/var/lib/docker/overlay2/bc33ae46693670bff7606bddd1b5cb43652418e1f2ce3aa164bbd09ac7035b19-init/diff:/var/lib/docker/overlay2/f2f6b70a5d2a354c631429de7c5203b36dc5299fe7683668842b2b2c222cdc0d/diff",
"MergedDir": "/var/lib/docker/overlay2/bc33ae46693670bff7606bddd1b5cb43652418e1f2ce3aa164bbd09ac7035b19/merged",
"UpperDir": "/var/lib/docker/overlay2/bc33ae46693670bff7606bddd1b5cb43652418e1f2ce3aa164bbd09ac7035b19/diff",
"WorkDir": "/var/lib/docker/overlay2/bc33ae46693670bff7606bddd1b5cb43652418e1f2ce3aa164bbd09ac7035b19/work"
},
"Name": "overlay2"
},
"Mounts": [],
"Config": {
"Hostname": "ae1fd5e514c5",
"Domainname": "",
"User": "",
"AttachStdin": false,
"AttachStdout": false,
"AttachStderr": false,
"Tty": false,
"OpenStdin": false,
"StdinOnce": false,
"Env": [
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
],
"Cmd": [
"bin/sh",
"-c",
"while true;do echo wang;slepp 1; done"
],
"Image": "centos",
"Volumes": null,
"WorkingDir": "",
"Entrypoint": null,
"OnBuild": null,
"Labels": {
"org.label-schema.build-date": "20200611",
"org.label-schema.license": "GPLv2",
"org.label-schema.name": "CentOS Base Image",
"org.label-schema.schema-version": "1.0",
"org.label-schema.vendor": "CentOS"
}
},
"NetworkSettings": {
"Bridge": "",
"SandboxID": "eaf2a5b705c534501ad56ce274a2a7c86a2744ad26129d4cd5c61326fb500b52",
"HairpinMode": false,
"LinkLocalIPv6Address": "",
"LinkLocalIPv6PrefixLen": 0,
"Ports": {},
"SandboxKey": "/var/run/docker/netns/eaf2a5b705c5",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"EndpointID": "cb605d167049d190dc71f41a76c3615abecbebaed942a88dbcf02d74a5ee96fa",
"Gateway": "172.17.0.1",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"MacAddress": "02:42:ac:11:00:02",
"Networks": {
"bridge": {
"IPAMConfig": null,
"Links": null,
"Aliases": null,
"NetworkID": "2a67b1b66338af38ab094dfcdf5ff52e9bede09879f518a7588133ca2a28ff9d",
"EndpointID": "cb605d167049d190dc71f41a76c3615abecbebaed942a88dbcf02d74a5ee96fa",
"Gateway": "172.17.0.1",
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"MacAddress": "02:42:ac:11:00:02",
"DriverOpts": null
}
}
}
}
]

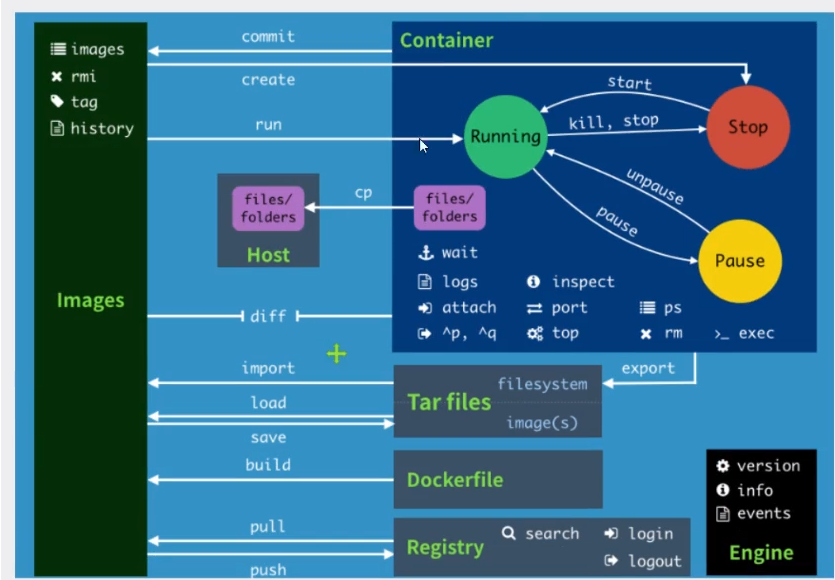
**进入当前正在运行的容器**

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
命令
docker exec -it 容器id /bin/bash
测试
[root@localhost /]# docker exec -it ae1fd5e514c5 /bin/bash
[root@ae1fd5e514c5 /]# ls
bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var
[root@ae1fd5e514c5 /]# ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 66 05:55 ? 00:08:58 bin/sh -c while true;do echo wang;slepp 1; done
root 45535 0 0 06:08 pts/0 00:00:00 /bin/bash
root 57547 0 0 06:08 pts/1 00:00:00 /bin/bash
root 67893 57547 0 06:08 pts/1 00:00:00 ps -ef
方式二
docker attach 容器id
测试
[root@localhost /]# docker attach ae1fd5e514c5
正在执行当前的代码
docker exec 进入容器后开启一个新的终端，可以在里面操作（常用）
docker attach 进入容器正在执行的终端，不会启动新的进程

**从容器内拷贝文件到主机**

docker cp 容器id：容器内路径 目的的主机路径
touch test.java 创建文件
查看当前主机目录下
[root@localhost home]# ls
wangzhiming179
[root@localhost home]# touch wangzhiming.java
[root@localhost home]# ls
wangzhiming179 wangzhiming.java
[root@localhost home]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d7c3b51a60b2 centos "/bin/bash" About a minute ago Up About a minute ecstatic\_sanderson
进入docker容器内部
[root@localhost home]# docker attach d7c3b51a60b2
[root@d7c3b51a60b2 /]# cd /home
[root@d7c3b51a60b2 home]# ls
在容器新建一个文件
[root@d7c3b51a60b2 home]# touch test.java
[root@d7c3b51a60b2 home]# exit
exit
[root@localhost home]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
[root@localhost home]# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d7c3b51a60b2 centos "/bin/bash" 3 minutes ago Exited (0) 38 seconds ago ecstatic\_sanderson
将文件拷贝到主机上
[root@localhost home]# docker cp d7c3b51a60b2:/home/test.java /home
[root@localhost home]# ls
test.java wangzhiming179 wangzhiming.java
拷贝是一个手动过程，未来我们可以使用-v卷技术，可以实现，自动同步

**小结**



**命令大全**

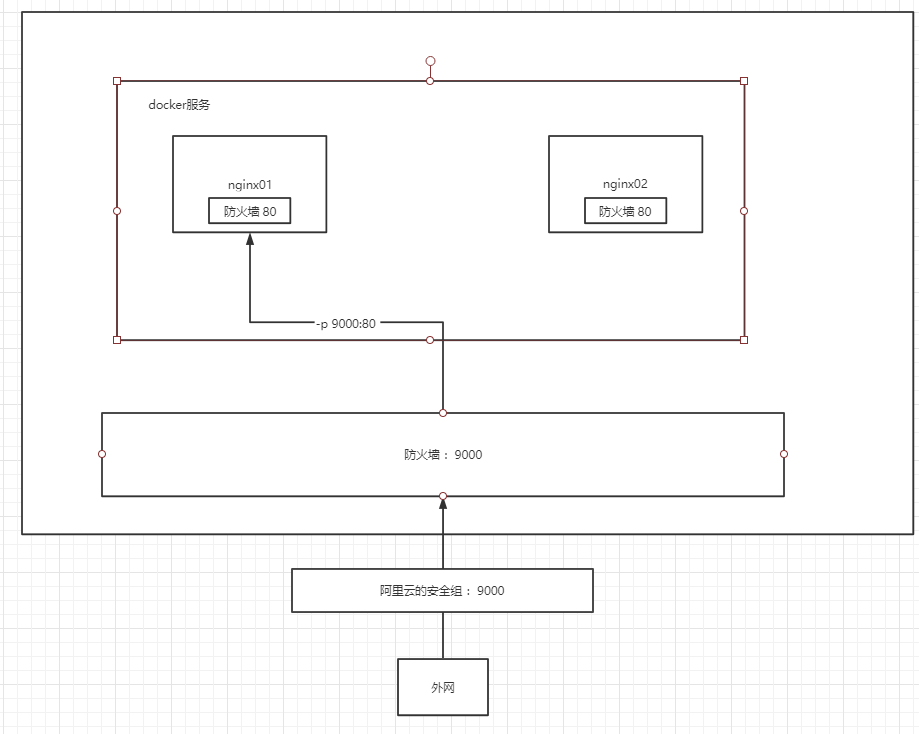
attach Attach local standard input, output, and error streams to a running container #当前shell下 attach连接指定运行的镜像
build Build an image from a Dockerfile # 通过Dockerfile定制镜像
commit Create a new image from a container's changes #提交当前容器为新的镜像
cp Copy files/folders between a container and the local filesystem #拷贝文件
create Create a new container #创建一个新的容器
diff Inspect changes to files or directories on a container's filesystem #查看docker容器的变化
events Get real time events from the server # 从服务获取容器实时时间
exec Run a command in a running container # 在运行中的容器上运行命令
export Export a container's filesystem as a tar archive #导出容器文件系统作为一个tar归档文件[对应import]
history Show the history of an image # 展示一个镜像形成历史
images List images #列出系统当前的镜像
import Import the contents from a tarball to create a filesystem image #从tar包中导入内容创建一个文件系统镜像
info Display system-wide information # 显示全系统信息
inspect Return low-level information on Docker objects #查看容器详细信息
kill Kill one or more running containers # kill指定docker容器
load Load an image from a tar archive or STDIN #从一个tar包或标准输入中加载一个镜像[对应save]
login Log in to a Docker registry #
logout Log out from a Docker registry
logs Fetch the logs of a container
pause Pause all processes within one or more containers
port List port mappings or a specific mapping for the container
ps List containers
pull Pull an image or a repository from a registry
push Push an image or a repository to a registry
rename Rename a container
restart Restart one or more containers
rm Remove one or more containers
rmi Remove one or more images
run Run a command in a new container
save Save one or more images to a tar archive (streamed to STDOUT by default)
search Search the Docker Hub for images
start Start one or more stopped containers
stats Display a live stream of container(s) resource usage statistics
stop Stop one or more running containers
tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE
top Display the running processes of a container
unpause Unpause all processes within one or more containers
update Update configuration of one or more containers
version Show the Docker version information
wait Block until one or more containers stop, then print their exit codes

**作业练习**

参考网址
https://www.cnblogs.com/yslss/p/12982155.html

Docker安装Nginx
docker searc Nginx 搜索镜像
docker pull Nginx下载镜像
运行测试
[root@localhost home]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
[root@localhost home]# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
centos latest 831691599b88 9 days ago 215MB
nginx latest 2622e6cca7eb 2 weeks ago 132MB
-d 后台运行
--name 给容器命名
-p 宿主机端口：容器内部端口
[root@localhost home]# docker run -d --name nginx01 -p 3344:80 nginx
0ab1978839862b8bda741dd025d6a7ba318c6cec217f1b08f86c698296dc3f43
[root@localhost home]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0ab197883986 nginx "/docker-entrypoint.…" 8 seconds ago Up 6 seconds 0.0.0.0:3344->80/tcp nginx01
[root@localhost home]# curl localhost:3344
进入容器后开启一个新的终端，可以在里面操作
[root@localhost home]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0ab197883986 nginx "/docker-entrypoint.…" 10 minutes ago Up 3 seconds 0.0.0.0:3344->80/tcp nginx01
[root@localhost home]# docker exec -it nginx01 /bin/bash
root@0ab197883986:/# ls
bin boot dev docker-entrypoint.d docker-entrypoint.sh etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@0ab197883986:/# whereis nginx
nginx: /usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx

端口暴露



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

docker装一个tomcat
官方使用
docker run -it --rm tomcat:9.0
我们之前的启动都是后台，停止容器之后，容器还是可以查到docker run -it --rm tomact:9.0 一般用来测试，用完即删
下载镜像后在启动
docker pull tomcat 最新版
启动运行
[root@localhost /]# docker run -d -p 3355:8080 --name tomcat01 tomcat
测试访问没有问题
进入容器
[root@localhost /]# docker exec -it tomcat01 /bin/bash
发现问题：1.linux命令少了，2.没有webapps。 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了，
保证最小可运行环境

思考问题：我们以后要部署项目，都需要进入容器内部？十分麻烦，我要是可以在容器外部提供一个映射路径，webapps，我们在外部放置项目，就会自动同步到内部就好了。

docker容器tomcat+网站 docker mysql

部署ex+kibana
es暴露的端口很多
es十分耗内存
es的数据一般需要放置到安全目录！挂载
--net somenetwork
下载启动elasticsearch
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" elasticsearch:7.8.0
启动linux就卡主了，docker stats查看cpu的状态
es是一个十分耗内存的 1.xg
查看docker stats
测试es是否启动成功
[root@localhost /]# curl localhost:9200
{
"name" : "05bd59c7fb84",
"cluster\_name" : "docker-cluster",
"cluster\_uuid" : "8AfT6tx\_RrG327H1T3UsaQ",
"version" : {
"number" : "7.8.0",
"build\_flavor" : "default",
"build\_type" : "docker",
"build\_hash" : "757314695644ea9a1dc2fecd26d1a43856725e65",
"build\_date" : "2020-06-14T19:35:50.234439Z",
"build\_snapshot" : false,
"lucene\_version" : "8.5.1",
"minimum\_wire\_compatibility\_version" : "6.8.0",
"minimum\_index\_compatibility\_version" : "6.0.0-beta1"
},
"tagline" : "You Know, for Search"
}
赶紧关闭，增加内存的限制

clipboard.png

赶紧关闭，增加内存的限制，修改配置文件 -e 环境配置修改
docker run -d --name elasticsearch02 -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" -e ES\_JAVA\_OPS="-Xms64M -Xmx512m" elasticsearch:7.8.0
docker stats

clipboard.png

[root@localhost /]# curl localhost:9200
{
"name" : "3f2d9aa9728e",
"cluster\_name" : "docker-cluster",
"cluster\_uuid" : "U0FTHqRwTBGTBUwFIs20mw",
"version" : {
"number" : "7.8.0",
"build\_flavor" : "default",
"build\_type" : "docker",
"build\_hash" : "757314695644ea9a1dc2fecd26d1a43856725e65",
"build\_date" : "2020-06-14T19:35:50.234439Z",
"build\_snapshot" : false,
"lucene\_version" : "8.5.1",
"minimum\_wire\_compatibility\_version" : "6.8.0",
"minimum\_index\_compatibility\_version" : "6.0.0-beta1"
},
"tagline" : "You Know, for Search"
}

**可视化**

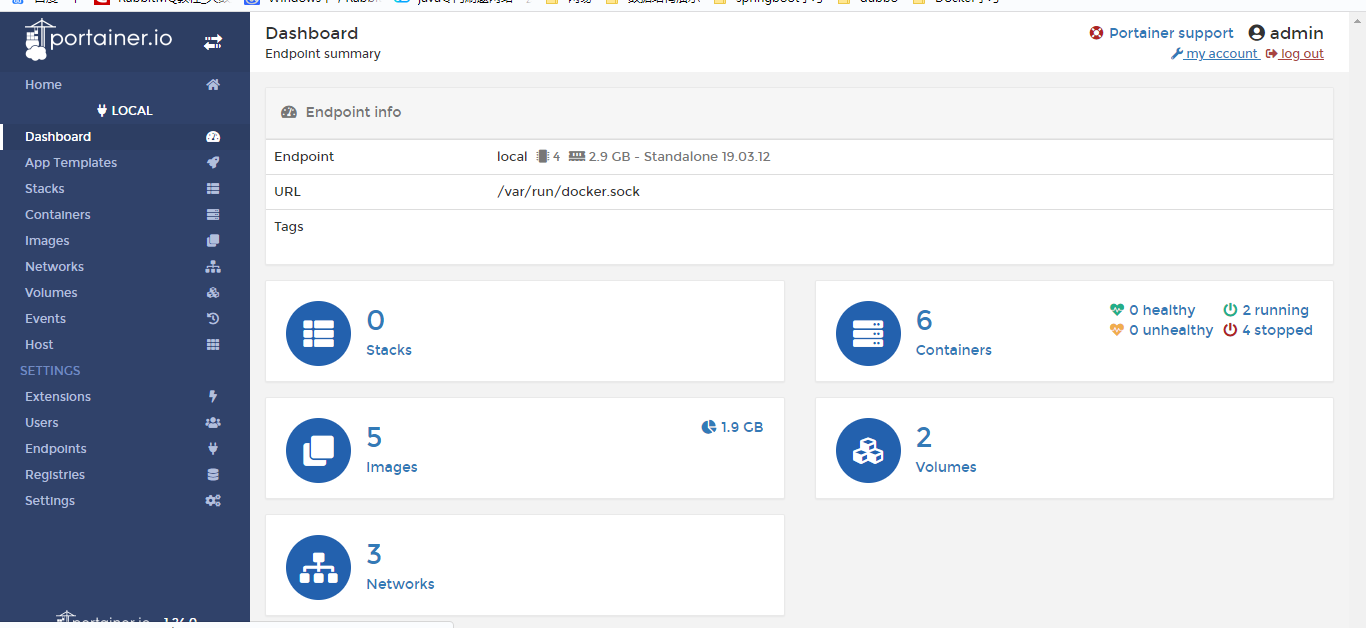
portainer(先用这个)
docker run -d -p 8088:9000 --restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true portainer/portainer
Rancher(CI/CD再用)

**什么是portainer？**

docker图形化界面管理工具！提供一个后台面板供我们操作！

docker run -d -p 8088:9000 --restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true portainer/portainer

访问测试: <http://192.168.1.105:8088> 通过它来访问：



**Docker镜像讲解**

**镜像是什么**

镜像是一种轻量级、可执行的独立软件保，用来打包软件运行环境和基于运行环境开发的软件，他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

**Docker镜像加载原理**

**UnionFs （联合文件系统）**

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，他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

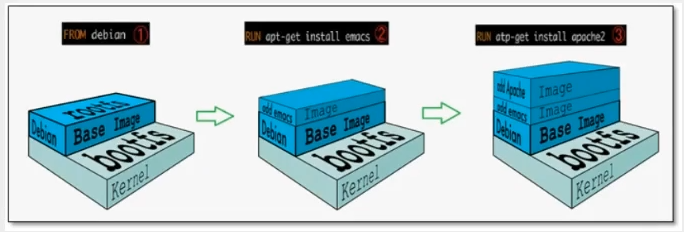
特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录

**Docker镜像加载原理**

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。

boots(boot file system）主要包含 bootloader和 Kernel, bootloader主要是引导加 kernel, Linux刚启动时会加bootfs文件系统，在 Docker镜像的最底层是 boots。这一层与我们典型的Linux/Unix系统是一样的，包含boot加載器和内核。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？

clipboard.png

对于个精简的OS,rootfs可以很小，只需要包合最基本的命令，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， boots基本是一致的， rootfs会有差別，因此不同的发行版可以公用bootfs.

虚拟机是分钟级别，容器是秒级！

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，可以看到是一层层的在下载

[root@localhost /]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
8559a31e96f4: Already exists
85a6a5c53ff0: Pull complete
b69876b7abed: Pull complete
a72d84b9df6a: Pull complete
5ce7b314b19c: Pull complete
04c4bfb0b023: Pull complete
Digest: sha256:800f2587bf3376cb01e6307afe599ddce9439deafbd4fb8562829da96085c9c5
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest

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

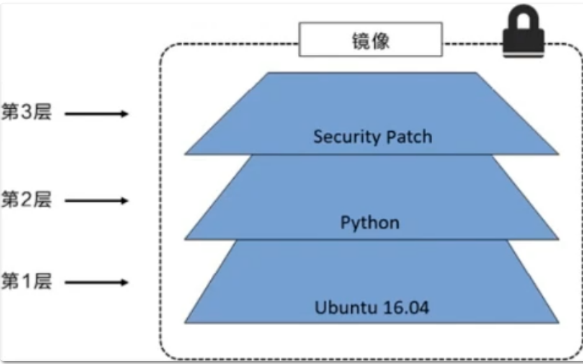
[root@localhost /]# docker inspect redis
[
{
"Id": "sha256:2355926154447ec75b25666ff5df14d1ab54f8bb4abf731be2fcb818c7a7f145",
"RepoTags": [
"redis:latest"
],
"RepoDigests": [
"redis@sha256:800f2587bf3376cb01e6307afe599ddce9439deafbd4fb8562829da96085c9c5"
],
"Parent": "",
"Comment": "",
"Created": "2020-06-10T09:46:52.678534887Z",
"Container": "1ee443d62a9e842d46b868a6ca3f61c0da44a5a6788a8871c89fe6e7c34f5f34",
"ContainerConfig": {
"Hostname": "1ee443d62a9e",
"Domainname": "",
"User": "",
"AttachStdin": false,
"AttachStdout": false,
"AttachStderr": false,
"ExposedPorts": {
"6379/tcp": {}
},
"Tty": false,
"OpenStdin": false,
"StdinOnce": false,
"Env": [
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
"GOSU\_VERSION=1.12",
"REDIS\_VERSION=6.0.5",
"REDIS\_DOWNLOAD\_URL=http://download.redis.io/releases/redis-6.0.5.tar.gz",
"REDIS\_DOWNLOAD\_SHA=42cf86a114d2a451b898fcda96acd4d01062a7dbaaad2801d9164a36f898f596"
],
"Cmd": [
"/bin/sh",
"-c",
"#(nop) ",
"CMD [\"redis-server\"]"
],
"ArgsEscaped": true,
"Image": "sha256:0b2611dbbf667ee7ad92bb5e4b9843ae53f9463fc8c00044b244d8dfd692f175",
"Volumes": {
"/data": {}
},
"WorkingDir": "/data",
"Entrypoint": [
"docker-entrypoint.sh"
],
"OnBuild": null,
"Labels": {}
},
"DockerVersion": "18.09.7",
"Author": "",
"Config": {
"Hostname": "",
"Domainname": "",
"User": "",
"AttachStdin": false,
"AttachStdout": false,
"AttachStderr": false,
"ExposedPorts": {
"6379/tcp": {}
},
"Tty": false,
"OpenStdin": false,
"StdinOnce": false,
"Env": [
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
"GOSU\_VERSION=1.12",
"REDIS\_VERSION=6.0.5",
"REDIS\_DOWNLOAD\_URL=http://download.redis.io/releases/redis-6.0.5.tar.gz",
"REDIS\_DOWNLOAD\_SHA=42cf86a114d2a451b898fcda96acd4d01062a7dbaaad2801d9164a36f898f596"
],
"Cmd": [
"redis-server"
],
"ArgsEscaped": true,
"Image": "sha256:0b2611dbbf667ee7ad92bb5e4b9843ae53f9463fc8c00044b244d8dfd692f175",
"Volumes": {
"/data": {}
},
"WorkingDir": "/data",
"Entrypoint": [
"docker-entrypoint.sh"
],
"OnBuild": null,
"Labels": null
},
"Architecture": "amd64",
"Os": "linux",
"Size": 104124452,
"VirtualSize": 104124452,
"GraphDriver": {
"Data": {
"LowerDir": "/var/lib/docker/overlay2/5b3dce1b97d50bc15d887e6927385bbd29bed4f15c9bc54d17a1a381f6715100/diff:/var/lib/docker/overlay2/40a0d866e6ed74a11ef2b0ea4e1a41f1cdd3dd67eee35da2855636eb878678bd/diff:/var/lib/docker/overlay2/0b5706e4fae9730d0a0e55a5821253856e5b01fb652f2302f0aa6a20f86b504f/diff:/var/lib/docker/overlay2/d95bbbe8d5070275d44dcd391b71f006beff99891ad7b275480007967a3855dc/diff:/var/lib/docker/overlay2/c63955602bef31519e951cd7cc55fbf9d7ca0b4ee3261bed14062e652a2ba66e/diff",
"MergedDir": "/var/lib/docker/overlay2/875b5c067eb8b1b3dce2bd14d2648d6a0426e265a4c6e4ed9af81fb3dd1557ab/merged",
"UpperDir": "/var/lib/docker/overlay2/875b5c067eb8b1b3dce2bd14d2648d6a0426e265a4c6e4ed9af81fb3dd1557ab/diff",
"WorkDir": "/var/lib/docker/overlay2/875b5c067eb8b1b3dce2bd14d2648d6a0426e265a4c6e4ed9af81fb3dd1557ab/work"
},
"Name": "overlay2"
},
"RootFS": {
"Type": "layers",
"Layers": [
"sha256:13cb14c2acd34e45446a50af25cb05095a17624678dbafbcc9e26086547c1d74",
"sha256:e6b49c7dcaac7a2ec2acc379da5f5b1bcc6a5d3badd72814fe945296216557bd",
"sha256:cdaf0fb0082b74223a224c39c2d2ea886c32f53b7e1d5b872d5354aae0df56b8",
"sha256:72d3a7e6fe022824ee2f852ca132030e22c644fbaf8287f4ea8044268abe40b7",
"sha256:67c707dbd847d8310d3b988c3e3d9d9eb53387ede0de472e36a15abbcb6c719c",
"sha256:7b9c5be81844318508f57a5b0574822dabaaed3dc25ee35d960feec3a9e941c4"
]
},
"Metadata": {
"LastTagTime": "0001-01-01T00:00:00Z"
}
}
]

**理解：**

所有的 Docker镜像都起始于一个基础镜像层，当进行修改或培加新的内容时，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 Linux16.04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，

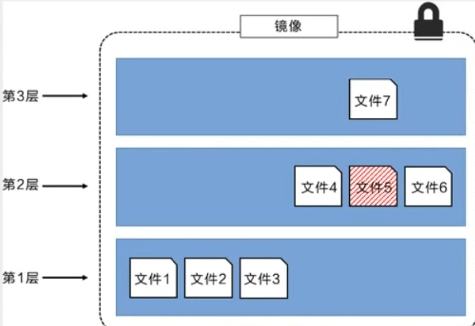
就会在基础镜像层之上创建第二个镜像层；如果继续添加一个安全补丁，就会创健第三个镜像层该像当前已经包含3个镜像层，如下图所示（这只是一个用于演示的很简单的例子）。



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別，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



文种情況下，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

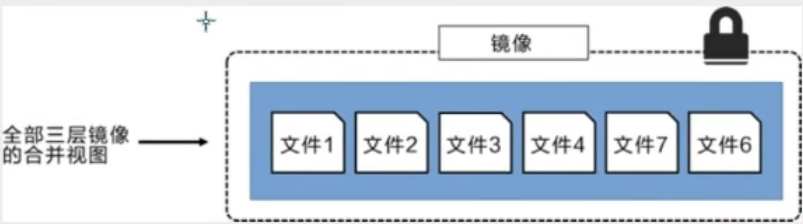
Docker通过存储引擎（新版本采用快照机制）的方式来实现镜像层堆栈，并保证多镜像层对外展示为统一的文件系统

Linux上可用的存储引撃有AUFS、 Overlay2、 Device Mapper、Btrfs以及ZFS。顾名思义，每种存储引擎都基于 Linux中对应的

件系统或者块设备技术，井且每种存储引擎都有其独有的性能特点。

Docker在 Windows上仅支持 windowsfilter 一种存储引擎，该引擎基于NTFS文件系统之上实现了分层和CoW [1]。

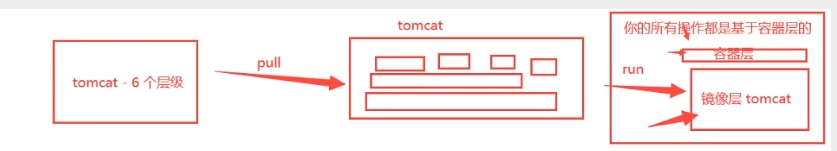
下图展示了与系统显示相同的三层镜像。所有镜像层堆并合井，对外提供统一的视图



特点

Docker 镜像都是只读的，当容器启动时，一个新的可写层加载到镜像的顶部！

这一层就是我们通常说的容器层，容器之下的都叫镜像层！



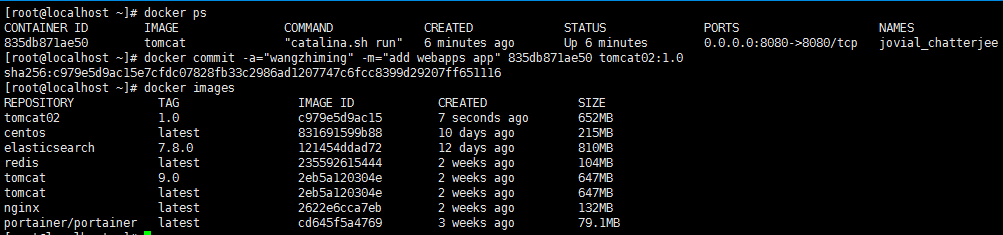
如何提交一个自己的镜像

**commit镜像**

docker commi 提交容器成为一个新的副本
命令和git类似
docker commit -m="提交描述信息" -a="作者" 容器id 目标镜像名:[tag]

实战测试

1.启动一个默认的Tomcat
2.发现这个默认的Tomcat 是没有webapps应用的，镜像的原因，官方镜像默认webapps下面是没有文件的！
3.我自己拷贝进去基本的文件
4.将我们修改后的容器通过commi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



如果你想要保存当前容器的状态，就可以通过commit来提交，获得一个镜像，就好比我们我们使用虚拟机的快照。

**容器数据卷**

**docker的理念**

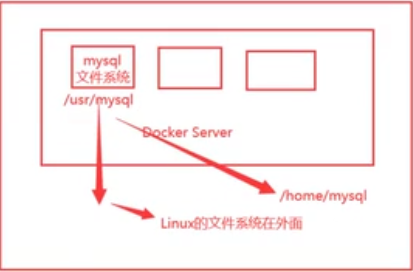
将应用和环境打包成一个镜像！

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数据可以存储本地！

容器之间可以有一个数据共享技术！Docker容器中产生的数据，同步到本地！

这就是卷技术！目录挂载，将我们的容器内的目录，挂载到LLinux上面！



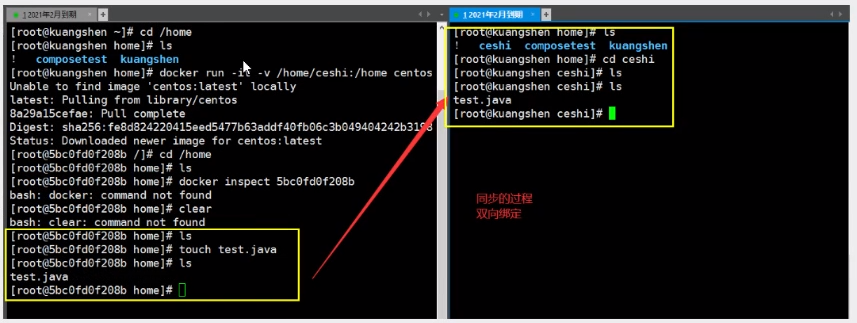
**总结一句话：容器的持久化和同步操作！容器间也是可以数据共享的！**

**使用数据卷**

方式一：直接使用命令来挂载-v
docker run -it -v 主机目录:容器目录
docker run -it -v -p 主机端口:容器端口
测试
[root@localhost home]# docker run -it -v /home/ceshi:/home centos /bin/bash
通过 docker inspect 容器id 查看
[root@localhost /]# docker inspect 0367379bba08



测试是否同步



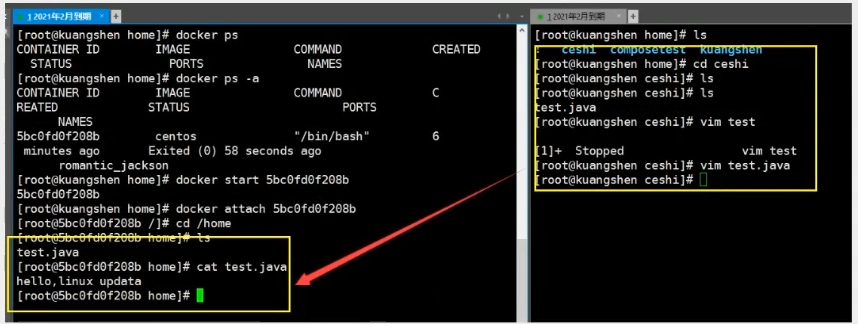
再来测试

1、停止容器

2、宿主机修改文件

3、启动容器

4、容器内的数据依旧是同步的



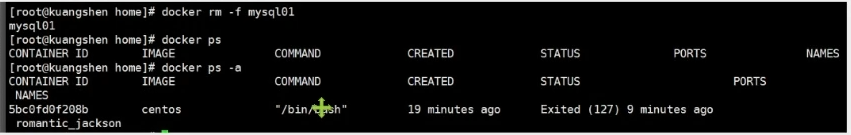
好处：我们以后修改只需要在本地修改即可，容器内会自动同步！

**实战安装mysql**

思考：MySQL的数据持久化的问题

获取mysql镜像
➜ ~ docker pull mysql:5.7
# 运行容器,需要做数据挂载 #安装启动mysql，需要配置密码的，这是要注意点！
# 参考官网hub
docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql:tag
#启动我们得
-d 后台运行
-p 端口映射
-v 卷挂载
-e 环境配置
-- name 容器名字
[root@localhost /]# docker run -d -p 3310:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v /home/mysql/data:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=123456 --name mysql01 mysql:5.7
# 启动成功之后，我们在本地使用sqlyog来测试一下
# sqlyog-连接到服务器的3310--和容器内的3306映射
# 在本地测试创建一个数据库，查看一下我们映射的路径是否ok！

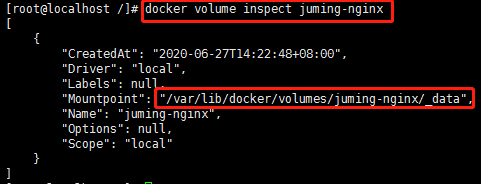
假设我们将容器删除



发现，我们挂载到本地的数据卷依旧没有丢失，这就实现了容器数据持久化功能。

**具名和匿名挂载**

# 匿名挂载
-v 容器内路径!
docker run -d -P --name nginx01 -v /etc/nginx nginx
# 查看所有的volume的情况
➜ ~ docker volume ls
DRIVER VOLUME NAME
local 33ae588fae6d34f511a769948f0d3d123c9d45c442ac7728cb85599c2657e50d
local
# 这里发现，这种就是匿名挂载，我们在 -v只写了容器内的路径，没有写容器外的路劲！
# 具名挂载
➜ ~ docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx
➜ ~ docker volume ls
DRIVER VOLUME NAME
local juming-nginx
# 通过 -v 卷名：容器内路径
# 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下如果指定了目录，docker volume ls 是查看不到的

如何确定是具名挂载还是匿名挂载，还是指定路径挂载
-v 容器内路径 匿名挂载
-v 卷名：容器内路径 具名挂载
-v /宿主机路径：容器内路径 指定路径挂载

拓展：

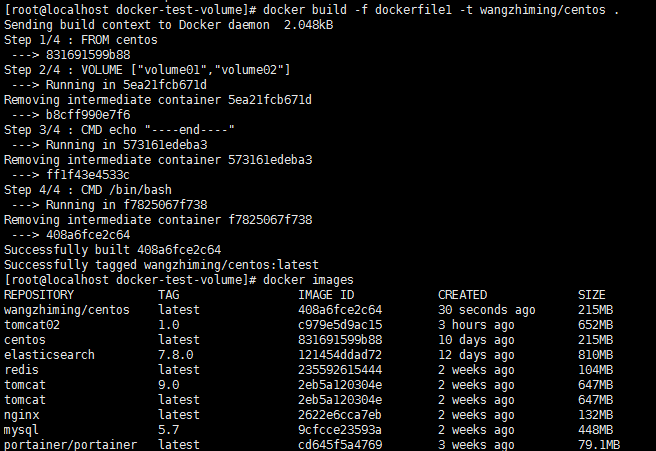
通过-v容器内路径：ro rw 改变读写权限
ro readonly 只读
rw readwrite 可读可写
一旦这个设置了容器权限，容器对我们挂载出来的内容就有限定了！
docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx:ro nginx
docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx:rw nginx
ro只要看到ro说明这个路径只能通过宿主机修改，容器内部无法操作 默认rw

**初始Dockerfile**

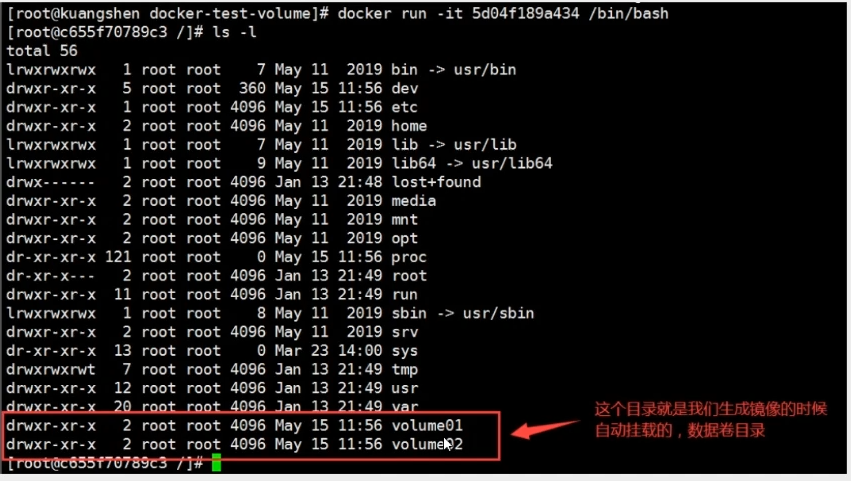
Dockerfile 就是用来构建docker镜像的构建文件！命令脚本！先体验一下！

通过这个脚本可以生成镜像，镜像是一层一层的，脚本一个个的命令，每个命令都是一层！

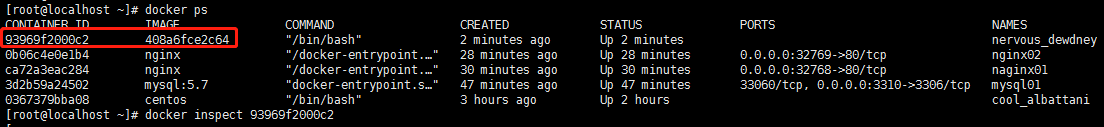
方式二
# 创建一个dockerfile文件，名字可以随便 建议Dockerfile
# 文件中的内容 指令(大写) 参数
FROM centos
VOLUME ["volume01","volume02"]
CMD echo "----end----"
CMD /bin/bash
#这里的每个命令，就是镜像的一层！

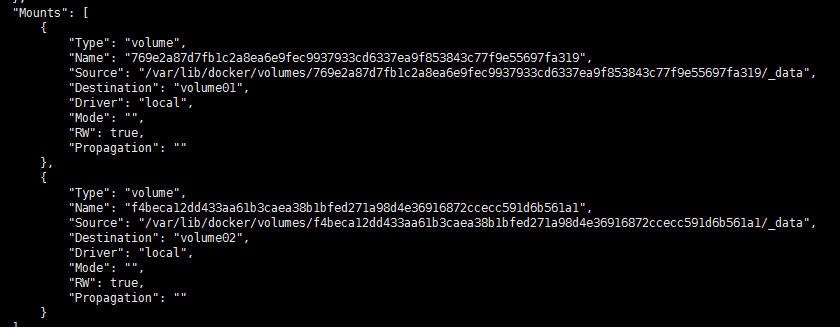


启动自己的容器



查找宿主机挂载路径





测试一下刚才的文件是否同步出去了！

这种方式使用的十分多，因为我们通常会构建自己的镜像！

假设构建镜像时候没有挂载卷，要手动镜像挂载 -v 卷名：容器内路径！

clipboard.png

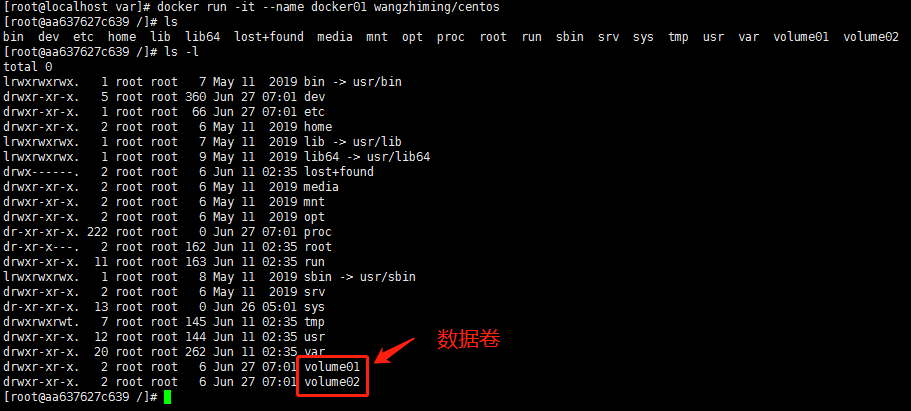
**数据卷容器**

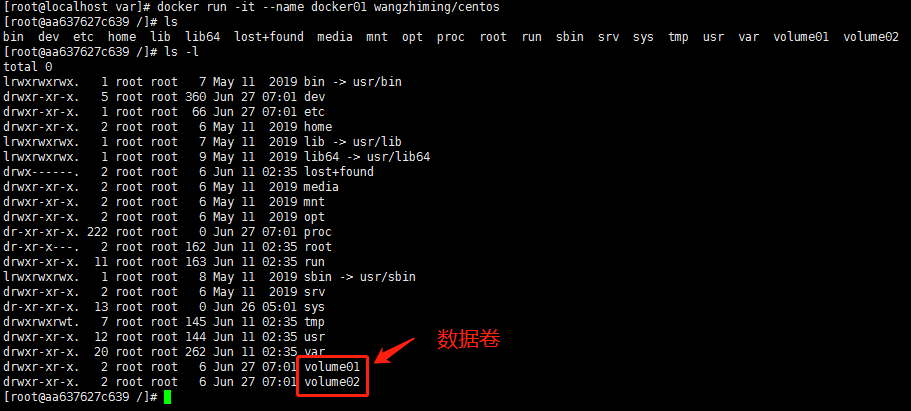
多个MySQL同步数据！

命名的容器挂载数据卷



--volumes-from list Mount volumes from the specified container(s)
# 测试，我们通过刚才启动的

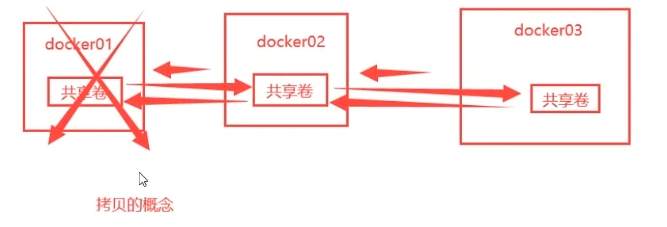




clipboard.png

测试：可以删除docker01，查看一下docker02和docker03是否可以访问这个文件

*测试依旧可以访问*



多个mysql实现数据共享

➜ ~ docker run -d -p 3310:3306 -v /etc/mysql/conf.d -v /var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=123456 --name mysql01 mysql:5.7
➜ ~ docker run -d -p 3311:3306 -e MYSQL\_ROOT\_PASSWORD=123456 --name mysql02 --volumes-from mysql01 mysql:5.7
# 这个时候，可以实现两个容器数据同步！

结论：

容器之间的配置信息的传递，数据卷容器的生命周期一直持续到没有容器使用为止。

但是一旦你持久化到了本地，这个时候，本地的数据是不会删除的！

**DockerFile**

dockerfile是用来构建docker镜像的文件！命令参数脚本！

构建步骤：

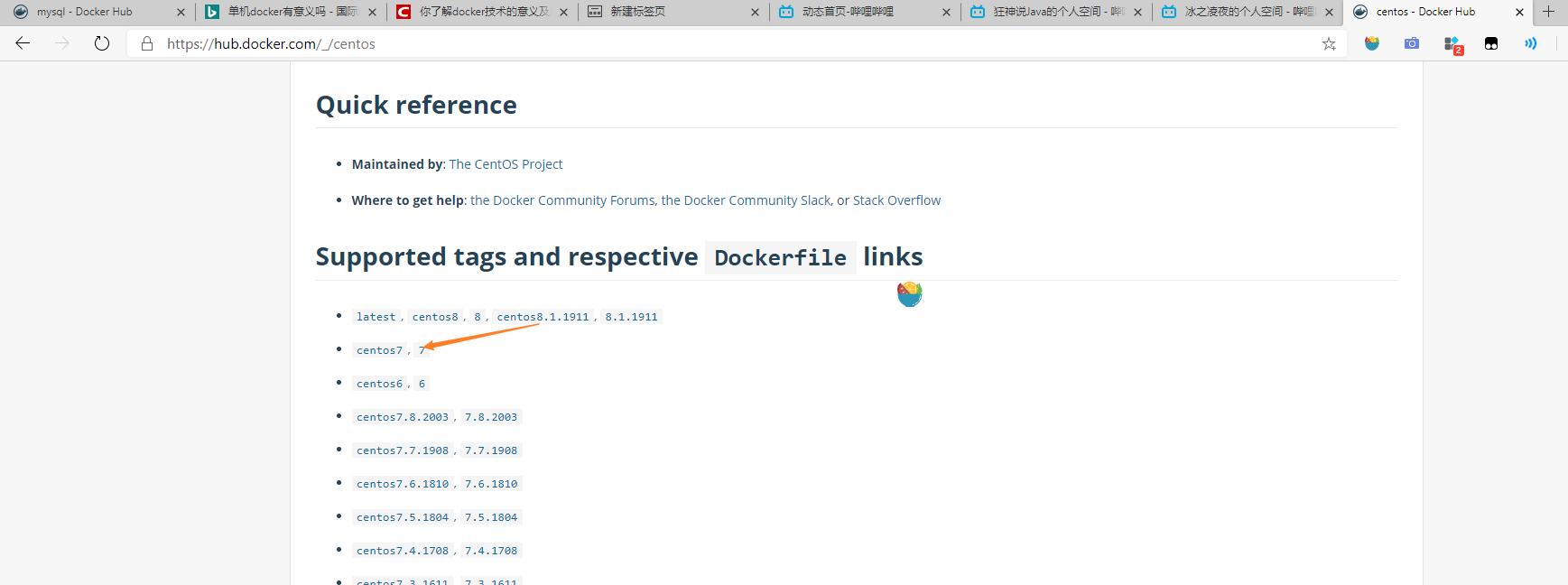
1、 编写一个dockerfile文件

2、 docker build 构建称为一个镜像

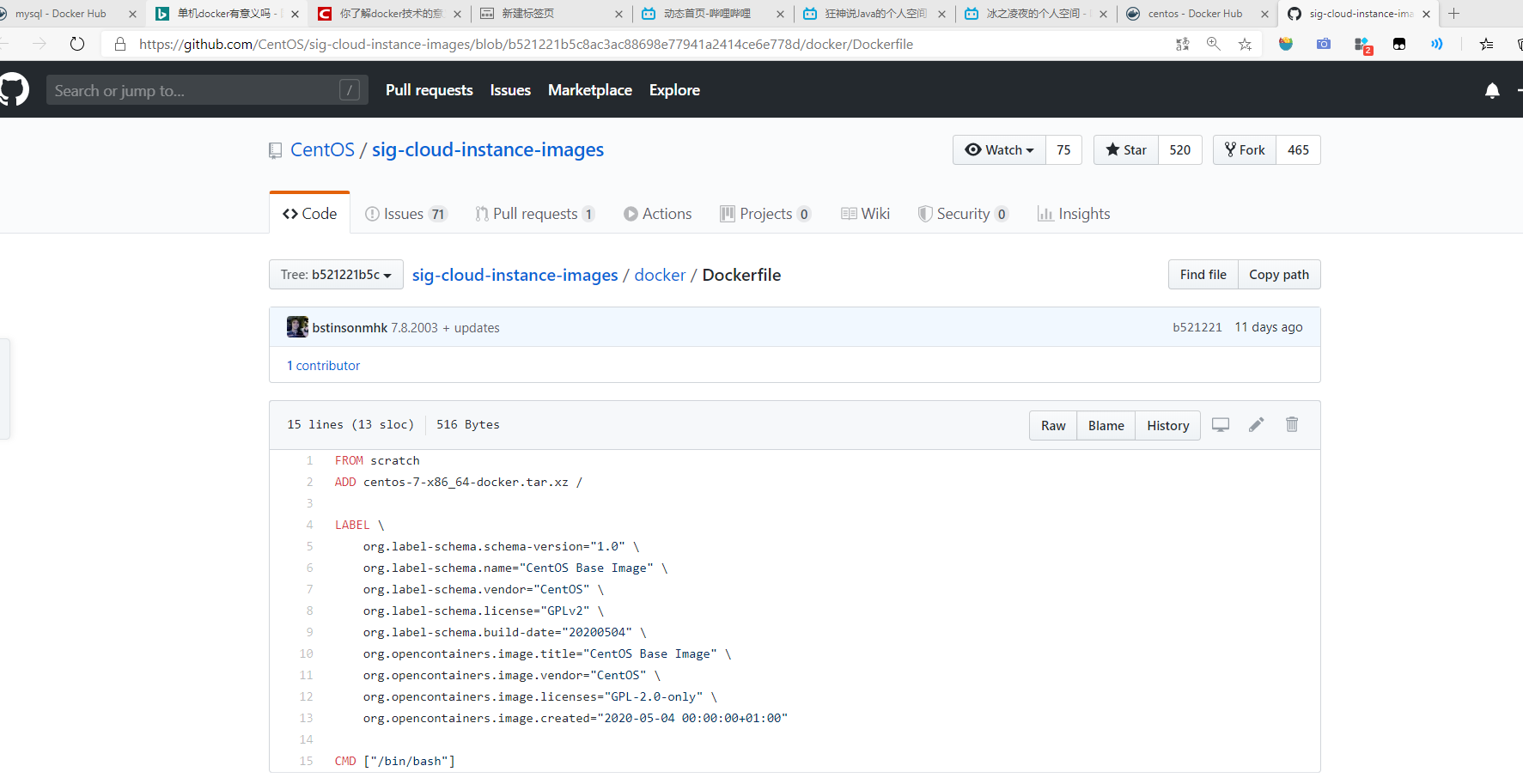
3、 docker run运行镜像

4、 docker push发布镜像（DockerHub 、阿里云仓库)

查看官方是怎么做的



点击后跳到一个Dockerfile



很多官方镜像都是基础包，很多功能没有，我们通常会自己搭建自己的镜像！

官方既然可以制作镜像，那我们也可以！

**DockerFile构建过程**

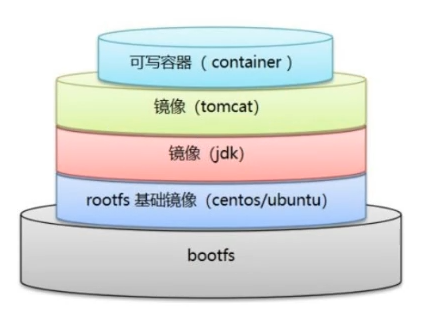
**基础知识：**

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

3、#表示注释

4、每一个指令都会创建提交一个新的镜像曾，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，这个文件十分简单！

Docker镜像逐渐成企业交付的标准，必须要掌握！

DockerFile：构建文件，定义了一切的步骤，源代码

DockerImages：通过DockerFile构建生成的镜像，最终发布和运行产品。

Docker容器：容器就是镜像运行起来提供服务。

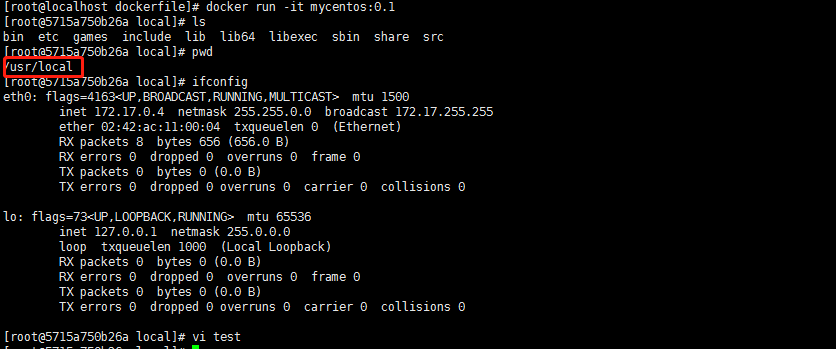
**DockerFile的指令**

FROM # 基础镜像，一切从这里开始构建
MAINTAINER # 镜像是谁写的， 姓名+邮箱
RUN # 镜像构建的时候需要运行的命令
ADD # 步骤，tomcat镜像，这个tomcat压缩包！添加内容 添加同目录
WORKDIR # 镜像的工作目录
VOLUME # 挂载的目录
EXPOSE # 保留端口配置
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。
ENTRYPOINT # 指定这个容器启动的时候要运行的命令，可以追加命令
ONBUILD # 当构建一个被继承 DockerFile 这个时候就会运行ONBUILD的指令，触发指令。
COPY # 类似ADD，将我们文件拷贝到镜像中
ENV # 构建的时候设置环境变量！

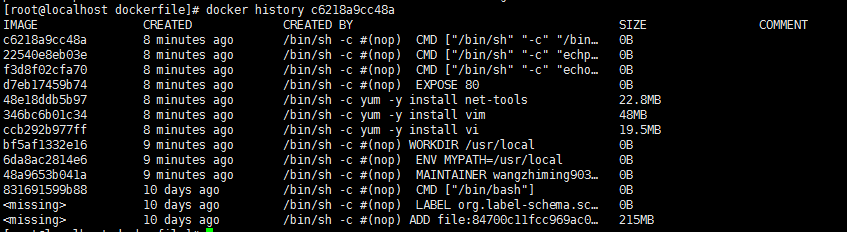
**实战测试**

Docker Gub中99%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进行构建

创建一个自己的centos
# 1.编写Dockerfile文件
[root@localhost dockerfile]# cat mydockerfile
FROM centos
MAINTAINER wangzhiming903308627@qq.com<>
ENV MYPATH /usr/local
WORKDIR $MYPATH
RUN yum -y install vi
RUN yum -y install vim
RUN yum -y install net-tools
EXPOSE 80
CMD echo $MYPATH
CMD echp ---END ---
CMD /bin/bash
# 2、通过这个文件构建镜像
# 命令 docker build -f 文件路径 -t 镜像名:[tag] .
docker build -f mydockerfile-centos -t mycentos:0.1
Successfully built c6218a9cc48a
Successfully tagged mycentos:0.1
3.测试运行



我们可以列出本地镜像的变更历史



CMD 和 ENTRYPOINT区别

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。
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**

# 编写dockerfile文件
$ vim dockerfile-test-cmd
FROM centos
CMD ["ls","-a"]
# 构建镜像
$ docker build -f dockerfile-test-cmd -t cmd-test:0.1 .
# 运行镜像
$ docker run cmd-test:0.1
.
..
.dockerenv
bin
dev
# 想追加一个命令 -l 成为ls -al
$ docker run cmd-test:0.1 -l
docker: Error response from daemon: OCI runtime create failed: container\_linux.go:349: starting container process caused "exec: \"-l\":
executable file not found in $PATH": unknown.
ERRO[0000] error waiting for container: context canceled
# cmd的情况下 -l 替换了CMD["ls","-l"]。 -l 不是命令所有报错