Docker笔记

前言

参考资料:

Docker官方文档

Docker官方安装文档

Docker安装包or二进制文件下载

Docker开源项目 - github

k8s官方文档

The Linux Kernel Archives

centos官网

Docker基础原理详解*lks1139230294*的博客-*CSDN*博客docker原理详解

Docker 的前世今生 - 哈喽沃德先生 - 博客园 (cnblogs.com)

Docker高级网络配置祯*min*的博客-*CSDN*博客docker network配置

Docker网络详解——原理篇*meltsnow*的博客-*CSDN*博客docker 网络

KubeEdge vs K3S：Kubernetes在边缘计算场景的探索勇往直前的专栏-*CSDN*博客k3s kubeedge

内核版本与发行版本（CentOS & Ubuntu）的对应关系

Docker 教程 | 菜鸟教程 (runoob.com)

Docker简介以及Docker历史 (biancheng.net)

Installation on Mac OS X | Docker 中文指南 (widuu.com)

容器和镜像的区别吗？ - 知乎 (zhihu.com)

Linux和UNIX的关系及区别（详解版） (biancheng.net)

Runtime options with Memory, CPUs, and GPUs | Docker Documentation
使用 docker 对容器资源进行限制 - DockOne.io
Index of /doc/Documentation/cgroup-v1/ (kernel.org)
Docker资源管理探秘：Docker背后的内核Cgroups机制-InfoQ

Docker概述

Docker的历史

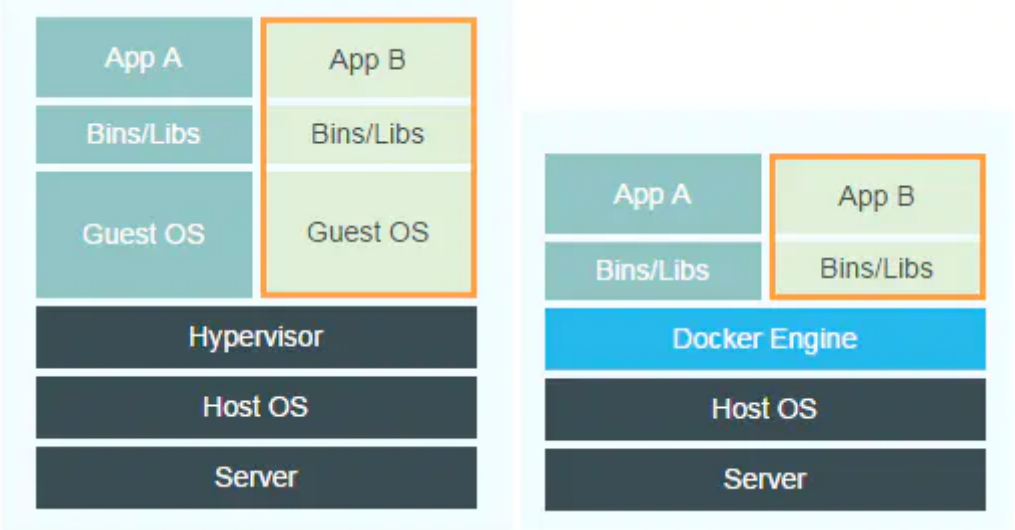
Docker 是 PaaS 提供商 dotCloud 开源的一个基于 LXC 的高级容器引擎，源代码托管在 Github 上，基于**go语言**并遵从Apache2.0协议开源。

Docker与虚拟机

虚拟机是硬件层虚拟化，虚拟出一套硬件，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。系统层隔离。

容器化技术不是模拟一个完整的操作系统。进程层的隔离(共享主机的内核)。

如果应用需要有比如内核模块的配合或者需要运行在不同的OS，用虚拟机更好。



Docker 架构

Docker 包括三个基本概念

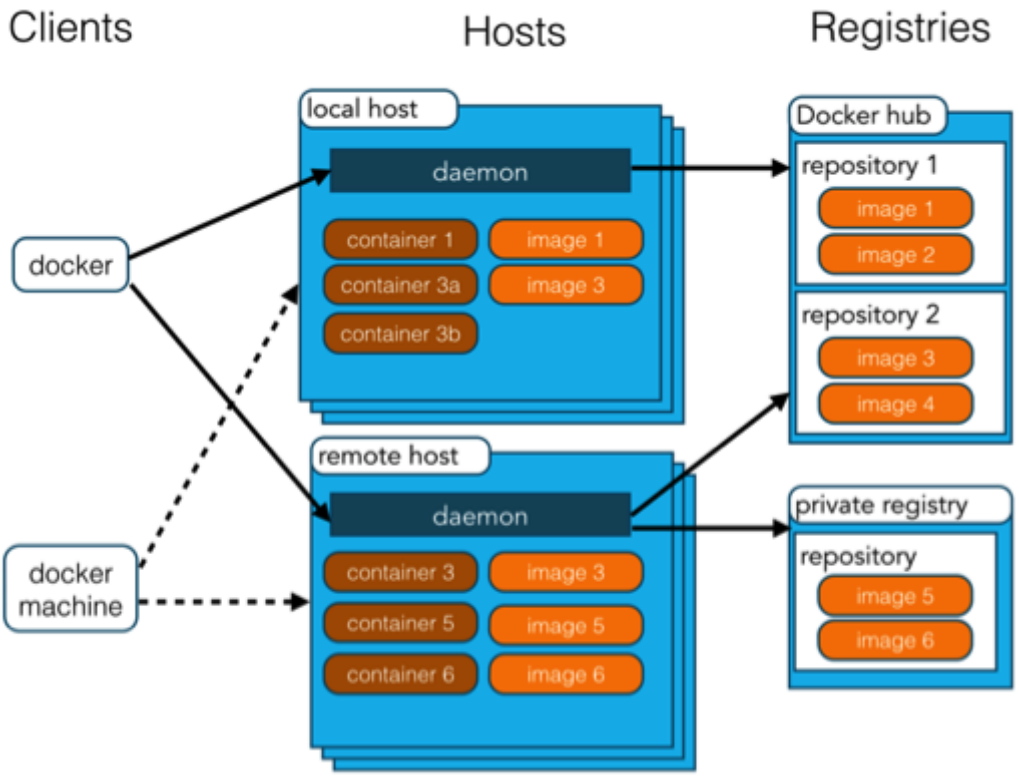
- **镜像（Image）**：Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统。
- **容器（Container）**：镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。
- **仓库（Repository）**：仓库可看成一个代码控制中心，用来保存镜像。

Docker 使用客户端-服务器 (C/S) 架构模式，使用远程API来管理和创建Docker容器。

Docker 容器通过 Docker 镜像来创建。

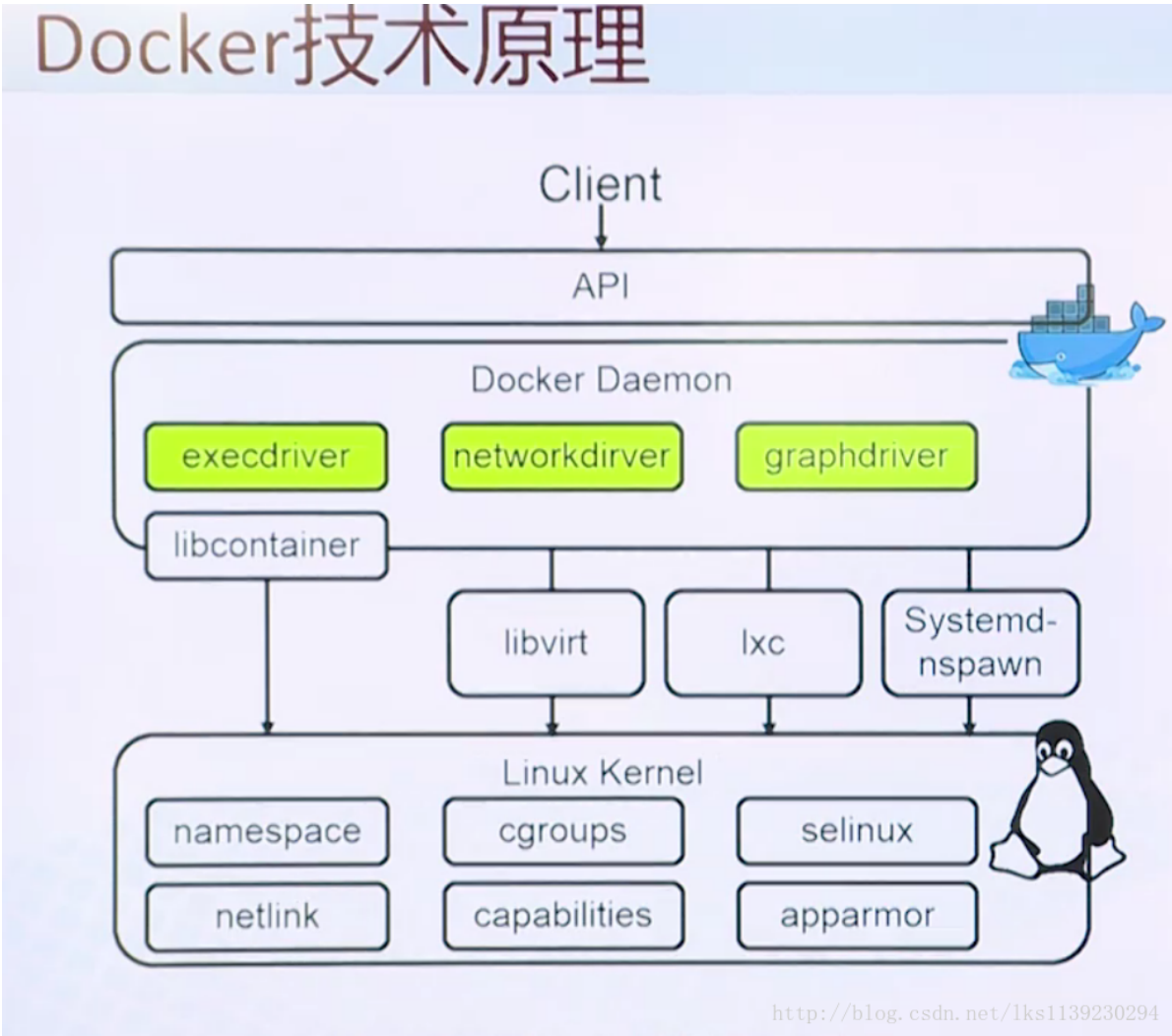
容器与镜像的关系类似于面向对象编程中的对象与类。

Docker	面向对象
容器	对象
镜像	类



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

Docker主要由**execdriver**，**networkdriver**，**graphdriver**来进行管理．其中execdriver负责配置信息的管理，然后通过libcontainer来与namespace，cgroups来完成容器的创建与及管理．而networkdriver主要是完成网络信息的配置，graphdriver主要是对容器镜像的管理．实际上就是通过libvirt，lxc等技术来完成Docker技术管理的。



安装Docker

Docker支持的平台

Windows

一、Docker for Windows Server

Docker Enterprise Edition for **Windows Server 2016**
https://store.docker.com/editions/enterprise/docker-ee-server-windows

二、Docker for Windows

Docker Community Edition for Windows
Requires Microsoft Windows 10 Professional or Enterprise 64-bit.
Docker CE for Windows is Docker designed to run on Windows 10. It is a native Windows application that provides an easy-to-use development environment for building, shipping, and running dockerized apps. Docker CE for Windows uses Windows-native Hyper-V virtualization and networking and is the fastest and most reliable way to develop Docker apps on Windows. Docker CE for Windows supports running both Linux and Windows Docker containers.

https://store.docker.com/editions/community/docker-ce-desktop-windows

三、For previous versions：Docker Toolbox

https://www.docker.com/products/docker-toolbox
To run Docker, your machine must have a **64-bit operating system running Windows 7 or higher**. Additionally, you **must make sure that virtualization is enabled** on your machine.

windows 容器提供了两种级别的隔离技术，分别是Windows Server container 和Hyper-V Container ，前者通过进程和命名空间隔离技术提供应用程序隔离。 Windows Server 容器与容器主机和该主机上运行的所有容器共享内核。 后者通过在高度优化的虚拟机中运行每个容器，在由 Windows Server 容器提供的隔离上扩展。 在此配置中，容器主机的内核不与 Hyper-V 容器共享。

下面这个说法，目前已经不准确了 -- win上也有基于win内核的容器化技术
*Docker*并非是一个通用的容器工具，它依赖于已存在并运行的*Linux*内核环境。
Docker 实质上是在已经运行的 *Linux* 下制造了一个隔离的文件环境，它执行的效率几乎等同于所部署的 *Linux*主机。
因此，*Docker*必须部署在 *Linux* 内核的系统上。如果其他系统想部署 *Docker* 就必须安装一个虚拟 *Linux* 环境。
*Windows*平台安装*Docker*本质上是先装一个*Linux*虚拟机，然后在虚拟机中部署*Docker*。

Linux

Linux操作系统环境要求

有一个shell脚本，用于检查系统是否具有Docker所需的依赖项，以及检查哪些功能可用
https://github.com/docker/docker/blob/master/contrib/check-config.sh

- 一般来说安装Docker(版本≥ 1.8.0)，要求Linux内核版本 ≥ 3.10
- `iptables` version 1.4 or higher
- `git` version 1.7 or higher
- A `ps` executable, usually provided by `procps` or a similar package.
- XZ Utils 4.9 or higher(一个压缩工具)
- A properly mounted `cgroupfs` hierarchy; a single, all-encompassing `cgroup` mount point is not sufficient. See Github issues #2683, #3485, #4568).

若docker启动不正常需要检查内核配置
cat /proc/filesystems
查看是否有cgroup和overlay，若不存在请检查内核配置。
cat /proc/self/cgroup
查看是否存在devices、cpuset、memory、cpu,cpuacct。

不存在的原因：
1． 内核配置没有选上
2． 系统启动后系统的初始化程序没有初始化配置，正常配置的systemd会做好cgroup的初始化

常见的主流Linux发行版安装方式(CentOS、ubuntu、debian)

- 通过个发行版的包管理工具安装(apt-get、yum)
- 官方提供的deb/rpm安装包
- 官方提供编译好的了二进制文件(x86_64、aarch64、armel、armhf、ppc64le、s390x)

常见Linux发行版需要安装三个软件包:
containerd.io、docker-ce-ce、docker-ce-cli

- containerd.io - daemon to interface with the OS API (in this case, LXC - Linux Containers), essentially decouples Docker from the OS, also provides container services for non-Docker container managers
- docker-ce - Docker daemon, this is the part that does all the management work, requires the other two on Linux
- docker-ce-cli - CLI tools to control the daemon, you can install them on their own if you want to control a remote Docker daemon

国产操作系统(中标麒麟、银河麒麟、中科方德、统信USO)

其他Linux发行版或一些嵌入式系统

- 官方提供编译好的了二进制文件(x86_64、aarch64、armel、armhf、ppc64le、s390x)

2021年开局Linux桌面100名排名

https://download.docker.com/linux/

表1: 基于Linux内核各发行版本安装Docker

Linux发行版本	支持的系统架构	Docker版本 / 安装方式	系统版本
CentOS	x86_64、amd64 ARM64、AARCH64	yum包管理工具/官方安装包	CentOS7及以上
Fedora	x86_64、amd64 ARM64、AARCH64	yum包管理工具/官方安装包	Fedora 32 Fedora 33 Fedora 34
Debian	x86_64、amd64 ARM、ARM64、AARCH64	dnf包管理工具/官方安装包	Debian Bullseye 11 (testing) Debian Buster 10 (stable)
Ubuntu	x86_64、amd64 ARM、ARM64、AARCH64	apt包管理工具/官方安装包	Ubuntu Hirsute 21.04 Ubuntu Groovy 20.10 Ubuntu Focal 20.04 (LTS) Ubuntu Bionic 18.04 (LTS) Ubuntu Xenial 16.04 (LTS)
Raspbian	ARM、ARM64、AARCH64	apt包管理工具/官方安装包	Raspbian Bullseye 11 (testing) Raspbian Buster 10 (stable)

表2: 支持安装Docker的非Linux内核系统

操作系统	支持的系统架构	Docker版本 / 安装方式	
Win10	x86_64/AMD64	Docker Desktop for Windows (基于win10的Hyper-V虚拟机)	
Win7、Win8	x86_64/AMD64	Docker toolbox (一个工具集, 包含Oracle VM Virtualbox)	
Mac	x86_64/AMD64	Docker Desktop for Mac	

Docker基础命令

Docker reference | Docker Documentation -- 官方文档

帮助命令

查看相关指令具体用法，特别docker run命令会附带很多参数，参考文档详细了解。

```
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 `command` --help   # 命令的具体使用
```

帮助文档的地址：https://docs.docker.com/reference/

镜像命令

导入docker镜像

```
docker load -i xxx.tar
```

查看已导入的docker镜像

```
docker images
```

生成docker镜像

```
$ docker build -t myapps .  # . 指Dockerfile所在当前目录
```

导出docker镜像

```
docker save -o xxx.tar xxx
```

docker images 查看所有本地主机上的镜像

```
[root@AlibabaECS ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
hello-world         latest              bf756fb1ae65        8 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            镜像大小

可选项
-a, --all       # 列出所有的镜像
-q, --quiet     # 只显示镜像的id
```

docker search 搜索镜像

```
[root@AlibabaECS ~]# docker search mysql
NAME                              DESCRIPTION
STARS               OFFICIAL            AUTOMATED
mysql                             MySQL is a widely used, open-source relation…
9911


可选项
--filter , -f       Filter output based on conditions provided

例:
docker search mysql -f=stars=5000
```

docker pull 下载镜像

下载镜像 docker pull 镜像名[:tag]

```
[root@AlibabaECS ~]# docker pull mysql
Using default tag: latest   # 如果不写tag,默认就是latest
latest: Pulling from library/mysql
bf5952930446: Pull complete # 分层下载，docker image的核心: 联合文件系统
8254623a9871: Pull complete
938e3e06dac4: Pull complete
ea28ebf28884: Pull complete
f3cef38785c2: Pull complete
894f9792565a: Pull complete
1d8a57523420: Pull complete
6c676912929f: Pull complete
ff39fdb566b4: Pull complete
fff872988aba: Pull complete
4d34e365ae68: Pull complete
7886ee20621e: Pull complete
Digest: sha256:c358e72e100ab493a0304bda35e6f239db2ec8c9bb836d8a427ac34307d074ed
# 签名
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest # 真实地址
```

```
指定版本下载
```

```
[root@AlibabaECS ~]# docker pull mysql:5.7
5.7: Pulling from library/mysql
bf5952930446: Already exists
8254623a9871: Already exists
938e3e06dac4: Already exists
ea28ebf28884: Already exists
f3cef38785c2: Already exists
894f9792565a: Already exists
1d8a57523420: Already exists
5f09bf1d31c1: Pull complete
1b6ff254abe7: Pull complete
74310a0bf42d: Pull complete
d398726627fd: Pull complete
Digest: sha256:da58f943b94721d46e87d5de208dc07302a8b13e638cd1d24285d222376d6d84
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7
```

```
docker rmi 删除镜像
[root@AlibabaECS ~]# docker rmi -f 容器id                    # 删除指定的容器
[root@AlibabaECS ~]# docker rmi -f  容器id 容器id 容器id    # 删除多个容器
[root@AlibabaECS ~]# docker rmi -f $(docker images -aq)  # 删除全部容器
```

容器命令

容器启动

```
#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
#docker pull centos

#新建容器并启动
#docker run [可选参数] image

#参数说明
--name = "Name"     容器名字用来区分容器
-d                  后台方式运行
-it                 使用交互方式运行，进入容器查看区分
-p                  指定容器的端口 -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@AlibabaECS bin]# docker run -it centos /bin/bash
[root@94d468db18da /]# ls  # 查看容器内的centos，基础版本，很多命令都是不完善的!
bin  etc   lib    lost+found  mnt  proc  run   srv  tmp  var
dev  home  lib64  media       opt  root  sbin  sys  usr

#退出容器
exit                # 直接容器停止并退出
Ctrl + P + Q        # 容器不停止退出

[root@94d468db18da /]# exit
```

列出所有的运行的容器

```
docker ps `参数`        # 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
```

删除容器

```
docker rm 容器id                    # 删除指定容器，不能删除正在运行的容器，如果要强制删除
rm -f
docker rm -f $(docker ps -aq)    # 删除所有的容器
docker ps -aq|xargs docker rm    # 删除所有的容器

#启动和停止容器的操作
docker start 容器id         # 启动容器
docker restart 容器id       # 重启容器
docker stop 容器id          # 停止当前正在运行的容器
docker kill 容器id          # 强制停止当前容器

常用其他命令
后台启动容器
```

查看容器日志

```
#docker logs -f -t --tail 容器ID
[root@AlibabaECS /]# docker run -d centos /bin/sh -c "while true;do echo
kuangshen;sleep 1;done"
-tf               # 显示日志
--tail number     # 要显示的日志条数

[root@AlibabaECS /]# docker logs -ft --tail f1178d5b0bd8
```

查看容器中的进程信息

docker top 容器id

```
root@ubuntu-Vostro-3268:/home# docker top 244d50e7bc40
UID           PID                 PPID                C                   STIME
           TTY                 TIME             CMD
root          20474               20444               0                   4月26
            pts/0               00:00:00         /bin/sh
```

查看容器信息

```
[root@AlibabaECS /]# docker inspect f1178d5b0bd8
[
    {
        "Id":
"f1178d5b0bd8eea4e0734056c03b35da8a829390d7000d90f863f56fe59af2a3",
        "Created": "2020-08-31T05:10:13.714768846Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while true;do echo kuangshen;sleep 1;done"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 21626,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2020-08-31T05:10:13.994851078Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image":
"sha256:0d120b6ccaa8c5e149176798b3501d4dd1885f961922497cd0abef155c869566",
        "ResolvConfPath":
"/var/lib/docker/containers/f1178d5b0bd8eea4e0734056c03b35da8a829390d7000d90f863
f56fe59af2a3/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/f1178d5b0bd8eea4e0734056c03b35da8a829390d7000d90f863
f56fe59af2a3/hostname",
        "HostsPath":
"/var/lib/docker/containers/f1178d5b0bd8eea4e0734056c03b35da8a829390d7000d90f863
f56fe59af2a3/hosts",
        "LogPath":
"/var/lib/docker/containers/f1178d5b0bd8eea4e0734056c03b35da8a829390d7000d90f863
f56fe59af2a3/f1178d5b0bd8eea4e0734056c03b35da8a829390d7000d90f863f56fe59af2a3-
json.log",
        "Name": "/stupefied_colden",
        "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/overlay2/00a62ec6bd23d1b97f6525f3617759eab6a0d3f283aee8b3591f5e
2dbd1d916f-
init/diff:/var/lib/docker/overlay2/f56573c4e87525abbd1aa5e97f8553016e5b63c6d1773
169e87e0b59afc5d845/diff",
                "MergedDir":
"/var/lib/docker/overlay2/00a62ec6bd23d1b97f6525f3617759eab6a0d3f283aee8b3591f5e
2dbd1d916f/merged",
                "UpperDir":
"/var/lib/docker/overlay2/00a62ec6bd23d1b97f6525f3617759eab6a0d3f283aee8b3591f5e
2dbd1d916f/diff",
                "WorkDir":
"/var/lib/docker/overlay2/00a62ec6bd23d1b97f6525f3617759eab6a0d3f283aee8b3591f5e
2dbd1d916f/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "f1178d5b0bd8",
            "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 kuangshen;sleep 1;done"
            ],
            "Image": "centos",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "org.label-schema.build-date": "20200809",
                "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":
"d83e7bbffe1ec2f7d31fa4887e986b5e10d0ce83a1a2d2e447617511f3fe71d3",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
            "SandboxKey": "/var/run/docker/netns/d83e7bbffe1e",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"310a86df2f690f9358063ad5ac71ec98eb16a7eab51e13f0e8f2b9ebe5239ea9",
            "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":
"04038c2f1d641f91c97253ba1e8dfc890f6a9846ba6cb7ea66235079d138c319",
                    "EndpointID":
"310a86df2f690f9358063ad5ac71ec98eb16a7eab51e13f0e8f2b9ebe5239ea9",
                    "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

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

查看docker状态

docker stats　　#查看容器状态

镜像-UnionFs

Docker分层原理与内部结构

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，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。

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上可用的存储引擎有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，并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]。
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!

Docker进阶命令

容器数据卷 -- 数据持久化

https://docs.docker.com/storage/volumes/

容器卷挂载： 匿名挂载、具名挂载、指定路径挂载
所有的docker容器内的卷，没有指定宿主机目录的情况下都是在/var/lib/docker/volumes/xxxx/_data下;

-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(docker volume ls查看不到的)

容器卷挂载权限和挂载模式
当使用systemd管理Docker守护程序的启动和停止时，在systemd单元文件中有一个选项可以控制Docker守护程序本身的装载传播，称为MountFlags。此设置的值可能会导致Docker看不到在装载点上所做的装载传播更改。例如，如果此值为slave，则可能无法在卷上使用共享或rshared传播。

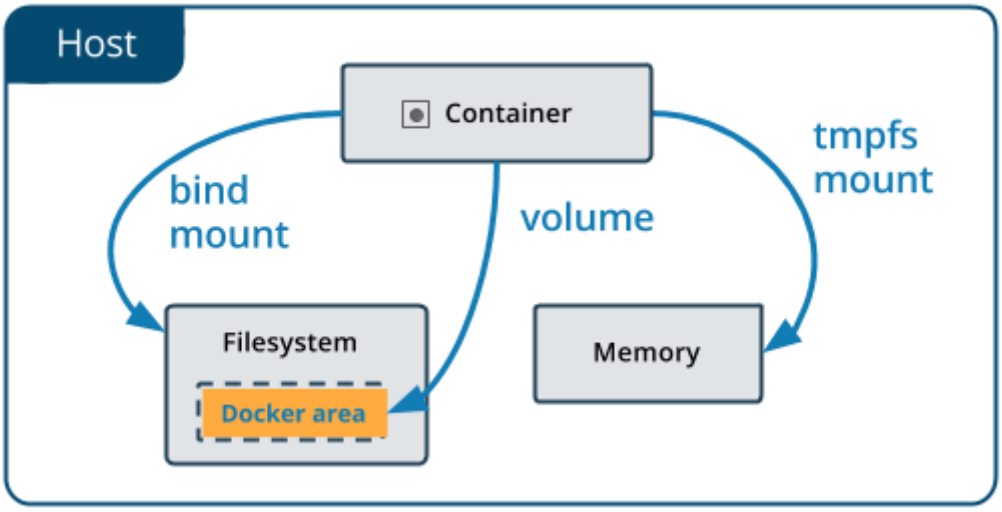
docker绑定数据卷默认模式是[private]. 使用挂载目录做容器的数据卷, 在宿主机上进行mount/umount操作不能同步到容器内
需要在容器进行绑定挂载前, 在宿主机的指定目录上, 先挂载远程目录.
容器启动时, -v 绑定数据卷设置模式位shared

```
--volume=[host-src:]container-dest[:<options>]

#options
ro                  #readonly  只读
rw                  #readwrite 可读可写(默认)

#共用其他容器的数据卷
--volumes-from=
```

容器TMPFS



The example below mounts an empty tmpfs into the container with the `rw`, `noexec`, `nosuid`, and `size=65536k` options.

```
$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my_image
```

Docker网络

Docker高级网络配置祯*min*的博客-*CSDN*博客docker network配置

Docker网络详解——原理篇*meltsnow*的博客-*CSDN*博客docker 网络

安装Docker时，它会自动创建三个网络，bridge（创建容器默认连接到此网络）、 none 、host
使用docker run创建Docker容器时，可以用 --net 选项指定容器的网络模式，Docker可以有以下4种网络模式：

host模式：　　使用 --net=host 指定。
none模式：　　使用 --net=none 指定。
bridge模式：　使用 --net=bridge 指定，默认设置。
container模式：使用 --net=container:NAME_or_ID 指定。

网络模式	简介
Host	容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口
Bridge	此模式会为每一个容器分配、设置IP等，并将容器连接到一个docker0虚拟网桥，通过docker0网桥以及Iptables nat表配置与宿主机通信
None	该模式关闭了容器的网络功能
Container	创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围

Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。

一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。但如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。

```
#docker0特点：默认，域名不能访问，--link可以打通连接

#自定义网络
#可以自定义网桥
#进一步控制哪些容器之间可以通讯
--driver bridge
--subnet 192.168.0.0/16     子网
--gateway 192.168.0.1       网关

[root@AlibabaECS ~]# docker network create --driver bridge --subnet
192.168.0.0/16 --gateway 192.168.0.1 mynet
dd7c8522864cb87c332d355ccd837d94433f8f10d58695ecf278f8bcfc88c1fc
[root@AlibabaECS ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
04038c2f1d64        bridge              bridge              local
81476375c43d        host                host                local
dd7c8522864c        mynet               bridge              local
64ba38c2cb2b        none                null                local

[root@AlibabaECS ~]# docker run -d -P --name tomcat-net-01 --net mynet tomcat
1de6f5994a480160d932de239b104b366ebd5b954e740a5ab8c0d5aeea8f5ba5
[root@AlibabaECS ~]# docker run -d -P --name tomcat-net-02 --net mynet tomcat
f26916a49e5ee239aee23584020e0d23d53d2e644d5cb5155d831edc0803d957
[root@AlibabaECS ~]# docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id":
"dd7c8522864cb87c332d355ccd837d94433f8f10d58695ecf278f8bcfc88c1fc",
        "Created": "2020-09-05T12:43:54.847233062+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "192.168.0.0/16",
                    "Gateway": "192.168.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "1de6f5994a480160d932de239b104b366ebd5b954e740a5ab8c0d5aeea8f5ba5":
{
                "Name": "tomcat-net-01",
                "EndpointID":
"c308999d4e51ed9e5975f3b4f3c1d468bfb08d93de7561d55062db055f44ef18",
                "MacAddress": "02:42:c0:a8:00:02",
```

```
                "IPv4Address": "192.168.0.2/16",
                "IPv6Address": ""
            },
            "f26916a49e5ee239aee23584020e0d23d53d2e644d5cb5155d831edc0803d957":
{
                "Name": "tomcat-net-02",
                "EndpointID":
"8d9dbdd6ca119559ef4f1dd82a36e0d279c0b8284fe19f36c6d992047937a764",
                "MacAddress": "02:42:c0:a8:00:03",
                "IPv4Address": "192.168.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```


有时Docker会运行一些网络服务，如果Docker使用的内部网络，那么我们从本机以外的地方就没法访问服务了。Docker提供了端口映射的功能，将本机的某个端口映射到容器内部开放的端口，因此我们可以通过访问本机的指定端口即可访问容器内部服务了。可以使用-P或者-p进行端口映射。

使用 -P 时，Docker 会随机映射一个端口到内部容器开放的网络端口。
使用 -p 时，可以指定要映射的端口，在一个指定端口上只可以绑定一个容器容器互联
默认情况下，各容器之间是互通的。当使用选项-icc=false重启docker后，容器之间就互相隔离了。容器的访问控制，主要通过 Linux 上的 iptables 防火墙来进行管理和实现。

可以通过-link name:alias命令连接指定容器，Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主主机上，从而避免了暴露数据库端口到外部网络上。

跨宿主机容器间的通讯之macvlan
跨宿主机容器间的通讯之overlay

Docker资源控制

Runtime options with Memory, CPUs, and GPUs | Docker Documentation
使用 docker 对容器资源进行限制 - DockOne.io
Index of /doc/Documentation/cgroup-v1/ (kernel.org)
Docker资源管理探秘：Docker背后的内核Cgroups机制-InfoQ

Docker使用内核的 namespace 来做容器之间的隔离，使用内核的 cgroups 来做容器的资源限制(CPU、内存和 IO)

Docker访问硬件资源

宿主机要能驱动硬件，容器共享的宿主机的驱动。

对边缘设备而言，在支持容器化运行的条件下，需要在容器内获取宿主机的硬件资源，完成与宿主机硬件资源的交互。通常在宿主机提供驱动的情况下，容器内需要通过SPI、I2C、UART、USB等协议完成数据的交互。

--device
–privileged

Dockefile

https://blog.51cto.com/14156658/2497164
https://blog.csdn.net/atlansi/article/details/87892016

构建Docker镜像 -- DockerFile

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              # 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
```