

Docker实战应用（一）

什么是Docker

Docker 最初是 dotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目，它是基于 dotCloud 公司多年云服务技术的一次革新，并于 2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 GitHub 上进行维护。

Docker 自开源后受到广泛的关注和讨论，至今其 GitHub 项目已经超过 4 万 6 千个星标和一万多个 fork。甚至由于 Docker 项目的火爆，在 2013 年底，dotCloud 公司决定改名为 Docker。

Docker 使用 Google 公司推出的 Go 语言 进行开发实现，基于 Linux 内核的 cgroup，namespace，以及 Union FS 等技术，对进程进行封装隔离，属于操作系统层面的**虚拟化技术**。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为**容器**。

官方的说法：

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker 是一个用于开发，交付和运行应用程序的开放平台。Docker 使您能够将应用程序与基础架构分开，从而可以快速交付软件。借助 Docker，您可以以与管理应用程序相同的方式来管理基础架构。通过利用 Docker 快速交付，测试和部署代码的方法，您可以大大减少编写代码和在生产环境中运行代码之间的延迟。

总之： Docker 是一种容器技术，它能够解决软件安装和迁移过程中环境问题。

为什么要使用Docker

作为一种新兴的虚拟化方式，**Docker** 跟传统的虚拟化方式相比具有众多的优势。



传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程



容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。

更高效的利用系统资源

由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，**Docker** 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

更快速的启动时间

传统的虚拟机技术启动应用服务往往需要数分钟，而 **Docker** 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。

一致的运行环境

开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 **Docker** 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 **「这段代码在我机器上没问题啊」** 这类问题。

持续交付和部署

对开发和运维人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。

使用 **Docker** 可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过 **Dockerfile** 来进行镜像构建，并结合持续集成(Continuous Integration)系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合持续交付(Continuous Delivery) 系统进行自动部署。

而且使用 **Dockerfile** 使镜像构建透明化，不仅仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。

更轻松的迁移

由于 Docker 确保了执行环境的一致性，使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

更轻松的维护和扩展

Docker 使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker 团队同各个开源项目团队一起维护了一大批高质量的 官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

对比传统虚拟机总结

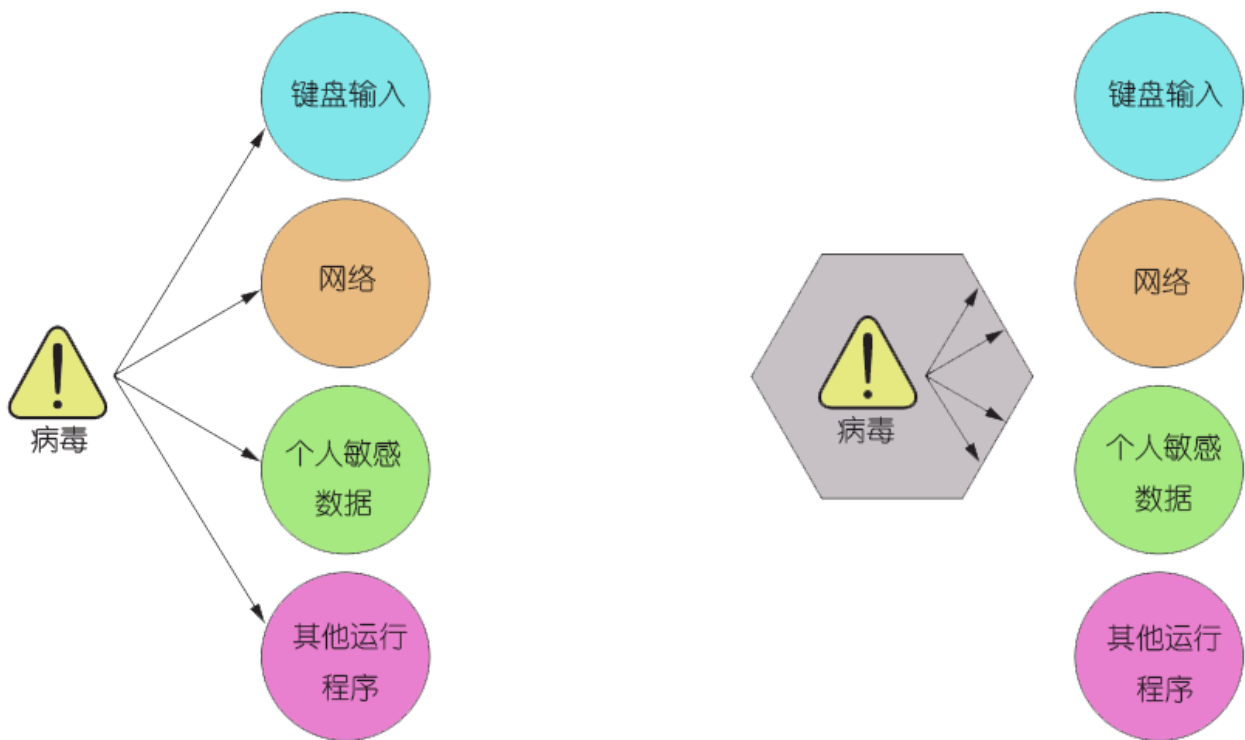
特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

安全性

Docker的使用还有一个重要原因，那就是安全性问题。原因有三：

- 1) 一个程序可能遇到特定的攻击。
- 2) 牛逼而资深的程序员写出来的程序它副作用也很牛逼。
- 3) 程序因为意外，也可以变成具有攻击的bug程序。

通过下图，我们理解一下有容器和没有容器的运行环境的安全性：

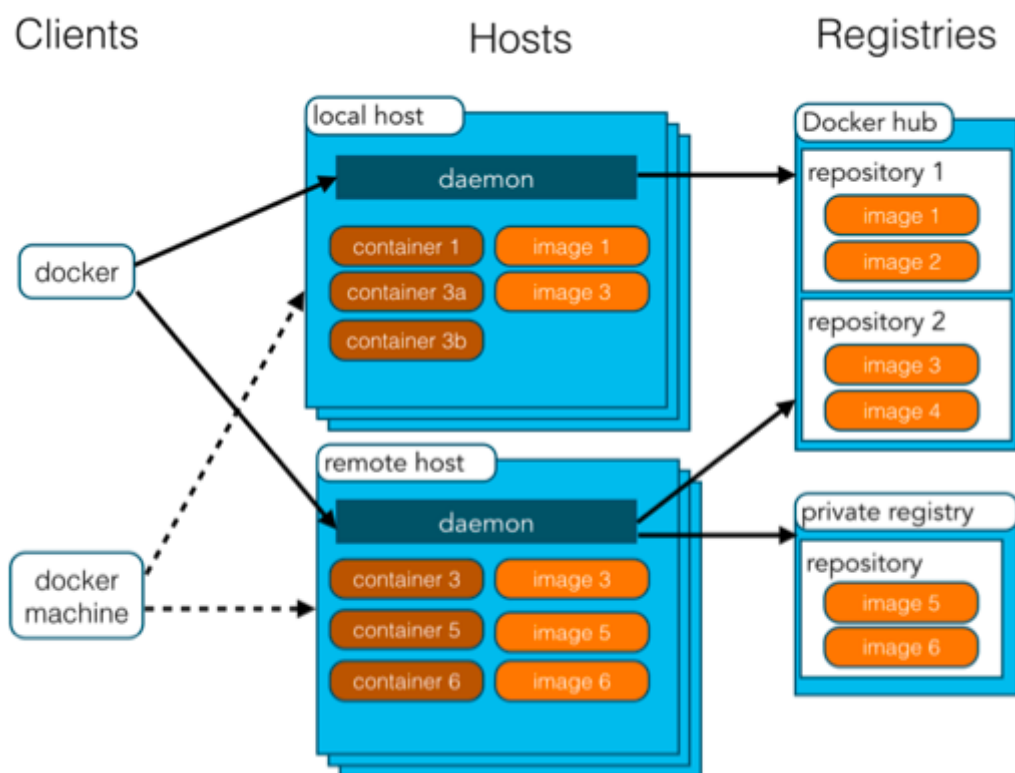


Docker架构

Docker 包括三个基本概念：

- 镜像 (Image)
- 容器 (Container)
- 仓库 (Repository)

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 镜像

我们都知道，操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。

而 Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu 16.04 最小系统的 root 文件系统。

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

分层存储

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 Union FS 的技术，将其设计为分层存储的架构。

所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。关于镜像构建，将会在后续相关章节中做进一步的讲解。

Docker 容器

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

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

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

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

数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器删除或者重新运行之后，数据却不会丢失。

Docker 仓库

仓库可看成一个代码控制中心，用来保存镜像。

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

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

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

以 Ubuntu 镜像 为例，`ubuntu` 是仓库的名字，其内包含有不同的版本标签，如，`16.04`, `18.04`。我们可以通过 `ubuntu:16.04`，或者 `ubuntu:18.04` 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 `ubuntu`，那将视为 `ubuntu:latest`。

公有 Docker 仓库服务

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

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

由于某些原因，在国内访问这些服务可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务 (**Registry Mirror**)，这些镜像服务被称为 **加速器**。常见的有 阿里云加速器、DaoCloud 加速器 等。使用加速器会直接从国内的地址下载 Docker Hub 的镜像，比直接从 Docker Hub 下载速度会提高很多。

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

私有 Docker 仓库服务

除了使用公开服务外，用户还可以在本地搭建私有 Docker 仓库。Docker 官方提供了 Docker Registry 镜像，可以直接使用做为私有 Registry 服务。

开源的 Docker 仓库镜像只提供了 Docker Registry API 的服务端实现，足以支持 `docker` 命令，不影响使用。但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。在官方的商业化版本 Docker Trusted Registry 中，提供了这些高级功能。

除了官方的 Docker Registry 外，还有第三方软件实现了 Docker Registry API，甚至提供了用户界面以及一些高级功能。比如，Harbor 和 Sonatype Nexus。

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 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安装

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

安装最新版

```
1  # 1、卸载旧版本的Docker
2  yum remove docker \
3  docker-client \
4  docker-client-latest \
5  docker-common \
6  docker-latest \
7  docker-latest-logrotate \
8  docker-logrotate \
9  docker-engine
10 # 2、yum源更新到最新
11 yum update
12 # 3、安装需要的软件包， yum-util提供yum-config-manager功能，另外两个是devicemapper
    驱动依赖的
13 yum install -y yum-utils device-mapper-persistent-data lvm2
14 # 4、设置yum国内安装镜像源
15 yum-config-manager --add-repo \
16 https://download.docker.com/linux/centos/docker-ce.repo
17 # 官方镜像：https://download.docker.com/linux/centos/docker-ce.repo
18 # 阿里镜像：http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
19 # 5、更新yum软件源缓存
20 # centos7
21 # yum makecache fast
22 # centos8
23 # dnf makecache
24 # 6、安装最新版docker，如果想要安装指定版本，参考下面的指定版本安装步骤执行后续安装
25 yum install -y \
26 docker-ce \
27 docker-ce-cli \
28 containerd.io \
29 docker-compose-plugin
30 # 7、查看docker版本，验证是否验证成功
31 docker -v
```

安装特定版

```
1  # 需要预先执行安装最新版的前5个步骤
2  # 1、列出并排序您存储库中可用的版本。此示例按版本号（从高到低）对结果进行排序。
3  yum list docker-ce --showduplicates | sort -r
4  # 列表如：
5  # docker-ce.x86_64 3:18.09.1-3.el7 docker-ce-stable
6  # docker-ce.x86_64 3:18.09.0-3.el7 docker-ce-stable
7  # docker-ce.x86_64 18.06.1.ce-3.el7 docker-ce-stable
```

```

8 # docker-ce.x86_64 18.06.0.ce-3.el7 docker-ce-stable
9 # 2、通过其完整的软件包名称安装特定版本
10 yum install -y \
11 docker-ce-<VERSION_STRING> \
12 docker-ce-cli-<VERSION_STRING> \
13 containerd.io \
14 docker-compose-plugin
15 # VERSION_STRING: 列表的（第二列），从第一个冒号（:）一直到第一个连接字符，连接字符
  为-。
16 # 例如：18.09.1或18.06.1.ce
17 # 3、查看docker版本，验证是否验证成功
18 docker -v

```

安装成功截图，下面是查看版本号。

```

If you would like to use Docker as a non-root user, you should now consider
adding your user to the "docker" group with something like:

    sudo usermod -aG docker your-user

Remember that you will have to log out and back in for this to take effect!

WARNING: Adding a user to the "docker" group will grant the ability to run
containers which can be used to obtain root privileges on the
docker host.
Refer to https://docs.docker.com/engine/security/security/#docker-daemo
n-attack-surface
for more information.
[root@localhost ~]# docker -v
Docker version 19.03.11, build 42e35e61f3

```

启动Docker

```

1 # 启动Docker命令
2 systemctl start docker
3 # 查看启动结果
4 systemctl status docker

```

```

[root@localhost ~]# systemctl start docker
[root@localhost ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor pres
  et: disabled)
   Active: active (running) since 日 2020-06-21 16:41:33 CST; 7s ago
     Docs: https://docs.docker.com
    Main PID: 20513 (dockerd)
       Tasks: 11
      Memory: 34.0M
     CGroup: /system.slice/docker.service
             └─20513 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/con...

```

建立 Docker 用户

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

docker安装完成后会默认创建 docker 用户组，但该用户组下没有用户。


```
1 # 创建一个docker用户
2 useradd mydocker
3 # 将用户加入到docker组
4 usermod -aG docker mydocker
5 # 给用户设置密码
6 passwd mydocker
7 # 查看是否正确修改组成功
8 id mydocker
9 # 看到类似的输出 uid=1001(mydocker) gid=1001(mydocker) 组
   =1001(mydocker),993(docker)
```

测试 Docker 是否正常工作

```
1 # 切换到mydocker用户
2 su mydocker
3 # 执行测试命令
4 docker run hello-world
```

若能正常输出以下信息信息，则说明安装成功并且说明docker能够正常运行。

```
[mydocker@localhost root]$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c3db9a9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

默认配置下，如果在 CentOS 使用 Docker CE 看到下面的这些警告信息：

```
1 WARNING: bridge-nf-call-iptables is disabled
2 WARNING: bridge-nf-call-ip6tables is disabled
```

请添加内核配置参数以启用这些功能，需要使用root用户

```
1 # 切换回管理员
2 sudo -s
3 # 启动功能命令
4 tee -a /etc/sysctl.conf <<-EOF
5 net.bridge.bridge-nf-call-ip6tables = 1
6 net.bridge.bridge-nf-call-iptables = 1
7 EOF
8 # 重新加载配置
9 sysctl -p
```

```
[root@localhost ~]# tee -a /etc/sysctl.conf <<-EOF
> net.bridge.bridge-nf-call-ip6tables = 1
> net.bridge.bridge-nf-call-iptables = 1
> EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
[root@localhost ~]# sysctl -p
vm.max_map_count = 655360
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
[root@localhost ~]#
```

卸载Docker

如果你想要卸载Docker引擎，可以执行下面的命令

```
1 yum remove docker-ce \
2 docker-ce-cli \
3 containerd.io \
4 docker-compose-plugin
```

主机上的镜像、容器、卷或自定义配置文件不会自动删除。要删除所有镜像、容器和卷，执行下面的命令：

```
1 rm -rf /var/lib/docker
2 rm -rf /var/lib/containerd
```

您必须手动删除任何已编辑的配置文件。

Docker镜像加速器配置

国内从 Docker Hub 拉取镜像有时会遇到困难，此时可以配置镜像加速器。国内很多云服务商都提供了国内加速器服务，例如：

- 网易云加速器 <https://hub-mirror.c.163.com>
- 百度云加速器 <https://mirror.baidubce.com>
- DAOCLOUD <https://www.daocloud.io/mirror>
- 阿里云加速器(需登录账号获取)

配置镜像

在 `/etc/docker/daemon.json` 中写入如下内容（如果文件不存在请新建该文件）

```
1 #如果文件不存在 新建文件
2 touch daemon.json
3 # 编辑文件 vi daemon.json 添加{}下面的内容
4 {
5     "registry-mirrors": [
6         "https://hub-mirror.c.163.com",
7         "https://mirror.baidubce.com",
8         "http://f1361db2.m.daocloud.io",
9         "https://registry.docker-cn.com"
10    ]
11 }
```

注意，一定要保证该文件符合 json 规范，否则 Docker 将不能启动。

重启服务

```
1 # 重新加载配置
2 systemctl daemon-reload
3 # 重启docker服务
4 systemctl restart docker
5 # 查看docker状态
6 systemctl status docker
```

检查加速器是否生效

执行 `docker info`，如果从结果中看到了如下内容，说明配置成功。

```
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
  127.0.0.0/8
Registry Mirrors:
  https://hub-mirror.c.163.com/
  https://mirror.baidubce.com/
  http://f1361db2.m.daocloud.io/
Live Restore Enabled: false
```

阿里云镜像获取方式

容器镜像服务

▼ 默认实例

镜像仓库

命名空间

授权管理

代码源

访问凭证

▶ 企业版实例

▼ 镜像中心

镜像搜索

我的收藏

镜像加速器

镜像加速器

加速器

使用加速器可以提升获取Docker官方镜像的速度

加速器地址

https://[REDACTED].mirror.aliyuncs.com 复制

操作文档

Ubuntu CentOS Mac Windows

1. 安装 / 升级Docker客户端

推荐安装 1.10.0 以上版本的Docker客户端，参考文档 [docker-ce](#)

2. 配置镜像加速器

针对Docker客户端版本大于 1.10.0 的用户

您可以通过修改daemon配置文件 `/etc/docker/daemon.json` 来使用加速器

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

云服务商

某些云服务商提供了仅供内部访问的镜像服务，当您的 Docker 运行在云平台时可以选择它们。

- Azure 中国镜像 <https://dockerhub.azk8s.cn>
- 腾讯云 <https://mirror.ccs.tencentyun.com>

Docker远程访问

开启访问

参考链接: <https://docs.docker.com/config/daemon/remote-access/>

使用下面命令编辑docker服务启动文件

```
1 systemctl edit docker.service
```

添加或修改以下行，替换为您自己的值，ip地址修改成功你自己服务器的ip地址

```
1 [Service]
2 ExecStart=
3 ExecStart=/usr/bin/dockerd -H fd:// -H tcp://192.168.220.128:2375 --
  containerd=/run/containerd/containerd.sock
```

然后保存文件，执行下面命令重新加载systemctl配置。

```
1 systemctl daemon-reload
```

重启Docker容器

```
1 systemctl restart docker
```

通过查看 netstat 的输出以确认 dockerd 正在侦听配置的端口，以检查更改是否已生效。

```
1 netstat -lntp | grep dockerd
```

```
[root@localhost ~]# systemctl edit docker.service
[root@localhost ~]# systemctl daemon-reload
[root@localhost ~]# systemctl restart docker.service
[root@localhost ~]# netstat -lntp | grep dockerd
tcp        0      0 192.168.220.128:2375  0.0.0.0:*          LISTEN      10916/dockerd
[root@localhost ~]#
```

开放防火墙端口

```
1 # 开放端口
2 firewall-cmd --add-port 2375/tcp --permanent
3 # 重新加载防火墙
4 firewall-cmd --reload
```

安全认证

由于开放了端口没有做任何安全保护，会引起安全漏洞，被人入侵、挖矿、CPU飙升这些情况都有发生。

为解决这个问题：我们只要使用安全传输层协议（TLS）进行传输并使用CA认证即可。

参考链接：<https://docs.docker.com/engine/security/protect-access/>

制作证书和秘钥

我们需要使用OpenSSL制作CA机构证书、服务端证书和客户端证书，以下操作均在安装Docker的Linux服务器上进行。

- 首先创建一个目录用于存储生成的证书和秘钥

```
1 mkdir /home/docker-ca && cd /home/docker-ca
```

- 创建CA证书私钥，期间需要输入两次用户名和密码，生成文件为ca-key.pem

```
1 openssl genrsa -aes256 -out ca-key.pem 4096
```

```
[root@localhost docker-ca]# openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
```

密码根据自己情况输入就行，比如我输入都是123456。

- 根据私钥创建CA证书，期间需要输入上一步设置的私钥密码，生成文件为ca.pem

```
1 | openssl req -new -x509 -days 365 -key ca-key.pem \
2 | -sha256 -subj "/CN=*" -out ca.pem
```

```
[root@localhost docker-ca]# openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -subj "/CN=*" -out ca.pem
Enter pass phrase for ca-key.pem:
[root@localhost docker-ca]#
```

- 创建服务端私钥，生成文件为server-key.pem

```
1 | openssl genrsa -out server-key.pem 4096
```

```
[root@localhost docker-ca]# openssl genrsa -out server-key.pem 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
[root@localhost docker-ca]#
```

- 创建服务端证书签名请求文件，用于CA证书给服务端证书签名，生成文件server.csr

```
1 | openssl req -subj "/CN=*" -sha256 -new \
2 | -key server-key.pem -out server.csr
```

```
[root@localhost docker-ca]# openssl req -subj "/CN=*" -sha256 -new -key server-key.pem -out server.csr
[root@localhost docker-ca]#
```

- 创建CA证书签名好的服务端证书，期间需要输入CA证书私钥密码，生成文件为server-cert.pem

```
1 | openssl x509 -req -days 365 -sha256 -in server.csr \
2 | -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem
```

```
[root@localhost docker-ca]# openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem
Signature ok
subject=CN = *
Getting CA Private Key
Enter pass phrase for ca-key.pem:
[root@localhost docker-ca]#
```

- 创建客户端私钥，生成文件为key.pem

```
1 | openssl genrsa -out key.pem 4096
```

```

[root@localhost docker-ca]# openssl genrsa -out key.pem 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.....+++++
e is 65537 (0x010001)
[root@localhost docker-ca]#

```

- 创建客户端证书签名请求文件，用于CA证书给客户证书签名，生成文件client.csr

```
1 openssl req -subj "/CN=client" -new -key key.pem -out client.csr
```

```

[root@localhost docker-ca]# openssl req -subj "/CN=client" -new -key key.pem -out client.csr
[root@localhost docker-ca]#

```

- 为了让秘钥适合客户端认证，创建一个扩展配置文件extfile-client.cnf

```
1 echo extendedKeyUsage = clientAuth > extfile-client.cnf
```

```

[root@localhost docker-ca]# echo extendedKeyUsage = clientAuth > extfile-client.cnf
[root@localhost docker-ca]#

```

- 创建CA证书签名好的客户端证书，期间需要输入CA证书私钥密码，生成文件为cert.pem

```

1 openssl x509 -req -days 365 -sha256 -in client.csr \
2 -CA ca.pem -CAkey ca-key.pem -CAcreateserial \
3 -out cert.pem -extfile extfile-client.cnf

```

```

[root@localhost docker-ca]# openssl x509 -req -days 365 -sha256 -in client.csr \
> -CA ca.pem -CAkey ca-key.pem -CAcreateserial \
> -out cert.pem -extfile extfile-client.cnf
Signature ok
subject=CN = client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
[root@localhost docker-ca]#

```

- 删除创建过程中多余的文件

```
1 rm -rf ca.srl server.csr client.csr extfile-client.cnf
```

- 最终生成文件如下，有了它们我们就可以进行基于TLS的安全访问了

```

[root@localhost docker-ca]# ls
ca-key.pem  ca.pem  cert.pem  key.pem  server-cert.pem  server-key.pem
[root@localhost docker-ca]# ll

```

```

1 ca.pem CA证书
2 ca-key.pem CA证书私钥
3 server-cert.pem 服务端证书
4 server-key.pem 服务端证书私钥
5 cert.pem 客户端证书
6 key.pem 客户端证书私钥

```

配置Docker支持TLS

- 编辑docker服务启动文件

```
1 systemctl edit docker.service
```

- 在原来的基础上往后添加启动参数

```
1 [Service]
2 ExecStart=
3 ExecStart=/usr/bin/dockerd -H fd:// -H tcp://192.168.220.128:2375 --
  containerd=/run/containerd/containerd.sock --tlsverify --tlscacert=/home/docker-
  ca/ca.pem --tlscert=/home/docker-ca/server-cert.pem --tlskey=/home/docker-
  ca/server-key.pem
```

- 然后保存文件，执行下面命令重新加载systemctl配置。

```
1 systemctl daemon-reload
```

- 重启Docker容器


```
1 systemctl restart docker
```


客户端访问


下载证书到本地磁盘，主要下载下面三个文件

```
1 ca.pem CA证书
2 cert.pem 客户端证书
3 key.pem 客户端证书私钥
```

比如我下载到我的本地电脑的E:\Workspace\docker-ca这个位置。

 ca.pem

 cert.pem

 key.pem

记住这个位置，后面打包部署的时候需要指定改目录的，具体使用我们在打包部署的时候会用到。

Docker命令

进程相关命令

```
1 # 启动Docker命令
2 systemctl start docker
3 # 停止Docker命令
4 systemctl stop docker
5 # 重启Docker命令
6 systemctl restart docker
7 # 查看启动结果
8 systemctl status docker
9 # 设置开机启动Docker
10 systemctl enable docker
```

镜像相关命令

我们可以直接输入 docker 命令来查看到 Docker 客户端的所有命令选项

可以通过命令 **docker command --help** 更深入的了解指定的 Docker 命令使用方法。

例如我们要查看 **docker stats** 指令的具体使用方法：

```
1 docker stats --help
```

查看镜像

查看本地所有的镜像

```
1 # 列出本地主机上的镜像
2 docker images
3 # 查看所用镜像的id
4 docker images -q
```

搜索镜像

从网络中查找需要的镜像

```
1 docker search 镜像名称
```

拉取镜像

从Docker仓库下载镜像到本地，镜像名称格式为 名称:版本号，如果版本号不指定则是最新的版本。

如果不知道镜像版本，可以去 [Docker Hub](#)搜索对应镜像查看。

```
1 docker pull 镜像名称:版本号
2 # 完成命令 docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]
```

具体的选项可以通过 `docker pull --help` 命令看到，这里我们说一下镜像名称的格式。

Docker 镜像仓库地址：地址的格式一般是 `<域名/IP>[:端口号]`。默认地址是 Docker Hub。

仓库名：如之前所说，这里的仓库名是两段式名称，即 `<用户名>/<软件名>`。对于 Docker Hub，如果不给出用户名，则默认为 `library`，也就是官方镜像。

如：

```
1 docker pull centos:7
```

上面的命令中没有给出 Docker 镜像仓库地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 `centos:7`，因此将会获取官方镜像 `library/centos` 仓库中标签为 `7` 的镜像。

从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的 ID 的前 12 位。并且下载结束后，给出该镜像完整的 sha256 的摘要，以确保下载一致性。

删除镜像

删除本地镜像

```
1 # 删除指定本地镜像
2 docker rmi 镜像id #这里也可以是<镜像名:tag>
3 # 删除所有本地镜像
4 docker rmi `docker images -q`
```

容器相关命令

创建容器

```
1 docker run [参数] 镜像名称:Tag <交互方式>
```

参数说明：

- `-i`：保持容器运行。通常与 `-t` 同时使用。加入 `it` 这两个参数后，容器创建后自动进入容器中，退出容器后，容器自动关闭。
- `-t`：为容器重新分配一个伪输入终端，通常与 `-i` 同时使用。
- `-d`：以守护（后台）模式运行容器。创建一个容器在后台运行，需要使用 `docker exec` 进入容器。退出后，容器不会关闭。
- `-it` 创建的容器一般称为交互式容器，`-id` 创建的容器一般称为守护式容器
- `--name`：为创建的容器命名。
- `/bin/bash`:交互方式为shell终端

命令示例：

```
1 # 创建容器c1并进入
2 docker run -it --name c1 centos:7 /bin/bash
3 # 创建容器并后台启动
4 docker run -id --name c2 centos:7
```

查看容器

```
1 # 查看正在运行的容器
2 docker ps
3 # 查看所有容器
4 docker ps -a
5 # 查看所有容器id
6 docker ps -aq
```

进入容器

```
1 docker exec [参数] 容器名称 /bin/bash
```

命令示例：

```
1 # 进入c2容器
2 docker exec -it c2 /bin/bash
```

启动容器

```
1 docker start 容器名称
```

命令示例：

```
1 # 启动c1容器
2 docker start c1
```

停止容器

```
1 docker stop 容器名称
```

命令示例：

```
1 # 停止c1容器
2 docker stop c1
```

删除容器

如果容器是运行状态则删除失败，需要停止容器才能删除

```
1 # 删除指定容器
2 docker rm 容器名称
3 # 删除所用容器
4 docker rm `docker ps -aq`
```

查看容器信息

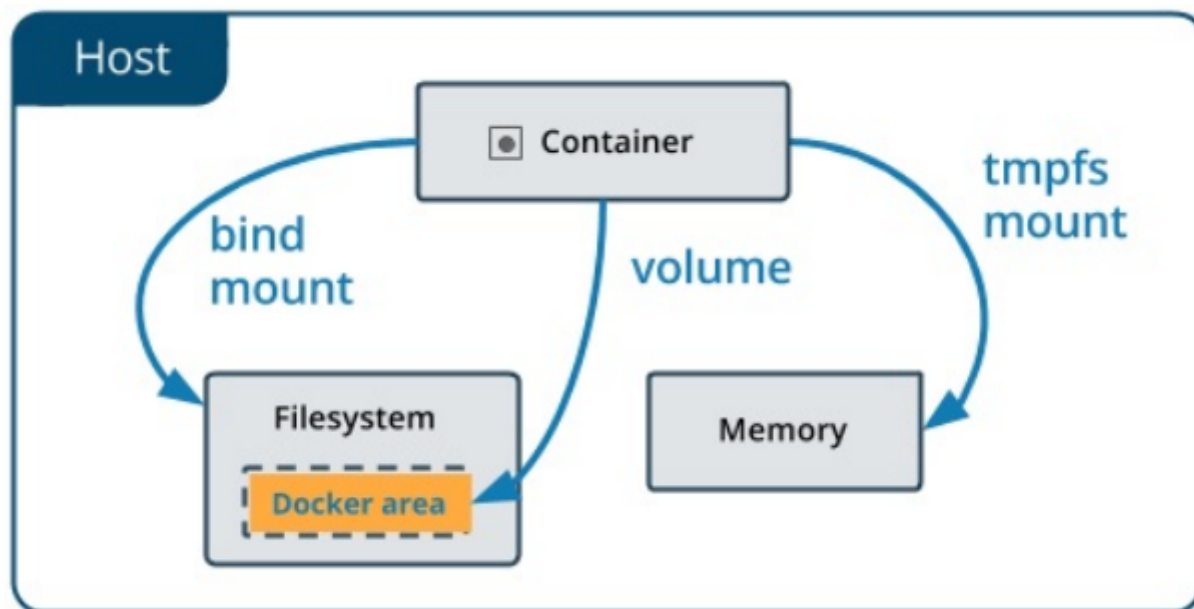
```
1 docker inspect 容器名称
```

更新容器配置

指令: `docker update [OPTIONS] CONTAINER [CONTAINER ...]`

```
1 # 演示设置容器跟随docker一起重启
2 docker update --restart always 容器名称
```

Docker 数据管理



数据卷

数据卷的概念

- 数据卷是宿主机中的一个目录或文件
- 当容器目录和数据卷目录绑定后，对方的修改会立即同步
- 一个数据卷可以被多个容器同时挂载
- 一个容器也可以被挂载多个数据卷
- 数据卷，默认会一直存在，即使容器被删除

数据卷的作用

- 容器数据持久化
- 外部机器和容器间接通信
- 容器之间数据交换

配置数据卷

创建启动容器时，使用 `-v` 参数 设置数据卷

```
1 docker run ... -v 宿主机目录(文件):容器内目录(文件) ... 镜像名:tag [交互方式]
```

注意事项:

- 目录必须是绝对路径
- 如果目录不存在，会自动创建
- 可以挂载多个数据卷

命令示例

```
1 #挂载单目录
2 docker run -it --name c1 -v /root/volume:/root/volum_container centos:7
3 #挂载多目录
4 docker run -it --name c2 -v /root/data:/root/data -v /root/data1:/root/data1
   centos:7
```

数据卷容器

多容器进行数据交换

- 多个容器挂载同一个数据卷(弊端: 如果容器多了, 创建容器的时候构建数据卷比较麻烦)
- 数据卷容器

数据卷容器实现步骤

创建数据卷容器, 使用 `-v` 指定数据卷

```
1 docker run -id --name 容器名称 -v 宿主机目录路径 镜像名:Tag
```

为容器设置数据卷容器, 使用 `--volumes-from` 设置数据卷容器

```
1 docker run -id --name 容器名称 --volumes-from 数据卷容器名 镜像名:Tag
```