Docker 学习笔记

背景

环境配置的难题

软件开发最大的麻烦事之一，就是环境配置。用户计算机的环境都不相同，你怎么知道自家的软件，能在那些机器跑起来？

用户必须保证两件事：操作系统的设置，各种库和组件的安装。只有它们都正确，软件才能运行。举例来说，安装一个 Python 应用，计算机必须有 Python 引擎，还必须有各种依赖，可能还要配置环境变量。

如果某些老旧的模块与当前环境不兼容，那就麻烦了。开发者常常会说："它在我的机器可以跑了"（It works on my machine），言下之意就是，其他机器很可能跑不了。

环境配置如此麻烦，换一台机器，就要重来一次，旷日费时。很多人想到，能不能从根本上解决问题，**软件可以带环境安装？**也就是说，**安装的时候，把原始环境一模一样地复制过来**。

虚拟机

虚拟机（virtual machine）就是带环境安装的一种解决方案。它可以在一种操作系统里面运行另一种操作系统，比如在 Windows 系统里面运行 Linux 系统。应用程序对此毫无感知，因为虚拟机看上去跟真实系统一模一样，而对于底层系统来说，虚拟机就是一个普通文件，不需要了就删掉，对其他部分毫无影响。

缺点。

（1）资源占用多

虚拟机会独占一部分内存和硬盘空间。它运行的时候，其他程序就不能使用这些资源了。哪怕虚拟机里面的应用程序，真正使用的内存只有 1MB，虚拟机依然需要几百 MB 的内存才能运行。

（2）冗余步骤多

虚拟机是完整的操作系统，一些系统级别的操作步骤，往往无法跳过，比如用户登录。

（3）启动慢

启动操作系统需要多久，启动虚拟机就需要多久。可能要等几分钟，应用程序才能真正运行。

Linux 容器

由于虚拟机存在这些缺点，Linux 发展出了另一种虚拟化技术：Linux 容器（Linux Containers，缩写为 LXC）。

Linux 容器不是模拟一个完整的操作系统，而是对进程进行隔离。或者说，**在正常进程的外面套了一个保护层**。对于容器里面的进程来说，它接触到的各种资源都是虚拟的，从而实现与底层系统的隔

离。

由于容器是进程级别的，相比虚拟机有很多优势。

（1）启动快

容器里面的应用，直接就是底层系统的一个进程，而不是虚拟机内部的进程。所以，启动容器相当于启动本机的一个进程，而不是启动一个操作系统，速度就快很多。

（2）资源占用少

容器只占用需要的资源，不占用那些没有用到的资源；虚拟机由于是完整的操作系统，不可避免要占用所有资源。另外，多个容器可以共享资源，虚拟机都是独享资源。

（3）体积小

容器只要包含用到的组件即可，而虚拟机是整个操作系统的打包，所以容器文件比虚拟机文件要小很多。

Docker 是什么

Docker 属于 Linux 容器的一种封装，提供简单易用的容器使用接口。它是目前最流行的 Linux 容器解决方案。

Docker 将应用程序与该程序的依赖，打包在一个文件里面。运行这个文件，就会生成一个虚拟容器。程序在这个虚拟容器里运行，就好像在真实的物理机上运行一样。有了 Docker，就不用担心环境问题。

image 文件

Docker 把应用程序及其依赖，打包在 image 文件里面。只有通过这个文件，才能生成 Docker 容器。image 文件可以看作是容器的模板。Docker 根据 image 文件生成容器的实例。同一个 image 文件，可以生成多个同时运行的容器实例。

```
# 列出本机的所有 image 文件。
$ docker image ls

# 删除 image 文件
$ docker image rm [imageName]
```

运行这个 image 文件。

```
$ docker container run hello-world
```

docker container run 命令会从 image 文件，生成一个正在运行的容器实例。

Docker 的存储驱动-overlay2

命令行学习

```
docker search kafka
docker search zookeeper
docker search kafka-manager
```

《深入浅出 Docker》

Nigel・Poulton

第一章-容器发展之路

落后的旧时代

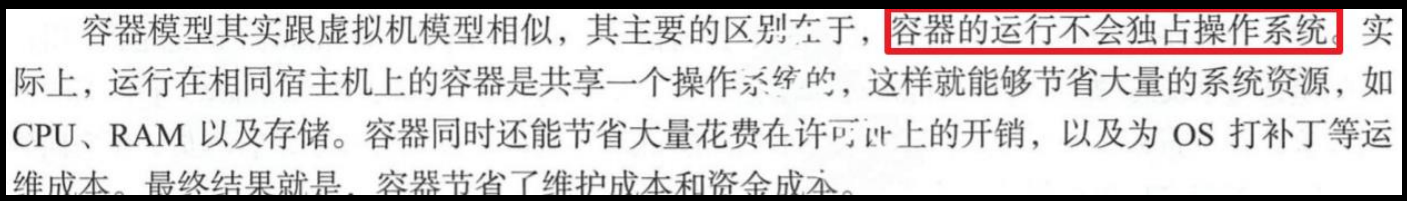
服务器部署
不稳定
利用率低
各种问题

Vmware

虚拟机的出现

VM 最大的缺点就是：依赖其专用的 OS，OS 会暂用额外的 CPU、RAM 等资源。
启动慢
移植性差

容器


容器模型其实跟虚拟机模型相似，其主要的区别在于，容器的运行不会独占操作系统。实际上，运行在相同宿主机上的容器是共享一个操作系统的，这样就能够节省大量的系统资源，如 CPU、RAM 以及存储。容器同时还能节省大量花费在许可证上的开销，以及为 OS 打补丁等运维成本。最终结果就是，容器节省了维护成本和资金成本。

Linux 容器

容器技术起源于 Linux
随容器发展影响大的技术：
1. 内核命名技术 Kernel Namespace
2. 控制组 Control Group
3. 联合文件系统 Union File System

4. Docker：Docker 让容器变得更加简单

Windows 容器

Windows 和 docker 合作，开发 Windows 容器

运行中的容器共享宿主机的内核

因此，Windows 容器需要运行在 Windows 宿主机上，Linux 容器需要运行在 Linux 宿主机上，不能迁移

Mac 容器

没有 Mac 容器
但是有 Docker for Mac 来运行 Linux 容器，这是通过在 Mac 上启动一个轻量级的 Linux VM

Kubernetes

Google 的一个开源项目，开源后迅速成为容器编排的领头羊
K8s 目前已经采用 Docker 作为其默认的容器运行时 container RUNTIME，不过也支持其他的

第二章-Docker

Docker?
- Docker 公司
- Docker 的容器运行时 RUNTIME 和编排引擎
- Docker 开源项目：Moby

Docker 简介

Docker：一种用于创建、管理、编排容器的软件技术，是 Moby 开源项目的一部分，Docker 是 Moby 的维护者

Docker 公司

位于美国旧金山
起初名为 dotCloud，PaaS 公司

13 年聘请 Ben Golub 为新的 CEO，公司重命名为 Docker，放弃 PaaS 平台，开启新的征程，“将 Docker 和容器技术推向全世界”

Docker 一词源于英国口语，以为码头工人：Dock Worker

Docker RUNTIME 和编排引擎

Docker 引擎是用于运行和编排容器的基础设施工具，类似 VMware 的核心管理程序 ESXi

Docker 引擎开源，有企业版 EE 和社区版 CE

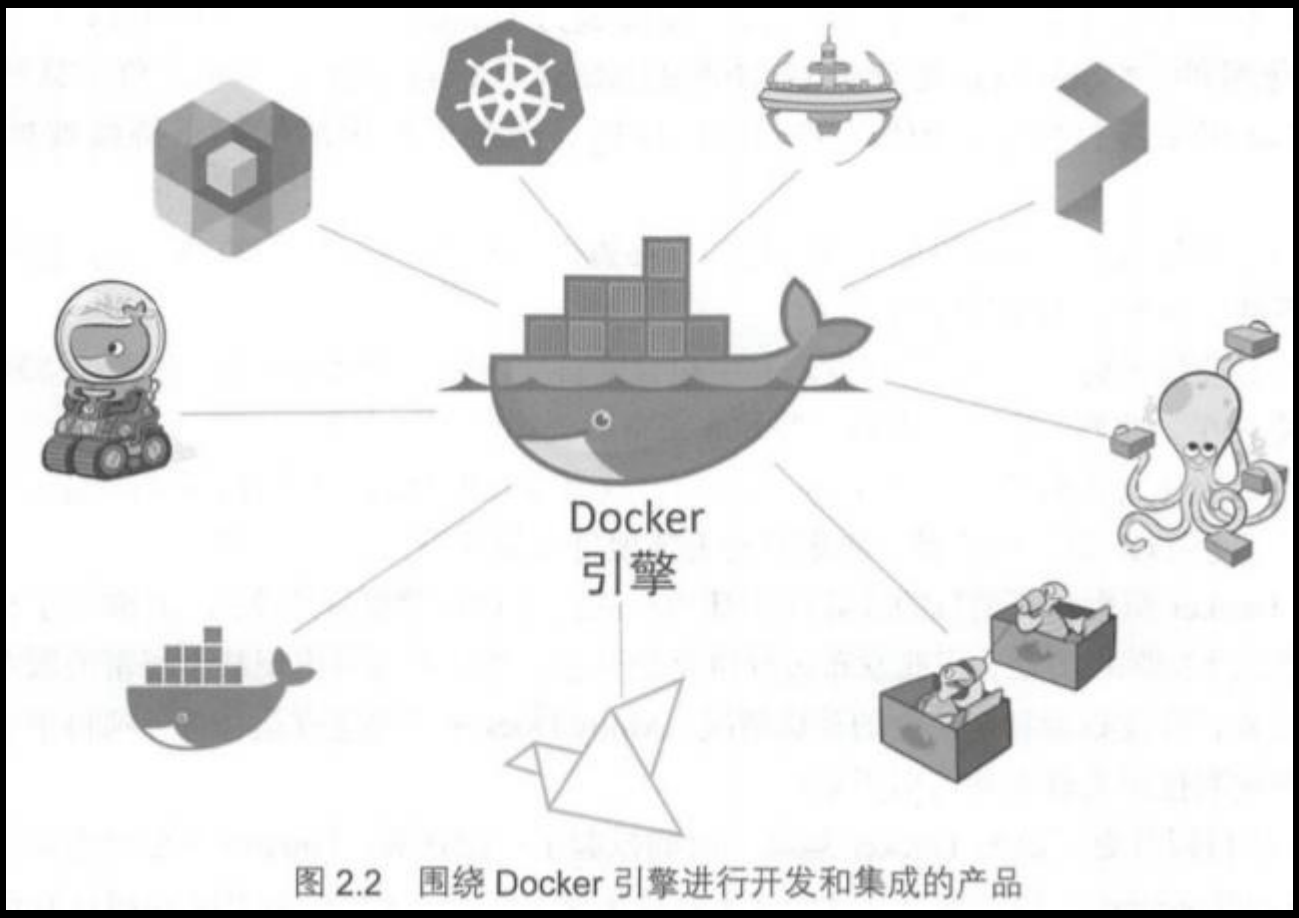


图 2.2　围绕 Docker 引擎进行开发和集成的产品

Docker 开源项目

https://github.com/moby/moby

该项目将 docker 拆分为更多的模块化组件

Docker 引擎位于 moby/moby 下

容器生态

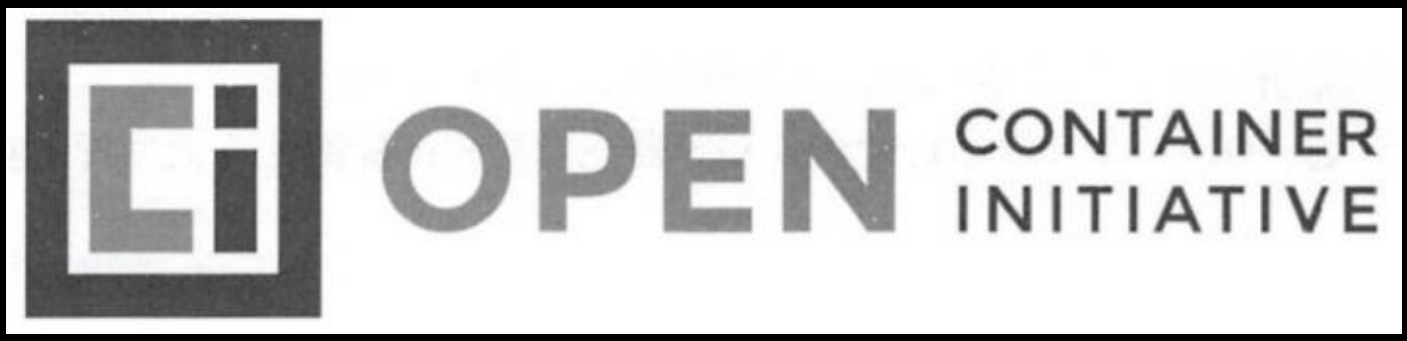
Docker 内置的组件都可以替换为第三方组件——可插拔

良性的竞争是创新之母

开放容器计划

The Open Container Initiative

OCI



OCI 是一个旨在对容器基础架构中的基础组件进行标准化的管理委员会

第三章-docker 安装

- 桌面安装。
  - Windows 版 Docker（Docker for Windows）。
  - Mac 版 Docker（Docker for Mac）。
- 服务器安装。
  - Linux。
  - Windows Server 2016。
- Docker 引擎升级。
- Docker 存储驱动的选择。

Linux 安装

（2）使用 wget 从 https://get.docker.com 获取并运行 Docker 安装脚本，然后采用 Shell 中管道（pipe）的方式来执行这个脚本。

```
$ wget -qO- https://get.docker.com/ | sh
```

（3）最好通过非 root 用户来使用 Docker。这时需要添加非 root 用户到本地 Docker Unix 组当中。下面的命令展示了如何把名为 npoulton 的用户添加到 Docker 组中，以及如何确认操作是否执行成功。请读者自行使用系统中的有效用户。

```
$ sudo usermod -aG docker npoulton

$ cat /etc/group | grep docker
docker:x:999:npoulton
```

如果读者当前登录用户就是要添加到 Docker 组中的用户的话，则需要重新登录，组权限设置才会生效。

Docker 存储驱动的选择

每个 Docker 都有一个本地存储空间，用于保存层叠的镜像层（Image Layer）以及挂载的容器文件系统。

默认情况下，容器的所有读写操作都是发生在其镜像层上或者挂载的文件系统中

以往，本地存储是通过存储驱动 Storage Driver 进行管理的，有时候也被称为 Graph Driver。虽然存储驱动在上层抽象设计中都采用了**栈式镜像层存储**和**写时复制 Copy On Write** 的设计思想，但是 Docker 在 Linux 底层支持几种不同的存储驱动的具体实现，每一种都采用不同的镜像层和 COW。

Linux 上可选的存储驱动：
1. AUFS 原始的
2. Overlay2：可能未来最佳
3. Device Mapper
4. Btrfs
5. ZFS

在 windows 上只支持 Windows Filter

每个 docker 只能选择一个存储驱动

可以修改/etc/docker/daemon.json 更换

```
{
  "storage-driver": "overlay2"
}
```

注意：修改存储驱动后现有的镜像将不可用

查看当前存储驱动：
Docker system info

选择建议：

- Red Hat Enterprise Linux：4.x 版本内核或更高版本 + Docker 17.06 版本或更高版本，建议使用 Overlay2。
- Red Hat Enterprise Linux：低版本内核或低版本的 Docker，建议使用 Device Mapper。
- Ubuntu Linux：4.x 版本内核或更高版本，建议使用 Overlay2。
- Ubuntu Linux：更早的版本建议使用 AUFS。
- SUSE Linux Enterprise Server：Btrfs。

第四章-纵观 Docker

运维视觉

两个组件：
1. Docker client
2. Docker server：deamon 引擎

使用 Linux 默认安装时，client 和 deamon 之间通过**本地 IPC/UNIX Socket** 通信 (/var/run/docker.sock)

```
> docker version
Client:
 Version:      18.01.0-ce
 API version:  1.35
 Go version:   go1.9.2
 Git commit:   03596f5
 Built: Wed Jan 10 20:11:05 2018
 OS/Arch:      linux/amd64
 Experimental: false
 Orchestrator: swarm

Server:
 Engine:
  Version:      18.01.0-ce
  API version:  1.35 (minimum version 1.12)
  Go version:   go1.9.2
  Git commit:   03596f5
  Built:        Wed Jan 10 20:09:37 2018
  OS/Arch:      linux/amd64
  Experimental: false
```

镜像

可以将对象理解为包含了 OS 文件系统和应用的对象

Docker image ls

获取镜像：pull 拉取

Docker pull

每个镜像都有自己唯一的 ID

```
ubuntu@VM-16-11-ubuntu:~$ docker image ls
REPOSITORY            TAG       IMAGE ID       CREATED         SIZE
minio/minio           latest    305cf2390a4e   10 days ago     184MB
zookeeper             latest    0839c2287ea0   3 weeks ago     269MB
hello-world           latest    d1165f221234   8 weeks ago     13.3kB
wurstmeister/kafka    latest    cc59b78d943f   4 months ago    438MB
```

容器

运行镜像，启动容器

Docker container run （简写为 docker run）

Run 名称告诉 deamon 启动新的容器

-it 参数表示：进入到容器内部，并连接到当前 shell

容器内部，**ps -elf** 可以查看运行的全部进程

退出交互

1. **Ctrl+P+Q：退出容器**
2. **Exec：退出容器并关闭容器**

Docker exec 也可以进入容器内部

开发视觉

容器即应用

通过 Dockerfile 构建镜像：

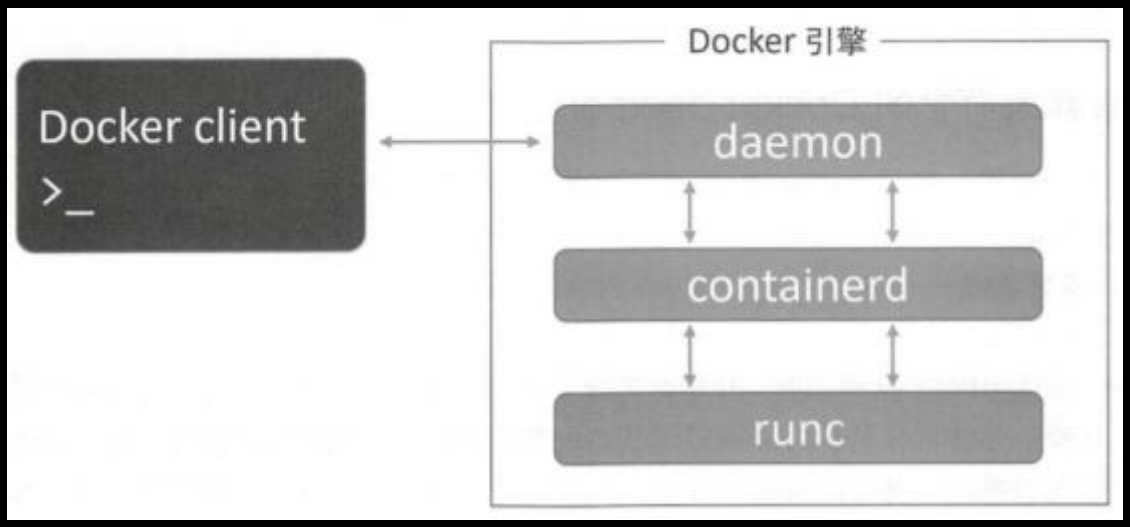
```
FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN  npm install
EXPOSE  8080
ENTRYPOINT ["node", "./app.js"]
```

在 dockerfile 目录下：
Docker image build -t image-name:tag

第五章-Docker 引擎

简介

Docker 引擎是用来运行并管理容器的核心软件
Docker 引擎采用模块化开发，模块：
1. Docker client
2. Docker 守护进程 deamon
3. Containerd
4. Runc



详解

一开始 deamon 是整体，但是这不好，于是拆分为多个模块

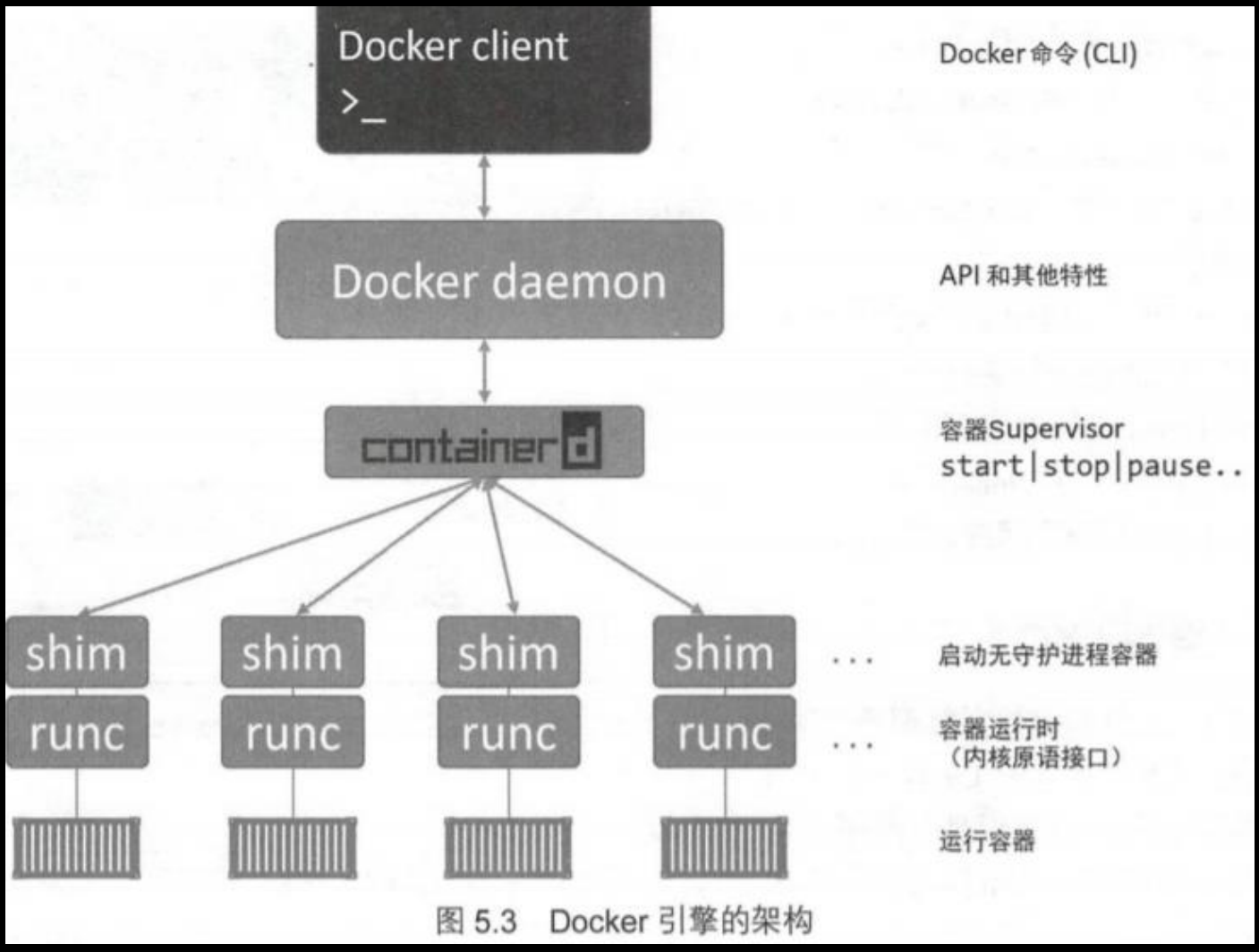


图 5.3　Docker 引擎的架构

OCI 的规范

17 年发布 OCI 的规范。

Deamon 不在包含任何容器运行时代码

所有的容器运行代码都在一个单独的 OCI 兼容层中

Docker 默认采用 runc 实现，runc 是 OCI 容器 RUNTIME 规范的参考实现
Containerd 组件确保了 docker 镜像能够以正确的 OCI Bundle 的格式传递给 runc

Runc

Runc 实质是一个轻量级的、针对 Libcontainer 进行包装的命令行交互工具
Runc 只有一个作用：创建容器
不过它是一个 CLI 包装器，实质就是一个独立的容器运行时工具

Containerd

包含容器的执行逻辑，主要任务是容**器的生命周期管理**

Containerd 以 deamon 的方式运行
Containerd 位于 deamon 和 runc 所在的 OCI 层之间

K8s 可以通过 cri-containerd 使用 containerd

Containerd 本专著与容器声明周期管理，后面也可以进行镜像管理

使用 docker run 执行命令时，docker client 将会把命令转化为合适 API，发送到 deamon
API 是 deamon 实现的，RESTful
Deamon 接收到命令后，将会向 container 发出调用，deamon 本身不包含任何容器代码
Deamon 通过 gRPC 和 containerd 通信。
Containerd 并不负责创建容器，而是指挥 runc 去做
Containerd 将 image 转化为 OCI bundle，并让 runc 基于此创建一个新的容器
Runc 和 OS 接口通信，基于必要的工具(Namespace、CGroup 等)创建容器
容器进程作为 runc 的子进程启动，启动完毕后，runc 退出

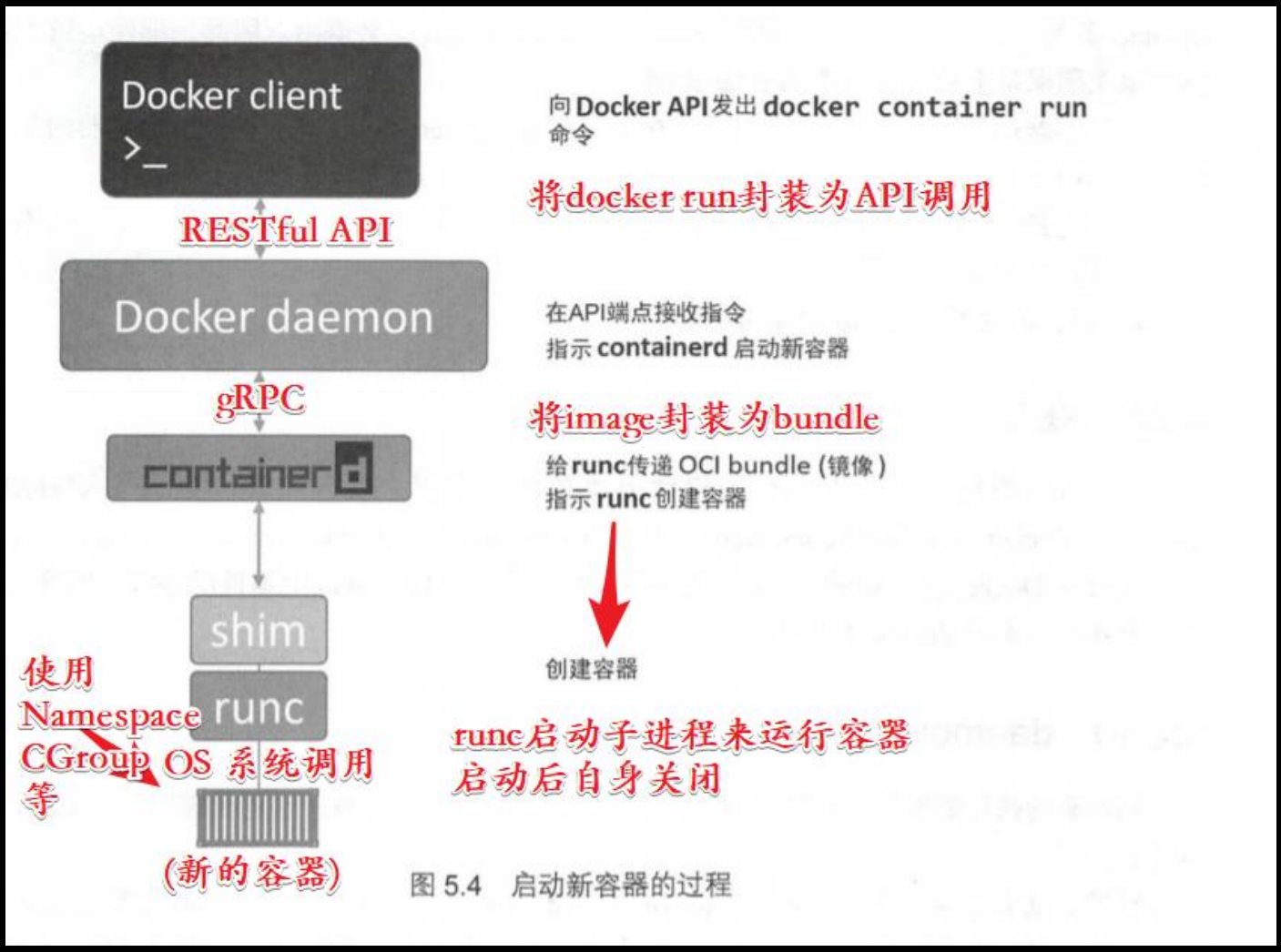


图 5.4 启动新容器的过程

Shim:

实现无 deamon 的容器不可或缺的工具

一旦容器的父进程 runc 退出，相关联的 **containerd-shim 进程就会称为容器的父进程**

Shim 职责:

1. 保持所有的 STDIN\STDOUT 流是开启状态，从而当 deamon 重启的时候，容器不会因为管道 pipe 的关闭而终止
2. 将容器的退出状态反馈给 deamon

在 Linux 系统中，前面谈到的组件由单独的二进制来实现，具体包括 dockerd(Docker daemon)、docker-containerd(containerd)、docker-containerd-shim (shim)和 docker-runc (runc)。

Deamon 的功能

其他功能都剥离出来了

Deamon 还剩:

1. 镜像管理
2. 镜像构建
3. RESTAPI
4. 身份验证
5. 安全
6. 核心网络

7. 编排

第六章-Docker 镜像

简介

Docker 镜像就像是停止运行的容器
镜像由多个层组成，每层叠加之后，从外部看来就如一个独立的对象，**镜像内部是一个精简的 OS，同时还包含应用运行所需要的文件和依赖包。**

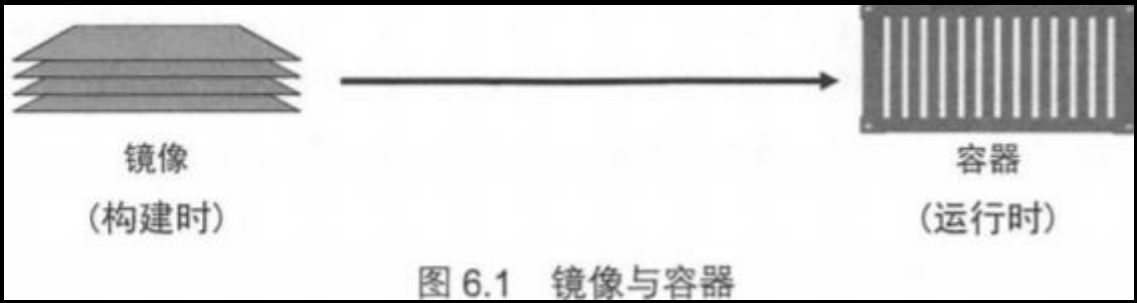


图 6.1 镜像与容器

镜像一旦被启动生成容器，二者就被关联了起来，容器不停，镜像不能删

镜像特点：比较小
容器追求小而快，构建镜像时需要把不需要的都裁减掉
例如：docker 镜像通常不会包含 6 个不用的 shell 让用户选择，通常有一个或者没有
镜像不包含内核，容器共享宿主机的内核

注意：Hyper-V 容器运行在专用的轻量级 VM 上，同时利用 VM 内部的 OS 内核(而不是宿主机的)

拉取镜像

Linux 下镜像仓库：/var/lib/docker/<store-driver>

Win 的镜像比 Linux 大

镜像命名和 tag

镜像名:标签 tag 定义了唯一的一个镜像
Docker image pull mongo -- 采用默认标签 latest

没有指定 tag 默认采用 latest

注意：标为 latest 的镜像不一定是最新的镜像

从非官方仓库拉取镜像时，需要指定镜像的用户名/组织名

```
docker image pull nigelpoulton/tu-demo:v2
```

如果读者希望从第三方镜像仓库服务获取镜像（非 Docker Hub），则需要在镜像仓库名称前加上第三方镜像仓库服务的 DNS 名称。假设上面的示例中的镜像位于 Google 容器镜像仓库服务（GCR）中，则需要在仓库名称前面加上 gcr.io，如 docker pull gcr.io/nigelpoulton/tu-demo:v2（这个仓库和镜像并不存在）。

Docker image pull -a imageName 表示拉取所有 tag 的镜像

Image 可以拥有多个 tag

```
$ docker image ls
REPOSITORY                TAG      IMAGE ID       CREATED      SIZE
nigelpoulton/tu-demo      v2       6ac21e..bead   1 yr ago     211.6 MB
nigelpoulton/tu-demo      latest   9b915a..1e29   1 yr ago     211.6 MB
nigelpoulton/tu-demo      v1       9b915a..1e29   1 yr ago     211.6 MB
```

Latest 一般指向最新的那个 image，那个 image 通常有自己的版本号

Dangling 镜像

```
$ docker image ls --filter dangling=true
REPOSITORY    TAG        IMAGE ID       CREATED      SIZE
<none>        <none>     4fd34165afe0   7 days ago   14.5MB
```

没有标签的镜像——悬虚镜像(dangling 镜像)

构建镜像时，如果 tag 存在，原来的镜像的 tag 被标记为<none>

Docker image prune 移除所有 dangling 镜像
Docker image prune -a 还会移除没有被使用的镜像

Docker 目前支持如下的过滤器。

- dangling：可以指定 true 或者 false，仅返回悬虚镜像（true），或者非悬虚镜像（false）。
- before：需要镜像名称或者 ID 作为参数，返回在之前被创建的全部镜像。
- since：与 before 类似，不过返回的是指定镜像之后创建的全部镜像。
- label：根据标注（label）的名称或者值，对镜像进行过滤。docker image ls 命令输出中不显示标注内容。

其他的过滤方式可以使用 reference。

下面就是使用 reference 完成过滤并且仅显示标签为 latest 的示例。

```
$ docker image ls --filter=reference="*:latest"
REPOSITORY    TAG       IMAGE ID        CREATED       SIZE
alpine        latest    3fd9065eaf02    8 days ago    4.15MB
test          latest    8426e7efb777    3 days ago    122MB
```

读者也可以使用--format 参数来通过 Go 模板对输出内容进行格式化。例如，下面的指令返回 Docker 主机上镜像的大小属性。

```
$ docker image ls --format "{{.Size}}"
99.3MB
111MB
82.6MB
88.8MB
```

```
$ docker image ls --format "{{.Repository}}: {{.Tag}}: {{.Size}}"
```

CLI 方式搜索镜像

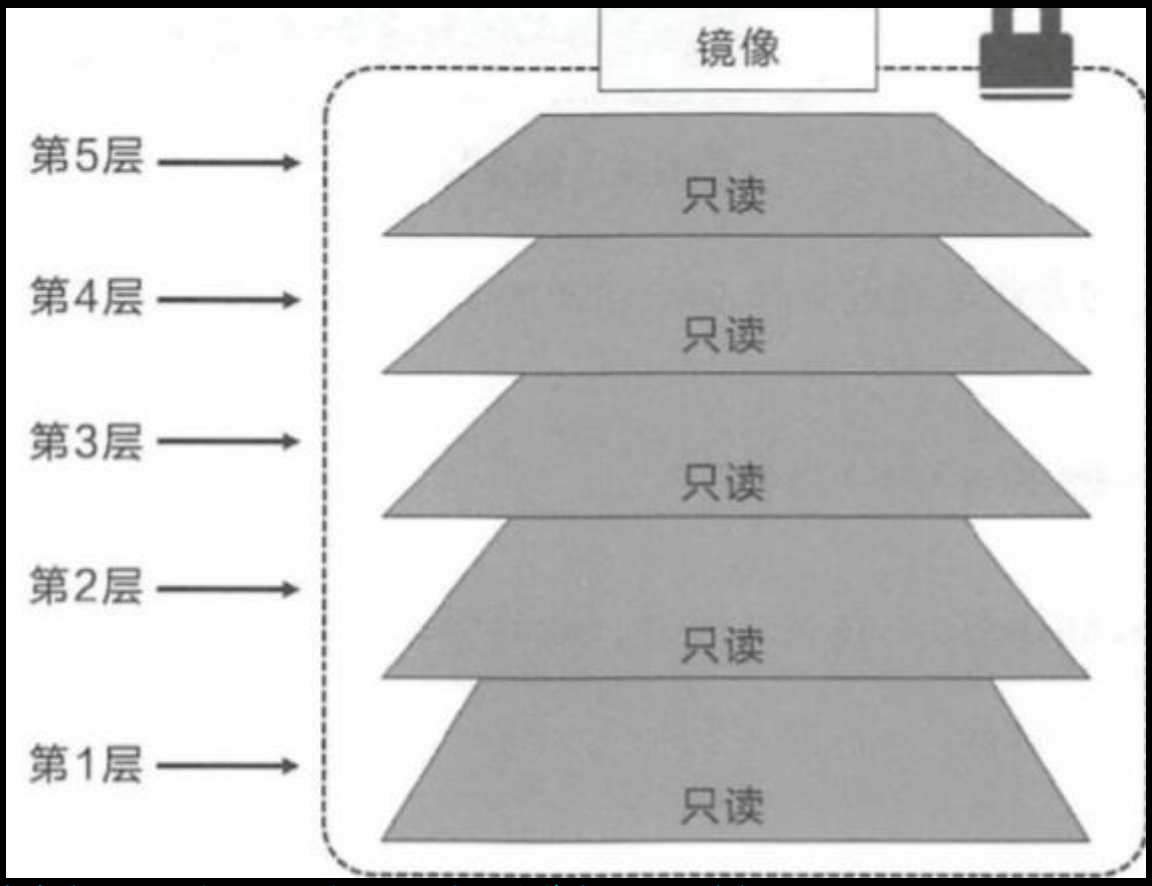
Docker search 允许通过 CLI 方式搜索 Docker Hub

需要注意，上面返回的镜像中既有官方的也有非官方的。读者可以使用--filter "is-official=true"，使命令返回内容只显示官方镜像。

```
$ docker search alpine --filter "is-official=true"
```

镜像和分层

Docker image 由一些**松耦合的只读镜像层**组成：



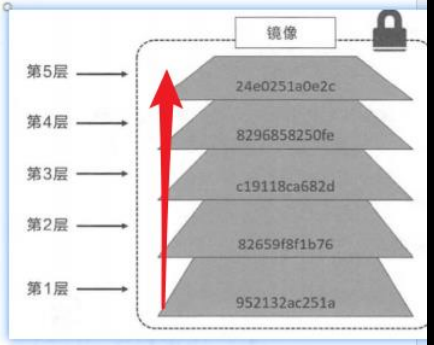
Docker 负责堆叠这些镜像层，并把他们表示为单个统一的对象

查看镜像层：

1. 方式 1：pull 的时候查看

再回顾一下 docker image pull ubuntu:latest 命令的输出内容。

```
$ docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
952132ac251a: Pull  complete
82659f8f1b76: Pull  complete
c19118ca682d: Pull  complete
8296858250fe: Pull  complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
```



在上面输出内容中，以 Pull complete 结尾的每一行都代表了镜像中某个被拉取的镜像层。可以看到，这个镜像包含 5 个镜像层。

2. 方式 2：**docker image inspect**

```
$ docker image inspect ubuntu:latest
[
    {
        "Id": "sha256:bd3d4369ae.......fa2645f5699037d7d8c6b415a10",
        "RepoTags": [
            "ubuntu:latest"

        <Snip>

        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:c8a75145fc...894129005e461a43875a094b93412",
                "sha256:c6f2b330b6...7214ed6aac305dd03f70b95cdc610",
                "sha256:055757a193...3a9565d78962c7f368d5ac5984998",
                "sha256:4837348061...12695f548406ea77feb5074e195e3",
                "sha256:0cad5e07ba...4bae4cfc66b376265e16c32a0aae9"
            ]
```

Layers 中是 **SHA256 散列值**标识的镜像层

Docker history image: 显示镜像的构建历史

所有的 image 都起始于一个基础镜像层，新的层加在其基础上

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 中对应的文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。

共享镜像层

多个镜像之间共享镜像层，可以有效节省空间

每个存储引擎都有自己的镜像分层、镜像共享、COW 实现

镜像摘要(镜像散列值)

如果一个镜像有 bug 但是已经被使用了，如何修复 bug 后更新这个镜像，在不修改 tag 的条件下需要使用镜像摘要 image digest(一种散列值)

每一个镜像都有一个基于其内容的散列值——摘要

```
$ docker image ls --digests alpine
REPOSITORY  TAG     DIGEST                IMAGE ID      CREATED       SIZE
alpine      latest  sha256:3dcd...f73a    4e38e38c8ce0  10 weeks ago  4.8 MB
```

```
sha256:3dcdb92d7432d56604d... 6d99b889d0626de158f73a
```

镜像是一系列**松耦合的独立层的集合**

镜像层才是实际数据存储的地方

镜像、镜像层都有一个散列——**基于内容的内容散列**

第七章-Docker 容器

容器是镜像的运行时实例

容器 vs.VM

VM 和容器最大的区别就是容器更加快和轻量级：**与 VM 需要运行在完整的 OS 上相比，容器会共享所在主机的 OS 内核**

容器和 VM 都依赖于宿主机

1. 在 VM 模型中：
   宿主机将资源分配给 VM
   在 VM 中可以安装应用

图 7.2　运行 4 个业务应用的物理服务器

2．在容器模型中：

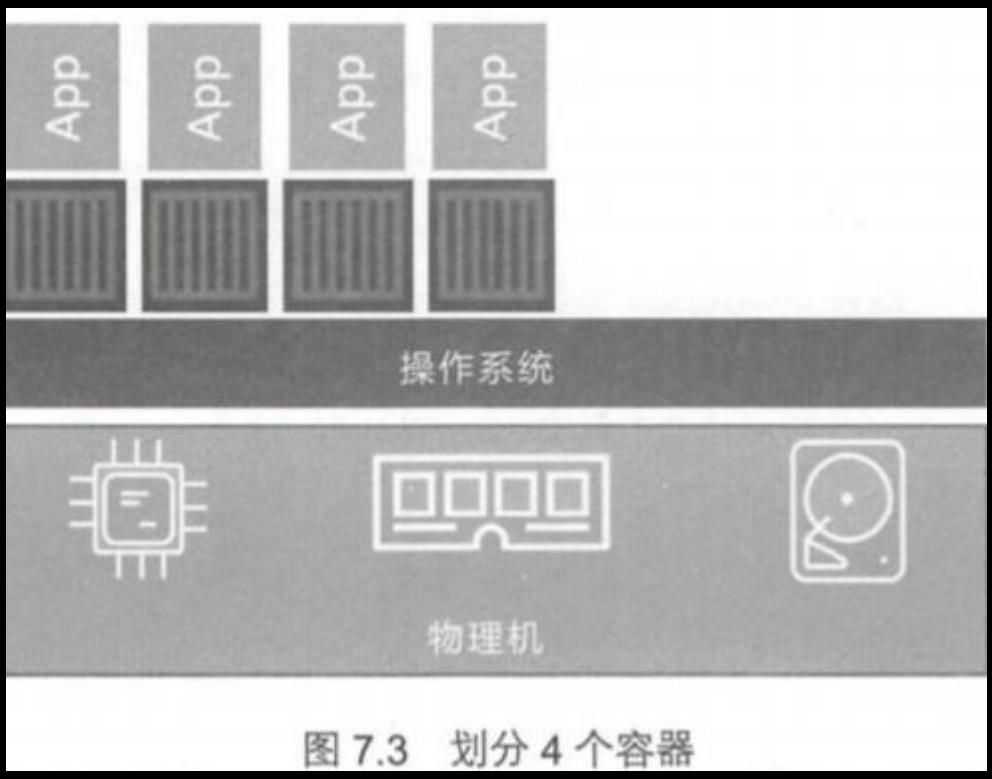


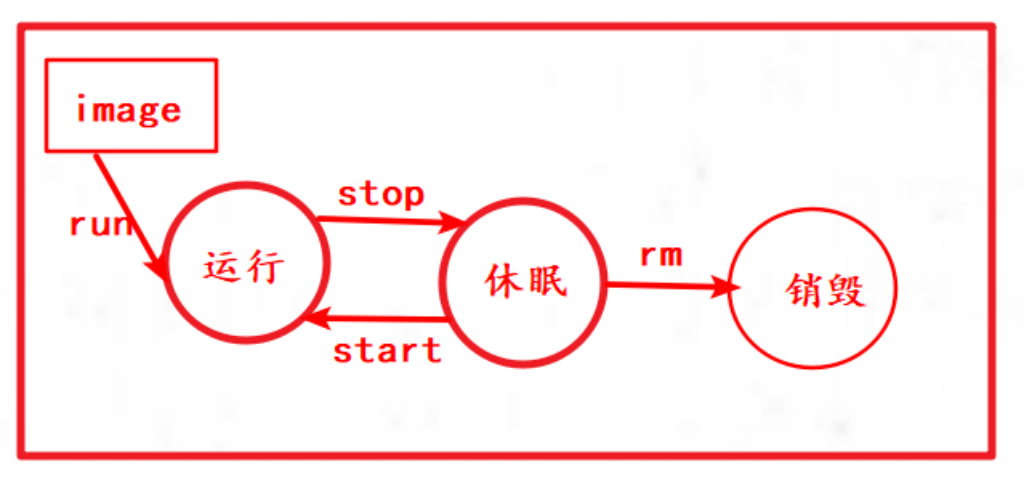
图 7.3　划分 4 个容器

从更高层面上来讲，Hypervisor 是硬件虚拟化（Hardware Virtualization）——Hypervisor 将硬件物理资源划分为虚拟资源；另外，容器是操作系统虚拟化（OS Virtualization）——容器将系统资源划分为虚拟资源。

容器的生命周期

容器在持久化上有缺点，不善于持久化

容器的生命周期：**创建-运行-休眠-销毁**



创建后，在销毁前，在容器中创建的文件是可以持久化的。
但是**卷(Volume)才是容器中存储持久化数据的首选方式**
容器被 rm 后，其中的数据会被删除，如果使用了 Volume，数据还是会被保存下来。

优雅地关闭容器

推荐先 stop 容器，在 rm 容器。不推荐直接 rm 容器

这背后的原理可以通过 Linux/POSIX 信号来解释。docker container stop 命令向容器内的 PID 1 进程发送了 **SIGTERM** 这样的信号。就像前文提到的一样，会为进程预留一个清理并优雅停止的机会。如果 10s 内进程没有终止，那么就会收到 **SIGKILL** 信号。这是致命一击。但是，进程起码有 10s 的时间来“解决”自己。

docker container rm <container> -f 命令不会先友好地发送 **SIGTERM**，这条命令会直接发出 **SIGKILL**。就像刚刚所打的比方一样，该命令悄悄接近并对容器发起致命一击。

重启策略与自我修复

重启策略应用于每一个容器，可以作为参数被强制传入 run 命令

容器支持的重启策略有：

1. **Always**:除非手动停止 stop 容器，否则会一直尝试重启停止的容器。--restart always
   注意：当 deamon 重启的时候，停止的容器也会重启
2. **Unless-stopped**：和 always 的区别是，当 deamon 重启时容器不会重启。--restart unless-stopped
3. **On-failed**:在退出容器但是返回值不是 0 的时候重启，deamon 重启的时候容器也会重启

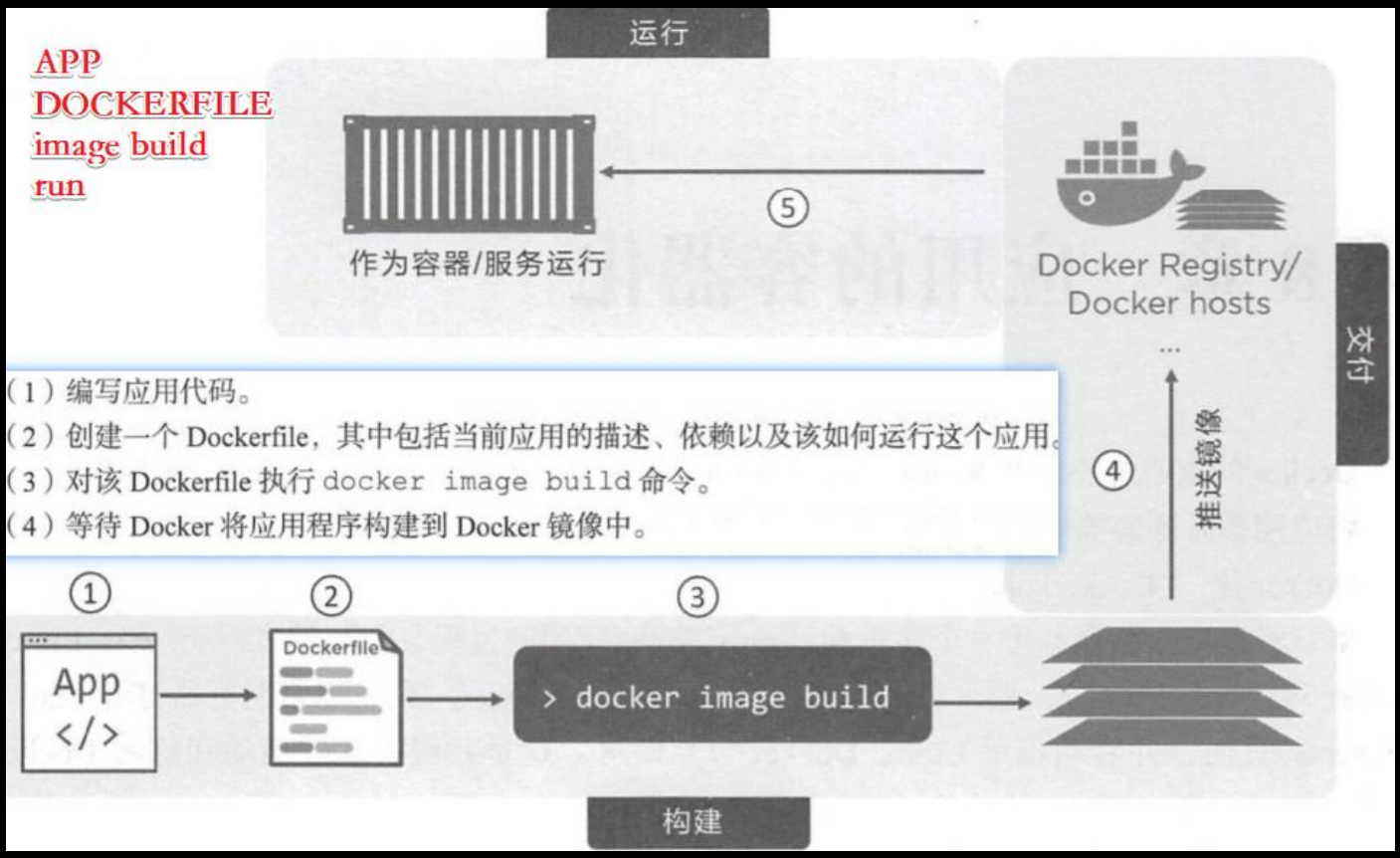
快速清理

快速清理全部容器的方法：不能在生产环境中使用

```
docker container rm $(docker container ls -aq) -f
```

第八章-应用的容器化

容器化 Dockerizing：将应用整合到容器中并运行起来的过程



Dockerfile 名字就是大写 D 开头，不能是其他的

Dockerfile 主要包括两个用途。

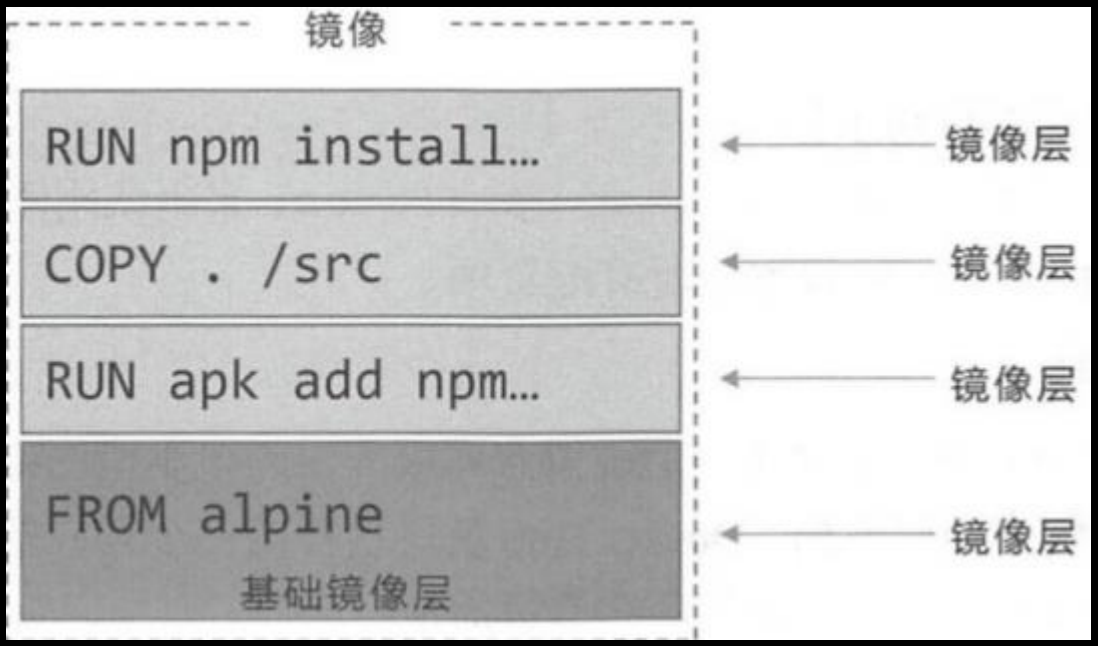
- 对当前应用的描述。
- 指导 Docker 完成应用的容器化（创建一个包含当前应用的镜像）。

示例：

```
FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

FROM alpine: 以 alpine 镜像作为当前镜像基础
LABLE maintainer:指定维护者
RUN :安装 nodejs\npm
COPY: 复制应用代码到镜像中
WORKDIR: 设置工作目录
RUN :npm install
EXPOSE:暴露的端口
ENTRYPOINT: 入口程序，运行应用

运行完后的镜像:



推送镜像

Docker image push
Docker hub 是默认的推送地址

推送前需要登录:
Docker login

推送 Docker 镜像之前，读者还需要为镜像打标签。这是因为 Docker 在镜像推送的过程中需要如下信息。

- Registry（镜像仓库服务）。
- Repository（镜像仓库）。
- Tag（镜像标签）。

读者无须为 Registry 和 Tag 指定值。当读者没有为上述信息指定具体值的时候，Docker 会默认 `Registry=docker.io`、`Tag=latest`。但是 Docker 并没有给 `Repository` 提供默认值，而是从被推送镜像中的 `REPOSITORY` 属性值获取。这一点可能不好理解，下面会通过一个完整

在本章节前面的例子中执行了 `docker image ls` 命令。在该命令对应的输出内容中可以看到，镜像仓库的名称是 web。这意味着执行 `docker image push` 命令，会尝试将镜像推送到 `docker.io/web:latest` 中。但是其实 `nigelpoulton` 这个用户并没有 `web` 这个镜像仓库的访问权限，所以只能尝试推送到 `nigelpoulton` 这个二级命名空间（Namespace）之下。因此需要使用 nigelpoulton 这个 ID，为当前镜像重新打一个标签。

```
$ docker image tag web:latest nigelpoulton/web:latest
```

```
docker image tag web:latest nigelpoulton/web:latest
```

图 8.6　确定镜像所要推送的目的仓库

多阶段构建

最佳实践

- 利用构建缓存
- 合并镜像
- No-install-recommends
- 不要安装 MSI 包(win)

第九章-Docker Compose

Docker compose 和 docker stack 类似

介绍

Docker compose 前身是 Fig
Fig 基于 Python，用于多容器管理

后 Fig 更名为 Docker compose
Docker compose 仍然是基于 python 的外部工具

安装

在 Linux 上安装 Docker Compose 分为两步。首先使用 curl 命令下载二进制文件，然后使用 chmod 命令将其置为可运行。

Compose YML 文件

默认名：**docker-compose.yml**

```yaml
version: "3.5"
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5000
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /code
  redis:
    image: "redis:alpine"
    networks:
      counter-net:

networks:
  counter-net:

volumes:
  counter-vol:
```

一级 key:

1. Version:必须、位于第一行。定义了 compose 文件格式(API 版本)，建议最新版
2. Services:定义不同的应用服务
3. Networks:指引 Docker 创建新的网络，默认情况下，会创建 bridge 网络——一种单主机网络，只能够实现同一主机上的容器的连接，也可以使用 driver 属性指定不同的网络类型。

   下面的代码可以用来创建一个名为 over-net 的 Overlay 网络，允许独立的容器（standalone container）连接（attachable）到该网络上。

   ```yaml
   networks:
     over-net:
     driver: overlay
     attachable: true
   ```

4. **Volumes**:指引 docker 创建新的卷

服务中:

- build：.指定 Docker 基于当前目录（.）下 Dockerfile 中定义的指令来构建一个新镜像。该镜像会被用于启动该服务的容器。
- command：python app.py 指定 Docker 在容器中执行名为 app.py 的 Python 脚本作为主程序。因此镜像中必须包含 app.py 文件以及 Python，这一点在 Dockerfile 中可以得到满足。
- ports：指定 Docker 将容器内（-target）的 5000 端口映射到主机（published）的 5000 端口。这意味着发送到 Docker 主机 5000 端口的流量会被转发到容器的 5000 端口。容器中的应用监听端口 5000。
- networks：使得 Docker 可以将服务连接到指定的网络上。这个网络应该是已经存在的，或者是在 networks 一级 key 中定义的网络。对于 Overlay 网络来说，它还需要定义一个 attachable 标志，这样独立的容器才可以连接上它（这时 Docker Compose 会部署独立的容器而不是 Docker 服务）。
- volumes：指定 Docker 将 counter-vol 卷（source:）挂载到容器内的/code（target:）。counter-vol 卷应该是已存在的，或者是在文件下方的 volumes 一级 key 中定义的。

默认情况下，docker-compose up 会查找名为 docker-compose.yml 或 docker-compose.yaml 的 Compose 文件。如果 Compose 文件是其他文件名，则需要通过-f 参数来指定。如下命令会基于名为 prod-equus-bass.yml 的 Compose 文件部署应用。

```
$ docker-compose -f prod-equus-bass.yml up
```

第十章-Docker Swarm

Docker Swarm 包含两方面：一个企业级的 Docker 安全集群，以及一个微服务应用编排引擎。

第十一章-Docker 网络

第十二章-Docker 覆盖网络

第十三章-卷与持久化数据

第十四章-使用 Docker Stack 部署应用

第十五章-Docker 安全

第十六章-企业版工具

第十七章-企业级特性

Overy2 模式，默认是宿主机的大小，你宿主机可以空间就是每个容器可用空间。你的云主机空间就是每个 40G，为什么是 40G 就是因为做了 overy2 模式，设置了全局大小，这样才会限制约束每个容器的空间

软件安装

Ubuntu 安装 docker

https://docker_practice.gitee.io/install/ubuntu.html

警告：切勿在没有配置 Docker APT 源的情况下直接使用 apt 命令安装 Docker.

Docker 支持以下版本的 Ubuntu 操作系统：

Focal 20.04 (LTS)

Bionic 18.04 (LTS)

Xenial 16.04 (LTS)

Docker 可以安装在 64 位的 x86 平台或 ARM 平台上。Ubuntu 发行版中，**LTS（Long-Term-Support）**长期支持版本，会获得 5 年的升级维护支持，这样的版本会更稳定，因此在生产环境中推荐使用 LTS 版本。

卸载旧版本

```
$ sudo apt-get remove docker \
               docker-engine \
               docker.io
```

使用 APT 安装：

由于 apt 源使用 HTTPS 以确保软件下载过程中不被篡改。因此，我们首先需要添加使用 HTTPS 传输的软件包以及 CA 证书。

```
$ sudo apt-get update

$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

鉴于国内网络问题，强烈建议使用国内源，官方源请在注释中查看。

为了确认所下载软件包的合法性，需要添加软件源的 GPG 密钥。

```
$ curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -

# 官方源
# $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

然后，我们需要向 sources.list 中添加 Docker 软件源

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu \
    $(lsb_release -cs) \
    stable"


# 官方源
# $ sudo add-apt-repository \
#    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
#    $(lsb_release -cs) \
#    stable"
```

安装 Docker

更新 apt 软件包缓存，并安装 docker-ce:

```
$ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

除此外 还可以使用脚本自动安装

启动 Docker

```
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

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

建立 docker 组:

```
$ sudo groupadd docker
```

将当前用户加入 docker 组:

```
$ sudo usermod -aG docker $USER
```

退出当前终端并重新登录，进行如下测试。

```
docker run hello-world
```

配置国内镜像源

在/etc/docker 下有文件 deamon.json
如果没有自己加一个
然后重启 docker

Build

```
//docker build
// . 表示当前目录 -f 参数指定 Dockerfile 文件  -t 表示 制作的镜像:tag
docker build -f Dockerfile  -t  edw/scratch-cms-backend:1.0.0  .
```

zookeeper

单机版

拉取镜像
```
docker pull zookeeper
```

将它部署在 /usr/local/zookeeper 目录下:
```
cd /usr/local && mkdir zookeeper && cd zookeeper
```
创建 data 目录, 用于挂载容器中的数据目录:
```
mkdir data
```

正式部署
```
docker run -d -e TZ="Asia/Shanghai" \
-p 2182:2182 \
-v /usr/local/zookeeper/data:/data \
-v /usr/local/zookeeper/conf/zoo.cfg:/conf/zoo.cfg \ (如果在配置中指定 data 目录, 则上面一行不用)
--name zookeeper \
--restart always zookeeper
```

```
docker run --name zookeeper -p 2182:2182 -v /usr/local/zookeeper/conf/zoo.cfg:/conf/zoo.cfg -v /usr/local/zookeeper/data:/data -d zookeeper --restart always zookeeper
```

```
-e TZ="Asia/Shanghai" # 指定上海时区
-d # 表示在一直在后台运行容器
-p 2181:2181 # 对端口进行映射, 将本地 2181 端口映射到容器内部的 2181 端口
--name # 设置创建的容器名称
-v # 将本地目录(文件)挂载到容器指定目录;
--restart always #始终重新启动 zookeeper
```

```
docker ps -a 查看状况
```

测试
使用 zk 命令行客户端连接 zk
```
docker run -it --rm --link zookeeper:zookeeper zookeeper zkCli.sh -server zookeeper
```

其它命令
```
# 查看 zookeeper 容器实例进程信息
docker top zookeeper

# 停止 zookeeper 实例进程
docker stop zookeeper
```

```
# 启动 zookeeper 实例进程
docker start zookeeper

# 重启 zookeeper 实例进程
docker restart zookeeper

# 查看 zookeeper 进程日志
docker logs -f zookeeper

# 杀死 zookeeper 实例进程
docker kill -s KILL zookeeper

# 移除 zookeeper 实例
docker rm -f -v zookeeper
```

进入容器:

```
docker exec -it 36080e8c431c bash
```

ConnectException: Connection refused:
如果是云服务器，zoo.cfg 里面的 ip 要是内网地址，不能是别名或者公网地址。

172.17.16.11

配置文件中添加

```
zookeeper.client.sasl=false
```

使用配置文件的方式 run，始终存在问题:

```
[main:ZooKeeperServerMain@81] - Unable to access datadir, exiting abnormally
```

```
docker run -d -e TZ="Asia/Shanghai" -p 9710:2181 -v /home/scratch-cms/zk/zoo.cfg:/conf/zoo.cfg --name scratch-cms_zookeeper --restart always zookeeper
```

不使用配置文件方式，直接指定端口、工作目录，是可以正常工作的

```
docker run  -e TZ="Asia/Shanghai" -d \
-p 9710:2181 \
-v /usr/local/zookeeper/data:/data \
--name scratch-cms_zookeeper \
--restart always zookeeper
```

集群方式安装

集群方式选择使用 docker-compose 来完成。

安装 docker-compose

```
curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

配置 docker-compose

编写配置文件，并将其命名为：docker-compose.yml（docker-compose 默认配置文件名）

```
version: '3.1'
networks:
  default:
    external:
      name: zookeeper_network
services:
  zoo1:
    image: zookeeper
    restart: always
    container_name: zoo1
    hostname: zoo1
    ports:
      - 9710:2181
    volumes:
      - "./zoo1/data:/data"
      - "./zoo1/datalog:/datalog"
    environment:
      ZOO_MY_ID: 1
      ZOO_SERVERS: server.1=zoo1:2888:3888;2181 server.2=zoo2:2888:3888;2181 server.3=zoo3:2888:3888;2181

  zoo2:
    image: zookeeper
    restart: always
    container_name: zoo2
    hostname: zoo2
    ports:
      - 9713:2181
    volumes:
      - "./zoo2/data:/data"
      - "./zoo2/datalog:/datalog"
    environment:
      ZOO_MY_ID: 2
      ZOO_SERVERS: server.1=zoo1:2888:3888;2181 server.2=zoo2:2888:3888;2181 server.3=zoo3:2888:3888;2181

  zoo3:
    image: zookeeper
    restart: always
    container_name: zoo3
    hostname: zoo3
    ports:
      - 9715:2181
    volumes:
      - "./zoo3/data:/data"
      - "./zoo3/datalog:/datalog"
    environment:
      ZOO_MY_ID: 3
      ZOO_SERVERS: server.1=zoo1:2888:3888;2181 server.2=zoo2:2888:3888;2181 server.3=zoo3:2888:3888;2181
```

此配置文件表示，Docker 需要启动三个 zookeeper 实例，并将 2181，2182，2183 三个端口号映射到容器内的 2181 这个端口上。

ZOO_MY_ID：表示 zk 服务的 ID，取值为 1-255 之间的整数，且必须唯一

ZOO_SERVERS：表示 zk 集群的主机列表

启动 zookeeper 集群

docker-compose up -d

该命令执行需要在 docker-compose 配置文件的目录下执行，结果如下：

```
[root@izbp13xko46hud9vfr5s94z conf]# docker-compose up -d
Starting zoo1 ... done
Starting zoo2 ... done
Starting zoo3 ... done
```

查看 zookeeper 集群实例

通过 docker ps 查看

通过 docker-compose ps 查看 （这个命令需要在 docker-compose 配置文件下执行。）

管理 docker-compose 服务

```
# 停止 docker-compose 服务 docker-compose stop
# 启动 docker-compose 服务 docker-compose start
# 重启 docker-compose 服务 docker-compose restart
```

查看 zookeeper 集群节点主从关系

使用 docker exec -it zoo1 /bin/bash 这个命令进入 zoo1 节点中，之后输入./bin/zkServer.sh statu 来查看节点主从关系

```
[root@izbp13xko46hud9vfr5s94z conf]# docker exec -it zoo1 /bin/bash
bash-4.4# ./bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /conf/zoo.cfg
Mode: follower
bash-4.4#
```

kafka

docker search kafka 可以看到可以安装的镜像

选择第一个

Docker pull wurstmeister/kafka

启动：

```
docker run --name kafka \

-p 9092:9092 \

-e KAFKA_ADVERTISED_HOST_NAME=kafka01 \

-e KAFKA_CREATE_TOPICS="test:1:1" \

-e KAFKA_ZOOKEEPER_CONNECT=服务器 IP:zookeeper 端口 \

-d  wurstmeister/kafka
```

```
docker run --name scratch-cms_kafka \
-p 9708:9092 \
-e KAFKA_ADVERTISED_HOST_NAME=scratch-cms_kafka \
```

```
-e KAFKA_ZOOKEEPER_CONNECT= 42.193.37.120:2181 \
-d  wurstmeister/kafka
```

```
docker run -d --name scratch-cms_kafka --publish 9708:9092 \
--link zookeeper \
--env KAFKA_ZOOKEEPER_CONNECT= 42.193.37.120:9710 \
--env KAFKA_ADVERTISED_HOST_NAME= 42.193.37.120 \
--env KAFKA_ADVERTISED_PORT=9092  \
wurstmeister/kafka
```

不行

这才行:

```
docker run -d --restart=always --log-driver json-file --log-opt max-size=100m --log-opt max-file=2 --name scratch-cms_kafka -p 9708:9092 -e KAFKA_BROKER_ID=0 -e KAFKA_ZOOKEEPER_CONNECT=172.17.0.9:2181/kafka -e KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://172.17.0.9:9092 -e KAFKA_LISTENERS=PLAINTEXT://0.0.0.0:9092 -v /etc/localtime:/etc/localtime wurstmeister/kafka
```

https://cloud.tencent.com/developer/article/1691991

还应该用公网 IP

```
listeners=INSIDE://0.0.0.0:9092,OUTSIDE://0.0.0.0:9708
advertised.listeners=INSIDE://localhost:9092,OUTSIDE://42.193.37.120:9708
listener.security.protocol.map=INSIDE:SASL_PLAINTEXT,OUTSIDE:SASL_PLAINTEXT,
PLAINTEXT: PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
inter.broker.listener.name=PLAINTEXT
```

```
listeners=INSIDE://0.0.0.0:9092,OUTSIDE://0.0.0.0:9708
advertised.listeners=INSIDE://localhost:9092,OUTSIDE://42.193.37.120:9708
listener.security.protocol.map=INSIDE: PLAINTEXT, OUTSIDE: PLAINTEXT, PLAINTEXT: PLAINTEXT
inter.broker.listener.name=PLAINTEXT
```

用这个: ? ? ?

```
listeners=PLAINTEXT://0.0.0.0:9092
advertised.listeners=PLAINTEXT://42.193.37.120:9708
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
inter.broker.listener.name=PLAINTEXT
```

```
docker run -d --restart=always --log-driver json-file --log-opt max-size=100m --log-opt max-file=2 --name scratch-cms_kafka -p 9708:9092 -e KAFKA_BROKER_ID=0 -
```

```
e                KAFKA_ZOOKEEPER_CONNECT=42.193.37.120:9710/kafka                -e
KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://172.17.0.9:9092                          -e
KAFKA_LISTENERS=PLAINTEXT://0.0.0.0:9092     -v     /etc/localtime:/etc/localtime
wurstmeister/kafka
```

listeners
是 kafka 真正 bind 的地址

advertised.listeners
是暴露给外部的 listeners，如果没有设置，会用 listeners

advertised_listeners 监听器会注册在 zookeeper 中

在公司内网部署 kafka 集群只需要用到 listeners，内外网需要作区分时 才需要用到 advertised.listeners。

那么先看看文字类描述：
listeners： 学名叫监听器，其实就是告诉外部连接者要通过什么协议访问指定主机名和端口开放的 Kafka 服务。
advertised.listeners：和 listener 相比多了个 advertised。Advertise 的含义表示宣称的、公布的，就是组监听器是 Broker 用于对外发布的。

```
listeners=INSIDE://172.17.0.10:9092,OUTSIDE://<公网 ip>:端口
advertised.listeners=INSIDE://172.17.0.10:9092,OUTSIDE://<公网 ip>:端口
listener.security.protocol.map=INSIDE:SASL_PLAINTEXT,OUTSIDE:SASL_PLAINTEXT
inter.broker.listener.name=INSIDE
```

advertised_listeners 监听器会注册在 zookeeper 中；

当我们对 172.17.0.10:9092 请求建立连接，kafka 服务器会通过 zookeeper 中注册的监听器，找到 INSIDE 监听器，然后通过 listeners 中找到对应的 通讯 ip 和 端口；

同理，当我们对 <公网 ip>:端口 请求建立连接，kafka 服务器会通过 zookeeper 中注册的监听器，找到 OUTSIDE 监听器，然后通过 listeners 中找到对应的 通讯 ip 和 端口 172.17.0.10:9094；

在 docker 中或者 在类似阿里云主机上部署 kafka 集群，这种情况下是 需要用到 advertised_listeners。

测试

进入容器

```
docker exec -it kafka bash
```

kafka 默认安装在/opt/kafka

cd opt/kafka

创建 topic

```
bin/kafka-topics.sh --create --zookeeper 42.193.37.120:9710/kakfa --
replication-factor 1 --partitions 1 --topic course_message
```

查看 topic

```
bin/kafka-topics.sh --list --zookeeper 42.193.37.120:9710/kafka
```

./kafka-topics.sh --replication-factor 1 --partitions 1 --create --topic project_message --zookeeper 42.193.37.120:9710/kafka

集群

依赖前面的 zoo123

```
version: '3.1'

networks:
  default:
    external:
      name: zookeeper_network
services:
  kafka1:
    image: wurstmeister/kafka
    restart: unless-stopped
    container_name: kafka1
    hostname: kafka1
    ports:
      - "9708:9092"
    external_links:
      - zoo1
      - zoo2
      - zoo3
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://42.193.37.120:9708    ## 宿主机 IP
      KAFKA_ADVERTISED_HOST_NAME: kafka1
      KAFKA_ADVERTISED_PORT: 9708
      KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181,zoo2:2181,zoo3:2181"
    volumes:
      - "./kafka/kafka1/data/:/kafka"


  kafka2:
    image: wurstmeister/kafka
    restart: unless-stopped
```

```
    container_name: kafka2
    hostname: kafka2
    ports:
      - "9716:9092"
    external_links:
      - zoo1
      - zoo2
      - zoo3
    environment:
      KAFKA_BROKER_ID: 2
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://42.193.37.120:9716    ## 宿主机 IP
      KAFKA_ADVERTISED_HOST_NAME: kafka2
      KAFKA_ADVERTISED_PORT: 9716
      KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181,zoo2:2181,zoo3:2181"
    volumes:
      - "./kafka/kafka2/data/:/kafka"

  kafka3:
    image: wurstmeister/kafka
    restart: unless-stopped
    container_name: kafka3
    hostname: kafka3
    ports:
      - "9717:9092"
    external_links:
      - zoo1
      - zoo2
      - zoo3
    environment:
      KAFKA_BROKER_ID: 3
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://42.193.37.120:9717    ## 宿主机 IP
      KAFKA_ADVERTISED_HOST_NAME: kafka3
      KAFKA_ADVERTISED_PORT: 9717
      KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181,zoo2:2181,zoo3:2181"
    volumes:
      - "./kafka/kafka3/data/:/kafka"

  kafka-manager: # Kafka 图形管理界面
    image: sheepkiller/kafka-manager:latest
    restart: unless-stopped
    container_name: kafka-manager
    hostname: kafka-manager
    ports:
      - "9718:9000"
    links:           # 连接本 compose 文件创建的 container
      - kafka1
      - kafka2
      - kafka3
    external_links:   # 连接外部 compose 文件创建的 container
      - zoo1
      - zoo2
      - zoo3
    environment:
      ZK_HOSTS: zoo1:2181,zoo2:2181,zoo3:2181
      KAFKA_BROKERS: kafka1:9708,kafka2:9716,kafka3:9717
```

Kafka 的配置

https://blog.csdn.net/weixin_38251332/article/details/105638535

https://www.jianshu.com/p/26495e334613

minio

```
docker search minio
docker pull minio/minio

docker run -p 9710:9710 --name scratch-cms_minio \
-d --restart=always \
-e "MINIO_ACCESS_KEY=root" \
-e "MINIO_SECRET_KEY=rootroot" \
-v /usr/local/minio/data:/data \
-v /usr/local/minio/config:/root/.minio \
minio/minio server /data



docker run -d -p 9000:9000 --name minio --restart=always -v /lhb/data-server/minio/data:/data -v /lhb/data-server/minio/config:/root/.minio 324b server /data

docker run -p 9000:9000 --name scratch-cms_minio -v /usr/local/minio/data:/data -v /usr/local/minio/config:/root/.minio -d b5f1d82e0a6d server /data


一直不行，原来是容器端口必须是9000，宿主机端口必须是外网可以访问的
docker run -p 9709:9000 --name scratch-cms_minio  -d \
  -e "MINIO_ACCESS_KEY=root" \
  -e "MINIO_SECRET_KEY=rootroot" \
  -v /mnt/data:/data \
  -v /mnt/config:/root/.minio \
  minio/minio server /data
```

Redis

```
docker run -d   -v /usr/local/redis/data:/data --name scratch-cms_redis -p 9706:6380 redis redis-server --requirepass "" --appendonly yes


docker run -d -p 9706:6380 -v $PWD/redis.conf:/usr/local/etc/redis/redis.conf -v $PWD/data:/data --name scratch-cms_redis redis redis-server /usr/local/etc/redis/redis.conf --appendonly yes  --requirepass "123456"


redis-cli.exe -h 42.193.37.120 -p 9706
```

获取配置文件 redis.conf
修改默认配置文件
bind 127.0.0.1 #注释掉这部分，这是限制 redis 只能本地访问
protected-mode no #默认 yes，开启保护模式，限制为本地访问
daemonize no#默认 no，改为 yes 意为以守护进程方式启动，可后台运行，除非 kill 进程（可选），改为 yes 会使配置文件方式启动 redis 失败
dir ./ #输入本地 redis 数据库存放文件夹（可选）
appendonly yes #redis 持久化（可选）

挂载配置文件:
docker run -p 9706:9706 --name scratch-cms_redis -v /home/scratch-cms/redis/redis.conf:/etc/redis/redis.conf -v /usr/local/docker/data:/data -d redis redis-server /etc/redis/redis.conf --appendonly yes

 # 不挂载配置文件: docker run --name redis -p 6379:6379 -d --restart=always redis redis-server --appendonly yes --requirepass ""

 -p 6379:6379 端口映射: 前表示主机部分,: 后表示容器部分。
--name myredis 指定该容器名称，查看和进行操作都比较方便。
-v 挂载目录，规则与端口映射相同。
-d redis 表示后台启动 redis
redis-server /etc/redis/redis.conf 以配置文件启动 redis，加载容器内的 conf 文件，最终找到的是挂载的目录/usr/local/docker/redis.conf
appendonly yes 开启 redis 持久化

mysql

```
docker run --restart=always --privileged=true -d \
 -p 9705:3306 --name scratch-cms_mysql \
 -e MYSQL_ROOT_PASSWORD=root mysql:latest
```