

本文档遵循知识共享许可协议(Creative Commons license 4.0),以下简称 CC 4.0, 使用者需遵循以下授权内容:

<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.zh>

您必须遵守下列条件:

1.署名: 使用本系列文档及文档内容, 您必须注明源作者信息(马哥教育张士杰老师, Email:2973707860@qq.com), 并提供本许可证的链接(<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.zh>),

同时需要声明您的内容是否经过修改, 你可以采用适当的方式将相关修改信息表述出来, 但是您不能在信息中以任何形式暗示授权人已正式认可了您以及您的使用行为。

2.非商业性使用: 您不能将本系列文档用于其他任何商业用途。

3.相同方式授权: 如您对本系列文档进行修改或使用本文档中内容, 那么您必须基于本协议分发您的作品(署名源作则马哥教育张士杰老师并使用本许可协议)。

您的权利:

当遵循本协议后, 您将有以下权限:

共享 — 允许以非商业性质复制本作品。

改编 — 在原基础上修改、复制或以本作品为基础进行重新编辑并用于个人非商业使用。

只要您遵守本许可协议条款, 许可人(马哥教育张士杰老师)将授权您使用本系列文档, 如果您违背本许可协议条款, 许可人(马哥教育张士杰老师)将有权收回您的授权并保留进一步追究其法律责任的权利。

<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

Under the following terms:

Attribution — You must give appropriate credit(www.magedu.com,Zhang ShiJie,Email:2973707860@qq.com), provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Docker



讲师：张士杰/杰哥(QQ:2973707860)

<http://www.magedu.com>

目录

一：简介：	9
1.1：docker 简介：	9
1.1.1：Docker 是什么：	9
1.1.2：Docker 的组成：	10
1.1.3：Docker 对比虚拟机：	11
1.1.4：Linux Namespace 技术：	13
1.1.5：Linux control groups：	20
1.1.6：容器管理工具：	22
1.1.6：Docker 的优势：	25
1.1.7：Docker 的缺点：	25
1.1.8：docker(容器)的核心技术：	25
1.1.9：docker(容器)的依赖技术：	28
1.2：Docker 安装及基础命令介绍：	29
1.2.1：下载 rpm 包安装：	30
1.2.2：通过修改 yum 源安装：	30
1.2.3：ubuntu 安装 docker、启动并验证服务：	31
1.2.4：验证 docker 版本：	32

1.2.5: 验证 docker0 网卡:	32
1.2.6: 验证 docker 信息:	32
1.2.7: 解决不支持 swap 限制警告:	34
1.2.8: docker 存储引擎:	34
1.2.9: docker 服务进程:	36
1.3: docker 镜像加速配置:	41
1.3.1: 获取加速地址:	41
1.3.2: 生成配置文件:	42
1.3.3: 重启 docke 服务:	42
1.4: Docker 镜像管理:	43
1.4.1: 搜索镜像:	45
1.4.2: 下载镜像:	45
1.4.3: 查看本地镜像:	46
1.4.4: 镜像导出:	47
1.4.5: 镜像导入:	47
1.4.6: 删 除镜像:	48
1.5: 容器操作基础命令:	48
1.5.1: 从镜像启动一个容器:	49
1.5.2: 显示正在运行的容器:	49
1.5.3: 显示所有容器:	49
1.5.4: 删除运行中的容器:	49
1.5.5: 随机映射端口:	49
1.5.6: 指定端口映射:	50
1.5.7: 查看容器已经映射的端口:	51
1.5.8: 自定义容器名称:	51
1.5.9: 后台启动容器:	51
1.5.10: 创建并进入容器:	51
1.5.11: 单次运行:	52
1.5.12: 传递运行命令:	52
1.5.13: 容器的启动和关闭:	52
1.5.14: 进入到正在运行的容器:	52
1.5.15: 查看容器内部的 hosts 文件:	55
1.5.16: 批量关闭正在运行的容器:	55
1.5.17: 批量强制关闭正在运行的容器:	55
1.5.18: 批量删除已退出容器:	56

1.5.19: 批量删除所有容器:	56
1.5.20: 指定容器 DNS:	56
1.5.21: 其他命令:	56
二: Docker 镜像与制作:	57
2.1: 手动制作 yum 版 nginx 镜像:	58
2.1.1: 下载镜像并初始化系统:	58
2.1.2: yum 安装并配置 nginx:	58
2.1.3: 关闭 nginx 后台运行:	58
2.1.4: 自定义 web 页面:	59
2.1.5: 提交为镜像:	59
2.1.6: 带 tag 的镜像提交:	59
2.1.7: 从自己镜像启动容器:	60
2.1.8: 访问测试:	60
2.2: Dockerfile 制作编译版 nginx 1.16.1 镜像:	60
2.2.1: 下载镜像并初始化系统:	61
2.2.2: 编写 Dockerfile:	61
2.2.3: 准备源码包与配置文件:	62
2.2.4: 执行镜像构建:	62
2.2.5: 构建完成:	63
2.2.6: 查看是否生成本地镜像:	64
2.2.7: 从镜像启动容器:	64
2.2.8: 访问 web 界面:	65
2.2.9: 镜像编译制作途中:	65
2.3: 手动制作编译版本 nginx 1.16.1 镜像:	65
2.3.1: 下载镜像并初始化系统:	65
2.3.2: 编译安装 nginx:	65
2.3.3: 关闭 nginx 后台运行:	66
2.3.4: 自定义 web 界面:	66
2.3.5: 创建用户及授权:	66
2.3.6: 在宿主机提交为镜像:	66
2.3.7: 从自己的镜像启动容器:	66
2.3.8: 访问测试:	67
2.3.9: 查看 Nginx 访问日志:	67
2.4: 自定义 Tomcat 业务镜像:	67
2.4.1: 构建 JDK 镜像:	67

2.4.2: 从 JDK 镜像构建 tomcat 8 Base 镜像:	70
2.4.3: 构建业务镜像 1:	72
2.4.4: 构建业务镜像 2:	75
2.5: 构建 haproxy 镜像:	76
2.5.1: 准备 Dockerfile:	76
2.5.2: 准备 haproxy 源码和配置文件:	77
2.5.3: 准备 haproxy 配置文件:	77
2.5.4: 准备镜像构建脚本:	78
2.5.5: 执行构建 haproxy 镜像:	79
2.5.6: 从镜像启动容器:	79
2.5.7: web 访问验证:	79
2.5.8: 访问 haproxy 控制端:	79
2.6: 基于官方 alpine 基础镜像制作自定义镜像:	80
2.7: 基于官方 Ubuntu 基础镜像制作自定义镜像:	80
2.8: 本地镜像上传至官方 docker 仓库:	81
2.8.1: 准备账户:	81
2.8.2: 填写账户基本信息:	82
2.8.3: 在虚拟机使用自己的账号登录:	82
2.8.4: 查看认证信息:	82
2.8.5: 给镜像做 tag 并开始上传:	83
2.8.6: 上传完成	83
2.8.7: 到 docker 官网验证:	83
2.8.8: 更换到其他 docker 服务器下载镜像:	83
2.8.9: 从镜像启动一个容器:	84
2.9: 本地镜像上传到阿里云:	84
2.9.1: 将镜像推送到 Registry	84
2.9.2: 阿里云验证镜像:	84
三: Docker 数据管理:	85
3.1: 数据类型:	86
3.1.1: 什么是数据卷(data volume):	87
3.1.2: 文件挂载:	90
3.1.3: 数据卷容器:	92
四: 网络部分:	95
4.1: docker 结合负载实现网站高可用:	96
4.1.1: 整体规划图:	96

4.1.2: 安装并配置 keepalived:	97
4.1.3: 安装并配置 haproxy:	98
4.1.4: 服务器启动 nginx 容器并验证:	101
4.2: 容器之间的互联:	103
4.2.1: 通过容器名称互联:	103
4.2.2: 通过自定义容器别名互联:	104
4.3: 创建自定义网络:	114
4.3.1: 创建自定义 docker 网络:	114
4.3.2: 创建不同网络的容器测试通信:	114
4.3.3: 当前 iptables 规则:	116
4.3.4: 如何与使用默认网络的容器通信:	116
4.3.5: 重新导入 iptables 并验证通信:	117
五: Docker 仓库之单机 Docker Registry:	118
5.1: 下载 docker registry 镜像:	118
5.2: 搭建单机仓库:	118
5.2.1: 创建授权使用目录:	118
5.2.2: 创建用户:	119
5.2.3: 验证用户名密码:	119
5.2.4: 启动 docker registry:	119
5.2.5: 验证端口和容器:	119
5.2.6: 测试登录仓库:	119
5.2.7: 在 Server1 登录后上传镜像:	120
5.2.8: Server 2 下载镜像并启动容器:	121
六: docker 仓库之分布式 Harbor	123
6.1: Harbor 功能官方介绍:	123
6.2: 安装 Harbor:	124
6.2.1: 服务器 1 安装 docker:	124
6.2.2: 服务器 2 安装 docker:	124
6.2.3: 下载 Harbor 安装包:	125
6.3: 配置 Harbor:	125
6.3.1: 解压并编辑 harbor.cfg:	125
6.3.2: 更新 harbor 配置:	126
6.3.3: 官方方式启动 Harbor:	127
6.3.4: 非官方方式启动:	130
6.4: 配置 docker 使用 harbor 仓库上传下载镜像:	131

6.4.1: 编辑 docker 配置文件:	131
6.4.2: 重启 docker 服务:	131
6.4.3: 验证能否登录 harbor:	131
6.4.4: 测试上传和下载镜像:	132
6.4.5: 验证从 harbor 服务器下载镜像并启动容器:	134
6.4.6: 从镜像启动容器并验证:	135
6.5: 实现 harbor 高可用:	136
6.5.1: 新部署一台 harbor 服务器:	136
6.5.2: 验证从 harbor 登录:	138
6.5.3: 创建一个 nginx 项目:	138
6.5.4: 在主 harbor 服务器配置同步测试:	139
6.5.5: 点击复制规则:	139
6.5.6: 主 harbor 编辑同步策略:	140
6.5.7: 主 harbor 查看镜像同步状态:	140
6.5.8: 从 harbor 查看镜像:	141
6.5.9: 测试从 harbor 镜像下载和容器启动:	141
6.6.: 实现 harbor 双向同步:	142
6.6.1: 在 docker 客户端导入 centos 基础镜像:	142
6.6.2: 镜像打 tag:	143
6.6.3: 上传到从 harbor:	143
6.6.4: 从 harbor 界面验证:	143
6.6.5: 从 harbor 创建同步规则:	143
6.6.6: 到主 harbor 验证镜像:	144
6.6.7: docker 镜像端测试:	144
6.7: harbor https 配置:	145
七: 单机编排之 Docker Compose:	146
7.1: 基础环境准备:	147
7.1.1: 安装 python-pip 软件包:	147
7.1.2: 安装 docker compose:	147
7.1.3: 验证 docker-compose 版本:	148
7.1.4: 查看 docker-compose 帮助:	149
7.2: 从 docker compose 启动单个容器:	150
7.2.1: 单个容器的 docker compose 文件:	150
7.2.2: 启动容器:	151
7.2.3: 启动完成:	151

7.2.4: web 访问测试: _____	152
7.2.5: 后台启动服务: _____	152
7.2.6: 自定义容器名称: _____	152
7.2.7: 验证容器: _____	153
7.2.8: 查看容器进程: _____	153
7.3: 从 docker compose 启动多个容器: _____	153
7.3.1: 编辑 docker-compose 文件: _____	153
7.3.2: 重新启动容器: _____	154
7.3.3: web 访问测试: _____	154
7.4: 定义数据卷挂载: _____	155
7.4.1: 创建数据目录和文件: _____	155
7.4.2: 编辑 compose 配置文件: _____	155
7.4.3: 重启容器: _____	155
7.4.4: 验证 web 访问: _____	156
7.4.5: 其他常用命令: _____	156
7.5: 实现单机版的 Nginx+Tomcat: _____	157
7.5.1: 制作 Haproxy 镜像: _____	157
7.5.2: 准备 nginx 镜像: _____	159
7.5.3: 准备 tomcat 镜像: _____	160
7.5.4: 编辑 docker compose 文件及环境准备: _____	160
7.5.6: 验证容器启动成功: _____	163
7.5.7: 查看启动日志: _____	164
7.5.8: 访问 haroxy 管理界面: _____	164
7.5.9: 访问 Nginx 静态页面: _____	165
7.5.10: 访问 tomcat 静态页面: _____	165
7.5.11: 访问 tomcat 动态页面: _____	165
7.5.12: 验证 Nginx 容器访问日志: _____	165

一：简介：

前言

统称来说，容器是一种工具，指的是可以装下其它物品的工具，以方便人类归纳放置物品、存储和异地运输，具体来说比如人类使用的衣柜、行李箱、背包等可以成为容器，但今天我们所说的容器是一种 IT 技术。

容器技术是虚拟化、云计算、大数据之后的一门新兴的并且是炙手可热的新技术，容器技术提高了硬件资源利用率、方便了企业的业务快速横向扩容、实现了业务宕机自愈功能，因此未来数年会是一个容器愈发流行的时代，这是一个对于 IT 行业来说非常有影响和价值的技术，而对于 IT 行业的从业者来说，熟练掌握容器技术无疑是一个很有前景的行业工作机会。

容器技术最早出现在 freebsd 叫做 jail。

1.1：docker 简介：

1.1.1：Docker 是什么：

首先 Docker 是一个在 2013 年开源的应用程序并且是一个基于 go 语言编写是一个开源的 PAAS 服务(Platform as a Service，平台即服务的缩写)，go 语言是由 google 开发，docker 公司最早叫 dotCloud 后由于 Docker 开源后大受欢迎就将公司改名为 Docker Inc，总部位于美国加州的旧金山，Docker 是基于 linux 内核实

现，Docker 最早采用 LXC 技术(LinuX Container 的简写，LXC 是 Linux 原生支持的容器技术，可以提供轻量级的虚拟化，可以说 docker 就是基于 LXC 发展起来的(0.1.5 (2013-04-17)，提供 LXC 的高级封装，发展标准的配置方法)，而虚拟化技术 KVM(Kernel-based Virtual Machine) 基于模块实现，Docker 后改为自己研发并开源的 runc 技术运行容器(1.11.0 (2016-04-13))。

Docker now relies on containerd and runc to spawn containers.

Docker 相比虚拟机的交付速度更快，资源消耗更低，Docker 采用客户端/服务端架构，使用远程 API 来管理和创建 Docker 容器，其可以轻松的创建一个轻量级的、可移植的、自给自足的容器，docker 的三大理念是 build(构建)、ship(运输)、run(运行)，Docker 遵从 apache 2.0 协议，并通过 (namespace 及 cgroup 等) 来提供容器的资源隔离与安全保障等，所以 Docker 容器在运行时不需要类似虚拟机（空运行的虚拟机占用物理机的一定性能开销）的额外资源开销，因此可以大幅提高资源利用率，总而言之 Docker 是一种用了新颖方式实现的轻量级虚拟机。类似于 VM 但是在原理和应用上和 VM 的差别还是很大的，并且 docker 的专业叫法是应用容器(Application Container)。

1.1.2: Docker 的组成:

<https://docs.docker.com/engine/docker-overview/>

Docker 主机(Host): 一个物理机或虚拟机，用于运行 Docker 服务进程和容器。

Docker 服务端(Server): Docker 守护进程，运行 docker 容器。

Docker 客户端(Client): 客户端使用 docker 命令或其他工具调用 docker API。

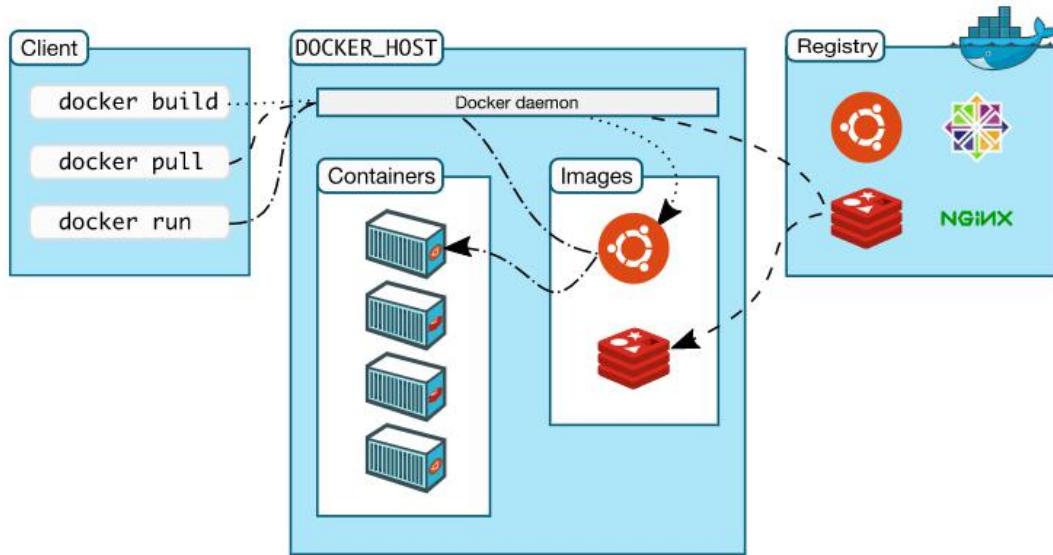
Docker 仓库(Registry): 保存镜像的仓库，类似于 git 或 svn 这样的版本控制系统。

Docker 镜像(Images): 镜像可以理解为创建实例使用的模板。

Docker 容器(Container): 容器是从镜像生成对外提供服务的一个或一组服务。

统，官方仓库: <https://hub.docker.com/>



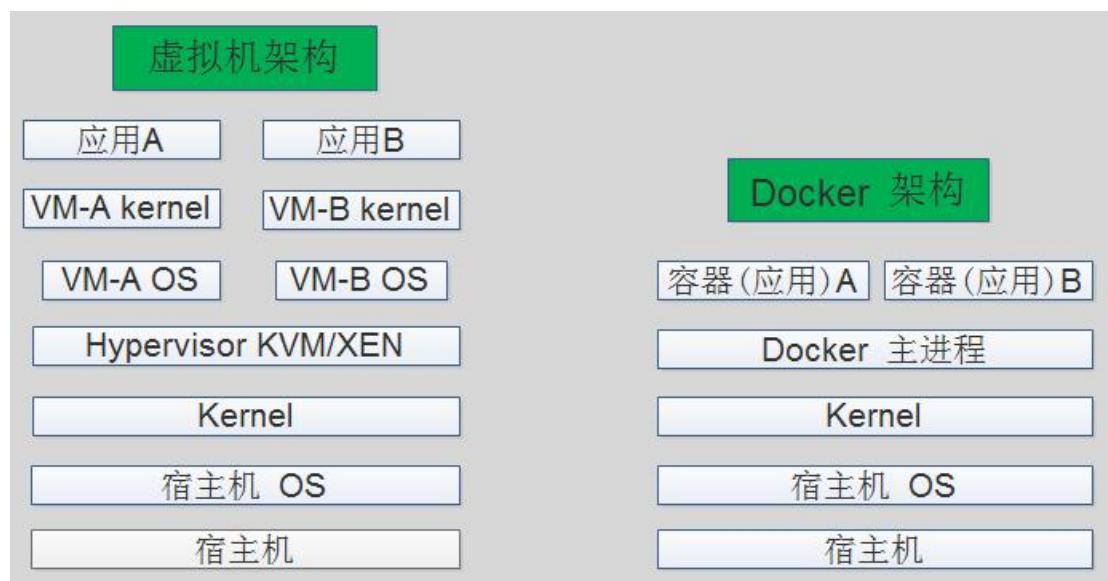


1.1.3: Docker 对比虚拟机:

资源利用率更高: 一台物理机可以运行数百个容器，但是一般只能运行数十个虚拟机。

开销更小: 不需要启动单独的虚拟机占用硬件资源。

启动速度更快: 可以在数秒内完成启动。

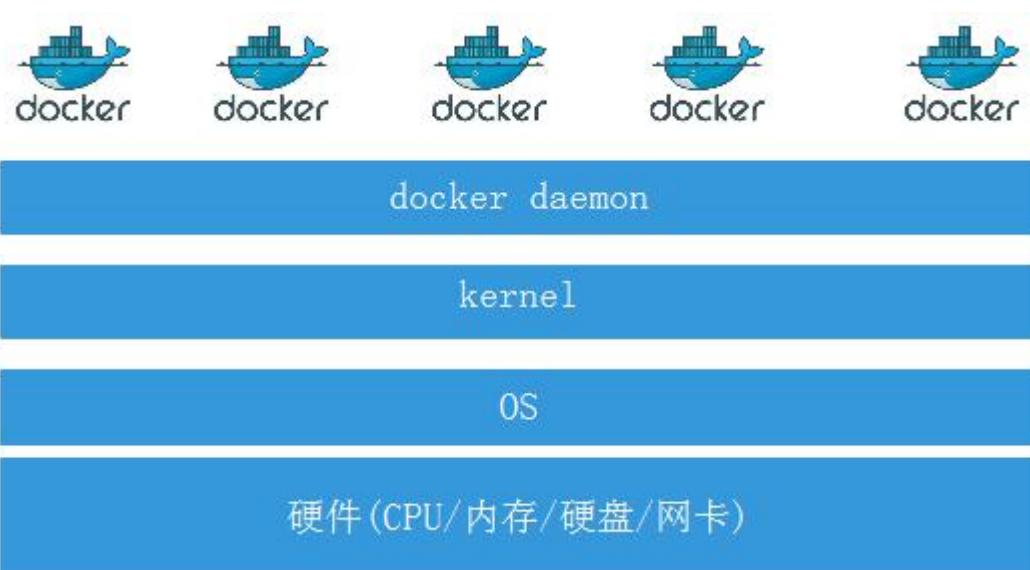


VM vs. Docker



Size	A large grey elephant icon.	A small white mouse icon.
Startup	A slow green turtle icon.	A fast dolphin icon.
Integration	A red angry face icon.	A yellow smiling face icon.

使用虚拟机是为了更好的实现服务运行环境隔离，每个虚拟机都有独立的内核，虚拟化可以实现不同操作系统的虚拟机，但是通常一个虚拟机只运行一个服务，很明显资源利用率比较低且造成不必要的性能损耗，我们创建虚拟机的目的是为了运行应用程序，比如 Nginx、PHP、Tomcat 等 web 程序，使用虚拟机无疑带来了一些不必要的资源开销，但是容器技术则基于减少中间运行环节带来较大的性能提升。



但是，如上图一个宿主机运行了 N 个容器，多个容器带来的以下问题怎么解决：

1. 怎么样保证每个容器都有不同的文件系统并且能互不影响？
2. 一个 docker 主进程内的各个容器都是其子进程，那么实现同一个主进程中不同类型的子进程？各个进程间通信能相互访问(内存数据)吗？
3. 每个容器怎么解决 IP 及端口分配的问题？
4. 多个容器的主机名能一样吗？
5. 每个容器都要不要有 root 用户？怎么解决账户重名问题？

以上问题怎么解决？

1.1.4: Linux Namespace 技术：

namespace 是 Linux 系统的底层概念，在内核层实现，即有一些不同类型的命名空间被部署在核内，各个 docker 容器运行在同一个 docker 主进程并且共用同一个宿主机系统内核，各 docker 容器运行在宿主机的用户空间，每个容器都要有类似于虚拟机一样的相互隔离的运行空间，但是容器技术是在一个进程内实现运行指定服务的运行环境，并且还可以保护宿主机内核不受其他进程的干扰和影响，如文件系统空间、网络空间、进程空间等，目前主要通过以下技术实现容器运行空间的相互隔离：

隔离类型	功能	系统调用参数	内核版本
MNT Namespace(mount)	提供磁盘挂载点和文件系统的隔离能力	CLONE_NEWNS	Linux 2.4.19
IPC Namespace(Inter-Process Communication)	提供进程间通信的隔离能力	CLONE_NEWPIC	Linux 2.6.19
UTS Namespace(UNIX Timesharing System)	提供主机名隔离能力	CLONE_NEWUTS	Linux 2.6.19
PID Namespace(Process Identification)	提供进程隔离能力	CLONE_NEWPID	Linux 2.6.24
Net Namespace(network)	提供网络隔离能力	CLONE_NEWWNET	Linux 2.6.29
User Namespace(user)	提供用户隔离能力	CLONE_NEWUSER	Linux 3.8

1.1.4.1: MNT Namespace:

每个容器都要有独立的根文件系统有独立的用户空间，以实现在容器里面启动服务并且使用容器的运行环境，即一个宿主机是 ubuntu 的服务器，可以在里面启动一个 centos 运行环境的容器并且在容器里面启动一个 Nginx 服务，此 Nginx 运行时使用的运行环境就是 centos 系统目录的运行环境，但是在容器里面是不能访问宿主机的资源，宿主机是使用了 chroot 技术把容器锁定到一个指定的运行目录里面。

例如：/var/lib/containerd/io.containerd.runtime.v1.linux/moby/**容器 ID**

启动三个容器用于以下验证过程：

Server: Docker Engine - Community

Engine:

```
Version:          18.09.7
API version:     1.39 (minimum version 1.12)
Go version:      go1.10.8
Git commit:      2d0083d
Built:           Thu Jun 27 17:26:28 2019
OS/Arch:         linux/amd64
Experimental:    false
```

```
# docker run -d --name nginx-1 -p 80:80 nginx
# docker run -d --name nginx-2 -p 81:80 nginx
# docker run -d --name nginx-3 -p 82:80 nginx
```

Debian 系统安装基础命令：

```
# apt update
# apt install procps (top 命令)
# apt install iputils-ping (ping 命令)
# apt install net-tools (网络工具)
```

验证容器的根文件系统：

```
[root@node1 ~]# cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
[root@node1 ~]# docker exec -it 5a7c6090b8adae677c8c sh
# ls / ← 容器的根文件系统，每个容器都有一个独立且完整的根文件系统。
bin  boot  dev  etc  home  index.html  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
#
```

1.1.4.2: IPC Namespace:

一个容器内的进程间通信，允许一个容器内的不同进程的(内存、缓存等)数据访问，但是不能夸容器访问其他容器的数据。

1.1.4.3: UTS Namespace:

UTS namespace (UNIX Timesharing System 包含了运行内核的名称、版本、底层体系结构类型等信息)用于系统标识，其中包含了 hostname 和域名 domainname，它使得一个容器拥有属于自己 hostname 标识，这个主机名标识独立于宿主机系统和其上的其他容器。

```

# cat /etc/issue
Debian GNU/Linux 9 \n \l

# apt-get update
Hit:1 http://security-cdn.debian.org/debian-security stretch/updates InRelease
Ign:2 http://cdn-fastly.deb.debian.org/debian stretch InRelease
Hit:3 http://cdn-fastly.deb.debian.org/debian stretch-updates InRelease
Hit:4 http://cdn-fastly.deb.debian.org/debian stretch Release
Reading package lists... Done
# uname -a
Linux 5a7c6090b8ad 3.10.0-327.el7.x86_64 #1 SMP Thu Nov 19 22:10:57 UTC 2015 x86_64 GNU/Linux
# hostname
5a7c6090b8ad
#

```

宿主机的内核版本

1.1.4.4: PID Namespace:

Linux 系统中，有一个 PID 为 1 的进程(init/systemd)是其他所有进程的父进程，那么在每个容器内也要有一个父进程来管理其下属的子进程，那么多个容器的进程通过 PID namespace 进程隔离(比如 PID 编号重复、器内的主进程生成与回收子进程等)。

例如：下图是在一个容器内使用 top 命令看到的 PID 为 1 的进程是 nginx：

```

top - 05:39:50 up 19:27, 0 users, load average: 0.00, 0.01, 0.05
Tasks: 4 total, 1 running, 3 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.1 us, 0.2 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 1868692 total, 514628 free, 292976 used, 1061088 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 1418708 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM TIME+ COMMAND
        1 root      20   0  32648  3244  2436 S  0.0  0.2  0:00.07 nginx
         6 nginx    20   0  33104  1600   340 S  0.0  0.1  0:00.00 nginx
      3839 root      20   0   4272   628   540 S  0.0  0.0  0:00.02 sh
      3844 root      20   0  41032  1776  1308 R  0.0  0.1  0:00.00 top

```

容器内的 Nginx 主进程与工作进程：

```

# ps -ef | grep nginx
root      1  0  Jun30 ?        00:00:00 nginx: master process nginx -g daemon off;
nginx     6  1  Jun30 ?        00:00:00 nginx: worker process
root  3846 3839  0 05:40 pts/0    00:00:00 grep nginx
#

```

那么宿主机的 PID 究竟与容器内的 PID 是什么关系？

容器 PID 追踪：

1.1.4.4.1：查看宿主机上的 PID 信息：

```
[root@node1 ~]# ps -ef | grep docker ← 查看docker相关的进程
root  5064  1  0 Jun30 ?      00:00:58 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
root  5487  5064  0 Jun30 ?      00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 80
    -container-ip 172.17.0.2 -container-port 80
root  5492  5061  0 Jun30 ?      00:00:06 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/5a7c6090b8adae677c8cd176f9130ce83d27e60093f9424cbe9a99c86d69c690 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  26180  5064  0 11:58 ?      00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 81
    -container-ip 172.17.0.3 -container-port 80
root  26186  5061  0 11:58 ?      00:00:00 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/c3f729d7be609297e8b2b1b57364257f286707b431ef5db1bfee899f4dec2d92 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  26318  5064  0 11:58 ?      00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 82
    -container-ip 172.17.0.4 -container-port 80
root  26324  5061  0 11:58 ?      00:00:01 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/c981ef6a234c5f2201fd90663d06d3fa7331b64a7246dal835abc8f689a6866 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  27554  21259  0 13:39 pts/1   00:00:00 docker exec -it 5a7c6090b8adae677c8c sh
root  27797  26765  0 14:40 pts/2   00:00:00 grep --color=auto docker
[root@node1 ~]#
[root@node1 ~]# ps -ef | grep 5061 ← 查看PID为5061是什么进程
root  5061  1  0 Jun30 ?      00:06:38 /usr/bin/containerd
root  5492  5061  0 Jun30 ?      00:00:06 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/5a7c6090b8adae677c8cd176f9130ce83d27e60093f9424cbe9a99c86d69c690 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  26186  5061  0 11:58 ?      00:00:00 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/c3f729d7be609297e8b2b1b57364257f286707b431ef5db1bfee899f4dec2d92 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  26324  5061  0 11:58 ?      00:00:01 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/c981ef6a234c5f2201fd90663d06d3fa7331b64a7246dal835abc8f689a6866 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  27799  26765  0 14:42 pts/2   00:00:00 grep --color=auto 5061
[root@node1 ~]#
[root@node1 ~]# ps -ef | grep 5492 ← 查看其中一个容器的PID信息
root  5492  5061  0 Jun30 ?      00:00:06 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/5a7c6090b8adae677c8cd176f9130ce83d27e60093f9424cbe9a99c86d69c690 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  5510  5492  0 Jun30 ?      00:00:00 nginx: master process nginx -g daemon off;
root  27574  5492  0 13:39 pts/0   00:00:00 sh
root  27801  26765  0 14:42 pts/2   00:00:00 grep --color=auto 5492
[root@node1 ~]#
```

1.1.4.4.2：查看容器中的 PID 信息：

```
[root@node1 ~]# ps -ef | grep 5492 ← 查看某个容器的PID信息
root  5492  5061  0 Jun30 ?      00:00:06 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/5a7c6090b8adae677c8cd176f9130ce83d27e60093f9424cbe9a99c86d69c690 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root  5510  5492  0 Jun30 ?      00:00:00 nginx: master process nginx -g daemon off;
root  27932  5492  0 14:56 pts/1   00:00:00 sh
root  27990  21259  0 14:56 pts/1   00:00:00 grep --color=auto 5492
[root@node1 ~]# docker exec -it 5a7c6090b8adae677c8c sh ← 进入到该容器中
#
# ps -ef | grep nginx ← 查看Nginx的进程信息
root     1     0  0 Jun30 ?      00:00:00 nginx: master process nginx -g daemon off;          Nginx的主进程PID为1，即容器的守护进程
nginx    6     1  0 Jun30 ?      00:00:00 nginx: worker process
root  3869  3863  0 06:56 pts/0   00:00:00 grep nginx ← Nginx工作进程
#
# top -n 1 ← 查看进程信息
top - 06:57:04 up 20:44,  0 users,  load average: 0.06, 0.03, 0.05
Tasks:  5 total,  1 running,  4 sleeping,  0 stopped,  0 zombie
%Cpu(s): 0.1 us, 0.2 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 1868692 total, 485620 free, 316580 used, 1066492 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 1392744 avail Mem          Nginx的主进程被作为容器的守护进程
#
# PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
# 1 root 20 0 32648 3244 2436 S 0.0 0.2 0:00.07 nginx
# 6 nginx 20 0 33104 1600 340 S 0.0 0.1 0:00.00 nginx
# 3855 root 20 0 4272 624 540 S 0.0 0.0 0:00.02 sh
# 3863 root 20 0 4272 624 540 S 0.0 0.0 0:00.02 sh
# 3870 root 20 0 41032 1780 1316 R 0.0 0.1 0:00.00 top          Nginx的工作进程，PID为6
```

1.1.4.5: Net Namespace:

每一个容器都类似于虚拟机一样有自己的网卡、监听端口、TCP/IP 协议栈等，Docker 使用 network namespace 启动一个 vethX 接口，这样你的容器将拥有它自己的桥接 ip 地址，通常是 docker0，而 docker0 实质就是 Linux 的虚拟网桥，网桥是在 OSI 七层模型的数据链路层的网络设备，通过 mac 地址对网络进行划分，并且在不同网络直接传递数据。

1.1.4.5.1：查看宿主机的网卡信息：

```
veth1c35fa8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::6ca7:c0ff:fe34:b180 prefixlen 64 scopeid 0x20<link>
        ether 6e:a7:c0:34:b1:80 txqueuelen 0 (Ethernet)
            RX packets 1973 bytes 118913 (116.1 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 2681 bytes 13127422 (12.5 MiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth2d4714c: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::7cd9:27ff:fec9:a979 prefixlen 64 scopeid 0x20<link>
        ether 7e:d9:27:c9:a9:79 txqueuelen 0 (Ethernet)
            RX packets 0 bytes 0 (0.0 B)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 8 bytes 648 (648.0 B)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth6a33bc3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::480a:19ff:fee3:9924 prefixlen 64 scopeid 0x20<link>
        ether 4a:0a:19:e3:99:24 txqueuelen 0 (Ethernet)
            RX packets 0 bytes 0 (0.0 B)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 8 bytes 648 (648.0 B)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

1.1.4.5.3：查看宿主机桥接设备：

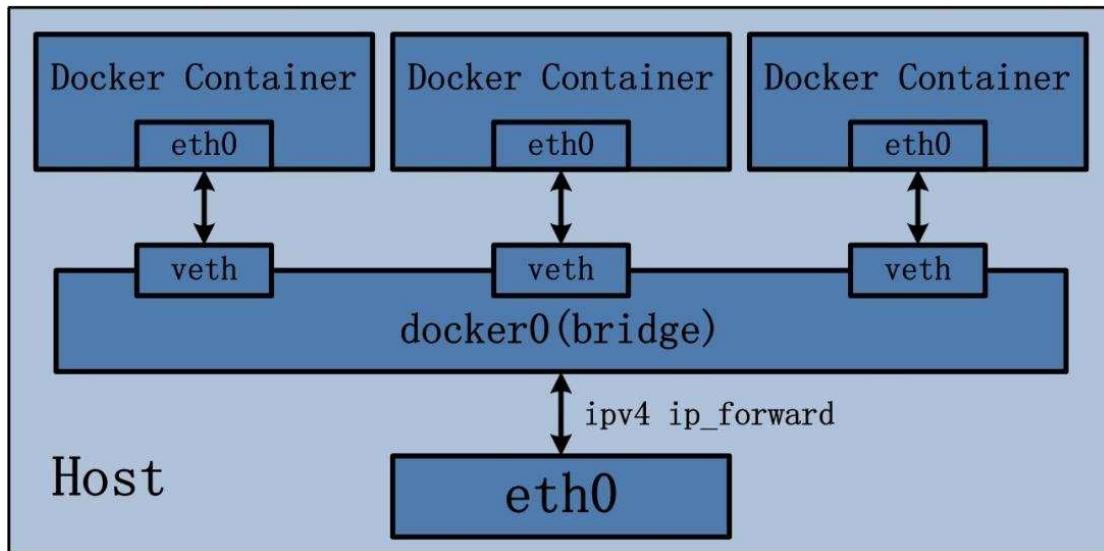
通过 `brctl show` 命令查看桥接设备。

```
[root@node1 ~]# docker exec -it c981ef6a234c sh
# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.4 netmask 255.255.0.0 broadcast 172.17.255.255
        ether 02:42:ac:11:00:04 txqueuelen 0 (Ethernet)
        RX packets 1757 bytes 9262963 (8.8 MiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 1232 bytes 72606 (70.9 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        loop txqueuelen 0 (Local Loopback)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

#
```

1.1.4.5.3: 实逻辑网络图:



1.1.4.5.4: 宿主机 iptables 规则:

```
[root@node1 ~]# iptables -t nat -vnL
Chain PREROUTING (policy ACCEPT 135 packets, 20914 bytes)
pkts bytes target  prot opt in     out    source               destination
  2   104 DOCKER  all  --  *      *      0.0.0.0/0          0.0.0.0/0           ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 38 packets, 8254 bytes)
pkts bytes target  prot opt in     out    source               destination

Chain OUTPUT (policy ACCEPT 105 packets, 6172 bytes)
pkts bytes target  prot opt in     out    source               destination
  0     0 DOCKER  all  --  *      *      0.0.0.0/0          !127.0.0.0/8         ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 105 packets, 6172 bytes)
pkts bytes target  prot opt in     out    source               destination
  80  5087 MASQUERADE  all  --  *      !docker0  172.17.0.0/16  0.0.0.0/0           源地址转换, 让容器通过宿主机地访问外网
    0     0 MASQUERADE  tcp   --  *      *      172.17.0.2       172.17.0.2          tcp dpt:80
    0     0 MASQUERADE  tcp   --  *      *      172.17.0.3       172.17.0.3          tcp dpt:80
    0     0 MASQUERADE  tcp   --  *      *      172.17.0.4       172.17.0.4          tcp dpt:80

Chain DOCKER (2 references)
pkts bytes target  prot opt in     out    source               destination
  0     0 RETURN   all  --  docker0 *      0.0.0.0/0          0.0.0.0/0           目的地址转换, 以实现从外宿主机部访问容器
  0     0 DNAT     tcp   --  !docker0 *      0.0.0.0/0          0.0.0.0/0           tcp dpt:80 to:172.17.0.2:80
  0     0 DNAT     tcp   --  !docker0 *      0.0.0.0/0          0.0.0.0/0           tcp dpt:81 to:172.17.0.3:80
  0     0 DNAT     tcp   --  !docker0 *      0.0.0.0/0          0.0.0.0/0           tcp dpt:82 to:172.17.0.4:80

[root@node1 ~]#
```

```
[root@node1 ~]# iptables -vnL
Chain INPUT (policy ACCEPT 5149 packets, 528K bytes)
pkts bytes target prot opt in     out      source          destination
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target prot opt in     out      source          destination
10160 29M DOCKER-USER all -- *    *      0.0.0.0/0      0.0.0.0/0
10160 29M DOCKER-ISOLATION-STAGE-1 all -- *    *      0.0.0.0/0      0.0.0.0/0
5953  29M ACCEPT   all -- *    docker0  0.0.0.0/0      0.0.0.0/0      ctstate RELATED,ESTABLISHED
0     0 DOCKER    all -- *    docker0  0.0.0.0/0      0.0.0.0/0
4207 194K ACCEPT   all -- docker0 !docker0 0.0.0.0/0      0.0.0.0/0
0     0 ACCEPT    all -- docker0 docker0  0.0.0.0/0      0.0.0.0/0
Chain OUTPUT (policy ACCEPT 4619 packets, 642K bytes)
pkts bytes target prot opt in     out      source          destination
Chain DOCKER (1 references)
pkts bytes target prot opt in     out      source          destination
0     0 ACCEPT   tcp  -- !docker0 docker0  0.0.0.0/0      172.17.0.2      tcp dpt:80
0     0 ACCEPT   tcp  -- !docker0 docker0  0.0.0.0/0      172.17.0.3      tcp dpt:80
0     0 ACCEPT   tcp  -- !docker0 docker0  0.0.0.0/0      172.17.0.4      tcp dpt:80
Chain DOCKER-ISOLATION-STAGE-1 (1 references)
pkts bytes target prot opt in     out      source          destination
4207 194K DOCKER-ISOLATION-STAGE-2 all -- docker0 !docker0 0.0.0.0/0      0.0.0.0/0
10160 29M RETURN  all -- *    *      0.0.0.0/0      0.0.0.0/0
Chain DOCKER-ISOLATION-STAGE-2 (1 references)
pkts bytes target prot opt in     out      source          destination
0     0 DROP     all -- *    docker0  0.0.0.0/0      0.0.0.0/0
4207 194K RETURN  all -- *    *      0.0.0.0/0      0.0.0.0/0
Chain DOCKER-USER (1 references)
pkts bytes target prot opt in     out      source          destination
10160 29M RETURN  all -- *    *      0.0.0.0/0      0.0.0.0/0
[root@node1 ~]#
```

1.1.4.6: User Namespace:

各个容器内可能会出现重名的用户和用户组名称，或重复的用户 UID 或者 GID，那么怎么隔离各个容器内的用户空间呢？

User Namespace 允许在各个宿主机的各个容器空间内创建相同的用户名以及相同的用户 UID 和 GID，只是会把用户的作用范围限制在每个容器内，即 A 容器和 B 容器可以有相同的用户名和 ID 的账户，但是此用户的有效范围仅是当前容器内，不能访问另外一个容器内的文件系统，即相互隔离、互补影响、永不相见。

```
root@docker-node1:~# docker run -it --rm nginx bash
root@23ff62ed080b:/# id ← 每个容器内都有超级管理员root及其他系统账户，且账户ID与其他容器相同
uid=0(root) gid=0(root) groups=0(root)
root@23ff62ed080b:/# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
nginx:x:101:101:nginx user,,,:/nonexistent:/bin/false
root@23ff62ed080b:/#
```

1.1.5: Linux control groups:

在一个容器，如果不对其做任何资源限制，则宿主机会允许其占用无限大的内存空间，有时候会因为代码 bug 程序会一直申请内存，直到把宿主机内存占完，为了避免此类的问题出现，宿主机有必要对容器进行资源分配限制，比如 CPU、内存等，Linux Cgroups 的全称是 Linux Control Groups，它最主要的作用，就是限制一个进程组能够使用的资源上限，包括 CPU、内存、磁盘、网络带宽等等。此外，还能够对进程进行优先级设置，以及将进程挂起和恢复等操作。

1.1.5.1: 验证系统 cgroups:

Cgroups 在内核层默认已经开启，从 centos 和 ubuntu 对比结果来看，显然内核较新的 ubuntu 支持的功能更多。

1.1.5.1.1: Centos 7.6 cgroups:

```
[root@linux-node4 ~]# cat /etc/redhat-release
CentOS Linux release 7.6.1810 (Core)
[root@linux-node4 ~]# uname -r
3.10.0-957.el7.x86_64
[root@linux-node4 ~]# cat /boot/config-3.10.0-957.el7.x86_64 | grep CGROUP
CONFIG_CGROUPS=y
# CONFIG_CGROUP_DEBUG is not set
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_CGROUP_HUGETLB=y
CONFIG_CGROUP_PERF=y
CONFIG_CGROUP_SCHED=y
CONFIG_BLK_CGROUP=y
# CONFIG_DEBUG_BLK_CGROUP is not set
CONFIG_NETFILTER_XT_MATCH_CGROUP=m
CONFIG_NET_CLS_CGROUP=y
CONFIG_NETPRIORITY_CGROUP=y
[root@linux-node4 ~]#
```

1.1.5.1.2: ubuntu cgroups:

```
root@s1:~# cat /etc/issue
Ubuntu 18.04 LTS \n \l

root@s1:~# uname -r
4.15.0-20-generic
root@s1:~# cat /boot/config-4.15.0-20-generic | grep CGROUP
CONFIG_CGROUPS=y
CONFIG_BLK_CGROUP=y
# CONFIG_DEBUG_BLK_CGROUP is not set
CONFIG_CGROUP_WRITEBACK=y
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_RDMA=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_HUGETLB=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_CGROUP_PERF=y
CONFIG_CGROUP_BPF=y
# CONFIG_CGROUP_DEBUG is not set
CONFIG_SOCK_CGROUP_DATA=y
CONFIG_NETFILTER_XT_MATCH_CGROUP=m
CONFIG_NET_CLS_CGROUP=m
CONFIG_CGROUP_NET_PRIO=y
CONFIG_CGROUP_NET_CLASSID=y
root@s1:~#
```

1.1.5.1.3: cgroups 中内存模块:

```
# cat /boot/config-4.15.0-54-generic | grep MEM | grep CG
CONFIG_MEMCG=y
CONFIG_MEMCG_SWAP=y
# CONFIG_MEMCG_SWAP_ENABLED is not set
CONFIG_SLUB_MEMCG_SYSFS_ON=y
```

1.1.5.1.4: cgroups 具体实现:

blkio: 块设备 IO 限制。

cpu: 使用调度程序为 cgroup 任务提供 cpu 的访问。

cpuacct: 产生 cgroup 任务的 cpu 资源报告。

cpuset: 如果是多核心的 cpu, 这个子系统会为 cgroup 任务分配单独的 cpu 和内存。

devices: 允许或拒绝 cgroup 任务对设备的访问。

freezer: 暂停和恢复 cgroup 任务。

memory: 设置每个 cgroup 的内存限制以及产生内存资源报告。

net_cls: 标记每个网络包以供 cgroup 方便使用。

ns: 命名空间子系统。

perf_event: 增加了对每 group 的监测跟踪的能力, 可以监测属于某个特定的 group 的所有线程以及运行在特定 CPU 上的线程。

1.1.5.1.5: 查看系统 cgroups:

```
[root@node1 ~]# ll      /sys/fs/cgroup/
total 0
drwxr-xr-x 5 root root  0 Jun 30 18:12 blkio
lrwxrwxrwx 1 root root 11 Jun 30 18:12 cpu -> cpu,cpuacct
lrwxrwxrwx 1 root root 11 Jun 30 18:12 cpuacct -> cpu,cpuacct
drwxr-xr-x 5 root root  0 Jun 30 18:12 cpu,cpuacct
drwxr-xr-x 3 root root  0 Jun 30 18:12 cpuset
drwxr-xr-x 5 root root  0 Jun 30 18:12 devices
drwxr-xr-x 3 root root  0 Jun 30 18:12 freezer
drwxr-xr-x 3 root root  0 Jun 30 18:12 hugetlb
drwxr-xr-x 5 root root  0 Jun 30 18:12 memory
drwxr-xr-x 3 root root  0 Jun 30 18:12 net_cls
drwxr-xr-x 3 root root  0 Jun 30 18:12 perf_event
drwxr-xr-x 5 root root  0 Jun 30 18:12 systemd
```

有了以上的 chroot、namespace、cgroups 就具备了基础的容器运行环境，但是还需要有相应的容器创建与删除的管理工具、以及怎么样把容器运行起来、容器数据怎么处理、怎么进行启动与关闭等问题需要解决，于是容器管理技术出现了。

1.1.6: 容器管理工具:

目前主要是使用 docker，早期有使用 lxc。

1.1.6.1: lxc:

LXC: LXC 为 Linux Container 的简写。可以提供轻量级的虚拟化，以便隔离进程和资源，官方网站：<https://linuxcontainers.org/>

Ubuntu 安装 lxc:

```
root@s1:~# apt install lxc lxd
Reading package lists... Done
Building dependency tree
Reading state information... Done
lxd is already the newest version (3.0.3-0ubuntu1~18.04.1).
lxc is already the newest version (3.0.3-0ubuntu1~18.04.1).
root@s1:~# lxc-checkconfig #检查内核对 lcx 的支持状况，必须全部为 lcx
root@s1:~# lxc-create -t 模板名称 -n lcx-test
root@s1:~# lxc-create -t download --name alpine12 -- --dist alpine --release
3.9 --arch amd64
Setting up the GPG keyring
Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs
```

```
--  
You just created an Alpinelinux 3.9 x86_64 (20190630_13:00) container.  
  
root@s1:~# lxc-start alpine12 #启动 lxc 容器  
root@s1:~# lxc-attach alpine12 #进入 lxc 容器  
~ # ifconfig  
eth0      Link encap:Ethernet HWaddr 00:16:3E:DF:54:94  
          inet addr:10.0.3.115 Bcast:10.0.3.255 Mask:255.255.255.0  
          inet6 addr: fe80::216:3eff:fedf:5494/64 Scope:Link  
             UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1  
             RX packets:18 errors:0 dropped:0 overruns:0 frame:0  
             TX packets:13 errors:0 dropped:0 overruns:0 carrier:0  
             collisions:0 txqueuelen:1000  
             RX bytes:2102 (2.0 KiB) TX bytes:1796 (1.7 KiB)  
  
lo       Link encap:Local Loopback  
          inet addr:127.0.0.1 Mask:255.0.0.0  
          inet6 addr: ::1/128 Scope:Host  
             UP LOOPBACK RUNNING MTU:65536 Metric:1  
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
             collisions:0 txqueuelen:1000  
             RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)  
  
~ # uname -a  
Linux alpine12 4.15.0-20-generic #21-Ubuntu SMP Tue Apr 24 06:16:15 UTC 2018  
x86_64 Linux  
~ # cat /etc/issue  
Welcome to Alpine Linux 3.9  
Kernel \r on an \m (\l)
```

命令备注：

-t 模板: -t 选项后面跟的是模板，模式可以认为是一个原型，用来说明我们需要一个什么样的容器(比如容器里面需不需要有 vim, apache 等软件). 模板实际上就是一个脚本文件(位于/usr/share/lxc/templates 目录)，我们这里指定 download 模板(lxc-create 会调用 lxc-download 脚本，该脚本位于刚说的模板目录中)是说明我们目前没有自己模板，需要下载官方的模板

--name 容器名称： 为创建的容器命名

-- :--用来说明后面的参数是传递给 download 脚本的，告诉脚本需要下载什么样的模板

--dist 操作系统名称： 指定操作系统

--release 操作系统： 指定操作系统，可以是各种 Linux 的变种

--arch 架构：指定架构，是 x86 还是 arm，是 32 位还是 64 位

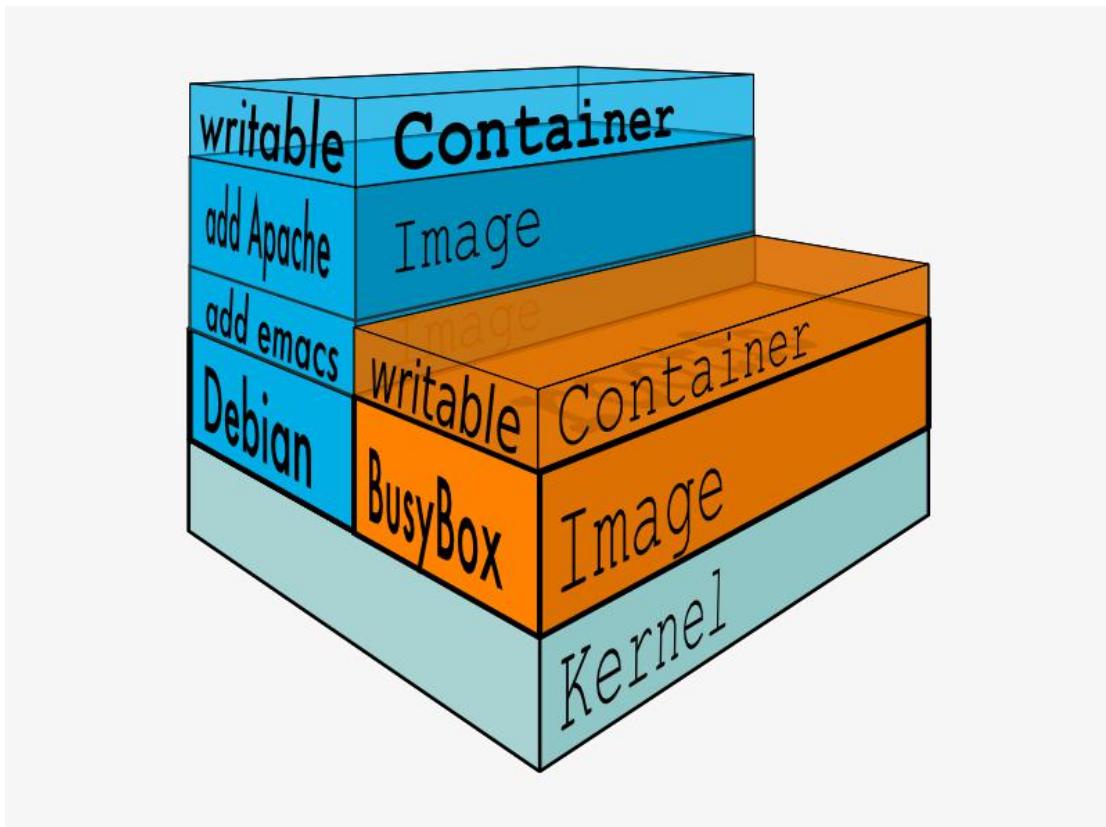
lxc 启动容器依赖于模板，清华模板源：

<https://mirrors.tuna.tsinghua.edu.cn/help/lxc-images/>，但是做模板相对较难，需要手动一步步创构文件系统、准备基础目录及可执行程序等，而且在大规模使用容器的场景很难横向扩展，另外后期代码升级也需要重新从头构建模板，基于以上种种原因便有了 docker。

1.1.6.2: docker:

Docker 启动一个容器也需要一个外部模板但是较多镜像，docke 的镜像可以保存在一个公共的地方共享使用，只要把镜像下载下来就可以使用，最主要的是可以在镜像基础之上做自定义配置并且可以再把其提交为一个镜像，一个镜像可以被启动为多个容器。

Docker 的镜像是分层的，镜像底层为库文件且只读层即不能写入也不能删除数据，从镜像加载启动为一个容器后会生成一个可写层，其写入的数据会复制到容器目录，但是容器内的数据在删除容器后也会被随之删除。



1.1.6.3: pouch:

<https://www.infoq.cn/article/alibaba-pouch>

<https://github.com/alibaba/pouch>

1.1.6: Docker 的优势:

快速部署：短时间内可以部署成百上千个应用，更快速交付到线上。

高效虚拟化：不需要额外的 hypervisor 支持，直接基于 linux 实现应用虚拟化，相比虚拟机大幅提高性能和效率。

节省开支：提高服务器利用率，降低 IT 支出。

简化配置：将运行环境打包保存至容器，使用时直接启动即可。

快速迁移和扩展：可在平台运行在物理机、虚拟机、公有云等环境，良好的兼容性可以方便将应用从 A 宿主机迁移到 B 宿主机，甚至是 A 平台迁移到 B 平台。

1.1.7: Docker 的缺点:

隔离性：各应用之间的隔离不如虚拟机彻底。

1.1.8: docker(容器)的核心技术:

1.1.8.1:容器规范:

容器技术除了的 docker 之外，还有 coreOS 的 rkt，还有阿里的 Pouch，为了保证容器生态的标准性和健康可持续发展，包括 Linux 基金会、Docker、微软、红帽谷歌和、IBM、等公司在 2015 年 6 月共同成立了一个叫 open container (OCI) 的组织，其目的就是制定开放的标准的容器规范，目前 OCI 一共发布了两个规范，分别是 **runtime spec** 和 **image format spec**，有了这两个规范，不同的容器公司开发的容器只要兼容这两个规范，就可以保证容器的可移植性和相互可操作性。

1.1.8.1.1:容器 runtime(runtime spec):

runtime 是真正运行容器的地方，因此为了运行不同的容器 runtime 需要和操作系统内核紧密合作相互支持，以便为容器提供相应的运行环境。

目前主流的三种 runtime:

Lxc: linux 上早期的 runtime，Docker 早期就是采用 lxc 作为 runtime。

runc: 目前 Docker 默认的 runtime，runc 遵守 OCI 规范，因此可以兼容 lxc。

rkt: 是 CoreOS 开发的容器 runtime，也符合 OCI 规范，所以使用 rktruntime 也可以运行 Docker 容器。

runtime 主要定义了以下规范，并以 json 格式保存在 /run/docker/runtime-runc/moby/容器 ID/state.json 文件，此文件会根据容器的状态实时更新内容：

版本信息：存放 OCI 标准的具体版本号。

容器 ID：通常是一个哈希值，可以在所有 state.json 文件中提取出容器 ID 对容器进行批量操作（关闭、删除等），此文件在容器关闭后会被删除，容器启动后会自动生成。

PID：在容器中运行的首个进程在宿主机上的进程号，即将宿主机的那个进程设置为容器的守护进程。

容器文件目录：存放容器 rootfs 及相应配置的目录，外部程序只需读取 state.json 就可以定位到宿主机上的容器文件目录。

容器创建：创建包括文件系统、namespaces、cgroups、用户权限在内的各项内容。

容器进程的启动：运行容器启动进程，该文件在

/run/containerd/io.containerd.runtime.v1.linux/moby/容器 ID/config.json。

容器生命周期：容器进程可以被外部程序关停，runtime 规范定义了对容器操作信号的捕获，并做相应资源回收的处理，避免僵尸进程的出现。

1.1.8.1.2: 容器镜像(image format spec):

OCI 容器镜像主要包含以下内容：

文件系统：定义以 layer 保存的文件系统，在镜像里面是 layer.tar，每个 layer 保存了和上层之间变化的部分，image format spec 定义了 layer 应该保存哪些文件，怎么表示增加、修改和删除的文件等操作。

manifest 文件：描述有哪些 layer，tag 标签及 config 文件名称。

config 文件：是一个以 hash 命名的 json 文件，保存了镜像平台，容器运行时容器运行时需要的一些信息，比如环境变量、工作目录、命令参数等。

index 文件：可选的文件，指向不同平台的 manifest 文件，这个文件能保证一个镜像可以跨平台使用，每个平台拥有不同的 manifest 文件使用 index 作为索引。

父镜像：大多数层的元信息结构都包含一个 parent 字段，指向该镜像的父镜像。

参数：

ID：镜像 ID，每一层都有 ID

tag 标签：标签用于将用户指定的、具有描述性的名称对应到镜像 ID

仓库：Repository 镜像仓库

os：定义类型

architecture：定义 CPU 架构

author：作者信息

create：镜像创建日期

1.1.8.2: 容器管理工具:

管理工具连接 runtime 与用户，对用户提供图形或命令方式操作，然后管理工具将用户操作传递给 runtime 执行。

lxc 是 lxd 的管理工具。

Runc 的管理工具是 docker engine，docker engine 包含后台 deamon 和 cli 两部分，大家经常提到的 Docker 就是指的 docker engine。

Rkt 的管理工具是 rkt cli。

1.1.8.3: 容器定义工具:

容器定义工具允许用户定义容器的属性和内容，以方便容器能够被保存、共享和重建。

Docker image: 是 docker 容器的模板，runtime 依据 docker image 创建容器。

Dockerfile: 包含 N 个命令的文本文件，通过 Dockerfile 创建出 docker image。

ACI(App container image): 与 docker image 类似，是 CoreOS 开发的 rkt 容器的镜像格式。

1.1.8.4: Registry:

统一保存镜像而且是多个不同镜像版本的地方，叫做镜像仓库。

Image registry: docker 官方提供的私有仓库部署工具。

Docker hub: docker 官方的公共仓库，已经保存了大量的常用镜像，可以方便大家直接使用。

Harbor: vmware 提供的自带 web 界面自带认证功能的镜像仓库，目前有很多公司使用。

```
172.18.200.101/project/centos:7.2.1511  
172.18.200.101/project/centos: latest  
172.18.200.101/project/java-7.0.59:v1  
172.18.200.101/project/java-7.0.59:v2
```

1.1.8.5: 编排工具:

当多个容器在多个主机运行的时候，单独管理容器是相当复杂而且很容易出错，而且也无法实现某一台主机宕机后容器自动迁移到其他主机从而实现高可用的目的，也无法实现动态伸缩的功能，因此需要有一种工具可以实现统一管理、动态伸缩、故障自愈、批量执行等功能，这就是容器编排引擎。

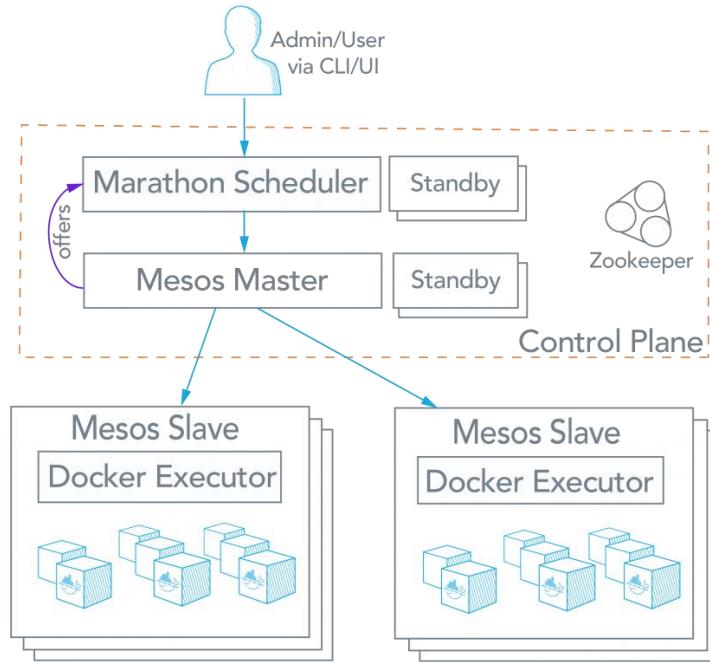
容器编排通常包括容器管理、调度、集群定义和服务发现等功能。

Docker swarm: docker 开发的容器编排引擎。

Kubernetes: google 领导开发的容器编排引擎，内部项目为 Borg，且其同时支持

docker 和 CoreOS。

Mesos+Marathon: 通用的集群组员调度平台，mesos(资源分配)与 marathon(容器编排平台)一起提供容器编排引擎功能。



Mesos Master: manages resources in cluster. Provides offers to Marathon
Mesos Slave: runs agents which report resources to master
Offer: a list of available CPU and memory resources for slave nodes
Standby: activated if current masters for Mesos/Marathon fail

Marathon Scheduler: registers with Mesos master to receive offers
Docker Executor: executes tasks from Marathon scheduler
Zookeeper: enables high availability of Mesos and Marathon

1.1.9: docker(容器)的依赖技术:

1.1.9.1: 容器网络:

docker 自带的网络 docker network 仅支持管理单机上的容器网络，当多主机运行的时候需要使用第三方开源网络，例如 calico、flannel 等。

1.1.9.2: 服务发现:

容器的动态扩容特性决定了容器 IP 也会随之变化，因此需要有一种机制可以自动识别并将用户请求动态转发到新创建的容器上，kubernetes 自带服务发现功能，需要结合 kube-dns 服务解析内部域名。

1.1.9.3: 容器监控:

可以通过原生命令 docker ps/top/stats 查看容器运行状态，另外也可以使用 heapster/Prometheus 等第三方监控工具监控容器的运行状态。

1.1.9.4: 数据管理:

容器的动态迁移会导致其在不同的 Host 之间迁移，因此如何保证与容器相关的数据也能随之迁移或随时访问，可以使用逻辑卷/存储挂载等方式解决。

1.1.9.5: 日志收集:

docker 原生的日志查看工具 docker logs，但是容器内部的日志需要通过 ELK 等专门的日志收集分析和展示工具进行处理。

1.2: Docker 安装及基础命令介绍:

官方网址: <https://www.docker.com/>

系统版本选择:

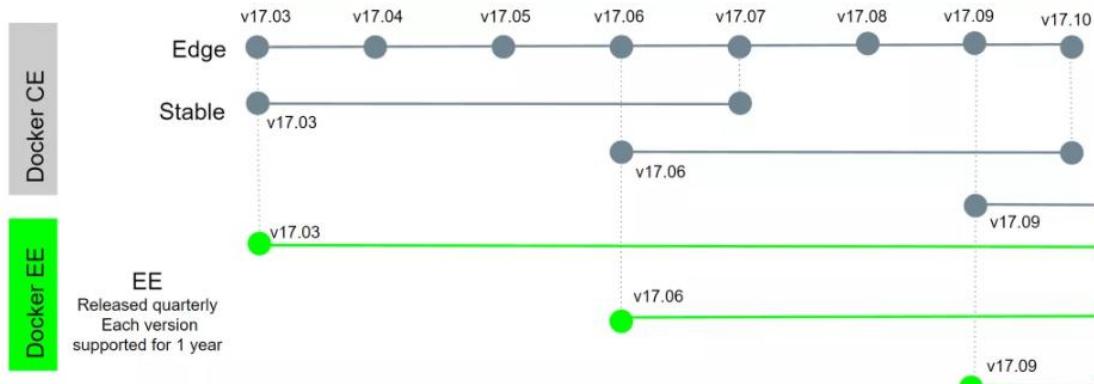
Docker 目前已经支持多种操作系统的安装运行，比如 Ubuntu、CentOS、Redhat、Debian、Fedora，甚至是还支持了 Mac 和 Windows，在 linux 系统上需要内核版本在 3.10 或以上，docker 版本号之前一直是 0.X 版本或 1.X 版本，但是从 2017 年 3 月 1 号开始改为每个季度发布一次稳版，其版本号规则也统一变更为 YY.MM，例如 17.09 表示是 2017 年 9 月份发布的，本次演示的操作系统使用 Centos 7.5 为例。

Docker 版本选择:

Docker 之前没有区分版本，但是 2017 年初推出(将 docker 更名为)新的项目 Moby，github 地址: <https://github.com/moby/moby>，Moby 项目属于 Docker 项目的全新上游，Docker 将是一个隶属于的 Moby 的子产品，而且之后的版本之后开始区分为 CE 版本（社区版本）和 EE（企业收费版），CE 社区版本和 EE 企业版本都是每个季度发布一个新版本，但是 EE 版本提供后期安全维护 1 年，而 CE 版本是 4 个月，本次演示的 Docker 版本为 18.03，以下为官方原文：

<https://blog.docker.com/2017/03/docker-enterprise-edition/>

Docker CE and EE are released quarterly, and CE also has a monthly “Edge” option. Each Docker EE release is supported and maintained for one year and receives security and critical bugfixes during that period. We are also improving Docker CE maintainability by maintaining each quarterly CE release for 4 months. That gets Docker CE users a new 1-month window to update from one version to the next.



与 kubernetes 结合使用的时候，要安装经过 kubernetes 官方测试通过的 docker 版本，避免出现不兼容等未知的及不可预估的问题发生，kubernetes 测试过的 docker 版本可以在 github 查询，具体如下：

<https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG-1.14.md#external-dependencies>

GitHub, Inc. [US] | github.com/kubernetes/kubernetes/blob/master/CHANGELOG-1.14.md#external-dependencies

External Dependencies

- Default etcd server and client have been updated to v3.3.10. (#71615, #70168)
- The list of validated docker versions has changed. 1.11.1 and 1.12.1 have been removed. The current list is 1.13.1, 17.03, 17.06, 17.09, 18.06, 18.09. (#72823, #72831)
- The default Go version was updated to 1.12.1. (#75422)
- CNI has been updated to v0.7.5 (#75455)
- CSI has been updated to v1.1.0. (#75391)

1.2.1：下载 rpm 包安装：

官方 rpm 包下载地址：

https://download.docker.com/linux/centos/7/x86_64/stable/Packages/

二进制下载地址：

<https://download.docker.com/>

https://mirrors.aliyun.com/docker-ce/linux/static/stable/x86_64/

阿里镜像下载地址：

https://mirrors.aliyun.com/docker-ce/linux/centos/7/x86_64/stable/Packages/

```
[root@docker-server1 ~]# yum install /usr/local/src/docker-ce-18.03.1.ce-1.el7.centos.x86_64.rpm
Loaded plugins: fastestmirror
Examining /usr/local/src/docker-ce-18.03.1.ce-1.el7.centos.x86_64.rpm: docker-ce-18.03.1.ce-1.el7.centos.x86_64
Marking /usr/local/src/docker-ce-18.03.1.ce-1.el7.centos.x86_64.rpm to be installed
Resolving Dependencies
--> Running transaction check
--> Package docker-ce.x86_64 0:18.03.1.ce-1.el7.centos will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package           Arch      Version            Repository
=====
Installing:
docker-ce        x86_64   18.03.1.ce-1.el7.centos   /docker-ce-18.03.1.ce-1.el7.centos.x86_64

Transaction Summary
=====
Install 1 Package

Total size: 151 M
Installed size: 151 M
Is this ok [y/d/N]:
```

1.2.2：通过修改 yum 源安装：

```
[root@docker-server1 ~]# rm -rf /etc/yum.repos.d/*
[root@docker-server1 ~]# wget -O /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo
```

```
[root@docker-server1 ~]# wget -O /etc/yum.repos.d/epel.repo  
http://mirrors.aliyun.com/repo/epel-7.repo  
[root@docker-server1 ~]# wget -O /etc/yum.repos.d/docker-ce.repo  
https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
```

```
[root@docker-server1 ~]# yum install docker-ce  
[root@docker-server1 ~]# yum install docker-ce  
Loaded plugins: fastestmirror  
Loading mirror speeds from cached hostfile  
* base: mirrors.aliyun.com  
* extras: mirrors.aliyun.com  
* updates: mirrors.aliyun.com  
Resolving Dependencies  
--> Running transaction check  
--> Package docker-ce.x86_64 0:18.03.1.ce-1.el7.centos will be installed  
--> Finished Dependency Resolution  
  
Dependencies Resolved  
  
=====  
           Package          Arch        Version  
=====  
Installing:  
  docker-ce           x86_64      18.03.1.ce-1.el7.centos  
  
Transaction Summary  
=====  
Install 1 Package  
  
Total download size: 35 M  
Installed size: 151 M  
Is this ok [y/d/N]: █
```

1.2:3: ubuntu 安装 docker、启动并验证服务：

```
root@docker-server1:~#apt-get install docker-ce=5:18.09.9~3-0~ubuntu-bionic  
docker-ce-cli=5:18.09.9~3-0~ubuntu-bionic  
root@docker-server1:~# systemctl start docker  
root@docker-server1:~# systemctl enable docker
```

验证 docker 服务启动：

```
root@docker-server1:~# systemctl enable docker  
Synchronizing state of docker.service with SysV service script with /lib/systemd/systemd-sysv-install.  
Executing: /lib/systemd/systemd-sysv-install enable docker  
root@docker-server1:~# systemctl status docker  
root@docker-server1:~# systemctl status docker  
● docker.service - Docker Application Container Engine  
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)  
   Active: active (running) since Tue 2019-09-10 11:30:38 CST; 1min 29s ago  
     Docs: https://docs.docker.com  
   Main PID: 658 (dockerd)  
     Tasks: 8  
    CGroup: /system.slice/docker.service  
           └─658 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock  
  
Sep 10 11:30:37 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:37.926047260+08:00" level=info msg="ccResolverWrapper: sending n  
Sep 10 11:30:37 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:37.926176137+08:00" level=info msg="ClientConn switching balanc  
Sep 10 11:30:37 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:37.926313495+08:00" level=info msg="pickfirstBalancer: HandleSub  
Sep 10 11:30:37 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:37.926668054+08:00" level=info msg="pickfirstBalancer: HandleSub  
Sep 10 11:30:38 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:38.254235175+08:00" level=info msg="Default bridge (docker0) is  
Sep 10 11:30:38 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:38.319915320+08:00" level=info msg="Loading containers: done."  
Sep 10 11:30:38 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:38.453279109+08:00" level=info msg="Docker daemon" commit=039a7d  
Sep 10 11:30:38 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:38.453384282+08:00" level=info msg="Daemon has completed initial  
Sep 10 11:30:38 docker-server1.magedu.net dockerd[658]: time="2019-09-10T11:30:38.503105424+08:00" level=info msg="API listen on /var/run/docke  
lines 1-19/19 (END)  
root@docker-server1:~# █
```

1.2.4: 验证 docker 版本:

```
root@docker-server1:~# docker version
Client:
  Version:          18.09.9
  API version:     1.39
  Go version:      go1.11.13
  Git commit:      039a7df9ba
  Built:           Wed Sep  4 16:57:28 2019
  OS/Arch:         linux/amd64
  Experimental:    false

Server: Docker Engine - Community
Engine:
  Version:          18.09.9
  API version:     1.39 (minimum version 1.12)
  Go version:      go1.11.13
  Git commit:      039a7df
  Built:           Wed Sep  4 16:19:38 2019
  OS/Arch:         linux/amd64
  Experimental:    false
root@docker-server1:~#
```

1.2.5: 验证 docker0 网卡:

在 docker 安装启动之后，默认会生成一个名称为 docker0 的网卡并且默认 IP 地址为 172.17.0.1 的网卡。

```
root@docker-server1:~# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
        inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
              ether 02:42:67:33:db:3e txqueuelen 0 (Ethernet)
              RX packets 0 bytes 0 (0.0 B)
              RX errors 0 dropped 0 overruns 0 frame 0
              TX packets 0 bytes 0 (0.0 B)
              TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.7.101 netmask 255.255.248.0 broadcast 192.168.7.255
        inet6 fe80::20c:29ff:fe44:8078 prefixlen 64 scopeid 0x20<link>
              ether 00:0c:29:44:80:78 txqueuelen 1000 (Ethernet)
              RX packets 342 bytes 216257 (216.2 KB)
              RX errors 0 dropped 0 overruns 0 frame 0
              TX packets 292 bytes 33970 (33.9 KB)
              TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

1.2.6: 验证 docker 信息:

```
root@docker-server1:~# docker info
Containers: 2 #当前主机运行的容器总数
Running: 1 #有几个容器是正在运行的
Paused: 0 #有几个容器是暂停的
```

```
Stopped: 1 #有几个容器是停止的
Images: 3 #当前服务器的镜像数
Server Version: 18.09.9 #服务端版本
Storage Driver: overlay2 #正在使用的存储引擎
    Backing Filesystem: xfs #后端文件系统, 即服务器的磁盘文件系统
    Supports d_type: true #是否支持 d_type
    Native Overlay Diff: true #是否支持差异数据存储
Logging Driver: json-file #日志类型
Cgroup Driver: cgroupfs #Cgroups 类型
Plugins: #插件
    Volume: local #卷
    Network: bridge host macvlan null overlay # overlay 夸主机通信
    Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog #日志类型
Swarm: inactive #是否支持 swarm
Runtimes: runc #已安装的容器运行时
Default Runtime: runc #默认使用的容器运行时
Init Binary: docker-init #初始化容器的守护进程, 即 pid 为 1 的进程
containerd version: 894b81a4b802e4eb2a91d1ce216b8817763c29fb #版本
runc version: 425e105d5a03fabd737a126ad93d62a9eeede87f # runc 版本
init version: fec3683 #init 版本
Security Options: #安全选项
    Apparmor #安全模块, https://docs.docker.com/engine/security/apparmor/
    seccomp #审计(操作), https://docs.docker.com/engine/security/seccomp/
    Profile: default #默认的配置文件
Kernel Version: 4.15.0-55-generic #宿主机内核版本
Operating System: Ubuntu 18.04.3 LTS #宿主机操作系统
OSType: linux #宿主机操作系统类型
Architecture: x86_64 #宿主机架构
CPUs: 1 #宿主机 CPU 数量
Total Memory: 1.924GiB #宿主机总内存
Name: docker-server1.magedu.net #宿主机 hostname
ID: ZFPD:UIA5:SR6E:Y6SS:52QL:5MPT:VDY3:ATVI:QMVG:HAFF:MN74:2HPD #宿主机 ID
Docker Root Dir: /var/lib/docker #宿主机数据保存目录
Debug Mode (client): false #client 端是否开启 debug
Debug Mode (server): false #server 端是否开启 debug
Registry: https://index.docker.io/v1/ #镜像仓库
Labels: #其他标签
Experimental: false #是否测试版
Insecure Registries: #非安全的镜像仓库
    127.0.0.0/8
Live Restore Enabled: false #是否开启活动重启(重启 docker-daemon 不关闭容器)
Product License: Community Engine #产品许可信息
```

```
WARNING: No swap limit support #系统警告信息(没有开启 swap 资源限制)
root@docker-server1:~#
```

1.2.7: 解决不支持 swap 限制警告:

```
root@docker-server1:~# vim /etc/default/grub
GRUB_DEFAULT=0
GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=2
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT=""
GRUB_CMDLINE_LINUX="net.ifnames=0 biosdevname=0 cgroup_enable=memory
swapaccount=1"
# update-grub
# reboot
```

1.2.8: docker 存储引擎:

目前 docker 的默认存储引擎为 overlay2, 不同的存储引擎需要相应的系统支持, 如需要磁盘分区的时候传递 `d-type` 文件分层功能, 即需要传递内核参数开启格式化磁盘的时候的指定功能。

历史更新信息:

<https://github.com/moby/moby/blob/master/CHANGELOG.md>

官方文档关于存储引擎的选择文档:

<https://docs.docker.com/storage/storagedriver/select-storage-driver/>

存储驱动类型:

AUFS (AnotherUnionFS) 是一种 Union FS, 是文件级的存储驱动。所谓 UnionFS 就是把不同物理位置的目录合并 mount 到同一个目录中。简单来说就是支持将不同目录挂载到同一个虚拟文件系统下的文件系统。这种文件系统可以一层一层地叠加修改文件。无论底下有多少层都是只读的, 只有最上层的文件系统是可写的。当需要修改一个文件时, AUFS 创建该文件的一个副本, 使用 CoW 将文件从只读层复制到可写层进行修改, 结果也保存在可写层。在 Docker 中, 底下的只读层就是 image, 可写层就是 Container, 是 Docker 18.06 及更早版本的首选存储驱动程序, 在内核 3.13 上运行 Ubuntu 14.04 时不支持 overlay2。

Overlay: 一种 Union FS 文件系统, Linux 内核 3.18 后支持。

overlay2: Overlay 的升级版, 到目前为止, 所有 Linux 发行版推荐使用的存储类型。devicemapper: 是 CentOS 和 RHEL 的推荐存储驱动程序, 因为之前的内核版本不支持 overlay2, 但是当前较新版本的 CentOS 和 RHEL 现在已经支持 overlay2, 因

此推荐使用 overlay2。

ZFS(Sun-2005)/btrfs(Oracle-2007): 目前没有广泛使用。

vfs: 用于测试环境，适用于无法使用 copy-on-write 文件系统的情况。此存储驱动程序的性能很差，通常不建议用于生产。

Docker 官方推荐首选存储引擎为 overlay2, devicemapper 存在使用空间方面的一些限制，虽然可以通过后期配置解决，但是官方依然推荐使用 overlay2，以下是从网上查到的部分资料：

<https://www.cnblogs.com/youruncloud/p/5736718.html>

```
[root@docker-server1 bin]# xfs_info /
meta-data=/dev/mapper/centos-root isize=512    agcount=4, agsize=6520320 blks
          =                      sectsz=512  attr=2, projid32bit=1
          =                      crc=1    finobt=0 spinodes=0
data     =                      bsize=4096   blocks=26081280, imaxpct=25
          =                      sunit=0    swidth=0 blks
naming   =version 2           bsize=4096   ascii-ci=0 ftype=1 ←
log      =internal            bsize=4096   blocks=12735, version=2
          =                      sectsz=512  sunit=0 blks, lazy-count=1
realtime =none                extsz=4096   blocks=0, rtextents=0
[root@docker-server1 bin]#
```

如果 docker 数据目录是一块单独的磁盘分区而且是 xfs 格式的，那么需要在格式化的时候加上参数-n ftype=1，否则后期在启动容器的时候会报错不支持 d-type。

```
[root@bj-zw-boss-cesupp-app-v-9-122 ~]# xfs_info /
meta-data=/dev/vda1              isize=256    agcount=31, agsize=524224 blks
          =                      sectsz=512  attr=2, projid32bit=1
          =                      crc=0    finobt=0
data     =                      bsize=4096   blocks=15727379, imaxpct=25
          =                      sunit=0    swidth=0 blks
naming   =version 2           bsize=4096   ascii-ci=0 ftype=0 ←
log      =internal            bsize=4096   blocks=2560, version=2
          =                      sectsz=512  sunit=0 blks, lazy-count=1
realtime =none                extsz=4096   blocks=0, rtextents=0
[root@bj-zw-boss-cesupp-app-v-9-122 ~]#
```

Centos 7.2 报错界面：

```
WARNING: overlay: the backing xfs filesystem is formatted [without d_type support], which leads to incorrect behavior.
Reformat the filesystem with ftype=1 to enable d_type support.
Running without d_type support will not be supported in future releases.
```

Centos 7.3 修复此问题：

```
[root@localhost ~]# cat /etc/redhat-release
CentOS Linux release 7.3.1611 (Core)
[root@localhost ~]#
[root@localhost ~]# uname -a
Linux localhost.localdomain 3.10.0-514.el7.x86_64 #1 SMP Tue Nov 22 16:42:41 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
[root@localhost ~]#
[root@localhost ~]# xfs_info /
meta-data=/dev/mapper/cl-root  isize=512    agcount=4, agsize=1277440 blks
                                =           sectsz=512   attr=2, projid32bit=1
                                =           crc=1     finobt=0 spinodes=0
data      =           bsize=4096   blocks=5109760, imaxpct=25
                                =           sunit=0   swidth=0 blks
naming    =version 2    bsize=4096   ascii-ci=0 ftype=1
log       =internal     bsize=4096   blocks=2560, version=2
                                =           sectsz=512   sunit=0 blks, lazy-count=1
realtime  =none         extsz=4096   blocks=0, rtextents=0
[root@localhost ~]#
```

1.2.9: docker 服务进程:

通过查看 docker 进程，了解 docker 的运行及工作方式

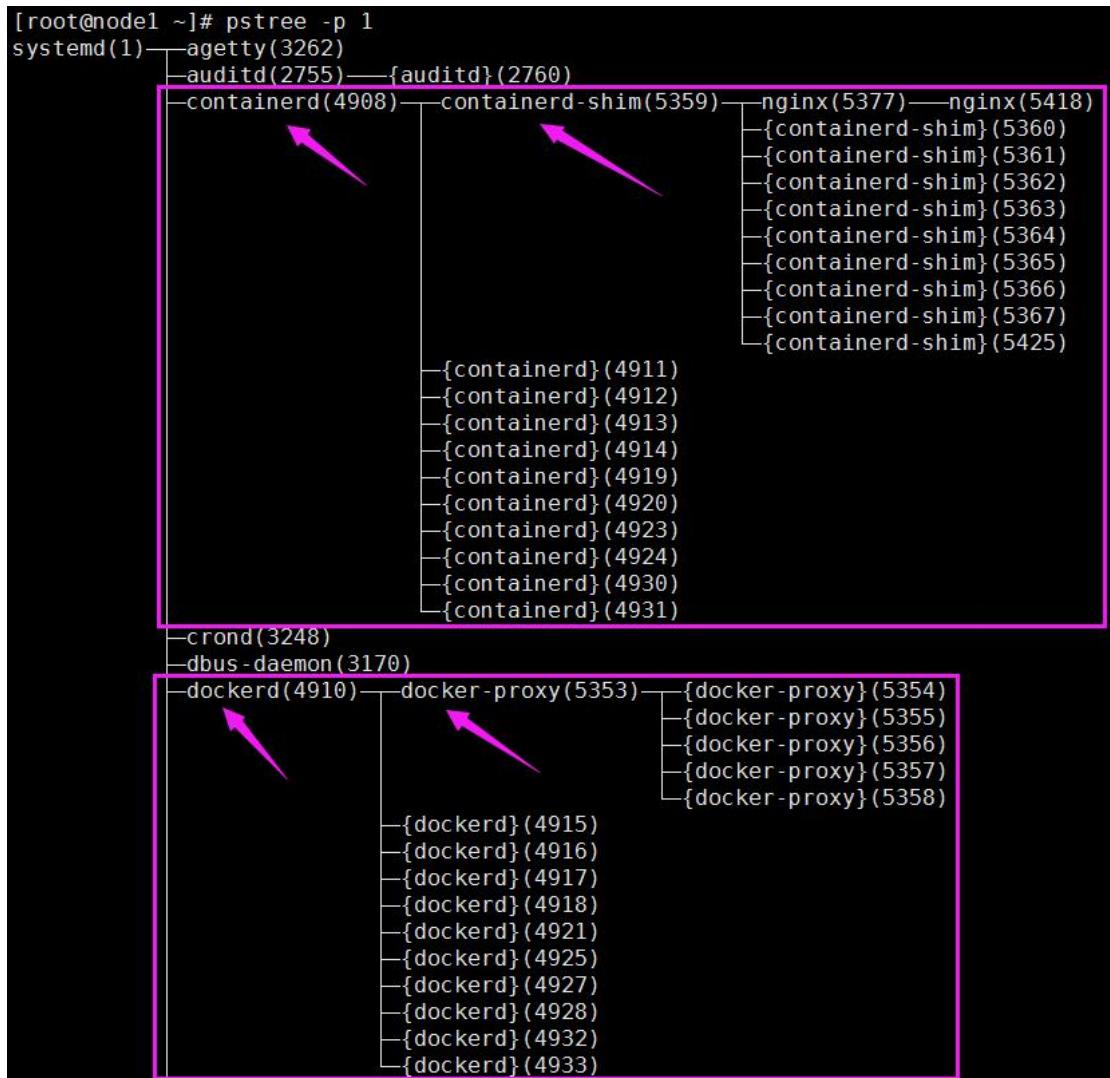
1.2.9.1: 查看宿主机进程树:

Docker 版本：

Server: Docker Engine - Community

Engine:

Version:	18.09.7
API version:	1.39 (minimum version 1.12)
Go version:	go1.10.8
Git commit:	2d0083d
Built:	Thu Jun 27 17:26:28 2019
OS/Arch:	linux/amd64
Experimental:	false



18.06 及之前的 docker 版本，进程关系：

```
[root@linux-node2 ~]# docker version
Client:
  Version:           18.06.3-ce
  API version:      1.38
  Go version:        go1.10.3
  Git commit:        d7080c1
  Built:             Wed Feb 20 02:26:51 2019
  OS/Arch:           linux/amd64
  Experimental:     false

Server:
  Engine:
    Version:          18.06.3-ce
    API version:      1.38 (minimum version 1.12)
    Go version:       go1.10.3
    Git commit:       d7080c1
    Built:            Wed Feb 20 02:28:17 2019
    OS/Arch:          linux/amd64
    Experimental:    false
[root@linux-node2 ~]# pstree -p 1
systemd(1)---audited(5427)---{audited}(5437)
          |   crond(6009)
          |   dbus-daemon(5935)
          |   dockerd(7324)---docker-containe(7331)---docker-containe(7535)---nginx(7554)---nginx(7588)
          |          |   {docker-containe}(7536)
          |          |   {docker-containe}(7537)
          |          |   {docker-containe}(7538)
          |          |   {docker-containe}(7539)
          |          |   {docker-containe}(7540)
          |          |   {docker-containe}(7541)
          |          |   {docker-containe}(7543)
          |          |   {docker-containe}(7544)
          |          |   {docker-containe}(7777)
          |          |   docker-containe(7627)---nginx(7646)---nginx(7682)
          |          |          |   {docker-containe}(7628)
          |          |          |   {docker-containe}(7629)
          |          |          |   {docker-containe}(7630)
          |          |          |   {docker-containe}(7631)
          |          |          |   {docker-containe}(7632)
          |          |          |   {docker-containe}(7633)
          |          |          |   {docker-containe}(7635)
          |          |          |   {docker-containe}(7636)
          |          |          |   {docker-containe}(7779)
          |          |          |   docker-containe(7716)---nginx(7736)---nginx(7768)
          |          |          |          |   {docker-containe}(7717)
          |          |          |          |   {docker-containe}(7718)
          |          |          |          |   {docker-containe}(7719)
          |          |          |          |   {docker-containe}(7720)
          |          |          |          |   {docker-containe}(7721)
          |          |          |          |   {docker-containe}(7722)
          |          |          |          |   {docker-containe}(7724)
          |          |          |          |   {docker-containe}(7728)
          |          |          |          |   {docker-containe}(7780)
          |          |          |          |   {docker-containe}(7332)
          |          |          |          |   {docker-containe}(7333)
          |          |          |          |   {docker-containe}(7334)
          |          |          |          |   {docker-containe}(7335)
          |          |          |          |   {docker-containe}(7336)
          |          |          |          |   {docker-containe}(7337)
          |          |          |          |   {docker-containe}(7340)
          |          |          |          |   {docker-containe}(7341)
          |          |          |          |   {docker-containe}(7454)
          |          |          |          |   {docker-containe}(7575)
          |          |          |          |   {docker-containe}(7576)
          |          |          |          |   {docker-containe}(7749)
          |          |          |   docker-proxy(7528)---{docker-proxy}(7529)
          |          |          |          |   {docker-proxy}(7530)
          |          |          |          |   {docker-proxy}(7531)
          |          |          |          |   {docker-proxy}(7532)
          |          |          |          |   {docker-proxy}(7533)
          |          |          |          |   {docker-proxy}(7534)
          |          |          |   docker-proxy(7622)---{docker-proxy}(7623)
          |          |          |          |   {docker-proxy}(7624)
          |          |          |          |   {docker-proxy}(7625)
          |          |          |          |   {docker-proxy}(7626)
          |          |          |   docker-proxy(7711)---{docker-proxy}(7712)
          |          |          |          |   {docker-proxy}(7713)
```

```

      └─{docker-proxy}(7714)
        ├─{docker-proxy}(7715)
        ├─{dockerd}(7325)
        ├─{dockerd}(7326)
        ├─{dockerd}(7327)
        ├─{dockerd}(7328)
        ├─{dockerd}(7329)
        ├─{dockerd}(7330)
        ├─{dockerd}(7338)
        ├─{dockerd}(7342)
        ├─{dockerd}(7343)
        ├─{dockerd}(7349)
        └─{dockerd}(7483)
      └─irqbalance(5908)

```

1.2.9.2: 查看 containerd 进程关系:

有四个进程:

dockerd: 被 client 直接访问, 其父进程为宿主机的 systemd 守护进程。

docker-proxy: 实现容器通信, 其父进程为 dockerd

containerd: 被 dockerd 进程调用以实现与 runc 交互。

containerd-shim: 真正运行容器的载体, 其父进程为 containerd。

```

[root@node1 ~]# ps -ef | grep containerd
root      4908      1  0 09:08 ?          00:00:31 /usr/bin/containerd
root      4910      1  0 09:08 ?          00:00:02 /usr/bin/dockerd -H fd:// --containe
rd=/run/containerd/containerd.sock
root      5359    4908  0 09:32 ?          00:00:00 containerd-shim -namespace moby -wor
kdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/5a7c6090b8adae677c8cd176f
9130ce83d27e60093f9424cbe9a99c86d69c690 -address /run/containerd/containerd.sock -cont
ainerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root      5539    4868  0 10:04 pts/0      00:00:00 grep --color=auto containerd
[root@node1 ~]#
[root@node1 ~]# ps -ef | grep docker-proxy
root      5353    4910  0 09:32 ?          00:00:00 /usr/bin/docker-proxy -proto tcp -ho
st-ip 0.0.0.0 -host-port 80 -container-ip 172.17.0.2 -container-port 80
root      5541    4868  0 10:04 pts/0      00:00:00 grep --color=auto docker-proxy
[root@node1 ~]#

```

1.2.9.3: containerd-shim 命令使用:

[root@node1 ~]# containerd-shim -h

Usage of containerd-shim:

- address string
grpc address back to main containerd
- containerd-binary containerd publish
path to containerd binary (used for containerd publish) (default "cont
- criu string
path to criu binary
- debug
enable debug output in logs
- namespace string
namespace that owns the shim
- runtime-root string
root directory for the runtime (default "/run/containerd/runc")
- socket string

```

abstract socket path to serve
-systemd-cgroup
    set runtime to use systemd-cgroup
-workdir string
    path used to storage large temporary data

```

1.2.9.4: 容器的创建与管理过程:

通信流程:

1. dockerd 通过 grpc 和 containerd 模块通信(runc)交换， dockerd 和 containerd 通信的 socket 文件: /run/containerd/containerd.sock。

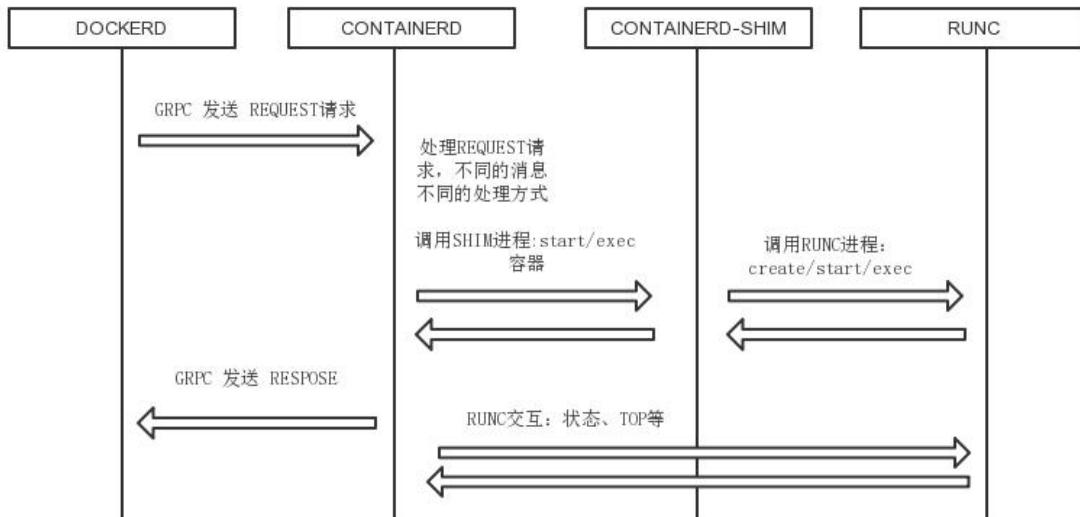
2. containerd 在 dockerd 启动时被启动，然后 containerd 启动 grpc 请求监听， containerd 处理 grpc 请求，根据请求做相应动作。

/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

3. 若是创建容器， containerd 拉起一个 container-shim 容器进程，并进行相应的创建操作。

4. container-shim 被拉起后， start/exec/create 拉起 runC 进程，通过 exit、control 文件和 containerd 通信，通过父子进程关系和 SIGCHLD(信号)监控容器中进程状态。

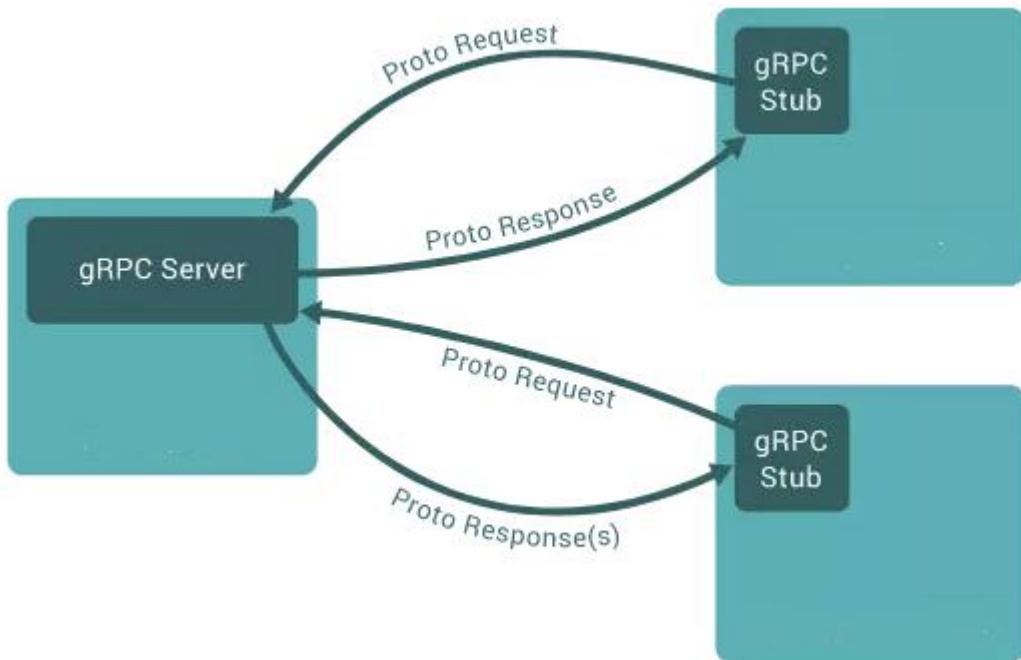
5. 在整个容器生命周期中， containerd 通过 epoll 监控容器文件，监控容器事件。



1.2.9.5: grpc 简介:

gRPC 是 Google 开发的一款高性能、开源和通用的 RPC 框架，支持众多语言客户端。

<https://www.grpc.io/>

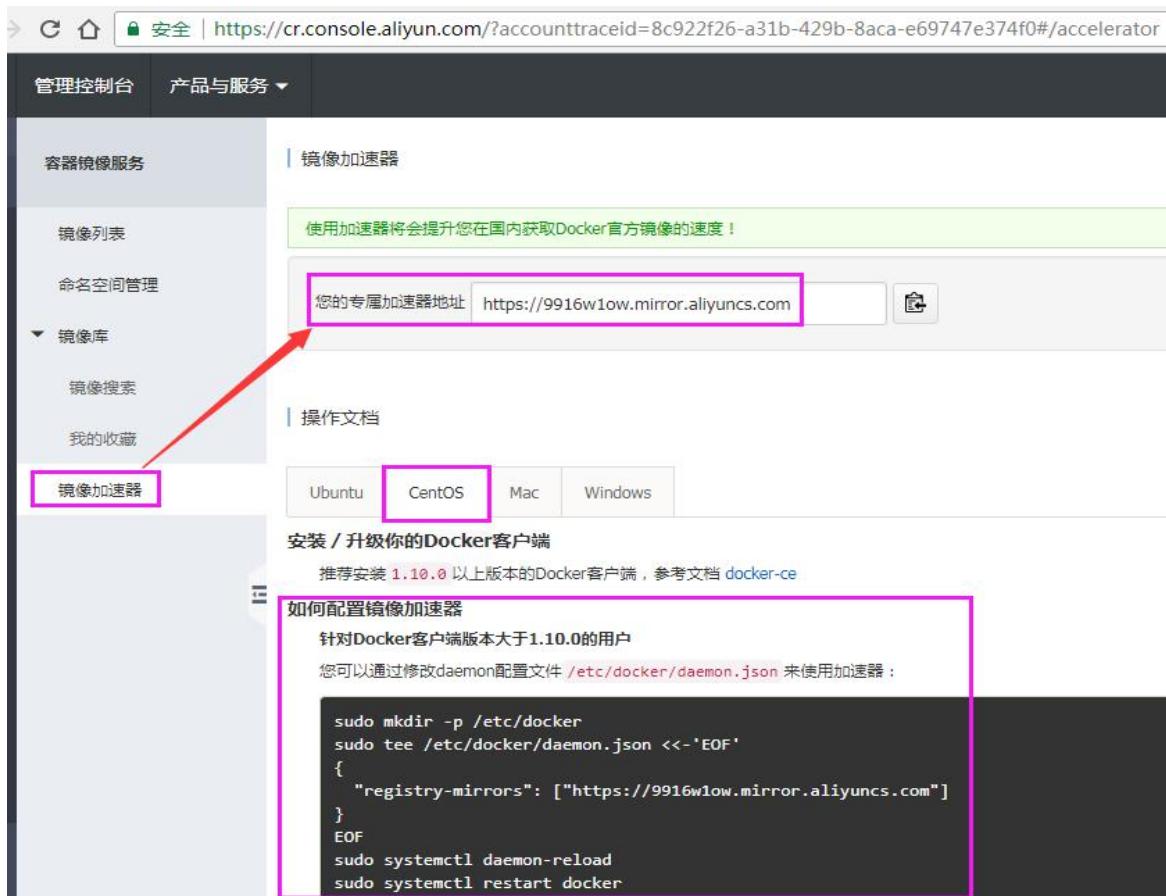


1.3: docker 镜像加速配置:

国内下载国外的镜像有时候会很慢，因此可以更改 docker 配置文件添加一个加速器，可以通过加速器达到加速下载镜像的目的。

1.3.1: 获取加速地址:

浏览器打开 <http://cr.console.aliyun.com>，注册或登录阿里云账号，点击左侧的镜像加速器，将会得到一个专属的加速地址，而且下面有使用配置说明：



1.3.2: 生成配置文件:

```
[root@docker-server1 ~]# mkdir -p /etc/docker
[root@docker-server1 ~]# sudo tee /etc/docker/daemon.json <<-'EOF'
> {
>   "registry-mirrors": ["https://9916w1ow.mirror.aliyuncs.com"]
> }
> EOF
{
  "registry-mirrors": ["https://9916w1ow.mirror.aliyuncs.com"]
}

[root@docker-server1 ~]# cat /etc/docker/daemon.json
{
  "registry-mirrors": ["https://9916w1ow.mirror.aliyuncs.com"]
}
```

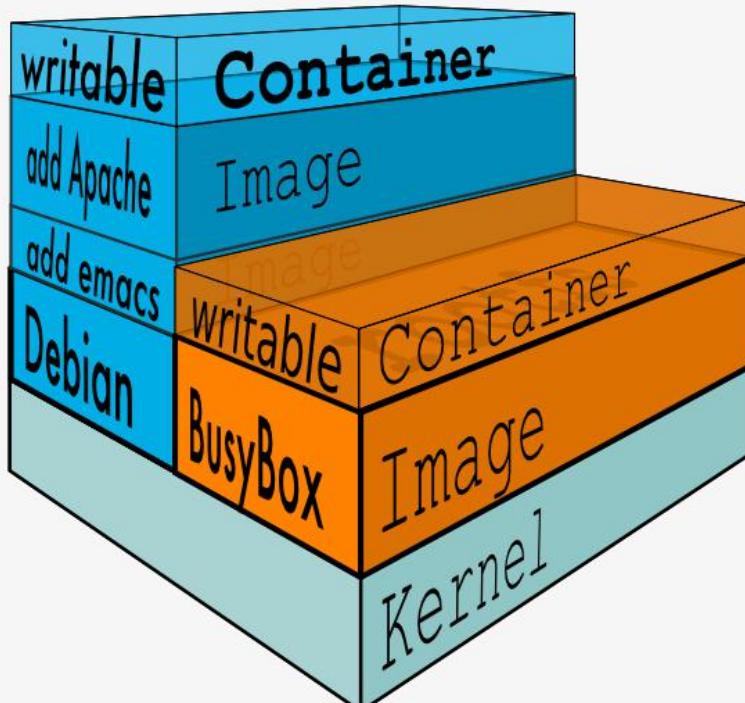
1.3.3: 重启 docke 服务:

```
[root@docker-server1 ~]# systemctl daemon-reload
[root@docker-server1 ~]# sudo systemctl restart docker
```

1.4: Docker 镜像管理:

Docker 镜像含有启动容器所需要的文件系统及所需要的内容，因此镜像主要用于创建并启动 docker 容器。

Docker 镜像含里面是一层层文件系统,叫做 Union File System (Union FS 联合文件系统) , 2004 年由纽约州立大学石溪分校开发, 联合文件系统可以将多个目录挂载到一起从而形成一整个虚拟文件系统, 该虚拟文件系统的目录结构就像普通 linux 的目录结构一样, docker 通过这些文件再加上宿主机的内核提供了一个 linux 的虚拟环境,每一层文件系统我们叫做一层 layer, 联合文件系统可以对每一层文件系统设置三种权限, 只读 (readonly) 、读写 (readwrite) 和写出 (whiteout-able) , 但是 docker 镜像中每一层文件系统都是只读的,构建镜像的时候,从一个最基本的操作系统开始,每个构建的操作都相当于做一层的修改,增加了一层文件系统,一层层往上叠加,上层的修改会覆盖底层该位置的可见性, 这也很容易理解, 就像上层把底层遮住了一样,当使用镜像的时候, 我们只会看到一个完全的整体, 不知道里面有几层也不需要知道里面有几层, 结构如下:



```
# pwd  
/usr/local/src  
#mkdir a b system  
#touch a/a.txt b/b.txt  
#mount -t aufs -o dirs=.:./b none ./system/
```

```
#tree system/
```

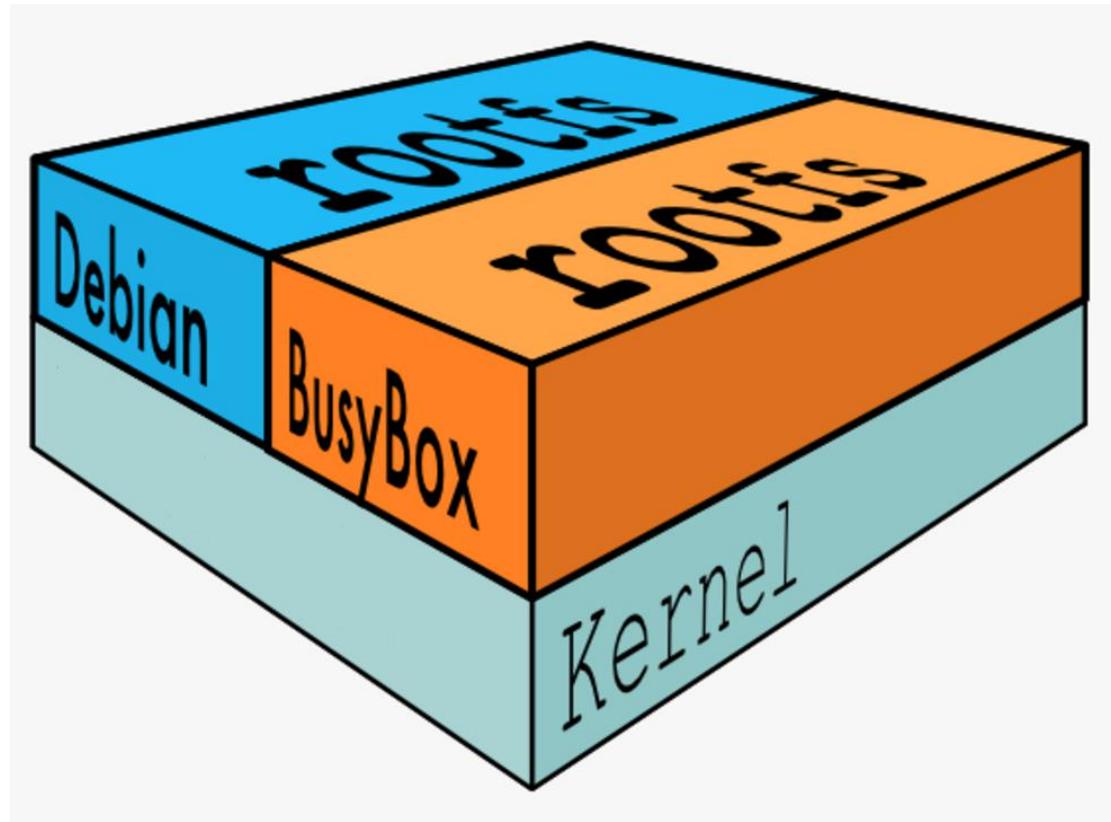
```
system/
├── a.txt
└── b.txt
```

```
0 directories, 2 file
```

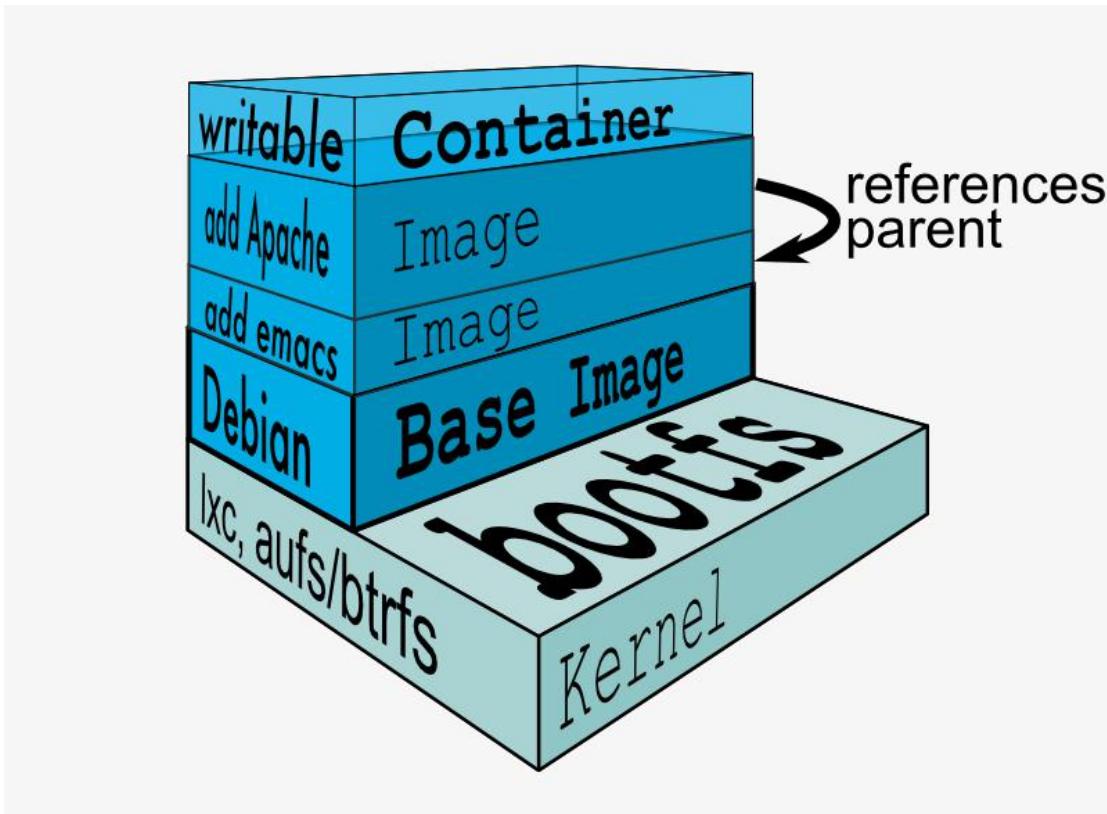
一个典型的 Linux 文件系统由 bootfs 和 rootfs 两部分组成, bootfs (boot file system) 主要包含 bootloader 和 kernel, bootloader 主要用于引导加载 kernel, 当 kernel 被加载到内存中后 bootfs 会被 umount 掉, rootfs (root file system) 包含的就是典型 Linux 系统中的 /dev, /proc, /bin, /etc 等标准目录和文件, 下图就是 docker image 中最基础的两层结构, 不同的 linux 发行版 (如 ubuntu 和 CentOS) 在 rootfs 这一层会有所区别。

但是对于 docker 镜像通常都比较小, 官方提供的 centos 基础镜像在 200MB 左右, 一些其他版本的镜像甚至只有几 MB, docker 镜像直接调用宿主机的内核, 镜像中只提供 rootfs, 也就是只需要包括最基本的命令、工具和程序库就可以了, 比如 alpine 镜像, 在 5M 左右。

下图就是有两个不同的镜像在一个宿主机内核上实现不同的 rootfs。



容器、镜像父镜像：



docker 命令是最常使用的 docker 客户端命令，其后面可以加不同的参数以实现相应功能，常用的命令如下：

1.4.1：搜索镜像：

在官方的 docker 仓库中搜索指定名称的 docker 镜像，也会有很多镜像。

```
[root@docker-server1 ~]# docker search centos:7.2.1511 #带指定版本号  
[root@docker-server1 ~]# docker search centos #不带版本号默认 latest
```

INDEX	NAME	DESCRIPTION	STARS
docker.io	docker.io/centos	The official build of CentOS.	3759
docker.io	docker.io/ansible/centos7-ansible	Ansible on Centos7	103
docker.io	docker.io/jdeathe/centos-ssh	CentOS-6 6.9 x86_64 / CentOS-7 7.4.1708 x8...	88
docker.io	docker.io/tutum/centos	Simple CentOS docker image with SSH access	33
docker.io	docker.io/imagine10255/centos6-lnmp-php56	centos6-lnmp-php56	31
docker.io	docker.io/gluster/gluster-centos	Official GlusterFS Image [CentOS-7 + Glu...	21
docker.io	docker.io/kinogmt/centos-ssh	CentOS with SSH	17
docker.io	docker.io/openshift/base-centos7	A Centos7 derived base image for Source-To...	11
docker.io	docker.io/centos/php-56-centos7	Platform for building and running PHP 5.6 ...	10
docker.io	docker.io/centos/python-35-centos7	Platform for building and running Python 3...	9
docker.io	docker.io/openshift/jenkins-2-centos7	A Centos7 based Jenkins v2.x image for use...	6
docker.io	docker.io/openshift/mysql-55-centos7	DEPRECATED: A Centos7 based MySQL v5.5 ima...	6
docker.io	docker.io/darksheer/centos	Base Centos Image -- Updated hourly	3
docker.io	docker.io/openshift/ruby-20-centos7	DEPRECATED: A Centos7 based Ruby v2.0 imag...	3

1.4.2：下载镜像：

从 docker 仓库将镜像下载到本地，命令格式如下：

```
# docker pull 仓库服务器:端口/项目名称/镜像名称:tag(版本)号
```

```
[root@docker-server1 ~]# docker pull alpine  
[root@docker-server1 ~]# docker pull nginx  
[root@docker-server1 ~]# docker pull hello-world  
[root@docker-server1 ~]# docker pull centos
```

下载中

```
[root@docker-server1 ~]# docker pull alpine  
Using default tag: latest  
Trying to pull repository docker.io/library/alpine ...  
latest: Pulling from docker.io/library/alpine  
88286f41530e: Downloading [=====>]  
88286f41530e: Pulling fs layer  
] 797.5 kB/1.99 MB
```

```
[root@docker-server1 ~]# docker pull centos  
Using default tag: latest  
Trying to pull repository docker.io/library/centos ...  
latest: Pulling from docker.io/library/centos  
d9aaaf4d82f24: Downloading [====>]  
d9aaaf4d82f24: Pulling fs layer  
] 7.875 MB/73.39 MB
```

下载完成

```
[root@docker-server1 ~]# docker pull alpine  
Using default tag: latest  
Trying to pull repository docker.io/library/alpine ...  
latest: Pulling from docker.io/library/alpine  
88286f41530e: Pull complete  
Digest: sha256:f006ecbb824d87947d0b51ab8488634bf69fe4094959d935c0c103f4820a417d  
[root@docker-server1 ~]#
```

```
[root@docker-server1 ~]# docker pull centos  
Using default tag: latest  
Trying to pull repository docker.io/library/centos ...  
latest: Pulling from docker.io/library/centos  
d9aaaf4d82f24: Pull complete  
Digest: sha256:eba772bac22c86d7d6e72421b4700c3f894ab6e35475a34014ff8de74c10872e  
[root@docker-server1 ~]#
```

1.4.3：查看本地镜像：

下载完成的镜像比下载的大，因为下载完成后会解压

```
[root@docker-server1 ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	cd5239a0906a	2 weeks ago	109MB
centos	latest	49f7960eb7e4	3 weeks ago	200MB
hello-world	latest	e38bc07ac18e	2 months ago	1.85kB
alpine	latest	3fd9065eaf02	5 months ago	4.15MB

REPOSITORY	# 镜像所属的仓库名称
TAG	# 镜像版本号（标识符），默认为 latest
IMAGE ID	# 镜像唯一 ID 标示
CREATED	# 镜像创建时间

VIRTUAL SIZE #镜像的大小

1.4.4: 镜像导出:

可以将镜像从本地导出为一个压缩文件，然后复制到其他服务器进行导入使用。

#导出方法 1:

```
[root@docker-server1 ~]# docker save centos -o /opt/centos.tar.gz  
[root@docker-server1 ~]# ll /opt/centos.tar.gz  
-rw----- 1 root root 205225472 Nov 1 03:52 /opt/centos.tar.gz
```

#导出方法 2:

```
[root@docker-server1 ~]# docker save centos > /opt/centos-1.tar.gz  
[root@docker-server1 ~]# ll /opt/centos-1.tar.gz  
-rw-r--r-- 1 root root 205225472 Nov 1 03:52 /opt/centos-1.tar.gz
```

#查看镜像内容:

```
[root@docker-server1 ~]# cd /opt/  
[root@docker-server1 opt]# tar xvf centos.tar.gz  
[root@docker-server1 opt]# cat manifest.json #包含了镜像的相关配置，配置文件、分层  
[{"Config": "196e0ce0c9fbb31da595b893dd39bc9fd4aa78a474bbdc21459a3ebe855b7768.json", "RepoTags": ["docker.io/centos:latest"], "Layers": ["892ebb5d1299cbf459f67aa070f29fdc6d83f4025c58c090e9a69bd4f7af436b/layer.tar"]}]
```

#分层为了方便文件的共用，即相同的文件可以共用

```
[{"Config": "配置文件.json", "RepoTags": ["docker.io/nginx:latest"], "Layers": ["分层1/layer.tar", "分层2/layer.tar", "分层3/layer.tar"]}]
```

1.4.5: 镜像导入:

将镜像导入到 docker

```
[root@docker-server1 ~]# scp /opt/centos.tar.gz 192.168.10.206:/opt/  
[root@docker-server2 ~]# docker load < /opt/centos.tar.gz  
[root@docker-server2 ~]# docker load < /opt/centos.tar.gz  
cf516324493c: Loading layer [=====> ] 177.7 MB/205.2 MB  
cf516324493c: Loading layer [> ] 557.1 kB/205.2 MB
```

验证镜像：

```
[root@docker-server2 ~]# docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
docker.io/centos    latest      196e0ce0c9fb    6 weeks ago   196.6 MB
[root@docker-server2 ~]#
```

1.4.6：删除镜像：

```
[root@docker-server1 opt]# docker rmi centos
[root@docker-server1 opt]# docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
docker.io/centos    latest      196e0ce0c9fb    6 weeks ago   196.6 MB
docker.io/alpine    latest      76da55c8019d    6 weeks ago   3.962 MB
[root@docker-server1 opt]# docker rmi centos
Untagged: centos:latest
Untagged: docker.io/centos@sha256:eba772bac22c86d7d6e72421b4700c3f894ab6e35475a34014ff8de74c10872e
Deleted: sha256:196e0ce0c9fb31da595b893dd39bc9fd4aa78a474bbdc21459a3ebbe855b7768
Deleted: sha256:c5f516324493c00941ac20020801553e87ed24c564fb3f269409ad138945948d4
[root@docker-server1 opt]# docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
docker.io/alpine    latest      76da55c8019d    6 weeks ago   3.962 MB
```

#获取运行参数帮助

```
[root@linux-docker opt]# docker daemon --help
```

总结：企业使用镜像及常见操作：

搜索、下载、导出、导入、删除

命令总结：

```
# docker load -i centos-latest.tar.xz #导入本地镜像
# docker save > /opt/centos.tar #centos #导出镜像
# docker rmi 镜像 ID/镜像名称 #删除指定 ID 的镜像，通过镜像启动容器的时候镜像不能被删除，除非将容器全部关闭
# docker rm 容器 ID/容器名称 #删除容器
# docker rm 容器 ID/容器名-f #强制删除正在运行的容器
```

1.5：容器操作基础命令：

命令格式：

```
# docker run [选项] [镜像名] [shell 命令] [参数]
# docker run [参数选项] [镜像名称, 必须在所有选项的后面] [/bin/echo 'hello world'] #单次执行, 没有自定义容器名称
# docker run centos /bin/echo 'hello world' #启动的容器在执行完 shell 命令就退出了
```

1.5.1：从镜像启动一个容器：

会直接进入到容器，并随机生成容器 ID 和名称

```
[root@docker-server1 ~]# docker run -it docker.io/centos bash  
[root@11445b3a84d3 /]#
```

```
#退出容器不注销  
ctrl+p+q
```

1.5.2：显示正在运行的容器：

```
[root@linux-docker ~]# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
11445b3a84d3	docker.io/centos	"bash"	7 minutes ago

1.5.3：显示所有容器：

包括当前正在运行以及已经关闭的所有容器：

```
[root@linux-docker ~]# docker ps -a
```

1.5.4：删除运行中的容器：

即使容正在运行当中，也会被强制删除掉

```
[root@docker-server1 ~]# docker rm -f 11445b3a84d3
```

```
[root@docker-server1 ~]# docker ps ← 删除前  
CONTAINER ID      IMAGE          COMMAND          CREATED  
11445b3a84d3      docker.io/centos   "bash"          28 minutes ago  
[root@docker-server1 ~]# docker rm -f 11445b3a84d3 ← 删除运行中的容器  
11445b3a84d3  
[root@docker-server1 ~]# docker ps ← 删除后  
CONTAINER ID      IMAGE          COMMAND          CREATED  
[root@docker-server1 ~]#
```

1.5.5：随机映射端口：

```
[root@docker-server1 ~]# docker pull nginx #下载 nginx 镜像
```

```
[root@docker-server1 ~]# docker run -P docker.io/nginx #前台启动并随机映射本地端口到容器的 80
```

#前台启动的会话窗口无法进行其他操作，除非退出，但是退出后容器也会退出

```
[root@docker-server1 ~]# docker run -P docker.io/nginx
```

#随机端口映射，其实是默认从 32768 开始

```
[root@docker-server1 ~]# ss -tnl
State      Recv-Q Send-Q          Local Address:Port
LISTEN      0      128              *:22
LISTEN      0      100             127.0.0.1:25
LISTEN      0      128              :::22
LISTEN      0      100              :::25
LISTEN      0      128              :::32768
[root@docker-server1 ~]# lsof -i:32768
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
docker-pr 2339 root    4u  IPv6  18860      0t0  TCP *:filenet-tms (LISTEN)
[root@docker-server1 ~]#
```

#访问端口：

```
192.168.10.205:32768
```

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

1.5.6：指定端口映射：

方式 1：本地端口 81 映射到容器 80 端口：

```
# docker run -p 81:80 --name nginx-test-port1 nginx
```

方式 2：本地 IP:本地端口:容器端口

```
# docker run -p 192.168.10.205:82:80 --name nginx-test-port2 docker.io/nginx
```

方式 3：本地 IP:本地随机端口:容器端口

```
# docker run -p 192.168.10.205::80 --name nginx-test-port3 docker.io/nginx
```

方式 4：本机 ip:本地端口:容器端口/协议，默认为 tcp 协议

```
# docker run -p 192.168.10.205:83:80/udp --name nginx-test-port4 docker.io/nginx
```

方式 5：一次性映射多个端口+协议：

```
# docker run -p 86:80/tcp -p 443:443/tcp -p 53:53/udp --name nginx-test-port5 docker.io/nginx
```

#命令截图

```
[root@docker-server1 ~]# docker run -d -p 81:80 --name nginx-test-port1 nginx
3214854df3f50871868deda1514e721f295de3649c8c1350785207763f93cf1e
[root@docker-server1 ~]# docker run -d -p 192.168.10.205:82:80 --name nginx-test-port2 docker.io/nginx
1f6c18cb23f37664ab010d1edd284054dfc5fd63d272164b490000b215399525
[root@docker-server1 ~]# docker run -d -p 192.168.10.205::80 --name nginx-test-port3 docker.io/nginx
ee5e51a925bae372b95fa84444d6d1626c2bc1014f827a4b7107fb545fb26d5e
[root@docker-server1 ~]# docker run -d -p 192.168.10.205:83:80/udp --name nginx-test-port4 docker.io/nginx
9137eb9aac025740051077138f57a9516fdc4a650e250c6534b8d30b58d6ad15
[root@docker-server1 ~]# docker run -d -p 86:80/tcp -p 443:443/tcp -p 53:53/udp --name nginx-test-port5 docker.io/nginx
f821d0cd5a996e42e5a253fab02ca773fe47727a428ba035fa4fd8c2cd93774e
[root@docker-server1 ~]#
```

#查看运行的容器：

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					
f821d0cd5a99	docker.io/nginx	"nginx -g 'daemon off"	6 minutes ago	Up 6 minutes	0.0.0.0:53->53/udp, 0.0.0.0:443->443/tcp
cp_0.0.0.0:86->80/tcp	nginx-test-port5	"nginx -g 'daemon off"	8 minutes ago	Up 8 minutes	80/tcp, 192.168.10.205:83->80/udp
9137eb9aac02	docker.io/nginx	"nginx -g 'daemon off"	9 minutes ago	Up 9 minutes	192.168.10.205:32768->80/tcp
ee5e51a9255a	docker.io/nginx	"nginx -g 'daemon off"	11 minutes ago	Up 11 minutes	192.168.10.205:82->80/tcp
1f6c18cb23f3	docker.io/nginx	"nginx -g 'daemon off"	13 minutes ago	Up 13 minutes	0.0.0.0:81->80/tcp
3214854df3f5	nginx	"nginx -g 'daemon off"	31 minutes ago	Up 31 minutes	0.0.0.0:32769->80/tcp
9aad776850b	docker.io/nginx	"nginx -g 'daemon off"	31 minutes ago	Up 31 minutes	0.0.0.0:32769->80/tcp
	nginx-test1				

#查看 Nginx 容器访问日志：

```
[root@docker-server1 ~]# docker logs nginx-test-port3 #一次查看
```

```
[root@docker-server1 ~]# docker logs -f nginx-test-port3 #持续查看
```

1.5.7：查看容器已经映射的端口：

```
[root@docker-server1 ~]# docker port nginx-test-port5
```

```
[root@docker-server1 ~]# docker port nginx-test-port5
53/udp -> 0.0.0.0:53
80/tcp -> 0.0.0.0:86
443/tcp -> 0.0.0.0:443
[root@docker-server1 ~]#
```

1.5.8：自定义容器名称：

```
[root@docker-server1 ~]# docker run -it --name nginx-test nginx
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e56a3829ba4a	nginx	"nginx -g 'daemon off"	4 seconds ago	Up 2 seconds	80/tcp	nginx-test

1.5.9：后台启动容器：

```
[root@docker-server1 ~]# docker run -d -P --name nginx-test1 docker.io/nginx
```

```
9aad776850bc06f516a770d42698e3b8f4ccce30d4b142f102ed3cb34399b31
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e56a3829ba4a	nginx	"nginx -g 'daemon off"	4 seconds ago	Up 2 seconds	80/tcp	nginx-test
[root@docker-server1 ~]# docker run -d -P -it --name nginx-test1 docker.io/nginx						
9aad776850bc06f516a770d42698e3b8f4ccce30d4b142f102ed3cb34399b31						
[root@docker-server1 ~]# docker ps						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
9aad776850b	docker.io/nginx	"nginx -g 'daemon off"	2 seconds ago	Up 2 seconds	0.0.0.0:32769->80/tcp	nginx-test1
[root@docker-server1 ~]#						

1.5.10：创建并进入容器：

```
[root@docker-server1 ~]# docker run -t -i --name test-centos2 docker.io/centos
```

```
/bin/bash
```

```
[root@a8fb69e71c73 /]# #创建容器后直接进入，执行exit退出后容器关闭
```

```
[root@docker-server1 ~]# docker run -t -i --name test-centos1 docker.io/centos /bin/bash
[root@80a7064883b0 /]# ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.2  0.0  11776  1820 ?        Ss   19:10   0:00 /bin/bash
root         13  0.0  0.0  47448  1664 ?        R+   19:11   0:00 ps -aux
[root@80a7064883b0 /]#
```

第一个进程

```
[root@docker-server1 ~]# docker run -d --name centos-test1 docker.io/centos
2cbbec43ba939476d798a5e1c454dd62d4d893ee12a09b587556ba6395353152
```

1.5.11：单次运行：

容器退出后自动删除：

```
[root@linux-docker opt]# docker run -it --rm --name nginx-delete-test
docker.io/nginx
```

1.5.12：传递运行命令：

容器需要有一个前台运行的进程才能保持容器的运行，通过传递运行参数是一种方式，另外也可以在构建镜像的时候指定容器启动时运行的前台命令。

```
[root@docker-server1 ~]# docker run -d centos /usr/bin/tail -f '/etc/hosts'
```

```
[root@docker-server1 ~]# docker run -d centos /usr/bin/tail -f '/etc/hosts'
3de260d211815663fa8be3ce2b477335dc16ce22c0e0367ccb55e339b8ea7a2a
[root@docker-server1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            NAMES
3de260d21181        centos              "/usr/bin/tail -f /etc/hosts"   3 seconds ago   pensive_stallman
91fc190cb538        nginx               "nginx -g 'daemon off;'"   13 minutes ago  test-nginx1
[root@docker-server1 ~]#
```

1.5.13：容器的启动和关闭：

```
[root@docker-server1 ~]# docker stop f821d0cd5a99
```

```
[root@docker-server1 ~]# docker start f821d0cd5a99
```

1.5.14：进入到正在运行的容器：

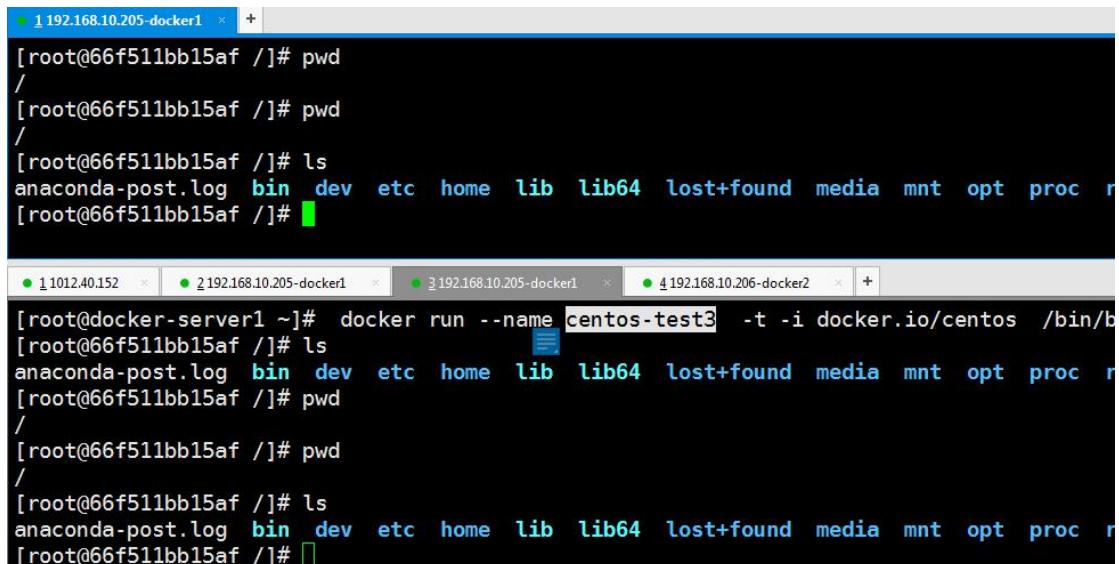
1.5.14.1：使用 attach 命令：

#使用方式为 docker attach 容器名，attach 类似于 vnc，操作会在各个容器界面显示，所有使用此方式进入容器的操作都是同步显示的且 exit 后容器将被关闭，且使用 exit 退出后容器关闭，不推荐使用，需要进入到有 shell 环境的容器，比

如 centos 为例：

```
[root@s1 ~]# docker run -it centos bash  
[root@63fbc2d5a3ec /]#  
[root@s1 ~]# docker attach 63fbc2d5a3ec  
[root@63fbc2d5a3ec /]#
```

在另外一个窗口启动测试页面是否同步：



```
[root@66f511bb15af /]# pwd  
/  
[root@66f511bb15af /]# ls  
anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc r  
[root@66f511bb15af /]#  
  
[root@docker-server1 ~]# docker run --name centos-test3 -t -i docker.io/centos /bin/bash  
[root@66f511bb15af /]# ls  
anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc r  
[root@66f511bb15af /]# pwd  
/  
[root@66f511bb15af /]# ls  
anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc r  
[root@66f511bb15af /]#
```

1.5.14.2：使用 exec 命令：

执行单次命令与进入容器，不是很推荐此方式，虽然 exit 退出容器还在运行

```
[root@docker-server1 ~]# docker exec -it centos-test3 /bin/bash  
[root@66f511bb15af /]# ll /opt/  
total 0  
[root@66f511bb15af /]# exit  
exit  
[root@docker-server1 ~]# docker ps  
CONTAINER ID IMAGE COMMAND CREATED  
66f511bb15af docker.io/centos "/bin/bash" 6 minutes ago  
test3
```

1.5.14.3：使用 nsenter 命令：

推荐使用此方式，nsenter 命令需要通过 PID 进入到容器内部，不过可以使用 docker inspect 获取到容器的 PID：

```
[root@docker-server1 ~]# yum install util-linux #安装 nsenter 命令  
[root@docker-server1 ~]# docker inspect -f "{{.NetworkSettings.IPAddress}}"  
91fc190cb538  
172.17.0.2
```

```
[root@docker-server1 ~]# docker inspect -f "{{.State.Pid}}" mydocker #获取到某个  
docker 容器的 PID，可以通过 PID 进入到容器内
```

```
[root@docker-server1 ~]# docker inspect -f "{{.State.Pid}}" centos-test3  
5892  
[root@docker-server1 ~]# nsenter -t 5892 -m -u -i -n -p  
[root@66f511bb15af /]# ls
```

1.5.14.4: 脚本方式:

将 nsenter 命令写入到脚本进行调用，如下：

```
[root@docker-server1 ~]# cat docker-in.sh  
#!/bin/bash  
docker_in(){  
    NAME_ID=$1  
    PID=$(docker inspect -f "{{.State.Pid}}" ${NAME_ID})  
    nsenter -t ${PID} -m -u -i -n -p  
}  
docker_in $1  
#测试脚本是否可以正常进入到容器且退出后仍然正常运行：  
[root@docker-server1 ~]# chmod a+x docker-in.sh  
[root@docker-server1 ~]# ./docker-in.sh centos-test3  
[root@66f511bb15af /]# pwd  
/  
[root@66f511bb15af /]# exit  
logout  
[root@docker-server1 ~]# ./docker-in.sh centos-test3  
[root@66f511bb15af /]# exit  
Logout
```

```
[root@docker-server1 ~]# chmod a+x docker-in.sh  
[root@docker-server1 ~]# ./docker-in.sh centos-test3  
[root@66f511bb15af /]# pwd  
/  
[root@66f511bb15af /]# exit  
logout  
[root@docker-server1 ~]# ./docker-in.sh centos-test3  
[root@66f511bb15af /]# exit  
logout  
[root@docker-server1 ~]#
```

1.5.15: 查看容器内部的 hosts 文件:

```
[root@docker-server1 ~]# docker run -i -t --name test-centos3 docker.io/centos  
/bin/bash  
[root@056bb4928b64 /]# cat /etc/hosts  
127.0.0.1 localhost  
::1 localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
ff00::0 ip6-mcastprefix  
ff02::1 ip6-allnodes  
ff02::2 ip6-allrouters  
172.17.0.4 056bb4928b64 #默认会将实例的 ID 添加到自己的 hosts 文件
```

#ping 容器 ID:

```
[root@056bb4928b64 /]# ping 056bb4928b64  
PING 056bb4928b64 (172.17.0.4) 56(84) bytes of data.  
64 bytes from 056bb4928b64 (172.17.0.4): icmp_seq=1 ttl=64 time=0.079 ms  
64 bytes from 056bb4928b64 (172.17.0.4): icmp_seq=2 ttl=64 time=0.048 ms  
64 bytes from 056bb4928b64 (172.17.0.4): icmp_seq=3 ttl=64 time=0.047 ms  
^C  
--- 056bb4928b64 ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2000ms  
rtt min/avg/max/mdev = 0.047/0.058/0.079/0.014 ms
```

1.5.16: 批量关闭正在运行的容器:

```
[root@docker-server1 ~]# docker stop $(docker ps -a -q) #正常关闭所有运行中的容器
```

```
[root@docker-server1 ~]# docker stop $(docker ps -a -q)  
7dd10f29cba0  
59f43d16a735  
926eb1b1ae4a
```

1.5.17: 批量强制关闭正在运行的容器:

```
[root@docker-server1 ~]# docker kill $(docker ps -a -q) #强制关闭所有运行中的容器
```

```
[root@docker-server1 ~]# docker kill $(docker ps -a -q)
7dd10f29cba0
59f43d16a735
926eb1b1ae4a
```

1.5.18：批量删除已退出容器：

```
[root@docker-server1 ~]# docker rm -f `docker ps -aq -f status=exited`
[root@docker-server1 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
96086942df72        jack/nginx-1.10.3:v1   "nginx"           2 minutes ago     Up 2 minutes
510253308acd        jack/nginx-1.10.3:v1   "nginx"           3 minutes ago     Exited (0) 3 minutes ago
[root@docker-server1 ~]# docker rm -f `docker ps -aq -f status=exited`
510253308acd
[root@docker-server1 ~]#
```

1.5.19：批量删除所有容器：

```
[root@docker-server1 ~]# docker rm -f $(docker ps -a -q)
```

```
[root@docker-server1 ~]# docker rm -f $(docker ps -a -q)
66f511bb15af
672f102519ec
2cbbec43ba93
9389494330fb
9b0deb509cab
5dc79ab21624
d874ed6a1f81
e594db0c1efd
a334959fce2
f071a8f46c25
5affff27a2403
2085a551868b
80a7064883b0
a8fb69e71c73
f821d0cd5a99
```

1.5.20：指定容器 DNS：

Dns 服务， 默认采用宿主机的 dns 地址

一是将 dns 地址配置在宿主机

二是将参数配置在 docker 启动脚本里面 --dns=1.1.1.1

1.5.21：其他命令：

```
# docker update 容器 --cpus 2 #更新容器配置信息

# docker events #获取 dockerd 的实时事件
2020-05-12T08:26:37.116277036+08:00 container create
ba55fe25733c123e7bf9ab5a8bcb78ddb83d864e2f012a2f0098c9bb9e4a66f2
```

```
(image=busybox, name=nifty_elgamal)
2020-05-12T08:26:37.118436296+08:00 container attach
ba55fe25733c123e7bf9ab5a8bcb78ddb83d864e2f012a2f0098c9bb9e4a66f2
(image=busybox, name=nifty_elgamal)
```

```
root@docker-node1:~# docker stop f8ae0c8dcaf8
f8ae0c8dcaf8
root@docker-node1:~# docker wait f8ae0c8dcaf8 #显示容器的退出状态码
137

# docker diff f8ae0c8dcaf8 #检查容器更改过的文件或目录
```

```
[root@docker-server1 ~]# docker run -it --dns 223.6.6.6 centos bash
[root@afeb628bf074 /]# cat /etc/resolv.conf
nameserver 223.6.6.6
```

二：Docker 镜像与制作：

Docker 镜像有没有内核？

从镜像大小上面来说，一个比较小的镜像只有十几 MB，而内核文件需要一百多兆，因此镜像里面是没有内核的，镜像在被启动为容器后将直接使用宿主机的内核，而镜像本身则只提供相应的 rootfs，即系统正常运行所必须的用户空间的文件系统，比如/dev/，/proc，/bin，/etc 等目录，所以容器当中基本是没有/boot 目录的，而/boot 当中保存的就是与内核相关的文件和目录。

```
[root@docker-server1 ~]# uname -r
3.10.0-862.el7.x86_64
[root@docker-server1 ~]# docker run -it --rm ubuntu bash
root@a5ae598f3fbe:/# cat /etc/issue
Ubuntu 18.04 LTS \n \l

root@a5ae598f3fbe:/# uname -r
3.10.0-862.el7.x86_64
root@a5ae598f3fbe:/#
root@a5ae598f3fbe:/# ll /boot/
total 0
drwxr-xr-x 2 root root 6 Apr 24 08:34 ../
drwxr-xr-x 1 root root 6 Jun 30 03:13 ...
root@a5ae598f3fbe:/# █
```

为什么没有内核？

由于容器启动和运行过程中是直接使用了宿主机的内核，所以没有直接调用过

物理硬件，所以也不会涉及到硬件驱动，因此也用不上内核和驱动，另外有内核的那是虚拟机。

2.1：手动制作 yum 版 nginx 镜像：

Docker 制作类似于虚拟机的模板制作，即按照公司的实际业务需求将需要安装的软件、相关配置等基础环境配置完成，然后将虚拟机再提交为模板，最后再批量从模板批量创建新的虚拟机，这样可以极大的简化业务中相同环境的虚拟机运行环境的部署工作，Docker 的镜像制作分为手动制作和自动制作(基于 DockerFile)，企业通常都是基于 Dockerfile 制作镜像，其中手动制作镜像步骤具体如下：

2.1.1：下载镜像并初始化系统：

基于某个基础镜像之上重新制作，因此需要先有一个基础镜像，本次使用官方提供的 centos 镜像为基础：

```
[root@docker-server1 ~]# docker pull centos
[root@docker-server1 ~]# docker run -it docker.io/centos /bin/bash
[root@37220e5c8410 /]# yum install wget -y
[root@37220e5c8410 /]# cd /etc/yum.repos.d/
[root@37220e5c8410 yum.repos.d]# rm -rf .//* #更改 yum 源
[root@37220e5c8410 yum.repos.d]# wget -O /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo
[root@37220e5c8410 yum.repos.d]# wget -O /etc/yum.repos.d/epel.repo
http://mirrors.aliyun.com/repo/epel-7.repo
```

2.1.2：yum 安装并配置 nginx：

```
[root@37220e5c8410 yum.repos.d]# yum install nginx -y #yum 安装 nginx
[root@37220e5c8410 yum.repos.d]# yum install -y vim wget pcre pcre-devel zlib
zlib-devel openssl openssl-devel iproute net-tools iotop #安装常用命令
```

2.1.3：关闭 nginx 后台运行：

```
[root@37220e5c8410 yum.repos.d]# vim /etc/nginx/nginx.conf #关闭 nginx 后台运行
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
```

```
pid /run/nginx.pid;  
daemon off; #关闭后台运行
```

2.1.4: 自定义 web 页面:

```
[root@37220e5c8410 yum.repos.d]# vim /usr/share/nginx/html/index.html  
[root@37220e5c8410 yum.repos.d]# cat /usr/share/nginx/html/index.html  
Docker Yum Nginx #自定义 web 界面
```

2.1.5: 提交为镜像:

在宿主机基于容器 ID 提交为镜像

```
root@docker-node1:~# docker commit --help  
Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]  
Create a new image from a container's changes  
Options:  
-a, --author string    Author (e.g., "John Hannibal Smith  
<hannibal@a-team.com>")  
-c, --change list      Apply Dockerfile instruction to the created image  
-m, --message string   Commit message  
-p, --pause            Pause container during commit (default true)
```

```
[root@docker-server1 ~]# docker commit -a "2973707860@qq.com" -m "nginx yum  
v1" --change="EXPOSE 80 443" f5f8c13d0f9f centos-nginx:v1  
sha256:7680544414220f6ff01e58ffc6bd59ff626af5dbbce4914ceb18e20f4ae0276a  
[root@docker-server1 ~]# docker commit -m "nginx image" f5f8c13d0f9f jack/centos-nginx  
sha256:7680544414220f6ff01e58ffc6bd59ff626af5dbbce4914ceb18e20f4ae0276a
```

2.1.6: 带 tag 的镜像提交:

提交的时候标记 tag 号:

#标记 tag 号，生产当中比较常用，后期可以根据 tag 标记创建不同版本的镜像以及创建不同版本的容器。

```
[root@docker-server1 ~]# docker commit -m "nginx image" f5f8c13d0f9f  
jack/centos-nginx:v1  
sha256:ab9759679eb586f06e315971d28b88f0cd3e0895d2e162524ee21786b98b24  
e8
```

```
[root@docker-server1 ~]# docker commit -m "nginx image" f5f8c13d0f9f jack/centos-nginx:v1  
sha256:ab9759679eb586f06e315971d28b88f0cd3e0895d2e162524ee21786b98b24e8  
[root@docker-server1 ~]#
```

2.1.7: 从自己镜像启动容器:

```
[root@docker-server1 ~]# docker run -d -p 80:80 --name my-centos-nginx  
jack/centos-nginx /usr/sbin/nginx  
ce4ee8732a0c4c6a10b85f5463396b27ba3ed120b27f2f19670fdff3bf5cdb62  
  
[root@docker-server1 ~]# docker run -d -p 80:80 --name my-centos-nginx jack/centos-nginx /usr/sbin/nginx  
ce4ee8732a0c4c6a10b85f5463396b27ba3ed120b27f2f19670fdff3bf5cdb62  
[root@docker-server1 ~]# ss -tnl  
State      Recv-Q Send-Q          Local Address:Port  
LISTEN      0      128              *:22  
LISTEN      0      100              127.0.0.1:25  
LISTEN      0      128              :::80 ←  
LISTEN      0      128              :::22  
LISTEN      0      100              :::1:25  
[root@docker-server1 ~]#
```

2.1.8: 访问测试:



```
[root@docker-server1 ~]# docker run -d -p 80:80 --name my-centos-nginx jack/centos-nginx /usr/sbin/nginx  
ce4ee8732a0c4c6a10b85f5463396b27ba3ed120b27f2f19670fdff3bf5cdb62  
[root@docker-server1 ~]# ss -tnl  
State      Recv-Q Send-Q          Local Address:Port  
LISTEN      0      128              *:22  
LISTEN      0      100              127.0.0.1:25  
LISTEN      0      128              :::80 ←  
LISTEN      0      128              :::22  
LISTEN      0      100              :::1:25  
[root@docker-server1 ~]#
```

2.2: DockerFile 制作编译版 nginx 1.16.1 镜像:

DockerFile 可以说是一种可以被 Docker 程序解释的脚本, DockerFile 是由一条条的命令组成的, 每条命令对应 linux 下面的一条命令, Docker 程序将这些 DockerFile 指令再翻译成真正的 linux 命令, 其有自己的书写方式和支持的命令, Docker 程序读取 DockerFile 并根据指令生成 Docker 镜像, 相比手动制作镜像的方式, DockerFile 更能直观的展示镜像是怎么产生的, 有了写好的各种各样 DockerFile 文件, 当后期某个镜像有额外的需求时, 只要在之前的 DockerFile 添加或者修改相应的操作即可重新生成新的 Docker 镜像, 避免了重复手动制作镜像的麻烦, 具体如下:

<https://docs.docker.com/engine/reference/builder/>

```
ADD  
COPY  
ENV
```

```
EXPOSE  
FROM  
LABEL  
STOP SIGNAL  
USER  
VOLUME  
WORKDIR  
RUN
```

2.2.1： 下载镜像并初始化系统：

```
[root@docker-server1 ~]# docker pull centos  
[root@docker-server1 ~]# docker run -it docker.io/centos /bin/bash  
[root@docker-server1 ~]# cd /opt/ # 创建目录环境  
[root@docker-server1 opt]# mkdir  
dockerfile/{web/{nginx,tomcat,jdk,apache},system/{centos,ubuntu,redhat}} -pv  
# 目录结构按照业务类型或系统类型等方式划分，方便后期镜像比较多的时候进行分类。  
  
[root@docker-server1 ~]# cd /opt/  
[root@docker-server1 opt]# mkdir dockerfile/{web/{nginx,tomcat,apache},system/{centos,ubuntu,redhat}} -pv  
mkdir: created directory 'dockerfile'  
mkdir: created directory 'dockerfile/web'  
mkdir: created directory 'dockerfile/web/nginx'  
mkdir: created directory 'dockerfile/web/tomcat'  
mkdir: created directory 'dockerfile/web/apache'  
mkdir: created directory 'dockerfile/system'  
mkdir: created directory 'dockerfile/system/centos'  
mkdir: created directory 'dockerfile/system/ubuntu'  
mkdir: created directory 'dockerfile/system/redhat'  
[root@docker-server1 opt]#
```

进入到指定的 Dockerfile 目录：

```
[root@docker-server1 opt]# cd dockerfile/web/nginx/  
[root@docker-server1 nginx]# pwd  
/opt/dockerfile/web/nginx
```

2.2.2： 编写 Dockerfile：

```
[root@docker-server1 nginx]# vim ./Dockerfile # 生成的镜像的时候会在执行命令的当前目录查找 Dockerfile 文件，所以名称不可写错，而且 D 必须大写  
  
#My Dockerfile  
#"#"为注释，等于 shell 脚本的中#  
#除了注释行之外的第一行，必须是 From xxx (xxx 是基础镜像)  
From centos # 第一行先定义基础镜像，后面的本地有效的镜像名，如果本地没有会从远程仓库下载，第一行很重要  
# 镜像维护者的信息  
MAINTAINER Jack.Zhang 123456@qq.com  
  
##### 其他可选参数  
#####
```

```

#USER #指定该容器运行时的用户名和 UID，后续的 RUN 命令也会使用这面指定的用户执行
#WORKDIR /a
#WORKDIR b #指定工作目录，最终为/a/b
#VOLUME ["/dir_1", "/dir_2" ..] 设置容器挂载主机目录
#ENV name jack #设置容器变量，常用于想容器内传递用户密码等
#####
#####

#执行的命令，将编译安装 nginx 的步骤执行一遍
RUN rpm -ivh http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm
RUN yum install -y vim wget tree lrzs gcc gcc-c++ automake pcre pcre-devel zlib
zlib-devel openssl openssl-devel iproute net-tools iotop
ADD nginx-1.16.1.tar.gz /usr/local/src/ #自动解压压缩包
RUN cd /usr/local/src/nginx-1.16.1 && ./configure --prefix=/usr/local/nginx
--with-http_sub_module && make && make install
RUN cd /usr/local/nginx/
ADD nginx.conf /usr/local/nginx/conf/nginx.conf
RUN useradd nginx -s /sbin/nologin
RUN ln -sv /usr/local/nginx/sbin/nginx /usr/sbin/nginx
RUN echo "test nginx page" > /usr/local/nginx/html/index.html
EXPOSE 80 443 #向外开放的端口，多个端口用空格做间隔，启动容器时候-p 需要使用此端向外映射，如：-p 8081:80，则 80 就是这里的 80
CMD ["nginx","-g","daemon off;"] #运行的命令，每个 Dockerfile 只能有一条，如果有两条则只有最后一条被执行

#如果在从该镜像启动容器的时候也指定了命令，那么指定的命令会覆盖 Dockerfile 构建的镜像里面的 CMD 命令，即指定的命令优先级更高，Dockerfile 的优先级较低一些，重新指定的指令优先级要高一些。

```

2.2.3：准备源码包与配置文件：

```

[root@docker-server1 nginx]# cp /usr/local/nginx/conf/nginx.conf . #使用其它服务器编译安装相同版本的 nginx 配置文件，而且配置文件中关闭 Nginx 进程后台运行
[root@docker-server1 nginx]# cp /usr/local/src/nginx-1.16.1.tar.gz . #nginx 源码包

```

2.2.4：执行镜像构建：

```
# docker build -t nginx:v1 .
```

```
Sending build context to Docker daemon 1.039MB
Step 1/15 : FROM centos:7.6.1810
--> f1cb7c7d58b7
Step 2/15 : MAINTAINER jack.zhang 2973707860@qq.com
--> Using cache
--> 5389bf8e6323
Step 3/15 : ENV name jack
--> Using cache
--> b7f0da176147
Step 4/15 : ENV PASS 123456
--> Using cache
--> fc38364909a5
Step 5/15 : RUN rpm -ivh http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm
--> Using cache
--> 24aa1b9c3302
Step 6/15 : RUN yum install -y vim wget tree lrzsz gcc gcc-c++ automake pcre pcre-devel zlib
s iotop
--> Using cache
--> 8e0045c0dd69
Step 7/15 : ADD nginx-1.16.1.tar.gz /usr/local/src/
--> Using cache
--> 0e76f05033e5
```

2.2.5：构建完成：

可以清晰看到各个步骤执行的具体操作

```
--> Using cache
--> 9607786e1dd6
Step 9/15 : RUN cd /usr/local/nginx/
--> Using cache
--> 35538102cb64
Step 10/15 : ADD nginx.conf /usr/local/nginx/conf/nginx.conf
--> Using cache
--> 7fffc6a056f15
Step 11/15 : RUN useradd nginx -s /sbin/nologin
--> Using cache
--> a7af7d5d9a01
Step 12/15 : RUN ln -sv /usr/local/nginx/sbin/nginx /usr/sbin/nginx
--> Using cache
--> c5006bd54ad1
Step 13/15 : RUN echo "test nginx page" > /usr/local/nginx/html/index.html
--> Using cache
--> 42d51d3555ed
Step 14/15 : EXPOSE 80 443
--> Using cache
--> 0775eda217dd
Step 15/15 : CMD ["nginx","-g","daemon off;"]
--> Using cache
--> 1fc72bde4677
Successfully built 1fc72bde4677
Successfully tagged nginx:v1
```

2.2.6: 查看是否生成本地镜像:

```
root@docker-server1:/opt/dockerfile/system/centos# docker images
REPOSITORY          TAG      IMAGE ID      CREATED             SIZE
nginx              v1       1fc72bde4677   5 minutes ago   572MB
<none>            <none>   cb2d8a0b4547   11 minutes ago  572MB
httpd              2.4.41   7d85cc3b2d80   2 weeks ago    154MB
alpine              latest   961769676411   2 weeks ago    5.58MB
nginx              latest   5a3221f0137b   3 weeks ago    126MB
mariadb             10.4.7   99c1098d5884   3 weeks ago    355MB
mariadb             10.4.7-bionic 99c1098d5884   3 weeks ago    355MB
ubuntu              18.04   a2a15febcd3   3 weeks ago    64.2MB
debian              9        f26939cc87ef   3 weeks ago    101MB
debian              stable   f5356914cf3c   3 weeks ago    114MB
ubuntu              16.04   5e13f8dd4cla   6 weeks ago    120MB
mysql               5.6.44   c30095c52827   7 weeks ago    256MB
nginx              1.14.2   295c7be07902   5 months ago   109MB
centos              7.6.1810  f1cb7c7d58b7   5 months ago   202MB
jumpserver/jms_all  1.4.8    e9274ba449e8   6 months ago   1.31GB
hello-world         latest   fce289e99eb9   8 months ago   1.84kB
root@docker-server1:/opt/dockerfile/system/centos#
```

2.2.7: 从镜像启动容器:

```
root@docker-server1:~# docker run -it -d -p 80:80 nginx:v1
8c48b74c1f86305eef18e8950fb25451c159aeda092902b6d52d6074cb0fe152
```

```
root@docker-server1:~# docker run -it -d -p 80:80 nginx:v1
8c48b74c1f86305eef18e8950fb25451c159aeda092902b6d52d6074cb0fe152
root@docker-server1:~# ss -tnl
State      Recv-Q      Send-Q      Local Address:Port
LISTEN      0           64          0.0.0.0:46189
LISTEN      0           128         0.0.0.0:55151
LISTEN      0           128         0.0.0.0:111
LISTEN      0           128         127.0.0.53%lo:53
LISTEN      0           128         0.0.0.0:22
LISTEN      0           128         127.0.0.1:6010
LISTEN      0           128         127.0.0.1:6011
LISTEN      0           128         127.0.0.1:6012
LISTEN      0           64          0.0.0.0:2049
LISTEN      0           128         0.0.0.0:34211
LISTEN      0           128         0.0.0.0:50059
LISTEN      0           128         [::]:39503
LISTEN      0           128         [::]:111
LISTEN      0           128         *:80
LISTEN      0           128         [::]:22
LISTEN      0           128         [::]:57273
LISTEN      0           128         [::]:6010
LISTEN      0           128         [::]:6011
LISTEN      0           128         [::]:6012
LISTEN      0           64          [::]:36001
LISTEN      0           64          [::]:2049
LISTEN      0           128         [::]:53889
LISTEN      0           128         *:3306
LISTEN      0           128         *:3307
root@docker-server1:~#
```

2.2.8: 访问 web 界面:

← → C ⌂ ⓘ 不安全 | 192.168.7.101

test nginx page

2.2.9: 镜像编译制作中:

制作镜像过程中，需要启动一个临时的容器进行相应的指令操作，操作完成后把此容器转换为 image。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
lafafa495eb9	0e76f05033e5	"/bin/sh -c 'cd /usr...'"	12 seconds ago	Up 11 seconds
urited_atmeida	nginx:v1	"nginx -g 'daemon of...'"	3 minutes ago	Up 3 minutes
ursing_robinson	mariadb:10.4.7-bionic	"docker-entrypoint.s..."	About an hour ago	Up About an hour
lant_burnell	mysql:5.6.44	"docker-entrypoint.s..."	About an hour ago	Up About an hour
quent_ellis				
root@docker-server1:/usr/local/src/nginx-1.16.1#				

2.3: 手动制作编译版本 nginx 1.16.1 镜像:

过程为在 centos 基础镜像之上手动编译安装 nginx，然后再提交为镜像。

2.3.1: 下载镜像并初始化系统:

```
[root@docker-server1 ~]# docker pull centos
[root@docker-server1 ~]# docker run -it docker.io/centos /bin/bash
[root@86a48908bb97 /]# yum install wget -y
[root@86a48908bb97 /]# cd /etc/yum.repos.d/
[root@86a48908bb97 yum.repos.d]# rm -rf ./ #更改 yum 源
[root@86a48908bb97 yum.repos.d]# wget -O /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo
[root@86a48908bb97 yum.repos.d]# wget -O /etc/yum.repos.d/epel.repo
http://mirrors.aliyun.com/repo/epel-7.repo
```

2.3.2: 编译安装 nginx:

```
[root@86a48908bb97 yum.repos.d]# yum install -y vim wget tree lrzs gcc
gcc-c++ automake pcre pcre-devel zlib zlib-devel openssl openssl-devel iproute
net-tools iotop #安装基础包
[root@86a48908bb97 yum.repos.d]# cd /usr/local/src/
[root@86a48908bb97 src]# wget http://nginx.org/download/nginx-1.16.1.tar.gz
[root@86a48908bb97 src]# tar xvf nginx-1.16.1.tar.gz
[root@86a48908bb97 src]# cd nginx-1.16.1
```

```
[root@86a48908bb97 nginx-1.16.1]# ./configure --prefix=/apps/nginx  
--with-http_sub_module  
[root@86a48908bb97 nginx-1.16.1]# make && make install  
[root@86a48908bb97 nginx-1.16.1]# cd /apps/nginx/
```

2.3.3: 关闭 nginx 后台运行:

```
[root@86a48908bb97 nginx]# vim /apps/nginx/conf/nginx.conf  
user    nginx;  
worker_processes  auto;  
daemon off;  
[root@86a48908bb97 nginx]# ln -sv /apps/nginx/sbin/nginx  /usr/sbin/nginx  #创建 nginx 命令软连  
:  
:
```

2.3.4: 自定义 web 界面:

```
[root@86a48908bb97 nginx]# mkdir  /apps/nginx/html/magedu  
[root@86a48908bb97 nginx]# echo "magedu" >  
/apps/nginx/html/magedu/index.html
```

2.3.5: 创建用户及授权:

```
[root@86a48908bb97 nginx]# useradd -u 2019 nginx -s /sbin/nologin  
[root@86a48908bb97 nginx]# chown  nginx.nginx /usr/local/nginx/ -R
```

2.3.6: 在宿主机提交为镜像:

```
[root@docker-server1 ~]# docker commit -m "test nginx" 86a48908bb97  
magedu-nginx:v1  
sha256:fce6e69410e58b8e508c7ffd2c5ff91e59a1144847613f691fa5e80bb68efbfa  
  
[root@docker-server1 ~]# docker commit -m "test nginx" 86a48908bb97 jack/nginx-test-image  
sha256:fce6e69410e58b8e508c7ffd2c5ff91e59a1144847613f691fa5e80bb68efbfa  
[root@docker-server1 ~]#
```

2.3.7: 从自己的镜像启动容器:

```
[root@docker-server1 ~]# docker run -d -p 80:80 --name magedu-centos-nginx  
magedu-nginx:v1 /usr/sbin/nginx  
8042aedec1d6412a79ac226c9289305087fc062b0087955a3a0a609c891e1122
```

备注： -name 是指定容器的名称， -d 是后台运行， -p 是端口映射， jack/nginx-test-image 是 xx 仓库下的 xx 镜像的 xx 版本，可以不加版本，不加版

本默认是使用 latest, 最后面的 nginx 是运行的命令, 即镜像里面要运行一个 nginx 命令, 所以才有了前面将 /usr/local/nginx/sbin/nginx 软连接到 /usr/sbin/nginx, 目的就是为了让系统可以执行此命令。

2.3.8: 访问测试:



2.3.9: 查看 Nginx 访问日志:

```
[root@8042aedec1d6 /]# tail -f /usr/local/nginx/logs/access.log
192.168.10.5 - - [23/Nov/2017:17:35:47 +0000] "GET / HTTP/1.1" 200 19 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
192.168.10.5 - - [23/Nov/2017:19:48:59 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
192.168.10.5 - - [23/Nov/2017:19:49:10 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
192.168.10.5 - - [23/Nov/2017:19:49:11 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
192.168.10.5 - - [23/Nov/2017:19:49:12 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
```

2.4: 自定义 Tomcat 业务镜像:

基于官方提供的 centos、debain、ubuntu、alpine 等基础镜像构建 JDK(Java 环境), 然后再基于自定义的 JDK 镜像构建出业务需要的 tomcat 镜像。

2.4.1: 构建 JDK 镜像:

先基于官方提供的基础镜像, 制作出安装了常用命令的自定义基础镜像, 然后在基础镜像的基础之上, 再制作 JDK 镜像、Tomcat 镜像等。

2.4.1.1: 自定义 Centos 基础镜像:

```
# docker pull centos
# mkdir
/opt/dockerfile/{web/{nginx,tomcat,jdk,apache},system/{centos,ubuntu,redhat}}
-pv
# cd /opt/dockerfile/system/centos/

# cat Dockerfile
# Centos Base Image
#
FROM centos:latest

MAINTAINER zhangshijie 2973707860@qq.com

RUN rpm -ivh http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm
```

```
RUN yum install -y vim wget tree lrzsz gcc gcc-c++ automake pcre pcre-devel zlib  
zlib-devel openssl openssl-devel iproute net-tools iotop  
RUN groupadd www -g 2020 && useradd www -u 2020 -g www #添加系统账户
```

```
# cat build-command.sh #通过脚本构建镜像  
#!/bin/bash  
docker build -t centos-base:v1 .
```

```
# bash build-command.sh #通过脚本进行镜像构建
```

自定义基础镜像制作过程中：

```
root@docker-server1:/opt/dockerfile/system/centos# bash build-command.sh  
Sending build context to Docker daemon 3.072kB  
Step 1/4 : FROM centos:latest  
--> 67fa590cf1c  
Step 2/4 : MAINTAINER zhangshijie 2973707860@qq.com  
--> Running in d94cceab9579  
Removing intermediate container d94cceab9579  
--> b54256e2b3ba  
Step 3/4 : RUN rpm -ivh http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm  
--> Running in 0b6425bae0ee  
warning: /var/tmp/rpm-tmp.LaEyaF: Header V3 RSA/SHA256 Signature, key ID 352c64e5: NOKEY  
Retrieving http://mirrors.aliyun.com/epel/epel-release-latest-7.noarch.rpm  
Preparing... ####  
Updating / installing... ####  
epel-release-7-11 ####  
Removing intermediate container 0b6425bae0ee  
--> 480b835296a4
```

自定义基础镜像制作完成：

```
perl-constant.noarch 0:1.27-2.el7  
perl-libs.x86_64 4:5.16.3-294.el7_6  
perl-macros.x86_64 4:5.16.3-294.el7_6  
perl-parent.noarch 1:0.225-244.el7  
perl-podlators.noarch 0:2.5.1-3.el7  
perl-threads.x86_64 0:1.87-4.el7  
perl-threads-shared.x86_64 0:1.43-6.el7  
vim-common.x86_64 2:7.4.160-6.el7_6  
vim-filesystem.x86_64 2:7.4.160-6.el7_6  
which.x86_64 0:2.20-7.el7  
  
Complete!  
Removing intermediate container 7e557326bfdf  
--> aac28115c2f3  
Successfully built aac28115c2f3  
Successfully tagged centos-base:v1
```

2.4.1.2：执行构建 JDK 镜像：

```
# cd /opt/dockerfile/web/jdk/
```

```
# cat Dockerfile
```

```
#JDK Base Image
```

```
FROM centos-base:v1

MAINTAINER zhangshijie "2973707860@qq.com"

ADD jdk-8u212-linux-x64.tar.gz /usr/local/src/
RUN ln -sv /usr/local/src/jdk1.8.0_212 /usr/local/jdk
ADD profile /etc/profile

ENV JAVA_HOME /usr/local/jdk
ENV JRE_HOME $JAVA_HOME/jre
ENV CLASSPATH $JAVA_HOME/lib/:$JRE_HOME/lib/
ENV PATH $PATH:$JAVA_HOME/bin

RUN rm -rf /etc/localtime && ln -snf /usr/share/zoneinfo/Asia/Shanghai
/etc/localtime
```

2.4.1.3: 上传 JDK 压缩包和 profile 文件:

将 JDK 压缩包上传到 Dockerfile 当前目录，然后执行构建：

```
# pwd
/opt/dockerfile/web/jdk
# tree

.
├── build-command.sh
├── Dockerfile
├── jdk-8u212-linux-x64.tar.gz
└── profile
```

0 directories, 4 files

2.4.1.4: 执行构建自定义 JDK 基础镜像:

构建并验证自定义 JDK 基础镜像

2.4.1.4.1: 通过脚本构建:

```
# cat build-command.sh
#!/bin/bash
docker build -t jdk-base:v8.212 .
```

2.4.1.4.2：执行构建：

```
Step 5/10 : ADD profile /etc/profile
--> 96a82ab6bd41
Step 6/10 : ENV JAVA_HOME /usr/local/jdk
--> Running in 299752cb2145
Removing intermediate container 299752cb2145
--> 6c218d58a031
Step 7/10 : ENV JRE_HOME $JAVA_HOME/jre
--> Running in 410023d4064e
Removing intermediate container 410023d4064e
--> 253e0d64355d
Step 8/10 : ENV CLASSPATH $JAVA_HOME/lib/:$JRE_HOME/lib/
--> Running in 618b2a89b8f5
Removing intermediate container 618b2a89b8f5
--> aea242862d97
Step 9/10 : ENV PATH $PATH:$JAVA_HOME/bin
--> Running in 2e6b157b91ec
Removing intermediate container 2e6b157b91ec
--> 54ad46a93278
Step 10/10 : RUN rm -rf /etc/localtime && ln -snf /usr/share/zoneinfo/Asia/Shanghai
--> Running in calf025ea396
Removing intermediate container calf025ea396
--> 8f1d91fe7af9
Successfully built 8f1d91fe7af9
Successfully tagged jdk-base:v8.212
```

2.4.1.5：验证镜像构建完成：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jdk-base	v8.212	8f1d91fe7af9	6 minutes ago	921MB
centos-base	v1	aac28115c2f3	17 minutes ago	515MB
magedu-nginx	v1	69fd53a4124d	2 hours ago	516MB
ubuntu1804-nginx-1.16.1	v1	959f293732b9	3 hours ago	394MB

2.4.1.6：验证镜像 JDK 环境：

从镜像创建出一个测试容器，验证容器的中的 JDK 环境是否可用。

```
[root@docker-server1 jdk]# docker run -it jack/ceontos-jdk bash
# docker run -it --rm jdk-base:v8.212 bash
[root@3373ee6a824e /]# java -version
java version "1.8.0_212"
Java(TM) SE Runtime Environment (build 1.8.0_212-b10)
Java HotSpot(TM) 64-Bit Server VM (build 25.212-b10, mixed mode)
```

2.4.2：从 JDK 镜像构建 tomcat 8 Base 镜像：

基于自定义的 JDK 基础镜像，构建出通用的自定义 Tomcat 基础镜像，此镜像后期会被多个业务的多个服务共同引用(相同的 JDK 版本和 Tomcat 版本)。

2.4.2.1：编辑 Dockerfile：

```
# pwd
/opt/dockerfile/web/tomcat/tomcat-base
```

```
# cat Dockerfile

#Tomcat Base Image
FROM jdk-base:v8.212

#env
ENV TZ "Asia/Shanghai"
ENV LANG en_US.UTF-8
ENV TERM xterm
ENV TOMCAT_MAJOR_VERSION 8
ENV TOMCAT_MINOR_VERSION 8.5.45
ENV CATALINA_HOME /apps/tomcat
ENV APP_DIR ${CATALINA_HOME}/webapps

#tomcat
RUN mkdir /apps
ADD apache-tomcat-8.5.45.tar.gz /apps
RUN ln -sv /apps/apache-tomcat-8.5.45 /apps/tomcat
```

2.4.2.2: 上传 tomcat 压缩包:

```
# tree
.
├── apache-tomcat-8.5.45.tar.gz
├── build-command.sh
└── Dockerfile
```

0 directories, 3 files

2.4.2.3: 通过脚本构建 tomcat 基础镜像:

```
[root@docker-server1 tomcat8-base]# cat build-command.sh
# cat build-command.sh
#!/bin/bash
docker build -t tomcat-base:v8.5.45 .
```

#执行构建:

```
# bash build-command.sh

Step 5/11 : ENV TOMCAT_MAJOR_VERSION 8
--> Using cache
--> c1147a213b67
Step 6/11 : ENV TOMCAT_MINOR_VERSION 8.5.45
--> Running in bf89e0ad3838
Removing intermediate container bf89e0ad3838
--> 0f21d5c64104
Step 7/11 : ENV CATALINA_HOME /apps/tomcat
--> Running in 658b9fdb462a
Removing intermediate container 658b9fdb462a
--> 4f77a194db29
Step 8/11 : ENV APP_DIR ${CATALINA_HOME}/webapps
--> Running in 0dec604b4e7b
Removing intermediate container 0dec604b4e7b
--> 0cacb1d8ba28
Step 9/11 : RUN mkdir /apps
--> Running in b6a829293f56
Removing intermediate container b6a829293f56
--> ee25399927e5
Step 10/11 : ADD apache-tomcat-8.5.45.tar.gz /apps
--> 67a28059acfa
Step 11/11 : RUN ln -sv /apps/apache-tomcat-8.5.45 /apps/tomcat
--> Running in 5d615ae7fc16
' /apps/tomcat' -> '/apps/apache-tomcat-8.5.45'
Removing intermediate container 5d615ae7fc16
--> 00e588ae3d60
Successfully built 00e588ae3d60
Successfully tagged tomcat-base:v8.5.45
```

2.4.2.4: 验证镜像构建完成:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tomcat-base	v8.5.45	00e588ae3d60	About a minute ago	936MB
jdk-base	v8.212	d20ed2b57d65	13 minutes ago	922MB
centos-base	v1	4e4d88944b3d	14 minutes ago	515MB
magedu-nginx	v1	69fd53a4124d	3 hours ago	516MB
ubuntu1804-nginx-1.16.1	v1	959f293732b9	3 hours ago	394MB

2.4.3: 构建业务镜像 1:

创建 tomcat app1 和 tomcat app2 两个目录，表示基于 tomcat 自定义基础镜像构建出不同业务的 tomcat app 镜像。

2.4.3.1: 准备 Dockerfile:

```
# pwd
```

```
/opt/dockerfile/web/tomcat/tomcat-app1

# cat Dockerfile
#Tomcat Web Image
FROM tomcat-base:v8.5.45

ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh
ADD myapp/* /apps/tomcat/webapps/myapp/

RUN chown www.www /apps/ -R

EXPOSE 8080 8009

CMD ["/apps/tomcat/bin/run_tomcat.sh"]
```

2.4.3.2: 准备自定义 myapp 页面:

```
# mkdir myapp
# echo "Tomcat Web Page" > myapp/index.html
# cat myapp/index.html
Tomcat Web Page
```

2.4.3.3: 准备容器启动执行脚本:

```
# cat run_tomcat.sh
#!/bin/bash
echo "1.1.1.1 abc.test.com" >> /etc/hosts
echo "nameserver 223.5.5.5" > /etc/resolv.conf

su - www -c "/apps/tomcat/bin/catalina.sh start"
su - www -c "tail -f /etc/hosts"
```

2.4.3.4: 准备构建脚本:

```
[root@docker-server1 tomcat-app1]# cat build-command.sh
# cat build-command.sh
#!/bin/bash
docker build -t tomcat-web:app1 .
```

2.4.3.5: 执行构建:

```
# pwd
/opt/dockerfile/web/tomcat/tomcat-app1
# chmod a+x *.sh
# tree
.
```

```
|── build-command.sh  
|── Dockerfile  
|── myapp  
|   └── index.html  
└── run_tomcat.sh
```

1 directory, 4 file

```
# bash build-command.sh
```

```
root@docker-server1:/opt/dockerfile/web/tomcat/tomcat-app1# bash build-command.sh  
Sending build context to Docker daemon 5.632kB  
Step 1/6 : FROM tomcat-base:v8.5.45  
--> 00e588ae3d60  
Step 2/6 : ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh  
--> c12b058de3bc  
Step 3/6 : ADD myapp/* /apps/tomcat/webapps/myapp/  
--> 33b5044a315f  
Step 4/6 : RUN chown www.www /apps/ -R  
--> Running in 47716ed9ad60  
Removing intermediate container 47716ed9ad60  
--> 70922c9df4d6  
Step 5/6 : EXPOSE 8080 8009  
--> Running in ca81969fd058  
Removing intermediate container ca81969fd058  
--> 187f1a11f3eb  
Step 6/6 : CMD ["/apps/tomcat/bin/run_tomcat.sh"]  
--> Running in 123ab9de45fa  
Removing intermediate container 123ab9de45fa  
--> 6363753dc615  
Successfully built 6363753dc615  
Successfully tagged tomcat-web:app1  
root@docker-server1:/opt/dockerfile/web/tomcat/tomcat-app1#
```

2.4.3.6: 从镜像启动容器测试:

```
# docker run -it -d -p 8080:8080 tomcat-web:app1  
ad2eef20efe9299deb3289137001690c19f14978a6cc34f8f383df5951ac98a0
```

```
root@docker-server1:/opt/dockerfile/web/tomcat/tomcat-app1# docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES  
ad2eef20efe9        tomcat-web:app1    "/apps/tomcat/bin/ru... "   About a minute ago   Up 1 minute          0.0.0.0:8080->8080   tomcat-web-1  
f78655fc1026        linux37-nginx:v1   "nginx"            4 hours ago       Up 4 hours          0.0.0.0:80->80     nginx-1  
563c9618d767        ubuntu:18.04      "bash"             4 hours ago       Up 4 hours          0.0.0.0:22->22     bash-1  
root@docker-server1:/opt/dockerfile/web/tomcat/tomcat-app1#
```

2.4.3.7: 访问测试:

```
← → ⌂ ⌂ ⓘ 不安全 | 192.168.7.101:8080/myapp/
```

Tomcat Web Page

2.4.4.: 构建业务镜像 2:

第二个 tomcat 服务镜像。

2.4.4.1: 准备 Dockerfile:

```
# pwd  
/opt/dockerfile/web/tomcat/tomcat-app2  
  
# cat Dockerfile  
#Tomcat Web Image  
FROM tomcat-base:v8.5.45  
  
ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh  
ADD myapp/* /apps/tomcat/webapps/myapp/  
  
RUN chown www.www /apps/ -R  
  
EXPOSE 8080 8009  
  
CMD ["/apps/tomcat/bin/run_tomcat.sh"]
```

2.4.4.2: 准备自定义页面:

```
# mkdir myapp  
# echo "Tomcat Web Page2" > myapp/index.html  
  
# cat myapp/index.html  
Tomcat Web Page2
```

2.4.4.3: 准备容器启动脚本:

```
# cat run_tomcat.sh  
#!/bin/bash  
echo "1.1.1.1 abc.test.com" >> /etc/hosts  
echo "nameserver 223.5.5.5" > /etc/resolv.conf  
  
su - www -c "/apps/tomcat/bin/catalina.sh start"  
su - www -c "tail -f /etc/hosts"
```

2.4.4.4: 准备构建脚本:

```
# cat build-command.sh  
#!/bin/bash  
docker build -t tomcat-web:app2 .
```

2.4.4.5: 执行构建:

```
# pwd
```

```
/opt/dockerfile/web/tomcat/tomcat-app2
# bash build-command.sh

root@docker-server1:/opt/dockerfile/web/tomcat/tomcat-app2# bash build-command.sh
Sending build context to Docker daemon 5.632kB
Step 1/6 : FROM tomcat-base:v8.5.45
--> 00e588ae3d60
Step 2/6 : ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh
--> Using cache
--> c12b058de3bc
Step 3/6 : ADD myapp/* /apps/tomcat/webapps/myapp/
--> 63083f84e133
Step 4/6 : RUN chown www.www /apps/ -R
--> Running in 33914bbe8494
Removing intermediate container 33914bbe8494
--> 53af1627facf
Step 5/6 : EXPOSE 8080 8009
--> Running in b7c76cdc7553
Removing intermediate container b7c76cdc7553
--> 6489f0731a8f
Step 6/6 : CMD ["/apps/tomcat/bin/run_tomcat.sh"]
--> Running in fc308e4467e8
Removing intermediate container fc308e4467e8
--> 5f6d1b2ffee8
Successfully built 5f6d1b2ffee8
Successfully tagged tomcat-web:app2
root@docker-server1:/opt/dockerfile/web/tomcat/tomcat-app2#
```

2.4.4.6: 从镜像启动容器:

```
# docker run -it -d -p 8081:8080 tomcat-web:app2
fa01784532f09c04da3ab0f81efdee7aefebc15c8f6dbfb00a9bbc566e6f597
```

2.4.4.7: 访问测试:

← → C ⌂ ⓘ 不安全 | 192.168.7.101:8081/myapp/

Tomcat Web Page2

2.5: 构建 haproxy 镜像:

构建出 haproxy 镜像，将 haproxy 通过容器的方式运行。

2.5.1: 准备 Dockerfile:

```
# pwd
/opt/dockerfile/web/haproxy

# cat Dockerfile
#Haproxy Base Image
FROM centos-base:v1
```

```
MAINTAINER zhangshijie "2973707860@qq.com"
```

```
RUN yum install -y yum install gcc gcc-c++ glibc glibc-devel pcre pcre-devel openssl  
openssl-devel systemd-devel net-tools vim iotop bc zip unzip zlib-devel lrzsz tree s  
creen lsof tcpdump wget ntpdate ADD haproxy-2.0.5.tar.gz /usr/local/src/  
RUN cd /usr/local/src/haproxy-2.0.5 && make ARCH=x86_64  
TARGET=linux-glibc USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1 USE_SYSTEMD=1  
USE_CPU_AFFINITY=1 PREFIX=/usr/local/hap  
roxy && make install PREFIX=/usr/local/haproxy && cp haproxy /usr/sbin/ &&  
mkdir /usr/local/haproxy/runADD haproxy.cfg /etc/haproxy/  
  
ADD run_haproxy.sh /usr/bin  
EXPOSE 80 9999  
CMD ["/usr/bin/run_haproxy.sh"]
```

2.5.2: 准备 haproxy 源码和配置文件:

```
# pwd  
/opt/dockerfile/web/haproxy  
  
# ll  
total 2496  
drwxr-xr-x 2 root root 117 Sep 12 15:31 ./  
drwxr-xr-x 7 root root 73 Sep 12 15:23 ../  
-rwxr-xr-x 1 root root 44 Sep 12 15:30 build-command.sh*  
-rw-r--r-- 1 root root 717 Sep 12 15:31 Dockerfile  
-rw-r--r-- 1 root root 2539226 Sep 12 15:22 haproxy-2.0.5.tar.gz  
-rw-r--r-- 1 root root 673 Sep 12 15:30 haproxy.cfg  
-rwxr-xr-x 1 root root 67 Aug 3 2018 run_haproxy.sh*
```

2.5.3: 准备 haproxy 配置文件:

```
# cat haproxy.cfg  
global  
chroot /usr/local/haproxy  
#stats socket /var/lib/haproxy/haproxy.sock mode 600 level admin  
uid 99  
gid 99  
daemon  
nbproc 1  
pidfile /usr/local/haproxy/run/haproxy.pid  
log 127.0.0.1 local3 info  
  
defaults  
option http-keep-alive
```

```
option forwardfor
mode http
timeout connect 300000ms
timeout client 300000ms
timeout server 300000ms

listen stats
    mode http
    bind 0.0.0.0:9999
    stats enable
    log global
    stats uri      /haproxy-status
    stats auth     haadmin:123456

listen web_port
    bind 0.0.0.0:80
    mode http
    log global
    balance roundrobin
    server web1  192.168.7.101:8080  check inter 3000 fall 2 rise 5
    server web2  192.168.7.102:8080  check inter 3000 fall 2 rise 5
```

2.5.4: 准备镜像构建脚本:

```
# cat build-command.sh
#!/bin/bash
docker build -t haproxy:2.0.5 .
```

2.5.5: 执行构建 haproxy 镜像:

```
'doc/peers.txt' -> '/usr/local/haproxy/doc/haproxy/peers.txt'
'doc/close-options.txt' -> '/usr/local/haproxy/doc/haproxy/close-options.txt'
'doc/SP0E.txt' -> '/usr/local/haproxy/doc/haproxy/SP0E.txt'
'doc/intro.txt' -> '/usr/local/haproxy/doc/haproxy/intro.txt'
Removing intermediate container 45bff4dc0378
--> f7db5efd1b2c
Step 6/9 : ADD haproxy.cfg /etc/haproxy/
--> d0ba50ebb3e8
Step 7/9 : ADD run_haproxy.sh /usr/bin
--> b293c2f423f8
Step 8/9 : EXPOSE 80 9999
--> Running in b4ef9fc48c66
Removing intermediate container b4ef9fc48c66
--> 2c63227039ab
Step 9/9 : CMD ["/usr/bin/run_haproxy.sh"]
--> Running in 2a3a05343bce
Removing intermediate container 2a3a05343bce
--> fc1e0e61d803
Successfully built fc1e0e61d803
Successfully tagged haproxy:2.0.5
root@docker-server1:/opt/dockerfile/web/haproxy#
```

2.5.6: 从镜像启动容器:

```
# docker run -it -d -p 80:80 -p 9999:9999  haproxy:2.0.5
d20e821a82e4a4a8b2d4b4c7f79f4750e5b3878bf10b43d30bd3ad2196687afa
```

2.5.7: web 访问验证:

← → ⌂ ⌂ ⓘ 不安全 | 192.168.7.101/myapp/

Tomcat Web Page

2.5.8: 访问 haproxy 控制端:

下图中 web2 服务器，需要将 tomcat 镜像从镜像服务器把 tomcat 镜像导出，然后 scp 到 web2 服务器，然后导入镜像，最后通过镜像启动一个容器并监听本机的 8080 端口即可，否则 web2 服务器状态显示异常。

HAProxy version 2.0.5, released 2019/08/16																					
Statistics Report for pid 7																					
> General process information																					
<p>pid = 7 (process #1, nbproc = 1, nbthread = 1) uptime = 0d 0h0m14s system limits: memmax = unlimited; ulimit-n = 1048575 maxsock = 1048575; maxconn = 524279; maxpipes = 0 current connns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 22.296 kbps Running tasks: 1/12; idle = 100 %</p>																					
<p>Display of:</p> <ul style="list-style-type: none"> • S • H • R • C <p>Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.</p>																					
stats			Queue			Session rate			Sessions			Bytes			Denied		Errors		Warnings		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Req	Conn	Resp	Retr	Redis	
Frontend				1	2	-	1	2	524 279	9			7 307	136 465	0	0	0			Status	
Backend	0	0		1	1		0	1	52 428	7	0	0s	7 307	136 465	0	0	7	0	0	0	
																			LastChk		
web_port			Queue			Session rate			Sessions			Bytes			Denied		Errors		Warnings		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Req	Conn	Resp	Retr	Redis	Status
Frontend				0	1	-	0	1	524 279	1			0 211	0	0	1					OPEN
web1	0	0	-	0	0		0	0	-	0	0	? 0	0	0	0	0	0	0	0	8m14s UP	
web2	0	0	-	0	0		0	0	-	0	0	? 0	0	0	0	0	0	0	0	L4OK in 1ms	
Backend	0	0		0	0		0	0	52 428	0	0	? 0	211	0	0	0	0	0	0	8m14s UP	
																			LastChk		

2.6: 基于官方 alpine 基础镜像制作自定义镜像:

```
FROM alpine
```

```
maintainer zhangshijie "2973707860@qq.com"
```

```
COPY repositories /etc/apk/repositories
RUN apk update && apk add iotop gcc libgcc libc-dev libcurl libc-utils pcre-dev
zlib-dev libnfs make pcre pcre2 zip unzip net-tools pstree wget libevent
libevent-dev iproute2
ADD nginx-1.16.1.tar.gz /opt/
RUN cd /opt/nginx-1.16.1 && ./configure --prefix=/apps/nginx && make && make
install && ln -sv /apps/nginx/sbin/nginx /usr/bin/
```

```
RUN addgroup -g 2019 -S nginx && adduser -s /sbin/nologin -S -D -u 2019 -G
nginx nginx
```

```
COPY nginx.conf /apps/nginx/conf/nginx.conf
```

```
ADD static.tar.gz /data/nginx/html
```

```
RUN chown nginx.nginx /data/nginx/ /apps/nginx/ -R
```

```
EXPOSE 80 443
```

```
CMD ["nginx"]
```

2.7: 基于官方 Ubuntu 基础镜像制作自定义镜像:

```
# cat Dockerfile
```

```
FROM ubuntu:18.04
```

```
maintainer zhangshijie "2973707860@qq.com"
```

```
COPY sources.list /etc/apt/sources.list

RUN apt update && apt install -y iproute2 ntpdate tcpdump telnet traceroute
nfs-kernel-server nfs-common lrzsz tree openssl libssl-dev libpcre3 libpcre3-dev
zlib1g-dev ntpdate tcpdump telnet traceroute gcc openssh-server lrzsz tree
openssl libssl-dev libpcre3 libpcre3-dev zlib1g-dev ntpdate tcpdump telnet
traceroute iotop unzip zip make && touch /tmp/linux.txt

ADD nginx-1.16.1.tar.gz /usr/local/src
RUN cd /usr/local/src/nginx-1.16.1 && ./configure --prefix=/apps/nginx && make &&
make install && ln -sv /apps/nginx/sbin/nginx /usr/bin && rm
-rf /usr/local/src/nginx-1.16.1 && rm -rf /usr/local/src/nginx-1.16.1.tar.gz
ADD nginx.conf /apps/nginx/conf/nginx.conf
ADD static.tar.gz /data/nginx/html

RUN groupadd -g 2019 nginx && useradd -g nginx -s /usr/sbin/nologin -u 2019
nginx && chown -R nginx.nginx /apps/nginx /data/nginx

EXPOSE 80 443

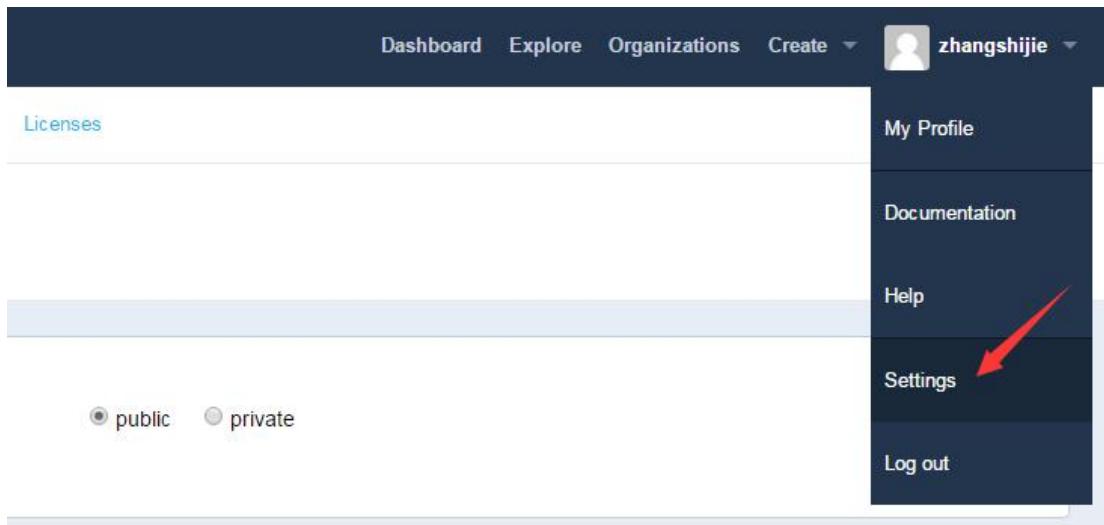
CMD ["nginx"]
```

2.8：本地镜像上传至官方 docker 仓库：

#将自制的镜像上传至 docker 仓库；<https://hub.docker.com/>，实现镜像夸主机备份。

2.8.1：准备账户：

登录到 docker hub 创建官网创建账户，登录后点击 settings 完善账户信息：



2.8.2: 填写账户基本信息:

The screenshot shows the "Account Information" section of the Docker Hub settings. It includes fields for Name (zhang), Location (China), City (Beijing), Website (https://hub.docker.com), and Email (rootroot@aliyun.com). A red box highlights these input fields. A blue "Save" button is located at the bottom right.

2.8.3: 在虚拟机使用自己的账号登录:

```
# # docker login  docker.io
root@docker-node3:~# docker login  docker.io
Login with your Docker ID to push and pull images from Docker Hub. If you don't have
one.
Username: zhangshijie
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

2.8.4: 查看认证信息:

```
#登录成功之后会在当前目录生成一个隐藏文件用于保存登录认证信息
```

```
# cat  /root/.docker/config.json
{
  "auths": {
    "https://index.docker.io/v1/": {
      "auth": "emhhbmdzaGlqaWU6emhhbmdAMTlZ"
    }
}
```

```
        },
        "HttpHeaders": {
            "User-Agent": "Docker-Client/18.09.9 (linux)"
        }
    }
root@docker-server1:/opt/dockerfile/system/centos#
```

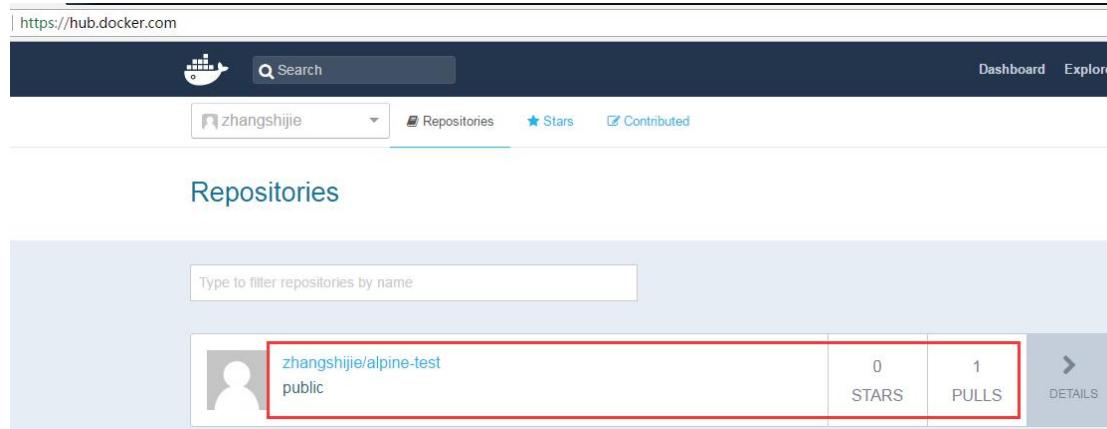
2.8.5: 给镜像做 tag 并开始上传:

```
[root@linux-docker ~]# docker images #查看镜像 ID
#为镜像做标记并上传至仓库
# docker tag alpine:latest docker.io/zhangshijie/alpine:latest
# docker push docker.io/zhangshijie/alpine:latest
```

2.8.6: 上传完成

```
root@docker-server1:~# docker tag alpine:latest docker.io/zhangshijie/alpine:latest
root@docker-server1:~# docker push docker.io/zhangshijie/alpine:latest
The push refers to repository [docker.io/zhangshijie/alpine]
03901b4a2ea8: Layer already exists
latest: digest: sha256:acd3ca9941a85e8ed16515bfc5328e4e2f8c128caa72959a58a127b7801ee01f size: 528
root@docker-server1:~#
```

2.8.7: 到 docker 官网验证:



2.8.8: 更换到其他 docker 服务器下载镜像:

```
[root@docker-server2 ~]# docker login https://hub.docker.com
Username: zhangshijie
Password:
Login Succeeded
[root@docker-server2 ~]# docker pull zhangshijie/alpine-test
Using default tag: latest
Trying to pull repository docker.io/zhangshijie/alpine-test ...
latest: Pulling from docker.io/zhangshijie/alpine-test
6d987f6f4279: Pull complete
Digest: sha256:641b95ddb2ea9dc2af1a0113b6b348ebc20872ba615204fbe12148e98fd6f23d
[root@docker-server2 ~]# docker images
REPOSITORY          TAG        IMAGE ID      CREATED       SIZE
docker.io/centos    latest     196e0ce0c9fb   7 weeks ago   196.6 MB
docker.io/zhangshijie/alpine-test  latest     76da55c8019d   7 weeks ago   3.962 MB
[root@docker-server2 ~]#
```

2.8.9: 从镜像启动一个容器:

```
[root@docker-server2 ~]# docker run -it docker.io/zhangshijie/alpine-test sh  
/ # ls  
bin dev etc home lib media mnt proc root run sbin srv  
/ # █
```

2.9: 本地镜像上传到阿里云:

将本地镜像上传至阿里云，实现镜像备份与统一分发的功能。

<https://cr.console.aliyun.com>

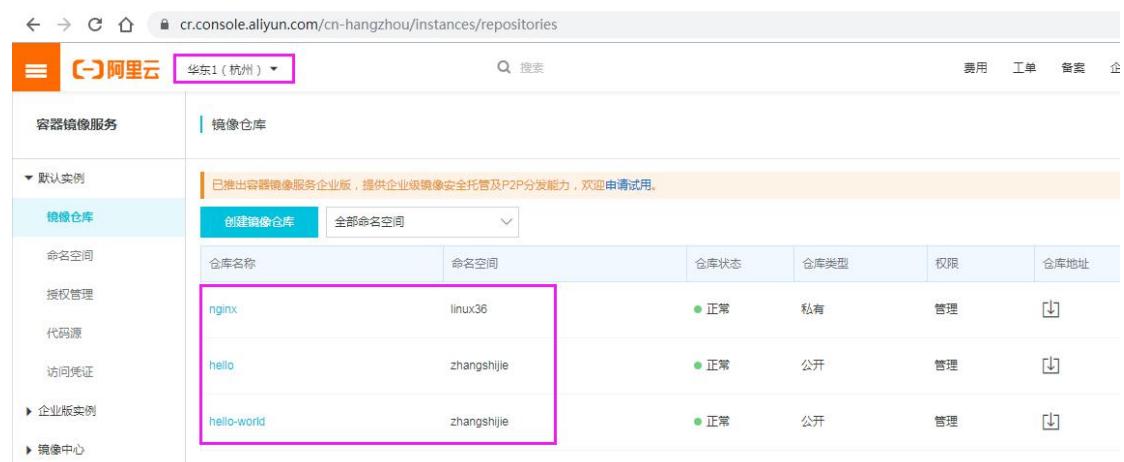
注册账户、创建 namespace、创建仓库、修改镜像 tag 及上传镜像

```
# docker login --username=rooroot@aliyun.com registry.cn-hangzhou.aliyuncs.com  
Password:  
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.  
Configure a credential helper to remove this warning. See  
https://docs.docker.com/engine/reference/commandline/login/#credentials-store  
  
Login Succeeded
```

2.9.1: 将镜像推送到 Registry

```
# docker login --username=rooroot@aliyun.com registry.cn-hangzhou.aliyuncs.com  
# docker tag [ImageId] registry.cn-hangzhou.aliyuncs.com/magedu/nginx:[镜像版本号]  
# docker push registry.cn-hangzhou.aliyuncs.com/ magedu /nginx:[镜像版本号]
```

2.9.2: 阿里云验证镜像:



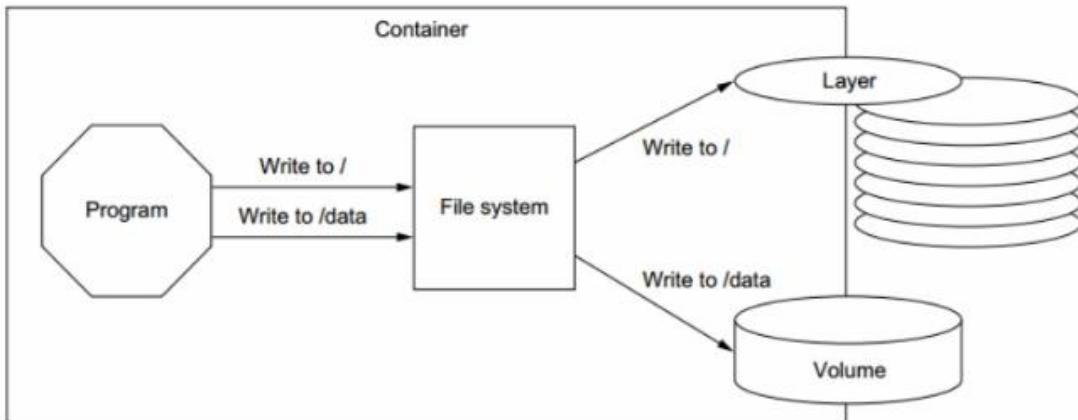
The screenshot shows the AliCloud Container Registry (CR) console. On the left, there's a sidebar with navigation links like '容器镜像服务', '默认实例', '命名空间', '授权管理', '代码源', '访问凭证', '企业版实例', and '镜像中心'. The main area has tabs for '阿里云' (selected), '华东1(杭州)' (region), and a search bar. A prominent yellow banner at the top says '已推出容器镜像服务企业版, 提供企业级镜像安全托管及P2P分发能力, 欢迎申请试用。'. Below the banner, under the '镜像仓库' tab, there's a '创建镜像仓库' button and a dropdown menu set to '全部命名空间'. A table lists three repositories: 'nginx' (namespace: 'linux36'), 'hello' (namespace: 'zhangshijie'), and 'hello-world' (namespace: 'zhangshijie'). Each row includes columns for '仓库名称' (repository name), '命名空间' (namespace), '仓库状态' (repository status), '仓库类型' (repository type), '权限' (permissions), and '仓库地址' (repository address). The 'hello' and 'hello-world' rows are highlighted with a pink border.

仓库名称	命名空间	仓库状态	仓库类型	权限	仓库地址
nginx	linux36	正常	私有	管理	山
hello	zhangshijie	正常	公开	管理	山
hello-world	zhangshijie	正常	公开	管理	山

三：Docker 数据管理：

如果正在运行中的容器修改生成了新的数据或者修改了现有的一个已经存在的文件内容，那么新产生的数据将会被复制到读写层进行持久化保存，这个读写层也就是容器的工作目录，此即“写时复制(COW) copy on write”机制。

如下图是将对根的数据写入到了容器的可写层，但是把 /data 中的数据写入到了一个另外的 volume 中用于数据持久化。



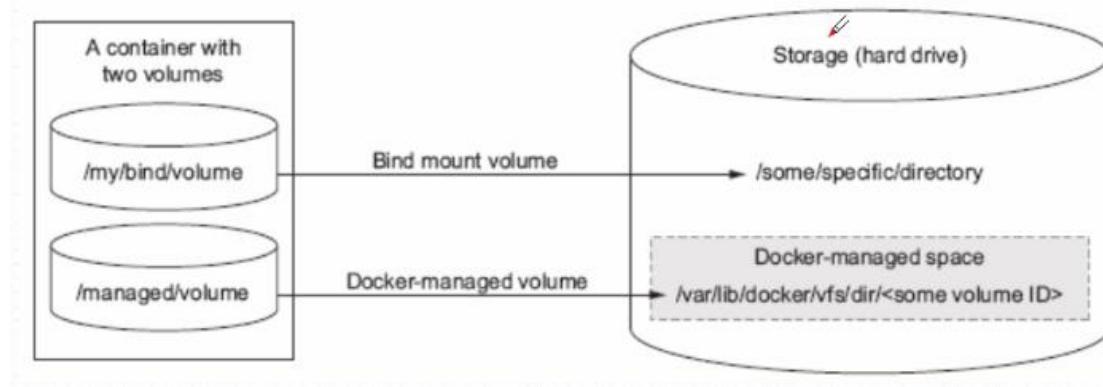
3.1: 数据类型:

Docker 的镜像是分层设计的，镜像层是只读的，通过镜像启动的容器添加了一层可读写的文件系统，用户写入的数据都保存在这一层当中。

如果要将写入到容器的数据永久保存，则需要将容器中的数据保存到宿主机的指定目录，目前 Docker 的数据类型分为两种：

一是数据卷 (data volume)，数据卷类似于挂载的一块磁盘，数据容器是将数据保存在一个容器上。

二是数据卷容器 (Data volume container)，数据卷容器是将宿主机的目录挂载至一个专门的数据卷容器，然后让其他容器通过数据卷容器读写宿主机的数据。



```
root@s1:~# docker inspect f55c55544e05 #查看指定 PID 的容器信息
```

```
"GraphDriver": {
    "Data": {
        "LowerDir": "/var/lib/docker/overlay2/d64676c6143b28b64c41c9155f67d3a79804fd1ba
fdaa586ffdd39e3c68fd385-init:diff:/var/lib/docker/overlay2/cb1f2fa35b6507872ba4df267d7898e1484e
388c632b2838cc18e1a7666bdbb5/diff",
        "MergedDir": "/var/lib/docker/overlay2/d64676c6143b28b64c41c9155f67d3a79804fd1ba
afdaa586ffdd39e3c68fd385/merged",
        "UpperDir": "/var/lib/docker/overlay2/d64676c6143b28b64c41c9155f67d3a79804fd1ba
fdaa586ffdd39e3c68fd385/diff",
        "WorkDir": "/var/lib/docker/overlay2/d64676c6143b28b64c41c9155f67d3a79804fd1ba
daa586ffdd39e3c68fd385/work"
    }
}
```

Lower Dir: image 镜像层(镜像本身, 只读)
Upper Dir: 容器的上层(读写)
Merged Dir: 容器的文件系统, 使用 Union FS(联合文件系统)将 lowerdir 和 upper
Dir: 合并给容器使用。
Work Dir: 容器在宿主机的工作目录

在容器生成数据:

```
# docker exec -it fa01784532f0 bash
[root@fa01784532f0 /]# dd if=/dev/zero of=file bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (105 MB) copied, 0.257476 s, 407 MB/s

[root@fa01784532f0 /]# md5sum file
2f282b84e7e608d5852449ed940bfc51  file

# cp anaconda-post.log  /opt/
```

数据在宿主机哪里?

```
root@docker-server1:~# ll /var/lib/docker/overlay2/f46a7c5dc7bfa7ca33af7783dc0fd08a3b440fd201b3
726cf428fdffbd276b2d/diff/
total 102400
drwxr-xr-x 7 root root    75 Sep 12 16:59 .
drwx----- 5 root root    69 Sep 12 15:10 ..
drwxr-xr-x 3 2019 2019    34 Sep 12 14:49 apps/
-rw-r--r-- 1 root root 104857600 Sep 12 16:59 file
drwxr-xr-x 3 root root    46 Sep 12 17:00 opt/
dr-xr-x-- 2 root root    27 Sep 12 16:46 root/
drwxrwxrwt 3 root root    28 Sep 12 15:10 tmp/
drwxr-xr-x 3 root root    17 Aug  1 09:09 var/
root@docker-server1:~# md5sum /var/lib/docker/overlay2/f46a7c5dc7bfa7ca33af7783dc0fd08a3b440fd201b3
01b3726cf428fdffbd276b2d/diff/file
2f282b84e7e608d5852449ed940bfc51  /var/lib/docker/overlay2/f46a7c5dc7bfa7ca33af7783dc0fd08a3b440fd201b3
0fd201b3726cf428fdffbd276b2d/diff/file ← 验证file文件的md5 值和容器中是否一致
root@docker-server1:~#
root@docker-server1:~# ll /var/lib/docker/overlay2/f46a7c5dc7bfa7ca33af7783dc0fd08a3b440fd201b3
726cf428fdffbd276b2d/opt/
total 12
drwxr-xr-x 3 root root    46 Sep 12 17:00 .
drwxr-xr-x 7 root root    75 Sep 12 16:59 ..
-rw-r--r-- 1 root root 12090 Sep 12 17:00 anaconda-post.log
drwxr-xr-x 2 root root     6 Sep 12 16:44 linux37/
root@docker-server1:~#
```

当容器被删除后, 宿主机的数据还在吗?

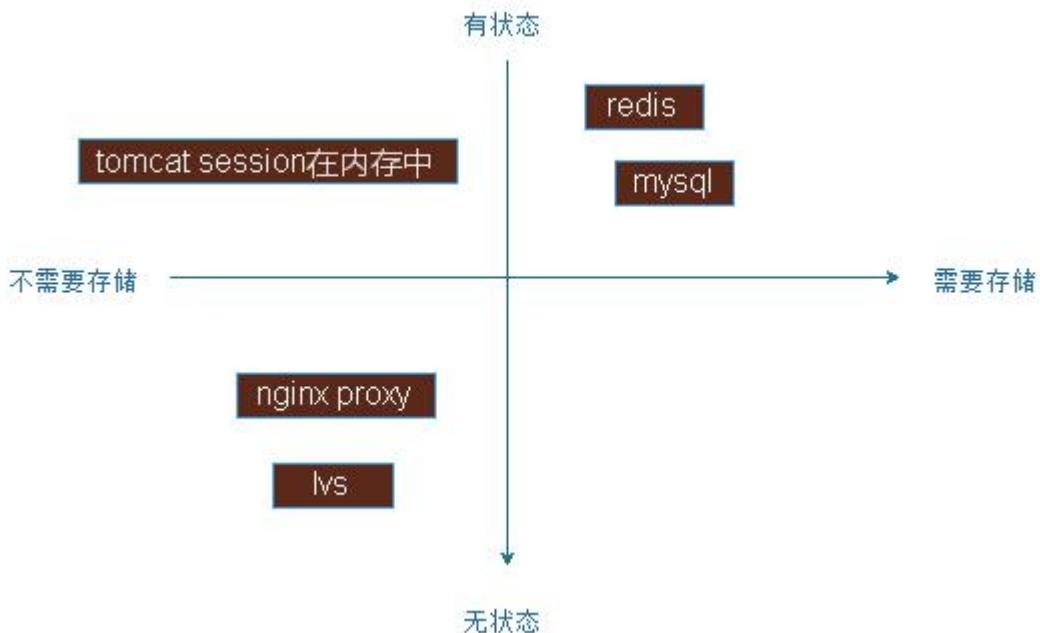
3.1.1: 什么是数据卷(data volume):

数据卷实际上就是宿主机上的目录或者是文件, 可以被直接 mount 到容器当中使用。

实际生产环境中, 需要针对不同类型的服务、不同类型的数据存储要求做相应的规划, 最终保证服务的可扩展性、稳定性以及数据的安全性。

如下图:

左侧是无状态的 http 请求服务，右侧为有状态。
下层为不需要存储的服务，上层为需要存储的部分服务。



3.1.1.1: 创建 APP 目录并生成 web 页面:

此 app 以数据卷的方式，提供给容器使用，比如容器可以直接宿主机本地的 web app，而需要将代码提前添加到容器中，此方式适用于小型 web 站点。

```
# mkdir /data/testapp -p  
# echo "testapp page" > /data/testapp/index.html  
# cat /data/testapp/index.html  
testapp page
```

3.1.1.2: 启动容器并验证数据:

启动两个容器，web1 容器和 web2 容器，分别测试能否在宿主机访问到宿主机的数据。

#注意使用-v 参数，将宿主机目录映射到容器内部，web2 的 ro 标示在容器内对该目录只读，默认是可读写的：

```
root@docker-server1:~# docker run -d --name web1 -v  
/data/testapp/:/apps/tomcat/webapps/testapp -p 8080:8080 tomcat-web:app1  
463bf7548b724c74c99d2c583831af680a0b9575a1d62206311898a759bc823b  
  
# docker run -d --name web2 -v /data/testapp/:/apps/tomcat/webapps/testapp  
:ro -p 8081:8080 tomcatweb:app2  
999ed545ca948551d3a51338a0080c1ea45b3f8446e7454b880e8c8cde269ba8
```

3.1.1.3: 进入到容器内测试写入数据:

```
root@docker-server1:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
999ed545ca94        tomcat-web:app2    "/apps/tomcat/bin/ru..."   2 minutes ago      Up 2 minutes
463bf7548b72        tomcat-web:app1    "/apps/tomcat/bin/ru..."   3 minutes ago      Up 3 minutes

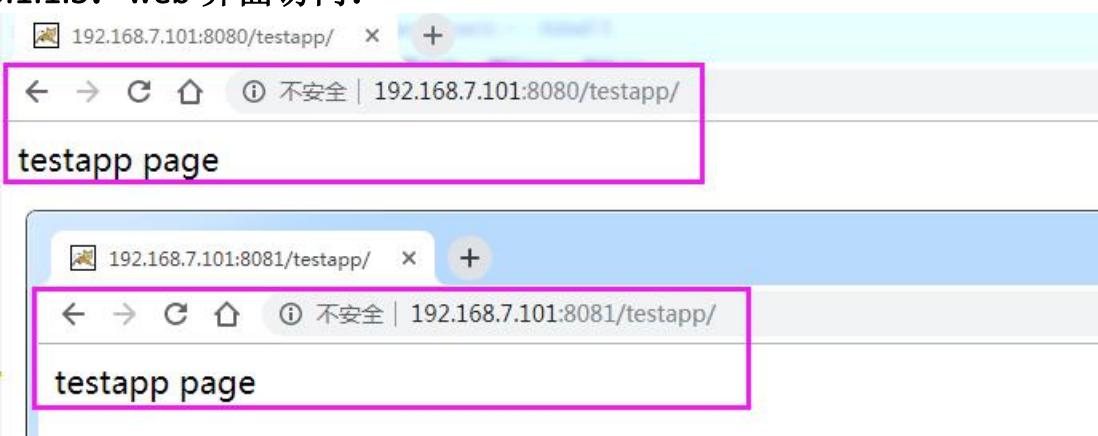
root@docker-server1:~# docker exec -it 463bf7548b72 bash
[root@463bf7548b72 /]# cat /apps/tomcat/webapps/testapp/index.html
testapp page
[root@463bf7548b72 /]# exit
exit
root@docker-server1:~# docker exec -it 999ed545ca94 bash
[root@999ed545ca94 /]# cat /apps/tomcat/webapps/testapp/index.html
testapp page
[root@999ed545ca94 /]# exit
exit
root@docker-server1:~#
```

3.1.1.4: 宿主机验证:

验证宿主机的数据是否正常

```
root@docker-server1:~# cat /data/testapp/index.html
testapp page
root@docker-server1:~#
```

3.1.1.5: web 界面访问:



3.1.1.6: 在宿主机或容器修改数据:

```
# echo "web v2" >> /data/testapp/index.html
# cat /data/testapp/index.html
testapp page
web v2
```

3.1.1.7: web 端访问验证数据:



3.1.1.8: 删除容器:

#创建容器的时候指定参数-v，可以删除/var/lib/docker/containers/的容器数据目录，但是不会删除数据卷的内容，如下：

```
root@docker-server1:~# docker ps
CONTAINER ID        IMAGE               COMMAND
999ed545ca94      tomcat-web:app2    "/apps/tomcat/bin/ru... "
463bf7548b72      tomcat-web:app1    "/apps/tomcat/bin/ru... "
root@docker-server1:~# docker rm -fv 999ed545ca94
999ed545ca94
root@docker-server1:~# docker rm -fv 463bf7548b72
463bf7548b72
root@docker-server1:~#
```

3.1.1.9: 验证宿主机的数据:

```
# cat /data/testapp/index.html
testapp page
web v2
```

3.1.1.10: 数据卷的特点及使用:

- 1、数据卷是宿主机的目录或者文件，并且可以在多个容器之间共同使用。
- 2、在宿主机对数据卷更改数据后会在所有容器里面会立即更新。
- 3、数据卷的数据可以持久保存，即使删除使用该容器卷的容器也不影响。
- 4、在容器里面的写入数据不会影响到镜像本身。

3.1.2: 文件挂载:

文件挂载用于很少更改文件内容的场景，比如 nginx 的配置文件、tomcat 的配置文件等。

3.1.2.1: 创建容器并挂载配置文件:

```
# ll /data/testapp/
total 28
drwxr-xr-x 2 root root   43 Sep 12 17:40 ./
```

```
drwxr-xr-x 4 root root 35 Sep 12 17:10 ../  
-rwxr-xr-x 1 root root 23611 Aug 5 2018 catalina.sh*  
-rw-r--r-- 1 root root 20 Sep 12 17:22 index.html
```

#自定义 JAVA 选项参数：

```
JAVA_OPTS="-server -Xms4g -Xmx4g -Xss512k -Xmn1g  
-XX:CMSInitiatingOccupancyFraction=65 -XX:+UseFastAccessorMethods  
-XX:+AggressiveOpts -XX:+UseBiasedLocking -XX:+DisableExplicitGC  
-XX:MaxTenuringThreshold=10 -XX:NewSize=2048M -XX:MaxNewSize=2048M  
-XX:NewRatio=2 -XX:PermSize=128m -XX:MaxPermSize=512m  
-XX:CMSFullGCsBeforeCompaction=5 -XX:+ExplicitGCI invokes Concurrent  
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled  
-XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m  
-XX:+UseFastAccessorMethods"
```

创建容器：

```
# docker run -it -d -p 8080:8080 -v  
/data/testapp/catalina.sh:/apps/tomcat/bin/catalina.sh:ro tomcat-web:app2  
2e90299a39163d7c3af97888b84f02f2c3481529fcf6a9da8f06fbb92bd05f7c
```

3.1.2.2：验证参数生效：

```
root@docker-server1:~# docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS  
PORTS NAMES  
2e90299a3916 tomcat-web:app2 "/apps/tomcat/bin/run_... " 51 seconds ago Up 49 sec  
8009/tcp, 0.0.0.0:8080->8080/tcp compassionate_feynman  
root@docker-server1:~# ps -ef | grep tomcat  
root 41430 41403 0 17:43 pts/0 00:00:00 /bin/bash /apps/tomcat/bin/run_tomcat.sh  
2019 41503 41430 6 17:43 ? 00:00:03 /usr/local/jdk/bin/java -Djava.util.logging.config.file=/apps/tomcat/conf/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -server -Xms1g -Xmx1g -Xss512k -Xmn1g -XX:CMSInitiatingOccupancyFraction=65 -XX:+UseFastAccessorMethods -XX:+AggressiveOpts -XX:+UseBiasedLocking -XX:+DisableExplicitGC -XX:MaxTenuringThreshold=10 -XX:NewSize=2048M -XX:MaxNewSize=2048M -XX:NewRatio=2 -XX:PermSize=128m -XX:MaxPermSize=512m -XX:CMSFullGCsBeforeCompaction=5 -XX:+ExplicitGCI invokes Concurrent -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -Djdk.tls.ephemeralDHKeySize=2048 -Djava.protocol.handler.pkgs=org.apache.catalina.webresources -Dignore.endorsed.dirs=-classpath /apps/tomcat/bin/bootstrap.jar:/apps/tomcat/bin/tomcat-juli.jar -Dcatalina.base=/apps/tomcat -Dcatalina.home=/apps/tomcat -Djava.io.tmpdir=/apps/tomcat/temp org.apache.catalina.startup.Bootstrap start  
root 41585 32727 0 17:43 pts/1 00:00:00 grep --color=auto tomcat  
root@docker-server1:~#
```

3.1.2.3：进入容器测试文件读写：

```
root@docker-server1:~# docker exec -it 2e90299a3916 bash  
[root@2e90299a3916 /]# echo "test" >> /apps/tomcat/bin/catalina.sh  
bash: /apps/tomcat/bin/catalina.sh: Read-only file system  
[root@2e90299a3916 /]#
```

3.1.2.3：如何一次挂载多个目录：

多个目录可以位于不同的目录下

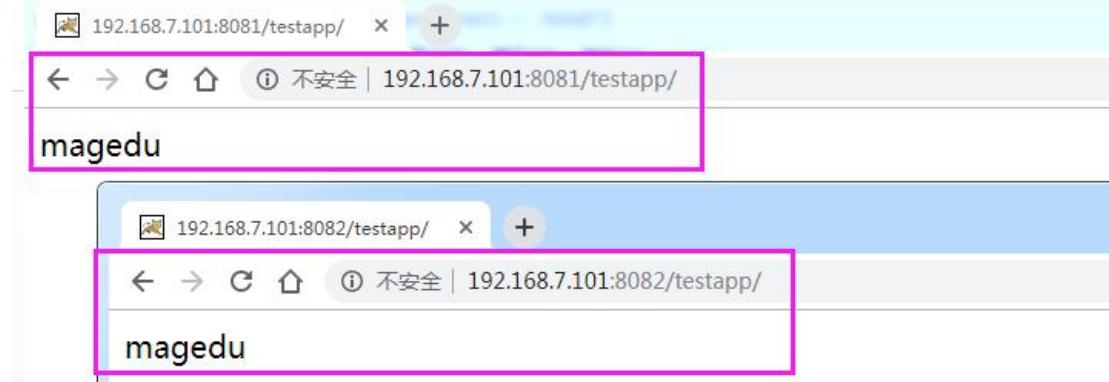
```
root@docker-server1:~# mkdir /data/magedu  
root@docker-server1:~# echo "magedu" >> /data/magedu/index.html
```

```
# docker run -d --name web1 -p 8081:8080 -v  
/data/testapp/catalina.sh:/apps/tomcat/bin/catalina.sh:ro -v  
/data/magedu:/apps/tomcat/webapps/testapp t  
omcatweb:app1  
491b5acd8426d125dc747b6a90990513d655fac4573597bd13afb330ca1c2dd4
```

再启动一个容器，验证宿主机目录或文件是否共享：

```
~# docker run -d --name web2 -p 8082:8080 -v  
/data/testapp/catalina.sh:/apps/tomcat/bin/catalina.sh:ro -v  
/data/magedu:/apps/tomcat/webapps/testapp t  
omcat-web:app2  
b07b9c09c8cadca8a8ef9e4e5c3969f3713a47280e9e5608fb8bd4e11ec45128
```

3.2.1.4：验证 web 访问：



3.1.2.5：数据卷使用场景：

- 1、日志输出
- 2、静态 web 页面
- 3、应用配置文件
- 4、多容器间目录或文件共享

3.1.3：数据卷容器：

数据卷容器功能是可以让数据在多个 docker 容器之间共享，即可以让 B 容器访问 A 容器的内容，而容器 C 也可以访问 A 容器的内容，即先要创建一个后台运行的容器作为 Server，用于卷提供，这个卷可以为其他容器提供数据存储服务，其他使用此卷的容器作为 client 端：

3.1.3.1：启动一个卷容器 Server：

先启动一个容器，并挂载宿主机的数据目录：

将宿主机的 catalina.sh 启动脚本和 magedu 的 web 页面，分别挂载到卷容器 server

端，然后通过 server 端共享给 client 端使用。

```
# docker rm -fv web1 web2

# docker run -d --name volume-server -v
/data/testapp/catalina.sh:/apps/tomcat/bin/catalina.sh:ro -v
/data/magedu:/apps/tomcat/webapps/magedu tomcat-web:app2

33bc02c22cae0261731e0aa24ae2c663322b7888ed0ba5958b91f71616df19c4
```

3.1.3.2: 启动两个端容器 Client:

```
# docker run -d --name web1 -p 8801:8080 --volumes-from volume-server
tomcat-web:app1
ac657e2cd03e12e57b6b093d563120279556138018e0bbc9655dda59507fed46

# docker run -d --name web2 -p 8802:8080 --volumes-from volume-server
tomcat-web:app2
ec50686a98fa7bbb0a3b53ed27f18d166236f132ab187c82bad37eb1ab329fa0
```

3.1.3.3: 分别进入容器测试读写:

读写权限依赖于源数据卷 Server 容器

```
root@docker-server1:~# docker exec -it ac657e2cd03e bash
[root@ac657e2cd03e /]# cat /apps/tomcat/webapps/magedu/index.html
magedu
[root@ac657e2cd03e /]# echo "magedu v2" >> /apps/tomcat/webapps/magedu/index.html
[root@ac657e2cd03e /]# 可写文件权限
[root@ac657e2cd03e /]# echo "magedu v2" >> /apps/tomcat/bin/catalina.sh
bash: /apps/tomcat/bin/catalina.sh: Read-only file system
[root@ac657e2cd03e /]# 只读文件
```

3.1.3.4: 测试访问 web 页面:



3.1.3.5: 验证宿主机数据:

```
# cat /data/magedu/index.html
magedu
magedu v2
```

3.1.3.4: 关闭卷容器 Server 测试能否启动新容器:

```
# docker stop volume-server
volume-server

#测试停止完成之后能否创建新的容器, 停止 volume server 是可以创建新容器的。
# docker run -d --name web3 -p 8803:8080 --volumes-from volume-server
tomcat-web:app2
1ab11ec011668bb04225702961bb1a0b520a47a7de49134cd7b301cffeaaf5dd
```

#停止 volume server 可以创建新容器

```
root@docker-server1:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
1ab11ec01166        tomcat-web:app2   "/apps/tomcat/bin/ru..."   2 minutes ago     Up 2 minutes      8009/tcp, 0.0.0.0:8803->8080/tcp   web3
ec50686a98fa        tomcat-web:app2   "/apps/tomcat/bin/ru..."   11 minutes ago    Up 11 minutes     8009/tcp, 0.0.0.0:8802->8080/tcp   web2
ac657e2cd03e        tomcat-web:app1   "/apps/tomcat/bin/ru..."   11 minutes ago    Up 11 minutes     8009/tcp, 0.0.0.0:8801->8080/tcp   web1
root@docker-server1:~#
```

3.1.3.6: 测试删除源卷容器 Server 创建容器:

将 volume server 删除，然后在测试能否创建基于 volume server 的新容器。

```
# docker rm -fv volume-server
volume-server
```

```
# docker run -d --name web4 -p 8804:8080 --volumes-from volume-server
tomcat-web:app2
docker: Error response from daemon: No such container: volume-server.
See 'docker run --help'.
```

```
root@docker-server1:~# docker rm -fv volume-server
volume-server
root@docker-server1:~#
root@docker-server1:~# docker run -d --name web4 -p 8804:8080 --volumes-from volume-server tomcat-web:app2
docker: Error response from daemon: No such container: volume-server.
See 'docker run --help'.
root@docker-server1:~#
```

3.1.3.7: 测试之前的容器是否正常:

已经运行的容器不受任何影响



3.1.3.8: 重新创建容器卷 Server:

```
# docker run -d --name volume-server -v  
/data/testapp/catalina.sh:/apps/tomcat/bin/catalina.sh:ro -v  
/data/magedu:/apps/tomcat/webapps/magedu tomcat-web:app2  
4cacb3a92de9ec7db948c41db04853793fdab50a2d70d37d31b4d55a7463ce2b
```

#创建出 volume server 之后，就可以创建基于 volume server 的新容器。

```
# docker run -d --name web4 -p 8804:8080 --volumes-from volume-server  
tomcat-web:app2  
244908c41af7fa709b96bbcb95388da1f138c4b7bbbf822ba3ed79929a1cd8b9
```

#在当前环境下，即使把提供卷的容器 Server 删除，已经运行的容器 Client 依然可以使用挂载的卷，因为容器是通过挂载访问数据的，但是无法创建新的卷容器客户端，但是再把卷容器 Server 创建后即可正常创建卷容器 Client，此方式可以用于线上共享数据目录等环境，因为即使数据卷容器被删除了，其他已经运行的容器依然可以挂载使用

#数据卷容器可以作为共享的方式为其他容器提供文件共享，类似于 NFS 共享，可以在生产中启动一个实例挂载本地的目录，然后其他的容器分别挂载此容器的目录，即可保证各容器之间的数据一致性。

四：网络部分：

主要介绍 docker 网络相关知识。

Docker 服务安装完成之后，默认在每个宿主机会生成一个名称为 docker0 的网卡其 IP 地址都是 172.17.0.1/16，并且会生成三种不能类型的网络，如下图：

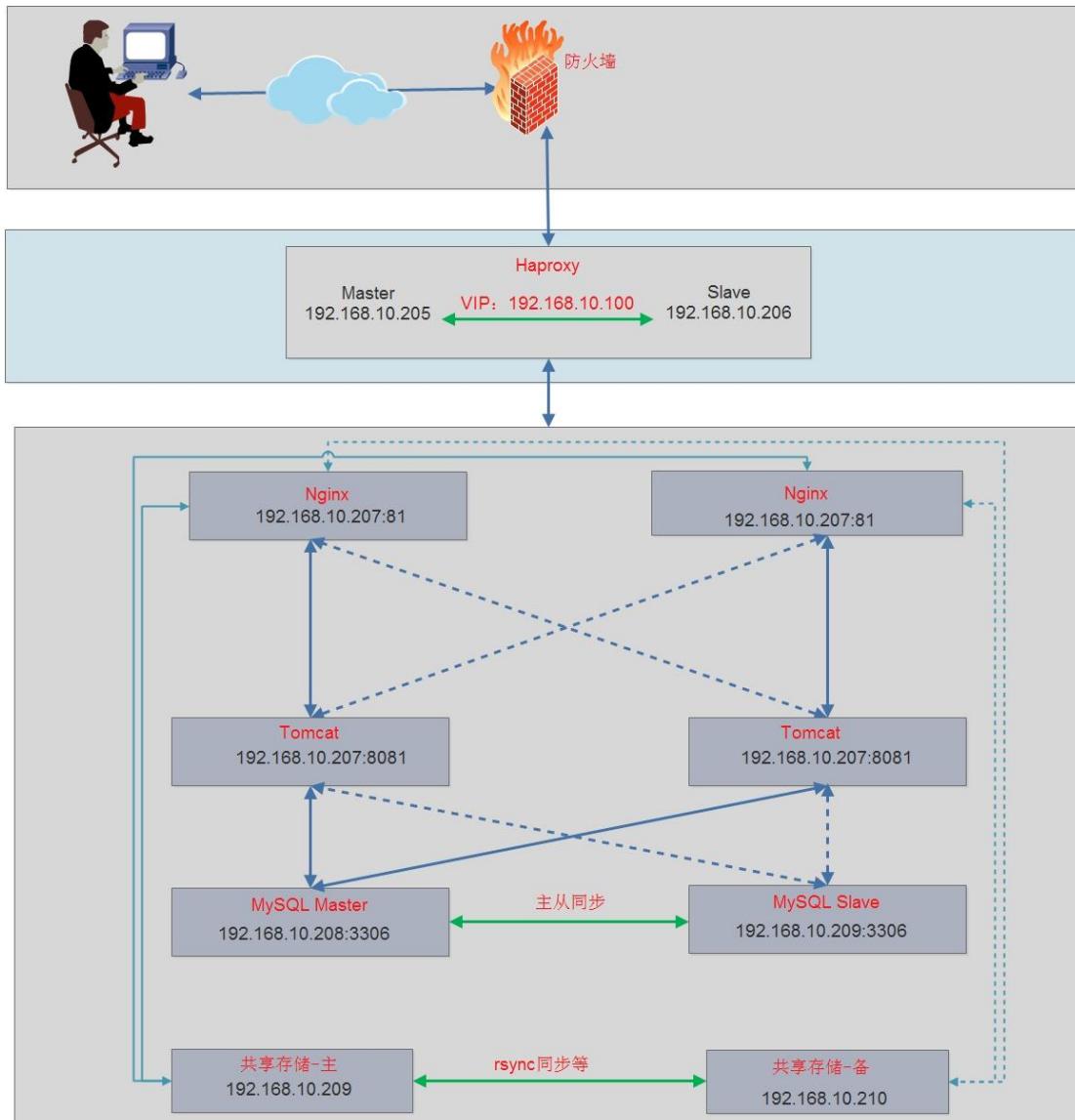
```
root@docker-server1:~# ifconfig docker0
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
      inet6 fe80::42:b5ff:fe2e:6447 prefixlen 64 scopeid 0x20<link>
        ether 02:42:b5:2e:64:47 txqueuelen 0 (Ethernet)
          RX packets 107435 bytes 5967418 (5.9 MB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 209379 bytes 949787805 (949.7 MB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
root@docker-server1:~# docker network list
NETWORK ID      NAME      DRIVER      SCOPE
96b8c5310386    bridge    bridge      local
06be5dcfad98    host      host       local
9374ba0c4a30    none     null       local
root@docker-server1:~#
```

4.1: docker 结合负载实现网站高可用:

4.1.1: 整体规划图:

下图为一个小型的网络架构图，其中 nginx 使用 docker 运行。



4.1.2: 安装并配置 keepalived:

4.1.2.1: Server1 安装并配置:

```
[root@docker-server1 ~]# yum install keepalived -y
[root@docker-server1 ~]# cat /etc/keepalived/keepalived.conf
vrrp_instance MAKE_VIP_INT {
    state MASTER
    interface eth0
    virtual_router_id 1
    priority 100
    advert_int 1
    unicast_src_ip 192.168.10.205
    unicast_peer {
        192.168.10.206
    }
}
```

```
authentication {
    auth_type PASS
    auth_pass 1111
}
virtual_ipaddress {
    192.168.10.100/24 dev eth0 label eth0:1
}
}
[root@docker-server1~]# systemctl restart keepalived && systemctl enable
keepalived
```

4.1.2.2: Server2 安装并配置:

```
[root@docker-server2 ~]# yum install keepalived -y
[root@docker-server2 ~]# cat /etc/keepalived/keepalived.conf
vrrp_instance MAKE_VIP_INT {
    state BACKUP
    interface eth0
    virtual_router_id 1
    priority 50
    advert_int 1
    unicast_src_ip 192.168.10.206
    unicast_peer {
        192.168.10.205
    }

    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.10.100/24 dev eth0 label eth0:1
    }
}
[root@docker-server2 ~]# systemctl restart keepalived && systemctl enable
keepalived
```

4.1.3: 安装并配置 haproxy:

4.1.3.1.: 各服务器配置内核参数:

```
[root@docker-server1 ~]# sysctl -w net.ipv4.ip_nonlocal_bind=1
[root@docker-server2 ~]# sysctl -w net.ipv4.ip_nonlocal_bind=1
```

4.1.3.2: Server1 安装并配置 haproxy:

```
[root@docker-server1 ~]# yum install haproxy -y
[root@docker-server1 ~]# cat /etc/haproxy/haproxy.cfg
global
maxconn 100000
uid 99
gid 99
daemon
nbproc 1
log 127.0.0.1 local0 info

defaults
option http-keep-alive
#option forwardfor
maxconn 100000
mode tcp
timeout connect 500000ms
timeout client 500000ms
timeout server 500000ms

listen stats
mode http
bind 0.0.0.0:9999
stats enable
log global
stats uri      /haproxy-status
stats auth     haadmin:q1w2e3r4ys

=====
frontend docker_nginx_web
    bind 192.168.10.100:80
    mode http
    default_backend docker_nginx_hosts

backend docker_nginx_hosts
    mode http
    #balance source
    balance roundrobin
    server 192.168.10.205    192.168.10.205:81 check inter 2000 fall 3 rise 5
    server 192.168.10.206    192.168.10.206:81 check inter 2000 fall 3 rise 5
```

4.1.3.3: Server2 安装并配置 haproxy:

```
[root@docker-server2 ~]# yum install haproxy -y
[root@docker-server2 ~]# cat /etc/haproxy/haproxy.cfg
```

```

global
maxconn 100000
uid 99
gid 99
daemon
nbproc 1
log 127.0.0.1 local0 info

defaults
option http-keep-alive
#option forwardfor
maxconn 100000
mode tcp
timeout connect 500000ms
timeout client 500000ms
timeout server 500000ms

listen stats
    mode http
    bind 0.0.0.0:9999
    stats enable
    log global
    stats uri      /haproxy-status
    stats auth     haadmin:q1w2e3r4ys

#####
frontend docker_nginx_web
    bind 192.168.10.100:80
    mode http
    default_backend docker_nginx_hosts

backend docker_nginx_hosts
    mode http
    #balance source
    balance roundrobin
    server 192.168.10.205  192.168.10.205:81 check inter 2000 fall 3 rise 5
    server 192.168.10.206  192.168.10.206:81 check inter 2000 fall 3 rise 5

```

4.1.3.4: 各服务器别分启动 haproxy:

```

[root@docker-server1 ~]# systemctl enable haproxy
Created symlink from /etc/systemd/system/multi-user.target.wants/haproxy.service
to /usr/lib/systemd/system/haproxy.service.
[root@docker-server1 ~]# systemctl restart haproxy

```

```
[root@docker-server2 ~]# systemctl enable haproxy  
Created symlink from /etc/systemd/system/multi-user.target.wants/haproxy.service  
to /usr/lib/systemd/system/haproxy.service.  
[root@docker-server2 ~]# systemctl restart haproxy
```

4.1.4: 服务器启动 nginx 容器并验证:

4.1.4.1: Server1 启动 Nginx 容器:

从本地 Nginx 镜像启动一个容器，并指定端口，默认协议是 tcp 方式

```
[root@docker-server1 ~]# docker rm -f `docker ps -a -q` #先删除之前所有的容器  
[root@docker-server1 ~]# docker run --name nginx-web1 -d -p 81:80  
jack/nginx-1.10.3:v1 nginx  
5410e4042f731d2abe100519269f9241a7db2b3a188c6747b28423b5a584d020
```

4.1.4.2: 验证端口:

```
[root@docker-server1 ~]# ss -tnl  
State      Recv-Q Send-Q          Local Address:Port  
LISTEN      0      128              192.168.10.100:80  
LISTEN      0      128              *:22  
LISTEN      0      100              127.0.0.1:25  
LISTEN      0      128              :::81  
LISTEN      0      128              :::22  
LISTEN      0      100              ::1:25  
[root@docker-server1 ~]#
```

4.1.4.3: 验证 web 访问:



← → C ⌂ ⓘ 192.168.10.205:81

test nginx page

4.1.4.3: Server2 启动 nginx 容器:

```
[root@docker-server2 ~]# docker run --name nginx-web1 -d -p 81:80  
jack/nginx-1.10.3:v1 nginx  
84f2376242e38d7c8ba7fabf3134ac0610ab26358de0100b151df6a231a2b56a
```

4.1.4.4: 验证端口:

```
[root@docker-server2 ~]# docker run --name nginx-web1 -d -p 81:80 jack/nginx-1.10.3:v1 nginx  
84f2376242e38d7c8ba7fabf3134ac0610ab26358de0100b151df6a231a2b56a  
[root@docker-server2 ~]# ss -tnl  
State      Recv-Q Send-Q          Local Address:Port  
LISTEN      0      128              192.168.10.100:80  
LISTEN      0      128              *:22  
LISTEN      0      100              127.0.0.1:25  
LISTEN      0      128              :::81  
LISTEN      0      128              :::22  
LISTEN      0      100              ::1:25  
[root@docker-server2 ~]#
```

4.1.4.5: 验证 web 访问:

test nginx page

4.1.4.6: 访问 VIP:

test nginx page

4.1.4.7: Server1 haproxy 状态页面:

General process information

```
pid = 4032 (process #1, nbproc = 1)
uptime = 0d 0h0m14s
system limits: memmax = unlimited; ulimit-n = 200014
maxsock = 200014; maxconn = 100000; maxpipes = 0
current conn = 1; current pipes = 0/0; conn rate = 0/sec
Running tasks: 1/7; idle = 100 %
```

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

stats																						
Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings				
Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
Frontend	0	2	-	1	2	100 000	2	-	400	262	0	0	0	0	0	0	0	0	0	0	OPEN	
Backend	0	0	0	0	0	10 000	0	0	400	262	0	0	0	0	0	0	0	0	0	0	1m41s UP	

docker_nginx_web																						
Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings				
Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
Frontend	0	0	-	0	0	100 000	0	-	0	0	?	0	0	0	0	0	0	0	0	0	OPEN	

docker_nginx_hosts																						
Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings				
Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
192.168.10.205	0	0	-	0	0	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	1m41s UP	L4OK in 0ms
192.168.10.206	0	0	-	0	0	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	1m41s UP	L4OK in 0ms
Backend	0	0	0	0	0	10 000	0	0	0	0	?	0	0	0	0	0	0	0	0	0	1m41s UP	

4.1.4.8: Server2 haproxy 状态页面:

General process information

```
pid = 3018 (process #1, nbproc = 1)
uptime = 0d 0h0m16s
system limits: memmax = unlimited; ulimit-n = 200014
maxsock = 200014; maxconn = 100000; maxpipes = 0
current conn = 1; current pipes = 0/0; conn rate = 0/sec
Running tasks: 1/7; idle = 100 %
```

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

stats																						
Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings				
Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
Frontend	0	2	-	1	2	100 000	2	-	400	262	0	0	0	0	0	0	0	0	0	0	OPEN	
Backend	0	0	0	0	0	10 000	0	0	400	262	0	0	0	0	0	0	0	0	0	0	3m16s UP	

docker_nginx_web																						
Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings				
Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
Frontend	0	0	-	0	0	100 000	0	-	0	0	0	0	0	0	0	0	0	0	0	0	OPEN	

docker_nginx_hosts																						
Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings				
Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
192.168.10.205	0	0	-	0	0	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	3m16s UP	L4OK in 0ms
192.168.10.206	0	0	-	0	0	0	0	-	0	0	?	0	0	0	0	0	0	0	0	0	3m16s UP	L4OK in 0ms
Backend	0	0	0	0	0	10 000	0	0	0	0	?	0	0	0	0	0	0	0	0	0	3m16s UP	

日志可以在 nginx 里面通过 syslog 传递给 elk 收集

指定 IP、协议和端口：

```
[root@linux-docker ~]# docker run --name nginx-web -d -p
```

```
192.168.10.22:80:80/tcp jack/centos-nginx nginx
```

```
[root@linux-docker ~]# docker run --name nginx-web-udp -d -p  
192.168.10.22:54:53/udp jack/centos-nginx nginx
```

4.2：容器之间的互联：

4.2.1：通过容器名称互联：

即在同一个宿主机上的容器之间可以通过自定义的容器名称相互访问，比如一个业务前端静态页面是使用 nginx，动态页面使用的是 tomcat，由于容器在启动的时候其内部 IP 地址是 DHCP 随机分配的，所以如果通过内部访问的话，自定义名称是相对比较固定的，因此比较适用于此场景。

```
#此方式最少需要两个容器之间操作：
```

```
#
```

4.2.1.1：先创建第一个容器，后续会使用到这个容器的名称：

```
# docker run -it -d --name tomcat-web1 -p 8801:8080 tomcat-web:app1  
96fd3426c786b032f252b709d4bb483590de8e57a99f401821634e4bd0045577
```

4.2.1.2：查看当前 hosts 文件内容：

```
# docker exec -it 96fd3426c786 bash  
[root@96fd3426c786 /]# cat /etc/hosts  
127.0.0.1 localhost  
::1 localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
ff00::0 ip6-mcastprefix  
ff02::1 ip6-allnodes  
ff02::2 ip6-allrouters  
172.17.0.2 96fd3426c786  
1.1.1.1 abc.test.com
```

4.2.1.3：创建第二个容器：

```
# docker run -it -d -p 80:80 --name magedu-nginx-web1 --link tomcat-web1  
magedu-nginx:v1  
e7796ad98c84d7e6148fd25e10c7026bdbe9a21fd5699995912340ab8906b9fc
```

4.2.1.4：查看第二个容器的 hosts 文件内容：

```
# docker exec -it e7796ad98c84 bash  
[root@e7796ad98c84 /]# cat /etc/hosts  
127.0.0.1 localhost  
::1 localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet
```

```
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2 tomcat-web1 96fd3426c786 #连接的对方容器的 ID 和容器名称
172.17.0.3 e7796ad98c84
```

4.2.1.5: 检测通信:

```
[root@e7796ad98c84 /]# ping tomcat-web1
PING tomcat-web1 (172.17.0.2) 56(84) bytes of data.
64 bytes from tomcat-web1 (172.17.0.2): icmp_seq=1 ttl=64 time=0.147 ms
64 bytes from tomcat-web1 (172.17.0.2): icmp_seq=2 ttl=64 time=0.141 ms
^C
--- tomcat-web1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.141/0.144/0.147/0.003 ms
[root@e7796ad98c84 /]# ping 96fd3426c786
PING tomcat-web1 (172.17.0.2) 56(84) bytes of data.
64 bytes from tomcat-web1 (172.17.0.2): icmp_seq=1 ttl=64 time=0.192 ms
64 bytes from tomcat-web1 (172.17.0.2): icmp_seq=2 ttl=64 time=0.150 ms
^C
--- tomcat-web1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.150/0.171/0.192/0.021 ms
[root@e7796ad98c84 /]#
```

4.2.2: 通过自定义容器别名互联:

上一步骤中，自定义的容器名称可能后期会发生变化，那么一旦名称发生变化，程序之间也要随之发生变化，比如程序通过容器名称进行服务调用，但是容器名称发生变化之后再使用之前的名称肯定是无法成功调用，每次都进行更改的话又比较麻烦，因此可以使用自定义别名的方式解决，即容器名称可以随意更，只要不更改别名即可，具体如下：

命令格式：

```
docker run -d --name 新容器名称 --link 目标容器名称:自定义的名称 -p 本地端口:容器端口 镜像名称 shell 命令
```

4.2.2.1: 启动第三个容器:

```
# docker run -it -d -p 81:80 --name magedu-nginx-web2 --link
tomcat-web1:java_server magedu-nginx:v1
6acb8a2b366ec31045b19f9c5a00bd7e811d95c5c46202aec09f140bf3420508
```

4.2.2.2: 查看当前容器的 hosts 文件:

```
root@docker-server1:~# docker exec -it 6acb8a2b366e bash
[root@6acb8a2b366e /]# cat /etc/hosts
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2 java_server 96fd3426c786 tomcat-web1
172.17.0.4 6acb8a2b366
```

4.2.2.3: 检查自定义别名通信:

```
[root@6acb8a2b366e /]# ping java_server
PING java_server (172.17.0.2) 56(84) bytes of data.
64 bytes from java_server (172.17.0.2): icmp_seq=1 ttl=64 time=0.273 ms
64 bytes from java_server (172.17.0.2): icmp_seq=2 ttl=64 time=0.138 ms
64 bytes from java_server (172.17.0.2): icmp_seq=3 ttl=64 time=0.134 ms
64 bytes from java_server (172.17.0.2): icmp_seq=4 ttl=64 time=0.139 ms
```

4.2.3.1: docker 网络类型:

Docker 的网络使用 docker network ls 命令看到有三种类型，下面将介绍每一种类型的具体工作方式：

Bridge 模式，使用参数 `-net=bridge` 指定，不指定默认就是 bridge 模式。

#查看当前 docke 的网卡信息：

```
root@docker-server1:~# docker network list
NETWORK ID      NAME      DRIVER      SCOPE
96b8c5310386    bridge    bridge      local
06be5dcfad98    host      host       local
9374ba0c4a30    none     null       local
```

Bridge: #桥接， 使用自定义 IP

Host: #不获取 IP 直接使用物理机 IP，并监听物理机 IP 监听端口

None: #没有网络

4.2.3.1.1: Host 模式:

Host 模式，使用参数 `-net=host` 指定。

启动的容器如果指定了使用 host 模式，那么新创建的容器不会创建自己的虚拟网卡，而是直接使用宿主机的网卡和 IP 地址，因此在容器里面查看到的 IP 信息就是宿主机的信息，访问容器的时候直接使用宿主机 IP+容器端口即可，不过容器的其他资源们必须文件系统、系统进程等还是和宿主机保持隔离。

此模式的网络性能最高，但是各容器之间端口不能相同，适用于运行容器端口比较固定的业务。

为避免端口冲突，先删除所有的容器：

先确认宿主机端口没有占用 80 端口：

```
[root@docker-server1 ~]# ss -tnl
State      Recv-Q Send-Q                               Local Address:Port
LISTEN      0      128                                *:22
LISTEN      0      100                               127.0.0.1:25
LISTEN      0      128                                :::22
LISTEN      0      100                               ::1:25
[root@docker-server1 ~]#
```

#启动一个新容器，并指定网络模式为 host

```
# docker run -d --name net_host --net=host magedu-nginx:v1
7d65d6106ca87d41b6c62677740a1cdd14a870234f7c15d0beac6b306583cff8
```

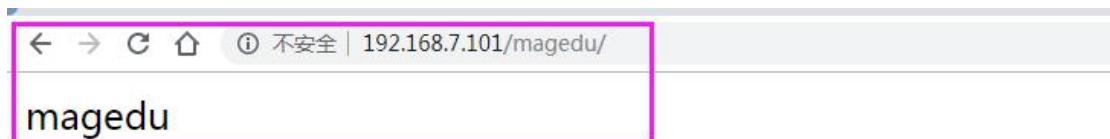
#验证网络信息：

```
root@docker-server1:~# docker run -it -d --name net_host --net=host magedu-nginx:v1 /apps/nginx/sbin/nginx
861e1eb7aaeff17878fc80a386f9dd38af1360f2b5d6d9bcf2624c9d3f06cbf1
root@docker-server1:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
861e1eb7aaef        magedu-nginx:v1   "/apps/nginx/sbin/ng..."   4 seconds ago    Up 3 seconds
root@docker-server1:~# docker exec -it 861e1eb7aaef bash
[root@docker-server1 /]# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
      inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
      inet6 fe80::42:b5ff:fe2e:6447 prefixlen 64 scopeid 0x20<link>
        ether 02:42:b5:2e:64:47 txqueuelen 0 (Ethernet)
          RX packets 107437 bytes 5967474 (5.6 MiB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 209384 bytes 949788235 (905.7 MiB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 192.168.7.101 netmask 255.255.248.0 broadcast 192.168.7.255
      inet6 fe80::20c:29ff:fe44:8078 prefixlen 64 scopeid 0x20<link>
        ether 00:0c:29:44:80:78 txqueuelen 1000 (Ethernet)
          RX packets 1080555 bytes 1447753085 (1.3 GiB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 933917 bytes 1114788169 (1.0 GiB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
      inet6 ::1 prefixlen 128 scopeid 0x10<host>
        loop txqueuelen 1000 (Local Loopback)
          RX packets 194 bytes 18664 (18.2 KiB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 194 bytes 18664 (18.2 KiB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
[root@docker-server1 /]#
```

#访问宿主机验证：



Host 模式不支持端口映射，当指定端口映射的时候会提示如下警告信息：

使用主机网络模式时，将丢弃已指定的端口：

```
# docker run -it -d --name net_host -p 80:80 --net=host magedu-nginx:v1
/apps/nginx/sbin/nginx
```

WARNING: Published ports are discarded when using host network mode

025ff64d057f095032ac6c271d5275d81cea1f73645e93877c3d696cb2280020

4.2.3.1.2: none 模式:

None 模式，使用参数 `--net=none` 指定

在使用 none 模式后，Docker 容器不会进行任何网络配置，其没有网卡、没有 IP 也没有路由，因此默认无法与外界通信，需要手动添加网卡配置 IP 等，所以极少使用，

命令使用方式：

```
# docker run -it -d --name net_none -p 80:80 --net=none magedu-nginx:v1
/apps/nginx/sbin/nginx
7d7125a2a53e0af9718e6f7b3407b093a1367499188d5493b8c4b4a56dff584
```

```
root@docker-server1:~# docker run -it -d --name net_none -p 80:80 --net=none magedu-nginx:v1 /apps/nginx/sbin/nginx
7d7125a2a53e0af9718e6f7b3407b093a1367499188d5493b8c4b4a56dff584
root@docker-server1:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
7d7125a2a53e        magedu-nginx:v1   "/apps/nginx/sbin/ng..."   4 seconds ago    Up 2 seconds
root@docker-server1:~# docker exec -it 7d7125a2a53e bash
[root@7d7125a2a53e ~]# ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
          loop txqueuelen 1000  (Local Loopback)
          RX packets 0 bytes 0 (0.0 B)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 0 bytes 0 (0.0 B)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
[root@7d7125a2a53e ~]#
```

4.2.3.1.3: Container 模式:

Container 模式，使用参数 `--net=container:名称或 ID` 指定。

使用此模式创建的容器需指定和一个已经存在的容器共享一个网络，而不是和宿主机共享网，新创建的容器不会创建自己的网卡也不会配置自己的 IP，而是和一个已经存在的被指定的容器东西 IP 和端口范围，因此这个容器的端口不能和被指定的端口冲突，除了网络之外的文件系统、进程信息等仍然保持相互隔离，两个容器的进程可以通过 **lo 网卡及容器 IP** 进行通信。

```
# docker rm -fv `docker ps -a -q`  
  
# docker run -it -d --name nginx-web1 -p 80:80 --net=bridge magedu-nginx:v1
/apps/nginx/sbin/nginx
8d6950bcf89f7744b2cc3d53733b4561f4c9e2ec2fb735f75972bedb0a4eb79a  
  
# docker run -it -d --name tomcat-web1 --net=container:nginx-web1
tomcat-web:app1 #直接使用对方的网络，此方式较少使用
bf77a272fe2b88b1888b321c96c95d6a27da8d5297366d6965ac1e3e26560083
```

```

root@docker-server1:~# docker exec -it bf77a272fe2b bash
[root@8d6950bcf89f /]# cat /etc/hosts
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      8d6950bcf89f
1.1.1.1 abc.test.com
[root@8d6950bcf89f /]# ss -tnl
State      Recv-Q Send-Q          Local Address:Port
LISTEN      0      1              127.0.0.1:8005
LISTEN      0     100             *:8009
LISTEN      0     100             *:8080
LISTEN      0     128             *:80
[root@8d6950bcf89f /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 172.17.0.2 netmask 255.255.0.0 broadcast 172.17.255.255
                ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)
                RX packets 20 bytes 1408 (1.3 KiB)
                RX errors 0 dropped 0 overruns 0 frame 0
                TX packets 5 bytes 270 (270.0 B)
                TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

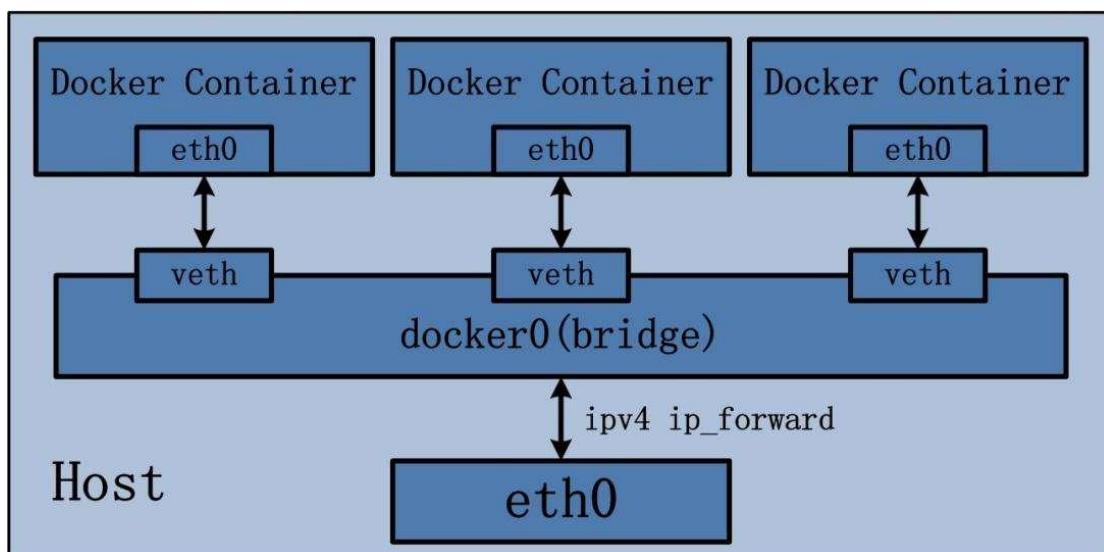
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        loop txqueuelen 1000 (Local Loopback)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

[root@8d6950bcf89f /]#

4.2.3.1.4: bridge 模式:

docker 的默认模式即不指定任何模式就是 bridge 模式，也是使用比较多的模式，此模式创建的容器会为每一个容器分配自己的网络 IP 等信息，并将容器连接到一个虚拟网桥与外界通信。



[root@docker-server1 ~]# docker network inspect bridge

```
root@docker-server1:~# docker network inspect bridge
[{"Name": "bridge",
 "Id": "96b8c531038689b224087eb3370ee109f7c060cbacf9be368b0e8e7ce76a9b16",
 "Created": "2019-09-12T08:55:31.760460181+08:00",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
     "Driver": "default",
     "Options": null,
     "Config": [
         {
             "Subnet": "172.17.0.0/16",
             "Gateway": "172.17.0.1"
         }
     ]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
     "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {
     "8d6950bcf89f7744b2cc3d53733b4561f4c9e2ec2fb735f75972bedb0a4eb79a": {
         "Name": "nginx-web1",
         "EndpointID": "8fe40b36b9aede9c92cd2f40d8f5e18f2d31a9ee068b451fadbf936f0ea763f",
         "MacAddress": "02:42:ac:11:00:02",
         "IPv4Address": "172.17.0.2/16",
         "IPv6Address": ""
     }
 }
}
```

```
# docker rm -fv `docker ps -a -q`  
# docker run -it -d --name nginx-web1 -p 80:80 --net=bridge magedu-nginx:v1  
/apps/nginx/sbin/nginx  
0dfe5a96ef1b7c8f0af820a097a4b140aed8809a787562c694d4982d8f0037b9
```

4.2.3.2: docker 夸主机互联之简单实现:

夸主机互联是说 A 宿主机的容器可以访问 B 主机上的容器，但是前提是保证各宿主机之间的网络是可以相互通信的，然后各容器才可以通过宿主机访问到对方的容器，实现原理是在宿主机做一个网络路由就可以实现 A 宿主机的容器访问 B 主机的容器的目的，复杂的网络或者大型的网络可以使用 google 开源的 k8s 进行互联。

4.2.3.2.1: 修改各宿主机网段:

Docker 的默认网段是 172.17.0.x/24,而且每个宿主机都是一样的，因此要做路由的前提就是各个主机的网络不能一致，具体如下：

问避免影响，先在各服务器删除之前穿件的所有容器。

```
# docker rm -f `docker ps -a -q`
```

4.2.3.2.2: 服务器 A 更改网段:

```
# vim /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
--bip=10.10.0.1/24
```

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --bip=10.10.0.1/24
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

4.2.3.2.3: 重启 docker 服务并验证网卡:

```
root@docker-server1:~# systemctl daemon-reload
root@docker-server1:~# systemctl restart docker
```

验证 A 服务器网卡:

```
root@docker-server1:~# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 10.10.0.1 netmask 255.255.255.0 broadcast 10.10.0.255
        ↓
    inet6 fe80::42:b5ff:fe2e:6447 prefixlen 64 scopeid 0x20<link>
        ether 02:42:b5:2e:64:47 txqueuelen 0 (Ethernet)
        RX packets 107459 bytes 5968318 (5.9 MB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 209416 bytes 949790050 (949.7 MB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

4.2.3.2.4: 服务器 B 更改网段:

```
# vim /lib/systemd/system/docker.service
--bip=10.20.0.1/24
```

```
# systemctl daemon-reload
# systemctl restart docker
```

4.2.3.2.5: 验证网卡:

```
root@docker-node2:~# systemctl daemon-reload
root@docker-node2:~# systemctl restart docker
root@docker-node2:~# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 10.20.0.1 netmask 255.255.255.0 broadcast 10.20.0.255
        ↓
    inet6 fe80::42:fdff:fe18:c409 prefixlen 64 scopeid 0x20<link>
        ether 02:42:fd:18:c4:09 txqueuelen 0 (Ethernet)
        RX packets 1626 bytes 90232 (90.2 KB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 3041 bytes 206554 (206.5 KB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

4.2.3.3: 在两个宿主机分别启动一个容器:

```
#Server1:  
root@docker-server1:~# docker run -it -p 8080:8080 tomcat-web:app1 bash  
[root@781e7f053c20 /]# ifconfig  
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
      inet 10.10.0.2 netmask 255.255.255.0 broadcast 10.10.0.255  
        ether 02:42:0a:0a:00:02 txqueuelen 0 (Ethernet)  
          RX packets 8 bytes 696 (696.0 B)  
          RX errors 0 dropped 0 overruns 0 frame 0  
          TX packets 0 bytes 0 (0.0 B)  
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  
  
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536  
      inet 127.0.0.1 netmask 255.0.0.0  
        loop txqueuelen 1000 (Local Loopback)  
          RX packets 0 bytes 0 (0.0 B)  
          RX errors 0 dropped 0 overruns 0 frame 0  
          TX packets 0 bytes 0 (0.0 B)  
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

#Server2:

```
root@docker-node2:~# docker run -it -p 8080:8080 tomcat-web:app2 bash  
[root@1ee02ccc1810 /]# ifconfig  
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
      inet 10.20.0.2 netmask 255.255.255.0 broadcast 10.20.0.255  
        ether 02:42:0a:14:00:02 txqueuelen 0 (Ethernet)  
          RX packets 8 bytes 696 (696.0 B)  
          RX errors 0 dropped 0 overruns 0 frame 0  
          TX packets 0 bytes 0 (0.0 B)  
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  
  
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536  
      inet 127.0.0.1 netmask 255.0.0.0  
        loop txqueuelen 1000 (Local Loopback)  
          RX packets 0 bytes 0 (0.0 B)  
          RX errors 0 dropped 0 overruns 0 frame 0  
          TX packets 0 bytes 0 (0.0 B)  
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

4.2.3.4: 添加静态路由:

在各宿主机添加静态路由，网关指向对方的 IP:

3.2.3.4.1: Server1 添加静态路由:

```
# route add -net 10.20.0.0/24 gw 192.168.7.102  
# iptables -A FORWARD -s 192.168.0.0/21 -j ACCEPT
```

#ping 对方容器 IP 是否通信:

```
[root@781e7f053c20 /]# ping 10.20.0.2  
PING 10.20.0.2 (10.20.0.2) 56(84) bytes of data.  
64 bytes from 10.20.0.2: icmp_seq=1 ttl=62 time=0.366 ms  
64 bytes from 10.20.0.2: icmp_seq=2 ttl=62 time=0.385 ms  
64 bytes from 10.20.0.2: icmp_seq=3 ttl=62 time=0.414 ms  
64 bytes from 10.20.0.2: icmp_seq=4 ttl=62 time=0.364 ms  
64 bytes from 10.20.0.2: icmp_seq=5 ttl=62 time=0.362 ms  
64 bytes from 10.20.0.2: icmp_seq=6 ttl=62 time=0.624 ms  
64 bytes from 10.20.0.2: icmp_seq=7 ttl=62 time=0.370 ms  
64 bytes from 10.20.0.2: icmp_seq=8 ttl=62 time=0.435 ms  
64 bytes from 10.20.0.2: icmp_seq=9 ttl=62 time=0.427 ms  
64 bytes from 10.20.0.2: icmp_seq=10 ttl=62 time=0.386 ms  
64 bytes from 10.20.0.2: icmp_seq=11 ttl=62 time=1.05 ms  
64 bytes from 10.20.0.2: icmp_seq=12 ttl=62 time=1.09 ms  
64 bytes from 10.20.0.2: icmp_seq=13 ttl=62 time=0.541 ms
```

4.2.3.4.2: server2 添加静态路由:

```
[root@docker-server2 ~]# route add -net 172.16.10.0/24 gw 192.168.10.205  
[root@docker-server2 ~]# iptables -A FORWARD -s 192.168.0.0/21 -j ACCEPT
```

#ping 对方容器 IP 是否通信:

```
[root@docker-server2 ~]# route add -net 172.16.10.0/24 gw 192.168.10.205  
[root@docker-server2 ~]# ping 172.16.10.2  
PING 172.16.10.2 (172.16.10.2) 56(84) bytes of data.  
64 bytes from 172.16.10.2: icmp_seq=1 ttl=63 time=1.77 ms  
64 bytes from 172.16.10.2: icmp_seq=2 ttl=63 time=0.321 ms  
64 bytes from 172.16.10.2: icmp_seq=3 ttl=63 time=0.286 ms  
64 bytes from 172.16.10.2: icmp_seq=4 ttl=63 time=0.540 ms  
^C  
--- 172.16.10.2 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3002ms  
rtt min/avg/max/mdev = 0.286/0.730/1.775/0.611 ms  
[root@docker-server2 ~]#
```

4.2.3.4.3: 抓包分析

```
[root@linux-docker2 ~]# tcpdump -i eth0 -vnn icmp
```

```
[root@docker-server2 ~]# tcpdump -i eth0 -vnn icmp  
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes  
03:07:26.674799 IP (tos 0x0, ttl 64, id 44969, offset 0, flags [DF], proto ICMP (1), length 84)  
    192.168.10.206 > 172.16.10.2: ICMP echo request, id 4803, seq 1, length 64  
03:07:26.675304 IP (tos 0x0, ttl 63, id 36435, offset 0, flags [none], proto ICMP (1), length 84)  
    172.16.10.2 > 192.168.10.206: ICMP echo reply, id 4803, seq 1, length 64  
03:07:27.675231 IP (tos 0x0, ttl 64, id 45532, offset 0, flags [DF], proto ICMP (1), length 84)  
    192.168.10.206 > 172.16.10.2: ICMP echo request, id 4803, seq 2, length 64  
03:07:27.675625 IP (tos 0x0, ttl 63, id 36627, offset 0, flags [none], proto ICMP (1), length 84)  
    172.16.10.2 > 192.168.10.206: ICMP echo reply, id 4803, seq 2, length 64  
03:07:28.675224 IP (tos 0x0, ttl 64, id 46308, offset 0, flags [DF], proto ICMP (1), length 84)  
    192.168.10.206 > 172.16.10.2: ICMP echo request, id 4803, seq 3, length 64  
03:07:28.675563 IP (tos 0x0, ttl 63, id 37360, offset 0, flags [none], proto ICMP (1), length 84)  
    172.16.10.2 > 192.168.10.206: ICMP echo reply, id 4803, seq 3, length 64
```

4.2.3.5: 测试容器间互联:

4.2.3.5.1: 宿主机 A 到宿主机 B 容器测试:

```
[root@docker-server1 ~]# ./docker-in.sh test-net-vm
[root@5d625a868718 /]# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 172.16.10.2 netmask 255.255.255.0 broadcast 0.0.0.0
        inet6 fe80::42:acff:fe10:a02 prefixlen 64 scopeid 0x20<link>
          ether 02:42:ac:10:0a:02 txqueuelen 0 (Ethernet)
            RX packets 41 bytes 3354 (3.2 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 33 bytes 2706 (2.6 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@5d625a868718 /]# ping 172.16.20.2
PING 172.16.20.2 (172.16.20.2) 56(84) bytes of data.
64 bytes from 172.16.20.2: icmp_seq=1 ttl=62 time=1.62 ms
64 bytes from 172.16.20.2: icmp_seq=2 ttl=62 time=0.354 ms
64 bytes from 172.16.20.2: icmp_seq=3 ttl=62 time=0.811 ms
64 bytes from 172.16.20.2: icmp_seq=4 ttl=62 time=0.437 ms
64 bytes from 172.16.20.2: icmp_seq=5 ttl=62 time=0.447 ms
^C
--- 172.16.20.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 0.354/0.734/1.622/0.471 ms
[root@5d625a868718 /]#
```

4.2.3.5.2: 宿主机 B 到宿主机 A 容器测试:

```
[root@docker-server2 ~]# ./docker-in.sh test-net-vm
[root@ce8fbe4c1d6b /]# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 172.16.20.2 netmask 255.255.255.0 broadcast 0.0.0.0
        inet6 fe80::42:acff:fe10:1402 prefixlen 64 scopeid 0x20<link>
          ether 02:42:ac:10:14:02 txqueuelen 0 (Ethernet)
            RX packets 33 bytes 2706 (2.6 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 45 bytes 3882 (3.7 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@ce8fbe4c1d6b /]# ping 172.16.10.2
PING 172.16.10.2 (172.16.10.2) 56(84) bytes of data.
64 bytes from 172.16.10.2: icmp_seq=1 ttl=62 time=0.624 ms
64 bytes from 172.16.10.2: icmp_seq=2 ttl=62 time=0.402 ms
64 bytes from 172.16.10.2: icmp_seq=3 ttl=62 time=0.341 ms
64 bytes from 172.16.10.2: icmp_seq=4 ttl=62 time=0.369 ms
64 bytes from 172.16.10.2: icmp_seq=5 ttl=62 time=0.371 ms
^C
--- 172.16.10.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 0.341/0.421/0.624/0.104 ms
[root@ce8fbe4c1d6b /]#
```

4.3：创建自定义网络：

可以基于 docker 命令创建自定义网络，自定义网络可以自定义 IP 地址范围和网关等信息。

4.3.1：创建自定义 docker 网络：

```
# docker network create --help
# docker network create -d bridge --subnet 10.100.0.0/24 --gateway 10.100.0.1
magedu-net # 创建自定义网络 magedu-net
954c8bda9a8c35cd8e9e76159ef04f29f89f054e72f29e92cc5fc4a7be1cf6da
```

验证网络：

```
# docker network list
```

NETWORK ID	NAME	DRIVER	SCOPE
78b0503ddcf3	bridge	bridge	local
06be5dcfad98	host	host	local
954c8bda9a8c	magedu-net	bridge	local
9374ba0c4a30	none	null	local

4.3.2：创建不同网络的容器测试通信：

4.3.2.1：使用自定义网络创建容器：

```
root@docker-server1:~# docker run -it -p 8080:8080 --name magedu-net-test
--net=magedu-net tomcat-web:app1 bash
[root@42dba1061dd1 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 10.100.0.2 netmask 255.255.255.0 broadcast 10.100.0.255
        ether 02:42:0a:64:00:02 txqueuelen 0 (Ethernet)
          RX packets 8 bytes 696 (696.0 B)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 0 bytes 0 (0.0 B)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
        loop txqueuelen 1000 (Local Loopback)
          RX packets 0 bytes 0 (0.0 B)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 0 bytes 0 (0.0 B)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@42dba1061dd1 /]# ping www.magedu.com
PING www.magedu.com (101.200.188.230) 56(84) bytes of data.
```

```
64 bytes from 101.200.188.230 (101.200.188.230): icmp_seq=1 ttl=127 time=5.62 ms
64 bytes from 101.200.188.230 (101.200.188.230): icmp_seq=2 ttl=127 time=4.91 ms
```

4.3.2.2: 创建默认网络容器:

```
root@node1:~# docker run -it --name bridge-container-test -p 8081:8080
tomcat-web:app1 bash
[root@ed06c2785c92 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.10.0.2  netmask 255.255.255.0  broadcast 10.10.0.255
        ether 02:42:0a:0a:00:02  txqueuelen 0  (Ethernet)
          RX packets 8  bytes 696 (696.0 B)
          RX errors 0  dropped 0  overruns 0  frame 0
          TX packets 0  bytes 0 (0.0 B)
          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
      inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
          RX packets 0  bytes 0 (0.0 B)
          RX errors 0  dropped 0  overruns 0  frame 0
          TX packets 0  bytes 0 (0.0 B)
          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@ed06c2785c92 /]# ping www.magedu.com
PING www.magedu.com (101.200.188.230) 56(84) bytes of data.
64 bytes from 101.200.188.230 (101.200.188.230): icmp_seq=1 ttl=127 time=5.40 ms
64 bytes from 101.200.188.230 (101.200.188.230): icmp_seq=2 ttl=127 time=5.47 ms
```

4.3.3: 当前 iptables 规则:

```
root@docker-server1:~# iptables -t nat -vnL
Chain PREROUTING (policy ACCEPT 4 packets, 270 bytes)
pkts bytes target prot opt in out source destination
  14  744 DOCKER all -- * 0.0.0.0/0 0.0.0.0/0 ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 1 packets, 52 bytes)
pkts bytes target prot opt in out source destination

Chain OUTPUT (policy ACCEPT 1 packets, 76 bytes)
pkts bytes target prot opt in out source destination
  0   0 DOCKER all -- * 0.0.0.0/0 !127.0.0.0/8 ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 1 packets, 76 bytes)
pkts bytes target prot opt in out source destination
  6  436 MASQUERADE all -- * !br-954c8bda9a8c 10.100.0.0/24 0.0.0.0/0
  7  554 MASQUERADE all -- * !docker0 10.10.0.0/24 0.0.0.0/0
  0   0 MASQUERADE tcp -- * 10.100.0.2 10.100.0.2 tcp dpt:8080
  0   0 MASQUERADE tcp -- * 10.10.0.2 10.10.0.2 tcp dpt:8080

Chain DOCKER (2 references)
pkts bytes target prot opt in out source destination
  0   0 RETURN all -- br-954c8bda9a8c * 0.0.0.0/0 0.0.0.0/0
  1  84 RETURN all -- docker0 * 0.0.0.0/0 0.0.0.0/0
  0   0 DNAT  tcp -- !br-954c8bda9a8c * 0.0.0.0/0 0.0.0.0/0 tcp dpt:8080 to:10.100.0.2:8080
  0   0 DNAT  tcp -- !docker0 * 0.0.0.0/0 0.0.0.0/0 0.0.0.0/0 tcp dpt:8081 to:10.10.0.2:8080
root@docker-server1:~#
```

```
root@docker-server1:~# iptables -vnL
Chain INPUT (policy ACCEPT 144 packets, 11460 bytes)
pkts bytes target prot opt in out source destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts bytes target prot opt in out source destination
  32 3810 DOCKER-USER all -- * 0.0.0.0/0 0.0.0.0/0
  32 3810 DOCKER-ISOLATION-STAGE-1 all -- * 0.0.0.0/0 0.0.0.0/0 ctstate RELATED,ESTABLISHED
  11 1740 ACCEPT all -- br-954c8bda9a8c 0.0.0.0/0 0.0.0.0/0
  0   0 DOCKER all -- br-954c8bda9a8c 0.0.0.0/0 0.0.0.0/0
  11 856 ACCEPT all -- br-954c8bda9a8c !br-954c8bda9a8c 0.0.0.0/0 0.0.0.0/0
  0   0 ACCEPT all -- br-954c8bda9a8c br-954c8bda9a8c 0.0.0.0/0 0.0.0.0/0
  49 4552 ACCEPT all -- docker0 0.0.0.0/0 0.0.0.0/0 ctstate RELATED,ESTABLISHED
  49 3816 DOCKER all -- docker0 docker0 0.0.0.0/0 0.0.0.0/0
  59 4526 ACCEPT all -- docker0 !docker0 0.0.0.0/0 0.0.0.0/0
  0   0 ACCEPT all -- docker0 docker0 0.0.0.0/0 0.0.0.0/0
  1  84 ACCEPT all -- * 192.168.0.0/21 0.0.0.0/0

Chain OUTPUT (policy ACCEPT 150 packets, 19429 bytes)
pkts bytes target prot opt in out source destination

Chain DOCKER (2 references)
pkts bytes target prot opt in out source destination
  0   0 ACCEPT tcp -- !br-954c8bda9a8c br-954c8bda9a8c 0.0.0.0/0 10.100.0.2 tcp dpt:8080
  0   0 ACCEPT tcp -- !docker0 docker0 0.0.0.0/0 10.10.0.2 tcp dpt:8080

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
pkts bytes target prot opt in out source destination
  11 856 DOCKER-ISOLATION-STAGE-2 all -- br-954c8bda9a8c !br-954c8bda9a8c 0.0.0.0/0 0.0.0.0/0
  59 4526 DOCKER-ISOLATION-STAGE-2 all -- docker0 !docker0 0.0.0.0/0 0.0.0.0/0
  179 15490 RETURN all -- * 0.0.0.0/0 0.0.0.0/0
```

4.3.4: 如何与使用默认网络的容器通信:

现在有一个 docker0(10.10.0.0/24) 网络一个自定义的 magedu-net(10.100.0.0/24) 网络，每个网络上分别运行了不同数量的容器，那么怎么才能让位于不同网络的容器可以相互通信呢？

```
# iptables-save > iptables.sh
```

```

-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o br-954c8bda9a8c -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o br-954c8bda9a8c -j DOCKER
-A FORWARD -i br-954c8bda9a8c ! -o br-954c8bda9a8c -j ACCEPT
-A FORWARD -i br-954c8bda9a8c -o br-954c8bda9a8c -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -s 192.168.0.0/21 -j ACCEPT
-A DOCKER -d 10.100.0.2/32 ! -i br-954c8bda9a8c -o br-954c8bda9a8c -p tcp -m tcp --dport 8080 -j ACCEPT
-A DOCKER -d 10.10.0.2/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 8080 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i br-954c8bda9a8c ! -o br-954c8bda9a8c -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
#-A DOCKER-ISOLATION-STAGE-2 -o br-954c8bda9a8c -j DROP
#-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
COMMIT
# Completed on Thu Sep 12 20:20:00 2019
"iptables.sh" 501 2324C

```

4.3.5: 重新导入 iptables 并验证通信:

```
#重新导入 iptables 规则:
# iptables-restore < iptables.sh
```

```
[root@ed06c2785c92 /]# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.10.0.2 netmask 255.255.255.0 broadcast 10.10.0.255
        ether 02:42:0a:0a:00:02 txqueuelen 0 (Ethernet)
          RX packets 28 bytes 2560 (2.5 KiB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 119 bytes 11292 (11.0 KiB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@ed06c2785c92 /]# ping 10.100.0.2
PING 10.100.0.2 (10.100.0.2) 56(84) bytes of data.
64 bytes from 10.100.0.2: icmp_seq=1 ttl=63 time=0.214 ms
64 bytes from 10.100.0.2: icmp_seq=2 ttl=63 time=0.168 ms
64 bytes from 10.100.0.2: icmp_seq=3 ttl=63 time=0.167 ms
64 bytes from 10.100.0.2: icmp_seq=4 ttl=63 time=0.109 ms
```

五：Docker 仓库之单机 Docker Registry：

Docker Registry 作为 Docker 的核心组件之一负责镜像内容的存储与分发，客户端的 docker pull 以及 push 命令都将直接与 registry 进行交互，最初版本的 registry 由 Python 实现，由于设计初期在安全性，性能以及 API 的设计上有着诸多的缺陷，该版本在 0.9 之后停止了开发，由新的项目 distribution（新的 docker register 被称为 Distribution）来重新设计并开发下一代 registry，新的项目由 go 语言开发，所有的 API，底层存储方式，系统架构都进行了全面的重新设计已解决上一代 registry 中存在的问题，2016 年 4 月份 registry 2.0 正式发布，docker 1.6 版本开始支持 registry 2.0，而八月份随着 docker 1.8 发布，docker hub 正式启用 2.1 版本 registry 全面替代之前版本 registry，新版 registry 对镜像存储格式进行了重新设计并和旧版不兼容，docker 1.5 和之前的版本无法读取 2.0 的镜像，另外，Registry 2.4 版本之后支持了回收站机制，也就是可以删除镜像了，在 2.4 版本之前是无法支持删除镜像的，所以如果你要使用最好是大于 Registry 2.4 版本的，目前最新版本为 2.7.x。

官方文档地址：<https://docs.docker.com/registry/>

官方 github 地址：<https://github.com/docker/distribution>

本部分将介绍通过官方提供的 docker registry 镜像来简单搭建一套本地私有仓库环境。

5.1：下载 docker registry 镜像：

```
[root@docker-server1 ~]# docker pull  registry
[root@docker-server1 ~]# docker pull  registry
Using default tag: latest
Trying to pull repository docker.io/library/registry ...
latest: Pulling from docker.io/library/registry
ab7e51e37a18: Pull complete
c8ad8919ce25: Pull complete
5808405bc62f: Pull complete
f6000d7b276c: Pull complete
f792fdcd8ff6: Pull complete
Digest: sha256:9d295999d330eba2552f9c78c9f59828af5c9a9c15a3fdb1351df03eaad04c6a
[root@docker-server1 ~]#
```

5.2：搭建单机仓库：

5.2.1：创建授权使用目录：

```
[root@docker-server1 ~]# mkdir /docker/auth  # 创建一个授权使用目录
```

5.2.2: 创建用户:

```
[root@docker-server1 ~]# cd /docker  
[root@docker-server1 docker]# docker run --entrypoint htpasswd registry -Bbn jack  
123456 > auth/htpasswd #创建一个用户并生成密码
```

5.2.3: 验证用户名密码:

```
[root@docker-server1 docker]# cat auth/htpasswd  
jack:$2y$05$8W2aO/2RXMrMzw/0M5pig..QXwUh/m/XPoW5H/XxloLLRDTepVGP6
```

5.2.4: 启动 docker registry:

```
[root@docker-server1 docker]# docker run -d -p 5000:5000 --restart=always  
--name registry1 -v /docker/auth:/auth -e "REGISTRY_AUTH=htpasswd" -e  
"REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" -e  
REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd registry  
ce659e85018bea3342045f839c43b66de1237ce5413c0b6b72c0887bece5325a
```

5.2.5: 验证端口和容器:

```
[root@docker-server1 docker]# docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS  
PORTS NAMES  
ce659e85018b registry "/entrypoint.sh /etc/" About a minute ago Up About a mi  
nute 0.0.0.0:5000->5000/tcp registry1  
[root@docker-server1 docker]# ss -tnl  
State Recv-Q Send-Q Local Address:Port Peer Address:Port  
LISTEN 0 128 *:22 *:  
LISTEN 0 100 127.0.0.1:25 *:  
LISTEN 0 128 :::5000 *:  
LISTEN 0 128 :::22 *:  
LISTEN 0 100 :::1:25 *:  
[root@docker-server1 docker]#
```

5.2.6: 测试登录仓库:

5.2.6.1: 报错如下:

```
[root@docker-server1 ~]# docker login 192.168.10.205:5000  
Username: jack  
Password:  
Error response from daemon: Get https://192.168.10.205:5000/v1/users/: http: server gave HTTP response to HTTPS client  
[root@docker-server1 ~]#
```

5.2.6.2: 解决方法:

编辑各 docker 服务器/etc/sysconfig/docker 配置文件如下:

```
[root@docker-server1 ~]# vim /etc/sysconfig/docker  
4 OPTIONS='--selinux-enabled --log-driver=journald'  
9 ADD_REGISTRY='--add-registry 192.168.10.205:5000'
```

```

10 INSECURE_REGISTRY='--insecure-registry 192.168.10.205:5000'
[root@docker-server1 ~]# systemctl restart docker

[root@docker-server2 ~]# vim /etc/sysconfig/docker
4 OPTIONS='--selinux-enabled --log-driver=journald'
5 if [ -z "${DOCKER_CERT_PATH}" ]; then
6     DOCKER_CERT_PATH=/etc/docker
7 fi
8
9 ADD_REGISTRY='--add-registry 192.168.10.205:5000'
10 INSECURE_REGISTRY='--insecure-registry 192.168.10.205:5000'
[root@docker-server2 ~]# systemctl restart docker

```

5.2.6.3: 验证各 docker 服务器登录:

#server1:

```

[root@docker-server1 ~]# docker login 192.168.10.205:5000
Username (jack): jack
Password:
Login Succeeded ←
[root@docker-server1 ~]#

```

#server2:

```

[root@docker-server2 ~]# docker login 192.168.10.205:5000
Username: jack
Password:
Login Succeeded ←

```

5.2.7: 在 Server1 登录后上传镜像:

5.2.7.1: 镜像打 tag:

```

[root@docker-server1 ~]# docker tag 192.168.10.205:5000/jack/nginx-1.10.3:v1

```

```

[root@docker-server1 ~]# docker images
REPOSITORY          TAG      IMAGE ID   CREATED        SIZE
docker.io/registry  latest   177391bcf802  10 days ago  33.26 MB
jack/nginx-1.10.3   v1      811aa2f1a1d   2 weeks ago  491.6 MB
jack/nginx-test-image v1      188c311c451a  2 weeks ago  472 MB
jack/nginx-test-image latest  8492484d2960  2 weeks ago  472 MB
jack/centos-nginx   v1      ab9759679eb5  2 weeks ago  388 MB
jack/centos-nginx   latest  768054441422  2 weeks ago  388 MB
docker.io/jenkins/jenkins lts    5f9cc03c136c  4 weeks ago  811.5 MB
docker.io/nginx     latest  40960efd7b8f   5 weeks ago  108.4 MB
docker.io/centos    latest  d123f4e55e12  5 weeks ago  196.6 MB
<none>              <none>  196e0ce0c9fb  12 weeks ago  196.6 MB
docker.io/alpine    latest  76da55c8019d   3 months ago  3.962 MB
docker.io/zhangshijie/alpine-test latest  76da55c8019d   3 months ago  3.962 MB
[root@docker-server1 ~]# docker tag jack/nginx-1.10.3:v1 192.168.10.205:5000/jack/nginx-1.10.3:v1

```

5.2.7.2: 上传镜像:

```
[root@docker-server1 ~]# docker push 192.168.10.205:5000/jack/nginx-1.10.3:v1
The push refers to a repository [192.168.10.205:5000/jack/nginx-1.10.3]
4637e87f0d85: Pushed
126f5a7b45c0: Pushed
5f0f33a5a6f3: Pushed
c13dc3f4d849: Pushed
25e653659466: Pushed
7ebe0b965c04: Pushed
9521c7d1e1db: Pushing [=====>] 32.27 MB/253.8 MB
3b7c2e371c0b: Pushed
cf516324493c: Pushing [=====>] 38.06 MB/196.6 MB
cf516324493c: Preparing
```

5.2.8: Server 2 下载镜像并启动容器:

5.2.8.1: 登录并从 docker registry 下载镜像:

```
[root@docker-server2 ~]# docker images
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
[redacted]
[root@docker-server2 ~]# docker login 192.168.10.205:5000
Username (jack): jack
Password:
Login Succeeded
[root@docker-server2 ~]# docker pull 192.168.10.205:5000/jack/nginx-1.10.3:v1
```

```
[root@docker-server2 ~]# docker pull 192.168.10.205:5000/jack/nginx-1.10.3:v1
Trying to pull repository 192.168.10.205:5000/jack/nginx-1.10.3 ...
sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616: Pulling from 192.168.10.205:5000/jack/nginx-1.10.3
d9aaaf4d82f24: Pull complete
6803d2ac9d4b: Pull complete
0055c23cd7e6: Pull complete
06eb480d99bc: Pull complete
949ff195d4e5: Pull complete
bf9b583eb345: Pull complete
aafdfdf6c706f: Extracting [=====>] 2.002 kB/2.002 kB
0b784962da77: Download complete
1d0566c17a28: Download complete
1d0566c17a28: Pulling fs layer
[redacted]
```

5.2.8.2: 验证镜像下载成功:

```
[root@docker-server2 ~]# docker images
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
[redacted]
[root@docker-server2 ~]# docker login 192.168.10.205:5000
Username (jack): jack
Password:
Login Succeeded
[root@docker-server2 ~]# docker pull 192.168.10.205:5000/jack/nginx-1.10.3:v1
Trying to pull repository 192.168.10.205:5000/jack/nginx-1.10.3 ...
sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616: Pulling from 192.168.10.205:5000/jack/nginx-1.10.3
d9aaaf4d82f24: Pull complete
6803d2ac9d4b: Pull complete
0055c23cd7e6: Pull complete
06eb480d99bc: Pull complete
949ff195d4e5: Pull complete
bf9b583eb345: Pull complete
aafdfdf6c706f: Pull complete
0b784962da77: Pull complete
1d0566c17a28: Pull complete
Digest: sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616
Status: Downloaded newer image for 192.168.10.205:5000/jack/nginx-1.10.3:v1
[root@docker-server2 ~]# docker images
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
192.168.10.205:5000/jack/nginx-1.10.3   v1              811aa2f1aa1d   2 weeks ago        491.6 MB
[root@docker-server2 ~]#
```

登录到 docker registry 并下载镜像

镜像下载完成

5.2.8.3: 从下载的镜像启动容器:

```
[root@docker-server2 ~]# docker run -d --name docker-registry -p 80:80
```

```
192.168.10.205:5000/jack/nginx-1.10.3:v1 nginx
2ba24f28362e1b039fbebeada94a332111c2882aa06987463ae033c630f5c9927c
```

```
[root@docker-server2 ~]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
192.168.10.205:5000/jack/nginx-1.10.3   v1        811aa2f1a1d   2 weeks ago   491.6 MB
[redacted]
[redacted]
[root@docker-server2 ~]# docker run -d --name docker-registry -p 80:80 192.168.10.205:5000/jack/nginx-1.10.3:v1 nginx
[redacted]
[root@docker-server2 ~]# ss -tnl
State      Recv-Q Send-Q      Local Address:Port          Peer Address:Port
LISTEN     0      128          *:22                  *:*
LISTEN     0      100          127.0.0.1:25           *:*
LISTEN     0      128          :::80                 :::*
LISTEN     0      128          :::22                 :::*
LISTEN     0      100          :::1:25               :::*
```

5.2.8.4：访问测试：

← → C ⌂ ① 192.168.10.206

test nginx page

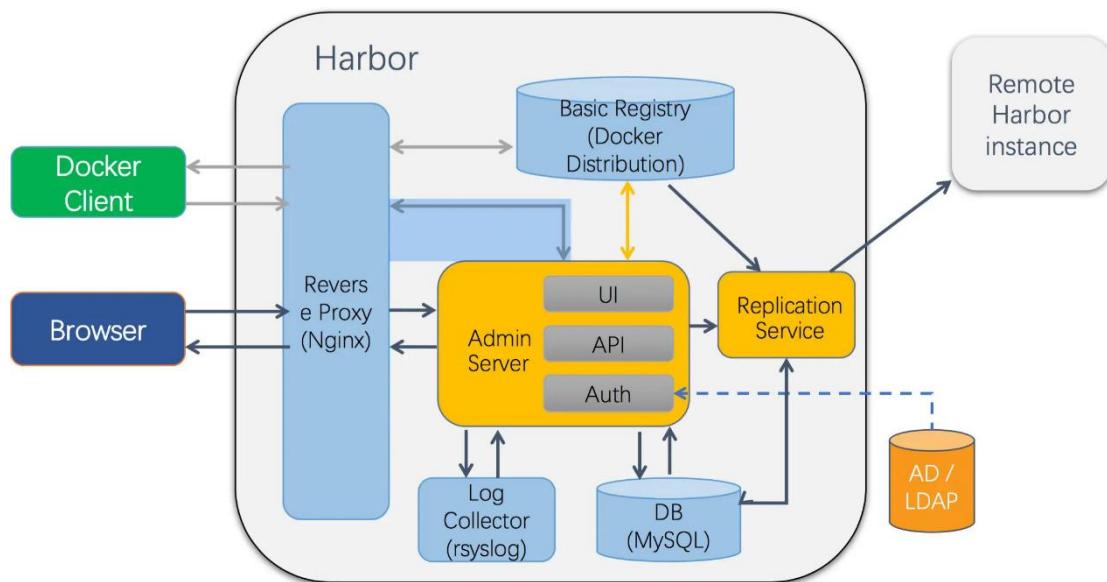
六：docker 仓库之分布式 Harbor

Harbor 是一个用于存储和分发 Docker 镜像的企业级 Registry 服务器，由 vmware 开源，其通过添加一些企业必需的功能特性，例如安全、标识和管理等，扩展了开源 Docker Distribution。作为一个企业级私有 Registry 服务器，Harbor 提供了更好的性能和安全。提升用户使用 Registry 构建和运行环境传输镜像的效率。Harbor 支持安装在多个 Registry 节点的镜像资源复制，镜像全部保存在私有 Registry 中，确保数据和知识产权在公司内部网络中管控，另外，Harbor 也提供了高级的安全特性，诸如用户管理，访问控制和活动审计等。

vmware 官方开源服务列表地址：<https://vmware.github.io/harbor/cn/>,

harbor 官方 github 地址：<https://github.com/vmware/harbor>

harbor 官方网址：<https://goharbor.io/>



6.1：Harbor 功能官方介绍：

基于角色的访问控制：用户与 Docker 镜像仓库通过“项目”进行组织管理，一个用户可以对多个镜像仓库在同一命名空间（project）里有不同的权限。

镜像复制：镜像可以在多个 Registry 实例中复制（同步）。尤其适合于负载均衡，高可用，混合云和多云的场景。

图形化用户界面：用户可以通过浏览器来浏览，检索当前 Docker 镜像仓库，管理项目和命名空间。

AD/LDAP 支：Harbor 可以集成企业内部已有的 AD/LDAP，用于鉴权认证管理。

审计管理：所有针对镜像仓库的操作都可以被记录追溯，用于审计管理。

国际化：已拥有英文、中文、德文、日文和俄文的本地化版本。更多的语言将会

添加进来。

RESTful API - RESTful API : 提供给管理员对于 Harbor 更多的操控，使得与其它管理软件集成变得更容易。

部署简单: 提供在线和离线两种安装工具，也可以安装到 vSphere 平台(OVA 方式)虚拟设备。

nginx: harbor 的一个反向代理组件，代理 registry、ui、token 等服务。这个代理会转发 harbor web 和 docker client 的各种请求到后端服务上。

harbor-adminserver: harbor 系统管理接口，可以修改系统配置以及获取系统信息。

harbor-db: 存储项目的元数据、用户、规则、复制策略等信息。

harbor-jobservice: harbor 里面主要是为了镜像仓库之前同步使用的。

harbor-log: 收集其他 harbor 的日志信息。

harbor-ui: 一个用户界面模块，用来管理 registry。

registry: 存储 docker images 的服务，并且提供 pull/push 服务。

redis: 存储缓存信息

webhook: 当 registry 中的 image 状态发生变化的时候去记录更新日志、复制等操作。

token service: 在 docker client 进行 pull/push 的时候负责 token 的发放。

6.2: 安装 Harbor:

下载地址: <https://github.com/vmware/harbor/releases>

安装文档:

https://github.com/vmware/harbor/blob/master/docs/installation_guide.md

6.2.1: 服务器 1 安装 docker:

本次使用当前 harbor 最新的稳定版本 1.7.5 离线安装包，具体名称为 harbor-offline-installer-v1.7.5.tgz

```
[root@docker-server1 ~]# yum install docker -y
[root@docker-server1 ~]# systemctl start docker
[root@docker-server1 ~]# systemctl enable docker
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to
/usr/lib/systemd/system/docker.service.
```

6.2.2: 服务器 2 安装 docker:

```
[root@docker-server2 ~]# yum install docker -y
[root@docker-server2 ~]# systemctl start docker
[root@docker-server2 ~]# systemctl enable docker
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to
/usr/lib/systemd/system/docker.service.
```

6.2.3: 下载 Harbor 安装包:

6.2.3.1: 下载离线完整安装包:

#推荐使用离线完整安装包

```
[root@docker-server2 ~]# cd /usr/local/src/  
[root@docker-server2 src]# wget  
https://github.com/vmware/harbor/releases/download/v1.7.5/harbor-offline-installer-v1.7.5.tgz
```

6.2.3.2: 下载在线安装包

#不是很推荐此方式

```
[root@docker-server2 src]# wget  
https://github.com/vmware/harbor/releases/download/v1.7.5/harbor-online-installer-v1.7.5.tgz
```

6.3: 配置 Harbor:

6.3.1: 解压并编辑 harbor.cfg:

```
[root@docker-server1 src]# tar xvf harbor-offline-installer-v1.7.5.tgz  
[root@docker-server1 src]# ln -sv /usr/local/src/harbor /usr/local/  
    '/usr/local/harbor' -> '/usr/local/src/harbor'  
[root@docker-server1 harbor]# cd /usr/local/harbor/  
  
[root@docker-server1 harbor]# yum install python-pip -y  
[root@docker-server1 harbor]# docker-compose start  
[root@docker-server1 harbor]# vim harbor.cfg  
  
[root@docker-server1 harbor]# grep "^[a-Z]" harbor.cfg  
hostname = 192.168.10.205  
ui_url_protocol = http  
db_password = root123  
max_job_workers = 3  
customize_crt = on  
ssl_cert = /data/cert/server.crt  
ssl_cert_key = /data/cert/server.key  
secretkey_path = /data  
admiral_url = NA  
clair_db_password = password  
email_identity = harbor  
email_server = smtp.163.com  
email_server_port = 25  
email_username = rorooot@163.com  
email_password = zhang@123
```

```
email_from = admin <rooroot@163.com>
email_ssl = false
harbor_admin_password = zhang@123
auth_mode = db_auth
ldap_url = ldaps://ldap.mydomain.com
ldap_basedn = ou=people,dc=mydomain,dc=com
ldap_uid = uid
ldap_scope = 3
ldap_timeout = 5
self_registration = on
token_expiration = 30
project_creation_restriction = everyone
verify_remote_cert = on
```

6.3.2: 更新 harbor 配置:

6.3.2.1: 首次部署 harbor 更新:

```
[root@docker-server1 harbor]# pwd
/usr/local/harbor #在 harbor 当前目录执行
[root@docker-server1 harbor]# ./prepare #更新配置
```

```
[root@docker-server1 harbor]# pwd
/usr/local/harbor
[root@docker-server1 harbor]# ./prepare
Generated and saved secret to file: /data/secretkey
Generated configuration file: ./common/config/nginx/nginx.conf
Generated configuration file: ./common/config/adminserver/env
Generated configuration file: ./common/config/ui/env
Generated configuration file: ./common/config/registry/config.yml
Generated configuration file: ./common/config/db/env
Generated configuration file: ./common/config/jobservice/env
Generated configuration file: ./common/config/jobservice/app.conf
Generated configuration file: ./common/config/ui/app.conf
Generated certificate, key file: ./common/config/ui/private_key.pem, cert file: ./common/config/registry/root.crt
The configuration files are ready, please use docker-compose to start the service.
[root@docker-server1 harbor]#
```

#执行完毕后会在当前目录生成一个 docker-compose.yml 文件，用于配置数据目录等配置信息：

6.3.2.2: 后期修改配置:

如果 harbor 运行一段时间之后需要更改配置，则步骤如下：

6.3.2.2.1: 停止 harbor:

```
[root@docker-server1 harbor]# pwd
/usr/local/harbor #harbor 的当前目录
[root@docker-server1 harbor]# docker-compose stop
```

6.3.2.2.2: 编辑 harbor.cfg 进行相关配置:

```
[root@docker-server1 harbor]# vim harbor.cfg
```

6.3.2.2.3: 更新配置:

```
[root@docker-server1 harbor]# ./prepare
[root@docker-server1 harbor]# ./prepare
Clearing the configuration file: ./common/config/adminserver/env
Clearing the configuration file: ./common/config/ui/env
Clearing the configuration file: ./common/config/ui/app.conf
Clearing the configuration file: ./common/config/ui/private_key.pem
Clearing the configuration file: ./common/config/db/env
Clearing the configuration file: ./common/config/jobservice/env
Clearing the configuration file: ./common/config/jobservice/app.conf
Clearing the configuration file: ./common/config/registry/config.yml
Clearing the configuration file: ./common/config/registry/root.crt
Clearing the configuration file: ./common/config/nginx/nginx.conf
loaded secret from file: /data/secretkey
Generated configuration file: ./common/config/nginx/nginx.conf
Generated configuration file: ./common/config/adminserver/env
Generated configuration file: ./common/config/ui/env
Generated configuration file: ./common/config/registry/config.yml
Generated configuration file: ./common/config/db/env
Generated configuration file: ./common/config/jobservice/env
Generated configuration file: ./common/config/jobservice/app.conf
Generated configuration file: ./common/config/ui/app.conf
Generated certificate, key file: ./common/config/ui/private_key.pem, cert file: ./common/config/registry/root.crt
The configuration files are ready, please use docker-compose to start the service.
[root@docker-server1 harbor]#
```

清理之前的配置并重新生成新的配置

6.3.2.3: 启动 harbor 服务:

```
[root@docker-server1 harbor]# docker-compose start
[root@docker-server1 harbor]# docker-compose start
Starting log      ... done
Starting adminserver ... done
Starting registry   ... done
Starting ui        ... done
Starting mysql     ... done
Starting jobservice ... done
Starting proxy     ... done
[root@docker-server1 harbor]#
```

6.3.3: 官方方式启动 Harbor:

6.3.3.1: 官方方式安装并启动 harbor:

```
[root@docker-server1 harbor]# yum install python-pip
[root@docker-server1 harbor]# pip install --upgrade pip
[root@docker-server1 harbor]# pip install docker-compose
[root@docker-server1 harbor]# ./install.sh #官方构建 harbor 和启动方式, 推荐此方法, 会下载官方的 docker 镜像:
```

6.3.3.2: 部署过程中:

```
[root@docker-server1 harbor]# ./install.sh
[Step 0]: checking installation environment ...
Note: docker version: 1.12.6
Note: docker-compose version: 1.17.1
[Step 1]: loading Harbor images ...
dd60b611baaa: Loading layer [=====] 133.2 MB/133.2 MB
abf0579c40fd: Loading layer [=====] 1.536 kB/1.536 kB
ea1fc7bed9c5: Loading layer [=====] 22.48 MB/22.48 MB
1d6671367c69: Loading layer [=====] 7.168 kB/7.168 kB
b322bb3e4765: Loading layer [=====] 5.339 MB/5.339 MB
0cf512d418ac: Loading layer [=====] 9.728 kB/9.728 kB
4a7cdc0b1a2b: Loading layer [=====] 2.56 kB/2.56 kB
ef1130526636: Loading layer [=====] 22.48 MB/22.48 MB
Loaded image: vmware/harbor-ui:v1.2.2
4a050fccec52: Loading layer [=====] 12.16 MB/12.16 MB
d918d73369ec: Loading layer [=====] 17.3 MB/17.3 MB
22898836924e: Loading layer [=====] 15.87 kB/15.87 kB
Loaded image: vmware/notary-photon:server-0.5.0
76c156eab077: Loading layer [=====] 512 B/15.87 kB
76c156eab077: Loading layer [=====] 65.73 MB/134 MB
76c156eab077: Loading layer [>] 557.1 kB/134 MB
```

6.3.3.3: 部署完成:

```
[Step 4]: starting Harbor ...
Creating network "harbor_harbor" with the default driver
Creating harbor-log ...
Creating harbor-log ... done
Creating harbor-db ...
Creating registry ...
Creating harbor-adminserver ...
Creating registry
Creating harbor-db
Creating harbor-adminserver ... done
Creating harbor-ui ...
Creating harbor-ui ... done
Creating harbor-jobservice ...
Creating nginx ...
Creating nginx
Creating harbor-jobservice ... done
✓ ----Harbor has been installed and started successfully.----
Now you should be able to visit the admin portal at http://192.168.10.205 .
For more details, please visit https://github.com/vmware/harbor .

[root@docker-server1 harbor]#
```

6.3.3.4: 查看本地的镜像:

```
[root@docker-server1 harbor]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
vmware/harbor-log   v1.2.2   36ef78ae27df  8 weeks ago  199.8 MB
vmware/harbor-jobservice  v1.2.2   e2af366cba44  8 weeks ago  164.1 MB
vmware/harbor-ui    v1.2.2   39efb472c253  8 weeks ago  177.8 MB
vmware/harbor-adminserver  v1.2.2   c75963ec543f  8 weeks ago  141.6 MB
vmware/harbor-db    v1.2.2   ee7b9fa37c5d  8 weeks ago  328.5 MB
vmware/nginx-photon  1.11.13  6cc5c831fc7f  9 weeks ago  144.2 MB
vmware/registry     2.6.2-photon  5d9100e4350e  3 months ago  173.1 MB
vmware/postgresql   9.6.4-photon  c562762cbd12  4 months ago  225.3 MB
vmware/clair        v2.0.1-photon  f04966b4af6c  5 months ago  297.1 MB
vmware/harbor-notary-db  mariadb-10.1.10  64ed814665c6  8 months ago  324.1 MB
vmware/notary-photon  signer-0.5.0   b1eda7d10640  8 months ago  155.7 MB
vmware/notary-photon  server-0.5.0   6e2646682e3c  9 months ago  156.9 MB
photon              1.0       e6e4e4a2balb  17 months ago  127.5 MB
[root@docker-server1 harbor]#
```

6.3.3.5: 查看本地端口:

```
[root@docker-server1 harbor]# ss -tnl
State      Recv-Q Send-Q          Local Address:Port
LISTEN      0      128              127.0.0.1:1514
LISTEN      0      128              *:22
LISTEN      0      100              127.0.0.1:25
LISTEN      0      128              :::80
LISTEN      0      128              :::22
LISTEN      0      100              :::125
LISTEN      0      128              :::443
LISTEN      0      128              :::4443
[root@docker-server1 harbor]#
```

6.3.3.6: web 访问 Harbor 管理界面:

VMware Harbor™

默认管理员admin
admin
.....| harbor.cfg 定义的密码

记住我 忘记密码

登录

注册账号

受欢迎的镜像仓库

名称

6.3.3.7: 登录成功后的界面:

项目

日志

> 系统管理

+ 项目

项目名称	访问级别	角色
library	公开	项目管理员

6.3.4: 非官方方式启动:

6.3.4.1: 非官方方式启动 harbor:

```
[root@docker-server2 harbor]# ./prepare
[root@docker-server2 harbor]# yum install python-pip -y
[root@docker-server2 harbor]# pip install --upgrade pip #升级 pip 为最新版本
[root@docker-server2 harbor]# pip install docker-compose #安装 docker-compose
命令
[root@docker-server2 harbor]# pip install docker-compose
Collecting docker-compose
  Downloading docker_compose-1.17.1-py2.py3-none-any.whl (108kB)
    100% |██████████| 112kB 239kB/s
Collecting texttable<0.10,>=0.9.0 (from docker-compose)
  Downloading texttable-0.9.1.tar.gz
Collecting backports.ssl-match-hostname>=3.5; python_version < "3.5" (from docker-compose)
  Downloading backports.ssl_match_hostname-3.5.0.1.tar.gz
Collecting docker<3.0,>=2.5.1 (from docker-compose)
  Downloading docker-2.6.1-py2.py3-none-any.whl (117kB)
    100% |██████████| 122kB 400kB/s
Collecting jsonschema<3,>=2.5.1 (from docker-compose)
  Downloading jsonschema-2.6.0-py2.py3-none-any.whl
Collecting cached-property<2,>=1.2.0 (from docker-compose)
  Downloading cached_property-1.3.1-py2.py3-none-any.whl
Collecting six<2,>=1.3.0 (from docker-compose)
  Downloading six-1.11.0-py2.py3-none-any.whl
Collecting ipaddress>=1.0.16; python_version < "3.3" (from docker-compose)
  Downloading ipaddress-1.0.19.tar.gz
Collecting enum34<2,>=1.0.4; python_version < "3.4" (from docker-compose)
```

6.3.4.2: 启动 harbor:

```
[root@docker-server2 harbor]# docker-compose up -d #非官方方式构建容器, 此步骤会从官网下载镜像, 需要相当长的时间
#执行过程如下:
```

```
[root@docker-server2 harbor]# docker-compose up -d
Creating network "harbor_harbor" with the default driver
Pulling log (vmware/harbor-log:v1.2.2)...
Trying to pull repository docker.io/vmware/harbor-log ...
v1.2.2: Pulling from docker.io/vmware/harbor-log
8cbd18f8deb2: Downloading [=>                                                 ] 991.3 kB/48.88 MB
07d028a1dbdd: Downloading [=>                                             ] 1.078 MB/35.09 MB
82f2b0c8d80e: Download complete
c451c419e7cf: Download complete
c3c2a289c925: Download complete
```

6.3.4.3: 查看本地镜像:

6.3.4.4: 验证本地端口:

6.3.4.5: web 访问 Harbor 界面:

6.4: 配置 docker 使用 harbor 仓库上传下载镜像:

6.4.1: 编辑 docker 配置文件:

注意: 如果我们配置的是 https 的话, 本地 docker 就不需要有任何操作就可以访问 harbor 了

```
[root@docker-server1 ~]# vim /etc/sysconfig/docker  
4 OPTIONS='--selinux-enabled --log-driver=journald --insecure-registry  
192.168.10.205'  
#其中 192.168.10.205 是我们部署 Harbor 的地址, 即 hostname 配置项值。配置  
完后需要重启 docker 服务。
```

6.4.2: 重启 docker 服务:

```
[root@docker-server1 ~]# systemctl stop docker  
[root@docker-server1 ~]# systemctl start docker
```

6.4.3: 验证能否登录 harbor:

```
[root@docker-server1 harbor]# docker login 192.168.10.205
```

```
[root@docker-server1 harbor]# docker login 192.168.10.205  
Username: admin  
Password:  
Login Succeeded
```



```
[root@docker-server1 harbor]#
```

6.4.4: 测试上传和下载镜像:

将之前单机仓库构建的 Nginx 镜像上传到 harbor 服务器用于测试

6.4.4.1: 导入镜像:

```
[root@docker-server1 harbor]# docker load < /opt/nginx-1.10.3_docker.tar.gz
```

```
[root@docker-server1 harbor]# docker load < /opt/nginx-1.10.3_docker.tar.gz
cf516324493c: Loading layer [=====] 205.2 MB/205.2 MB
3b7c2e371c0b: Loading layer [=====] 20.5 MB/20.5 MB
9521c7d1e1db: Loading layer [=====] 260.3 MB/260.3 MB
7ebe0b965c04: Loading layer [=====] 5.659 MB/5.659 MB
25e653659466: Loading layer [=====] 15.16 MB/15.16 MB
c13dc3f4d849: Loading layer [=====] 6.656 kB/6.656 kB
5f0f33a5a6f3: Loading layer [=====] 310.3 kB/310.3 kB
126f5a7b45c0: Loading layer [=====] 2.56 kB/2.56 kB
4637e87f0d85: Loading layer [=====] 4.096 kB/4.096 kB
Loaded image: 192.168.10.205:5000/jack/nginx-1.10.3:v1
[root@docker-server1 harbor]#
```

6.4.4.2: 验证镜像导入成功:

```
[root@docker-server1 harbor]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
192.168.10.205:5000/jack/nginx-1.10.3   v1      811aa2f1aa1d    3 weeks ago   108.5 MB
vmware/harbor-log      v1.2.2   36ef78ae27df    8 weeks ago   199.8 MB
vmware/harbor-jobservice  v1.2.2   e2af366cba44    8 weeks ago   164.1 MB
vmware/harbor-ui        v1.2.2   39efb472c253    8 weeks ago   177.8 MB
vmware/harbor-adminserver v1.2.2   c75963ec543f    8 weeks ago   141.6 MB
vmware/harbor-db        v1.2.2   ee7b9fa37c5d    8 weeks ago   328.5 MB
vmware/nginx-photon     1.11.13   6cc5c831fc7f    9 weeks ago   144.2 MB
vmware/registry         2.6.2-photon  5d9100e4350e    3 months ago   173.1 MB
vmware/postgresql        9.6.4-photon  c562762cbd12    4 months ago   225.3 MB
vmware/clair             v2.0.1-photon  f04966b4af6c    5 months ago   297.1 MB
vmware/harbor-notary-db mariadb-10.1.10  64ed814665c6    8 months ago   324.1 MB
vmware/notary-photon     signer-0.5.0   b1eda7d10640    8 months ago   155.7 MB
vmware/notary-photon     server-0.5.0   6e2646682e3c    9 months ago   156.9 MB
photon                  1.0      e6e4e4a2ba1b    17 months ago   127.5 MB
[root@docker-server1 harbor]#
```

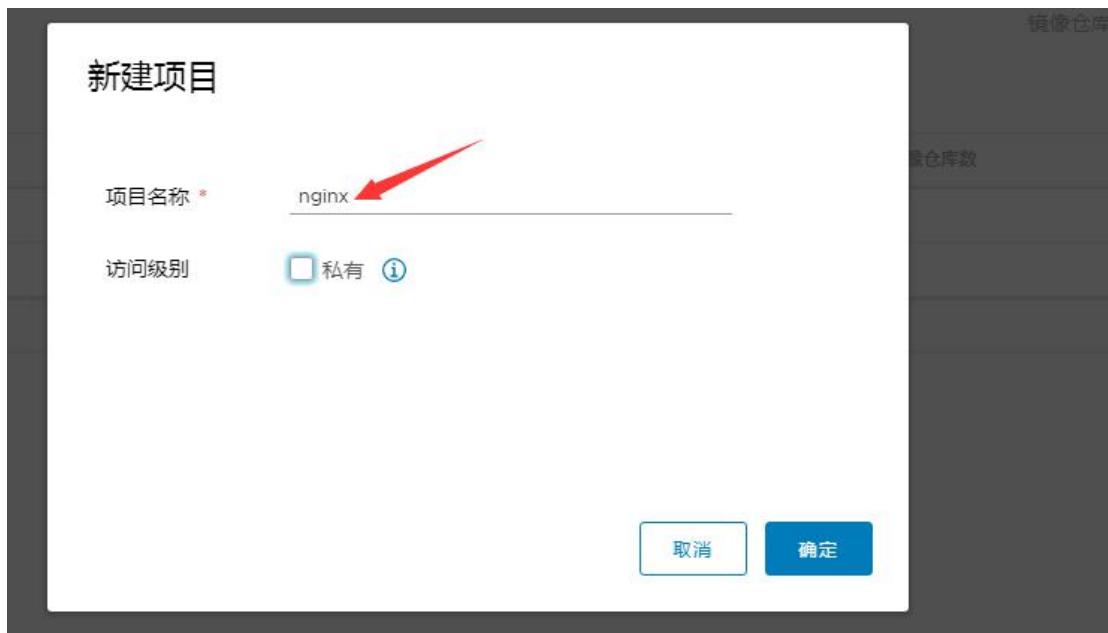
6.4.4.3: 镜像打 tag:

#修改 images 的名称，不修改成指定格式无法将镜像上传到 harbor 仓库，格式为: Harbor IP/项目名/image 名字:版本号:

```
[root@docker-server1 harbor]# docker tag
192.168.10.205:5000/jack/nginx-1.10.3:v1 192.168.10.205/nginx/nginx_1.10.3:v1
[root@docker-server1 harbor]# docker images
```

```
[root@docker-server1 harbor]# docker tag 192.168.10.205:5000/jack/nginx-1.10.3:v1 192.168.10.205/nginx/nginx_1.10.3:v1
[root@docker-server1 harbor]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
docker.io/nginx      latest   f895b3fb9e30    6 days ago   108.5 MB
192.168.10.205/nginx_nginx_1.10.3   v1      811aa2f1aa1d    3 weeks ago   491.6 MB
192.168.10.205:5000/jack/nginx-1.10.3   v1      811aa2f1aa1d    3 weeks ago   491.6 MB
vmware/harbor-log      v1.2.2   36ef78ae27df    8 weeks ago   199.8 MB
vmware/harbor-jobservice  v1.2.2   e2af366cba44    8 weeks ago   164.1 MB
vmware/harbor-ui        v1.2.2   39efb472c253    8 weeks ago   177.8 MB
vmware/harbor-adminserver v1.2.2   c75963ec543f    8 weeks ago   141.6 MB
vmware/harbor-db        v1.2.2   ee7b9fa37c5d    8 weeks ago   328.5 MB
vmware/nginx-photon     1.11.13   6cc5c831fc7f    9 weeks ago   144.2 MB
vmware/registry         2.6.2-photon  5d9100e4350e    3 months ago   173.1 MB
vmware/postgresql        9.6.4-photon  c562762cbd12    4 months ago   225.3 MB
vmware/clair             v2.0.1-photon  f04966b4af6c    5 months ago   297.1 MB
vmware/harbor-notary-db mariadb-10.1.10  64ed814665c6    8 months ago   324.1 MB
vmware/notary-photon     signer-0.5.0   b1eda7d10640    8 months ago   155.7 MB
vmware/notary-photon     server-0.5.0   6e2646682e3c    9 months ago   156.9 MB
photon                  1.0      e6e4e4a2ba1b    17 months ago   127.5 MB
[root@docker-server1 harbor]#
```

6.4.4.4: 在 harbor 管理界面创建项目:



6.4.4.4: 将镜像 push 到 harbor:

#格式为： docker push 镜像名:版本

```
[root@docker-server1 harbor]# docker push 192.168.10.205/nginx/nginx_1.10.3:v1
The push refers to a repository [192.168.10.205/nginx/nginx_1.10.3]
4637e87f0d85: Pushed
126f5a7b45c0: Pushed
5f0f33a5a6f3: Pushed
c13dc3f4d849: Pushed
25e653659466: Pushing [=====>] 13.38 MB/15.04 MB
7ebe0b965c04: Pushing [=====>] 2.256 MB/5.342 MB
9521c7d1e1db: Pushing [>] 3.246 MB/253.8 MB
3b7c2e371c0b: Pushing [==>] 922.1 kB/20.48 MB
cf516324493c: Waiting
cf516324493c: Preparing
```

6.4.4.5: push 完成:

```
[root@docker-server1 harbor]# docker push 192.168.10.205/nginx/nginx_1.10.3:v1
The push refers to a repository [192.168.10.205/nginx/nginx_1.10.3]
4637e87f0d85: Pushed
126f5a7b45c0: Pushed
5f0f33a5a6f3: Pushed
c13dc3f4d849: Pushed
25e653659466: Pushed
7ebe0b965c04: Pushed
9521c7d1e1db: Pushed
3b7c2e371c0b: Pushed
cf516324493c: Pushed
v1: digest: sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616 size: 2203
[root@docker-server1 harbor]#
```

6.4.4.6: harbor 界面验证镜像上传成功:

项目名称	访问级别	角色	镜像仓库数	创建时间
library	公开	项目管理员	0	2017/12/18 下午11:49
nginx	私有	项目管理员	1	2017/12/19 上午2:28

6.4.4.7: 验证镜像信息:

名称	标签数	下载数
nginx/nginx_1.10.3	1	0
v1	docker pull 192.168.10.205/nginx/nginx_1.10.3:v1	Jack.Zhang 2017/11/25 上午2:49 1.12.6 123456@qq.com

6.4.5: 验证从 harbor 服务器下载镜像并启动容器:

6.4.5.1: 更改 docker 配置文件:

目前凡是需要从 harbor 镜像服务器下载 image 的 docker 服务都要更改，不更改的话无法下载：

```
[root@docker-server2 ~]# vim /etc/sysconfig/docker
4 OPTIONS='--selinux-enabled --log-driver=journald --insecure-registry
192.168.10.205'
```

6.4.5.2: 重启 docker:

```
[root@docker-server2 ~]# systemctl stop docker
[root@docker-server2 ~]# systemctl start docker
```

6.4.5.3: 验证从 harbor 下载镜像:

6.4.5.5.1: 查看下载命令:

#harbor 上的每个镜像里面自带 pull 命令

名称	标签数	下载数
nginx/nginx_1.10.3	1	0

镜像下载命令

6.4.5.5.2: 执行下载:

```
[root@docker-server2 ~]# docker pull 192.168.10.205/nginx/nginx_1.10.3:v1
```

```
[root@docker-server2 ~]# docker pull 192.168.10.205/nginx/nginx_1.10.3:v1
Trying to pull repository 192.168.10.205/nginx/nginx_1.10.3 ...
sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616: Pulling from 192.168.10.205/nginx/nginx_1.10.3
d9aaaf4d82f24: Downloading [=====] 23.71 MB/73.39 MB
6803d2ac9d4b: Download complete
0055c23cd7e6: Downloading [=====] 51.17 MB/97.22 MB
06eb480d99bc: Download complete
949ff195d4e5: Download complete
bf9b583eb345: Download complete
aafdfdf6c706f: Download complete
0b784962da77: Download complete
1d0566c17a28: Download complete
d0566c17a28: Pulling fs layer
```

6.4.5.5.3: 验证镜像下载完成:

```
[root@docker-server2 ~]# docker pull 192.168.10.205/nginx/nginx_1.10.3:v1
Trying to pull repository 192.168.10.205/nginx/nginx_1.10.3 ...
sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616: Pulling from 192.168.10.205/nginx/nginx_1.10.3
d9aaaf4d82f24: Pull complete
6803d2ac9d4b: Pull complete
0055c23cd7e6: Pull complete
06eb480d99bc: Pull complete
949ff195d4e5: Pull complete
bf9b583eb345: Pull complete
aafdfdf6c706f: Pull complete
0b784962da77: Pull complete
1d0566c17a28: Pull complete
Digest: sha256:ddc45cda500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616
Status: Downloaded newer image for 192.168.10.205/nginx/nginx_1.10.3:v1
[root@docker-server2 ~]# docker images
REPOSITORY          TAG           IMAGE ID        CREATED         SIZE
192.168.10.205/nginx/nginx_1.10.3   v1            811aa2f1aa1d   3 weeks ago    491.6 MB
```

6.4.6: 从镜像启动容器并验证:

6.4.6.1: 启动容器:

```
[root@docker-server2 ~]# docker run -d -p 80:80 -p 443:443
192.168.10.205/nginx/nginx_1.10.3:v1 nginx
89901f9badf74809f6abccc352fc7479f1490f0ebe6d6e3b36d689e73c3f9027
```

6.4.6.2: 验证端口:

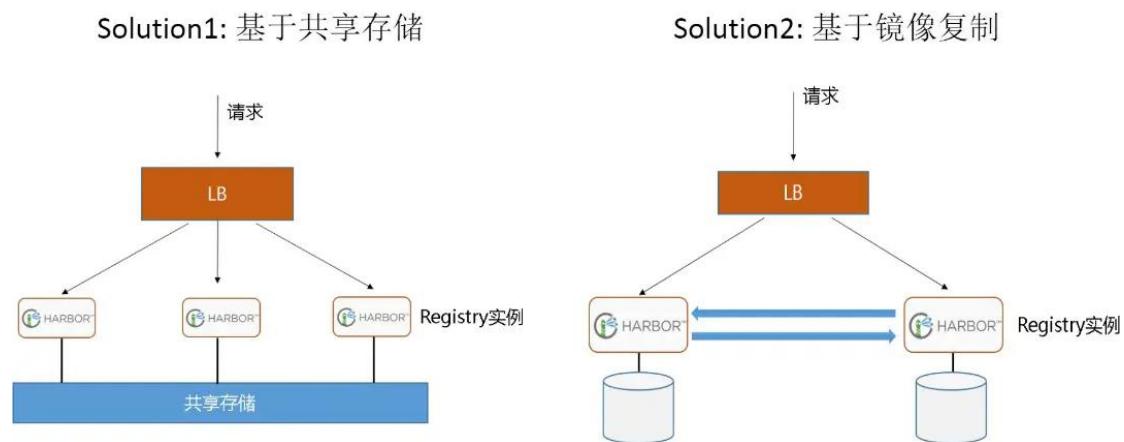
```
[root@docker-server2 ~]# ss -tnl
State      Recv-Q Send-Q                               Local Address:Port
LISTEN      0      128                                *:22
LISTEN      0      100                               127.0.0.1:25
LISTEN      0      128                                :::80
LISTEN      0      128                                :::22
LISTEN      0      100                               ::1:25
LISTEN      0      128                                :::443
[root@docker-server2 ~]# lsof -i:80
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
docker-pr 2680 root    4u  IPv6  20791      0t0  TCP *:http (LISTEN)
[root@docker-server2 ~]#
```

6.4.6.3: 验证 web 访问:



6.5: 实现 harbor 高可用:

高可用实现方式:



Harbor 支持基于策略的 Docker 镜像复制功能，这类似于 MySQL 的主从同步，其可以实现不同的数据中心、不同的运行环境之间同步镜像，并提供友好的管理界面，大大简化了实际运维中的镜像管理工作，已经有用很多互联网公司使用 harbor 搭建内网 docker 仓库的案例，并且还有实现了双向复制的案例，本文将实现单向复制的部署：

6.5.1: 新部署一台 harbor 服务器:

```
[root@docker-server2 ~]# cd /usr/local/src/
```

```
[root@docker-server2 src]# tar xf harbor-offline-installer-v1.7.5.tgz
[root@docker-server2 src]# ln -sv /usr/local/src/harbor /usr/local/
  '/usr/local/harbor' -> '/usr/local/src/harbor'
[root@docker-server2 src]# cd /usr/local/harbor/
[root@docker-server2 harbor]# grep "^[a-Z]" harbor.cfg
hostname = 192.168.10.206
ui_url_protocol = http
db_password = root123
max_job_workers = 3
customize_crt = on
ssl_cert = /data/cert/server.crt
ssl_cert_key = /data/cert/server.key
secretkey_path = /data
admiral_url = NA
clair_db_password = password
email_identity = harbor-1.7.5
email_server = smtp.163.com
email_server_port = 25
email_username = rooroot@163.com
email_password = zhang@123
email_from = admin <rooroot@163.com>
email_ssl = false
harbor_admin_password = zhang@123
auth_mode = db_auth
ldap_url = ldaps://ldap.mydomain.com
ldap_basedn = ou=people,dc=mydomain,dc=com
ldap_uid = uid
ldap_scope = 3
ldap_timeout = 5
self_registration = on
token_expiration = 30
project_creation_restriction = everyone
verify_remote_cert = on

[root@docker-server2 harbor]# yum install python-pip -y
[root@docker-server2 harbor]# pip install --upgrade pip
[root@docker-server2 harbor]# pip install docker-compose
[root@docker-server2 harbor]# ./install.sh
```

```
[Step 4]: starting Harbor ...
Creating network "harbor_harbor" with the default driver
Creating harbor-log ...
Creating harbor-log ... done
Creating registry ...
Creating harbor-adminserver ...
Creating harbor-db ...
Creating harbor-adminserver
Creating registry
Creating harbor-adminserver ... done
Creating harbor-ui ...
Creating harbor-ui ... done
Creating harbor-jobservice ...
Creating nginx ...
Creating harbor-jobservice
Creating harbor-jobservice ... done

✓ ----Harbor has been installed and started successfully.----→

Now you should be able to visit the admin portal at http://192.168.10.206 .
For more details, please visit https://github.com/vmware/harbor .

[root@docker-server2 harbor]#
```

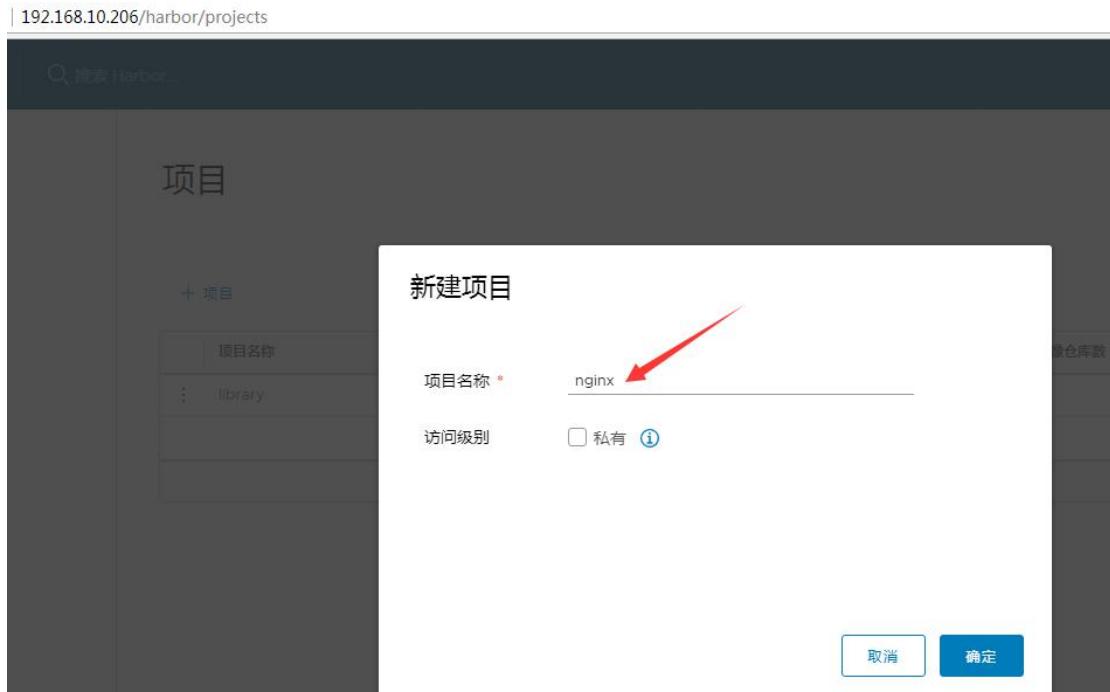
6.5.2: 验证从 harbor 登录:



The screenshot shows the Harbor web interface. At the top, there is a header bar with a back arrow, a forward arrow, a refresh icon, and a link to '192.168.10.206/harbor/projects'. Below the header, there is a navigation bar with a 'vm' icon and the word 'Harbor'. A search bar is also present. On the left side, there is a sidebar with a '项目' tab selected, followed by '日志', '系统管理' (with sub-options '用户管理', '复制管理', and '配置管理'), and other tabs like '镜像' and '构建'. The main content area is titled '项目' and shows a table with one row. The table has columns for '项目名称' (Library), '访问级别' (公开), and '角色' (项目管理员). There is also a '+ 项目' button.

6.5.3: 创建一个 nginx 项目:

#与主 harbor 项目名称保持一致:



6.5.4: 在主 harbor 服务器配置同步测试:

项目名称	访问级别	角色	镜像仓库数
library	公开	项目管理员	0
nginx	公开	项目管理员	1

6.5.5: 点击复制规则:

nginx 项目管理员

镜像仓库 成员 日志 复制

+ 复制规则

名称	描述	目标名	上次起始时间

未发现任何复制规则!

6.5.6: 主 harbor 编辑同步策略:

The screenshot shows the 'New Rule' configuration dialog in Harbor. The 'Name' field is set to 'Nginx 镜像备份'. The 'Description' field contains 'Nginx 镜像备份'. The 'Enabled' checkbox is checked. The 'Target Name' field is set to '192.168.10.206' with the 'Create Target' checkbox checked. The 'Target URL' field is set to 'http://192.168.10.206'. The 'Username' field is set to 'admin'. The 'Password' field is set to '*****'. A success message '测试连接成功。确认测试连接成功' is displayed above the 'Test Connection' button. The 'Test Connection' button is highlighted with a red box.

6.5.7: 主 harbor 查看镜像同步状态:

The screenshot shows the 'Replication Status' page in Harbor. It displays a table of replication rules. One rule is listed: 'Nginx 镜像备份' with target '192.168.10.206', last run on '2017/12/19 上午3:51', and status '启用'. Below this, a table of replication tasks is shown. One task is listed: 'nginx/nginx_1.10.3' with status 'finished', operation 'transfer', created at '2017/12/19 上午3:51', and updated at '2017/12/19 上午3:51'. A red arrow points to the 'finished' status in the task table.

6.5.8: 从 harbor 查看镜像:

The screenshot shows the Harbor interface at the URL 192.168.10.206/harbor/projects/2/repositories. The project 'nginx' is selected. The '镜像仓库' tab is active. A red arrow points from the text '验证从harbor镜像同步成功且下载地址为从harbor服务器IP' to the 'Pull命令' column for the 'v1' tag, which contains the command 'docker pull 192.168.10.206/nginx/nginx_1.10.3:v1'. Another red arrow points to the '私有' (Private) status of the 'nginx' project in the table below.

名称	标签数	下数
nginx/nginx_1.10.3	1	0

标签	Pull命令	作者	创建时间	Docker 版本	架构	操作系统
v1	docker pull 192.168.10.206/nginx/nginx_1.10.3:v1	Jack.Zhang 123456@qq.com	2017/11/25 上午2:49	1.12.6	amd64	linux

6.5.9: 测试从 harbor 镜像下载和容器启动:

6.5.9.1: docker 客户端配置使用 harbor:

#本次新部署了一台 docker 客户端, IP 地址为 192.168.10.207

```
[root@docker-server3 ~]# vim /etc/sysconfig/docker
```

```
4      OPTIONS='--selinux-enabled      --log-driver=journald      --insecure-registry
192.168.10.206'
```

6.5.9.2: 重启 docker 服务:

```
[root@docker-server3 ~]# systemctl restart docker
```

6.5.9.3: 从 harbor 项目设置为公开:

The screenshot shows the Harbor interface at the URL 192.168.10.206/harbor/projects. The '项目' tab is active. A red arrow points from the text '把项目设置为公开, 否则docker客户端需要登录之后才能下载镜像' to the '访问级别' (Access Level) column for the 'nginx' project, which is currently set to '私有' (Private). The 'library' project has its access level set to '公开' (Public).

项目名称	访问级别	角色	镜像仓库数
library	公开	项目管理员	0
nginx	私有	项目管理员	1

6.5.9.4: 设置项目为公开访问:

项目名称	访问级别	角色	镜像仓库数
library	公开	项目管理员	0
nginx	公开	项目管理员	1

6.5.9.5: docker 客户端下载镜像:

```
[root@docker-server3 ~]# docker pull 192.168.10.206/nginx/nginx_1.10.3:v1
Trying to pull repository 192.168.10.206/nginx/nginx_1.10.3 ...
sha256:ddc45cd4500c0982eb8856fe69f1cf6b6b3e5714e72e70c073d7b0021846b616: Pulling from 192.168.10.206/nginx/nginx_1.10.3
d9aaaf4d82f24: Extracting [>          ] 557.1 kB/73.39 MB
6803d2ac9d4b: Download complete
0055c23cd7e6: Downloading [=====] 71.43 MB/97.22 MB
06eb480d99bc: Download complete
949ff195d4e5: Download complete
bf9b583eb345: Download complete
aafdfd6c706f: Download complete
0b784962da77: Download complete
1d0566c17a28: Download complete
1d0566c17a28: Pulling fs layer
```

6.5.9.6: docker 客户端从镜像启动容器:

```
[root@docker-server3 ~]# docker run -d -p 80:80 -p443:443
192.168.10.206/nginx/nginx_1.10.3:v1 nginx
0b496bc81035291b80062d1fba7d4065079ab911c2a550417cf9e593d353c20b
```

6.5.9.7: 验证 web 访问:



#至此，高可用模式的 harbor 仓库部署完毕

6.6.: 实现 harbor 双向同步:

6.6.1: 在 docker 客户端导入 centos 基础镜像:

```
[root@docker-server3 ~]# docker load -i /opt/centos.tar.gz
[root@docker-server3 ~]# vim /etc/sysconfig/docker
4 OPTIONS='--selinux-enabled --log-driver=journald --insecure-registry
192.168.10.206'
```

6.6.2: 镜像打 tag:

```
[root@docker-server3 ~]# docker tag docker.io/centos 192.168.10.206/nginx/centos_base
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
192.168.10.206/nginx/nginx_1.10.3	latest	811aa2f1aa1d	3 weeks ago	491.6 MB
192.168.10.206/nginx/nginx_1.10.3	v1	811aa2f1aa1d	3 weeks ago	491.6 MB
docker.io/centos	latest	196e0ce0c9fb	3 months ago	196.6 MB
[root@docker-server3 ~]# docker tag docker.io/centos 192.168.10.206/nginx/centos_base				
[root@docker-server3 ~]# docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
192.168.10.206/nginx/nginx_1.10.3	latest	811aa2f1aa1d	3 weeks ago	491.6 MB
192.168.10.206/nginx/nginx_1.10.3	v1	811aa2f1aa1d	3 weeks ago	491.6 MB
192.168.10.206/nginx/centos_base	latest	196e0ce0c9fb	3 months ago	196.6 MB
docker.io/centos	latest	196e0ce0c9fb	3 months ago	196.6 MB

```
[root@docker-server3 ~]#
```

6.6.3: 上传到从 harbor:

```
[root@docker-server3 ~]# docker push 192.168.10.206/nginx/centos_base
```

```
[root@docker-server3 ~]# docker push 192.168.10.206/nginx/centos_base
The push refers to a repository [192.168.10.206/nginx/centos_base]
cf516324493c: Mounted from nginx/nginx_1.10.3
latest: digest: sha256:822de5245dc5b659df56dd32795b08ae42db4cc901f3462fc509e91e97132dc0 size: 529
[root@docker-server3 ~]#
```

6.6.4: 从 harbor 界面验证:

The screenshot shows the Harbor interface for the 'nginx' repository. The repository was created from the 'nginx_1.10.3' Dockerfile. It contains two tags: 'nginx/nginx_1.10.3' (v1) and 'nginx/centos_base'. The 'nginx/centos_base' tag has 1 download.

6.6.5: 从 harbor 创建同步规则:

规则方式与主 harbor 相同，写对方的 IP+用户名密码，然后点测试连接，确认可以测试连接通过。

修改规则

名称*	nginx images backup
描述	nginx images backup
启用	<input checked="" type="checkbox"/>
上次起始时间	2017/12/19 上午5:05
目标名*	207-206
目标URL*	http://192.168.10.205
用户名	admin
密码
测试连接成功。	
<input type="button" value="测试连接"/> <input type="button" value="取消"/> <input type="button" value="确定"/>	
更新时间	2017/12/19 下午6:30
2017/12/19 上午5:06	
2017/12/19 上午5:05	
2017/12/19 上午5:05	

6.6.6: 到主 harbor 验证镜像:

192.168.10.205/harbor/projects/2/repositories

搜索 Harbor...

< 项目

nginx 项目管理员

镜像仓库 成员 日志 复制

	名称	标签数	下载数
: >	nginx/nginx_1.10.3	2	6
: >	nginx/centos_base	1	0

6.6.7: docker 镜像端测试:

6.6.7.1: 下载 centos 基础镜像:

```
[root@docker-server1 harbor]# docker pull 192.168.10.205/nginx/centos_base
Using default tag: latest
Trying to pull repository 192.168.10.205/nginx/centos_base ...
sha256:822de5245dc5b659df56dd32795b08ae42db4cc901f3462fc509e91e97132dc
```

```
O: Pulling from 192.168.10.205/nginx/centos_base
```

Digest:

```
sha256:822de5245dc5b659df56dd32795b08ae42db4cc901f3462fc509e91e97132dc0
```

```
0
```

```
[root@docker-server1 harbor]# docker pull 192.168.10.205/nginx/centos_base
Using default tag: latest
Trying to pull repository 192.168.10.205/nginx/centos_base ...
sha256:822de5245dc5b659df56dd32795b08ae42db4cc901f3462fc509e91e97132dc0: Pulling from 192.168.10.205/nginx/centos_base

Digest: sha256:822de5245dc5b659df56dd32795b08ae42db4cc901f3462fc509e91e97132dc0
Status: Downloaded newer image for 192.168.10.205/nginx/centos_base:latest
[root@docker-server1 harbor]# docker images
REPOSITORY          TAG      IMAGE ID   CREATED    SIZE
docker.io/nginx     latest   f895b3fb9e30  7 days ago  108.5 MB
192.168.10.205/nginx/nginx_1.10.3  v1       811aa2f1aa1d  3 weeks ago  491.6 MB
192.168.10.205:5000/jack/nginx-1.10.3  v1       811aa2f1aa1d  3 weeks ago  491.6 MB
vmware/harbor-log   v1.2.2   36ef78ae27df  8 weeks ago  199.8 MB
vmware/harbor-jobservice  v1.2.2   e2af366cb44  8 weeks ago  164.1 MB
vmware/harbor-ui    v1.2.2   39efb472c253  8 weeks ago  177.8 MB
vmware/harbor-adminserver  v1.2.2   c75963ec543f  8 weeks ago  141.6 MB
vmware/harbor-db    v1.2.2   ee7b9fa37c5d  8 weeks ago  328.5 MB
vmware/nginx-photon  1.11.13  6cc5c831fc7f  9 weeks ago  144.2 MB
192.168.10.205/nginx/centos_base  latest   196e0ce0c9fb  3 months ago  196.6 MB
vmware/registry     2.6.2-photon  5d9100e4350e  3 months ago  173.1 MB
vmware/postgresql   9.6.4-photon  c562762cbd12  4 months ago  225.3 MB
vmware/clair        v2.0.1-photon  f04966b4af6c  5 months ago  297.1 MB
vmware/harbor-notary-db  mariadb-10.1.10  64ed814665c6  8 months ago  324.1 MB
vmware/notary-photon  signer-0.5.0   b1eda7d10640  8 months ago  155.7 MB
vmware/notary-photon  server-0.5.0   6e2646682e3c  9 months ago  156.9 MB
photon              1.0       e6e4e4a2balb  17 months ago  127.5 MB
[root@docker-server1 harbor]#
```

6.6.7.2: 从镜像启动容器:

```
[root@docker-server1 ~]# docker run -it --name centos_base
```

```
192.168.10.205/nginx/centos_base bash
```

```
[root@771f5aa0d089 /]#
```

```
[root@docker-server1 ~]# docker run -it --name centos_base 192.168.10.205/nginx/centos_base bash
[root@771f5aa0d089 /]# ls
anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var
[root@771f5aa0d089 /]#
```

6.7: harbor https 配置:

```
# openssl genrsa -out /usr/local/src/harbor/certs/harbor-ca.key 2048
# openssl req -x509 -new -nodes -key /usr/local/src/harbor/certs/harbor-ca.key
-subj "/CN=harbor.magedu.net" -days 7120 -out
/usr/local/src/harbor/certs/harbor-ca.crt
```

```
# vim harbor.cfg
hostname = harbor.magedu.net
ui_url_protocol = https
ssl_cert = /usr/local/src/harbor/certs/harbor-ca.crt
ssl_cert_key = /usr/local/src/harbor/certs/harbor-ca.key
harbor_admin_password = 123456
# ./install.sh
```

```
# yum install docker-ce-18.06.3.ce-3.el7.x86_64.rpm  
# yum install docker-compose  
# mkdir /etc/docker/certs.d/harbor.magedu.net -p  
# cp certs/harbor-ca.crt /etc/docker/certs.d/harbor.magedu.net/  
# docker login harbor.magedu.net
```

登录测试：

七：单机编排之 Docker Compose：

当在宿主机启动较多的容器时候，如果都是手动操作会觉得比较麻烦而且容器出错，这个时候推荐使用 docker 单机编排工具 docker-compose，docker-compose 是 docker 容器的一种单机编排服务，docker-compose 是一个管理多个容器的工具，比如可以解决容器之间的依赖关系，就像启动一个 nginx 前端服务的时候会调用后端的 tomcat，那就得先启动 tomcat，但是启动 tomcat 容器还需要依赖数据库，那就还得先启动数据库，docker-compose 就可以解决这样的嵌套依赖关系，其完全可以替代 docker run 对容器进行创建、启动和停止。

docker-compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排，docker-compose 将所管理的容器分为三层，分别是工程（project），服务（service）以及容器（container）。

```
#github 地址 https://github.com/docker/compose
```

```
# cat docker-compose.yml  
version: '3.6'  
services:  
  nginx-server:  
    image: nginx  
    container_name: nginx-web1  
    expose:
```

```
- 80  
- 443  
ports:  
- "80:80"  
- "443:443"
```

7.1：基础环境准备：

7.1.1：安装 python-pip 软件包：

python-pip 包将安装一个 pip 的命令，pip 命令是一个 python 安装包的安装工具，其类似于 ubuntu 的 apt 或者 redhat 的 yum，但是 pip 只安装 python 相关的安装包，可以在多种操作系统安装和使用 pip。

Ubuntu:

```
# apt update  
# apt install -y python-pip #python2  
# apt install -y python3-pip #python3
```

Centos:

```
# yum install epel-release  
# yum install -y python-pip  
# pip install --upgrade pip  
Setting up python-enum34 (1.1.6-2) ... #####  
Setting up python-dbus (1.2.6-1) ...#####  
Setting up python-ipaddress (1.0.17-1) ...#####  
Setting up python-pip (9.0.1-2.3~ubuntu1.18.04.1) ...#####  
Setting up python-all (2.7.15~rc1-1) ...  
Setting up python-setuptools (39.0.1-2) ...#####  
Setting up python-keyrings.alt (3.0-1) ...#####  
Setting up python-all-dev (2.7.15~rc1-1) ...#####  
Setting up python-cryptography (2.1.4-1ubuntu1.3) ...#####  
Setting up python-secretstorage (2.3.1-2) ...  
Setting up python-keyring (10.6.0-1) ...#####  
Processing triggers for libc-bin (2.27-3ubuntu1) ...#####
```

7.1.2：安装 docker compose：

通过 apt、yum 及官方二进制安装 docker-compose

7.1.2.1：apt 或 yum 安装：

```
# pip install docker-compose #python2  
# pip3 install docker-compose #python3
```

```

root@docker-node1:~# pip install docker-compose
Collecting docker-compose
  Downloading https://files.pythonhosted.org/packages/2e/93/b8fb6532487fcc40f5c607ac428a609e7f74fb2
-py2.py3-none-any.whl (137kB)
    100% |████████████████████████████████| 143kB 1.5MB/s
Collecting docopt<1,>=0.6.1 (from docker-compose)
  Downloading https://files.pythonhosted.org/packages/a2/55/8f8cab2af404cf578136ef2cc5dfb50baa1761b
Requirement already satisfied: six<2,>=1.3.0 in /usr/lib/python2.7/dist-packages (from docker-compos
Collecting backports.ssl-match-hostname<4,>=3.5; python_version < "3.5" (from docker-compose)
  Downloading https://files.pythonhosted.org/packages/ff/2b/8265224812912bc5b7a607c44bf7b027554e1b97
ostname-3.7.0.1.tar.gz
Collecting texttable<2,>=0.9.0 (from docker-compose)
  Downloading https://files.pythonhosted.org/packages/82/a8/60df592e3a100a1f83928795aca210414d72cebd
z
Collecting jsonschema<4,>=2.5.1 (from docker-compose)
  Downloading https://files.pythonhosted.org/packages/c5/8f/51e89ce52a085483359217bc72cdbf6e75ee595d
py3-none-any.whl (56kB)
    100% |████████████████████████████████| 61kB 3.9MB/s
Collecting dockerpty<1,>=0.4.1 (from docker-compose)
  Downloading https://files.pythonhosted.org/packages/8d/ee/e9ecce4c32204a6738e0a5d5883d3413794d7498
z
Collecting subprocess32<4,>=3.5.4; python_version < "3.2" (from docker-compose)

```

7.1.2.2：官方二进制安装：

官方安装教程：<https://docs.docker.com/compose/install/>

官方二进制下载地址：<https://github.com/docker/compose/releases>

Assets 11	
docker-compose-Darwin-x86_64	9.94 MB
docker-compose-Darwin-x86_64.sha256	95 Bytes
docker-compose-Darwin-x86_64.tgz	9.96 MB
docker-compose-Darwin-x86_64.tgz.sha256	99 Bytes
docker-compose-Linux-x86_64	12.1 MB
docker-compose-Linux-x86_64.sha256	94 Bytes
docker-compose-Windows-x86_64.exe	9.96 MB
docker-compose-Windows-x86_64.exe.sha256	101 Bytes
run.sh	2.52 KB
Source code (zip)	
Source code (tar.gz)	

```

# wget
https://github.com/docker/compose/releases/download/1.x.y/docker-compose-Linu
x-x86\_64
# cp docker-compose-Linux-x86_64 /usr/bin/docker-compose
# chmod a+x /usr/bin/docker-compose

```

7.1.3：验证 docker-compose 版本：

```

# docker-compose version
docker-compose version 1.29.1, build c34c88b2
docker-py version: 5.0.0
CPython version: 3.7.10
OpenSSL version: OpenSSL 1.1.0l  10 Sep 2019

```

7.1.4: 查看 docker-compose 帮助:

<https://docs.docker.com/compose/reference/> #官方文档

```
[root@docker-server3 ~]# docker-compose --help
```

Usage:

```
  docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]  
  docker-compose -h | --help
```

选项如下：

```
-f, - file FILE #指定 Compose 模板文件， 默认为 docker-compose.yml。  
-p, - project-name NAME #指定项目名称， 默认将使用当前所在目录名称作为项  
目名。  
--verbose #显示更多输出信息  
--log-level LEVEL #定义日志级别 (DEBUG, INFO, WARNING, ERROR, CRITICAL)  
--no-ansi #不显示 ANSI 控制字符  
-v, --version #显示版本  
  
#命令选项， 需要在 docker-compose.yml 文件目录执行， 带#的不常用  
#build #修改 Dockerfile 后,通过 docker-compose 重新构建镜像并重建服务  
#bundle #从当前 docker compose 文件生成一个以当前目录为名称的从 Compose  
文件生成一个分布式应用程序捆绑包 (DAB) 。  
config -q #查看当前配置， 没有错误不输出任何信息  
#create #创建服务后容器会退出  
down #停止和删除所有容器、 网络、 镜像和卷等资源  
#events #从容器接收实时事件， 可以指定 json 日志格式， 如  
    docker-compose events --json  
  
#exec #进入指定容器进行操作  
    # docker-compose ps -f --services  
    # docker exec -it nginx-web1 sh  
help #显示帮助细信息  
#images #显示当前服务器的 docker 镜像信息  
kill #强制终止运行中的容器  
    # docker-compose kill -s SIGKILL nginx-server #SIG 是信号名的通用前缀，  
    KILL 是指让一个进程立即终止的动作,合并起来 SIGKILL 就是发送给一个进程使进  
程立即终止的信号。  
  
logs #查看容器的日志  
    # docker-compose logs --tail="10" -f 10 nginx-server  
#pause #暂停服务  
    # docker-compose ps --service  
    # docker-compose pause nginx-server  
  
#port #查看端口
```

```

# docker-compose port --protocol=tcp nginx 80
ps #列出容器
pull #重新拉取镜像
#push #上传镜像
#restart #重启服务
rm #删除已经停止的服务
#run #一次性运行容器,等于 docker run --rm
scale #设置指定服务运行的容器个数
    # docker-compose scale nginx-server=2 #动态伸缩每个 service 的副本数

start #启动服务
    # docker-compose stop nginx-server
stop #停止服务
    # docker-compose start nginx-server

top #显示容器运行状态
# docker-compose top
magedu_nginx-server_1
      UID      PID      PPID      C      STIME     TTY      TIME          CMD
-----
root      34113   34094      0   13:42      ?  00:00:00  nginx: master process nginx -g daemon off;
systemd+  34197   34113      0   13:42      ?  00:00:00  nginx: worker process

m43_nginx-server_2
      UID      PID      PPID      C      STIME     TTY      TIME          CMD
-----
root      34007   33985      0   13:42      ?  00:00:00  nginx: master process nginx -g daemon off;
systemd+  34093   34007      0   13:42      ?  00:00:00  nginx: worker process

unpause #取消暂定状态中的 server
    # docker-compose unpause nginx-server
up #创建并启动容器
version #显示 docker-compose 版本信息

```

7.2: 从 docker compose 启动单个容器:

目录可以在任意目录，推荐放在有意义的位置。

```

# cd /opt/
# mkdir magedu
# cd magedu/

```

7.2.1: 单个容器的 docker compose 文件:

编写一个 yml 格式的配置 docker-compose 文件，启动一个 nginx 服务，由于格式

为 yml 格式，因此要注意前后的缩进及上下行的等级关系。

```
# pwd  
/opt/maged  
  
# cat docker-compose.yml  
service-nginx-web:  
    image: 192.168.7.103/linux37/ubuntu-nginx:1.16.1  
    expose:  
        - 80  
        - 443  
    ports:  
        - "80:80"  
        - "443:443"
```

7.2.2：启动容器：

必须要在 docker compose 文件所在的目录执行：

```
# pwd  
/opt/magedu  
  
# docker-compose up -d #不加是 d 前台启动  
root@docker-node5:/opt/magedu# docker-compose up -d  
Pulling service-nginx-web (192.168.7.103/linux37/ubuntu-nginx:1.16.1)...  
1.16.1: Pulling from linux37/ubuntu-nginx  
35c102085707: Pull complete  
251f5509d51d: Pull complete  
8e829fe70a46: Pull complete  
6001e1789921: Pull complete  
227e7bf3fbde: Pull complete  
3296b408efdf: Extracting [> ] 557.1kB/123.9MB  
dcca4040235: Download complete  
9a521ecab4d3: Download complete  
f458df8bb68d: Download complete  
78bb5a31f0fe: Download complete  
02b868b0de52: Download complete
```

7.2.3：启动完成：

镜像下载完成后将容器创建完成并成功运行。

```
root@docker-node5:/opt/magedu# docker-compose up -d
Pulling service-nginx-web (192.168.7.103/linux37/ubuntu-nginx:1.16.1)...
1.16.1: Pulling from linux37/ubuntu-nginx
35c102085707: Pull complete
251f5509d51d: Pull complete
8e829fe70a46: Pull complete
6001e1789921: Pull complete
227e7bf3fbde: Pull complete
3296b408efdf: Pull complete
dccca4040235: Pull complete
9a521ecab4d3: Pull complete
f458df8bb68d: Pull complete
78bb5a31f0fe: Pull complete
02b868b0de52: Pull complete
Digest: sha256:dd6b652b8d9a33cc7dfada454456bf4ede2a4a4b3370334c4e23934d0c83084b
Status: Downloaded newer image for 192.168.7.103/linux37/ubuntu-nginx:1.16.1
Creating magedu_service-nginx-web_1 ... done
root@docker-node5:/opt/magedu#
```

7.2.4: web 访问测试:



7.2.5: 后台启动服务:

#容器的在启动的时候，会给容器自定义一个名称，在 service name 后面加_1。

```
# docker-compose up -d
root@docker-node5:/opt/magedu# docker-compose stop
Stopping magedu_service-nginx-web_1 ... done
root@docker-node5:/opt/magedu#
root@docker-node5:/opt/magedu# docker-compose up -d
Starting magedu_service-nginx-web_1 ... done
root@docker-node5:/opt/magedu#
```

```
# docker-compose ps
Name          Command   State           Ports
magedu_service-nginx-web_1    nginx     Up      0.0.0.0:443->443/tcp, 0.0.0.0:80->80/tcp
```

7.2.6: 自定义容器名称:

```
# cat docker-compose.yml
service-nginx-web:
  image: 192.168.7.103/linux37/ubuntu-nginx:1.16.1
  container_name: nginx-web1
  expose:
    - 80
```

```

- 443

ports:
- "80:80"
- "443:443"

# docker-compose up -d
Recreating magedu_service-nginx-web_1 ... done

# docker-compose ps
      Name      Command     State            Ports
-----+-----+-----+-----+
nginx-web1    nginx      Up       0.0.0.0:443->443/tcp, 0.0.0.0:80->80/tcp

```

7.2.7: 验证容器:

```

[root@docker-server3 docker-compose]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
af5a311eb032        192.168.10.206/nginx/nginx_1.10.3   "nginx"           51 seconds ago   Up 50 sec
onds               0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
[root@docker-server3 docker-compose]#

```

7.2.8: 查看容器进程:

```

[root@docker-server3 docker-compose]# docker-compose ps
root@docker-node5:/opt/magedu# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
e6cf012957bf        192.168.7.103/ubuntu-nginx:1.16.1   "nginx"           2 minutes ago   Up 2 minutes
u_service-nginx-web_1
root@docker-node5:/opt/magedu# 

```

7.3: 从 docker compose 启动多个容器:

7.3.1: 编辑 docker-compose 文件:

```

# pwd
/opt/magedu

# cat docker-compose.yml
service-nginx-web:
  image: 192.168.7.103/ubuntu-nginx:1.16.1
  container_name: nginx-web1
  expose:
    - 80
    - 443
  ports:
    - "80:80"
    - "443:443"

```

```

service-tomcat-app1:
  image: 192.168.7.103/linux37/linux37-tomcat:app1
  container_name: tomcat-app1
  expose:
    - 8080
  ports:
    - "8080:8080"

```

7.3.2: 重新启动容器:

```

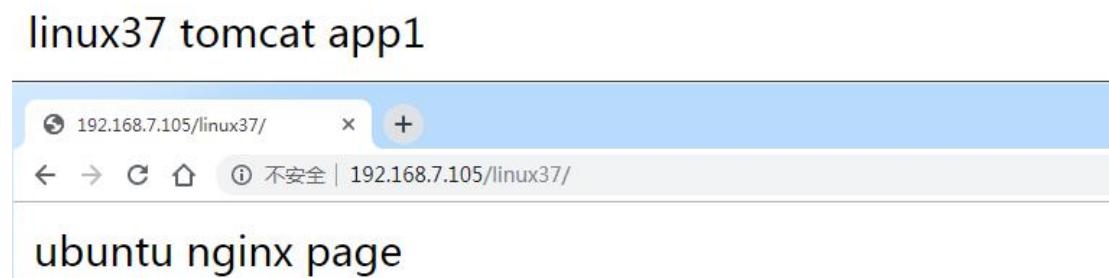
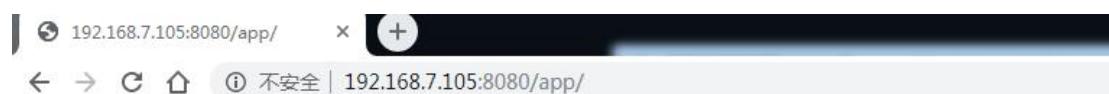
# pwd
/opt/magedu

# docker-compose stop
# docker-compose up -d

root@docker-node5:/opt/magedu# ss -tnl
State      Recv-Q      Send-Q      Local Address:Port          Peer Address:Port
LISTEN      0            128          0.0.0.0:111           0.0.0.0:*
LISTEN      0            128          0.0.0.0:56337          0.0.0.0:*
LISTEN      0            128          127.0.0.53%lo:53        0.0.0.0:*
LISTEN      0            128          0.0.0.0:22           0.0.0.0:*
LISTEN      0            128          127.0.0.1:6010          0.0.0.0:*
LISTEN      0            128          127.0.0.1:6011          0.0.0.0:*
LISTEN      0            128          0.0.0.0:60923          0.0.0.0:*
LISTEN      0            64           0.0.0.0:2049           0.0.0.0:*
LISTEN      0            128          0.0.0.0:46117          0.0.0.0:*
LISTEN      0            64           0.0.0.0:41447          0.0.0.0:*
LISTEN      0            128          [::]:55885             [::]:*
LISTEN      0            128          [::]:111              [::]:*
LISTEN      0            128          *:8080                *:*
LISTEN      0            128          [::]:111              [::]:*
LISTEN      0            128          [::]:53041             [::]:*
LISTEN      0            128          [::]:22               [::]:*
LISTEN      0            128          [::]:6010              [::]:*
LISTEN      0            128          *:443                 *:*
LISTEN      0            128          [::]:6011              [::]:*
LISTEN      0            64           [::]:36859              [::]:*
LISTEN      0            128          [::]:57117              [::]:*
LISTEN      0            64           [::]:2049              [::]:*
root@docker-node5:/opt/magedu#

```

7.3.3: web 访问测试:



7.4: 定义数据卷挂载:

7.4.1: 创建数据目录和文件:

```
# mkdir -p /data/nginx/magedu  
# echo "magedu test page" > /data/nginx/magedu/index.html
```

7.4.2: 编辑 compose 配置文件:

```
# cat docker-compose.yml  
service-nginx-web:  
    image: 192.168.7.103/linux37/ubuntu-nginx:1.16.1  
    container_name: nginx-web1  
    volumes:  
        - /data/nginx/magedu:/apps/nginx/html  
    expose:  
        - 80  
        - 443  
    ports:  
        - "80:80"  
        - "443:443"  
#   links:  
#     - nginx-web1
```

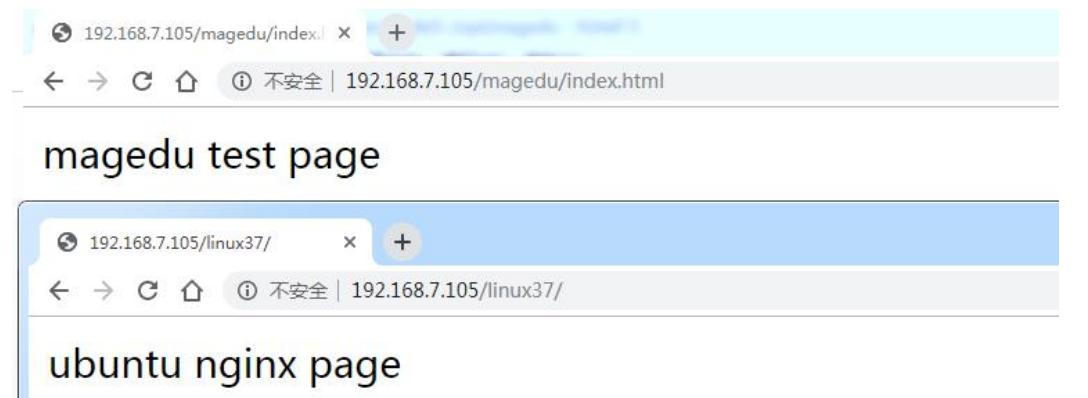
```
service-tomcat-app1:  
    image: 192.168.7.103/linux37/linux37-tomcat:app1  
    container_name: tomcat-app1  
    expose:  
        - 8080  
    ports:  
        - "8080:8080"
```

7.4.3: 重启容器:

```
# docker-compose stop  
# docker-compose up -d  
  
# docker-compose up -d  
tomcat-app1 is up-to-date  
Recreating nginx-web1 ... done
```

7.4.4: 验证 web 访问:

注: 同一个文件, 数据卷的优先级比镜像内的文件优先级高



7.4.5: 其他常用命令:

7.4.5.1: 重启单个指定容器:

```
# docker-compose restart service-nginx-web #写容器的服务名称
Restarting nginx-web1 ... done
root@docker-node5:/opt/magedu# head -n3 docker-compose.yml
service-nginx-web:
  image: 192.168.7.103/linux37/ubuntu-nginx:1.16.1
  container_name: nginx-web1
root@docker-node5:/opt/magedu# docker-compose restart service-nginx-web
Restarting nginx-web1 ... done
root@docker-node5:/opt/magedu#
```

7.4.5.2: 重启所有容器:

```
# docker-compose restart
root@docker-node5:/opt/magedu# docker-compose restart
Restarting nginx-web1 ... done
Restarting tomcat-app1 ... done
root@docker-node5:/opt/magedu#
```

7.4.5.3: 停止和启动单个容器:

```
# docker-compose stop service-tomcat-app1
# docker-compose start service-tomcat-app1
root@docker-node5:/opt/magedu# docker-compose stop service-tomcat-app1
Stopping tomcat-app1 ... done
root@docker-node5:/opt/magedu# docker-compose start service-tomcat-app1
Starting service-tomcat-app1 ... done
root@docker-node5:/opt/magedu#
```

7.4.5.4: 停止和启动所有容器:

```
# docker-compose stop
# docker-compose start
root@docker-node5:/opt/magedu# docker-compose stop
Stopping nginx-web1 ... done
Stopping tomcat-app1 ... done
root@docker-node5:/opt/magedu# docker-compose start
Starting service-tomcat-app1 ... done
Starting service-nginx-web ... done
root@docker-node5:/opt/magedu# █
```

7.5: 实现单机版的 Nginx+Tomcat:

编写 docker-compose.yml 文件，实现单机版本的 nginx+tomcat 的动静分离 web 站点，要求从 nginx 作为访问入口，当访问指定 URL 的时候转发至 tomcat 服务器响应。

7.5.1: 制作 Haproxy 镜像:

当前目录文件:

```
# pwd
/opt/dockerfile/web/linux37/haproxy
```

```
# tree
.
├── build-command.sh
├── dockerfile
├── haproxy-2.0.5.tar.gz
├── haproxy.cfg
└── run_haproxy.sh
```

0 directories, 5 files

7.5.1.1: dockerfile 文件:

```
# cat dockerfile
FROM linux37-centos-base:7.6.1810

maintainer zhangshijie "2973707860@qq.com"

RUN yum install -y yum install gcc gcc-c++ glibc glibc-devel pcre pcre-devel openssl
openssl-devel systemd-devel net-tools vim iotop bc zip unzip zlib-devel lrzsz tree
```

```

screen lsof tcpdump wget ntpdate
ADD haproxy-2.0.5.tar.gz /usr/local/src

RUN cd /usr/local/src/haproxy-2.0.5 && make ARCH=x86_64
TARGET=linux-glibc USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1 USE_SYSTEMD=1
USE_CPU_AFFINITY=1 PREFIX=/usr/local/haproxy && make install PREFIX=/usr/local/haproxy && cp haproxy /usr/sbin/ &&
mkdir /usr/local/haproxy/run
ADD haproxy.cfg /etc/haproxy/

ADD run_haproxy.sh /usr/bin
EXPOSE 80 9999
CMD ["/usr/bin/run_haproxy.sh"]

```

7.5.1.2: haproxy.cfg 配置文件:

```

# cat haproxy.cfg
# cat  haproxy.cfg
global
chroot /usr/local/haproxy
#stats socket /var/lib/haproxy/haproxy.sock mode 600 level admin
uid 99
gid 99
daemon
nbproc 1
pidfile /usr/local/haproxy/run/haproxy.pid
log 127.0.0.1 local3 info

defaults
option http-keep-alive
option forwardfor
mode http
timeout connect 300000ms
timeout client  300000ms
timeout server  300000ms

listen stats
  mode http
  bind 0.0.0.0:9999
  stats enable
  log global
  stats uri      /haproxy-status
  stats auth     haadmin:123456

listen  web_port_80

```

```
bind 0.0.0.0:80
mode http
log global
balance roundrobin
server web1 127.0.0.1:8800 check inter 3000 fall 2 rise 5

listen web_port_443
bind 0.0.0.0:443
mode http
log global
balance roundrobin
server web1 127.0.0.1:8843 check inter 3000 fall 2 rise 5
```

7.5.1.3: haproxy 运行脚本:

```
# cat run_haproxy.sh
#!/bin/bash

haproxy -f /etc/haproxy/haproxy.cfg
tail -f /etc/hosts
```

7.2.1.4: build-command 脚本:

```
# cat build-command.sh
#!/bin/bash

docker build -t 192.168.7.103/linux37/linux37-centos-haproxy:2.0.5 .
docker push 192.168.7.103/linux37/linux37-centos-haproxy:2.0.5
```

7.2.1.5: 执行镜像构建:

```
# bash build-command.sh

Removing intermediate container 84ddb49490e6
--> e59bb9608038
Successfully built e59bb9608038
Successfully tagged 192.168.7.103/linux37/linux37-centos-haproxy:2.0.5
The push refers to repository [192.168.7.103/linux37/linux37-centos-haproxy]
f1ca3cfc7d8e: Pushed
306e0fa6cdab: Pushed
f49df9d0e224: Pushing [=====] 12.75MB/67.06MB
32f7869be04d: Pushing [=====] 4.499MB/10.37MB
6f016a5ede2d: Pushing [=====] 27.77MB/116.7MB
3e6358a13843: Waiting
877b494a9f30: Waiting
```

7.5.2: 准备 nginx 镜像:

参考步骤 2.3 2.6 及 2.7

7.5.3: 准备 tomcat 镜像:

参考步骤 2.4

7.5.4: 编辑 docker compose 文件及环境准备:

7.5.4.1: 编辑 docker compose 文件:

```
# pwd  
/opt/maged  
  
# cat docker-compose.yml  
service-haproxy:  
    image: 192.168.7.103/linux37/linux37-centos-haproxy:2.0.5  
    container_name: haproxy  
    expose:  
        - 80  
        - 443  
        - 9999  
    ports:  
        - "80:80"  
        - "443:443"  
        - "9999:9999"  
    links:  
        - service-nginx-web  
  
service-nginx-web:  
    image: 192.168.7.103/linux37/ubuntu-nginx:1.16.1  
    container_name: nginx-web1  
    volumes:  
        - /data/nginx/magedu:/apps/nginx/html/magedu  
        - /data/nginx/static:/apps/nginx/html/static  
    expose:  
        - 80  
        - 443  
    # ports:  
    #     - "8800:80"  
    #     - "8443:443"  
    links:  
        - service-tomcat-app1  
        - service-tomcat-app2  
  
service-tomcat-app1:  
    image: 192.168.7.103/linux37/linux37-tomcat:app1  
    container_name: tomcat-app1
```

```
volumes:
  - /data/tomcat/webapps/magedu:/data/tomcat/webapps/app/magedu
expose:
  - 8080
# ports:
#   - "8801:8080"

service-tomcat-app2:
  image: 192.168.7.103/linux37/linux37-tomcat:app2
  container_name: tomcat-app2
  volumes:
    - /data/tomcat/webapps/magedu:/data/tomcat/webapps/app/magedu
  expose:
    - 8080
# ports:
#   - "8802:8080"
```

7.5.4.2: 准备 nginx 静态文件:

```
# mkdir /data/nginx/static
# echo "Nginx static page" > /data/nginx/static/index.html
上传宿主机图片到静态文件路径
```

```
# tree /data/nginx/static/
/data/nginx/static/
├── 1.jpeg
└── index.html
```

```
0 directories, 2 files
```

7.5.4.3: 准备 nginx.conf 配置文件:

```
#在 nginx 配置文件中，将用户访问 app 目录的请求通过 upstream 服务器组转发至后端 tomcat 容器。
```

```
# pwd
/opt/dockerfile/system/ubuntu

# grep -v "#" nginx.conf | grep -v "^\$"
user  nginx;
worker_processes  1;
daemon off;
events {
    worker_connections  1024;
}
http {
```

```

include      mime.types;
default_type application/octet-stream;
sendfile      on;
keepalive_timeout 65;
upstream tomcat_webserver {
    server service-tomcat-app1:8080;
    server service-tomcat-app2:8080;
}
server {
    listen      80;
    server_name localhost;
    location / {
        root      html;
        index     index.html index.htm;
    }
    location /linux37 {
        root /data/nginx/html;
        index index.html;
    }
    location /app {
        proxy_pass  http://tomcat_webserver;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Real-IP $remote_addr;
    }
    error_page  500 502 503 504  /50x.html;
    location = /50x.html {
        root      html;
    }
}
}

```

7.5.4.4: 准备 tomcat 页面文件:

```

# mkdir /data/tomcat/webapps/magedu -p
# cat /data/tomcat/webapps/magedu/showhost.jsp
<%@page import="java.util.Enumeration"%>
<br />
host:
<%try{out.println(""+java.net.InetAddress.getLocalHost().getHostName());}catch(Exc
eption e){}%>
<br />
remoteAddr: <%=request.getRemoteAddr()%>
<br />
remoteHost: <%=request.getRemoteHost()%>

```

```

<br />
sessionId: <%=request.getSession().getId()%>
<br />
serverName:<%=request.getServerName()%>
<br />
scheme:<%=request.getScheme()%>
<br />
<%request.getSession().setAttribute("t1","t2");%>
<%
    Enumeration en = request.getHeaderNames();
    while(en.hasMoreElements()){
        String hd = en.nextElement().toString();
            out.println(hd+" : "+request.getHeader(hd));
            out.println("<br />");
    }
%>

```

7.5.5: 启动容器:

```

# pwd
/opt/magedu
# docker-compose up -d
d111424f0dc4: Pull complete
7d967c85bef4: Pull complete
f80b6401069c: Pull complete
Digest: sha256:d2c2a81d81481c6742540fc4d6247677d48071f5fcf2bb7bcfe06d4d92175f76
Status: Downloaded newer image for 192.168.7.103/linux37/ubuntu-nginx:1.16.1
Pulling service-haproxy (192.168.7.103/linux37/linux37-centos-haproxy:2.0.5)...
2.0.5: Pulling from linux37/linux37-centos-haproxy
d8d02d457314: Already exists
5fd9af99db6d: Already exists
5b12d347ad0f: Pull complete
833c90b6ed8a: Pull complete
25d8530d28a7: Pull complete
4484db0b8665: Pull complete
cd4576a9a293: Pull complete
Digest: sha256:5b0f933c5b2c350cd14d1afd3fac9467e0e4df94063577c2a2bfc0444f973428
Status: Downloaded newer image for 192.168.7.103/linux37/linux37-centos-haproxy:2.0.5
Creating tomcat-app2 ... done
Creating tomcat-app1 ... done
Creating nginx-web1 ... done
Creating haproxy ... done
root@docker-node5:/opt/magedu#

```

7.5.6: 验证容器启动成功:

```

root@docker-node5:/opt/magedu# docker-compose ps
      Name          Command   State           Ports
----- 
haproxy      /usr/bin/run_haproxy.sh     Up      0.0.0.0:443->443/tcp, 0.0.0.0:80->80/tcp, 0.0.0.0:9999->9999/tcp
nginx-web1   nginx                  Up      443/tcp, 80/tcp
tomcat-app1  /apps/tomcat/bin/run_tomcat.sh Up      8080/tcp, 8443/tcp
tomcat-app2  /apps/tomcat/bin/run_tomcat.sh Up      8080/tcp, 8443/tcp
root@docker-node5:/opt/magedu#

```

7.5.7: 查看启动日志:

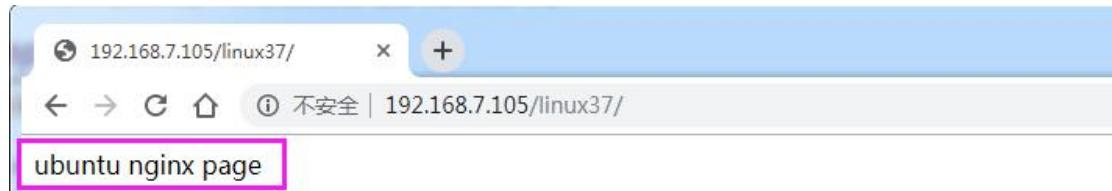
```
[root@docker-server3 docker-compose]# docker-compose logs -f
```

```
root@magedu:/opt/magedu# docker-compose logs -f
ERROR: No such service: - f
root@magedu:/opt/magedu# docker-compose logs -f
Attaching to haproxy, nginx-web1, tomcat-app2, tomcat-app1
tomcat-app1          Tomcat started.
tomcat-app1          127.0.0.1      localhost
tomcat-app1          ::1      localhost ip6-localhost ip6-loopback
tomcat-app1          fe00::0      ip6-localnet
tomcat-app1          ff00::0      ip6-mcastprefix
tomcat-app1          ff02::1      ip6-allnodes
tomcat-app1          ff02::2      ip6-allrouters
tomcat-app1          172.17.0.2    094bd40f081f
tomcat-app1          192.168.100.103 k8s-harbor1.local.com
haproxy              127.0.0.1      localhost
haproxy              ::1      localhost ip6-localhost ip6-loopback
haproxy              fe00::0      ip6-localnet
haproxy              ff00::0      ip6-mcastprefix
haproxy              ff02::1      ip6-allnodes
haproxy              ff02::2      ip6-allrouters
haproxy              172.17.0.4    nginx-web1 16c95ab02a49
haproxy              172.17.0.4    service-nginx-web 16c95ab02a49 nginx-web1
haproxy              172.17.0.5    a4fa3c63e832
tomcat-app2          Tomcat started.
tomcat-app2          127.0.0.1      localhost
tomcat-app2          ::1      localhost ip6-localhost ip6-loopback
tomcat-app2          fe00::0      ip6-localnet
tomcat-app2          ff00::0      ip6-mcastprefix
tomcat-app2          ff02::1      ip6-allnodes
tomcat-app2          ff02::2      ip6-allrouters
tomcat-app2          172.17.0.3    d6789e7059cf
tomcat-app2          192.168.100.103 k8s-harbor1.local.com
```

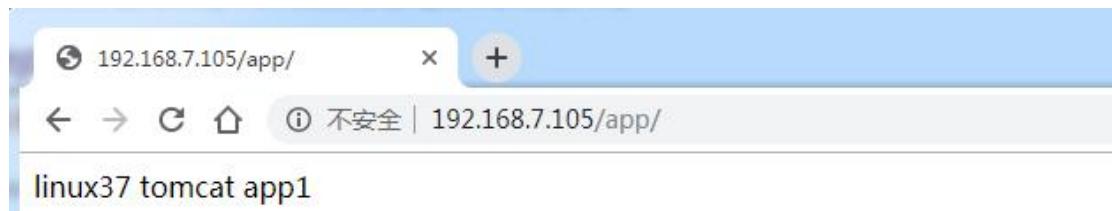
7.5.8: 访问 haroxy 管理界面:

<http://192.168.7.105:9999/haproxy-status>

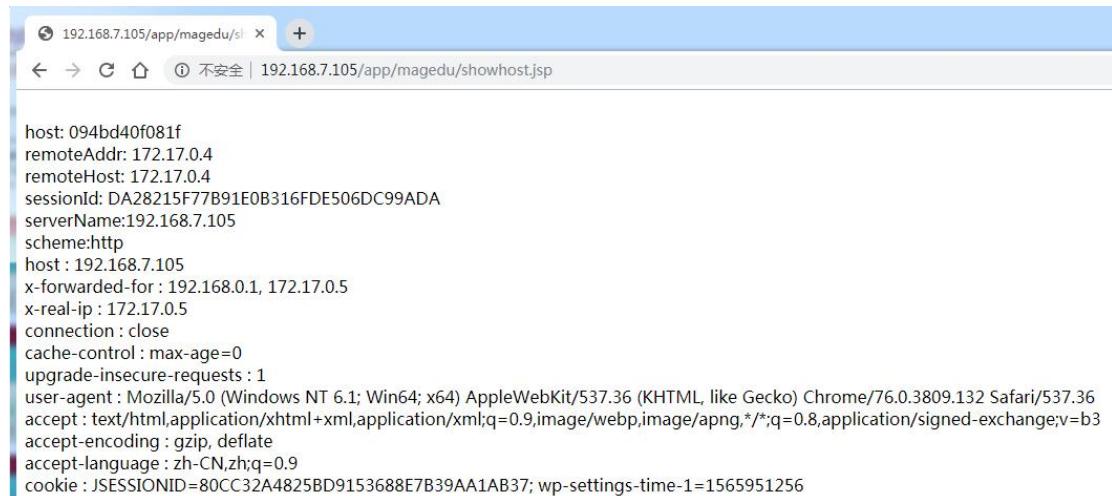
7.5.9: 访问 Nginx 静态页面:



7.5.10: 访问 tomcat 静态页面:



7.5.11: 访问 tomcat 动态页面:



7.5.12: 验证 Nginx 容器访问日志:

```
# docker exec -it 16c95ab02a49 bash
root@16c95ab02a49:/# tail /apps/nginx/logs/access.log
172.17.0.5 - [18/Sep/2019:07:06:12 +0000] "GET /app/magedu/showhost.jsp
HTTP/1.1" 200 842 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
AppleWebKit/537.36 (KHTML, like Ge
cko) Chrome/76.0.3809.132 Safari/537.36"172.17.0.5 - [18/Sep/2019:07:06:44
+0000] "GET /app HTTP/1.1" 302 5 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.
132 Safari/537.36"172.17.0.5 - [18/Sep/2019:07:07:04 +0000] "GET /linux37/
HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36
```

(KHTML, like Gecko) Chrome/76.0.

3809.132 Safari/537.36"172.17.0.5 - - [18/Sep/2019:07:07:04 +0000] "GET /linux37/
HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36

(KHTML, like Gecko) Chrome/76.0.

3809.132 Safari/537.36"