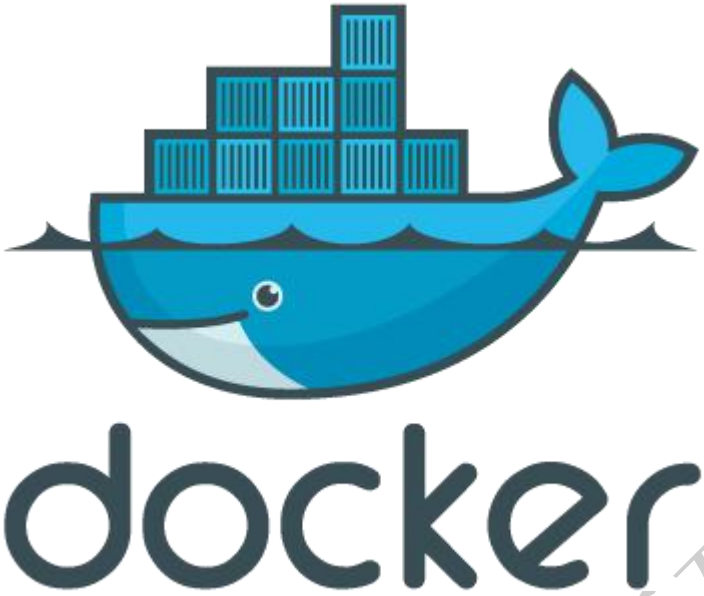
Docker学习手册



作者：itcast

版本：V 1.0

文档编号：

日期：2017年12月11日

目录

第 1 章 Docker基础

1.1 docker简介

在这一部分我们主要讲两个方面：

docker是什么、docker特点

1.1.1 docker是什么

docker是什么？

Docker是一个开源的容器引擎，它基于LXC容器技术，使用Go语言开发。

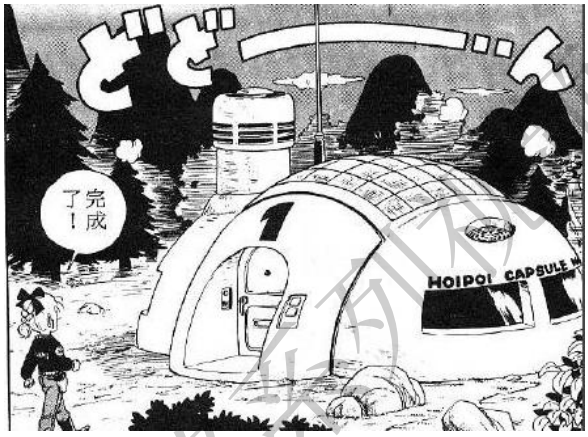
源代码托管在Github上，并遵从Apache2.0协议。

Docker采用C/S架构，其可以轻松的为任何应用创建一个轻量级的、可移植的、自给自足的容器。

简单来说：Docker就是一种快速解决生产问题的一种技术手段。

Docker生活场景：

图一：动画片《七龙珠》里面的胶囊



图二：1号胶囊启动后的效果

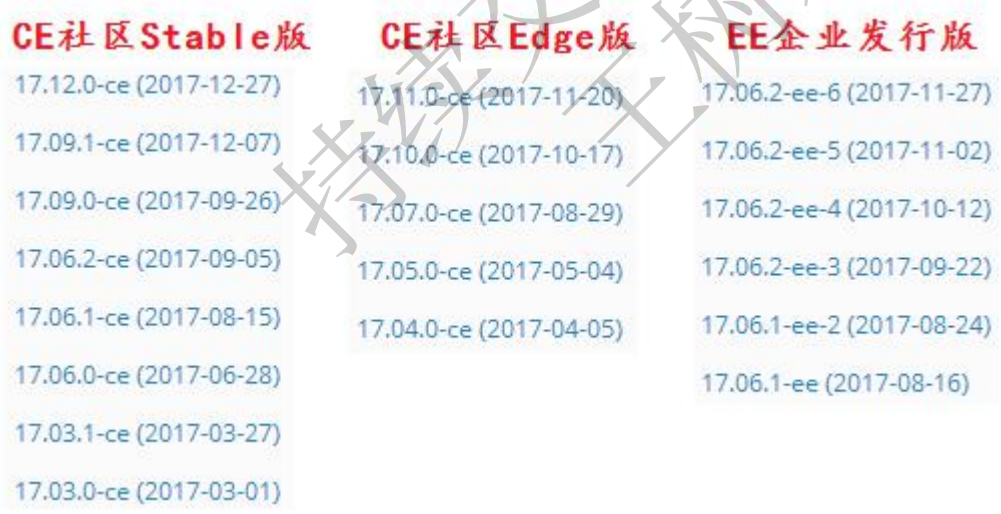
docker历程：

自2013年开始出现以来，发展势头很猛，赶上了好时候。

docker 从1.13x开始，版本分为了社区版ce和企业版ee，并且基于年月的时间线行驶

目前最新版本

社区版ce：17.12.0-ce(2017-12-27)　　企业版ee：17.06.2-ee-6(2017-11-27)



注：

Stable：gives you reliable updates every quarter

Edge：gives you new features every month

官方强烈要求：全部升级到1.12版本以上

官方资料：

Docker 官网：http://www.docker.com

Github Docker 源码：https://github.com/docker/docker

1.1.2 docker特点

三大理念：

构建：

龙珠里的胶囊，将你需要的场景构建好，装在一个小胶囊里

运输：

随身携带着房子、车子等，非常方便

运行：

只需要你轻轻按一下胶囊，找个合适的地方一放，就ok了

优点：

多：　适用场景多

快：　环境部署快、更新快

好：　好多人在用，东西好

省：　省钱省力省人工

缺点：

太腻歪人：　依赖操作系统

不善于沟通：依赖网络

不善理财：　银行U盾等场景不能用

1.2 docker快速入门

1.2.1 官方要求

为什么用ubuntu学docker

Platform	Docker CE x86_64	Docker CE ARM	Docker CE ARM64	Docker CE IBM Z (s390x)	Docker EE x86_64	Docker EE IBM Z (s390x)
CentOS	✓				✓	
Debian	✓	✓	✓			
Fedora	✓					
Microsoft Windows Server 2016					✓	
Oracle Linux					✓	
Red Hat Enterprise Linux					✓	✓
SUSE Linux Enterprise Server					✓	✓
Ubuntu	✓	✓	✓	✓	✓	✓

图片来源：https://docs.docker.com/engine/installation/#server

docker要求的ubuntu环境

OS requirements

To install Docker CE, you need the 64-bit version of one of these Ubuntu versions:

- Artful 17.10 (Docker CE 17.11 Edge and higher only)
- Zesty 17.04
- Xenial 16.04 (LTS)
- Trusty 14.04 (LTS)

ubuntu主机环境需求

执行命令

```
uname -a
ls -l /sys/class/misc/device-mapper
```

执行效果

```
root@admina-virtual-machine:~# uname -a
Linux admina-virtual-machine 4.8.0-36-generic #36~16.04.1-Ubuntu SMP Sun Feb 5 0
9:39:57 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
root@admina-virtual-machine:~# ls -l /sys/class/misc/device-mapper
lrwxrwxrwx 1 root root 0 Jan  7 23:56 /sys/class/misc/device-mapper -> ../../dev
ices/virtual/misc/device-mapper
```

1.2.2 部署docker

官网参考:

https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#upgrade-docker-after-using-the-convenience-script

安装基本软件

```
apt-get update
apt-get install apt-transport-https ca-certificates curl software-properties-common -y
```

使用官方推荐源

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

使用阿里云的源**{推荐}**

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

软件源升级

```
apt-get update
```

安装docker

```
apt-get install docker-ce -y
```

注:

可以指定版本安装docker:

```
apt-get install docker-ce=<VERSION> -y
```

查看支持的docker版本

```
apt-cache madison docker-ce
```

测试docker

```
docker version
```

网卡区别：

安装前：只有ens33和lo网卡

安装后：docker启动后，多出来了docker0网卡，网卡地址172.17.0.1

1.2.3 docker加速器

在国内使用docker的官方镜像源，会因为网络的原因，造成无法下载，或者一直处于超时。所以我们使用 daocloud 的方法进行加速配置。

方法：

访问 daocloud.io 网站，登录 daocloud 账户

点击右上角的 加速器

在新窗口处会显示一条命令，

我们执行这条命令

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://74f21445.m.daocloud.io
```

修改daemon.json文件，增加绿色背景字体内容

```
# cat /etc/docker/daemon.json
{"registry-mirrors": ["http://74f21445.m.daocloud.io"], "insecure-registries": []}
```

注意：

docker cloud加速器的默认内容是少了一条配置，所以我们要编辑文件在后面加上绿色背景的内容，然后再重启docker

重启docker

```
systemctl restart docker
```

1.2.4 docker 其他简介

docker软件的基本命令格式：

systemctl [参数] docker

参数详解：

start	开启服务
stop	关闭
restart	重启
status	状态

删除docker命令：

```
apt-get purge docker-ce -y
rm -rf /var/lib/docker/
rm -rf /etc/docker
```

docker基本目录简介

/etc/docker/	docker的认证目录
/var/lib/docker/	docker的应用目录

第 2 章 Docker 核心技术

Docker的核心技术内容很多，我们分别从以下五个方面来介绍Docker的核心技术

镜像、容器、仓库、数据、网络

2.1 docker镜像管理

2.1.1 镜像简介

Docker镜像是什么？

它是一个只读的文件，就类似于我们安装操作系统时候所需要的那个iso光盘镜像，通过运行这个镜像来完成各种应用的部署。

这里的镜像就是一个能被docker运行起来的一个程序。

2.1.2 搜索、查看、获取

搜索镜像

命令格式：

```
docker search [image_name]
```

命令演示：

```
docker search ubuntu
```

获取镜像

命令格式：

```
docker pull [image_name]
```

命令演示：

```
docker pull ubuntu
docker pull nginx
```

注释：

获取的镜像在哪里？

/var/lib/docker 目录下，具体详见docker仓库知识

查看镜像

命令格式：

```
docker images <image_name>
```

命令演示：

```
docker images
```

镜像的ID唯一标识了镜像，如果ID相同，说明是同一镜像。TAG信息来区分不同发行版本，如果不指定具体标记，默认使用latest标记信息

docker images -a 列出所有的本地的images（包括已删除的镜像记录）

2.1.3 重命名、删除

镜像重命名

命令格式：

```
docker tag [old_image]:[old_version] [new_image]:[new_version]
```

命令演示：

```
docker tag nginx:latest sswang-nginx:v1.0
```

删除镜像

命令格式：

```
docker rmi [image_id/image_name:image_version]
```

命令演示：

```
docker rmi 3fa822599e10
```

注意：

如果一个image_id存在多个名称，那么应该使用name:tag的格式删除镜像

2.1.4 导出、导入

导出镜像

将已经下载好的镜像，导出到本地，以备后用。

命令格式：

```
docker save -o [镜像名称] [镜像]
```

导出镜像

```
docker save -o nginx.tar sswang-nginx
```

导入镜像

为了更好的演示效果，我们先将nginx的镜像删除掉

```
docker rmi nginx:v1.0
docker rmi nginx
```

导入镜像命令格式：

```
docker load < [image.tar_name]
docker load --input [image.tar_name]
```

导入镜像文件

```
docker load < nginx.tar
```

2.1.5 历史、创建

查看镜像历史

查看镜像历史命令格式：

```
docker history [image_name]
```

我们获取到一个镜像，想知道他默认启动了哪些命令或者都封装了哪些系统层，那么我们可以使用docker history这条命令来获取我们想要的信息

```
docker history sswang-nginx:v1.0
```

根据模板创建镜像

登录系统模板镜像网站：

https://download.openvz.org/template/precreated/

找到一个镜像模板进行下载，比如说ubuntu-14.04-x86_64-minimal.tar.gz，地址为：

https://download.openvz.org/template/precreated/ubuntu-14.04-x86_64-minimal.tar.gz

命令格式：

```
cat 模板文件名.tar | docker import - [自定义镜像名]
```

演示效果：

```
cat ubuntu-14.04-x86_64-minimal.tar.gz | docker import - ubuntu-mini
```

2.2 容器管理

2.2.1 容器简介

容器是什么？

容器就类似于我们运行起来的一个操作系统，而且这个操作系统启动了某些服务。

这里的容器指的是运行起来的一个Docker镜像。

2.2.2 查看、启动

查看容器

命令格式：docker ps

```
# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
        NAMES
```

注意：

管理docker容器可以通过名称，也可以通过ID

ps是显示正在运行的容器， -a是显示所有运行过的容器，包括已经不运行的容器

启动容器

启动容器有三种方式

1、基于镜像新建一个容器并启动

2、将关闭的容器重新启动

3、守护进程方式启动docker

创建新容器并启动

命令格式：docker run <参数，可选> [docker_image] [执行的命令]

启动一个镜像，输入信息后关闭容器

```
docker run nginx /bin/echo "hello docker"
```

注意：

docker run 其实 是两个命令的集合体 docker create + docker start

启动已终止的容器

在生产过程中，常常会出现运行和不运行的容器，我们使用 start 命令开起一个已关闭的容器

命令格式：docker start [container_id]

守护进程方式启动容器

更多的时候，需要让Docker容器在后台以守护形式运行。此时可以通过添加-d参数来实现

命令格式：docker run -d [image_name] command ...

守护进程方式启动容器

```
 docker run -d nginx
```

2.2.3 关闭、删除

关闭容器

在生产中，我们会以为临时情况，要关闭某些容器，我们使用 stop 命令来关闭某个容器

命令格式：docker stop [container_id]

关闭容器id

```
docker stop 8005c40a1d16
```

删除容器

删除容器有两种方法：

正常删除 -- 删除已关闭的

强制删除 -- 删除正在运行的

正常删除容器

命令格式： docker rm [container_id]

删除已关闭的容器

```
docker rm 1a5f6a0c9443
```

强制删除运行容器

命令格式：docker rm -f [container_id]

删除正在运行的容器

```
docker rm -f 8005c40a1d16
```

拓展批量关闭容器

命令格式：docker rm -f $(docker ps -a -q)

2.2.4 进入、退出

进入容器我们学习三种方法：

1、创建容器的同时进入容器

2、手工方式进入容器

3、生产方式进入容器

创建并进入容器

命令格式：docker run --name [container_name] -it [docker_image] /bin/bash

```
~]# docker run -it --name sswang-nginx nginx /bin/bash
root@7c5a24a68f96:/# echo "hello world"
hello world
root@7c5a24a68f96:/# exit
exit
```

docker 容器启动命令参数详解：

--name:给容器定义一个名称

-i:则让容器的标准输入保持打开。

-t:让docker分配一个伪终端,并绑定到容器的标准输入上

/bin/bash:执行一个命令

退出容器：

方法一：exit

方法二：Ctrl + D

手工方式进入容器

命令格式：

docker exec -it 容器id /bin/bash

效果演示：

```
docker exec -it d74fff341687 /bin/bash
```

生产方式进入容器

我们生产中常用的进入容器方法是使用脚本，脚本内容如下

```
#!/bin/bash
# 定义进入仓库函数
docker_in(){
  NAME_ID=$1
  PID=$(docker inspect -f "{{ .State.Pid }}" $NAME_ID)
  nsenter -t $PID -m -u -i -n -p
}
docker_in $1
```

赋权执行

```
chmod +x docker_in.sh
```

进入指定的容器，并测试

```
./docker_in.sh b3fbcba852fd
```

2.2.5 基于容器创建镜像

方式一：

命令格式：

docker commit -m '改动信息' -a "作者信息" [container_id] [new_image:tag]

命令演示：

进入一个容器，创建文件后并退出

```
./docker_in.sh d74fff341687
mkdir /sswang
exit
```

创建一个镜像

```
docker commit -m 'mkdir /sswang' -a "sswang" d74fff341687 sswang-nginx:v0.2
```

查看镜像

```
docker images
```

启动一个容器

```
docker run -itd sswang-nginx:v0.2 /bin/bash
```

进入容器进行查看

```
./docker_in.sh ae63ab299a84
ls
```

方式二：

命令格式：

```
docker export [容器id] > 模板文件名.tar
```

命令演示：

创建镜像

```
docker export ae63ab299a84 > gaoji.tar
```

导入镜像

```
cat gaoji.tar | docker import - sswang-test
```

2.2.6 日志、信息

查看容器运行日志

命令格式：

```
docker logs [容器id]
```

命令效果：

```
docker logs 7c5a24a68f96
```

查看容器详细信息

命令格式：

```
docker inspect [容器id]
```

命令效果：

查看容器全部信息

```
docker inspect 930f29ccdf8a
```

查看容器网络信息

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 930f29ccdf8a
```

查看容器端口信息

命令格式：

```
docker port [容器 id]
```

命令效果：

```
docker port 930f29ccdf8a
```

2.3 仓库管理

2.3.1 仓库简介

仓库是什么？

仓库就类似于我们在网上搜索操作系统光盘的一个镜像站。

这里的仓库指的是Docker镜像存储的地方。

Docker的仓库有两大类：

公有仓库：Docker hub、Docker cloud、等

私有仓库：registry、harbor、等

docker私有仓库：

不安全的

安全的

传输数据携带tls标识

和仓库相关的命令：

```
docker login [仓库名称]
docker pull [镜像名称]
docker push [镜像名称]
docker search [镜像名称]
```

我们接下来就用registry来部署一个私有的仓库

2.3.2 私有仓库部署

创建仓库流程

1、根据registry镜像创建容器

2、配置仓库权限

3、提交镜像到私有仓库

4、测试

实施方案

下载registry镜像

```
docker pull registry
```

启动仓库容器

```
docker run -d -p 5000:5000 registry
```

检查容器效果

```
curl 127.0.0.1:5000/v2/_catalog
```

配置容器权限

```
vim /etc/docker/daemon.json
{"registry-mirrors": ["http://74f21445.m.daocloud.io"], "insecure-registries": ["192.168.8.14:5000"]}
```

注意：

私有仓库的ip地址是宿主机的ip，而且ip两侧有双引号

重启docker服务

```
systemctl restart docker
systemctl status docker
```

启动容器

```
docker start 315b5422c699
```

标记镜像

```
docker tag ubuntu-mini 192.168.8.14:5000/ubuntu-14.04-mini
```

提交镜像

```
docker push 192.168.8.14:5000/ubuntu-14.04-mini
```

下载镜像

```
docker pull 192.168.8.14:5000/ubuntu-14.04-mini
```

2.4 数据管理

docker的镜像是只读的，虽然依据镜像创建的容器可以进行操作，但是我们不能将数据保存到容器中，因为容器会随时关闭和开启，那么如何将数据保存下来呢？

答案就是：数据卷和数据卷容器

2.4.1 数据卷简介

什么是数据卷？

就是将宿主机的某个目录，映射到容器中，作为数据存储的目录，我们就可以在宿主机对数据进行存储

缺点是：太单一了

docker 数据卷命令详解

```
# docker run --help
。。。
  -v, --volume list              Bind mount a volume (default [])
                                 挂载一个数据卷，默认为空
```

我们可以使用命令 docker run 用来创建容器，可以在使用docker run 命令时添加 -v 参数，就可以创建并挂载一个到多个数据卷到当前运行的容器中。

-v 参数的作用是将宿主机的一个目录作为容器的数据卷挂载到docker容器中，使宿主机和容器之间可以共享一个目录，如果本地路径不存在，Docker也会自动创建。

-v 宿主机文件:容器文件

2.4.2 数据卷实践

关于数据卷的管理我们从两个方面来说：

1、目录

2、普通文件

数据卷实践 之 目录

命令格式：

```
docker run -itd --name [容器名字] -v [宿主机目录]:[容器目录] [镜像名称] [命令(可选)]
```

命令演示：

创建测试文件

```
echo "file1" > /tmp/file1.txt
```

启动一个容器，挂载数据卷

```
docker run -itd --name test1 -v /tmp:/test1 nginx
```

测试效果

```
~# docker exec -it a53c61c77 /bin/bash
root@a53c61c77bde:/# cat /test1/file1.txt
file1
```

数据卷实践 之 文件

命令格式：

```
docker run -itd --name [容器名字] -v [宿主机文件]:[容器文件] [镜像名称] [命令(可选)]
```

命令演示：

创建测试文件

```
echo "file1" > /tmp/file1.txt
```

启动一个容器，挂载数据卷

```
docker run -itd --name test2 -v /tmp/file1.txt:/nihao/nihao.sh nginx
```

测试效果

```
~# docker exec -it 84c37743 /bin/bash
root@84c37743d339:/# cat /nihao/nihao.sh
file1
```

2.5 网络管理

Docker 网络很重要，重要的，我们在上面学到的所有东西都依赖于网络才能工作。我们从两个方面来学习网络：端口映射和网络模式

为什么先学端口映射呢？

在一台主机上学习网络，学习端口映射最简单，避免过多干扰。

2.5.1 端口映射详解

默认情况下，容器和宿主机之间网络是隔离的，我们可以通过端口映射的方式，将容器中的端口，映射到宿主机的某个端口上。这样我们就可以通过 宿主机的ip+port的方式来访问容器里的内容

Docker的端口映射

1、随机映射 -P(大写)

2、指定映射 -p 宿主机端口:容器端口

注意：

生产场景一般不使用随机映射，但是随机映射的好处就是由docker分配，端口不会冲突，

不管哪种映射都会影响性能，因为涉及到映射

2.5.2 随机映射实践

随机映射我们从两个方面来学习：

默认随机映射

指定主机随机映射

默认随机映射

命令格式：

```
docker run -d -P [镜像名称]
```

命令效果：

启动一个 nginx 镜像

```
docker run -d -P nginx
```

查看效果

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
930f29ccdf8a        nginx               "nginx -g 'daemon of…"   3 seconds ago       Up 3 seconds        0.0.0.0
:32768->80/tcp      peaceful_pike
```

注意：

宿主机的32768被映射到容器的80端口

-P 自动绑定所有对外提供服务的容器端口，映射的端口将会从没有使用的端口池中自动随机选择，但是如果连

续启动多个容器的话，则下一个容器的端口默认是当前容器占用端口号+1

在浏览器中访问 http://192.168.8.14:32768/，效果显示：

注意：

浏览器输入的格式是： docker容器宿主机的ip:容器映射的端口

2.5.3 指定映射实践

指定端口映射我们从三个方面来讲：

指定端口映射

指定多端口映射

指定端口协议映射

指定端口映射

命令格式：

```
docker run -d -p [宿主机ip]:[宿主机端口]:[容器端口] --name [容器名字] [镜像名称]
```

注意：

如果不指定宿主机ip的话，默认使用 0.0.0.0，

端口映射默认使用的协议是tcp协议。如果想用udp的话，格式如下

```
docker run -d -p [宿主机ip]:[宿主机端口]:[容器端口]/udp --name [容器名字] [镜像名称]
```

使用udp协议的业务一般使用在 dns业务

命令实践：

现状我们在启动容器的时候，给容器指定一个访问的端口 1199

```
docker run -d -p 192.168.8.14:1199:80 --name nginx-1 nginx
```

查看新容器ip

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 0ad3acfbfb76
```

查看容器端口映射

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
0ad3acfbfb76        nginx               "nginx -g 'daemon of…"   37 seconds ago      Up 36 seconds       192.168
.8.14:1199->80/tcp  nginx-1
930f29ccdf8a        nginx               "nginx -g 'daemon of…"   25 minutes ago      Up 25 minutes       0.0.0.0
:32768->80/tcp      peaceful_pike
```

查看宿主机开启端口

```
root@admina-virtual-machine:~# netstat -tnulp | grep docker-proxy
tcp        0      0 192.168.8.14:1199       0.0.0.0:*               LISTEN      58951/docker-proxy
tcp6       0      0 :::32768                :::*                    LISTEN      58487/docker-proxy
```

查看浏览器效果：

2.5.4 网络管理基础

docker网络命令

查看网络命令帮助

```
~# docker network help
...
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```

查看当前主机网络

```
~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
8a18574f0f27        bridge              bridge              local
0925619f069d        host                host                local
e8800439307e        none                null                local
```

查看bridge网络的信息

```
~# docker network inspect 8a18574f0f27
[
    {
        "Name": "bridge",
        ...
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
        ...
```

2.5.5 网络模式简介

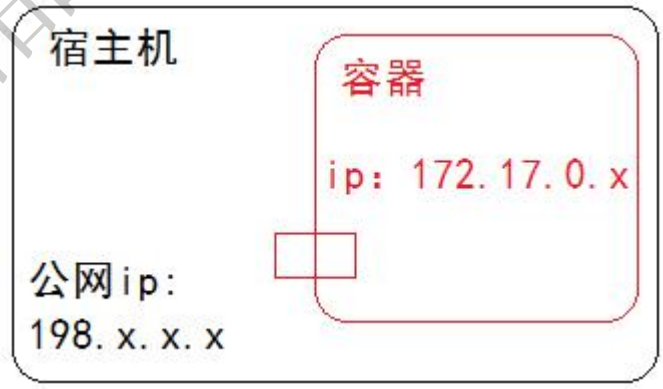
docker的网络模式

bridge模式：

简单来说：就是穿马甲，打着宿主机的旗号，做自己的事情。

Docker的默认模式，它会在docker容器启动时候，自动配置好自己的网络信息，同一宿主机的所有容器都在一个网络下，彼此间可以通信。类似于我们vmware虚拟机的nat模式。

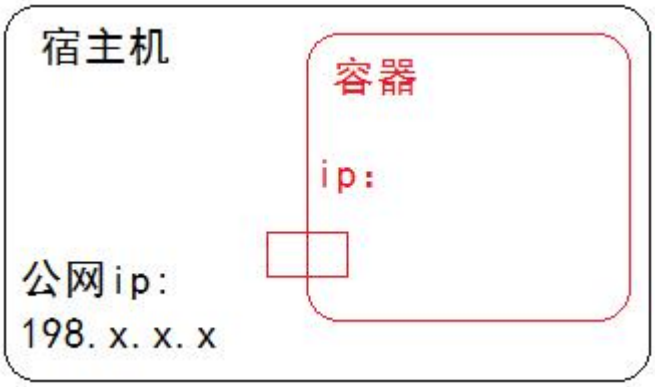
利用宿主机的网卡进行通信，因为涉及到网络转换，所以会造成资源消耗，网络效率会低。



host模式：

简单来说，就是鸠占鹊巢，用着宿主机的东西，干自己的事情。容器使用宿主机的ip地址进行通信。

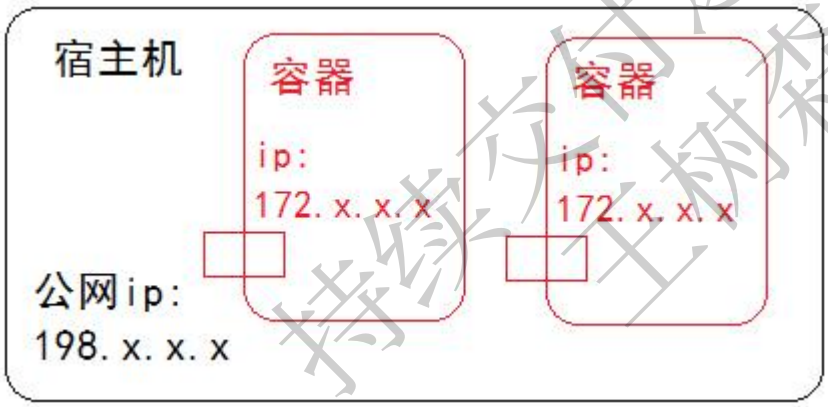
特点：容器和宿主机共享网络



container模式：

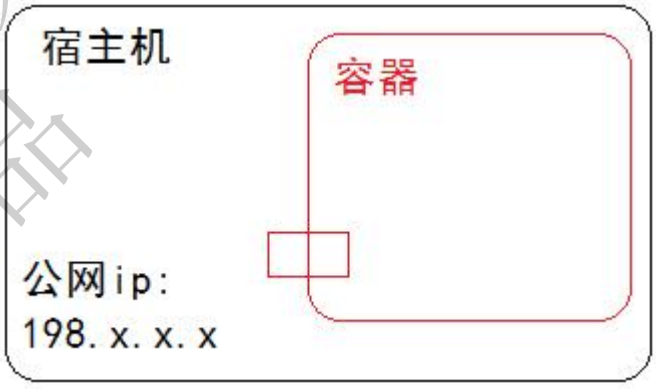
新创建的容器间使用使用已创建的容器网络，类似一个局域网。

特点：容器和容器共享网络



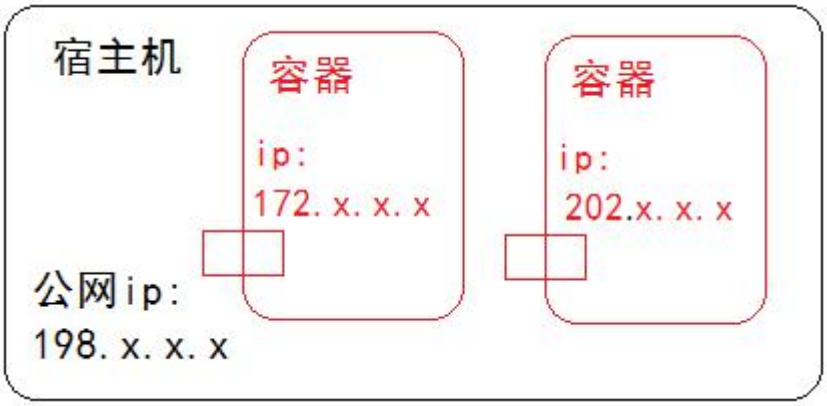
none模式：

这种模式最纯粹，不会帮你做任何网络的配置，可以最大限度的定制化。

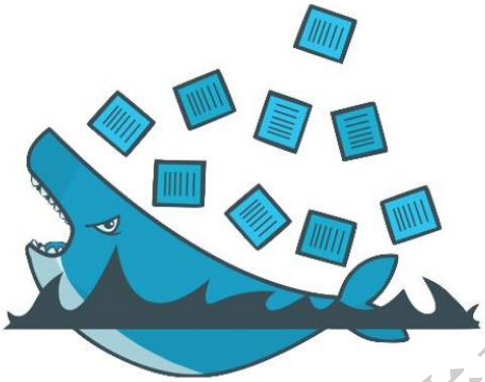


overlay模式：

容器彼此不再同一网络，而且能互相通行。



第 3 章 Docker 高级实践



在这一部分我们主要来介绍一些Docker的高级内容：

Dockerfile 和 Docker compose

3.1 Dockerfile

Dockerfile我们从下面的几个方面来介绍：

Dockerfile简介

Dockerfile快速入门

Dockerfile详解

Dockerfile简单实践

3.1.1 Dockerfile简介

什么是Dockerfile

Dockerfile类似于我们学习过的脚本，将我们在上面学到的docker镜像，使用自动化的方式实现出来。

Dockerfile的作用

1、找一个镜像：　　ubuntu
2、创建一个容器：　　docker run　ubuntu
3、进入容器：　　docker exec -it 容器 命令
4、操作：　　各种应用配置
　　....
5、构造新镜像：　　docker commit

Dockerfile 使用准则

1、大：　首字母必须大写D
2、空：　尽量将Dockerfile放在空目录中。
3、单：　每个容器尽量只有一个功能。
4、少：　执行的命令越少越好。

Dockerfile文件内容：

首行注释信息
指令(大写) 参数

Dockerfile 基础四指令：

基础镜像信息　　从哪来?
维护者信息　　我是谁?
镜像操作指令　　怎么干?
容器启动时执行指令　　嗨！！！

Dockerfile使用命令：

构建镜像命令格式:

```
docker build -t [镜像名]:[版本号] [Dockerfile所在目录]
```

构建样例:

```
docker build -t nginx:v0.2 /opt/dockerfile/nginx/
```

参数详解:

-t　　指定构建后的镜像信息,
/opt/dockerfile/nginx/　　则代表Dockerfile存放位置，如果是当前目录，则用 .(点)表示

3.1.2 Dockerfile快速入门

接下来我们快速的使用Dockerfile来创建一个定制化的镜像：nginx。

创建Dockerfile专用目录

```
mkdir /docker/images/nginx -p
cd /docker/images/nginx
```

创建Dockerfile文件

```
# 构建一个基于 ubuntu 的 docker 定制镜像
# 基础镜像
FROM ubuntu
# 镜像作者
MAINTAINER President.Wang 000000@qq.com
```

```
# 执行命令
RUN apt-get update
RUN apt-get install nginx -y

# 对外端口
EXPOSE 80
```

构建镜像

```
docker build -t nginx:v0.1 .
```

检查构建历史：

```
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
cf4899c5e48f   43 seconds ago  /bin/sh -c #(nop)  EXPOSE 80                    0B
13819e86fb10   43 seconds ago  /bin/sh -c apt-get install nginx -y             56.5MB
9e6d0167a088   2 minutes ago   /bin/sh -c apt-get update                       39.5MB
2817b46c27cc   4 minutes ago   /bin/sh -c #(nop)  MAINTAINER President.Wang…   0B
```

注意：

因为容器没有启动命令，所以肯定访问不了

修改一下Dockerfile文件：

```
修改 RUN 命令为一个
RUN apt-get update && apt-get install nginx -y
```

第二次构建镜像

```
docker build -t ubuntu-nginx:v0.2 .
```

查看构建历史

```
IMAGE          CREATED             CREATED BY                                      SIZE      COMMENT
b69834f6fa96   About a minute ago  /bin/sh -c #(nop)  EXPOSE 80                    0B
363530490797   About a minute ago  /bin/sh -c apt-get update && apt-get install…   96MB
2817b46c27cc   11 minutes ago      /bin/sh -c #(nop)  MAINTAINER President.Wang…   0B
```

思考：

两次构建有什么区别

```
REPOSITORY   TAG       IMAGE ID       CREATED             SIZE
nginx        v0.2      b69834f6fa96   About a minute ago  207MB
nginx        v0.1      cf4899c5e48f   8 minutes ago       207MB
nginx        latest    3f8a4339aadd   2 weeks ago         108MB
```

3.1.3 基础指令详解

基础指令

FROM

格式：

```
FROM <image>
FROM <image>:<tag>。
```

解释：

FROM 是 Dockerfile 里的第一条而且只能是除了首行注释之外的第一条指令

可以有多个 FROM 语句，来创建多个 image

FROM 后面是有效的镜像名称，如果该镜像没有在你的本地仓库，那么就会从远程仓库 Pull 取，如果远程也没有，就报错失败

下面所有的 系统可执行指令 在 FROM 的镜像中执行。

MAINTAINER

格式：

```
MAINTAINER <name>
```

解释：
指定该 dockerfile 文件的维护者信息。类似我们在 docker commit 时候使用-a 参数指定的信息

RUN

格式：
RUN <command> (shell 模式)
RUN["executable", "param1", "param2"]。 (exec 模式)
解释：
表示**当前镜像构建时候**运行的命令，如果有确认输入的话，一定要在命令中添加 -y
如果命令较长，那么可以在命令结尾使用 \ 来换行
生产中，推荐使用上面数组的格式
注释：
shell 模式：类似于 /bin/bash -c command
举例： RUN echo hello
exec 模式：类似于 RUN ["/bin/bash", "-c", "command"]
举例： RUN ["echo", "hello"]

EXPOSE

格式：
EXPOSE <port> [<port>...]
解释：
设置 Docker 容器对外暴露的端口号，Docker 为了安全，不会自动对外打开端口，如果需要外部提供访问，还需要启动容器时增加-p 或者-P 参数对容器的端口进行分配。

3.1.4 运行时指令详解

容器运行时指令

CMD

格式：
CMD ["executable","param1","param2"] (exec 模式)推荐
CMD command param1 param2 (shell 模式)
CMD ["param1","param2"] 提供给 ENTRYPOINT 的默认参数；
解释：
CMD 指定**容器启动时**默认执行的命令
每个 Dockerfile **只运行一条 CMD 命令**，如果指定了多条，只有最后一条会被执行
如果你在启动容器的时候使用 docker run 指定的运行命令，那么会**覆盖** CMD 命令。
举例： CMD ["/usr/sbin/nginx","-g","daemon off"]

ENTRYPOINT

格式：
ENTRYPOINT ["executable", "param1","param2"] (exec 模式)
ENTRYPOINT command param1 param2 (shell 模式)
解释：
和 CMD 类似都是配置容器启动后执行的命令，并且不会被 docker run 提供的参数覆盖。
每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个起效。
生产中我们可以同时使用 ENTRYPOINT 和 CMD，
想要在 docker run 时被覆盖，可以使用 "docker run --entrypoint"

CMD指令实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，末尾增加下面一句话：
CMD ["/usr/sbin/nginx","-g","daemon off;"]
```

构建镜像

```
docker build -t ubuntu-nginx:v0.3 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-3 -d ubuntu-nginx:v0.3
docker ps
```

根据镜像创建容器,创建时候，添加执行命令/bin/bash

```
docker run -p 80 --name nginx-4 -d ubuntu-nginx:v0.3 /bin/bash
docker ps
```

ENTRYPOINT指令实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，修改末尾的 CMD 为 ENTRYPOINT：
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
```

构建镜像

```
docker build -t ubuntu-nginx:v0.4 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-5 -d ubuntu-nginx:v0.4
docker ps
```

根据镜像创建容器,创建时候，添加执行命令/bin/bash

```
docker run -p 80 --name nginx-6 -d ubuntu-nginx:v0.4 /bin/bash
docker ps
```

根据镜像创建容器,创建时候，使用 --entrypoint参数，添加执行命令/bin/bash

```
docker run -p 80 --entrypoint "/bin/bash" --name nginx-7 -d ubuntu-nginx:v0.4
docker ps
```

对比：

三次执行效果

```
CONTAINER ID        IMAGE                 COMMAND                  CREATED             STATUS                          PORTS                     NAMES
c62b6a5e5d18        ubuntu-nginx:v0.4     "/bin/bash"              3 seconds ago       Exited (0) 2 seconds ago                                  nginx-7
2b1828cbf8f0        ubuntu-nginx:v0.4     "/usr/sbin/nginx -g …"   13 minutes ago      Exited (1) 13 minutes ago                                 nginx-6
e6bb09500367        ubuntu-nginx:v0.4     "/usr/sbin/nginx -g …"   14 minutes ago      Up 14 minutes                   0.0.0.0:32771->80/tcp     nginx-5
```

CMD ENTRYPOINT 综合使用实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，修改末尾的 ENTRYPOINT
ENTRYPOINT ["/usr/sbin/nginx"]
CMD ["-g"]
```

构建镜像

```
docker build -t ubuntu-nginx:v0.5 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-8 -d ubuntu-nginx:v0.5
docker ps
```

根据镜像创建容器,创建时候，不添加执行命令，覆盖cmd的参数 -g "daemon off;"

```
docker run -p 80 --name nginx-9 -d ubuntu-nginx:v0.5 -g "daemon off;"
docker ps
```

查看效果

```
CONTAINER ID        IMAGE                 COMMAND                  CREATED             STATUS                          PORTS                     NAMES
9a2c6fb933c5        ubuntu-nginx:v0.5     "/usr/sbin/nginx -g …"   3 seconds ago       Up 2 seconds                    0.0.0.0:32781->80/tcp     nginx-9
b8de4875745c        ubuntu-nginx:v0.5     "/usr/sbin/nginx -g"     24 seconds ago      Exited (1) 23 seconds ago                                 nginx-8
```

注释：

任何docker run设置的命令参数或者CMD指令的命令，都将作为ENTRYPOINT 指令的命令参数，追加到ENTRYPOINT指令之后

3.1.5 文件编辑指令详解

目录文件编辑指令

ADD

格式：

```
ADD <src>... <dest>
ADD ["<src>",... "<dest>"]
```

解释：

将指定的 <src> 文件复制到容器文件系统中的 <dest>

src 指的是宿主机，dest 指的是容器

所有拷贝到 container 中的文件和文件夹权限为 0755,uid 和 gid 为 0

如果文件是可识别的压缩格式，则 docker 会帮忙解压缩

注意：

1、如果源路径是个文件，且目标路径是以 / 结尾， 则 docker 会把目标路径当作一个目录，会把源文件拷贝到该目录下；

如果目标路径不存在，则会自动创建目标路径。

2、如果源路径是个文件，且目标路径是不是以 / 结尾，则 docker 会把目标路径当作一个文件。

如果目标路径不存在，会以目标路径为名创建一个文件，内容同源文件；

如果目标文件是个存在的文件，会用源文件覆盖它，当然只是内容覆盖，文件名还是目标文件名。

如果目标文件实际是个存在的目录，则会源文件拷贝到该目录下。 注意，这种情况下，最好显示的以 / 结尾，以避免混淆。

3、如果源路径是个目录，且目标路径不存在，则 docker 会自动以目标路径创建一个目录，把源路径目录下的文件拷贝进来。

如果目标路径是个已经存在的目录，则 docker 会把源路径目录下的文件拷贝到该目录下。

4、**如果源文件是个压缩文件，则 docker 会自动帮解压到指定的容器目录中**。

COPY

格式：

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"]
```

解释：

COPY 指令和 ADD 指令功能和使用方式类似。只是 COPY 指令不会做自动解压工作。

单纯复制文件场景，Docker 推荐使用 COPY

VOLUME

格式：

```
VOLUME ["/data"]
```

解释：

VOLUME 指令可以在镜像中创建挂载点，这样只要通过该镜像创建的容器都有了挂载点

通过 VOLUME 指令创建的挂载点，无法指定主机上对应的目录，是自动生成的。

举例：

```
VOLUME ["/var/lib/tomcat7/webapps/"]
```

ADD实践

拷贝普通文件

Dockerfile文件内容

```
...
# 执行命令
...
```

```
# 增加文件
ADD ["sources.list","/etc/apt/sources.list"]
...
```

构建镜像

```
docker build -t ubuntu-nginx:v0.6 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-10 -d ubuntu-nginx:v0.6
docker ps
```

进入容器查看效果

拷贝压缩文件

Dockerfile文件内容

```
...
# 执行命令
...
# 增加文件
ADD ["linshi.tar.gz","/nihao/"]
...
```

构建镜像

```
docker build -t ubuntu-nginx:v0.7 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-11 -d ubuntu-nginx:v0.7
docker ps
```

进入容器查看效果

COPY实践

修改Dockerfile文件内容:

```
...
# 执行命令
...
# 增加文件
COPY index.html /var/www/html/
...
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
```

index.html 文件内容:

```
<h1>你好 docker</h1>
```

构建镜像

```
docker build -t ubuntu-nginx:v0.8 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-12 -d ubuntu-nginx:v0.8
docker ps
```

浏览器访问nginx查看效果

VOLUME实践

修改Dockerfile文件内容:

```
# 在上一个 Dockerfile 文件内容基础上，在 COPY 下面增加一个 VOLUME
VOLUME ["/data/"]
```

```
...
```

构建镜像

```
docker build -t ubuntu-nginx:v0.9 .
```

创建容器

```
docker run -itd --name nginx-13 ubuntu-nginx:v0.9
```

查看镜像信息

```
docker inspect nginx-13
```

验证操作

```
docker run -itd --name vc-nginx-1 --volumes-from nginx-11 nginx
docker run -itd --name vc-nginx-2 --volumes-from nginx-11 nginx
```

进入容器1

```
docker exec -it vc-nginx-1 /bin/bash
echo 'nihao itcast' > data/nihao.txt
```

进入容器2

```
docker exec -it vc-nginx-2 /bin/bash
cat data/nihao.txt
```

3.1.6 环境指令详解

环境设置指令

USER

格式：

```
USER daemon
```

解释：

指定运行容器时的用户名和 UID，后续的 RUN 指令也会使用这里指定的用户。

如果不输入任何信息，表示默认使用 root 用户

ENV

格式：

```
ENV <key> <value>
ENV <key>=<value> ...
```

解释：

设置环境变量，可以在 RUN 之前使用，然后 RUN 命令时调用，容器启动时这些环境变量都会被指定

WORKDIR

格式：

```
WORKDIR /path/to/workdir                    (shell 模式)
```

解释：

切换目录，为后续的 RUN、CMD、ENTRYPOINT 指令配置工作目录。 相当于 cd

可以多次切换(相当于 cd 命令)，

也可以使用多个 WORKDIR 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如

举例：

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

则最终路径为 /a/b/c。

ARG

格式：

```
ARG <name>[=<default value>]
```

解释：

ARG 指定了一个变量在 docker build 的时候使用，可以使用--build-arg <varname>=<value>来指定参数的值，不过如果构建的时候不指定就会报错。

ENV实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，在 RUN 下面增加一个 ENV
ENV NIHAO=helloworld
```

...

构建镜像

```
docker build -t ubuntu-nginx:v0.10 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-13 -d ubuntu-nginx:v0.10
docker exec -it 54f86f714083 /bin/bash
echo $NIHAO
```

WORKDIR实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，在 RUN 下面增加一个 WORKDIR
WORKDIR /nihao/itcast/
RUN ["touch","itcast.txt"]
```

...

构建镜像

```
docker build -t ubuntu-nginx:v0.11 .
```

根据镜像创建容器,创建时候，不添加执行命令

```
docker run -p 80 --name nginx-14 -d ubuntu-nginx:v0.11
docker exec -it 54f76f520083 /bin/bash
ls
```

3.1.7 触发器指令详解

触发器指令

ONBUILD

格式：

```
ONBUILD [command]
```

解释：

当一个镜像A被作为其他镜像B的基础镜像时，这个触发器才会被执行，
新镜像B在构建的时候，会插入触发器中的指令。

触发器实践

修改Dockerfile文件内容：在COPY 前面增加 ONBUILD

```
# 构建一个基于 ubuntu 的 docker 定制镜像
# 基础镜像
FROM ubuntu

# 镜像作者
MAINTAINER President.Wang 000000@qq.com
```

```
# 执行命令
RUN apt-get update
RUN apt-get install nginx -y

# 增加文件
ONBUILD COPY index.html /var/www/html/

# 对外端口
EXPOSE 80

# 执行命令 ENTRYPOINT:
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
```

构建镜像

```
docker build -t ubuntu-nginx:v0.12 .
```

根据镜像创建容器，

```
docker run -p 80 --name nginx-15 -d ubuntu-nginx:v0.12
docker ps
```

访问容器页面，是否被更改

构建子镜像Dockerfile文件

```
FROM ubuntu-nginx:v0.12
MAINTAINER President.Wang 000000@qq.com
EXPOSE 80
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
```

构建子镜像

```
docker build -t ubuntu-nginx-sub:v0.1 .
```

根据镜像创建容器，

```
docker run -p 80 --name nginx-16 -d ubuntu-nginx-sub:v0.1
docker ps
```

访问容器页面，是否被更改

3.1.8 Dockerfile构建过程

Dockerfile构建过程：

从基础镜像1运行一个容器A
遇到一条Dockerfile指令，都对容器A做一次修改操作
执行完毕一条命令，提交生成一个新镜像2
再基于新的镜像2运行一个容器B
遇到一条Dockerfile指令，都对容器B做一次修改操作
执行完毕一条命令，提交生成一个新镜像3
...

构建过程镜像介绍

构建过程中，创建了很多镜像，这些中间镜像，我们可以直接使用来启动容器，通过查看容器效果，从侧面能看到我们每次构建的效果。

提供了镜像调试的能力

构建缓存

我们第一次构建很慢，之后的构建都会很快，因为他们用到了构建的缓存。

不适用构建缓存方法常见两种：

全部不同缓存：

```
docker build --no-cache -t [镜像名]:[镜像版本] [Dockerfile位置]
```

部分使用缓存：

```
ENV REFRESH_DATE 2018-01-12
```

只要构建的缓存时间不变，那么就用缓存，如果时间一旦改变，就不用缓存了

样例：

```
# 构建一个基于 ubuntu 16.04 的 docker 定制镜像
# 基础镜像
FROM ubuntu-16.04

# 镜像作者
MAINTAINER President.Wang 000000@qq.com

# 创建构建刷新时间
ENV REFRESH_DATE 2018-01-12

# 执行命令
RUN apt-get update
RUN apt-get install nginx -y

# 对外端口
EXPOSE 80
```

构建历史：

查看构建过程查看

```
docker history
```

3.3 Docker compose

Docker compose是一种docker容器的任务编排工具

官方地址：https://docs.docker.com/compose/

3.3.1 简介

任务编排介绍

场景：

我们在工作中为了完成业务目标，首先把业务拆分成多个子任务，然后对这些子任务进行顺序组合，当子任务按照方案执行完毕后，就完成了业务目标。

任务编排，就是对多个子任务执行顺序进行确定的过程。

常见的任务编排工具：

单机版：docker compose

集群版：

Docker swarm	Docker
Mesos	Apache
kubernetes	Google

docker compose是什么

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see the list of features.

docker compose的特点；

本质：docker 工具
对象：应用服务
配置：YAML 格式配置文件
命令：简单
执行：定义和运行容器

docker compose的配置文件

docker-compose.yml
文件后缀是yml
文件内容遵循 ymal格式

docker 和 Docker compose

Compose file format	Docker Engine release		Compose file format	Docker Engine release
3.4	17.09.0+		2.3	17.06.0+
3.3	17.06.0+		2.2	1.13.0+
3.2	17.04.0+		2.1	1.12.0+
3.1	1.13.1+		2.0	1.10.0+
3.0	1.13.0+		1.0	1.9.1.+

官方地址：
https://docs.docker.com/compose/overview/

3.3.2 快速入门

docker compose 安装

安装依赖工具

```
apt-get install python-pip -y
```

安装编排工具

```
pip install docker-compose
```

查看编排工具版本

```
docker-compose version
```

查看命令帮助

```
docker-compose --help
```

compose简单配置文件

docker-compose.yml 文件内容

```
version: '2'
services:
  web1:
    image: nginx
    ports:
      - "9999:80"
```

```
    container_name: nginx-web1

  web2:
    image: nginx
    ports:
      - "8888:80"
    container_name: nginx-web2
```

运行一个容器

后台启动：

```
docker-compose up -d
```

注意：

如果不加-d，那么界面就会卡在前台

查看运行效果

```
docker-compose ps
```

3.3.3 命令详解

注意：

所有命令尽量都在docker compose项目目录下面进行操作

项目目录：docker-compose.yml所在目录

compose服务启动、关闭、查看

后台启动：

```
docker-compose up -d
```

删除服务

```
docker-compose down
```

查看正在运行的服务

```
docker-compose ps
```

服务开启、关闭、删除

启动一个服务

```
docker-compose start <服务名>
```

注意：

如果后面不加服务名，会停止所有的服务

停止一个服务

```
docker-compose stop <服务名>
```

注意：

如果后面不加服务名，会停止所有的服务

删除服务

```
docker-compose rm
```

注意：

这个docker-compose rm不会删除应用的网络和数据卷。

其他信息查看

查看运行的服务

```
docker-compose ps
```

查看服务运行的日志

```
docker-compose logs -f
```

注意：

加上-f 选项，可以持续跟踪服务产生的日志

查看服务依赖的镜像

```
docke-compose  images
```

进入服务容器

```
docker-compose exec <服务名> <执行命令>
```

查看服务网络

```
docker network ls
```

项目相关

```
build            Build or rebuild services
```

根据提供的dockerfile，自动构建镜像，类似于docker build -t ...

```
create           Create services
pause            Pause services
unpause          Unpause services
restart          Restart services
start            Start services
stop             Stop services
```

注意：

如果后面不加服务名，会停止所有的服务

服务相关

```
up               Create and start containers
down             Stop and remove containers, networks, images, and volumes
ps               List containers
rm               Remove stopped containers
events           Receive real time events from containers
kill             Kill containers
logs             View output from containers
scale            Set number of containers for a service
```

其他命令

```
config           Validate and view the Compose file
help             Get help on a command
images           List images
run              Run a one-off command
top              Display the running processes
version          Show the Docker-Compose version information
bundle           Generate a Docker bundle from the Compose file
port             Print the public port for a port binding
```

第 4 章 Docker拓展

4.1 虚拟化、容器化、云计算

4.1.1 虚拟化基础知识

虚拟化是什么？

就是本来没有，但是通过某种特殊的手段，让你以为有，而且确信不已。

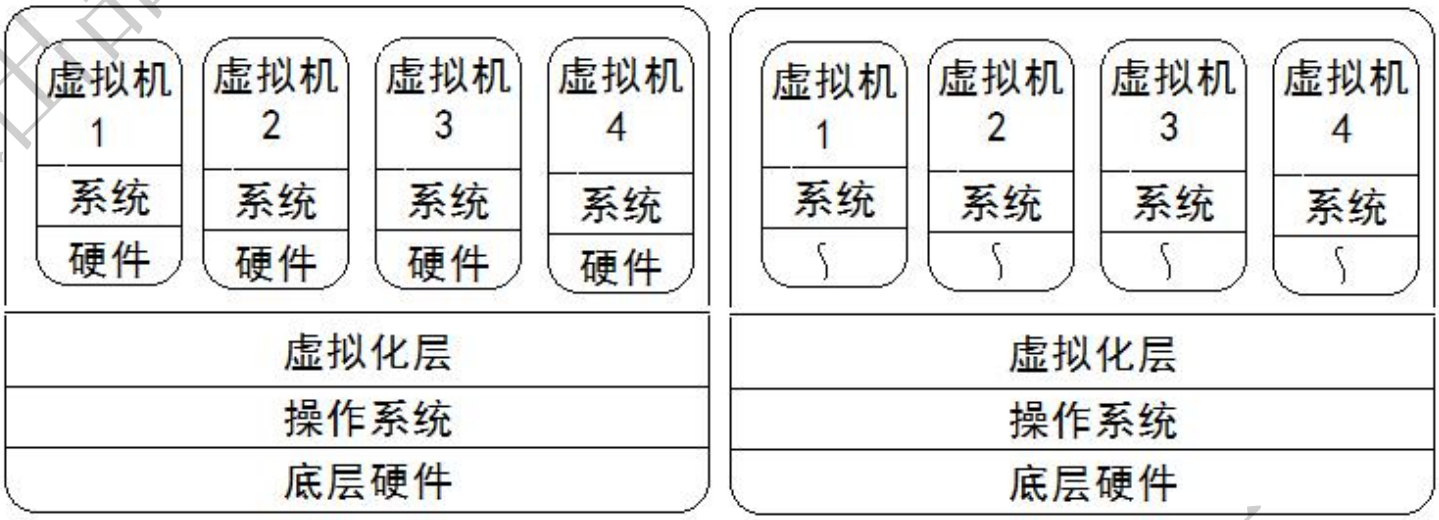
这些手段就是虚拟化技术。

虚拟化的目的：

在时间上和空间上突破我们工作的限制，提升工作效率。

时间上：多种工作在一时间段内同时进行

空间上：在一台物理主机上，虚拟出来多台主机，多台主机共同做一件事情。



常见的虚拟化技术：

全虚拟化：

使用软件方式，虚拟出一个功能完整的主机，虚拟出来的主机自以为是一个真实的主机

常见的技术：kvm

最早出现在1966年

半虚拟化：

结合软件方式，虚拟出一个部分功能的主机，虚拟出来的主机知道自己是一个傀儡。

常见的技术：xen

硬件辅助虚拟化：

借助主机硬件的功能，虚拟出一个完全功能的主机。他是全虚拟化技术的一个特殊表现形式

常见技术：VT-x/EPT或者AMD-v/RVI

最早出现在1972年

三种虚拟化技术的演变：

全虚拟化出来后，因为性能的问题，研究出来了半虚拟化，科技进步太快，硬件虚拟化出来后，性能直接提升。

4.1.2 容器化基础知识

容器是什么：

容器本质上就是一个应用项目的运行状态，特点是实现了某种特殊业务功能，普遍适用于核心业务之外的其他应用。

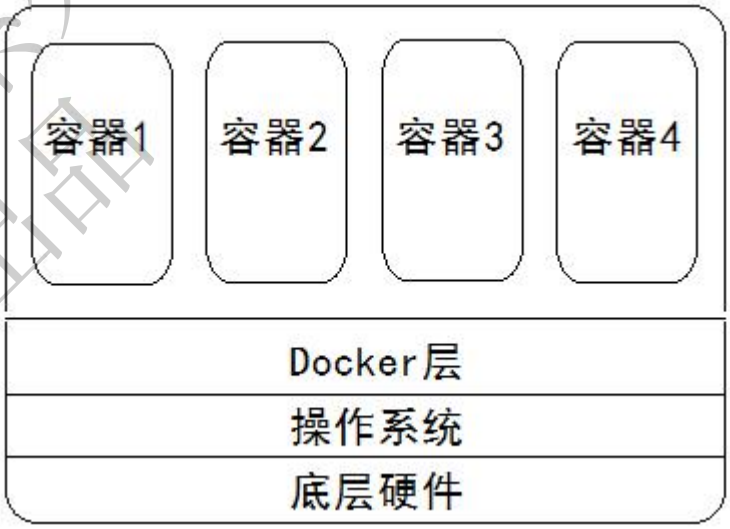
容器化技术，是虚拟化技术的另外一种实现。

容器的特点：

依赖操作系统：借助操作系统实现虚拟功能。

资源利用率高：占用资源少，启动删除自由。

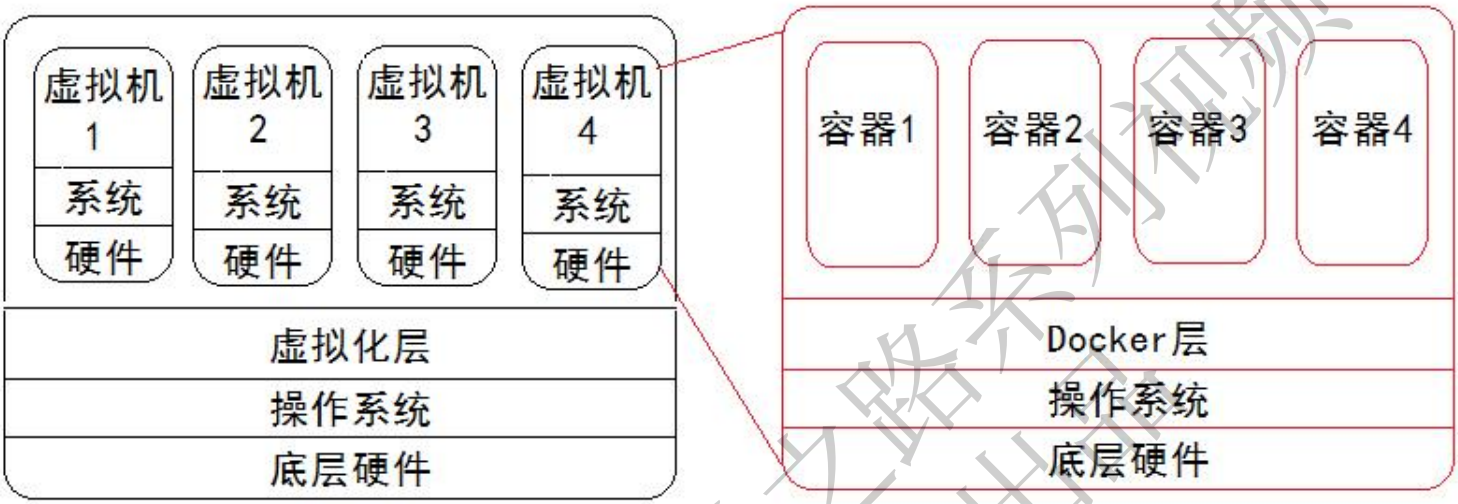
适用 范 围广：所有业务都能使用容器来实现。



常见的容器技术：

LXC、Pouch、Docker、Rocket、LXD

目前比较流行：虚拟化+容器化



4.1.3 云计算基础知识

云计算是什么：

云计算借助于虚拟技术，在物理资源基础上，虚拟出来一个资源池，通过调度资源，大大的提高资源的利用效率。

所以说云计算本质上是一种资源使用和交付的模式。

云计算的特点：

基于网络、按需使用、弹性计费

常见云计算技术：

公有云：

定义：其他公司创造的资源池

获取方式：阿里云、aws、XX云

常见实现技术：openstack

私有云：

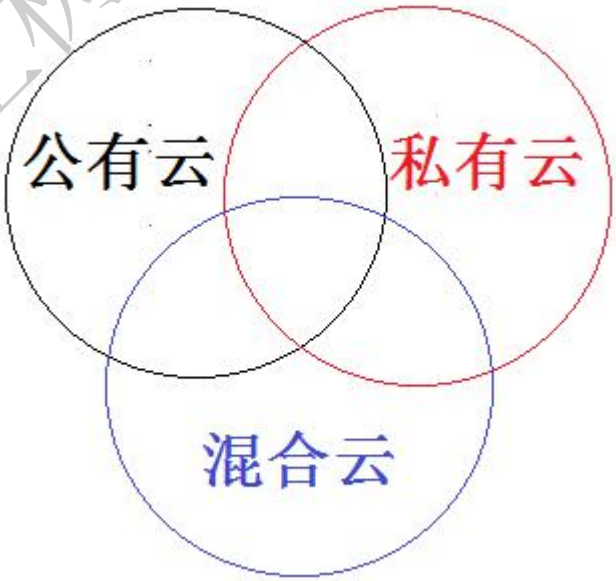
定义：自己公司创造的资源池

获取方式：自己公司技术团队

场景实现方式：cloudstack

混合云：

定义：公有云和私有云的混合使用。



4.1.4 三者区别

虚拟化/容器化 vs 云计算

虚拟化和容器化都是一种技术，而云计算是资源交付的模式，他们是两种不相关的事物

虚拟化 vs 容器化

虚拟化是一种大型的虚拟技术，而容器化是一种轻量级的虚拟技术

在某些特定场景，推荐用虚拟化技术实现：

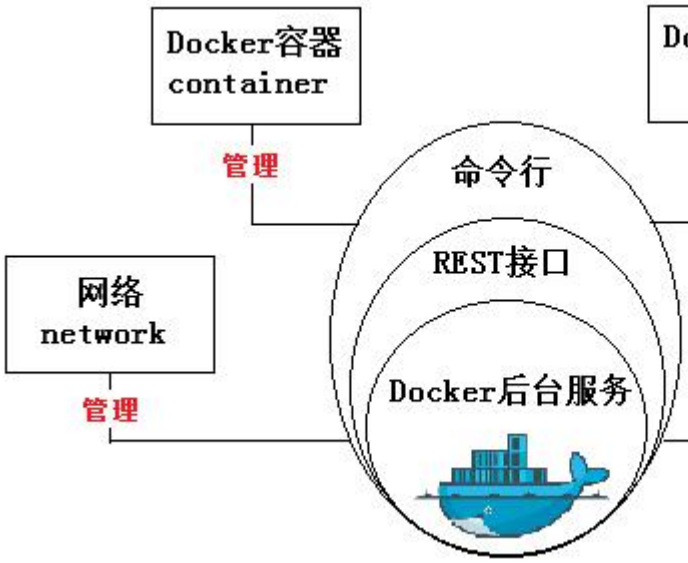
涉及到银行支付功能的U盾场景，只能使用虚拟化技术

投入产出比，不适合的场景，推荐使用虚拟化场景

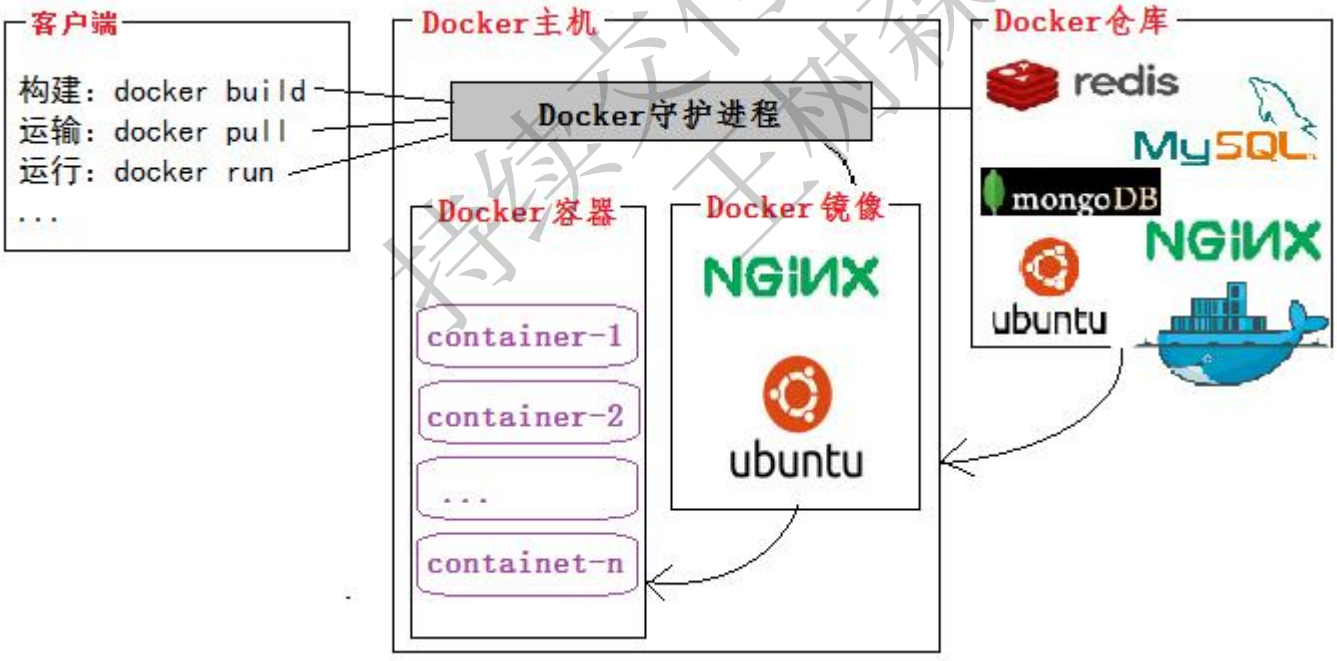
4.2 Docker原理详解

4.2.1 Docker架构

Docker架构图



Docker运行流程图



4.2.2 nameserver & CGroup

Docker是基于LXC的容器技术，Docker主要继承的技术是namespace和CGroup

对于容器的资源隔离，LXC给出的方法是container，本质上它是linux系统的 namespace。Docker通过linux的 pid、net、ipc、mnt、uts、user这六类的namespace将容器的进程、网络、消息、文件系统、UTS和操作系统资源隔离开。

对于容器的资源限制，LXC给出的方法是Cgroup，Cgroup 全称Control group，Docker把它继承过来了。
CGroup其实就是通过创建一个虚拟的文件系统交给容器使用，同时还能对容器的容量做出限制。
通过CGroup可以实现的功能：
资源限制、优先级分配、资源统计、任务控制

4.2.3 镜像 & 容器