

Docker容器

快速入门

基础知识

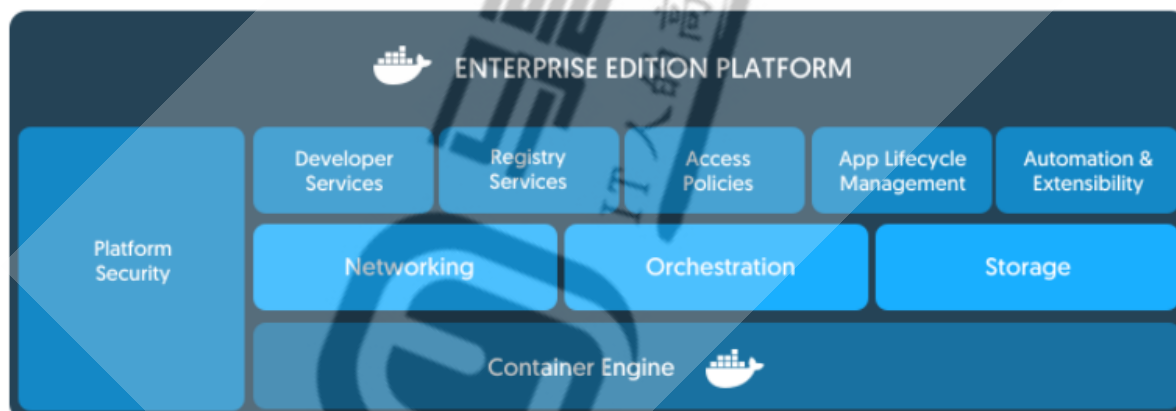
学习目标：了解 docker定位、基本组成等。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

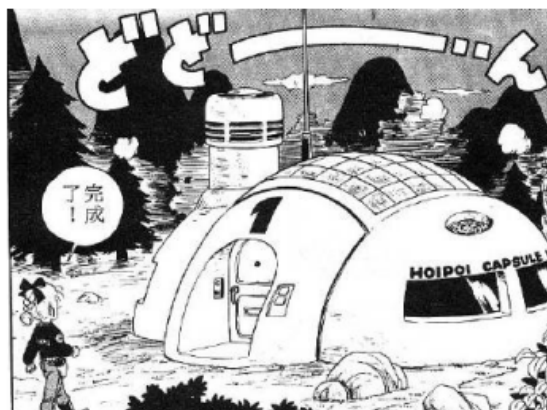
基础知识

docker是什么

Docker is the company driving(推动) the container movement and the only container platform provider to address every application across the hybrid cloud(混合云). Today's businesses are under pressure to digitally transform(数字化转型) but are constrained(限制) by existing applications and infrastructure while rationalizing an increasingly diverse portfolio of clouds, datacenters and application architectures. Docker enables true independence between applications and infrastructure and developers and IT ops to unlock their potential and creates a model for better collaboration and innovation.



生活场景



我的理解：Docker就是一种快速解决业务稳定环境的一种技术手段。

结构组成

角色组成

组件	描述
Docker 镜像	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。
Docker 容器	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。
Docker 仓库	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。

官方资料

Docker 官网: <http://www.docker.com>
Github Docker 源码: <https://github.com/docker/docker>

小结

原理解析

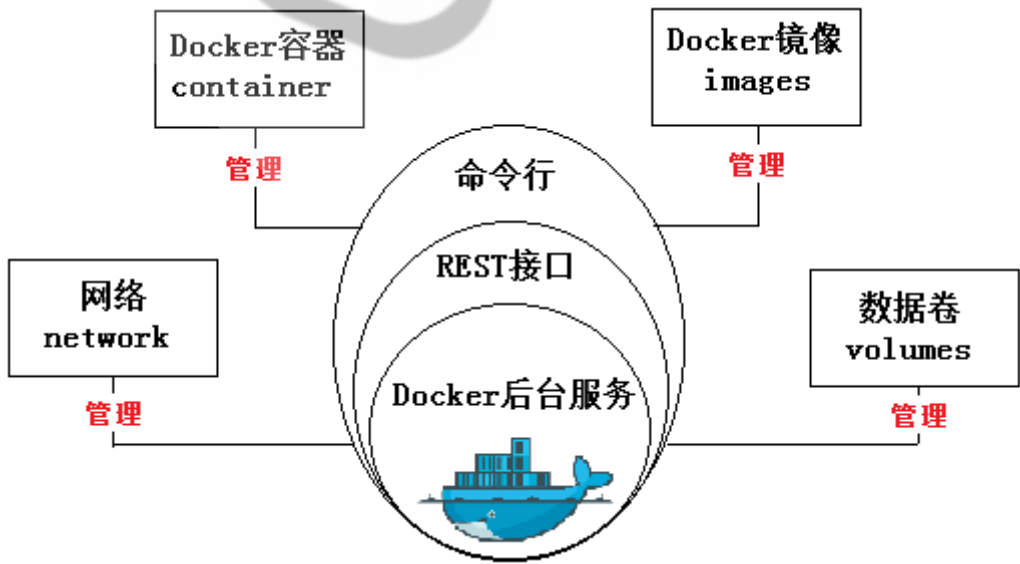
学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

架构图

Docker Engine是一个C/S架构的应用程序，主要包含下面几个组件



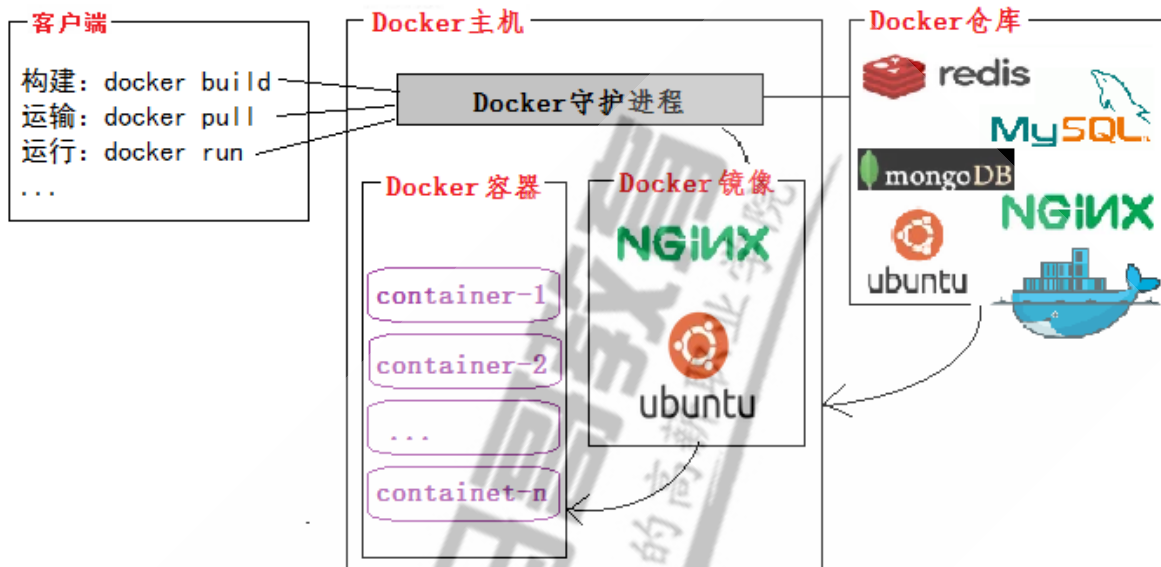
常驻后台进程Dockerd

一个用来和 Dockerd 交互的 REST API Server

命令行CLI接口，通过和 REST API 进行交互（我们经常使用的 docker 命令）

基本流程

Docker 使用 C/S 体系的架构，Docker 客户端与 Docker 守护进程通信，Docker 守护进程负责构建，运行和分发 Docker 容器。Docker 客户端和守护进程可以在同一个系统上运行，也可以将 Docker 客户端连接到远程 Docker 守护进程。Docker 客户端和守护进程使用 REST API 通过UNIX套接字或网络接口进行通信。



简单实践

小结

软件部署

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

源配置

安装依赖软件

```
apt-get update
```

```
apt-get install apt-transport-https ca-certificates curl gnupg lsb-release -y
```

使用官方推荐源

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

```
echo \  
"deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]  
https://download.docker.com/linux/ubuntu \  
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >  
/dev/null
```

```
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1}

```
curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo gpg --  
dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg  
echo \  
"deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]  
https://mirrors.aliyun.com/docker-ce/linux/ubuntu \  
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >  
/dev/null
```

使用清华的源{推荐2}

```
curl -fsSL https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/ubuntu/gpg |  
sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg  
echo \  
"deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]  
https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/ubuntu \  
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >  
/dev/null
```

检查

```
apt-get update
```

查看docker软件版本信息

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

安装最新docker

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

检查服务

```
systemctl [start|stop|restart|status|...] docker  
docker version|info
```

注意

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

检查效果

```
# docker version
```

查看docker 的版本信息

```
Client:
  Version:      20.10.7
  API version:  1.41
  ...

Server:
  Version:      20.10.7
  API version:  1.41 (minimum version 1.12)
  ...
```

```
# docker info
```

查看docker的基本属性

```
...
Docker Root Dir: /var/lib/docker
Debug Mode: false
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false
```

测试效果

```
docker run hello-world
```

```
root@python-auto:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:df5f5184104426b65967e016ff2ac0bfcd44ad7899ca3bbcf8e44e4461491a9e
Status: Downloaded newer image for hello-world:latest

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

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

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

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

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

核心目录

```
软件数据目录  /var/lib/docker/
专属服务目录  /etc/docker
```

加速器实践

需求

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。我们可以使用很多国内的公共源仓库，比如，我们这里采用 **daocloud** 的方法进行加速配置。

访问 daocloud.io 网站，登录 daocloud 账户

登录 DaoCloud 帐号

邮箱/用户名

密码 忘记密码?

验证码

登录

或使用以下帐号登录

Github 微信

点击右上角的 加速器



在新窗口处会显示一条命令

配置 Docker 镜像站

Linux

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://f1361db2.m.daocloud.io
```

该脚本可以将 `--registry-mirror` 加入到你的 Docker 配置文件 `/etc/docker/daemon.json` 中。适用于 Ubuntu14.04、Debian、CentOS6、CentOS7、Fedora、Arch Linux、openSUSE Leap 42.1，其他版本可能有细微不同。更多详情请访问文档。

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://f1361db2.m.daocloud.io
```

注意：

这个地址是每个账号专有的，当然也可以被其他人使用

当然了，阿里云账号也有对应的账号，比如 `"https://kcmn5udq.mirror.aliyuncs.com"`

执行完命令代码后，会自动生成一个文件 `/etc/docker/daemon.json`，内容效果如下

```
root@python-auto:~# curl -sS https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://f1361db2.m.daocloud.io
docker version >= 1.12
{"registry-mirrors": ["http://f1361db2.m.daocloud.io"]}
Success.
You need to restart docker to take effect: sudo systemctl restart docker.service
root@python-auto:~# cat /etc/docker/daemon.json
{"registry-mirrors": ["http://f1361db2.m.daocloud.io"]}
```

但是默认情况下，这个文件少了一个配置属性，我们需要修改一下

Docker 版本在 1.12 或更高

创建或修改 `/etc/docker/daemon.json` 文件，修改为如下形式（请将 **加速地址** 替换为在[加速器](#)页面获取的专属地址）

```
{
  "registry-mirrors": [
    "加速地址"
  ],
  "insecure-registries": []
}
```

```
root@python-auto:~# cat /etc/docker/daemon.json
{"registry-mirrors": ["http://f1361db2.m.daocloud.io"], "insecure-registries": []}
```

重启docker

```
systemctl restart docker
```

再次查看效果

```
root@python-auto:~# docker info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Build with BuildKit (Docker Inc., v0.5.1-docker)
  scan: Docker Scan (Docker Inc., v0.8.0)

Server:
 Containers: 1
  107: 0000000000000000000000000000000000000000000000000000000000000000
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Registry Mirrors:
  http://f1361db2.m.daocloud.io/
 Live Restore Enabled: false
```

远程连接

需求

有些时候，我们需要 远程跨主机执行 **docker** 相关的操作，对于 **docker** 软件来说，它提供了这种远程登录的命令，主要是通过两种方式：

socket方式 `unix:///var/run/docker.sock`

tcp方式 tcp://0.0.0.0:2375

简单实践

修改启动文件

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

定制启动配置

```
# cat /etc/docker/daemon.json
{..., "insecure-registries": ["192.168.8.12:5000"], "hosts": [
    "unix:///var/run/docker.sock",
    "tcp://0.0.0.0:12306"
]}
```

加载配置

```
systemctl daemon-reload
```

重启服务

```
systemctl start docker.service
```

检查效果

```
docker info
netstat -tnulp | grep docker
docker -H tcp://192.168.8.12:12306 images
```

小结

镜像管理

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

镜像

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。

镜像相关命令


```
root@python-auto:~# docker help | grep image
image      Manage images
manifest   Manage Docker image manifests and manifest lists
trust       Manage trust on Docker images
build       Build an image from a Dockerfile
commit      Create a new image from a container's changes
history     Show the history of an image
images      List images
import      Import the contents from a tarball to create a filesystem image
load        Load an image from a tar archive or STDIN
pull        Pull an image or a repository from a registry
push        Push an image or a repository to a registry
rmi         Remove one or more images
save        Save one or more images to a tar archive (streamed to STDOUT by default)
search      Search the Docker Hub for images
```

简单实践

搜索

```
docker search [image_name]
```

获取

```
docker pull [image_name]
```

查看

```
docker images <image_name>
```

历史

```
docker history [image_name]
```

标签

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

删除

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

导出

```
docker save -o [包文件] [镜像]
```

导入

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

小结

容器管理

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

容器

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

容器相关命令

```
root@python-auto:~# docker help | grep container
A self-sufficient runtime for containers
  container  Manage containers
  attach     Attach local standard input, output, and error streams to a running container
  commit     Create a new image from a container's changes
  cp         Copy files/folders between a container and the local filesystem
  create     Create a new container
  diff       Inspect changes to files or directories on a container's filesystem
  exec       Run a command in a running container
  export     Export a container's filesystem as a tar archive
  kill       Kill one or more running containers
  logs       Fetch the logs of a container
  pause      Pause all processes within one or more containers
  port       List port mappings or a specific mapping for the container
  ps         List containers
  rename     Rename a container
  restart    Restart one or more containers
  rm         Remove one or more containers
  run        Run a command in a new container
  start      Start one or more stopped containers
  stats      Display a live stream of container(s) resource usage statistics
  stop       Stop one or more running containers
  top        Display the running processes of a container
  unpause    Unpause all processes within one or more containers
  update     Update configuration of one or more containers
  wait       Block until one or more containers stop, then print their exit codes
```

简单实践

查看

查看所有：

```
docker ps -a
```

查看部分：

```
docker ps --filter status=exited -q
```

启动

```
docker run -itd <image_name>
```

常见参数:

-e, --env list	Set environment variables
--env-file list	Read in a file of environment variables
-i, --interactive	Keep STDIN open even if not attached
-t, --tty	Allocate a pseudo-TTY
-d, --detach	Run container in background and print
container ID	
--name string	Assign a name to the container
-h, --hostname string	Container host name
--rm	Automatically remove the container when it
exits	
--restart string	Restart policy to apply when a container
exits (default "no")	

启动

```
docker start [container_id]
```

关闭

```
docker stop [container_id]
```

删除

```
docker rm [-f] [container_id]
```

技巧:

```
docker rm -f $(docker ps -a -q)
```

进入

```
docker exec -it [container_id] /bin/bash
```

属性解析

-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`

提交

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

日志

```
docker logs [container_id]
```

属性

查看所有属性：

```
docker inspect [container_id]
```

格式化输出属性：

专用格式：`docker inspect -f '{{json .NetworkSettings.Ports}}'`

定制格式：`docker inspect -f 'Hello from container {{.Name}}' [container_id]`

高阶格式信息输出：

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' [container_id]
```

属性解析：

`{{}}` 表示模板指令，里面存放属性的名称

`"."` 表示“当前上下文，表示容器元数据的整个数据结构，两个属性间的 `.` 代表上下级

`range` 用于遍历结构内返回值的所有数据

小结

数据管理

学习目标：

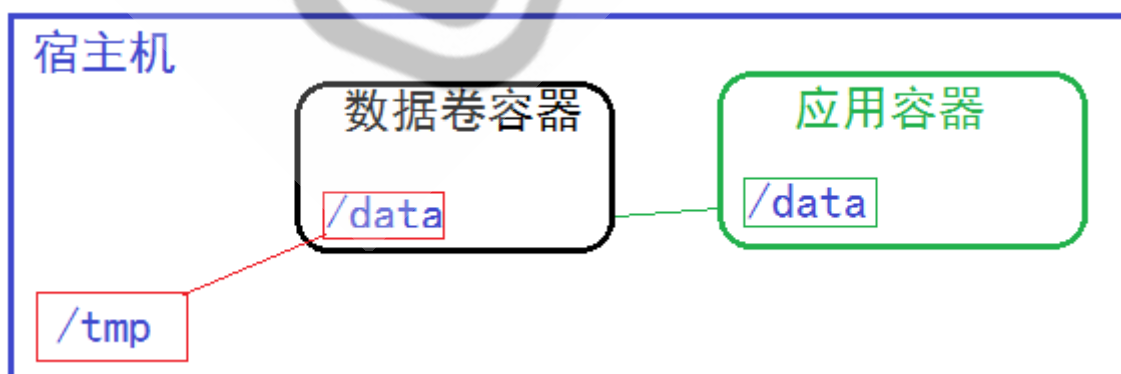
这一节，我们从 基础知识、数据卷实践、数据卷容器实践、小结 三个方面来学习。

基础知识

简介

`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



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

数据卷容器

将宿主机的某个目录，使用容器的方式来表示，然后其他的应用容器将数据保存在这个容器中，达到大量应用数据同时存储的目的

命令解析

```
root@python-auto:~# docker run --help | grep volume
-v, --volume list          Bind mount a volume
--volume-driver string      Optional volume driver for the container
--volumes-from list        Mount volumes from the specified container(s)
```

数据卷操作

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

注意：

- v 宿主机文件:容器文件 可以存在多个，表示同时挂载多个
- 宿主机文件尽量用绝对路径，容器文件即使不存在，Docker自动创建

数据卷容器操作

1 创建数据卷容器

```
docker create -v [宿主机文件]:[容器数据卷目录] --name [容器名字] [镜像名称] [命令(可选)]
```

2 使用数据卷容器

```
docker run --volumes-from [数据卷容器名字] -d --name [容器名字] [镜像名称] [命令(可选)]
```

数据卷实践

目录文件

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
```

普通文件

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
```

数据卷容器实践

创建数据卷

```
docker create -v /data --name v-test nginx
```

挂载数据卷

创建 vc-test1 容器

```
docker run --volumes-from 4693558c49e8 -d --name vc-test1 nginx /bin/bash
```

创建 vc-test2 容器

```
docker run --volumes-from 4693558c49e8 -d --name vc-test2 nginx /bin/bash
```

确认效果

进入vc-test1，操作数据卷容器

```
~# docker exec -it vc-test1 /bin/bash
root@c408f4f14786:/# ls /data/
root@c408f4f14786:/# echo 'v-test1' > /data/v-test1.txt
root@c408f4f14786:/# exit
```

进入vc-test2，确认数据卷

```
~# docker exec -it vc-test2 /bin/bash
root@7448eee82ab0:/# ls /data/
v-test1.txt
root@7448eee82ab0:/# echo 'v-test2' > /data/v-test2.txt
root@7448eee82ab0:/# exit
```

回到vc-test1进行验证

```
~# docker exec -it vc-test1 /bin/bash
root@c408f4f14786:/# ls /data/
v-test1.txt  v-test2.txt
root@c408f4f14786:/# cat /data/v-test2.txt
v-test2
```

回到宿主机查看/data/目录

```
~# ls /data/
~#
```

小结

备份还原

学习目标：

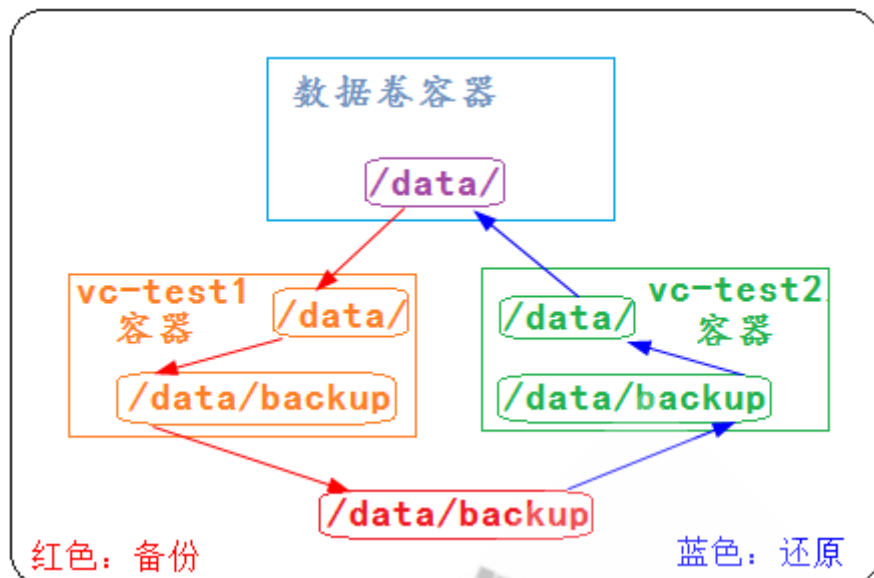
这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

需求

基于数据卷和数据卷容器知识，实现容器应用内部的数据备份还原实践

思路



方案解析

数据备份方案：

- 1 创建一个挂载数据卷容器的容器
- 2 挂载宿主机本地目录作为备份数据卷
- 3 将数据卷容器的内容备份到宿主机本地目录挂载的数据卷中
- 4 完成备份操作后销毁刚创建的容器以及关联的容器卷

命令格式：

```
docker run --rm --volumes-from [数据卷容器id/name] -v [宿主机目录]:[容器目录]  
[镜像名称] [备份命令]
```

数据恢复方案

- 1、创建一个新的数据卷容器
- 2、创建一个新容器，挂载数据卷容器，同时挂载本地的备份目录作为数据卷
- 3、将要恢复的数据解压到容器中
- 4 完成还原操作后销毁刚创建的容器以及关联的容器卷

命令格式：

```
docker run --rm -itd --volumes-from [数据要恢复的容器] -v [宿主机备份目录]:[容器备份目录] [镜像名称] [解压命令]
```

简单实践

备份实践

创建备份目录

```
mkdir /backup/
```

创建备份的容器

```
docker run --rm --volumes-from 4693558c49e8 -v /backup:/backup nginx tar zcf /backup/data.tar.gz /data
```

验证操作

```
ls /backup
```

```
zcat /backup/data.tar.gz
```

注意:

解压的时候, 如果使用目录的话, 一定要在解压的时候使用 `-C` 制定挂载的数据卷容器, 不然的话容器数据是无法恢复的, 因为容器中默认的`backup`目录不是数据卷, 即使解压后, 也看不到文件。

还原实践

删除源容器内容

```
~# docker exec -it vc-test1 bash
root@c408f4f14786:/# rm -rf /data/*
```

恢复数据

```
docker run --rm --volumes-from 4693558c49e8 -v /backup:/backup nginx tar xf /backup/data.tar.gz -C /
```

注意:

解压的时候, 会自动携带压缩前的`data`目录名称

验证

```
~# docker exec -it vc-test1 bash
root@c408f4f14786:/# ls /data/data/
v-test1.txt v-test2.txt
```

小结

网络管理

学习目标:

这一节, 我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

Docker 网络实现方式有两种:

端口映射

- 将容器内服务端口与宿主机端口关联在一起, 通过"宿主机ip:宿主机port"达到访问容器服务效果

网络模式

- 借助于独立的docker网卡功能实现访问容器服务的效果

命令格式：

```
docker -P|p [镜像名称]
```

注意：

-P(大写) 指的是容器应用PORT随机映射到宿主机上的PORT

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

生产场景一般不使用随机映射，好处是由docker分配，宿主机端口不会冲突，

-p(小写) 宿主机IP:宿主机PORT:容器PORT

宿主机IP不写表示"0.0.0.0",宿主机PORT不写表示随机端口，容器PORT必须指定
可以同时多个端口进行映射绑定

指定端口映射，在标准化场景下使用频率高，

简单实践

默认随机映射

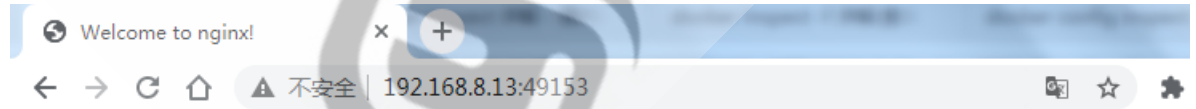
```
docker run -d -P nginx
```

结果显示：

多次执行后，宿主机的端口会自动出现

```
root@python-auto:~# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
acc27a9299a5	nginx	"/docker-entrypoint..."	7 seconds ago	Up 6 seconds	0.0.0.0:49156->80/tcp
p, :::49156->80/tcp		infallible_einstein			
5e0c269f4629	nginx	"/docker-entrypoint..."	8 seconds ago	Up 8 seconds	0.0.0.0:49155->80/tcp
p, :::49155->80/tcp		brave_chaum			
1f5d4de46e44	nginx	"/docker-entrypoint..."	10 seconds ago	Up 9 seconds	0.0.0.0:49154->80/tcp
p, :::49154->80/tcp		focused_wing			
cc2745aafb7a	nginx	"/docker-entrypoint..."	12 seconds ago	Up 11 seconds	0.0.0.0:49153->80/tcp
p, :::49153->80/tcp		busy_johnson			



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.

指定端口映射

```
docker run -d -p 192.168.8.13:777:80 --name nginx-1 nginx
docker run -d -p 192.168.8.13::80 --name nginx-2 nginx
docker run -d -p :666:80 --name nginx-3 nginx
docker run -d -p ::80 --name nginx-4 nginx
```

```
root@python-auto:~# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
a38ffd7682fb	nginx	"/docker-entrypoint..."	6 seconds ago	Up 5 seconds	0.0.0.0:49154->80/tcp,
:::49153->80/tcp	nginx-4				
d616ddfd8f74	nginx	"/docker-entrypoint..."	8 seconds ago	Up 7 seconds	0.0.0.0:666->80/tcp, :
:::666->80/tcp	nginx-3				
04cbd787bfb	nginx	"/docker-entrypoint..."	9 seconds ago	Up 7 seconds	192.168.8.13:49153->80
/tcp	nginx-2				
ecff0418dab7	nginx	"/docker-entrypoint..."	9 seconds ago	Up 8 seconds	192.168.8.13:777->80/t
cp	nginx-1				

多端口映射

```
docker run -d -p 520:443 -p 6666:80 --name nginx-3 nginx
```

```
root@python-auto:~# docker run -d -p 520:443 -p 6666:80 --name nginx-3 nginx
845c58fc4825d152aea0091610e36d030b9a1b7227262314ae862d64ad6c803b
root@python-auto:~# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
845c58fc4825	nginx	"/docker-entrypoint..."	3 seconds ago	Up 2 seconds	0.0.0.0:6666->80/tcp,
:::6666->80/tcp, 0.0.0.0:520->443/tcp, :::520->443/tcp	nginx-3				

小结

网络模型

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

基本命令

```
root@python-auto:~# docker network help
```

Usage: docker network COMMAND

Manage networks

Commands:

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

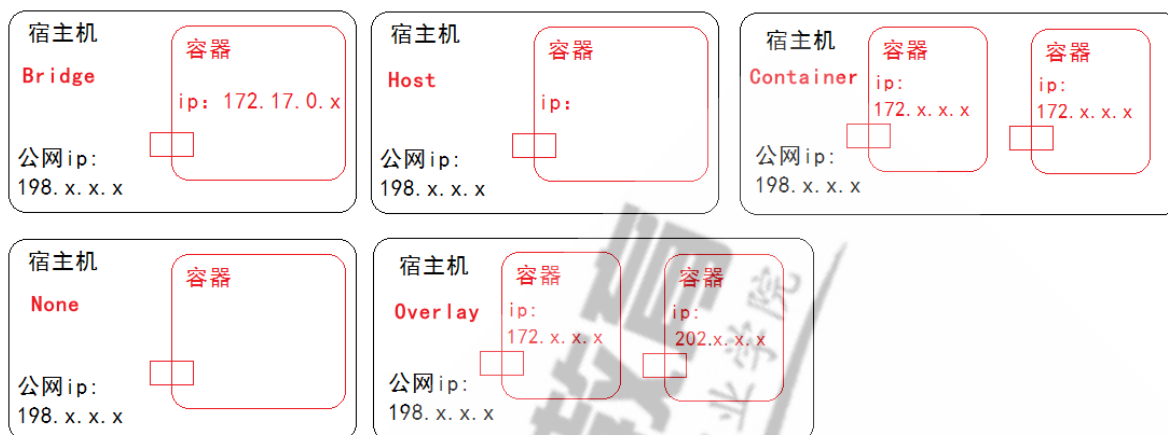
默认网络

```
root@python-auto:~# docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
1f2823967022	bridge	bridge	local
20a347d752f6	host	host	local
7648b673e9d8	none	null	local

五种模型

模型名称	特点	备注
bridge	默认模式，地址转换	网络效率低
host	使用宿主机的ip地址	效率高
container	容器和容器共享网络	局域网
none	无任何网络配置	最干净、最复杂
overlay	容器间跨网段通信	第三方工具



简单实践

命令语法

创建网络:

```
docker network create --driver [网络类型] [网络名称]
```

使用网络:

```
docker run --net=[网络名称] -itd --name=[容器名称] [镜像名称]
```

端口网络

```
docker network disconnect [网络名] [容器名]
```

连接网络

```
docker network connect [网络名] [容器名]
```

创建实践

命令演示:

```
docker network create --driver bridge bridge-test
```

查看主机网络类型

```
~# docker network ls
```

```
NETWORK ID          NAME                DRIVER              SCOPE
8a18574f0f27        bridge              bridge              local
172ae1f1f3f5        bridge-test         bridge              local
...
```

查看新建网络的网络信息

```
~# docker network inspect bridge-test
```

```
[
  {
    "Name": "bridge-test",
    ...
    "Config": [
```

```
{
  "Subnet": "172.18.0.0/16",
  "Gateway": "172.18.0.1"
...
}
```

宿主机又多出来一个网卡设备

```
~# ifconfig
br-172ae1f1f3f5 Link encap:Ethernet  HWaddr 02:42:18:4e:ac:92
    inet addr:172.18.0.1  Bcast:172.18.255.255  Mask:255.255.0.0
    ...
```

应用实践

容器启动时候使用网络:

```
docker run --net=bridge-test -itd --name=nginx-new-bri nginx
```

查看该容器的ip信息

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}
{{end}}' nginx-new-bri
```

断开实践

命令演示:

```
docker network disconnect bridge-test nginx-new-bri
```

效果展示:

```
docker network inspect bridge-test
```

连接实践

命令演示:

```
docker network connect bridge-test nginx-new-bri
```

效果展示:

```
docker network inspect bridge-test
```

小结

仓库管理

学习目标:

这一节, 我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

仓库

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。

分类

公有仓库: Docker hub、Docker cloud、等
私有仓库: registry、harbor等
本地仓库: 在当前主机存储镜像的地方。

相关命令

```
root@python-auto:~# docker help | grep -E 'rep|reg'
login      Log in to a Docker registry
logout     Log out from a Docker registry
pull       Pull an image or a repository from a registry
push       Push an image or a repository to a registry
```

简单实践

基本流程

- 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

仓库配置

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": ["192.168.8.13:5000"]}
```

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
```

查看效果

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
```

小结

Dockerfile

基础知识

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

镜像创建

- | | |
|-----------|-----------------------|
| 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 |

问题：

慢，繁琐、手工易出错

Dockerfile简介

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。

注意：

Dockerfile 在使用的时候，首字母必须大写。

主要内容

- | | |
|-----------|------|
| 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！！ |

命令语法

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，默认是以构建后的docker image的id号为镜像名称
/opt/dockerfile/nginx/ 代表Dockerfile存放位置，如果是当前目录，则用 .(点)表示

简单实践

准备工作

```
创建Dockerfile专用目录  
mkdir /data/docker/base/  
cd /data/docker/base/
```

准备文件

```
cp /home/python/.config/pip/pip.conf ./  
~# ls /data/docker/base/  
Dockerfile pip.conf
```

定制Dockerfile

```
# 构建一个基于python3的定制镜像  
# 基础镜像  
FROM ubuntu  
  
# 作者信息  
MAINTAINER sswang@163.com  
  
# 执行操作  
RUN apt-get update  
RUN apt-get install python3 -y && apt-get install python3-pip -y  
WORKDIR /root/.pip/  
ADD ./pip.conf ./pip.conf  
WORKDIR /  
  
# 入口指令  
ENTRYPOINT ["/bin/bash"]
```

构建镜像

构建镜像

```
docker build -t python3:v0.1 .
```

使用新镜像启动一个容器，查看效果

```
docker run -it --name python python3:v0.1
```

容器检查

```
docker ps
```

小结

构建指令

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础指令

FROM

语法：

```
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

MAINTAINER

语法：

```
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

语法：

```
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

执行模式：

模式	格式	示例
shell模式	类似于 <code>/bin/bash -c command</code>	<code>RUN echo hello</code>
exec 模式	类似于 <code>RUN ["/bin/bash", "-c", "command"]</code>	<code>RUN ["echo", "hello"]</code>

EXPOSE

语法:

```
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。

ENTRYPOINT

语法:

```
ENTRYPOINT ["executable", "param1", "param2"] (exec 模式)
ENTRYPOINT command param1 param2 (shell模式)
```

解释:

每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个起效。

其他指令

ADD

语法:

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

解释:

将指定的 文件复制到容器文件系统中的

src 指的是宿主机，dest 指的是容器

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中(无论目标是文件还是目录，都会当成目录处理)。

COPY

语法:

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

解释:

单纯复制文件场景，Docker推荐使用COPY

VOLUME

语法:

```
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。

ENV

语法:

```
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

语法:

```
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

小结

构建解析

学习目标：

这一节，我们从 基础知识、小结 三个方面来学习。

基础知识

构建过程

- 1) 从基础镜像1创建一个容器A
- 2) 遇到一条Dockerfile指令，都对容器A做一次修改操作
- 3) 执行完一条指令，提交生成一个新镜像2
- 4) 再基于新的镜像2运行一个容器B
- 5) 遇到一条Dockerfile指令，都对容器B做一次修改操作
- 6) 执行完一条指令，提交生成一个新镜像3
- ...

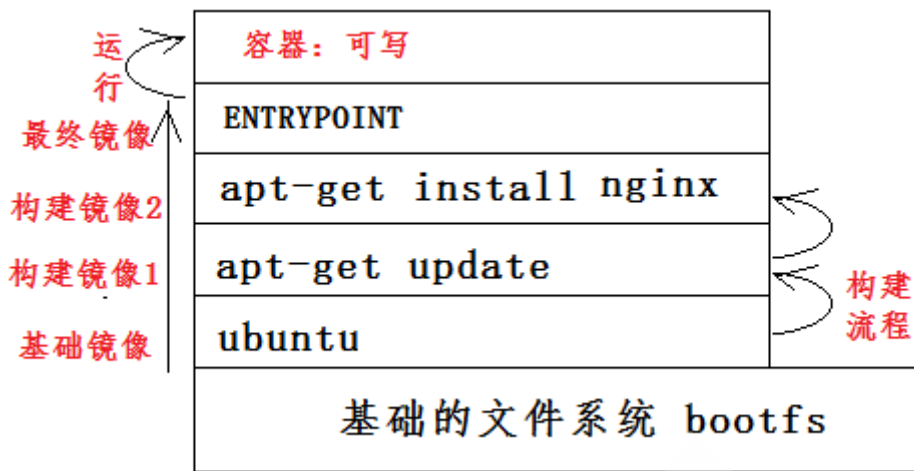
构建历史

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

我们可以通过docker history <镜像名> 来查看整个构建过程所产生的镜像

镜像原理



对于Docker镜像文件整体来说，它是一个只读的文件，但是根据我们对构建过程的理解，我们发现，对于Docker镜像还有很多更深层的东西：

1 镜像文件是基于分层机制实现的

- 最底层是bootfs用于启动容器之前系统引导的文件系统，容器启动完毕后，卸载该部分内容以便节省资源

- 其他层是rootfs，有内核挂载为只读模式，而后通过"联合挂载"在其基础上挂载一个"可写层"

2 下层镜像是上层镜像的父镜像，最底层的称为基础镜像

- 最上层的是可写的，其他各层都是只读的。

拓展：执行的步骤越多越好呢？还是越少越好？

构建缓存

第一次构建很慢，之后的构建都会很快，因为它们用到了构建的镜像缓存。

不使用构建缓存方法：

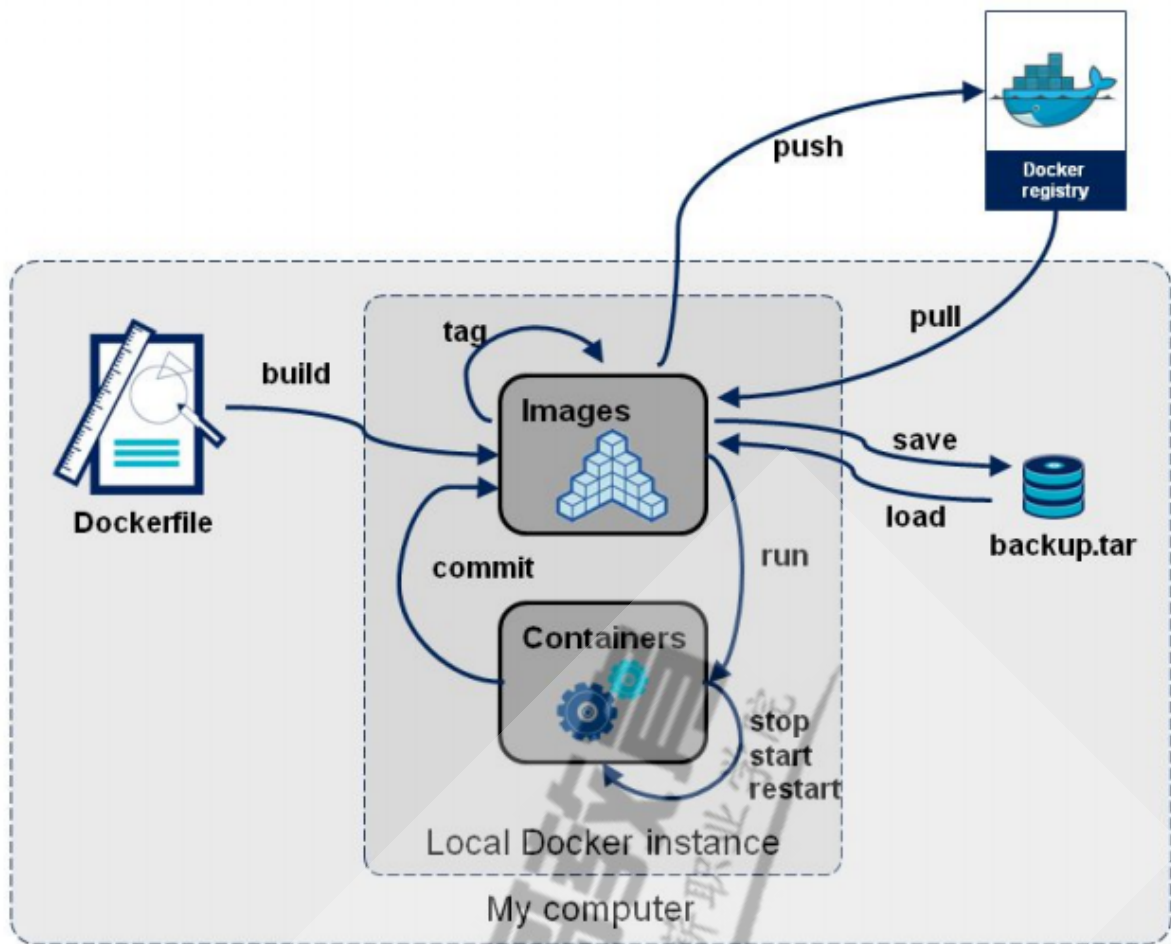
全部不用缓存：

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

部分不用缓存：

更改Dockerfile的部分代码即可

镜像的完整流程



使用原则

简介

在工作中，我们经常会因为业务需求，而定制各种各样的Docker镜像，由于Dockerfile的便捷性，所以我们经常会基于Dockerfile来创建我们业务场景中所需要的各种镜像。

根据我自己的工作经验，我们在使用Dockerfile的过程中，一般只需要关注三个方面即可：

- 1 Dockerfile在使用的过程中，构建的指令越少越好，能合并的就合并。
- 2 基于Docker镜像的分层特性，我们最好按照项目的架构级别来定制不同层的镜像
- 3 Dockerfile构建的过程中，功能越简单越好，最好只有一个

代码合并

原来效果

```

COPY ./setuptools_scm-3.3.3-py2.py3-none-any.whl /data/softs/
RUN apt-get update -y
RUN apt-get install nginx -y --allow-unauthenticated
RUN mkdir /data/meiduo -p && cd /data/meiduo
RUN apt-get install -y libjpeg-dev libfreetype6-dev libssl-dev --allow-unauthenticated
RUN pip3 install /data/softs/setuptools_scm-3.3.3-py2.py3-none-any.whl
RUN pip install -r /data/softs/requirements.txt
RUN pip3 uninstall haystack -y
RUN rm -rf /usr/local/lib/python3.7/site-packages/haystack/
RUN pip3 uninstall django-haystack -y
RUN pip3 install django-haystack==2.8.1
RUN apt-get clean
RUN apt-get autoclean

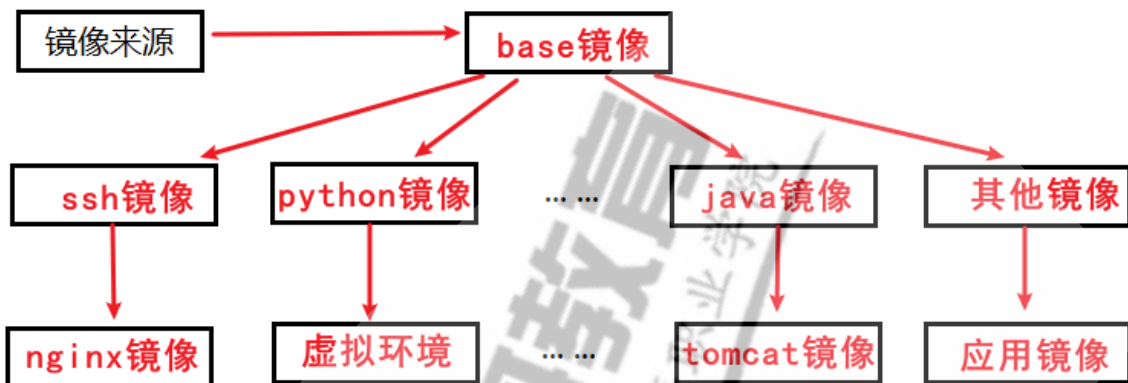
```

```
RUN rm -rf /var/lib/apt/lists/*
RUN rm -rf ~/.cache/pip/
```

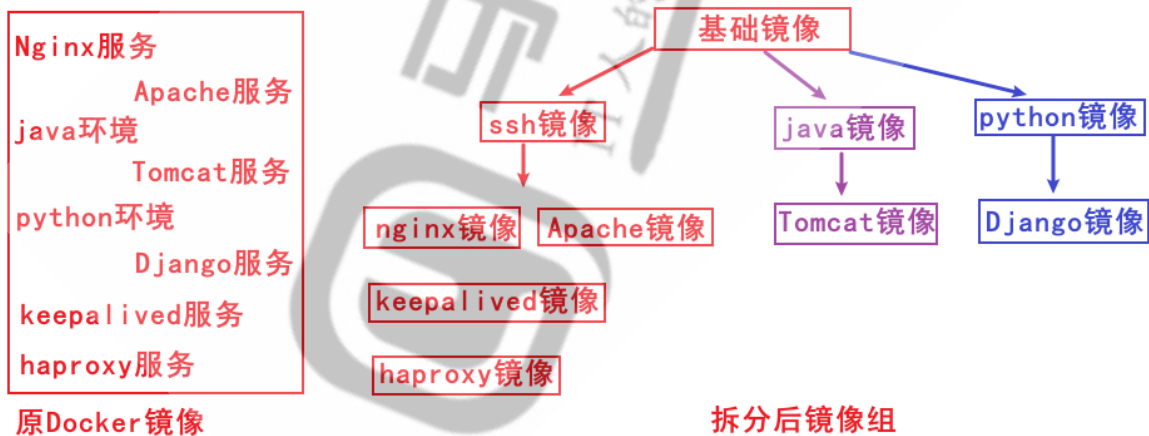
改造效果

```
RUN apt-get update -y && apt-get install nginx libjpeg-dev libfontconfig-dev
libssl-dev -y --allow-unauthenticated && pip3 install
/data/softs/setuptools_scm-3.3.3-py2.py3-none-any.whl && pip3 install -r
/data/softs/requirements.txt && pip3 uninstall haystack -y && rm -rf
/usr/local/lib/python3.7/site-packages/haystack/ && pip3 uninstall django-
haystack -y && pip3 install django-haystack==2.8.1 && apt-get clean && apt-get
autoclean && rm -rf /var/lib/apt/lists/* && rm -rf ~/.cache/pip/
```

分层效果



功能效果



小结

基础镜像构建

学习目标：

这一节，我们从 ssh镜像、java镜像、小结 三个方面来学习。

ssh镜像

需求

我们定制一个标准的ssh基础镜像，用于我们远程连接或者进行其他的应用镜像的基础镜像

简单实践

创建Dockerfile专用目录

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

创建密钥认证

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub > authorized_keys
```

定制Dockerfile

构建一个基于ubuntu的ssh定制镜像

基础镜像

FROM ubuntu

镜像作者

MAINTAINER President.Wang 000000@qq.com

安装 ssh 服务

```
RUN apt-get update && apt-get install -y openssh-server curl vim net-tools &&
mkdir -p /var/run/sshd && mkdir -p /root/.ssh && sed -i
"s/*.pam_loginuid.so/#&/" /etc/pam.d/sshd && apt-get autoclean && apt-get clean
&& apt-get autoremove
```

复制配置文件到相应位置,并赋予脚本可执行权限

```
ADD authorized_keys /root/.ssh/authorized_keys
```

对外端口

EXPOSE 22

启动ssh

```
CMD ["/usr/sbin/sshd","-D"]
```

测试效果

构建镜像

```
docker build -t ubuntu-ssh .
```

使用新镜像启动一个容器，查看效果

```
docker run -d -p 10086:22 ubuntu-ssh
```

容器检查

```
docker ps
```

```
docker port c03d146b64d4
```

ssh查看效果

```
ssh 192.168.8.12 -p 10086
```

java镜像

需求

定制标准的 java环境，以便于基于java环境的 业务环境使用

简单实践

创建Dockerfile专用目录

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

定制Dockerfile

构建一个基于ubuntu-ssh定制java镜像

基础镜像

FROM ubuntu-ssh

镜像作者

MAINTAINER President.wang 000000@qq.com

添加文件到容器

ADD jdk-8u121-linux-x64.tar.gz /data/server/

RUN ln -s /data/server/jdk1.8.0_121 /data/server/java

定制环境变量

ENV JAVA_HOME /data/server/java

ENV JRE_HOME \$JAVA_HOME/jre

ENV CLASSPATH \$JAVA_HOME/lib/:\$JRE_HOME/lib/

ENV PATH \$PATH:\$JAVA_HOME/bin

测试效果

构建镜像

```
docker build -t ubuntu-jdk:8u121 .
```

使用新镜像启动一个容器，查看效果

```
docker run -it --rm ubuntu-jdk:8u121 bash
```

容器检查

```
docker exec -it 04b5e238cf68 /bin/bash
```

```
java -version
```

```
echo $JAVA_HOME
```

编译镜像

需求

定制标准的 可编译 环境，以便于后续其他软件的编译安装使用

简单实践

创建Dockerfile专用目录

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

下载lua软件

```
wget http://www.lua.org/ftp/lua-5.4.3.tar.gz
```

定制Dockerfile

构建一个基于ubuntu定制编辑环境镜像

```
# 基础镜像
FROM ubuntu
# 镜像作者
MAINTAINER President.Wang 000000@qq.com

# 添加文件到容器
ADD lua-5.4.3.tar.gz /usr/local/src/
RUN apt-get update && apt install make gcc build-essential libssl-dev zlib1g-dev
libpcre3 libpcre3-dev libsystemd-dev libreadline-dev -y && cd
/usr/local/src/lua-5.4.3/src/ && make linux && apt-get autoclean && apt-get
clean && apt-get autoremove

# 启动ssh
CMD ["/bin/bash"]
```

测试效果

```
构建镜像
docker build -t ubuntu-make:v0.1 .

使用新镜像启动一个容器，查看效果
docker run --rm -it ubuntu-make:v0.1 bash

容器检查
cd /usr/local/src/lua-5.4.3/src
./lua -v
```

小结

应用镜像构建

学习目标：

这一节，我们从 tomcat 镜像、应用镜像、小结 三个方面来学习。

tomcat 镜像

需求

基于 java 环境镜像定制 tomcat 镜像

简单实践

```
创建Dockerfile专用目录
mkdir /docker/images/tomcat -p
cd /docker/images/tomcat

获取应用文件
wget https://mirrors.tuna.tsinghua.edu.cn/apache/tomcat/tomcat-
10/v10.0.10/bin/apache-tomcat-10.0.10.tar.gz

定制Dockerfile
```

```
# 构建一个基于ubuntu-jdk定制tomcat镜像
# 基础镜像
FROM ubuntu-jdk:8u121
# 镜像作者
MAINTAINER President.wang 000000@qq.com

# 定制环境变量
ENV TZ "Asia/Shanghai"
ENV LANG en_US.UTF-8
ENV TERM xterm
ENV TOMCAT_MAJOR_VERSION 10
ENV TOMCAT_MINOR_VERSION 10.0.10
ENV CATALINA_HOME /apps/tomcat
ENV APP_DIR ${CATALINA_HOME}/webapps

# 添加文件到容器
ADD apache-tomcat-10.0.10.tar.gz /apps
RUN ln -s /apps/apache-tomcat-10.0.10 /apps/tomcat

# 定制容器的启动命令
CMD ["/bin/bash"]
```

测试效果

构建镜像

```
docker build -t ubuntu-tomcat:v10.0.10 .
```

使用新镜像启动一个容器，查看效果

```
docker run -it --rm -p 8080:8080 ubuntu-tomcat:v10.0.10 bash
```

注意：

因为在构建镜像的时候，已经启动了命令bash，所以可以直接进入进去

容器检查

```
/apps/tomcat/bin/catalina.sh start
netstat -tnulp
```

```
root@640d46e7ea18:/# /data/server/tomcat/bin/catalina.sh start
Using CATALINA_BASE:   /data/server/tomcat
Using CATALINA_HOME:   /data/server/tomcat
Using CATALINA_TMPDIR: /data/server/tomcat/temp
Using JRE_HOME:        /usr/local/jdk/jre
Using CLASSPATH:        /data/server/tomcat/bin/bootstrap.jar:/data/server/tomcat/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
root@640d46e7ea18:/# netstat -tnulp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:8080             0.0.0.0:*               LISTEN      15/java
tcp        0      0 127.0.0.1:8005           0.0.0.0:*               LISTEN      15/java
root@640d46e7ea18:/#
```

应用镜像

需求

基于基础的 tomcat镜像，运行两个专用的tomcat项目

简单实践

创建Dockerfile专用目录

```
mkdir /docker/web/tomcat/tomcat-app{1,2} -p
cd /docker/web/tomcat
```

定制tomcat-app1

```
cd /docker/images/tomcat/
tar xf apache-tomcat-10.0.10.tar.gz
cp apache-tomcat-10.0.10/conf/server.xml /docker/web/tomcat/tomcat-app1
```

修改 tomcat-app1/server.xml 配置文件

开放8009端口

```
<Connector protocol="AJP/1.3"
            address="::1"
            port="8009"
            redirectPort="8443" />
```

定制应用目录

```
<Host name="localhost" appBase="/data/server/tomcat/webapps"
      unpackWARs="true" autoDeploy="false">
```

定制 tomcat-app1 的应用首页

```
cd /docker/web/tomcat/tomcat-app1
mkdir ROOT
echo "Welcome to Tomcat-app1" > ROOT/index.jsp
tar zcf ROOT.tar.gz ROOT
```

定制tomcat服务启动脚本

```
# cat tomcat_service.sh
#!/bin/bash
# 定制容器里面的tomcat服务启动脚本
```

定制dns解析文件

```
echo "nameserver 192.168.8.2" > /etc/resolv.conf
```

定制服务启动命令

```
/apps/tomcat/bin/catalina.sh start
```

定制信息的输出

```
tail -f /etc/hosts
```

定制Dockerfile

构建一个基于ubuntu-tomcat定制tomcat app镜像

基础镜像

```
FROM ubuntu-tomcat:v10.0.10
```

镜像作者

```
MAINTAINER President.Wang 000000@qq.com
```

增加相关文件

```
ADD server.xml /apps/tomcat/conf/server.xml
ADD tomcat_service.sh /apps/tomcat/bin/tomcat_service.sh
ADD ROOT.tar.gz /data/server/tomcat/webapps/
```

开放tomcat服务端口

```
EXPOSE 8080 8009
```

定制容器的启动命令

```
CMD ["/bin/bash", "/apps/tomcat/bin/tomcat_service.sh"]
```

测试效果

构建镜像

```
docker build -t tomcat-web:app1 .
```

使用新镜像启动一个容器，查看效果

```
docker run -d -p 8080:8080 tomcat-web:app1
```

容器检查

```
# docker port suspicious_wright
```

```
# curl 127.0.0.1:8080
```

```
Welcome to Tomcat-app1
```

同样的方式定制第二个tomcat web镜像

准备文件

```
cd /docker/web/tomcat
```

```
cp -a tomcat-app1/* tomcat-app2/
```

```
cd tomcat-app2/
```

```
sed -i 's#app1#app2#' ROOT/index.jsp
```

构建镜像

```
docker build -t tomcat-web:app2 .
```

测试效果

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

```
curl 127.0.0.1:8081
```

小结

haproxy镜像

学习目标

这一节，我们从 haproxy镜像、简单实践、小结 三个方面来学习。

haproxy镜像

需求

基于基础的 tomcat镜像，运行两个专用的tomcat项目

简单实践

创建Dockerfile专用目录

```
mkdir /docker/web/haproxy -p
```

```
cd /docker/web/haproxy
```

获取源码文件

```
wget https://www.haproxy.org/download/2.4/src/haproxy-2.4.2.tar.gz
```

定制配置文件 haproxy.cfg

```
global
    chroot /apps/haproxy
    maxconn 4096
    uid 99
    gid 99
    daemon
    nbproc 1
    pidfile /apps/haproxy/run/haproxy.pid
    log 127.0.0.1 local3 info

defaults
    log global
    mode http
    option http-keep-alive
    option forwardfor
    retries 3
    maxconn 4096
    timeout connect 120s
    timeout client 120s
    timeout server 120s

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

listen web_port
    bind 0.0.0.0:80
    mode http
    log global
    balance roundrobin
    server web1 192.168.8.12:8080 check inter 2000 fall 3 rise 5
    server web2 192.168.8.13:8080 check inter 2000 fall 3 rise 5
```

注意:

如果主机资源有限的话，可以用一台主机来进行测试。

定制haproxy启动脚本

```
# cat haproxy_service.sh
#!/bin/bash
# 定制容器里面的haproxy服务启动脚本

# 定制服务启动命令
haproxy -f /etc/haproxy/haproxy.cfg

# 定制信息的输出
tail -f /etc/hosts
```

定制Dockerfile

```
# 构建一个基于ubuntu-ssh定制haproxy镜像
# 基础镜像
FROM ubuntu-make:v0.1
# 镜像作者
MAINTAINER President.wang 000000@qq.com
```

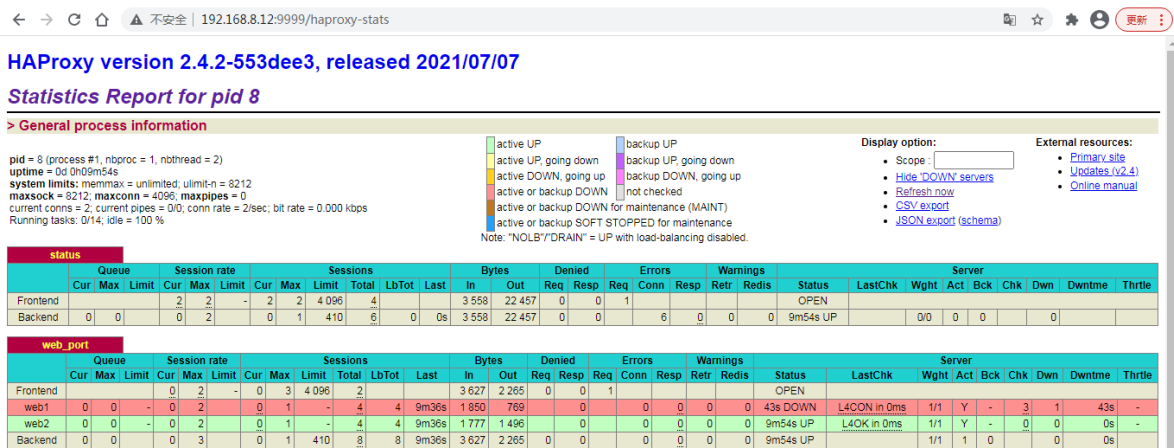


```

proxy:2.4.2 .
:
9999:9999 ubuntu-haproxy:2.4.2

```

尝试关闭一个容器，再来查看效果



其他内容

虚拟vs容器

学习目标：

这一节，我们从 虚拟化、容器化、小结 三个方面来学习。

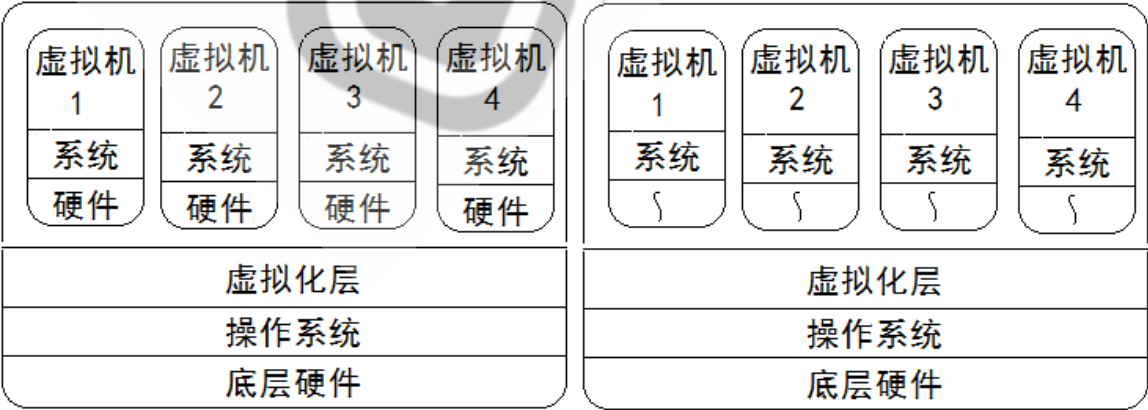
虚拟化

虚拟化

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。

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：多种工作在一段时间内同时进行
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。



虚拟机
1

虚拟机
2

虚拟机
3

虚拟机
4

系统

系统

系统

系统

⌋

⌋

⌋

⌋

虚拟化层

操作系统

底层硬件

容器化

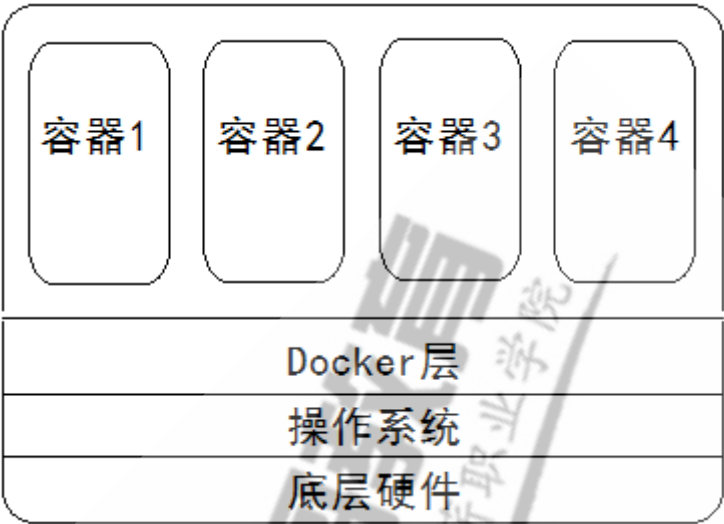
容器

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。

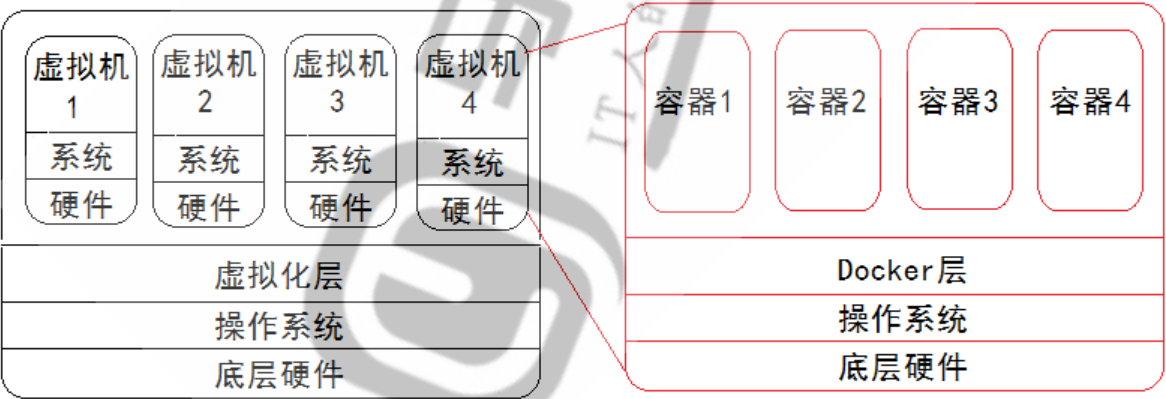
容器化技术，是虚拟化技术的另外一种实现。

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。



组合效果



小结

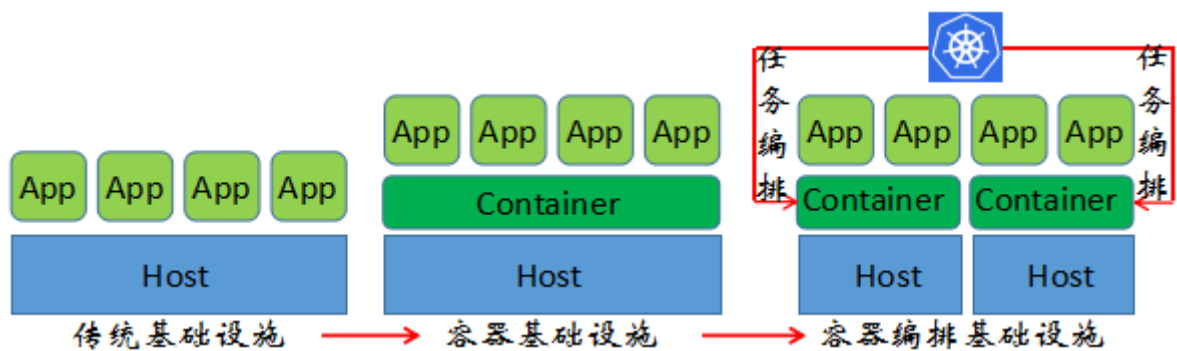
任务编排

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

应用设施演变

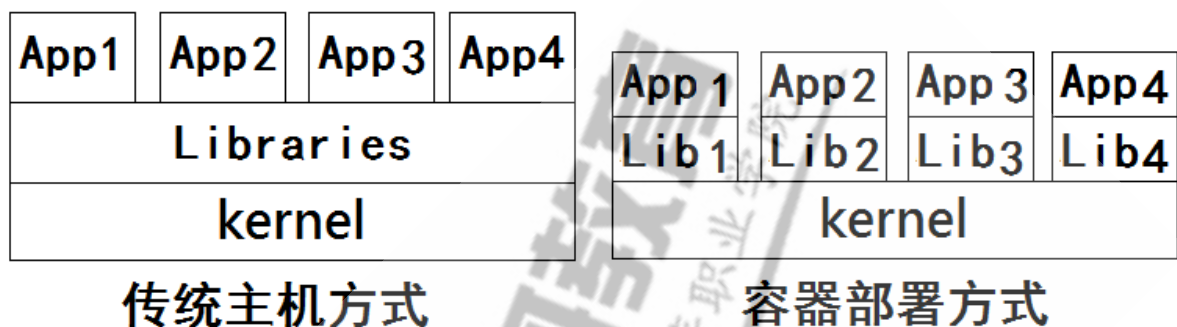


传统基础设施：应用环境和主机紧密结合

容器基础设施：应用环境和主机实现隔离

容器编排设施：解决主机容量对应用规模的限制，应用的同一管理

为什么用容器？



传统主机：部署复杂、成本高、运行慢

容器部署：部署简单、成本小、运行快

需求

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

任务编排，就是对多个子任务执行顺序进行确定的过程。

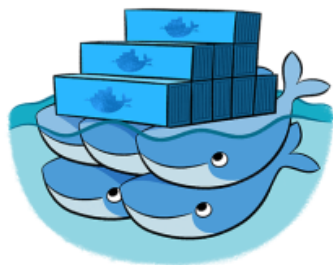
简单实践

编排工具

单机版：docker compose

集群版：容器调度平台

2017年Kubernetes以77%的市场份额成为行业事实标准



Docker-Docker swarm



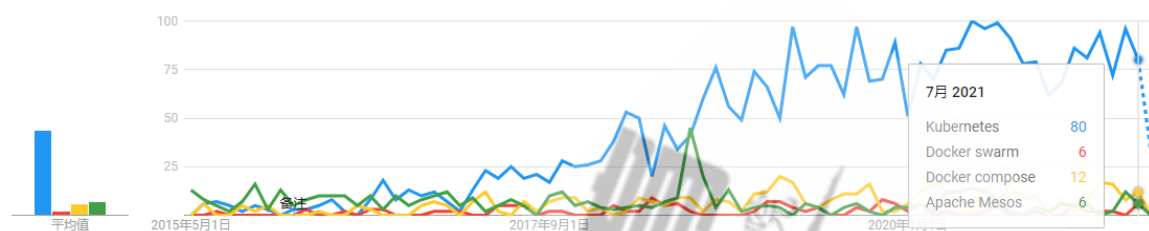
Apache-Mesos



Google-kubernetes

发展趋势

热度随时间变化的趋势 ?



为什么Kubernetes脱颖而出？



Planet Scale



Never Outgrow



Run K8s Anywhere

Google多年内部应用实践
庞大的开源社区和业界大厂鼎力支持
企业级应用的常态：快速迭代
先进的思想和架构设计，开放兼容标准 *****

小结

网络进阶

学习目标：

这一节，我们从 基础知识、容器互联、小结 三个方面来学习。

基础知识

简介

我们之前说过对于docker来说，默认采用的网络模型是 bridge，本质上是一种网络地址转换。

所以 同一宿主机上 采用默认网络模式启动的所有容器，都可以自由通信

因为 不同宿主机上 的容器的启动ip是重复的，所以原则上，默认不能自由通信

实践效果

运行容器

```
docker run -it --rm ubuntu-ssh bash
```

测试网络

```
apt install iputils-ping -y
```

```
ping -c1 www.baidu.com
```

```
ping 192.168.8.13
```

结果显示：网络是正常的，可以正常的连接宿主机能访问到的任何主机

查看路由

```
root@023494e67192:/# route -n
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	172.17.0.1	0.0.0.0	UG	0	0	0	eth0
172.17.0.0	0.0.0.0	255.255.0.0	U	0	0	0	eth0

退出容器，回到ubuntu宿主机，查看docker虚拟网卡

```
ifconfig docker0
```

查看虚拟网桥效果

```
root@python-auto:~# brctl show
```

bridge name	bridge id	STP enabled	interfaces
docker0	8000.0242933bba59	no	veth5953a39 vethb03c5c6

查看宿主机的网络转发功能

```
root@python-auto: # cat /proc/sys/net/ipv4/ip_forward  
1
```

默认情况下，同一宿主机的多个容器是可以自由通信的

启动两个容器

```
docker run --name net-test1 -d busybox sleep 36000
```

```
docker run --name net-test2 -d busybox sleep 36000
```

自由的ping通

```
# docker exec -it net-test1 sh
```

```
/ # ping 172.17.0.4 -c1
```

```
# docker exec -it net-test2 sh
```

```
/ # ping 172.17.0.3 -c1
```

我们可以通过 dockerd的参数来进行同一主机上的网络禁止通信

启动属性

```
root@python-auto:~# dockerd --help
```

```
--icc
```

Enable inter-container

communication (default true)

修改启动文件

```
root@python-auto:~# grep 'icc' /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd --icc=false -H fd:// --
containerd=/run/containerd/containerd.sock
```

重载启动文件

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

启动两个容器

```
docker start net-test1
docker start net-test2
```

查看同一主机的多容器通信效果

```
root@python-auto:~# docker exec -it net-test2 sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
150: eth0@if151: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
/ # ping -c 1 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
/ # route -n
Kernel IP routing table
Destination        Gateway            Genmask           Flags Metric Ref    Use Iface
0.0.0.0            172.17.0.1        0.0.0.0           UG    0     0        0 eth0
172.17.0.0         0.0.0.0           255.255.0.0       U     0     0        0 eth0
```

我们还可以更换docker容器本身的网络桥接设备

启动属性

```
root@python-auto:~# dockerd --help
-b, --bridge string
```

Attach containers to a network bridge

创建虚拟网桥

```
brctl addbr br0
brctl stp br0 on
ifconfig br0 10.0.1.12/24 up
brctl show
ifconfig br0
```

启动容器测试

```
docker start net-test1
docker exec -it net-test1 ping -c1 10.0.1.12
```

修改启动文件

```
root@python-auto:~# grep 'icc' /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -b=br0 -H fd:// --
containerd=/run/containerd/containerd.sock
```

重载启动文件


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

启动两个容器

```
docker start net-test1
docker inspect net-test1
```

查看同一主机的多容器通信效果

```
docker exec -it net-test1 ping -c1 www.baidu.com
docker exec -it net-test1 ping -c1 192.168.8.13
```

容器互联

需求

我们知道，容器默认启动的时候，**docker**会为其分配动态的ip地址和相关的信息，而且随着**docker**的重启，同一容器的ip也会发生不断的改变，那么我们就需要有一种能够不用查看容器ip，就可以自由的与指定容器通信的方式。

-- 容器名称互联

容器名称互联

所谓的容器名称互联，其实说的就是 我们不再以容器ip的方式进行通信，而是基于容器的名称来实现自由的通信，根据容器名称的生成方式，这里主要有两种分类：

默认的容器名称互联 和 自定义容器名称互联。

命令参数

```
root@python-auto:~# docker run --help
--link list                Add link to another container
--name string              Assign a name to the container
```

注意：

需要两个同时使用

--link的完整写法是 --link 目标容器名称:目标容器别名

通主机容器名称互联

第一个容器不要使用 --link，因为没有别的容器可连接

```
docker run --name net-name1 -d busybox sleep 36000
```

查看主机名效果

```
root@python-auto:~# docker exec -it net-name1 hostname
e8c9563c4ddb
root@python-auto:~# docker exec -it net-name1 tail -n1 /etc/hosts
10.0.1.1          e8c9563c4ddb
```

第二个容器启动的时候，使用 --link关联其他容器

```
docker run --name net-name2 --link net-name1 -d busybox sleep 36000
```

查看主机名效果

```
root@python-auto:~# docker exec -it net-name2 tail -n2 /etc/hosts
10.0.1.1          net-name1 e8c9563c4ddb
10.0.1.2          5121cf33a16f
root@python-auto:~# docker exec -it net-name2 ping -c1 net-name1
PING net-name1 (10.0.1.1): 56 data bytes
```



```
64 bytes from 10.0.1.1: seq=0 ttl=64 time=0.124 ms
```

启动第三个容器，给目标容器起一个别名

```
docker run --name net-name3 --link net-name1:server1 --link net-name2:server2 -d  
busybox sleep 36000
```

查看主机名效果

```
root@python-auto:~# docker exec -it net-name3 tail -n3 /etc/hosts  
10.0.1.1      server1 e8c9563c4ddb net-name1  
10.0.1.2      server2 5121cf33a16f net-name2  
10.0.1.3      302d5f7446c3  
root@python-auto:~# docker exec -it net-name3 ping -c1 server1  
PING server1 (10.0.1.1): 56 data bytes  
64 bytes from 10.0.1.1: seq=0 ttl=64 time=0.126 ms
```

```
root@python-auto:~# docker exec -it net-name3 ping -c1 server2  
PING server2 (10.0.1.2): 56 data bytes  
64 bytes from 10.0.1.2: seq=0 ttl=64 time=0.123 ms
```

小结

网络模型进阶

学习目标

这一节，我们从 基础知识、小结 两个方面来学习

基础知识

网络模型基础

网络模型分类：

bridge	容器连接到虚拟网桥，借助于宿主机的nat模式来上网，默认模式，自由方便，不可控
host	借助于宿主机的ip地址来进行通信，性能好，多主机容器间无网络隔离，受宿主机端口限制
none	容器本身不包含任何网络信息，需要自己的高度定制
container	容器间公用网络空间，一个进阶版的容器名称通信
overlay	

host模型进阶实践

host模型特点：直接使用宿主机的网络ip和容器本身的ip地址进行通信

```
192.168.8.12主机开启 haproxy和tomcat-app1 容器  
docker run -d --network host ubuntu-haproxy:2.4.2  
docker run -d --network host tomcat-web:app2  
netstat -tnulp | grep 80
```

```
192.168.8.13主机开启 tomcat-app2 容器  
docker run -d --network host tomcat-web:app1  
netstat -tnulp | grep 8080
```

浏览器访问 <http://192.168.8.12:9999/haproxy-stats>效果

注意:

由于host模式直接将容器端口关联到宿主主机上了,所以无法进行端口映射了

使用主机模式启动端口映射

```
docker run -d --network host -p 8888:8080 tomcat-web:app2
```

```
netstat -tnulp | grep 88
```

结果显示:

即时容器能够启动成功,也无法实现端口映射。

none模型

host模型特点: 容器本身没有进行任何网络的定制,甚至都没有自己的网卡设备,需要自己定制网络效果

启动一个none模式的容器

```
docker run -d --network none busybox sleep 36000
```

查看网络信息

```
docker inspect 99d653a3d32f
```

进入容器查看效果

```
root@python-auto:~# docker run -d --network none busybox sleep 36000
99d653a3d32f8b99f0ea7d41155e24198caef09bb35fdc620ce7506c93d87835
root@python-auto:~# docker exec -it 99d653a3d sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
/ # route -n
Kernel IP routing table
Destination        Gateway            Genmask           Flags Metric Ref    Use Iface
/ # ping www.baidu.com
ping: bad address 'www.baidu.com'
/ #
```

container模型

host模型特点: 容器间共享一个网络空间,网络空间里面的容器是否能够与主机通信,主要有第一个容器来决定

还原之前的网络环境

```
# docker rm -f $(docker ps -a -q)
```

```
# grep dockerd /lib/systemd/system/docker.service
```

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

```
# systemctl daemon-reload
```

```
# systemctl restart docker
```

第一个容器定制名称(使用桥接模型)

```
docker run -d --name web1 busybox sleep 3600
```

第二个容器使用container模型

```
docker run -d --network container:web1 --name web2 busybox sleep 36000
```

```
docker exec -it web2 ping -c1 www.baidu.com
```

结果显示:

第二个容器可以自由的与互联网通信

第一个容器定制名称(使用桥接模型)

```
docker run -d --network=none --name none1 busybox sleep 3600
```

第二个容器使用container模型

```
docker run -d --network container:none1 --name none2 busybox sleep 36000
```

```
docker exec -it none2 ping -c1 www.baidu.com
```

小结

资源隔离

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

资源隔离

需求

在一个操作系统中，启动了多个应用容器，那么这些应用容器如何来保障互相的操作不受影响？

-- 资源隔离

简介

linux内核实现namespace的主要目的，就是为了实现轻量级虚拟化技术服务。在同一个namespace下的进程合一感知彼此的变化，而对外界的进程一无所知。

Docker通过linux的 pid、net、ipc、mnt、uts、user、time这七类的namespace将容器的进程、网络、消息、文件系统、UTS、时间和操作系统资源隔离开。从而让容器中的进程产生错觉，仿佛自己置身一个独立的系统环境中，以达到隔离的目的。

namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名或域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户组和用户组
Time	CLONE_NEWTIME	启动和单调时钟

```

root@python-auto:/var/lib/containerd# ps aux | grep docker | grep -v grep
root      7482   0.0  1.6 946924 98440 ?        Ssl  09:08   0:06 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/container
d.sock
root@python-auto:/var/lib/containerd# ll /proc/7482/ns/
总用量 0
dr-x--x--x  2 root root 0 7月 27 09:08 ./
dr-xr-xr-x  9 root root 0 7月 27 09:08 ../
lrwxrwxrwx  1 root root 0 7月 27 09:08 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx  1 root root 0 7月 27 15:00 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx  1 root root 0 7月 27 15:00 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx  1 root root 0 7月 27 15:00 net -> 'net:[4026531992]'
lrwxrwxrwx  1 root root 0 7月 27 15:00 pid -> 'pid:[4026531836]'
lrwxrwxrwx  1 root root 0 7月 27 15:35 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx  1 root root 0 7月 27 15:35 time -> 'time:[4026531834]'
lrwxrwxrwx  1 root root 0 7月 27 15:35 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx  1 root root 0 7月 27 15:00 user -> 'user:[4026531837]'
lrwxrwxrwx  1 root root 0 7月 27 15:00 uts -> 'uts:[4026531838]'

```

简单实践

准备工作

随便启动几个容器

```
docker start nginx
```

```
docker run -d busybox sleep 3600
```

基本信息

```

# docker inspect nginx-3
[
  {
    ...,
    "Image":
      "sha256:08b152afcfcae220e9709f00767054b824361c742ea03a9fe936271ba520a0a4b",
    "ResolvConfPath":
      "/var/lib/docker/containers/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae86
      2d64ad6c803b/resolv.conf",
    "HostnamePath":
      "/var/lib/docker/containers/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae86
      2d64ad6c803b/hostname",
    "HostsPath":
      "/var/lib/docker/containers/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae86
      2d64ad6c803b/hosts",
    "LogPath":
      "/var/lib/docker/containers/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae86
      2d64ad6c803b/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae862d64ad6c803b-
      json.log",
    ...,
    "GraphDriver": {
      "Data": {
        ...,
        "MergedDir":
          "/var/lib/docker/overlay2/6b518bf9522d8f2bc03a4bc9e2dd4a8913d53055078e1999595597
          aba6ada311/merged",
        ...
      },
      "Name": "overlay2"
    },
    "Mounts": [],
    "Config": {
      ...,
      "SandboxKey": "/var/run/docker/netns/be46cb060421",
      ...,
      "EndpointID":
        "04b7d16be4520daba67bc9484015bf880abe769b29f740e33c211c4660064c1d",
      ...,
      "Networks": {

```

```

        "bridge": {
            "IPAMConfig": null,
            "Links": null,
            "Aliases": null,
            "NetworkID":
"66aa1a0fa13b95ce23564daffa6a617f828fa542f0aa53efd34624650faab337",
            "EndpointID":
"04b7d16be4520daba67bc9484015bf880abe769b29f740e33c211c4660064c1d",
            "Gateway": "172.17.0.1",
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "02:42:ac:11:00:02",
            "DriverOpts": null
        }
    }
}
]

```

文件系统隔离

```

root@python-auto:/var/lib/containerd# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS          NAMES
b1cd5c3c282c   busybox   "tail -f"               57 minutes ago Up 56 minutes          gracious_euclid
f5f697d2af24   busybox   "sleep 3600"           58 minutes ago Up 58 minutes          gracious_shockley
root@python-auto:/var/lib/containerd# docker exec -it b1cd5c3c282c cat /proc/self/mountinfo | head -5
1155 490 0:67 / / rw,relatime master:504 - overlay overlay rw,lowerdir=/var/lib/docker/overlay2/L/LDZED63KHWJJA5V6QVTRY3AC3:/var/lib/docker/overlay2/L/HVLM44R4GCIQSW34F2JFCRW5W,upperdir=/var/lib/docker/overlay2/5d128125eb3938b63d076f40657df07274590ac78d70b085acc82046530bf746/diff,workdir=/var/lib/docker/overlay2/5d128125eb3938b63d076f40657df07274590ac78d70b085acc82046530bf746/work
1156 1155 0:70 / /proc rw,nosuid,nodev,noexec,relatime - proc proc rw
1157 1155 0:71 / /dev rw,nosuid - tmpfs tmpfs rw,size=65536k,mode=755
1158 1157 0:72 / /dev/pts rw,nosuid,noexec,relatime - devpts devpts rw,gid=5,mode=620,ptmxmode=666
1159 1155 0:73 / /sys ro,nosuid,nodev,noexec,relatime - sysfs sysfs ro
root@python-auto:/var/lib/containerd# docker exec -it f5f697d2af24 cat /proc/self/mountinfo | head -5
1069 435 0:54 / / rw,relatime master:471 - overlay overlay rw,lowerdir=/var/lib/docker/overlay2/L/OZIVD6WDY2AGP5DAL4T5Y3NU7M:/var/lib/docker/overlay2/L/HVLM44R4GCIQSW34F2JFCRW5W,upperdir=/var/lib/docker/overlay2/074b1cd75bc65d0d73f0609d0a1d0176794fba0d9c50fff86aaa3c67a26d68b3/diff,workdir=/var/lib/docker/overlay2/074b1cd75bc65d0d73f0609d0a1d0176794fba0d9c50fff86aaa3c67a26d68b3/work
1070 1069 0:57 / /proc rw,nosuid,nodev,noexec,relatime - proc proc rw
1071 1069 0:58 / /dev rw,nosuid - tmpfs tmpfs rw,size=65536k,mode=755
1072 1071 0:59 / /dev/pts rw,nosuid,noexec,relatime - devpts devpts rw,gid=5,mode=620,ptmxmode=666
1073 1069 0:60 / /sys ro,nosuid,nodev,noexec,relatime - sysfs sysfs ro

root@python-auto:/var/lib/containerd# ls /var/lib/docker/overlay2/6b518bf9522d8f2bc03a4bc9e2dd4a8913d53055078e1999595597aba6ada311/merged
bin      dev      docker-entrypoint.sh  home  lib64  mnt  proc  run  srv  tmp  var
boot     docker-entrypoint.d  etc      lib   media  opt  root  sbin sys  usr
root@python-auto:/var/lib/containerd# docker exec -it nginx-3 /bin/bash
root@845c58fc4825:/# ls
bin      dev      docker-entrypoint.sh  home  lib64  mnt  proc  run  srv  tmp  var
boot     docker-entrypoint.d  etc      lib   media  opt  root  sbin sys  usr

```

UTC资源隔离

```

root@python-auto:/var/lib/containerd# cat /var/lib/docker/containers/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae862d64ad6c803b/hostname
845c58fc4825
root@python-auto:/var/lib/containerd# cat /var/lib/docker/containers/845c58fc4825d152aea0091610e36d030b9a1b7227262314ae862d64ad6c803b/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     845c58fc4825
root@python-auto:/var/lib/containerd#

```

IPC资源隔离

```

root@python-auto:/var/lib/containerd# ipcmk -Q
消息队列 id: 0
root@python-auto:/var/lib/containerd# ipcs

----- 消息队列 -----
键          msqid      拥有者    权限      已用字节数 消息
0x15b9eeb9  0                root      644        0           0

----- 共享内存段 -----
键          shmid      拥有者    权限      字节        连接数    状态

----- 信号量数组 -----
键          semid      拥有者    权限      nsems

root@python-auto:/var/lib/containerd# docker exec -it f5f697d2af24 ipcs

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes        nattch     status

----- Semaphore Arrays -----
key          semid      owner      perms      nsems

```

ipcmk -Q 建立一个消息队列
ipcs 查看消息情况

PID资源隔离

```

root@python-auto:/var/lib/containerd# docker run -d busybox sleep 3600
f5f697d2af2427dbccf8f2bb7a95f32e7d35c41108305a2a53cffe90b35c3c1f
root@python-auto:/var/lib/containerd# docker run -d busybox tail -f
b1cd5c3c282c3246d104598f9585c2be421cfab0f44f75fa665547acbe11036f
root@python-auto:/var/lib/containerd# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS   NAMES
b1cd5c3c282c   busybox   "tail -f"                4 seconds ago Up 3 seconds             gracious_euclid
f5f697d2af24   busybox   "sleep 3600"             About a minute ago Up About a minute         gracious_shockley
root@python-auto:/var/lib/containerd# docker exec -it b1cd5c3c282c pstree
tail
root@python-auto:/var/lib/containerd# docker exec -it f5f697d2af24 pstree
sleep
root@python-auto:/var/lib/containerd# pstree
systemd--ManagementAgent--6*[{ManagementAgent}]
--ModemManager--2*[{ModemManager}]
--NetworkManager--2*[{NetworkManager}]
--VGAuthService
--accounts-daemon--2*[{accounts-daemon}]
--acpid

```

```

root@python-auto:/var/lib/containerd# docker inspect --format '{{.State.Pid}}' b1cd5c3c282c
69167
root@python-auto:/var/lib/containerd# docker inspect --format '{{.State.Pid}}' f5f697d2af24
69083

```

```

root@python-auto:~# docker exec -it web1 top -n 1
Mem: 5559284K used, 563224K free, 5816K shrd, 230920K buff, 4231496K cached
CPU:  5.0% usr  5.0% sys  0.0% nic 90.0% idle  0.0% io  0.0% irq  0.0% irq
Load average: 0.00 0.00 0.00 3/538 35
  PID  PPID  USER    STAT  VSZ %VSZ CPU %CPU COMMAND
   29    0 root     R    1324  0.0  1  0.0 top -n 1
    1    0 root     S    1312  0.0  1  0.0 sleep 3600
root@python-auto:~# top -n 1
top - 03:44:26 up 7:27,  2 users,  load average: 0.00, 0.00, 0.00
任务: 290 total,  1 running, 289 sleeping,  0 stopped,  0 zombie
%Cpu(s):  5.1 us,  5.1 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  5979.0 total,  564.9 free,  771.6 used,  4642.5 buff/cache
MiB Swap:  2048.0 total,  2041.2 free,  6.8 used.  4919.3 avail Mem

  进程号  USER    PR  NI   VIRT   RES   SHR  %CPU  %MEM    TIME+  COMMAND
    51638 root    20   0   20644   4008   3240 R   11.1   0.1   0:00.07 top
      1 root    20   0 168924 12988   8356 S    0.0   0.2   0:19.55 systemd

```

NET资源隔离


```

root@python-auto:/var/lib/containerd# ls /var/lib/docker/network/files/
local-kv.db
root@python-auto:/var/lib/containerd# docker exec -it b1cd5c3c282c ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
12: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
root@python-auto:/var/lib/containerd# docker exec -it f5f697d2af24 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever

```

User资源隔离

```

root@python-auto:/var/lib/containerd# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS   NAMES
b1cd5c3c282c   busybox   "tail -f"               49 minutes ago Up 49 minutes          gracious_euclid
f5f697d2af24   busybox   "sleep 3600"            50 minutes ago Up 50 minutes          gracious_shockley
root@python-auto:/var/lib/containerd# docker exec -it f5f697d2af24 id
uid=0(root) gid=0(root) groups=10(wheel)
root@python-auto:/var/lib/containerd# docker exec -it b1cd5c3c282c id
uid=0(root) gid=0(root) groups=10(wheel)
root@python-auto:/var/lib/containerd# id
uid=0(root) gid=0(root) 组=0(root)

```

每个容器都有自己的用户和用户组，可以是重复的

资源控制

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

虽然我们目前说到容器，大部分时候，指的还是Docker，其实容器技术，并不是从docker才出来的，它经历了非常多的技术更新换代，最终才出现了docker。Docker的资源隔离思路是来源于其他的技术，同样，资源控制也是来源于其他技术。

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

默认情况下，资源限制的功能已经开启了

```

root@python-auto:/var/lib/containerd# grep CGROUP /boot/config-5.8.0-63-generic
CONFIG_CGROUPS=y
CONFIG_BLK_CGROUP=y
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_BLK_CGROUP_RWSTAT=y
# CONFIG_BLK_CGROUP_IOLATENCY is not set
CONFIG_BLK_CGROUP_IOCOST=y
# CONFIG_BFQ_CGROUP_DEBUG is not set
CONFIG_NETFILTER_XT_MATCH_CGROUP=m
CONFIG_NET_CLS_CGROUP=m
CONFIG_CGROUP_NET_PRI0=y
CONFIG_CGROUP_NET_CLASSID=y

```

系统cgroup的体现

```

root@python-auto:/var/lib/containerd# ls /sys/fs/cgroup/ -l
总用量 0
dr-xr-xr-x 6 root root 0 7月 27 09:05 blkio
lrwxrwxrwx 1 root root 11 7月 27 09:05 cpu -> cpu,cpuacct
lrwxrwxrwx 1 root root 11 7月 27 09:05 cpuacct -> cpu,cpuacct
dr-xr-xr-x 6 root root 0 7月 27 09:05 cpu,cpuacct
dr-xr-xr-x 3 root root 0 7月 27 09:05 cpuset
dr-xr-xr-x 6 root root 0 7月 27 09:05 devices
dr-xr-xr-x 3 root root 0 7月 27 09:05 freezer
dr-xr-xr-x 3 root root 0 7月 27 09:05 hugetlb
dr-xr-xr-x 6 root root 0 7月 27 09:05 memory
lrwxrwxrwx 1 root root 16 7月 27 09:05 net_cls -> net_cls,net_prio
dr-xr-xr-x 3 root root 0 7月 27 09:05 net_cls,net_prio
lrwxrwxrwx 1 root root 16 7月 27 09:05 net_prio -> net_cls,net_prio
dr-xr-xr-x 3 root root 0 7月 27 09:05 perf_event
dr-xr-xr-x 6 root root 0 7月 27 09:05 pids
dr-xr-xr-x 2 root root 0 7月 27 09:05 rdma
dr-xr-xr-x 6 root root 0 7月 27 09:05 systemd
dr-xr-xr-x 5 root root 0 7月 27 09:05 unified

```

以cpu为例，查看效果

```

root@python-auto:~# tree /sys/fs/cgroup/cpu/docker/
/sys/fs/cgroup/cpu/docker/
├── b1cd5c3c282c3246d104598f9585c2be421cfab0f44f75fa665547acbe11036f
│   ├── cgroup.clone_children
│   ├── cgroup.procs
│   ├── cpuacct.stat
│   ├── cpuacct.usage
│   ├── cpuacct.usage_all
│   ├── cpuacct.usage_percpu
│   ├── cpuacct.usage_percpu_sys
│   ├── cpuacct.usage_percpu_user
│   ├── cpuacct.usage_sys
│   ├── cpuacct.usage_user
│   ├── cpu.cfs_period_us
│   ├── cpu.cfs_quota_us
│   └── cpu.shares

```



```

|   ├── cpu.stat
|   ├── cpu.uclamp.max
|   ├── cpu.uclamp.min
|   ├── notify_on_release
|   └── tasks
└── cgroup.clone_children
└── ...
└── f5f697d2af2427dbccf8f2bb7a95f32e7d35c41108305a2a53cffe90b35c3c1f
    |   ...
    |   └── tasks
    └── notify_on_release
        └── tasks

2 directories, 54 files

```

简单实践

简介

默认情况下，容器没有资源的 "使用限制" -- 可以使用尽可能多的主机资源。

Docker提供了控制容器使用资源的方法，可以限制当前容器可以使用的内存数量和cpu资源。我们可以通过 docker run的参数来进行简单的控制

```

root@python-auto:~# docker run --help | grep -E 'cpu|oom|memory'
--cpu-period int          Limit CPU CFS (Completely Fair Scheduler) period
--cpu-quota int           Limit CPU CFS (Completely Fair Scheduler) quota
--cpu-rt-period int       Limit CPU real-time period in microseconds
--cpu-rt-runtime int      Limit CPU real-time runtime in microseconds
-c, --cpu-shares int       CPU shares (relative weight)
--cpus decimal            Number of CPUs
--cpuset-cpus string       CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems string       MEMs in which to allow execution (0-3, 0,1)
--kernel-memory bytes     Kernel memory limit
-m, --memory bytes        Memory limit
--memory-reservation bytes Memory soft limit
--memory-swap bytes       Swap limit equal to memory plus swap: '-1' to enable unlimited swap
--memory-swappiness int    Tune container memory swappiness (0 to 100) (default -1)
--oom-kill-disable        Disable OOM Killer
--oom-score-adj int       Tune host's OOM preferences (-1000 to 1000)

```

网上有一个非常好的容器压力测试工具，docker-stress-ng 我们可以基于该镜像来进行各种容量的限制测试

下载镜像

```
docker pull lorel/docker-stress-ng
```

查看容器的命令帮助

```
docker run -it --rm lorel/docker-stress-ng
```

```

...
-c N, --cpu N          start N workers spinning on sqrt(rand())      设定cpu的
数量
--cpu-ops N            stop when N cpu bogo operations completed
-l P, --cpu-load P     load CPU by P %, 0=sleep, 100=full load (see -c)
--cpu-method m         specify stress cpu method m, default is all
...
-m N, --vm N           start N workers spinning on anonymous mmap      设定
线程数
--vm-bytes N           allocate N bytes per vm worker (default 256MB) 默认
容量

```

内存容量限制

```
# 开启两个work的容器，每个work默认占256m，当前容器占 512m
root@python-auto:~# docker run --name mem_stress -it --rm lorel/docker-stress-ng
--vm 2
stress-ng: info: [1] defaulting to a 86400 second run per stressor
stress-ng: info: [1] dispatching hogs: 2 vm

# 另开一个终端，查看状态
root@python-auto:~# docker stats
```

```
root@python-auto:~# docker stats
CONTAINER ID   NAME          CPU %       MEM USAGE / LIMIT   MEM %       NET I/O     BLOCK I/O    PIDS
bfe700399d3a   mem_stress    0.13%      516.7MiB / 5.839GiB  8.64%      2.66kB / 0B  0B / 0B      5
b1cd5c3c282c   gracious_euclid 0.01%      2.105MiB / 5.839GiB  0.04%      4.46kB / 0B  0B / 0B      1
f5f697d2af24   gracious_shockley 0.00%      1.438MiB / 5.839GiB  0.02%      3.81kB / 0B  0B / 0B      1
```

```
# 关闭容器后，调整最大的容器内存量为 100m
root@python-auto:~# docker run --name mem_stress -it --rm -m 100m lorel/docker-stress-ng --vm 2
stress-ng: info: [1] defaulting to a 86400 second run per stressor
stress-ng: info: [1] dispatching hogs: 2 vm

# 另开一个终端，查看状态
root@python-auto:~# docker stats
```

```
root@python-auto:~# docker stats
CONTAINER ID   NAME          CPU %       MEM USAGE / LIMIT   MEM %       NET I/O     BLOCK I/O    PIDS
317f6e699048   mem_stress    0.03%      99.95MiB / 100MiB   99.95%      2.37kB / 0B  176kB / 0B    5
b1cd5c3c282c   gracious_euclid 0.00%      2.105MiB / 5.839GiB  0.04%      4.46kB / 0B  0B / 0B      1
f5f697d2af24   gracious_shockley 0.00%      1.438MiB / 5.839GiB  0.02%      3.81kB / 0B  0B / 0B      1
```

cpu资源控制实践

```
# 查看当前的cpu梳理
root@python-auto:~# lscpu | grep CPU
CPU 运行模式:          32-bit, 64-bit
CPU:                    2
在线 CPU 列表:         0,1
...

# 开启使用两个cpu的容器
root@python-auto:~# docker run --name cpu_stress -it --rm lorel/docker-stress-ng --cpu 2
stress-ng: info: [1] defaulting to a 86400 second run per stressor
stress-ng: info: [1] dispatching hogs: 2 cpu

# 查看状态
docker stats --no-stream
```

```
root@python-auto:~# docker stats --no-stream
CONTAINER ID   NAME          CPU %       MEM USAGE / LIMIT   MEM %       NET I/O     BLOCK I/O    PIDS
1854a41b4ec2   cpu_stress    197.70%     7.422MiB / 5.839GiB  0.12%      2.37kB / 0B  0B / 0B      3
b1cd5c3c282c   gracious_euclid 0.01%      2.105MiB / 5.839GiB  0.04%      4.46kB / 0B  0B / 0B      1
f5f697d2af24   gracious_shockley 0.00%      1.438MiB / 5.839GiB  0.02%      3.98kB / 0B  0B / 0B      1
root@python-auto:~# cat /sys/fs/cgroup/cpuset/docker/1854a41b4ec28ff3e4ae6468d7dcdf695351f35cf9399bd77e3cab55783a7cb2/cpuset.cpus
0-1
```

调整cpu的使用数量

```
root@python-auto:~# docker run --name cpu_stress -it --rm --cpus 1
lore1/docker-stress-ng --cpu 2
stress-ng: info: [1] defaulting to a 86400 second run per stressor
stress-ng: info: [1] dispatching hogs: 2 cpu
^Cstress-ng: info: [1] successful run completed in 438.80s
root@python-auto:~# docker run --name cpu_stress -it --rm --cpus 1.5
lore1/docker-stress-ng --cpu 2
stress-ng: info: [1] defaulting to a 86400 second run per stressor
stress-ng: info: [1] dispatching hogs: 2 cpu
^X^Cstress-ng: info: [1] successful run completed in 17.57s

# 查看状态
docker stats --no-stream
```

```
root@python-auto:~# docker stats --no-stream
CONTAINER ID   NAME          CPU %      MEM USAGE / LIMIT   MEM %      NET I/O     BLOCK I/O   PIDS
b94e366eca2d   cpu_stress    99.07%     7.543MiB / 5.839GiB  0.13%      3.19kB / 0B  0B / 0B     3
b1cd5c3c282c   gracious_euclid 0.01%     2.105MiB / 5.839GiB  0.04%      4.57kB / 0B  0B / 0B     1
f5f697d2af24   gracious_shockley 0.00%     1.438MiB / 5.839GiB  0.02%      4.09kB / 0B  0B / 0B     1
root@python-auto:~# docker stats --no-stream
CONTAINER ID   NAME          CPU %      MEM USAGE / LIMIT   MEM %      NET I/O     BLOCK I/O   PIDS
9bc5a66edf00   cpu_stress    146.58%    7.074MiB / 5.839GiB  0.12%      2.48kB / 0B  0B / 0B     3
b1cd5c3c282c   gracious_euclid 0.00%     2.105MiB / 5.839GiB  0.04%      4.57kB / 0B  0B / 0B     1
f5f697d2af24   gracious_shockley 0.00%     1.438MiB / 5.839GiB  0.02%      4.09kB / 0B  0B / 0B     1
root@python-auto:~#
```

定制cpu绑定效果

```
root@python-auto:~# docker run --name cpu_stress -it --rm --cpus 1 --cpuset-
cpus 1 lore1/docker-stress-ng --cpu 2

# 查看状态
docker stats --no-stream
```

```
root@python-auto:~# docker stats --no-stream
CONTAINER ID   NAME          CPU %      MEM USAGE / LIMIT   MEM %      NET I/O     BLOCK I/O   PIDS
ff58b4fdad3e   cpu_stress    95.04%     6.891MiB / 5.839GiB  0.12%      2.1kB / 0B  0B / 0B     3
b1cd5c3c282c   gracious_euclid 0.01%     2.105MiB / 5.839GiB  0.04%      4.57kB / 0B  0B / 0B     1
f5f697d2af24   gracious_shockley 0.00%     1.438MiB / 5.839GiB  0.02%      4.09kB / 0B  0B / 0B     1
root@python-auto:~# cat /sys/fs/cgroup/cpuset/docker/ff58b4fdad3e8665a19e2c1a40d617f9aa75f8527d32272f8c6d563f3d94
854a/cpuset.cpus
1
```