Docker

学习目标：

- 掌握Docker基础知识，能够理解Docker镜像与容器的概念
  模板　　看作是服务器
- 完成Docker安装与启动
- 掌握Docker镜像与容器相关命令
- 掌握Tomcat Nginx 等软件的常用应用的安装
- 掌握docker迁移与备份相关命令
- 能够运用Dockerfile编写创建容器的脚本
- 能够搭建与使用docker私有仓库

1 Docker简介

1.1 什么是虚拟化

在计算机中，虚拟化（英语：Virtualization）是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来，打破实体结构间的不可切割的障碍，使用户可以比原本的组态更好的方式来应用这些资源。这些资源的新虚拟部份是不受现有资源的架设方式，地域或物理组态所限制。一般所指的虚拟化资源包括计算能力和资料存储。

在实际的生产环境中，虚拟化技术主要用来解决高性能的物理硬件产能过剩和老的旧的硬件产能过低的重组重用，透明化底层物理硬件，从而最大化的利用物理硬件　对资源充分利用

虚拟化技术种类很多，例如：软件虚拟化、硬件虚拟化、内存虚拟化、网络虚拟化(vip)、桌面虚拟化、服务虚拟化、虚拟机等等。

1.2 什么是Docker

Docker 是一个开源项目，诞生于 2013 年初，最初是 dotCloud 公司内部的一个业余项目。它基于 Google 公司推出的 Go 语言实现。 项目后来加入了 Linux 基金会，遵从了 Apache 2.0 协议，项目代码在 GitHub 上进行维护。

Docker 自开源后受到广泛的关注和讨论，以至于 dotCloud 公司后来都改名为 Docker Inc。Redhat 已经在其 RHEL6.5 中集中支持 Docker；Google 也在其 PaaS 产品中广泛应用。

Docker 项目的目标是实现轻量级的操作系统虚拟化解决方案。 Docker 的基础是 Linux 容器（LXC）等技术。

在 LXC 的基础上 Docker 进行了进一步的封装，让用户不需要去关心容器的管理，使得操作更为简便。用户操作 Docker 的容器就像操作一个快速轻量级的虚拟机一样简单。

为什么选择Docker?

（1）上手快。

用户只需要几分钟，就可以把自己的程序“Docker化”。Docker依赖于“写时复制”（copy-on-write）模型，使修改应用程序也非常迅速，可以说达到“随心所致，代码即改”的境界。

随后，就可以创建容器来运行应用程序了。大多数Docker容器只需要不到1秒中即可启动。由于去除了管理程序的开销，Docker容器拥有很高的性能，同时同一台宿主机中也可以运行更多的容器，使用户尽可能的充分利用系统资源。

（2）职责的逻辑分类

使用Docker，开发人员只需要关心容器中运行的应用程序，而运维人员只需要关心如何管理容器。Docker设计的目的就是要加强开发人员写代码的开发环境与应用程序要部署的生产环境一致性。从而降低那种“开发时一切正常，肯定是运维的问题（测试环境都是正常的，上线后出了问题就归结为肯定是运维的问题）”

（3）快速高效的开发生命周期

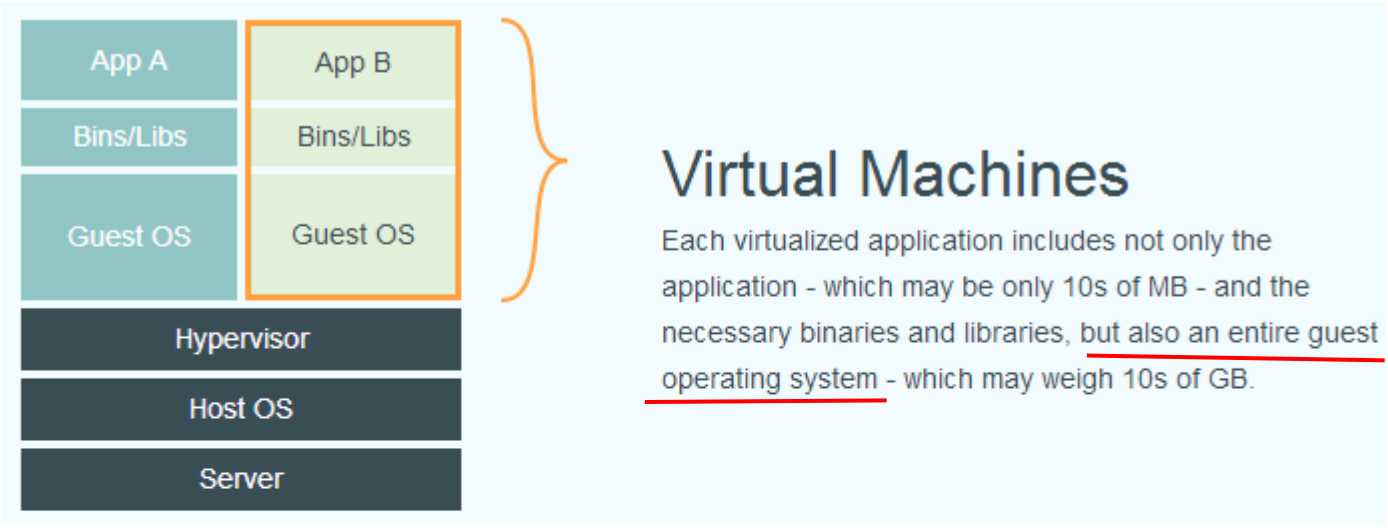
Docker的目标之一就是缩短代码从开发、测试到部署、上线运行的周期，让你的应用程序具备可移植性，易于构建，并易于协作。（通俗一点说，Docker就像一个盒子，里面可以装很多物件，如果需要这些物件的可以直接将该大盒子拿走，而不需要从该盒子中一件件的取。）

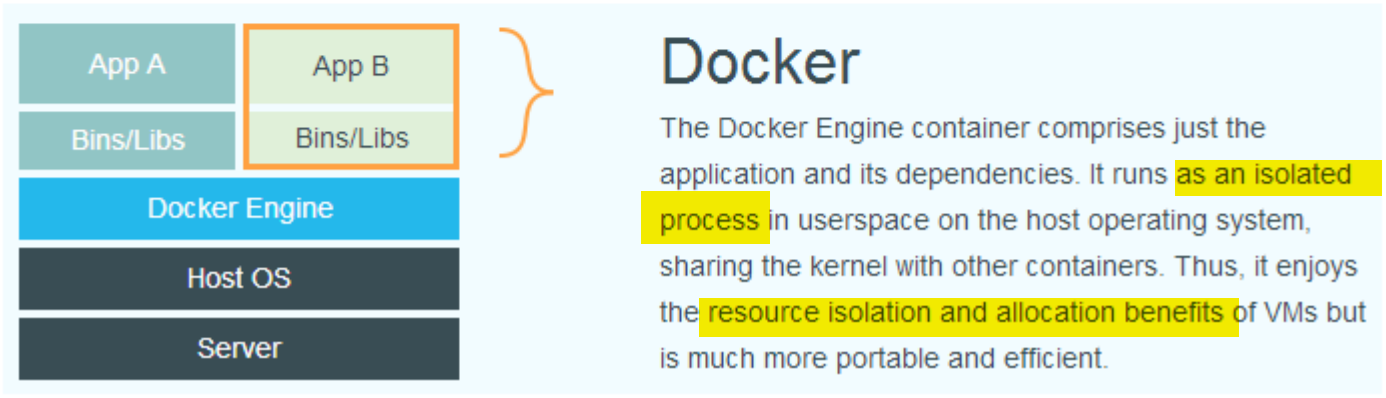
（4）鼓励使用面向服务的架构

Docker还鼓励面向服务的体系结构和微服务架构。Docker推荐单个容器只运行一个应用程序或进程，这样就形成了一个分布式的应用程序模型，在这种模型下，应用程序或者服务都可以表示为一系列内部互联的容器，从而使分布式部署应用程序，扩展或调试应用程序都变得非常简单，同时也提高了程序的内省性。（当然，可以在一个容器中运行多个应用程序）

1.3 容器与虚拟机比较

下面的图片比较了 Docker 和传统虚拟化方式的不同之处，可见容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，而传统方式则是在硬件层面实现。



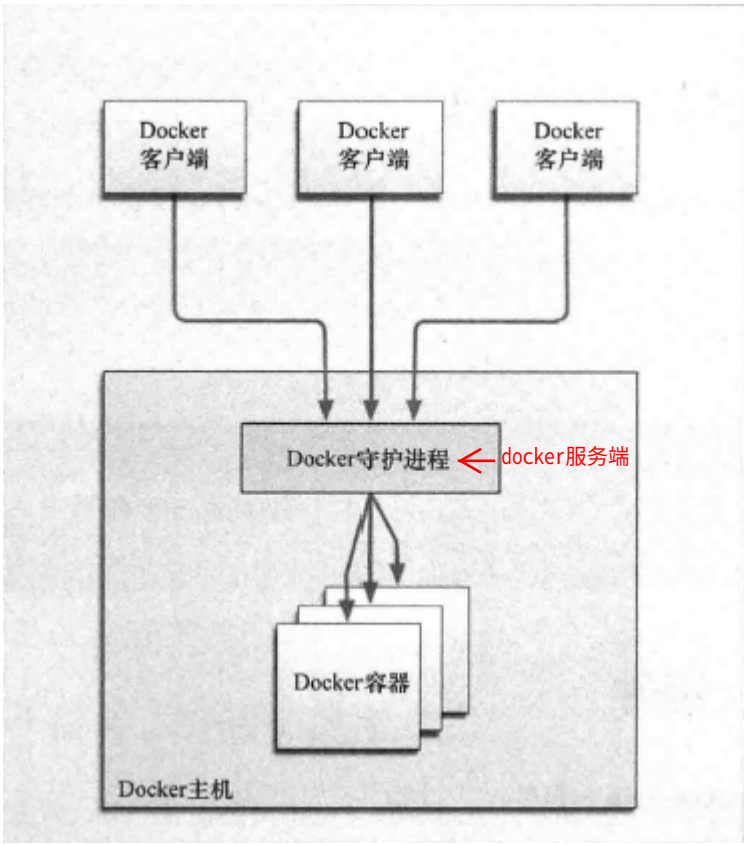


与传统的虚拟机相比，Docker优势体现为启动速度快、占用体积小。

1.4 Docker 组件

1.4.1 Docker服务器与客户端

Docker是一个客户端-服务器（C/S）架构程序。Docker客户端只需要向Docker服务器或者守护进程发出请求，服务器或者守护进程将完成所有工作并返回结果。Docker提供了一个命令行工具Docker以及一整套RESTful API。你可以在同一台宿主机上运行Docker守护进程和客户端，也可以从本地的Docker客户端连接到运行在另一台宿主机上的远程Docker守护进程。



1.4.2 Docker镜像与容器

镜像是构建Docker的基石。用户基于镜像来运行自己的容器。镜像也是Docker生命周期中的“构建”部分。镜像是基于联合文件系统的一种层式结构，由一系列指令一步一步构建出来。例如：

添加一个文件；

执行一个命令；　　一个镜像可以有很多容器，就像一个类可以有很多对象一样

打开一个窗口。

也可以将镜像当作容器的“源代码”。镜像体积很小，非常“便携”，易于分享、存储和更新。

Docker可以帮助你构建和部署容器，你只需要把自己的应用程序或者服务打包放进容器即可。容器是基于镜像启动起来的，容器中可以运行一个或多个进程。我们可以认为，镜像是Docker生命周期中的构建或者打包阶段，而容器则是启动或者执行阶段。 容器基于镜像启动，一旦容器启动完成后，我们就可以登录到容器中安装自己需要的软件或者服务。

所以Docker容器就是：

一个镜像格式；

一些列标准操作；

一个执行环境。

Docker借鉴了标准集装箱的概念。标准集装箱将货物运往世界各地，Docker将这个模型运用到自己的设计中，唯一不同的是：集装箱运输货物，而Docker运输软件。

和集装箱一样，Docker在执行上述操作时，并不关心容器中到底装了什么，它不管是web服务器，还是数据库，

或者是应用程序服务器什么的。所有的容器都按照相同的方式将内容“装载”进去。

Docker也不关心你要把容器运到何方：我们可以在自己的笔记本中构建容器，上传到Registry，然后下载到一个物理的或者虚拟的服务器来测试，在把容器部署到具体的主机中。像标准集装箱一样，Docker容器方便替换，可以叠加，易于分发，并且尽量通用。

1.4.3 Registry（注册中心）

Docker用Registry来保存用户构建的镜像。Registry分为公共和私有两种。Docker公司运营公共的Registry叫做Docker Hub。用户可以在Docker Hub注册账号，分享并保存自己的镜像（说明：在Docker Hub下载镜像巨慢，可以自己构建私有的Registry）。

https://hub.docker.com/

2 Docker安装与启动

2.1 安装Docker

Docker官方建议在Ubuntu中安装，因为Docker是基于Ubuntu发布的，而且一般Docker出现的问题Ubuntu是最先更新或者打补丁的。在很多版本的CentOS中是不支持更新最新的一些补丁包的。

由于我们学习的环境都使用的是CentOS，因此这里我们将Docker安装到CentOS上。注意：这里建议安装在CentOS7.x以上的版本，在CentOS6.x的版本中，安装前需要安装其他很多的环境而且Docker很多补丁不支持更新。

root：itcast

请直接挂载课程配套的Centos7.x镜像

（1）yum 包更新到最新

```
sudo yum update
```

（2）安装需要的软件包， yum-util 提供yum-config-manager功能，另外两个是devicemapper驱动依赖的

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

（3）设置yum源为阿里云

```
sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
```

（4）安装docker

```
sudo yum install docker-ce
```

（5）安装后查看docker版本

```
docker -v
```

2.2 设置ustc的镜像

ustc是老牌的linux镜像服务提供者了，还在遥远的ubuntu 5.04版本的时候就在用。ustc的docker镜像加速器速度很快。ustc docker mirror的优势之一就是不需要注册，是真正的公共服务。

https://lug.ustc.edu.cn/wiki/mirrors/help/docker

编辑该文件：

```
vi /etc/docker/daemon.json
```

在该文件中输入如下内容：

中国科技大学的镜像加速器：中科大的加速器不用注册，直接使用地址

```
{ "registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"] }
```

2.3 Docker的启动与停止

systemctl命令是系统服务管理器指令

启动docker：

```
systemctl start docker
```

停止docker：

```
systemctl stop docker
```

重启docker：

```
systemctl restart docker
```

查看docker状态：

```
systemctl status docker
```

开机启动：

```
systemctl enable docker
```

查看docker概要信息

```
docker info
```

查看docker帮助文档

```
docker --help
```

3 常用命令

3.1 镜像相关命令

3.1.1 查看镜像

```
docker images
```

REPOSITORY：镜像名称

TAG：镜像标签 版本

IMAGE ID：镜像ID id有可能重复

CREATED：镜像的创建日期（不是获取该镜像的日期）

SIZE：镜像大小

使用find ./ -name java命令可以找出来，可以在docker目录下搜索，缩小范围

这些镜像都是存储在Docker宿主机的/var/lib/docker目录下

3.1.2 搜索镜像

如果你需要从网络中查找需要的镜像，可以通过以下命令搜索

```
docker search 镜像名称
```

NAME：仓库名称

DESCRIPTION：镜像描述

STARS：用户评价，反应一个镜像的受欢迎程度

OFFICIAL：是否官方

AUTOMATED：自动构建，表示该镜像由Docker Hub自动构建流程创建的

3.1.3 拉取镜像

拉取镜像默认从中央仓库中下载镜像到本地

```
docker pull 镜像名称
```

docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]

例如，我要下载centos7镜像

```
docker pull centos:7
```

3.1.4 删除镜像

按镜像ID删除镜像

先停止容器，删除所有容器才可以删除镜像

学成day20pdf

```
docker rmi 镜像ID
```

删除所有镜像

docker image prune，删除none镜像

```
docker rmi `docker images -q`
```

3.2 容器相关命令

3.2.1 查看容器

查看日志 docker logs -f gitlab

查看正在运行的容器

```
docker ps
```

查看所有容器

```
docker ps -a
```

查看最后一次运行的容器

```
docker ps -l
```

查看停止的容器

```
docker ps -f status=exited
```

3.2.2 创建与启动容器

创建容器常用的参数说明：

创建容器命令：docker run

-i：表示运行容器

-t：表示容器启动后会进入其命令行。加入这两个参数后，容器创建就能登录进去。即分配一个伪终端。

--name :为创建的容器命名。

-v：表示目录映射关系（前者是宿主机目录，后者是映射到宿主机上的目录），可以使用多个－v做多个目录或文件映射。注意：最好做目录映射，在宿主机上做修改，然后共享到容器上。 目录不存在会自动给创建

-d：在run后面加上-d参数,则会创建一个守护式容器在后台运行（这样创建容器后不会自动登录容器，如果只加-i -t两个参数，创建后就会自动进去容器）。

-p：表示端口映射，前者是宿主机端口，后者是容器内的映射端口。可以使用多个-p做多个端口映射 通过宿主机的端口，来访问容器

（1）交互式方式创建容器

--restart=always

```
docker run -it --name=容器名称 镜像名称:标签 /bin/bash
```

这时我们通过ps命令查看，发现可以看到启动的容器，状态为启动状态

退出当前容器 退出自动关闭容器

```
exit
```

（2）守护式方式创建容器：

```
docker run -di --name=容器名称 镜像名称:标签
```

不指定标签默认为latest

也可以使用imageID

登录守护式容器方式：

```
docker exec -it 容器名称 (或者容器ID) /bin/bash
```

3.2.3 停止与启动容器

停止容器：

```
docker stop 容器名称（或者容器ID）
```

启动容器：

```
docker start 容器名称（或者容器ID）
```

3.2.4 文件拷贝

如果我们需要将文件拷贝到容器内可以使用cp命令

```
docker cp 需要拷贝的文件或目录 容器名称:容器目录
```

也可以将文件从容器内拷贝出来

```
docker cp mycentos2:/usr/local/anaconda-ks.cfg anaconda-ks2.cfg
```

```
docker cp 容器名称:容器目录 需要拷贝的文件或目录
```

3.2.5 目录挂载

我们可以在创建容器的时候，将宿主机的目录与容器内的目录进行映射，这样我们就可以通过修改宿主机某个目录的文件从而去影响容器。 创建容器 添加-v参数 后边为 宿主机目录:容器目录，例如：

```
docker run -di -v /usr/local/myhtml:/usr/local/myhtml --name=mycentos3 centos:7
```

如果你共享的是多级的目录，可能会出现权限不足的提示。

这是因为CentOS7中的安全模块selinux把权限禁掉了，我们需要添加参数 --privileged=true 来解决挂载的目录没有权限的问题

3.2.6 查看容器IP地址

我们可以通过以下命令查看容器运行的各种数据

```
docker inspect 容器名称（容器ID）
```

也可以直接执行下面的命令直接输出IP地址 提取所需要的信息

```
docker inspect --format='{{.NetworkSettings.IPAddress}}' 容器名称（容器ID）
```

3.2.7 删除容器

删除指定的容器： 先停止再删除

```
docker rm 容器名称（容器ID）
```

4 应用部署

4.1 MySQL部署

（1）拉取mysql镜像

```
docker pull centos/mysql-57-centos7
```

（2）创建容器

```
docker run -di --name=tensquare_mysql -p 33306:3306 -e MYSQL_ROOT_PASSWORD=123456 mysql
```

-p 代表端口映射，格式为 宿主机映射端口:容器运行端口

-e 代表添加环境变量 MYSQL*ROOT*PASSWORD 是root用户的登陆密码

（3）远程登录mysql　　通过宿主机对容器mysql进行访问

连接宿主机的IP ,指定端口为33306

容器在哪里呢？

4.2 tomcat部署

（1）拉取镜像

```
docker pull tomcat:7-jre7
```

（2）创建容器　　sftp 用put命令传入文件，全路径文件名

创建容器 -p表示地址映射

```
docker run -di --name=mytomcat -p 9000:8080 -v /usr/local/webapps:/usr/local/tomcat/webapps tomcat:7-jre7
```

容器内tomcat的路径

怎么知道的？

4.3 Nginx部署

（1）拉取镜像

找到nignx的根目录

```
docker pull nginx
```

（2）创建Nginx容器

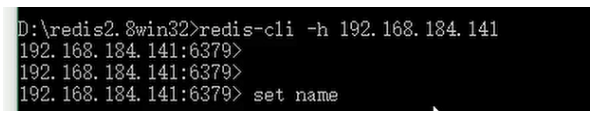
```
docker run -di --name=mynginx -p 80:80 nginx
```

4.4 Redis部署

（1）拉取镜像

通过主机的redis客户端远程连接，先进入目录下运行命令

```
docker pull redis
```



（2）创建容器

```
docker run -di --name=myredis -p 6379:6379 redis
```

5 迁移与备份

5.1 容器保存为镜像

我们可以通过以下命令将容器保存为镜像　　容器已经做过一些设置，包括导入一些东西，因此可以在容器的基础上保存为镜像，再创建容器

```
docker commit mynginx mynginx_i
```

容器名字　　新镜像名字

5.2 镜像备份

把镜像放在另外一个服务器上

我们可以通过以下命令将镜像保存为tar 文件

```
docker save -o mynginx.tar mynginx_i
```

镜像名字

5.3 镜像恢复与迁移

首先我们先删除掉mynginx_img镜像 然后执行此命令进行恢复

```
docker load -i mynginx.tar
```

-i 输入的文件

执行后再次查看镜像，可以看到镜像已经恢复

6 Dockerfile

功能类似于把设置好了的容器转化为镜像作为基础镜像

6.1 什么是Dockerfile

带操作系统的像centos这些　　加入了jdk，mysql这些

Dockerfile是由一系列命令和参数构成的脚本，这些命令应用于基础镜像并最终创建一个新的镜像。

1、对于开发人员：可以为开发团队提供一个完全一致的开发环境； 2、对于测试人员：可以直接拿开发时所构建的镜像或者通过Dockerfile文件构建一个新的镜像开始工作了； 3、对于运维人员：在部署时，可以实现应用的无缝移植。

6.2 常用命令

命令	作用
FROM image_name:tag	定义了使用哪个基础镜像启动构建流程　需要存在
MAINTAINER user_name	声明镜像的创建者
ENV key value	设置环境变量 (可以写多条)　　jdk的环境变量
RUN command	是Dockerfile的核心部分(可以写多条)
ADD source_dir/file dest_dir/file	将宿主机的文件复制到容器内，如果是一个压缩文件，将会在复制后自动解压
COPY source_dir/file dest_dir/file	和ADD相似，但是如果有压缩文件并不能解压
WORKDIR path_dir	设置工作目录　　所有命令都是相对于这个目录

6.3 使用脚本创建镜像

步骤：

（1）创建目录

```
mkdir -p /usr/local/dockerjdk8
```

（2）下载jdk-8u171-linux-x64.tar.gz并上传到服务器（虚拟机）中的/usr/local/dockerjdk8目录

（3）创建文件Dockerfile `vi Dockerfile`

输入以下内容

```
#依赖镜像名称和ID
FROM centos:7
#指定镜像创建者信息
MAINTAINER ITCAST
#切换工作目录
WORKDIR /usr
RUN mkdir  /usr/local/java
#ADD 是相对路径jar,把java添加到容器中
ADD jdk-8u171-linux-x64.tar.gz /usr/local/java/

#配置java环境变量
ENV JAVA_HOME /usr/local/java/jdk1.8.0_171
ENV JRE_HOME $JAVA_HOME/jre
ENV CLASSPATH $JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE_HOME/lib:$CLASSPATH
ENV PATH $JAVA_HOME/bin:$PATH
```


```
#依赖镜像名称和ID
FROM centos:7
#指定镜像创建者信息
MAINTAINER ITCAST
#切换工作目录
WORKDIR /usr
RUN mkdir  /usr/local/java
#ADD 是相对路径jar,把java添加到容器中
ADD jdk-8u171-linux-x64.tar.gz /usr/local/java/

#配置java环境变量
ENV JAVA_HOME /usr/local/java/jdk1.8.0_171
ENV JRE_HOME $JAVA_HOME/jre
ENV CLASSPATH $JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE_HOME/lib:$CLASSPATH
ENV PATH $JAVA_HOME/bin:$PATH
```


（4）执行命令构建镜像

```
docker build -t='jdk1.8' .
```

.指定dockerfile的目录

-t指定镜像的名称

注意后边的空格和点，不要省略

（5）查看镜像是否建立完成

```
docker images
```

7 Docker私有仓库

企业用的，比较方便

7.1 私有仓库搭建与配置

（1）拉取私有仓库镜像（此步省略）

```
docker pull registry
```

（2）启动私有仓库容器

```
docker run -di --name=registry -p 5000:5000 registry
```

（3）打开浏览器 输入地址http://192.168.72.139:5000/v2/_catalog
看到{"repositories":[]}并且内容为空,表示私有仓库搭建成功

（4）修改daemon.json　　让docker信任这个私有仓库，

```
vi /etc/docker/daemon.json
```

添加以下内容，保存退出。

```
{
"registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"],
"insecure-registries":["192.168.184.141:5000"]
}
```

```
json {"insecure-registries":["192.168.72.139:5000"]}
```

此步用于让 docker信任私有仓库地址

（5）重启docker 服务

client与Registry交互，为了安全，默认将采用https访问，但我们在搭建私有仓库时并未配置指定任何tls相关的key和crt文件，https访问定然失败。所以直接

```
systemctl restart docker
```

7.2 镜像上传至私有仓库

（1）标记此镜像为私有仓库的镜像

```
docker tag jdk1.8 192.168.72.139:5000/jdk1.8
```

（2）再次启动私服容器

```
docker start registry
```

（3）上传标记的镜像

```
docker push 192.168.72.139:5000/jdk1.8
```

（4）用curl查看仓库中的镜像

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

显示{"repositories":["ubuntu"]}

拉取私有仓库镜像时加上ip地址，有参数

4 push 到私有仓库
client与Registry交互，为了安全，默认将采用https访问, 但我们在搭建私有仓库时并未配置指定任何tls相关的key和crt文件，https访问定然失败。所以直接

```
1    docker push 10.255.1.25/istio-release/pilot:1.0.0
```

是无法将这个镜像push到私有仓库的。需要在docker的配置文件 `/etc/docker/daemon.json` （没有的话需要新建）中增加 “insecure-registries“参数配置

```
1  #启用不安全的注册表
2  {
3    "insecure-registries" : ["10.255.1.25"]
4  }
```

启用不安全的注册表后，Docker将执行以下步骤： 首先，尝试使用HTTPS。 如果HTTPS可用但证书无效，请忽略有关证书的错误。 如果HTTPS不可用，会回退到HTTP
重新加载配置文件，重启docker服务使配置生效

```
#systemctl daemon-reload
#systemctl resatrt docker
```

docker ps 默认的显示内容过多，当值过长时就会导致折行，可读性很差，所以希望只显示自己关心的某些列。

可以自己指定显示的模板，例如：

```
docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Names}}\t{{.Ports}}\t{{.Status}}" -a
```

```
docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Ports}}"
```

- table - 表示显示表头列名
- {{.ID}} - 容器ID
- {{.Command}} - 启动执行的命令

显示结果：

```
$ docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Ports}}"
CONTAINER ID        NAMES                                       PORTS
db3df460fe14        dev-peer0.org1.example.com-fabcar-1.0
b6f803814cce        cli
10724ca7364f        peer0.org1.example.com                      0.0.0.0:7051->7051/tcp, 0.0.0.0:7053
20d930e6e9f7        ca.example.com                              0.0.0.0:7054->7054/tcp
```

可用的占位符

名称	含义
.ID	容器ID
.Image	镜像ID
.Command	执行的命令
.CreatedAt	容器创建时间
.RunningFor	运行时长
.Ports	暴露的端口
.Status	容器状态
.Names	容器名称
.Label	分配给容器的所有标签
.Mounts	容器挂载的卷
.Networks	容器所用的网络名称

Docker 实战（二十七）Docker 容器之间的通信

最近在修改我以前写的 Docker 镜像，才发现我一直都没有把 Docker 用好，连 Docker 的容器之前如何通信都不知道。之前的做法是把不同的环境安装在一个 Docker 容器中，就不存在容器间通信的问题。但是 Docker 推荐的用法是一个 Docker 容器只运行一个进程，所以我将以前写的 Docker 镜像进行了重构。下面来总结下 Docker 容器之间的通信。

Docker 的网络模式

docker 目前支持以下 5 种网络模式：

docker run 创建 Docker 容器时，可以用 –net 选项指定容器的网络模式。

host 模式 ： 使用 –net=host 指定。与宿主机共享网络，此时容器没有使用网络的 namespace，宿主机的所有设备，如 Dbus 会暴露到容器中，因此存在安全隐患。

container 模式 ： 使用 –net=container:NAME_or_ID 指定。指定与某个容器实例共享网络。

none 模式 ： 使用 –net=none 指定。不设置网络，相当于容器内没有配置网卡，用户可以手动配置。

bridge 模式 ： 使用 –net=bridge 指定，默认设置。此时 docker 引擎会创建一个 veth 对，一端连接到容器实例并命名为 eth0，另一端连接到指定的网桥中（比如 docker0），因此同在一个主机的容器实例由于连接在同一个网桥中，它们能够互相通信。容器创建时还会自动创建一条 SNAT 规则，用于容器与外部通信时。如果用户使用了-p 或者-Pe 端口端口，还会创建对应的端口映射规则。

自定义模式 ： 使用自定义网络，可以使用 docker network create 创建，并且默认支持多种网络驱动，用户可以自由创建桥接网络或者 overlay 网络。

默认是桥接模式，网络地址为 172.17.0.0/16，同一主机的容器实例能够通信，但不能跨主机通信。

host 模式

如果启动容器的时候使用 host 模式，那么这个容器将不会获得一个独立的 Network Namespace，而是和宿主机共用一个 Network Namespace。容器将不会虚拟出自己的网卡，配置自己的 IP 等，而是使用宿主机的 IP 和端口。

container 模式

这个模式指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。

none 模式

这个模式和前两个不同。在这种模式下，Docker 容器拥有自己的 Network Namespace，但是，并不为 Docker 容器进行任何网络配置。也就是说，这个 Docker 容器没有网卡、IP、路由等信息。需要我们自己为 Docker 容器添加网卡、配置 IP 等。

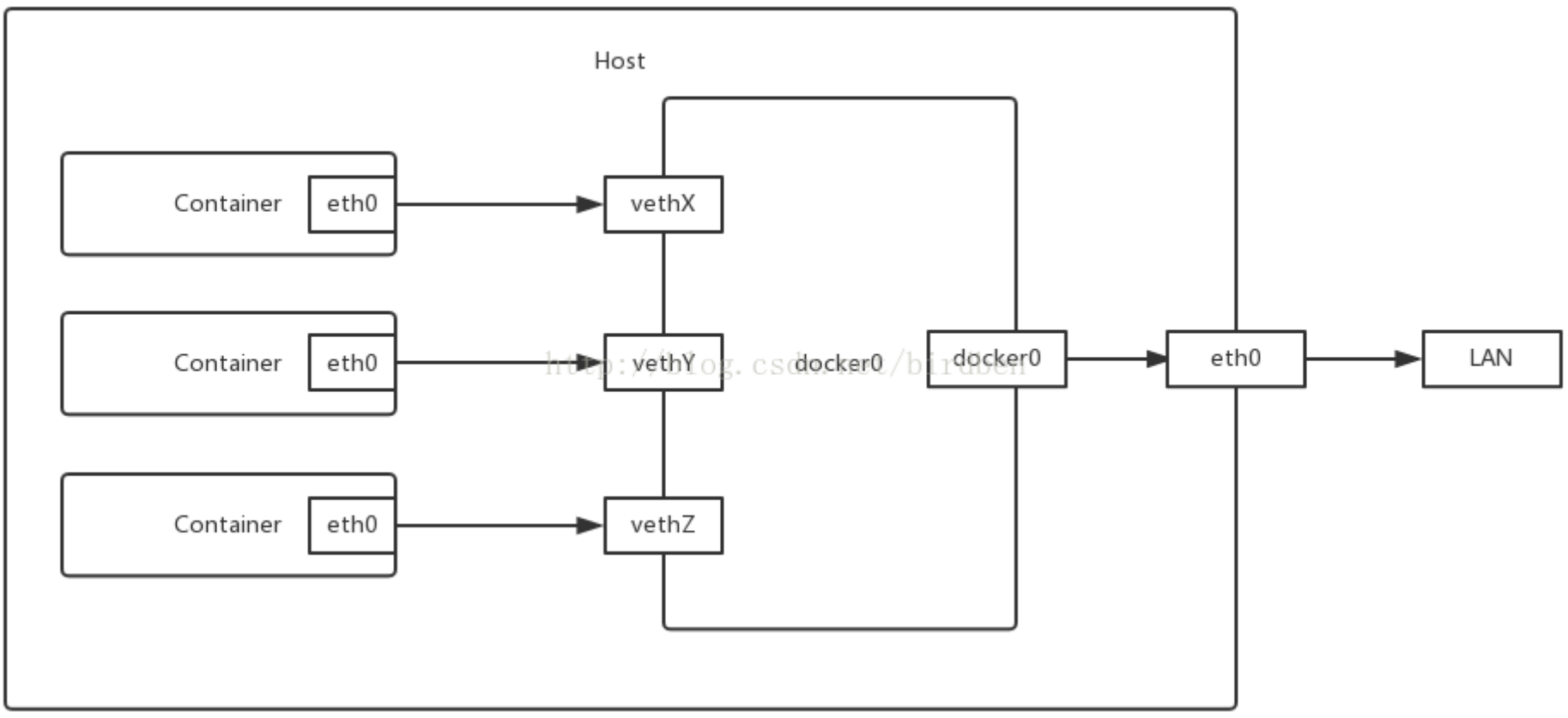
bridge 模式

bridge 模式是 Docker 默认的网络设置，此模式会为每一个容器分配 Network Namespace、设置 IP 等，并将一个主机上的 Docker 容器连接到一个虚拟网桥上。

当 Docker server 启动时，会在主机上创建一个名为 docker0 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。

接下来就要为容器分配 IP 了，Docker 会从 RFC1918 所定义的私有 IP 网段中，选择一个和宿主机不同的 IP 地址和子网分配给 docker0，连接到 docker0 的容器就从这个子网中选择一个未占用的 IP 使用。如一般 Docker 会使用 172.17.0.0/16 这个网段，并将 172.17.42.1/16 分配给 docker0 网桥（在主机上使用 ifconfig 命令是可以看到 docker0 的，可以认为它是网桥的管理接口，在宿主机上作为一块虚拟网卡使用）

当创建一个 Docker 容器的时候，同时会创建了一对 veth pair 接口（当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包）。这对接口一端在容器内，即 eth0；另一端在本地并被挂载到 docker0 网桥，名称以 veth 开头（例如 vethAQI2QT）。通过这种方式，主机可以跟容器通信，容器之间也可以相互通信。Docker 就创建了在主机和所有容器之间一个虚拟共享网络。



同主机不同容器之间通信

这里同主机不同容器之间通信主要使用 Docker 桥接（Bridge）模式。该 bridge 接口在本地一个单独的 Docker 宿主机上运行，并且它是我们后面提到的所有三种连接方式的背后机制。

```
$ ifconfig docker0
docker0     Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
            inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
            UP BROADCAST MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

连接方式

方式一：可以通过使用容器的 IP 地址来通信。这种方式会导致 IP 地址的硬编码，不方便迁移，并且容器重启后 IP 地址可能会改变，除非使用固定的 IP 地址。

方式二：可以通过宿主机的 IP 加上容器暴露出的端口号来通信。这种方式比较单一，只能依靠监听在暴露出的端口的进程来进行有限的通信。

方式三：可以使用容器名，通过 docker 的 link 机制通信。这种方式通过 docker 的 link 机制可以通过一个 name 来和另一个容器通信，link 机制方便了容器去发现其它的容器并且可以安全的传递一些连接信息给其它的容器。使用 name 给容器起一个别名，方便记忆和使用。即使容器重启了，地址发生了变化，不会影响两个容器之间的连接。

```
#  查看容器的内部 IP
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' $CONTAINER_ID
```

```
# Elasticsearch 容器
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' 4d5e7a1058de
172.17.0.2
# Kibana 容器
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' 4f26e64bfe82
172.17.0.4
```

方式一：使用容器的 IP 地址来通信

```
# 进入 Kibana 容器
$ docker exec -it 4f26e64bfe82 /bin/bash
# 在 Kibana 容器使用 ES 容器的 IP 地址来访问 ES 服务
$ curl -XGET 'http://172.17.0.2:9200/_cat/health?pretty'
1493707223 06:40:23 ben-es yellow 1 1 11 11 0 0 11 0 - 50.0%
```

方式二：使用宿主机的 IP 加上容器暴露出的端口号来通信

```
# 进入 Kibana 容器
$ docker exec -it 4f26e64bfe82 /bin/bash
# 在 Kibana 容器使用宿主机的 IP 地址来访问 ES 服务（我这里本机的 IP 地址是 10.10.1.129）
$ curl -XGET 'http://10.10.1.129:9200/_cat/health?pretty'
1493707223 06:40:23 ben-es yellow 1 1 11 11 0 0 11 0 - 50.0%
```

方式三：使用 docker 的 link 机制通信

```
# 先启动 ES 容器，并且使用--name 指定容器名称为：elasticsearch_2.x_yunyu
$ docker run -itd -p 9200:9200 -p 9300:9300 --name elasticsearch_2.x_yunyu birdben/elasticsearch_2.x:v2
# 启动 Kibana 容器，并且使用--link 指定关联的容器名称为 ES 的容器名称：elasticsearch_2.x_yunyu
$ docker run -itd -p 5601:5601 --link elasticsearch_2.x_yunyu --name kibana_4.x_yunyu birdben/kibana_4.x:v2
# 查看运行的容器
$ docker ps
CONTAINER ID        IMAGE                              COMMAND                  CREATED             STATUS              PORTS
4f26e64bfe82        birdben/kibana_4.x:v2              "docker-entrypoint..."   25 hours ago        Up 15 minutes       0.0.0.0:5601->5601/tcp
kibana_4.x_yunyu
4d5e7a1058de        birdben/elasticsearch_2.x:v2       "docker-entrypoint..."   26 hours ago        Up 19 hours         0.0.0.0:9200->9200/tcp,
0.0.0.0:9300->9300/tcp                                 elasticsearch_2.x_yunyu
# 在 Kibana 容器使用--link 的容器名称来访问 ES 服务
$ curl -XGET 'http://elasticsearch_2.x_yunyu:9200/_cat/health?pretty'
1493707223 06:40:23 ben-es yellow 1 1 11 11 0 0 11 0 - 50.0%
```

实际上–link 机制就是在 Docker 容器中的/etc/hosts 文件中添加了一个 ES 容器的名称解析。有了这个名称解析后就可以不使用 IP 来和目标容器通信了，除此之外当目标容器重启，Docker 会负责更新/etc/hosts 文件，因此可以不用担心容器重启后 IP 地址发生了改变，解析无法生效的问题。

Kibana 容器的/etc/hosts 文件

```
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      4d5e7a1058de
```

ES 容器的/etc/hosts 文件

```
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      elasticsearch_2.x_yunyu 4d5e7a1058de
172.17.0.4      4f26e64bfe82
```

当 docker 引入网络新特性后，link 机制变的有些多余，但是为了兼容早期版本，–link 机制在默认网络上的功能依旧没有发生变化，docker 引入网络新特性后，内置了一个 DNS Server，但是只有用户创建了自定义网络后，这个 DNS Server 才会起作用。

跨主机不同容器之间通信

（待续）

（待续）

参考文章：

https://jiajially.gitbooks.io/dockerguide/content/chapter_network_pro/index.html

https://opskumu.gitbooks.io/docker/content/chapter6.html

http://tonybai.com/2016/01/15/understanding-container-networking-on-single-host/

https://yq.aliyun.com/articles/55912

https://yq.aliyun.com/articles/30345?spm=5176.100239.blogcont40494.28.FnfzAV

http://int32bit.me/2016/05/10/Docker%E5%AE%9E%E7%8E%B0%E8%B7%A8%E4%B8%BB%E6%9C%BA%E9%80%9A%E4%BF%A1/

FAQ

挂载文件和目录是不一样的

挂载的文件修改权限可以传导