目录

[1技术类型 1](#_Toc529350559)

[2 Docker 1](#_Toc529350560)

# 1技术类型

1、Iaas(Infrastructure as a Service) 基础设施即服务，通过Internet向用户提供存储、数据库、cpu、网络、GPU等资源。

2、Saas(Software as a Service) 软件即服务，通过internet(云)向用户提供应用程序的服务。

3、PAAS平台即(Platform-as-a-Service：平台即服务)，把应用服务的运行和开发环境作为一种服务提供的商业模式

IaaS主要提供了虚拟计算、存储、数据库等基础设施服务，SaaS为用户提供了基于云的应用，PaaS则为开发人员提供了构建应用程序的环境。

# 2 Docker

## 1 基本概念

特性:

更高效的利用资源(不需要额外的虚拟化管理软件)

更简单的更新管理

更方便的迁移部署

更快速的启动

更容易扩展

传统虚拟化方式:

虚拟出所需的硬件资源之后，在其上运行一套完整的操作系统，然后在该系统上运行所需的应用程序。



Docker: 容器虚拟化

不需要虚拟出硬件资源，容器的应用程序直接运行于宿主内核，容器内没自己的内核，更轻便。



|  |  |  |
| --- | --- | --- |
| 特性 | 容器 | 虚拟机 |
| 启动 | 秒级 | 分钟级 |
| 硬盘使用 | MB | GB |
| 性能 | 接近原生 | 若于原生 |
| 系统支持量 | 几千个容器 | 几十个虚拟机 |

## 2 三个概念

镜像images： 镜像包含操作系统完整的root文件系统。

分层存储框架。镜像不像ISO打包文件那样，镜像是一组文件。

容器

仓库

## 3.基本指令

拉取镜像: docker pull image\_name

查看已有的镜像:docker images

查看帮助:docker help

查看镜像列表:docker search nginx

创建并启动一个容器: docker run –name=container\_name images

启动一个容器:docker start container\_name/container\_id（已经创建的容器）

进入一个容器:docker attach container\_name

停止容器:docker stop container\_id

退出:ctrl+p

导出镜像: docker save –o image\_name.tar image\_name

删除镜像:docker rmi image\_name

基于创建好的容器自定义镜像:docker commit –m “con\_name” –a”author” con\_id image\_name

创建一个容器的同时进入这个容器 docker run –it –name=con\_name images

放入后台运行: docker run –d –p image\_name

启动一个容器:基于一个镜像新建并启动；启动已存在的终止的容器

载入镜像:sudo docker load --input ubuntu14.04.tar

查看docker 容器的日志:docker logs container\_name/container\_id

-f 跟踪实时日志 –t –since= –until

--tail=xxx 从日志末尾显示xxx行的日志

查看docker 的网络: docker network ls

创建一个docker网络: docker network create –d bridge

--subnet=192.168.0.0/24

--gateway=192.168.0.120

-- ip-range=192.168.0.0/24 my-docker

运行dockerfile并给dockerfile创建的镜像建立名字: docker build –t mysql:3.5.34 ‘pwd’

docker build –t second:v1.0 .注意最后有个点，代表使用当前路径的Dockerfile进行构件

-t 指定镜像名称 -f 指定Dockerfile的路径

Dockerfile 是一个文本格式的配置文件。

Dockerfile分为四个部分:基础镜像信息、维护者信息、镜像操作指令和容器启动指令

FROM 基础镜像信息 可以使用多个FROM，如果在同一个Dockerfile中创建多个镜像，可以使用多个FROM（每个镜像一次）

FORM<image> FROM Ubuntu:trusty

#MAINTAINER 指定维护者信息

#MAINTAINER username<xxx.com>

#ENV指定一个环境变量，会被后续RUN指令使用，并在容器运行时保持。

#ENV <key> <value>

ENV DEBIAN\_FROMEND noninteractive（设置环境变量）

#RUN 镜像的操作指令 image支持的命令

#ADD ADD [source directory or URL] [destination directory]

它的基本作用是从源系统的文件系统上复制文件到目标容器的文件系统。可以用“\”换行

复制指定的<src>到容器的<dest>中，<src>可以是Dockerfile所在的目录的一个相对路径；可以是URL，也可以是tar.gz（自动解压）

#COPY

复制本地主机的 <src> （ 为 Dockerfile 所在目录的相对路径）到容器中的 <dest> （当使用本地目录为源目录时，推荐使用 COPY）

#EXPOSE

格式：EXPOSE <port>  [ <port> ...]

告诉Docker服务端暴露端口，在容器启动时需要通过 -p 做端口映射

#WORKDIR

格式：WORKDIR /path/to/workdir

为后续的 RUN 、 CMD 、 ENTRYPOINT 指令配置工作目录。（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）

#CMD 设置容器启动时执行的操作 只能存在一条(多个CMD命令存在的话只执行最后一个CMD 命令，因此只需要一个CMD命令)，调用方式有三种

# 方式一，运行一个可执行文件，并提供参数(like an exec, this is the preferred form)

CMD ["executable","param1","param2"]

# 方式二，利用”/bin/sh -c”去执行， (as a shell)

CMD command param1 param2

# 方式三，作为ENTRYPOINT的默认参数

CMD ["param1", "param2"]

在使用docker run imagename command新建并启动容器的时候，command会替换dockerfile里的CMD命令，如上面我们创建的docker镜像，如果后面输入了hello docker，则不会输出hello world了，本来dockerfile里面指定了输出hello world。

$ docker run hello\_docker echo "hello docker"

hello docker

$ docker run hello\_docker

hello world

该操作可以是执行自定义脚本，也可以是执行系统命令。

#ENTRYPOINT 功能和CMD 一样，区别在于ENTRYPOINT后面的携带的参数不会被docker run 提供的参数覆盖，而CMD 会被覆盖

CMD指令指定的容器启动时命令可以被docker run指定的命令覆盖，而ENTRYPOINT指令指定的命令不能被覆盖，而是将docker run指定的参数当做ENTRYPOINT指定命令的参数。

docker run –d –P training/webapp python app.py

-d 后台运行

-P 将容器内部使用的网络端口映射到使用的主机上

-p 5000:5000

docker run –d –p 127.0.0.1:5000:5000 training/webapp python app.py

网络端口的快捷方式

docker port 可以查看某个容器的确定端口映射到主机上的端口号

Dockerfile 创建ubuntu的新用户，并以新用户登录。

RUN useradd --create-home --no-log-init --shell /bin/bash mynewuser

RUN adduser mynewuser sudoRUN echo 'mynewuser:mynewpassword' | chpasswd

USER mynewuser

WORKDIR /home/mynewuser

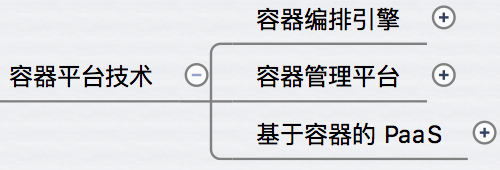
## 4.docker 理论

### 4.1Docker的 技术组成

容器生态系统:容器核心技术、容器平台技术、容器支持技术。

容器核心技术:让container在host上运行起来

容器平台技术:让容器作为集群在分布式环境中运行



容器编排引擎 kubernetes docker swarm

基于容器的应用一般会采用微服务架构。在这种架构下，应用被划分为不同的组件，并以服务的形式运行在各自的容器中，通过 API 对外提供服务。为了保证应用的高可用，每个组件都可能会运行多个相同的容器。这些容器会组成集群，集群中的容器会根据业务需要被动态地创建、迁移和销毁。

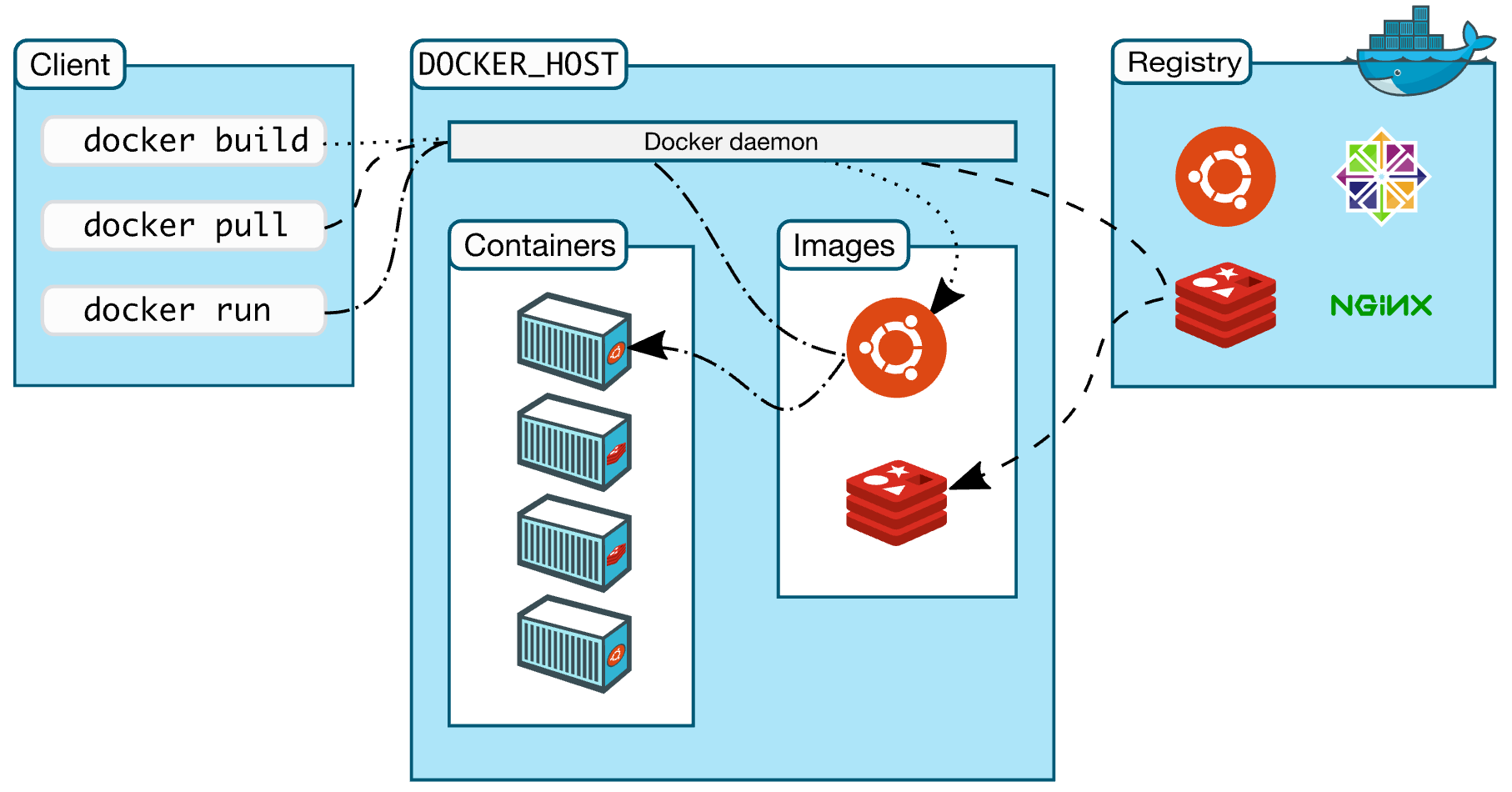
大家可以看到，这样一个基于微服务架构的应用系统实际上是一个动态的可伸缩的系统。这对我们的部署环境提出了新的要求，我们需要有一种高效的方法来管理容器集群。而这，就是容器编排引擎要干的工作。

所谓编排（orchestration），通常包括容器管理、调度、集群定义和服务发现等。通过容器编排引擎，容器被有机的组合成微服务应用，实现业务需求。

Deis、Flynn 和 Dokku 都是开源容器 PaaS 的代表。

容器支持的技术:网络、数据、服务状态等

### 4.2 Docker的基本框架图



### 5 docker数据管理

5.1 数据卷 Data Volumes 容器内的数据直接映射到localhost

数据卷可以在容器之间共享和重用，容器间的传递数据将会变得更高效和方便

数据卷会一直存在，直到没有容器使用后可以安全的删除

数据卷更新不影响镜像

数据卷的修改会立马生效，在容器内和本地

创建数据卷: docker volume create –d local test

绑定数据卷

Sudo docker run –d –P –name web –v /webapp:/opt/webapp training/webapp python app.py

5.2 数据卷容器 Data Volumes Containers

### 6 web服务与应用

Web协议基础

Tcp 三次握手，四次挥手

TCP可以看成一种字节流，他会处理IP层或以下的层的丢包、重复、按需到达等问题。是面向链接的协议。TCP提供可靠的、面向连接的服务。

序列号的作用是使得一个TCP接收端可丢弃重复的报文段，记录以杂乱次序到达的报文段。因为TCP使用IP来传输报文段，而IP不提供重复消除或者保证次序正确的功能。另一方面，TCP是一个字节流协议，绝不会以杂乱的次序给上层程序发送数据。因此TCP接收端会被迫先保持大序列号的数据不交给应用程序，直到缺失的小序列号的报文段被填满。

头部长度以32位字为单位，也就是以4bytes为单位，它只有4位，最大为15，因此头部最大长度为60字节，而其最小为5，也就是头部最小为20字节（可变选项为空）。

SYN位字段会消耗一个序列号，这意味着使用重传进行可靠传输。而不消耗序列号的ACK则不是。

Apache

7 docker 分层技术

# 3 openStack

## 3.1 OpenStack 基本介绍

组件：

1 Horizon 控制台 web展示界面操作平台

2 Nove 计算 负责创建，调度，销毁云主机

3 Neutron 网络 负责实现SDN

4 Swift 对象存储 结构化存储数据

5 Cinder 块存储 提供持久化块存储，为云主机提供附加云盘

6 Glance 镜像 提供镜像服务 装机使用

7 Keystone 认证

8 Ceilometer 计费

KVM(Kernel-based Virtual Machine) 开放虚拟化技术

是一个开源的系统虚拟化模块，需要硬件支持。

# K8s

## 1、Node

Node作为集群中的工作节点，运行真正的应用程序，在Node上Kubernetes管理的最小运行单元是Pod。Node上运行着Kubernetes的Kubelet、kube-proxy服务进程，这些服务进程负责Pod的创建、启动、监控、重启、销毁、以及实现软件模式的负载均衡。

Node包含的信息：

Node地址：主机的IP地址，或Node ID。

Node的运行状态：Pending、Running、Terminated三种状态。

Node Condition：…

Node系统容量：描述Node可用的系统资源，包括CPU、内存、最大可调度Pod数量等。

其他：内核版本号、Kubernetes版本等。

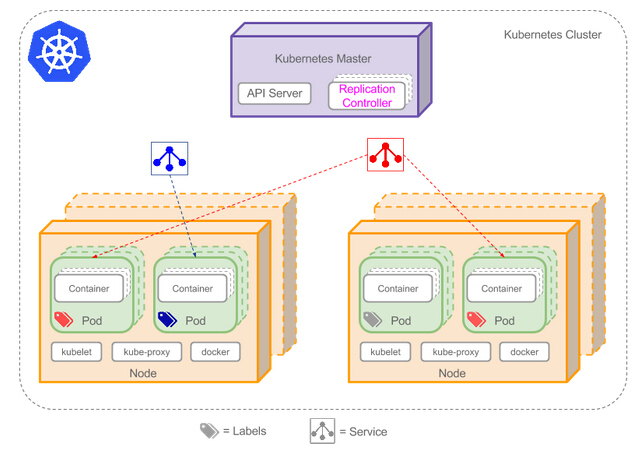
查看Node信息：

kubectl describe node

## 2、Pod

Pod是Kubernetes最基本的操作单元，包含一个或多个紧密相关的容器，一个Pod可以被一个容器化的环境看作应用层的“逻辑宿主机”；一个Pod中的多个容器应用通常是紧密耦合的，Pod在Node上被创建、启动或者销毁；每个Pod里运行着一个特殊的被称之为Pause的容器，其他容器则为业务容器，这些业务容器共享Pause容器的网络栈和Volume挂载卷，因此他们之间通信和数据交换更为高效，在设计时我们可以充分利用这一特性将一组密切相关的服务进程放入同一个Pod中。

同一个Pod里的容器之间仅需通过localhost就能互相通信。



一个Pod中的应用容器共享同一组资源：

PID命名空间：Pod中的不同应用程序可以看到其他应用程序的进程ID；

网络命名空间：Pod中的多个容器能够访问同一个IP和端口范围；

IPC命名空间：Pod中的多个容器能够使用SystemV IPC或POSIX消息队列进行通信；

UTS命名空间：Pod中的多个容器共享一个主机名；

Volumes（共享存储卷）：Pod中的各个容器可以访问在Pod级别定义的Volumes；

Pod的生命周期通过Replication Controller来管理；通过模板进行定义，然后分配到一个Node上运行，在Pod所包含容器运行结束后，Pod结束。

Kubernetes为Pod设计了一套独特的网络配置，包括：为每个Pod分配一个IP地址，使用Pod名作为容器间通信的主机名等。

## 3、Service

在Kubernetes的世界里，虽然每个Pod都会被分配一个单独的IP地址，但这个IP地址会随着Pod的销毁而消失，这就引出一个问题：如果有一组Pod组成一个集群来提供服务，那么如何来访问它呢？**Service！**

一个Service可以看作一组提供相同服务的Pod的对外访问接口，Service作用于哪些Pod是通过Label Selector来定义的。

* 拥有一个指定的名字（比如my-mysql-server）；
* 拥有一个虚拟IP（Cluster IP、Service IP或VIP）和端口号，销毁之前不会改变，只能内网访问；
* 能够提供某种远程服务能力；
* 被映射到了提供这种服务能力的一组容器应用上；

如果Service要提供外网服务，需指定公共IP和NodePort，或外部负载均衡器；

NodePort   
系统会在Kubernetes集群中的每个Node上打开一个主机的真实端口，这样，能够访问Node的客户端就能通过这个端口访问到内部的Service了

## 4、Volume

Volume是Pod中能够被多个容器访问的共享目录。

## 5、Label

Label以key/value的形式附加到各种对象上，如Pod、Service、RC、Node等，以识别这些对象，管理关联关系等，如Service和Pod的关联关系。

## 6、RC（Replication Controller）

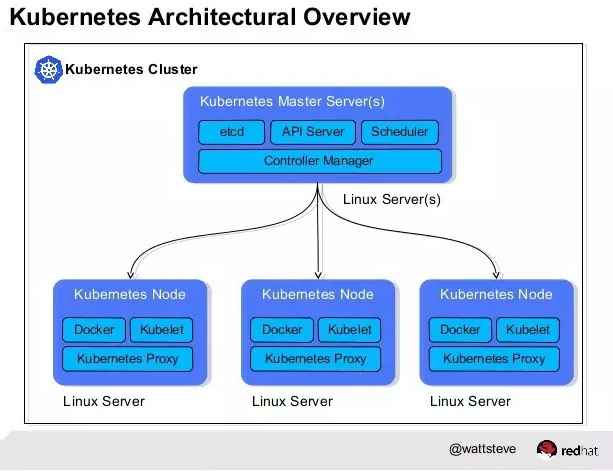
* 目标Pod的定义；
* 目标Pod需要运行的副本数量；
* 要监控的目标Pod标签（Lable）；

Kubernetes通过RC中定义的Lable筛选出对应的Pod实例，并实时监控其状态和数量，如果实例数量少于定义的副本数量（Replicas），则会根据RC中定义的Pod模板来创建一个新的Pod，然后将此Pod调度到合适的Node上启动运行，直到Pod实例数量达到预定目标。

## 二、Kubernetes总体架构

### Master和Node

Kubernetes将集群中的机器划分为一个Master节点和一群工作节点（Node）。其中，Master节点上运行着集群管理相关的一组进程**etcd、API Server、Controller Manager、Scheduler**，后三个组件构成了Kubernetes的总控中心，这些进程实现了整个集群的资源管理、Pod调度、弹性伸缩、安全控制、系统监控和纠错等管理功能，并且全都是自动完成。在每个Node上运行**Kubelet、Proxy、**[**Docker**](http://lib.csdn.net/base/docker)**daemon**三个组件，负责对本节点上的Pod的生命周期进行管理，以及实现服务代理的功能。



**流程**   
通过Kubectl提交一个创建RC的请求，该请求通过API Server被写入etcd中，此时Controller Manager通过API Server的监听资源变化的接口监听到这个RC事件，分析之后，发现当前集群中还没有它所对应的Pod实例，于是根据RC里的Pod模板定义生成一个Pod对象，通过API Server写入etcd，接下来，此事件被Scheduler发现，它立即执行一个复杂的调度流程，为这个新Pod选定一个落户的Node，然后通过API Server讲这一结果写入到etcd中，随后，目标Node上运行的Kubelet进程通过API Server监测到这个“新生的”Pod，并按照它的定义，启动该Pod并任劳任怨地负责它的下半生，直到Pod的生命结束。

随后，我们通过Kubectl提交一个新的映射到该Pod的Service的创建请求，Controller Manager会通过Label标签查询到相关联的Pod实例，然后生成Service的Endpoints信息，并通过API Server写入到etcd中，接下来，所有Node上运行的Proxy进程通过API Server查询并监听Service对象与其对应的Endpoints信息，建立一个软件方式的负载均衡器来实现Service访问到后端Pod的流量转发功能。

* etcd   
  用于持久化存储集群中所有的资源对象，如Node、Service、Pod、RC、Namespace等；API Server提供了操作etcd的封装接口API，这些API基本上都是集群中资源对象的增删改查及监听资源变化的接口。
* API Server   
  提供了资源对象的唯一操作入口，其他所有组件都必须通过它提供的API来操作资源数据，通过对相关的资源数据“全量查询”+“变化监听”，这些组件可以很“实时”地完成相关的业务功能。
* Controller Manager   
  集群内部的管理控制中心，其主要目的是实现Kubernetes集群的故障检测和恢复的自动化工作，比如根据RC的定义完成Pod的复制或移除，以确保Pod实例数符合RC副本的定义；根据Service与Pod的管理关系，完成服务的Endpoints对象的创建和更新；其他诸如Node的发现、管理和状态监控、死亡容器所占磁盘空间及本地缓存的镜像文件的清理等工作也是由Controller Manager完成的。
* Scheduler   
  集群中的调度器，负责Pod在集群节点中的调度分配。
* Kubelet   
  负责本Node节点上的Pod的创建、修改、监控、删除等全生命周期管理，同时Kubelet定时“上报”本Node的状态信息到API Server里。
* Proxy   
  实现了Service的代理与软件模式的负载均衡器。

客户端通过Kubectl命令行工具或Kubectl Proxy来访问Kubernetes系统，在Kubernetes集群内部的客户端可以直接使用Kuberctl命令管理集群。Kubectl Proxy是API Server的一个反向代理，在Kubernetes集群外部的客户端可以通过Kubernetes Proxy来访问API Server。

API Server内部有一套完备的安全机制，包括认证、授权和准入控制等相关模块。

# 附录

## Dockerfile 模板

FROM centos

MAINTAINER yaolin

COPY jdk1.7.0\_79 jdk1.7.0\_79

ADD websocket.jar app.jar

ENV JAVA\_HOME=/jdk1.7.0\_79

ENV PATH=$JAVA\_HOME/bin:$PATH

ENV CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

EXPOSE 8080

ENTRYPOINT ["java","-jar","/app.jar"]