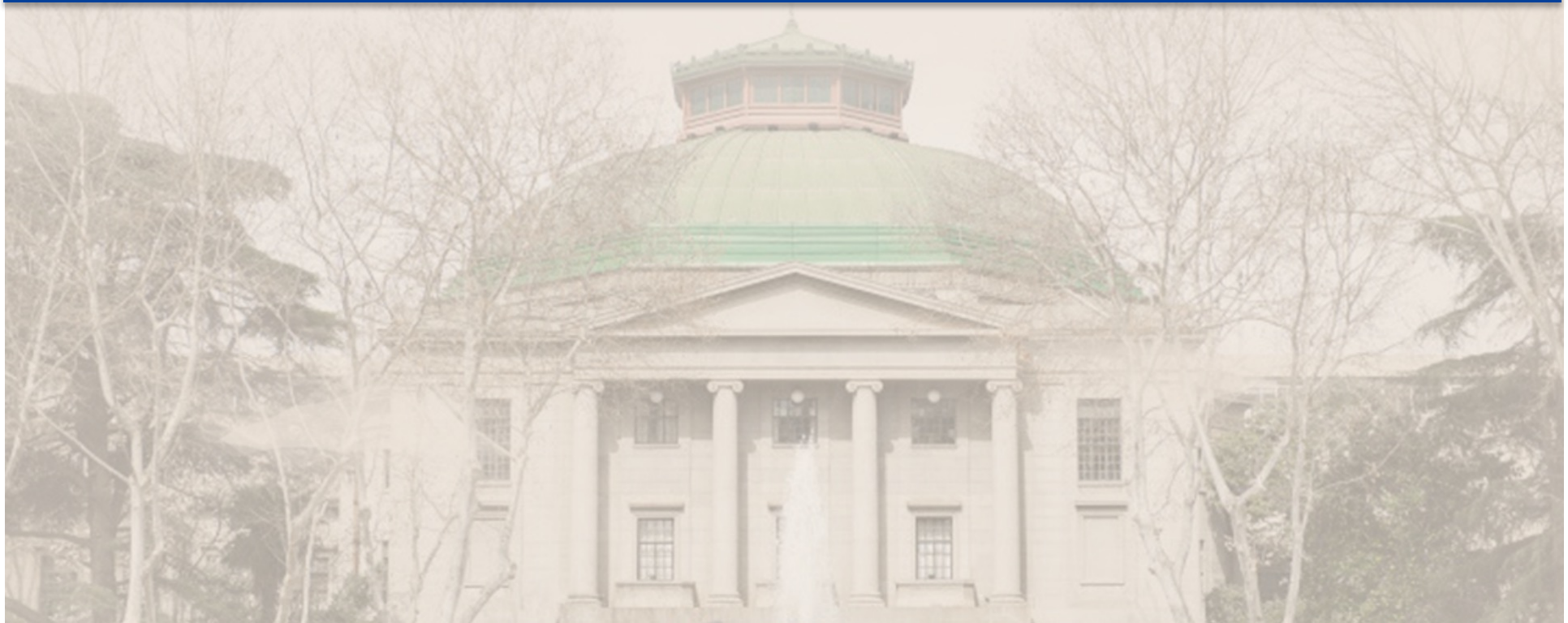
“云原生”与容器基本概念



1 为什么要开设云原生技术公开课?

云原生技术发展简史

2004~2007

Google 大规模使用
容器（cgroups）技术

2008.01

cgroups 合并进入
Linux 内核主干

2013.03

Docker 项目正式发布

2014.06

Kubernetes 项目发布

2015.07

CNCF （云原生基金会）成立
22 个创始成员
Kubernetes 成为第一个 CNCF 项目

2017.07

CNCF 成立2周年
170个成员
14个基金会项目

2018.07

CNCF 成立3周年
195个成员
19个基金会项目
11个孵化项目

我们正处于时代的关键节点

2013 年，Docker 项目发布

使得全操作系统语义的沙盒技术唾手可得，对传统 PaaS 产业“降维打击”

2014 年，Kubernetes 项目发布

Google Borg/Omega 系统思想借助开源社区“重生”，“容器设计模式”的思想正式确立。为什么选择开源?

2015~2016 年，容器编排“三国争霸”

Docker Swarm，Mesos，Kubernetes 在容器编排领域展开角逐。为什么要竞争？各自优势为何?

2017 年，Kubernetes 项目事实标准确立

Docker 公司宣布在核心产品内置 Kubernetes 服务，Swarm 项目逐渐停止维护。原因为何?

2018 年，云原生技术理念逐步萌芽

Kubernetes 和容器成为所有云厂商上的既定标准，以“云”为核心的软件研发思想逐步形成

2019？

2019 年 – 云原生技术普及元年

腾讯、阿里、华为、字节全面上云

以“云”为核心的软件研发思想，逐步成为默认选项

Kubernetes 等云原生技能成为技术人员必修课，大量工作岗位涌现

“会 Kubernetes”已经远远不够，“懂 Kubernetes”、“会云原生架构”的重要性日益凸显

1 容器与镜像

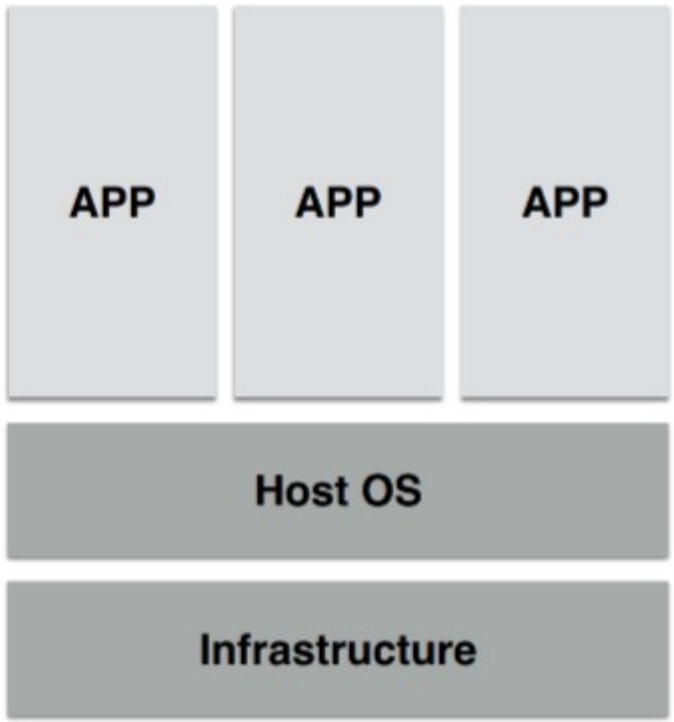
2 容器生命周期

3 容器项目的架构

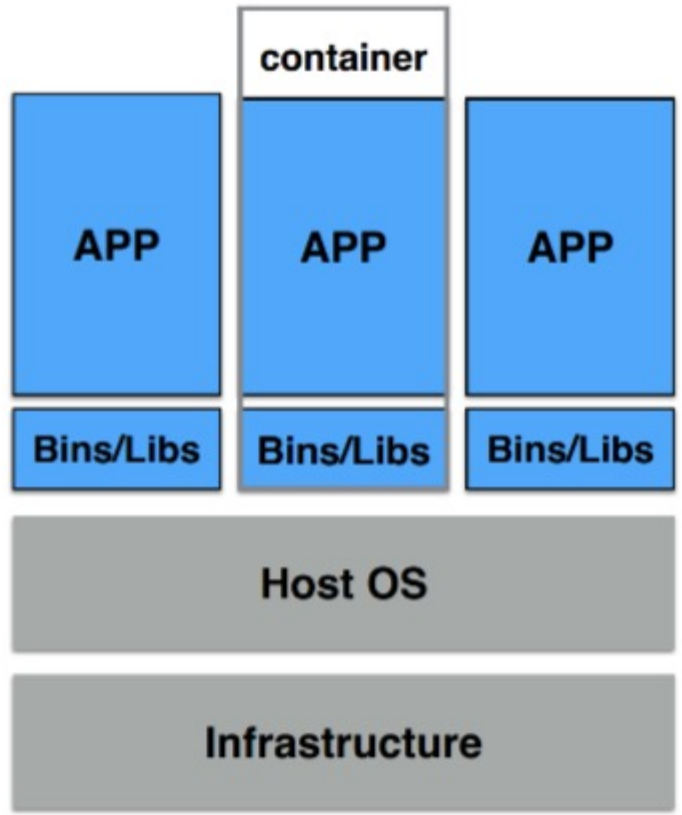
4 容器 vs VM

5 课后实践

什么是容器?



* 进程可见、可相互通信
* 共享同一份文件系统



* 资源视图隔离 – namespace
* 控制资源使用率 – cgroup
* 独立的文件系统 – chroot

什么是容器？

容器，是一个视图隔离、资源可限制、独立文件系统的进程集合。

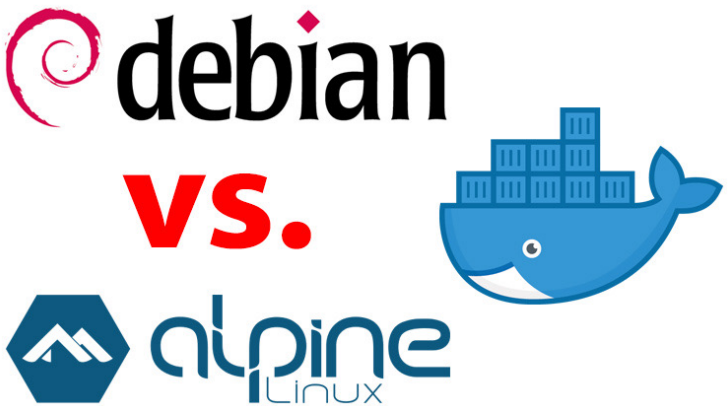
* 视图隔离 – 如能看见部分进程；独立主机名 等等；
* 控制资源使用率 – 如 2G 内存大小；CPU 使用个数 等等；

什么是镜像？

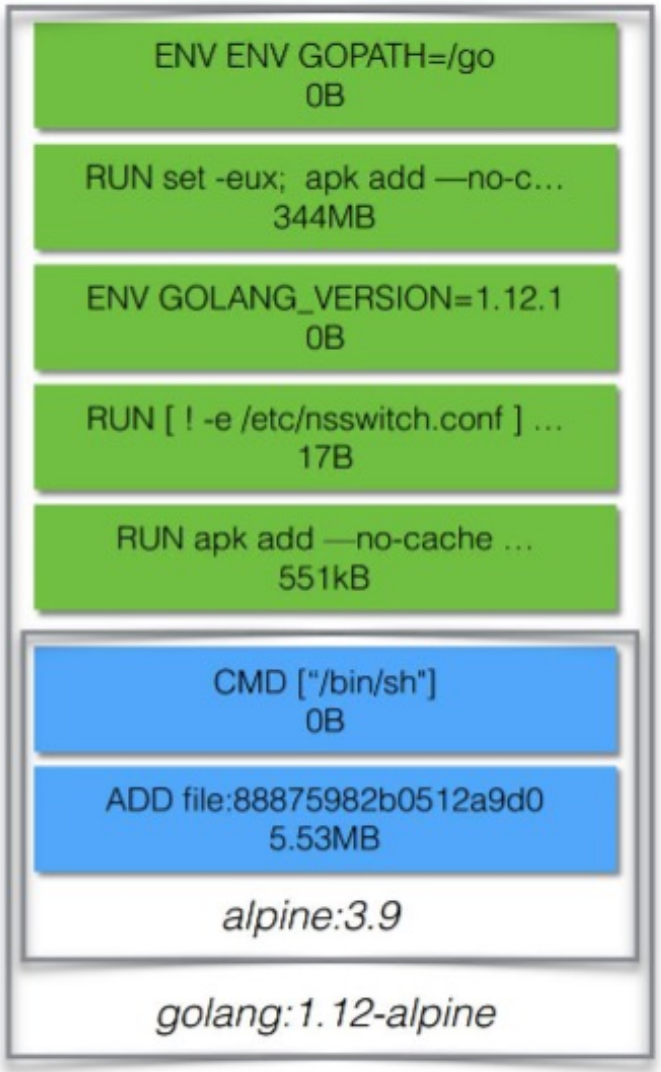
运行容器所需要的所有文件集合 – 容器镜像

Dockerfile – 描述镜像构建步骤

构建步骤所产生出文件系统的变化 – changeset
★类似 disk snapshot
★提高分发效率，减少磁盘压力



Benchmarking Debian vs Alpine as a Base Docker Image



如何构建镜像?

编写 Dockerfile – app:v1

$ docker build . –t app:v1

docker registry – 镜像数据的存储和分发

$ docker push app:v1

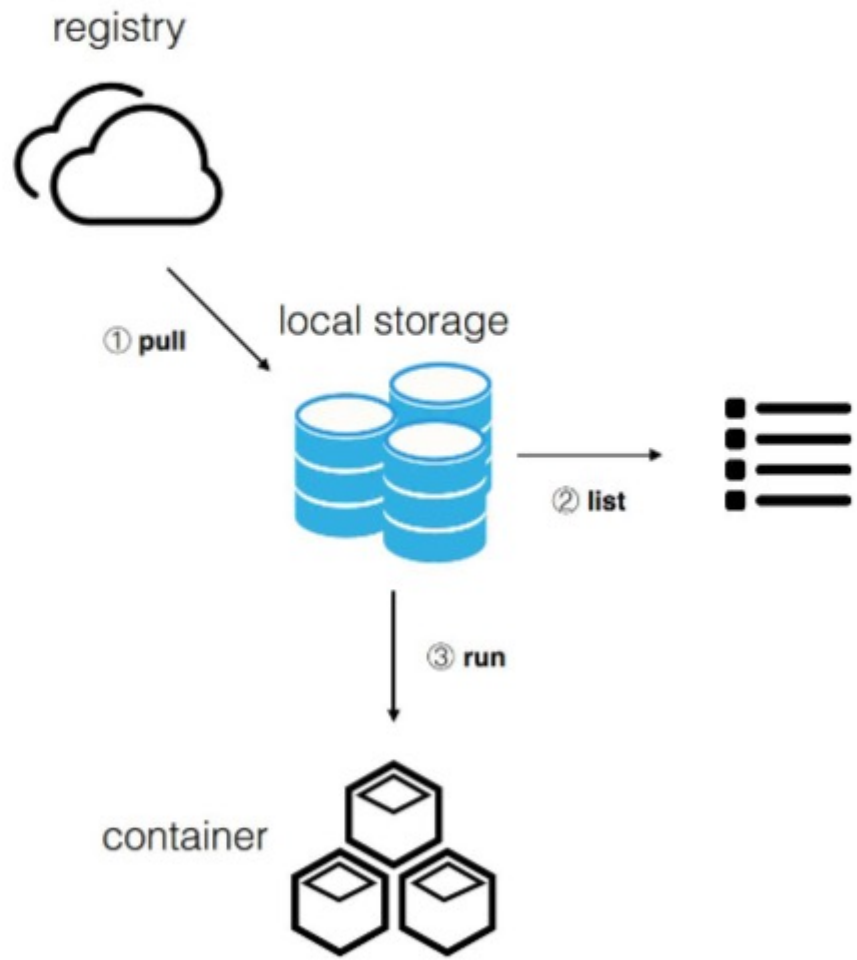
```
1  # base on golang:1.12-alpine image
2  FROM golang:1.12-alpine
3
4  # setting current working dir (PWD -> /go/src/app)
5  WORKDIR /go/src/app
6
7  # copy local files into /go/src/app
8  COPY . .
9
10 # get all the dependencies
11 RUN go get -d -v ./...
12
13 # build the application and install it
14 RUN go install -v ./...
15
16 # by default, run the app
17 CMD ["app"]
```

如何运行容器?

① 从 docker registry 下载镜像 – docker pull busybox:1.25

② 查看本地镜像列表 – docker images

③ 选择相应的镜像并运行 – docker run [–d] —name demo busybox:1.25 top



小节

容器 – 和系统其他部分隔离开的进程集合

镜像 – 容器所需要的所有文件集合 – Build once, Run anywhere

1 容器与镜像

2 容器生命周期

3 容器项目的架构

4 容器 vs VM

5 课后实践

2 容器生命周期

容器运行时的生命周期

单进程模型

① Init 进程生命周期 = 容器生命周期
② 运行期间可运行 exec 执行运维操作

数据持久化

① 独立于容器的生命周期
② 数据卷 – docker volume vs bind

```
# bind host dir into container
$ docker run -v /tmp:/tmp busybox:1.25 sh -c "date > /tmp/demo.log"

# check result
$ cat /tmp/demo.log
Tue Apr  9 02:17:55 UTC 2019

# let it handled by docker container engine
$ docker create volume demo

# demo is volume name
$ docker run -v demo:/tmp busybox:1.25 sh -c "date > /tmp/demo.log"

# check result
$ docker run -v demo:/tmp busybox:1.25 sh -c "cat /tmp/demo.log"
Tue Apr  9 02:19:57 UTC 2019
```

1 容器与镜像

2 容器生命周期

3 容器项目的架构

4 容器 vs VM

5 课后实践

3 容器项目的架构

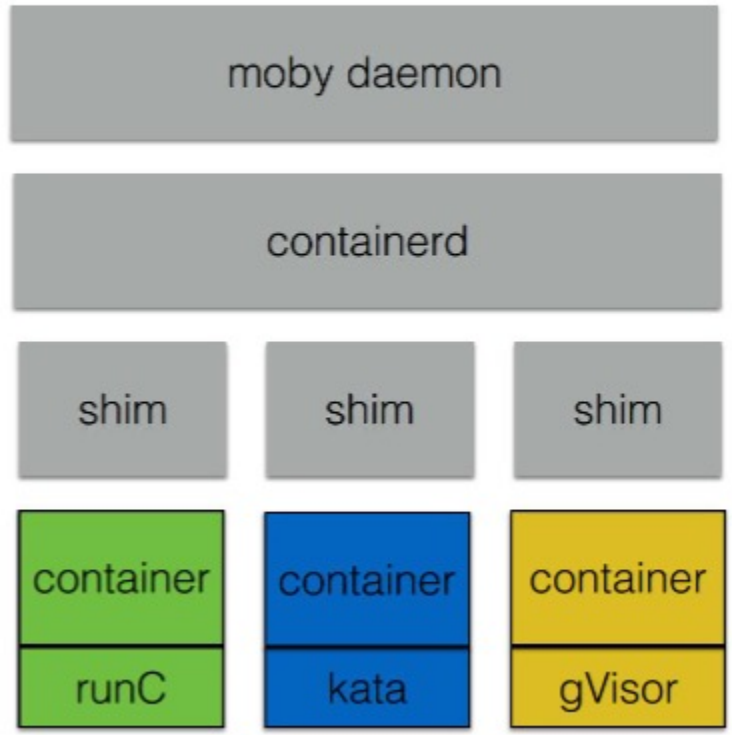
moby 容器引擎架构

containerd

① 容器运行时管理引擎，独立于 moby daemon
② containerd-shim 管理容器生命周期，可被 containerd 动态接管

容器运行时

① 容器虚拟化技术方案
② runC kata gVisor



1 容器与镜像

2 容器生命周期

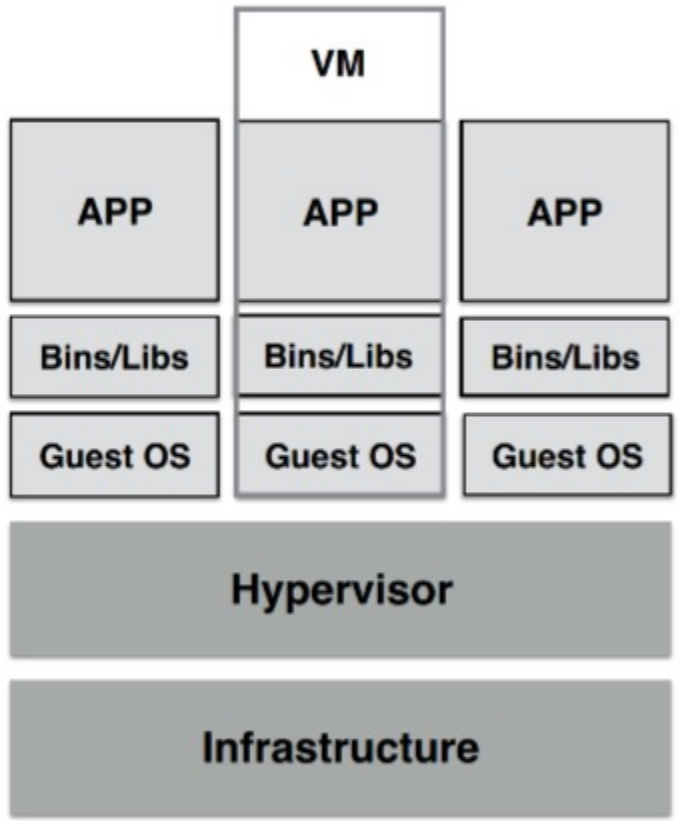
3 容器项目的架构

4 容器 vs VM

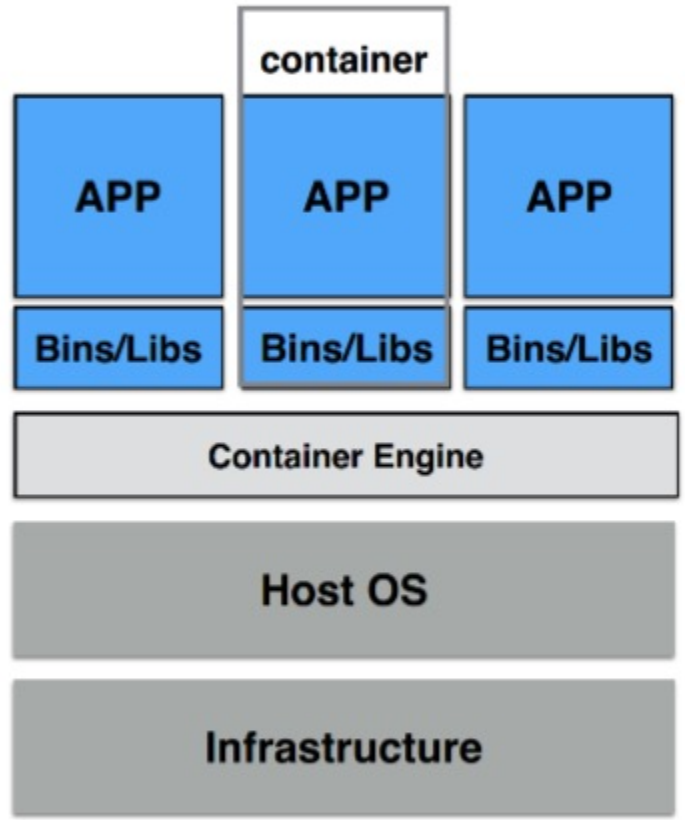
5 课后实践

4 容器 vs VM

容器和VM之间的差异



① 模拟硬件资源，需要 Guest OS
② 应用拥有 Guest OS 所有资源
③ 更好地隔离效果 – Hypervisor 需要消耗更多地资源



① 无 Guest OS，进程级别的隔离
② 启动时间更快
③ 隔离消耗资源少 – 隔离效果弱于 VM

1 容器与镜像

2 容器生命周期

3 容器项目的架构

4 容器 vs VM

5 课后实践

举例： WAR 包 + Tomcat 的容器化

- 方法一：把WAR包和Tomcat打包进一个镜像
  - 无论是WAR 包和 Tomcat 更新都需要重新制作镜像
- 方法二：镜像里只打包Tomcat。使用数据卷（hostPath）从宿主机上将WAR包挂载进Tomcat容器
  - 需要维护一套分布式存储系统
- 有没有更通用的方法?

打包制作docker镜像

- 打包 WAR 包和 Tomcat (示例的war包下载：https://www.middlewareinventory.com/blog/sample-web-application-war-file-download/#Code_and_Explanation)
- 创建 Dockerfile 文件并构建镜像
- 运行构建镜像：docker run
- 此时已经成功将 WAR 包部署到 Tomcat

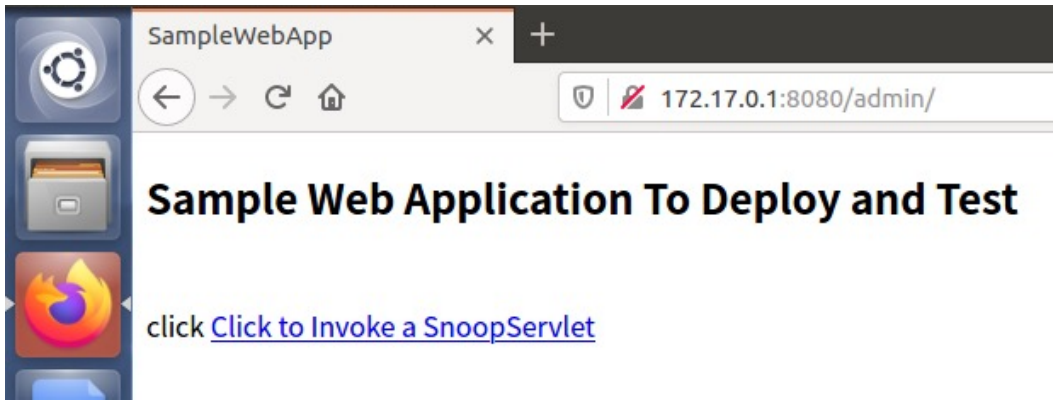
```
# 先创建一个文件夹为docker-admin
mkdir docker-admin

# 进入文件夹docker-admin 并创建一个Dockerfile
cd docker-admin && vim Dockerfile
```

```
seucyber@localhost: ~/docker-admin
FROM docker.io/tomcat
MAINTAINER rstyro
COPY admin.war /usr/local/tomcat/webapps
```

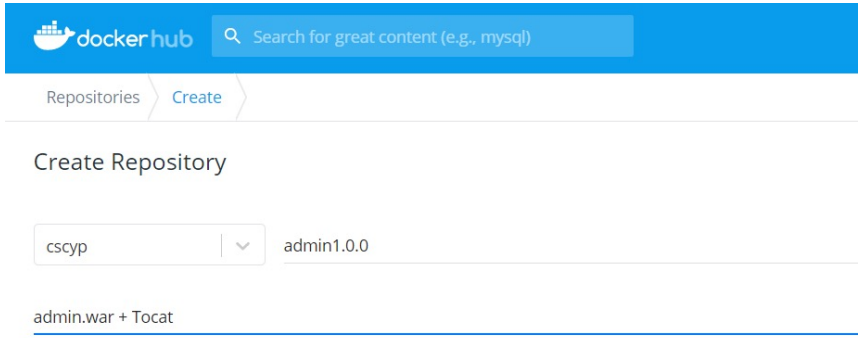
```
# 给它取名 admin 本机端口映射 8080
docker run --name=admin -p 8080:8080 -d admin:1.0.0
```

本地浏览器可以访问查看：



打包docker镜像上传 docker hub

- 先申请Docker hub 帐号， 并创建仓库：https://hub.docker.com/
- 本地为创建好的镜像打标签
- 本地登录自己的 docker hub 账号
- 将镜像推送上去：docker push
- 此时已经成功推送镜像，可以提供下载




```
docker tag <existing-image> <hub-user>/<repo-name>[:<tag>]
这里的tag不指定就是latest。
```

1. 在本地登录docker hub 帐号,命令如下:

```
root@master:~# docker  login
Username: lidnyun
Password:
Email: 邮箱地址
WARNING: login credentials saved in /root/.docker/config.json
Login Succeeded
```

2.push镜像，命令如下:

```
docker push <hub-user>/<repo-name>:<tag>
```

大致过程如下：

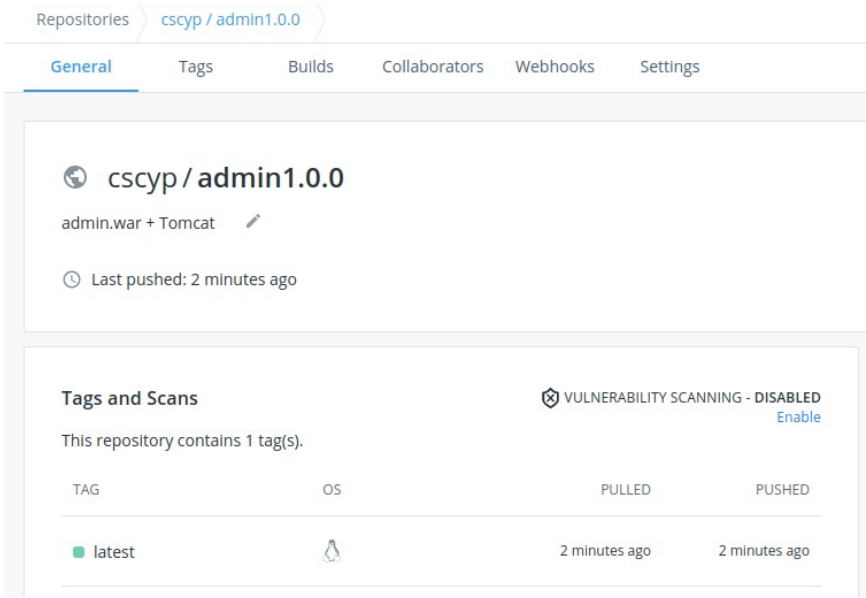
参考资料：https://blog.csdn.net/qq_32923745/article/details/80817395

https://blog.csdn.net/ximenghappy/article/details/66971035

```
seucyber@master:~/docker-admin$ sudo docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
cscyp/admin1.0.0    latest              b25f49cc2290        About an hour ago   649MB
admin               1.0.0               b25f49cc2290        About an hour ago   649MB
tomcat              latest              040bdb29ab37        8 weeks ago         649MB
bestwu/wechat       latest              53c371b7016c        18 months ago       941MB
bestwu/qq           office              d1a0bdfead00        2 years ago         792MB
```

```
seucyber@localhost:~/docker-admin$ ifconfig
docker0   Link encap:以太网  硬件地址 02:42:cc:34:d7:27
          inet 地址:172.17.0.1  广播:172.17.255.255  掩码:255.255.0.0
          inet6 地址: fe80::42:ccff:fe34:d727/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:75211 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:136456 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:3593164 (3.5 MB)  发送字节:205310462 (205.3 MB)
```

```
seucyber@localhost:~/docker-admin$ sudo docker build -t admin:1.0.0 .
[sudo] seucyber 的密码:
Sending build context to Docker daemon  11.26kB
Step 1/3 : FROM docker.io/tomcat
 ---> 040bdb29ab37
Step 2/3 : MAINTAINER rstyro
 ---> Using cache
 ---> 409848d77673
Step 3/3 : COPY admin.war /usr/local/tomcat/webapps
 ---> Using cache
 ---> b25f49cc2290
Successfully built b25f49cc2290
Successfully tagged admin:1.0.0
```




```
seucyber@master:~/docker-admin$ sudo docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, he
ad over to https://hub.docker.com to create one.
Username: cscyp
Password:
WARNING! Your password will be stored unencrypted in /home/seucyber/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
seucyber@master:~/docker-admin$ sudo docker push cscyp/admin1.0.0
The push refers to repository [docker.io/cscyp/admin1.0.0]
1b58ba180c4e: Pushed
9ddc8cd8299b: Mounted from library/tomcat
c9132b9a3fc8: Mounted from library/tomcat
8e2e6f2527c7: Mounted from library/tomcat
500f722b156b: Mounted from library/tomcat
7a9b35031285: Mounted from library/tomcat
7496c5e8691b: Mounted from library/tomcat
aa7af8a465c6: Mounted from library/tomcat
ef9a7b8862f4: Mounted from library/tomcat
a1f2f42922b1: Mounted from library/tomcat
4762552ad7d8: Mounted from library/tomcat
latest: digest: sha256:632714d281ba3b34ec8ca1648a25e401adc98c18c5c3e6d0f44a9e84ca85cc12 size: 2629
```

再来看一个真实操作系统里的例子

- 举例：helloworld 程序
  - helloworld 程序实际上是由一组进程（Linux 里的线程）组成
  - 这 4 个进程共享 helloworld 程序的资源，相互协作，完成 helloworld 程序的工作

```
$ pstree -p
…
   |-helloworld,3062
   |    |-{api},3063
   |    |-{main},3064
   |    |-{log},3065
   |    `-{compute},3133
```

思考

- Kubernetes = 操作系统 (比如：Linux)
- 容器 = 进程 (Linux 线程)
- Pod = ?
  - **进程组** (Linux 线程组)

“进程组”

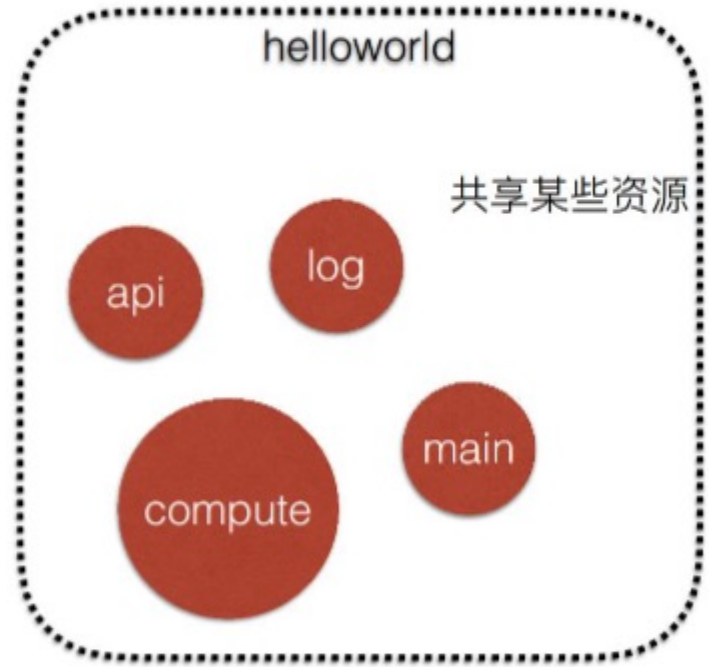
- 举例：
  - helloworld 程序由 4 个进程组成，这些进程之间共享某些文件
- 问题：helloworld 程序如何用容器跑起来呢?
  - 解法一：在一个 Docker 容器中，启动这 4 个进程
    - 疑问：容器 PID = 1 的进程就是应用本身比如 main 进程，那么“谁”来负责管理剩余的 3 个进程?
      - **容器是“单进程”模型!**
      - 除非：
        - 应用进程本身具备“进程管理”能力（这意味着：helloworld 程序需要具备 systemd的能力）
        - 或者，容器的 PID=1进程改成 systemd
          - 这会导致：管理容器 = 管理 systemd != 直接管理应用本身

```
$ pstree -p
…
   |-helloworld,3062
   |    |-{api},3063
   |    |-{main},3064
   |    |-{log},3065
   |    `-{compute},3133
```

Pod = “进程组”

```
$ pstree -p
…
   |-helloworld,3062
   |    |-{api},3063
   |    |-{main},3064
   |    |-{log},3065
   |    `-{compute},3133
```

```
apiVersion: v1
kind: Pod
metadata:
  name: helloworld
spec:
  containers:
  - name: api
    image: api
    ports:
    - containerPort: 80
  - name: main
    image: main
  - name: log
    image: log
    volumeMounts:
    - name: log-storage
  - name: compute
    image: compute
    volumeMounts:
    - name: data-storage
```



Pod：一个逻辑单位，多个容器的组合
Kubernetes 的原子调度单位

来自 Google Borg 的思考

- Google 的工程师们发现，在 Borg 项目部署的应用，往往都存在着类似于“进程和进程组”的关系。更具体地说，就是这些应用之间有着密切的协作关系，使得它们必须部署在同一台机器上并且共享某些信息

- Large-scale cluster management at Google with Borg, EuroSys’15

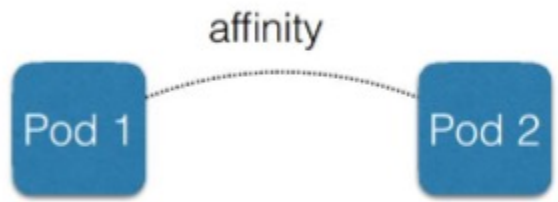
为什么 Pod 必须是原子调度单位?

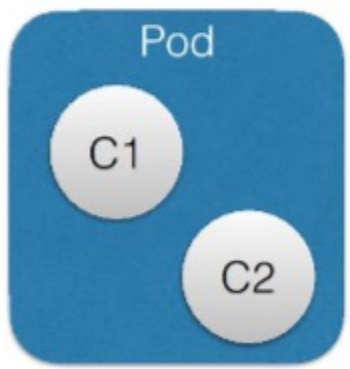
- 举例：两个容器紧密协作
  - App：业务容器，写日志文件
  - LogCollector: 转发日志文件到 ElasticSearch 中
- 内存要求:
  - App: 1G
  - LogCollector: 0.5G
- 当前可用内存:
  - Node_A: 1.25G
  - Node_B: 2G
- 如果 App 先被调度到了 Node_A 上，会怎么样?

- Task co-scheduling 问题
  - Mesos：资源囤积（resource hoarding）：
    - 所有设置了Affinity约束的任务都达到时，才开始统一进行调度
    - 调度效率损失和死锁
  - Google Omega：乐观调度处理冲突：
    - 先不管这些冲突，而是通过精心设计的回滚机制在出现了冲突之后解决问题。
    - 复杂
  - Kubernetes：Pod

再次理解 Pod

- **亲密关系 - 调度解决**
  - 两个应用需要运行在同一台宿主机上
- **超亲密关系 - Pod 解决**
  - 会发生直接的文件交换
  - 使用localhost或者Socket文件进行本地通信
  - 会发生非常频繁的RPC调用
  - 会共享某些Linux Namespace（比如，一个容器要加入另一个容器的Network Namespace）
  - …





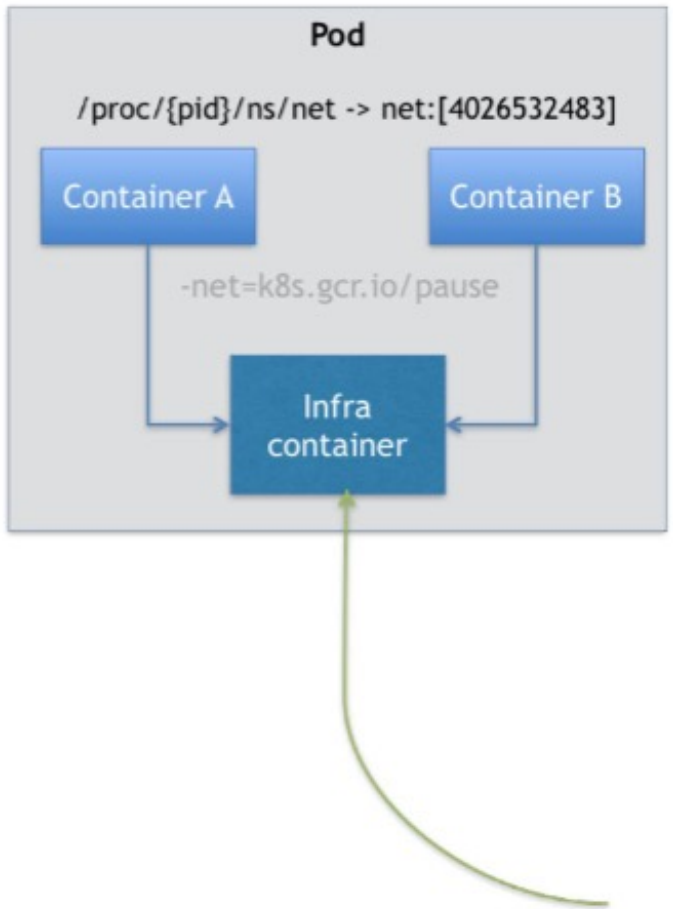
2 Pod 的实现机制

Pod 要解决的问题

- 如何让一个 Pod 里的多个容器之间最高效的共享某些资源和数据?
  - 容器之间原本是被 Linux Namespace 和 cgroups 隔离开的

1. 共享网络

- 容器 A 和 B
  - 通过 Infra Container 的方式共享同一个 Network Namespace：
    - 镜像：k8s.gcr.io/pause；汇编语言编写的、永远处于“暂停”；大小100~200 KB
  - 直接使用localhost进行通信
  - 看到的网络设备跟Infra容器看到的完全一样
  - 一个Pod只有一个IP地址，也就是这个Pod的Network Namespace对应的IP地址
    - 所有网络资源，都是一个Pod一份，并且被该Pod中的所有容器共享
  - 整个 Pod的生命周期跟Infra容器一致，而与容器A和B无关



2. 共享存储

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:
  restartPolicy: Never
  volumes:
  - name: shared-data
    hostPath:
      path: /data
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: debian-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello from the debian container > /
pod-data/index.html"]
```

- shared-data 对应在宿主机上的目录会被同时绑定挂载进了上述两个容器当中

本节总结

- Pod 是 Kubernetes 项目里实现“容器设计模式”的核心机制
- “容器设计模式”是 Google Borg 的大规模容器集群管理最佳实践之一
  - 也是 Kubernetes 进行复杂应用编排的基础依赖之一
- 所有“设计模式”的本质都是：解耦和重用