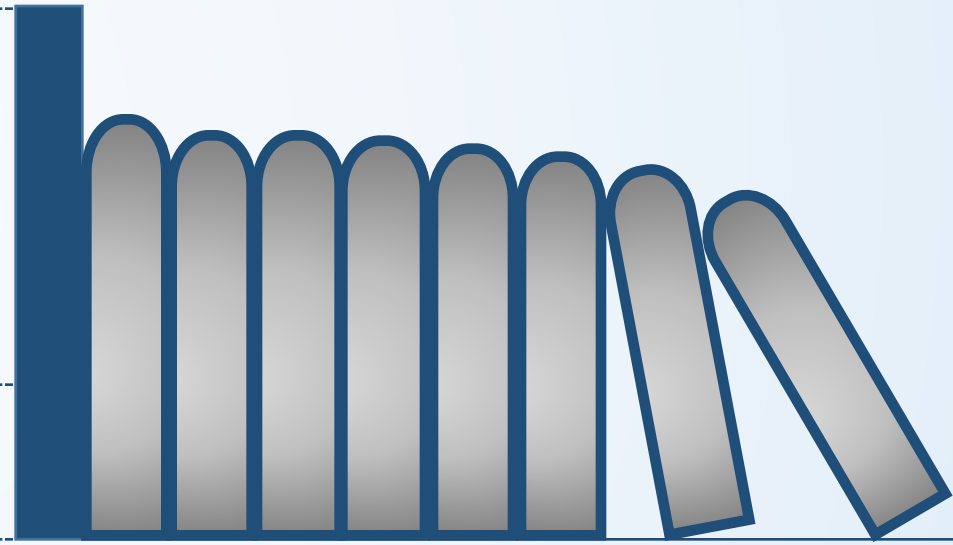
Docker镜像与容器



主讲人：宋小金

目录

1 镜像分层结构

2 镜像写时复制

3 镜像基本操作

4 镜像组织结构

5 搭建镜像仓库

预期收获

- 了解镜像分层
- 了解镜像采用的技术
- 掌握镜像的基本操作
- 了解镜像的组织结构

Docker镜像基本知识

Docker镜像介绍：

Docker借鉴了Git利用分层的优点，是*Docker Image分层*变为可能，并且借鉴了Github理念实现了DockerHub。Docker一直宣称的卖点是*一次部署，到处运行*。

其核心主要是Docker Image（镜像）支持的：

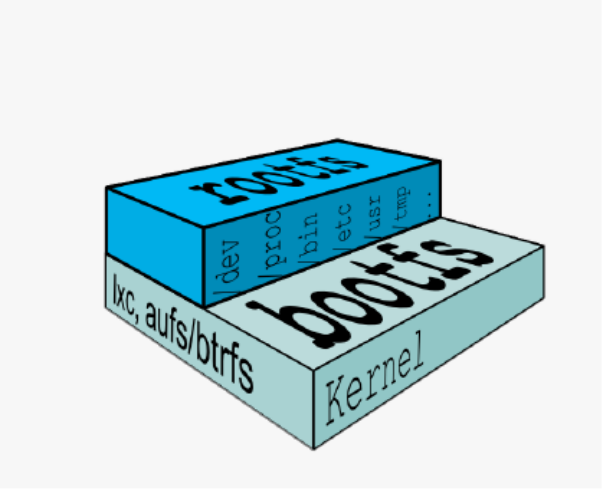
- Docker通过把应用的*运行时环境和应用打包*在一块，解决了部署环境依赖的问题。
- 另外，通过*文件系统分层*的概念，通过*分层复用*，大幅的节省了磁盘空间。

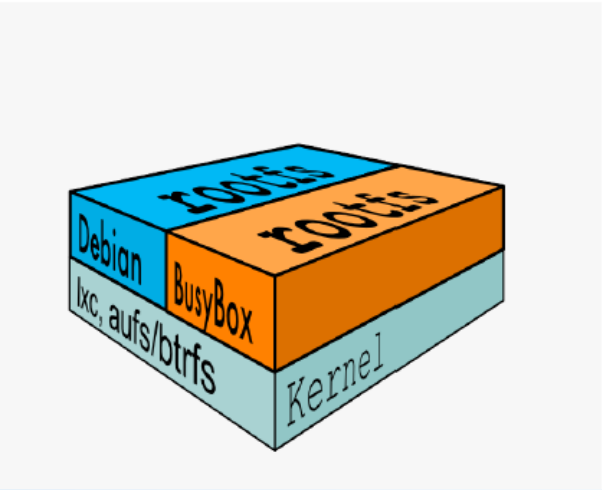
Docker镜像基本知识

- registry.abc.net/orgname/imagename:tag
  - registry.abc.net：Remote registry的地址
  - （docker.io,gcr.io）
  - orgname:（optional）组织区分的镜像集合
  - imagename:镜像名称
  - tag:标签
- docker.io/oracle/mysql:v5.7.1
- k8s.gcr.io/kube-apiserver-amd64:v1.11.0

小心latest，千万别被 latest tag 给误导了。latest 其实并没有什么特殊的含义。当没指明镜像 tag 时，Docker 会使用默认值 latest，仅此而已，所以我们在使用镜像时最好还是避免使用 latest，明确指定某个 tag，比如 httpd:2.3

Docker镜像基本知识



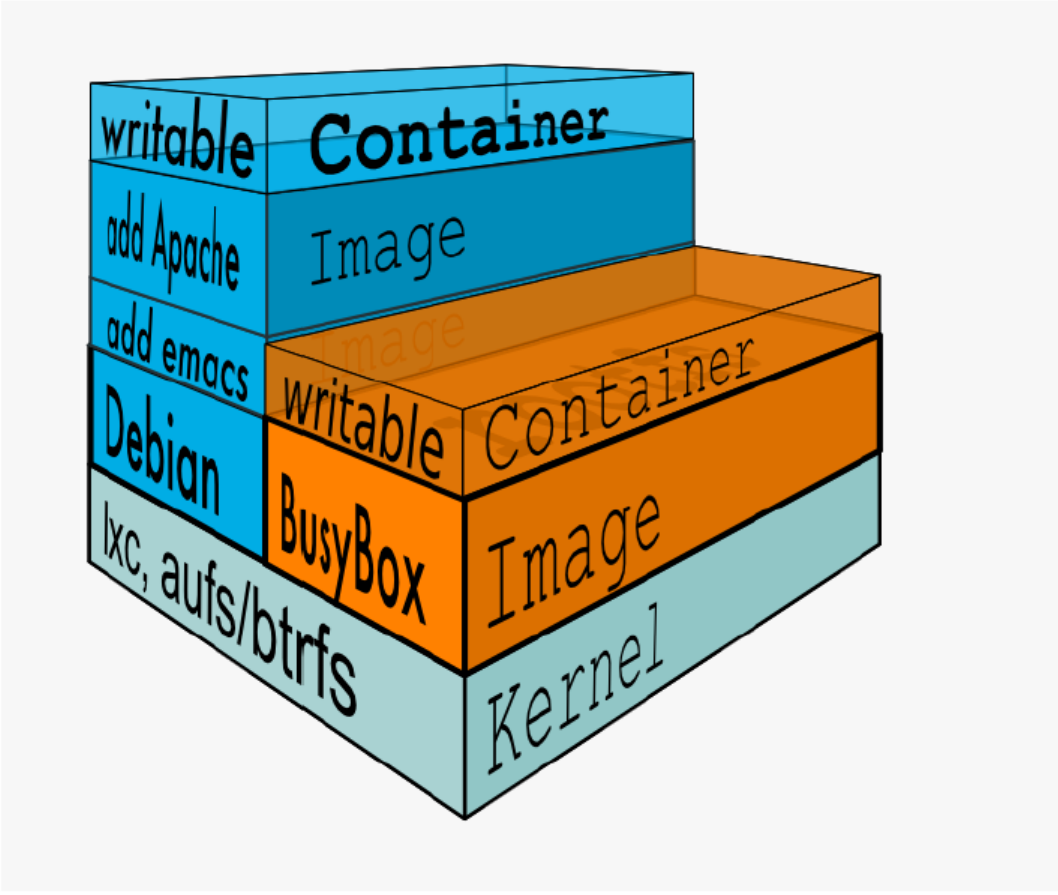


典型的Linux文件系统

- bootfs(bootfilesystem)
  - Bootloader - 引导加载kernel
  - Kernel - 当kernel被加载到内存中后umount bootfs
- rootfs(rootfilesystem)
  - /dev，/proc，/bin，/etc等标准目录和文件
- 对于不同的linux发行版,bootfs基本是一致的
- 但rootfs会有差别

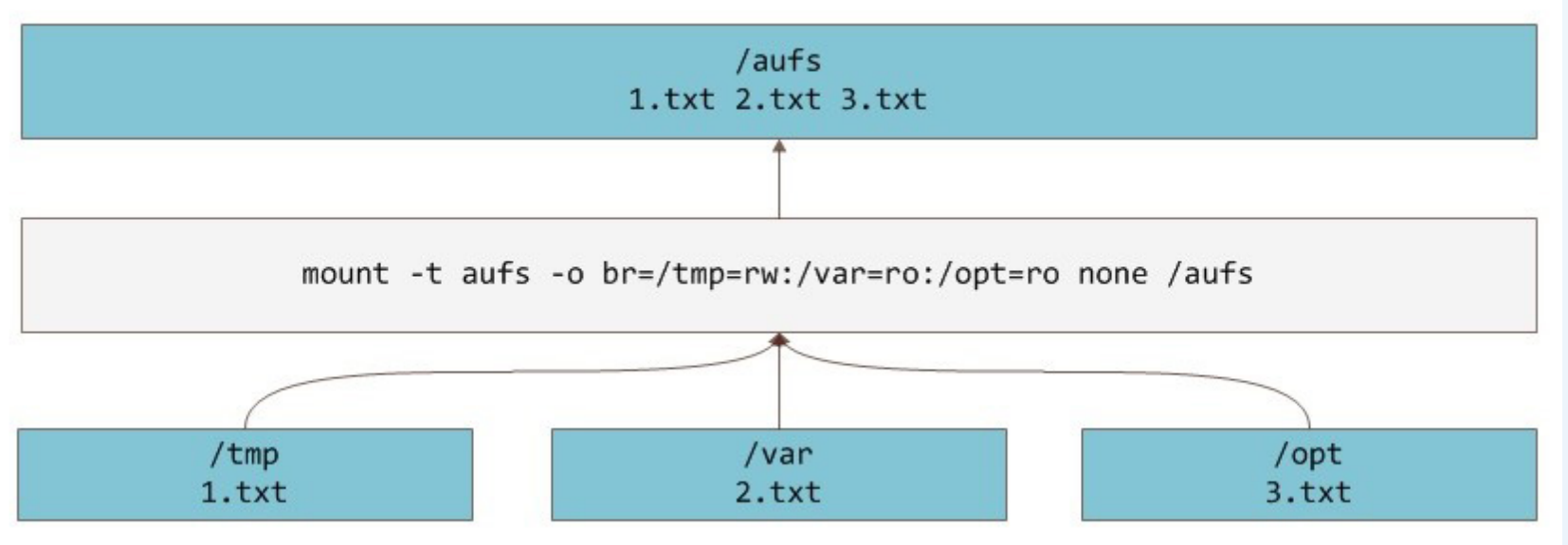
Docker的文件系统如何启动

- Linux
  - 在启动后，首先将 rootfs 置为 readonly, 进行一系列检查, 然后将其切换为 “readwrite” 供用户使用。
- Docker
  - 也是将 rootfs 以readonly方式加载并检查，然而接下来利用 union mount 的将一个 readwrite 文件系统挂载在 readonly 的rootfs之上
  - 并且允许再次将下层的 file system设定为readonly 并且向上叠加
  - 这样一组readonly和一个writeable的结构构成一个container的运行目录, 每一个被称作一个Layer。



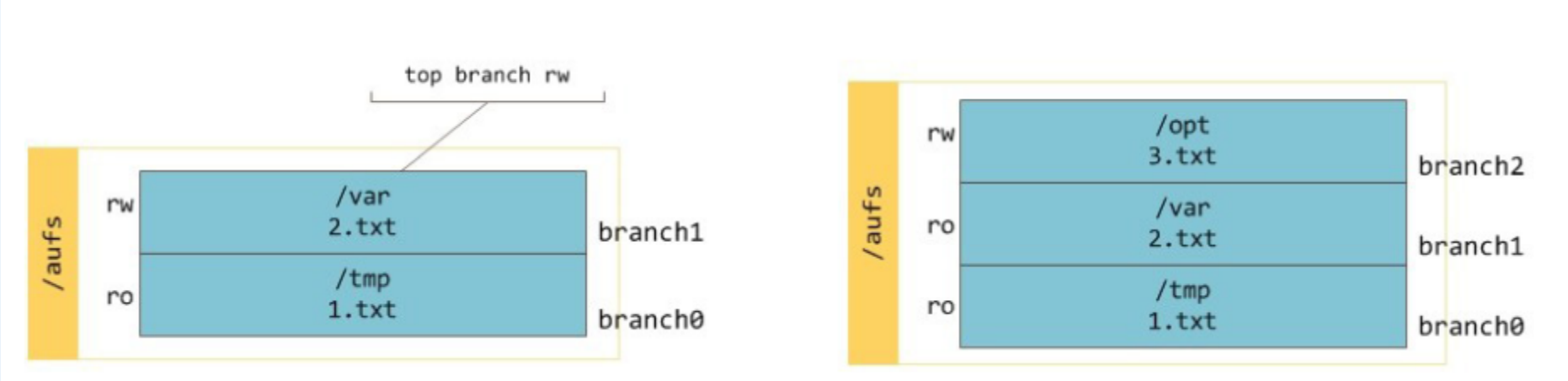
AUFS

AUFS是一种Union File System，所谓UnionFS就是把不同物理位置的目录合并mount到同一个目录中。UnionFS的一个最主要的应用是，把一张CD/DVD和一个硬盘目录给联合 mount在一起，然后，你就可以对这个只读的CD/DVD上的文件进行修改（当然，修改的文件存于硬盘上的目录里）



AUFS

AUFS是将多个目录合并成一个虚拟文件系统，成员目录称为虚拟文件系统的一个分支（branch），每个branch可以指定 readwrite/whiteout-able/readonly权限，只读（ro），读写（rw），写隐藏（wo）。一般情况下，aufs只有最上层的branch具有读写权限，其余branch均为只读权限。只读branch只能逻辑上修改，AUFS每层branch可以动态的增加删除，每增加一层，下层默认置为ro，最上一层为rw。删除branch是在aufs挂载点移除，并未删除挂载目录

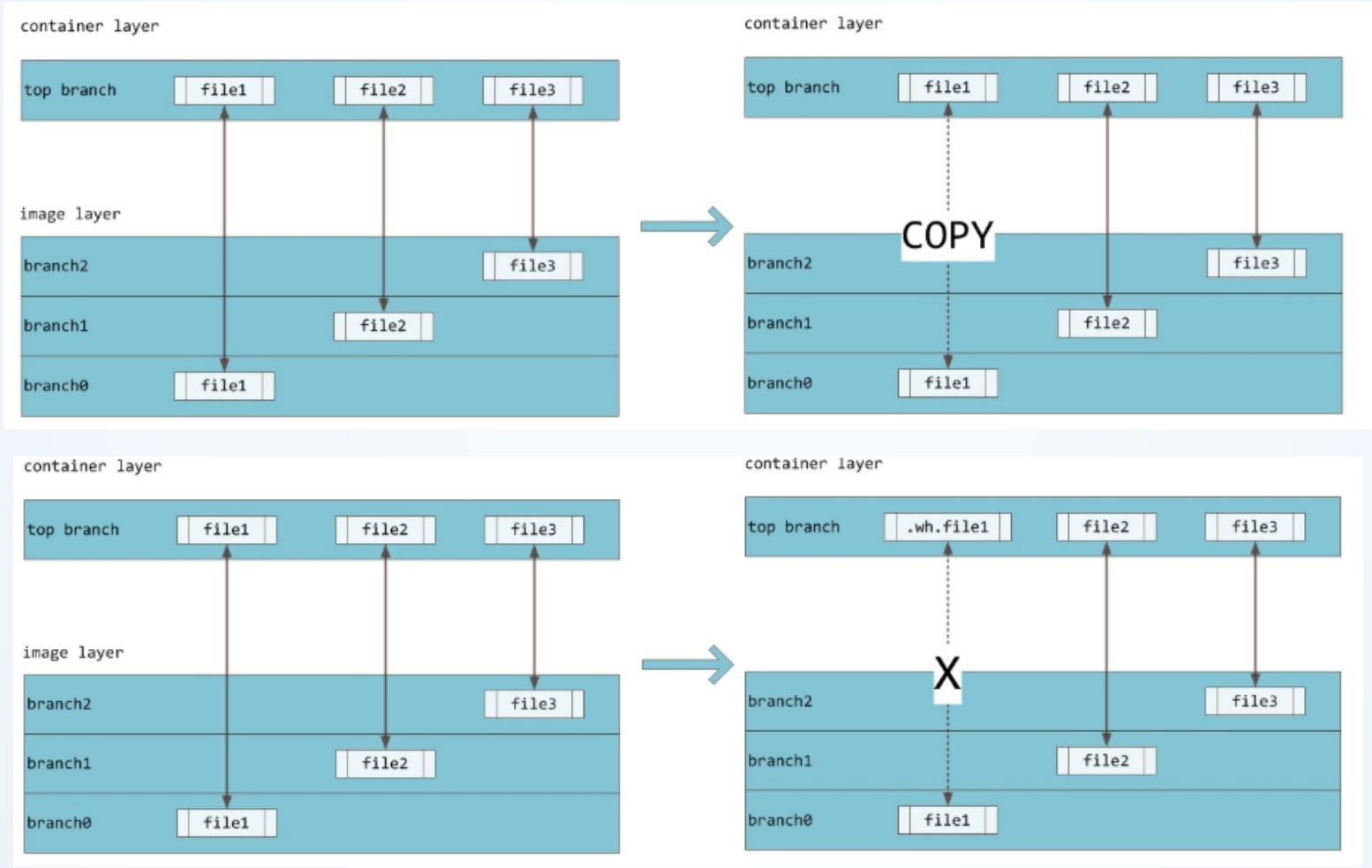


AUFS的文件操作

当需要修改一个文件，而该文件位于低层branch时，顶层branch会直接复制低层branch的文件至顶层再进行修改，而低层的文件不变，这种方式即是CoW技术（写复制），AUFS默认支持Cow技术，当容器删除一个低层branch文件时，只是在顶层branch对该文件进行重命名并隐藏，实际并未删除文件，只是不可见，这种方式即AUFS的whiteout（写隐藏）

- 添加文件：创建文件时，新文件被添加到branch中
- 读取文件 ：读取某个文件时，会从上往下依次在各镜像层中查找此文件。一旦找到，打开并读入内存。
- 修改文件 ：修改已存在的文件时，会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后修改之。
- 删除文件 ：删除文件时，也是从上往下依次在镜像层中查找此文件。找到后，会在branch中记录下此删除操作。

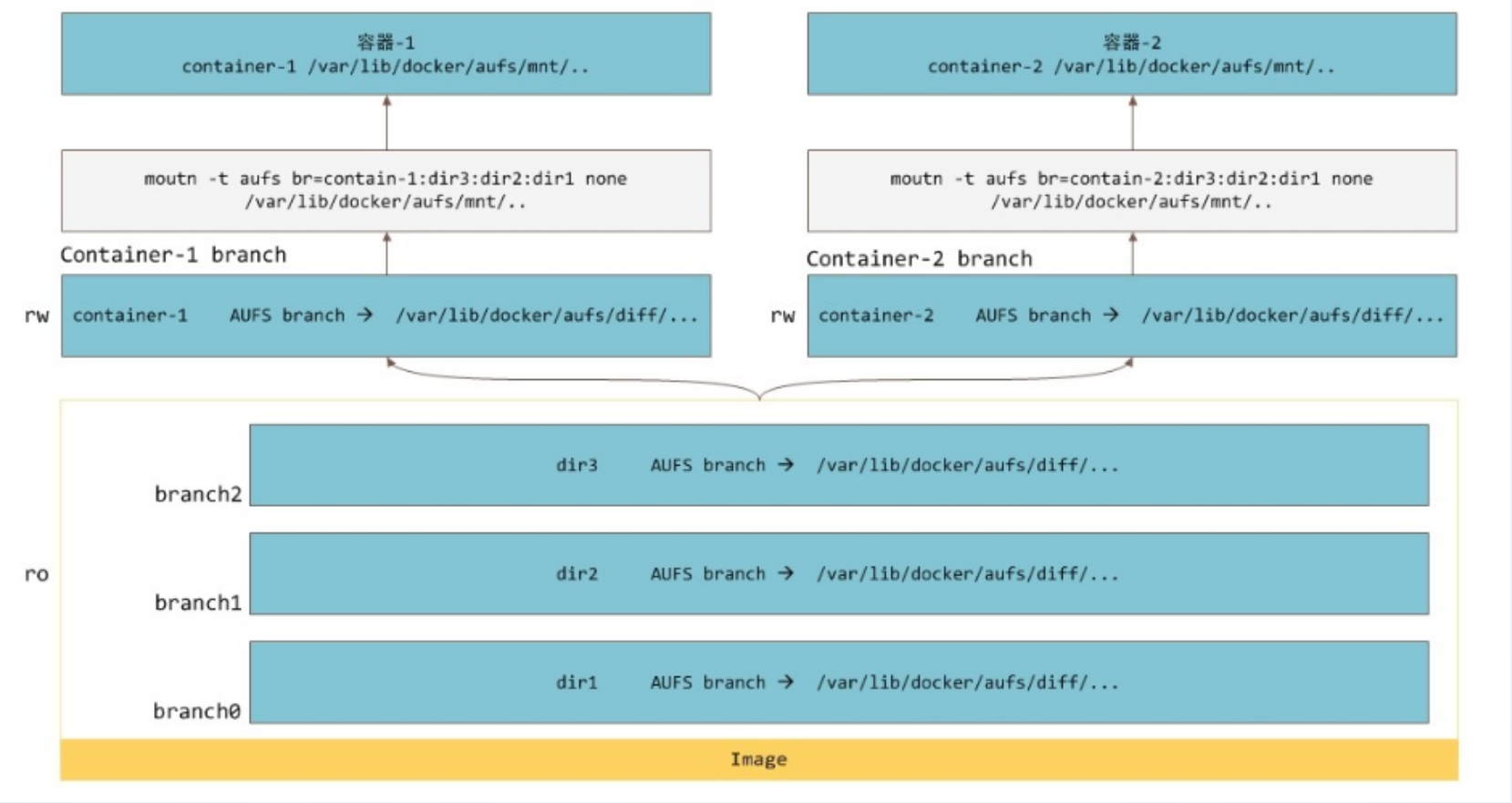
AUFS的文件操作


container layer
top branch
file1
file2
file3
image layer
branch2
file3
branch1
file2
branch0
file1
container layer
top branch
file1
file2
file3
COPY
branch2
file3
branch1
file2
branch0
file1
container layer
top branch
file1
file2
file3
image layer
branch2
file3
branch1
file2
branch0
file1
container layer
top branch
.wh.file1
file2
file3
X
branch2
file3
branch1
file2
branch0
file1

Docker中的AUFS

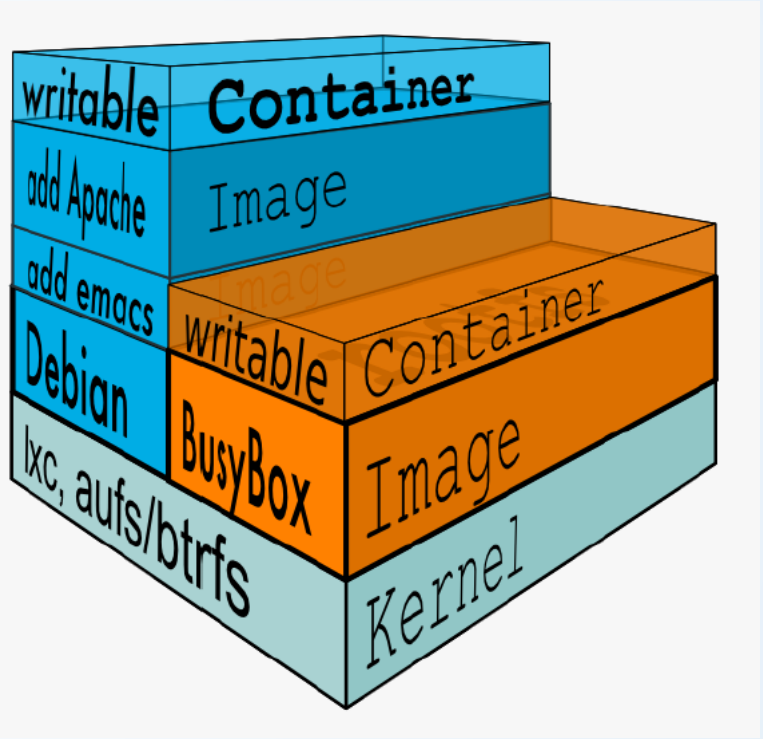
Docker镜像（Image）是由一个或多个AUFS branch组成，并且所有的branch均为只读权限。简单来说，AUFS所有robranch按照一定顺序堆积构成Docker Image镜像

在运行容器的时候，创建一个AUFS branch位于image层之上，具有rw权限，并把这些branch联合挂载到一个挂载点下。这就是Docker能够一个镜像运行多个容器的原理所在



写时复制（Copy-on-Write）

- 容器启动时，一个新的可写层被加载到镜像的顶部，这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”
- 所有对容器的改动，无论添加、删除、还是修改文件都只会发生在容器层中
- 只有容器层是可写的，容器层下面的所有镜像层都是只读的
- 只有当需要修改时才复制一份数据，这种特性被称作 Copy-on-Write。可见，容器层保存的是镜像变化的部分，不会对镜像本身进行任何修改



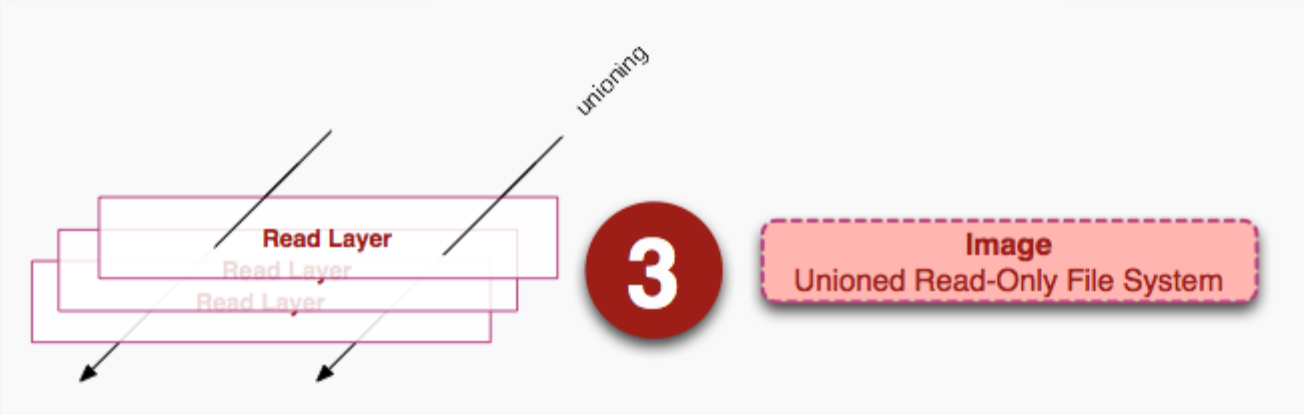
AUFS的好处

- 节省存储空间：多个容器可以共享基础镜像（Base Image）存储
- 快速部署：如果要部署多个容器，基础镜像（Base Image）可以避免多次拷贝
- 内存更省：因为多个容器共享Base Image，以及OS的Disk缓存机制，多个容器中的进程命中缓存内容的几率大大增加
- 允许在不更改基础镜像的同时修改其目录中的文件：所有写操作都发生在最上层的writeable层中，这样可以大大增加Base Image能共享的文件内容

镜像的定义

镜像（Image）就是一堆只读层（read-only layer）的统一视角

从左边我们看到了多个只读层，它们重叠在一起。除了最下面一层，其它层都会有一个指针指向下一层。这些层是Docker内部的实现细节，并且能够在主机（译者注：运行Docker的机器）的文件系统上访问到。统一文件系统（union file system）技术能够将不同的层整合成一个文件系统，为这些层提供了一个统一的视角



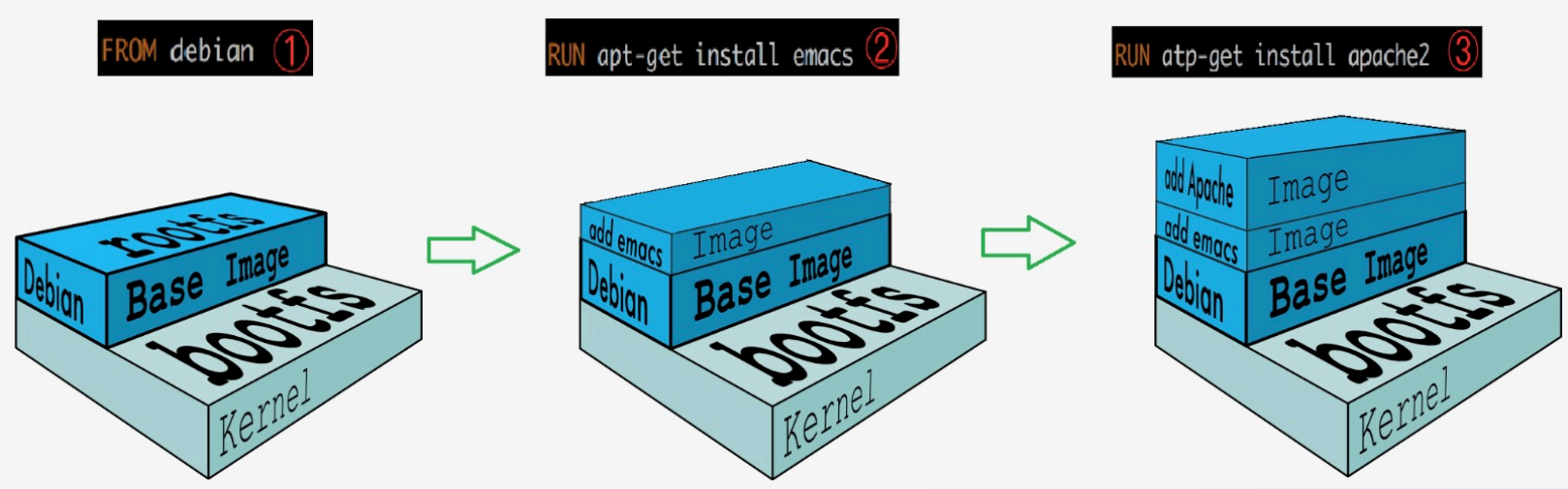
镜像的分层

镜像的一层被称为一个layer，类似于Git仓库一次commit

```
FROM debian ①
RUN apt-get install emacs ②
RUN atp-get install apache2 ③
CMD ["/bin/bash"] ④
```

构建过程如右图所示，

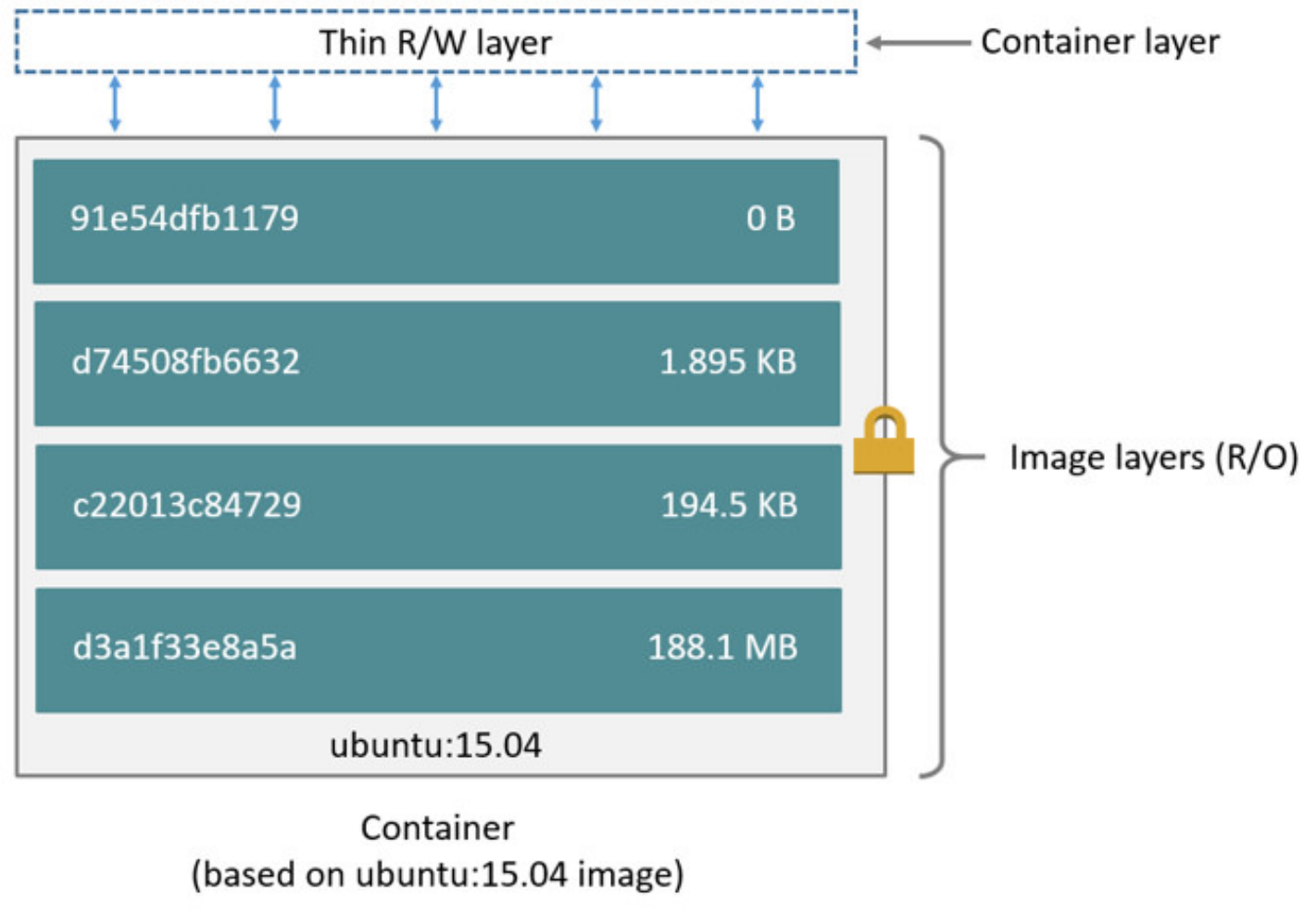
新镜像是从基础镜像一层一层叠加生成。



镜像与分层

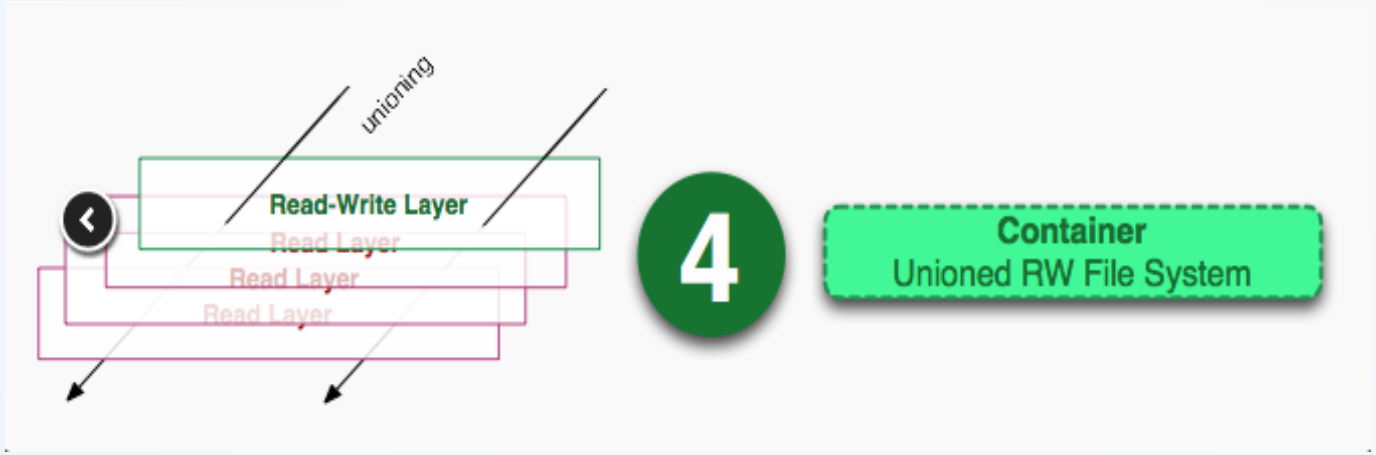
1. Docker镜像是多个的，堆叠的分层的只读文件系统
2. 每一层的变化是基于下一层的
3. 相邻层之间的改动是有延续性的
4. 从docker 1.10之后，每个镜像层的由一致性的hash生成id，取代了旧版使用随机生成的UUID.

```
FROM ubuntu:15.10
COPY . /app
RUN make /app
CMD python /app/app.py
```



容器的定义

- 容器（container）的定义和镜像（image）几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的
- 要点：容器 = 镜像 + 读写层，并且容器的定义并没有提及是否要运行容器



Dockerfile

Dockerfile是docker构建镜像的基础，也是docker区别于其他容器的重要特征，正是有了Dockerfile，docker的自动化和可移植性才成为可能

编写Dockerfile命令：

- FROM：从一个基础镜像构建新的镜像
  - FROM ubuntu
- MAINTAINER：维护者信息
  - MAINTAINER William <wlj@nicescale.com>
- RUN：非交互式运行shell命令
  - RUN apt-get -y update
  - RUN apt-get -y install nginx

Dockerfile

编写Dockerfile命令：

- ENV：设置环境变量
  - ENV MYSQL 5.7
- WORKDIR /path/to/workdir：设置工作目录
  - WORKDIR /var/www
- USER：设置用户ID
  - USER nginx
- VOLUME <#dir>：设置volume
  - VOLUME [‘/data’]
- EXPOSE：暴露哪些端口
  - EXPOSE 80 443

Dockerfile

编写Dockerfile命令：

- ENTRYPOINT [‘executable’, ‘param1’,’param2’]：执行命令
  - ENTRYPOINT ["/usr/sbin/nginx"]
- CMD [“param1”,”param2”]
  - CMD ["start"]

注意：

- ENTRYPOINT指令和CMD指令虽然是在Dockerfile中定义，但是在构建镜像的时候并不会被执行，只有在执行docker run命令启动容器时才会起作用
- 在Dockerfile中，只能有一个ENTRYPOINT指令，如果有多个ENTRYPOINT指令则以最后一个为准
- 在Dockerfile中，只能有一个CMD指令，如果有多个CMD指令则以最后一个为准
- 在Dockerfile中，ENTRYPOINT指令或CMD指令，至少必有其一，如果设置了ENTRYPOINT，则CMD将作为参数

Dockerfile范例

```
FROM centos:7
MAINTAINER fengzp <fengzp@gzyitop.com>
ENV LANG en_US.UTF-8
ENV TOMCAT_VERSION 8.5.13
ENV CATALINA_HOME /opt/apache-tomcat-$TOMCAT_VERSION
ENV PATH $CATALINA_HOME/bin:$PATH
ENV JDK_VERSION 1.8.0

WORKDIR $CATALINA_HOME

RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai  /etc/localtime
RUN yum -y install java-$JDK_VERSION-openjdk-devel && rm -rf /var/cache/yum/*
ENV JAVA_HOME /usr/lib/jvm/java-openjdk

RUN yum -y install wget
RUN cd /opt/ && wget "http://apache.fayea.com/tomcat/tomcat-8/v$TOMCAT_VERSION
    /bin/apache-tomcat-$TOMCAT_VERSION.tar.gz"
RUN cd /opt/ && tar -zxf apache-tomcat-$TOMCAT_VERSION.tar.gz
RUN cd /opt/ && rm -rf apache-tomcat-$TOMCAT_VERSION.tar.gz
RUN chmod +x $CATALINA_HOME/bin/*.sh

VOLUME $CATALINA_HOME/webapps
VOLUME $CATALINA_HOME/logs
VOLUME $CATALINA_HOME/conf

EXPOSE 8080

CMD $CATALINA_HOME/bin/startup.sh && tail -F $CATALINA_HOME/logs/catalina.out
```

镜像构建最佳实践

- ✓**使用统一的base镜像** — Centos, Ubuntu
- ✓**使用小型基础镜像** — 生成镜像也较小
- ✓**动静分离** — 经常变化的内容和基本不会变化的内容要分开，把不怎么变化的内容放在下层，创建出来不同基础镜像供上层使用。
- ✓**最小原则：只安装必需的东西**
- ✓**一个原则：每个镜像只有一个功能** — 不要在容器里运行多个不同功能的进程，每个镜像中只安装一个应用的软件包和文件，需要交互的程序通过 pod 或者容器之间的网络进行交流。
- ✓**使用更少的层** — 尽量把相关的内容放到同一个层，使用换行符进行分割

镜像构建最佳实践

✓**切勿映射公有端口** — 镜像应该可以多次运行在任何主机上

✓**不要在构建中升级软件版本** — 应在基础镜像中更新，类似于类的继承，在基类里实现

✓**利用cache来加快构建速度** — docker build --cache-from 参数可以手动指定一个镜像来使用它的缓存。

✓ **版本控制和自动构建** — 最好把Dockerfile和对应的应用代码一起放到版本控制中，然后能够自动构建镜像。

Docker镜像实操演示

参照附件：03　Docker镜像实操演示.txt

镜像仓库

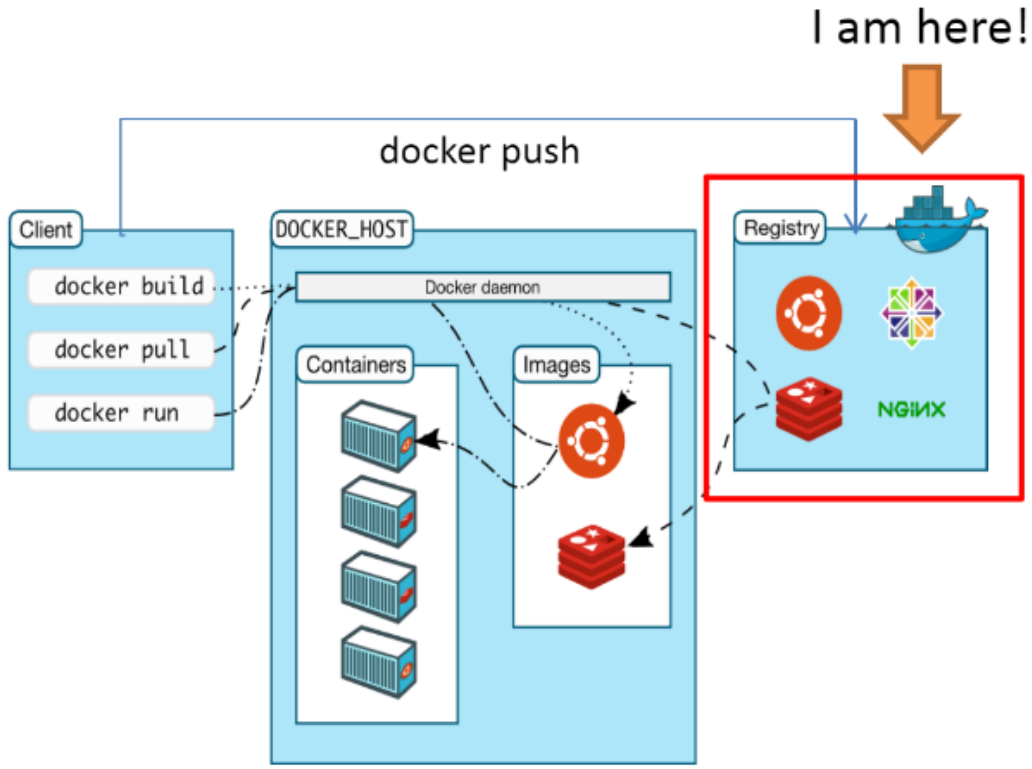
Harbor

Docker Registry

镜像仓库

What is Docker Registry

- Docker Registry : 官方镜像存储、管理和分发工具
- 最新实现是distribution，实现了registry2.0协议
- 官方仓库: hub.docker.com
- 国内一般采用加速器



启动一个registry

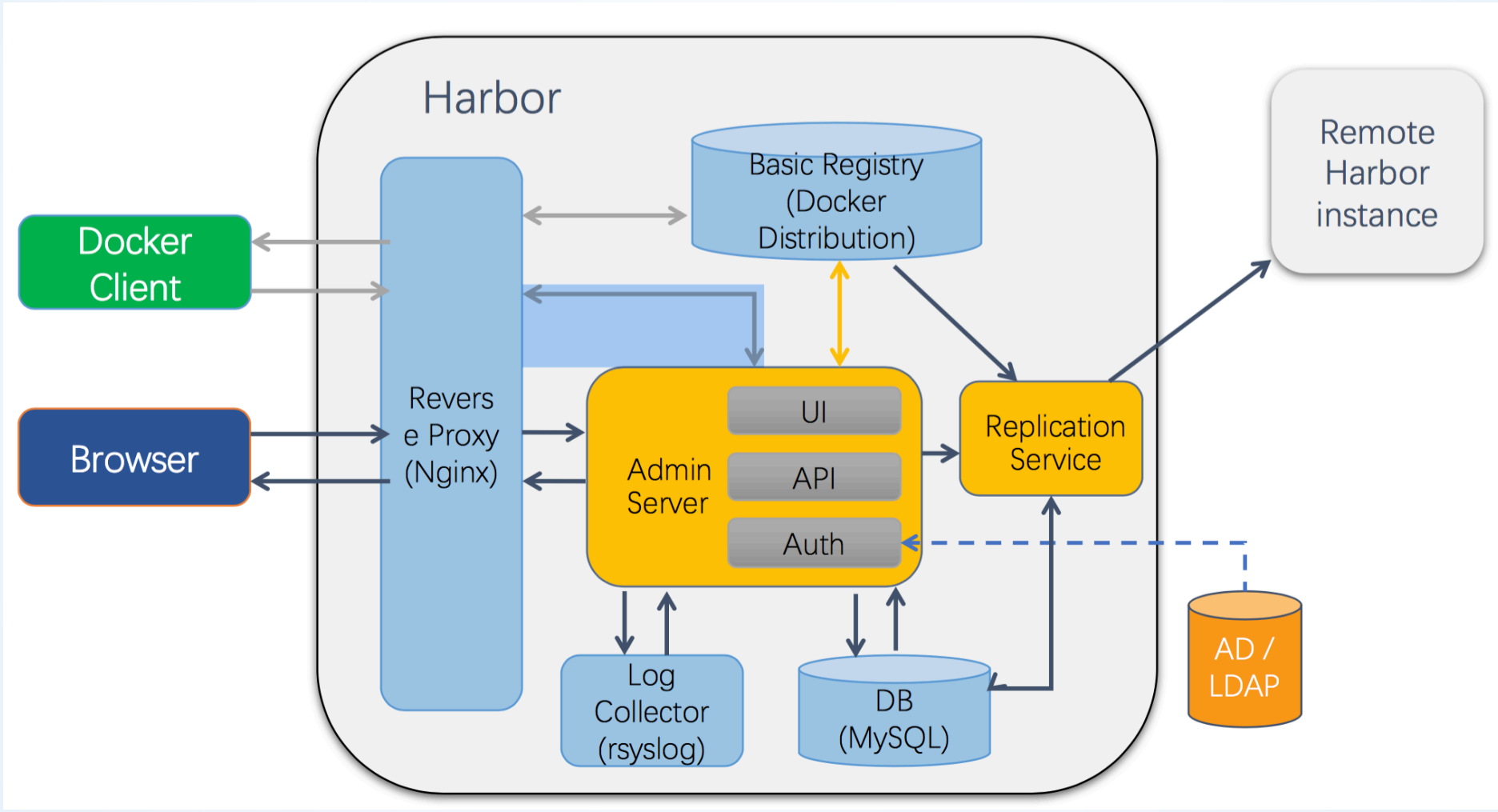
```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

镜像仓库

Features of HARBOR™

- VMware中国团队开源的企业级镜像仓库项目，聚焦镜像仓库的企业级需求：
  - 支持基于角色的访问控制RBAC；
  - 支持镜像复制策略（PUSH）；
  - 支持无用镜像数据的自动回收和删除；
  - 支持LDAP/AD认证；
  - Web UI；
  - 提供审计日志功能；
  - 提供RESTful API，便于扩展；
  - 支持中文&部署Easy。

Harbor架构分析


Harbor
Basic Registry
(Docker
Distribution)
Remote
Harbor
instance
Docker
Client
Browser
Revers
e Proxy
(Nginx)
Admin
Server
UI
API
Auth
Replication
Service
Log
Collector
(rsyslog)
DB
(MySQL)
AD /
LDAP

Harbor组件介绍

1. Nginx：负责请求转发，URL以 /v2/ 开始的请求会被转发到 Docker Registry 中，其它请求由 Admin Server 处理；

2. Admin Server：Harbor 的主体模块，提供 Web UI 和 RESTful API 以及 Auth 相关功能；

3. Replication Service：提供多个 Harbor 实例之间的镜像同步功能；

4. MySQL：Admin Server 和 Replication Service 所用到的数据库；

5. Docker Registry：Docker 官方镜像仓库；

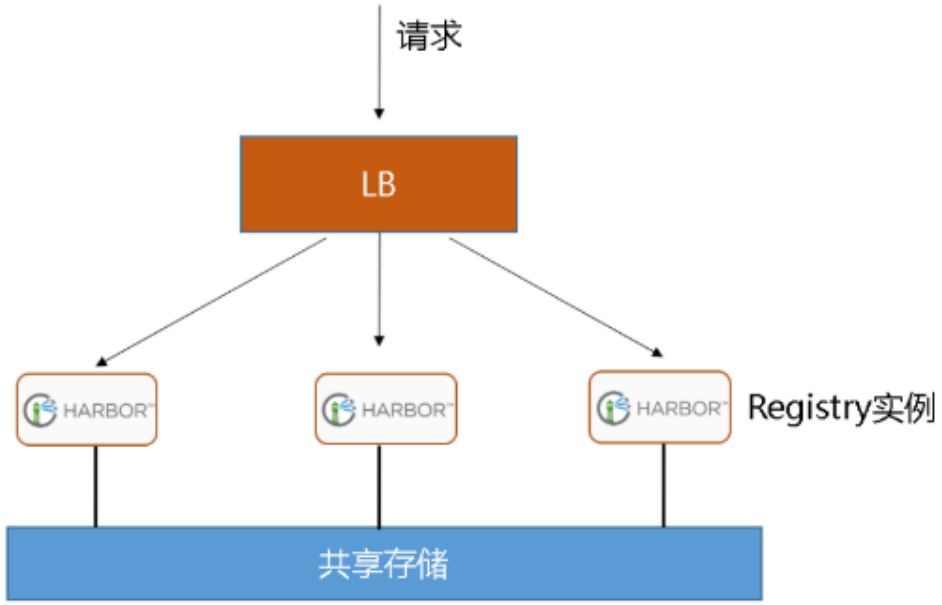
6. Image Storage：镜像的存储介质，可以是本地磁盘，或者分布式存储，根据 Docker Registry 的配置而不同；

7. Log Collector：通过rsyslog收集容器日志。

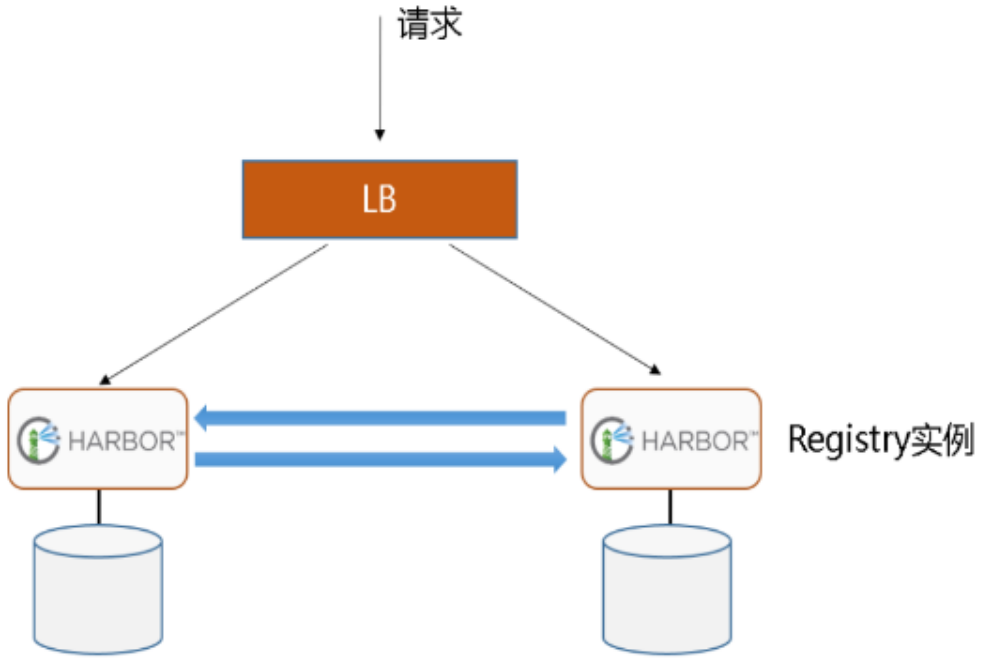
镜像仓库高可用部署

HA Solutions

Solution1: 基于共享存储



Solution2: 基于镜像复制



镜像仓库高可用部署

Cons and Pros

Solution1: 基于共享存储

- 优点
  - 标准方案，Scaling好
  - 数据实时一致
- 不足
  - 门槛高，需要具备现成的共享存储
  - 搭建难度略高

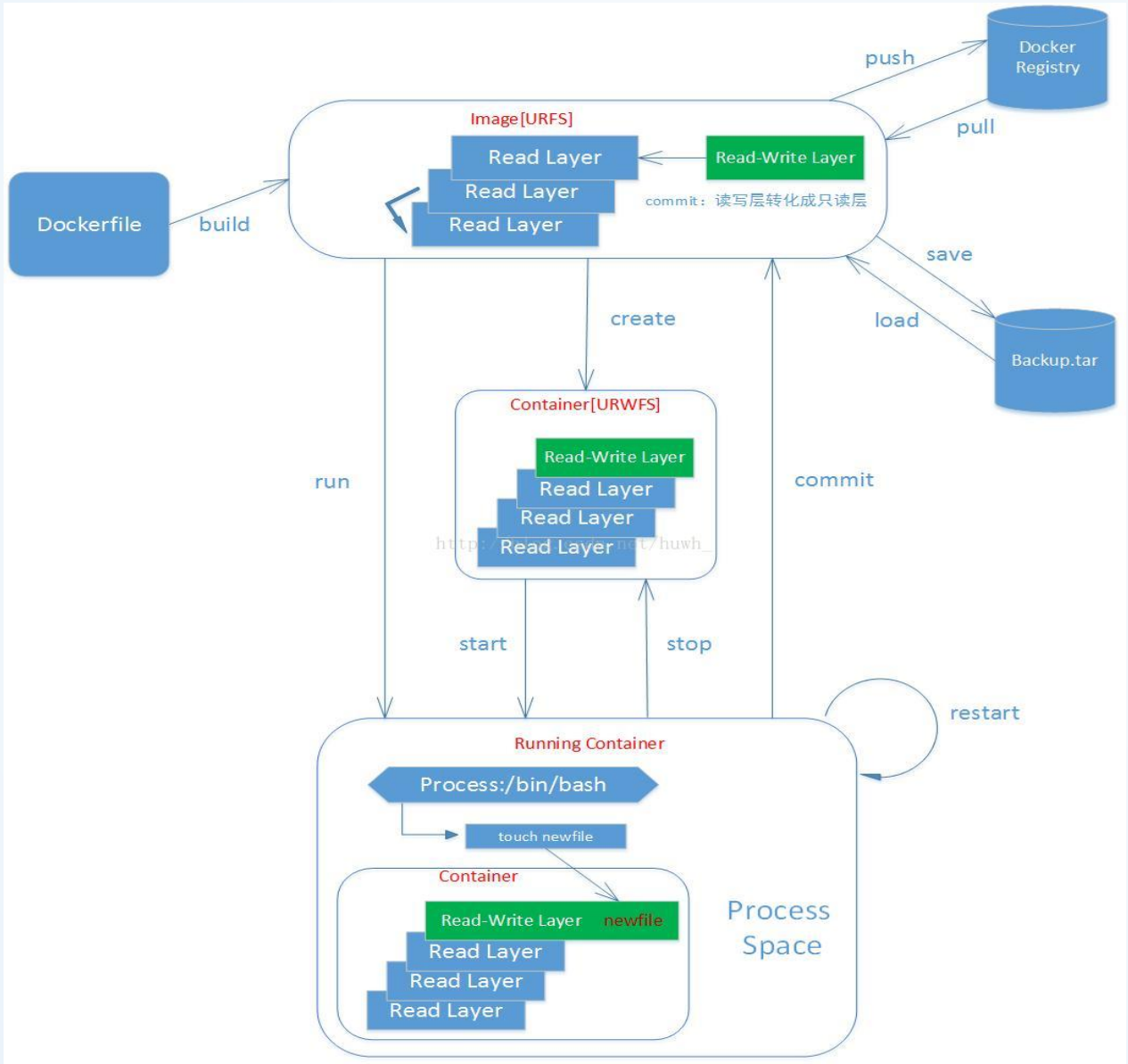
Solution2: 基于镜像复制

- 优点
  - 门槛低，搭建简便
- 不足
  - Scaling差，甚至是不能
  - 镜像复制延迟，导致数据阶段性不一致
  - 添加Project时，需手工维护复制规则

Harbor的安装与配置

参照附件：04　Harbor的安装与配置.txt

容器运行状态图


Docker Registry
push
pull
Image[URFS]
Read Layer
Read Layer
Read Layer
Read-Write Layer
commit：读写层转化成只读层
Dockerfile
build
save
load
Backup.tar
create
Container[URWFS]
Read-Write Layer
Read Layer
Read Layer
Read Layer
run
commit
start
stop
restart
Running Container
Process:/bin/bash
touch newfile
Container
Read-Write Layer
newfile
Read Layer
Read Layer
Read Layer
Process Space

课程回顾

已学知识要点

- *了解镜像分层技术*
- *了解镜像的写时复制与组织结构*
- *掌握镜像相关操作*
- *掌握镜像仓库搭建配置*