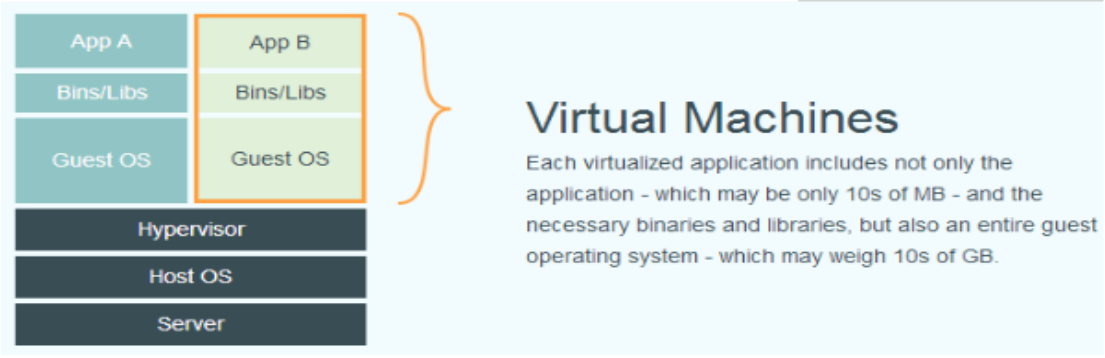
Docker 入门到实践

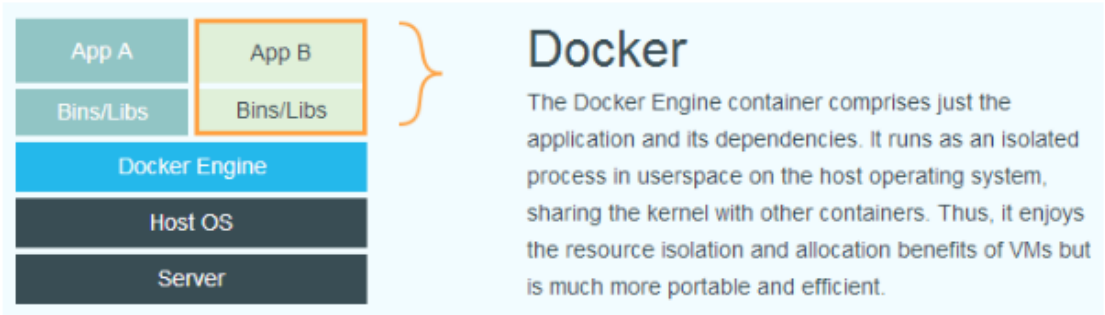
Dokcer 介绍

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口

Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。使得 Docker 技术比虚拟机技术更为轻便、快捷

传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便





为什么要使用 Docker

1、 更高的利用系统资源
2、 更快的启动时间
3、 一致的运行环境
4、 持续交付和部署
5、 更轻松的迁移
6、 更轻松的维护和扩展

对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

基本概念

Docker 包括三个基本概念：

- 镜像
- 容器
- 仓库

Docker 镜像

Docker 镜像（Image）类似于虚拟机镜像，可以理解为面向 Docker 引擎的只读模板

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

Docker 容器

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的 类 和 实例 一样，镜像是静态的定义，容器是镜像运行时的实体。容器是从镜像创建的应用运行实例，可以将其启动、停止、重启、删除

容器的实质是进程。但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间。甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立与宿主的系统下操作一样

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失

Docker 最佳实践：容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用数据卷(Volume)、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对手宿主(或网络存储)发生读写，其性能和稳定性更高

Docker 仓库

Docker 集中存放镜像文件的应用

安装 Docker

Docker 在 1.13 版本之后，从 2017 年的 3 月 1 日开始，版本命名规则变为如下：

项目	说明
版本格式	YY.MM
stable 版本	每个季度发行
edge 版本	每个月发行
当前 Docker CE 版本	17.10.0

同时 Docker 划分为 CE 和 EE。CE 版本即社区版(免费，支持周期三个月)，EE 即企业版，强调安全，付费使用

使用脚本安装 Docker

```
curl -fsSL get.docker.com -o get-docker.sh
sudo sh get-docker.sh --mirror Aliyun
sudo groupadd docker
sudo usermod -aG docker $USER
```

使用镜像

Docker 运行容器前需要本地对应的镜像，如果镜像不存在，Docker 会用镜像仓库下载(默认是 Docker Hub 公共注册服务器中的仓库)

获取镜像：

docker pull [选项] [Docker Registry 地址]<仓库名>:<标签>

Docker Registry 地址：地址的格式一般是 <域名/IP>[:端口号]。默认地址是 Docker Hub。

仓库名：如之前所说，这里的仓库名是两段式名称，既 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。

运行镜像：

docker run -it ubuntu14.04 bash

-it：这是两个参数，一个是 -i：交互式操作，一个是 -t 终端。我们这里打算进入 bash 执行一些命令并查看返回结果，因此我们需要交互式终端。

--rm：这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 docker rm。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 --rm 可以避免浪费空间。

ubuntu:14.04：这是指用 ubuntu:14.04 镜像为基础来启动容器。

bash：放在镜像名后的是命令，这里我们希望有个交互式 Shell，因此用的是 bash

列出镜像：

docker images

查看容器变动

docker diff ubuntu14.04

提交变更后的镜像

```
docker commit [选项] <容器 ID 或容器名> [<仓库名>[:<标签>]]
docker commit \
    --author "Tao Wang <twang2218@gmail.com>" \
    --message "修改了默认网页" \
    webserver \
    nginx:v2
```

查看镜像更改历史记录：

```
docker history nginx:v2
```

更改后可以直接运行该镜像

删除镜像

```
docker rmi [选项] <镜像 1> [<镜像 2> ...]
```

使用 Dockerfile 定制镜像

FROM：FROM <image>或者 FROM <image>:<tag>或者 FROM <image>@<digest>
指定构建依赖的基础镜像，FROM 指令必须作为 Dockerfile 中第一条没有被注释的指令

MAINTAINER：MAINTAINER <name>指定镜像的 Author 字段

RUN：shell 形式，command 是作为/bin/sh -c 的参数进行执行的，即为 shell 的子进程。RUN <command> exec 形式，直接执行 RUN ["executable", "param1", "param2"]

RUN 指令是在当前镜像上执行命令，并且提交执行之后的结果，作为最新的一层 layer，并且后续的 Dockerfile 指令会在 RUN 指令执行完生成的最新镜像上继续执行。

CMD：exec 形式，直接执行，推荐使用该形式

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

第二种形式，作为 ENTRYPOINT 指令的默认参数

CMD ["param1","param2"]

第三种形式，shell 形式，作为/bin/sh -c 的参数进行执行，即为 shell 的子进程

CMD command param1 param2

在一个 Dockerfile 文件中，只能有一个 CMD 指令，如果有多于一条 CMD 指令，那只有最后一条 CMD 指令会生效。 CMD 指令的主要目的是提供容器运行时的默认值，这些默认值可以包括一个可执行文件名，加上执行时的一些参数，或者不包含可执行文件名，只提供参数，但是必须通过增加一个 ENTRYPOINT 指令来指定可执行文件名

LABEL：LABEL <key>=<value> <key>=<value> <key>=<value> ...

LABEL 指令给一个镜像增加元信息 metadata，一个 LABEL 是一个键值对，为了在 LABEL 值中包含空格或者换行符时，需要使用双引号"或者反斜杠\。下面是一些用例

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

EXPOSE：EXPOSE <port> [<port>...]

EXPOSE 指令告诉 Docker 容器运行时监听指定网络端口。EXPOSE 指令并不会使得容器的端口可以被容器所在的主机被访问。为了达到能被主机访问的目的，必须使用-p 或者-P 参数

ENV：ENV <key> <value> ENV <key>=<value> ...

ENV 指令设置镜像的环境变量，可在实际启动容器时使用 docker run --env <key>=<value>进行覆盖。

ADD：

有两种形式

ADD <src>... <dest>

第二种形式用于路径或者文件名包含空格的情况

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

如果 src 是文件路径，则必须是相对于构建上下文 context 的相对路径，且不能引用构建上下文目录之外的内容。dest 必须是绝对路径，或者是工作路径 WORKDIR 的相对路径。如果 dest 不存在，则将自动创建，如果 dest 不以/结尾，则将被认为是一个文件，而不是目录。

COPY：

有两种形式

COPY <src>... <dest>

第二种形式用于路径或者文件名包含空格的情况

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

与 ADD 类似，区别在于 src 不能是网络链接 URL

官方最佳实践中推荐尽可能使用 COPY 命令，最佳使用 ADD 命令的场景，就是需要自动解压的的场景。另外使用 ADD 会导致镜像构建存失效，从而令镜像构建变得比较缓慢

ENTRYPOINT:

有两种形式 exec 形式，推荐形式

ENTRYPOINT ["executable", "param1", "param2"] ,shell 形式，command 是作为 /bin/sh -c 的参数进行执行的，即为 shell 的子进程

ENTRYPOINT command param1 param2 ,ENTRYPOINT 指令允许你指定容器启动时的启动进程。

VOLUME: VOLUME ["/data"]

VOLUME 指令指定了一个挂载点，并给该挂载点命名，表明该挂载点的数据卷来着于主机的某个目录或者共享了其他容器的目录，该挂载点的内容不会随镜像的分发而分发。

USER: USER daemon

USER 指令设置启动镜像时的用户或者 UID，随后所有在 Dockerfile 文件内的 RUN，CMD 以及 ENTRYPOINT 指令都将该用户作为执行用户。

WORKDIR: WORKDIR /path/to/workdir

WORKDIR 指令设置工作目录，随后所有在 Dockerfile 文件内的 RUN，CMD 以及 ENTRYPOINT 指令都将该目录作为当前目录，并执行相应的命令。

案例

```
#基础系统信息，基于 ubuntu 14.04 构建的
FROM ubuntu:14.04
MAINTAINER Alex McLain <alex@alexmclain.com>
RUN apt-get -qq update
#安装 apache、hg、php5
RUN apt-get -y install apache2 apache2-utils curl mercurial php5 php5-cli php5-mcrypt
# TODO: Remove
#是的，vim 确实很大，不安装为好
RUN apt-get -y install vim
RUN echo "colorscheme delek" > ~/.vimrc
# Configure hgweb
ADD hg/add.php /etc/default/hgweb/hg/
ADD hg/hgweb.config /etc/default/hgweb/hg/
ADD hg/hgweb.cgi /etc/default/hgweb/hg/
ADD hg/hgusers /etc/default/hgweb/hg/
# Configure Apache
ADD apache/hg.conf /etc/default/hgweb/apache/
RUN rm /etc/apache2/sites-enabled/*
RUN a2enmod rewrite && a2enmod cgi
ADD load-default-scripts /bin/
RUN chmod u+x /bin/load-default-scripts
#创建一个挂载点，本机或其他容器可以将其挂载。启动时用-v 参数进行挂载
VOLUME /var/hg
VOLUME /etc/apache2/sites-available
#暴露的端口号，启动时要通过-p 参数指定
EXPOSE 80
#启动时执行的命令
```

```
CMD load-default-scripts && service apache2 start && /bin/bash
```

镜像构建上下文(Context)

docker build [选项] <上下文路径/URL/->

docker build -t nginx:v3 .

如果注意，会看的 docker build 命令最后有一个.。.表示当前目录，而 Dockerfile 就在当前目录，因此不少初学者以为这个路径是在制定 Dockerfile 所在路径，这么理解其实不准确。如果对应上面的命令格式，你可能会发现，这是指定上下文路径

首先我们要理解 docker build 的工作原理。Docker 在运行时分为 Docker 引擎和客户端工具。Docker 的引擎提供了一组 REST API，被称为 Docker Remote API，而如 docker 命令这样的客户端工具，则是通过这组 API 与 Docker 引擎交换，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种 docker 功能，但实际上，一切都是使用远程调用形式在服务端(Docker 引擎)完成。也因为这种 C/S 设计，让我们操作远程服务器的 Docker 引擎变得轻而易举

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进行镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非本地构建，而是在服务端，也就是 Docker 引擎中构建的

这就引入上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件

如果在 Dockerfile 中这么写： COPY ./package.json /app/

这并不是要复制执行 docker build 命令所在目录下的 package.json，也不是复制 Dockerfile 所在目录下的 package.json，而是复制上下文目录下的 package.json

因此，COPY 这类指令中的源文件的路径都是相对路径。这也就是初学者经常会问的为什么 COPY ../package.json /app/或者 COPY /opt/xxx /app/无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要哪些文件，应该将它们复制到上下文目录中去

现在就可以理解刚才的命令 docker build –t nginx:v3 . 中的这个. ，实际上是指定上下文的目录，docker build 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像

一般来说，应该将 Dockerfile 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用.gitignore 一样的语法写一个.dockerignore，该文件用于提出不需要作为上下文传递给 Docker 引擎的

在默认情况下，如果不额外指定 Dockerfile 的话，会将上下文目录下的名为 Dockerfile 的文件作为 Dockerfile。实际上 Dockerfile 的文件名并不要求必须为 Dockerfile，而且并不要求必须位于上下文目录中，比如可以用 –f ../Dockerfile.php 参数指定某个文件作为 Dockerfile

直接使用 GIT repo 进行构建

Docker buid 还支持从 URL 构建，比如可以直接从 GIT repo 中构建:

docker build https://github.com/twang2218/gitlab-ce-zh.git#:8.14

如果所给出的 URL 不是这个 GIT repo，而是个 tar 压缩包，那么 Docker 引擎会下载这个包，并自动解压，以其作为上下文，开始构建

注意事项：

对于容器而言，其启动程序就是容器应用程序，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其他辅助进程不是它需要关心的东西。

例如 service nginx start 命令，则是希望 upstart 来以后台守护进程形式启动 nginx 服务。而 CMD service nginx start 会被理解为 CMD[“sh”, ”c”, ” service nginx start”]，因此主进程实际上是 sh。那么当 service nginx start 命令结束以后，sh 也就结束了，sh 做完主进程退出，自然就会令容器退出。正确的做法是直接执行 nginx 可执行文件，并要求以前台形式运行：

CMD[“nginx”, ”-g”, ” daemon off;”]

在 Shell 中，连续两行是同一个进程环境，因此前一个命令修改的内存状态，会直接影响后一个命令;而在 Dockerfile 中，这两行 RUN 命令的执行环境根本不同，是两个完全不同的容器。

操作 Docker 容器

当利用 `docker run` 来创建容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载
- 利用镜像创建并启动一个容器
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个 ip 地址给容器
- 执行用户指定的应用程序
- 执行完毕后容器被终止

Docker 数据管理

在容器中管理数据主要有两种方式：

数据卷（Data volumes）

数据卷容器（Data volume containers）

数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

数据卷可以在容器之间共享和重用

对数据卷的修改会立马生效

对数据卷的更新，不会影响镜像

数据卷默认会一直存在，即使容器被删除

***注意：**数据卷的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的数据卷。

创建一个数据卷

在用 docker run 命令的时候，使用 -v 标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用可以挂载多个数据卷。

下面创建一个名为 web 的容器，并加载一个数据卷到容器的 /webapp 目录。

$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py

***注意：**也可以在 Dockerfile 中使用 VOLUME 来添加一个或者多个新的卷到由该镜像创建的任意容器。

删除数据卷

数据卷是被设计用来持久化数据的，它的生命周期独立于容器，Docker 不会在容器被删除后自动删除数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的数据

卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 docker rm -v 这个命令。无主的数据卷可能会占据很多空间，要清理会很麻烦。Docker 官方正在试图解决这个问题，相关工作的进度可以查看这个 PR

挂载一个主机目录作为数据卷

使用 -v 标记也可以指定挂载一个本地主机的目录到容器中去

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

Docker 挂载数据卷的默认权限是读写，用户也可以通过 :ro 指定为只读。

$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py

加了 :ro 之后，就挂载为只读了。

查看数据卷的具体信息

在主机里使用以下命令可以查看指定容器的信息

$ docker inspect web

挂载一个本地主机文件作为数据卷

-v 标记也可以从主机挂载单个文件到容器中

$ sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/bash

这样就可以记录在容器输入过的命令了。

*注意：如果直接挂载一个文件，很多文件编辑工具，包括 vi 或者 sed --in-place，可能会造成文件 inode 的改变，从 Docker 1.1 .0 起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。

数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。

首先，创建一个名为 dbdata 的数据卷容器：

$ sudo docker run -d -v /dbdata --name dbdata training/postgres echo Data-only container for postgres

然后，在其他容器中使用 --volumes-from 来挂载 dbdata 容器中的数据卷。

$ sudo docker run -d --volumes-from dbdata --name db1 training/postgres $ sudo docker run -d --volumes-from dbdata --name db2 training/postgres

可以使用超过一个的 --volumes-from 参数来指定从多个容器挂载不同的数据卷。 也可以从其他已经挂载了数据卷的容器来级联挂载数据卷。

$ sudo docker run -d --name db3 --volumes-from db1 training/postgres

***注意：**使用 --volumes-from 参数所挂载数据卷的容器自己并不需要保持在运行状态。

如果删除了挂载的容器（包括 dbdata、db1 和 db2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 docker rm -v 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。利用数据卷容器来备份、

恢复、迁移数据卷

可以利用数据卷对其中的数据进行进行备份、恢复和迁移。

备份

首先使用 --volumes-from 标记来创建一个加载 dbdata 容器卷的容器，并从主机挂载当前目录到容器的 /backup 目录。命令如下：

$ sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

容器启动后，使用了 tar 命令来将 dbdata 卷备份为容器中 /backup/backup.tar 文件，也就

是主机当前目录下的名为 backup.tar 的文件。

恢复

如果要恢复数据到一个容器，首先创建一个带有空数据卷的容器 dbdata2。

$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash

然后创建另一个容器，挂载 dbdata2 容器卷中的数据卷，并使用 untar 解压备份文件到挂载的容器卷中。

$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar xvf /backup/backup.tar

为了查看/验证恢复的数据，可以再启动一个容器挂载同样的容器卷来查看

$ sudo docker run --volumes-from dbdata2 busybox /bin/ls /dbdata

容器网络

外部访问容器

容器中可以运行一些网络应用，要让外部也可以访问这些应用，**可以通过 -P 或 -p 参数来指定端口映射。**

当使用 -P 标记时，Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。

使用 docker ps 可以看到，本地主机的 49155 被映射到了容器的 5000 端口。此时访问本机的 49155 端口即可访问容器内 web 应用提供的界面。

$ sudo docker run -d -P training/webapp python app.py $ sudo docker ps -l CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES bc533791f3f5 training/webapp:latest python app.py 5 seconds ago Up 2 seconds 0.0.0.0:49155->5000/tcp nostalgic_morse

同样的，可以通过 docker logs 命令来查看应用的信息。

$ sudo docker logs -f nostalgic_morse * Running on http://0.0.0.0:5000/ 10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 - 10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -

-p（小写的）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有 ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort。

映射所有接口地址

使用 hostPort:containerPort 格式本地的 5000 端口映射到容器的 5000 端口，可以执行

$ sudo docker run -d -p 5000:5000 training/webapp python app.py

此时默认会绑定本地所有接口上的所有地址。

映射到指定地址的指定端口

可以使用 ip:hostPort:containerPort 格式指定映射使用一个特定地址，比如 localhost 地址 127.0.0.1

$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py

映射到指定地址的任意端口

使用 ip::containerPort 绑定 localhost 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口。

$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py

还可以使用 udp 标记来指定 udp 端口

$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py

查看映射端口配置

使用 docker port 来查看当前映射的端口配置，也可以查看到绑定的地址

$ docker port nostalgic_morse 5000 127.0.0.1:49155.

注意：容器有自己的内部网络和 ip 地址（使用 docker inspect 可以获取所有的变量，Docker 还可以有一个可变的网络配置。）

-p 标记可以多次使用来绑定多个端口

例如

$ sudo docker run -d -p 5000:5000 -p 3000:80 training/webapp python app.py

容器互联

容器的连接（linking）系统是除了端口映射外，另一种跟容器中应用交互的方式。该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

自定义容器命名

连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器命名。虽然当创建容器的时候，系统默认会分配一个名字。自定义命名容器有 2 个好处：

自定义的命名，比较好记，比如一个 web 应用容器我们可以给它起名叫 web

当要连接其他容器时候，可以作为一个有用的参考点，比如连接 web 容器到 db 容器

使用 --name 标记可以为容器自定义命名。

$ sudo docker run -d -P --name web training/webapp python app.py

使用 docker ps 来验证设定的命名。

$ sudo docker ps -l CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES aed84ee21bde training/webapp:latest python app.py 12 hours ago Up 2 seconds 0.0.0.0:49154->5000/tcp web

也可以使用 docker inspect 来查看容器的名字

$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde /web

注意：容器的名称是唯一的。如果已经命名了一个叫 web 的容器，当你要再次使用 web 这个名称的时候，需要先用 docker rm 来删除之前创建的同名容器。

在执行 docker run 的时候如果添加 --rm 标记，则容器在终止后会立刻删除。注意，--rm 和 -d 参数不能同时使用。

容器互联

使用 --link 参数可以让容器之间安全的进行交互。

下面先创建一个新的数据库容器。

$ sudo docker run -d --name db training/postgres

删除之前创建的 web 容器

$ docker rm -f web

然后创建一个新的 web 容器，并将它连接到 db 容器

$ sudo docker run -d -P --name web --link db:db training/webapp python app.py

此时，db 容器和 web 容器建立互联关系。

--link 参数的格式为 --link name:alias，其中 name 是要链接的容器的名称，alias 是这个连接的别名。

使用 docker ps 来查看容器的连接

$ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 349169744e49 training/postgres:latest su postgres -c '/usr About a minute ago Up About a minute 5432/tcp db, web/db aed84ee21bde training/webapp:latest python app.py 16 hours ago Up 2 minutes 0.0.0.0:49154->5000/tcp web

可以看到自定义命名的容器，db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主

主机上。在启动 db 容器的时候并没有使用 -p 和 -P 标记，从而避免了暴露数据库端口到外部网络上。

Docker 通过 2 种方式为容器公开连接信息：

环境变量

更新 /etc/hosts 文件

使用 env 命令来查看 web 容器的环境变量

```
$ sudo docker run --rm --name web2 --link db:db training/webapp env . . .
DB_NAME=/web2/db DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432 DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432 DB_PORT_5000_TCP_ADDR=172.17.0.5 . . .
```

其中 DB_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。

除了环境变量，Docker 还添加 host 信息到父容器的 /etc/hosts 的文件。下面是父容器 web 的 hosts 文件

```
$ sudo docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts 172.17.0.7 aed84ee21bde . . . 172.17.0.5 db
```

这里有 2 个 hosts，第一个是 web 容器，web 容器用 id 作为他的主机名，第二个是 db 容器的 ip 和主机名。 可以在 web 容器中安装 ping 命令来测试跟 db 容器的连通。

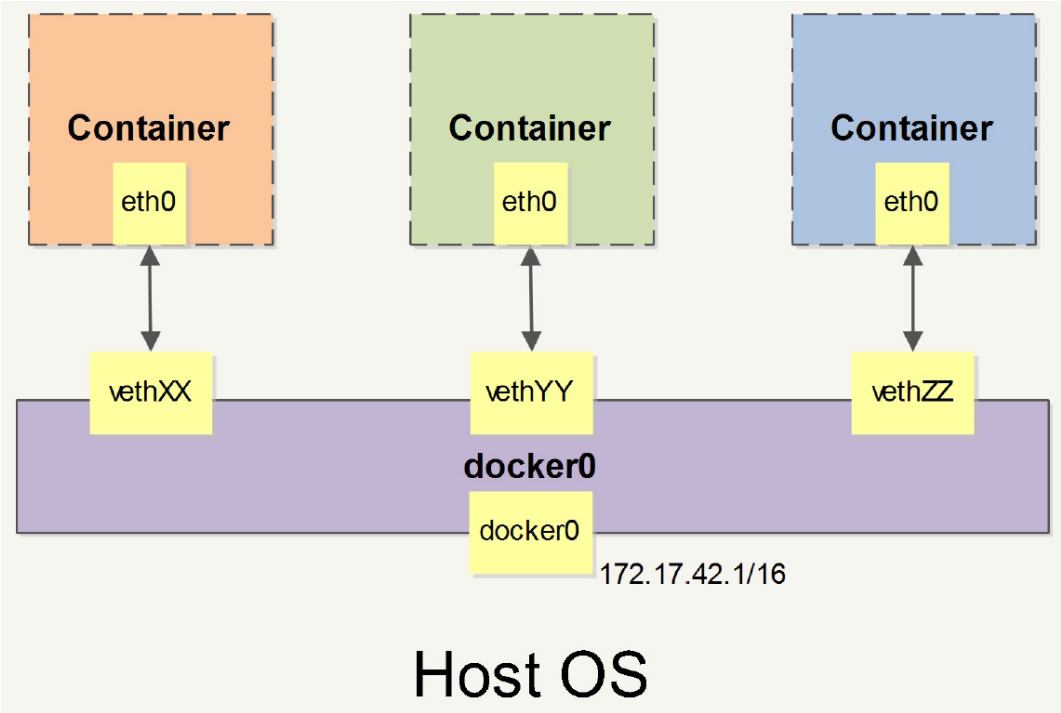
```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db PING db (172.17.0.5): 48 data bytes 56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms 56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms 56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

用 ping 来测试 db 容器，它会解析成 172.17.0.5。 *注意：官方的 ubuntu 镜像默认没有安装 ping，需要自行安装。

用户可以链接多个父容器到子容器，比如可以链接多个 web 到 db 容器上。

高级网络配置

当 Docker 启动时，会自动在主机上创建一个 docker0 虚拟网桥，实际上是 Linux 的一个 bridge，可以理解为一个软件交换机。它会挂载到它的网口之间进行转发



快速配置指南

下面是一个跟 Docker 网络相关的命令列表。

其中有些命令选项只有在 Docker 服务启动的时候才能配置，而且不能马上生效。

-b BRIDGE or --bridge=BRIDGE --指定容器挂载的网桥

--bip=CIDR --定制 docker0 的掩码

-H SOCKET... or --host=SOCKET... --Docker 服务端接收命令的通道

--icc=true|false --是否支持容器之间进行通信

--ip-forward=true|false --请看下文容器之间的通信

--iptables=true|false --是否允许 Docker 添加 iptables 规则

--mtu=BYTES --容器网络中的 MTU

下面 2 个命令选项既可以在启动服务时指定，也可以 Docker 容器启动（docker run）时候指定。在 Docker 服务启动的时候指定则会成为默认值，后面执行 docker run 时可以覆盖设置的默认值。

--dns=IP_ADDRESS... --使用指定的 DNS 服务器

--dns-search=DOMAIN... --指定 DNS 搜索域

最后这些选项只有在 docker run 执行时使用，因为它是针对容器的特性内容。

-h HOSTNAME or --hostname=HOSTNAME --配置容器主机名

--link=CONTAINER_NAME:ALIAS --添加到另一个容器的连接

--net=bridge|none|container:NAME_or_ID|host --配置容器的桥接模式

-p SPEC or --publish=SPEC --映射容器端口到宿主主机

-P or --publish-all=true|false --映射容器所有端口到宿主主机

Docker 底层实现

Docker 底层的核心技术包括 Linux 上的命名空间(Namespaces)、控制组(Control Group)、Union 文件系统(Union file systems)和容器格式(Container format)

NameSpace：内核级别，环境隔离

名称空间是 Linux 内核一个强大的特性。每个容器都有自己单独的名称空间，运行在其中的应用都像是在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响

PID NameSpace：linux2.6.24，PID 隔离

不同用户的进程就是通过 pid 命名空间隔离开的，且不同命名空间中可以有相同 pid。所有的 LXC 进程在 Docker 中的父进程为 Docker 进程。每个 LXC 进程具有不同的命名空间。同时由于允许嵌套，因此可以很方便的实现嵌套的 Docker 容器

Network NameSpace：linux2.6.29，网络设备、网络栈、端口等网络资源隔离

有了 pid 命名空间，每个命名空间中的 pid 能够相互隔离，但是网络端口还是共享 host 的端口。网络隔离是通过 net 命名空间实现的，每个 net 命名空间有独立的网络设备，IP 地址，路由表，/proc/net 目录。这样每个容器的网络就能隔离开来。Docker 默认采用 veth 的方式，将容器中的虚拟网卡同 host 上的一个 Docker 网桥 docker0 连接起来

User NameSpace： linux 3.8，用户和用户组资源隔离

每个容器可以有不同的用户和组 id，也就是说可以在容器内用容器内部的太湖执行程序而非主机上的用户

IPC NameSpace：linux 2.6.19，信号量、消息队列和共享内存的隔离

容器总进程交换才是采用了 Linux 常见的进程间交换方法，包括信号量、消息队列和共享内存等。然而同 VM 不同的是，容器的进程间交互实际上还是 host 上具有相同 pid 命

名空间中的进程间交互，因此需要在 IPC 资源申请时加入命名空间信息，每个 IPC 资源有一个唯一的 32 位 ID

UTS NameSpace：linux 2.6.19，主机名和域名的隔离

UTS(“UNIX Time-sharing System”)命名空间允许每个容器拥有独立的 hostname 和 domain name，使其在屋里上可以被视作一个独立的节点而非主机上的一个进程

Mount NamSpace：linux 2.4.19，挂载点（文件系统）隔离

类似 chroot，将一个进程放在一个特定的目录执行。Mnt 命名空间允许不同命名空间的进程看到的文件结构不同。这样每个命名空间中的进程所看到的文件目录就被隔离开了。同 chroot 不同，每个命名空间中的容器在/proc/mounts 的信息只包含所在命名空间的 mount pront

API：clone(),setns(),unshare();

CGroup:Linux Control Group 控制组 linux 2.6.24

控制组是 Linux 内核的一个特性，主要用来对共享资源进行隔离、限流、审计等。只有能控制分配到容器的资源，才能避免当多个容器同时运行时的对系统资源的竞争。控制组技术最早是由 Googloe 的程序 2006 年提出，Linux 内核自 2.6.24 开始支持。控制组可以提供对容器的内存、CPU、磁盘 I/O 等资源的限制和审计管理

内核级别，现在、控制与一个进程组群的资源

资源：CPU,内存,IO

功能：

资源限制

优先级控制

审计统计

挂起进程，恢复进程

联合文件系统(UnionFS)

联合文件系统是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下

联合文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像(没有父镜像)，可以制作各种具体的应用镜像。另外，不同 Docker 容器就可以共享一些基础的文件系统层，同时加上自己独有的改动层，大大提高了存储的效率

Docker 中使用的 AUFS 就是一种联合文件系统。AUFS 支持为每一个成员目录设定只读、读写和写出权限，同时 AUFS 里有一个类似分层的概念，对只读权限的分支可以逻辑上进行增量地修改(不影响只读部分的)

Docker 目前支持的联合文件系统种类包括 AUFS，btrfs，vfs 和 DeviceMapper

容器格式

最初，Docker 采用了 LXC 中的容器格式。自 1.20 版本开始，Docker 也开始支持新的 libcontainer 格式，并作为默认选项

Docker 网络实现

Docker 的网络实现其实就是利用了 Linux 上的网络命名空间和虚拟网络设备(特别是 veth pair)

Docker 中的网络接口默认都是虚拟的接口。虚拟接口的优势之一是转发效率较高。Linux 通过在内核中进行数据复制来实现虚拟接口之间的数据转发，发送接口的发送缓存中的数据包被之间复制到接收接口的接收缓存中。对于本地系统和容器内系统看来就像是一个正常的

以太网卡，只是它不需要真正同外包网络设备通信，速度要快很多。Docker 容器网络就利用了这项技术。它再本地主叫和容器内分别创建了一个虚拟接口，并让它们彼此连通(这样的一对接口叫做 veth pair)

创建网络参数

Docker 创建一个容器的时候，会执行如下操作：

- 创建一对虚拟接口，分别放到本地主机和新容器中；
- 本地主机一端桥接到默认的 docker0 或指定网桥上，并具有一个唯一的名字，如 veth65f9；
- 容器一端放到新容器中，并修改名字作为 eth0，这个接口只在容器的命名空间可见；
- 从网桥可用地址段中获取一个空闲地址分配给容器的 eth0，并配置默认路由到桥接网卡 veth65f9。

完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

ectd

Etcd 是 CoreOS 团队于 2013 年 6 月发起的一个管理配置信息和服务发现的项目，它的目标是构建一个高可用的分布式键值数据库，基于 Go 语言实现。在分布式系统中，各种服务的配置信息的管理分享，服务的发现是一个很基本同时也是很重要的问题

- 简单：支持 REST 风格的 HTTP+JSON API
- 安全：支持 HTTPS 方式的访问
- 快速：支持并发 1k/s 的写操作
- 可靠：支持分布式结构，基于 Raft 的一致性算法

热门镜像介绍

Ubuntu

最小化使用方法

docker run --name some-ubuntu -i -t ubuntu

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/ubuntu 查看。

Centos

最小化使用方法

docker run --name some-centos -i -t centos bash

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/centos 查看。

MySQL

最小化使用方法

docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql

docker run --name some-app --link some-mysql:mysql -d application-that-uses-mysql

docker run -it --link some-mysql:mysql --rm mysql sh -c 'exec mysql -h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/mysql 查看。

MongoDB

最小化使用方法

docker run --name some-mongo -d mongo

docker run --name some-app --link some-mongo:mongo -d application-that-uses-mongo

docker run -it --link some-mongo:mongo --rm mongo sh -c 'exec mongo "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/test"'

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/mongo 查看。

Redis

最小化使用方法

docker run --name some-redis -d redis

启动持久化存储

docker run --name some-redis -d redis redis-server --appendonly yes

默认数据存储位置在 VOLUME/data。可以使用 --volumes-from some-volume-container 或 -v /docker/host/dir:/data 将数据存放到本地。

使用其他应用连接到容器，可以用

docker run --name some-app --link some-redis:redis -d application-that-uses-redis

或者通过 redis-cli

docker run -it --link some-redis:redis --rm redis sh -c 'exec redis-cli -h "$REDIS_PORT_6379_TCP_ADDR" -p "$REDIS_PORT_6379_TCP_PORT"'

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/redis 查看

Nginx

最小化使用方法

下面的命令将作为一个静态页面服务器启动。

docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx

用户也可以不使用这种映射方式，通过利用 Dockerfile 来直接将静态页面内容放到镜像中，内容为

FROM nginx

COPY static-html-directory /usr/share/nginx/html

之后生成新的镜像，并启动一个容器。

docker build -t some-content-nginx .

docker run --name some-nginx -d some-content-nginx

开放端口，并映射到本地的 8080 端口。

docker run --name some-nginx -d -p 8080:80 some-content-nginx

Nginx 的默认配置文件路径为 /etc/nginx/nginx.conf，可以通过映射它来使用本地的配置文件，例如

docker run --name some-nginx -v /some/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx

使用配置文件时，为了在容器中正常运行，需要保持 daemon off;

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/nginx 查看

WordPress

最小化使用方法

启动容器需要 MySQL 的支持，默认端口为 80。

docker run --name some-wordpress --link some-mysql:mysql -d wordpress

启动 WordPress 容器时可以指定的一些环境参数包括：

-e WORDPRESS_DB_USER=... 缺省为“root”

-e WORDPRESS_DB_PASSWORD=... 缺省为连接 mysql 容器的环境变量 MYSQL_ROOT_PASSWORD 的值

-e WORDPRESS_DB_NAME=... 缺省为“wordpress”

-e WORDPRESS_AUTH_KEY=..., -e WORDPRESS_SECURE_AUTH_KEY=..., -e WORDPRESS_LOGGED_IN_KEY=..., -e WORDPRESS_NONCE_KEY=..., -e WORDPRESS_AUTH_SALT=..., -e WORDPRESS_SECURE_AUTH_SALT=..., -e WORDPRESS_LOGGED_IN_SALT=..., -e WORDPRESS_NONCE_SALT=... 缺省为随机 sha1 串

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/wordpress 查看

Node.js

最小化使用方法

在项目中创建一个 Dockerfile。

```
FROM node:0.10-onbuild
# replace this with your application's default port
EXPOSE 8888
```

然后创建镜像，并启动容器。

```
docker build -t my-nodejs-app
docker run -it --rm --name my-running-app my-nodejs-app
```

也可以直接运行一个简单容器。

docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp node:0.10 node your-daemon-or-scri

Dockerfile

请到 https://github.com/docker-library/docs/tree/master/node 查看。

客户端命令

可以通过 man docker-COMMAND 或 docker help COMMAND 来查看这些命令的具体用法。

attach：依附到一个正在运行的容器中；

build：从一个 Dockerfile 创建一个镜像；

commit：从一个容器的修改中创建一个新的镜像；

cp：在容器和本地宿主系统之间复制文件中；

create：创建一个新容器，但并不运行它；

diff：检查一个容器内文件系统的修改，包括修改和增加；

events：从服务端获取实时的事件；

exec：在运行的容器内执行命令；

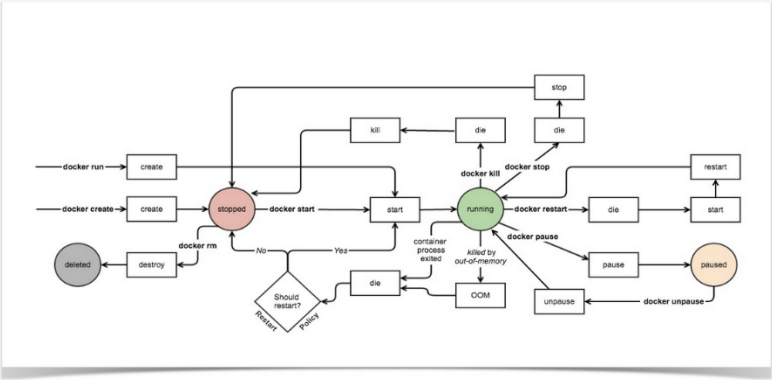
export：导出容器内容为一个 tar 包；

history：显示一个镜像的历史信息；
images：列出存在的镜像；
import：导入一个文件（典型为 tar 包）路径或目录来创建一个本地镜像；
info：显示一些相关的系统信息；
inspect：显示一个容器的具体配置信息；
kill：关闭一个运行中的容器 (包括进程和所有相关资源)；
load：从一个 tar 包中加载一个镜像；
login：注册或登录到一个 Docker 的仓库服务器；
logout：从 Docker 的仓库服务器登出；
logs：获取容器的 log 信息；
network：管理 Docker 的网络，包括查看、创建、删除、挂载、卸载等；
node：管理 swarm 集群中的节点，包括查看、更新、删除、提升/取消管理节点等；
pause：暂停一个容器中的所有进程；

Docker 命令查询

port：查找一个 nat 到一个私有网口的公共口；
ps：列出主机上的容器；
pull：从一个 Docker 的仓库服务器下拉一个镜像或仓库；
push：将一个镜像或者仓库推送到一个 Docker 的注册服务器；
rename：重命名一个容器；
restart：重启一个运行中的容器；
rm：删除给定的若干个容器；
rmi：删除给定的若干个镜像；
run：创建一个新容器，并在其中运行给定命令；
save：保存一个镜像为 tar 包文件；
search：在 Docker index 中搜索一个镜像；
service：管理 Docker 所启动的应用服务，包括创建、更新、删除等；
start：启动一个容器；
stats：输出（一个或多个）容器的资源使用统计信息；
stop：终止一个运行中的容器；
swarm：管理 Docker swarm 集群，包括创建、加入、退出、更新等；
tag：为一个镜像打标签；
top：查看一个容器中的正在运行的进程信息；
unpause：将一个容器内所有的进程从暂停状态中恢复；
update：更新指定的若干容器的配置信息；
version：输出 Docker 的版本信息；
volume：管理 Docker volume，包括查看、创建、删除等；
wait：阻塞直到一个容器终止，然后输出它的退出符。

container 事件状态图



docker 命令分布图