Docker 入门教程

2013年发布至今， Docker 一直广受瞩目，被认为可能会改变软件行业。

但是，许多人并不清楚 Docker 到底是什么，要解决什么问题，好处又在哪里？本文就来详细解释，帮助大家理解它，还带有简单易懂的实例，教你如何将它用于日常开发。

一、环境配置的难题

软件开发最大的麻烦事之一，就是环境配置。用户计算机的环境都不相同，你怎么知道自家的软件，能在那些机器跑起来？

用户必须保证两件事：操作系统的设置，各种库和组件的安装。只有它们都正确，软件才能运行。举例来说，安装一个 Python 应用，计算机必须有 Python 引擎，还必须有各种依赖，可能还要配置环境变量。

如果某些老旧的模块与当前环境不兼容，那就麻烦了。开发者常常会说："它在我的机器可以跑了"（It works on my machine），言下之意就是，其他机器很可能跑不了。

环境配置如此麻烦，换一台机器，就要重来一次，旷日费时。很多人想到，能不能从根本上解决问题，软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。

二、虚拟机

虚拟机（virtual machine）就是带环境安装的一种解决方案。它可以在一种操作系统里面运行另一种操作系统，比如在 Windows 系统里面运行 Linux 系统。应用程序对此毫无感知，因为虚拟机看上去跟真实系统一模一样，而对于底层系统来说，虚拟机就是一个普通文件，不需要了就删掉，对其他部分毫无影响。

虽然用户可以通过虚拟机还原软件的原始环境。但是，这个方案有几个缺点。

（1）资源占用多

虚拟机会独占一部分内存和硬盘空间。它运行的时候，其他程序就不能使用这些资源了。哪怕虚拟机里面的应用程序，真正使用的内存只有 1MB，虚拟机依然需要几百 MB 的内存才能运行。

（2）冗余步骤多

虚拟机是完整的操作系统，一些系统级别的操作步骤，往往无法跳过，比如用户登录。

（3）启动慢

启动操作系统需要多久，启动虚拟机就需要多久。可能要等几分钟，应用程序才能真正运行。

三、Linux 容器

由于虚拟机存在这些缺点，Linux 发展出了另一种虚拟化技术：Linux 容器（Linux Containers，缩写为 LXC）。

Linux 容器不是模拟一个完整的操作系统，而是对进程进行隔离。或者说，在正常进程的外面套了一个保护层。对于容器里面的进程来说，它接触到的各种资源都是虚拟的，从而实现与底层系统的隔离。

由于容器是进程级别的，相比虚拟机有很多优势。

（1）启动快

容器里面的应用，直接就是底层系统的一个进程，而不是虚拟机内部的进程。所以，启动容器相当于启动本机的一个进程，而不是启动一个操作系统，速度就快很多。

（2）资源占用少

容器只占用需要的资源，不占用那些没有用到的资源；虚拟机由于是完整的操作系统，不可避免要占用所有资源。另外，多个容器可以共享资源，虚拟机都是独享资源。

（3）体积小

容器只要包含用到的组件即可，而虚拟机是整个操作系统的打包，所以容器文件比虚拟机文件要小很多。

总之，容器有点像轻量级的虚拟机，能够提供虚拟化的环境，但是成本开销小得多。

四、Docker 是什么？

Docker 属于 Linux 容器的一种封装，提供简单易用的容器使用接口。它是目前最流行的 Linux 容器解决方案。

Docker 将应用程序与该程序的依赖，打包在一个文件里面。运行这个文件，就会生成一个虚拟容器。程序在这个虚拟容器里运行，就好像在真实的物理机上运行一样。有了 Docker，就不用担心环境问题。

总体来说，Docker 的接口相当简单，用户可以方便地创建和使用容器，把自己的应用放入容器。容器还可以进行版本管理、复制、分享、修改，就像管理普通的代码一样。

五、Docker 的用途

Docker 的主要用途，目前有三大类。

（1）提供一次性的环境。比如，本地测试他人的软件、持续集成的时候提供单元测试和构建的环境。

（2）提供弹性的云服务。因为 Docker 容器可以随开随关，很适合动态扩容和缩容。

（3）组建微服务架构。通过多个容器，一台机器可以跑多个服务，因此在本机就可以模拟出微服务架构。

六、Docker 的安装

Docker 是一个开源的商业产品，有两个版本：社区版（Community Edition，缩写为 CE）和企业版（Enterprise Edition，缩写为 EE）。企业版包含了一些收费服务，个人开发者一般用不到。下面的介绍都针对社区版。

Docker CE 的安装请参考官方文档。

> - Mac
> - Windows
> - Ubuntu
> - Debian
> - CentOS
> - Fedora
> - 其他 Linux 发行版

安装完成后，运行下面的命令，验证是否安装成功。

```
$ docker version
# 或者
$ docker info
```

Docker 需要用户具有 sudo 权限，为了避免每次命令都输入`sudo`，可以把用户加入 Docker 用户组（官方文档）。

```
$ sudo usermod -aG docker $USER
```

Docker 是服务器----客户端架构。命令行运行`docker`命令的时候，需要本机有 Docker 服务。如果这项服务没有启动，可以用下面的命令启动（官方文档）。

```
# service 命令的用法
$ sudo service docker start

# systemctl 命令的用法
$ sudo systemctl start docker
```

六、image 文件

Docker 把应用程序及其依赖，打包在 image 文件里面。只有通过这个文件，才能生成 Docker 容器。image 文件可以看作是容器的模板。Docker 根据 image 文件生成容器的实例。同一个 image 文件，可以生成多个同时运行的容器实例。

image 是二进制文件。实际开发中，一个 image 文件往往通过继承另一个 image 文件，加上一些个性化设置而生成。举例来说，你可以在 Ubuntu 的 image 基础上，往里面加入 Apache 服务器，形成你的 image。

```
# 列出本机的所有 image 文件。
$ docker image ls

# 删除 image 文件
$ docker image rm [imageName]
```

image 文件是通用的，一台机器的 image 文件拷贝到另一台机器，照样可以使用。一般来说，为了节省时间，我们应该尽量使用别人制作好的 image 文件，而不是自己制作。即使要定制，也应该基于别人的 image 文件进行加工，而不是从零开始制作。

为了方便共享，image 文件制作完成后，可以上传到网上的仓库。Docker 的官方仓库 Docker Hub 是最重要、最常用的 image 仓库。此外，出售自己制作的 image 文件也是可以的。

七、实例：hello world

下面，我们通过最简单的 image 文件"hello world"，感受一下 Docker。

需要说明的是，国内连接 Docker 的官方仓库很慢，还会断线，需要将默认仓库改成国内的镜像网站，具体的修改方法在下一篇文章的第一节。有需要的朋友，可以先看一下。

首先，运行下面的命令，将 image 文件从仓库抓取到本地。

```
$ docker image pull library/hello-world
```

上面代码中，`docker image pull`是抓取 image 文件的命令。`library/hello-world`是 image 文件在仓库里面的位置，其中`library`是 image 文件所在的组，`hello-world`是 image 文件的名字。

由于 Docker 官方提供的 image 文件，都放在`library`组里面，所以它的是默认组，可以省略。因此，上面的命令可以写成下面这样。

```
$ docker image pull hello-world
```

抓取成功以后，就可以在本机看到这个 image 文件了。

```
$ docker image ls
```

现在，运行这个 image 文件。

```
$ docker container run hello-world
```

`docker container run`命令会从 image 文件，生成一个正在运行的容器实例。

注意，`docker container run`命令具有自动抓取 image 文件的功能。如果发现本地没有指定的 image 文件，就会从仓库自动抓取。因此，前面的`docker image pull`命令并不是必需的步骤。

如果运行成功，你会在屏幕上读到下面的输出。

```
$ docker container run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

... ...
```

输出这段提示以后，`hello world`就会停止运行，容器自动终止。

有些容器不会自动终止，因为提供的是服务。比如，安装运行 Ubuntu 的 image，就可以在命令行体验 Ubuntu 系统。

```
$ docker container run -it ubuntu bash
```

对于那些不会自动终止的容器，必须使用`docker container kill`命令手动终止。

```
$ docker container kill [containID]
```

八、容器文件

image 文件生成的容器实例，本身也是一个文件，称为容器文件。也就是说，一旦容器生成，就会同时存在两个文件： image 文件和容器文件。而且关闭容器并不会删除容器文件，只是容器停止运行而已。

```
# 列出本机正在运行的容器
$ docker container ls

# 列出本机所有容器，包括终止运行的容器
$ docker container ls --all
```

上面命令的输出结果之中，包括容器的 ID。很多地方都需要提供这个 ID，比如上一节终止容器运行的`docker container kill`命令。

终止运行的容器文件，依然会占据硬盘空间，可以使用`docker container rm`命令删除。

```
$ docker container rm [containerID]
```

运行上面的命令之后，再使用`docker container ls --all`命令，就会发现被删除的容器文件已经消失了。

九、Dockerfile 文件

学会使用 image 文件以后，接下来的问题就是，如何可以生成 image 文件？如果你要推广自己的软件，势必要自己制作 image 文件。

这就需要用到 Dockerfile 文件。它是一个文本文件，用来配置 image。Docker 根据 该文件生成二进制的 image 文件。

下面通过一个实例，演示如何编写 Dockerfile 文件。

十、实例：制作自己的 Docker 容器

下面我以 koa-demos 项目为例，介绍怎么写 Dockerfile 文件，实现让用户在 Docker 容器里面运行 Koa 框架。

作为准备工作，请先下载源码。

```
$ git clone https://github.com/ruanyf/koa-demos.git
$ cd koa-demos
```

10.1 编写 Dockerfile 文件

首先，在项目的根目录下，新建一个文本文件`.dockerignore`，写入下面的内容。

```
.git
node_modules
npm-debug.log
```

上面代码表示，这三个路径要排除，不要打包进入 image 文件。如果你没有路径要排除，这个文件可以不新建。

然后，在项目的根目录下，新建一个文本文件 Dockerfile，写入下面的内容。

```
FROM node:8.4
COPY . /app
WORKDIR /app
RUN npm install --registry=https://registry.npm.taobao.org
EXPOSE 3000
```

上面代码一共五行，含义如下。

- `FROM node:8.4`：该 image 文件继承官方的 node image，冒号表示标签，这里标签是`8.4`，即8.4版本的 node。
- `COPY . /app`：将当前目录下的所有文件（除了`.dockerignore`排除的路径），都拷贝进入 image 文件的`/app`目录。
- `WORKDIR /app`：指定接下来的工作路径为`/app`。
- `RUN npm install`：在`/app`目录下，运行`npm install`命令安装依赖。注意，安装后所有的依赖，都将打包进入 image 文件。
- `EXPOSE 3000`：将容器 3000 端口暴露出来， 允许外部连接这个端口。

10.2 创建 image 文件

有了 Dockerfile 文件以后，就可以使用`docker image build`命令创建 image 文件了。

```
$ docker image build -t koa-demo .
# 或者
$ docker image build -t koa-demo:0.0.1 .
```

上面代码中，`-t`参数用来指定 image 文件的名字，后面还可以用冒号指定标签。如果不指定，默认的标签就是`latest`。最后的那个点表示 Dockerfile 文件所在的路径，上例是当前路径，所以是一个点。

如果运行成功，就可以看到新生成的 image 文件`koa-demo`了。

```
$ docker image ls
```

10.3 生成容器

`docker container run`命令会从 image 文件生成容器。

```
$ docker container run -p 8000:3000 -it koa-demo /bin/bash
# 或者
$ docker container run -p 8000:3000 -it koa-demo:0.0.1 /bin/bash
```

上面命令的各个参数含义如下：

- `-p`参数：容器的 3000 端口映射到本机的 8000 端口。
- `-it`参数：容器的 Shell 映射到当前的 Shell，然后你在本机窗口输入的命令，就会传入容器。
- `koa-demo:0.0.1`：image 文件的名字（如果有标签，还需要提供标签，默认是 latest 标签）。
- `/bin/bash`：容器启动以后，内部第一个执行的命令。这里是启动 Bash，保证用户可以使用 Shell。

如果一切正常，运行上面的命令以后，就会返回一个命令行提示符。

```
root@66d80f4aaf1e:/app#
```

这表示你已经在容器里面了，返回的提示符就是容器内部的 Shell 提示符。执行下面的命令。

```
root@66d80f4aaf1e:/app# node demos/01.js
```

这时，Koa 框架已经运行起来了。打开本机的浏览器，访问 http://127.0.0.1:8000，网页显示"Not Found"，这是因为这个 demo没有写路由。

这个例子中，Node 进程运行在 Docker 容器的虚拟环境里面，进程接触到的文件系统和网络接口都是虚拟的，与本机的文件系统和网络接口是隔离的，因此需要定义容器与物理机的端口映射（map）。

现在，在容器的命令行，按下 Ctrl + c 停止 Node 进程，然后按下 Ctrl + d （或者输入 exit）退出容器。此外，也可以用`docker container kill`终止容器运行。

```
# 在本机的另一个终端窗口，查出容器的 ID
$ docker container ls

# 停止指定的容器运行
$ docker container kill [containerID]
```

容器停止运行之后，并不会消失，用下面的命令删除容器文件。

```
# 查出容器的 ID
$ docker container ls --all

# 删除指定的容器文件
$ docker container rm [containerID]
```

也可以使用`docker container run`命令的`--rm`参数，在容器终止运行后自动删除容器文件。

```
$ docker container run --rm -p 8000:3000 -it koa-demo /bin/bash
```

10.4 CMD 命令

上一节的例子里面，容器启动以后，需要手动输入命令`node demos/01.js`。我们可以把这个命令写在 Dockerfile 里面，这样容器启动以后，这个命令就已经执行了，不用再手动输入了。

```
FROM node:8.4
COPY . /app
WORKDIR /app
RUN npm install --registry=https://registry.npm.taobao.org
EXPOSE 3000
CMD node demos/01.js
```

上面的 Dockerfile 里面，多了最后一行`CMD node demos/01.js`，它表示容器启动后自动执行`node demos/01.js`。

你可能会问，`RUN`命令与`CMD`命令的区别在哪里？简单说，`RUN`命令在 image 文件的构建阶段执行，执行结果都会打包进入 image 文件；`CMD`命令则是在容器启动后执行。另外，一个 Dockerfile 可以包含多个`RUN`命令，但是只能有一个`CMD`命令。

注意，指定了`CMD`命令以后，`docker container run`命令就不能附加命令了（比如前面的`/bin/bash`），否则它会覆盖`CMD`命令。现在，启动容器可以使用下面的命令。

```
$ docker container run --rm -p 8000:3000 -it koa-demo:0.0.1
```

10.5 发布 image 文件

容器运行成功后，就确认了 image 文件的有效性。这时，我们就可以考虑把 image 文件分享到网上，让其他人使用。

首先，去 hub.docker.com 或 cloud.docker.com 注册一个账户。然后，用下面的命令登录。

```
$ docker login
```

接着，为本地的 image 标注用户名和版本。

```
$ docker image tag [imageName] [username]/[repository]:[tag]
# 实例
$ docker image tag koa-demos:0.0.1 ruanyf/koa-demos:0.0.1
```

也可以不标注用户名，重新构建一下 image 文件。

```
$ docker image build -t [username]/[repository]:[tag] .
```

最后，发布 image 文件。

```
$ docker image push [username]/[repository]:[tag]
```

发布成功以后，登录 hub.docker.com，就可以看到已经发布的 image 文件。

十一、其他有用的命令

docker 的主要用法就是上面这些，此外还有几个命令，也非常有用。

（1）docker container start

前面的`docker container run`命令是新建容器，每运行一次，就会新建一个容器。同样的命令运行两次，就会生成两个一模一样的容器文件。如果希望重复使用容器，就要使用`docker container start`命令，它用来启动已经生成、已经停止运行的容器文件。

```
$ docker container start [containerID]
```

（2）docker container stop

前面的`docker container kill`命令终止容器运行，相当于向容器里面的主进程发出 SIGKILL 信号。而`docker container stop`命令也是用来终止容器运行，相当于向容器里面的主进程发出 SIGTERM 信号，然后过一段时间再发出 SIGKILL 信号。

```
$ bash container stop [containerID]
```

这两个信号的差别是，应用程序收到 SIGTERM 信号以后，可以自行进行收尾清理工作，但也可以不理会这个信号。如果收到 SIGKILL 信号，就会强行立即终止，那些正在进行中的操作会全部丢失。

（3）docker container logs

`docker container logs`命令用来查看 docker 容器的输出，即容器里面 Shell 的标准输出。如果`docker run`命令运行容器的时候，没有使用`-it`参数，就要用这个命令查看输出。

```
$ docker container logs [containerID]
```

（4）docker container exec

`docker container exec`命令用于进入一个正在运行的 docker 容器。如果`docker run`命令运行容器的时候，没有使用`-it`参数，就要用这个命令进入容器。一旦进入了容器，就可以在容器的 Shell 执行命令了。

```
$ docker container exec -it [containerID] /bin/bash
```

（5）docker container cp

`docker container cp`命令用于从正在运行的 Docker 容器里面，将文件拷贝到本机。下面是拷贝到当前目录的写法。

```
 $ docker container cp [containID]:[/path/to/file] .
```