

Desvendando o Dockerfile



Daniel Artine
28/08/2019

COMPARTILHE



Esse artigo faz parte da
Formação DevOps

Resolvi me aventurar no mundo do **Dockerfile** e entender para que ele serve efetivamente, quando usar, o que é, e o que faz cada uma das principais instruções do Dockerfile?

Tudo começa com um Dockerfile

Meu objetivo inicial é rodar um container bem simples que contenha apenas o Ubuntu e tenha o Java instalado. Como posso executar essa tarefa?

Existem duas possibilidades: A primeira é escrever o Dockerfile e criar o container a partir dele. A segunda é utilizar o comando [docker run](#) para que ele busque a imagem na internet e rode o container para nós.

Mas, quando se usa o comando `docker run`, ele vai até o Docker Hub, busca essa imagem e baixa para nós. E como essa imagem foi gerada? Isso mesmo, com um Dockerfile feito por outra pessoa!

O que é o Dockerfile ?

O Dockerfile nada mais é do que **um meio que utilizamos para criar nossas próprias imagens**. Em outras palavras, ele serve como a receita para construir um container, permitindo definir um ambiente personalizado e próprio para meu projeto pessoal ou empresarial.

Há ainda um outro ponto muito interessante que deve ser explorado (vejo muitas pessoas confundirem quando ainda estão aprendendo) para entendermos melhor o conceito e começarmos a compreender o Dockerfile mais a fundo: qual é exatamente a diferença entre um container e uma imagem?

Imagem x Container

Uma imagem nada mais é do que uma representação imutável de como será efetivamente construído um container. Por conta disso, nós não rodamos ou inicializamos imagens, nós fazemos isso com os containers.

O ponto que temos que entender agora é o seguinte: escrevemos um Dockerfile, construímos uma imagem a partir dele executando o comando **docker build**, e, por fim, criamos e rodamos o container com o comando **docker run**. O container é o fim enquanto a imagem é o meio.



Etapas da criação de um container

Escolhi uma pasta no meu computador e nela criei um arquivo chamado Dockerfile (com letra maiúscula e sem extensão mesmo). Nele, adicionei a seguinte linha:

```
FROM ubuntu:18.04
```

Então, iniciei o processo de criação da imagem. Para isso, abri o terminal e acessei a pasta que contém o Dockerfile. Após isso, executei o comando **docker build .** (com o ponto) e o docker começou a construir a imagem a partir do arquivo.

Ao terminar o processo, executei o comando **docker image ls**, e obtive a seguinte saída:

```
danielartine@artine ~/docker$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu 18.04 7698f282e524 2 weeks ago 69.9MB
```

O resultado acima mostra uma imagem a partir do Dockerfile e ela está pronta para rodar um container com o comando **docker run**

Agora que já sei o que é, e para que serve o Dockerfile, decidi aprender qual o propósito de cada uma das instruções desta ferramenta.

Entendendo as instruções do Dockerfile

FROM

A instrução **FROM** é a mais utilizada para a criação de Dockerfiles, sabe o motivo? Simplesmente porque ela é obrigatória 😊

Com essa instrução, pode-se definir qual será o ponto de partida da imagem que criaremos com o nosso Dockerfile, ou seja, se eu quiser utilizar a imagem do Java para produzir meu container, basta que eu especifique para utilizar a imagem do **openjdk** como base. Ficaria algo como:

```
FROM openjdk
```

Mas e caso eu queira criar uma imagem do zero absoluto? Sem me basear em imagem alguma?

Para isso, posso utilizar a imagem **scratch**.

```
FROM scratch
```

Essa imagem nada mais é do que simplesmente... nada! Com ela, consigo criar uma imagem completamente do zero sem utilizar nada de ninguém.

RUN

A instrução **RUN** é bem interessante. Ela pode ser executada uma ou mais vezes e, com ela, posso definir quais serão os comandos executados na etapa de criação de camadas da imagem. Mas como assim?

Temos o seguinte Dockerfile:

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
```

Quando executo o comando `docker build .`, além de baixar a imagem do Ubuntu 18.04 para colocar na minha imagem, o processo de criação também executará os comandos para atualizar os repositórios do Ubuntu através do `apt-get update`, e para instalar o Java 8

utilizara o `apt-get install openjdk-8-jdk-y`. Ele executará uma série de downloads, produzindo o seguinte resultado:

```
Processing triggers for libc-bin (2.27-3ubuntu1) ...
Processing triggers for libgdk-pixbuf2.0-0:amd64 (2.36.11-2) ...
Removing intermediate container 7cb03c8c74e6
--> 41afdc6b059f
Successfully built 41afdc6b059f
danielartine@artine ~$ docker
```

Como resultado em minha máquina, foi gerada uma imagem com o ID **41afdc6b059f** (em sua máquina será algo completamente diferente, não se assuste).

Se criarmos um container a partir dessa imagem, os dois comandos já terão sido executados, já que **esses foram executados no momento de criação da imagem, não da criação do container**. Então qualquer container criado a partir dessa imagem terá o repositório do Ubuntu na mesma versão e o Java 8 instalado!

Para criar um container utilizando a imagem gerada e podendo manter a execução do container, executei o comando passando o ID da imagem gerada na minha máquina através do comando `docker run -it 41afdc6b059f`.

O `-it` serve para rodar o container em modo interativo, ou seja, eu quero interagir com ele efetivamente. Caso não utilizasse esse parâmetro, o container subiria e cairia logo em seguida.

```
danielartine@artine ~$ docker run -it 41afdc6b059f
root@80f188e9f9bf:/# java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (build 1.8.0_212-8u212-b03-0ubuntu1.18.04.1-b03)
OpenJDK 64-Bit Server VM (build 25.212-b03, mixed mode)
root@80f188e9f9bf:/#
```

Lembra lá no início da explicação sobre o `RUN` quando falei que ele podia ser usado uma ou mais vezes? Então, isso faz uma baita diferença na hora da criação de imagens, pois, como foi dito, cada `RUN` criará uma etapa na criação da imagem. Mas, o que isso muda?

O grande diferencial da instrução `RUN` é que cada camada gerada por ele poderá ser reutilizada na criação de outras imagens.

Então, para testar, alterei meu Dockerfile para criar uma imagem nova com elementos em comum da outra imagem:

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
RUN touch arquivo-de-boas-vindas
```

Ele conseguirá reutilizar diversas camadas e tornará o processo muito mais rápido.

Caso eu queira adicionar um comando novo qualquer, ao executar o comando `docker build` para gerar essa nova imagem, vejam o resultado:

```
danielartine@artine ~$ docker build -t
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM ubuntu:18.04
--> 7698f282e524
Step 2/4 : RUN apt-get update
--> Using cache
--> 694ab280c9b2
Step 3/4 : RUN apt-get install openjdk-8-jdk -y
--> Using cache
--> 41afdc6b059f
Step 4/4 : RUN touch arquivo-de-boas-vindas
--> Running in b88809757f57
Removing intermediate container b88809757f57
--> 2790edeb6496
Successfully built 2790edeb6496
```

Vejam só que elegante! O Docker conseguiu reutilizar as camadas de outra imagem através do uso de cache para construir uma nova, tornando o processo muito mais rápido, sem necessidade de fazer downloads repetidos!

O `RUN` aceita parâmetros de dois jeitos:

```
RUN apt-get install openjdk-8-jdk -y OU RUN ["apt-get", "install", "openjdk-8-jdk", "-y"]
```

Ou seja, podemos passar os comandos separadamente entre aspas dentro dos colchetes. O resultado é o mesmo

CMD e ENTRYPOINT

Não há muito mistério na instrução **CMD**, ela na verdade é bem parecida com a instrução

RUN, com algumas poucas diferenças. Vamos ver quais são!

A sintaxe é a mesma, podemos passar os parâmetros do mesmo modo que a instrução RUN, alterando o último RUN por um CMD:

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
CMD touch arquivo-de-boas-vindas
```

Se construirmos uma imagem com o Dockerfile acima, veremos que ele não executou esse comando na etapa de criação.

Isso aconteceu porque na verdade a instrução CMD executa o comando apenas quando criamos o container e não passamos nenhum parâmetro para ele, ou seja, quando executarmos o comando `docker run -it` nessa imagem.

Caso passássemos algo como `docker run -it <id da imagem> /bin/bash`, ele sobrescreveria o comando `CMD touch arquivo-de-boas-vindas` e executaria apenas o `/bin/bash`.

Um outro teste também poderia ser:

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
CMD touch arquivo-de-boas-vindas
CMD touch outro-arquivo
```

Ao inicializar o container vi que apenas o outro-arquivo foi criado, mas por quê? O que acontece é que podemos ter quantos CMD quisermos, mas no fim das contas apenas será executado o último CMD, sem nenhum erro aparente!

E o **ENTRYPOINT**? Essa instrução faz exatamente a mesma coisa, porém seus parâmetros não são sobrescritos igual ao CMD.

ADD e COPY

Os nomes dessas instruções são bem intuitivos.

O papel do ADD é fazer a cópia de um arquivo, diretório ou até mesmo fazer o download de uma URL de nossa máquina host e colocar dentro da imagem

Eu utilizei o ADD para copiar o arquivo chamado `arquivo-host` da minha máquina para dentro da imagem, com o nome `arquivo-host-transferido`:

```
FROM ubuntu:18.04
RUN apt-get update
RUN ["apt-get", "install", "openjdk-8-jdk", "-y"]
ADD arquivo-host arquivo-host-transferido
```

Se criarmos uma imagem e rodarmos um container, teremos o seguinte resultado:



```
root@e411ee5f2:/# ls
arquivo-host-transferido bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys usr var
```

O arquivo está dentro do container

A instrução `ADD` também tem alguns efeitos interessantes, como: caso o arquivo que esteja sendo passado seja um arquivo de extensão `tar`, ele fará a descompressão automaticamente, além do fato já mencionado de poder fazer download de arquivos por URLs.

A instrução `COPY`, permite apenas a passagem de arquivos ou diretórios, diferente do `ADD`, que permite downloads. Como podemos ver abaixo:

```
FROM ubuntu:18.04
RUN apt-get update
RUN ["apt-get", "install", "openjdk-8-jdk", "-y"]
COPY arquivo-host arquivo-host-transferido
```

A próxima instrução mostrará como podemos documentar de maneira sucinta o nosso Dockerfile para quem for utilizá-lo.

EXPOSE

Há uma certa dúvida quanto ao uso dessa instrução. Muitas pessoas pensam que o **EXPOSE** serve para definir em qual porta nossa aplicação rodará dentro do container, mas na verdade o propósito é servir apenas para documentação.

Essa instrução não publica a porta efetivamente, já que o propósito dela é fazer uma comunicação entre quem escreveu o Dockerfile e quem rodará o container.

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
EXPOSE 8080
```

Logo, o Dockerfile acima não faz a publicação da porta, apenas serve como documentação.

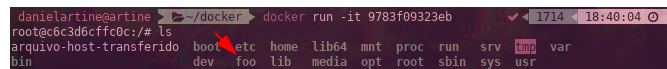
Agora vamos entender como podemos compartilhar informações entre o nosso container e nossa máquina host.

VOLUME

Essa instrução cria uma pasta em nosso container que será compartilhada entre o container e o host, funcionando do seguinte modo:

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install openjdk-8-jdk -y
VOLUME /foo
```

Quando criarmos um container dessa imagem, ele criará uma pasta chamada foo:



```
danielartine@artine ~ -/docker$ docker run -it 9783f09323eb
root@c6c3d6cffc0c:/# ls
arquivo-host-transferido boot etc home lib64 mnt proc run srv tmp var
bin dev foo lib media opt root sbin sys usr
```

Todo arquivo criado dentro dessa pasta será acessível a partir da máquina host no caminho `/var/lib/docker/volumes !`

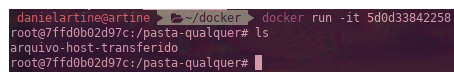
Por fim, como pode ser feita a organização de trabalho do container? Será que devemos trabalhar em qualquer pasta indefinidamente?

WORKDIR

Essa instrução tem o propósito de definir o nosso ambiente de trabalho. Com ela, definimos onde as instruções **CMD**, **RUN**, **ENTRYPOINT**, **ADD** e **COPY** executarão suas tarefas, além de definir o diretório padrão que será aberto ao executarmos o container.

```
FROM ubuntu:18.04
RUN apt-get update
RUN ["apt-get", "install", "openjdk-8-jdk", "-y"]
WORKDIR /pasta-qualquer
COPY arquivo-host arquivo-host-transferido
```

Ao acessarmos o container gerado por essa imagem, teremos o seguinte resultado:



```
danielartine@artine ~ -/docker$ docker run -it 5d0d33842258
root@7ffd0b02d97c:/pasta-qualquer# ls
arquivo-host-transferido
root@7ffd0b02d97c:/pasta-qualquer#
```

Ou seja, além de inicializarmos o container nessa pasta, o COPY colocou o arquivo dentro da pasta definida!

Conclusão

O processo de criação de imagens é de suma importância para quem quer aprender Docker. Vimos que através do Dockerfile podemos utilizar diversas instruções para atingir um determinado fim, seja a instrução FROM para utilizar uma imagem como base para a nossa, a instrução RUN para definir as camadas da nossa imagem, CMD e ENTRYPOINT para configurar o container em sua inicialização, ADD e COPY para movermos arquivos de nossa máquina para o futuro container, EXPOSE para documentar as questões de portas do container, VOLUME para conseguir transitar arquivos entre a máquina host e virtual, e, por fim, WORKDIR para definirmos o nosso ambiente de trabalho.

Aqui na **Alura**, temos um [curso sobre Docker](#), nele você aprenderá tudo sobre o que é um container, como fazer eles se comunicarem, além de aprender como criar suas próprias imagens para personalizar seus container.



Daniel Artine

Daniel é instrutor na Alura e analista de desenvolvimento sênior na Stone Age. Possui certificação Docker e formação em Ciência da Computação pela Universidade Federal do Rio de Janeiro.

Leia também:

- [Criando volumes com Docker](#)
- [Replicando ambientes com Docker](#)
- [Começando com Docker](#)
- [Criando um repositório local de imagens Docker](#)
- [Docker Compose para compor uma aplicação](#)

Vea outros artigos sobre [DevOps](#)

Quer mergulhar em tecnologia e aprendizagem?

Receba a newsletter que o nosso CEO escreve pessoalmente, com insights do mercado de trabalho, ciência e desenvolvimento de software

Escreva seu email

ME INSCREVA

AOVS Sistemas de Informática S.A
CNPJ 05.555.382/0001-33

NOSSAS REDES E APPS



NAVEGAÇÃO

PLANOS
TODOS OS CURSOS
ALURA CASES
GUIA DE CARREIRA
INSTRUTORES
TRABALHE CONOSCO
COMO VIRAR INSTRUTOR
ARTIGOS
PARA ESCOLAS
PODCASTS
POLÍTICA DE PRIVACIDADE
TERMOS DE USO
SOBRE NÓS
COMO FUNCIONA
DEV EM <T>

NOVIDADES E LANÇAMENTOS

Seu e-mail

RECEBER

ALURA NA SUA EMPRESA

Desenvolva seu time com apoio da nossa solução

CONHEÇA MAIS

COMUNIDADE

SCUBA DEV
IMERSÕES
DEPOIMENTOS
NOSSOS INSTRUTORES

FALE COM A GENTE

WHATSAPP
EMAIL E TELEFONE

BLOG

PROGRAMAÇÃO
FRONT-END
DATA SCIENCE
DEVOPS
UX & DESIGN
MOBILE
INOVAÇÃO & GESTÃO



Nós, da Alura, somos uma das Scale-Ups selecionadas pela Endeavor, programa de aceleração das empresas que mais crescem no país.



Fomos uma das 7 startups selecionadas pelo Google For Startups a participar do programa Growth Academy em 2021.

CURSOS

Cursos de Programação	Lógica Python PHP Java .NET Node JS C Computação Jogos IoT
Cursos de Front-end	HTML, CSS React Angular JavaScript jQuery
Cursos de Data Science	Ciência de dados BI SQL e Banco de Dados Excel Machine Learning NoSQL Estatística
Cursos de DevOps	AWS Azure Docker Segurança IaC Linux
Cursos de UX & Design	Photoshop e Illustrator Usabilidade e UX Vídeo e Motion 3D
Cursos de Mobile	React Native Flutter iOS e Swift Android, Kotlin Jogos
Cursos de Inovação & Gestão	Métodos Ágeis Softskills Liderança e Gestão Startups Vendas

Grupo Alura

EDUCAÇÃO



Caelum



Casa do Código

EDUCAÇÃO ONLINE



Alura



Alura Para Empresas



Alura LATAM



Alura Start



MusicDot



Alura Língua



PM3 - Cursos de Produto

COMUNIDADE



Hipsters ponto Tech



Scuba Dev



Layers ponto Tech



Like a Boss



Carreira sem Fronteiras



Hipsters ponto Jobs



GUJ