Docker: précisions sur le Dockerfile

Table of Contents

Dans cet article, on passera à la loupe le Dockerfile, le fichier de description d'une image et d'un conteneur Docker . Même si le nombre de commandes est réduit, leur subtilité n'apparaît pas à la première utilisation. Le site officiel demeure la meilleure source de documentation. On ne s'attardera que sur les commandes de base – `ADD`, `COPY`, `CMD` et `ENTRYPOINT`.

Les layers :

Dans le Dockerfile, chaque commande (`ADD`, `COPY`, `CMD`, `ENTRYPOINT`, ...) fait l'objet d'ajout d'une nouvelle couche (COW) à l'image. Et les couches, moins on en a, mieux c'est pour la taille des images. Sachant qu'en plus, le nombre de couches a une limite (42 auparavant et 127 depuis peu). Même si la contrainte du nombre de lignes (ou couches) est quasiment écartée, regrouper les commandes facilite la compréhension du Dockerfile.

Tip : Regrouper les commandes autant que possible

```
* Example 1. Dockerfile (avec 4 couches)

RUN curl -fsSL
http://archive.apache.org/dist/maven/maven-3/$MAVEN_VERSION/binaries/apache-
maven-$MAVEN_VERSION-bin.tar.gz
RUN tar xzf apache-maven-$MAVEN_VERSION-bin.tar.gz  - -C /usr/share
RUN mv /usr/share/apache-maven-$MAVEN_VERSION /usr/share/maven
```

```
RUN ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
```

```
* Example 2. Dockerfile “refactore” en une seule commande

RUN curl -fsSL
http://archive.apache.org/dist/maven/maven-3/$MAVEN_VERSION/binaries/apache-
maven-$MAVEN_VERSION-bin.tar.gz | tar xzf - -C /usr/share \
  && mv /usr/share/apache-maven-$MAVEN_VERSION /usr/share/maven \
  && ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
```

On réduit le nombre de couches a une et on a gagne en lisibilité.

Le cache

Toutes les couches d’une image sont mises dans le cache. Quand on lance le build d’une image Docker, il faut d’abord commencer par chercher dans le cache des couches qu’il pourrait réutiliser pour optimiser le temps de construction de l’image en évitant les téléchargements inutiles. De ce fait, en cas de modification, seules les lignes concernées par la modification seront réexecutées.

```
FROM java:openjdk-8-jdk
MAINTAINER Yakhya DABO
ENV MAVEN_VERSION 3.3.3
ENV M2_HOME /usr/share/maven
ENV PROJECT_DIR /usr/src/app
...

...
8. RUN mkdir -p $PROJECT_DIR
9. COPY config/settings.xml $M2_HOME/conf/
10. RUN curl -fsSL
http://archive.apache.org/dist/maven/maven-3/$MAVEN_VERSION/binaries/apache-
maven-$MAVEN_VERSION-bin.tar.gz | tar xzf - -C /usr/share \
  && mv /usr/share/apache-maven-$MAVEN_VERSION /usr/share/maven \
  && ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
11. VOLUME $PROJECT_DIR
12. WORKDIR $PROJECT_DIR
```

Si on apporte une modification sur la ligne 9, le cache invalide toutes ses filles, les lignes 10, 11, … qui seront réexecutées. Les lignes 0, 1, … 8 n’étant pas des filles de 9, elles ne seront pas concernées par les modifications, c’est le cache qui sera utilisé.

Deux astuces pour bien profiter du cache :

- Placer les lignes qui changent souvent le plus bas possible (ajout d’un jar par exemple, COPY).
- Mettre les opérations coûteuses le plus haut possible dans le Dockerfile afin d’éviter de les réexécuter à chaque modification (téléchargement par exemple, RUN curl).

Les Directives

La directive FROM spécifie une image à partir de laquelle Toute référence d'image valide peut être utilisée.

```
FROM node:5
```

La directive WORKDIR définit le répertoire de travail actuel dans le conteneur, ce qui équivaut à exécuter cd dans le conteneur. RUN cd ne changera pas le répertoire de travail en cours.

```
WORKDIR /usr/src/app
```

La directive RUN exécute la commande donnée à l'intérieur du conteneur.

```
RUN npm install cowsay knock-knock-jokes
```

La directive COPY copie le fichier ou le répertoire spécifié dans le premier argument à partir du contexte de génération (le path transmis au docker build path) vers l'emplacement du conteneur spécifié par le deuxième argument.

```
COPY cowsay-knockknock.js .
```

La directive CMD spécifie une commande à exécuter lorsque l'image est exécutée et qu'aucune commande n'est donnée. Il peut être remplacé en transmettant une commande à docker run .

```
CMD node cowsay-knockknock.js
```

Il existe de nombreuses autres instructions et options.

Organisation d'un dockerfile

Exemple simple

```
# Base image
FROM python:2.7-alpine

# Metadata
MAINTAINER John Doe <johndoe@example.com>

# System-level dependencies
RUN apk add --update \
```

```
    ca-certificates \
    && update-ca-certificates \
    && rm -rf /var/cache/apk/*

# App dependencies
COPY requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

# App codebase
WORKDIR /app
COPY . ./

# Configs
ENV DEBUG true
EXPOSE 5000
CMD ["python", "app.py"]
```

Classement du contenu

Le `dokerfile` présenté ci-dessus est organisé de la maière suivante

- Déclaration d'image de base (`FROM`)
- Métadonnées (par exemple `MAINTAINER` , `LABEL`)
- Installation des dépendances du système (`RUN`: apt-get install , apk add)
- Copie du fichier de dépendances de l'application (`COPY`: bower.json , package.json , build.gradle , requirements.txt)
- Installation de dépendances d'application (`RUN`: npm install pip install)
- Copier l'intégralité du code
- Configuration des configurations d'exécution par défaut (`CMD`, `ENTRYPOINT` ,`ENV` ,`EXPOSE`)

Ces commandes sont destinées à optimiser le temps de création à l'aide du mécanisme de mise en cache intégré de Docker.

Règle de base: Les parties qui changent souvent (par exemple, la base de code) doivent être placées au bas du fichier Dockerfile et vice-versa. Les parties qui changent rarement (par exemple les dépendances) doivent être placées en haut.

Directives ADD et COPY

Ces deux commandes sont facilement sujet à confusion, de par leur nom mais aussi de par leur syntaxe : elles semblent faire la même chose, sauf que dans la pratique, ce n’est pas tout le temps le cas.

```
ADD  src dest
COPY   src dest
```

- **src**: un répertoire (ou fichier) du host relatif au `contextDir`.
- **dest**: un nom de fichier (/var/opt/fileName), ou un répertoire (/var/opt/) du conteneur

```
Une petite précision : dans la commande “docker build”, on spécifie le
context du Dockerfile.

 $ docker build -f dockerfileDir contextDir

ou

 $ docker build contextDir  // (si contextDir = buildDir)
```

ADD

ADD peut aussi faire la même chose, mais le src peut être une URL. Dans ce cas, docker se charge du téléchargement et de placer le fichier téléchargé dans dest. Si src est un fichier zippé et dest un répertoire docker dezzipe src (ADD file.zip /var/opt/).

COPY

COPY se contente tout simplement de prendre un fichier (ou répertoire) du host et de le mettre dans le conteneur.

On peut se contenter de COPY pour les opérations simples de copie du host vers le conteneur, et de coupler RUN avec les utilitaires existants tels que tar, unzip, wget, curl, … si on a besoin de zipper ou de télécharger des fichiers.

Directives CMD et ENTRYPOINT

Il existe deux directives Dockerfile pour spécifier quelle commande exécuter par défaut dans les images construites. Si vous spécifiez uniquement `CMD` alors docker exécutera cette commande en utilisant le `ENTRYPOINT` par défaut, à savoir `/bin/sh -c`. On peut remplacer soit le point d’entrée, soit la commande lorsqu'on démarre l’image construite.

Si on spécifie les deux, alors `ENTRYPOINT` spécifie l'exécutable du processus de conteneur et `CMD` sera fourni comme paramètre de cet exécutable.

CMD

Par exemple, si `Dockerfile` contient

```
FROM ubuntu:16.04
CMD ["/bin/date"]
```

On utilise la directive ENTRYPOINT par défaut de `/bin/sh -c` avec `/bin/date` (par défaut). La commande du processus de conteneur sera `/bin/sh -c /bin/date` . Lorsqu'on exécutera cette image, elle affichera par défaut la date actuelle

```
$ docker build -t test .
$ docker run test
Tue Jul 19 10:37:43 UTC 2016
```

On peut spécifier la commande à passer sur la ligne de commande, auquel cas il exécutera la commande spécifiée.

```
$ docker run test /bin/hostname
bf0274ec8820
```

ENTRYPOINT

Lorsqu'on spécifie une directive ENTRYPOINT , Docker utilise cet exécutable et la directive CMD spécifie le ou les paramètres par défaut de la commande. Donc, si votre Dockerfile contient:

```
FROM ubuntu:16.04
ENTRYPOINT ["/bin/echo"]
CMD ["Hello"]
```

Alors lord fe l'exécution ça va produire

```
$ docker build -t test .
$ docker run test
Hello
```

On peut fournir différents paramètres, mais ils seront tous exécutés dans `/bin/echo`

```
$ docker run test Hi
Hi
```

Pour remplacer le point d'entrée répertorié dans le fichier Docker (par exemple, si on souhaite exécuter une commande différente de echo dans ce conteneur), il faut spécifier le paramètre `--entrypoint` sur la ligne de commande:

```
$ docker run --entrypoint=/bin/hostname test
b2c70e74df18
```

Généralement, on utilise la directive ENTRYPOINT pour pointer vers uns script que l'on vous souhaite exécuter et CMD vers les paramètres par défaut.

Directive EXPOSE

Expose un port dans le fichier `Dockerfile`

Description

```
EXPOSE <port> [<port>...]
```

EXPOSE informe Docker que le conteneur écoute les ports réseau spécifiés au moment de l'exécution. EXPOSE ne rend pas les ports du conteneur accessibles à l'hôte. Pour ce faire, vous devez utiliser l'option `-p` pour publier une plage de ports ou l'option `-P` pour publier tous les ports exposés. On peut exposer un numéro de port et le publier en externe sous un autre numéro.

Exemple:

À l'intérieur du fichier Docker:

```
EXPOSE 8765
```

Pour accéder à ce port depuis la machine hôte, inclure cet argument dans le commande `docker run` :

```
-p 8765:8765
```