@pottava 2018年03月09日に更新

# 効率的に安全な Dockerfile を作るには

Docker dockerfile



⚠ この記事は最終更新日から1年以上が経過しています。

## 結論

以下の手順で作るのが効率的です。

1. ベースにする Docker イメージを決める
2. `docker run -it <docker-image> sh` でコンテナ内部で作業
3. 1行ずつ、うまくいったらどこかにメモ
4. 失敗したらいったん `exit` して再度 `docker run`
5. ファイルの取り込みやポートの外部公開が必要ならオプション付きで `docker run`
6. 全部うまくいったら Dockerfile にする

ネットで見たことはないですが、もし **docker build で試行錯誤しながら Dockerfile を作るとしたら、それはさすがに苦行です。**
遅い

デバッグしにくい！コンテナ爆発しろ！！って気持ちになります。
これが原因で「Docker 使えない

便利じゃない



」と思っていたのならそれは勘違いです。

# 手順詳説

試しに ip-api.com にリバースプロキシするだけの Nginx イメージを作ってみます。

## 1. ベースにする Docker イメージを決める

個人的には Alpine Linux が圧倒的にオススメですが、
今回は CentOS をベースにしてみます。

## 2. `docker run -it <docker-image> sh` でコンテナ内部で作業

`docker run -it centos:6 /bin/bash` で CentOS 6 のコンテナを起動し
せっせと Nginx をインストールしていきます。
`epel-release` を入れて、 `nginx` を入れて・・

このとき
各ツールの `y` オプションや `yes` を使い、対話なしで実行できるようにしましょう。
また、ファイルの上書きなどで不慮のエラーが起きないよう
`cp -f` や `ln -f` のように強制オプションをつけたり、
不要なログを抑えるために `curl -s` や `wget -q` の利用を検討します。

## 3. 1行ずつ、うまくいったらどこかにメモ

成功したコマンドはどこかに転記しておきましょう。
隣に開いたエディタなどにコピペしていきます。

`yum search` などセットアップ上本質的でない作業は当然省きますが、
ベテランさんだと無意識にやってしまうかもしれない
設定値変更に伴うバックアップファイルの作成なども省きましょう。
(例: `cp /etc/foo/default.conf /etc/foo/default.conf.org` )

## 4. 失敗したらいったん `exit` して再度 `docker run`

設定値がおかしくなってしまった、クリーンな環境でやり直したい
そんなときは `exit` して 2 からやり直します。

とはいえ毎回最初からやるのは非効率。
うまくいくところまででいったん `exit & docker commit` で
イメージを作っておけば、次回はそこから作業を再開できて便利です。
Docker ほんと便利。

## 5. オプション付きで `docker run` が必要なケース

今回のように、**外からの接続を前提にしている** イメージの場合、
予め `docker run -it -p 80:80 centos:6 /bin/bash`
のようにポートをホスト側に公開しておくと確認作業が捗ります。

また、
**外部ファイルを取り込む**（Dockerfile でいう ADD / COPY）必要がありそうなら
`docker run -it -v $(pwd):/tmp/share centos:6 /bin/bash`
のようにしてホスト側の作業フォルダをコンテナ内部に
マウントしておくと検証が進めやすいです。

## 6. 全部うまくいったら Dockerfile にする

まとめるとこんな Dockerfile になりました。

```
FROM centos:6

RUN set -x && \
    yum install -y epel-release && \
    yum install -y nginx && \
    sed -i -e "s/index  index.html index.htm/proxy_pass http:\/\/ip-api.com\/json/"
        /etc/nginx/conf.d/default.conf && \
    ln -sf /dev/stdout /var/log/nginx/access.log && \
    ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

ちゃんと動くか確認してみましょう！

（以下を実行する場合はローカルに↑の Dockerfile を保存してください）

```
$ docker build -t test-image .
$ docker run --name test-container -d -p 80:80 test-image
$ curl -s <DockerホストIP> | jq -r '.query'
```

IPアドレスが返ってきましたか？[1]

コンテナを破棄しておきましょう。

```
$ docker stop test-container
$ docker rm test-container
```

# Tips

## Dockerfile ベストプラクティス

- Dockerfile のベストプラクティス（公式ドキュメント翻訳）
  ある程度 Docker に慣れてきてからでもいいので、Docker イメージの、
  特に**レイヤーとは何か**を理解しておくのが大切だと思います。
  `.dockerignore` の利用なども公式ならではの記述ですね。

- Dockerイメージの最適化 - ワザノバ | wazanova
  一般公開目的ではなく、実運用などで極限までイメージを小さくしたいなら
  `Flatten your image` の手法を使うのは有効だと思います。
  **「レイヤーはイメージ全体のサイズを増やす方にしか働かない」**法則を
  ある意味避けることができ、副次的に こんな裏技的用途 にも使えます。

## `docker commit` と `docker build`

そもそも Docker イメージを作る方法はいくつかあるわけですが
`commit` でイメージを作る過程を **可視化するためにあるのが** `build` 。

本記事の手順はこの概念にのっとっています。
つまり本来は、コンテナ内部で作業した後に外に出て
`commit` して Docker イメージを作るプロセスが先にある。
Dockerfile は単にそれをまとめたものであり、 `build` するための手順書。

Dockerfile でビルドした Docker イメージであれば、
作成されるまでの手順を公開しやすく（Docker Hub の自動構築 など）
ユーザには比較的[2] 安心して使ってもらえるわけです。

## `docker run` の rm オプション

`docker run --rm ..` のように `--rm` をつけておくと、シェルから抜けたときの
Docker コンテナの残骸を後で `docker rm` で掃除する必要がなくなり便利です。

ただし Dockerfile を作る場合、 `commit` を駆使して作業途中の状態を作ったり
そのイメージをベースに `run` することで作業効率を改善できるので

散らかること前提で `rm` オプションは使わないほうがいいかもしれません。

## set -x

最終的に Dockerfile にコマンドをまとめるとなると、レイヤー削減のために
RUN には **&&オペレータで一連の操作を一つにまとめた** 黒魔術のようなコマンドが
記述されると思いますが、途中で処理が失敗したときにどこまで進んだかわかるよう、
実行したコマンドそのものを標準エラー出力にだすよう `set -x` しておくと便利です。

## チェックサムや署名の検証をする

タイトルにある後半の「安全性」に関して。

バイナリダウンロード元にチェックサムや署名についての情報があるなら
Dockerfile の中で検証ロジックを入れておくのがよさそうです。
もし一般に公開する Docker イメージであれば、安心感増します！

- Node.js でのチェックサム検証例
- Node.js での署名検証例

1. `jq` をインストールしていない方は `curl -s <DockerホストIP>` だけでお試しください。 ↩
2. `ADD / COPY` や `RUN wget/curl` で外部ファイルを持ち込んでいると Dockerfile だけでは その由来を説明することはできなくなるわけですが。。 ↩

pottava @pottava

フォロー