DockerFileハンズオン

DockerFfileとは

今までは開発などに利用するDockerコンテナを作成するにあたり、各種設定などを一々コマンドラインに入力して行っていた。
しかし、一連の手順をコマンドラインのみで行うのはコマンドの入力ミスなどが起こりやすく、そもそも効率が悪い。
そこでDockerでは`Dockerfile`と呼ばれるファイルに各種命令を記述することで、自動的に望みのコンテナを作成することができる仕組みが存在する。
本章では具体的にapacheが稼働するコンテナのためのDockerfileを記述することを通じてDockerfileについて学ぶ。

Apacheコンテナの作成

まずはDockerfileを作成する。
基本的にDockerfileはどこに作成しても良いが、後述する**ビルドコンテキスト**との兼ね合いがあるため、自身の階層以下にディレクトリが存在しないような場所に作成する。
今回作成するDockerfileは以下のようになる。

```
FROM centos:latest (1)
RUN yum install -y httpd iproute && yum clean all (2)
RUN echo "Hello Apache." > /var/www/html/index.html (3)
ENTRYPOINT [ "/usr/sbin/httpd","-DFOREGROUND"] (4)
```

項番	概要
1	親イメージとしてcentosを指定
2	httpd(apache)をイメージ内で利用できるように指定
3	apcheのindexページに使用するファイルをコンテナ内の所定の場所に書き込む
4	コンテナ起動時にhttpdをforegroundで実行する命令をコンテナ内で行うように指定

これを`Dockerfile`というファイル名で適当なディレクトリに保存し、そのディレクトリ階層で以下のコマンドを実行することでイメージをビルドする。

```
docker build -t centos:apache .
```

コマンドラインに各種情報が流れるが、`docker images`でcentos:apacheが存在すればビルドは成功している。

以下のコマンドを実行し、コンテナを起動する。

```
docker run -itd -p 8081:80 centos:apache
```

ブラウザなどで、`http://127.0.0.1:8081/index.html`にアクセスし、**Hello Apache**が表示されれば成功である。

イメージのビルド

Dockerfileの構造の前にまずはDockerにおけるビルドに関して解説する。
ビルドする際はdocker buildコマンドに**コンテキスト(パス or URL)とDockerfile**の情報を付与して行う。

コンテキストはDockerのビルドに必要なファイルが存在するパスやURLを示し、通常コンテキストで指定されたパスorURLをルートディレクトリとして、その配下のディレクトリ、ファイルを**全て**Dockerデーモンに送信してビルドを行う。
今回のビルドは`docker build -t centos:apache .`というコマンドを実行しているが、この場合、コンテキストは`.`つまり現在のディレクトリを指定している。
また、ディレクトリだけでなくURLなども指定することができるため、githubのリポジトリを指定することができる。
この時、内部的にはリポジトリのクローンを行い、ファイルなどを取得している。
また、Dockerfileの情報を与えていないが、オプションなどで指定しない場合、コンテキストのルートディレクトリに保存されている`Dockerfile`という名称のファイルを探して自動的に使用する。
URLを指定してリポジトリのクローンを取得した場合は、そのリポジトリのルートにあるDockerfileを使用する。
パスを指定した時、Dockerfileを別の名称やディレクトリなどに保存して利用したい場合は`-f`オプションを利用することができる。

注意点として、前述したようにコンテキストは基本的に**指定した場所以下のディレクトリファイルを全てDockerデーモンに送信する**ため、コンテキスト配下に不要なファイルなどが多量に含まれている場合、イメージのビルドに非常に時間がかかる可能性がある。
また、ビルド時にDockerはなるべくキャッシュを利用するため、Dockerfileの書き方によっては意図したような挙動を示さないことがある。
キャッシュを使用しないビルドを強制する場合は、`--no-cache`を使用する。

.dockerignore

コンテキスト配下にデーモンに対して送信してほしくないファイルが存在する場合、コンテキストルートに`.dockerignore`ファイルを作成し、送信したくないファイルを記述することで実現できる。
ファイルの中身は`.gitignore`と同じように除外したいファイルやディレクトリを記述する。 例を以下に示す。

```
root
 ├ Dockerfile
 └ testfile
   └ test.md
```

以上のようなディレクトリを作成する。
test.mdの中身は特に何でもよい。

Dockerfileの中身を以下に示す。

```
FROM centos:latest
COPY ./testfile /testdir/
ENTRYPOINT [ "/bin/bash" ]
```

COPYコマンドでtestfileディレクトリ配下のファイル(今回はtest.md)をコンテナのtestdirにコピーすることを指示している。
これをこのままビルドし、コンテナを起動してみる。
rootディレクトリに移動し、以下のコマンドを実行する。

`docker build -t <<任意のイメージ名>>:<<任意のタグ名>> .`

ビルドが成功し、コンテナ内を確認してみると、ルートディレクトリ直下に`testfile`ディレクトリが存在し、その中に`test.md`が存在することがわかる。

次に、Dockerfileと同じ階層に`.dockerignore`を以下のように記述して配置する。

```
/testfile/
```

配置後に先ほどのビルドコマンドをタグ名を変更して実行すると、`.dockerignore`によって`testfile`ディレクトリが無視されるため、`COPY failed: stat /var/lib/docker/tmp/docker-builder607533665/testfile: no such file or directory`をエラーが出てビルドが失敗することがわかる。

Dockerfileの構造

Dockerfileは前述のようにイメージを作成するための命令を記述したテキストファイルになるが、記述方法には当然ルールがある。
Dockerfileでは、基本的に**コマンド**とそのコマンドに対するパラメータのセットで一行ずつ記述され、イメージの元になるベースイメージを指定する`FROM`コマンドが含まれている。
コマンドは一つのコマンドが一つのレイヤを構成し、そのコマンドは上から順に**個別に**実行される。
そのため、例えば`CMD cd /foo`などでディレクトリを移動したとしてもそれ以降のCMD命令などには影響されない。
通常作業用ディレクトリを変更する場合は**WORKDIR**コマンドを利用する。
また、Dockerfileに記述されたコマンドはその一つ一つがイメージのレイヤーになる。
そのためコマンドを多数記述した場合はイメージサイズが非常に大きくなる可能性があるため、後述のマルチステージビルドなどの機能を利用して、なるべくサイズを減らすことが望ましい。

コマンド

Dockerfileで記述可能なコマンドは20近く存在し、それぞれ説明すると非常に長くなるため概要を記述する。
より詳細が知りたい場合はDockerfileのリファレンスを参照すること。
また、コマンドは大文字、小文字を区別しないが慣習として大文字を使用する。

コマンド	概要

コマンド	概要
FROM	ベースイメージを指定する タグ、ダイジェストを設定できる
RUN	ビルド時にコマンドを実行する
CMD	コンテナ起動時に行うコマンドを設定する
LABEL	イメージにメタデータを付与する
EXPOSE	コンテナが公開するポートを指定する ホスト側とのバインドは行われない
ENV	コンテナの環境変数を設定する
ADD	コンテキスト上のファイル、ディレクトリをコンテナ内に追加する
COPY	コンテキスト上のファイル、ディレクトリをコンテナ内に追加する
ENTRYPOINT	コンテナ起動時に行うコマンドを設定する
VOLUME	イメージでDockerのvolumeをマウントする場所を指定する
USER	この記述以降のRUNやCMD、ENTRYPOINTを実行する際のユーザーを指定する
WORKDIR	この記述以降のRUNやCMD、ENTRYPOINTを実行する際のディレクトリ
ARG	ビルド時にCLI上から渡せる引数を定義する
ONBUILD	記述したDockerfileをベースイメージとしたDockerfileのビルド時に実行するコマンドを設定する
STOPSIGNAL	コンテナ終了時に送信するシステムコールシグナルを設定する
HEALTHCHECK	コンテナのヘルスチェック方法を指定する
SHELL	この記述以降のRUNコマンドなどを実行するシェルを変更する

また、上記のコマンド以外にも**MAINTAINER**というコマンドが存在するが、廃止予定なため省く。

注意

Dockerfileを利用するにあたり注意すべき事柄を以下に示す。

環境変数

ENVやARGなどで指定した情報は、イメージに組み込まれてしまう。
イメージに組み込まれた場合、基本的にそのイメージを取得した場合はdocker historyなどで誰もが見れてしまう状態になる。
そのため各種パスワードなどをこの方法でやり取りする場合はセキュリティインシデントの危険性がきわめて高くなる。
このようなことを避けるための手法を以下に示す。

- Docker Composeのsecrets、KubernatesのSecretを利用する
- コンテナ起動時に環境変数として渡す

- BuildKitの`--mount=type=secret`や`--mount=type=ssh`を利用する
- マルチステージビルドを使う
- ネットワーク越しに取得する

CMDとENTRYPOINT

CMDとENTRYPOINTは同じようにコンテナ起動時に実行するコマンドを記述するようなコマンドのように見えるが、振る舞いが異なる。
簡潔に説明すると、**CMDはコンテナ起動時にコマンドを指定しない場合のデフォルトコマンド**、**ENTRYPOINTは必ず実行するコマンド**であり、**両方存在する場合はCMDはENTRYPOINTに対する引数とみなされる。**

また、CMDは前述のように簡単に上書き可能であるが、ENTRYPOINTも起動時に`--entrypoint`オプションを使用すれば上書き可能である。

RUNコマンド

RUNコマンドでビルド時に様々なコマンドを実行する場合、それぞれ別のRUNコマンドで実行する場合、その一つ一つがイメージレイヤとなってしまい、イメージの容量が増大する。
それを避けるため、RUNコマンド内に記述する命令をそれぞれ`&&`で結合し、一つに纏めてしまうことが推奨される。

マルチステージビルド

今までのDockerfileでは、通常一つのDockerfileから生成されるイメージは常に1つである。
この場合、Dockerで開発している場合はプログラムのビルド環境のコンテナと実行環境やテスト環境のコンテナを用意する場合はそれぞれのDockerfileを作成、保守する必要があり、効率的ではない。
マルチステージビルド機能を利用する場合、同一のDockerfile内に複数のステージのイメージの記述が可能になり、イメージサイズの低減、細やかなコンテナイメージの作成が可能になる。

利用方法

今回はgoのビルドを行う環境とgoの実行環境を分離する形のマルチステージビルドを行うこととする。
なお、記述するGoのソースコードやDockerfileの内容はdocker-composeハンズオンとほぼ同じである。
今回のディレクトリ構成は以下のようになる。

```
go
  ├ Dockerfile
  └ src
    └ main.go
```

Dockerfileでは、ビルドステージを定義し、main.goファイルをビルドするイメージ、runステージでそのビルド済みバイナリを実行するイメージをそれぞれ記述する必要がある。
なお、利用するmain.goは以下のようになる。

Goのソースコードである`main.go`は以下のようになる。

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello World")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8000", nil)
}
```

Dockerfileは以下のようになる。

```
FROM golang:latest AS build  //(1)

COPY ./src/main.go /usr/go/src/
WORKDIR /usr/go/src
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build main.go

FROM alpine:latest as run //(2)

COPY --from=build /usr/go/src/main /usr/go/main  //(3)
WORKDIR /usr/go
CMD ["./main"]
```

項番	概要
1	goのビルドを行うステージを定義する
2	実際にビルドされたバイナリファイルを実行するステージを定義する
3	ビルドステージでビルドしたバイナリを実行ステージにコピーする

実際にビルドする際は以下のコマンドを実行する。

```
docker build --target <stage> -t <イメージ名>:<タグ>.
```

これでgorunイメージからコンテナを作成し、起動し`http://localhost:8000/`にアクセスするとレスポンスが返却されることがわかり、マルチステージビルドが成功していることがわかる。
なお、コンテナ起動時には`-p`オプションを付与し、ポートのバインディングを行うこと。