Docker Compose

概要

前述したように、DockerコンテナはCLIから各種設定を行ったり、Dockerfileを作成することで、テキストベースで管理などが出来る。
しかしながら、複数のコンテナが関連したサービスを考えるとき、CLIベースでは当然管理に難があり、DockerfileだけでもDockerfile毎にコマンド実行などを行って管理しなければいけない為、関連するコンテナの数が増えれば増えるだけ管理が難しくなってくる。

それらの欠点を解消するため、Dockerでは`Docker Compose`を利用して複数のコンテナを管理することができる。
Docker Composeはyamlファイルベースで管理を行う特徴を持ち、また、プロジェクト名を付与することで単一ホスト上で複数の環境を分離して管理することができる。
実際にDocker Composeを利用しながら解説を行う。

Docker Composeの利用

導入

`Docker for Windows`を利用している場合、基本的にはDockerインストール時にDocker Composeもインストールされているはずである。
`docker-compose --version`コマンドを実行してバージョン情報が取得できていれば問題ない。

単一のコンテナを立ち上げる

Docker Composeの入門編として、まずは単一のコンテナをDocker Composeを用いて立ち上げる。
今回はGoでWebAPIにアクセスすると単純に`Hello World`と返却するアプリケーションを作成する。

全体的なディレクトリ構成は以下のようになる。

```
/
┣ docker-compose.yml
┗ go
  ┗ src
    ┗ main.go
```

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)
}
```

docker-compose.ymlは以下のようになる。

```
version: '3' #1
services: #2
  go: #3
    image: golang:latest #4
    container_name: 'hello-go' #5
    volumes: #6
        - ./go/src:/usr/go
    working_dir: /usr/go #7
    command: [go, run, main.go] #8
    ports: #9
        - 8000:8000
```

項番	概要
1	composeファイルの形式のバージョンを指定
2	作成するコンテナはこのservices以下に記述する
3	作成するサービス名を記述し、設定は以下に書き込む
4	利用するイメージを記述する
5	コンテナ名を明示的に示したい場合は記述する 記述がない場合はサービス名などから自動的に設定される
6	srcディレクトリ以下をコンテナにマウントする
7	ワーキングディレクトリをmain.goがある階層に指定する
8	コンテナ立ち上げ時にmain.goを起動するように指定する
9	コンテナのポートとホストのポートをバインディングする

上記のファイル類を用意したのち、docker-compose.ymlが存在するディレクトリで`docker-compose up`を実行するとコンテナが立ち上がり、`http://localhost:8000/`にアクセスすると、`Hello World`が表示される。

なお、コンテナなどが不要になった場合は`docker-compose down`コマンドでコンテナの削除や作成されたネットワークの削除を自動的に行ってくれる。

以上のように、単純に公開されているイメージに設定を付与してコンテナを立ち上げる場合はdocker-compose.ymlを記述するだけで事足りる。
しかし、ユーザー独自にimageをビルドして利用する場合は別途Dockerfileを記述する必要があることに注意すること。

docker-compose.ymlでマルチステージビルドを行う

docker-composeでもマルチステージビルドを利用することができる。
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。

```
/
┣ docker-compose.yml
┗ go
  ┣ Dockerfile
  ┗ src
    ┗ main.go
```

今回のケースでは、imageのビルドを行う必要があるため、マルチステージビルドに対応したDockerfileを記述する。
Dockerfileに関してそれぞれbuildステージでmain.goファイルをビルドし、runステージでそのビルド済みバイナリを実行する形になる。
なお、利用するmain.goは前回と同様なので省略する。

```
FROM golang:latest AS build

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

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

以下に今回使用するdocker-compose.ymlを示す。

```
version: '3'
services:
  go:
    container_name: 'hello-go'
    build: #1
      context: ./go #2
      target: run #3
```

```
    ports:
        - 8000:8000
```

項番	概要
1	ビルドを伴う場合はbuild以下に記述を行う
2	ビルドコンテキストを指定する
3	この部分に対象となるビルドステージを記述する

これも前回と同じように`docker-compose up`コマンドでコンテナが立ち上がり、非常に軽量なalpineイメージから作成されたコンテナでプログラムが実行され、Webページにアクセスできることがわかる。

複数種類のコンテナを立ち上げる

続いてまさにdocker-composeを利用する上で最もメリットのある複数コンテナを管理する場合の説明を行う。
今回はgo言語で記述したwebページとredisを組み合わせ、訪問回数を表示するようにする。

今回のディレクトリ構造は以下のようになる。

```
/
├ docker-compose.yml
├ go
|  ├ Dockerfile
|  └ src
|     └ main.go
└ redis/data  #1
```

項番	概要
1	今回はredisのデータを永続化するので永続化データの格納場所を用意しておく なお用意しなくても自動的に作成される

Dockerfileは以下のようになる。

```
FROM golang:latest AS build

COPY ./src/main.go /usr/go/src/
RUN go get github.com/gomodule/redigo/redis #1
WORKDIR /usr/go/src
CMD ["go", "run", "main.go"]
```

項番	概要
1	Goで利用するredis用のライブラリをコンテナ内に用意する

main.goは以下のようになる。

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gomodule/redigo/redis"
)

func handler(w http.ResponseWriter, r *http.Request) {

    fmt.Fprintf(w, "Hello World %d", countUp())

}

func countUp() int {

    c, err := redis.Dial("tcp", "redis:6379") //1
    if err != nil {
        panic(err)
    }

    count, err := redis.Int(c.Do("GET", "count"))
    plusCount := count
    plusCount++
    c.Do("SET", "count", plusCount)
    defer c.Close()
    return count
}

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

項番	概要
1	docker-composeを利用している場合、自動的に個別のネットワークが作成され、各サービス間でサービス名前解決を行えるため`redis:6379`でgolangコンテナとredisコンテナの接続ができる。

docker-compose.ymlは以下のようになる。

```
version: '3'
services:
  go:
    container_name: 'hello-go'
```

```
    build:
      context: ./go
    ports:
        - 8000:8000
  redis:
    image: redis:latest
    volumes:
        - ./redis/data:/data #1
    command: redis-server --appendonly yes #2
```

項番	概要
1	redisコンテナ内でredisのデータが保存される`/data`ディレクトリとホスト側の`/redis/data`をバインドしてデータの永続化をする
2	データ永続化の為commandを上書きする

main.goの説明でもあったように、docker-composeを利用した場合、明示しなければ自動的にコンテナネットワークが作成され、コンテナ間の通信で名前解決を利用することができる。 また、`/redis/data`でデータの永続化を行えるため、コンテナを停止したとしてもredisのデータは失われない。

これも前回と同様に`docker-compose up`を実行することで、各種サービスが起動する。
curlなどで`http://localhost:8000/`にアクセスすると`Hello world <カウント>`が表示され、アクセスするたびにカウントが上がっていくことがわかる。