Docker Hands-on

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎

目的

できるようになること

- コンテナを仮想マシンに代わる手軽な開発環境として使えるようになる
- 複数のコンテナが協調して動作する環境を立ち上げられるようになる

扱わないこと

- Docker の内部の仕組み
- Docker の本番環境での利用

※一部説明を省略・簡略しているところもある

スライドのコマンド

実際に入力してもらう部分は $ または # から始まる

```
$ ...       // ホストで実行するコマンド

# ...       // コンテナ内部で実行するコマンド
```

特に重要な部分は太字にしている

```
$ ...

# ...
```

Part 1. Docker の基礎

Agenda

- Part 1. Docker の基礎
  - **コンテナの起動・停止・破棄**
  - コンテナの様々な起動オプション
  - 実行中のコンテナに対する操作
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎

用語

- **コンテナ**
  - プロセスとその実行環境 （filesystem, network, etc.） をカプセル化したもの
- **メインプロセス**
  - コンテナの中核となる単一のプロセス
- **イメージ**
  - コンテナをアーカイブし、データとしてやりとりできるようにしたもの
- **ホスト**
  - コンテナを動かすためのマシン（各自の Mac 等）

イメージのダウンロード

イメージを取得する方法は２つ

1. Webからダウンロードする ← まずはこちらから
2. 自分で作る ← あとで解説

以下のコマンドでUbuntuのイメージをダウンロードできる

```
$ docker image pull ubuntu
```

このあと使う他のイメージもダウンロードしておく

```
$ docker image pull nginx
$ docker image pull python
```

コンテナの起動

次のコマンドでコンテナを起動できる

`docker container run <image> <command>`

先程 pull した ubuntu イメージからコンテナを起動してみる

`$ docker container run ubuntu echo hello`

command で起動したプロセス（この場合は `echo hello`）がメインプロセス

メインプロセスの生死 = コンテナの起動・停止

この場合 `echo hello` はすぐ終了するので、コンテナはすぐ停止する

起動中コンテナの一覧

今度は sleep コマンドをメインプロセスとしてコンテナを起動してみる

```
$ docker container run ubuntu sleep infinity
```

このプロセスは自動では終了しない

他のターミナルで下記を実行すると、コンテナが実行中であることがわかる

```
$ docker container list
```

```
CONTAINER ID    IMAGE     COMMAND             CREATED          STATUS          PORTS     NAMES
944883bf35c7    ubuntu    "sleep infinity"    3 seconds ago    Up 8 seconds              nervous_kowalevski
```

コンテナの停止

IDを指定して先程起動したコンテナを停止する

$ docker container stop <container ID>

今度は一覧に何も表示されなない

$ docker container list

コンテナは停止したが、まだ破棄されていないので --all をつけると見れる

$ docker container list --all

STATUS が Exited... となっているはず

コンテナの再起動

IDを指定して先程停止したコンテナを再起動する

$ docker container start <container ID>

コンテナが再起動されていることを確認

```
$ docker container list
```

※メインプロセスは最初に指定したものから変えられない

もう一度コンテナを停止しておく

```
$ docker container stop <container ID>
```

コンテナの破棄

IDを指定して先程停止したコンテナを破棄する

$ docker container rm <container ID>

破棄したので --all をつけてもコンテナはない

```
$ docker container list --all
```

再起動もできない

```
$ docker start <container ID>    # エラー
```

ここまでのおさらい

`docker image pull`	イメージをダウンロード
`docker container run`	コンテナを起動
`docker container stop|start|rm`	コンテナを停止|再起動|破棄
`docker container list`	コンテナの一覧を表示

※今回はフルコマンドを用いたが、普通はショートカットコマンドを用いる

```
docker image pull                       =>  docker pull
docker container run|stop|start|rm      =>  docker run|stop|start|rm
docker container list                   =>  docker ps    // !!
```

Agenda

- Part 1. Docker の基礎
  - コンテナの起動・停止・破棄
  - **コンテナの様々な起動オプション**
  - 実行中のコンテナに対する操作
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎

--rm : 停止時に即コンテナを破棄

通常はコンテナが停止しても破棄されるわけではない

```
$ docker run ubuntu echo hello

$ docker ps --all    // 停止中のコンテナが見える
```

再利用しないようなコンテナは、停止後即破棄することができる

```
$ docker run --rm ubuntu echo hello

$ docker ps --all    // コンテナは破棄されたので存在しない
```

--detach : バックグラウンドで起動

以下のように起動すると docker コマンド自体はすぐ終了する

```
$ docker run --detach ubuntu sleep infinity
```

しかしコンテナはバックグラウンドで動いている

```
$ docker ps    // バックグラウンドで実行中のコンテナが見える
```

バックグラウンド実行中のコンテナは終了するのを忘れないように

```
$ docker stop <container ID>
```

--interactive --tty：キーボード入力を有効化

以下のようにすると・・・

```
$ docker run --interactive --tty ubuntu bash
```

プロンプトが表示され、キーボード入力を受け付ける普通のシェルとして動作

```
# echo hello    // => hello
```

exit するとシェル（コンテナのメインプロセス）が終了してコンテナも停止

```
# exit
```

--interactive --tty：キーボード入力を有効化

これを応用すればPythonなどのREPLをコンテナ内で起動したりもできる

```
$ docker run --interactive --tty python python
```

それぞれのオプションの意味

- --interactive：コンテナの標準入力を開いておく
- --tty：コンテナの標準入力にターミナル（キーボード入力）を接続する

この２つのオプションを省略して -it と指定することが多い

```
$ docker run -it ubuntu bash
```

--env：環境変数を設定

コンテナ内で参照する環境変数を設定

```
$ docker run -it --env HOGE=fuga ubuntu bash
```

```
# echo $HOGE    // => fuga
```

ファイルから読み込むことも可能

```
$ docker run -it --env-file vars.env ubuntu bash
```

vars.env の例：

```
HOGE=fuga
PIYO=poyo
```

--volume : ホストとコンテナのディレクトリを同期

まずはホストに適当なディレクトリとファイルを作成

```
$ mkdir data && echo hello > data/hello.txt
```

作成したディレクトリをコンテナ内のディレクトリと同期する

```
$ docker run -it --volume $PWD/data:/tmp/data ubuntu bash
```

コンテナの中で同期したファイルを読み書きできる

```
# ls /tmp/data                          // => hello.txt が見える
# echo bye > /tmp/data/hello.txt        // => hello.txt に書き込めり
```

--publish : ホストのポートをコンテナに接続

以下を実行してブラウザで http://localhost:8080/ にアクセスすると・・・

```
$ docker run --publish 8080:80 nginx nginx -g 'daemon off;'
```

Welcome to nginx!

ターミナルに戻ると標準出力にログが出ている

```
172.17.0.1 - - [03/Apr/2018:11:23:37 +0000] "GET / ..." ...
```

MySQLやRailsも同じようにポートを接続して起動できる

デフォルトコマンドを実行

通常は docker run <image> <command> としてコマンドを実行する

```
$ docker run -it python python
```

この <command> を省略するとイメージのデフォルトコマンドが実行される

```
$ docker run -it python    // PythonのREPLが起動
```

イメージにはたいてい何らかのデフォルトコマンドが設定されている

```
$ docker run -it ubuntu                    // => bash が起動
$ docker run --publish 8080:80 nginx       // => nginx が起動
```

ここまでのおさらい

- `docker run` のオプションでコンテナの起動方法を変えられる
- `docker run <image>` でデフォルトコマンドを実行できる

主要なオプションには省略形が用意されている (`docker run --help` で確認できる)

```
--detach       =>  -d
--env          =>  -e
--interactive  =>  -i
--volume       =>  -v
--tty          =>  -t
--publish      =>  -p
```

Agenda

- Part 1. Docker の基礎
  - コンテナの起動・停止・破棄
  - コンテナの様々な起動オプション
  - **実行中のコンテナに対する操作**
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎

実行中のコンテナ内で別のコマンドを実行

コンテナをバックグラウンドで起動

```
$ docker run -d ubuntu sleep infinity
```

ホストからこのコンテナの中に新しく別のコマンドを実行できる

```
$ docker container exec <container ID> ps
```

```
  PID TTY          TIME CMD
    1 ?        00:00:00 sleep
    5 ?        00:00:00 ps
```

ホストとコンテナの間でファイルをコピー

コンテナを起動

```
$ docker run -it ubuntu
```

ホストで適当にファイルを作成し、コンテナの中にコピー

```
$ echo hello > test.txt
```

$ docker container cp test.txt <container ID>:/tmp

コンテナの中でファイルを確認してみる

```
# ls /tmp    // => test.txt があるはず
```

メインプロセスの出力を取得

コンテナをバックグラウンドで起動

```
$ docker run -d -p 8080:80 nginx
```

ブラウザで http://localhost:8080/ にアクセスしてログを残す

コンテナの標準出力・標準エラー出力に記録されたログを確認

```
$ docker container logs <container ID>
```

```
172.17.0.1 - - [03/Apr/2018:11:23:37 +0000] "GET / ..." ...
```

おさらい

`docker container exec`	コンテナ内で別コマンドを実行
`docker container cp`	コンテナとホストの間でファイルをコピー
`docker container logs`	メインプロセスの出力を取得

もちろんショートカットコマンドも使える

```
docker container exec  =>  docker exec
docker container cp    =>  docker cp
docker container logs  =>  docker logs
```

Part 1. はここまで

起動したままのコンテナがあると思うので自分で停止してください

方法はこれまでのスライドを参考に・・・

一通り停止したら下記コマンドで停止済みコンテナを一括破棄できます

```
$ docker container prune
```

Part 2. Dockerfile の基礎

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
  - **イメージのビルド**
  - 主な命令
  - Dockerfile のポイント
- Part 3. Docker Compose の基礎

用語

- **ビルドコンテナ**
  - ビルド中に一時的に作成されるコンテナ
- **ビルドディレクトリ**
  - ホストでイメージのビルド作業を行うディレクトリ (Dockerfile などを置く)
- **ワーキングディレクトリ**
  - ビルドコンテナ内のカレントディレクトリ
- **ビルドキャッシュ**
  - ビルド途中でイメージの内容をキャッシュしたもの

イメージの取得方法

イメージを取得する方法は２つ

1. Webからダウンロードする ← Part 1. で紹介
2. 自分で作る ← 今度はこちら

自分で作る場合でも、普通は既存のイメージをベースにカスタマイズする

これから使うサンプルをダウンロードしておく

```
$ git clone git@github.com:yubessy/docker-handson-example.git
$ cd docker-handson-example
```

Dockerfile

イメージの作成手順を記述するファイル（見たことがあるはず）

```
FROM python:3.6
RUN groupadd appgroup && \
    useradd --system --group appgroup appuser && \
    mkdir /opt/myapp && \
    chgrp -R appgroup /opt/myapp && \
    chown -R appuser /opt/myapp && \
    pip install flask
COPY --chown=appuser:appgroup main.py /opt/myapp/main.py
USER appuser
WORKDIR /opt/myapp
CMD ["python", "main.py"]
```

イメージのビルド

以下のコマンドで Dockerfile からイメージをビルドできる

$ docker image build --tag myapp:v0.0.1 .

--tag <name>:<tag>: ビルドしたイメージに名前とタグを付与]

.: ビルドディレクトリのパス

コマンドとオプションは省略可能

$ docker build -t myapp:v0.0.1 .

ビルドしたイメージの利用

ビルドしたイメージは普通のイメージと同じように使える

```
$ docker run -p 8080:8080 myapp:v0.0.1
```

ブラウザで http://localhost:8080/ にアクセスしてみる

`main.py` のコードも見ておくこと

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
  - イメージのビルド
  - **Dockerfile の主な命令**
  - Dockerfile のポイント
- Part 3. Docker Compose の基礎

FROM : ベースイメージを指定

ベースにするイメージを指定

例: Python 3.6 製アプリケーションなら

```
FROM python3.6
```

原則として Dockerfile は `FROM` で書き始めなければならない

RUN : コマンドを実行

これを使ってベースイメージをカスタマイズしていく

&& や || など、シェルスクリプトと同じ記法が使える

例: apt-get で mysqlclient をインストール

```
RUN apt-get update && \
    apt-get install -y mysqlclient
```

命令途中の改行は \ で

COPY : ファイルをイメージに含める

ビルドディレクトリ以下の任意のファイルをイメージに含める

```
COPY example.txt /tmp/example.txt
```

. や * のようなパターンも使える

```
COPY . .
```

--chown でファイルのオーナーを変更することもできる

```
COPY --chown=appuser:appgroup main.py /opt/myapp/main.py
```

USER, WORKDIR : ユーザやディレクトリを切り替え

コンテナ内の実行ユーザやワーキングディレクトリを切り替える

```
RUN echo `whoami`:`pwd`
# root:/
USER appuser
WORKDIR /tmp
RUN echo `whoami`:`pwd`
# appuser:/tmp
```

RUN だけでなく COPY や CMD の実行ユーザも変わる

ビルド中に RUN で cd や su は使えないことに注意！

CMD：デフォルトコマンドを設定

docker run <image> で実行されるコマンドを指定

["cmd", "arg1", "arg2"] のような記法を推奨

```
FROM ubuntu:latest
# 通常は bash が起動するが、それを上書きできる
CMD ["echo", "hello"]
```

その他

上に挙げたものはよく使う一部の命令のみで、他にも様々な命令がある

- ARG
- ENTRYPOINT

詳しくは公式ドキュメントを参照

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
  - イメージのビルド
  - Dockerfile の主な命令
  - **Dockerfile のポイント**
- Part 3. Docker Compose の基礎

開発中に変更されやすい命令を後に書く

命令を１つ実行する毎にビルドキャッシュが生成されている

これにより命令の実行結果が変わらない場合に２回目以降のビルドが速くなる

→ 開発中に変更されやすい命令ほど後に書いたほうがビルドが速くなる

```
# main.py を編集するとしてキャッシュ1からビルドされる

FROM python:3.6
RUN pip install
# => キャッシュ1

COPY main.py /opt/myapp/main.py
# => キャッシュ2
```

ひとつの RUN や COPY に複数コマンドをまとめる

命令を 1 つ実行する毎にビルドキャッシュが生成されている

→ 命令を細かく分けると無駄にビルドキャッシュが生成される

→ ビルドが遅くなる・ディスクを圧迫する

```
# 良くない例
RUN groupadd appgroup
RUN useradd --system --group appgroup appuser
RUN mkdir /opt/myapp
RUN chgrp -R appgroup /opt/myapp
RUN chown -R appuser /opt/myapp
```

必要なファイルだけを COPY する

COPY . . などで複数のファイルをまとめてコピーするときは

以下のようなコピーすると問題があるファイルに気をつける

- log/ : 開発中に生成されたたくさんのログ
- .env : DBパスワードなどが書かれた環境設定

これらをイメージに含めないよう .dockerignore を書いておく

```
log/
.env
```

アプリケーション実行用のユーザを用意する

例ではアプリケーション実行専用のユーザをわざわざ作っていた

```
FROM python:3.6
RUN groupadd appgroup && \
    useradd --system --group appgroup appuser
# ...
USER appuser
```

これがないと root でプロセスが実行され予期せぬ問題が発生することも

Part 2. はここまで

身近な Dockerfile を読んで、中でどんな処理を行っているか見てみてください

Part 3. Docker Compose の基礎

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎
  - **Docker Compose とは**
  - サービスとネットワーク
  - Docker Compose のベストプラクティス

Docker Compose とは

前回まで:

- Part 1: 単一のコンテナの操作
- Part 2: イメージの作成

今回:

- Part 3: **複数**のコンテナの操作

複数のコンテナを扱う理由

複数のコンテナを操作できると何が嬉しい

1. ひとつずつ docker run とかせずに一度に起動できる
    - 例: APコンテとDBコンテナを一度に起動
2. コンテナ同士がネットワークを介して通信できる
    - 例: APコンテナがDBコンテナにクエリを発行

docker-compose.yml の基本

docker-compose.yml に複数のコンテナをまとめて定義する

```yaml
version: '3' # ファイルフォーマットのバージョン

services:
  app:
    build: ./app
    ports:
      - 3000:3000
    depends_on:
      - db # コンテナの起動順をDBコンテナより後にする

  db:
    image: mysql
    environment:
      MYSQL_ALLOW_EMPTY_PASSWORD: 'yes'
```

docker-compose コマンドの基本

docker-compose.yml の定義を元にコンテナやイメージを操作する

- docker-compose up: コンテナをまとめて起動
- docker-compose down: コンテナをまとめて終了・削除
- docker-compose run: 特定のコンテナを単一で起動
- docker-compose exec: 起動中のコンテナ内でコマンドを実行
- docker-compose pull: イメージをまとめて取得
- docker-compose build: イメージをまとめてビルド

他にもたくさんあるので docker-compose --help で確認しておく

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎
  - Docker Compose とは
  - **サービスとネットワーク**
  - Docker Compose のベストプラクティス

サービスとは

docker-compose.yml の service

= 同一の設定から作成される 1 つ以上のコンテナの集合

※開発環境で使う場合はたいてい 1 サービス 1 コンテナ

```
version: '3' # バージョン

services:
  app: # サービス名
    # 以下イメージ・コンテナの設定
    build: ./app
    ports: ...
```

サービスの主な設定

docker build|run|pull などのコマンドの引数と対応するもの

```
build:       # docker build <path> <- path
image:       # docker pull|run <image> <- image
command:     # docker run <image> <command> <- command
environment: # docker run --env
env_file:    # docker run --env-file
ports:       # docker run --publish
volumes:     # docker run --volume
stdin_open:  # docker run --interactive
tty:         # docker run --tty
```

Docker Compose のネットワーク

docker-compose up で起動されるサービスはデフォルトでは同一のネットワーク内に存在する

サービス名がそのままホスト名になっているので他のコンテナから通信できる

```
version: '3'
services:
  web-server:
    image: nginx:latest
  web-client:
    image: ubuntu:latest
    command: bash -c 'sleep 5; wget http://web-server/'
    depends_on:
      - web-server
```

高度なネットワークの設定

サービスを複数のネットワークに配置することも可能 (開発環境用途ではほぼ使用しない)

```yaml
version: "3"
services:
  proxy:
    build: ./proxy
    networks: [frontend]
  app:
    build: ./app
    networks: [frontend, backend]
  db:
    image: mysql
    networks: [backend]
networks:
  frontend: {}
  backend: {}
```

Agenda

- Part 1. Docker の基礎
- Part 2. Dockerfile の基礎
- Part 3. Docker Compose の基礎
  - Docker Compose とは
  - サービスとネットワーク
  - **Docker Compose のベストプラクティス**

Git Submodule と組み合わせる

複数レポジトリ構成の環境を up 一発で立ち上げられるように

```
.
├── docker-compose.yml
├── db          # DB初期化処理など
├── frontend # フロント (Node.js) の git repo
├── webapp   # Webアプリ (Rails) の git repo
└── batchjob # バッチ処理 (Python) の git repo
```

公式イメージの機能を活用する

MySQLなどの公式イメージには様々な機能がある

```
db:
  image: mysql:5.7
  volumes:
    - ./db:/docker-entrypoint-initdb.d # DB初期化用SQLを自動実行
  environment:
    MYSQL_ALLOW_EMPTY_PASSWORD: 'yes'  # root のパスワードを空に
```

参考: https://hub.docker.com/_/mysql/

YAML のアンカー・エイリアスを利用する

docker-compose.yml の x- から始まる項目は自由に利用できる

これとYAMLのアンカー/エイリアスを組み合わせると便利

```yaml
version: '3.4'
x-db-env-files: &db-env-files # エイリアス
  - ./env/db.env
  - ./env/db.secret
services:
  webapp:
    ...
    env_file: *db-env-files # アンカー
  batchjob:
    ...
    env_file: *db-env-files # アンカー
```

Part 3. はここまで

Part 1, 2, 3 で一通りのことはできるようになったはず

今後は公式ドキュメントや本で勉強してください