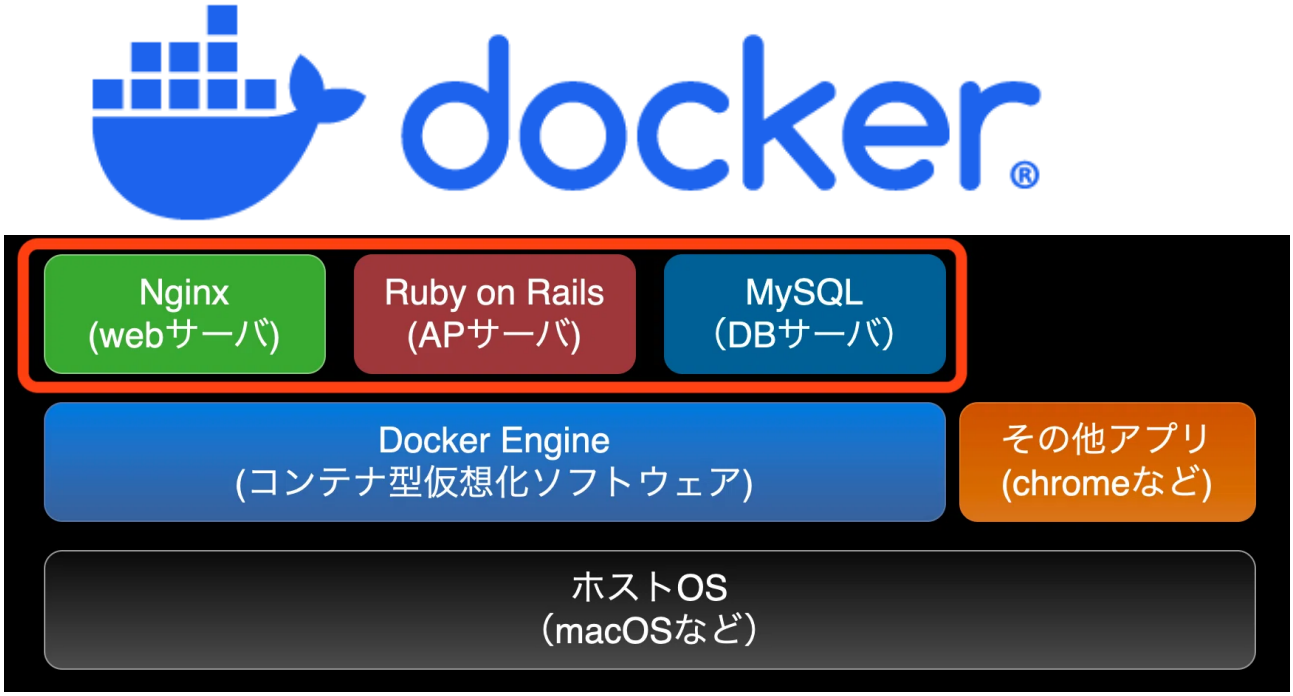
Docker 補助資料

■ Docker

- **コンテナ型仮想環境**を作成、実行、管理することができる
- たとえばRailsを用いた開発を行いたい時など、Railsがインストールされたコンテナを起動することでローカルPC環境を汚すことなく素早くRailsを実行することができる。
- 異なるコンピュータ環境でも簡単に同じ環境をつくることができるため、開発メンバー間で環境を共有したり、開発したコンテナをそのまま本番環境にデプロイすることができる



WindowsやMacOS,Linux系のOSなどにDockerをインストール、Docker上でコンテナとしてアプリケーションを動作させることができます
通常なら、いくつものアプリケーションをバージョンや依存関係に注意しながら、一つ一つ構築していく必要があるところ、コンテナを用いれば、ゲームのカセットを差し替えるくらい手軽にアプリケーションの動作環境を作成できます

Dockerを使用して作成したコンテナ、プログラムを
そのままAWSなどのクラウドサーバーにデプロイし本番環境としてリリースすることも流行っており
以前のように、サーバーを丁寧に構築して慎重に開発を進めていくやり方から、コンテナを使用して環境を構築→うまく動かなければコンテナを削除・修正・再ビルド
といった手返しよく環境を作っていくスタイルが浸透してきました

一台一台のサーバーをペットのように可愛がるのではなく、家畜のように簡単に処分したり増やしたりするというところから「ペットから家畜へ」という言葉が使われます

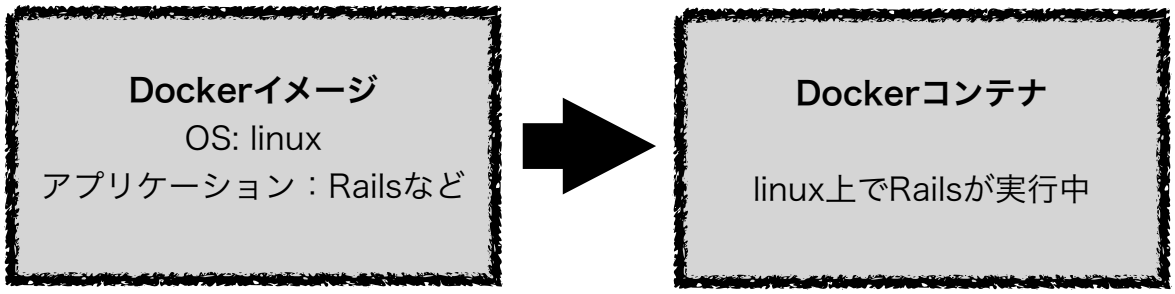
同じような仮想化技術で
ホストOS型仮想化があります
（OracleVirtualBoxやVMware Player）
これらも開発環境をチームメンバーで共有することなどが可能ですが、Dockerをはじめとしたコンテナ型仮想化に比べてホストOS型は少々手返しが良くない印象です
コンテナ型仮想化は、コンテナの作成、利用、削除が軽快に行えるところが魅力のひとつです

いくつかの演習を通してDockerを利用してみましょう

■Dockerイメージ・Dockerコンテナ

コンテナでアプリケーションを実行する一連の手順を学ぶ前にDockerイメージとDockerコンテナの関係性について紹介しておきます

名称	役割
Dockerイメージ	Dockerコンテナを構成するファイルシステムや実行するアプリケーションや設定をまとめたもので、コンテナを作成するための利用されるテンプレートとなるもの
Dockerコンテナ	Dockerイメージを基に作成され、具現化されたファイルシステムとアプリケーションが実行されている状態



コンテナが生成されるときにDockerイメージで定義された内容が具現化され、Docker上で動作するコンテナが作られます
※一つのDockerイメージから複数のコンテナを生成できます
※イメージやコンテナを別マシンにコピーすることもできます

■とりあえず使ってみる

事前準備：dockerをインストール

「helloworld」というファイル名でシェルスクリプトを用意します
※シェルスクリプトはOSなどを操作するコマンドなどをファイルに記述しておき、まとめて実行することができるプログラムのようなものです
変数などの便利機能もありますが、ひとまずシンプルに

```
$ helloworld
1    #!/bin/sh
2
3    echo "Hello, World!"
```

echoコマンドは画面に文字を出力する命令です
今回はRailsやMySQLなどの大きなアプリケーションではなく簡単なスクリプトになりますが、これをアプリケーションとみなして、Dockerで動作させてみます

このアプリケーションをDockerコンテナに詰め込んでいきます
Dockerfileやアプリケーションの実行ファイルからDockerコンテナの元となるイメージを作ることを、Dockerイメージをビルドすると言います
シェルスクリプトと同じフォルダにDockerがどんなイメージを作成・実行するかを定義する**Dockerfile**を作成します
Dockerfileはベース（コンテナの雛形）となるDockerイメージ（OS）を**FROM**で定義できます
ここではUbuntuというOSのDockerイメージを指定します
COPYでは作成したhelloworldをホスト側からDockerコンテナ内の/usr/local/binにコピーしています

RUNはDockerコンテナ内で任意のコマンドを実行できる仕組みです
ここではhelloworldスクリプトに実行権限を与えています
この一連の流れがDockerビルド時に実行され、新たなDockerイメージとして生成されます
CMDは出来上がったイメージをDockerコンテナとして実行する前に行われるコマンドを定義します
ここは事実上、アプリケーションを実行するコマンドを指定することになります

Dockerfileを基にビルド、実行してみましょう
※Dockerfileがあるフォルダでdocker image buildコマンドを実行します

```
$ docker image build -t helloworld:latest .
Sending build context to Docker daemon.  97.5MB
```

ビルドが終わったら、docker container run コマンドで
Dockerコンテナを実行するのが基本的な流れです

```
$ docker containerrun hello world:latest
Hello, World!
```

このようにアプリケーションや必要なファイルを、
Dockerイメージ（OS）に同梱して、コンテナとして実行していくのがDockerの基本的なスタイルです
今回の例はシェルスクリプトをUbuntuに同梱してコンテナとして実行しています

今回はechoを実行するだけのスクリプトを指定しているため、
Dockerコンテナは起動後すぐにスクリプトを実行し終了します
実際の開発では、Dockerコンテナに配置するアプリケーションはWebアプリケーションやAPIアプリケーション、Webサーバー、DBサーバーのように常に稼働し続けるプロセスがほとんどを占めます
今回の例よりもアプリケーションは複雑になりファイル数も多くなりますし、Dockerfileも複雑になります

■ Dockerfile

Dockerfileの内容についてもう少し説明します
※ここで紹介するもの以外にも様々なキーワードがありますが
とりあえずよく使うものを紹介します

FROM

FROMは作成するDockerイメージのベースとなるイメージを指定します
Dockerfileでイメージをビルドする際、まず最初にFROMで指定されたイメージをダウンロードしてから実行されます
FROMで取得するDockerイメージはDocker Hubというレジストリに公開されているものです
RubyやRails、Python、MySQL、などさまざまな言語やフレームワークのイメージを公式が準備してくれており、それを利用しながら独自のイメージを作成することができます
特定のバージョンのRubyを使いたい、、といったときにはバージョンを指定してイメージを取得してくることも可能です

RUN

RUNはDockerイメージビルド時にDockerコンテナ内で実行するコマンドを定義します
RUNの引数にはDockerコンテナ内で実行するコマンドをそのまま指定します

COPY

COPYはDockerを動作させているホストマシン上のファイルやディレクトリをDockerコンテナ内にコピーするための命令です

CMD

CMDはDockerコンテナとして実行する際に、コンテナ内で実行するプロセスを指定します
イメージをビルドするためのRUNに対して、CMDはコンテナ起動時に１度実行されます
RUNでアプリケーションの更新や配置、CMDでアプリケーションそのものを動作させると考えてください
CMDでは１つのコマンドを空白で分割し、配列化した形式で指定します
(例)　$go run /echo/main.go
CMD [“go”, “run”, “/echo/main.go”]

■ DockerでRuby開発環境を構築する

Dockerを利用してRubyの開発環境を構築する例を紹介します

作業用ディレクトリの作成

```
mkdir ruby_study
cd ruby_study
```

- mkdir ruby_study
  作業用ディレクトリ（ruby_study）を作成
- cd ruby_study
  作業用ディレクトリ（ruby_study）に移動

必要なファイルの準備

```
touch Dockerfile docker-compose.yml Gemfile Gemfile.lock
```

touch Dockerfile docker-compose.yml Gemfile Gemfile.lock
各ファイルをまとめて作成

Dockerfileの作成

```
FROM ruby:3.2.2
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev nodejs
WORKDIR /app
COPY Gemfile /app/Gemfile
COPY Gemfile.lock /app/Gemfile.lock
RUN bundle install
ENV LANG=ja_JP.UTF-8
ENV TZ=Asia/Tokyo
```

- FROM ruby:3.2.2
  Dockerイメージとバージョンを指定
- RUN apt-get update -qq && apt-get install -y build-essential libpq-dev nodejs
  Rubyの実行に必要なパッケージをインストール
- WORKDIR /app
  コンテナ内で作業するディレクトリを指定
- COPY Gemfile /app/Gemfile
  ホストマシンからGemfileをコンテナ内にコピー
- COPY Gemfile.lock /app/Gemfile.lock
  ホストマシンからGemfile.lockをコンテナ内にコピー
- RUN bundle install
  bundle installコマンドを実行
- ENV LANG=ja_JP.UTF-8
  環境変数LANGを設定
- ENV TZ=Asia/Tokyo
  環境変数TZ（タイムゾーン）を設定

docker-compose.ymlの作成

Docker-compose.ymlは、複数のDockerコンテナを束ねて一つのサービスとして利用するためのファイルです
※今回はRubyコンテナのみ作るので、ひとつのコンテナのためにdocker-compose.ymlを作りますが、このファイルに複数のコンテナを繋ぐ記述をすることで
複数コンテナを繋げて動作させることができます
（例： Railsコンテナ➕MySQLコンテナ➕NGINXコンテナといった感じ）

```
version: '3.7'
services:
  app:
    build: .
    tty: true
    ports:
      - 3000:3000
    volumes:
      - .:/app
```

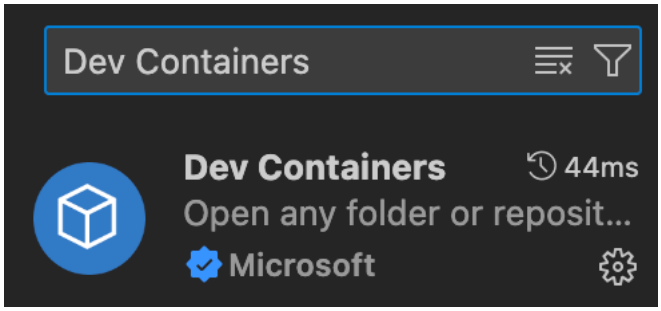
- version: '3.7'
  Docker Composeのバージョンを指定
- services:
  セクション名
- app:
  サービスの名前を設定
- build: .
  ビルドするDockerfileのPathを指定
- tty: true
  コンテナの起動時にttyを有効化する設定
- ports:
    - 3000:3000
  ホストマシンのポート3000をコンテナのポート3000にマッピングする設定
- volumes:
    - .:/app
  ホストマシンとコンテナでファイルを共有する設定（同期みたいなイメージ！）

コンテナの構築・起動

```
docker compose up -d
```

docker-compose.ymlで作成したdocker環境では
上記のコマンドでコンテナを作成し、動作させます
-d オプションは、デタッチモードを指定するものです。このオプションを使用すると、コンテナがバックグラウンドで実行され、ターミナルをブロックせずに他の作業を続けることができます。

このコマンドでRubyのプログラムが実行できる
コンテナが作成されPC上で起動されている状態となりました
コンテナに接続して作業する方法はいくつかありますが、vscodeの拡張機能に「Dev Containers」というものがあり、とても便利です
Vscodeから、動作しているコンテナに接続し、コンテナの中のフォルダやファイルを開けたり、ターミナルからコマンドを実行することができます



Rubyのソースコードを作成
Dev Containersでvscodeから実行中のコンテナに接続したのち、適当なフォルダにRubyのファイルを作成して、実行してみましょう

app.rb

```
def hello_ruby
  puts "Hello Ruby!"
end

hello_ruby
```

```
ruby app.rb
```

うまくいけば、Hello Ruby!とターミナルに表示されたかと思います

ローカルPC環境を汚すことなく、Rubyの開発環境が準備できたことになります

実験が終わったら、コンテナを停止しましょう

```
docker compose down
```

■ DockerでMySQL開発環境を構築する

Dockerを使ってMySQL環境を構築します

作業用ディレクトリの作成

```
mkdir mysql-study
```

必要なファイルの準備

```
touch Dockerfile docker-compose.yml .env
```

.envの作成

環境変数を定義するためのファイルを作成します。

.env

```
MYSQL_ROOT_PASSWORD=mysql
MYSQL_DATABASE=mysql_study
MYSQL_USER=mysql
MYSQL_PASSWORD=mysql
```

MySQLにログインするためのIDやパスワードを設定しています
任意のものに変更することも可能です

Dockerfileの作成

```
FROM mysql:8.0-debian
RUN apt-get update && \
    apt-get install -y locales
RUN locale-gen ja_JP.UTF-8
RUN localedef -f UTF-8 -i ja_JP ja_JP.UTF-8
ENV LANG=ja_JP.UTF-8
ENV TZ=Asia/Tokyo
```

- FROM mysql:8.0-debian
  Dockerイメージとバージョンを指定
- apt-get update
  パッケージリストを更新しより新しいバージョンのパッケージをインストールするための記述
- apt-get install -y locales
  日本語を扱うために必要なパッケージをインストール
- RUN locale-gen ja_JP.UTF-8
  日本語を扱うための記述
- RUN localedef -f UTF-8 -i ja_JP ja_JP.UTF-8
  日本語を扱うための記述
- ENV LANG=ja_JP.UTF-8
  環境変数LANGを設定するための記述
- ENV TZ=Asia/Tokyo
  環境変数TZ（タイムゾーン）を設定するための記述

docker-compose.ymlの作成

```
version: '3.7'
services:
  db:
    build: .
    platform: linux/amd64
    ports:
      - 3306:3306
    volumes:
      - ./db/data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DATABASE}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
```

- version: '3.7'
  Docker Composeのバージョンを指定
- services:
  セクション名
- db:
  サービスの名前を設定
- build: .
  ビルドするDockerfileのPathを指定
- platform: linux/amd64
  Apple silicon搭載のMacで動作させるための記述
- ports:
    - 3306:3306
  ホストマシンのポート3306をコンテナのポート3306にマッピングための記述
- volumes:
    - ./db/data:/var/lib/mysql
  データの永続化のための記述
- environment:
  - rootのパスワードを定義
  - データベースを定義
  - userを定義
  - userのパスワードを定義

特徴的な部分はvolumes:の部分です
コンテナを停止しても、データベース内のデータが消えてしまうのを防ぐため、コンテナ内で扱ったデータの保存先を設定しています
これによりコンテナを停止してもデータが残り続け、データの永続化が実現できます

コンテナの構築・起動

```
docker compose up -d
```

起動が終わったら、前のページで紹介した
vscodeのDev Containersなどを使用して
コンテナ内に接続しましょう

接続できたら、コンテナ内のターミナルを開き
「mysql -u mysql -p」と入力
パスワード「mysql」を入力

MySQLが動作していることを確認してください

実験が終わったら、コンテナを停止しましょう

```
docker compose down
```

■ DockerでRuby on Rails開発環境を構築する