# 本勉強会で扱うプロジェクト

プロジェクトリンクを参照

## 使用場面

モデル・ロード・パターンでは推論サーバを配備する際、サーバイメージを Pull した後に推論サーバを起動し、その後にモデルファイルをロードして推論サーバを本稼働させます。モデルファイルのロード元を変数等で変更することによって、推論サーバで稼働するモデルを柔軟に変更することも可能です。

## 必須要件：

- Python 3.8 以上
- Docker
- Kubernetes クラスターまたは minikube

## フォルダ構造

```
├─
│  .dockerignore
│  Dockerfile
│  makefile
│  README.md
│  requirements.txt
│  run.sh
│  __init__.py
│
├─manifests
│      deployment.yml
│      namespace.yml
│
├─models
│      default_iris_svc.onnx
│      label.json
│
├─model_loader
│      Dockerfile
│      main.py
│      requirements.txt
│      __init__.py
│
└─src
    │  configurations.py
    │  constants.py
    │  __init__.py
    │
    ├─app
    │  │  app.py
    │  │  __init__.py
```

```
    |   |
    |   └─routers
    |           routers.py
    |           __init__.py
    |
    ├─ml
    |       prediction.py
    |       __init__.py
    |
    └─utils
            logging.conf
            profiler.py
            __init__.py
```

# Makefileの詳細（用意されたコマンド）

Makefileで指定されたコマンドについて解説

---

- `make build_all`
  - 推論用Docker イメージおよびモデルロード用Dockerイメージのビルド

注意：

- "Docker build"コマンドについて
  - "-t" オプションでビルドするDockerのレポジトリを指定
  - "-f" オプションで使用するDockerfileを指定

> step1
> 変数の定義は以下の通り

```
DOCKER_REPOSITORY := shibui/ml-system-in-actions
ABSOLUTE_PATH := $(shell pwd)
DOCKERFILE := Dockerfile
IMAGE_VERSION := 0.0.1
MODEL_LOAD_PATTERN := model_load_pattern
MODEL_LOAD_PATTERN_PORT := 8000
MODEL_LOADER := model_loader
```

> step2
> Docker Hub の"shibui/ml-system-in-actions"のレポジトリにある"model_loader_pattern_api"イメージをビルド

参照リンク：DockerHub model_loader_pattern_api_0.0.1

```
.PHONY: build_api
build_api:
    docker build \
```

```
    -t $(DOCKER_REPOSITORY):\
    $(MODEL_LOAD_PATTERN)_api_$(IMAGE_VERSION) \
    -f $(DOCKERFILE) \
    .
```

ここで使用するDockerfileはこちら

```
FROM python:3.8-slim

ENV PROJECT_DIR model_load_pattern
WORKDIR /${PROJECT_DIR}
ADD ./requirements.txt /${PROJECT_DIR}/
RUN apt-get -y update && \
    apt-get -y install apt-utils gcc && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    pip install --no-cache-dir -r requirements.txt

COPY ./src/ /${PROJECT_DIR}/src/
COPY ./models/label.json /${PROJECT_DIR}/models/label.json

ENV LABEL_FILEPATH /${PROJECT_DIR}/models/label.json
ENV LOG_LEVEL DEBUG
ENV LOG_FORMAT TEXT

COPY ./run.sh /${PROJECT_DIR}/run.sh
RUN chmod +x /${PROJECT_DIR}/run.sh
CMD [ "./run.sh" ]
```

基本的には、以下の点に注意：

- 使用されたコマンドの一覧
    - FROM : ベースとなるイメージ
    - ENV : 環境変数の定義
      "PROJECT_DIR"の変数に"model-load-pattern"をいれる。
    - ADD : ファイルをDocker内に移動
    - RUN : 実行
    - COPY : ディレクトリのコピー

> step3
> Docker Hub の"shibui/ml-system-in-actions"のレポジトリにある"model_loader_pattern_loader"イメージをビルド

参照リンク：DockerHub model_loader_pattern_loader_0.0.1

- "-f"オプションで指定されているDockerfileがデフォルトのパスでないことに注意
  ⇒同ディレクトリの"model_loader"フォルダ内のDockerfileを参照している。

```
.PHONY: build_loader
build_loader:
    docker build \
    -t $(DOCKER_REPOSITORY):\
    $(MODEL_LOAD_PATTERN)_loader_$(IMAGE_VERSION) \
    -f $(MODEL_LOADER)/$(DOCKERFILE) \
    .
```

ここで実行されるDockerfileはこちら

```
FROM python:3.8-slim

ENV PROJECT_DIR model_load_pattern
WORKDIR /${PROJECT_DIR}
COPY ./model_loader/requirements.txt /${PROJECT_DIR}/
COPY ./model_loader/main.py /${PROJECT_DIR}/src/main.py
RUN apt-get -y update && \
    apt-get -y install apt-utils gcc && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    pip install --no-cache-dir -r requirements.txt && \
    touch src/__init__.py
```

> step3
> load_apiとbuild_loaderの実行

```
.PHONY: build_all
build_all: build_api build_loader
```

---

- `make deploy`

デプロイするコマンド."apply"コマンドでkubernetesで使用するpodを作成する。("Pod"についてはPodとはを参照のこと) 正確には、リソースの変更を反映し、前回実行時との差分を抽出する。

```
-f [Pod定義ファイル]：
```

Podを定義するyamlファイルを指定する。

```
.PHONY: deploy
deploy:
    kubectl apply -f manifests/namespace.yml
    kubectl apply -f manifests/deployment.yml
```

ここでは、"manifests"フォルダの"namespace.yml"、"deployment.yml"の変更の反映を実行している。（Kubernetesの操作は"kubectl"で操作することできる。Kubernetesの操作については"Kubernetesの基本操作"を参照.）

各.yaml(いわゆるマニフェスト)ファイルには、kubernetesの処理を実行する内容が記載されている。

- `namespace.yml`

名前空間を指定するマニフェスト内容：

```
apiVersion: v1
kind: Namespace
metadata:
  name: model-load
```

- `deployment.yml`

デプロイ時に利用する"Pod"の作成を行う。ここでは、直接"Pod"を作成するのではなく、"Deployment"オブジェクトを生成することで"Pod"の一括生成を行っている。また、このファイルでは"Pod"の作成だけでなく、"service"オブジェクトによる、各Podへの管理を行っている。

"Service"は配下のPodへ通信を振り分けるもので、技術的 にはプロキシやNATなどの仕組みを用いたロードバランサーに相当する。ワーカーノードが異なるにもかかわらず、1つのIPアドレスでアクセスできる。この機構はワーカーノード 内のkube-proxyによって実現されている。

内容：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-load
  namespace: model-load
  labels:
    app: model-load
spec:
  replicas: 4
  selector:
    matchLabels:
      app: model-load
  template:
    metadata:
      labels:
        app: model-load
    spec:
      containers:
        - name: model-load
          image: shibui/ml-system-in-actions:model_load_pattern_api_0.0.1
          ports:
            - containerPort: 8000
          resources:
```

```
            limits:
              cpu: 500m
              memory: "300Mi"
            requests:
              cpu: 500m
              memory: "300Mi"
          volumeMounts:
            - name: workdir
              mountPath: /workdir
          env:
            - name: MODEL_FILEPATH
              value: "/workdir/iris_svc.onnx"
      initContainers:
        - name: model-loader
          image: shibui/ml-system-in-actions:model_load_pattern_loader_0.0.1
          imagePullPolicy: Always
          command:
            - python
            - "-m"
            - "src.main"
            - "--gcs_bucket"
            - "ml_system_model_repository"
            - "--gcs_model_blob"
            - "iris_svc.onnx"
            - "--model_filepath"
            - "/workdir/iris_svc.onnx"
          volumeMounts:
            - name: workdir
              mountPath: /workdir
      volumes:
        - name: workdir
          emptyDir: {}

---
apiVersion: v1
kind: Service
metadata:
  name: model-load
  namespace: model-load
  labels:
    app: model-load
spec:
  ports:
    - name: rest
      port: 8000
      protocol: TCP
  selector:
    app: model-load

---
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: model-load
```

```
    namespace: model-load
    labels:
      app: model-load
  spec:
    scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: model-load
    minReplicas: 3
    maxReplicas: 10
    metrics:
      - type: Resource
        resource:
          name: cpu
          target:
            type: Utilization
            averageUtilization: 50
```

---

- `kubectl -n model-load get pods,deploy,svc`

ここでは、複数のコマンドが実行されていることに注意。

- コマンド1つ目

  ```
  kubectl -n model-load get pods
  ```

ここでは、"model-load"と呼ばれる名前空間を指定し、その名前空間上で作られた"Pod"を確認するコマンド。

- コマンド2つ目

  ```
  kubectl -n model-load get deploy
  ```

  デプロイを確認するコマンド。

- コマンド3つ目

  ```
  kubectl -n model-load get svc
  ```

  サービス一覧を確認するコマンド

---

- `make delete`

```

```

```
.PHONY: delete
delete:
    kubectl delete ns model-load
```

"model-load"の名前空間上に作成したオブジェクトをすべて破棄。

---

# Kubernetesの基本操作

Kubernetesは基本的は"kubectl"で操作する。 基本的な構文は以下の通り。

```
kubectl [コマンド] [オプション]
```

### マニフェストとは

---

kubectlでは、リソース（Kubernetesオブジェクト）を作成したり変更したりする操作をする一方、これらの設定値は数が多く、1つひとつkubectlの引数で設定すると、とても膨大になる。そこでリソースの情報は、ファイルとして記述しておき、それをkubectlに読み込ませるようにするのが一般的。リソースに関するデータを「マニフェスト（Manifest）」と呼び、それを記述したファイルを「マニフェストファイル」と呼ぶ。マニフェストファイルは、JSON形式もしくはYAML形式で記述する。JSON形式は、どちらというと機械的にやり取りすることを目的としたもので、人間がその設定ファイルを読み書きするのであれば、もっぱらYAML形式が使われる。

### 代表的なコマンド

---

| コマンド | 説明 |
|---|---|
| create | リソースの作成 |
| apply | リソースの変更を反映 |
| delete | リソースの削除 |

### Namespace

---

Kubu

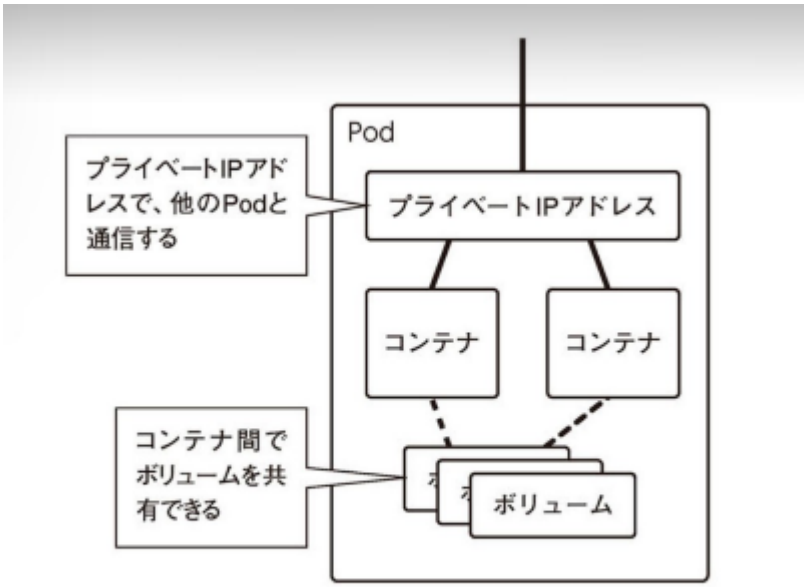
# 単語集

## Podとは

Kubernetesにおける最小実行単位のこと。一つ以上の（基本一つ）コンテナ、いくつかのボリュームを含む。基本的な使い方としては、**最小単位の機能**を表現するものであるので、複数の処理を同時に同じpodに

含めるのは避ける必要がある。ボリュームはコンテナ間のデータ共有に使用が可能. また、POdには動的なプライベートIPアドレスが割り当てられ、含まれるコンテナはそのIPアドレスを共有することで他のPodと通信可能。

- 注意
    - Pod内でのコンテナ同士がlocalhostで通信できる
    - 含まれているコンテナ同士で競合するポート番号を利用できない



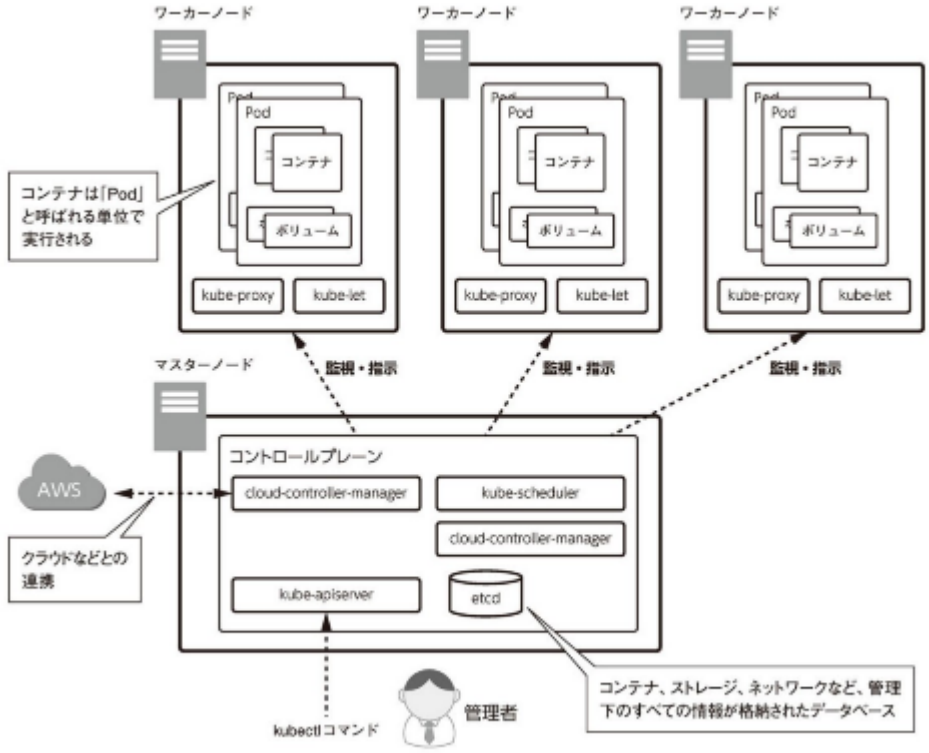

図表9-9　Pod

## kubernetesとは

- Kubernetesを使うモチベーション
    - 冗長性
    - スケーラビリティ
- Kubernetesとは 複数のコンテナ間をつなぐネットワークやストレージについて複数台のサーバーにまたがってコンテナを動かすプラットフォーム。基本的にはDockerコンテナの管理に使用されている。

システムを構成するサーバー群のことを"Kubernetes クラスター"と呼ぶ。クラスター を構成するサーバー群は役割によって「マスターノード」と「ワーカーノード」に分かれる。

- 役割
    - マスターノード
      クラスター全体を統括管理するためのシステムをインストール下サーバー群。管理者はマスターノードを通じて指示出す。
    - ワーカーノード ネットワークやストレージなどを構成し、実際にコンテナを動かすサーバー群。ワーカーノード内では個アンテナ、ネットワーク、ストレージを"Pod"単位で管理する。




図表9-5　Kubernetesの構成

# 参考資料：

Kubenetesのコマンド集： https://qiita.com/suzukihi724/items/241f7241d297a2d4a55c

マニフェストへの記述： https://qiita.com/soymsk/items/69aeaa7945fe1f875822

Kubernetesの名前空間： https://qiita.com/jackchuka/items/a1456d8cab03651ddbf8

kubectl apply -f manifests/ https://qiita.com/tkusumi/items/0bf5417c865ef716b221