令和6年度文部科学省委託「専門職業人材の最新技能アップデートのための専修学校リカレント教育推進」事業

コンテナ技術システム構築教材資料

目次

第一章 Dockerの利用……1

第二章 Kubernetesの基本……35

第三章 Kubernetesのシステム構築……63

第四章 Kubernetesのネットワーク通信……87

第五章 Kubernetesのストレージとデータ管理……111

第六章 Kubernetesのスケジューリング……143

第七章 Kubernetesのリソース管理……171

第八章 Kubernetesの可用性管理……199

第九章 パブリッククラウド上のコンテナサービス①……223

第十章 パブリッククラウド上のコンテナサービス②……253

確認テスト回答……277

ー演習課題ー……281

第一章 Dockerの利用

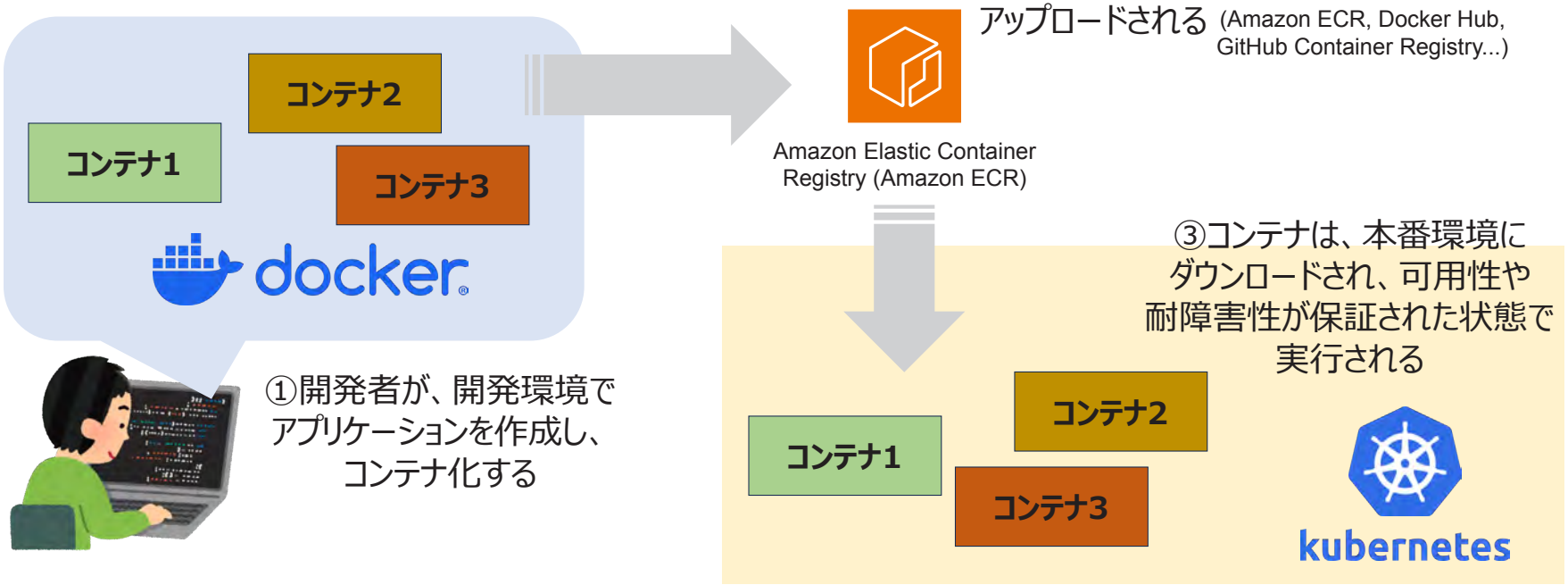
本コース受講の前提

- **本コースは、コンテナの基本知識を有する方を対象としています。**
  受講にあたって、以下の内容を理解していることを前提とします：
  - コンテナと仮想化の違い
  - 名前空間や cgroup を利用したコンテナの隔離方法
  - Dockerの基本コンセプト
  - Docker Hubの役割
  - Linuxの基本コマンド
  - Webアプリケーション開発の一般的な方法

Dockerのシステム開発での位置付け

DockerとKubernetesの一般的な使い分け

- Docker：単一のマシン上で開発環境
- Kubernetes：複数のマシン上で本番環境



第一章 Dockerの利用

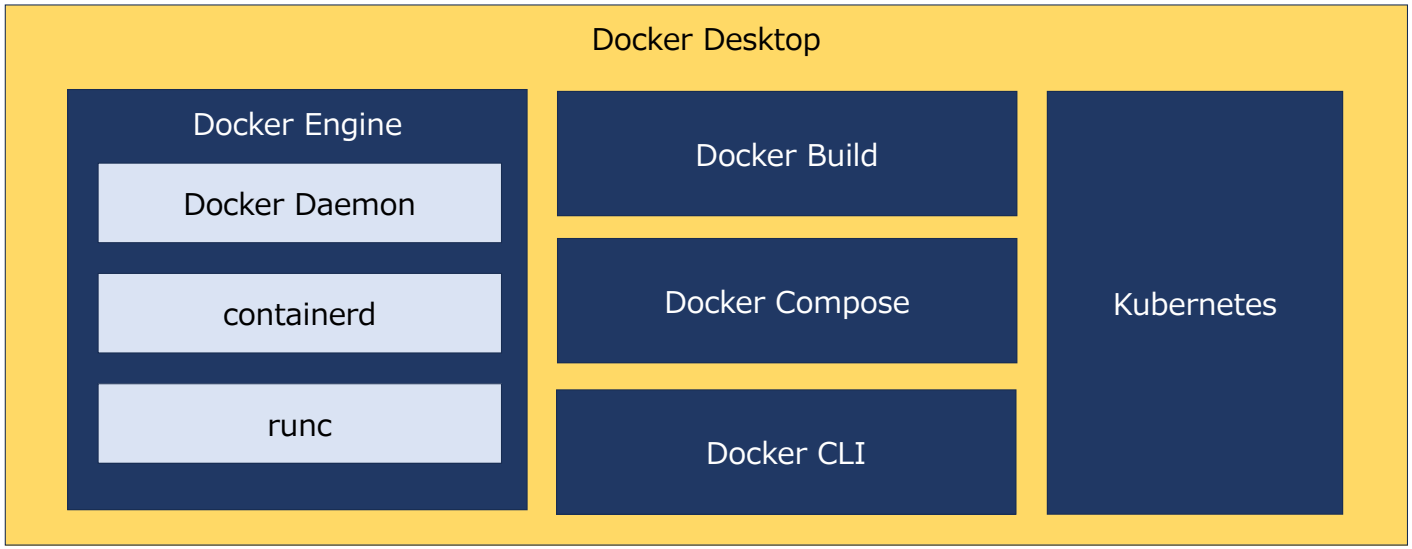
事前準備：Docker Desktop/Docker Engineのインストール

- **後続の演習にDockerを利用するため、Docker DesktopもしくはDocker Engineをインストールしておく必要があります。**

	Docker Desktop	Docker Engine
製品内容	Docker Engineをベースにした、初心者に優しいGUI付きのアプリケーション	コンテナ運用管理ツール。Docker製品群の心臓部
オープンソース	いいえ	はい（Apache License 2.0）
Kubernetes	最小構成のKubernetesを内蔵	なし
対応OS	主にWindowsとMac（LinuxのGnomeやKDEなどのデスクトップ環境にも対応）	主にLinux
コスト	大規模な企業（従業員数250人以上、または年間収益が1,000万米ドルを超える場合）がDocker Desktopを商用利用するには、有料のサブスクリプションが必要	無償

Docker Desktopの仕組み

- 条件によっては有償ですが、Docker Desktopは開発環境においてコンテナの作成やテストを行うツールとして高い人気を誇っています。
- 最近では、テスト用の最小構成のKubernetesをDocker Desktop上で実行できるようになるなど、開発の利便性がさらに向上しています。

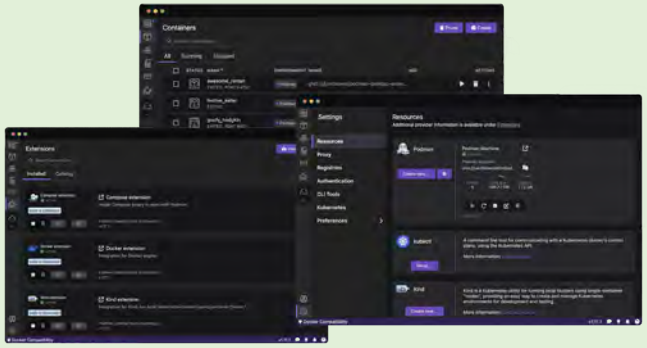


Docker以外の選択肢

- **Docker Desktopを無償で利用できない場合、またはLinux版のDocker Engineを利用したくない場合、WindowsやMacで利用可能な他の選択肢がいくつかあります。**

Podman Desktop

ルートレスコンテナを安全かつ簡単に管理できるオープンソース、無料のGUIツールで、Docker CLI互換性も備えています。



Rancher Desktop

コンテナとKubernetes環境を統合的に管理できるオープンソースのデスクトップアプリケーションです。

※本コースではDocker Desktopを前提としていますが、他のツールでも同様に演習を進めることが可能です。
ただし、コマンドや操作に若干の違いがありますので、各ツールのドキュメントを参照してください。

Dockerを利用するシナリオ

- **Docker開発やテストにおける具体的な利用方法を、3つのシナリオに分けて紹介します。**

シナリオ１　コンテナの実行

コンテナ利用にもっとも一般的な作業はコンテナの実行です。開発環境の立ち上げや各種テストでは、頻繁にコンテナを実行する必要があります。

シナリオ２　コンテナイメージの作成

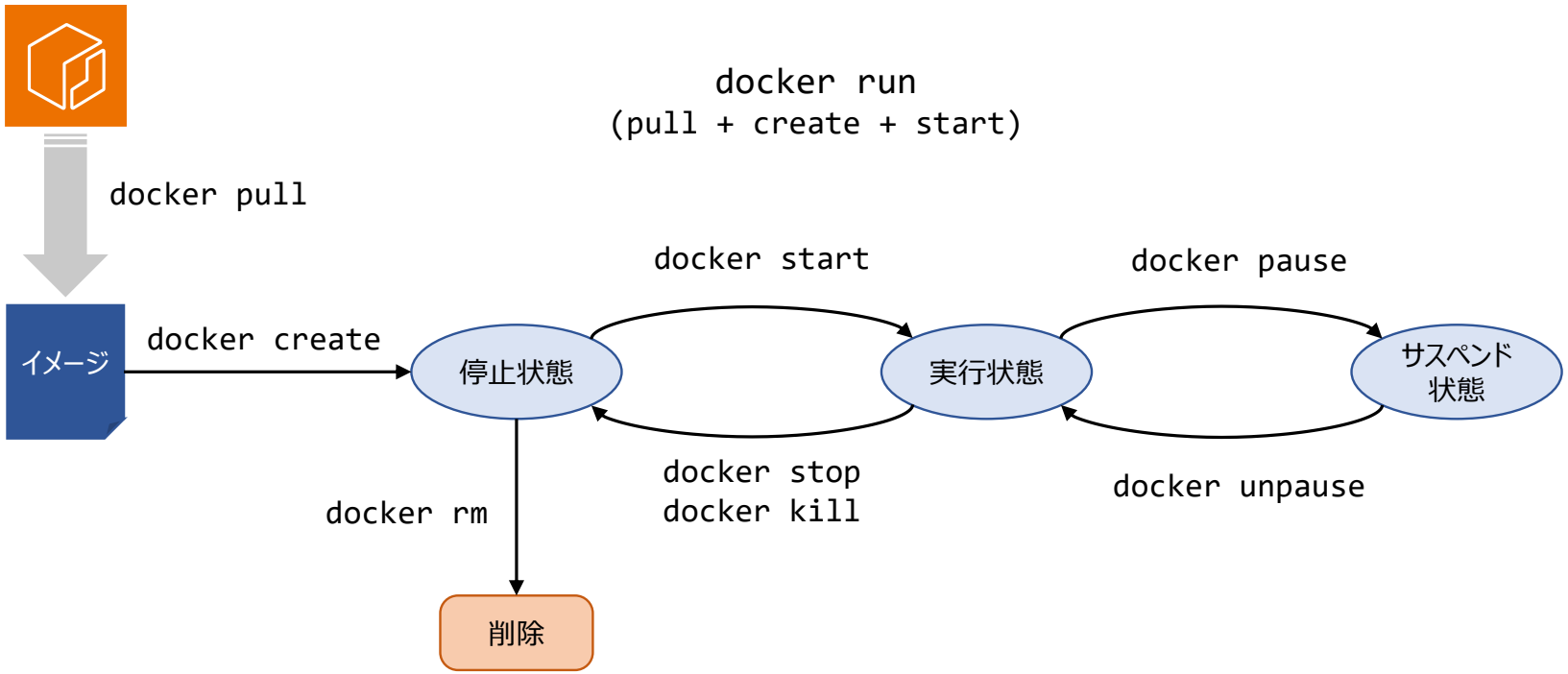
開発したアプリケーションをコンテナ化するというプロセスです。
コンテナは一度作成すれば永続的に利用するわけではなく、アプリケーションの変更があれば、再度コンテナを作成する必要があります。

シナリオ３　複数コンテナの一括管理

通常、アプリケーションは複数のコンテナから構成されます。複数のコンテナの依存関係の管理や、通信の管理などが必要です。

シナリオ１：コンテナの実行

- Dockerでは、Docker CLIを使用してコマンドを実行することで、コンテナのライフサイクル（状態の変化）を管理します。



シナリオ１：コンテナの実行

■ イメージファイルの取得

- デフォルトでは、「docker pull」コマンドはデフォルトのレジストリ（docker.io）からイメージを取得するようになっています。

```
docker pull httpd
```

デフォルトのレジストリより、「http」というイメージを取得（ダウンロード）する

- AWS ECRなど、サードパーティーのレジストリには、まずログインしてからpullを実行します。

```
aws ecr get-login-password | docker login --username AWS --password-stdin https://<account-id>.dkr.ecr.<region>.amazonaws.com
```

- まずはAWS CLIのコマンドで、パスワードを取得
- ユーザー名は「AWS」で固定
- パスワードは、パイプより受け渡される

シナリオ１：コンテナの実行

■ 取得したイメージの確認

ローカルにあるイメージの一覧を表示する

```
docker image ls
```

```
REPOSITORY                                  TAG       IMAGE ID       CREATED         SIZE
ubuntu                                      latest    b1d9df8ab815   4 weeks ago     78.1MB
redis                                       latest    b5e874b32a79   2 months ago    117MB
httpd                                       latest    494b2b45fd74   5 months ago    147MB
```

イメージの詳細を調査する

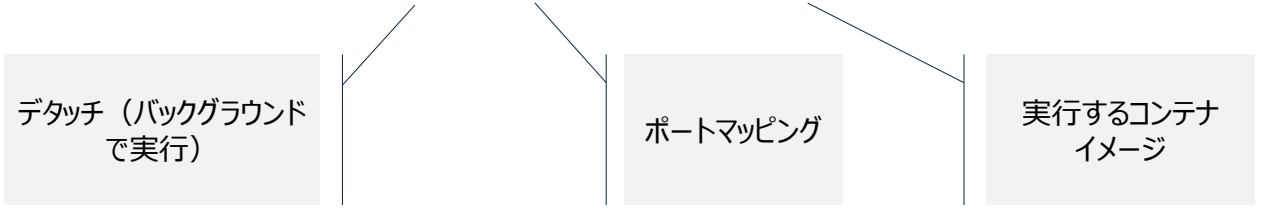
```
docker inspect httpd
```

```
[
    {
        "Id": "sha256:494b2b45fd74cbf7eb7dc9cfeda02b26c9450e26719afaf1914635832217c4ce",
        "RepoTags": [
            "httpd:latest"
        ],
        "RepoDigests": [
            "httpd@sha256:f4c5139eda466e45814122d9bd8b886d8ef6877296126c09b76dbad72b03c336"
        ],
        "Parent": "",
...
```

シナリオ１：コンテナの実行

コンテナの実行

```
docker run -d -p 80:80 httpd
```



実行中のコンテナの確認

```
docker ps
```

実行中のコンテナの一覧を表示する

```
CONTAINER ID   IMAGE     COMMAND              CREATED          STATUS          PORTS                  NAMES
51da63f71e35   httpd     "httpd-foreground"   3 seconds ago    Up 3 seconds    0.0.0.0:80->80/tcp     peaceful_buck
```

シナリオ１：コンテナの実行

コンテナでのコマンド実行、シェルアクセス

```
docker exec 51da63f71e35 pwd
/usr/local/apache2
```

対象コンテナ

実行するコマンド

```
docker exec -it 51da63f71e35 /bin/sh
#
```

ここでコマンドなどを
入力可能

シェルなど、インタラクティブな操作が必要なコマンドを実行する際に、「-it」を用いる

シナリオ１：コンテナの実行

ログ取得

```
docker logs 51da63f71e35
```

```
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2.
Set the 'ServerName' directive globally to suppress this message

AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2.
Set the 'ServerName' directive globally to suppress this message

[Sun Dec 22 11:30:34.893312 2024] [mpm_event:notice] [pid 1:tid 1] AH00489: Apache/2.4.62 (Unix)
configured -- resuming normal operations

[Sun Dec 22 11:30:34.894859 2024] [core:notice] [pid 1:tid 1] AH00094: Command line: 'httpd -D
FOREGROUND'
```

シナリオ１：コンテナの実行

■ コンテナのプライマリプロセスへのアタッチ

```
docker attach 51da63f71e35
```

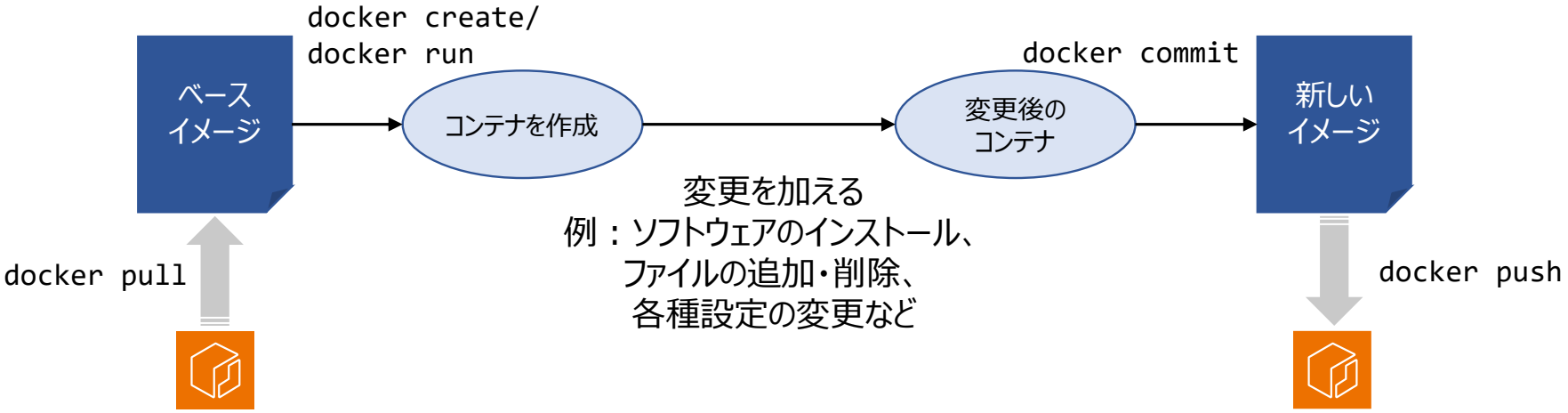
```
^C ⬅ ここで「Ctrl+C」を押す

[Sun Dec 22 11:55:04.731182 2024] [mpm_event:notice] [pid 1:tid 1] AH00491: caught SIGTERM, shutting
down
```

- コンテナのプライマリプロセスとは、コンテナ内で最初に起動されるプロセスであり、通常はコンテナのライフサイクル全体を管理する役割を持つプロセスのことを指します。
- Dockerの場合、CMDやENTRYPOINTで指定されたコマンドがプライマリプロセスとして実行されます（CMDやENTRYPOINTは後述のDockerfileで定義します）。

シナリオ２：コンテナイメージの作成

コンテナイメージを作成するには、主に2つの方法があります。

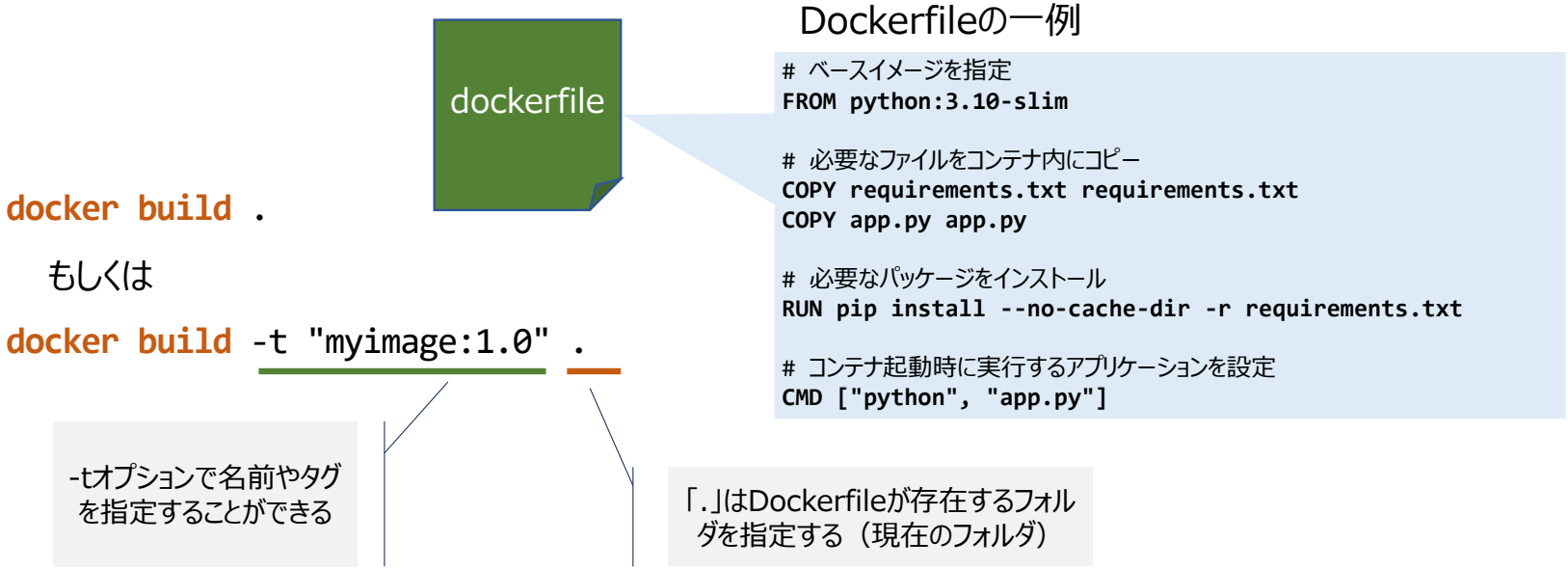
- ＜方法１＞ベースとなるイメージを選定し、そこからコンテナを作成します。
  そのコンテナにログインしてアプリや必要なファイルを追加した後、**`docker commit`**コマンドを使ってコンテナをイメージ化します。



シナリオ２：コンテナイメージの作成

コンテナイメージ作成のもう1つの方法です。

- ＜方法２＞Dockerfileを作成して、方法1の手順を自動化します。再現性や管理の観点から、可能な限りこの方法を利用することを推奨します。
- Dockerfileとは、コンテナイメージを作成するための設計図やレシピのようなもので、テキスト形式のファイルです。



シナリオ２：コンテナイメージの作成

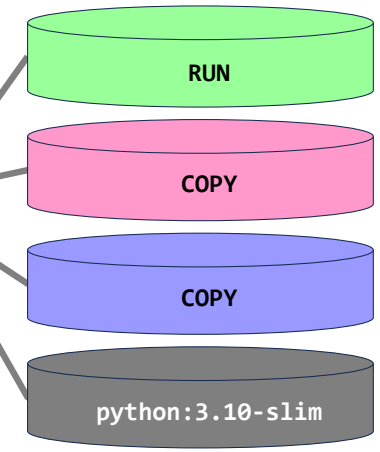
Dockerfileの書き方：よく使うコマンド

コマンド	説明	使用例
FROM	ベースとなるイメージを指定します。Dockerfileはこのコマンドから始まります。	`FROM python:3.9`
RUN	イメージを構築する際に実行するコマンドを指定します。通常はソフトウェアのインストールなどに使用します。	`RUN apt-get update && ¥` `apt-get install -y curl`
COPY	ファイルやディレクトリをホストからイメージ内にコピーします。	`COPY app.py /app/`
ENV	環境変数を設定します。	`ENV APP_ENV=production`
WORKDIR	コンテナ内での作業ディレクトリを設定します。	`WORKDIR /app`
EXPOSE	コンテナがリッスンするポートを指定します。	`EXPOSE 8080`
VOLUME	コンテナで使用するボリュームを指定します。	`VOLUME /data`
CMD	コンテナ実行時に実行されるデフォルトのコマンドを指定します。	`CMD ["python", "app.py"]`
ENTRYPOINT	コンテナ実行時のエントリーポイントを指定します。CMDと組み合わせて引数を指定することも可能です。	`ENTRYPOINT ["docker-entrypoint.sh"]`

シナリオ２ ：コンテナイメージの作成

- **Dockerイメージは、複数のレイヤーが積み重なった状態（スタック）で構成されています。**
  - 最下層にはベースイメージ（例: FROM ubuntu）があり、その上にDockerfileの命令（RUN, COPY, ADDなど）ごとに新しいレイヤーが追加されます。
  - Dockerfileに変更があった場合、レイヤーが再作成されます。
  - イメージからコンテナを実行する際、イメージに存在する各レイヤーは読み取り専用（Read-Only）となり、その上に読み書き可能なレイヤーが一時的に追加されます。これにより、コンテナで加えた変更はイメージに反映されません。

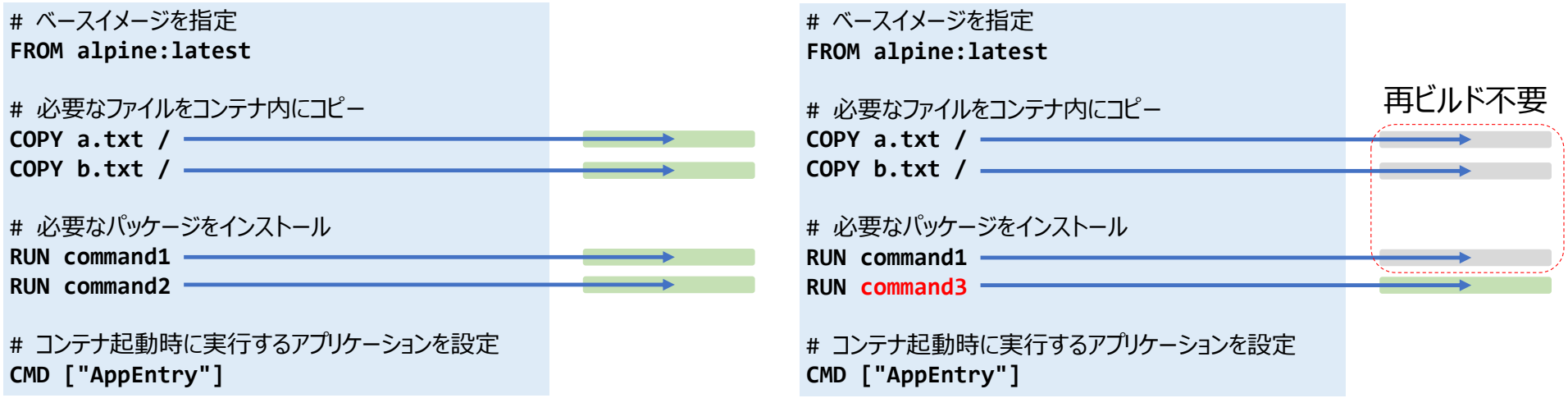
```
# ベースイメージを指定
FROM python:3.10-slim

# 必要なファイルをコンテナ内にコピー
COPY requirements.txt requirements.txt
COPY app.py app.py

# 必要なパッケージをインストール
RUN pip install --no-cache-dir -r requirements.txt

# コンテナ起動時に実行するアプリケーションを設定
CMD ["python", "app.py"]
```



シナリオ２：コンテナイメージの作成

レイヤー化のメリット

- イメージビルド時には、以前のレイヤーのキャッシュが利用されます。例えば、Dockerfile の一部のみを変更した場合、変更された部分以外のレイヤーは再度ビルドせずに流用できます。
- また、同じレイヤーを複数のイメージで共有できるため、ディスク容量の無駄を減らすことができます。



シナリオ２ ：コンテナイメージの作成

■ コンテナイメージの最適化

- 複数のコマンド（COPYやRUN）を1つにまとめることで、余分なレイヤーを作成せずに済み、イメージサイズを小さくすることができます。
- 一方で、キャッシュの活用が制限されるため、コマンド間の関連性を考慮してこの方法を利用する必要があります。

```
# ベースイメージを指定
FROM alpine:latest

# 必要なファイルをコンテナ内にコピー
COPY a.txt /
COPY b.txt /

# 必要なパッケージをインストール
RUN command1
RUN command2

# コンテナ起動時に実行するアプリケーションを設定
CMD ["AppEntry"]
```

```
# ベースイメージを指定
FROM alpine:latest

# 必要なファイルをコンテナ内にコピー
COPY a.txt b.txt /

# 必要なパッケージをインストール
RUN command1 && command2

# コンテナ起動時に実行するアプリケーションを設定
CMD ["AppEntry"]
```

シナリオ２：コンテナイメージの作成

ベースイメージの選定

- イメージのサイズ、アプリケーションに必要なツールやライブラリが含まれているかを考慮して選択します。

ベースディストリビューション	よく使うベースイメージ	
	OSのみ	アプリケーション実行環境付き
なし	scratch 0バイト	
軽量なディストリビューション	busybox 5MB alpine 8MB	python:alpine 45MB
通常のディストリビューション	debian 117MB ubuntu 78MB	python:slim 126MB python 1.02GB

シナリオ２ ：コンテナイメージの作成

■ マルチステージビルド

- 1つのdockerfile内に、複数の「FROM」を利用して、ステージ（段階）を分けて異なる環境をビルドすることができます。イメージを区別するために、FROMの後ろに「as <ステージ名>」をつけます。
- JavaやGoなど、ビルドが必要なアプリケーションでは、ソースコードやビルドツールがコンテナにコピーする必要があります。マルチステージビルドを利用すると、不要なビルドツールやソースコードを最終イメージに含めないため、イメージサイズが大幅に小さくなり、セキュリティリスクが低減します。
- イメージ間のファイルコピーは、「COPY --from=<イメージ名>」を利用します。

```
# ステージ1: ビルド
FROM golang:1.19 as builder
WORKDIR /app

# アプリケーションをコピーしてビルドする
COPY . .
RUN go build -o myapp .
```

1つ目のステージ

```
# ステージ2: 実行用コンテナの作成
FROM alpine:latest
WORKDIR /app

# ビルドステージよりコンパイル済みのバイナリをコピー
COPY --from=builder /app/myapp .

# アプリケーションのエントリーポイント
CMD ["./myapp"]
```

2つ目のステージ

シナリオ２：コンテナイメージの作成

■ イメージの保存

- 作成したコンテナイメージは、そのままdocker runで実行することができますが、他のマシンで使用したり、他の人と共有したりするためには、イメージをどこかに保存する必要があります。
- 保存方法として、Docker HubやAWS ECRなどのレジストリに保存（アップロード）する方法とローカルファイルとして保存する方法があります。

レジストリに保存

```
docker tag -t myapp:latest username/myapp:latest
```

Docker Hubでユーザー登録した際のユーザー名（Docker Hubにイメージを保存するにはユーザー登録が必要）

```
docker push username/myapp:latest
```

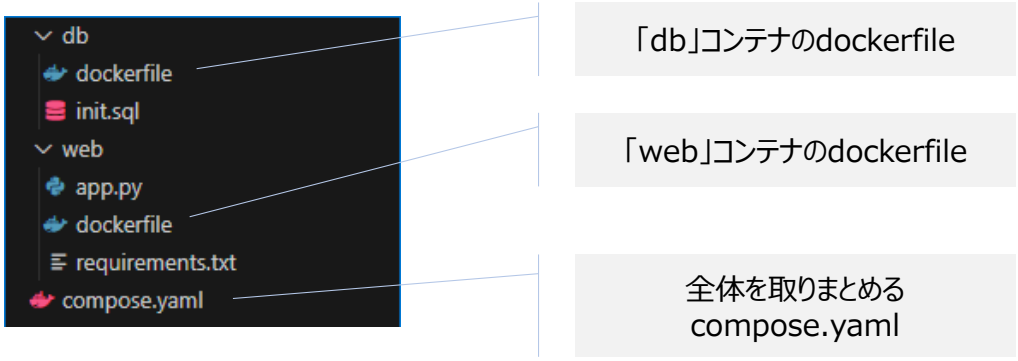
ローカルファイルとして保存

```
docker save -o myapp.tar myapp:latest
```

シナリオ：複数コンテナの一括管理

Docker Compose

- Docker Compose は、複数のコンテナを一括で管理・運用するためのツールです。
- 例えばWebサーバ、データベース、キャッシュサーバなど、複数コンテナで構成されているアプリケーションの場合、すべてのコンテナを一括で起動、停止、再構築でき、各サービスの設定や依存関係を簡単に定義できます。
- Docker Composeは、Docker Desktopに含まれます。ただし、Docker Engine（Linux）に含まれていないため別途インストールが必要です。



`docker compose up`

「docker compose up」コマンドは、自動的にcompose.yamlという設定ファイルを読み込み、その指示に従って必要なコンテナを作成し、起動します。

シナリオ：複数コンテナの一括管理

「compose.yaml」ファイルは、YAML形式で記述されます。

- YAML形式は、人間に読みやすいデータ形式で、JSONやXMLと同様に構造化データを扱う仕様です。

基本的な構造

```
key: value
```

リスト

```
items:
  - item1
  - item2
  - item3
```

オブジェクト（ネスト）

```
person:
  name: "taro takana"
  age: 30
  married: true
```

compose.yamlの例

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

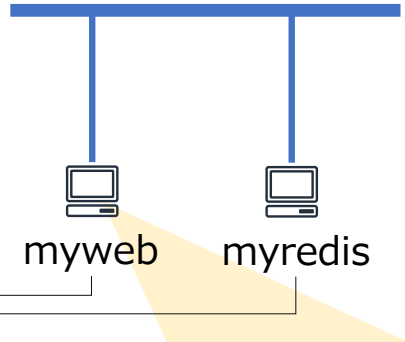
この例では、2つのコンテナが起動されます。
1つは、同じフォルダ内にあるDockerfileの指示に従ってビルドされるWebサーバーです。もう1つは、公式のredis:alpineイメージをそのまま使用して起動します。
また、ポートマッピングなどの追加設定は不要で、2つのコンテナ間は自由に通信できるようになります。

シナリオ：複数コンテナの一括管理

- 「compose.yaml」で起動したコンテナは、「サービス名」で通信できます。
  - サービス名とは、それぞれのコンテナを識別するための名前です。
  - このcompose.yamlの例では、赤枠のmywebやmyredisがサービス名です。

compose.yaml

```
services:
  myweb:
    build: .
    ports:
      - "8000:5000"
  myredis:
    image: "redis:alpine"
```



「web」コンテナの中では、このようなPythonコードを使用して、redisサーバにアクセスできます。

```
import redis
cache = redis.Redis(host='myredis', port=6379)
...
```

シナリオ：複数コンテナの一括管理

Docker Composeのコマンド

```
docker compose up -d
```

compose.yamlに従い、コンテナを起動する。ビルドされていない場合、ビルドも実行する

バックグラウンドで実行する

```
docker compose down
```

コンテナを停止し、削除する

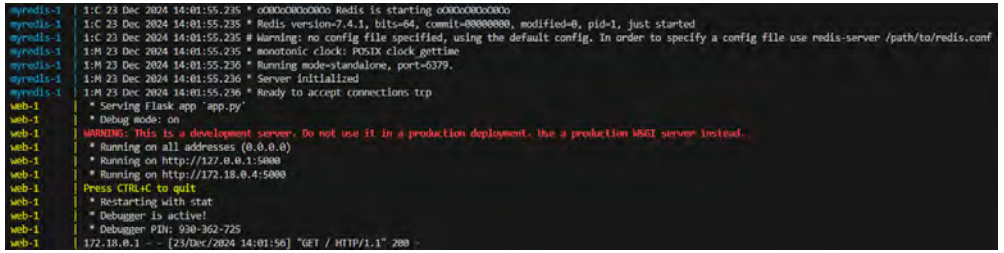
```
docker compose build
```

コンテナをビルドする。Dockerfileやアプリケーションに変更があった場合このコマンドを実行する

docker composeを使用しても、dockerコマンド（docker ps、docker exec等々）は引き続き利用できます。

```
docker compose logs
```

複数のコンテナのログを集約して表示する



シナリオ：複数コンテナの一括管理

コンテナ間の依存関係

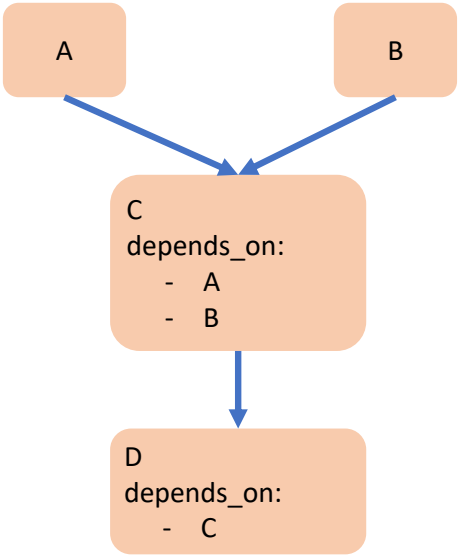
- コンテナ間は依存関係があります。例えば、Webサーバーがデータベースやキャッシュサーバー（Redisなど）に依存している場合、依存するコンテナが先に起動している必要があります。
- このような依存関係を定義するには、「depends_on」を利用します。

```
services:
  web:
    build: .
    depends_on:
      - redis
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

webがredisに依存するように設定する

この設定により

docker compose up : redis → webの順に起動する
docker compose down : web → redisの順に停止する



シナリオ：複数コンテナの一括管理

コンテナの自動再起動

- 意図しない停止を防ぎたい場面などで、コンテナの自動再起動を有効にすることができます。

```
services:
  web:
    build: .
    restart: always
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

restart設定を利用することで、再起動ポリシーを設定できる

再起動ポリシー	動作	主なシナリオ
no	再起動しない	テスト環境、一時的なジョブ、デバッグ作業
on-failure	異常終了時に再起動（成功終了の場合は停止）	回復可能なエラーやクラッシュのある小規模サービス
always	常に再起動（手動停止の場合を除く）	中断不可なサービス、長期間稼働する環境

確認テスト1

Q1: Docker Desktopは、どのような場面で使用しても無償である。この記述は正しいか？最も適切なものを選択してください。

1. 正しい
2. 正しくない

Q2: 実行状態のDockerコンテナを停止するには、以下のどのコマンドを使用できるか？当てはまるものをすべて選択してください。

1. docker stop
2. docker pause
3. docker kill
4. docker terminate

確認テスト2

Q3: docker inspectコマンドの説明として正しいものはどれか？当てはまるものをすべて選択してください。

1. docker inspectの結果は、デフォルトでHTMLで表示される
2. docker inspectの結果は、デフォルトでJSONで表示される
3. docker inspectは、イメージだけでなくコンテナなどの詳細を調べることもできる
4. docker inspectは、イメージの詳細のみを調べることもできる

Q4: 「docker run -d -p 8000:80 nginx」このコマンドの説明として正しいものはどれか？当てはまるものをすべて選択してください。

1. =-dは,コンテナ終了時にデータを保持する指示
2. =-dは,コンテナをバックグラウンドで実行する指示
3. コンテナの8080番ポートを、ローカルホストの80番にマッピングする
4. コンテナの80番ポートを、ローカルホストの8080番にマッピングする

確認テスト3

Q5: あるDockerイメージでは、CMDまたはENTRYPOINTでhttpdをエントリーポイントとして指定している。次の記述のうち正しいものはどれか？当てはまるものをすべて選択してください。

1. 「docker exec -it <コンテナID> httpd」を使用して、そのコンテナ内のhttpdの実行状況を確認できる
2. 「docker attach <コンテナID>」を使用して、そのコンテナ内のhttpdの実行状況を確認できる
3. 「docker exec -it <コンテナID> /bin/sh」を使用して、そのコンテナのシェルにアクセスできる
4. 「docker attach <コンテナID> /bin/sh」を使用して、そのコンテナのシェルにアクセスできる

Q6: あるプログラマーが自作したPythonプログラムをコンテナとして配布しようとしている。Linuxに詳しくないため、可能な限りLinuxでの環境構築をせず、また配布ファイルのサイズを小さくしたいと考えている。この条件に最も適したベースイメージを選択してください。

1. scratch
2. busybox
3. python-slim
4. python

確認テスト4

Q7: あるプログラムをDockerコンテナ化しようとしている。このプログラムはコンパイルが必要で、コンパイルに使用するツールのサイズが非常に大きい。しかし、一度コンパイルすれば実行時にコンパイルツールは必要ない。イメージサイズをできるだけ小さくするには、以下のどの方法を採用すべきか。最も適切なものを選択してください。

1. プログラムをローカルPCでコンパイルしてから、実行可能ファイルだけCOPYコマンドでイメージにコピーする
2. ベースイメージはscratchを使用する
3. 一度コンパイルツールをコンテナにインストールしてから削除する（RUN apt remove）
4. マルチステージビルドを使用する

Q8: Docker Composeで起動した複数のコンテナについて、正しい記述は次のうちどれか？当てはまるものをすべて選択してください。

1. デフォルトで、コンテナ間は、ストレージを共有している
2. デフォルトで、コンテナ間は、サービス名で相互に通信できる
3. コンテナ間の依存関係は、Docker Composeによって自動的に管理される
4. 意図せず停止を防ぐには、「restart」を設定することで自動再起動は可能

第二章 Kubernetesの基本

第二章 Kubernetesの基本

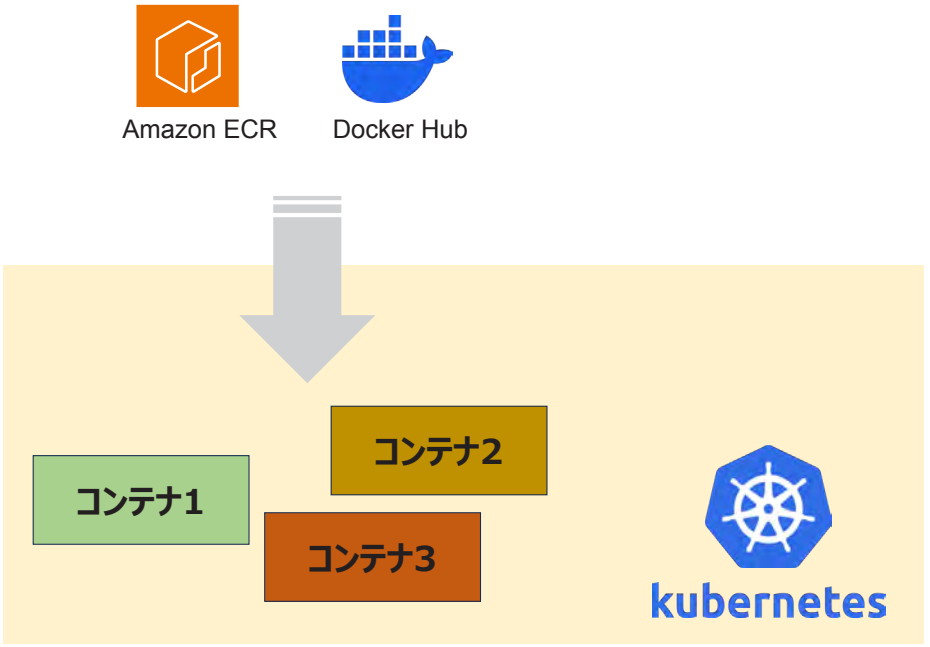
この章の内容

- Kubernetesの概要及びアーキテクチャ、コンポーネント
- Kubernetesのテスト環境の構築
- 演習：kubectlを使用したKubernetesの基本操作
- 補足情報など

Kubernetesのシステム開発における位置づけ

- **Kubernetes（Kubernetes）は、コンテナ化されたアプリケーションのオーケストレーションプラットフォームです。**
  - 「オーケストレーション」という言葉は抽象的に感じるかもしれませんが、いったん以下のように理解して問題ありません：
    オーケストレーション＝「コンテナの実行環境 ＋ 管理 ＋ 自動化」

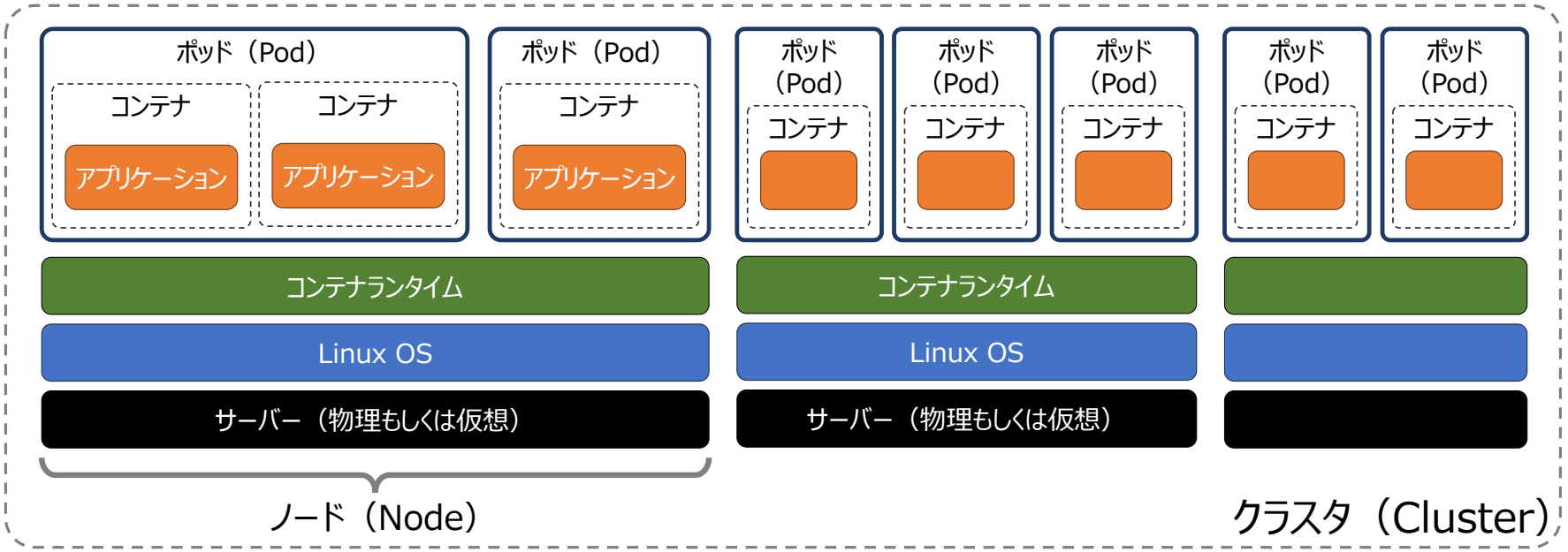
- **Kubernetesは、本番用の実行環境なのでDocker Desktopなどに比べて以下の注意点があります。**
  - **構築手順の複雑さ**： KubernetesのセットアップはDocker Desktopに比べて手順が多く、専門的な知識が求められます。
  - **必要なサーバー数**：可用性を確保するためには、少なくとも3台以上のサーバー（物理サーバーまたは仮想マシン）が必要です。アプリケーションのリソース要件を考慮すると、最低でも5台以上のサーバーが推奨されます。
  - **イメージ作成機能の限定性**： Kubernetesは既存のコンテナイメージを実行するためのプラットフォームであり、イメージの作成自体を行うツールではありません。そのため、イメージ作成に関する機能は限定的であり、別途Dockerや他のイメージビルドツールを使用する必要があります。



Kubernetesの基本用語

理解しておきたい基本用語

- ノード（Node）：Kubernetesクラスタ内の1台の物理サーバーまたは仮想マシン。コンテナを実行するためのリソースを提供します。
- クラスタ（Cluster）：複数のノードで構成されるKubernetesの管理単位。クラスタ内のノードが協力してアプリケーションのデプロイや管理を行います。
- ポッド (Pod)：Kubernetesにおける最小の実行単位。1つ以上のコンテナを含むグループです。

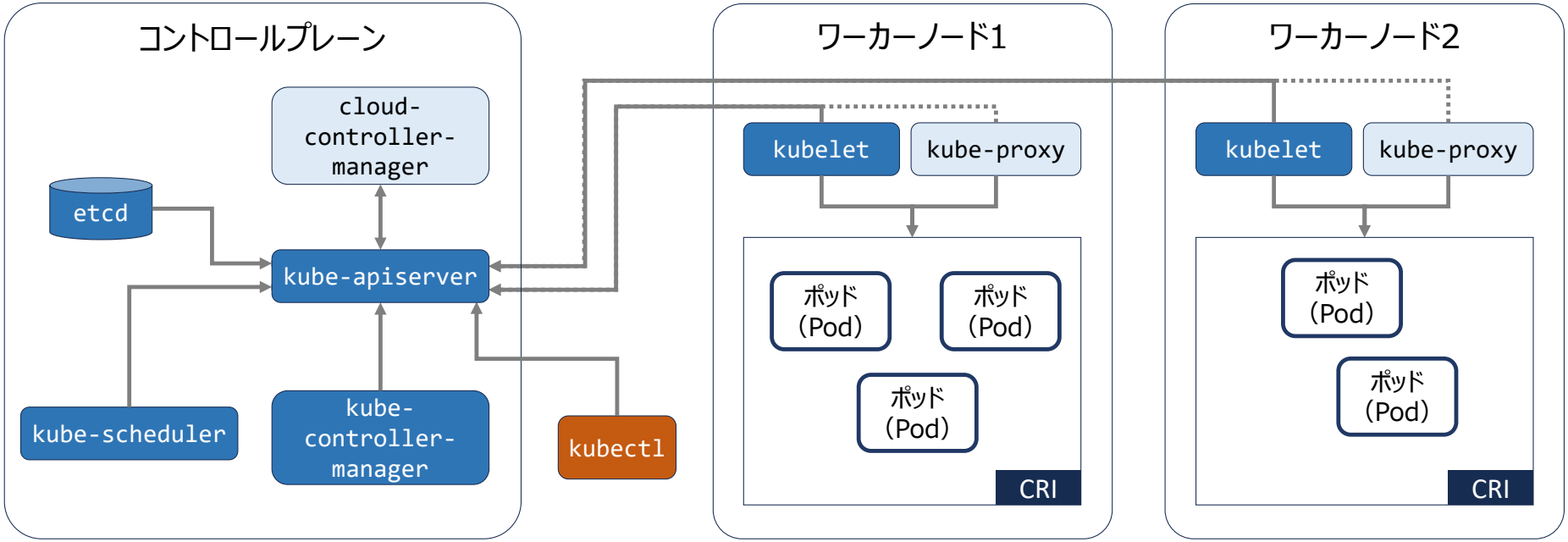


Kubernetesのアーキテクチャ

必須

オプション

- **Kubernetesクラスタは、コントロールプレーンとノードから構成されています。**
  - コントロールプレーン（Control Plane）：クラスタを管理するための主要なコンポーネント群であり、クラスタ全体の状態を監視し、リソースのスケジューリングや管理を行います。
  - ノード（Node）またはワーカーノード（Worker Node）：クラスタ内で実際にコンテナ（ポッド）を実行するためのノードです。



Kubernetesのアーキテクチャ

コントロールプレーン

コンポーネント	説明
kube-apiserver	クラスタのすべてのリクエストを受け付け、リソースの状態を管理するAPIサーバーです。外部からのリクエスト（例：ポッドの作成や削除）を受け入れ、適切に処理します。
kube-scheduler	ポッドをどのワーカーノードに配置するかを決定するスケジューラーです。リソースの使用状況やポッドの要求に基づき、最適なノードにポッドを割り当てます。
kube-controller-manager	クラスタ内のリソースの状態を管理する複数のコントローラを実行します。例えば、ポッドの数を保つためのレプリケーションコントローラや、デプロイメントコントローラなどが含まれます。
etcd	クラスタの状態を永続的に保存する分散型キーバリューストアです。etcd は、すべてのKubernetesリソース（ポッド、サービス、設定など）のデータを管理し、クラスタの整合性を保つ役割を果たします。
cloud-controller-manager（c-c-m）	クラウドプロバイダー（AWS, Azure, GCPなど）と連携してインフラストラクチャ管理を行うコンポーネントです。、クラウドリソース（ロードバランサー、ボリューム、ネットワーク設定など）を動的にプロビジョニングし、Kubernetes クラスタで利用できるようにします。

ワーカーノード

コンポーネント	説明
kubelet	クラスタ内の各ワーカーノードにインストールされるエージェントで、kube-apiserverと通信してポッドの状態を管理します。
kube-proxy	kube-proxy は、ネットワークトラフィックのルーティングと負荷分散を担当し、Kubernetesサービスとポッド間の通信を管理します。

Kubernetesのアーキテクチャ

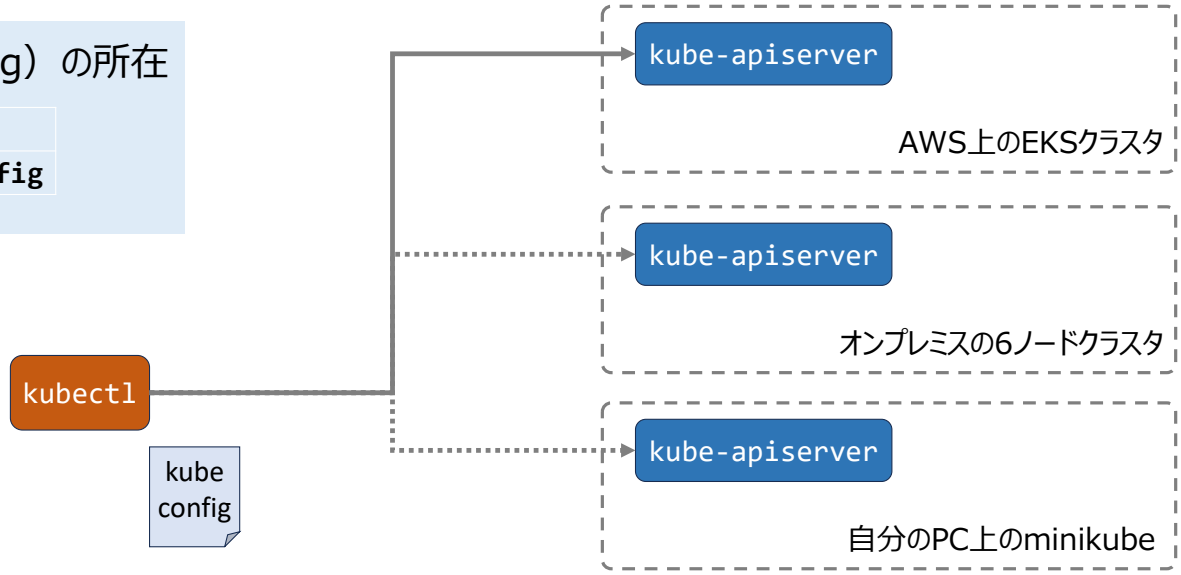
kubectl は、Kubernetes クラスタと対話するためのコマンドラインツールです。

- kubectl は、Kubernetes API サーバー（kube-apiserver）に対してリクエストを送り、クラスタ内のリソース（ポッド、サービス、デプロイメントなど）の状態を取得・更新・削除することができます。
- 通常、Kubernetesクラスタ内ではなく、利用者のPCなどにインストールします。

kubectlの接続設定ファイル（kubeconfig）の所在

Linux	`~/.kube/config`
Windows	`C:¥Users¥<UserName>¥.kube¥config`

kubectlは、
コンテキストを切り替えて、
複数の環境に接続することができます。



Kubernetesのテスト環境の構築

- Kubernetes環境を構築するには、複数台の物理または仮想サーバーが必要になり、構築手順も複雑です。
- しかし、ローカル環境（例: 手元のノートPC）で手軽にKubernetesのテスト環境を構築できるツールも存在します。

minikube

minikubeは、ローカルマシンにKubernetesクラスタを簡単にセットアップできるツールで、テスト環境を迅速に構築するのに適しています。

Windows、macOS、Linuxで動作し、**仮想マシン**または**コンテナ**をノードとして使用してフル機能のKubernetesクラスタを構築できます。また、**複数ノードのクラスタ**にも対応しています。

kindは、「Kubernetes in Docker」の略で、Dockerコンテナを使用してローカルのKubernetesクラスタを構築するためのツールです。

以前は、複数ノードのクラスタやDockerのサポートが差別化ポイントでしたが、最近はminikubeも同様の機能を備えています。

CI/CDパイプラインに統合しやすい点から、特にCI/CD環境でよく使われています。

docker

Kubernetes for Docker Desktop

Kubernetes for Docker Desktopは、Docker Desktopに組み込まれた機能の一つです。

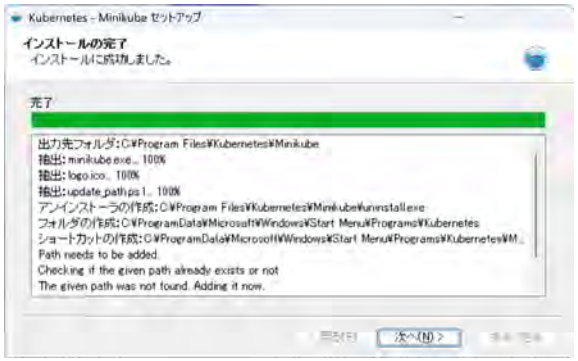
GUIでKubernetesを有効化する設定を切り替えるだけで、Kubernetesクラスタを簡単にセットアップできます。複雑な設定やコマンド操作は不要で、すぐに利用を開始できるのが大きな魅力です。

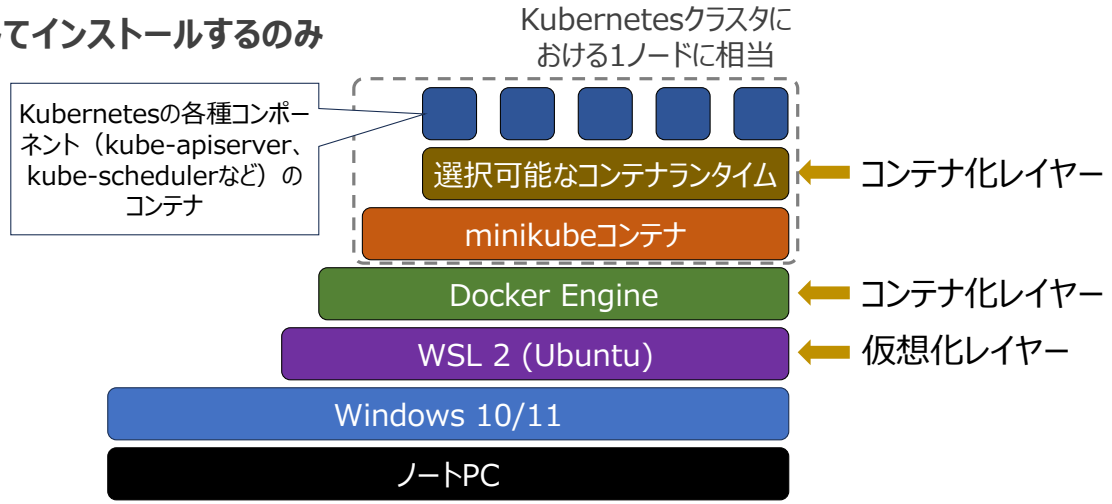
シングルノードのクラスタのみサポートしています。

本コースでは、minikubeとKubernetes for Docker Desktopを例に解説します。次のスライド以降の手順に従い、どちらかをインストールして引き続き受講してください。

minikubeのセットアップ

- **minikubeは、動作要件を満たさないマシンでは動作しません。**
  - 2論理CPU / 2GBメモリ / 20GB以上の空き容量
- **minikubeは、複数の「ドライバー」をサポートしています。**
  - ドライバーとは、「ノード」をシミュレーションするためのバックエンド技術です。
  - サポートしているドライバーは、Docker / VirtualBox / Hyper-V / QEMUなどあります。
  - 最も性能が良いのは、Dockerです。
    本コースでは、Windows + WSL2 + Docker Desktopの構成を例として説明します。
- **Windowsでのインストールは簡単：ダウンロードしてインストールするのみ**





minikubeでのクラスタ構築

- **minikubeをインストールした後、Kubernetesクラスタを構築して起動します。**
  - まず、Docker Desktopを起動し、左下に「Engine Running」が表示されるまで待ちます。
  - 次に、以下のコマンドを実行します。
    `minikube start --driver=docker`

```
C:\>minikube start --driver=docker
* Microsoft Windows 11 Home 10.0.26100.2605 Build 26100.2605 上の minikube v1.34.0
* ユーザーの設定に基づいて docker ドライバーを使用します
* root 権限を持つ Docker Desktop ドライバーを使用
* Starting "minikube" primary control-plane node in "minikube" cluster
* Pulling base image v0.0.45 ...
    > gcr.io/k8s-minikube/kicbase...:  161.63 MiB / 487.90 MiB  33.13% 1.64 MiB
```

Kubernetesのコンテナランタイムを指定することもできます。例えばcontainerdを指定するには：

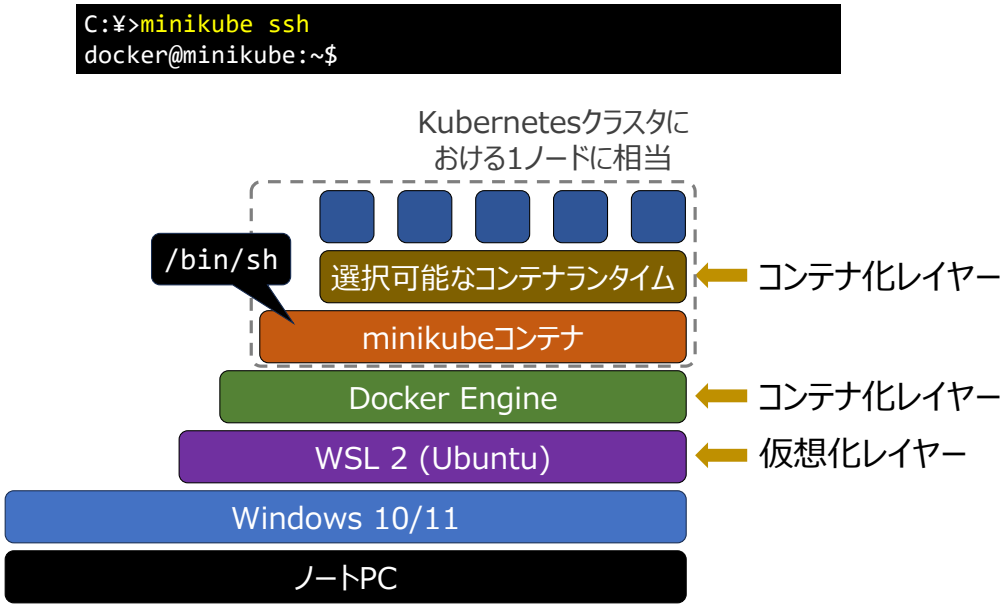
`minikube start --container-runtime=containerd --driver=docker`

起動後、minikube status
コマンドでステータスを確認

```
C:¥>minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

minikubeで構築したクラスタを確認する

- 「minikube status」コマンド以外に、「minikube ssh」というコマンドもあります。こちらは、minikubeのコンテナに直接シェルアクセスする機能です。




```
C:¥>minikube ssh
docker@minikube:~$ sudo systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/lib/systemd/system/kubelet.service; disabled;
vendor preset: enabled)
    Drop-In: /etc/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: active (running) since Tue 2024-12-02 14:00:54 UTC; 2h
11min ago
       Docs: http://kubernetes.io/docs/
   Main PID: 1636 (kubelet)
      Tasks: 24 (limit: 18952)
     Memory: 57.9M
     CGroup: /docker/ad96e3206461259d747
```

上記のように、minikubeのコンテナ内に、kubeletが動作していることを確認できます。
kubeletは、コンテナとしてではなく、デーモンとして動作します。

minikubeで構築したクラスタを停止/削除する

- minikubeで構築したクラスタを停止するには、stopオプションを使用します。

```
minikube stop
```

```
C:\>minikube stop
* 「 minikube」 ノードを停止しています...
* SSH 経由で「 minikube」 の電源をオフにしています...
* 1 台のノードが停止しました。
```

- クラスタを削除する場合は、deleteオプションを使用します。この操作により、クラスタ内のすべてのコンテナやデータが完全に削除されるため、十分に注意して実施してください。

```
minikube delete --all
```

```
C:\>minikube delete --all
* docker の「 minikube」 を削除しています...
* C:\Users\small\.minikube\machines\minikube を削除しています...
* クラスター「 minikube」 の全てのトレースを削除しました。
* 全てのプロファイルの削除に成功しました
```

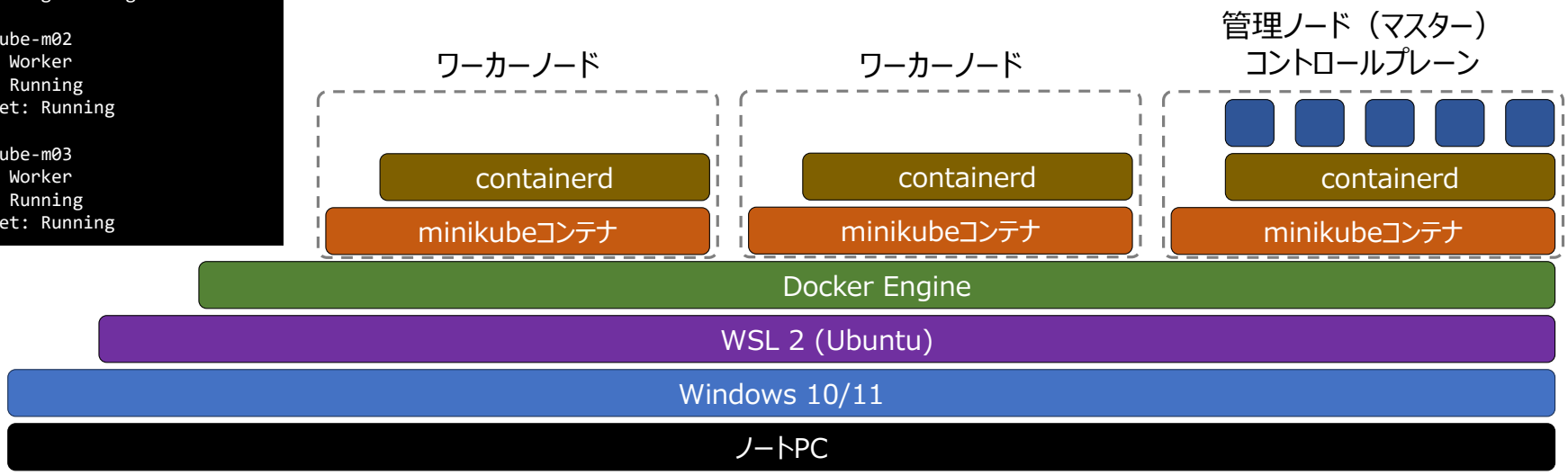
minikubeで複数ノードクラスタを構築する

- 「minikube start --node 3」のように、簡単に複数ノードのクラスタを構築できます：

```
minikube start --nodes 3 --container-runtime=containerd --driver=docker
```

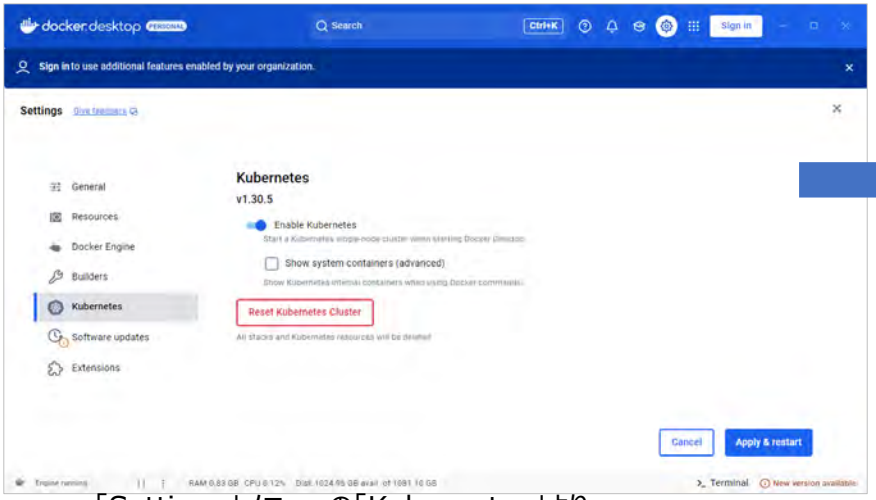
```
C:¥>minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

minikube-m02
type: Worker
host: Running
kubelet: Running

minikube-m03
type: Worker
host: Running
kubelet: Running
```

- 注意点として、minikubeでは、合計ノード数に関係なく、コントロールプレーン用の管理ノードは必ず1つのみ作成されます。その他のノードはすべてワーカーノードとして動作します。

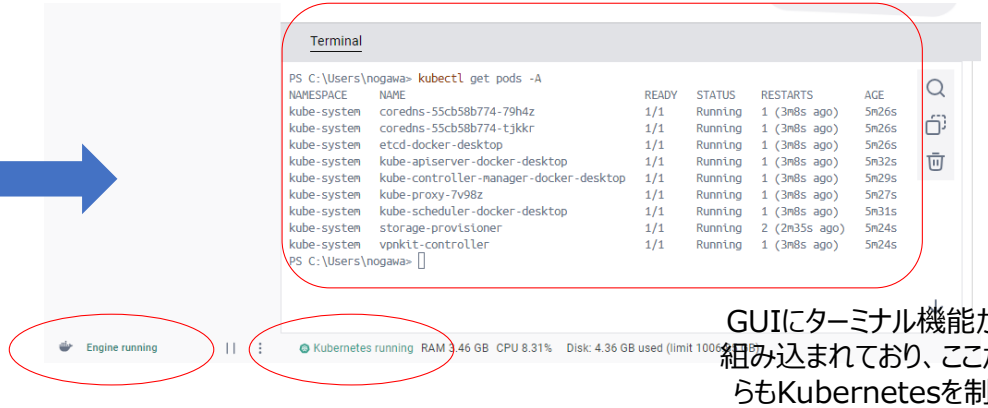


Kubernetes for Docker Desktopのセットアップ

- minikubeのセットアップも簡単ですが、それ以上に手軽なのがDocker Desktopに内蔵されているKubernetes機能です。
- 設定をワンクリックするだけで、Kubernetesを有効または無効に切り替えられます。
- minikubeと比べると、シングルノードのみのサポートなど制限はありますが、初心者がKubernetesを学習する際には十分実用的です。



「Settings」メニューの「Kubernetes」より、「**Enable Kubernetes**」をオンにして、Dockerを再起動するだけで準備完了



Dockerが実行中であることを示す「Engine running」

Kubernetesが実行中であることを示す「Kubernetes running」

GUIにターミナル機能が組み込まれており、ここからもKubernetesを制御するためのコマンドを実行できる

演習

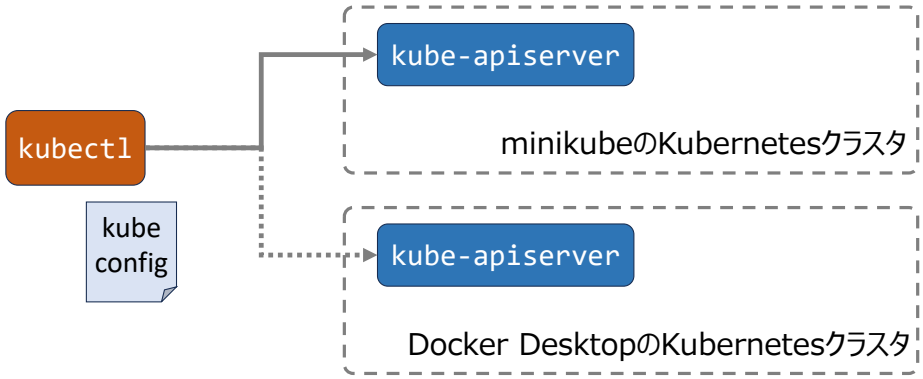
- では、一緒にいくつかの演習を進めてみましょう。
- 演習の前提条件として、minikubeまたはKubernetes for Docker Desktopがすでに設定されていること、そしてインターネット接続があることが必要です。
- なお、minikubeを使用する場合は、事前にminikube start コマンドでクラスタが作成されていることを確認してください。

演習：Kubernetesクラスタの基本操作

- Kubernetesは、基本的にkubectlというCLIツールから操作します。
- minikubeもしくはKubernetes for Docker Desktopでクラスタ構築後、以下のコマンドで構築したクラスタの情報を確認できます。
  `kubectl cluster-info`

```
C:\>kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:47472
CoreDNS is running at https://127.0.0.1:47472/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

- 上記は、minikubeの場合の出力です。Docker Desktopの場合、URLやポート番号など若干異なります。
- もし同じPCにminikubeとKubernetes for Docker Desktop両方インストール/有効化した場合、kubectlを利用する際に、「kubectl config set-contexts」コマンドで操作対象を指定しておく必要があります。



演習：クラスタへのアプリケーションのデプロイメント

- **Kubernetesクラスタが作成されているので、その上にアプリケーションを実際にデプロイしてみます。**
- **この演習でデプロイするアプリケーションは、「echo-server」という、受け取ったリクエストやデータをそのまま返す（HTMLページとして表示する）Webサーバーです。**
- **使用するコマンドは2行です。**
  - ①は、「デプロイメント」というものを作成します。デプロイメントの詳細は次の章で紹介しますが、いったん「1つもしくは複数のコンテナの集合体」と理解して問題ありません。
  - ②は、①で作成したデプロイメントを、外部アクセスを可能にするコマンドです。この2行のコマンドにより、「hello-minikube」というWebアプリケーションは、ネットワーク経由でアクセス可能になります。

①
```
C:¥>kubectl create deployment hello-minikube --image=kicbase/echo-server:1.0
deployment.apps/hello-minikube created
```

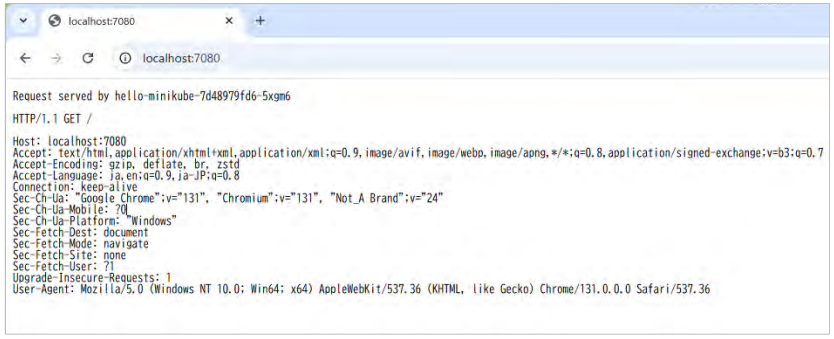
②
```
C:¥>kubectl expose deployment hello-minikube --type=NodePort --port=8080
service/hello-minikube exposed
```

演習：クラスタへのアプリケーションのデプロイメント

- しかし、今回利用する演習環境・テスト環境（minikube/Kubernetes for Docker Desktop）はWindows/WSL2/Dockerの上と結構複雑な構造となっているため、単にブラウザでlocalhost:8080と入力してもアクセスできません。
- その場合、便利な「kubectl port-forward」を使用することで、kubectlが実行しているマシンから、対象クラスタのポッドまでトンネリングすることができます。

```
C:¥>kubectl port-forward service/hello-minikube 7080:8080
Forwarding from 127.0.0.1:7080 -> 8080
Forwarding from [::1]:7080 -> 8080
```

- 「kubectl port-forward」が実行されている状態で、ブラウザで「http://localhost:7080」とたたくと、echo-serverのページが表示されます。port-forwardを終了するには、Ctrl + Cを使用します。



演習：kubectlの基本操作

- ポッド（Pod）はKubernetesクラスタにおける最小の実行単位です。ポッドの一覧を表示するには、以下のコマンドを使用します：
  `kubectl get pods`

- Kubernetesのコントロールプレーンも、実はほとんどコンテナ化されています。以下のコマンドで、コントロールプレーンのポッドを含めて表示することができます。「-A」は、「すべてのネームスペース」という意味です。
  `kubectl get pods -A`

ネームスペース（Namespace）は、クラスタ内でリソース（Podなど）を分離し、管理しやすくするための論理的な区分けです。

先ほど作成したhello-minikubeは、「default」というネームスペースに入っている

コントロールプレーンのポッドは、「kube-system」というネームスペースに入っている

```
C:¥>kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
hello-minikube-7d48979fd6-5xgm6   1/1     Running   0          82m
```

```
C:¥>kubectl get pods -A
NAMESPACE     NAME                                READY   STATUS    RESTARTS      AGE
default       hello-minikube-7d48979fd6-5xgm6     1/1     Running   0             82m
kube-system   coredns-6f6b679f8f-rrwth            1/1     Running   0             10h
kube-system   etcd-minikube                       1/1     Running   0             10h
kube-system   kindnet-7l5fp                       1/1     Running   0             10h
kube-system   kindnet-ghln5                       1/1     Running   0             10h
kube-system   kindnet-q5tj5                       1/1     Running   0             10h
kube-system   kube-apiserver-minikube             1/1     Running   0             10h
kube-system   kube-controller-manager-minikube    1/1     Running   0             10h
kube-system   kube-proxy-k7jgm                    1/1     Running   0             10h
kube-system   kube-proxy-tjgtw                    1/1     Running   0             10h
kube-system   kube-proxy-vfx58                    1/1     Running   0             10h
kube-system   kube-scheduler-minikube             1/1     Running   0             10h
kube-system   storage-provisioner                 1/1     Running   1 (10h ago)   10h
```

※この画面は、minikubeで構成した3ノードクラスタの場合のサンプルです。ノード数や環境によって画面表示が異なります。

演習：kubectlの基本操作

- ノードの一覧を取得するには、こちらのコマンドを利用します：
  `kubectl get nodes`
- また、多くのgetコマンドは、「-o wide」をつけることでより詳細な情報を表示することができます：
  `kubectl get nodes -o wide`
  `kubectl get pods -o wide`

```
C:¥>kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
minikube       Ready    control-plane   10h   v1.31.0
minikube-m02   Ready    <none>          10h   v1.31.0
minikube-m03   Ready    <none>          10h   v1.31.0
```

※この画面は、minikubeで構成した3ノードクラスタの場合のサンプルです。ノード数や環境によって画面表示が異なります。

```
C:¥>kubectl get nodes -o wide
NAME           STATUS   ROLES           AGE   VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION                     CONTAINER-RUNTIME
minikube       Ready    control-plane   11h   v1.31.0   192.168.49.2   <none>        Ubuntu 22.04.4 LTS   5.15.167.4-microsoft-standard-WSL2   containerd://1.7.21
minikube-m02   Ready    <none>          11h   v1.31.0   192.168.49.3   <none>        Ubuntu 22.04.4 LTS   5.15.167.4-microsoft-standard-WSL2   containerd://1.7.21
minikube-m03   Ready    <none>          11h   v1.31.0   192.168.49.4   <none>        Ubuntu 22.04.4 LTS   5.15.167.4-microsoft-standard-WSL2   containerd://1.7.21
```

```
C:¥>kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE    IP           NODE           NOMINATED NODE   READINESS GATES
hello-minikube-7d48979fd6-5xgm6   1/1     Running   0          103m   10.244.2.2   minikube-m03   <none>           <none>
```

演習：kubectlの基本操作

- ポッドやノードなどのさらなる詳細情報を確認するには、以下のコマンドを使用します。

```
kubectl describe pod <PodName>
kubectl describe node <NodeName>
```

```
C:¥>kubectl describe pod hello-minikube
Name:             hello-minikube-7d48979fd6-5xgm6
Namespace:        default
Priority:         0
Service Account:  default
Node:             minikube-m03/192.168.49.4
Start Time:       Wed, 4 Dec 2024 11:02:48 +0900
Labels:           app=hello-minikube
                  pod-template-hash=7d48979fd6
Annotations:      <none>
Status:           Running
IP:               10.244.2.2
IPs:
  IP:           10.244.2.2
Controlled By:  ReplicaSet/hello-minikube-7d48979fd6
Containers:
  echo-server:
    Container ID:   containerd://e94bddebf5badf2a514927ec62aab179c80c60e35fde24807d688963aa01e15f
    Image:          kicbase/echo-server:1.0
    Image ID:       docker.io/kicbase/echo-server@sha256:127ac38a2bb9537b7f252addff209ea6801edcac8a92c8b1104dacd66a583ed6
    Port:           <none>
    Host Port:      <none>
<以下省略>
...
```

※この画面は、minikubeで構成した3ノードクラスタの場合のサンプルです。ノード数や環境によって画面表示が異なります。

演習：kubectlの基本操作

- ポッドやノードなどのさらなる詳細情報を確認するには、以下のコマンドを使用します。

```
kubectl describe pod <PodName>
kubectl describe node <NodeName>
```

```
C:¥>kubectl describe node minikube
Name:               minikube
Roles:              control-plane
Labels:             beta.kubernetes.io/arch=amd64
...
<中略>
...
System Info:
  Machine ID:                 ffc92387ea1c44d3ae2d13130fe15393
  System UUID:                ffc92387ea1c44d3ae2d13130fe15393
  Boot ID:                    56753443-5d70-43a9-933e-168f0fa14930
  Kernel Version:             5.15.167.4-microsoft-standard-WSL2
  OS Image:                   Ubuntu 22.04.4 LTS
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.7.21
  Kubelet Version:            v1.31.0
  Kube-Proxy Version:
PodCIDR:                      10.244.0.0/24
PodCIDRs:                     10.244.0.0/24
<以下省略>
...
```

ワーカーノードの場合は、ここが「<none>」

ここに、クラスタ作成時（minikube start）に、--container-runtimeで指定したランタイムになる（docker, cri-oなど）

※この画面は、minikubeで構成した3ノードクラスタの場合のサンプルです。ノード数や環境によって画面表示が異なります。

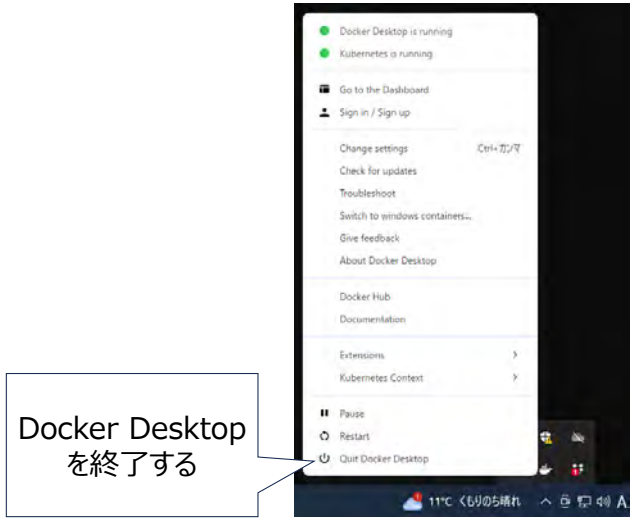
演習：kubectlの基本操作

- 演習の最後に、デプロイしたecho-serverを削除します。
  `kubectl delete deployment <DeploymentName>`

```
C:¥>kubectl delete deployment hello-minikube
deployment.apps "hello-minikube" deleted
```

- minikubeのクラスタは、stopコマンドで停止します。
  `minikube stop`

```
C:¥>minikube stop
* 「minikube-m03」ノードを停止しています...
* SSH 経由で「minikube-m03」の電源をオフにしています...
* 「minikube-m02」ノードを停止しています...
* SSH 経由で「minikube-m02」の電源をオフにしています...
* 「minikube」ノードを停止しています...
* SSH 経由で「minikube」の電源をオフにしています...
* 3 台のノードが停止しました。
```



Docker Desktop
を終了する

- Kubernetes for Docker Desktopでは、クラスタ停止の機能がないため、クラスタを停止したい場合、Docker Desktopを終了するかことになります。

補足情報：minikubeでの複数クラスタ

- Kubernetes for Docker Desktopの場合、1つのクラスタのみ作成可能で、そのクラスタに含まれるノードは、1台のみです。
- 一方で、minikubeの場合、複数ノードのクラスタを、複数作成し同時に起動することができます。また、クラスタ作成の際に、Kubernetesのバージョンを指定することも可能です。例えば古いバージョンを指定して、アプリケーションとの互換性確認などに便利です。

```
C:¥>minikube start -p test1 --container-runtime=cri-o --kubernetes-version=v1.29.5
```

```
C:¥>minikube profile list
|----------|-----------|------------|--------------|------|---------|---------|-------|----------------|--------------------|
| Profile  | VM Driver |  Runtime   |      IP      | Port | Version | Status  | Nodes | Active Profile | Active Kubecontext |
|----------|-----------|------------|--------------|------|---------|---------|-------|----------------|--------------------|
| minikube | docker    | containerd | 192.168.49.2 | 8443 | v1.31.0 | Running |     3 | *              |                    |
| test1    | docker    | crio       | 192.168.58.2 | 8443 | v1.29.5 | Running |     1 |                | *                  |
|----------|-----------|------------|--------------|------|---------|---------|-------|----------------|--------------------|
```

- 複数クラスタの場合、minikubeでは複数の「profile」として管理します。minikubeの各種コマンドは、「-p」で明示的にprofileを指定するか、「minikube profile」コマンドで、profileを切り替えて使用します。

```
C:¥>minikube profile test1
* 無事 minikube のプロファイルが test1 に設定されました
```

```
C:¥>minikube status
test1
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

```
C:¥>minikube status -p minikube
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
<以下省略>
...
```

- 同様に、kubectlのコンテキストも切り替えが必要です。

```
C:¥>kubectl config get-contexts
CURRENT   NAME       CLUSTER    AUTHINFO   NAMESPACE
          minikube   minikube   minikube   default
*         test1      test1      test1      default

C:¥>kubectl config set-context minikube
Context "minikube" modified.
```

補足情報：minikube内蔵のkubectl

- kubectlのバージョンを確認すると、kubectlのバージョンがサーバー（クラスタ）のバージョンと異なることがあります。
- これは、今回構築したテスト環境のkubectlがDocker Desktopに付属しているものであり、一方でクラスタはminikubeを使用して構築されているためです。
- kubectlのバージョンとクラスタのバージョンをできるだけ一致させることが推奨されますが、必須ではありません。一般的に、kubectlはクラスタに対してAPIリクエストを送信するクライアントツールであり、若干のバージョンの違いは許容される場合があります。

```
C:¥>kubectl version
Client Version: v1.30.5
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: v1.31.0
```

- minikubeで作成したクラスタと同じバージョンのkubectlを利用したい場合、minikube内蔵のkubectlを使用することができます。minikube kubectlコマンドで呼び出すことができます。

```
minikube kubectl -p test1 -- get pods -A
```

このコマンドの注意点として、
--より前の部分のパラメータやオプションは**minikube**に適用され、
--より後ろの部分のパラメータやオプションは**kubectl**に適用されるという点です。

```
C:¥>minikube kubectl -p test1 -- version
Client Version: v1.29.5
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: v1.29.5
```

確認テスト1

Q1: Kubernetesのコントロールプレーンに含まれるコンポーネントの説明として、正しいのはどれか？正しいものをすべて選択してください。

1. kube-apiserverは、外部からのリクエスト（例えばPodの作成）を受け付けて処理する
2. kube-schedulerは、Podをどのノードに配置するかを決定する
3. etcdは、コンテナのネットワーク機能を抽象化し、コンテナ同士が通信して連携することを可能にする
4. cloud-controller-managerは、クラウドプロバイダー（AWS

Q2: Kubernetesの用語についての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. ポッド（Pod）：Kubernetesにおける最小の実行単位。1つのPodは1つのコンテナである
2. ノード（Node）：コンテナを実行するためのリソースを提供する。AWSの場合はEC2などがノードに当たる
3. クラスタ（Cluster）：複数のノードで構成されるKubernetesの管理単位
4. ネームスペース（Namespace）：複数のクラスタを束ね、管理しやすくするための論理的な区分け

確認テスト2

Q3: minikubeで構築したKubernetesクラスタについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. １クラスタあたり、複数ノードを構成することができる
2. 複数のクラスタを構成できる
3. Dockerをドライバとして使用するため、Dockerがない場合は動作しない
4. 様々なコンテナランタイム（containerd
5. crioなど）をサポートする

Q4: Kubernetes for Docker Desktopについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. １クラスタあたり、複数ノードを構成することができる
2. 複数のクラスタを構成できる
3. Docker Desktopに含まれているため、別途インストールする必要はない
4. 様々なコンテナランタイム（containerd
5. crioなど）をサポートする

確認テスト3

Q5: Kubernetesの基本的な操作方法について、正しいのはどれか？最も適切なものを選択してください。

1. 通常は、kubectlというコマンドを経由して操作する
2. 通常は、デフォルトで備え付けのWeb管理画面から操作する
3. 通常は、プログラムを作成してAPI経由で操作する
4. 通常は、minikubeというコマンドを経由して操作する

第三章 Kubernetesのシステム構築

第三章 Kubernetesのシステム構築

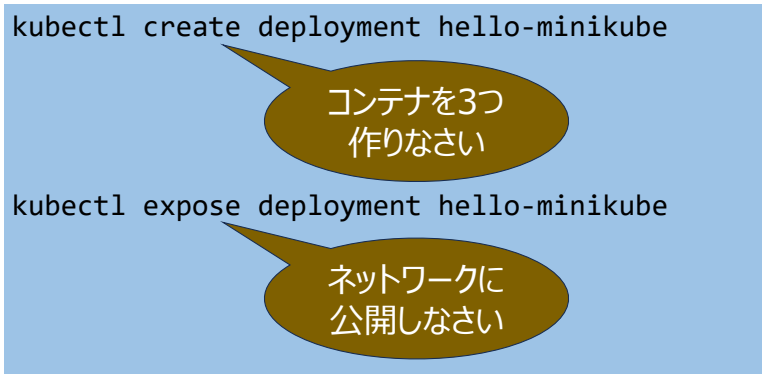
この章の内容

- マニフェストによるデプロイメント
- デプロイメント（Deployment）の利用

Kubernetesクラスタのデプロイ方法

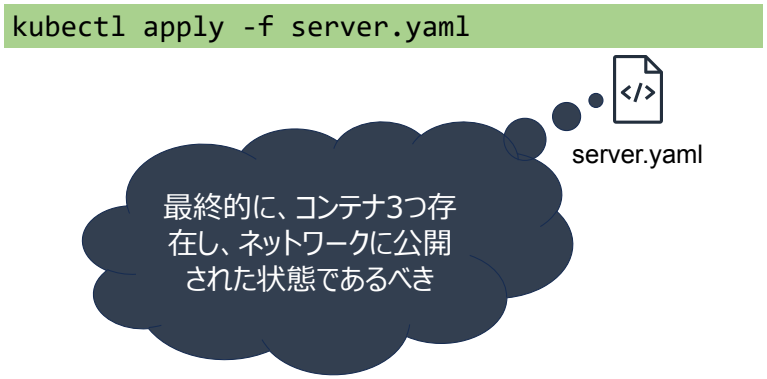
Kubernetesは、2種類のデプロイ方法をサポートしています。

命令型（imperative）



- システムが実行すべき手順や順序を具体的に記述します。
- 手順を明示的に指定するため、システムの挙動を細かく制御できる一方、システムの状態の管理が必要です。システム状態の認識に間違いがあると、期待と異なる結果になる可能性があります。

宣言型（declarative）



- 管理者は「最終的にどうあるべきか（Desired State）」をマニフェストというファイルに記述します。
- Kubernetesがその状態に到達するように、自動的にリソースを調整します。
- 手動で細かい調整を行う必要が減り、運用がシンプルになります。

Kubernetesのマニフェスト

- マニフェスト（manifest）は、Kubernetesクラスタ内でリソース（Pod、Service、Deployment、ConfigMap、Secret など）を宣言的に定義したファイルです。通常、YAML形式が用いられます。
- マニフェストは、主に以下の部分から構成されます。

```
# マニフェストのサンプルです。
apiVersion: v1
kind: Pod
metadata:
  name: webserver
  labels:
    component: web
spec:
  containers:
    - name: echo-server
      image: kicbase/echo-server:1.0
      ports:
        - containerPort: 8080
```

「＃」から始まる行は、コメントです。

apiVersionと、対象リソースの種別です。
この例では、「Pod」という種類のリソースを作成します。

名前などのメタデータです。

対象リソースのスペック、つまり理想状態です。
この例では、Podの中に含まれるコンテナのイメージやネットワークポート
など宣言的に定義しています。

演習：マニフェストでecho-serverをデプロイする

- VS Codeやテキストエディタなどで、webserver.yamlを作成します。内容は、左側のとおりにしてください。また、インデントに十分注意してください。
- 「kubectl apply」コマンドで、作成したマニフェスト（YAMLファイル）を適用します。

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
  labels:
    component: web
spec:
  containers:
    - name: echo-server
      image: kicbase/echo-server:1.0
      ports:
        - containerPort: 8080
```

webserver.yamlファイル
（マニフェスト）

```
C:¥pj>kubectl apply -f webserver.yaml
pod/webserver created
```

kubectl get podsで、STATUSが**Running**になっていることを確認してから、port-forwardを実行します。（STATUSがRunningになるまで、数分間かかることがあります）

```
C:¥pj>kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
webserver   1/1     Running   0          4m38s

C:¥pj>kubectl port-forward pod/webserver 7080:8080
Forwarding from 127.0.0.1:7080 -> 8080
Forwarding from [::1]:7080 -> 8080
```

ブラウザでhttp://localhost:7080を開き、echo-serverが動作することを確認します。

演習：マニフェストでecho-serverをデプロイする

`apiVersion: v1`	APIのバージョンを指定する
`kind: Pod`	作成しようとしているリソースの種類（kind）は、Podと指定する
`metadata:`	リソースのメタデータ（名前や識別情報など）を定義する
`  name: webserver`	ポッド名は、「webserver」とする
`  labels:`	ラベル（キーと値のペア）を定義する
`    component: web`	キーがcomponent、値がwebというラベルをPodに付与する
`spec:`	Podの仕様（スペック）を定義する
`  containers:`	Pod内に含まれるコンテナの一覧を定義する
`    - name: echo-server`	1個目のコンテナは、名前が「echo-server」とする
`      image: kicbase/echo-server:1.0`	コンテナイメージを指定する
`      ports:`	公開するポートを宣言する（なくても通信はできる）
`        - containerPort: 8080`	コンテナ内部で8080番ポートが公開されていることを宣言する

演習：マニフェストでecho-serverをデプロイする

- マニフェストを利用した「get」「describe」「delete」コマンドもあります。

```
C:¥pj>kubectl get -f webserver.yaml
NAME        READY   STATUS    RESTARTS   AGE
webserver   1/1     Running   0          34m


C:¥pj>kubectl describe -f webserver.yaml
Name:             webserver
Namespace:        default
Priority:         0
Service Account:  default
Node:             minikube-m02/192.168.49.3
<以下省略>
...

C:¥pj>kubectl delete -f webserver.yaml
pod "webserver" deleted
```

演習：マニフェストでecho-serverをデプロイする

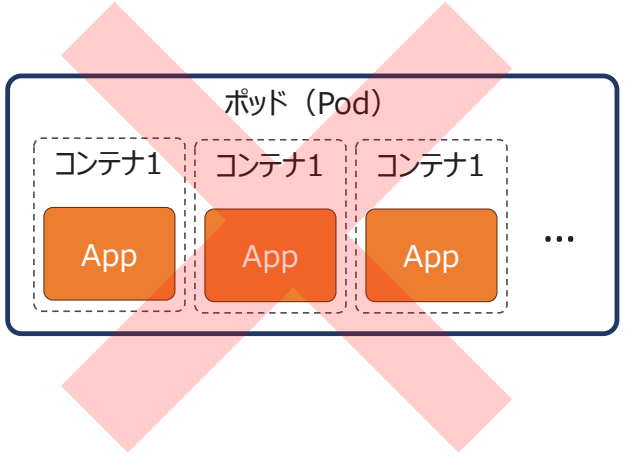
- **「kubectl apply -f webserver.yaml」もう1回実行します。Podは2つ作成されましたか？**
  - 命令型のコマンド（例えばkubectl create）を2回実行すると、リソースは2つ作成されます（同じ名前を指定すると2回目の作成に失敗しますが）
  - 宣言型のコマンド（例えばkubectl apply）を何回実行しても、リソースが再度作成されることはありません。
  - この「ある操作を何度行っても同じ結果になる性質」は、「冪等（べきとう）性」と言います。
- **冪等性は、IaC（Infrastructure as Code）において非常に重要です。**
  - IaCでは、インフラストラクチャがコードとして定義され、繰り返し適用できるようになります。冪等性を保証することで、何度実行してもリソースの状態が一貫しており、望ましい状態に保たれるため、予測可能な運用が可能になります。
- **そのため、Kubernetesは、原則宣言型で利用します。**
  - テストなどにおいては、命令型を利用してもまったく問題ありません。
  - システムの状態に影響しない、参照系のコマンド（get, describeなど）は、命令型を使用しても構いません。
  - 一部、宣言型で対応が難しい操作もあります。例えば多数のリソースがある中で、一部のリソースのみ残して、ほかはすべて削除するなど。この場合、もちろん命令型のコマンドを使用しても問題ありません。

デプロイメント（Deployment）の利用

- 次は、デプロイメント（Deployment）というリソースについて学習していきます。
- Kubernetesでは、実際に直接Podを作成したり使用したりするケースはほとんどありません。Deploymentを経由してPodを利用することが一般的です。

システムのスケールアウトとポッド（Pod）

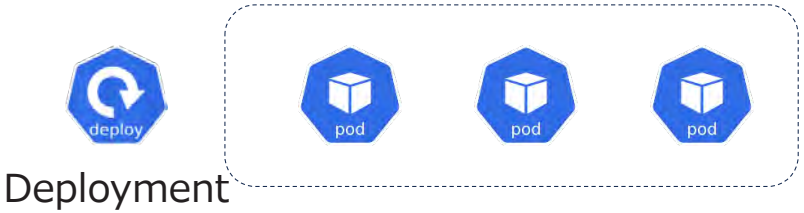
- **例えばECショップのWebアプリケーションがあるとします。ユーザー数が急増し、システムのリソースが足りなくなった場合、システムのスケールアウトが必要になります。では、具体的にどのようにスケールアウトしますか？**
  - 物理サーバー：物理サーバーを追加する（購入して、設置、インストール、設定…）
  - 仮想マシン：仮想マシンを追加する（数分間～）
  - コンテナ：コンテナを追加する（数秒間～）
- **では、Kubernetesクラスタでは、どのようにコンテナを追加しますか？**
  - Podに複数のコンテナを含むことができるため、Pod内でコンテナを追加することでスケールアウトできると思われがちですが、それは誤りです。
  - 詳細は次の章で説明しますが、通常「1 Pod = 1 コンテナ」と考えて問題ありません。
- **そのため、システムのスケールアウトは、Pod単位の追加になります。**



Pod内にコンテナを追加することでスケールアウトすることはできません。

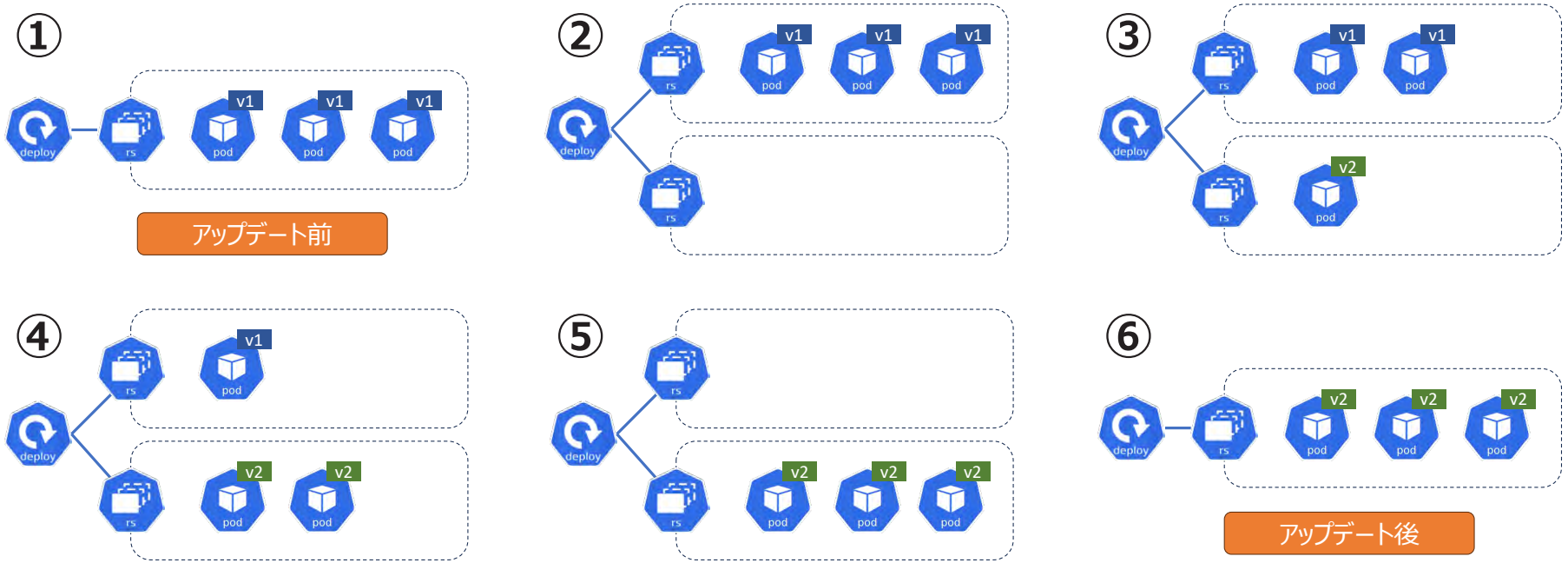
デプロイメント（Deployment）とは

- **デプロイメント（Deployment）は、複数の、同様なPodを管理するための仕組みです。**
- **Deploymentの具体的な機能は：**
  - **指定した数のPodを確実に維持します。**もしPodが予期せず停止したり削除された場合、Deploymentは自動的に新しいPodを起動して、望ましい状態を保ちます。
  - **ローリングアップデート&ロールバック。**コンテナのバージョンアップが必要な場合、Deploymentは段階的に新しいPodを展開します。これにより、ユーザーが常にサービスにアクセスできる状態が保たれます。また、問題が発生した場合、Deploymentは以前のバージョンにロールバックすることができます。
  - **スケーリング。**DeploymentはPodの数を増減させることができ、アプリケーションのトラフィック量に応じてスケーリングを簡単に行えます。



DeploymentとReplicaSet

- デプロイメント（Deployment）は、内部でレプリカセット（ReplicaSet）というものを利用します。
- 通常、1つのDeploymentには1つのReplicaSetが関連付けられていますが、ローリングアップデートが行われる際には、新しいReplicaSetが作成され、古いReplicaSetと新しいReplicaSetの2つが同時に存在することになります。



ローリングアップデート時のReplicaSetの状態遷移の一例です。設定によってプロセスが異なります。

演習：Deploymentの作成

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment
spec:
  replicas: 4
  selector:
    matchLabels:
      city: tokyo
  template:
    metadata:
      labels:
        city: tokyo
    spec:
      containers:
        - name: web
          image: docker.io/httpd:latest
          ports:
            - containerPort: 80
```

apiVersionは、Podと異なりますので注意してください。

kind（種別）はDeployment

レプリカの数、つまり維持したいPodの個数

「Deployment」と「Pod」を関連付けするには「selector」を利用します。ここのラベルと一致すればDeploymentの管理下に入ります。

Pod作成時のテンプレートです。Deploymentが動的にPodを作成するので、Podを直接定義するのではなく、テンプレートを通じてPodを定義します。

Deploymentのselectorのラベルと一致させる必要があります。

spec部分は、Podと同じ記述方法です。

「template」のインデントに十分注意してください

演習：Deploymentの作成

- 前頁のYAMLをデプロイします。

  ```
  kubectl apply -f deployment.yaml
  ```

- デプロイの結果を確認します。「READY」は立ち上がったPodの数で、最初の数秒間は0や1ですが、数十秒後すべて立ち上がることを確認できます。

  ```
  kubectl get -f deployment.yaml
  ```

```
C:¥pj>kubectl apply -f deployment.yaml
deployment.apps/mydeployment created

C:¥pj>kubectl get -f deployment.yaml
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
mydeployment   0/4     4            0           5s

C:¥pj>kubectl get -f deployment.yaml
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
mydeployment   1/4     4            1           13s

C:¥pj>kubectl get -f deployment.yaml
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
mydeployment   3/4     4            3           16s

C:¥pj>kubectl get -f deployment.yaml
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
mydeployment   4/4     4            4           21s
```

演習：Deploymentの作成

- **READYが「4/4」になったら、Podの一覧を確認したうえ、１つのPodを削除してみます。**
  kubectl get pods -o wide
  kubectl delete pod <確認したPodの名前>

- **異なる名前で即座に新たにPodが起動することを確認します。また、IPアドレスが変更されていることも確認できます。**
  kubectl get pods -o wide

```
C:¥pj>kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE   IP            NODE       NOMINATED NODE   READINESS GATES
mydeployment-5c6bdd9bcb-9pcmc   1/1     Running   0          13s   10.244.0.20   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-lnnx6   1/1     Running   0          13s   10.244.0.19   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-vgx5p   1/1     Running   0          13s   10.244.0.21   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-vj55m   1/1     Running   0          13s   10.244.0.22   minikube   <none>           <none>

C:¥pj>kubectl delete pod mydeployment-5c6bdd9bcb-vgx5p
pod "mydeployment-5c6bdd9bcb-vgx5p" deleted

C:¥pj>kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE   IP            NODE       NOMINATED NODE   READINESS GATES
mydeployment-5c6bdd9bcb-9pcmc   1/1     Running   0          77s   10.244.0.20   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-lnnx6   1/1     Running   0          77s   10.244.0.19   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-vj55m   1/1     Running   0          77s   10.244.0.22   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-xnk6k   1/1     Running   0          4s    10.244.0.23   minikube   <none>           <none>
```

演習：Deploymentのスケーリング

- YAMLファイルを編集して、「replicas: 4」を「replicas: 6」に変更します。
- 再度applyコマンドでデプロイします。Podが追加されたことを確認できます。

```
C:¥pj> kubectl apply -f deployment.yaml
deployment.apps/mydeployment configured

C:¥pj>kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP            NODE       NOMINATED NODE   READINESS GATES
mydeployment-5c6bdd9bcb-9pcmc   1/1     Running   0          2m37s   10.244.0.20   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-lnnx6   1/1     Running   0          2m37s   10.244.0.19   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-vj55m   1/1     Running   0          2m37s   10.244.0.22   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-xnk6k   1/1     Running   0          2m4s    10.244.0.23   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-px8vw   1/1     Running   0          8s      10.244.0.24   minikube   <none>           <none>
mydeployment-5c6bdd9bcb-xbghq   1/1     Running   0          8s      10.244.0.25   minikube   <none>           <none>
```

- Deploymentをスケーリングする（拡張・縮小）には、上記のようにマニフェストを編集して再度applyすることが推奨です。これは**宣言的**な方法です。
- 一方、非推奨ではありますが、**命令的**なコマンドもあります。例えば以下のコマンドは、Pod数を3にスケールインします：
  `kubectl scale deployment mydeployment --replicas=3`
- 最後は、Deploymentを削除します。
  `kubectl delete -f deployment.yaml`

演習：ローリングアップデート

- 次にローリングアップデートの演習を実施します。
- この演習では、右側のマニフェストを使用します。
  - レプリカ数は3です。理想状態では3つのPodが稼働します。
  - コンテナイメージは、nginxのやや古いバージョン、1.19です。この演習では、この近店を新しいバージョンのnginxのコンテナへアップデートします。
- では、さっそくマニフェストを適用し、ステータスを確認します。

```
kubectl apply -f nginx-deployment.yaml
kubectl get pods
```

```
C:¥pj>kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created

C:¥pj>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6b69486c79-h97hx   1/1     Running   0          22s
nginx-deployment-6b69486c79-jxz2t   1/1     Running   0          22s
nginx-deployment-6b69486c79-sj4wl   1/1     Running   0          22s
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.19
          ports:
            - containerPort: 80
```

演習：ローリングアップデート

- マニフェストを修正し、「nginx:1.19」を「nginx:1.21」に変更して保存します。

```
spec:
  containers:
    - name: nginx
      image: nginx:1.21
```

- マニフェストを再度適用します。
  `kubectl apply -f nginx-deployment.yaml`
- Deployment/ReplicaSet/Podの状態を確認します。
  `kubectl get deployment`
  `kubectl get replicaset`
  `kubectl get pods`

```
C:¥pj>kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     1            3           2m59s

C:¥pj>kubectl get replicaset
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-6b69486c79   3         3         3       3m1s
nginx-deployment-7cfcf9b64b   1         1         0       5s

C:¥pj>kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
nginx-deployment-6b69486c79-h97hx   1/1     Running             0          3m3s
nginx-deployment-6b69486c79-jxz2t   1/1     Running             0          3m3s
nginx-deployment-6b69486c79-sj4wl   1/1     Running             0          3m3s
nginx-deployment-7cfcf9b64b-f5425   0/1     ContainerCreating   0          7s
```

レプリカ数が3なのに、同時に4つのPodが存在することを確認できます。

演習：ローリングアップデート

- ローリングアップデート完了後、再度ReplicaSetを確認します。
  アップデート完了後も、ReplicaSetは削除されません。これは、Kubernetesがロールバックをサポートするために、古いReplicaSetを保持するからです。

```
C:¥pj>kubectl get replicaset
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-6b69486c79   0         0         0       3m25s
nginx-deployment-7cfcf9b64b   3         3         3       29s
```

nginx:1.19

nginx:1.21

- 試しに、マニフェストのnginxバージョンをさらに1個あげて、1.22に変更します。再度適用したうえ、ReplicaSetを確認します。

```
C:¥pj>kubectl get replicaset
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-6b69486c79   0         0         0       18m
nginx-deployment-7cfcf9b64b   0         0         0       15m
nginx-deployment-846985c665   3         3         3       12m
```

nginx:1.19

nginx:1.21

nginx:1.22

```
spec:
  containers:
    - name: nginx
      image: nginx:1.22
```

- このように、古いReplicaSetは削除されることなく、保持されます。
- ReplicaSetのリビジョン数は、デフォルトは10です。つまりデフォルトでは10世代まで保持されます。この値は、Deploymentの「revisionHistoryLimit」パラメータで変更できます。

演習：ロールバック

- **ロールバックは、2つの方法があります。**
  - 宣言型：マニフェストを修正し（例えばnginxを1.22から1.21に変更）、適用する
  - 命令型：kubectl rollout undoコマンド
- **通常は宣言型が推奨ですが、ロールバックの際は命令型の方法が推奨です。**
  - ロールバックは、アップデートする際に何かの問題で一時的に古いバージョンに戻す「縮退」です。
  - そのため、通常「理想的な状態」はやはりアップデート後のバージョンです。
  - 「理想的な状態」をマニフェストで保持しながら、一時的に古いバージョンに戻すために、命令型のほうが適切です。
- **実際にロールバックを実施してみます。1個前のリビジョンに戻っていることを確認します。**

```
kubectl rollout undo deployment nginx-deployment
```

```
C:¥pj>kubectl rollout undo deployment nginx-deployment
deployment.apps/nginx-deployment rolled back

C:¥pj>kubectl get replicaset
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-6b69486c79   0         0         0       30m
nginx-deployment-7cfcf9b64b   3         3         3       27m
nginx-deployment-846985c665   0         0         0       24m
```

nginx:1.19
nginx:1.21
nginx:1.22

再度「kubectl rollout undo」を実行しても、1.19に戻ることはありません。
1.22に戻ります。
指定したリビジョンに戻るには、
「--to-revision」オプションを利用します。

- **最後は、Deploymentを削除します。**

```
kubectl delete -f nginx-deployment.yaml
```

確認テスト1

Q1: Kubernetesでの環境構築（デプロイ）についての記述のうち、正しいのはどれか？正しいものをすべて選択してください。

1. 命令型は、実行すべきコマンドや順序を明確に記述する
2. 宣言型は、実行するコマンドなどをファイルとしてまとめて自動化を図る
3. マニフェストは、通常宣言型のアプローチと考えられる
4. 宣言型は冪等性を保証する
5. 宣言型のほうが自動化の度合いが高いので、すべての操作を宣言型で行うべき

Q2: Deploymentについての記述のうち、正しいのはどれか？正しいものをすべて選択してください。

1. Deploymentは、様々な種類のPodをまとめて管理するための仕組みである
2. 通常は、Pod単体で稼働させるのではなく、Deployment経由でPodを使用する
3. Deploymentは、内部でReplicaSetを利用している
4. Deploymentでは、維持したいPod数を指定する際に、必ず２以上指定する必要がある

確認テスト2

Q3: Deployment内にあるPodを手動で削除した場合の動作について、正しいのはどれか？最も適切なものを選択してください。

1. Deploymentは、実際のPod数に合わせて、レプリカ数を変更する
2. Deploymentは、必ず削除されたPodと同じIPアドレスでPodを再作成する
3. Deploymentは、Podを再作成するが、IPアドレスは必ずしも削除されたPodと同一ではない
4. Deploymentは、削除されたPodを復元して、起動する

Q4: Deploymentでは、Podを増やすことはできるが、減らすことはできない。この記述は正しいか？最も適切なものを選択してください。

1. 正しい
2. 正しくない

確認テスト3

Q5: Deploymentを使用したローリングアップデートについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. ローリングアップデートの際に、Podが複数のReplicaSetにて動作することがある
2. ローリングアップデートは、ソフトウェアの古いバージョンから新しいバージョンへの更新にしか利用できない
3. ローリングアップデートでは、それぞれのPod内のコンテナの更新なので、Podが再作成されることはない
4. ローリングアップデートは、古いPodを少しずつ終了し、新しいPodを順次起動する

Q4: ロールバックについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. ロールバックを使用するには、Deploymentなどが必要で、Pod単体では利用不可
2. ロールバックは、Kubernetesが古いPodを削除せず保存しているため、過去のバージョンに戻せるわけだ
3. ロールバックは、通常マニフェストを通じて行うべきだ
4. ロールバックは、１個前のバージョンにしか戻れない
5. ロールバックするには、kubectl rollout undoコマンドを使用する

第四章 Kubernetesのネットワーク通信

第四章 Kubernetesのネットワーク通信

この章の内容

- **この章では、Kubernetesクラスタにおけるコンテナ間の通信、およびコンテナと外部との通信について学びます。Kubernetesでは、主に「サービス（Service）」という仕組みを利用して、さまざまな通信を実現しています。**
- **この章の主な内容は以下の通りです。**
  - Pod内のネットワーク構造
  - サービス（Service）の利用方法
    - ClusterIP
    - NodePort
    - LoadBalancer
    - ExternalName

演習：Pod内のネットワーク構造

- Dockerでは、2つのコンテナが同時に80番ポートを利用しても特に問題ありませんでした。コンテナは仮想的に独立したマシンであるため、この挙動は当然です。
- では、Kubernetesではどうでしょうか？試しに、同一Podに2つのコンテナを立ち上げてみましょう。2つのコンテナはどちらもWebサーバーで、80番ポートを利用します。それぞれ、apacheとnginxです。
- kubectl applyでデプロイすると、問題なくPodは作成されたようです。しかし、しばらく経ってから確認すると、エラーが発生し、CrashLoopBackOffという再起動のループに陥っている状態であることがわかります。

two-webserver.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: two-webserver
spec:
  containers:
  - name: apache
    image: docker.io/httpd:latest
  - name: nginx
    image: docker.io/nginx:latest
```

```
C:¥pj>kubectl apply -f two-webserver.yaml
pod/two-webserver created

C:¥pj>kubectl get pods
NAME            READY   STATUS             RESTARTS      AGE
two-webserver   1/2     CrashLoopBackOff   6 (88s ago)   7m45s
```

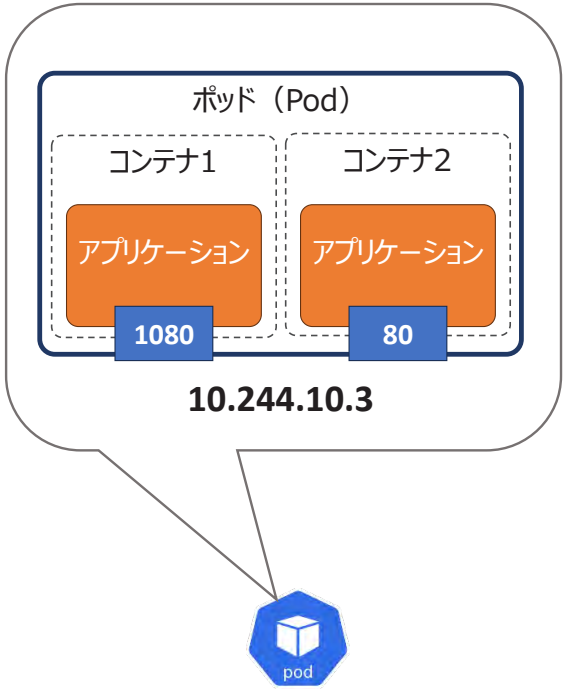
演習：Pod内のネットワーク構造

- そこで、それぞれのコンテナのログを確認します。以下のコマンドを使用します。
  kubectl logs <Pod名> -c <コンテナ名>
- ログを確認すると、apacheは起動に成功したものの、nginxは80番ポートがすでに使用されているため、bind()に失敗していることがわかります。

```
C:¥pj>kubectl logs two-webserver -c apache
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.244.0.9. Set the 'ServerName' directive globally to suppress
this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.244.0.9. Set the 'ServerName' directive globally to suppress
this message
[Thu Dec 26 04:24:22.361628 2024] [mpm_event:notice] [pid 1:tid 1] AH00489: Apache/2.4.62 (Unix) configured -- resuming normal operations
[Thu Dec 26 04:24:22.361822 2024] [core:notice] [pid 1:tid 1] AH00094: Command line: 'httpd -D FOREGROUND'

C:¥pj>kubectl logs two-webserver -c nginx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
(中略)
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/12/26 04:30:33 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address already in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)
2024/12/26 04:30:33 [emerg] 1#1: bind() to [::]:80 failed (98: Address already in use)
nginx: [emerg] bind() to [::]:80 failed (98: Address already in use)
2024/12/26 04:30:33 [notice] 1#1: try again to bind() after 500ms
2024/12/26 04:30:33 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address already in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)
...
```

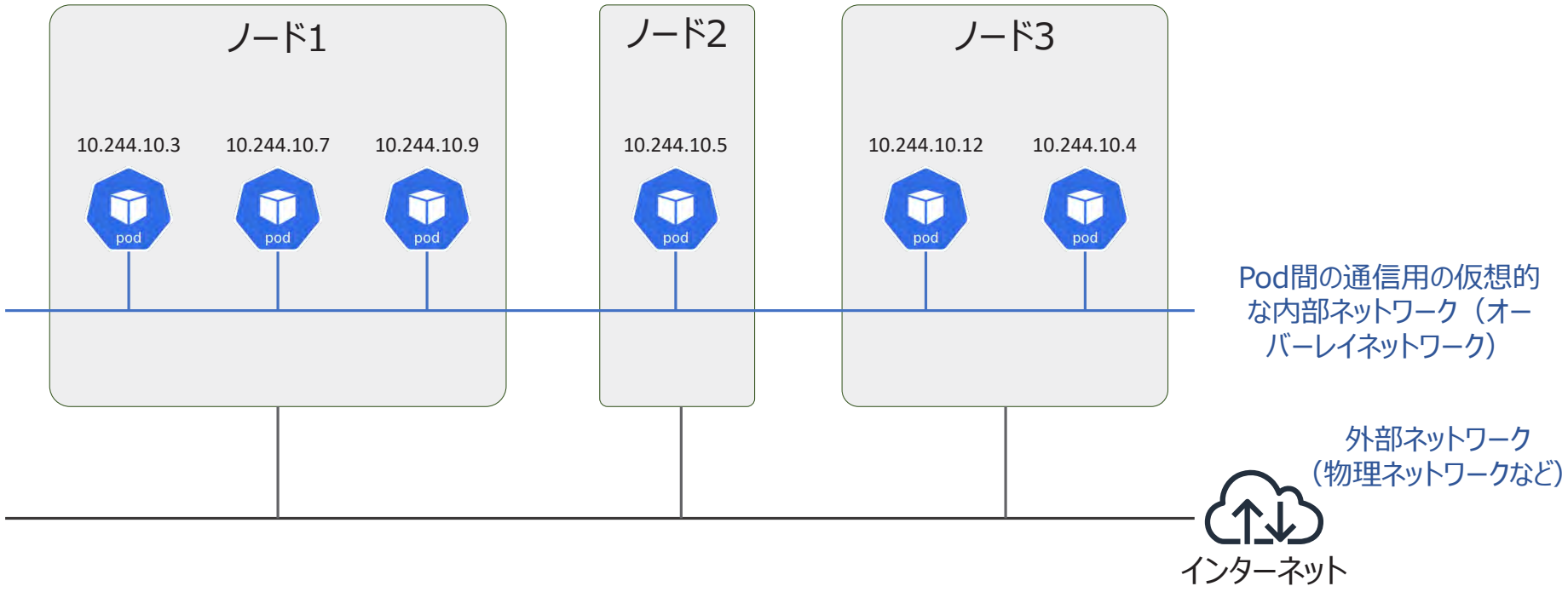
演習：Pod内のネットワーク構造

- **Pod内のコンテナは、同じネットワーク名前空間を共有します。**
  - そのため、ネットワークの観点から見ると、複数のコンテナは「1台のマシン」のように動作し、分離されていません。
  - すべてのコンテナは、同じIPアドレス（PodのIPアドレス）を共有します。
  - 例えば、コンテナ2にあるアプリケーションが、コンテナ1にあるアプリケーションにアクセスする場合、<localhost:ポート番号>の形式で通信が可能です。
- **同じPod内のコンテナは、CPUやメモリなどのリソースを共有することなく、独立して動作します。**
- **Podに含まれる複数のコンテナは、必ず同じノード上に配置されます。**
- **このような特性を持つPodですが、原則として、1Pod = 1コンテナで実装すべきです。ただし、特定のユースケースや監視・メトリクス収集など、密に連携しリソースを共有するケースでは、複数のコンテナを同じPodに配置することが一般的な実装方法となります。**



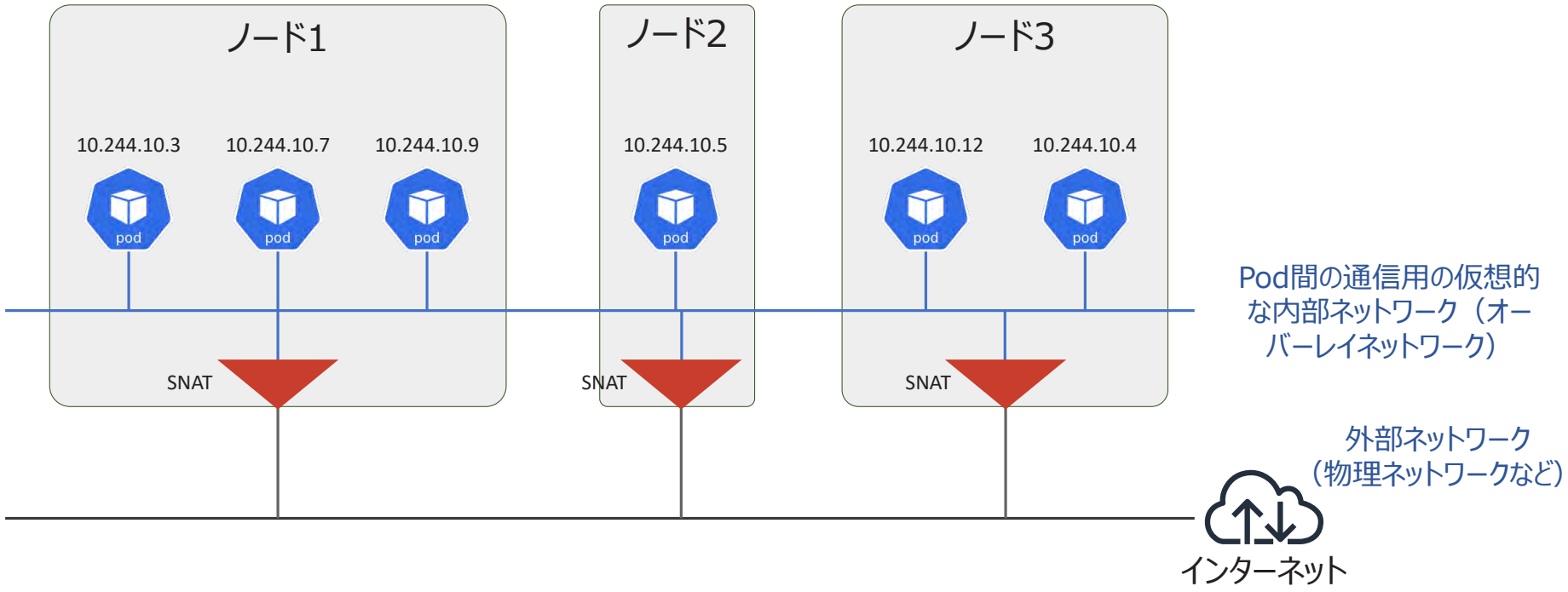
ポッド・ネットワーク

- Kubernetesでは、Podは仮想的な専用内部ネットワークを通じて通信します。いわゆる「オーバーレイネットワーク」です。オーバーレイ技術としてVxLANなどが利用されます。



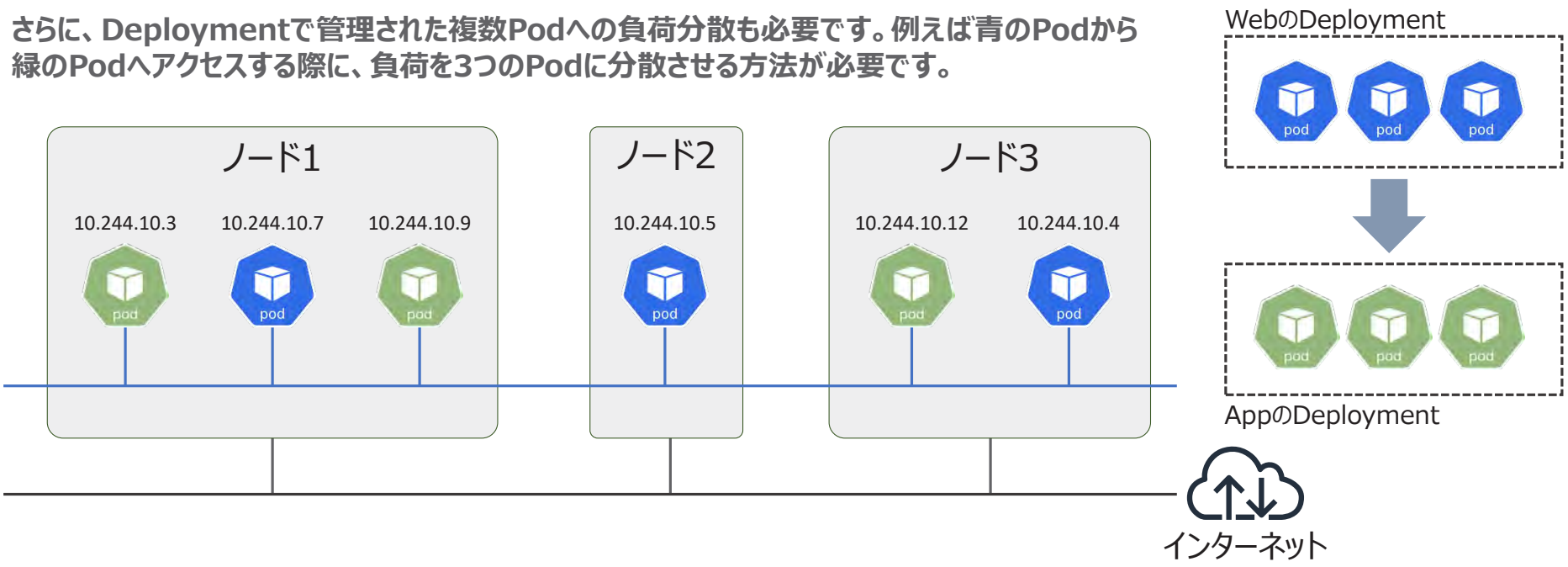
ポッド・ネットワーク

- Pod内にあるアプリケーションから外部へと通信するために、SNAT（ソースNAT）を利用します。通常、それぞれのノードでSNATが実装され、Podから外部への通信を可能にします。



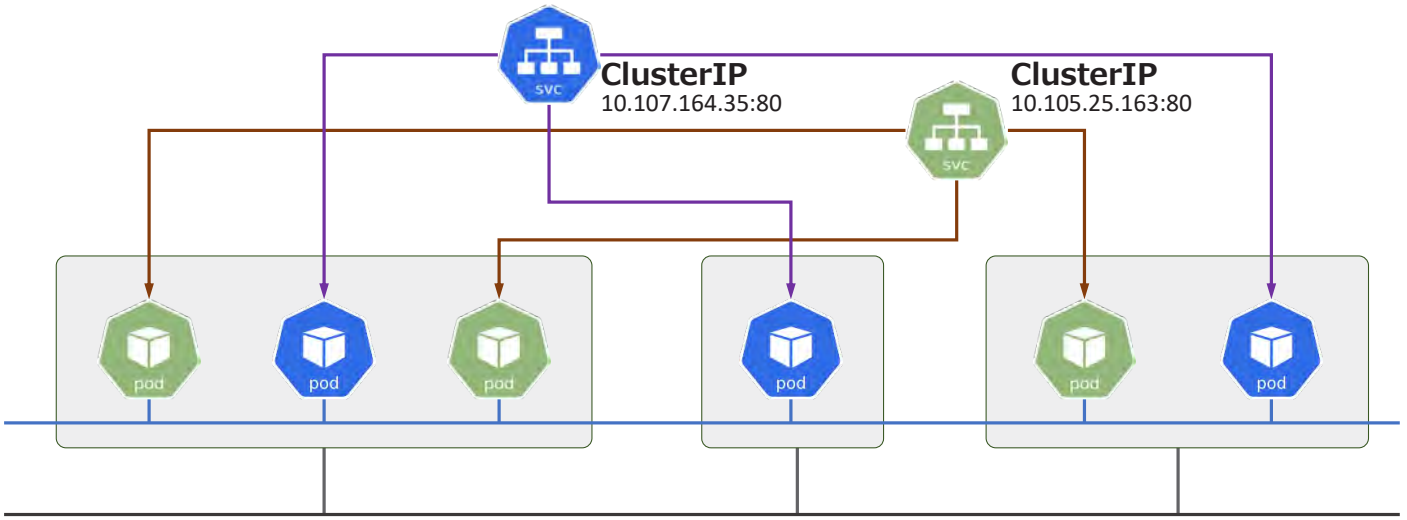
ポッド・ネットワークの制限

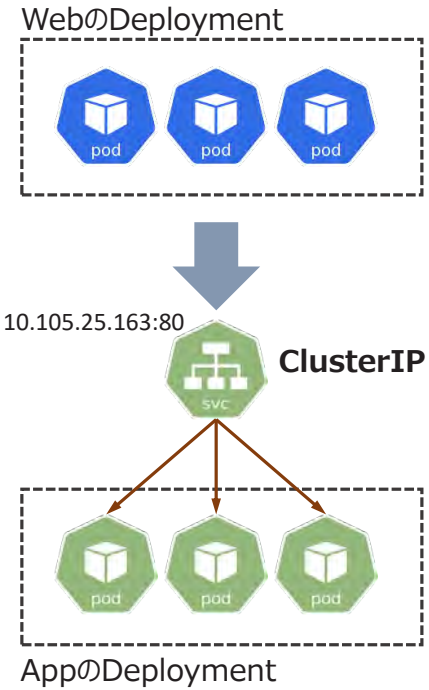
- 通常は、PodがDeploymentによって管理され、動的に作成・削除されます。そのため、IPアドレスも固定ではありません。
- そのため、Pod間のアクセスは、PodのIPアドレスを利用することはできなくなります（アクセス先のIPアドレスがいつまで有効なのかわからないためです）。
- さらに、Deploymentで管理された複数Podへの負荷分散も必要です。例えば青のPodから緑のPodへアクセスする際に、負荷を3つのPodに分散させる方法が必要です。



サービス・ネットワーク

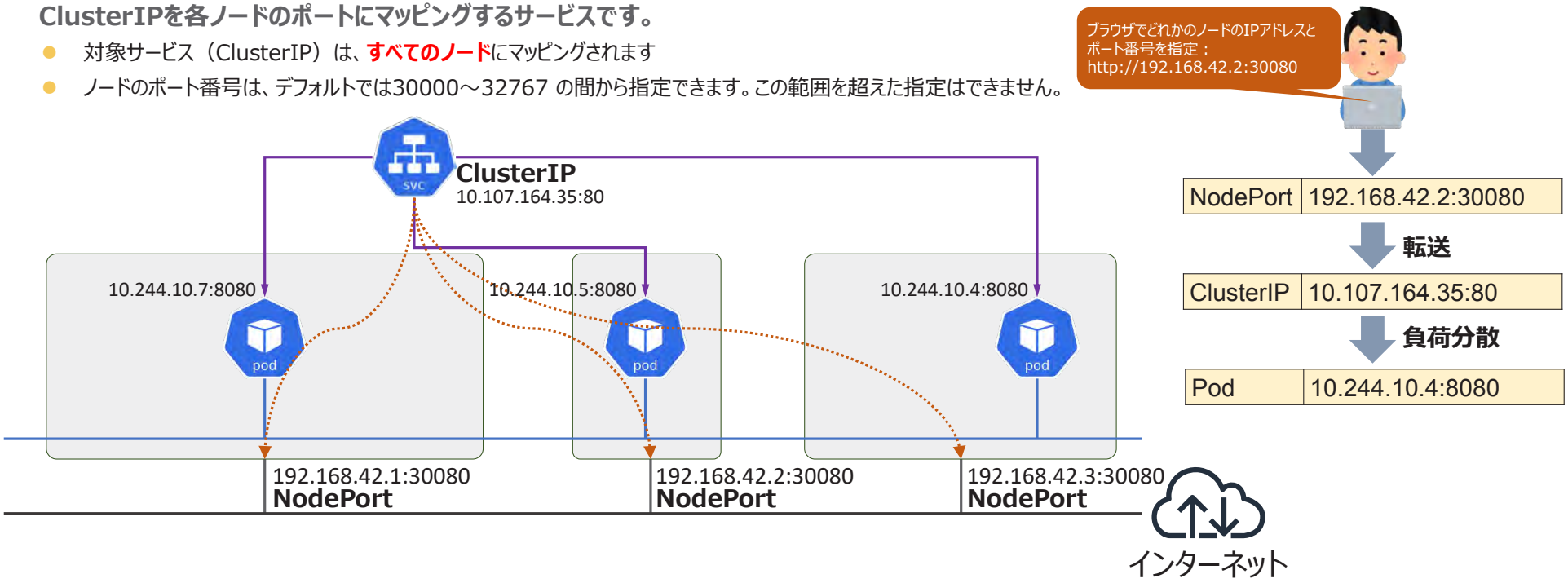
- これを解決してくれるのは、「サービス（Service）」という抽象的なリソースです。Kubernetesのサービスとは、クラスタ内の一連のPodに対して、**ネットワークアクセス**を提供するための抽象的な仕組みです。
- サービスには、いくつかの「タイプ（Type）」があります。基本的なタイプは、ClusterIPです。ClusterIPというサービスは、**固定の仮想IPアドレス**を提供し、リクエストを適切な**バックエンドのPodにルーティング**します。
- ClusterIPのIPアドレスは、PodのIPアドレスと異なるレンジを利用します（通常10.96.0.0/12を利用）。





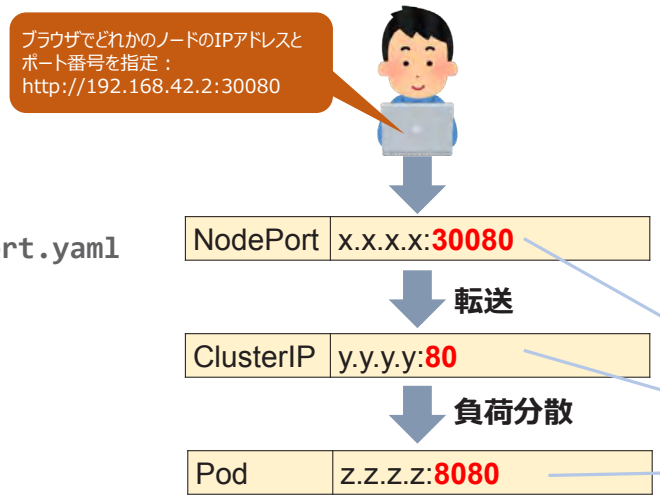
外部からのアクセス方法：NodePort

- ClusterIPは、IPアドレスが固定でないPodに対して、「**固定IP**」と「**L4の負荷分散**」機能を提供します。
- しかし、ClusterIPは、PodのIPアドレスと同様にあくまでも内部向けのIPアドレスです。外部からアクセスすることはできません。
- 外部からClusterIPにアクセスするには、NodePortを利用するのが1つの方法です。NodePortもClusterIPと同様にサービスの1つであり、ClusterIPを各ノードのポートにマッピングするサービスです。
  - 対象サービス（ClusterIP）は、**すべてのノード**にマッピングされます
  - ノードのポート番号は、デフォルトでは30000～32767 の間から指定できます。この範囲を超えた指定はできません。



演習：NodePortとClusterIPの作成

- では、実際にNodePort/ClusterIPを作成してみましょう。
- NodePortを作成すると、自動的にClusterIPが作成されるため、YAMLファイルではNodePortのみ記述すれば大丈夫です。
- 1つのYAMLファイルに複数のリソース（右側の例ではDeployment、NodePortの2つ）を定義する場合、「---」で区切ります。
- 右側のコードをデプロイします：
  `kubectl apply -f nodeport.yaml`



nodeport.yaml

```yaml
# Deploymentの定義
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: kicbase/echo-server:1.0
          ports:
            - containerPort: 8080
---
# NodePortの定義
apiVersion: v1
kind: Service
metadata:
  name: nodeport-web
spec:
  selector:
    app: web
  ports:
    - nodePort: 30080
      port: 80
      targetPort: 8080
  type: NodePort
```

演習：NodePortとClusterIPの作成

- YAMLをデプロイした後、デプロイしたリソースを確認します。
- 作成したサービスを確認するには、以下のコマンドを利用します。

  `kubectl get services`

  `kubectl describe service <サービス名>`

```
C:¥pj>kubectl get deployment
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
deployment-web   3/3     3            3           12s

C:¥pj>kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
deployment-web-69555c7f4c-lxlbc   1/1     Running   0          15s
deployment-web-69555c7f4c-p29gr   1/1     Running   0          15s
deployment-web-69555c7f4c-pjbvg   1/1     Running   0          15s

C:¥pj>kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP        91m
nodeport-web   NodePort    10.106.89.92   <none>        80:30080/TCP   57s

C:¥pj>kubectl describe service nodeport-web
Name:                     nodeport-web
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=web
Type:                     NodePort
<以下省略>
...
```

「kubernetes」というClusterIPは、クラスタ作成時に自動的に作成されたシステムのサービスです。
もう1つの「nodeport-web」は前頁のYAMLで作成したNodePortです。

演習：NodePortとClusterIPの作成

- これで、ノードの30080番ポートにアクセスすれば、echo-serverにアクセスできるようになります。
  （複数のノードで構成したKubernetesクラスタの場合、どのノードでも構いません）

- アクセス方法は、環境によって異なります。

minikube / Docker

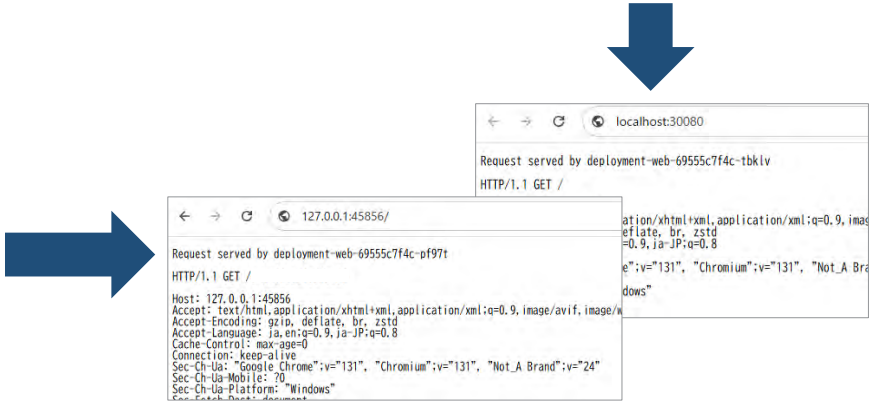
Kubernetes for Docker Desktop

minikubeの場合、サービスをローカルPCにマッピングする必要があります。以下のコマンドを利用します：

`minikube service <サービス名>`

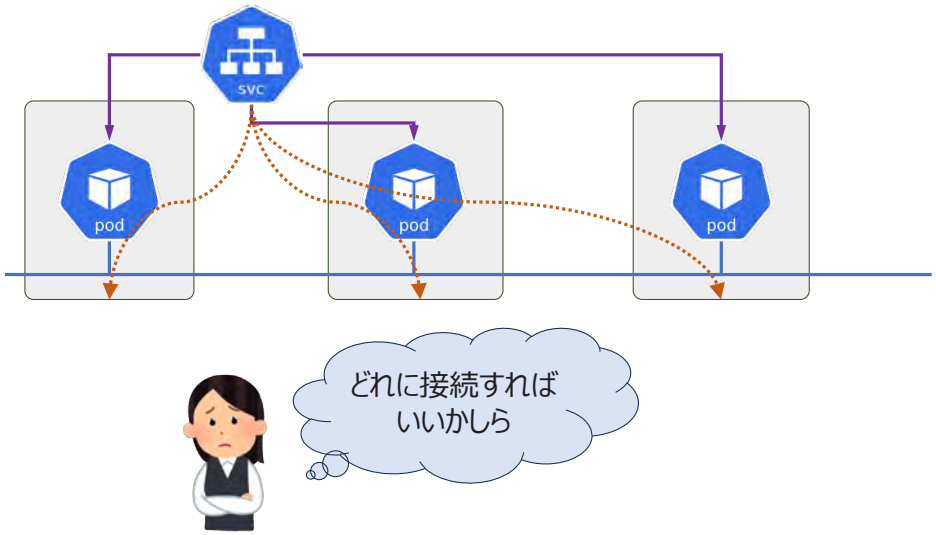
```
C:¥pj>minikube service nodeport-web
|-----------|--------------|-------------|-------------------------|
| NAMESPACE |     NAME     | TARGET PORT |           URL           |
|-----------|--------------|-------------|-------------------------|
| default   | nodeport-web |          80 | http://192.168.67.2:30080 |
|-----------|--------------|-------------|-------------------------|
* nodeport-web サービス用のトンネルを起動しています。
|-----------|--------------|-------------|-------------------------|
| NAMESPACE |     NAME     | TARGET PORT |           URL           |
|-----------|--------------|-------------|-------------------------|
| default   | nodeport-web |             | http://127.0.0.1:45856 |
|-----------|--------------|-------------|-------------------------|
```

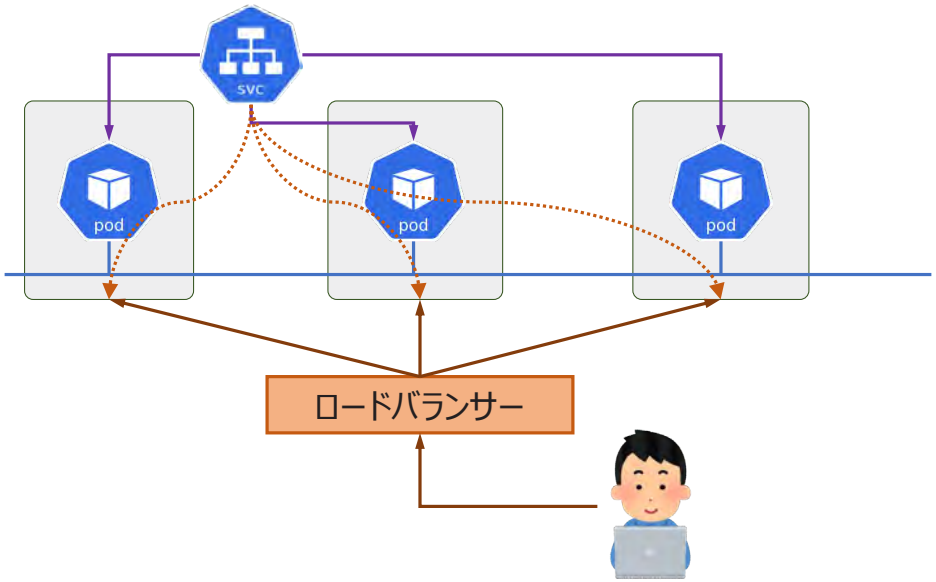
Kubernetes for Docker Desktopの場合、ノードがローカルPCになります。そのまま「localhost:30080」にアクセスできます。



演習：NodePortとClusterIPの作成

- このように、NodePortを利用することで、アプリケーションを外部に公開することが可能です。
- しかし、NodePortを利用する場合、ノードのIPアドレスやポートをそのままユーザーに公開することは避けるべきです。セキュリティリスク、可用性の低下、負荷分散の問題が発生する可能性があるためです。
- NodePortの一般的な利用方法としては、前段にロードバランサーを設置し、NodePortを間接的に使用する形が推奨されます。





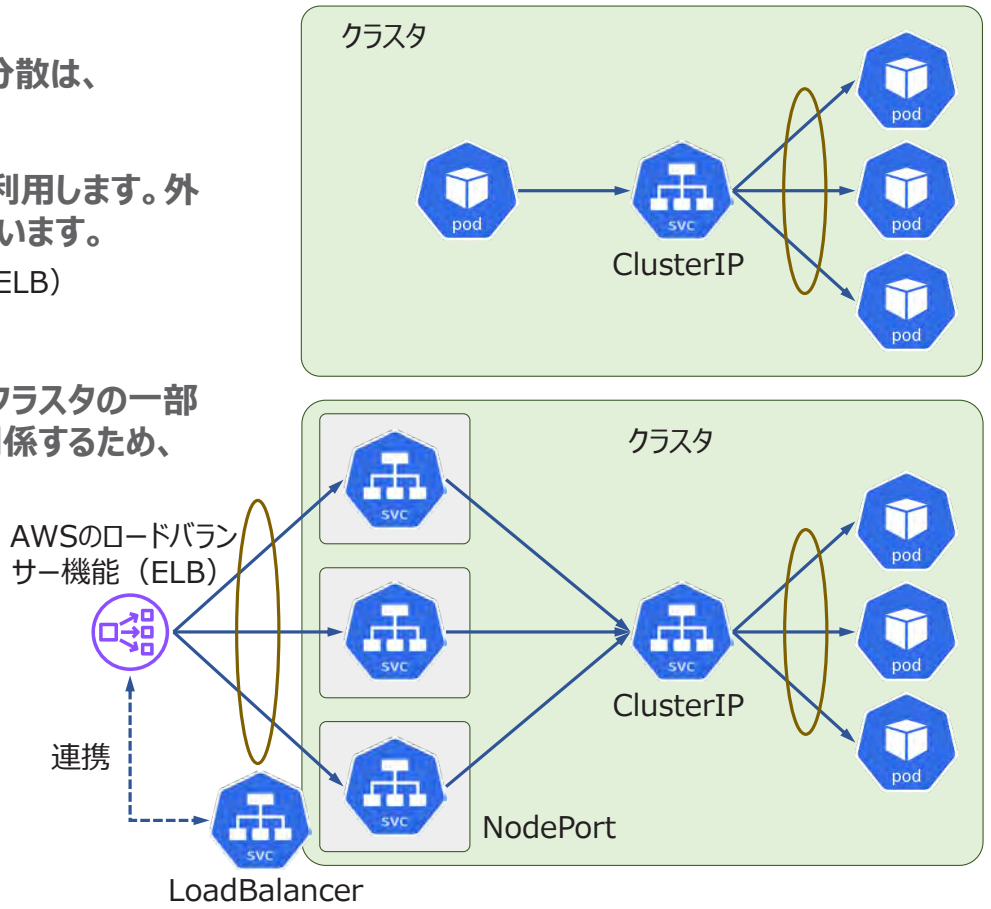
- こちらの演習は以上です。作成したリソースはすべて削除してください。

ロードバランサー

- **Kubernetesクラスタ内部での複数Pod対する負荷分散は、ClusterIPが用いられます。**
- **アプリケーションを外部に公開する際に、NodePortを利用します。外部のロードバランサーは、複数ノード間の負荷分散を行います。**
  - クラウド環境では、例えばAWSのElastic Load Balancer（ELB）
  - オンプレミス環境では、例えばHAProxyやF5など
- **この「外部のロードバランサー」は、当然Kubernetesクラスタの一部ではありません。しかし、Kubernetesクラスタと密に関係するため、Kubernetes内から制御する手段が必要です。**

KubernetesのサービスリソースにはClusterIPやNodePortのほかに、「**LoadBalancer**」というタイプがあります。

LoadBalancerサービスは、クラウド上の実際のロードバランサーなどと連携して、複数ノード間の負荷分散機能を提供します。



演習：LoadBalancerサービスと疑似ロードバランサー

- LoadBalancerサービスは、原則AWSやAzureなどのクラウドでのみ動作するように設計されているため、ローカルPCで構築したテスト環境で本格的なロードバランサーを構築することが難しいです。
- そのため、本演習では、疑似ロードバランサー(*)を利用します。minikubeでは以下の事前準備が必要ですが、Kubernetes for Docker Desktopは事前準備不要です。

minikubeの事前準備

```
C:¥pj>minikube addons enable metallb
! metallb is a 3rd party addon and is not maintained or verified by
minikube maintainers, enable at your own risk.
! metallb does not currently have an associated maintainer.
  - quay.io/metallb/speaker:v0.9.6 イメージを使用しています
  - quay.io/metallb/controller:v0.9.6 イメージを使用しています
* 'metallb' アドオンが有効です
```

metallbというアドオンを追加します。ベアメタル（オンプレミス環境）で使えるロードバランサーです。

```
C:¥pj>minikube node list
minikube        192.168.49.2
minikube-m02    192.168.49.3
minikube-m03    192.168.49.4
```

左記のコマンドで、ノードのIPアドレスを確認します。デフォルトの単一ノードのクラスタの場合、1行のみ表示されるはずです。

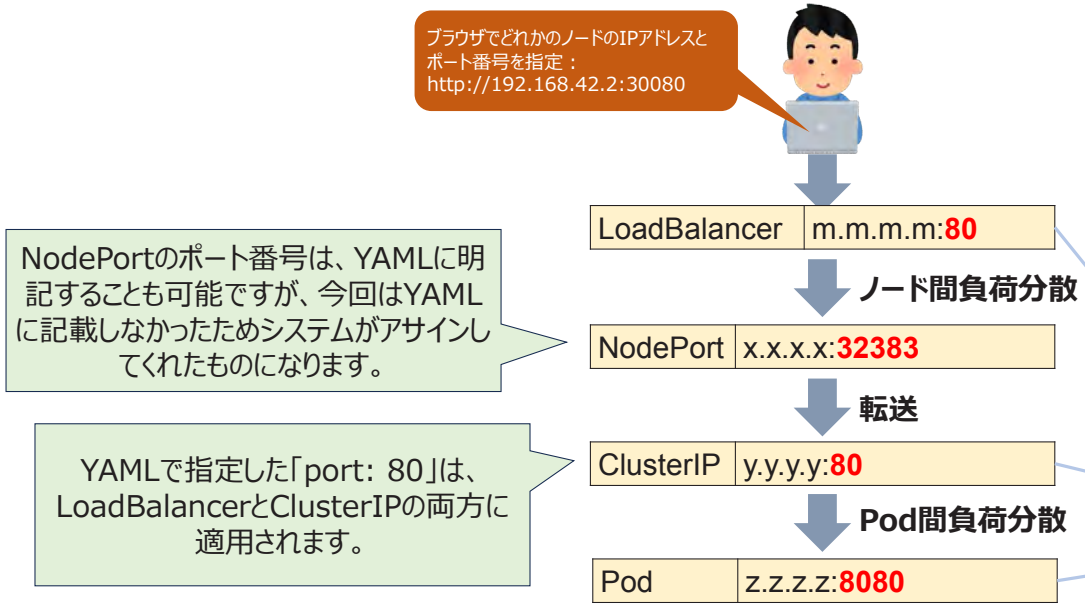
```
C:¥pj>minikube addons configure metallb
-- Enter Load Balancer Start IP: 192.168.49.51
-- Enter Load Balancer End IP: 192.168.49.60
  - quay.io/metallb/speaker:v0.9.6 イメージを使用しています
  - quay.io/metallb/controller:v0.9.6 イメージを使用しています
* metallb は正常に設定されました
```

左記のコマンドで、metallbが利用するIPアドレス範囲を指定します。ノードのIPアドレスと同じサブネットの範囲を指定してください。

*厳密に言うと、MetalLBは疑似ロードバランサーではなく、オンプレミス環境向けの実用的なロードバランサーです。ただし、本講座ではクラウド上のKubernetesによる本番環境を想定しているため、クラウドのロードバランサーをシミュレーションする意味合いで「疑似」と表現しています。

演習：LoadBalancerサービスと疑似ロードバランサー

- 準備が整ったので、マニフェストでリソースを定義します。
- Deployment部分は、前の演習（NodePort）と同じです。
- LoadBalancerサービスの部分について、内部的にはNodePort/ClusterIPを利用していますが、NodePort/ClusterIPを明示的に記載する必要はありません。




```yaml
# Deploymentの定義
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: kicbase/echo-server:1.0
          ports:
            - containerPort: 8080
---
# LoadBalancerの定義
apiVersion: v1
kind: Service
metadata:
  name: dummy-loadbalancer
spec:
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

演習：LoadBalancerサービスと疑似ロードバランサー

- **それでは、マニフェストを適用します。**

```
C:¥pj>kubectl apply -f loadbalancer.yaml
deployment.apps/deployment-web created
service/dummy-loadbalancer created
```

- **適用後、サービスを確認します。minikubeとKubernetes for Docker Desktopの出力結果が異なります。**

●minikube

```
C:¥pj>kubectl get services
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)        AGE
dummy-loadbalancer  LoadBalancer   10.101.192.193   192.168.49.52   80:32383/TCP   4s
kubernetes          ClusterIP      10.96.0.1        <none>          443/TCP        2d18h
```

外部IP（EXTERNAL-IP）は、metallbに設定したIPアドレス範囲からアサインされています。

●Kubernetes for Docker Desktop

```
C:¥pj>kubectl get services
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
dummy-loadbalancer  LoadBalancer   10.101.179.51    localhost     80:32058/TCP   4s
kubernetes          ClusterIP      10.96.0.1        <none>        443/TCP        5d22h
```

外部IP（EXTERNAL-IP）は、「localhost」です。

演習：LoadBalancerサービスと疑似ロードバランサー

- 最後に、ロードバランサーの動作を確認します。
- Kubernetes for Docker Desktopは非常にシンプルで、ブラウザで「http://localhost」を開くだけで確認できます。
- minikubeの場合は、いくつかの方法がありますが、1つご紹介します。

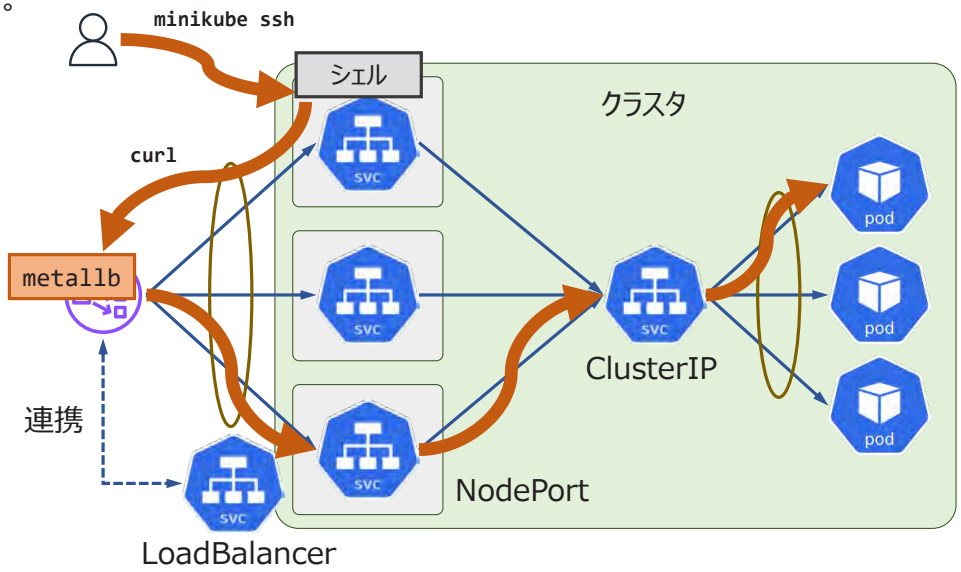
●minikube
まず、minikube sshコマンドを使用してノードのシェルにアクセスします。

```
C:¥pj>minikube ssh
docker@minikube:~$
```

次に、curlコマンドを使用して、metallbロードバランサーのIPアドレスに対してリクエストを送信します。（ロードバランサーのIPアドレスは、kubectl get servicesコマンドの出力にあるEXTERNAL-IP欄に記載されています）

```
docker@minikube:~$ curl http://192.168.49.52
Request served by deployment-web-69555c7f4c-5vgb5

HTTP/1.1 GET /

Host: 192.168.49.52
Accept: */*
User-Agent: curl/7.81.0
```

確認後、exitコマンドでシェルを終了します。

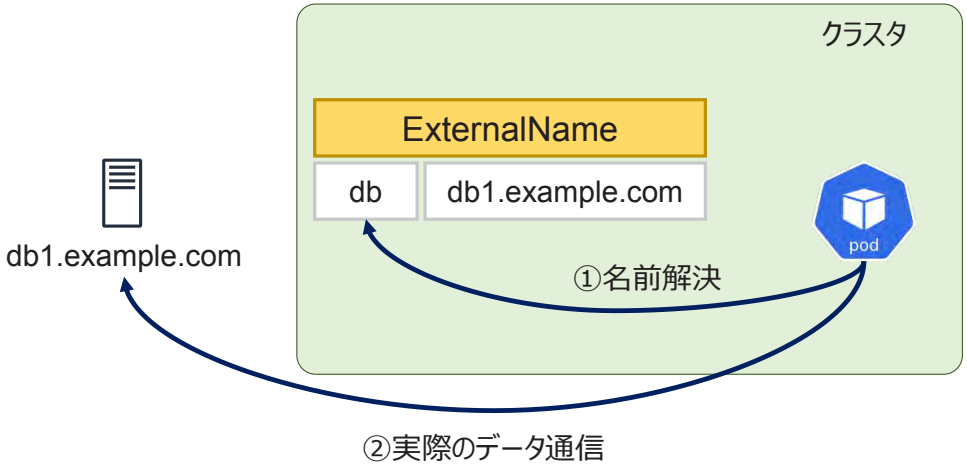


ExternalNameサービス

- もう1つのサービスタイプは、ExternalNameです。
- これは非常にシンプルな機能で、負荷分散や転送は行いません。単に、クラスタ外の名前（例：db1.example.comのようなFQDN）をクラスタ内の名前にマッピングするだけの役割を持ちます。
- 外部リソースを抽象化することで、Podから見ると外部の名前と内部の名前を区別する必要がなくなります。
- また、変更は即時に反映されるため、Podの再起動などの操作は不要です。

```
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  type: ExternalName
  externalName: db1.example.com
```

マニフェストもとてもシンプルです。



確認テスト1

Q1: 同じPod内の複数のコンテナについての記述のうち、正しいのはどれか？最も適切なものを選択してください。

1. 通常、異なるIPアドレスを持っているが、同じポートを利用できない
2. 通常、同じIPアドレスを持っており、同じポートを利用できない
3. 通常、異なるIPアドレスを持っており、同じポートを利用しても問題ない
4. 通常、同じIPアドレスを持っているが、同じポートを利用しても問題ない

Q2: 通常、Pod内のアプリケーションは、インターネット上のファイルなど外部のファイルをダウンロードする際に、どのように外部ネットワークにアクセスしているか？最も適切なものを選択してください。

1. アプリケーションは、SNATを利用してノードが所在するネットワークに到達し、外部にアクセスする
2. アプリケーションは、ノードが所在するネットワークにルーティングされ、外部にアクセスする
3. アプリケーションは、DNATを利用してノードが所在するネットワークに到達し、外部にアクセスする
4. アプリケーションは、kube-proxyというプロキシサーバーを使用して、外部にアクセスする

確認テスト2

Q3: ClusterIPについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. ClusterIPは、Deploymentとは１対１の関係にある
2. ClusterIPは、トラフィックを関連付けしたPodに負荷分散して転送する
3. ClusterIPのIPアドレスは、PodのネットワークのIPレンジから割り当てる
4. ClusterIPのIPアドレスは、直接外部からアクセスすることはできない
5. ClusterIPのIPアドレスは、固定IPである

Q4: NodePortについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. NodePortを使用すると、外部からPod内のアプリケーションにアクセスできる
2. NodePortで利用可能なポート範囲は、30000番以上のすべてのポートである
3. NodePortは、必要なノードにのみ設定できる
4. NodePortを利用する際に、NodePortのポート番号などを直接ユーザーに公開して、アクセスしてもらう
5. NodePort内部ではClusterIPを利用しているが、NodePortを作成する際に明示的にClusterIPを作成する必要はない

確認テスト3

Q5: LoadBalancerについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. LoadBalancerは、Kubernetesクラスタ内部でのL7負荷分散の機能である
2. LoadBalancerを作成すると、必ずNodePortおよびClusterIPも作成される
3. LoadBalancerは、通常クラウド上の負荷分散機能と連動する
4. LoadBalancerは、クラスタ内でPod間の通信を最適化するために使用される

Q6: ExternalNameは、Kubernetesクラスタ内のアプリケーションに名前をつけて、外部からより便利にアクセスしてもらうための仕組みである。この記述は正しいか？最も適切なものを選択してください。

1. 正しい
2. 正しくない

第五章 Kubernetesの
ストレージとデータ管理

第五章 Kubernetesのストレージとデータ管理

この章の内容

- **この章では、データの管理について学習します。**
- **物理サーバーと異なり、Podはいつ削除されても不思議ではありません。そのため、永続化が必要なデータはどのように扱うべきかは課題です。Kubernetesは、データの性質に応じて様々な管理方法を提供しています。**
- **この章の主な内容は以下の通りです。**
  - Kubernetesのストレージボリューム
    - 一時的なボリューム（emptyDir）
    - 永続的なボリューム（PV）
  - 設定情報の管理（ConfigMapとSecret）

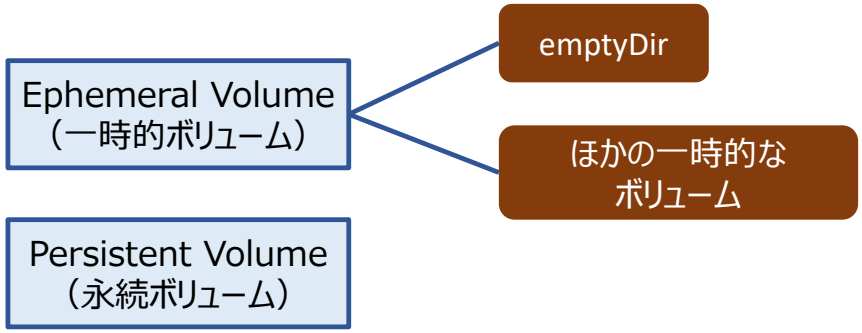
Kubernetesのストレージボリューム

- コンテナと物理サーバーの最も重要な違いは、コンテナが短命（エフェメラル）であることです。Deploymentの講義でご紹介した通り、コンテナは動的に作成・削除されるため、いつ削除されても不思議ではありません。そのため、**データの永続化**の仕組みが必要です。
- また、同じPod内のコンテナであっても、互いのファイルに直接アクセスすることはできません。そのため、複数のコンテナ間で**データ共有**の仕組みが必要です。
- これらの課題を解決する方法はいくつかあります。たとえば、データベースやAWS S3などのオブジェクトストレージといった、Kubernetesクラスタ外のサービスを利用するケースがよく見られます。
- 一方、Kubernetes内のソリューションとして、Kubernetesは2種類の「ボリューム」を提供しています。「ボリューム」はコンテナとは独立した存在であり、通常はコンテナに紐づけて利用します。

Ephemeral Volume（一時的ボリューム）	Ephemeral（エフェメラル）とは「儚い、短命な」という意味で、通常はコンテナ/Podが消滅するとボリューム上のデータも失われる仕様です。
Persistent Volume（永続ボリューム）	Persistent（パーシステント）とは「持続的な」という意味で、Podのライフサイクルとは独立して存在するストレージリソースです。

一時的なボリューム：emptyDir

- emptyDirはKubernetesのEphemeral Volume（一時的ボリューム）の一種です。その名の通り、「空のディレクター」です。
- Pod作成時に自動的に作成され、Podが削除されるとその内容も消去されます。
- では、演習を通じてemptyDirの特徴を確認してみます。



演習：emptyDirの利用

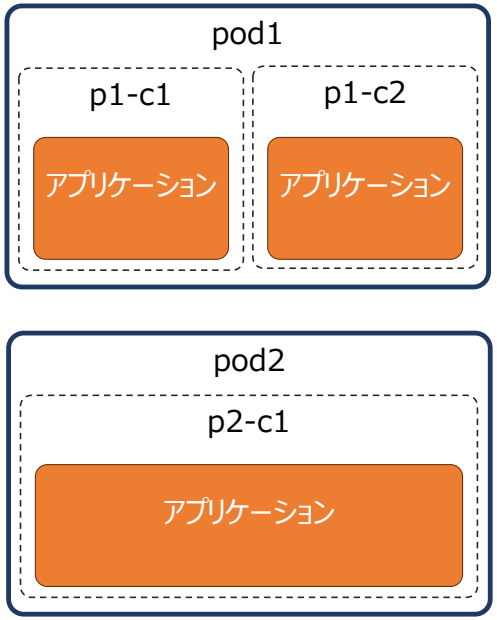
- この演習では、2つのPodを構築します。
- マニフェストが長いため、この資料ではスライド2枚に分割して表示します。
- このスライドのコードと次のスライドにあるコードを「---」で繋いで1つのYAMLファイルとして作成してもいいですし、それぞれ別のYAMLファイルとして作成しても問題ありません。
- 1つのファイルとして作成した場合、今まで通りのコマンドで適用します：
  `kubectl apply -f file.yaml`
- 2回のファイルとして作成した場合、以下ようにファイルをカンマで区切ることで複数のYAMLファイルを適用することが可能です：
  `kubectl apply -f file1.yaml,file2.yaml`

マニフェストYAMLファイル（1/2）

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  containers:
    - name: p1-c1
      image: docker.io/busybox:latest
      stdin: true
      tty: true
      volumeMounts:
      - name: shared-storage
        mountPath: /data
    - name: p1-c2
      image: docker.io/busybox:latest
      stdin: true
      tty: true
      volumeMounts:
      - name: shared-storage
        mountPath: /data
  volumes:
  - name: shared-storage
    emptyDir: {}
```

演習：emptyDirの利用

- 前頁のYAMLの続きです。
- このマニフェストは、2つのPodを定義しており、それぞれの名前はpod1とpod2です。
- pod1にはさらに2つのコンテナがあり、それぞれp1-c1、p1-c2です。pod2には1つのコンテナのみ（p2-c1）含まれています。



マニフェストYAMLファイル（2/2）

```
apiVersion: v1
kind: Pod
metadata:
  name: pod2
spec:
  containers:
    - name: p2-c1
      image: docker.io/busybox:latest
      stdin: true
      tty: true
      volumeMounts:
      - name: shared-storage
        mountPath: /data
  volumes:
  - name: shared-storage
    emptyDir: {}
```

演習：emptyDirの利用

YAMLファイルの内容を解説します。

stdin/ttyの設定は、docker runコマンドの「-it」オプションとほぼ同じ役割を果たします。この2つのオプションがない場合、コンテナは作成直後に終了してしまいます。

containerに設定されている「volumeMounts」は、YAMLの末尾で定義されたボリュームをどのようにマウントするかを指定する設定です。

「volumes」のインデントに注意してください。「containers」と同じレベルになります。つまり、コンテナとは独立した概念としてボリュームを定義します。このボリュームの種類は「emptyDir」です。

```
spec:
  containers:
    - name: p1-c1
      image: docker.io/busybox:latest
      stdin: true
      tty: true
      volumeMounts:
      - name: shared-storage
        mountPath: /data
    - name: p1-c2
      image: docker.io/busybox:latest
      stdin: true
      tty: true
      volumeMounts:
      - name: shared-storage
        mountPath: /data
  volumes:
  - name: shared-storage
    emptyDir: {}
```

演習：emptyDirの利用

- このマニフェストでは、2つのPodの3つのコンテナに、それぞれボリュームを「/data」にマウントしました。では、実際はどうでしょうか？
- 下記コマンドで、コンテナのシェルにアクセスします。
  `kubectl exec -it <Pod名> -c <コンテナ名> -- <実行したいコマンド及びオプション>`

```
C:¥pj>kubectl exec -it pod1 -c p1-c1 -- /bin/sh
/ # ls /data
/ # echo hello > /data/myfile
/ # ls /data
myfile
/ # exit
```

コンテナ「p1-c1」にログインします。
/dataの配下に、「myfile」を作成します。

```
C:¥pj>kubectl exec -it pod1 -c p1-c2 -- /bin/sh
/ # ls /data
myfile
/ # cat /data/myfile
hello
/ # exit
```

コンテナ「p1-c2」にログインします。
/dataの配下に、先ほど作成した「myfile」
が存在することを確認します。

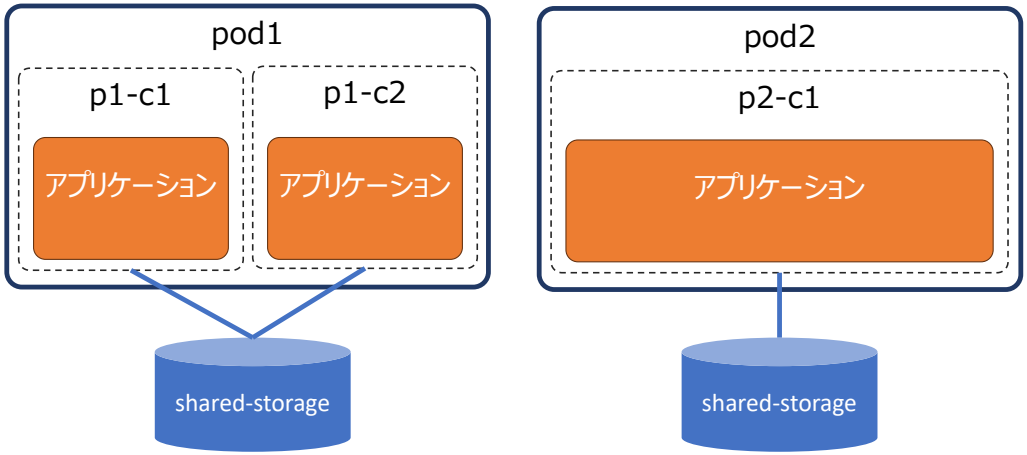
```
C:¥pj>kubectl exec -it pod1 -c p2-c1 -- /bin/sh
/ # ls /data
/ # exit
```

コンテナ「p2-c1」にログインします。
/dataの配下に、何もないことを確認します。

- 最後に、Podをすべて削除して、再度作成してみます。そして、「/data」にファイルが存在するかを確認してください。

演習：emptyDirの利用

- **この演習では、ボリュームの特徴を確認しました。**
  - ボリュームは、Pod単位で定義し、コンテナ単位でマウントします。
  - ボリュームは、同じPod内の複数コンテナで共有できます。
- **また、emptyDirで実装したボリュームは、Podのライフサイクルと一致し、Podが作成されるときに空のディレクトリが作成され、Podが削除されるとemptyDir内のデータも削除されます。**
  - キャッシュや一時ファイルの保管、またPod内のコンテナ間のデータ共有に使用されます。



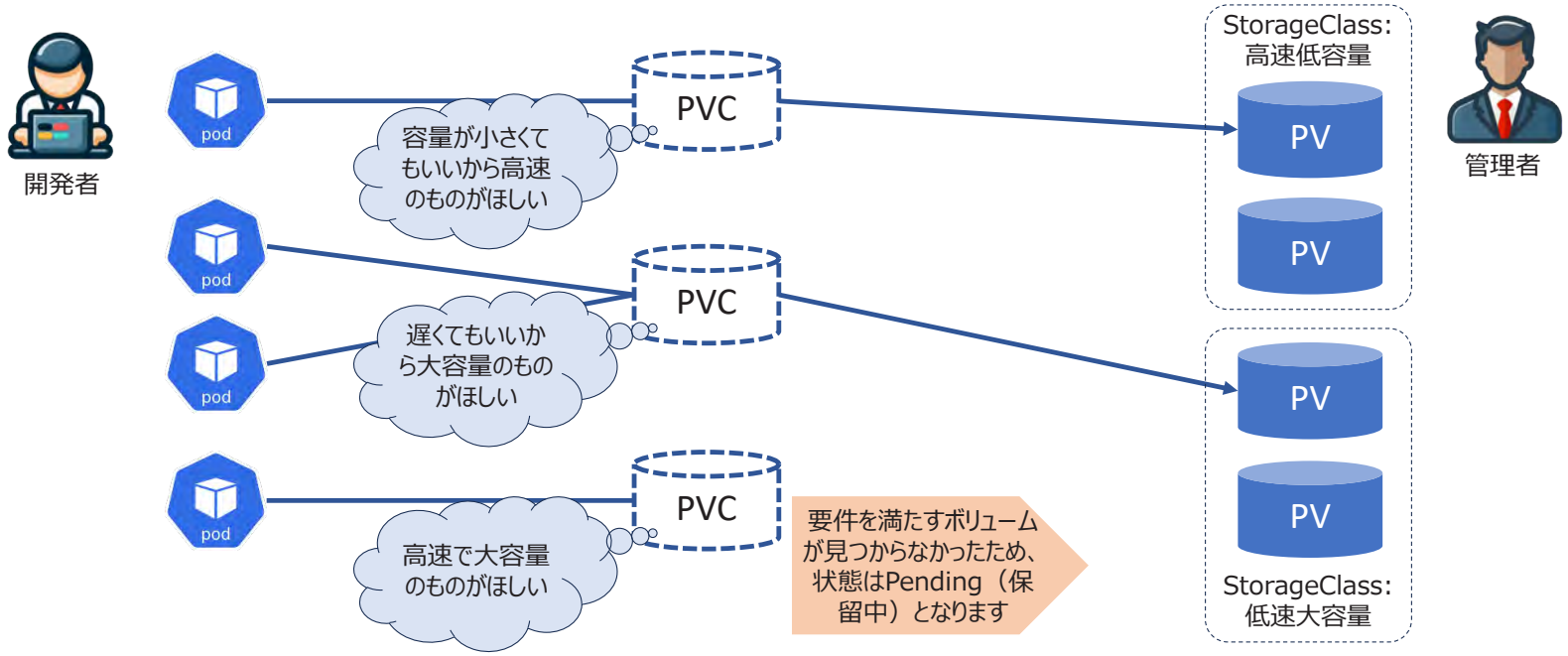
また、emptyDirは通常、ノードのディスクを媒体として使用しますが、以下のような設定でメモリ上にデータを保存できます。

```
volumes:
 - name: temp-storage
   emptyDir:
     medium: "Memory"
```

永続ボリュームのプロビジョニング

- **永続ボリュームをプロビジョニングするには、次の2つのステップが必要です。**
  - （１）管理者（インフラを管理する人）が、PV（Persistent Volume）と呼ばれるボリュームを作成する。
  - （２）開発者（インフラを利用する人）が、PVを利用するためにPVC（Persistent Volume Claim、利用申請のようなもの）を作成する。

これにより、開発者と管理者の役割が明確に分離され、システムの柔軟性が向上します。



演習：PVとPVCの作成

■ 今回の演習で利用するマニフェストです。まずはマニフェストの中身を確認しましょう。


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-pvc
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - mountPath: /data
          name: pvc-volume
  volumes:
    - name: pvc-volume
      persistentVolumeClaim:
        claimName: pod-pvc
```

Pod定義の「volumes」にある「claimName」を使用して、PVCと紐づけします。


```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pod-pvc
spec:
  storageClassName: ""
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

PVCの定義には、最低限でも以下の記述が必要です：
- キャパシティ（容量）
- アクセスモード
- StorageClass名

ピンポイントで「このPVをお願いします」という設定も可能ですが、それでも上記の条件を記載する必要があります。


```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: host-pv
spec:
  storageClassName: ""
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/pv0001/
```

PVの定義には、最低限でも以下の記述が必要です：
- キャパシティ（容量）
- アクセスモード
- StorageClass名

また、今回はテスト環境であるため、ノードのローカルディスクを利用します（hostPath）。**Podが他のノードに移動されるとアクセスできなくなる**ので、テスト以外の利用は不可です。

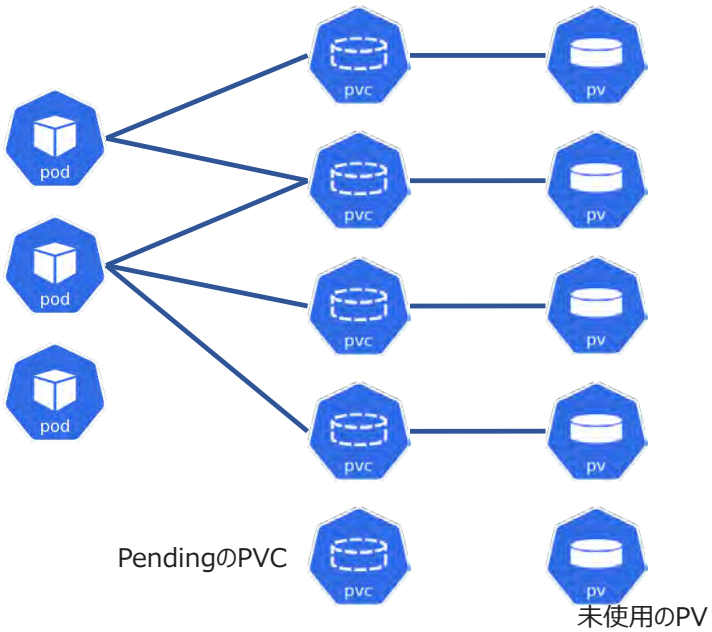
Pod/PVC/PV間の関係

PodとPVCは、多対多の関係にあります。

- 1つのPodに複数のボリュームを紐づけることが可能であり、そのため1つのPodに複数のPVCを設定できます。
- また、複数のPodが同じボリュームを共有することも可能であり、複数のPodに同じPVCを設定することができます。

PVCとPVは1対1の関係です。

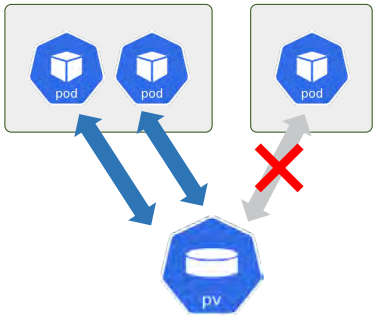
- PVは1つのPVCにのみバインド可能であり、その逆も同様です。
- 当然、作成されたものの利用されていないPVが存在する場合もあれば、要件未達でPVCがPending（保留中）になる状況もあります。



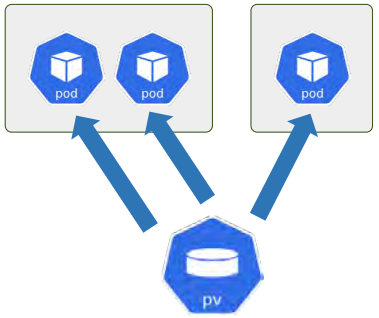
PV/PVCのアクセスモードについて

■ アクセスモードは、Podがボリュームに対して読み書きする際のモードを定義します。

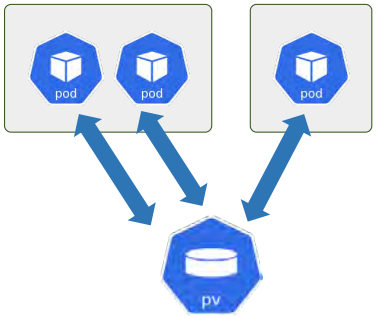
RWO	ReadWriteOnce	単一ノード上のPodからのみ、読み書きできます
ROX	ReadOnlyMany	複数ノード上のPodから、読み取り専用でアクセスできます。
RWX	ReadWriteMany	複数ノード上のPodから、読み書きできます。
RWOP	ReadWriteOncePod	Kubernetes v1.29からの新機能で、単一のPodからのみ、読み書きできます。



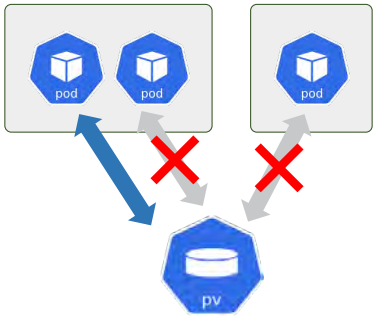
RWO



ROX



RWX



RWOP

演習：PVとPVCの作成

- では、前頁のYAMLをデプロイしてみましょう。
- デプロイ後、作成されたPVとPVCを確認します。どちらもSTATUSが「Bound」になりますが、これは「バインド済み（紐づけ済み）」という意味です。
  - PVの情報には、どのPVCによって要求（CLAIM）されたかが記載されています。
  - PVCの情報には、どのPV（VOLUME）を要求できたかが記載されています。

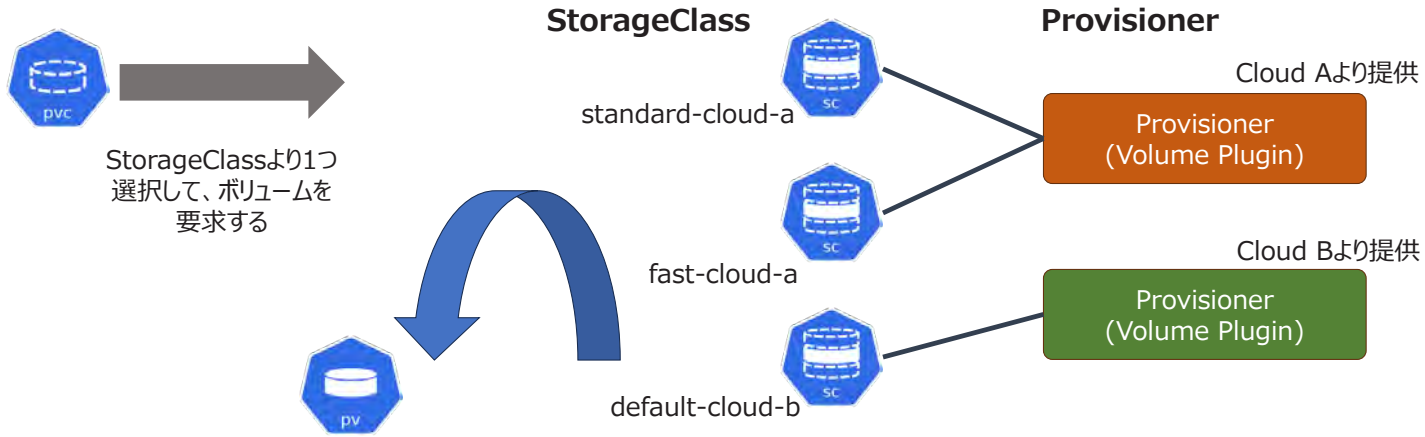
RECLAIM POLICY（PV利用終了後の処理）
Delete：PVを削除する
Retain：PV上のデータを保持する（データを含めて再利用可能）
Recycle：PV上のデータを削除するが、PVの削除せず再利用可能な状態にする

```
C:¥pj>kubectl apply -f pv.yaml
persistentvolume/host-pv created
persistentvolumeclaim/pod-pvc created
pod/pod-using-pvc created

C:¥pj>kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM             STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
host-pv   1Gi        RWO            Retain           Bound    default/pod-pvc                  <unset>                          6s

C:¥pj>kubectl get pvc
NAME      STATUS   VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
pod-pvc   Bound    host-pv   1Gi        RWO                           <unset>                 8s

C:¥pj>kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
pod-using-pvc   1/1     Running   0          11s
```

- もしもPVCの要求が未達の場合（例えば、対応可能なPVが存在しない場合）、PVCのSTATUSがPending（保留中）になり、Podも作成されません。

ボリュームの動的プロビジョニング

- 先ほどの演習の通り、ボリュームを利用するにはPVとPVCが必要です。
- PVのプロビジョニングはインフラ管理者が行う作業であり、アプリケーションが必要とするストレージ容量や特性に応じてPVを準備する作業は、時間がかかり、運用負荷が高まります。また大規模なクラスタでは、事前にニーズを把握して静的に準備することは非現実的です。
- 「なぜPVCとPVの管理を分離するのか？開発者自身が必要なときにPVまで作成すればいいのでは？」と思うかもしれません。しかし、この問題の根本的な理由は、「ほとんどの本番環境において、PVの実際のリソースはKubernetesクラスタの外にある」という点にあります。
  - たとえば、AWS環境では、PVのほとんどがAmazon EBSやAmazon EFSなどによって構成されます。
  - つまり、マニフェストや「kubectl apply」だけで、無からボリュームを作り出すことはできません（今までの方法では）。
- この課題を解決するために、Kubernetesは「動的プロビジョニング」という仕組みを提供しています。



演習：ボリュームの動的プロビジョニング

■ 事前準備：minikubeを利用している場合は、以下のコマンドでstorage-provisionerを有効にしておきます。

```
C:¥pj>minikube addons enable storage-provisioner
💡  storage-provisioner is an addon maintained by minikube. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  The 'storage-provisioner' addon is enabled
```

Pod定義は先ほどのPV/PVC演習とまったく同じものです。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-pvc
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - mountPath: /data
          name: pvc-volume
  volumes:
    - name: pvc-volume
      persistentVolumeClaim:
        claimName: pod-pvc
```

PVCの定義は先ほどのPV/PVC演習用のものからstorageClassNameの行のみ削除しています。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pod-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

PVは動的に作成されるため、定義は不要です。

演習：ボリュームの動的プロビジョニング

- デプロイの前に、まずStorageClassを確認します。minikubeとKubernetes for Docker Desktopでは、出力に若干の違いがあります。

minikube

```
C:¥pj>kubectl get storageclasses
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate           false                  7h3m
```

Kubernetes for Docker Desktop

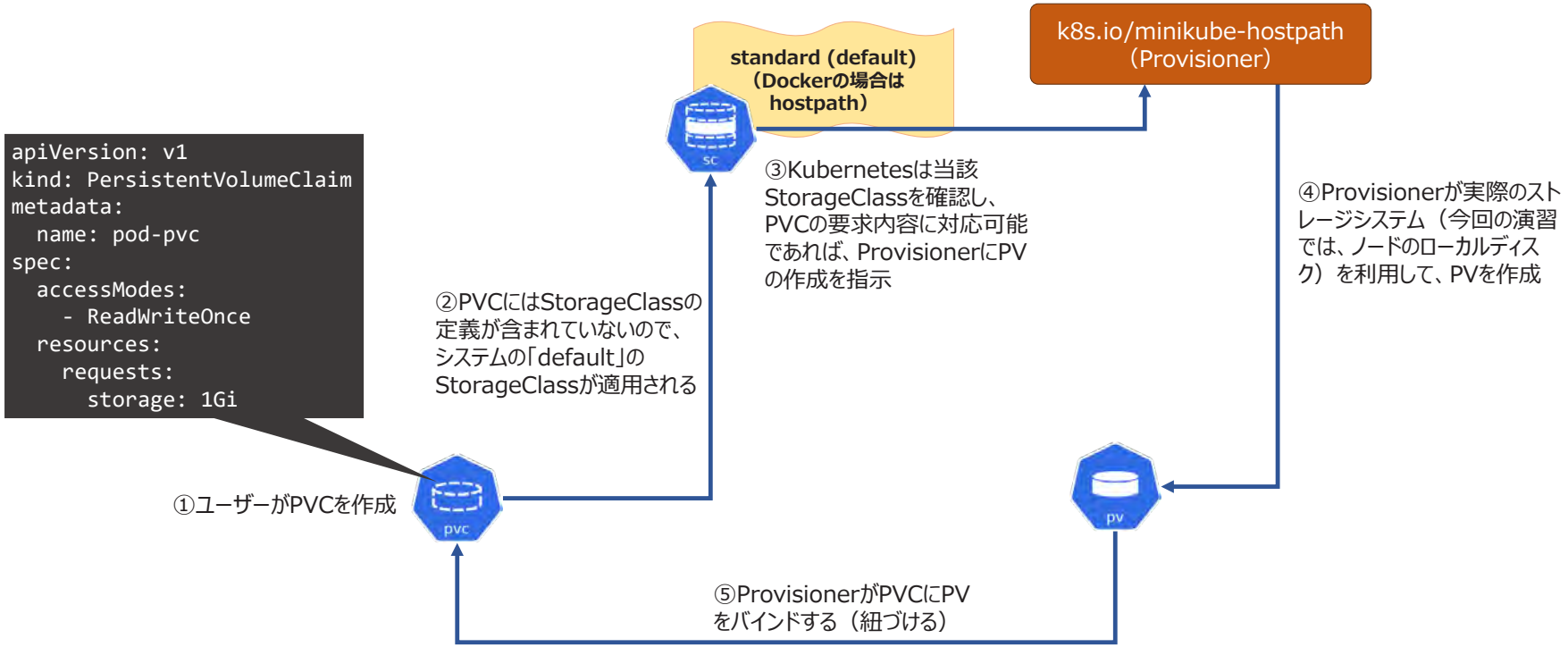
```
C:¥pj>kubectl get storageclasses
NAME                 PROVISIONER          RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
hostpath (default)   docker.io/hostpath   Delete          Immediate           false                  6d7h
```

- マニフェストをデプロイし、デプロイ結果を確認します。

```
C:¥pj>kubectl apply -f dpv.yaml
persistentvolumeclaim/pod-pvc created
pod/pod-using-pvc created

C:¥pj>kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
pod-pvc   Bound    pvc-5e0f553e-218d-43d4-be55-0ae7f87b44f3   1Gi        RWO            standard       <unset>                 48s

C:¥pj>kubectl get pv
NAME                  CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM             STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
pvc-5e0...(一部省略)   1Gi        RWO            Delete           Bound    default/pod-pvc   standard       <unset>                          50s
```

演習：ボリュームの動的プロビジョニング

- PVC作成時に、バックグラウンドで多くの処理が実行されます。



演習：ボリュームの動的プロビジョニング

- ここまでは、いくつかの種類のリソースを紹介しました。
  Pod, Deployment, Service, PV, PVC...
- これらのリソースを定義する言語、つまりマニフェスト（YAML）において、これらのリソースの記述法は、「APIオブジェクト」と言います。
- APIオブジェクトの詳細仕様、つまりマニフェストでこれらのAPIオブジェクトを作成する際に、どのような属性が利用できるか、どのような機能なのかについて、簡単に調べられるコマンドがあります：
  `kubectl explain <APIオブジェクト>`
  `kubectl explain <APIオブジェクト>.<属性>...`
- コマンドの出力は英語で表示されますが、内容を理解することで、より高度なYAMLファイルを記述する際に役立ちます。

```
C:¥pj>kubectl explain PersistentVolumeClaim
KIND:       PersistentVolumeClaim
VERSION:    v1

DESCRIPTION:
    PersistentVolumeClaim is a user's request for and claim to a persistent
    volume

FIELDS:
  apiVersion    <string>
    APIVersion defines the versioned schema of this representation of an object.
    Servers should convert recognized schemas to the latest internal value, and
    may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#resources

  kind  <string>
    Kind is a string value representing the REST resource this object
    represents. Servers may infer this from the endpoint the client submits
...

C:¥pj>kubectl explain PersistentVolumeClaim.spec.volumeMode
KIND:       PersistentVolumeClaim
VERSION:    v1

FIELD: volumeMode <string>
ENUM:
    Block
    Filesystem
...
```

ConfigMapとSecret

- **Podは、動的に作成・削除されますが、Podが存在する期間中はイミュータブル（不変）です。これは、「Immutable Infrastructure」の原則に基づいています。**
- **永続化が必要なデータは、PVCの仕組みを利用してPVに保存すればいいのですが、アプリケーションの設定情報を提供する仕組みが別途必要です。**
  - 例えば、データベースを必要とするWebアプリケーションの場合、データベースのエンドポイントURLやユーザー名、パスワードなどの情報です。
  - もちろん、これらの設定情報をコンテナイメージに組み込むことは可能ですが、イメージとして汎用性が失われます。
- **アプリケーションの設定情報を提供する仕組みは、Kubernetesとして2つ提供しています。**
  - ConfigMap：アプリケーションの設定情報、構成情報を提供する仕組み（主に平文のデータを扱う）
  - Secret：設定情報、構成情報のうち、機密データ（例: パスワード、プライベートキーなど）を提供する仕組み

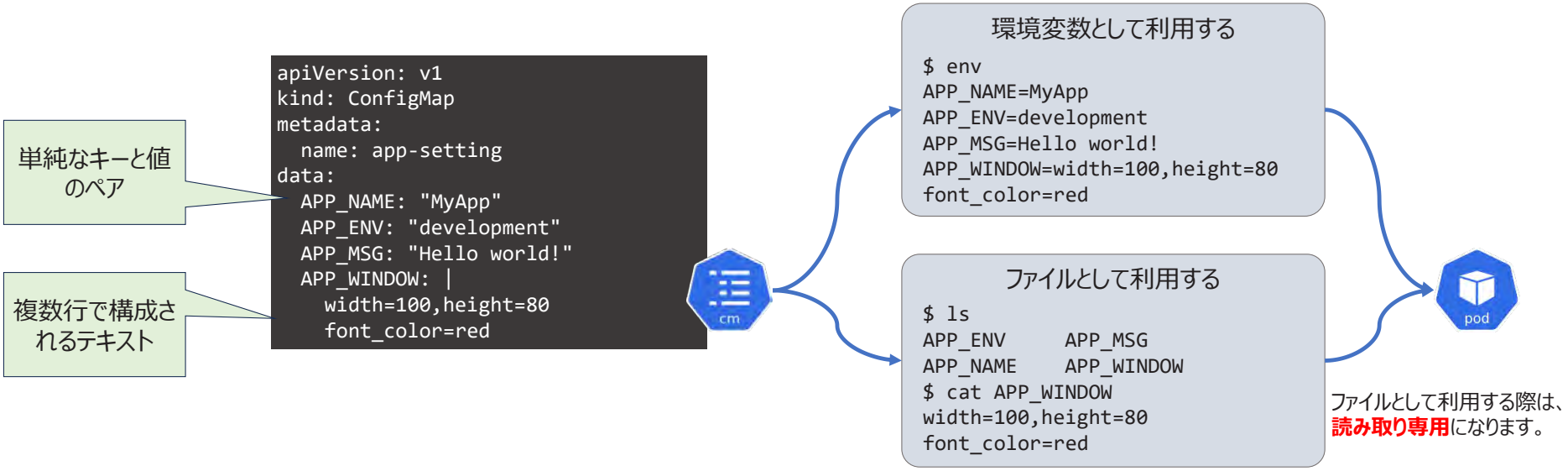
演習：ConfigMapの利用

ConfigMapの定義

- ConfigMapは、PodやPVなどのリソースと同様にYAMLで定義します（コマンドでも作成可能です）。

ConfigMapの利用

- Pod定義で定義済みのConfigMapを参照する形で利用します。
- 主に、環境変数、ボリュームの2つの形態で利用できます。



演習：ConfigMapの利用

- **ConfigMapを利用するためのPodの定義を見てみましょう。**
  - containersのenv属性は、ConfigMapからではなく環境変数の名前の値をコンテナ定義に直書きすることもできます。
  - 環境変数として利用する場合、valueFrom配下のconfigMapKeyRefを利用します。

環境変数APP_NAMEには、ConfigMap「app-setting」のAPP_NAMEという項目の値を利用します。
環境変数名とConfigMapでのキー名と必ずしも一致させる必要はありません。

ConfigMapをファイルとしてマウントします。
「items」以降の3行はオプションです。
この3行を記載しない場合、ConfigMap「app-setting」で定義したすべての項目がそれぞれ1つのファイルとして、所定のパスにマウントされます。

ファイルも、ConfigMapで定義した名称と異なるファイル名を付けることができます。ここでは、「default.config」という名前で、ConfigMap「app-setting」のAPP_WINDOWというキーの値を格納します。

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: cmcontainer
      image: docker.io/busybox:latest
      stdin: true
      tty: true
      env:
      - name: APP_NAME
        valueFrom:
          configMapKeyRef:
            name: app-setting
            key: APP_NAME
      volumeMounts:
        - name: config-volume
          mountPath: /etc/myapp-config
  volumes:
    - name: config-volume
      configMap:
        name: app-setting
        items:
        - key: APP_WINDOW
          path: default.config
```

演習：ConfigMapの利用

- ではこの2つのYAML（ConfigMapを定義するものと、Podを定義するもの）を適用します。適用後、作成したConfigMapを確認します。
- ConfigMapは2つ存在しますが、「kube-root-ca.crt」はシステム用のものになります。

```
C:¥pj>kubectl get configmap
NAME               DATA   AGE
app-setting        4      10s
kube-root-ca.crt   1      25h

C:¥pj>kubectl describe configmap app-setting
Name:         app-setting
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
APP_ENV:
----
development
APP_MSG:
----
Hello world!
APP_NAME:
----
MyApp
APP_WINDOW:
----
width=100,height=80
font_color=red
...
```

演習：ConfigMapの利用

- ConfigMapが作成されていることを確認後、コンテナのシェルにアクセスしてみます。
- 下記のように、環境変数やファイルが設定されていることがわかります。ConfigMapのすべての内容がPodに取り入れられるのではなく、Pod定義で選択したもののみ環境変数もしくはファイルとして使用可能になります。

```
C:¥pj>kubectl exec -it mypod -- /bin/sh
/ #

/ # env | grep APP
APP_NAME=MyApp

/ # ls -l /etc/myapp-config/
total 0
lrwxrwxrwx    1 root     root            21 Dec 29 07:52 default.config -> ..data/default.config

/ # cat /etc/myapp-config/default.config
width=100,height=80
font_color=red
```

- ConfigMapに関して重要な注意事項があります。ConfigMapの内容を変更しても、すでに起動中のPodには影響しません。反映させるには、Podの再作成が必要です。
  - 例えば、今回の演習で作成した「app-setting」というConfigMapを変更して、再度「kubectl apply」しても、すでに作成済みのPodには反映されません（これから作成されるPodには反映されますが）。
- 演習終了後、不要なリソースを削除してクリーンアップします。

演習：Secretの利用

- ConfigMapは、通常の設定を管理するのに適していますが、パスワードなどの秘匿性の高い設定（例えばDBへのアクセスパスワード）には適していません。
- 秘匿性の高い設定情報を管理するには、Secretを利用します。
- Secretは、ConfigMapと同様にマニフェストもしくはコマンドで作成できます。では、簡単なSecretを作成してみます。

ここでは、SecretのType（種類）をOpaque（オペーク）に設定します。
Opaqueとは、特に体系化されていない任意の値を意味します。
Opaque以外に、TLS証明書やBasic認証用の認証情報など、さまざまなType（種類）があります。

datacセクションでに、キーと値のペアで設定情報を記述できます。
ただし、値は平文ではなく、必ずBASE64でエンコードしなければなりません。
（平文で記述すると、エラーが表示れ、YAMLの適用に失敗します）

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  password: cGFzc3dvcmQ=
```

この文字列は、「password」という文字をBASE64でエンコードしたものです

- WindowsでBASE64エンコード･デコード方法を、2つ紹介します。BASE64は暗号化ではないことにご注意ください。

①WSLなど、任意のLinux環境を開きます。そして以下のコマンドを利用します。

```
echo hello | base64
aGVsbG8K
echo aGVsbG8K | base64 -d
hello
```

②Powershell環境を開き、以下のコマンドを利用します。

```
[Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes("hello"))
aGVsbG8K
[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String("aGVsbG8K"))
hello
```

演習：Secretの利用

- Secretの利用側（Pod）では、ConfigMapと同様に、環境変数もしくはボリュームとしてマウントして、Secretを利用します。
- ここでは、Pod定義のYAMLファイルを省略しますが、一部のみサンプルとして掲載します。ConfigMapとほぼ同じ利用法ですが、ConfigMapの「configMapKeyRef」を「secretKeyRef」に変更するだけで大丈夫です。
- では実際にデプロイしてみます。デプロイ後、デプロイしたSecretを確認します。

```
env:
      - name: APP_NAME
        valueFrom:
          configMapKeyRef:
            name: app-setting
            key: APP_NAME
      - name: PASS
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: password
```

```
C:¥pj>kubectl get secret
NAME             TYPE     DATA   AGE
db-credentials   Opaque   1      32s

C:¥pj>kubectl describe secret db-credentials
Name:         db-credentials
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  8 bytes
```

一見すると、パスワードが表示されていないため、秘匿性が高いように思われます。しかし、実際のところはどうでしょうか？

演習：Secretの利用

- APIオブジェクトの定義を出力するコマンド（kubectl get … -o yaml）を使用すると、BASE64でエンコードされた情報はそのまま表示されました。また、コンテナのシェルにアクセスすれば、平文を確認することも可能です。Secretを使用しているからといって安全とは限らないので、注意が必要です。

```
C:¥pj>kubectl get secret db-credentials -o yaml
apiVersion: v1
data:
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"password":"cGFzc3dvcmQ="},"kind":"Secret","metadata":{"annotations":{},"name":"db-
credentials","namespace":"default"},"type":"Opaque"}
  creationTimestamp: "2024-12-29T08:41:32Z"
  name: db-credentials
  namespace: default
  resourceVersion: "97616"
  uid: cb6bc90d-0ebe-4261-be31-8179cba3996f
type: Opaque

C:¥pj>kubectl exec -it mypod -- /bin/sh
/ # echo $PASS
password
/ #
```

Secretを実際に利用する際は、多くの場合、追加の暗号化を適用します。特にAWSなどのパブリッククラウド環境では、クラウドと連携してSecretの管理や暗号化を行うことが一般的です。

確認テスト1

Q1: emptyDirについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. emptyDirは、Pod作成時に自動的に作成され、Podが削除されるとその内容も消去される
2. emptyDirは、Pod間でデータ共有に使用できる
3. emptyDirは、同一Pod内のコンテナ間でデータ共有に使用できる
4. emptyDirは、メモリに作成することができる

Q2: PersistentVolume（PV）とPersistentVolumeClaim（PVC）の関係についての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. PVCは、Podが必要とするストレージをリクエストするためのリソースである
2. PVは、クラスタ外部のストレージリソースを表現するオブジェクトである
3. PVとPVCがバインドされると、PVはそのPVC専用となる
4. 同一のPVは複数のPVCにバインドすることが可能である

確認テスト2

Q3: PodとPersistentVolumeClaim（PVC）の関係についての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. 複数のPodが同じPVCを指定できる
2. Podは、PVCを複数指定して異なるボリュームにアクセスすることができる
3. Podは、PVCを指定する際に必ずストレージクラスも同時に指定する必要がある
4. Podが削除されると、PVCにバインドされているPVも自動的に削除される

Q4: PersistentVolume（PV）およびPersistentVolumeClaim（PVC）のアクセスモードに関する記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. ReadWriteOnceは、単一のPodのみが読み書き可能であることを意味する
2. ReadOnlyManyは、複数のPodが同時に読み取り可能であることを意味する
3. ReadWriteManyは、複数のPodが同時に読み書き可能であることを意味する
4. ReadWriteOncePodは、単一のPodのみがPVにアクセスできることを保証する
5. PVCで指定するアクセスモードはPVのアクセスモードと一致している必要はない

確認テスト3

Q5: ボリュームの動的プロビジョニングに関する記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. 動的プロビジョニングでは、PVCが作成されたときにPVが自動的にプロビジョンされる
2. 動的プロビジョニングはオンプレミス環境では使用できない
3. 動的プロビジョニングが失敗した場合、PVCは自動的に削除される
4. 動的プロビジョニングを使用するには、PVCでStorageClassを指定する必要がある
5. 動的プロビジョニングを利用すると、手動でPVを作成する手間が省ける

Q6: PVCを作成する際に、storageClassNameを指定しなかった場合、どのような挙動になるか？最も正しいものを選択してください。

1. 自動的にすべてのStorageClassにバインドされる
2. デフォルトのStorageClassが設定されていれば、そのStorageClassが使用される
3. どのStorageClassも使用されず、常に作成が失敗する
4. デフォルトのStorageClassがなくても、Kubernetesが自動的に新しいStorageClassを作成する

確認テスト4

Q7: ConfigMapは通常、どのようにPodから利用するか？当てはまるものをすべて選択してください。

1. 環境変数として利用する
2. コンテナのイメージに直接バンドルすることで利用する
3. ボリュームとしてマウントし、ファイルとして利用する
4. 実行時にPodのラベルに基づいて自動的に設定される

Q8: Secretに関する記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. Secretは、環境変数としてPodに渡すことができる
2. Secretは、ボリュームとしてマウントすることで、ファイルとしてPodから利用できる
3. Secretは、値をBase64で暗号化して保存するため、ConfigMapよりセキュアである
4. 作成済のSecretを、どんなkubectlコマンドでもその内容を確認することはできない

第六章 Kubernetesのスケジューリング

この章の内容

- **本章は、Podのスケジューリング（Podをどのノードに配置するかを決定するプロセス）について演習を通じて解説します。**
  - ノードの指定（nodeSelector）
  - ノードアフィニティの設定（nodeAffinity）
  - Podアフィニティおよびアンチアフィニティの設定（podAffinity, podAntiAffinity）
  - Taint及びToleration
  - Podの分散（topologySpreadConstraints）

演習についての注意事項

- 本章の内容は、Kubernetesのスケジューリングになります。スケジューリングとは、Podをどのノードに配置するかを決定するプロセスのことです。
- そのため、ノードが1つしかない環境では、正しく動作を確認することができません。本章の演習内容を実機で確認するには、minikubeやkindで3つ以上のノードを備えたクラスタを作成してください。
- minikubeで複数ノードのクラスタを作成するには、以下のコマンドを利用します。

```
# 3ノードのクラスタを作成
minikube start --nodes=3

# ドライバ、コンテナランタイムなどを指定して、3ノードのクラスタを作成
minikube start --driver=docker --container-runtime=containerd --kubernetes-version=latest --nodes=3

# 特定のプロファイルで作成
minikube start -p myprofile --driver=docker --container-runtime=containerd --kubernetes-version=latest --nodes=3
```

- また、複数のプロファイルを作成した場合、適宜minikube profileコマンド及びkubectl use-contextコマンドでプロファイル/コンテキストの切り替えを実施してください。

ノードのラベル

- ラベル（Label）は、メタデータの一種で、Podやノード、サービスなど様々なリソースに付与することができます。getコマンドで「--show-labels」オプションを使用すると、ラベルを確認することができます。

```
C:¥pj>kubectl get services --show-labels
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE   LABELS
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   45h   component=apiserver,provider=kubernetes

C:¥pj> kubectl get nodes --show-labels
NAME           STATUS   ROLES           AGE   VERSION   LABELS
minikube       Ready    control-plane   45h   v1.31.0   beta.kubernetes.io/arch...(省略)
minikube-m02   Ready    <none>          45h   v1.31.0   beta.kubernetes.io/arch ...(省略)
minikube-m03   Ready    <none>          45h   v1.31.0   beta.kubernetes.io/arch ...(省略)
```

- Podやサービスなどのラベル指定は、通常リソースを作成する際に、マニフェストの「metadata」部分で指定します。
しかし、ノードの場合は、ノードは通常マニフェストによって作成されるわけではないので、以下のようにコマンドで直接指定することが多いです（ノードの動的プロビジョニングも可能ですが、9章と10章で紹介します）。

ノードにラベルを付与する：
```
kubectl label nodes <node-name> <key>=<value>
```

ノードのラベルを書き換える：
```
kubectl label nodes <node-name> <key>=<value> --overwrite
```

ノードにラベルを削除する
```
kubectl label nodes <node-name> <key>-
```

```
C:¥pj>kubectl label node minikube vendor=dell
node/minikube labeled

C:¥pj>kubectl label node minikube vendor=hp
error: 'vendor' already has a value (dell), and --overwrite is false

C:¥pj>kubectl label node minikube vendor=hp --overwrite
node/minikube labeled

C:¥pj>kubectl label node minikube vendor-
node/minikube unlabeled
```

ノードのラベル

- minikubeの場合、ノードは最初から多数のラベルが付与されています。kubectlのdescribeコマンドで、これらのラベルをより詳細に確認できます。
- またminikubeでは、複数ノードのクラスタの場合、1番目のノードとそれ以外のノードのラベルが異なります。
- 以下では、わかりやすくするために、相違点を水色でマークしています。

```
C:¥pj>kubectl describe node minikube
Name:               minikube
Roles:              control-plane
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=minikube
                    kubernetes.io/os=linux
                    minikube.k8s.io/commit=210b...
                    minikube.k8s.io/name=minikube
                    minikube.k8s.io/primary=true
                    minikube.k8s.io/updated_at=2024_12_28T14_53_51_0700
                    minikube.k8s.io/version=v1.34.0
                    node-role.kubernetes.io/control-plane=
                    node.kubernetes.io/exclude-from-external-load-balancers=
                    vendor=dell
```

こちらは、kubectl labelコマンドで手動で追加したラベルです。

```
C:¥pj>kubectl describe node minikube-m02
Name:               minikube-m02
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=minikube-m02
                    kubernetes.io/os=linux
                    minikube.k8s.io/commit=210b...
                    minikube.k8s.io/name=minikube
                    minikube.k8s.io/primary=false
                    minikube.k8s.io/updated_at=2024_12_28T14_54_16_0700
                    minikube.k8s.io/version=v1.34.0
```

システムが定義したラベルも、手動で追加したラベルも、Podの配置先を指定する際に利用できます。

演習：ノードの指定

- では演習を通じて、Podを特定のノードに配置する方法を学習します。
- この演習は、複数ノードのKubernetesクラスタを必要とします。
- 以下のコマンドで、ノードの一覧を確認してから、任意の1つのノードにラベルを付与します：

```
C:¥pj>kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
minikube       Ready    control-plane   46h   v1.31.0
minikube-m02   Ready    <none>          46h   v1.31.0
minikube-m03   Ready    <none>          46h   v1.31.0

C:¥pj>kubectl label node minikube-m02 vendor=dell
node/minikube labeled
```

- ラベルが正しく付与されていることを確認するには、kubectl describe nodeコマンドでも問題ありませんが、getの「-l」オプションも利用可能です：

```
C:¥pj>kubectl get nodes -l vendor=dell
NAME           STATUS   ROLES    AGE   VERSION
minikube-m02   Ready    <none>   46h   v1.31.0
```

- そして、右側のマニフェストを使用して、2つのPodを作成します。specでは「nodeSelector」が指定されています。では、この2つのPodは、それぞれどのノードに配置されるのでしょうか？

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  nodeSelector:
    vendor: dell
  containers:
  - name: nginx
    image: nginx:1.21.6
    ports:
    - containerPort: 80
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod2
spec:
  nodeSelector:
    vendor: fujitsu
  containers:
  - name: nginx
    image: nginx:1.21.6
    ports:
    - containerPort: 80
```

演習：ノードの指定

- **マニフェストをデプロイして結果を確認します。**
  - Pod1は、予想通りminikube-m02に配置されています。これは、指定したラベル（vendor=dell）がこのノードにのみ存在しているためです。
  - Pod2はPending（保留中）状態となり、起動しませんでした。詳細をdescribeコマンドで確認すると、FailedSchedulingエラーが表示されました。メッセージには「3つのノードのいずれも、Podのnode affinity/selectorに一致しない」と書いてあります。

```
C:¥pj>kubectl get pods -o wide
NAME   READY   STATUS    RESTARTS   AGE
NAME   READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
pod1   1/1     Running   0          93s   10.244.1.7   minikube-m02   <none>           <none>
pod2   0/1     Pending   0          93s   <none>       <none>         <none>           <none>
```

```
C:¥pj>kubectl describe pod pod2
Name:             pod2
Namespace:        default
Priority:         0
...
Node-Selectors:              vendor=fujitsu
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  20s   default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affinity/selector.
preemption: 0/3 nodes are available: 3 Preemption is not helpful for scheduling.
```

演習：ノードの指定

- では、ここで手動で「vendor=fujitsu」というラベルを任意のノードに追加します。これにより、Pending状態のPodが配置されるかどうかを確認します。

```
C:¥pj>kubectl label node minikube-m03 vendor=fujitsu
node/minikube-m03 labeled

C:¥pj>kubectl get pods -o wide
NAME    READY   STATUS              RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
pod1    1/1     Running             0          10m   10.244.1.7   minikube-m02   <none>           <none>
pod2    0/1     ContainerCreating   0          10m   <none>       minikube-m03   <none>           <none>

C:¥pj>kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP            NODE           NOMINATED NODE   READINESS GATES
pod1    1/1     Running   0          10m   10.244.1.7    minikube-m02   <none>           <none>
pod2    1/1     Running   0          10m   10.244.2.16   minikube-m03   <none>           <none>
```

- 上記のように、ラベルを付与するとすぐにPodがスケジュールされ、数秒後には完全に起動しました。

ノードセレクター（nodeSelector）は、KubernetesにおいてPodを特定のノードにスケジュールするための簡単で直接的な方法です。例えば、Podでvendor=dellを指定すれば、そのラベルを持つノードのみにPodが配置されます。

設定は簡単ですが、単一条件しか指定できず柔軟性に欠けるため、複雑な条件にはノードアフィニティ（nodeAffinity）が推奨されます。

アフィニティ設定

- **nodeSelectorは最もシンプルな方法ですが、細かい条件でより複雑なロジックでノードを選定する場合、アフィニティ（Affinity）を利用します。**
- **アフィニティとは、親和性や近接性を指し、Podやノードの配置に関する関係性を定義するものです。例えば、**
  - 「このPodは特定のノードに配置したい」
  - 「このPodは他の特定のPodと同じノードに配置したい」
  - 「このPodは特定のPodと異なるノードに配置したい」

  といった条件を指定できます。これにより、Podとノード間や、Pod同士の適切な配置を柔軟にコントロールできます。
- **Kubernetesでは、3種類のアフィニティをサポートしています。**

アフィニティの種類	目的	具体例
ノードアフィニティ（nodeAffinity）	ノードのラベルに基づいてPodをスケジュール	vendor=dellのラベルを持つノードに配置
ポッドアフィニティ（podAffinity）	他のPodのラベルに基づいて特定のPodの近くに配置	app=backendのラベルを持つPodが所在するノードに配置
ポッドアンチアフィニティ（podAntiAffinity）	他のPodのラベルに基づいて特定のPodを離して配置	同じノードにapp=frontendのPodを配置しない

ノードアフィニティ（nodeAffinity）

- ノードアフィニティ（nodeAffinity）は、ノードセレクター（nodeSelector）と同様な目的の仕組みですが、より高度な設定が可能です。
- ノードアフィニティには、2つの指定方法があります。
  - **必須条件**として指定する：満たされなければスケジュール不可
    使用するキーワード：
    `requiredDuringSchedulingIgnoredDuringExecution`
  - **優先条件**として指定する：満たされなくてもスケジュール可能
    使用するキーワード：
    `preferredDuringSchedulingIgnoredDuringExecution`
- 右側の例は、前の演習で使用したノードセレクターと同様に、「vendor=dell」のラベルが付いたノードを選択する内容です。
- 書き方は少し複雑になりますが、これにより、より高度な条件指定が可能になります。では、その仕組みを確認してみましょう。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-vendor
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: vendor
            operator: In
            values:
              - dell
  containers:
  - name: nginx
    image: nginx:1.21.6
    ports:
    - containerPort: 80
```

spec配下に、まずaffinity階層があり、その配下にnodeAffinityを指定

必須条件として指定

具体的に指定する条件

ノードアフィニティ（nodeAffinity）

- **ノードアフィニティは、複雑な条件にも対応できます。以下のシナリオを想像してください。**
- **ある会社は、3種類の物理サーバーをKubernetesクラスタのノードとして使用しています。あるPodの配置条件は：**
  - メーカーから購入したサーバー（method=vendor）の場合、CPUがIntelもしくはAMDで、メモリが2000MBを超えるノードが必要です。
  - 自作サーバー（method=diy）の場合は、CPUを問わず、メモリが1000MBを超えていれば問題ありません。
  - 流用資産などのその他サーバー（method=other）の場合は、特にスペックの制約はありません。
- **これをプログラミング言語風に表現すると、以下のようになります。**

```
(method==vendor AND (cpu==intel OR cpu==amd) AND memory>2000) OR (method==diy AND memory>1000) OR method==other
```

- **仮に以下の3台のノードがあるとしたら、Podはどのノードにスケジュールされますでしょうか？**

minikube
method=vendor
cpu=intel
memory=1500

minikube-m02
method=diy
cpu=samsung
memory=1500

minikube-m03
method=diy
memory=800

演習：ノードアフィニティ

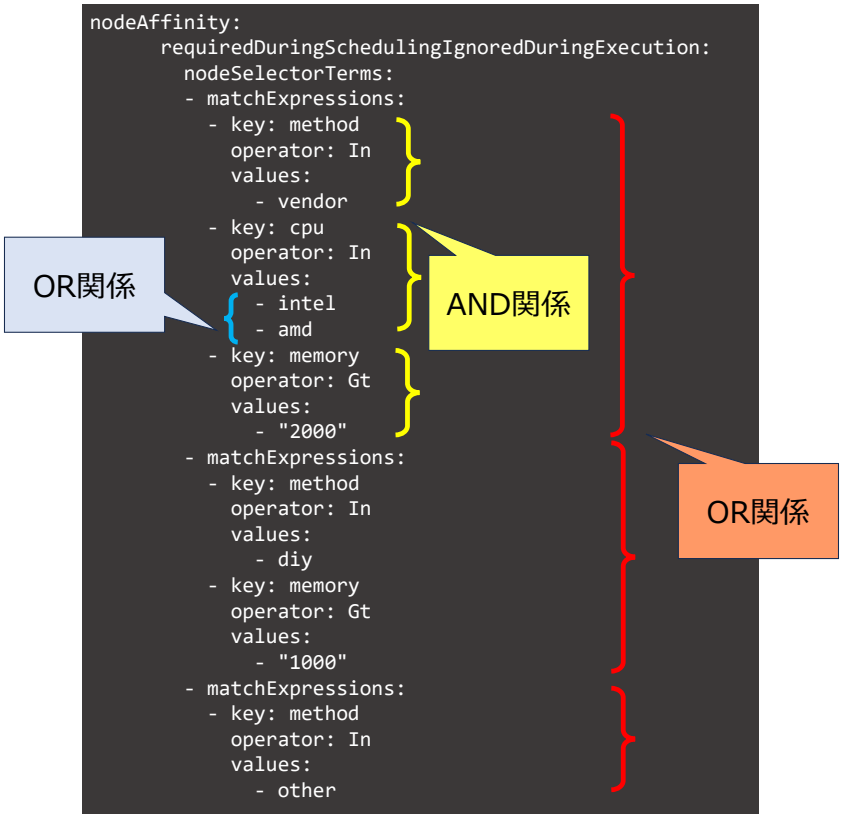
- 演習の前に、ノードを確認して、不要なレベルを削除しておきましょう。また、この演習では、3台のノード（minikube, minikube-m02, minikube-m03）が存在することを前提とします。
- そして、以下のコマンドでそれぞれのノードにラベルを付与します。

```
C:¥pj>kubectl label nodes minikube method=vendor cpu=intel memory=1500
node/minikube labeled

C:¥pj>kubectl label nodes minikube-m02 method=diy cpu=samsung memory=1500
node/minikube-m02 labeled

C:¥pj>kubectl label nodes minikube-m03 method=diy memory=800
node/minikube-m03 labeled
```

- 右側のYAMLをデプロイします。Podがどのノードにスケジュールされるかを確認します。
  - Podは「minikube-m02」で起動しているはずです。
  - ノード「minikube」は、methodとcpuの条件は満たしていますが、memoryが2000未満のため、選択されませんでした。
  - ノード「minikube-m03」は、methodがdiyで、この場合memoryが1000を超える必要がありますが、800しかないため、選択されませんでした。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-vendor
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: method
            operator: In
            values:
              - vendor
          - key: cpu
            operator: In
            values:
              - intel
              - amd
          - key: memory
            operator: Gt
            values:
              - "2000"
        - matchExpressions:
          - key: method
            operator: In
            values:
              - diy
          - key: memory
            operator: Gt
            values:
              - "1000"
        - matchExpressions:
          - key: method
            operator: In
            values:
              - other

  containers:
  - name: nginx
    image: nginx:1.21.6
    ports:
    - containerPort: 80
```

演習：ノードアフィニティ



- ノードアフィニティの定義を詳細に見てみましょう。
- 必須条件を示すrequiredDuringSchedulingIgnoredDuringExecutionの配下に、1つの要素しかありません。それはnodeSelectorTermsです。
- nodeSelectorTermsの配下に、1個もしくは複数のmatchExpressionsがあります。これらのmatchExpressionsの間は、OR関係になります。
- matchExpressionsの配下に、複数の条件文が記載されています。条件文の間は、AND関係です。
- 条件文は、key / operator / valuesの3つの要素から構成されます。比較演算子（operator）は、以下のようなものが使用可能です。

演算子	意味	valuesの記述方法
In	ノードのkeyラベルのvalue値がvaluesの中に含まれている	複数指定可能（OR関係）
NotIn	ノードのkeyラベルのvalue値がvaluesの中に含まない	複数指定可能（OR関係）
Exists	ノードに指定のkeyラベルが存在する	使用不可
NotExists	ノードに指定のkeyラベルが存在しない	使用不可
Gt	ノードのkeyラベルのvalue値がvaluesの値より大きい	1つのみ指定可能
Lt	ノードのkeyラベルのvalue値がvaluesの値より小さい	1つのみ指定可能

ノードアフィニティ（nodeAffinity）

- ここまでは、必須条件の場合のノードアフィニティの設定方法を学習しました。
  では、優先条件（preferredDuringSchedulingIgnoredDuringExecution）の場合はどのように設定しますか？

- 例えば、以下のようにラベルが付与された4台のノードに対して、右側のYAMLをデプロイするとします。このYAMLは、PodではなくDeploymentを利用して、一括で複数のPodをデプロイしています。

- ノードアフィニティには、2つの優先条件が定義されています。
  1つ目の条件は、ウェイトが80で、[disktype=ssd]のラベルを評価します。
  2つ目の条件は、ウェイトが50で、[region=tokyo]のラベルを評価します。

- 結果はどうなりますでしょうか？

minikube	minikube-m02	minikube-m03	minikube-m04
disktype=ssd	region=tokyo	disktype=ssd	
		region=tokyo	

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment
spec:
  replicas: 10
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 80
              preference:
                matchExpressions:
                  - key: disktype
                    operator: In
                    values:
                      - ssd
            - weight: 50
              preference:
                matchExpressions:
                  - key: region
                    operator: In
                    values:
                      - tokyo
      containers:
      - name: nginx
        image: nginx:1.21.6
```

ノードアフィニティ（nodeAffinity）

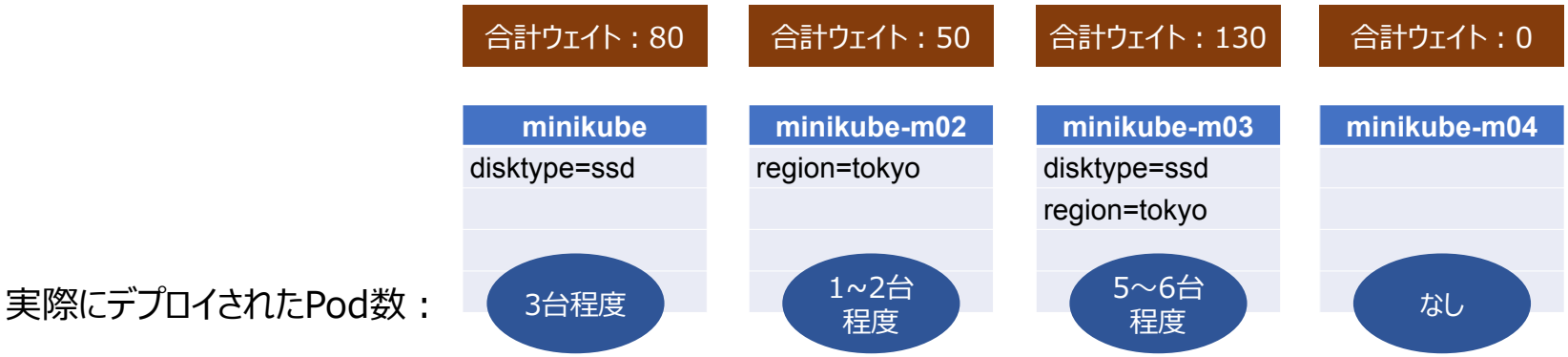
ウェイト（weight）の計算

- ウェイトは、1から100までの整数で指定できる値です。値が大きいほど、その条件を満たすノードがスケジューリングの候補として強く優先されます。
- 複数条件がある場合、ノードが満たした各条件のウェイト値が合算されます。

ウェイトに基づいたスケジューリング

- 合計ウェイトが最も高いノードが優先されます。ただし、スケジューラは分散配置戦略やリソース状況も考慮するため、必ずしもすべてのPodがそのノードに集中するわけではありません。結果的に、合計ウェイトが高いノードほど多くのPodが配置されますが、リソースのバランスに基づいて適切に分散されます。

例えば、以下は結果の一例です。



Podのアフィニティとアンチアフィニティ

- **Podのアフィニティとアンチアフィニティを理解する前に、まずトポロジドメインを解説します。**
- **Podのアフィニティとアンチアフィニティは、Podを「近くに配置」「離して配置」するためのものですが、その前に「近くに」と「離して」を定義しなければなりません。**
- **その定義には、やはりラベルを使用します。どれか1つのラベルを利用して、ノードが所属する領域をマークすればよいでしょう。**
  - 例えば、以下の例では「color」というラベルを使用すると、node1とnode2は同じくred、node3とnode4は同じくgreen、そしてnode5はyellowとして、3つの領域（ゾーン）に分けることができます。
  - 一方、「size」というラベルを使用する場合、node1、node2、node3は同じゾーンに入り、node4とnode5は別のゾーンに入ります。
  - また、「region」というラベルを使用すると、各ノードがそれぞれ異なるゾーンに分類されます。
- **「ゾーン分けに使用するラベル名」は、トポロジキー（topologyKey）と言います。また、トポロジキーによって分割された個々のゾーンは、トポロジキードメイン（topology domain）と言います。**

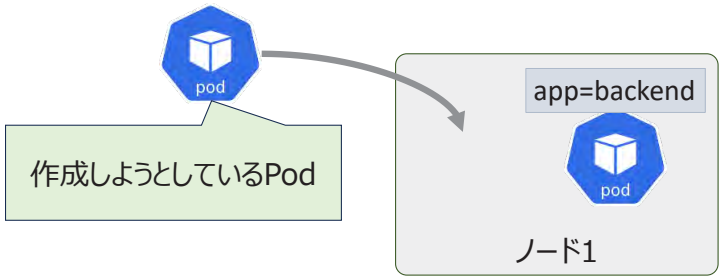
node1	node2	node3	node4	node5
color=red	color=red	color=green	color=green	color=yellow
size=small	size=small	size=small	size=big	sizu=big
shape=circle	shape=box	shape=box	shape=box	shape=box
region=tokyo	region=osaka	region=sapporo	region=nagoya	region=fukuoka

これにより、以下のように定義できます。

- **「近くに配置」：同じトポロジドメインに配置する**
- **「離して配置」：異なるトポロジドメインに配置する**

Podのアフィニティとアンチアフィニティ

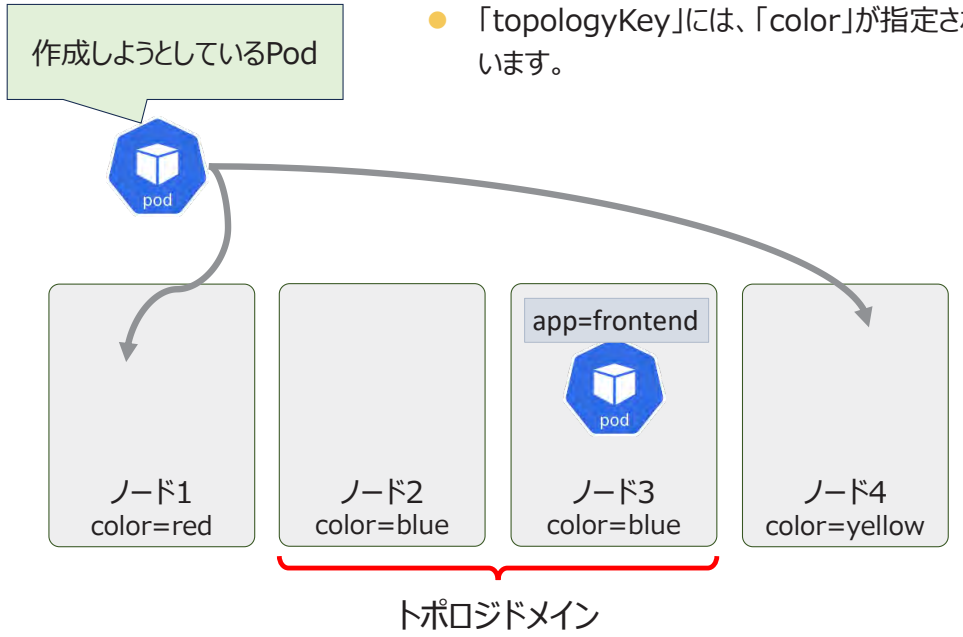
- **まずはPodのアフィニティを見てみます。**
- **例えばこちらのポッドアフィニティを使用してPodを作成すると、必ず「app=backend」のラベルを持つ既存Podと、同じノード上に配置されます。**
  - 必須条件（requiredDuringSchedulingIgnoredDuringExecution）としての指定なので、必ずそのように配置されます。
  - 「labelSelector」に記載されているのは、アフィニティ対象のPodのラベルです。
  - 「topologyKey」には、デフォルトのラベル「kubernetes.io/hostname」が指定されています。このラベルの値は、ノードのホスト名になりますので、「**ホスト名単位でトポロジドメインを分割する**」ことを意味しています。




```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: app
              operator: In
              values:
                - backend
        topologyKey: "kubernetes.io/hostname"
```

Podのアフィニティとアンチアフィニティ

- 次は、Podのアンチアフィニティです。
- こちらのポッドアンチアフィニティの例では、このマニフェストによって作成されるPodは、必ずノード2とノード3を避けて配置されることになります。
  - 「topologyKey」には、「color」が指定されています。「**colorの値をベースにトポロジドメインを分割する**」ことを意味しています。




```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: app
              operator: In
              values:
                - frontend
        topologyKey: "color"
```

同時指定時の動作

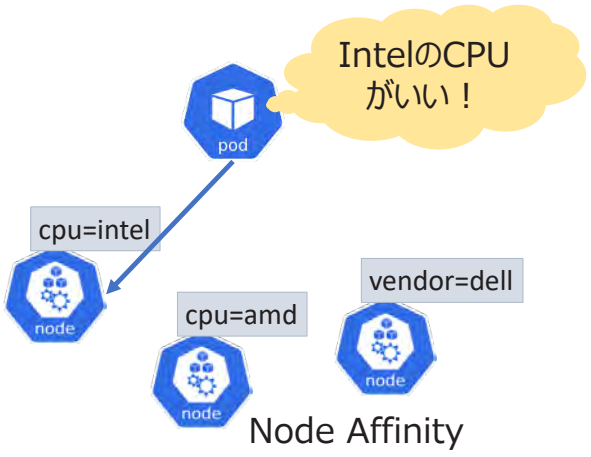
- **nodeSelect, nodeAffinity, podAffinity, podAntiAffinityが同時に指定された場合の動作は以下の通りです。**
  - まず、それぞれの必須条件（requiredDuringSchedulingIgnoredDuringExecution）がANDで評価されます。この際、nodeSelectorに記載されている内容は必須条件として扱われます。
  - 次に、それぞれの必須条件をANDで評価した結果、矛盾がある場合は、Podの作成がPending（保留中）になります。一方で、矛盾がなければ、すべての必須条件を満たすノードが選出されます。
  - その後、選出されたノードについて、優先条件（preferredDuringSchedulingIgnoredDuringExecution）が評価され、ウェイトの観点で最も優先されるノードが選定されます。
  - 最終的に、ウェイト情報やリソースの使用状況を考慮して、Podがスケジュールされます。

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-affinity-pod
spec:
  nodeSelector:
    disktype: ssd
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
       ...
      preferredDuringSchedulingIgnoredDuringExecution:
       ...
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
       ...
      preferredDuringSchedulingIgnoredDuringExecution:
       ...
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
       ...
      preferredDuringSchedulingIgnoredDuringExecution:
       ...
  containers:
  - name: nginx
    image: nginx:1.21.6
```

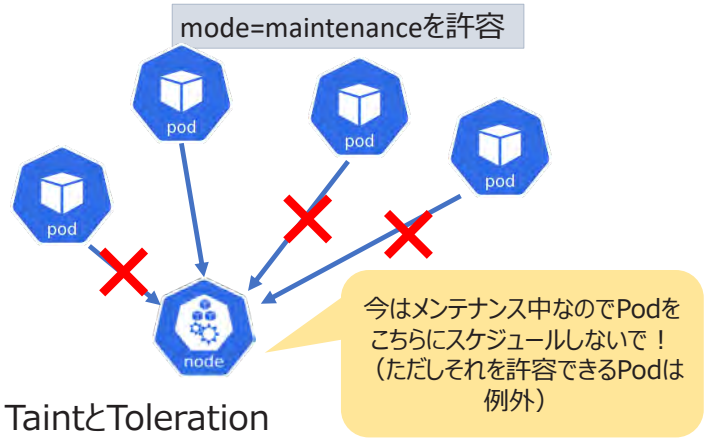
TaintとToleration

- **Podスケジューリングに関連して、もう1つ重要な概念があります。それがTaint（テイント）とToleration（トラレーション）です。これらの用語は日本語訳がまだ定着していないため、ここではローマ字で表記します。**
  - Taint：TaintはNodeに設定するプロパティで、特定の条件を満たさないPodがそのNodeにスケジュールされるのを防ぎます。
  - Toleration：TolerationはPodに設定するプロパティで、特定のTaintを許容することで、対応するNodeにスケジュール可能になります。つまりTaintの制約を緩和することができるプロパティです。

Node Affinity	TaintとToleration
Pod側の設定：PodがスケジュールされるNodeや他のPodとの関係性を制御する仕組み。	Node側の設定：NodeにTaintを設定し、それをPodが許容（Toleration）することでスケジュール可能にする仕組み。
主にPodがNodeを「選択」するために使用	主にNodeがPodを「拒否」するために使用



Node Affinity



TaintとToleration

演習：TaintとToleration

- レプリカ数が1のシンプルなDeploymentを作成します。
- 作成後、作成されたPodの稼働しているノードを確認します。このスライドの例では、minikube-m04で動作していることがわかります。
- 以下のコマンドで、Taintを設定します。Taintは、key=value:effectの形式で定義します。「effect」は「効果」で、下記の表から指定できます。

```
kubectl taint nodes <ノード名> <Taint>
```

NoSchedule	Podをこのノードにスケジュールしない
PreferNoSchedule	可能であればPodをこのノードにスケジュールしない
NoExecute	Podをこのノードにスケジュールしない。またすでにこのノードで稼働しているPodをEvict（削除）する

Evictされると、Podの場合そのまま削除されますが、Deploymentの場合は別のノードで再作成され、起動されます。

```
C:¥pj>kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP            NODE           NOMINATED NODE   READINESS GATES
myapp-5c6bdd9bcb-jc5gh   1/1     Running   0          12s   10.244.4.17   minikube-m04   <none>           <none>

C:¥pj>kubectl taint nodes minikube-m04 env=prod:NoExecute
node/minikube-m04 tainted

C:¥pj>kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP            NODE           NOMINATED NODE   READINESS GATES
myapp-5c6bdd9bcb-qnpk9   1/1     Running   0          2s    10.244.2.89   minikube-m03   <none>           <none>
```

- 今回は、「NoExecute」のTaintを追加したため、DeploymentがPodを他のノードでPodを再起動することを確認できます。

演習：TaintとToleration

- 次に、先ほど作成したTaintを許容（Toleration）できるPodを作成してみます。
- まずは、念のためTaintの存在を確認します。

```
C:¥pj>kubectl describe node minikube-m04
Name:               minikube-m04
...
Taints:             env=prod:NoExecute
...
```

- 右側のマニフェストを使用して、Podを作成してみますが、試しにオレンジ色枠内の「tolerations」の部分を削除してデプロイしてみると、PodがPending（保留中）状態になり、作成されないことを確認できます。
- また、Podの詳細情報を確認すると、
  「1つのノードには{env: prod}というTaintが設定されており、Podがそれを許容していません。
  残りの3つのノードはNode AffinityまたはSelectorの条件に一致しません。」
  というメッセージを確認できます。

```
C:¥pj>kubectl describe pod myapp
...
Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  11s   default-scheduler  0/4 nodes are available: 1 node(s) had untolerated taint {env: prod}, 3
node(s) didn't match Pod's node affinity/selector. preemption: 0/4 nodes are available: 4 Preemption is not helpful for
scheduling.
```

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    component: web
spec:
  nodeSelector:
    kubernetes.io/hostname: minikube-m04
  tolerations:
    - key: "env"
      operator: "Equal"
      value: "prod"
  containers:
    - name: nginx
      image: nginx:1.21.6
```

演習：TaintとToleration

- 一度Podを削除して、「tolerations」の部分を追加した上て、再度デプロイします。
- Taintがノードに設定されている場合でも、Podに「tolerations」の設定があるため、そのノードに問題なくスケジュールされることを確認します。

```
C:¥pj>kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP            NODE           NOMINATED NODE   READINESS GATES
myapp   1/1     Running   0          3s    10.244.4.18   minikube-m04   <none>           <none>
```

- 演習終了後、以下のコマンドを使用して、Taintを削除します。

  `kubectl taint nodes <ノード名> <key>-`

```
C:¥pj>kubectl taint nodes minikube-m04 env-
node/minikube-m04 untainted
```

- TaintとTolerationの主な利用例としては、「ノードを専用にする」や、「メンテナンス時の管理」などが挙げられます。

NoSchedule	• 特定のノードを専用に使いたい場合（例：GPU用Node）。この場合、そのノードを利用するアプリケーションに対して、Tolerationを設定します。 • ノードに性能関連の不具合があり、新規ノードのスケジューリングを避けたい場合
PreferNoSchedule	• 極力専用にしたいが、他のノードにリソースがなければスケジュールしてもよい場合
NoExecute	• ハードウェア障害やパフォーマンスの問題があるノードから、Podを退避させるために使用

Podの分散配置

- **KubernetesでPodをスケジュールする際に、必ずしもすべてのノードに均等に分散させるわけではありません。ノードの負荷やリソース状況を考慮するため、同一DeploymentのPodでも1台のノードに集中してしまうことはあり得ます。**
- **厳密にPodをノードに均等に分散させるには、topologySpreadConstraintsを使用します。**
- **topologySpreadConstraintsは、Podのspecの属性です。**
  - **topologyKey**：PodアフィニティのtopologyKeyと同じで、トポロジードメインを分割するために使用するラベルを指定します。この例ではホスト名を指定しているため、ノードごとにトポロジードメインが形成されます。
  - **maxSkew**： 1以上の整数で、最大の偏り許容値です。例えば2に設定した場合、任意の2つのトポロジードメインに含まれる対象Pod数の差は、最大でも2以内に収まるようスケジュールします。
  - **whenUnsatisfiable**：maxSkewを守れない場合の動作を指定します。
    DoNotSchedule：条件を満たせない場合にスケジューリングを拒否します。
    ScheduleAnyway：条件を満たせない場合でも可能な限りスケジューリングします。
  - **labelSelector**：スケジューリング対象のPodをラベルで指定します。

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-deployment
spec:
  replicas: 10
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: kubernetes.io/hostname
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: web
      containers:
      - name: nginx
        image: nginx:1.21.6
```

確認テスト1

Q1: Kubernetesにおいて、ラベルに関する記述のうち、正しいものはどれか。正しいものをすべて選択してください。

1. 1つのノードには、1つのラベルのみ付与できる。
2. ノードのラベルは削除できるが、置き換えることはできない。
3. ラベルはノードだけでなく、サービスやPodにも付与できる。
4. 「minikube.k8s.io」から始まるラベルは、minikubeが管理しているラベルである。

Q2: Podのマニフェストにある nodeSelector セクションについて、正しい記述はどれか。正しいものをすべて選択してください。

1. nodeSelector では、1つのラベルのみ指定可能である。
2. nodeSelector では、複数のラベルを指定可能である。
3. 指定したラベルを持つノードが存在しない場合、Podは任意のノードにスケジュールされる。
4. 指定したラベルを持つノードが存在しない場合、Podは Pending 状態になり、スケジュールされない。

確認テスト2

Q3: Podのアフィニティ（Affinity）設定について、正しい記述はどれか。正しいものをすべて選択してください。

1. podAffinity と podAntiAffinity を同時に使用することはできない。
2. アフィニティと nodeSelector を同時に使用することはできない。
3. アフィニティは、nodeSelector よりも細かい条件を指定でき、複雑なロジックでノードを選択できる。
4. nodeAffinity と podAffinity を同時に使用することができる。

Q4: Affinityの設定方法について正しい記述は以下のどれか？正しいものをすべて選択してください。

1. requiredDuringSchedulingIgnoredDuringExecutionセクションに記載される条件は、満たされなければスケジュールされない
2. preferredDuringSchedulingIgnoredDuringExecutionセクションに記載される条件は、満たされなくてもスケジュール可能
3. requiredDuringSchedulingIgnoredDuringExecutionや preferredDuringSchedulingIgnoredDuringExecutionは、nodeAffinityにのみ利用可能。podAffinityでは利用不可
4. preferredDuringSchedulingIgnoredDuringExecutionでは、weightを使用することで、どの条件を優先的に考慮するかを指定できる

確認テスト3

Q5: nodeAffinity指定の際に、「topologyKey: "abc"」という設定をした。この設定の意味は次のどれか?最も適切なものを選択してください。

1. 「abc」というラベルを持つノードは、すべて1つのトポロジードメインとしてみなされる
2. 「abc」というラベルを持つノードは、abcラベルの値ごとに異なるトポロジードメインとしてみなされる
3. 「abc」というドメイン名を持つノードは、すべて1つのトポロジードメインとしてみなされる
4. 「topologyKey」というラベルを持つノードは、値がabcであればnodeAffinityの設定対象としてみなされる

Q6: TaintとTolerationについて正しい記述は以下のどれか。正しいものをすべて選択してください。

1. Taintはノードに設定する
2. TolerationはPodに設定する
3. Taintをノードに設定すると、そのノード上に実行されているすべてのPodが停止される
4. TolerationをPodに設定すると、ノードのTaintがすべて無視される

確認テスト4

Q7: あるDeploymentに所属するPodには、topologySpreadConstraintsが設定されている。maxSkewは2で、topologyKeyはホスト名を指定している。また、whenUnsatisfiableはDoNotScheduleに設定されている。対象Podは、ノードAに3個、ノードBに4個、ノードCに5個それぞれすでに存在する。このDeploymentをスケールアウトし、さらに1個のPodを増やした場合、そのPodはどのノードにスケジュールされるか？最も適切なものを選択してください。

1. 必ずノードAにスケジュールされる
2. 必ずノードCにスケジュールされる
3. ノードAもしくはノードBにスケジュールされる
4. ノードAもしくはノードCにスケジュールされる
5. ノードBもしくはノードCにスケジュールされる

第七章 Kubernetesのリソース管理

第七章 Kubernetesのリソース管理

この章の内容

- **本章は、リソース管理について紹介します。**
  - Podレベルのリソース管理（requests/limitsの設定、QoSクラス）
  - クラスタレベルのリソース管理（Namespace、ResourceQuota、LimitRange）
  - 高度なリソース管理（metrics-server、HPAとVPA）

Podレベルのリソース管理

- **Kubernetesでは、「Request/Limit」という機能を使用して、コンテナのCPU/メモリリソースを制御します。**
  - Request（リクエスト）：コンテナ用に確保されるリソースの最低保証量です。
  - Limit（リミット）：コンテナが使用するリソースの最大量です。

CPU

CPUとは、コンピューティングの処理時間の単位で、**コア**で測定されます。

- 「cpu: 1」とは、CPUの1コアを全部使用する（そのコアの100%の時間を占用する）、という意味です。
- コアより小さい量を表すためにミリコア（m）を使用することができます。例えば「cpu: 500m」（「cpu: 0.5」と同じ意味）ならコアの半分を使用するという意味です。

メモリ

メモリは、**バイト**で測定されます。

メモリの単位では、2進数接頭辞（Mi, Gi, Ti...）と10進数接頭辞（M, G, T...）が利用可能ですが、**原則2進数接頭辞を使用**してください。違いは以下の例のようにです。

1 Gi = 1024 Mi　（2進数接頭辞）
1 G = 1000 M　（10進数接頭辞）

メモリは**非圧縮性リソース**です。つまり、CPUと同じように引き伸ばすことができません。プロセスが動作するのに十分なメモリを得られない場合、そのプロセスは強制終了されます。

```
resources:
  requests:
    cpu: 100m
    memory: 4Mi
  limits:
    cpu: 0.8
    memory: 20Mi
```

演習：Requests/Limitsの設定

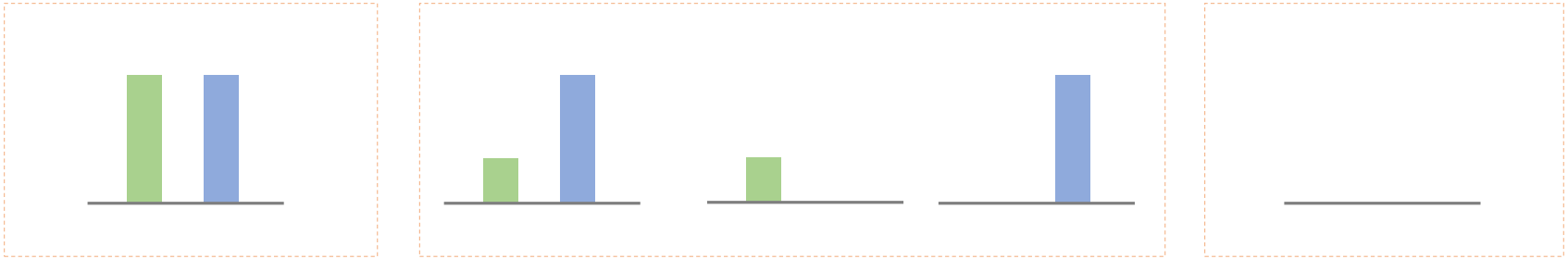
- では、実際にコンテナのRequests/Limitsを設定してみます。
- 右側のマニフェストをデプロイします。Pod作成完了後、Podの詳細情報を確認します。
- 「Containers」の詳細情報に、Requests/Limitsが記載されていることを確認します。

```
C:¥pj>kubectl describe pods myapp
Name:             myapp
Namespace:        default
Priority:         0
...
Containers:
  nginx:
...
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     800m
      memory:  20Mi
    Requests:
      cpu:        100m
      memory:     4Mi
    Environment:  <none>
```

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    component: web
spec:
  containers:
    - name: nginx
      image: nginx:1.21.6
      resources:
        requests:
          cpu: 100m
          memory: 4Mi
        limits:
          cpu: 0.8
          memory: 20Mi
```

QoSクラス

- **コンテナに設定されたリソースのRequests/Limitsは、PodのQoSクラス（QoS Class）を決定します。**
- **Kubernetesは、ノードのリソースが不足した際にPodを退避させる（削除する）判断を行うために、QoSクラスを使用します。**
- **QoSクラスの判断基準は以下の通りです。**
  - Guaranteed（保証クラス）：Pod内の全てのコンテナで、CPUやメモリのRequestsとLimitsが**等しく設定**されている場合。
  - Burstable（バースト可能クラス）：保証クラスの要件を満たさないが、少なくともPod内の1つのコンテナに対して、RequestsまたはLimitsが設定されている場合
  - BestEffort（ベストエフォートクラス）：Pod内の全てのコンテナで、RequestsもLimitsも**設定されていない**場合。



Guaranteed（保証クラス）　Burstable（バースト可能クラス）　BestEffort（ベストエフォートクラス）

QoSクラス

- PodのQoSクラスを確認するには、kubectl describeコマンドを使用します。
- 下記は、3つのPodの設定例と、そのPodのQoSクラスです。
  - 一番右側の例では、resourcesのセクションそのものを省略しても問題ありませんが、ここでは明示的にresourcesセクションに何も記述がないことを示しています。

```
resources:
  requests:
    cpu: 300m
    memory: 20Mi
  limits:
    cpu: 300m
    memory: 20Mi
```

```
resources:
  requests:
    cpu: 100m
    memory: 4Mi
  limits:
    cpu: 300m
```

```
resources: {}
```

```
C:¥pj>kubectl describe pods qos1
Name:             qos1
Namespace:        default
Priority:         0
Service Account:  default
...
QoS Class:                   Guaranteed
```

```
C:¥pj>kubectl describe pods qos2
Name:             qos2
Namespace:        default
Priority:         0
Service Account:  default
...
QoS Class:                   Burstable
```

```
C:¥pj>kubectl describe pods qos3
Name:             qos3
Namespace:        default
Priority:         0
Service Account:  default
...
QoS Class:                   BestEffort
```

クラスタレベルのリソース管理

- **大規模なKubernetesクラスタでは、個々のPodだけでなく、チーム間や環境間のリソース配分も重要な課題です。**
  - 複数のチームがPodを運用する場合、特定のチームのPodが暴走してシステム全体に影響を与えることはないでしょうか？
  - 開発環境、ステージング環境、本番環境が同じクラスタに共存する際、本番環境に十分なリソースを確保するにはどうすればよいでしょうか？
- **課題を解決するには、まずチームごと、環境ごとにリソースを論理的に分割するための仕組みを用意する必要があります。これは、Namespace（名前空間）です。**
  - コンテナのコア技術となる、LinuxカーネルのNamespace（名前空間）とは無関係です。
- **Kubernetesには、インストール直後からいくつかのデフォルトのNamespaceが用意されています。それぞれの役割は以下の通りです。**

Namespace	役割	利用方法
default	ユーザーがNamespaceを指定しない場合のデフォルト	テストや簡易リソース配置に利用
kube-system	Kubernetesシステムコンポーネント専用	ユーザーが利用すべきではない
kube-public	クラスタ全体で公開するリソース用	主に情報公開用（あまり使用されない）
kube-node-lease	ノードのハートビート管理。内部的なノード状態管理に利用	通常は利用しない

Namespaceの一覧を確認するには、以下のコマンドを使用します。
`kubectl get namespaces`
`kubectl get ns`　（nsはnamespacesのショートカットです）

```
C:¥pj>kubectl get ns
NAME              STATUS   AGE
default           Active   5d5h
kube-node-lease   Active   5d5h
kube-public       Active   5d5h
kube-system       Active   5d5h
```

Namespaceの作成

- Podやサービスなどほかのリソースと同様に、Namespaceはコマンドでの作成（命令型）とマニフェストでの作成（宣言型）をサポートしています。
- どちらも使用する場面があるので、両方紹介します。

コマンドを使用した作成・確認・削除

```
C:¥pj>kubectl create ns team-sales
namespace/team-sales created

C:¥pj>kubectl describe ns team-sales
Name:         team-sales
Labels:       kubernetes.io/metadata.name=team-sales
Annotations:  <none>
Status:       Active

No resource quota.

No LimitRange resource.

C:¥pj>kubectl delete ns team-sales
namespace "team-sales" deleted
```

マニフェストを使用した作成

```
apiVersion: v1
kind: Namespace
metadata:
  name: team-finance
```

Namespaceのクォーター管理

- **Namespace毎に、使用可能なリソースを制限することができます。**
- **「ResourceQuota」というAPIオブジェクトを使用して定義します。**
- **ResourceQuotaは、主に3種類のリソースに対して制限することができます。**

クォーターの種類	内容
計算資源に対するクォーター	Namespace内の合計CPUやメモリなどに対して上限を設定
ストレージ資源に対するクォーター	Namespace内の合計PVC数や容量に対して上限を設定
オブジェクト数のクォーター	各種オブジェクト（例：Pod数、ConfigMap数、Deployment数…）の最大数を設定

例えば、右側のResourceQuotaは、以下のように制限をかけています。
対象Namespaceは、「example-namespace」です。

- CPUの「request」の合計値は最大で10
- メモリの「limits」の合計値は最大で32GiB
- Pod数は、最大で50

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-quota
  namespace: example-namespace
spec:
  hard:
    requests.cpu: "10"
    limits.memory: "32Gi"
    pods: "50"
```

演習：クォーター管理

- それでは、演習を通じてResourceQuotaの動作を確認してみます。
- まずは、「team1」というNamespaceを作成します。ここでは、コマンドを使用して直接作成します。

```
C:¥pj>kubectl create ns team1
namespace/team1 created
```

- 次は、以下のマニフェストを使用して、クォーターを作成します。

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team1-quota
  namespace: team1
spec:
  hard:
    requests.cpu: 800m
    limits.memory: 400Mi
    requests.storage: 1Gi
    pods: "4"
```

- 作成後、Namespaceの詳細を確認し、クォーターが適用されていることを確認します。

```
C:¥pj>kubectl describe ns team1
Name:         team1
Labels:       kubernetes.io/metadata.name=team1
Annotations:  <none>
Status:       Active

Resource Quotas
  Name:             team1-quota
  Resource          Used  Hard
  --------          ---   ---
  limits.memory     0     400Mi
  pods              0     4
  requests.cpu      0     800m
  requests.storage  0     1Gi

No LimitRange resource.
```

演習：クォーター管理

- 続いて、以下のマニフェストを適用します。再度クォーターの状況を確認します。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
  namespace: team1
spec:
  containers:
    - name: nginx
      image: nginx:latest
      volumeMounts:
        - mountPath: /data
          name: pvc-volume
      resources:
        requests:
          cpu: 100m
          memory: 4Mi
        limits:
          cpu: 0.8
          memory: 20Mi
  volumes:
    - name: pvc-volume
      persistentVolumeClaim:
        claimName: web-pvc
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: web-pvc
  namespace: team1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

それぞれのmetadataに、Namespaceを記載することが重要です。
この記載がない場合、デフォルトのnamespaceでリソースが作成されることになります。

```
C:¥pj>kubectl describe ns team1
Name:         team1
Labels:       kubernetes.io/metadata.name=team1
Annotations:  <none>
Status:       Active

Resource Quotas
  Name:             team1-quota
  Resource          Used  Hard
  --------          ---   ---
  limits.memory     20Mi  400Mi
  pods              1     4
  requests.cpu      100m  800m
  requests.storage  1Gi   1Gi

No LimitRange resource.
```

- クォーターの状況を確認するには、もう1つのコマンドを利用できます：
  `kubectl describe quota -n <Namespace名>`

演習：クォーター管理

- では、requestsやlimitsが定義されていないPodを作成しようとすると、どうなるかを確認してみます。

webapp.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app-no-quota
spec:
  containers:
    - name: nginx
      image: nginx:latest
```

マニフェストにNamespaceの指定がない場合、kubectlコマンドの「-n」オプションを使用して、Namespaceを指定することができます。
Namespaceの指定がない場合、「default」というNamespaceで作成されます。

```
C:¥pj>kubectl apply -f webapp.yaml -n team1
Error from server (Forbidden): error when creating "simple2.yaml": pods "web-app-no-quota"
is forbidden: failed quota: team1-quota: must specify limits.memory for: nginx;
requests.cpu for: nginx

C:¥pj>kubectl apply -f webapp.yaml
pod/web-app-no-quota created
```

- ResourceQuotaでrequests/limitsのクォーターが設定された場合、名前空間内のPodにおいて、すべてのPodにそのリソースのrequests/limitsが設定されていないとエラーになります。
  理由は、ResourceQuotaが有効な場合、Kubernetesはリソースの消費量を管理するために各Podやコンテナのrequests/limits値を知る必要があるからです。

演習：クォーター管理

- では、クォーター制限を超えるリソースを作成しようとすると、どうなりますか？

large.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: large-web-app
  namespace: team1
spec:
  containers:
    - name: nginx
      image: nginx:latest
      resources:
        requests:
          cpu: 100m
        limits:
          memory: 390Mi
```

```
C:¥pj>kubectl apply -f large.yaml
Error from server (Forbidden): error when creating "large.yaml": pods "large-web-app" is
forbidden: exceeded quota: team1-quota, requested: limits.memory=390Mi, used:
limits.memory=20Mi, limited: limits.memory=400Mi
```

- 作成しようとしているPod自体は単独で見るとクォータ制限を超えていません。
  しかし、既存のリソースと合算すると制限を超えてしまうため、作成に失敗しました。
- 本演習は以上です。終了後は、使用したリソースを削除してクリーンアップを行ってください。

LimitRange：個々のオブジェクトのリソース制限

- ResourceQuotaを使用することで、1つのNamespace全体に対してリソース制限を設定できました。それでは、個々のオブジェクト（Podなど）が利用できるリソースについて、全体的なルールをどのように設定すればよいでしょうか？
- この場合は、「LimitRange」というAPIオブジェクトを使用します。
- LimitRangeは、Namespace内で作成される各リソース（PodやContainerなど）のリソース使用量に対して、最小値、最大値、またはデフォルト値を設定することができます。
- LimitRangeは、以下の表のとおり、5つの設定可能な項目を持っています。設定対象はコンテナ、Pod、PVCの3種類です。ただし、この3種類の対象に対して、5つの設定項目がすべて適用されるわけではありません。

設定可能な項目		設定項目の意味	コンテナ Container	ポッド Pod	PVC
制約（Constraint）	Min	最小値	設定可能	設定可能	設定可能
	Max	最大値	設定可能	設定可能	設定可能
	LimitRequestRatio	Limits / Requestsの比例	設定可能	設定可能	
デフォルト値（Default）	Default	limitsのデフォルト値	設定可能		
	DefaultRequest	requestsのデフォルト値	設定可能		

演習：LimitRange

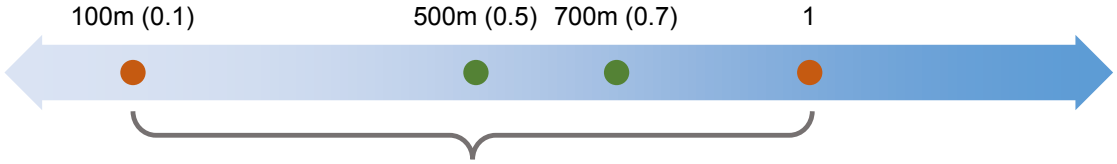
- まずは、以下のマニフェストを使用して、LimitRangeを作成します。

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-limit
  namespace: team1
spec:
  limits:
  - type: Container
    default:
      cpu: 700m
    defaultRequest:
      cpu: 500m
    max:
      cpu: "1"
    min:
      cpu: 100m
```

- 作成したLimitRangeを確認します。

```
C:¥pj>kubectl describe ns team1
Name:         team1
Labels:       kubernetes.io/metadata.name=team1
Annotations:  <none>
Status:       Active

No resource quota.

Resource Limits
 Type       Resource  Min   Max  Default Request  Default Limit  Max Limit/Request Ratio
 ----       --------  ---   ---  ---------------  -------------  -----------------------
 Container  cpu       100m  1    500m             700m           -
```

もし前の演習のResourceQuotaを削除していない場合、こちらにResourceQuotaの内容が表示されます。



左側のマニフェストの意味は次のとおりです。
個々のコンテナでは、CPUのrequestsおよびlimitsを100mから1の範囲で設定できます。
また、requestsを設定しなかった場合は500m、limitsを設定しなかった場合は700mに自動的に設定されます。

演習：LimitRange

- 次に、そのNamespaceにPodを作成します。Podの設定では、特にlimitsやrequestsを指定せず作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app-no-quota
  namespace: team1
spec:
  containers:
    - name: nginx
      image: nginx:latest
```

- 作成したPodを確認すると、デフォルトのlimitsやrequestsが設定されていることを確認できます。

```
C:¥pj>kubectl describe pod web-app-no-quota -n team1
Name:             web-app-no-quota
Namespace:        team1
Priority:         0
Service Account:  default
Node:             minikube-m02/192.168.49.3
...
 Ready:          True
    Restart Count:  0
    Limits:
      cpu:        700m
    Requests:
      cpu:        500m
    Environment:  <none>
...
```

- 試しにいったんPodを削除し、cpuのlimitsを1.2に設定してから、再度適用してみます。

```
 resources:
        limits:
          cpu: "1.2"
```

```
C:¥pj>kubectl apply -f simple3.yaml
Error from server (Forbidden): error when creating
"pod.yaml": pods "web-app-no-quota" is forbidden: maximum
cpu usage per Container is 1, but limit is 1200m
```

- 本演習は以上です。

metrics-server

- Kubernetesでは、より高度なリソース管理を可能にするには、「metrics-server」というアドオンを追加する必要があります。
- metrics-serverは、リソース使用量を収集・提供します。
- このアドオンの追加方法は、環境によって異なります。

方法1：minikubeでのみ利用可能

```
C:¥pj>minikube addons enable metrics-server
💡  metrics-server is an addon maintained by Kubernetes.
For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at:
https://github.com/kubernetes/minikube/blob/master/OWNERS
    ▪ Using image registry.k8s.io/metrics-server/metrics-
server:v0.7.2
🌟  The 'metrics-server' addon is enabled
```

方法2：Kubernetes for Docker Desktop及びminikubeで利用可能

```
C:¥pj>kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
service/metrics-server created
deployment.apps/metrics-server created
...
```

metrics-server

- どちらかの方法でmetrics-serverを追加後、metrics-serverが動作していることを確認します。
（このコマンドから推測できるように、metric-serverは「kube-system」で動作するDeploymentです）

```
C:¥pj>kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS      AGE
coredns-6f6b679f8f-f58tw           1/1     Running   1 (25h ago)   9d
...
metrics-server-d5865ff47-6z2k8     1/1     Running   0             44s
...
```

- metrics-server はインストール直後にノードや Pod のメトリクスを収集し始めますが、このプロセスには時間がかかります。

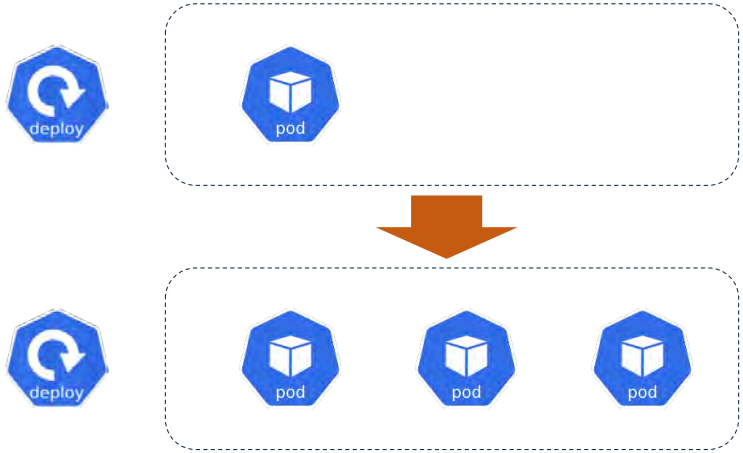
- 数分経過後、以下のコマンドでノードやPodのリソース使用状況を確認します（Podが存在しない場合は結果が表示されません）。「kubectl top」コマンドは、ノードやPodのリソース状況を確認するのに非常に便利です。

```
C:¥pj>kubectl top node
NAME           CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube       163m         1%     847Mi           5%
minikube-m02   32m          0%     235Mi           1%
minikube-m03   32m          0%     239Mi           1%
minikube-m04   23m          0%     143Mi           0%

C:¥pj>kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
some-app-fdcf7d7b6-r6912    0m           12Mi
```

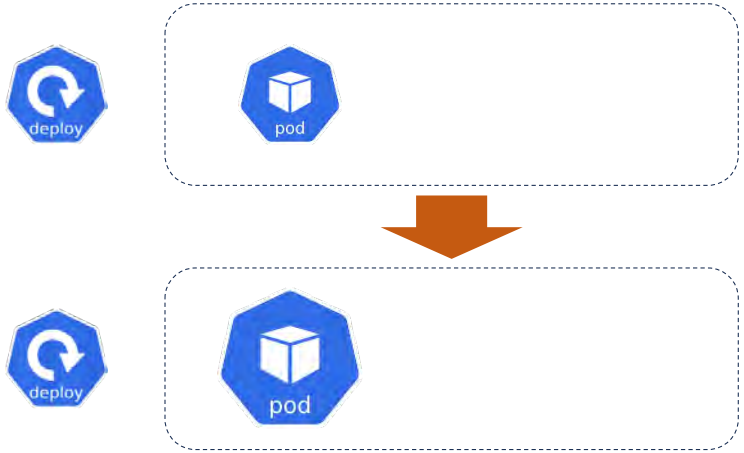
metrics-serverを活用した自動スケーリング

- 以前、Deploymentで複数のPodをデプロイする際に、手動でのスケーリング（拡大・縮小）を紹介しました。
- 自動的にリソース状況に応じてスケーリングするには、metrics-serverが必要になります。自動スケーリングには、2種類の方法があります。

HPA (Horizontal Pod Autoscaler)
水平にポッドをスケールする機能です。
Pod数の増減を行うことで負荷の変化に対応します。



VPA (Vertical Pod Autoscaler)
垂直にポッドをスケールする機能です。
Pod数の維持して、Podのスペックを調整することで
（例えばCPU/メモリの増減）負荷の変化に対応します。



HPAとVPA

HPAとVPAは、それぞれ単独で使用することも、併用することも可能です。

- ただし、HPA と VPA を同時に使用する場合、リソース管理の優先順位が複雑になるため、一般にはどちらか一方を選択することが推奨されます。

一般的には HPAのほうが VPA よりも多く使われています。

- Kubernetes の分散アーキテクチャと親和性が高く、スケールアウトを前提とした設計に適しています。
- VPA は動作中の Pod を再作成してリソースを調整する場合があり、システムに一時的なダウンタイムを引き起こす可能性があります。

HPAの動作概要

- HPAは、デフォルトでは、15秒おきにmetrics-serverより使用状況の情報を取得します。

演習：HPA

- 以下のマニフェストを使用して、DeploymentとHPAを作成します。

Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
      - name: demo-app
        image: nginx
        resources:
          requests:
            cpu: 100m
```

HPA

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: demo-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: demo-app
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- Deploymentを作成する際の注意点として、コンテナ定義には必ずCPUのrequestsを記載する必要があります。
  - HPAは、このrequestsの情報を基に必要なレプリカ数を計算します。
- HPAの設定では、「Resource」という種類を使用しており、CPU使用率のターゲットを50%に設定しています。
- 今回の演習では、初期設定として2レプリカのDeploymentを作成します。ただし、HPAは1～5の範囲で動的にスケーリングします。そのため、演習中、5分ほど経過した時点でCPUリソースが低い状態が続くと、HPAが発動してレプリカ数を1に減らす動作を確認できるはずです。

演習：HPA

- 今回の演習では「type: Resource」をメトリクスとして使用していますが、HPAでは以下の5種類のメトリクスをサポートしています。
- その中でも、シンプルで標準的なResourceは最も広く使用されています。

メトリクス種別	説明	使用する場面	設定例	特徴

Resource	Pod全体のCPUやメモリの使用量を基にスケーリング	CPUやメモリ使用率でスケーリングしたい場合	`name: cpu`
			または
			`name: memory`
				Kubernetes標準のリソースメトリクス。Metrics Serverが必要。

ContainerResource	特定のコンテナのCPUやメモリの使用量を基にスケーリング	Pod内の特定のコンテナのリソース使用量を基準にしたい場合	`container: nginx` `name: cpu`	特定コンテナにフォーカスしたリソースメトリクス。複数コンテナを持つPodで便利。
Pods	Pod単位で収集されるカスタムメトリクス（例: HTTPリクエスト数やアプリ特有の指標）を基にスケーリング	アプリケーション固有のメトリクス（例: 1 Podあたりのリクエスト数、成功したジョブ数）でスケーリング	`metric: http_requests_per_second` `averageValue: "10"`	カスタムメトリクスAPIが必要。Podごとの平均値を基にスケーリングする。

Object	特定のオブジェクト（例: Queueやカスタムリソース）のステータスに基づいてスケーリング	外部システムのステータスやKubernetes内のリソースステータスに応じてスケーリング	`metric: queue_length`
			`describedObject: { kind: Queue,`
			`name: my-queue }`
				特定のオブジェクトやリソース状態を基準にスケーリング。外部システムとも連携可能。

External	Kubernetes外部のメトリクス（例: クラウドプロバイダの監視メトリクス）を基にスケーリング	AWS CloudWatchやDatadogなど、外部監視ツールからのメトリクスを利用してスケーリング	`metric: external_queue_length` `targetValue: "100"`	External Metrics APIが必要。Kubernetes外のリソースメトリクスを扱える。

演習：HPA

- では、先ほどのマニフェストを適用します。適用直後に、Pod数が設定通り2であることを確認してください。また、HPAはCPU情報を直ちに取得できない場合があり、その際は<unknown>と表示されることがあります。

```
C:¥pj>kubectl get deployment
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
demo-app   2/2     2            2           16s

C:¥pj>kubectl get hpa
NAME           REFERENCE             TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
demo-app-hpa   Deployment/demo-app   cpu: <unknown>/50%   1         5         2          24s
```

- 数分後に再度確認します。Pod数が1に減少し、HPAがCPUの使用状況（この例では0%）を表示するようになります。

```
C:¥pj>kubectl get deployment
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
demo-app   1/1     1            1           5m26s

C:¥pj>kubectl get hpa
NAME           REFERENCE             TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
demo-app-hpa   Deployment/demo-app   cpu: 0%/50%   1         5         2          5m30s
```

演習：HPA

- また、DeploymentとHPAの詳細を確認することで、どのタイミングでどのような変更が行われたかを把握できます。

```
C:¥pj>kubectl describe deployment demo-app
Name:                   demo-app
Namespace:              default
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  14m    deployment-controller  Scaled up replica set demo-app-fdcf7d7b6 to 2
  Normal  ScalingReplicaSet  9m38s  deployment-controller  Scaled down replica set demo-app-fdcf7d7b6 to 1 from 2
```

レプリカ数を2から1にスケールダウン

```
C:¥pj>kubectl describe hpa demo-app-hpa
Name:                                                  demo-app-hpa
Namespace:                                             default
...
Conditions:
  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    ReadyForNewScale  recommended size matches current size
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a replica count from cpu resource utiliza      ercentage of request)
  ScalingLimited  True    TooFewReplicas    the desired replica count is less than the minimum replica count
Events:
  Type     Reason                        Age                   From                       Message
  ----     ------                        ----                  ----                       -------
  Warning  FailedGetResourceMetric       4m49s (x3 over 5m19s)  horizontal-pod-autoscaler  failed to get cpu utilization: ...
  Warning  FailedComputeMetricsReplicas  3m49s (x4 over 4m34s)  horizontal-pod-autoscaler  invalid metrics (1 invalid out of 1), ...
  Normal   SuccessfulRescale            18s                   horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

最初は、数回にわたってCPU使用率情報を取得できない旨の警告メッセージが表示されます。しかしその後、メトリクスがターゲットを下回っているため、レプリカ数を1に変更する旨のメッセージが表示されます。

- 本演習は以上です。

確認テスト1

Q1: Pod のマニフェストにおいて、「resources」セクションの「requests」に「cpu: 200m」と記載があった。この記述の意味として正しいものをすべて選択してください。

1. 200m とは 200 ミリ秒という意味で、requests として CPU 時間 200 ミリ秒を指定する
2. 200m とは 200 ミリコアという意味で、requests として 1 CPU コアの 20% を指定する
3. requests とは最低限必要なリソースの量であり、ノードで 200m のリソースを割り当てできない場合、Pod はスケジュールされない
4. requests とはコンテナが使用するリソースの最大量であり、ノード上で最大でも 200m のリソースを消費する

Q2: QoS クラスについての記述のうち、正しいものをすべて選択してください。

1. 複数のコンテナを持つ Pod 内で、1 つのコンテナに対して CPU とメモリの Requests と Limits を同じ値に設定した場合、Pod の QoS クラスが Guaranteed になる
2. 1 つのコンテナのみを持つ Pod において、CPU の Requests と Limits を同じ値に設定し、メモリは何も設定しなかった場合、Pod の QoS クラスは Burstable になる
3. Pod 内のすべてのコンテナに対して Limits を設定したが、Requests はまったく設定しなかった場合、QoS クラスは BestEffort になる
4. Requests や Limits をまったく設定していない場合、QoS クラスも設定されない
5. ノードのリソースが不足して Pod を退避させる際に、QoS クラスが使用される

確認テスト2

Q3: Kubernetes クラスタには、システム専用のネームスペースが存在し、通常のユーザーが利用すべきではないネームスペースがある。それは次のうちどれか。最も適切なものを選択してください。

1. default
2. kube-system
3. kube-public
4. kube-node-lease

Q4: 次の ResourceQuota を使用して制限できる例として、正しいものをすべて選択してください。

1. 最大 Pod 数を 100 個までに制限する
2. 最大ノード数を 8 までに制限する
3. CPU の Requests の合計を 20 までに制限する
4. メモリの Limits のそれぞれの Pod で設定可能な値を 200Mi までに制限する
5. PVC の合計数を 100 までに制限する

確認テスト3

Q5: 次のうち、LimitRange を使用して設定できるものはどれか。正しいものをすべて選択してください。

1. Pod の CPU 使用量の最大値
2. PVC のストレージ容量の最小値
3. Pod のメモリの requests のデフォルト値
4. PVC のストレージ容量の最大値と最小値の比例

Q6: 「kubectl top」コマンドについての記述のうち、正しいものをすべて選択してください。

1. ノードや Pod のリソース状況を確認できる
2. ResourceQuota や LimitRange の設定状況を確認できる
3. metrics-server を必要とする
4. AWS などのクラウド環境でしか利用できない

確認テスト4

Q7: HPA と VPA に関する記述のうち、正しいものを
すべて選択してください。

1. HPA は Pod の CPU やメモリを増減することで
負荷の変化に対応する
2. VPA は Pod 数の増減を行うことで負荷の変化
に対応する
3. HPA の方が広く使用されている
4. HPA では、最大 / 最小レプリカ数を定義する

第八章 Kubernetesの可用性管理

第八章 Kubernetesの可用性管理

この章の内容

本章は、Kubernetesの可用性管理を紹介します。

- Probeの設定と利用

  Probeは、Kubernetesにおいてヘルスチェックの重要な手段です。

- Kubernetesのコントローラー

  Kubernetesは、コントローラーという仕組みを利用してPodを管理し、可用性を高めています。
  Kubernetesで主に利用される3種類のコントローラーを紹介します。

  - Deployment/ReplicaSet
  - StatefulSet
  - DaemonSet

Probe（プローブ）の設定と利用

- 可用性管理には、死活監視（ヘルスチェック）の情報を利用しなければなりません。
- Kubernetesクラスタにおいて、主にProbe（プローブ）という仕組みを利用して、死活監視などを行います。
- Probeの設定対象は、Podではなく、個々のコンテナです。
- Kubernetesには3種類のProbeがあります。どのProbeも、4種類の方法から1つ選択できます。

Probe	機能
Liveness Probe	**存続性チェック**。いわゆる「死活監視」に該当するProbeで、対象コンテナが正常に動作しているかを確認します。
Readiness Probe	**準備性チェック**。対象コンテナが「リクエストを受け付けられる状態かどうか」を確認します。
Startup Probe	**起動チェック**。コンテナが「起動に成功したかどうか」を確認します。

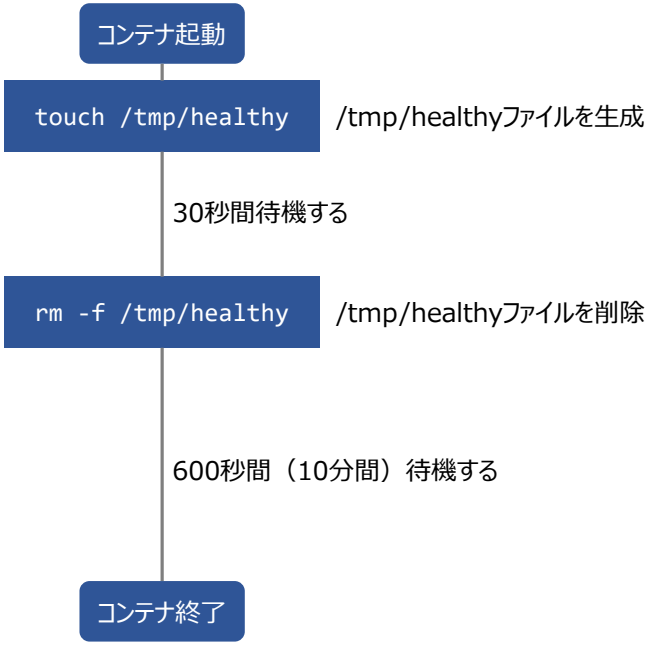
確認方法	確認内容
httpGet	指定されたHTTP/HTTPSエンドポイントにリクエストを送り、応答コードで判定します。
tcpSocket	指定されたポートに対してTCP接続を試み、接続可能かどうかで判定します。
exec	コンテナ内で指定されたコマンドを実行し、その終了コードが0の場合を成功とみなします。
grpc	gRPCサービスの特定のメソッドにリクエストを送り、その応答で判定します。 （Kubernetes v1.24以降で正式サポート）

演習：Probeの利用

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
      failureThreshold: 2
      successThreshold: 1
```

この部分がProbeの定義です。
詳細は次に説明します。

- このコンテナで実行されるコマンドを時系列で見てみます。



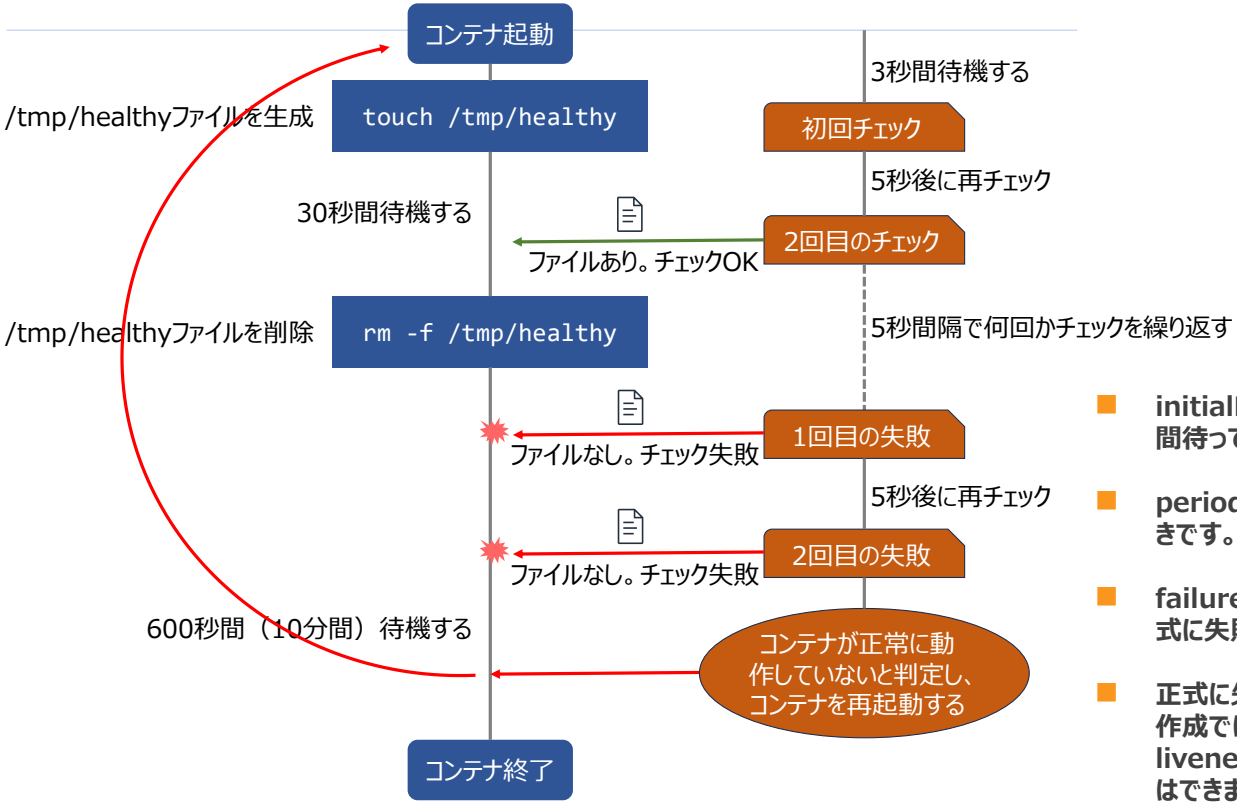
演習：Probeの利用

- このセクションの名称は「livenessProbe」で、存続性チェック（Liveness Probe）を意味します。
- livenessProbeの下には、まず「exec」があります。これは、チェック方法としてコンテナ内で指定されたコマンドを実行し、その終了コードを確認する方法を指定しています。
- 続いて、その次に記載されている4つの設定項目は、それぞれ以下の通りです。

```
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
      failureThreshold: 2
      successThreshold: 1
```

設定項目	設定内容	デフォルト値
initialDelaySeconds	Probe（チェック）が最初に実行されるまでの待機時間（Pod起動後の秒数）。	0秒
periodSeconds	Probe（チェック）を実行する間隔（秒単位）。	10秒
failureThreshold	Probe（チェック）が最終的に「失敗」とみなされるまでの連続失敗回数。	3回
successThreshold	Probe（チェック）が最終的に「成功」とみなされるまでの連続成功回数。	1回（livenessProbeの場合は、1のみ設定可能）

演習：Probeの利用




```
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
      failureThreshold: 2
      successThreshold: 1
```

- initialDelaySecondsが3なので、コンテナの起動後にまず3秒間待ってからProbe（チェック）を開始します。
- periodSecondsが5なので、Probe（チェック）の間隔は5秒おきです。
- failureThresholdが2なので、2回連続して失敗した時点で正式に失敗と判定します。
- 正式に失敗と判定された場合、コンテナを再起動します（Podの再作成ではない点に注意してください）。この挙動はlivenessProbeの仕様に基づくものであり、カスタマイズすることはできません。

演習：Probeの利用

- では、実際にマニフェストを適用して、想定通りに動作するかを確認します。
- マニフェストを適用後、直ちにPodのステータスを確認します。Eventsの部分に注目します。

```
C:¥pj>kubectl describe pod liveness
Name:             liveness-exec
Namespace:        default
...
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  13s   default-scheduler  Successfully assigned default/liveness-exec to minikube-m03
  Normal  Pulling    12s   kubelet            Pulling image "registry.k8s.io/busybox"
  Normal  Pulled     10s   kubelet            Successfully pulled image "registry.k8s.io/busybox" in 2.121s...
  Normal  Created    10s   kubelet            Created container liveness
  Normal  Started    10s   kubelet            Started container liveness
```

- Pod/コンテナは、10秒前から起動しているため、この時点で対象ファイルは存在していると考えられます。
- livenessProbeの初回チェックが完了しており、その結果は「成功」であると推測されます。
- しかし、Podのイベントには、この成功結果が記録されていない点に注意が必要です。

演習：Probeの利用

- Podが起動してから、約40秒～50秒経過後再度Podのステータスを確認します。

```
C:¥pj>kubectl describe pod liveness
Name:             liveness-exec
Namespace:        default
...
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  52s                default-scheduler  Successfully assigned default/liveness-exec to minikube-m03
  Normal   Pulling    52s                kubelet            Pulling image "registry.k8s.io/busybox"
  Normal   Pulled     50s                kubelet            Successfully pulled image "registry.k8s.io/busybox" in 2.121s...
  Normal   Created    50s                kubelet            Created container liveness
  Normal   Started    50s                kubelet            Started container liveness
  Warning  Unhealthy  13s (x2 over 18s)  kubelet            Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory
  Normal   Killing    13s                kubelet            Container liveness failed liveness probe, will be restarted
```

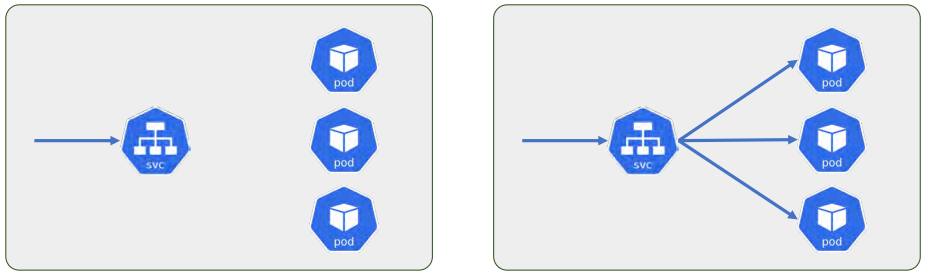
2回の失敗があったため、Unhealthyと判定されています。

コンテナは、再起動される旨のメッセージが記録されています。

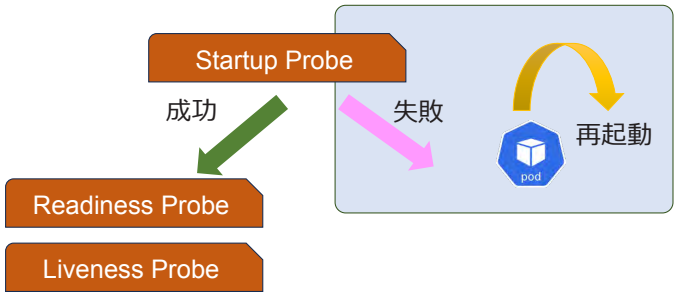
- デフォルト設定では、livenessProbeがコンテナを再起動する回数に制限はありません。そのため、この演習の例では、再起動が永遠に繰り返されることになります。
- この演習は以上で終了です。

Probeの比較

- Liveness Probeのほかに、Readiness ProbeとStartup Probeもあります。



Readiness Probe
Probeが成功するまで、PodはServiceから除外されます。Probeが成功すると、Serviceに再び追加され、トラフィックが振り分けられるようになります。アプリケーションの初期化や一時的なエラーをトラフィックに反映しないための仕組みですので、トラフィック制御に重要です。



Startup Probe
Probe失敗時にはコンテナが再起動されます。
また、Probe成功するまではLiveness ProbeやReadiness Probeが適用されることはありません。起動時間が長いアプリケーション向けに設定し、Liveness Probeが起動中に失敗するのを防ぐために使用します。

	Liveness Probe	Readiness Probe	Startup Probe
目的	コンテナが正常に動作を続けているかを確認	コンテナがトラフィックを処理する準備ができているかを確認	コンテナが正しく起動したかを確認
動作タイミング	コンテナ起動後に継続して動作	コンテナ起動後に継続して動作	コンテナ起動時に1回限り（コンテナ再起動後に再実行される）
成功時や失敗時の挙動	Probeに失敗したときは、コンテナを再起動する	Probeが成功するまでに、PodがServiceから除外される。成功するとServiceからトラフィックが振り分けられるようになる	Probe失敗時は、コンテナを再起動する。また、成功するまでは、Liveness ProbeやReadiness Probeは適用されない。

第八章 Kubernetesの可用性管理

Kubernetesのコントローラー

- Kubernetesにおけるコントローラーは、クラスタ内のリソース（例: PodやNodeなど）の状態を継続的に監視し、「現在の状態」を「望ましい状態」に保つ役割を担うコンポーネントです。
- 第3章で学習したDeploymentとReplicaSetは、Kubernetesのコントローラーの一種です。これらは、アプリケーションの可用性を確保する上で非常に有効です。
- Kubernetesには、さらに2種類の主要なコントローラーがあります：StatefulSetとDaemonSetです。それでは、この2種類のコントローラーの機能を確認してみましょう。

DeploymentとReplicaSet

ReplicaSetはレプリカ数の管理とPodの自己修復を担当し、DeploymentはReplicaSetを使用してアプリケーションのデプロイ、ローリングアップデートやロールバックを担当します。

StatefulSet

StatefulSetは、状態を持つアプリケーションを管理するために設計されたコントローラーです。永続ストレージの保証や起動順序の保証など、データベースなどに適しています。

DaemonSet

DaemonSetは、クラスタ内の各ノードに特定のPodを1つずつ確実に配置するための仕組みを提供します。各ノードでLinuxのデーモンのように動作するためログ収集や監視などに利用します。

StatefulSet

- StatefulSetは、「ステートフル」なPodを管理するためのコントローラーです。
- StatefulSetは、DeploymentやReplicaSetと同様に複数のPodをデプロイする仕組みですが、それぞれ異なる特徴を持っています。

	Deployment/ReplicaSet	StatefulSet
デプロイされるPodの名前	Deploymentを使用して作成されたPodには、「myapp-5c6bdd9bcb-s45nd」のように、**ランダムな文字列を含む名前**が付与されます。	StatefulSetを使用して作成されたPodには、「myapp-0」「myapp-1」のように、**順序付きの一意な名前**が付与されます。
順序制御	Podの作成・更新・削除に順序は関係ありません。	Podの作成・更新・削除は順序を維持します。作成は必ず昇順（myapp-0→myapp-1の順番）で、更新と削除は必ず降順（作成と逆の順番）で行われます。
ストレージ	PVCを使用して、Podにストレージをアサインすることは可能ですが、テンプレートを使用するのですべてのPodが**同じPVCを共有する**ことになります。	専用の「volumeClaimTemplates」を利用して、それぞれのPodに**異なるPVCを割り当てる**ことができます。
ネットワークIDの安定性	原則として、**個々のPodに直接アクセスすることはできず**、Service（ClusterIP）を経由してアクセスする形になります。	各Podには安定したネットワークID（ホスト名やDNS名）が付与されるため、他のアプリケーションやPodはStatefulSetによって管理されている特定の**Podに直接接続することが可能**です。

演習：StatefulSet

- それでは、演習を通じてStatefulSetの動作を確認してみます。
- 右側にあるマニフェストを使用します。このYAMLファイルの構造は、Deploymentと似ていることがわかります。

この部分は、Deploymentの記述方法とほぼ同様です。「replicas」とありますが、ReplicaSetを使用しているわけではありません。あくまでも「Pod数」です。

Podごとに異なるPVCをアサインするための仕組みです。この例では、RWOアクセスモードの、1GBのストレージをそれぞれのPodに対してPVCをアサインすることを意味します。

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "web"
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
  volumeClaimTemplates:
  - metadata:
      name: userdata
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

演習：StatefulSet

- マニフェストを適用した後、直ちにPodの作成状況を確認します。

```
C:¥pj>kubectl apply -f statefulset.yaml
statefulset.apps/web created

C:¥pj>kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          2s
web-1   0/1     Pending   0          1s

C:¥pj>kubectl get pods
NAME    READY   STATUS              RESTARTS   AGE
web-0   1/1     Running             0          4s
web-1   0/1     ContainerCreating   0          3s

C:¥pj>kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          5s
web-1   1/1     Running   0          4s
web-2   0/1     Pending   0          1s

C:¥pj>kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          10s
web-1   1/1     Running   0          9s
web-2   1/1     Running   0          6s
web-3   1/1     Running   0          4s
```

このように、Podが順番に作成されることが確認できます。

また、Podの名前はランダムではなく、順序付きの一意の名前であることも確認できます。

演習：StatefulSet

- StatefulSetで作成されたPodは、Pod名だけでなく、Podに紐づけられたPVCの名前も固定され一意となっています。

```
C:¥pj>kubectl get pvc
NAME             STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
userdata-web-0   Bound    pvc-f381fc09-eab5-4374-b77e-39e04126df62   1Gi        RWO            standard       <unset>                 51m
userdata-web-1   Bound    pvc-f656616e-7ce6-45bc-9799-41e4b4b6144a   1Gi        RWO            standard       <unset>                 51m
userdata-web-2   Bound    pvc-518b17b5-0aa6-4e57-b133-4eea061a9233   1Gi        RWO            standard       <unset>                 51m
userdata-web-3   Bound    pvc-57ddd5c6-e997-4a8b-a494-65a32c8e23b6   1Gi        RWO            standard       <unset>                 51m
```

PVC名は、
{volumeClaimTemplateの名前}-{StatefulSetの名前}-連番
のフォーマットで固定されています。

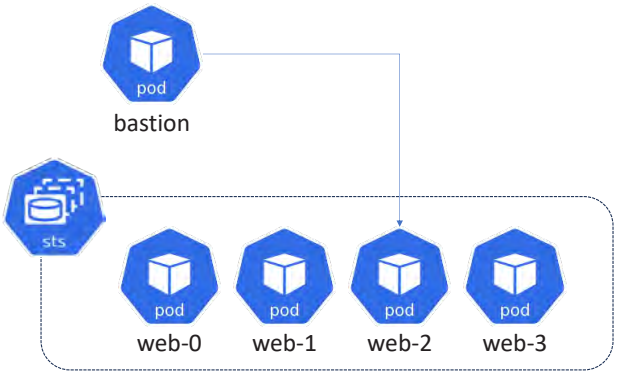
- StatefulSetの更新を確認してみます。マニフェストにあるnginxのバージョンを、1.19から1.21に変更して、再度デプロイしてみます。

```
C:¥pj>kubectl apply -f statefulset.yaml
statefulset.apps/web created

C:¥pj>kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
web-0     1/1     Running   0          8s
web-1     1/1     Running   0          12s
web-2     1/1     Running   0          16s
web-3     1/1     Running   0          20s
```

AGE欄を確認すると、バージョンアップの順番が作成時とは逆順であることがわかります。

演習：StatefulSet

- StatefulSetのPodは、名前とストレージボリュームが固定で一意ですが、IPアドレスは固定ではありません。Podが削除されて再作成される際には、IPアドレスが変更される可能性があります。
- Deploymentの場合は、前段にServiceを配置し、ClusterIPを使用して固定IPアドレスを確保できます。ClusterIPは、トラフィックを複数のPodに対して負荷分散します。
  - しかし、これはすべてのPodが同一の役割を持つDeploymentで可能な仕組みです。一方、StatefulSetでは各Podが異なる役割や状態を持つため、ランダムに負荷分散することはできません。
  - そのため、StatefulSetでは、IPアドレスが変更しても、各Podに直接アクセスできる仕組みを提供しています。それが、DNSによる名前解決です。
- 次の演習では、踏み台サーバーを作成し、そこからStatefulSetのPodにアクセスします。踏み台Podのマニフェストは右側の通りです。



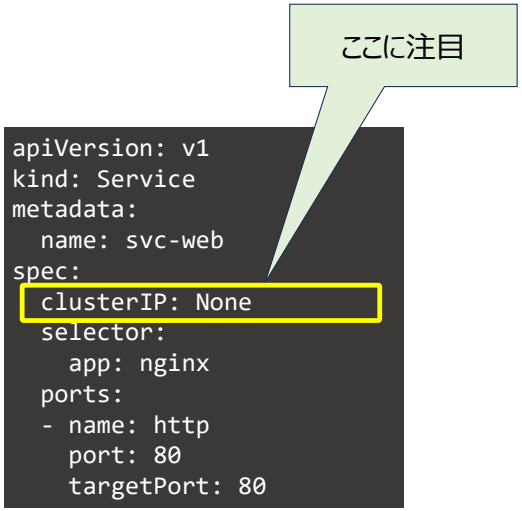

```
apiVersion: v1
kind: Pod
metadata:
  name: bastion
spec:
  containers:
    - name: busybox
      image: busybox
      stdin: true
      tty: true
```

演習：StatefulSet

- 踏み台をデプロイ後、踏み台からnslookupを実行して名前解決を試みます。

```
C:¥pj>kubectl exec -it bastion -- nslookup web-2.svc-web.default.svc.cluster.local
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find web-2.svc-web.default.svc.cluster.local: NXDOMAIN

** server can't find web-2.svc-web.default.svc.cluster.local: NXDOMAIN
```

- 名前解決に失敗しました。なぜでしょうか？
- これは、Podの名前解決には「Headless Service」という特殊なServiceが必要だからです。
  - 「Headless Service」は、「clusterIP: None」を指定した、IPアドレスの持たない、負荷分散機能がない特殊なServiceです。
  - 各Podに固有のDNS名を付与し、特定のPodへの直接アクセスを可能にします。特にStatefulSetのような、状態を持つアプリケーションで安定したネットワークIDを必要とする場合に利用されます。
- では、Headless Serviceを作成します。マニフェストは右側の通りです。



演習：StatefulSet

- 再度踏み台からnslookupを実行して名前解決を試みます。

```
C:¥pj>kubectl exec -it bastion -- nslookup web-2.svc-web.default.svc.cluster.local
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:   web-2.svc-web.default.svc.cluster.local
Address: 10.244.0.14
```

- 問題なく名前解決ができたため、これで名前を使用してクラスタ内の通信が可能になります。

- StatefulSetを削除します。削除完了後、PVCを再度確認します。
  - StatefulSetでは、Podが削除されても、そのストレージ保持されます。同じ名前で再作成時に同じデータにアクセス可能です。
  - 本演習は以上です。

```
C:¥pj>kubectl delete -f statefulset.yaml
statefulset.apps "web" deleted

C:¥pj>kubectl get pvc
NAME             STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
userdata-web-0   Bound    pvc-f381fc09-eab5-4374-b77e-39e04126df62   1Gi        RWO            standard       <unset>                 4h11m
userdata-web-1   Bound    pvc-f656616e-7ce6-45bc-9799-41e4b4b6144a   1Gi        RWO            standard       <unset>                 4h11m
userdata-web-2   Bound    pvc-518b17b5-0aa6-4e57-b133-4eea061a9233   1Gi        RWO            standard       <unset>                 4h11m
userdata-web-3   Bound    pvc-57ddd5c6-e997-4a8b-a494-65a32c8e23b6   1Gi        RWO            standard       <unset>                 4h11m
```

DaemonSet

- **各ノードに1つのPodを確実に実行したい場合は、DaemonSetを使用します。**
- **演習でDaemonSetの動作を確認しましょう。右側のマニフェストを使用します。**
  - 全体の記述方法は、ほとんどDeploymentと同じです。
  - ただし、Deploymentとの主な相違点として、レプリカ数の記載がない点が挙げられます。DaemonSetは各ノードに1つのPodを実行するため、レプリカ数は自動的にノード数と一致します。

DaemonSetの演習には、複数ノードが必要です。
minikubeの場合は、複数ノードのクラスタを作成しておいてください。

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemon
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

演習：DaemonSet

- マニフェストを適用します。Podとノードを確認します。

```
C:¥pj>kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
daemon-f4wsq   1/1     Running   0          31s   10.244.0.3   minikube       <none>           <none>
daemon-lgf7d   1/1     Running   0          31s   10.244.2.2   minikube-m03   <none>           <none>
daemon-rtr69   1/1     Running   0          31s   10.244.1.2   minikube-m02   <none>           <none>

C:¥pj>kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
minikube       Ready    control-plane   11d   v1.31.0
minikube-m02   Ready    <none>          11d   v1.31.0
minikube-m03   Ready    <none>          11d   v1.31.0
```

それぞれのノードに、確実に1つのPodが作成されています。

演習：DaemonSet

- minikubeのノードを1つ追加します。

```
C:¥pj>minikube node add
😄  Adding node m04 to cluster minikube as [worker]
👍  Starting "minikube-m04" worker node in "minikube" cluster
🚜  Pulling base image v0.0.45 ...
🔥  Creating docker container (CPUs=2, Memory=2666MB) ...
❗  Failing to connect to https://registry.k8s.io/ from inside the minikube container
💡  To pull new external images, you may need to configure a proxy:
https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
📦  Preparing Kubernetes v1.31.0 on containerd 1.7.21 ...
🔎  Verifying Kubernetes components...
🎉  Successfully added m04 to minikube!
```

- 再度Pod情報を確認します。

```
C:¥pj>kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP           NODE           NOMINATED NODE   READINESS GATES
daemon-45cf9   1/1     Running   0          69s     10.244.3.2   minikube-m04   <none>           <none>
daemon-f4wsq   1/1     Running   0          4m32s   10.244.0.3   minikube       <none>           <none>
daemon-lgf7d   1/1     Running   0          4m32s   10.244.2.2   minikube-m03   <none>           <none>
daemon-rtr69   1/1     Running   0          4m32s   10.244.1.2   minikube-m02   <none>           <none>
```

新しいPodが追加されていること
を確認できます。

確認テスト1

Q1: Probe（プローブ）は、次のどの方法を使用してコンテナを監視するか？当てはまるものをすべて選択してください。

1. HTTP/HTTPS の GET メソッドを使用する。
2. DNS のレコードを調べる。
3. コンテナ内でコマンドを実行し、戻り値を判定する。
4. kubectl コマンドを使用して確認する。

Q2: Kubernetes を使用して Web サービスを提供している。コンテナが Web リクエストを受け付けられるようになるまで、Service からトラフィックを振り分けないようにしたい。この場合、使用すべき最も適切な Probe はどれか？

1. Liveness Probe
2. Readiness Probe
3. Startup Probe
4. Web Probe

確認テスト2

Q3: Startup Probe の機能についての記述のうち、正しいものをすべて選択してください。

1. コンテナ起動時に 1 回のみ動作する。
2. コンテナが起動されるまで何回も動作する。
3. コンテナの起動を加速するために使用する。
4. Liveness Probe や Readiness Probe のコンテナ起動中の失敗を防ぐために使用する。

Q4: StatefulSet についての記述のうち、正しいものをすべて選択してください。

1. StatefulSet に含まれる Pod には、すべて同じ名前が付与される。
2. Pod の作成や更新の順番は一定である。
3. それぞれの Pod に異なる PersistentVolumeClaim (PVC) を割り当てることができる。
4. 通常、ClusterIP を経由して Pod にアクセスする。

確認テスト3

Q5: StatefulSet に含まれる Pod にアクセスするために必要な Service について、最も適切なものを選択してください。

1. LoadBalancer の Service を使用する必要がある。
2. NodePort の Service を使用する必要がある。
3. Ingress を使用する必要がある。
4. Headless Service を使用する必要がある。

Q6: あるアプリケーションは 4 つの Pod から構成されている。このアプリケーションを 3 つのノードで構成する Kubernetes クラスタにデプロイする際に、耐障害性を高めるために可能な限り Pod を分散させたい。この場合、最も適切なコントローラーはどれか？

1. Deployment
2. ReplicaSet
3. StatefulSet
4. DaemonSet

第九章 パブリッククラウド上のコンテナサービス①

第九章 パブリッククラウド上のコンテナサービス①

この章の内容

- **本章は、パブリッククラウド上のコンテナサービスをご紹介します。**
- **マイクロソフトAzureのコンテナサービスにも触れますが、主にAWSクラウドのEKSを中心に解説します。なお、時間の制約により、各種AWSサービスそのものの詳細な説明は割愛させていただきます。**
- **本章では、主に以下のトピックを取り上げます。**
  - Azure上の手軽に使えるコンテナサービス：ACI
  - Amazon EKSのクラスタ構築
  - EKSのロードバランサー連携
  - アプリケーション公開とIngress

パブリッククラウド上のコンテナサービス

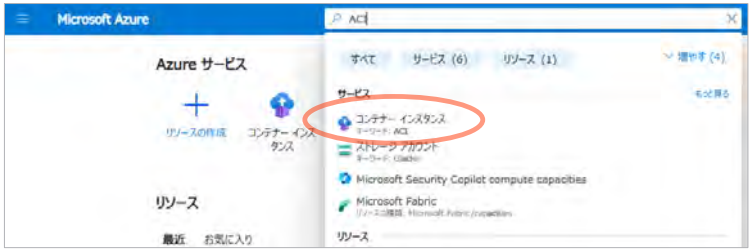
- **すべて主要なパブリッククラウドプロバイダー（AWS、Azure、Google Cloudなど）は、コンテナを効率的に活用するためのサービスを提供しています。**
- **これらのコンテナサービスは、大きく2つの主要なカテゴリに分けられます。**
  - Kubernetesを基盤とするコンテナサービス：オーケストレーションにKubernetesを使用します。kubectlコマンドも利用可能です。
  - 独自のオーケストレーション基盤を採用したコンテナサービス：簡易的なコンテナ実行環境を提供するものや、クラウド側とより深く連携したサービスです。

	Kubernetesベースのコンテナサービス	独自/その他コンテナサービス
AWS	Amazon Elastic Kubernetes Service (EKS)	Amazon Elastic Container Service (ECS)
Azure	Azure Kubernetes Service (AKS)	Azure Container Instances (ACI)
GCP（Google）	Google Kubernetes Engine (GKE)	

- **AWSの場合、さらに実行環境・基盤として、EC2もしくはFargateが選択可能です。**
- **この組み合わせにより、以下の4通りの運用が可能です。**
  - ECS x EC2
  - ECS x Fargate
  - EKS x EC2
  - EKS x Fargate

Azure ACI

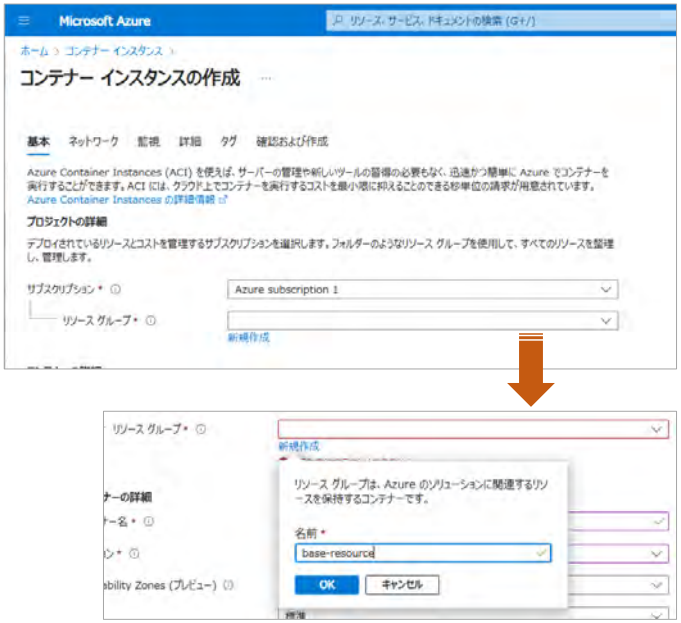
- Azure Container Instances（ACI）は、マイクロソフトAzure上で提供されるコンテナサービスです。
- Kubernetesのような高度な機能は備えていませんが、複雑な管理や設定が不要で、単純なワークロードやテスト環境に適しています。また、AWSのECSよりもシンプルで手軽に利用できる点が特徴です。
- それでは、画面を通じてACIの利用イメージをご紹介します。



①Azureポータル画面で「ACI」を検索し、「コンテナ インスタンス」を選択して、ACIの画面を開きます。

②コンテナ インスタンスの画面で「コンテナ インスタンスの作成」をクリックします。

Azure ACI



③コンテナ インスタンスの作成画面で、新しいリソースグループを作成し、それを選択します。

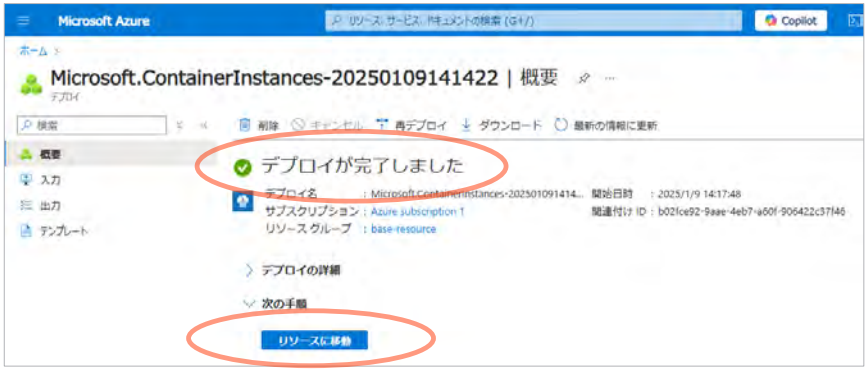
④その他の必要な情報を入力し、「確認および作成」をクリックします。

コンテナ名

東日本（Japan East）を選択

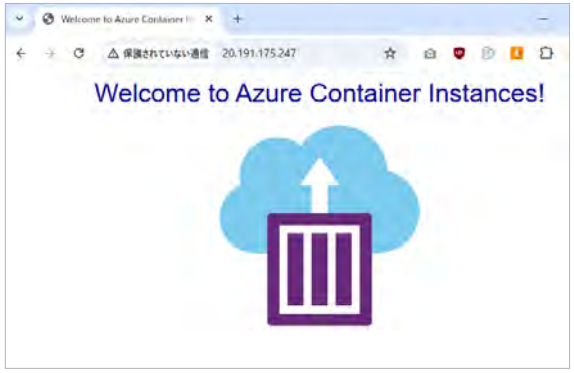
aci-helloworldのイメージを使用

Azure ACI

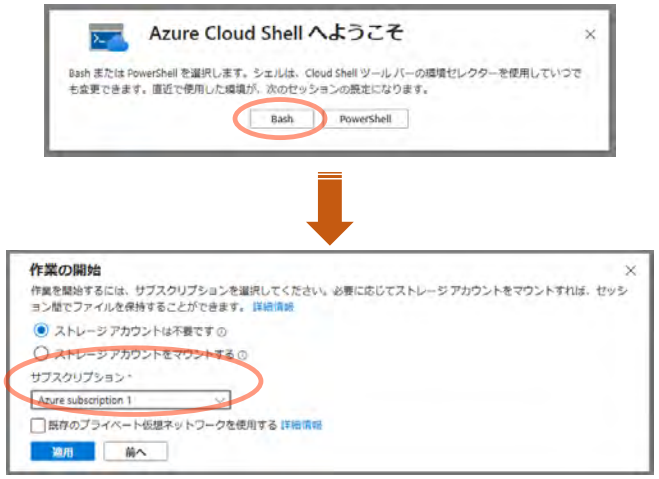


⑤作成が開始され、しばらくするとステータスが「デプロイが完了しました」に変わります。「リソースに移動」をクリックします。

⑥コンテナの情報が表示されます。IPアドレス（Public）の欄にあるIPアドレスをコピーしてブラウザでアクセスすると、右側のような画面が表示されます。



Azure ACI



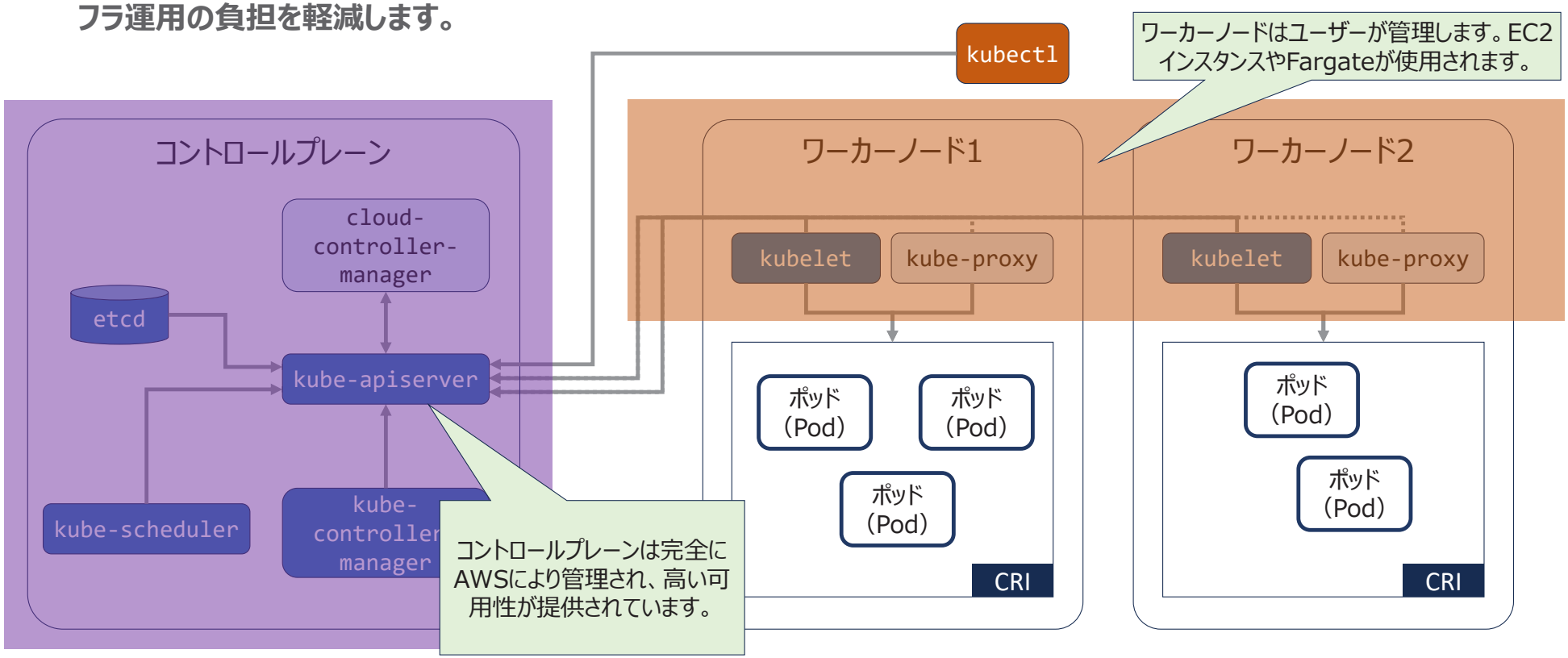
⑦Azure ACIは、管理画面だけでなく、コマンドラインを利用したコンテナ管理もサポートしています。
ここでは、Cloud Shellを使用してCLI操作を体験します。画面右上のCloud Shellアイコンをクリックします。初めてCloud Shellを起動する場合は、シェルの選択やストレージアカウントの選択画面が表示されます。この場合、Bashを使用し、その他の設定はデフォルト値を選択してください。

```
user [ ~ ]$ az container list -o table
Name      ResourceGroup    Status     Image                                                IP:ports           Network    CPU/Memory       OsType    Location
--------  ---------------  ---------  ---------------------------------------------------  -----------------  ---------  ---------------  --------  ----------
test-aci  base-resource    Succeeded  mcr.microsoft.com/azuredocs/aci-helloworld:latest  20.191.175.247:80  Public     1.0 core/1.5 gb  Linux     japaneast
```

⑧Cloud Shellでは、上記のコマンドを使用してコンテナの一覧を表示します。
このコマンドは、Dockerの「docker ps」やKubernetesの「kubectl get pods」と似た機能を持っています。

Amazon EKS

- AWS上のKubernetesのマネージドサービスです。AWSがクラスタのセットアップ、スケーリング、管理を支援し、インフラ運用の負担を軽減します。



Amazon EKSクラスタの構築

- ローカルPCでは、minikubeやDocker Desktopを使用して、テスト用のKubernetesクラスタを構築しました。
- Amazon EKSでは、構築方法をいくつか提供しています。

方法１：AWSマネージメントコンソールを利用する方法

GUI操作で簡単に設定可能なため、初心者でも利用しやすい点が特徴です。複雑なCLIコマンドを使用せずにEKSの基礎構築を迅速に行えるため、主に小規模な環境や学習目的に適しています。

方法２：eksctlというコマンドツールを利用する方法

eksctlは、kubectlのようにコマンド操作もしくはYAML形式のマニフェストによる定義が可能です。コード化された構成により、複雑なクラスタ構築も自動化でき、大規模環境や再現性のあるクラスタ構築に向いています。

演習：Amazon EKSクラスタの構築

- **では、実際の演習を通じてAmazon EKSの構築方法を学習していきます。**
- **ここでは、最も広く使用されているeksctlを使用する方法を紹介します。**
  - eksctlは、以前はサードパーティ製品でしたが、現在はAWSの公式ツールとして提供されています。
- **eksctlは、WindowsのローカルPCにもインストール可能です。ただし、AWS認証情報の設定やAWS CLIのインストールなど、いくつかの前提条件が必要です。そのため、ここでは手順を簡略化するために、AWSのCloud Shellにeksctlをインストールして使用します。**
  - AWS Cloud Shellは、AWSが提供するブラウザベースの統合シェル環境です。AWS CLI、kubectlなどが事前にインストールされており、認証情報の設定も不要なLinux環境です。

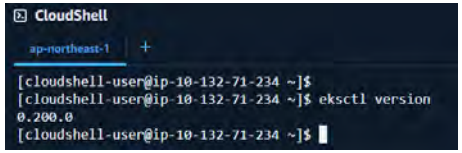
AWS Cloud Shellを開始するには、AWSマネジメントコンソールにログインし、画面上部にあるCloud Shellアイコンをクリックしてください。

演習：Amazon EKSクラスタの構築

- AWS Cloud Shellより、以下のコマンドでeksctlをインストールします：

```
$ curl -LO https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl_linux_amd64.tar.gz
$ mkdir -p ~/bin && tar -xzf eksctl_linux_amd64.tar.gz -C ~/bin
$ echo 'export PATH=~/bin:$PATH' >> ~/.bashrc
$ source ~/.bashrc
```

- インストール完了後、eksctlコマンドが問題なく使用できることを確認します：



- 次に、クラスタを作成します。今回の演習では、すべてデフォルト設定のままでクラスタを作成します。

```
$ eksctl create cluster
```

- クラスタの作成には15～20分程度かかるため、eksctlコマンドの出力を確認しながらお待ちください。

演習：Amazon EKSクラスタの構築

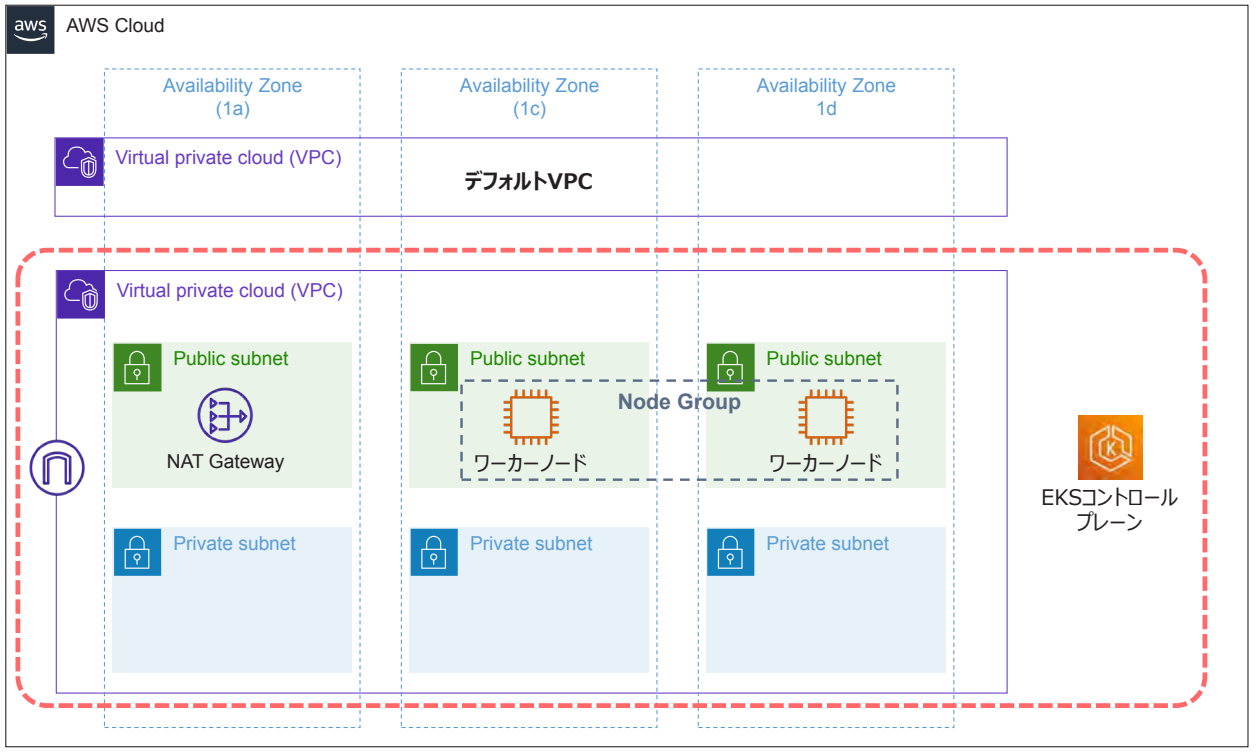
- **eksctl create clusterコマンドによって、どのようなリソースが作成されたのかを確認しましょう。ここでは、AWSの観点とKubernetesの観点の両方から見ていきます。**

赤い枠線内は、eksctlコマンドによって作成された主なAWSコンポーネントです。

- EKSコントロールプレーン
- VPC及びサブネット
  - Node Group（複数ワーカーノードを束ねた論理的なグループ）
  - Internet Gateway
  - NAT Gateway

これらのコンポーネントは、AWSマネージメントコンソールを使用してそれぞれ確認することができます。

*今回、パラメータを指定せずデフォルト設定でクラスタを作成したため、このような構成になっています。
ただし、ワーカーノードをパブリックサブネットに配置することはセキュリティ上のリスクが高いため、本番環境ではPrivateサブネットを使用することが推奨されます。



演習：Amazon EKSクラスタの構築

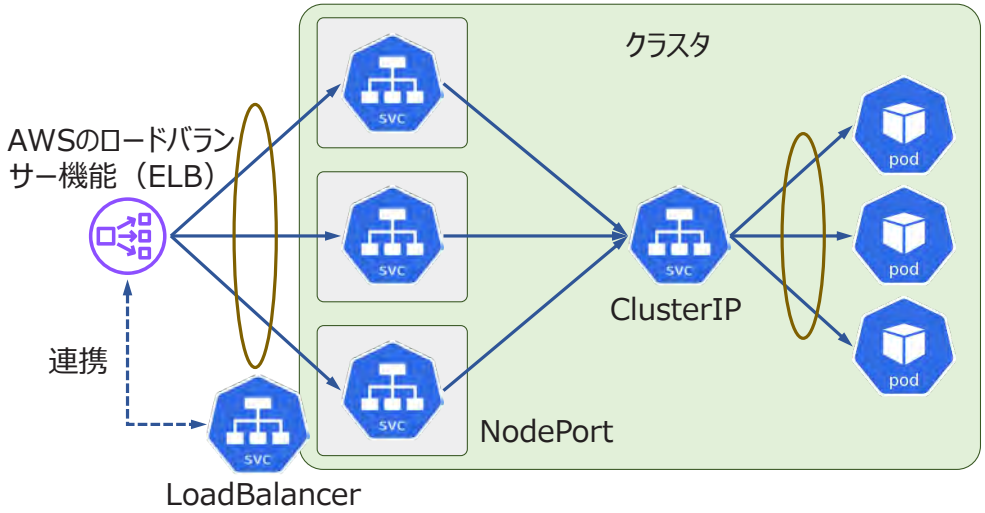
- では、Kubernetes観点から確認してみます。kubectlコマンドを使用します。

```
$ kubectl get nodes
NAME                                              STATUS   ROLES    AGE     VERSION
ip-192-168-32-252.ap-northeast-1.compute.internal   Ready    <none>   8m12s   v1.30.7-eks-59bf375
ip-192-168-85-239.ap-northeast-1.compute.internal   Ready    <none>   8m14s   v1.30.7-eks-59bf375


$ kubectl get pods -A
NAMESPACE     NAME                       READY   STATUS    RESTARTS   AGE
kube-system   aws-node-dz4xd             2/2     Running   0          9m13s
kube-system   aws-node-vn6t8             2/2     Running   0          9m15s
kube-system   coredns-5d797594d7-qfc59   1/1     Running   0          12m
kube-system   coredns-5d797594d7-t9sgx   1/1     Running   0          12m
kube-system   kube-proxy-kmj7m           1/1     Running   0          9m13s
kube-system   kube-proxy-r5m59           1/1     Running   0          9m15s
```

- ローカルPCのクラスタと異なり、EKSではワーカーノードのみ表示されます。コントロールプレーンノードは、AWSによって管理されているため、ユーザーには見えません。
- Podの一覧を確認すると、kube-proxyやcorednsなど、ワーカーノード上で実行されるKubernetesのシステムコンポーネントしかありません。

EKSでの応用例：LoadBalancerサービス

- 第四章では、疑似ロードバランサーのmetallbを使用してLoadBalancerサービスの挙動を確認しました。ここでは、クラウド上の実際のロードバランサーと連携してみます。
- AWSでは、KubernetesのLoadBalancerサービスが作成されると、自動的にAWSのELB（Elastic Load Balancing）サービスと連携し、NLB（Network Load Balancer）もしくはCLB（Classic Load Balancer）が作成されます。
- では、演習を通じて動作を確認します。右側の第四章で使用したマニフェストをそのままEKSでも利用できます。手順は次のスライドで説明します。




```yaml
# Deploymentの定義
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: kicbase/echo-server:1.0
          ports:
            - containerPort: 8080
---
# LoadBalancerの定義
apiVersion: v1
kind: Service
metadata:
  name: dummy-loadbalancer
spec:
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

四章の再掲

演習：EKSでのLoadBalancerサービス

- AWS Cloud Shellを使用する際に、ローカルPCで作成したYAMLファイルを簡単にアップロードできます。
- アップロードするには、AWS Cloud Shellの右上のアクションメニューより、「ファイルのアップロード」を選択します。
- ローカルのKubernetesクラスタと同様に、kubectlコマンドを使用して適用して、作成されたリソースを確認します。


```
$ kubectl apply -f loadbalancer.yaml
deployment.apps/deployment-web created
service/dummy-loadbalancer created


$ kubectl get -f loadbalancer.yaml
NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/deployment-web   3/3     3            3           110s

NAME                           TYPE           CLUSTER-IP      EXTERNAL-IP                                   PORT(S)        AGE
service/dummy-loadbalancer   LoadBalancer   10.100.30.223   a1...72.ap-northeast-1.elb.amazonaws.com   80:30779/TCP   110s
```

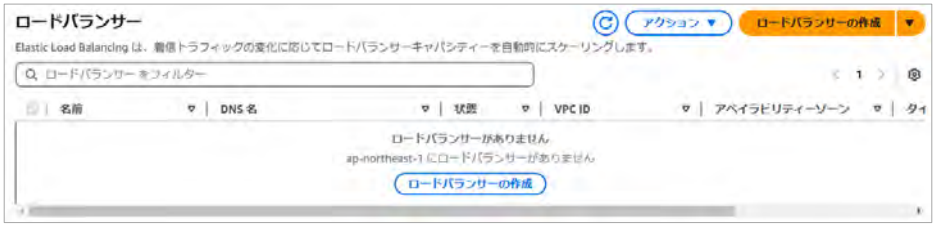
演習：EKSでのLoadBalancerサービス

- AWSのマネージメントコンソールから、ロードバランサーが自動的に作成されることを確認できます。
- ロードバランサーの詳細情報を確認すると、種類がClassicで、つまりCLB（Classic Load Balancer）であることがわかります。
- kubectlコマンドの出力に表示されている「External IP」もしくはAWSマネージメントコンソール画面に表示されている当該ロードバランサーのDNSをブラウザで開くと、echo-serverへ接続できます。

CLBが利用可能になるまで、5分以上かかることがあります。AWSマネジメントコンソールで対象のロードバランサーの詳細情報画面を開き、「ターゲットインスタンス」タブに表示されているEC2インスタンスがすべて「稼働中」と表示されるまでお待ちください。

演習：EKSでのLoadBalancerサービス

- 動作確認後、作成したリソースを削除します（kubectl delete）。
- KubernetesクラスタのLoadBalancerサービスを削除すると、AWS側のロードバランサー（CLB）も連動して削除されることを確認できます。

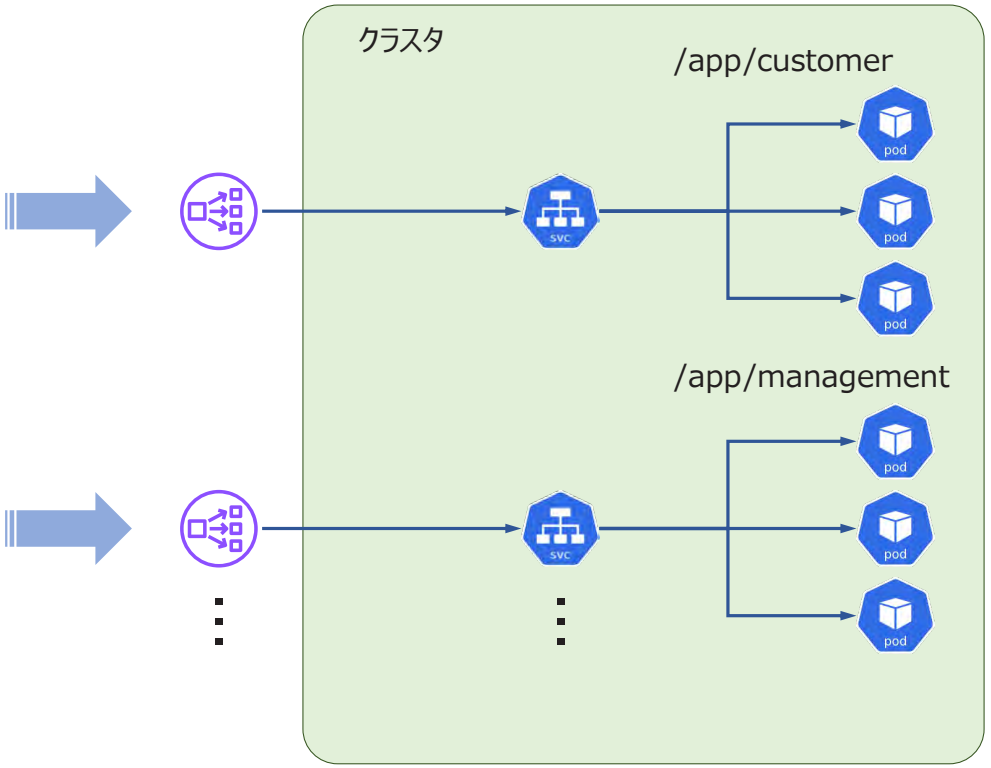


- 本演習は以上です。

この演習ではCLBを作成しましたが、CLBはレガシーのロードバランサーであり、現在は推奨されていません。本演習は、同じマニフェストが異なる環境（ローカルPCとAWSクラウド）で適用された場合の結果を確認することを目的としており、推奨される操作ではありません。

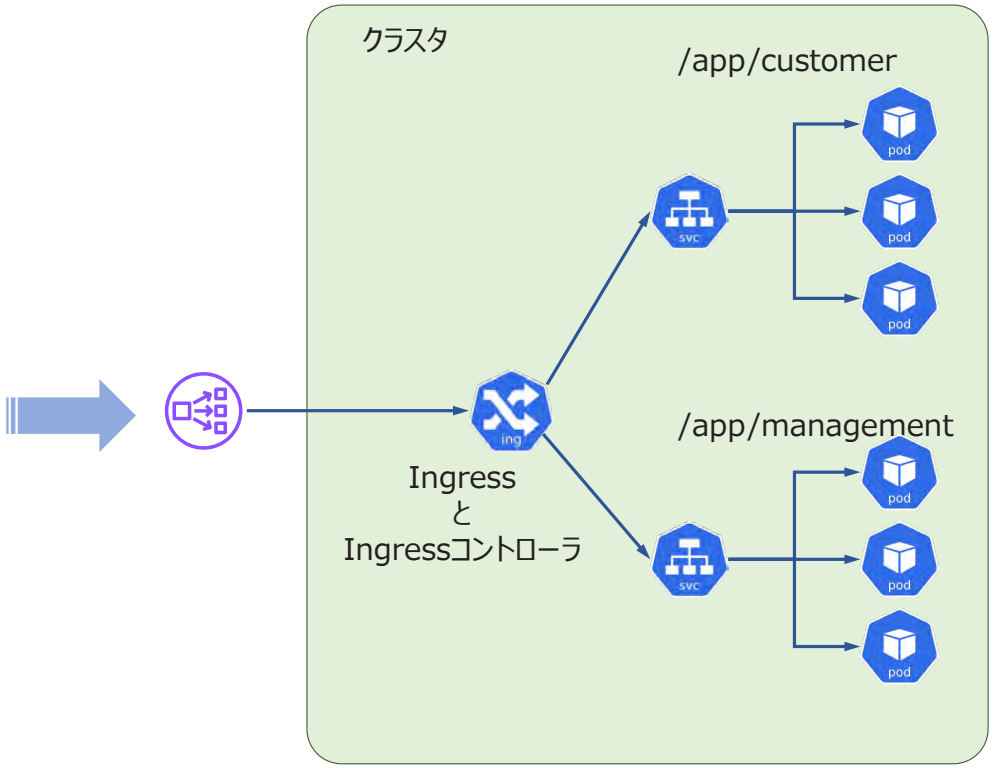
Kubernetes アプリケーションの公開

- これまでに、LoadBalancerタイプのServiceを使用して、Kubernetesクラスタ内部のサービス（例: Webアプリケーション）を外部に公開することができました。
- しかし、各サービスごとに異なるポート（NodePort）や外部IPアドレスが割り当てられるため、サービスが増えるたびに新しいポートやロードバランサーを管理する必要があります。
- また、LoadBalancerタイプのServiceはレイヤ4のTCP/IPベースのロードバランサーに限定されており、HTTPを考慮した高度なルーティング機能は持っていません。
- そのため、Kubernetesクラスタ上のHTTPベースのサービスを公開する場合は、「Ingress」というリソースを利用することが推奨されます。



Kubernetes アプリケーションの公開

- Ingressは、外部トラフィックを受け入れる単一のエントリーポイントを提供し、クラスタ内の複数のサービスやアプリケーションにトラフィックを振り分けます。
- **Ingressリソース**は、APIオブジェクトとして「kind: ingress」で定義します。
- Ingressリソース単体では動作しません。必ず**Ingressコントローラ**というコンポーネントをKubernetesクラスタにインストールしておく必要があります。
- AWS環境では、主にAWS社が提供する「AWS Load Balancer Controller (LBC)」をIngressコントローラとして利用します。LBCは、AWSのELB（Elastic Load Balancing）と連動します。



Kubernetes アプリケーションの公開

- Service（LoadBalancerタイプ）とIngressについてAWSでの動作を整理しました。
- CLB（Classic Load Balancer）が非推奨となっているため、EKSでELBを利用する際は、基本的にAWS Load Balancer Controllerの使用が推奨されています。

リソースの種類	機能	AWS Load Balancer Controllerインストール済み	AWS Load Balancer Controller未インストール（他のIngressコントローラも未インストール）
kind: Service type: LoadBalancer	L4ロードバランサー	**NLB**がプロビジョニングされる	**CLB**がプロビジョニングされる （推奨しないレガシー構成）
kind: Ingress	L7ロードバランサー、 HTTPルーティング	**ALB**がプロビジョニングされる	×（動作しない）

AWS Load Balancer Controllerのインストール

- **AWS Load Balancer Controllerのインストールは、基本的に公式ドキュメントの手順に沿って行いますが、執筆時点の最新版の手順の概要を説明します。バージョンによって手順が異なるのでご注意ください。**

- **(1) ServiceAccountの作成**
  - ServiceAccountは、KubernetesのAPIオブジェクトです。KubernetesクラスタにAWSのELBを制御する権限を与えるために、ServiceAccountを作成します。

```
$ curl -O https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.11.0/docs/install/iam_policy.json
...

$ aws iam create-policy ¥
     --policy-name AWSLoadBalancerControllerIAMPolicy ¥
     --policy-document file://iam_policy.json
{
    "Policy": {
        "PolicyName": "AWSLoadBalancerControllerIAMPolicy",
...

$ eksctl utils associate-iam-oidc-provider --cluster my-cluster --approve
2024-12-11 15:44:31 [ℹ]  will create IAM Open ID Connect provider for cluster "floral-wardrobe-1736420773" in "ap-northeast-1"
2024-12-11 15:44:31 [✓]  created IAM Open ID Connect provider for cluster "floral-wardrobe-1736420773" in "ap-northeast-1"
```

IAMポリシーのJSONファイルをダウンロードします

ダウンロードしたファイルを使用してポリシーを作成します

OIDCプロバイダーを作成します
赤文字の部分は、実際のクラスタ名に置き換えて実行してください

AWS Load Balancer Controllerのインストール

```
$ eksctl create iamserviceaccount ¥
  --cluster=floral-wardrobe-1736420773 ¥
  --namespace=kube-system ¥
  --name=aws-load-balancer-controller ¥
  --role-name AmazonEKSLoadBalancerControllerRole ¥
  --attach-policy-arn=arn:aws:iam::329599631627:policy/AWSLoadBalancerControllerIAMPolicy ¥
  --approve
2024-12-11 15:45:12 [i]  1 iamserviceaccount (kube-system/aws-load-balancer-controller) was included (based on the include/exclude
rules)
...
2024-12-11 15:45:42 [i]  created serviceaccount "kube-system/aws-load-balancer-controller"
```

ServiceAccountを作成します。赤文字の部分は、実際のクラスタ名、実際のAWSアカウントIDに置き換えて実行してください

(2) Helmのインストール

- 後続の手順では、Helmを使用するため、まずインストールを行います。HelmはKubernetesにおいて、Linuxのrpmやyumのように、アプリケーションやパッケージの管理を行うツールです。
- Helmのインストールは、公式サイトのインストール手順を参照して行います。ここでは参考までに、執筆時点の最新版の手順を記載します。

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

AWS Load Balancer Controllerのインストール

(3) AWS Load Balancer Controllerのインストール

```
$ helm repo add eks https://aws.github.io/eks-charts
"eks" has been added to your repositories

$ helm repo update eks
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "eks" chart repository
Update Complete. ⎈Happy Helming!⎈

$ helm install aws-load-balancer-controller eks/aws-load-balancer-controller ¥
  -n kube-system ¥
  --set clusterName=floral-wardrobe-1736420773 ¥
  --set serviceAccount.create=false ¥
  --set serviceAccount.name=aws-load-balancer-controller
NAME: aws-load-balancer-controller
LAST DEPLOYED: Thu Dec 12 02:15:07 2024
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
AWS Load Balancer controller installed!
```

赤文字の部分は、実際のクラスタ名置き換えて実行してください

AWS Load Balancer Controllerのインストール

(4) インストール後の確認

```
$ kubectl get pods -A
NAMESPACE     NAME                                            READY   STATUS    RESTARTS   AGE
kube-system   aws-load-balancer-controller-586687bcb6-dw4dt   1/1     Running   0          31s
kube-system   aws-load-balancer-controller-586687bcb6-w29s8   1/1     Running   0          31s
kube-system   aws-node-dz4xd                                  2/2     Running   0          2d14h
kube-system   aws-node-vn6t8                                  2/2     Running   0          2d14h
kube-system   coredns-5d797594d7-qfc59                        1/1     Running   0          2d15h
kube-system   coredns-5d797594d7-t9sgx                        1/1     Running   0          2d15h
kube-system   kube-proxy-kmj7m                                1/1     Running   0          2d14h
kube-system   kube-proxy-r5m59                                1/1     Running   0          2d14h
```

Load Balancer Controllerは、Podとしてデプロイされます

演習：Ingress/ALBの動作確認

- では、演習を通じてIngress/ALBの動作を確認してみます。この演習のマニフェストは、5つから構成されています。
- この演習では、2つのアプリケーションをデプロイしますが、どちらもecho-serverを使用して疑似的にアプリケーションとして見せます。どちらもDeploymentであり、それぞれのレプリカ数は2と3です。

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello-app-1
  template:
    metadata:
      labels:
        app: hello-app-1
    spec:
      containers:
      - name: hello-container
        image: k8s.gcr.io/echoserver:1.4
        ports:
        - containerPort: 8080
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-app-2
  template:
    metadata:
      labels:
        app: hello-app-2
    spec:
      containers:
      - name: hello-container
        image: k8s.gcr.io/echoserver:1.4
        ports:
        - containerPort: 8080
```

2つのDeploymentの相違点は、名前、レプリカ数、ラベルとセレクタの4つです。

2つのServiceの相違点は、名前とセレクタの2つです。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-service-1
spec:
  selector:
    app: hello-app-1
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-service-2
spec:
  selector:
    app: hello-app-2
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

演習：Ingress/ALBの動作確認

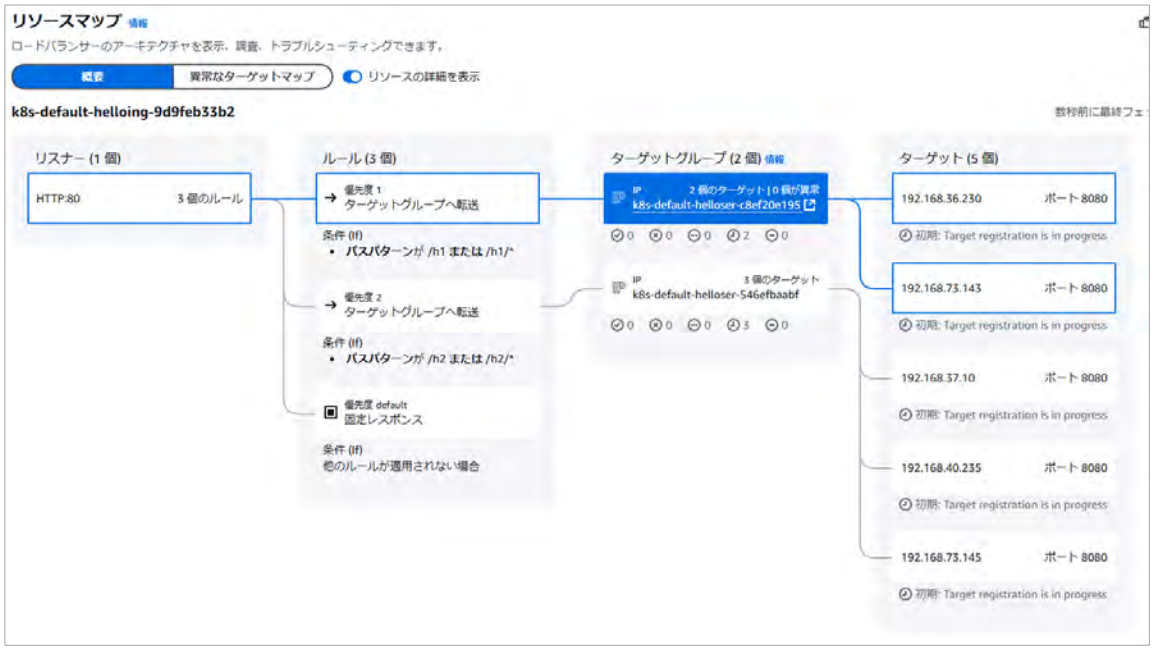
- **Ingressリソースの記述方法は以下の通りです。**
  - kind: Ingressを指定します。
  - specのingressClassNameをalbに設定します。これは、複数のIngressコントローラーが存在する場合、どのコントローラーが提供する実装を選択するかの指定です。
  - rules配下に、HTTPルーティングの詳細を記載します。ここでは、/h1のパスにアクセスされた場合、「hello-server-1」のClusterIPに転送し、/h2のパスにアクセスされた場合、「hello-server-2」のClusterIPに転送する内容が記載されています。
  - アノテーション（annotations）を使用してALBの詳細設定を制御しています。これらの設定に従って、AWS側でALBが作成されます。アノテーションは、他にも多数存在しますが、ここでは3つ紹介します。

アノテーション	設定内容
alb.ingress.kubernetes.io/scheme	インターネット向けかプライベートかを設定
alb.ingress.kubernetes.io/target-type	NodePort経由か、直接PodのIPアドレスにアクセスするかを設定
alb.ingress.kubernetes.io/listen-ports	ALBが使用するポートを設定

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-ingress
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}]'
spec:
  ingressClassName: alb
  rules:
  - http:
      paths:
      - path: /h1
        pathType: Prefix
        backend:
          service:
            name: hello-service-1
            port:
              number: 80
      - path: /h2
        pathType: Prefix
        backend:
          service:
            name: hello-service-2
            port:
              number: 80
```

演習：Ingress/ALBの動作確認

- マニフェストを適用します。ALBの作成には数分かかりますが、作成が完了すると、リソースマップでHTTPパスとPodの対応関係を確認できるようになります。
- また、「http://<ALBエンドポイントDNS名>/h1」にアクセスするとapp1に接続でき、パスの末尾を「h2」に変更することでapp2に接続できることを確認します。
- 本演習は以上です。作成したリソースを削除してください。ただし、EKSクラスタは次の章で使用するため、削除せずに残してください。




```
$ kubectl get pods -o wide
NAME                    READY   STATUS    RESTARTS   AGE   IP              NODE                                                NOMINATED NODE   READINESS GATES
app1-5d74b6bdbc-kqcrg   1/1     Running   0          39s   192.168.36.230   ip-192-168-32-252.ap-northeast-1.compute.internal   <none>           <none>
app1-5d74b6bdbc-nx47m   1/1     Running   0          39s   192.168.73.143   ip-192-168-85-239.ap-northeast-1.compute.internal   <none>           <none>
app2-f54768cbb-hr5cb    1/1     Running   0          39s   192.168.73.145   ip-192-168-85-239.ap-northeast-1.compute.internal   <none>           <none>
app2-f54768cbb-kzmxp    1/1     Running   0          39s   192.168.40.235   ip-192-168-32-252.ap-northeast-1.compute.internal   <none>           <none>
app2-f54768cbb-p6wz2    1/1     Running   0          39s   192.168.37.10    ip-192-168-32-252.ap-northeast-1.compute.internal   <none>           <none>
```

確認テスト1

Q1: Azure ACIについての記述のうち、正しいのはどれか？当てはまるものをすべて選択してください。

1. Kubernetesベースのコンテナサービス
2. 軽量で高度な機能を備えていないが、複雑な管理や設定が不要
3. Azureの管理画面からのみ操作可能
4. 単純なワークロードやテスト環境に適している

Q2: Amazon EKSの特徴について、正しいのはどれか？当てはまるものをすべて選択してください。

1. コントロールプレーンは、AWSによって管理される
2. AWSのさまざまなサービスと連携できる
3. AWS用の特殊なKubernetesを使用している
4. ワーカーノードは、EC2インスタンスやFargateを使用することができる

確認テスト2

Q3: eksctlによって構築されるKubernetesクラスタには、含まれるAWSリソースは次のどれか？当てはまるものをすべて選択してください。

1. EKSコントロールプレーン
2. VPCとサブネット
3. Internet Gateway
4. NAT Gateway
5. EC2インスタンス（ワーカーノード）

Q4: Ingress についての記述のうち、正しいものをすべて選択してください。

1. L4 で動作する
2. L7 で動作する
3. 外部トラフィックを受け入れるエントリーポイントである
4. AWS 環境でのみ利用可能

確認テスト3

Q5: AWS Load Balancer Controller（LBC）をインストール・設定済みの EKS 環境において、LBC と連携して LoadBalancer タイプの Service を作成した場合、AWS 上にどのようなリソースがプロビジョニングされるか。最も適切なものを選択してください。

1. Classic Load Balancer（CLB）
2. Application Load Balancer（ALB）
3. Network Load Balancer（NLB）
4. 上記のどれでもない

第十章 パブリッククラウド上のコンテナサービス②

この章の内容

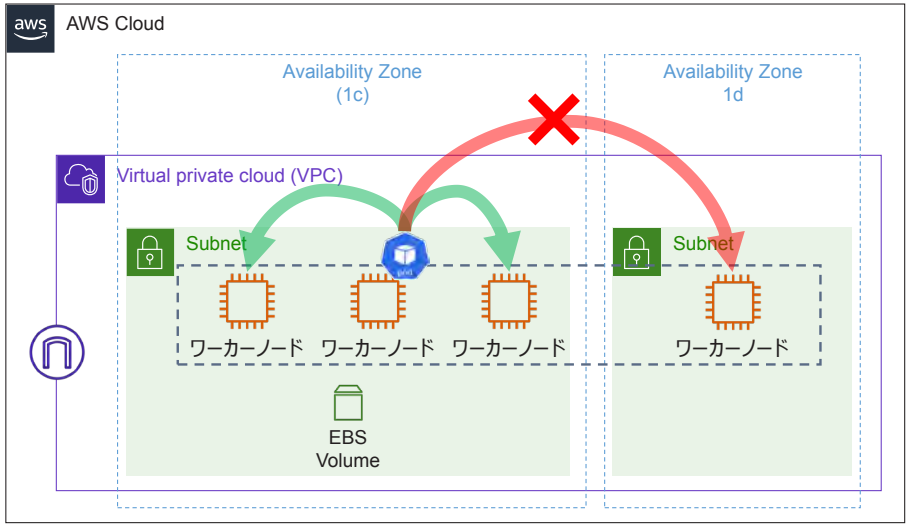
- 本章は、引き続きパブリッククラウド上のコンテナサービスをご紹介します。
- AWS上のEKS（Elastic Kubernetes Service）を中心に、以下の内容を紹介します。
  - EKSでのストレージ（EBSやEFSとの連携）
  - ConfigMap/Secret連携

EKSでのストレージ

- データの永続化には、永続的ボリューム（PV）をPodに関連付けします。第5章では、minikubeを使用して作成したローカルのKubernetesクラスタで、ノードのローカルストレージを利用し、静的または動的にPVを作成しました。
- EKSにおいても、PV/PVCの仕組みはローカル環境の場合と変わりません。
- EKSでは、PVのバックエンド（実体）として、EBSやEFSなど複数種類のストレージを利用することが可能です。

Amazon Elastic Block Store (Amazon EBS)
EBSボリュームは、ブロックストレージであり、AWSにおいて最も一般的に利用されるストレージボリュームです。

EBSボリュームの注意点は
- EBSボリュームは、単一のPodでの使用を前提としており、複数のPodからの共有には適していません。
- EBSボリュームは単一のアベイラビリティゾーン（AZ）内でのみ使用可能です。そのため、EBSボリュームを使用するPodは、別のAZに移動することができません。



演習：EBSボリュームの利用

- EKSでは、EBSボリュームを動的にプロビジョニングすることができます。ただし、デフォルトでは必要なプロビジョナー（Provisioner, CSIドライバとも言います）がインストールされていないため、インストールが必要です。
- インストール手順は、AWS公式ドキュメントにありますが、ここで手順例を記載します。

```
$ eksctl create iamserviceaccount ¥
        --name ebs-csi-controller-sa ¥
        --namespace kube-system ¥
        --cluster floral-wardrobe-1736420773 ¥
        --role-name AmazonEKS_EBS_CSI_DriverRole ¥
        --role-only ¥
        --attach-policy-arn arn:aws:iam::aws:policy/service-role/AmazonEBSCSIDriverPolicy ¥
        --approve
2024-12-13 08:21:20 [!]  serviceaccounts in Kubernetes will not be created or modified, since the option --role-only is used
2024-12-13 08:21:20 [i]  1 task: { create IAM role for serviceaccount "kube-system/ebs-csi-controller-sa" }
...

$ eksctl create addon ¥
        --name aws-ebs-csi-driver ¥
        --version latest ¥
        --cluster floral-wardrobe-1736420773 ¥
        --service-account-role-arn arn:aws:iam::329599631627:role/AmazonEKS_EBS_CSI_DriverRole ¥
        --force
2024-12-13 08:22:11 [i]  Kubernetes version "1.30" in use by cluster "floral-wardrobe-1736420773"
2024-12-13 08:22:11 [i]  IRSA is set for "aws-ebs-csi-driver" addon; will use this to configure IAM permissions
...
```

IAMロールを作成します

CSIドライバ（アドオン）をインストールします

演習：EBSボリュームの利用

- インストール完了後、AWSマネージメントコンソールからでも確認可能です。

- また、kubectlコマンドでもkube-systemネームスペースに、関連するPodが作成されていることを確認できます。

```
$ kubectl get pods -n kube-system
NAME                                  READY   STATUS    RESTARTS   AGE
...
ebs-csi-controller-5578675b85-c4kjm   6/6     Running   0          15m
ebs-csi-controller-5578675b85-mmtml   6/6     Running   0          15m
ebs-csi-node-cbq8d                    3/3     Running   0          15m
ebs-csi-node-lpdxm                    3/3     Running   0          15m
...
```

- これでEBSボリュームのプロビジョニングが準備完了です。

演習：EBSボリュームの利用

- それでは、PVCを作成します。PVCを作成する前に、まず既存のStorageClassを確認します。
- 以下の通り、初期値としてgp2というStorageClassが存在します。EBSは複数種類のストレージボリュームを提供しているので、それに合わせて新規にStorageClassを作成することも可能です。

```
$ kubectl get storageclass
NAME   PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
gp2    kubernetes.io/aws-ebs   Delete          WaitForFirstConsumer   false                  3d16h


$ kubectl describe storageclass gp2

Name:            gp2
IsDefaultClass:  No
Annotations:     kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"s...

Provisioner:           kubernetes.io/aws-ebs
Parameters:            fsType=ext4,type=gp2
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

「gp2」という名前のStorageClassが存在することを確認できます。

WaitForFirstConsumerとは、PVをPVCにバインドする際に実際にボリュームをプロビジョニングせず、PodがPVCを利用しようとする際に初めてボリュームをプロビジョニングする方法です。これにより、ストレージのローカリティ（ノードの地理的近接性やネットワークの効率）を最適化できます。

演習：EBSボリュームの利用

- PVCとPod作成に使用するマニフェストはこちらの通りです。
- ローカルKubernetes環境で使用したマニフェストとほぼ同一ですが、storageClassNameを"gp2"に変更しています。
- マニフェストを適用し、AWSマネージメントコンソールで適用前後のEBSの状況を確認します。

マニフェスト適用前

マニフェスト適用後


```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-pvc
spec:
  storageClassName: "gp2"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-ebs
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - mountPath: /data
          name: pvc-volume
  volumes:
    - name: pvc-volume
      persistentVolumeClaim:
        claimName: ebs-pvc
```

演習：EBSボリュームの利用

- また、kubectlコマンドでもPV/PVCのステータスを確認することができます。

```
$ kubectl get pvc
NAME      STATUS   VOLUME             CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
ebs-pvc   Bound    pvc-9e01712b-...   1Gi        RWO            gp2            <unset>                 6s

$ kubectl get pv
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM             STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
pvc-9e01...   1Gi        RWO            Delete           Bound    default/ebs-pvc   gp2            <unset>                          98s
```

- 本演習は以上です。作成したPodやPVCを削除してください。

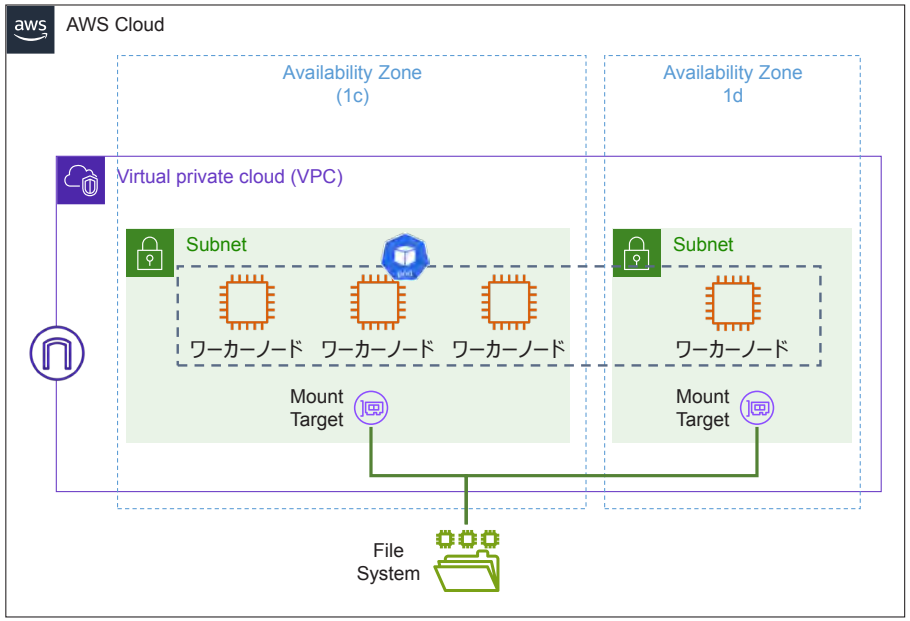
EKSでのストレージ

EKSで利用できるストレージのバックエンドは、EBSのほかにEFSも広く使われています。

Amazon Elastic File System (Amazon EFS)
NFSファイルサーバーのマネージドサービスです。

EFSの特徴はEBSと異なります。
- EFSは、NFSベースであるため、複数Podからの共有が可能です。またアベイラビリティゾーン（AZ）を跨いだ使用や共有もできます。
- EBSはブロックストレージですが、EFSはファイルベースのストレージです。

EFSを使用するには、まず「**ファイルシステム**（File System）」を作成します。そして、そのファイルシステムに特定のサブネットからアクセスできるよう、「**マウントターゲット**（Mount Target）」をそのサブネットに作成する必要があります。



演習：Amazon EFSの利用

- 演習を通じてAmazon EFSをEKSから利用する方法を学習していきます。
- まずは、Amazon EFSの作成と設定を行います。Amazon EFSは本講座の範囲外ですので、概要レベルで説明します。
- Amazon EFSの作成は、マネージメントコンソール画面からも、CLIからも可能です。ここでは、マネージメントコンソールから作成する方法を紹介します。

AWSのマネージメントコンソールより、EFSのサービスを開き、「ファイルシステムの作成」をクリックします。
作成する際に、**VPCは必ずEKSクラスタのVPC**を選択してください。

演習：Amazon EFSの利用

- **次に、EFS用のプロビジョナー（CSIドライバー）をインストールします。Helmを利用した方法を紹介します。**
  - Cloud Shellの場合、一度ログアウトして再度ログインするとインストール済みのHelmが消えてしまうことがあります。もしHelmコマンドが見つからないエラーが表示された場合、前章の手順で再度Helmをインストールしてください。

```
$ helm repo add aws-efs-csi-driver https://kubernetes-sigs.github.io/aws-efs-csi-driver/
"aws-efs-csi-driver" has been added to your repositories

$ helm repo update aws-efs-csi-driver
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "aws-efs-csi-driver" chart repository
Update Complete. ⎈Happy Helming!⎈

$ helm install aws-efs-csi-driver --namespace kube-system aws-efs-csi-driver/aws-efs-csi-driver
NAME: aws-efs-csi-driver
LAST DEPLOYED: Fri Jan 17 06:25:05 2025
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
To verify that aws-efs-csi-driver has started, run:

    kubectl get pod -n kube-system -l "app.kubernetes.io/name=aws-efs-csi-driver,app.kubernetes.io/instance=aws-efs-csi-driver"
```

演習：Amazon EFSの利用

- **EFSのプロビジョナーをインストールするだけでは、StorageClass は自動的に作成されません。公式サイトから最新版のマニフェストをダウンロードし、必要な修正を加えた上で適用し、StorageClass を作成してください。**

https://raw.githubusercontent.com/kubernetes-sigs/aws-efs-csi-driver/master/examples/kubernetes/dynamic_provisioning/specs/storageclass.yaml

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
parameters:
  provisioningMode: efs-ap
  fileSystemId: fs-92107410
  directoryPerms: "700"
  gidRangeStart: "1000" # optional
  gidRangeEnd: "2000" # optional
  basePath: "/dynamic_provisioning" # optional
  subPathPattern: "${.PVC.namespace}/${.PVC.name}" # optional
  ensureUniqueDirectory: "true" # optional
  reuseAccessPoint: "false" # optional
```

fileSystemIdは、ダミーなので必ず実際のファイルシステムのIDに置き換えてください

ダウンロードしたファイルをテキストエディタで開き、「fileSystemId」の部分を実際のEFSファイルシステムのIDに置き換えて保存します。

※このマニフェストのほかの部分も必要に応じて変更することは可能ですが、今回はそのまま使用します。
また、詳細情報は公式サイトを参照してください。

演習：Amazon EFSの利用

- 修正後のStorageClass作成用マニフェストをCloud Shellにアップロードして適用します。
- 適用後、StorageClassの一覧を確認します。新しく「efs-sc」というStorageClassが作成されていることを確認できます。

```
$ kubectl apply -f storageclass.yaml
storageclass.storage.k8s.io/efs-sc created

$ kubectl get sc
NAME     PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
efs-sc   efs.csi.aws.com         Delete          Immediate              false                  6s
gp2      kubernetes.io/aws-ebs   Delete          WaitForFirstConsumer   false                  7d22h
```

- StorageClass が作成されていることを確認した後、右側の例のようにPVCを作成できます。なお、Podのマニフェストは省略します。
- Amazon EFSは、RWX（ReadWriteMany、複数ノードからの読み書き） をサポートしています。さらに、特に容量の上限がないため、永続的なストレージとしてステートフルワークロード、ログ管理、CI/CD、機械学習、データ共有など、幅広い用途で活用できます。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: efs-sc
  resources:
    requests:
      storage: 5Gi
```

EFSは、RWXに対応しています

設定情報・シークレットの管理

- オンプレミス環境のKubernetesクラスタでは、ConfigMap や Secret を使用してアプリケーションの設定情報やシークレットを管理します。
- 一方、AWSクラウド上のEKSでは、AWSクラウドから提供されるサービスを利用して設定情報やシークレットを管理することが可能です。
- AWSが提供するサービスと概要は、次の通りです。

Kubernetesリソース	AWSでの対応サービス	AWSサービスの概要
ConfigMap	 AWS Systems Manager Parameter Store	アプリケーション設定やパラメータを安全に管理するサービス。KMSを使用した暗号化が可能。
Secret	 AWS Secrets Manager	機密情報（パスワード、APIキーなど）を安全に保存・管理し、自動ローテーションが可能なサービス。

演習：設定情報・シークレットの管理

- では演習を通じてConfigMap/SecretのAWSサービスとの連携を確認してみます。
- この連携を行うには、いくつかの準備作業が必要です。
  - IAMポリシーの作成
  - IAM Service Accountの作成（IAMロールの作成）
  - Secrets Store CSI ドライバーのインストール
  - AWS Provider and Config Provider (ASCP)のインストール

IAMポリシーの作成
EKSクラスタからAWSのサービスにアクセスするには、適切なIAMポリシーを作成する必要があります。

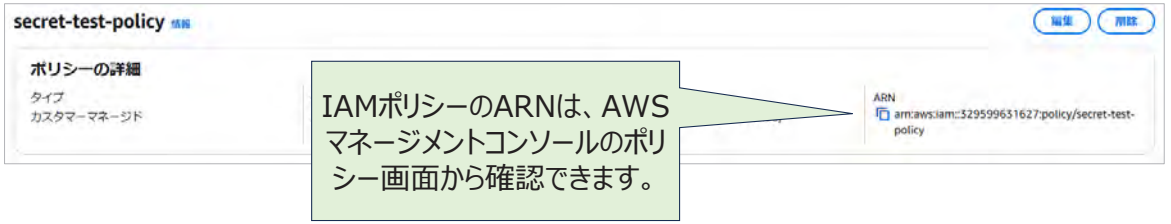
右側は、AWSのSecrets ManagerやParameter Storeにアクセスするための IAMポリシーサンプルです。こちらを使用してIAMポリシーを作成します。作成手順はAWS関連ドキュメントをご確認ください。

※このサンプルでは、 Secrets ManagerやParameter Storeのすべてのオブジェクトの読み取り権限を付与しています。実際の本番環境では、最小権限の原則に則ってより細かく権限を設定することを推奨しています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret",
                "ssm:GetParameters"
            ],
            "Resource": "*"
        }
    ]
}
```

演習：設定情報・シークレットの管理

- IAMポリシーを作成後、ARNを控えてIAM Service Accountを作成します。



IAM Service Accountの作成

```
$ eksctl create iamserviceaccount ¥
        --name configuration-sa ¥
        --namespace default ¥
        --cluster floral-wardrobe-1736420773 ¥
        --role-name AmazonEKS_ASCP_DriverRole ¥
        --attach-policy-arn arn:aws:iam::329599631627:policy/secret-test-policy ¥
        --approve
2024-12-13 08:21:20 [ℹ] 1 task: {
    2 sequential sub-tasks: {
        create IAM role for serviceaccount "kube-system/configuration-sa",
        create serviceaccount "kube-system/configuration-sa",
...
```

クラスタ名およびポリシーのARNは、実際の値に置き換えてください。

演習：設定情報・シークレットの管理

- 次に、Secrets Store CSI ドライバーをインストールします。
- Secrets Store CSI ドライバーとは、外部（AWS/Azure/GCPなど）のシークレット管理システムやパラメータ管理システムから、シークレットや設定情報をPod内のボリュームとしてマウントできる仕組みです。


```
$ helm repo add secrets-store-csi-driver https://kubernetes-sigs.github.io/secrets-store-csi-driver/charts
"secrets-store-csi-driver" has been added to your repositories

$ helm install csi-secrets-store secrets-store-csi-driver/secrets-store-csi-driver --namespace kube-system
NAME: csi-secrets-store
LAST DEPLOYED: Fri Jan 17 11:45:27 2025
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The Secrets Store CSI Driver is getting deployed to your cluster.

To verify that Secrets Store CSI Driver has started, run:

  kubectl --namespace=kube-system get pods -l "app=secrets-store-csi-driver"

Now you can follow these steps https://secrets-store-csi-driver.sigs.k8s.io/getting-started/usage.html
to create a SecretProviderClass resource, and a deployment using the SecretProviderClass.
```

演習：設定情報・シークレットの管理

- ASCP（AWS Secrets Manager and Config Provider for Secret Store CSI Driver）をインストールします。

```
$ helm repo add aws-secrets-manager https://aws.github.io/secrets-store-csi-driver-provider-aws
"aws-secrets-manager" has been added to your repositories

$ helm install -n kube-system secrets-provider-aws aws-secrets-manager/secrets-store-csi-driver-provider-aws
NAME: secrets-provider-aws
LAST DEPLOYED: Fri Jan 17 11:50:16 2025
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

- インストール完了後、Secrets Store CSI ドライバーとASCPが正常に動作していることを確認します。

```
$ kubectl --namespace=kube-system get pods
...
csi-secrets-store-secrets-store-csi-driver-92dpk                  3/3     Running   0          4m55s
csi-secrets-store-secrets-store-csi-driver-rrqnj                  3/3     Running   0          4m55s
...
secrets-provider-aws-secrets-store-csi-driver-provider-awskk2xb   1/1     Running   0          14s
secrets-provider-aws-secrets-store-csi-driver-provider-awswvvnq   1/1     Running   0          13s
```

Secrets Store CSI ドライバー

ASCP

以上で準備完了です。

演習：設定情報・シークレットの管理

- **ではテスト用のPodを作成します。その前に、まずSecretProvierClassを作成します。**
  - SecretProvierClass とは、Secrets Store CSI Driver に関連するリソースで、外部のシークレット管理システム（例: AWS Secrets ManagerやParameter Storeなど）からKubernetesクラスタ内の Pod にシークレットをマウントするための設定を定義するものです。

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: aws-secrets
spec:
  provider: aws
  parameters:
    objects: |
      - objectName: "test-secret"
        objectType: "secretsmanager"
      - objectName: "test-param"
        objectType: "ssmparameter"
```

- **次にPodを作成します。SecretProvierClassをボリュームとしてマウントします。**

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-app
  namespace: default
spec:
  serviceAccountName: configuration-sa
  containers:
  - name: busybox
    image: busybox
    command: [ "sleep", "3600" ]
    volumeMounts:
    - name: secrets-store
      mountPath: "/mnt/secrets"
      readOnly: true
  volumes:
  - name: secrets-store
    csi:
      driver: secrets-store.csi.k8s.io
      readOnly: true
      volumeAttributes:
        secretProviderClass: "aws-secrets"
```

演習：設定情報・シークレットの管理

- マニフェストを適用します。
- 適用後、対象コンテナのシェルにアクセスします。
  - /mnt/secretsディレクトリに、AWS Secrets Managerに格納されている値も、Parameter Storeに格納されている値も問題なく表示されていることを確認します。

```
$ kubectl apply -f sec-test.yaml
secretproviderclass.secrets-store.csi.x-k8s.io/aws-secrets created
pod/secret-app created

$ kubectl exec -it secret-app -- /bin/sh
/ # ls /mnt/secrets/ -l
total 8
-rw-r--r--    1 root     root            19 Jan 19 08:22 test-param
-rw-r--r--    1 root     root            29 Jan 19 08:22 test-secret

/ # cat /mnt/secrets/test-param
my-test-param-value

/ # cat /mnt/secrets/test-secret
{"my-api-key":"my-api-value"}
```

試しに、AWS Secrets Manager または Parameter Store の値を変更し、再度上記の cat コマンドを実行してみてください。
値は反映されましたか？

この設定の仕様上、Podのライフサイクル中に Secrets Manager や Parameter Store で変更があっても、その変更は自動的には反映されません。
シークレットがマウントされるのは Podの起動時のみです。もちろん、リアルタイムで反映させる仕組みもありますが、ここでは割愛します。

おわりに

- 演習は以上です。作成したリソースはすべて削除して構いません。
- また、すべてのリソースやデータを完全に削除する必要がある場合は、AWSアカウントを閉鎖することも可能です。
- AWSアカウントを閉鎖するには、ルートユーザーでログインし、画面右上のメニューから「アカウント」を選択し、「アカウントを閉鎖」をクリックしてください。これにより、さらなる料金の発生を防ぐことができます。

確認テスト1

Q1: EBS ボリュームを EKS で利用する場合、次の記述のうち正しいものをすべて選択してください。

1. 複数の Pod で共有するのに適している
2. gp2 や gp3 など、異なる性能のボリュームを選択できる
3. AZ をまたいで別のワーカーノードに移動できる
4. コンテナからはブロックストレージとして認識される

Q2: EFS ファイルシステムを EKS で利用する場合、次の記述のうち正しいものをすべて選択してください。

1. NFS や CIFS など、複数のファイルシステムをサポートする
2. マウントターゲットはパブリックサブネットに作成する必要がある
3. AZ をまたいで使用できる
4. コンテナからはファイルストレージとして認識される

確認テスト2

Q3: Secrets Store CSI の機能として最も適切なものを選択してください。

1. AWS Secrets Manager と直接連携する
2. Kubernetes クラスタ内部のシークレットを管理する
3. 外部のシークレット管理システムから情報を取得し、Pod のボリュームとしてマウントする
4. 外部の暗号化されたボリュームを PVC として Pod にマウントする

確認テスト回答

第一章

Q1　2

Q2　1|3

Q3　2|3

Q4　2|4

Q5　2|3

Q6　3

Q7　4

Q8　2|4

第二章

Q1　1|2|4

Q2　2|3

Q3　1|2|4

Q4　3

Q5　1

第三章

Q1　1|3|4

Q2　2|3

Q3　3

Q4　2

Q5　1|4

Q6　1|5

第四章

Q1　2

Q2　1

Q3　2|4|5

Q4　1|5

Q5　2|3

Q6　2

第五章

Q1　1|3|4

Q2　1|2|3

Q3　1|2

Q4　2|3|4

Q5　1|4|5

Q6　2

Q7　1|3

Q8　1|2

第六章

Q1　3|4

Q2　1|4

Q3　3|4

Q4　1|2|4

Q5　2

Q6　1|2|

Q7　3

第七章

Q1　2|3

Q2　2|5

Q3　2

Q4　1|3|5

Q5　1|2

Q6　1|3

Q7　3|4

第八章

Q1　1|3

Q2　2

Q3　1|4

Q4　2|3

Q5　4

Q6　1

第九章

Q1　2|4

Q2　1|2|4

Q3　1|2|3|4|5

Q4　2|3

Q5　3

第十章

Q1　2|4

Q2　3|4

Q3　3

ー 演習課題ー

演習課題①

課題 1

ローカル環境にDockerをインストールし、公式の nginx イメージを使ってコンテナを起動してください。その際、ポート 8080 でアクセスできるように設定してください。

課題 2

Python 3.9 をベースとしたDockerイメージを作成し、Flask をインストールするDockerfileを作成してください。その後、コンテナをビルドして起動してください。

課題 3

kubectl を使用して、新しいNamespace my-namespace を作成し、その中に nginx のPodをデプロイしてください。

課題 4

Google Kubernetes Engine (GKE) または Amazon Elastic Kubernetes Service (EKS) を使用して、クラウド環境にKubernetesクラスタを作成し、Podをデプロイしてください。

令和6年度
令和6年度文部科学省委託「専門職業人材の最新技能アップデートのための専修学校リカレント教育推進」事業
情報技術者の技能アップデートのためのリカレント教育推進事業

コンテナ技術システム構築教材資料

令和7年2月

一般社団法人全国専門学校情報教育協会
〒164-0003 東京都中野区東中野1-57-8 辻沢ビル3F
電話：03-5332-5081 FAX：03-5332-5083