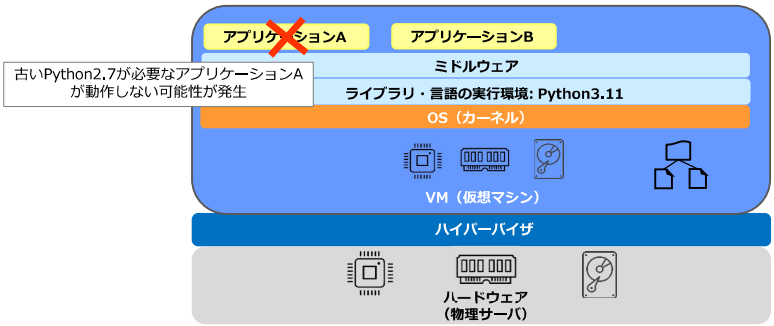
1 Docker 概要

仮想サーバでアプリケーションを動作させる場合の課題①

・アプリケーションの起動が完了するまでの作業に手間が発生
- アプリケーションの依存関係のインストール
- アプリケーションのインストール
- 設定ファイルの編集
- アプリケーションの起動

仮想サーバでアプリケーションを動作させる場合の課題②

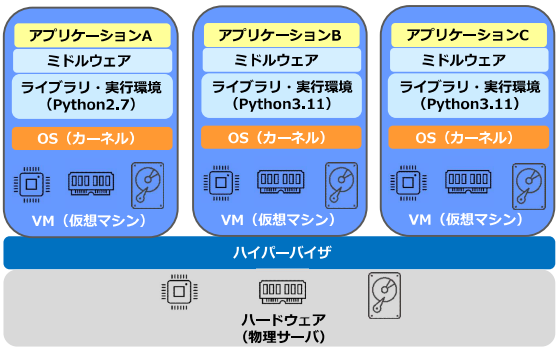
・同じサーバ上で異なるバージョンの依存関係（ライブラリ・言語の実行環境・ミドルウェア等）を必要とする複数のアプリケーションを動かしたい場合、バージョンの競合が起きる可能性が発生
- アプリA： 古くに作られたもので Python 2.7 が必要
- アプリB： 最新機能を使うために Python 3.11 が必要



仮想サーバでアプリケーションを動作させる場合の課題②の対応方法

・１つの仮想マシンにつき1つのアプリケーションを起動

・依存関係のバージョンを上げても、
他のアプリケーションに影響がない状況にすることが可能

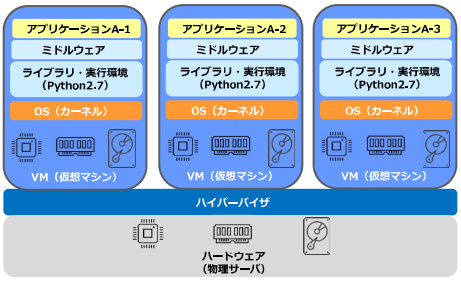


5

仮想サーバでアプリケーションを動作させる場合の課題③

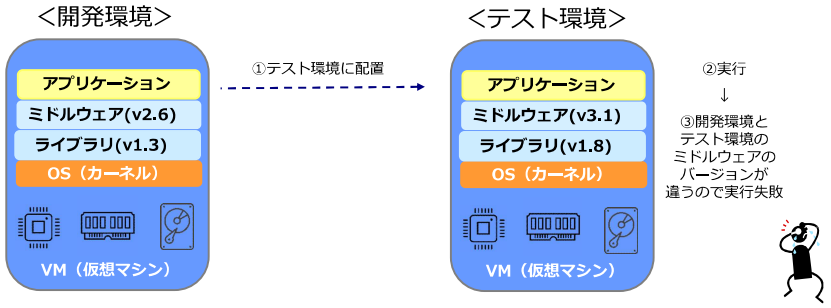
・1つのアプリケーションを１つの仮想マシンを起動した場合、様々な課題が発生

- 仮想マシンのOS（カーネル）の稼働に**多くのCPU・メモリ・ストレージの割り当てが必要**
- ライセンスが必要なOSの場合、**ライセンス費用が増加**
- アプリケーションをスケールアウト（増加）したい場合、
仮想マシンを新規作成＋起動する必要があり、仮想マシンのOSの起動に数分時間が必要
  - アプリケーションがサービスを提供可能になるまで時間が掛かり、
スパイクアクセス（アクセスの急増）の対応が困難



仮想サーバでアプリケーションを動作させる場合の課題④

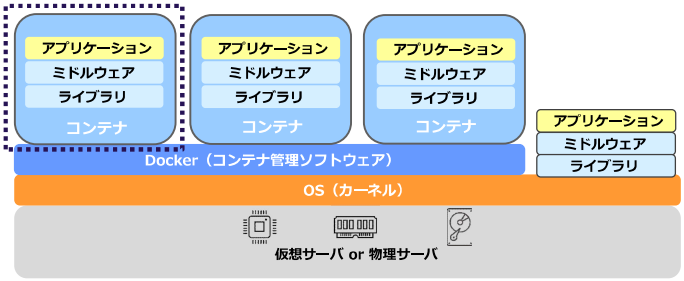
・開発・テスト・本番など各環境のサーバのアプリケーションの依存関係のバージョンの不一致があると、別環境でアプリケーションを動作させようとした際にうまく動作しない状況が発生
  - **アプリケーションの改修・機能追加に時間が掛かる**状況に



7

コンテナ

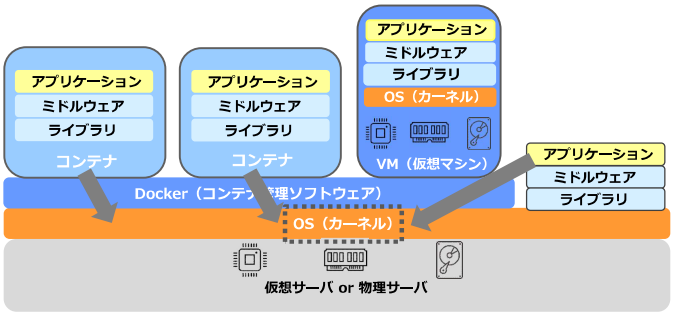
・アプリケーションを動作させることができる論理的な空間
  - OSの機能を使用し、OS上にコンテナを作成可能
  - アプリケーションとその依存関係をパッケージ化（イメージ化）し、どのコンピュータでもアプリケーションを動作させることが可能



8

コンテナの特徴（仮想マシンと異なる点①）

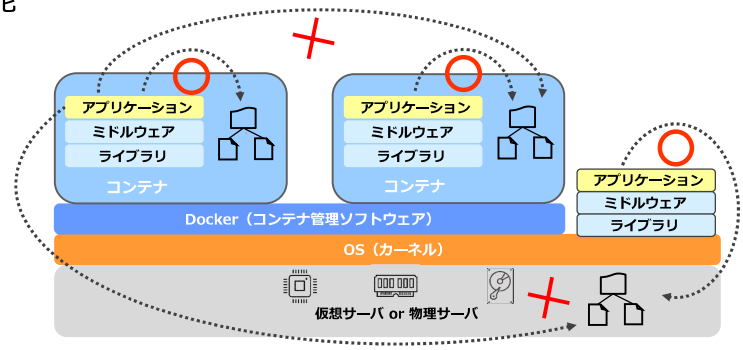
- コンテナ内ではOSは起動しない
  - コンテナ内のアプリケーションはDockerホストのOSを使用
  - コンテナ内のアプリケーションはDockerホストのOS上で動作するアプリケーションと同じ形で動作



11

コンテナの特徴（仮想マシンと同じ点①）

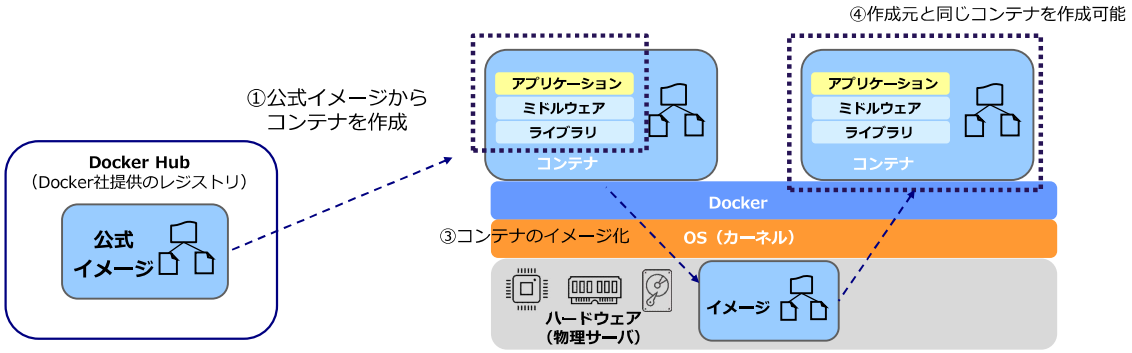
- コンテナは独自のファイルシステム（ディレクトリ階層構造とファイル）を保有
  - アプリケーションとアプリケーションを動作させるために必要な依存関係（ライブラリ・言語の実行環境・ミドルウェア）などを格納可能
- コンテナ内のアプリケーションは同じコンテナ内のファイルシステムにのみアクセス可能



12

独自作成のイメージからコンテナを作成

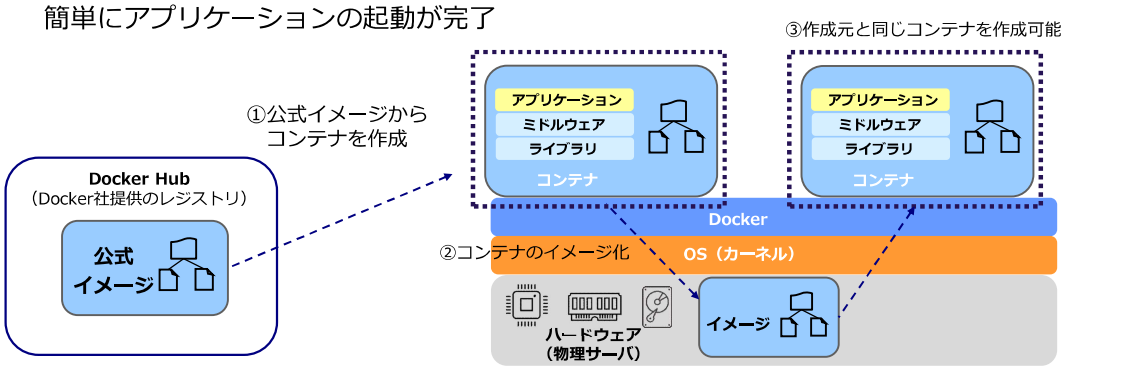
- イメージの作成元と同じコンテナを作成可能
  - 作成元と同じ依存関係・アプリケーションが配置されているコンテナが作成
    - コンテナのスケールアウトが可能
    - 他のユーザが同じコンテナを作成可能



19

アプリケーションのコンテナ化による仮想マシンでの課題の解消①

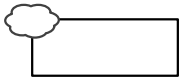
- サーバのOS上に簡単にアプリケーションを起動させることが可能
  - Docker Hubの公式イメージからコンテナを作成すると依存関係等まで準備されたコンテナを作成可能
  - 独自作成のコンテナイメージからコンテナを作成することで、簡単にアプリケーションの起動が完了



20

2 コンテナ・イメージの作成と管理

イメージ関連の基本用語

・レジストリ 
- イメージの保存・配布を行う場所

・リポジトリ 
- 特定のイメージのすべてのバージョンをグループ化したもの

・イメージ 
- コンテナの作成元となるリソース

・タグ
- リポジトリ内のイメージのバージョンを識別するための名前

35

レジストリ

・イメージを配置できるHTTPサーバ
- イメージからコンテナを作成できるのは、そのイメージがあるDockerホストのみ可能
- もし別のDockerホストでコンテナを作りたい場合は、イメージをレジストリに配置すると別のDockerホストでもイメージをダウンロードしてコンテナを作成可能に

36

コンテナ作成時にコンテナ内でシェルを起動し、コンテナ内でコマンドを実行できる状態にする方法

- 「**コンテナ内で実行するコマンド**」にシェルを起動するコマンド（bashコマンドやshコマンド等）を指定することで可能
  - 実行形式:

    $ docker container run -it リポジトリ名[:タグ] bash

    - -itオプションを指定することでコンテナ内で起動したシェルを利用可能に
      - -i: コンテナのメインプロセス（シェル）へのキーボード入力（標準入力）を端末から行えるように接続
      - -t: コンテナのメインプロセス（シェル）からの出力（標準出力）を端末に接続

  - コンテナの作成後、すぐコンテナ内でコマンドを実行したい場合に使用

55

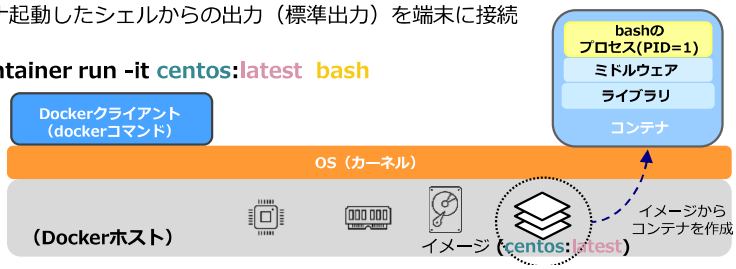
コンテナ作成時にコンテナ内でシェルを起動し、コンテナ内ですぐコマンドを実行できる状態にする方法

- 「**コンテナ内で実行するコマンド**」にシェルを起動するコマンド（bashコマンドやshコマンド等）を指定することで可能
  - 実行形式:

    $ docker container run -it リポジトリ名[:タグ] bash

    - -itオプションを指定することでコンテナ内で起動したシェルを利用可能に
      - -i: コンテナに起動したシェルへのキーボード入力（標準入力）を端末から行えるように接続
      - -t: コンテナ起動したシェルからの出力（標準出力）を端末に接続

    $ docker container run -it centos:latest bash



57

Docker Hub のイメージの検索とイメージのダウンロード

1. Docker Hub にある AlmaLinuxOS 関連のイメージを検索します。NAME または DESCRIPTION フィールドに「almalinux」という文字が記載されているイメージを検索します。

```
[user01@ip-192-168-1-1 ~]$ docker search almalinux
NAME                  DESCRIPTION                                     STARS     OFFICIAL
almalinux/almalinux   DEPRECATION NOTICE: This image is deprecated…   9
almalinux             The official build of AlmaLinux OS.             192
[OK]
```

2. Docker Hub のリポジトリ名を引数に指定し、そのリポジトリに置かれているイメージのタグを一覧表示する関数「docker-image-taglist」を作成します。

```
[user01@ip-192-168-1-1 ~]$ function docker-image-taglist { curl -s
https://hub.docker.com/v2/repositories/library/$1/tags?page_size=30   | sed
"s/,/¥n/g" | grep '"name"' | sed 's/:/ /g' | sed 's/¥"//g'|  cut -d ' '  -f
2; }
```

3. 引数に「almalinux」と指定して docker-image-taglist を実行し、Docker Hub の公式リポジトリ「almalinux」にあるイメージのタグを一覧表示します。

```
[user01@ip-192-168-1-1 ~]$ docker-image-taglist  almalinux
latest
minimal
9.6-minimal-20250611
9.6-minimal
9.6-20250611
9.6
9-minimal
9
8.10-minimal-20250611
8.10-minimal
（以下略）
```

4. Docker Hub 公式の「almalinux」のイメージの「latest」のタグが付いているイメージをダウンロードします。ダウンロード前に docker image ls コマンドを実行して Docker ホストにあるイメージを確認します（デフォルトでは Docker ホストに almalinux イメージは存在しません）。

```
[user01@ip-192-168-1-1 ~]$ docker image ls
REPOSITORY    TAG       IMAGE ID            CREATED       SIZE
```

5. docker image pull コマンドを実行して Docker Hub にある almalinux:latest イメージを Docker ホストにダウンロードします。

```
[user01@ip-192-168-1-1 ~]$ docker image pull almalinux:latest
latest: Pulling from library/almalinux
```

6. docker image ls コマンドを実行し、Docker ホストに almalinux:latest イメージがダウンロードされていることを確認します。

```
[user01@ip-192-168-1-1 ~]$ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
almalinux    latest    623706a2d956   13 hours ago   189MB
```

7. docker image pull コマンドを実行して Docker Hub にある ubuntu の latest タグが付いたイメージをダウンロードします。

```
[user01@ip-192-168-1-1 ~]$ docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
```

8. docker image ls コマンドを実行すると、Docker ホストに ubuntu イメージがダウンロードされていることが確認できます。

```
[user01@ip-192-168-1-1 ~]$ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
almalinux    latest    623706a2d956   13 hours ago   189MB
ubuntu       latest    bf16bdcff9c9   13 days ago    78.1MB
```

3 Dockerfile

Dockerfileに記載する命令①

- FROM命令:
  - ベースとして使用するイメージ名を指定
  - 書式:

```
FROM イメージ名
```

  - 記載例:

```
FROM almalinux:9
```

    - Dockerfileの最初の命令にはFROMを記載
    - 2章で独自アプリが動作するコンテナを作成した際のコンテナの作成元としたDocker Hubの公式イメージを指定した箇所に相当する命令
    - 可能な限り最新の公式イメージ名の指定が推奨
      （定常的に脆弱性の修正や設定が確認されているため）

```
# ベースイメージの指定
FROM almalinux:9

# Maintainerの登録
LABEL maintainer="abc@example.com"

# Node.jsのインストール等
RUN dnf -y install nodejs procps && mkdir /app

# Dockerホストのemoji.jsをコンテナの
# /appディレクトリにコピー
COPY ./emoji.js /app/emoji.js

# ポート3000でリッスン
EXPOSE 3000

# アプリケーションemoji.jsの起動
ENTRYPOINT ["node", "/app/emoji.js"]
```

Dockerfileに記載する命令①

- FROM命令（続き）：
  - 脆弱性発見のリスクの低減・コンテナの起動の高速化のため、可能な限り小さいサイズのイメージの指定が推奨
  - Dockerの推奨はalpineの公式イメージの指定
    （サイズが非常に小さいLinuxディストリビューション）
  - FROM命令が参照されると、DockerホストにFROM命令に記載されたイメージがダウンロードされ、
    コンテナとイメージが自動的に作成


```
# ベースイメージの指定
FROM almalinux:9

# Maintainerの登録
LABEL maintainer="abc@example.com"

# Node.jsのインストール等
RUN dnf -y install nodejs procps && mkdir /app

# Dockerホストのemoji.jsをコンテナの
# /appディレクトリにコピー
COPY ./emoji.js /app/emoji.js

# ポート3000でリッスン
EXPOSE 3000

# アプリケーションemoji.jsの起動
ENTRYPOINT ["node", "/app/emoji.js"]
```

Dockerfileに記載する命令②

- LABEL命令:
  - イメージに「キー = 値」形式で自由に情報を登録できる命令
  - 書式:

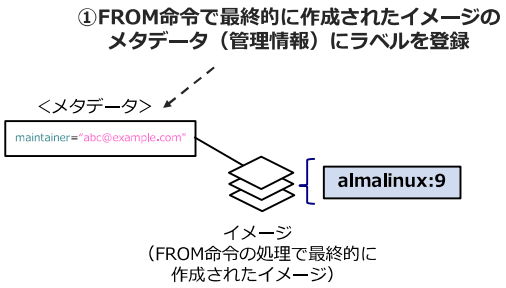
    LABEL キー=値
  - 記載例:

    LABEL maintainer="abc@example.com"
    - Dockerfileやイメージの不明点を問い合わせられるよう、イメージのメンテナー（管理者）の連絡先の登録が推奨
      - その場合はキーに「maintainer」、値に「管理者のメールアドレス」を登録
      - イメージのラベルのキーと値はdocker image inspectコマンドで確認可能

```
# ベースイメージの指定
FROM  almalinux:9

# Maintainerの登録
LABEL  maintainer="abc@example.com"

# Node.jsのインストール等
RUN  dnf -y install  nodejs procps && mkdir /app

# Dockerホストのemoji.jsをコンテナの
# /appディレクトリにコピー
COPY  ./emoji.js  /app/emoji.js

# ポート3000でリッスン
EXPOSE  3000

# アプリケーションemoji.jsの起動
ENTRYPOINT  ["node", "/app/emoji.js"]
```

Dockerfileに記載する命令②

- LABEL命令でバックグラウンドで自動的に実施される処理
  - １つ前の命令で最終的に作られたイメージのメタデータにラベルの情報が登録されるのみ
  - コンテナやイメージは作成されない




```
# ベースイメージの指定
FROM  almalinux:9

# Maintainerの登録
LABEL  maintainer="abc@example.com"

# Node.jsのインストール等
RUN  dnf -y install  nodejs procps && mkdir /app

# Dockerホストのemoji.jsをコンテナの
# /appディレクトリにコピー
COPY  ./emoji.js  /app/emoji.js

# ポート3000でリッスン
EXPOSE  3000

# アプリケーションemoji.jsの起動
ENTRYPOINT  ["node", "/app/emoji.js"]
```