Docker勉強会 vol.1

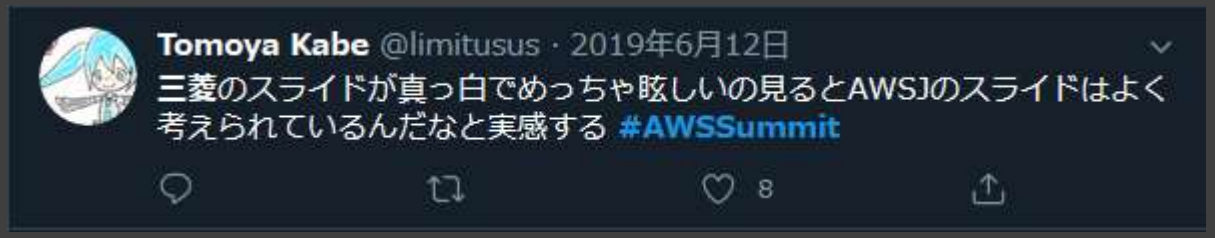
～Dockerの基本～

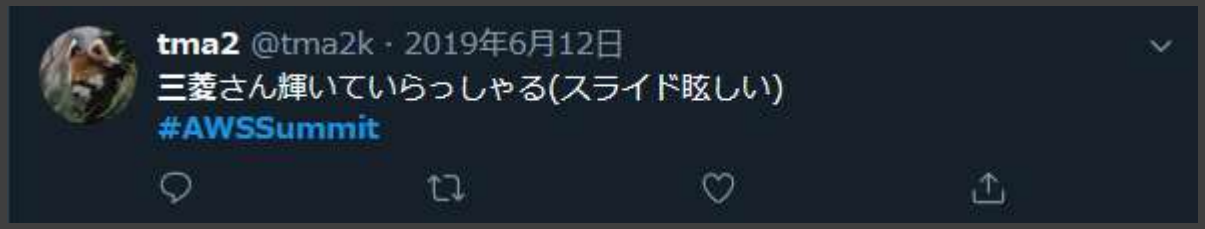
2020年8月21日

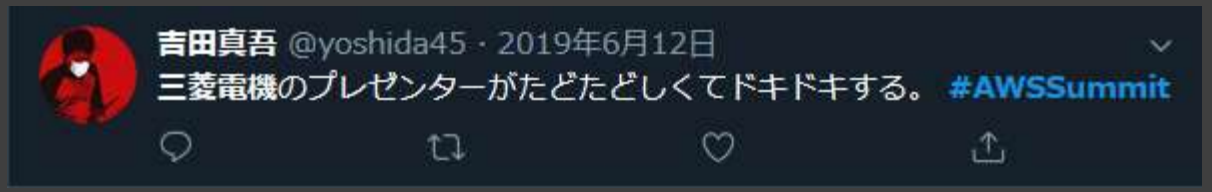
#AWSSummitの経験を生かして

当日資料のみ

- 暗い会場で白背景のスライドは使ってはならない(戒め)







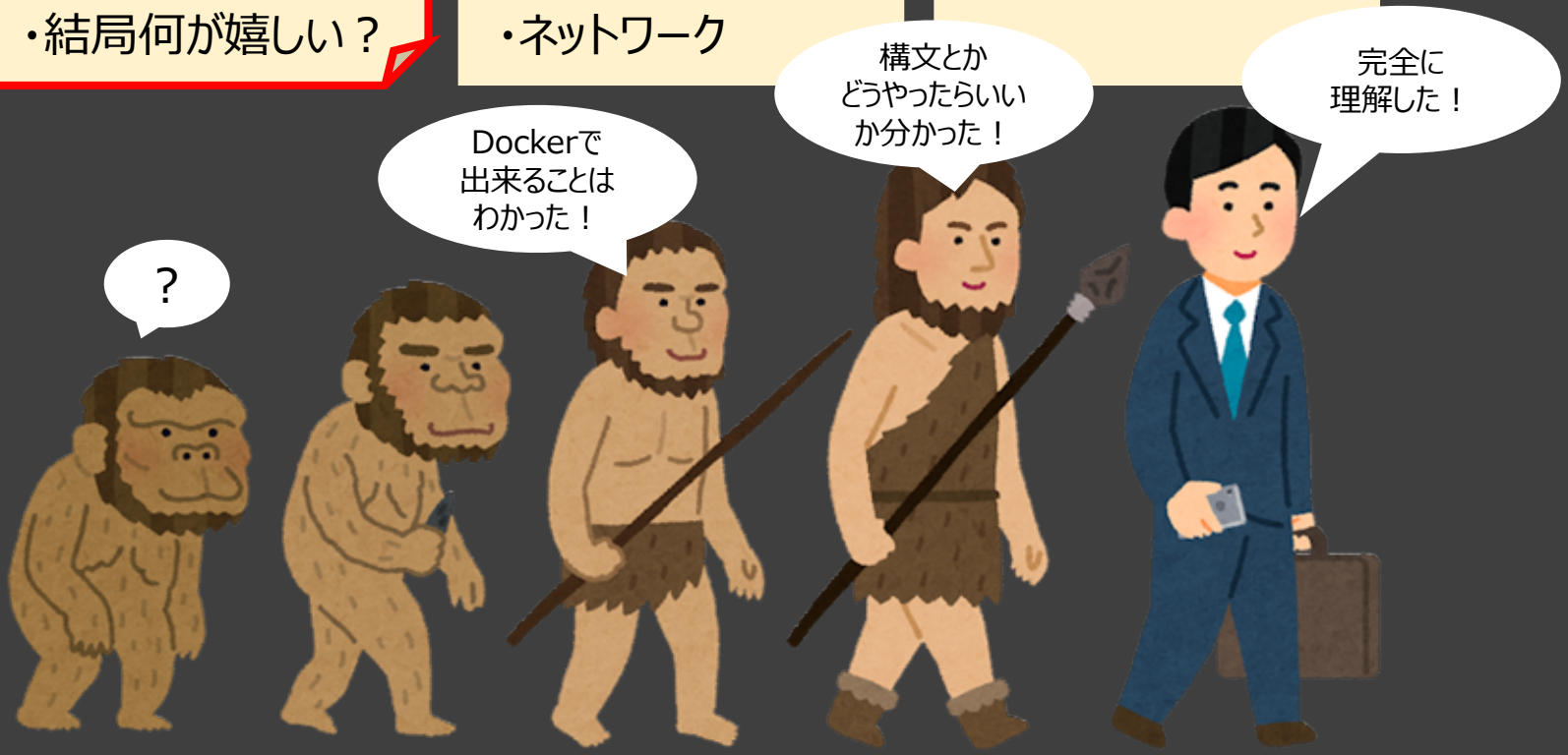
社外発表の時は煽られるので気を付けよう！ヨシ！☞

参考にしたサイトとか

- 本家HP
  - https://docs.docker.jp/engine/introduction/understanding-docker.html
- コンテナデザインパターン
  - https://docs.microsoft.com/ja-jp/azure/architecture/patterns/
- Dockerhub
  - https://hub.docker.com/

アジェンダと今日のゴール

Dockerを知る	使い方を知る	使ってみる
・Dockerって何？ ・アーキテクチャ ・基礎技術 ・結局何が嬉しい？	・イメージの作り方 ・コンテナ起動、終了 ・ボリューム ・ネットワーク	・ハンズオン



イラストに他意はありません

Dockerを知る～Dockerって何?～

- 公式HPの解説

Docker とは何ですか？

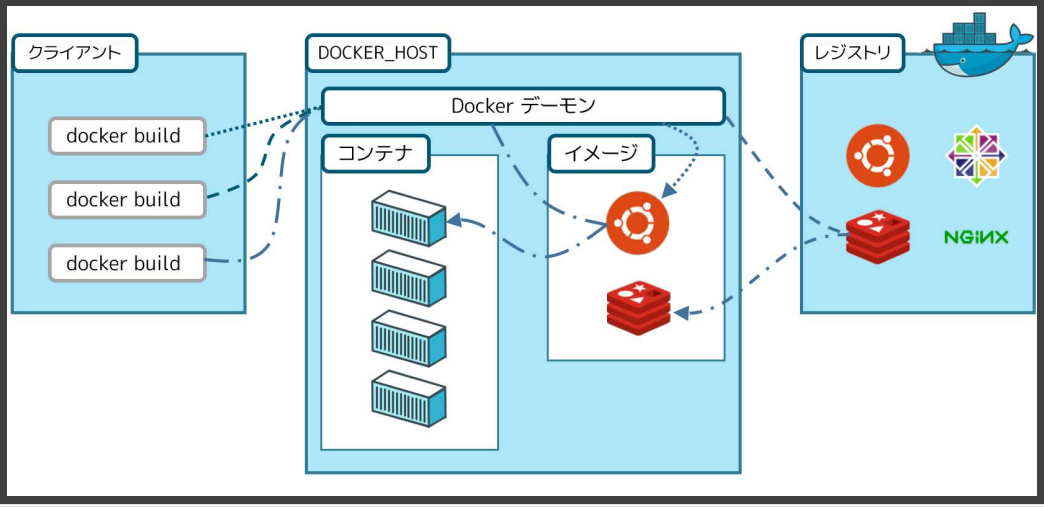
Docker はアプリケーションを開発（developing）・転送（shipping）・実行（running）するための、オープンなプラットフォームです。Docker はアプリケーションをより速く運ぶ（deliver）するために設計されました。Docker を使うことで、アプリケーションを基盤から分離し、アプリケーションを管理するようにインフラを扱えるようにします。Dockerはコードの転送をより速くし、テストを速くし、デプロイを速くし、コードの記述とコードの実行におけるサイクルを短くします。

Docker はこれを、軽量なコンテナ仮想化プラットフォームを使ったワークフローとツールの連携で実現します。これがアプリケーションの管理とデプロイの手助けになるでしょう。

- 雑に言うと
  - Docker＝コンテナ(＝アプリ)管理のプラットフォーム
  - コンテナ＝基盤から分離された実行コンポーネント

Dockerの基本を知る～アーキテクチャ～

- 本家HPのアーキテクチャ図

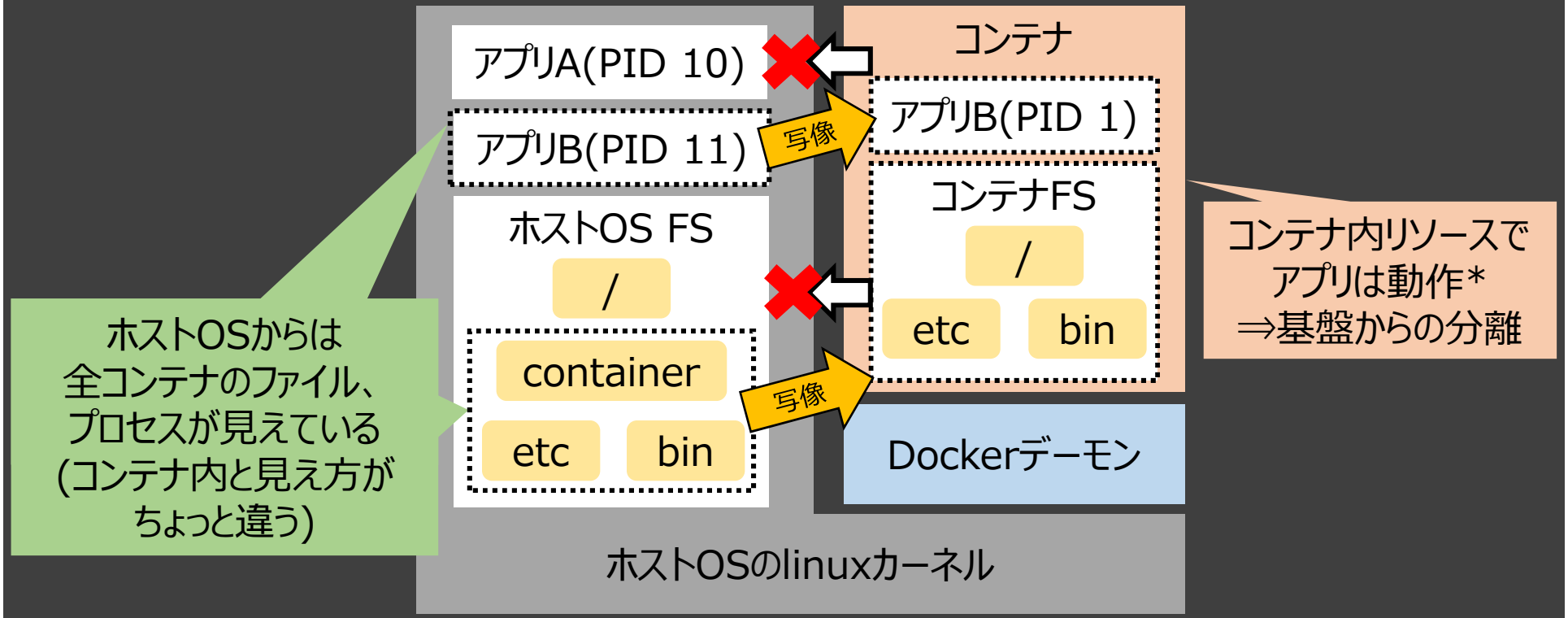


要素名	機能
Dockerデーモン	Dockerコンテナの構築、実行、配布を担う /var/run/docker.socketでRestful APIを公開している
クライアント	dockerコマンド。Dockerデーモンと通信する
イメージ	コンテナのテンプレート。OS*、アプリやM/Wが格納されている。読専
コンテナ	イメージをもとに作成された実行コンポーネント ストレージ、ネットワークなど起動時にコンテナの特徴を付与する
レジストリ	パブリックなイメージ置き場。Dockerhubなど

*本家HPにOSと書いているけど、厳密にはOSじゃないと思う

Dockerを知る～アーキテクチャ～

- コンテナにフォーカスを当ててみる
  - コンテナはホストOSとカーネルを共有
  - ホストOSから見ると、コンテナは**ただのプロセスとファイル群**
  - コンテナからホストOS、他コンテナは見えない*



*設定や構築方法によっては例外あり

補足：ディストリビューションが違うのに動く？

- ホストOS Ubuntu、コンテナ CentOSみたいなのができる
  - ディストリビューションが違ってもカーネルABIが一緒
    - ABI=Application Binary Interface
      アプリとカーネルのシステムコールI/F
    - ディストリビューションによってパッケージ管理方法とか違うけど、
      使ってるシステムコールが同じなのでダイジョウブ！

- 逆に言うと、カーネルが一緒じゃないと絶対に動きません
  - FreeBSDはlinuxバイナリ互換機能で頑張って動くらしい
  - Docker for windowsは、Hyper-VなどでLinux OSを立ち上げ、Linux OS上のDockerデーモンとやり取りしている
  - WSLに期待！

Dockerを知る～アーキテクチャ～

- ここまでのまとめ
  - Dockerはコンテナ管理プラットフォーム
  - コンテナは実行コンポーネント
    - イメージから作られる
    - アプリのリソースとプロセス群
    - ホストOS、周りのコンテナのことは認識しない
  - ↑↑があるから
    - 基盤から分離できる！
    - イメージを持ってくるだけで簡単にアプリが動かせる！

Dockerを知る～基礎技術～

- 基盤からの分離を実現するための基礎技術
  - cgroup
  - namespace
    - cgroup、namespaceともにLinuxカーネルの機能
  - ↑が分かればDockerでできることの大体は

Dockerを知る～基礎技術～

- cgroup(コントロールグループ)
  - リソース割り当てを制御・監視する単位。プロセスで構成される
  - 木構造で設定を継承
    - 親の設定を継承。子は親が許可したものだけ設定可能
  - 設定可能項目は以下(設定方法や詳細は割愛)
    - 割愛しているが、このほかにもレポート系のサブシステムもある

No.	サブシステム名	意味
1	blkio	ブロックデバイスIOへのアクセス(帯域幅)を設定
2	cpu	cgroup間のCPU時間の配分を設定
3	cpuset	割り当てるCPU、メモリノードを設定
4	devices	デバイスへのアクセス可否、権限(RW)を設定
5	net_cls	発信するパケットにタグをつけ、タグ毎に優先度をつけられるようにする
6	net_prio	用いるNIC毎にトラフィックの優先度を設定
7	**ns**	**プロセスが所属するnamespaceを設定**

Dockerを知る～基礎技術～

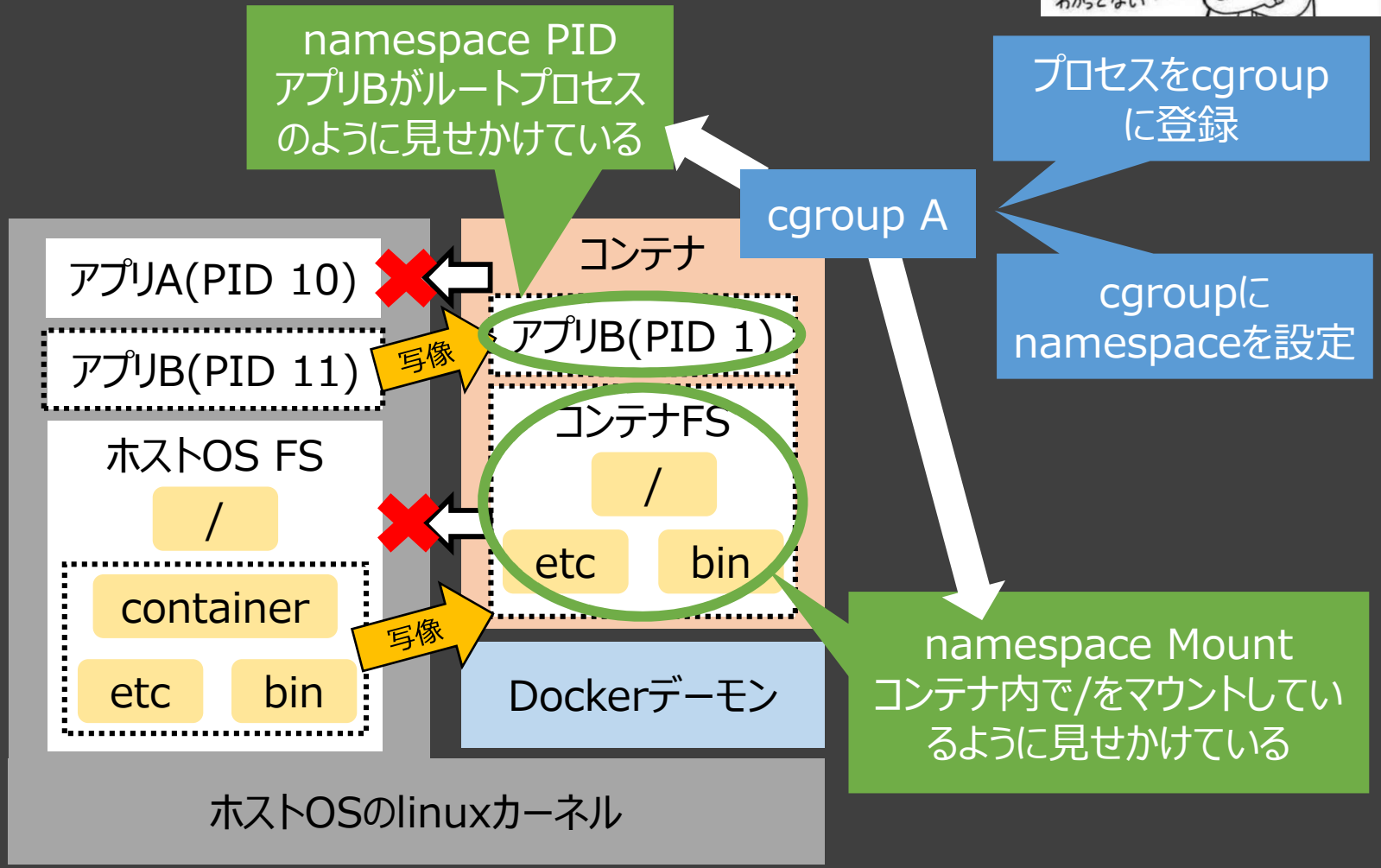
- namespace
  - namespaceに所属するプロセスに対して、自分たちが専用の分離されたグローバルリソースを持っているかのように見せる仕組み*
    - グローバルリソース毎にnamespaceが存在
      - IPCは見えるようにするけど、Networkは別とかできる
    - グローバルリソースの一覧は以下

No.	namespace	意味
1	IPC	System V IPC, POSIX メッセージキュー
2	Network	ネットワークデバイス、スタック、ポートなど
3	Mount	マウントポイント
4	PID	プロセス ID
5	User	ユーザー ID とグループ ID
6	UTS	ホスト名と NIS ドメイン名

*man(7) namespaceより

Dockerを知る～基礎技術～

- まとめ
  - ～アーキテクチャ～で説明した図はこうなっていた



Dockerを知る～結局何が嬉しい?～

- 開発シーン
  - イメージがあれば、何回でもコンテナを作れる
    - 開発中に汚れた環境をいつでも初期状態にやり直せる
      ⇒「俺の環境では動くんだけど」からの脱却
    - 開発→本番環境への移行も簡単
      - 環境変数など外部から設定を与えられるようにすると
        開発・本番環境の切替え作業がほぼ不要

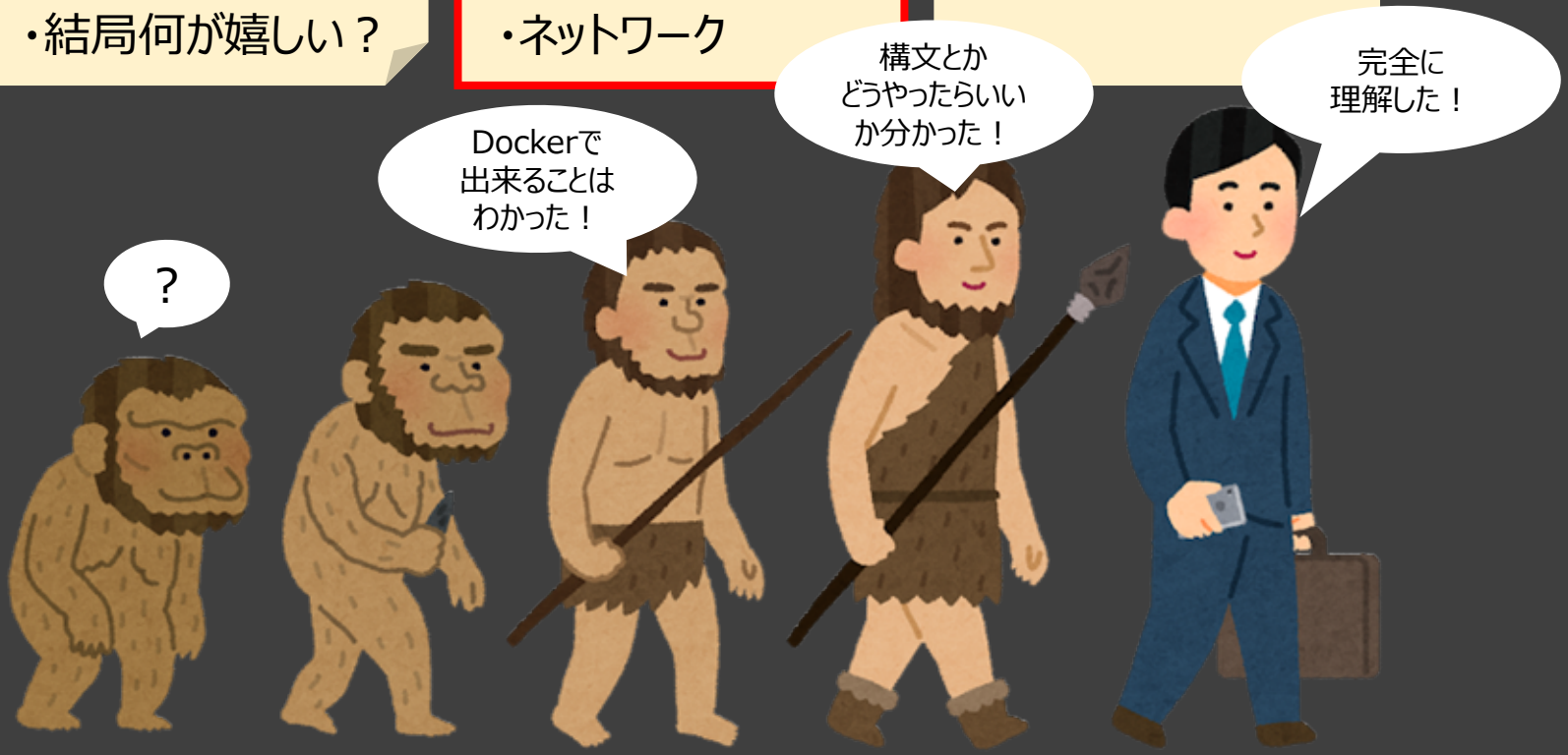
  - コンテナによってリソースが隔離されているので環境を汚さない
    - 複数バージョンのランタイム(例:python2、3)の共存
    - ライブラリの依存関係競合も起こしにくい

Dockerを知る～結局何が嬉しい?～

- 運用シーン
  - 機能でコンテナを分離すれば・・・
    - バージョンアップ対象のコンテナだけ差替え
      - 変なことになったら、旧Ver.のコンテナに差替えておしまい
    - 機能の抜き差しも簡単
      - コンテナデザインパターン
        - サイドカー、アンバサダー、アダプターetc.
    - スケールアウトも簡単
      - 機能の実行環境がワンセットになっているので、レプリカをどこでも作れる
        - Docker swarm、Kubernetes etc.

アジェンダと今日のゴール

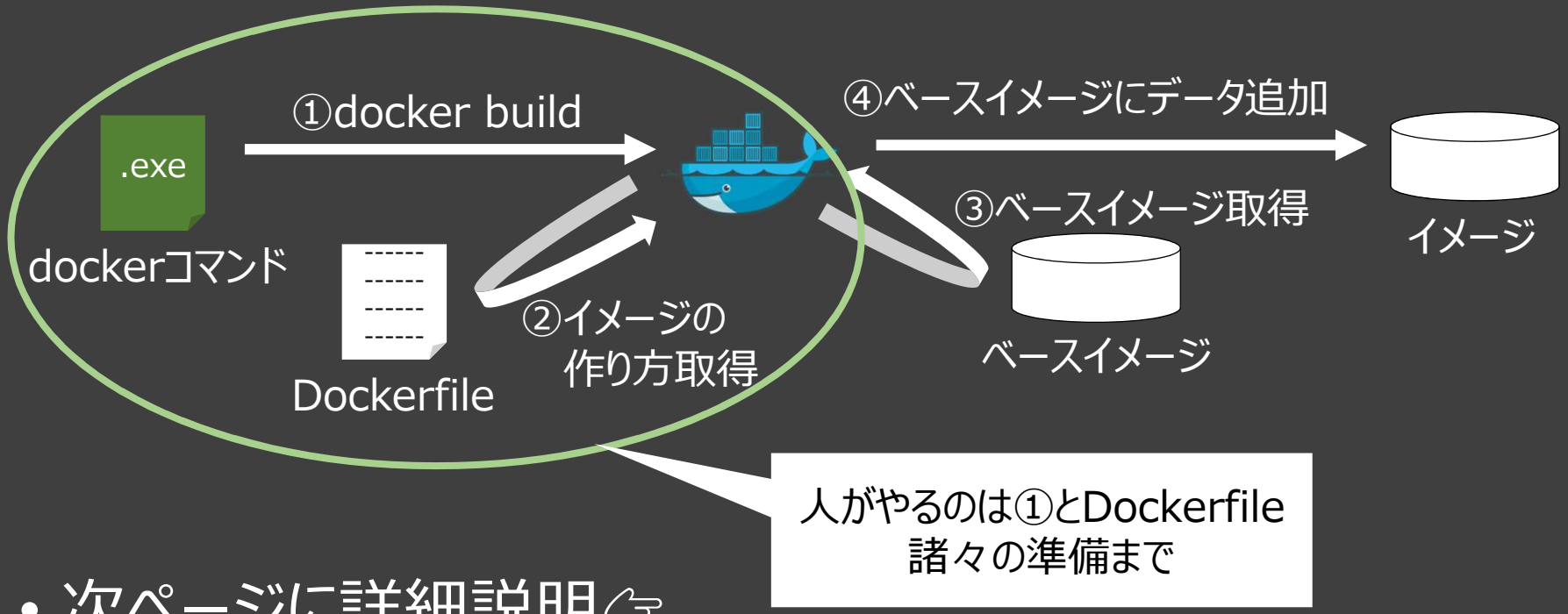
Dockerを知る	使い方を知る	使ってみる
・Dockerって何？ ・アーキテクチャ ・基礎技術 ・結局何が嬉しい？	・イメージの作り方 ・コンテナ起動、終了 ・ボリューム ・ネットワーク	・ハンズオン



イラストに他意はありません

使い方を知る～イメージの作り方～

- Dockerfileに基づき、ベースイメージに対してデータやメタデータを追加することで新規イメージを作成する

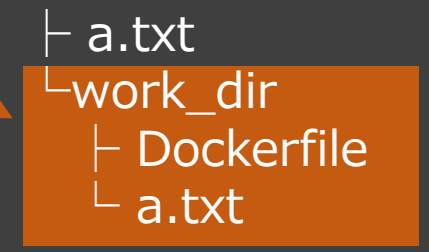


- 次ページに詳細説明☞

使い方を知る～イメージの作り方～

- ①docker build [option] <path> | <URL>
  - 細かいオプションはココ
    - https://docs.docker.jp/engine/reference/commandline/build.html
  - <path>、<URL>で指定したコンテキストを使ってイメージを構築
    - コンテキスト
      - <path>や<URL>で指定した場所にあるファイル群
      - build時はコンテキスト内のファイル群のみ参照可能
        - ~/をコンテキストとしたとき、/etcのファイルは参照不可
    - デフォルトでは、コンテキスト直下にあるDockerfileを読む

```
@home> sudo docker build –t sample:1.0 ./work_dir
#  ./work_dir/Dockerfileに基づきイメージが作成される
#  -t sample:1.0は、イメージ名:TAGを設定している(後述)
```



使い方を知る～イメージの作り方～

- ②イメージの作り方取得
  - Dockerfileのリファレンスはココ
    - https://docs.docker.jp/engine/reference/builder.html
  - よく使う命令

命令	処理
FROM <イメージ名>[:<TAG>]	ベースイメージの指定。TAG指定がない場合はlatestタグが使われる
RUN <コマンド>	イメージを作る際にコンテナ中で実行するコマンド
ADD <ファイル> or ADD <フォルダ> or ADD <URI>	イメージにコンテキスト内のファイルを追加する ・リモートファイル(https://～)利用可能 ・圧縮ファイルを展開してくれる
COPY <ファイル> or COPY <フォルダ>	イメージにコンテキスト内のファイルをコピーする ・ADDの逆
ENV <環境変数名> <値>	イメージの環境変数を設定する
ENTRYPOINT <コマンド>*	コンテナ実行時に実行するコマンド
CMD <コマンド>* *コマンドは文字列 or Json形式	コンテナ実行時に実行するコマンド ・ENTRYPOINTが設定されている場合は、ENTRYPOINTの引数扱いになる 例) ENTRYPOINT [“ping”]、CMD [“8.8.8.8”, “–c”, “100”] ⇒ping 8.8.8.8 –c 100 が実行される

使い方を知る～イメージの作り方～

- ③ベースイメージ取得
  - DockerfileのFROMに書かれているイメージを取得する
    - ローカルに<イメージ名>:<TAG>に一致するものがない場合リモート(Dockerhub)から自動で取得してくる

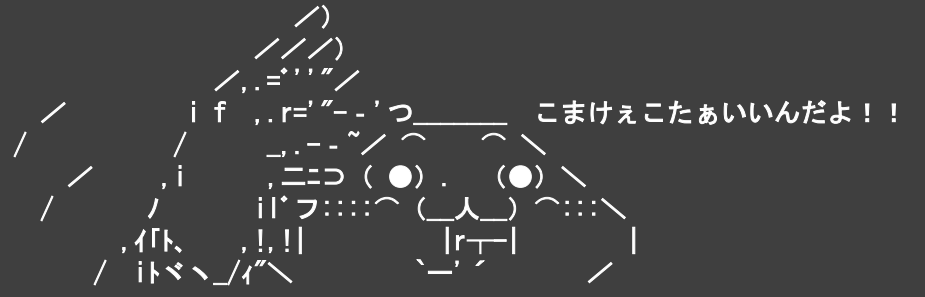
```
@home> sudo docker images
REPOSITORY    TAG      ...      SIZE # 空でした

@home> echo “FROM centos” | sudo docker build –t test -

@home> sudo docker images
REPOSITORY    TAG      ...      SIZE
test          latest   ...      215MB # 作ったイメージ
centos        latest   ...      215MB # ベースイメージ
```

    - ベースイメージcentosがローカルリポジトリに追加されている
      - buildの過程でDockerhubからpullされている

使い方を知る～イメージの作り方～

- ④ベースイメージにデータ追加
  - Dockerfileに基づいてベースイメージにデータを追加
    - 必要なライブラリ(RUN yum、RUN apt)
    - 必要なファイル(ADD、COPY)
    - 起動時コマンド(CMD、ENTRYPOINT) とか
  - 作ったイメージは、ベースイメージ+差分で管理されている
    - OverlayFS
      - Dockerfileの命令毎にレイヤーができる
        - Tmpファイル作って消したつもりでもレイヤーは残るので Dockerfileの書き方次第でサイズが大きくなったり。。

          ⇒今日はまずやってみる！



使い方を知る～イメージの作り方～

- Hello Worldイメージ サンプル

```
home
 ├ Dockerfile
 └ hello.sh
```

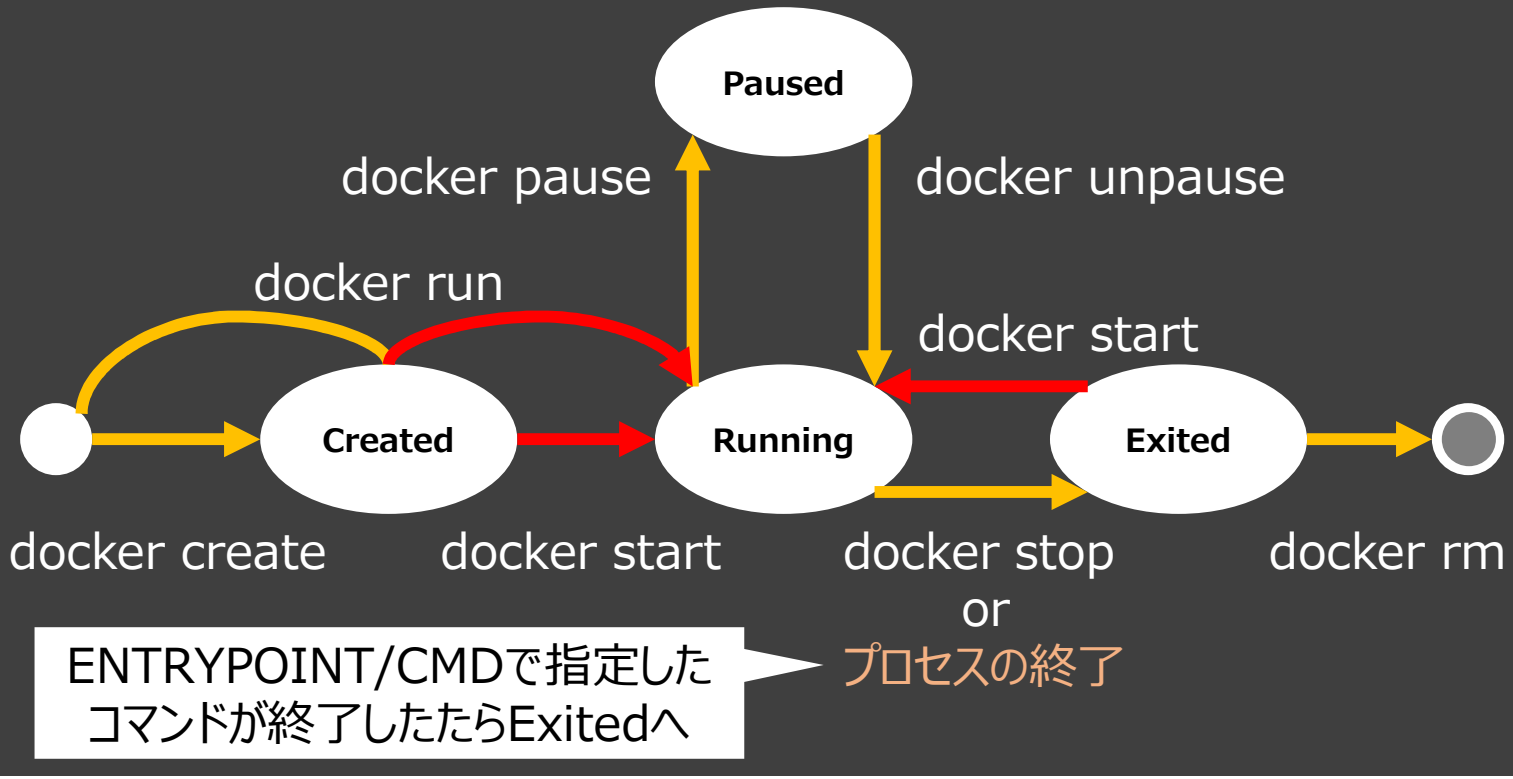
```
FROM centos:latest
ADD hello.sh /root/hello.sh
WORKDIR /root
RUN chmod +x hello.sh

ENTRYPOINT [“./hello.sh”]
CMD [“world”]
```

```
#!/bin/bash
echo hello $@
```

```
@home> sudo docker build –t hello:1.0 .

@home> sudo docker images
REPOSITORY    TAG      ...      SIZE
hello         1.0      ...      215MB # 作ったイメージ
centos        latest   ...      215MB # ベースイメージ

@home> sudo docker run hello:1.0
hello world
```

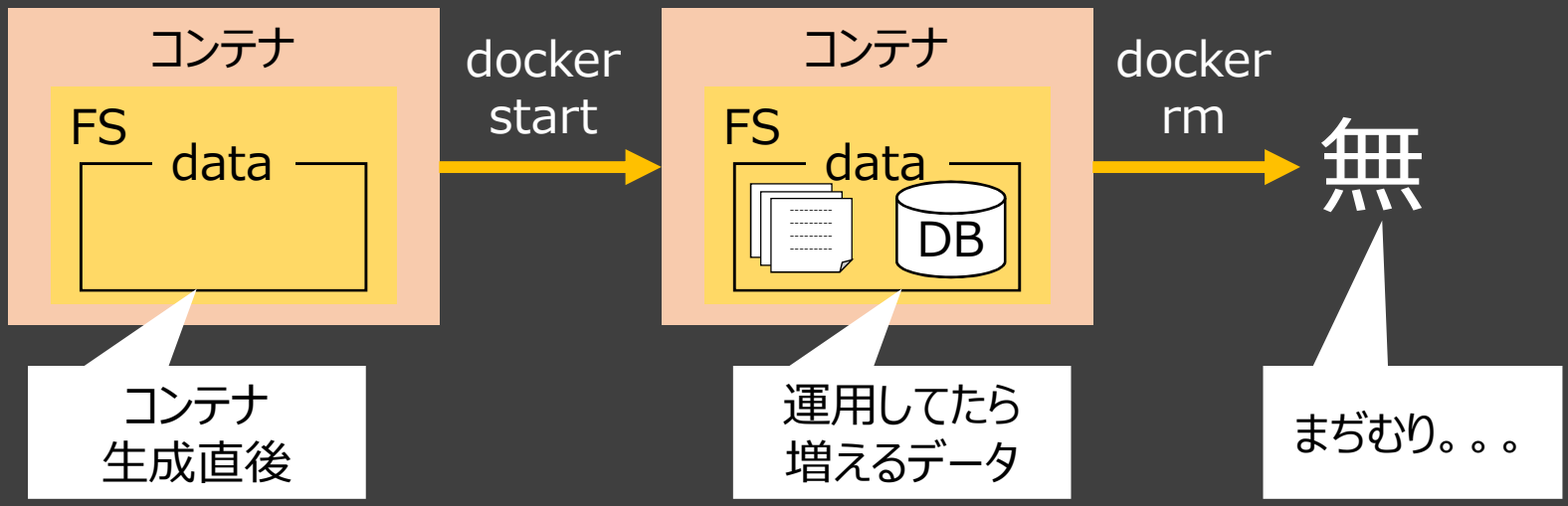
使い方を知る～コンテナの起動・終了～

- コンテナのライフサイクルと遷移するコマンド/イベントを示す
  - ENTRYPOINT/CMDは赤線の遷移時に実行される
  - コンテナは明示的に削除(docker rm)しないと残り続ける
    - コンテナが削除されるまで、コンテナ内のファイルは保存される
  - dead、restarting、removingみたいな奴は除外



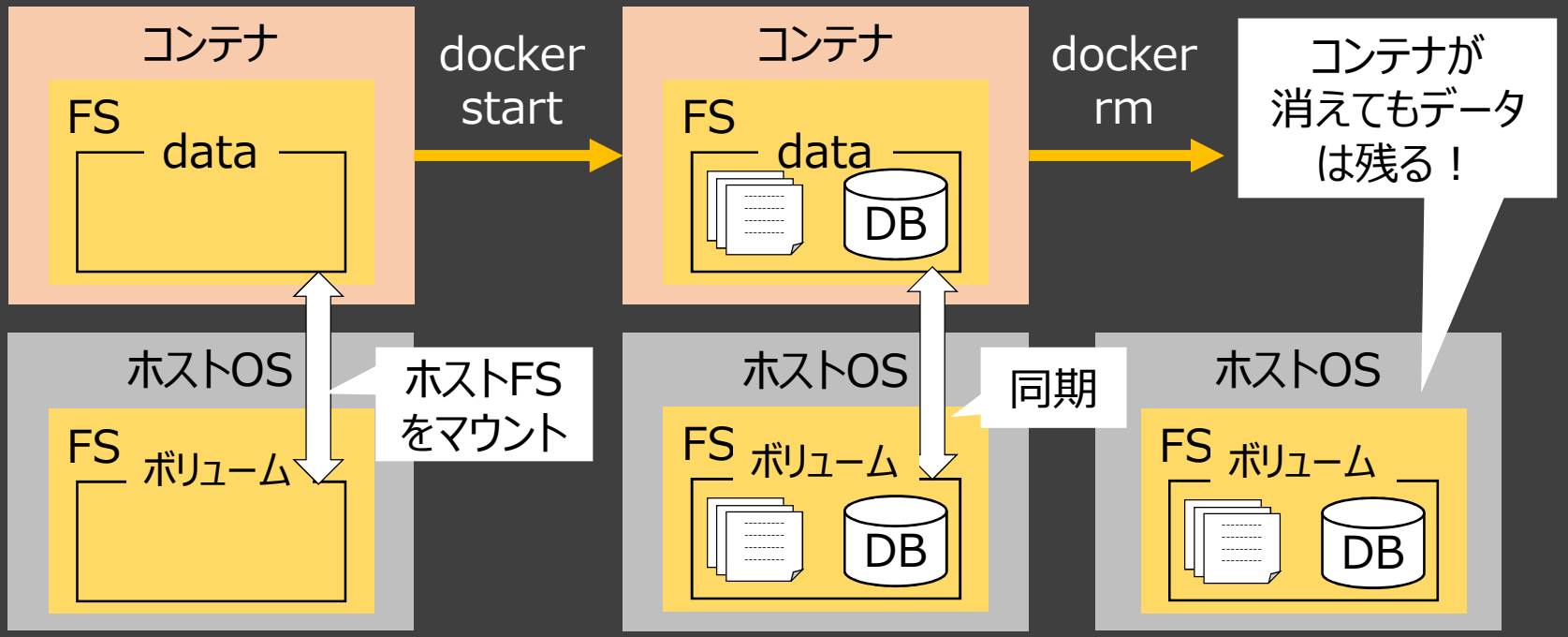
使い方を知る～ボリューム～

- 「コンテナが削除されるまで、コンテナ内のファイルは保存される」
  - 逆に言うと、コンテナ消したらファイルが消える
    - ログ、DBとかコンテナ起動後に増えるファイルもある
  - コンテナ消すときって結構ある
    - コンテナのバージョンアップ(イメージが別なので別コンテナ)
    - コンテナを動かすPCを変える etc.



使い方を知る～ボリューム～

- ボリュームを使ってデータを永続化する
  - 雑に言うと、ボリューム=ホストOSのFS(ファイル、ディレクトリOK)
  - ボリュームは、複数コンテナから同時にマウントできる
  - コンテナが消えてもボリュームは消えない
    - 逆に言うと、明示的に消さないとゴミが(ry
    - 作られているボリュームは”docker volume ls”で確認



使い方を知る～ボリューム～

- ボリュームの作り方
  - ×DockerfileのVOLUMEに記載
    - ホストOSのどこをマウントするかの指定不可
      - docker buildの時点でどこで動くかわからない
  - △sudo docker volume create
    - run/createの時に作れるから特に。。。
  - ○sudo docker (run|create) --mountオプション
    # --volumeオプションもあるけど非推奨になった
    - 3種類のマウントタイプが存在
      - Volumes
      - Bind mounts
      - tempfs mounts

使い方を知る～ボリューム～

- Volumes
  - ホストOSのデータを共有する必要がない場合(Docker推奨)
  - ボリュームの実態は/var/lib/docker/volumes/以下にある

```
@home> sudo docker run –-mount type=volume,src=v1,dst=/root ...
```

- Bind mounts
  - ホストOSのファイル/ディレクトリをbindする
    - ホストOSとファイルを共有・同期したいときに使う(/etc以下とか)

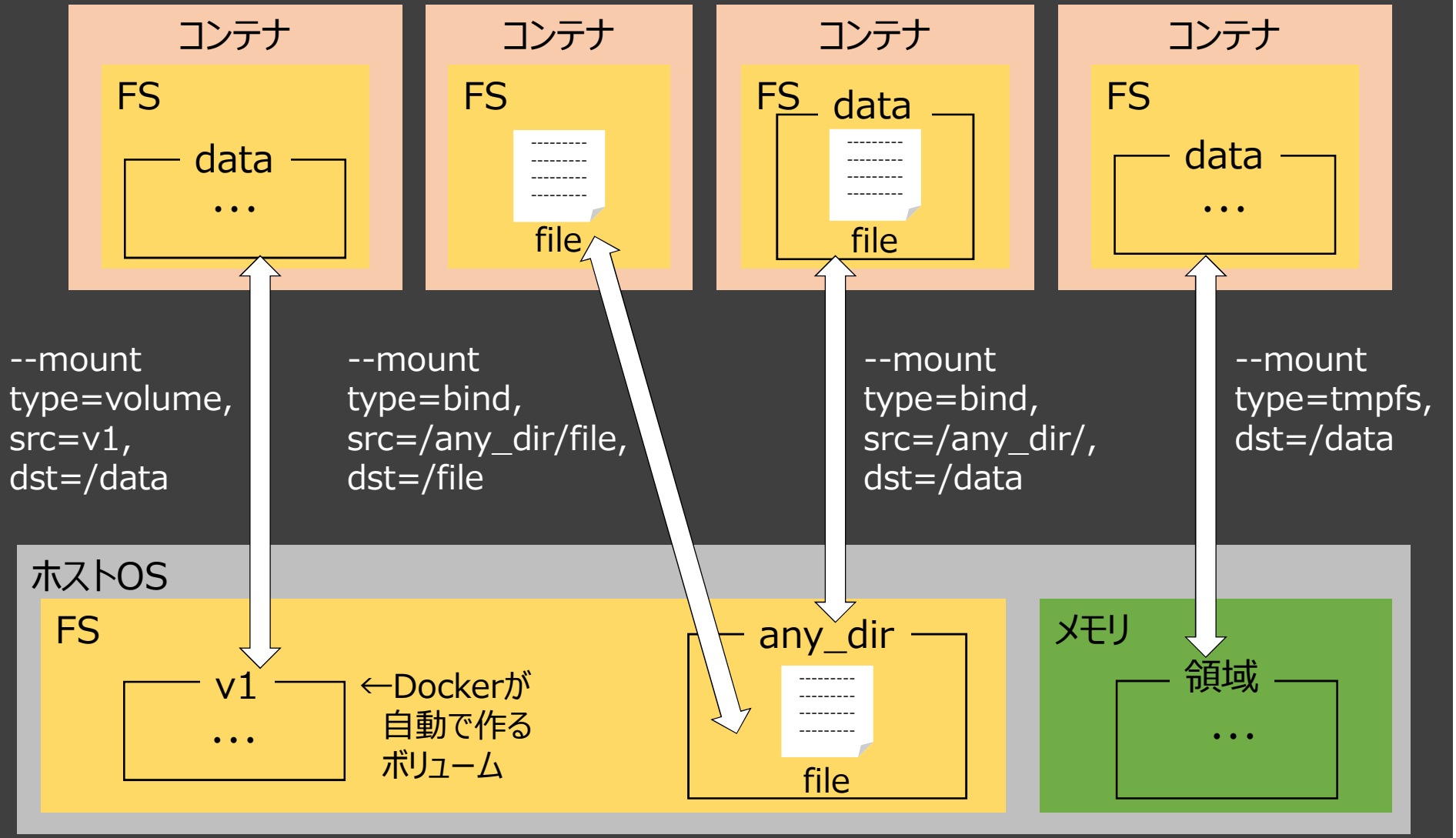
```
@home> sudo docker run –-mount type=bind,src=/etc/,dst=/etc ...
```

- tempfs mounts
  - メモリ領域をディレクトリとしてマウント(/tmpと一緒)
  - コンテナを停止すると消える

```
@home> sudo docker run –-mount type=tmpfs,dst=/tmp ...
```

使い方を知る～ボリューム～

- まとめ

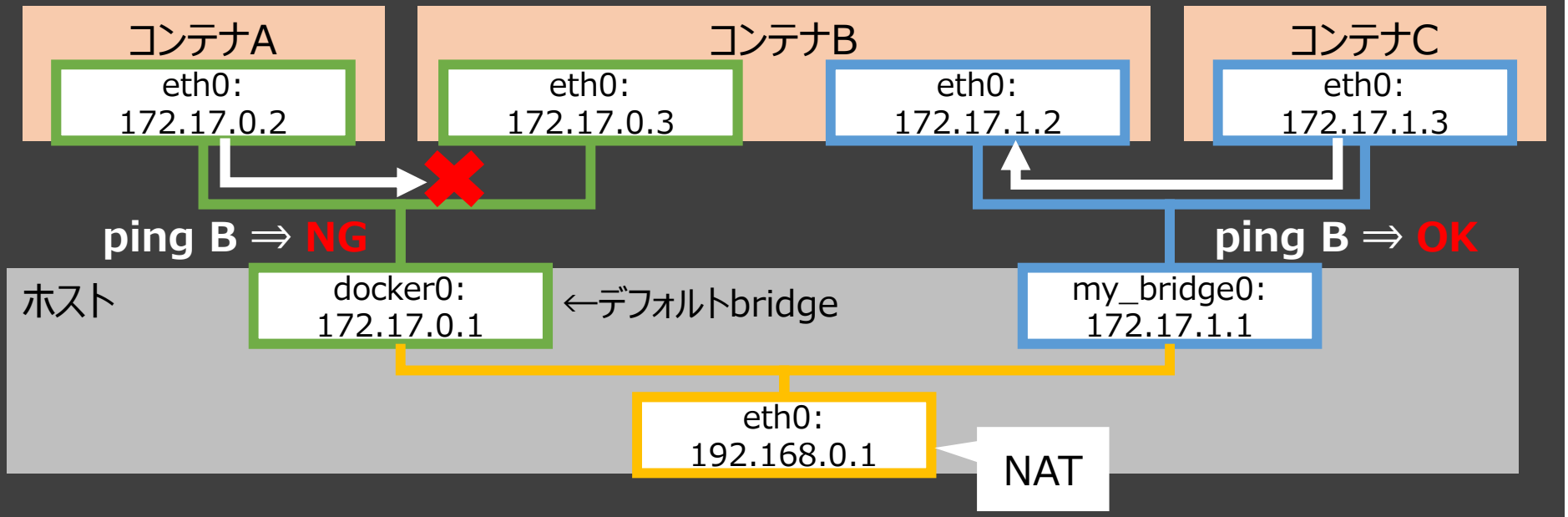


使い方を知る～ネットワーク～

- コンテナを外部に公開するためにはネットワーク設定が必須
  - ネットワークは5種類存在
    - 今日説明するのは以下の3つ
      - bridge
        - デフォルト
        - ユーザ定義
      - host
    - その他は。。。
      - container
        - 他のコンテナのネットワークスタックを利用
      - none
        - ネットワークに接続しない

使い方を知る～ネットワーク～

- bridge
  - コンテナだけのネットワーク
  - ホストOSがNATになっている
    - 外部⇒コンテナ通信はポートフォワーディングが必要
  - デフォルトとユーザ定義の違い
    - ユーザ定義ネットワークはネットワーク内で名前解決可能
      - 昔は--linkで名前解決もどきをしていたけど非推奨に



使い方を知る～ネットワーク～

- bridge設定
  - ユーザ定義ネットワークの作成

```
@home> sudo docker network create –d bridge my_bridge
```

  - コンテナをユーザ定義ネットワークに所属させる
    - オレンジ文字を削除するとデフォルトbridgeに所属
    - ユーザ定義brigdeではコンテナ名がホスト名となる

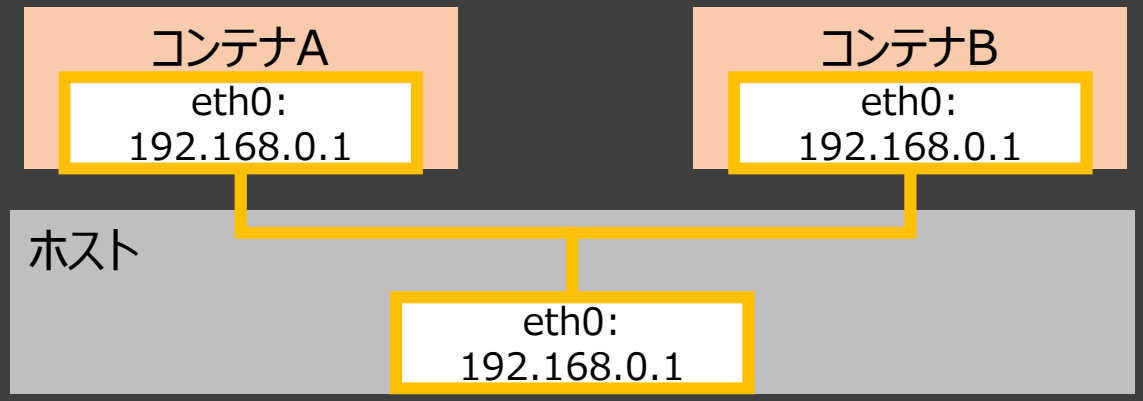
```
@home> sudo docker run –t –d –-net my_bridge centos bash
@home> sudo docker run –t –d –-net my_bridge centos bash
```

  - ポートフォワーディング設定をする
    - -pオプションで設定
      - ホストIP:8080はコンテナIP:80にフォワーディングされる

```
@home> sudo docker run –t –d -p 8080:80 centos bash
```

使い方を知る～ネットワーク～

- host
  - ホストのネットワークI/Fをコンテナも使う
    - 外部⇒コンテナ通信の設定は不要(ホスト上のプロセスと一緒)
    - 開けるポートが他と被らないように注意する必要あり
      - bridgeの場合複数コンテナが同じポート開けていても無害

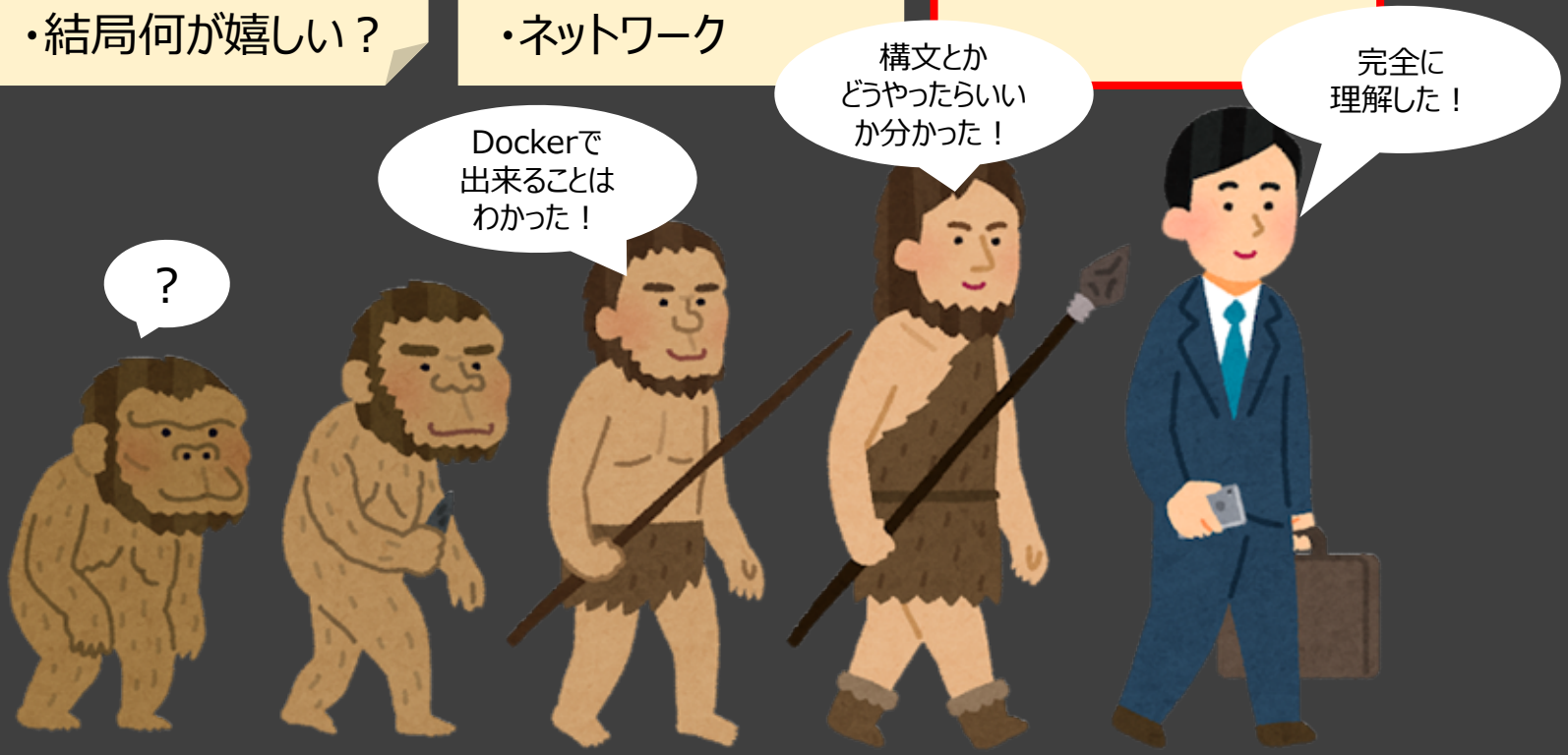


  - コンテナをホストのネットワークに所属させる

```
@home> sudo docker run –t –d –-net host centos bash
```

アジェンダと今日のゴール

Dockerを知る	使い方を知る	使ってみる
・Dockerって何？ ・アーキテクチャ ・基礎技術 ・結局何が嬉しい？	・イメージの作り方 ・コンテナ起動、終了 ・ボリューム ・ネットワーク	・ハンズオン



イラストに他意はありません