Dockerで仮想環境の用意

DL-Box等の共有のリソース上で他人とは違う環境(ドライバのバージョン違いなど)が欲しい場合に、Dockerを用いて自分専用の仮想環境を作る手順を説明する。

Dockerとは

ざっくり言ってしまえばOSレベルで仮想環境を用意できる便利なツール。Dockerを用いることで(使いたいマシン内で仮想的に)別のPCを用意できるといえばイメージしやすいかも。ただし、CUDAはマシン本体に入っているものより新しいバージョンのものは動かない模様。

Docker公式ドキュメント

環境構築までの流れ

おおまかには

1. Dockerのインストール
2. Dockerイメージのダウンロード or ビルド
3. Dockerコンテナの作成

となる。Dockerコンテナが仮想環境に相当するものになる。研究室内のDL-Boxについては恐らく全てにDockerをインストール済みなので1の手順は特に気にする必要はないはず。大体の環境は誰かが既にイメージを用意しているのでそれを探してきてダウンロードした方が楽。

Dockerをsudoなしで実行できるようにする

もともとsudoなしでDocker関連のコマンドが使えるなら設定しなくても大丈夫。

```
# dockerグループがなければ作る
sudo groupadd docker

# 現行ユーザをdockerグループに所属させる
sudo gpasswd -a $USER docker

# dockerデーモンを再起動する
sudo systemctl restart docker

# exitして再ログインすると反映される
exit
```

Dockerイメージの用意

Dockerイメージはネット上で公開されているものをダウンロードするか、自分でビルドすることで作成できる。

- ダウンロードしてイメージを用意する

  代表的な機械学習ライブラリであれば大体 nvidiaのサイト からダウンロードできる。ここになくても、例えばpython 3.7が動く環境が欲しいとき「docker python 3.7」と検索すれば大抵ヒットする。

  ```
  # イメージのダウンロード
  docker pull [イメージ名]:[タグ]
  ```

  イメージ名はその名の通りイメージの名前を指す。タグについてはそのイメージのバージョンのようなものと思えば良い。タグは省略されてる場合もある。以下に具体例をいくつか示しておく。イメージ内部のドライバやソフトのバージョンについてはReleaseNotes(例：PyTorch)などに書かれているので、古いバージョンのCUDAなどが必要な際はその辺りを参考にダウンロードすると良い。

  ```
  # CUDA: 11.0、PyTorch: 1.7.0
  docker pull nvcr.io/nvidia/pytorch:20.08-py3

  # CUDA: 10.2、PyTorch: 1.5.0
  docker pull nvcr.io/nvidia/pytorch:20.03-py3

  # Minecraftサーバー
  docker pull itzg/minecraft-server
  ```

- Dockerfileを用いてビルドしてイメージを用意する

  Dockerfileを記述してカスタマイズした環境を用意することもできる

  ```
  docker build -t [イメージ名]:[タグ] [Dockerfileのパス]
  ```

  Dockerfileの記述法は以下あたりを参考に。

  - Dockerfile 記述のベスト・プラクティス(公式)
  - 【入門】Dockerfileの基本的な書き方
  - Dockerfileの書き方を深掘りしつつまとめてみた

特にエラーがなければイメージの用意は出来たはず。以下のコマンドで用意したイメージがあるかを確認する

```
docker images
```

Dockerイメージの削除

Dockerイメージはそこそこの容量があるので不要なイメージは削除したほうが良い。ただ、他の人が使っているかもしれないので削除の際はしっかり確認すること

```
docker rmi [イメージ名]:[タグ]
```

イメージ名やタグは以下のコマンドで表示される表のうちREPOSITORYやTAGの部分

```
docker images
```

Dockerコンテナを作成

用意したイメージをベースにコンテナを用意する。ベースのイメージは[イメージ名]:[タグ]という形で指定。
イメージ名やタグの確認方法は上のDockerイメージの削除の項目と同じ

```
docker run [オプション] イメージ [コマンド] [引数...]
```

詳細なオプションについては公式ガイドを参照

よく使うやつ

```
# 大体の環境はこのオプション入れておけば大丈夫。他に欲しいオプションがあれば上の公式ガイドとかを参考に。
docker run -d --gpus all -it --name="example1" --shm-size=8g -v [ホストディレクトリの絶対パス]:[コンテナの絶対パス] [イメージ名]:[タグ]

# -d :コンテナをバックグラウンドで実行
# --gpus all :マシンのGPUを全部使う
# -it :コンテナ内の標準入力に擬似TTYを接続。対話シェルが使える。
# --name="example1" :作成するコンテナの名前。この例だとexample1というコンテナが作成される。作る際は自分の名前を入れるなど分かりやすくしよう。当たり前だが他のコンテナと被らないように。
# --shm-size=8g :共有メモリを8GB用意。なくても良いオプションだが、学習などの際に共有メモリが足りない的なエラーを吐く可能性が高くなる。大体は8GBで足りるが、それでも足りなかったら適当に増やせばOK。
# -v [ホストディレクトリの絶対パス]:[コンテナの絶対パス] :ディレクトリのマウント。[ホストディレクトリの絶対パス]に自分のプログラムやデータがあるディレクトリを指定することで、Docker内では[コンテナの絶対パス]でアクセスできる。極力他人のデータが入ってる場所や共有ディレクトリは指定しないように。逆にここを設定しないとマシンに入ってるデータが扱えなくて色々困ると思う。

# --rm :コンテナ停止時に自動でコンテナを削除してくれる。ここには無いが、適当にテストがてらコンテナを立てて後で削除するとかであれば、このオプションを入れておくことで削除の手間が省ける。
```

こちらも特にエラーを吐かなければ作成できたはず。以下のコマンドで作成されたコンテナがあるか確認。
NAMESの欄に指定した名前があればOK。

```
docker ps --all
```

Dockerコンテナに接続

以下のコマンドで起動中のコンテナを確認できる。用意したコンテナが起動しているかどうかを確認

```
docker ps
```

起動していたら以下のコマンドでコンテナに接続

```
docker exec -it [コンテナ名] bash
```

環境に入れたらOK。適当なコマンドで必要なソフトやドライバがあるか、バージョンは合っているかなどを確認しよう。

Dockerコンテナのその他の操作

```
# 起動中のコンテナを表示
docker ps

# 停止中のものを含めた全てのコンテナを表示
docker ps --all

# 起動
docker start [コンテナ名]

# 停止
docker stop [コンテナ名]

# 接続 - 個人用のコンテナであればどちらでも良いが、基本的に上で紹介したexecの方が扱いやすい
docker attach [コンテナ名]

# 名前の変更 - 対象のコンテナが起動している場合は先に停止
docker rename [before] [after]

# 削除 - 対象のコンテナが起動している場合は先に停止
docker rm [コンテナ名]
```
