RDEToolKitを用いたRDE構造化処理用 Dockerコンテナ作成手順書

リリース**1.3.0 (20250526)**

国立研究開発法人 物質・材料研究機構

目次

1. はじめに

ようこそ、RDE構造化処理の開発の世界へ。

本書は、RDE構造化処理開発の一環としての、Dockerイメージ作成から、Dockerイメージを利用した開発方法までを解説します。

> RDE構造化処理は、主にPython言語を利用して構築されます。そのため本書ではPython言語を利用する形で記述します。

1.1 概要

RDEの構造化処理プログラムは、RDEシステム上ではDockerコンテナとして利用されます。そのため、どのような開発手法をとったとしても、最終的には、Dockerイメージを作成する必要があります。

多くの場合は、先にローカル環境(Windows,MacあるいはLinux等)上にPythonをインストールし、構造化処理プログラムを開発します。開発が完了した時点でそれを含めたDockerイメージを作成します。

一方で、ローカル環境にPython実行環境を導入できない、あるいは想定するPythonバージョンの用意が容易でないなどの場合は、開発のためのDockerイメージを作成し、それを利用して構造化処理プログラムを作成します。

Dockerイメージ をDockerコンテナとして起動し、その中にあるファイルを変更しても、Dockerコンテナを再起動した時点で元に戻ってしまいます。そのため、開発中は構造化処理プログラムをDockerイメージ外、つまりローカル環境側に置くなどの工夫が必要になります。開発したスクリプトは、Dockerイメージ外に置かれますので、Dockerコンテナの再起動の影響を受けずに保持されます。

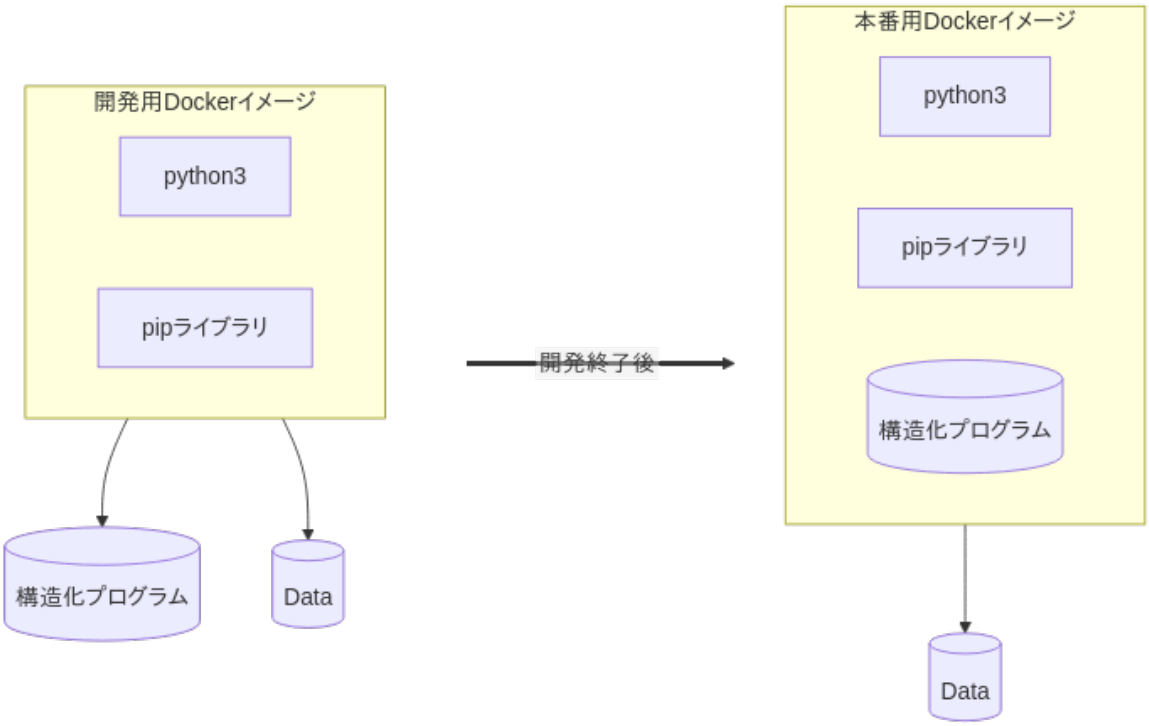
> DockerにはDockerイメージ外に開発したスクリプトを保存する方法がいくつか存在しますが、本書では バインドマウント と呼ばれる方法を用いて、ローカル環境側のファイルシステムに保存します。そのため、ファイルの所有者や書き込み権限など気をつけなければいけない点もありますが、Docker外からスクリプトを変更することも可能です。

開発が終了し、テスト実行で問題がないことを確認した後、開発した構造化処理プログラムを含む形でDockerイメージを再作成します。このDockerイメージを使ったDockerコンテナを起動してテスト実行を行い、問題が無ければ開発終了です。

本書は、主に後者、つまりDockerを用いた方法での構造化処理プログラム開発の手順について解説します。

> 実際にRDE環境で利用するデータセットを構築するには、コンテナレジストリと呼ばれるサーバ上にDockerイメージをアップロードしたり、ソースコード一式を提出していただいたりする必要がありますが、それらの作業はについては本書の範囲外とします。詳細については別途問合せください。

Dockerを利用した開発処理のイメージを以下に示します。



稼働しているDockerイメージを `コンテナ` と言います。このコンテナ内でファイル(main.pyなど)を修正して(保存して)も、コンテナを停止した時点で変更内容は廃棄されて、次にDockerコンテナを起動した際には、Dockerイメージ作成時の状態で起動します。

開発した構造化処理プログラムを保存(→永続化)できるように、スクリプトとデータをDockerイメージ外に置いて、コンテナを起動し、そのコンテナ上で開発(Pythonスクリプト作成)を行います。

> 以後、コンテナ内の環境と区別するため、dockerを実行しているLinux等の環境を、**ローカル環境** と呼ぶことにします。

起動したコンテナ上で `main.py` を実行し、想定されたファイルが出力されるかを確認します。

スクリプトが異常終了したり、想定した結果ではないファイルが生成された場合は、コンテナ上で再びPythonスクリプトを修正します。

以降は、コンテナ上で `開発(修正)` → `実行(確認)` → `開発(修正)` → `実行(確認)` → …… を繰り返します。

最終的には構造化処理プログラムのソースコードはDockerイメージに含まれている必要があります。開発が完了した(Pythonソースコードを修正する必要がなくなった)時点で、それらを組み込んだDockerイメージを実行用として再作成します。

このようにPythonスクリプトの開発と実行をコンテナ上で行いますので、ローカル環境上にはPython環境を用意する必要はありません。

1.2 対象読者

以下のような方々を対象とします。

- RDEToolKitを利用して構造化処理を開発する開発者
- 開発自体は他者(他社)に任せているが、最終的にRDE環境にDockerイメージとして登録する作業に携わる方

> 本書の利用にあたっては、ある程度の Dockerに関する基礎知識 を有していることが必要となります。

1.3 注意

- 本書ではRDEToolKit v1.2.0を使った場合の例を表示しています。バージョンによっては動作が異なる可能性があります。
- 本書では、"フォルダ"と"ディレクトリ"を同じ意味で使用しています。

1.4 既知の問題(制約)

ファイル名(スクリプト名や、データファイル名)に日本語を使うことは可能ですが、文字コードの違い(UTF-8かシフトJISか)、その他の原因で正しく処理されない場合が考えられます。

ファイル名に日本語を含む名前を付けることはできる限り避けてください。

また、スクリプト内に日本語を書く場合は、文字コード `UTF-8` で記述してください。

> 構造化処理プログラム内で読み込む データファイル の中に含まれる日本語の取扱については、構造化処理プログラムをどのように記述するか次第となります。こちらは特に制限はありませんので、文字コード判別してから処理を行うなど、構造化処理プログラムの中で適切な記述を行ってください。

2. 環境準備

2.1 Dockerのインストール

Windows、MacおよびLinuxの環境において、Docker Desktopのインストールが最も簡単ですが、組織によっては有償ライセンスを購入する必要があるため、本書ではDocker Desktopを使わない方法でインストールします。

> 企業でDocker Desktopを利用する条件
>
> 大企業（従業員が 251 人以上、または、年間収入が 1,000 万米ドル以上 ）における Docker Desktop の商用利用には、有料サブスクリプション契約が必要です。
>
> 参考: Docker is Updating and Extending Our Product Subscriptions - Docker

Docker Desktopを使わない場合は、Docker公式サイトが提供しているDocker Engineを利用します。Docker Engineは、"dockerd デーモンプロセス"とそれを利用する"API"、コマンドラインインターフェースとしての"dockerコマンド"からなっています。

Docker Engineを具体的に利用する方法は、OSによって異なります。また同じOSであってもいくつもの利用方法があります。

本書では、以下のような環境での利用を想定します。

- Windows環境においては、WSL2環境を利用してLinuxを用意し、その環境上でDocker Engineを利用します。
- Mac環境においては、"lima + Docker Engine"、"Rancher Desktop"を利用する方法などがありますが、"lima + Docker Engine"を利用します。
- Linux環境においては、Docker Engineがそのまま利用できます。

2.1.1 Windows 環境

Windows環境では、WSL2環境を使用したLinux環境を利用します。

WSL(Windows Subsystem for Linux、Linux用Windowsサブシステム）とは、LinuxのプログラムをWindows 10/11およびWindows Server上で実行するための仕組みです。最初のバージョンであるWSL1はLinuxシステムコールをWindowsシステムコールへ変換する方式でしたが、現在主流のWSL2はLinuxカーネルそのものをHyper-Vで実行する方式に変更されています。

WSL2環境が利用可能になっていない場合は、先にそちらの準備を行ってください。

> WSL2環境のインストールについては本書では触れません。マイクロソフトが提供しているドキュメント(https://learn.microsoft.com/ja-jp/windows/wsl/install)等を参考にインストールしてください。

以下では、WSL2環境が利用でき、その上で `Ubuntu` が利用可能になっていることを前提に進めます。

> 以下に示すやり方は、本稿作成時点のものになります。WindowsのバージョンやDocker Engineのバージョンにより変更される可能性があります。Docker Engineのインストール前に、公式ドキュメントを確認してください。
>
> 参考: Install Docker Engine on Ubuntu - docker docs

PowerShellを利用して、WSL2上のLinuxを起動します。

以下は、Ubuntu-24.04を起動する場合の例です。

```
PS C:\Windows\System32> wsl -d Ubuntu-24.04
$
```

> プロンプトが"$"に変わりましたのでLinux環境に移行したことが分かります。なお、プロンプトは設定により変更できますので、上記とは異なるプロンプトが表示される可能性があります。

2つのコマンドを使ってDocker Engineをインストールします。

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

プロキシ環境下でインストールする場合は、先に環境変数を指定して実行してください。

> プロキシサーバを"proxy.example.com:8080"として記述します。環境に合わせて変更してください。

```
export http_proxy=http://proxy.example.com:8080
export https_proxy=http://proxy.example.com:8080

curl -fsSL https://get.docker.com -o get-docker.sh
sudo -E sh get-docker.sh
```

> 環境によっては、さらに事前設定を行う必要が有る場合があります。それぞれの環境で必要な処理を実施してください。
>
> デフォルトでは `sudo` コマンドは、環境変数を引き継ぎません。プロキシの設定は引き継ぐ必要がありますので、`-E` オプションを使う必要があります。

さらに、Dockerをrootユーザー以外で実行できるように設定します。

```
sudo usermod -aG docker $USER
```

また、WSL2でdockerdをデーモンとして管理するためにsystemdのサポートを有効化します。`/etc/wsl.conf` にsystemdを有効化するオプションを以下の通り記述します。

> systemd サポート - Microsoft Support

```
sudo vi /etc/wsl.conf
```

以下の内容を書き込んで保存します。

```
# /etc/wsl.conf
[boot]
systemd=true
```

> 新しいディストリビューション(Ubuntu 24.04LTSなど)では、上記設定が最初から入っている場合があります。

最後に、WSLを再起動するとDockerが使用できるようになります。

```
$ exit

wsl --shutdown

wsl -d Ubuntu-24.04
```

> "-d"オプションの後に指定する部分は、利用するディストリビューションによって変わります。上記は"Ubuntu 24.04LTS"を使う場合の例です。また、上の例では、`exit` はWSL2上のLinuxに対するコマンド、wslコマンドは、Windows上で実行するコマンドです。

2.1.2 Mac 環境

limaは、Mac環境上で Linux 仮想マシンを動かす、つまり、Windows環境における `WSL2` 的な機能を提供してくれるMac版のツールです。この仮想環境の上でDockerを動作させます。

limaとdocker、docker-composeをbrewでインストールします。

```
brew install lima docker docker-compose
```

次にlima上にdocker vmを作るための設定を入手し、一部設定を書き換え保存します。

> 設定ファイルには使うCPU core数、メモリなどが設定されており、自分の環境に合わせて設定することが可能ですが、ここではまず動作させるため標準設定を使用します。

```
curl https://raw.githubusercontent.com/lima-vm/lima/master/examples/docker.yaml | sed -e 's%- location: "~"%- location: "~"\n
writable: true%g' > ./tempconf.yaml
```

docker vmを作成します。

```
limactl start --name=docker ./tempconf.yaml
```

設定ファイルは一度作成したら不要であるため削除可能です。

```
rm ./tempconf.yaml
```

作成が完了すると、以下のようなサンプルコマンドが3行表示されるので実行します。

```
# [user]のところは自分のhomeのユーザー名を指定する。
docker context create lima-docker --docker "host=unix://Users/[user]/.lima/docker/sock/docker.sock"
docker context use lima-docker
```

環境変数を設定後、dockerコマンドがdocker vmに接続します。この設定をしないとdocker runができないためご注意ください。

```
export DOCKER_HOST="unix:///Users/[user]/.lima/docker/sock/docker.sock"
docker run hello-world
```

> exportコマンドによる環境変数設定は `.zshrc` または `.zprofile` (bashを利用している場合は `.bashrc` や `.bash_profile`)に記述することを推奨します。

VMを起動/停止し、また起動状態の確認するには、`limactl` コマンドを利用します。

```
limactl list
# 起動
limactl start doker
# 停止
limactl stop doker
```

> proxy環境下での注意
>
> vm作成時、LocalPCの環境変数 `http_proxy` などを引き継ぎます。その結果、lima上のdocker vmに書き込まれてしまうので、設定を削除したい場合は以下の手順で実行します。
>
> `ssh -p 49476 -i ~/.lima/_config/user -o NoHostAuthenticationForLocalhost=yes 127.0.0.1`
>
> ログイン後、rootで作業したいときは、`sudo su` とする。
>
> limaのdocker vmの上で/etc/enviromentでproxyの設定が引き継がれていたのが原因であるため、これを無効にします。

また、limaのVMは自動では起動してくれないので自動起動させます。やり方は複数ありますが、ここではショートカットアプリを使用する方法を示します。

- ショートカットアプリを起動し新規作成します。
- Appタブのターミナルから シェルスクリプトを実行 をダブルクリックし、スクリプト入力欄に以下の処理を追加します。

```
eval "$(/opt/homebrew/bin/brew shellenv)"
limactl start docker
```

- ショートカット一覧画面に戻ると作ったショートカットが出来ているため、右クリックから[Dockに追加]を選択します。
- Dockに出たアイコンを右クリックし[オプション]->[ログイン時に開く]

2.1.3 Linux 環境

Linuxでは、Docker Engineが利用できます。以下は一般的な手順ですが、Linuxディストリビューションによって異なる場合がありますので、公式のドキュメントを参照してください。

> 参考: Install Docker Engine - docker docs

1. OS標準提供のDocker Engineのアンインストール (インストール済みの場合)
2. apt/dnf等インストールツールに、Dockerリポジトリの追加
3. apt/dnf等を使って、Docker Engineのインストール
4. dockerdデーモンの起動(および自動起動設定)

> 一般的にOS標準提供のDocker Engineは最新版より古いものであることが多いため、Docker公式から提供されている最新版をインストールして利用するようにします。ここでは、インストールの手順詳細については記述しません。上で示した公式ドキュメントを参照して作業を実施してください。

2.2 プロキシ設定

実行する環境によっては、Dockerに対してプロキシの設定を追加する必要があります。

ここでは、WSL2のLinux環境の場合の設定例を示しますが、他の環境のLinux上でもほぼ同じ設定が可能です。

> プロキシサーバを"proxy.example.com:8080"として記述します。環境に合わせて変更してください。

2.2.1 dockerデーモン向けの設定

```
$ sudo systemctl edit docker

:
[Service]
Environment = 'http_proxy=http://proxy.example.com:8080' 'https_proxy=http://proxy.example.com:8080'
```

> "### Edits below this comment will be discarded"という行の、上に追加する必要があります。この行の下に追加した場合は追加されません。

編集後、上記設定内容を有効にし、dockerデーモンを再起動します。

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

`docker info` コマンドにより、有効になったことを確認します。

```
$ docker info | grep -i proxy
 HTTP Proxy: http://proxy.example.com:8080
 HTTPS Proxy: http://proxy.example.com:8080
```

2.2.2 dockerコマンド向けの設定

ホームディレクトリに以下のフォルダを作成します。

```
$ mkdir .docker
$ vi .docker/config.json
```

設定する内容は以下の様にします。

```
{
  "proxies": {
    "default": {
      "httpProxy": "http://proxy.example.com:8080",
      "httpsProxy": "http://proxy.example.com:8080"
    }
  }
}
```

> 本書では、これ以降"WSL2 / Ubuntu 24.04LTS + Docker Engine"を使った環境を想定して記述することにします。
> その他の環境で利用される場合は、適宜読み替えてください。

2.3 dockerコマンド確認

構造化処理の開発を進める前に、`docker` コマンドが利用できることを確認します。

Linux上で以下のコマンドを実行します。

```
$ docker --version
Docker version 27.5.1, build 9f9e405
```

> バージョンは随時更新されますので、上記と同じバージョンとなるとは限りません。
>
> "Command 'docker' not found"のようなメッセージが出力される場合は、dockerのインストールを見直し、必要に応じて再度インストールしてください。

2.4 dockerd稼働確認

`hello-world` コンテナを使って、`docker` コマンドおよび `dockerデーモン` の稼働確認を行います。

> コンテナ とはアプリやインフラなどを入れた箱であり、イメージ とはコンテナを実行するために必要なファイルシステムのことです。
>
> `hello-world`コンテナ は、dockerhub公式に提供されているコンテナの1つで、実行すると"Hello from Docker!"と表示するだけのコンテナです。

docker runコマンドでイメージを取得、コンテナの実行をします。

```
docker run hello-world
```

結果は以下の様になります。

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:d1b0b5888fbb59111dbf2b3ed698489c41046cb9d6d61743e37ef8d9f3dda06f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

> 中程に"Hello from Docker!"と表示されているのが分かります。Digest: など一部の項目は実行の都度変わる可能性があります。

上の実行は、初回実行なのでDockerイメージがダウンロードされていませんでした。そのため最初にダウンロード処理が実行されました。2回目以降の実行では、すでにイメージをダウンロードしているので、表示が短くなります。

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

（以下省略）
```

> "Hello from Docker!"からの表示となり、それ以降は1回目と同じ表示となります。

この表示により、Docker環境が準備完了であると判断できます。

失敗例をいくつかあげます。

2.4.1 失敗例1：dockerデーモンが起動していない

```
$ docker run hello-world
docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?.
See 'docker run --help'.
```

この場合は、dockerデーモンが起動してません。dockerデーモンを起動します。

```
sudo systemctl start docker
```

2.4.2 失敗例2：プロキシ設定不良

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
docker: Error response from daemon: Get "https://registry-1.docker.io/v2/": read tcp MMM.MMM.MMM.MMM:47868->NNN.NNN.NNN.NNN:443:
read: connection reset by peer.
See 'docker run --help'.
```

> "MMM.MMM.MMM.MMM"や"NNN.NNN.NNN.NNN"は、環境に応じたIPアドレスが入ります。

この場合は、dockerコマンドやdockerデーモンに対するプロキシサーバの設定に不備があることが考えられます。前述のプロキシ設定の項を参考に設定を見直してください。

dockerデーモン設定を変更した場合は、dockerデーモンの再起動が必要です。

```
sudo systemctl restart docker
```

2.4.3 失敗例3：ソケットファイル(docker.sock)アクセスの権限不足

```
(venv) $ docker run hello-world
permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://
%2Fvar%2Frun%2Fdocker.sock/v1.47/containers/json": dial unix /var/run/docker.sock: connect: permission denied
```

この場合は、idコマンドで `docker` グループに所属しているかを確認して、所属していない場合は `docker` グループへの追加(追加方法は前述)を実施してください。

```
$ id
uid=1003(devel) gid=1003(devel) groups=1003(devel),27(sudo),100(users),989(docker)
```

> ユーザ名 `devel` での実行例です。他に所属しているグループやGIDなどは、実行する環境により変わります。上記のように `docker` グループに所属していれば問題ありません。

上記に問題がない場合でもエラーが消えない場合は、Linuxシステムを再起動してください。

```
sudo reboot
```

WSL2の場合は、一度抜けてwslのshutdownを実施後、再度起動します。

```
$ exit
logout

PS C:\Windows\System32> wsl --shutdown
PS C:\Windows\System32> wsl -d Ubuntu-24.04
```

> `$` で始まる行は、WSL上で稼働しているUbuntuで実行するコマンド、`PS` で始まる行は、Windows上のPowerShellのコマンドです。

3. 開発用Dockerイメージの作成

前述の様に、Dockerを利用した開発では、最初に開発用Dockerイメージを作成し、構造化処理プログラムの開発を行った後、それを含む形で本番実行用のDockerイメージを作成し直します。

本章では、まず最初に `開発用のDockerイメージ` を作成します。

3.1 作業用フォルダの作成

ホームディレクトリに、作業用のディレクトリを作成し、そこに移動します。

以下では、`rde-docker/` フォルダを作成し利用する場合の例を示します。

```
mkdir $HOME/rde-docker
cd $HOME/rde-docker
```

Dockerイメージを作成するための設定ファイルやPythonスクリプトを格納するフォルダとして、`dev_image/` フォルダを使うことにします。

フォルダを作成し、移動します。

```
mkdir dev_image
cd dev_image
```

3.2 必要な設定ファイルの用意

3.2.1 構造化処理プログラムの用意

これから開発していくので無くてもいいのですが、必要最低限のファイルとして"main.py"のみ用意することにします。

以下の内容で、`$HOME/rde-docker/dev_image/main.py` を作成します。

```
import rdetoolkit


def main():
    rdetoolkit.workflows.run()

if __name__ == '__main__':
    main()
```

> Pythonスクリプトなので、インデントに注意してください。

3.2.2 vimrcファイルの用意

後述するように、ファイルの編集にvimを使うことを想定しています。

今回は以下の内容で `$HOME/rde-docker/dev_image/vimrc` を用意し、後ほどDockerイメージに組み込みます。

```
syntax on
set clipboard=unnamed,autoselect
```

> 上の内容は、"構文ハイライトを有効"にし、"右クリックによるペーストを有効"にするものです。この設定は任意設定項目なので、必要が無ければ設定不要ですし、逆に他に必要な設定が有る場合は、ここに追加してください。

3.2.3 requirements.txtの用意

同じフォルダに、以下の内容で `requirements.txt` を用意します。

```
rdetoolkit==1.2.0
```

> 上の例ではバージョン番号を指定してrdetoolkitをインストールしています。"=="の前後に空白を挟むとエラーになりますので注意してください。

3.2.4 Dockerfileの用意

同じフォルダに、以下の内容で `Dockerfile` というファイルを用意します。

```
FROM python:3.12-bookworm

# treeコマンドインストール
RUN apt-get update && apt-get install -y \
    tree \
    vim \
&& rm -rf /var/lib/apt/lists/*

# vim設定ファイルをコピー
COPY vimrc /root/.vimrc

# appディレクトリを作成
WORKDIR /app

# requirements.txt設置
COPY requirements.txt /app

# 必要なPythonライブラリのインストール
RUN pip install --upgrade pip \
 && pip install -r requirements.txt

# プログラムや設定ファイルなどをコピーする
COPY main.py /app
```

> 構造化処理プログラムの実行には `vim` や `tree` コマンドは不要ですが、本マニュアル作成の都合上インストールしています。また構造化処理プログラムを、コンテナ内で編集するため `vim` をインストールしています。他のエディタで編集する場合は、適宜置き換えてください。
>
> vim設定ファイルは、コピー時に名称が変える必要があります(先頭にドット(.)を付与する)。

この時点では、`` `$HOME/rde-docker/dev_image/ `` には、4つのファイルだけが存在している状態です。

```
$ ls $HOME/rde-docker/dev_image/
Dockerfile  main.py  requirements.txt  vimrc
```

3.3 Dockerイメージの名前とバージョンの決定

作成するDockerイメージの名前とバージョンを決めます。ここでは以下を使うことにします。

項目	値	備考
名前	rde-sample	
バージョン	v0.1	開発版はv0.x(xは1からはじめ、必要に応じ加算)、本番版でv1.0に変更

3.4 Dockerイメージの作成

`Dockerfile` があるフォルダ上にいることを確認します。

```
$ pwd
/home/[ユーザ名]/rde-docker/dev_image
```

上記情報を使って、イメージを作成します。Dockerイメージを作成するには `docker build` コマンドを使います。

> そのため、Dockerイメージを作成することを、"ビルドする"という場合があります。

```
$ docker build -t rde-sample:v0.1  ./
 :
 :
 => exporting to image                                          0.9s
 => => exporting layers                                         0.9s
 => => writing image sha256:bcc3b7c02b33792f1b26 ..... 98cc68    0.0s
 => => naming to docker.io/library/rde-sample:v1.0              0.0s
```

> 実行の都度出力は変わります。
>
> `-t` オプションで、上で決めたイメージの名前とバージョンを指定して作成します。名前とバージョンはコロン(:)を挟んで連結する必要があります。
>
> プロキシ環境下で `docker build` する場合は、$http_proxyや$https_proxy環境変数を設定する必要があります。これは「`docker image pull` 実行時は、systemctlで追加した設定を使う」が「`docker image build` コマンドがベースイメージを pull するときは、シェルの環境変数を参照する」という仕様のためです。

環境によっては、`docker build` コマンドが `SSLCertVerificationError` で正常に終了しない場合があります。その場合はDockerfileの `RUN pip install ～` の前に以下の行を追加してください。

```
ARG PIP_TRUSTED_HOST="pypi.python.org files.pythonhosted.org pypi.org pypi.io"
```

`docker image ls` コマンドで、確認します。

```
$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
```

```
rde-sample    v0.1      b61962ef8995   7 minutes ago   1.64GB
hello-world   latest    74cc54e27dc4   5 weeks ago     10.1kB
```

> IMAGE ID や CREATED 欄に表示される値は、実行される環境に合わせて表示されるので、実際の表記は上記とは異なります。

3.5 コンテナ実行

構造化処理プログラムの開発に入る前に、`docker run` コマンドで、dockerイメージを実行し、問題無く稼働することを確認します。`-it` オプションを付けることで、イメージの中に移動します。

```
docker run -it <Dockerイメージ名>:<バージョン> <起動コマンド>
```

`<Dockerイメージ名>:<バージョン>` には、上記で指定したものを、`<起動コマンド>` には、"/bin/bash"を指定します。

```
$ docker run -it --rm rde-sample:v0.1 /bin/bash
root@50e097b072ce:/app#
```

> プロンプトが変わったことで、Dockerイメージ内にいることが分かります。"@"以降の文字列(上記例だと"50e097b072ce"の部分)は実行の都度変わります。
>
> `--rm` は、`exit` コマンドによりコンテナから抜けた際にコンテナを削除することを指定します。必須ではありません。
>
> Dockerfile内で `WORKDIR /app` と指定しているので、`/app` に移動した状態から始まります。

```
root@50e097b072ce:/app# ls
main.py  requirements.txt
root@50e097b072ce:/app# python --version
Python 3.12.10
```

> イメージをビルドした時期により、pythonのマイナーバージョンが異なる場合があります。

RDEToolKitがインストールされていることを確認します。

```
root@50e097b072ce:/app# python -m rdetoolkit version
1.2.0
```

> pypi上のRDEToolKitは随時バージョンアップが行われます。異なるバージョンが表示されることがあることに注意してください。
>
> pythonコマンドが利用出来ない場合は `python3` コマンドを使ってください。

このバージョンでは、この時点で `main.py` を実行すると以下のようなエラーが表示されます。"invoice.schema.json"は必須ファイルですが、当該ファイルを用意していませんので、このエラーが出るのは想定通りです。次章以降で開発作業を実施していきますので、このエラーは無視してください。

```
root@50e097b072ce:/app# python main.py
  0%|                                                                                                    | 0/1 [00:00<?, ?it/s]
Traceback (most recent call last):
  File "/usr/local/lib/python3.12/site-packages/rdetoolkit/workflows.py", line 229, in run
    status = invoice_mode_process(str(idx), srcpaths, rdeoutput_resource, custom_dataset_function)
```

```
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/site-packages/rdetoolkit/modeproc.py", line 369, in invoice_mode_process
    invoice_validate(resource_paths.invoice.joinpath("invoice.json"), schema_path)
  File "/usr/local/lib/python3.12/site-packages/rdetoolkit/validation.py", line 267, in invoice_validate
    raise FileNotFoundError(emsg)
FileNotFoundError: The schema and path do not exist: invoice.schema.json


============================================================
Custom Traceback (simplified and more readable):
============================================================

Traceback (simplified message):
Call Path:
   File: /usr/local/lib/python3.12/site-packages/rdetoolkit/workflows.py, Line: 229 in run()
    └─ File: /usr/local/lib/python3.12/site-packages/rdetoolkit/modeproc.py, Line: 369 in invoice_mode_process()
        └─ File: /usr/local/lib/python3.12/site-packages/rdetoolkit/validation.py, Line: 267 in invoice_validate()
            └─> L267: raise FileNotFoundError(emsg)

Exception Type: FileNotFoundError
Error: The schema and path do not exist: invoice.schema.json
```

> 実行するRDEToolKitのバージョンにより、表示される行番号は変更される可能性があります。

コンテナから抜けます。

```
root@50e097b072ce:/app# exit
exit
```

3.5.1 コンテナ内でのプロンプト表記について

上で示したように、コンテナ内に入っている場合はプロンプトが `root@(コンテナID):(カレントディレクトリ)#` のように変化します。コンテナIDは、実行する度に変更されますので都度異なるプロンプトが表示されます。

そのため、本書で示す手順通りにやっても、異なるプロンプトが表示されます。

本書において、一連の流れであってもプロンプトが変わっていることがありますが、転記のタイミングによるものですので無視してください。

> 以降の章でも同様となります。

4. 開発

前章で作成したDockerイメージを使って構造化処理プログラムの開発していきます。

4.1 ソースフォルダ作成

ソースコードを保存するフォルダを、ローカル環境に用意します。

`app/` フォルダを作成します。

```
cd $HOME/rde-docker
mkdir app
```

> プロンプトを確認し、コンテナ内から抜けていることを確認してから上記を実行してください。コンテナ内にいる場合は `exit` コマンドでコンテナから抜けます。また、カレントディレクトリが `dev_image/` でないことを確認してから `app/` フォルダを作成してください。

4.2 データフォルダ作成

実際の運用環境では `data/` フォルダは、`app/container/` フォルダの下に用意され、画面で入力された内容や入力ファイルなどが自動的に配置されます。

```
app
├── container
|    ├── data（本来の配置位置）
:
```

ここでは開発の都合上 `app/` と同じ階層に作ることにします。

```
cd $HOME/rde-docker
mkdir data
```

`data/` フォルダ下は以下の様になります。

```
app
├── container（この時点では未作成）
|    ├── data（本来の配置位置）（この時点では未作成）
:
data（今回の設置位置）
:
```

4.3 コンテナ起動

最終的には、コンテナ内の"/app"フォルダにスクリプトを配置したDockerイメージを作成しますが、上記で作成したダミーのスクリプトファイル(main.py)が配置されています。

それらダミーとして作成したものと、これからコンテナ上で新規に作成するものを区別するために、ここでは `/app2` にプログラムを、`/app2/data` の下にデータを配置することにします。

> Dockerイメージを再作成する際に考慮すればいいので、そのまま"/app"を使ってもいいですし、別の名前のディレクトリを使用しても構いません。

上記ディレクトリを使用する形でコンテナを起動します。

```
$ docker run -it --rm -v ${PWD}/app:/app2 -v ${PWD}/data:/app2/data rde-sample:v0.1 /bin/bash
root@cfafcd0f658a:/app#
```

上記例では、`-v` フラグを使って、2つのフォルダ(`app/` と `data/`)を、コンテナ内の別のフォルダに配置しています。

> 前述の様に、`-v ${PWD}/app:/app2` は、ホスト上(→Dockerを実行しているサーバ)の `${PWD}/app` (→上で用意したソースフォルダ)を、コンテナ内の `/app2` フォルダにマウントし、もう1つの `-v ${PWD}/data:/app2/data` は、`${PWD}/app` (→上で用意したデータフォルダ)を、コンテナ内の `/app2/data` フォルダにマウントします。

4.4 フォルダ構成初期化

`app2/` の下に、RDEToolKitが想定するフォルダ構成を、コンテナ内で"初期化"します。

最初に、`app2/フォルダに移動します。

> 別のフォルダを利用する場合は、そのフォルダに移動してください。

```
root@cfafcd0f658a:/app# cd /app2
```

続いて、RDEToolKitの `init` サブコマンドを実行し、フォルダ構成を初期化し、必要最低限のファイルを生成します。

```
root@cfafcd0f658a:/app2# python -m rdetoolkit init
Ready to develop a structured program for RDE.
Created: /app2/container/requirements.txt
Created: /app2/container/Dockerfile
Created: /app2/container/data/invoice/invoice.json
Created: /app2/container/data/tasksupport/invoice.schema.json
Created: /app2/container/data/tasksupport/metadata-def.json
Created: /app2/templates/tasksupport/invoice.schema.json
Created: /app2/templates/tasksupport/metadata-def.json
Created: /app2/input/invoice/invoice.json

Check the folder: /app2
Done
```

> 上述の様に、ローカル環境の `${PWD}/data` フォルダは、コンテ内の `/app2/data` にマウントされていますが、上で作成した `data/` フォルダは `/app2/container` の下にあるものです。そのため、`/app2/container/data/` 下の内容を、ローカル環境の `${PWD}/data` フォルダに移す必要があります(後述)。

出来たばかりの構造化処理プログラムを実行してみます。

```
root@cfafcd0f658a:/app2# cd container/
root@cfafcd0f658a:/app2/container# ls -F
Dockerfile  data/  main.py  modules/  requirements.txt
```

```
root@cfafcd0f658a:/app2/container# python main.py
```

> 実際にはプログレスバーが表示されます。本書では、これ以降も、プログレスバー表示は重要ではないため表示省略します。

ここでは、エラー表示なく終了したことを確認します。

フォルダ構造の初期化が完了しましたので、いったんコンテナから抜けます。

```
root@cfafcd0f658a:/app2/container# exit
exit
$
```

ローカル環境のフォルダを確認確認します。

```
$ tree app
app
├── container
│   ├── Dockerfile
│   ├── data
│   │   ├── attachment
│   │   ├── inputdata
│   │   ├── invoice
│   │   │   └── invoice.json
│   │   ├── invoice_patch
│   │   ├── logs
│   │   ├── main_image
│   │   ├── meta
│   │   ├── nonshared_raw
│   │   ├── other_image
│   │   ├── raw
│   │   ├── structured
│   │   ├── tasksupport
│   │   │   ├── invoice.schema.json
│   │   │   └── metadata-def.json
│   │   ├── temp
│   │   └── thumbnail
│   ├── main.py
│   ├── modules
│   └── requirements.txt
├── data
├── input
│   ├── inputdata
│   └── invoice
│       └── invoice.json
└── templates
    └── tasksupport
        ├── invoice.schema.json
        └── metadata-def.json

24 directories, 9 files
```

> `python main.py` を実行しないで確認すると、initサブコマンド実行のメッセージに示されたとおり、`data/invoice/invoice.json` フォルダと `data/tasksupport` フォルダの2つだけ存在することになります。他のフォルダが存在しない場合は、上記で示している手順で `python main.py` を実行してください。

4.5 データフォルダの初期化

この時点で、ローカル環境上に用意した `data/` フォルダには何も入っていません。

```
$ tree data
data

0 directories, 0 files
```

上の `init` サブコマンドで初期化された `data/` フォルダの内容を、カレントディレクトリにある `data/` フォルダに移動します。

```
$ cd ${HOME}/rde-docker
$ sudo mv app/container/data/* ./data

$ tree data
data
├── attachment
├── inputdata
├── invoice
│   └── invoice.json
├── invoice_patch
├── logs
├── main_image
├── meta
├── nonshared_raw
├── other_image
├── raw
├── structured
├── tasksupport
│   ├── invoice.schema.json
│   └── metadata-def.json
├── temp
└── thumbnail

15 directories, 3 files
```

> `data/` フォルダの下にフォルダが3つしかない場合は、コンテナ内で `python main.py` が未実施な場合が考えられます。再度コンテナを起動し、その中で(`init` サブコマンドで作成された) `main.py` を実行してください。ただし、当該3フォルダ以外は、後続の `python main.py` 処理実行時に作成されますので、必ずしも `python main.py` を実行する必要はありません。

次に、ローカル環境での書き換え時に"sudo"コマンドを使わなくてもよいように、`app/` と `data/` のオーナーを"root"から、開発で使用しているユーザ(→本マニュアル上では"devel"ユーザ)に変更しておきます。

```
sudo chown -R ${USER} app/
sudo chown -R ${USER} data/
```

カレントディレクトリを確認します。

```
$ pwd
/home/devel/rde-docker
```

現在のmain.pyの内容を確認します。

```
$ cat app/container/main.py
# The following script is a template for the source code.
```

```
import rdetoolkit

rdetoolkit.workflows.run()
```

> RDEToolKitのinit処理にて生成される内容のままです。

この状態で、`-v`オプションの設定を本番実行用のDockerイメージを想定したものに変更して、再度コンテナを実行します。

```
docker run -it --rm -v ${PWD}/app/container:/app2 -v ${PWD}/data:/app2/data rde-sample:v0.1 /bin/bash
```

> 実際の本番環境では app2/ フォルダではなく、app/ フォルダを利用するため、本番実行環境用のDockerイメージと完全に同じというわけではありません。
>
> 1個目のマウント元が ${PWD}/app から ${PWD}/app/container に変更になったことに注意してください。
>
> 2個目のマウント設定により、コンテナ内での main.py 実行で作成されたファイルは、マウント元の${HOME}/rde-docker/dataで確認することができるようになります。

フォルダの構成状況を確認すると以下の様になります。

```
root@7d92e5033f5e:/app# cd /app2
root@7d92e5033f5e:/app2# tree
.
├── Dockerfile
├── data
│   ├── attachment
│   ├── inputdata
│   ├── invoice
│   │   └── invoice.json
│   ├── invoice_patch
│   ├── logs
│   ├── main_image
│   ├── meta
│   ├── nonshared_raw
│   ├── other_image
│   ├── raw
│   ├── structured
│   ├── tasksupport
│   │   ├── invoice.schema.json
│   │   └── metadata-def.json
│   ├── temp
│   └── thumbnail
├── main.py
├── modules
└── requirements.txt

17 directories, 6 files
```

念のため、エラーなく実行できることを確認します。

```
root@7d92e5033f5e:/app2# python main.py
root@7d92e5033f5e:/app2#
```

コンテナ内から抜けます。

```
root@7d92e5033f5e:/app2# exit
exit
```

4.6 データ配置

開発で実際に使うデータを、用意します。

本来の開発であれば、ここで実際に利用されるデータなどを配置しますが、今回はダミーデータを用意することにします。

> ハンズオンチュートリアルで使用したものを使います。すでに同名のファイルがある場合は上書きしてください。
>
> コンテナ内で作成しても、コンテナ外で作成しても構いません。コンテナ内で作成する場合は、`data/inputdata/` 部分を `/app2/data/inputdata/` と読み替えて作成してください。

4.6.1 入力データファイル

`data/inputdata/sample.data` として、以下内容で作成します。

```
[METADATA]
data_title=data_title 2
measurement_date=2023/1/25 23:08
x_label=x(unit)
y_label=y(unit)
series_number=3
[DATA]
series1
16
1,101
2,102
3,103
4,104
5,105
6.2,106
7
8,108
9.5,109
10,101
11,103
12,105
13,104
14,100
15,98
16,82
[DATA]
series2
11
1,101
2,102
3,103
4,104
5,105
6,106
9,109
10,90
12,60
13,52
14,41
[DATA]
series3
12
1,101
1.5,101.2
2,102
3,103
4,104
```

```
5,105
6,106
9,109
10,90
12,60
13,52
14,41
```

4.6.2 送り状ファイル

`data/invoice/invoice.json` として、以下の内容で作成します。

> 既にあるファイルは不要ですので、一旦すべて削除してから以下の内容に置き換えてください。


```
{
    "datasetId": "8fdec13d-bca2-4808-8753-2bb7e4e9c927",
    "basic": {
        "dateSubmitted": "2023-01-26",
        "dataOwnerId": "119cae3d3612df5f6cf7085fb8deaa2d1b85ce963536323462353734",
        "dataName": "NIMS_TRIAL_20230126b",
        "instrumentId": "413e53fb-aec9-41f8-ae55-3f88f6cd8d41",
        "experimentId": null,
        "description": ""
    },
    "custom": {
        "measurement_date": "2023-01-25",
        "invoice_string1": "送状文字入力値1",
        "invoice_string2": null,
        "is_devided": "devided",
        "is_private_raw": "private"
    },
    "sample": {
        "sampleId": "4d6de819-4b42-4dfe-9619-a0a0588653bc",
        "names": [
            "試料"
        ],
        "composition": null,
        "referenceUrl": null,
        "description": null,
        "generalAttributes": [
            {
                "termId": "3adf9874-7bcb-e5f8-99cb-3d6fd9d7b55e",
                "value": null
            },
            {
                "termId": "0aadfff2-37de-411f-883a-38b62b2abbce",
                "value": null
            },
            {
                "termId": "0444cf53-db47-b208-7b5f-54429291a140",
                "value": null
            },
            {
                "termId": "e2d20d02-2e38-2cd3-b1b3-66fdb8a11057",
                "value": null
            }
        ],
        "specificAttributes": [
            {
                "classId": "52148afb-6759-23e8-c8b8-33912ec5bfcf",
                "termId": "70c2c751-5404-19b7-4a5e-981e6cebbb15",
                "value": null
            },
            {
                "classId": "961c9637-9b83-0e9d-e60e-ffc1e2517afd",
```

```
                "termId": "70c2c751-5404-19b7-4a5e-981e6cebbb15",
                "value": null
            },
            {
                "classId": "01cb3c01-37a4-5a43-d8ca-f523ca99a75b",
                "termId": "dc27a956-263e-f920-e574-5beec912a247",
                "value": null
            }
        ],
        "ownerId": "119cae3d3612df5f6cf7085fb8deaa2d1b85ce963536323462353734"
    }
}
```

4.6.3 タスクサポートフォルダに置くファイル

ここでは、`data/tasksupport/invoice.schema.json` と `data/tasksupport/metadata-def.json` を用意します。

> 既にあるファイルはダミーですので、一旦すべて削除してから以下の内容に置き換えてください。

`data/tasksupport/invoice.schema.json` として、以下の内容で作成します。

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "TEST_NIMS_Sample_Template_v0.0",
    "description": "None",
    "type": "object",
    "required": [
        "custom"
    ],
    "properties": {
        "custom": {
            "type": "object",
            "label": {
                "ja": "固有情報",
                "en": "Custom Information"
            },
            "required": [],
            "properties": {
                "measurement_date": {
                    "label": {
                        "ja": "計測年月日",
                        "en": "measurement_date"
                    },
                    "type": "string",
                    "format": "date"
                },
                "invoice_string1": {
                    "label": {
                        "ja": "文字列1",
                        "en": "invoice_string1"
                    },
                    "type": "string"
                },
                "invoice_string2": {
                    "label": {
                        "ja": "文字列2",
                        "en": "invoice_string2"
                    },
                    "type": "string"
                },
                "is_devided": {
                    "label": {
                        "ja": "複数ファイル区分",
                        "en": "is_devided"
                    },
```

```
                    "type": "string"
                },
                "is_private_raw": {
                    "label": {
                        "ja": "公開区分",
                        "en": "is_private_raw"
                    },
                    "type": "string"
                }
            }
        }
    }
}
```

`data/tasksupport/metadata-def.json` として、以下の内容で作成します。

```
{
    "data_title": {
        "name": {
            "ja": "データ名",
            "en": "data title"
        },
        "schema": {
            "type": "string"
        },
        "order": 1,
        "mode": "Derived from data",
        "original_name": "data_title"
    },
    "measurement_date": {
        "name": {
            "ja": "測定日時",
            "en": "measurement date"
        },
        "schema": {
            "type": "string",
            "format": "date-time"
        },
        "order": 2,
        "mode": "Derived from data",
        "original_name": "measurement_date"
    },
    "x_label": {
        "name": {
            "ja": "独立変数(X軸ラベル)",
            "en": "x-label"
        },
        "schema": {
            "type": "string"
        },
        "order": 3,
        "mode": "Derived from data",
        "original_name": "x_label"
    },
    "y_label": {
        "name": {
            "ja": "従属変数(Y軸ラベル)",
            "en": "y-label"
        },
        "schema": {
            "type": "string"
        },
        "order": 4,
        "mode": "Derived from data",
        "original_name": "y_label"
    },
    "series_number": {
        "name": {
```

```
            "ja": "系列数",
            "en": "series number"
        },
        "schema": {
            "type": "integer"
        },
        "order": 5,
        "unit": "PCS",
        "mode": "Derived from data",
        "original_name": "series_number"
    },
    "series_name": {
        "name": {
            "ja": "系列名",
            "en": "series name"
        },
        "schema": {
            "type": "string"
        },
        "order": 6,
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_name"
    },
    "series_data_count": {
        "name": {
            "ja": "系列ごとデータ数",
            "en": "data count by series"
        },
        "schema": {
            "type": "integer"
        },
        "order": 7,
        "unit": "PCS",
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_data_count"
    },
    "series_data_mean": {
        "name": {
            "ja": "系列ごと平均値",
            "en": "data mean by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 8,
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_data_mean"
    },
    "series_data_median": {
        "name": {
            "ja": "系列ごと中央値",
            "en": "data median by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 9,
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_data_median"
    },
    "series_data_max": {
        "name": {
            "ja": "系列ごと最大値",
            "en": "data max by series"
        },
        "schema": {
```

```
            "type": "number"
        },
        "order": 10,
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_data_max"
    },
    "series_data_min": {
        "name": {
            "ja": "系列ごと最小値",
            "en": "data min by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 11,
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_data_min"
    },
    "series_data_stdev": {
        "name": {
            "ja": "系列ごと標準偏差",
            "en": "data stdev by series"
        },
        "schema": {
            "type": "number"
        },
        "order": 12,
        "mode": "Analysis value",
        "variable": 1,
        "original_name": "series_data_stdev"
    },
    "measurement date": {
        "name": {
            "ja": "送状状測定日時",
            "en": "measurement date from invoice"
        },
        "schema": {
            "type": "string",
            "format": "date-time"
        },
        "order": 13,
        "mode": "Invoice",
        "original_name": "measurement date"
    },
    "invoice_number1": {
        "name": {
            "ja": "送状状数値入力値1",
            "en": "invoice_number1"
        },
        "schema": {
            "type": "number"
        },
        "order": 14,
        "mode": "Invoice",
        "original_name": "invoice_number1"
    },
    "invoice_number2": {
        "name": {
            "ja": "送状状数値入力値2",
            "en": "invoice_number2"
        },
        "schema": {
            "type": "number"
        },
        "order": 15,
        "mode": "Invoice",
        "original_name": "invoice_number2"
    },
```

```
    "invoice_string1": {
        "name": {
            "ja": "送状文字入力値1",
            "en": "invoice_string1"
        },
        "schema": {
            "type": "string"
        },
        "order": 16,
        "mode": "Invoice",
        "original_name": "invoice_string1"
    },
    "invoice_string2": {
        "name": {
            "ja": "送状文字入力値2",
            "en": "inboice_string2"
        },
        "schema": {
            "type": "string"
        },
        "order": 17,
        "mode": "Invoice",
        "original_name": "inboice_string2"
    },
    "invoice_list1": {
        "name": {
            "ja": "送状状選択値1",
            "en": "invoice_list1"
        },
        "schema": {
            "type": "string"
        },
        "order": 18,
        "mode": "Invoice",
        "original_name": "invoice_list1"
    }
}
```

4.7 コンテナ起動

コンテナを起動します。

```
$ cd ${HOME}/rde-docker/
$ docker run -it --rm -v ${PWD}/app/container:/app2 -v ${PWD}/data:/app2/data rde-sample:v0.1 /bin/bash
root@b103229368dc:/app#
```

`app2/` フォルダに移動します。

```
cd /app2
```

4.8 スクリプト開発

本来の開発であれば、main.pyおよびmodules/フォルダ下のスクリプトを作成していくことになりますが、本章ではデータ同様にダミースクリプトを用意することにします。

> ここでは、別途提供されている『ハンズオンチュートリアル』で使用したものを使います。この構成が必ずしも推奨のものではないことに留意ください。

```
root@b103229368dc:/app2# ls -F
Dockerfile  data/  main.py  modules/  requirements.txt
```

4.8.1 main.py

```
import rdetoolkit
from rdetoolkit.config import Config, SystemSettings

from modules.datasets_process import dataset


def main():
    config = Config(
         system=SystemSettings(
             save_raw=False,
             save_nonshared_raw=False
         )
    )
    rdetoolkit.workflows.run(
        custom_dataset_function=dataset,
        config=config
    )

if __name__ == '__main__':
    main()
```

> すでにある main.py を、上記内容にて上書きします。inputdata/フォルダにあるファイルを、自動的にraw/フォルダおよびnonshared_raw/フォルダにコピーする機能を抑止するような設定を追加しています。

4.8.2 modules/datasets_process.py

上で確認したmodules/フォルダ配下に、実行したい処理が記述されたファイルを配置していきます。

最初に `main.py` の `custom_dataset_function` に指定した `dataset_process.py` を、以下の内容で作成します。

```
import io
import statistics as st
from pprint import pprint

import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
from rdetoolkit.errors import catch_exception_with_message
from rdetoolkit.exceptions import StructuredError
from rdetoolkit.models.rde2types import RdeInputDirPaths, RdeOutputResourcePath
from rdetoolkit.invoicefile import InvoiceFile
#from rdetoolkit.rde2util import read_from_json_file, write_to_json_file
from rdetoolkit.rde2util import Meta
```

```
def custom_module(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    # Check input file
    input_dir = srcpaths.inputdata
    input_files = list(input_dir.glob("*"))
    if len(input_files) == 0:
        raise StructuredError("ERROR: input data not found")
    elif len(input_files) > 1:
        raise StructuredError("ERROR: input data should be one file")
    else:
        raw_file_path = input_files[0]
        if raw_file_path.suffix.lower() != ".data":
            raise StructuredError(
                f"ERROR: input file is not '*.data' : {raw_file_path.name}"
            )
    # Read invoice file
    invoice_file = srcpaths.invoice  / "invoice.json"
    invoice = InvoiceFile(invoice_file)

    # Check public or private
    is_private_raw = False if invoice["custom"]["is_private_raw"] == "share" else True

    # Backup(=Copy) invoice.json to shared/nonshared folder
    raw_dir = resource_paths.nonshared_raw if is_private_raw else resource_paths.raw
    invoice_file_backup = raw_dir / "invoice.json.orig"
    InvoiceFile.copy_original_invoice(invoice_file, invoice_file_backup)

    # Rewrite invoice
    original_data_name = invoice.invoice_obj["basic"]["dataName"]
    additional_title = "(2024)"
    if original_data_name.find(additional_title) < 0:
        # update title if not applied yet
        invoice.invoice_obj["basic"]["dataName"] = original_data_name + " / " + additional_title
        # overwrite original invoice
        invoice_file_new = resource_paths.invoice  / "invoice.json"
        invoice.overwrite(invoice_file_new)

    # Read input data
    DELIM = "="
    raw_data_df = None
    raw_meta_obj = None
    #
    input_dir = srcpaths.inputdata
    input_file = input_dir / "sample.data"

    with open(input_file) as f:
        lines = f.readlines()
    # omit new line codes (\r and \n)
    lines_strip = [line.strip() for line in lines]

    meta_row = [i for i, line in enumerate(lines_strip) if "[METADATA]" in line]
    data_row = [i for i, line in enumerate(lines_strip) if "[DATA]" in line]
    if (meta_row != []) & (data_row !=[]):
        meta = lines_strip[(meta_row[0]+1):data_row[0]]
        # metadata to dict
        raw_meta_obj = dict(map(lambda x: tuple([x.split(DELIM)[0],
                    DELIM.join(x.split(DELIM)[1:])]), meta))
    else:
        raise StructuredError("ERROR: invalid RAW METADATA or DATA")
    # read data to data.frame
    if (int(raw_meta_obj["series_number"])!= len(data_row)):
        raise StructuredError("ERROR: unmatch series number")
    raw_data_df = []
    for i in data_row:
        series_name = lines_strip[i+1]
        cnt = int(lines_strip[i+2])
        csv = "".join(lines[(i+3):(i+3+cnt)])
        temp_df = pd.read_csv(io.StringIO(csv),header=None)
        temp_df.columns = ["x", series_name]
        raw_data_df.append(temp_df)
```

```
# Retrieve Meta data

# create Meta instance
metadata_def_file = srcpaths.tasksupport / "metadata-def.json"
meta_obj = Meta(metadata_def_file)

# get meta data
# #1 Read Input File
# -> input fileは前述の処理にて作成済み

# #2 Read Inovoice
# -> invoice.invoice_obj["custom"] は前述の処理にて作成(→読み込み)済み

# #3 From raw data numeric data part
s_name = []
s_count = []
s_mean = []
s_median = []
s_max = []
s_min = []
s_stdev = []
for df in raw_data_df:
    d = df.dropna(axis=0)
    y = d.iloc[:, 1]
    s_name.append(d.columns[1])
    s_count.append(len(y))
    s_mean.append("{:.2f}".format(st.mean(y)))
    s_median.append("{:.2f}".format(st.median(y)))
    s_max.append(max(y))
    s_min.append(min(y))
    s_stdev.append("{:.2f}".format(st.stdev(y)))

meta_vars = {
    "series_name": s_name,
    "series_data_count": s_count,
    "series_data_mean": s_mean,
    "series_data_median": s_median,
    "series_data_max": s_max,
    "series_data_min": s_min,
    "series_data_stdev": s_stdev,
}

# Merge 2 types of metadata
const_meta_info = raw_meta_obj | invoice.invoice_obj["custom"]
repeated_meta_info = meta_vars

# Set metadata to meta instance
meta_obj.assign_vals(const_meta_info)
meta_obj.assign_vals(repeated_meta_info)

# Write metadata
metadata_json = resource_paths.meta / "metadata.json"
meta_obj.writefile(metadata_json)

# Write CSV file(s)
for d in raw_data_df:
    fname = d.columns[1].replace(" ", "") + ".csv"
    csv_file_path = resource_paths.struct / fname
    d.to_csv(csv_file_path, header=True, index=False)

# Write Graph
title = const_meta_info["data_title"]
x_label = const_meta_info["x_label"]
y_label = const_meta_info["y_label"]

## by series
for d in raw_data_df:
    x = d.iloc[:, 0]
    y = d.iloc[:, 1]
```

```
        label = d.columns[1]
        fname = resource_paths.other_image / f"{label}.png"
        fig, ax = plt.subplots(figsize=(5, 5), facecolor="white")
        ax.plot(x, y, label=label)
        ax.set_title(title)
        ax.set_xlabel(x_label)
        ax.set_ylabel(y_label)
        ax.legend()
        fig.savefig(fname)
        plt.close(fig)

    ## all series
    fig, ax = plt.subplots(figsize=(5, 5), facecolor="lightblue")
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.set_title(title)
    for d in raw_data_df:
        x = d.iloc[:, 0]
        y = d.iloc[:, 1]
        label = d.columns[1]
        ax.plot(x, y, label=label)
    ax.legend()
    fname = resource_paths.main_image / "all_series.png"
    fig.savefig(fname)
    plt.close(fig)

    # Write thumbnail images
    src_img_file_path = resource_paths.main_image / "all_series.png"
    out_img_file_path = resource_paths.thumbnail / "thumbnail.png"
    closure_w = 286
    closure_h = 200

    Image.MAX_IMAGE_PIXELS = None   # オープンする画像のピクセルサイズ制限を解除する
    img_org = Image.open(src_img_file_path)
    ratio_w = closure_w / img_org.width
    ratio_h = closure_h / img_org.height
    ratio = min(ratio_w, ratio_h)
    img_re = img_org.resize((int(img_org.width * ratio), int(img_org.height * ratio)), Image.BILINEAR)  # image magickの>デフォルトに
合わせてバイリニアとしている
    img_re.save(out_img_file_path)


@catch_exception_with_message(error_message="ERROR: failed in data processing", error_code=50)
def dataset(srcpaths: RdeInputDirPaths, resource_paths: RdeOutputResourcePath) -> None:
    custom_module(srcpaths, resource_paths)
```

4.9 実行

構造化処理プログラムを動かしてみます。

```
$ docker run -it --rm -v ${PWD}/app/container:/app2 -v ${PWD}/data:/app2/data rde-sample:v0.1 /bin/bash

root@633a74823e8b:/app2# python main.py
Traceback (most recent call last):
  File "/app2/main.py", line 4, in <module>
    from modules.datasets_process import dataset
  File "/app2/modules/datasets_process.py", line 6, in <module>
    import matplotlib.pyplot as plt
ModuleNotFoundError: No module named 'matplotlib'
```

上記スクリプト実行のために必要なモジュールがインストールされていません。

Dockerイメージを作り直します。

`exit` コマンドを使いDockerコンテナから抜け、requirements.txtを以下の様に変更します。

```
$ vi dev_image/requirements.txt

$ cat dev_image/requirements.txt
rdetoolkit==1.2.0
matplotlib==3.10.3
```

> `matplotlib==3.10.3` の行を追記します。

Dockerイメージを再作成します。

```
cd dev_image
docker build -t rde-sample:v0.1  ./
```

> 同じ名前(とバージョン)で作っていますので、上書きになります。

再度実行(docker run)します。

```
$ cd ${HOME}/rde-docker
$ docker run -it --rm -v ${PWD}/app/container:/app2 -v ${PWD}/data:/app2/data rde-sample:v0.1 /bin/bash

root@38dc486bc3a5:/app# cd /app2
root@38dc486bc3a5:/app2# python main.py
root@38dc486bc3a5:/app2#
```

正常に終了しました。出力されるファイルを確認すると以下の様になっています。

```
root@38dc486bc3a5:/app2# tree data
data
├── attachment
├── inputdata
│   └── sample.data
├── invoice
│   └── invoice.json
├── invoice_patch
├── logs
├── main_image
│   └── all_series.png
├── meta
│   └── metadata.json
├── nonshared_raw
│   └── invoice.json.orig
├── other_image
│   ├── series1.png
│   ├── series2.png
│   └── series3.png
├── raw
├── structured
│   ├── series1.csv
│   ├── series2.csv
│   └── series3.csv
├── tasksupport
│   ├── invoice.schema.json
│   └── metadata-def.json
├── temp
└── thumbnail
    └── thumbnail.png

15 directories, 14 files
```

上記の例の場合、ファイル構成から、問題無く想定のファイルが出来ていることが確認できます。実際の開発では、ファイルの内容も含めて確認する必要があります。

なんらかの異常がある場合は、`data/job.faild` ファイルの内容や端末に表示されるエラー表示を確認して修正します。

4.10 2回目以降の実行

スクリプトを修正して、再度実行する場合は、以下のような点に注意してください。

4.10.1 job.failedファイルの削除

前回処理の結果と混同するなどの混乱を避けるため、`data/job.failed` が存在している場合は削除します。

```
root@38dc486bc3a5:/app# cd /app2
root@38dc486bc3a5:/app2# rm -f data/job.failed
```

> 前回の処理が正常に終了した場合は、`data/job.failed` ファイルは生成されません。削除するのは、当該ファイルが存在する場合のみです。

4.10.2 生成ファイルの削除

構造化処理プログラムの実行により生成したファイルを削除します。

> 削除せずに実行した場合、存在しているファイルが最新の実行によって生成されたのか、あるいは前回の実行によって生成されたのかが、すぐには判別できない場合があります。そういった事態を避けるために、実行の前に、前回の実行で生成したファイルの削除処理を行うことをお勧めします。

4.10.3 ログファイルのクリア

設定によっては、`data/logs/rdesys.log` にログを出力します。

基本的に追記なので、ログのクリアを実行しなくても特に問題はありませんが、必要に応じてログをクリアしてください。

```
root@38dc486bc3a5:/app2# :>data/logs/rdesys.log
```

> `:` は何もしないコマンドです。標準出力には何も出力しないため、結果rdesys.logが0バイトのファイルになります。ログファイルは存在していなければ新規に作成されますので、`rm data/logs/rdesys.log` でファイルを削除しても構いません。

4.10.4 実行 (あるいはデバッグ)

初回実行と同様に、構造化処理プログラムを実行します。

何らかの異常がある場合は、`main.py` や `modules/` フォルダ下のスクリプトファイルを修正します。

以後、望む状態になるまで、開発と実行を繰り返します。

4.11 (参考)スクリプトファイルやその他生成ファイルのオーナー

本書の手順通りに作成したDockerコンテナ内で作成したファイル、あるいは構造化処理プログラムの実行により生成されたファイルのオーナーは `root` ユーザです。

> 前章で、一部ファイルのオーナーの変更処理を実行しています。そのため、すべてのファイルのオーナーが `root` ユーザではありません。

Dockerコンテナ外からファイルを編集したい場合など、`Permission denied` により編集出来ない場合があります。

その場合は、`sudo` コマンドを利用するか、ファイル/フォルダのオーナーを変更する必要があります。

`sudo` コマンドを利用する例:

```
cd app/container
sudo vi modules/datasets_process.py
```

オーナーを変更する場合の例:

```
$ sudo chown -R devel modules （ユーザ"devel"を利用している場合）
$ vi modules/datasets_process.py
```

> ファイルを削除する場合も、上と同様に、`sudo` コマンドを利用するか、オーナーを変更してから削除処理を実行する必要があります。

5. 本番用Dockerイメージ作成

構造化処理プログラムの開発が一通り済んだら、そのプログラムファイルを組み込んだ、新しいDockerイメージを作成して、その確認を行います。

ここまで使っていたDockerイメージでは構造化処理プログラム、つまりPythonスクリプトはホスト側(Dockerを実行しているサーバ上、つまりローカル環境上)のフォルダ上にあり、Dockerコンテナを再起動してもその変更内容が消えずに有効になるような状態で開発をしていました。つまり、いわば `開発用コンテナ` を利用していたことになります。

実際のRDE実行環境で利用されるDockerイメージ、すなわち `本番用コンテナ` では、構造化処理プログラムはDockerイメージの中に含まれている必要があります。

本章では、構造化処理プログラムをDockerイメージ内に含めるようにDockerfileを作成し、Dockerイメージを再生成します。その後、そのDockerイメージを使ったコンテナを使い構造化処理に問題無いことを確認します。

この時点で、コンテナ内にいる場合は、抜けます。

```
root@38dc486bc3a5:/app2# exit
exit
$
```

5.1 フォルダ移動

`app/container` フォルダに、新しいDockerイメージを作成するためのひな形が用意されているので、これを変更していきます。

`app/container` に移動します。

```
cd ${HOME}/rde-docker
cd app/container
```

> 前述の様に、コンテナ外での編集には `sudo` の利用が必要となる場合があります。必要に応じて利用してください。

5.2 requirements.txt

RDEToolKitおよびその依存パッケージ以外のpipパッケージを利用する場合は、requirements.txtにそのパッケージ名を追記してください。

> 特に追加パッケージがない場合は、変更不要です。
>
> また、指定時は、以下の様にバージョン番号も含めての記述を推奨します。

requirements.txtの例：

```
rdetoolkit==1.2.0
pydantic-xml==2.12.1
```

上記の例の場合、`rdetoolkit バージョン1.2.0`とそれに伴う依存パッケージの他に、`pydantic-xml バージョン2.12.1` およびその依存パッケージが導入されます。

> "#"で始まる行はコメント行です。そのままでもいいですし、削除してしまっても構いません。

```
vi requirements.txt
```

前述の開発環境と同様に、`matplotlib`を指定します。

```
rdetoolkit==1.2.0
matplotlib==3.10.3
```

5.3 Dockerfile

`app/container/`フォルダにある、ひな形として提供されているDockerfileは以下の内容となっています。

```
FROM python:3.11.9

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY main.py /app
COPY modules/ /app/modules/
```

以下の様に書き換えます。

```
FROM python:3.12-bookworm

# treeコマンドインストール
RUN apt-get update && apt-get upgrade -y \
&& rm -rf /var/lib/apt/lists/*

# appディレクトリを作成
WORKDIR /app

# requirements.txt設置
COPY requirements.txt /app

# 必要なPythonライブラリのインストール
RUN pip install --upgrade pip \
 && pip install -r requirements.txt

# プログラムや設定ファイルなどをコピーする
```

```
COPY main.py /app
COPY modules/ /app/modules/
```

1. OSパッケージのアップデートは任意です。必要と思われる場合に追記してください。その他必要なOSパッケージが有る場合はここでインストール処理を追記してください。

> 開発用イメージとは異なり、treeコマンドのインストール や、vimの設定ファイルの設置などは実施していません。

1. pip自体のアップデート処理を追加します。

> 構造化処理プログラムの中で"RDEToolKitが必要とするpipパッケージ以外のpipパッケージ"を利用する場合は、requirements.txtにバージョン番号を含めて追記します。

1. `modules/` フォルダのコピー処理を追加します。

`container/` フォルダにいることを確認し、Dockerイメージを再構築します。

```
cd $HOME/rde-docker/app/container

docker build -t rde-sample:v1.0 ./
```

> 本番用イメージということで、バージョン番号を"v1.0"に変更しています。Proxy環境下にいる場合は、`$http_proxy` や `$https_proxy` などに適切な設定を行ってから再構築作業を行ってください。

5.4 確認

先に、開発中に作成されたファイルが残っているので、削除しておきます。

```
cd ${HOME}/rde-docker/

sudo chown -R $USER data/

rm -rf ./data/logs
rm -f ./data/job.failed
rm -rf ./data/main_image ./data/other_image ./data/meta ./data/raw ./data/nonshared_raw ./data/structured ./data/thumbnail
rm -rf ./data/attachment ./data/invoice_patch ./data/temp
```

> 前章で作成したプログラムでは、`data/raw/` フォルダにファイルを出力しませんが、RDEToolKitの設定を変更する前のテスト実行などでコピーされている場合があります。上記の様に削除して実行した後、意図せず `data/raw/` フォルダ下にファイルが存在する場合は、構造化処理プログラムを見直してください。

コンテナを実行します。

```
$ cd ${HOME}/rde-docker/

$ docker run -it --rm -v ${PWD}/data:/app2/data rde-sample:v1.0 /bin/bash
root@4443d3a722af:/app# cd /app2
root@4443d3a722af:/app2# python /app/main.py
```

treeコマンドはインストールされていないため実行できませんが、仮にDockerイメージ作成時にtreeコマンドをインストールした場合の実行例を以下に示します。

```
root@4443d3a722af:/app2# tree
.
└── data
    ├── attachment
    ├── inputdata
    │   └── sample.data
    ├── invoice
    │   └── invoice.json
    ├── invoice_patch
    ├── logs
    │   └── rdesys.log
    ├── main_image
    │   └── all_series.png
    ├── meta
    │   └── metadata.json
    ├── nonshared_raw
    │   └── invoice.json.orig
    ├── other_image
    │   ├── series1.png
    │   ├── series2.png
    │   └── series3.png
    ├── raw
    ├── structured
    │   ├── series1.csv
    │   ├── series2.csv
    │   └── series3.csv
    ├── tasksupport
    │   ├── invoice.schema.json
    │   └── metadata-def.json
    ├── temp
    └── thumbnail
        └── thumbnail.png

16 directories, 15 files
```

RDEToolKitを利用した構造化処理プログラムでは、data/ フォルダとしてカレントディレクトリにあるものが使われます。そのため、python main.py ではなく cd /app2 の後に、python /app/main.py を実行しています。

また、本来は入力データを nonshared_raw/ フォルダまたは raw/ フォルダのいずれかにコピーします。今回は、main.py の中で双方へのコピー処理を無効にしましたので、上記の様に入力データ(sample.data)がコピーされていません。双方無効にした場合は、コピー処理を追記する必要があることに注意してください。

以上

6. 変更履歴

1.3.0 (2025.05.26)

- 対応RDEToolKit : v1.2.0
- サンプルのPythonバージョンを3.11から3.12に変更

1.2.0 (2025.02.28)

- 対応RDEToolKit : v1.1.1
- 開発の章が4章と5章に分かれていた部分を1つの章に統合

1.1.0 (2025.01.15)

- 対応RDEToolKit : v1.0.2