Docker を用いた Risa/Asir の実行環境構築: つまずいた点とその解決

中野 竜之介

北海道大学大学院理学院数学専攻博士課程 2 年

上の QR コードから講演スライドにアクセスできます.

This work was supported by JST SPRING, Grant Number JPMJSP2119.

自己紹介

- 中野竜之介 (NAKANO Ryunosuke)
- 北海道大学大学院理学院数学専攻 D2
- 専門：特殊関数論と代数幾何の交わり
  - キーワード：Lauricella F_D, テータ関数, Thomae 型公式, 算術幾何平均
- 数学ソフトウェアとの関わり：
  - Julia と Maple を用いて計算を行い, 自身の結果の検証に利用している
- GKZ 超幾何系にも関心を持ち, グレブナー道場の購入および Risa/Asir の導入を行った

想定する聞き手と今日共有したいこと

想定する聞き手

- 数学ソフトウェアには関心がある
- Docker やコンテナにはまだ慣れていないかもしれない
- 研究室や講習会で同じ環境を配りたい

今日共有したいこと

- Dockerfile からイメージとコンテナができる流れ
- build が通っても安心できない理由
- log をどう分けて読むと見通しがよくなるか

今回は Docker 一般を網羅するのではなく, Risa/Asir の事例を手がかりに, 実行環境をどう整理したかを話す.

今日の流れ

1. Docker の最小語彙を確認する
2. `OpenXM` と `OpenXM_contrib2` の最小版 Dockerfile を見る
3. build.log を読みながら, 不足をどう切り分けたかを見る
4. つまずいた点を, PATH・文書生成・描画の三つに分けて整理する
5. 完全版 Dockerfile を, 用途別の束として読む

なぜ Docker の話が必要だったか

- README どおりに入れたつもりでも, 手元の環境差で止まることがある
- 依存パッケージや PATH が暗黙だと, どこが前提条件なのか共有しにくい
- 半年後には自分でも再現しづらく, 試行錯誤の履歴も散らばりやすい
- Docker が役に立つのは, 自分の手元で動かすだけでなく, 他人に渡せる形で残したいとき
- 今回は, 完成版 Dockerfile を見せるより, どこでつまずき, どう切り分けたかを共有したい

Docker の最小語彙

Dockerfile

- 環境を作る手順書
- `FROM`, `RUN`, `COPY`, `CMD` を書く

イメージ

- 実行前のひな型
- 読み取り中心のテンプレート

コンテナ

- イメージから起動した実体
- 一時的な書き込み層を持つ

Dockerfile $\xrightarrow{\texttt{docker build}}$ イメージ $\xrightarrow{\texttt{docker run}}$ コンテナ

コンテナは仮想マシンと何が違うか

コンテナ

- ホスト側のカーネルを共有して動く
- 起動が比較的軽い
- 同じカーネル系列で使うことが前提になる

仮想マシン

- guest OS ごと持つ
- 分離は強いが, そのぶんやや重い
- OS 全体を別に持っていると考えやすい

今回押さえたいのは, コンテナは「完全に別の OS」ではなく, ホスト側のカーネルを共有して動いているという点である.

イメージとコンテナを層として見る

コンテナの書き込み層
OpenXM の build 成果物
`OpenXM` と `OpenXM_contrib2` の取得
依存パッケージの導入
`FROM ubuntu:24.04`

- `RUN` を 1 行足すたびに, イメージの層が 1 枚増えると思うと見通しがよい
- イメージは読み取り中心で, コンテナはその上に一時的な書き込み層を載せて動く
- Dockerfile を直すことは, 環境の層の作り方を直すことに対応する

数学ソフトウェアと Docker の相性

向いている点

- 研究用ソフトの依存関係を環境ごと固定しやすい
- 同じ環境を配りやすい
- 手元の OS を汚さず複数版を試しやすい
- 試行錯誤を Dockerfile として残せる

注意点

- GUI, 日本語処理, 印刷系は追加設定が要ることが多い
- コンテナ内の書き込みは, そのままでは永続化されない
- 依存関係の意味付けまでは自動では整理されない

今回の題材

- `OpenXM` と `OpenXM_contrib2` のソースを取得して build する
- 最初は「最低限のパッケージで最後まで build が通る Dockerfile」を作る
- その後, log を読みながら不足していたものを足していく

最小の Dockerfile → log の整理 → 完全版 Dockerfile の読み方

成功した Dockerfile そのものより, 途中で出た error をどう読むかを共有したい.

最小の Dockerfile

```
FROM ubuntu:latest

RUN apt-get update && apt-get upgrade -y && apt-get install -y \
    git build-essential autoconf curl \
    libncurses5-dev libncursesw5-dev libncurses-dev \
    libtinfo-dev bison file flex xorg-dev \
    && apt-get clean && rm -rf /var/lib/apt/lists/*

RUN git clone https://github.com/openxm-org/OpenXM
RUN git clone https://github.com/openxm-org/OpenXM_contrib2

WORKDIR /OpenXM/src
RUN make configure
RUN make install
```

まずは「最後まで build が通るか」だけを見て, 観測の基準点を作る.

README の手作業を Dockerfile に写す

手元で打つとき

- apt で必要なものを入れる
- `git clone` でソースを取る
- `/OpenXM/src` へ移動する
- `make configure`
- `make install`

Dockerfile にすると

- `RUN apt-get install ...`
- `RUN git clone ...`
- `WORKDIR /OpenXM/src`
- `RUN make configure`
- `RUN make install`

configure とは何か

環境確認

- gcc があるか
- ヘッダやライブラリが見つかるか

小さな試験

- 短いコードを実際に compile する
- 使える機能をその場で判定する

設定生成

- Makefile や `config.h` を生成する
- 環境に応じて build 条件を決める

Docker 上で configure が止まるときは, かなりの確率で依存関係が足りていない.

build のしかたと log の残し方

```
DOCKER_BUILDKIT=1 docker build --no-cache -t risa-asir . \
  2>&1 | tee build.log
```

- `DOCKER_BUILDKIT=1`：build の仕組みを新しい実装で有効にする
- `--no-cache`：以前の層を使い回さず, 毎回きちんと検証する
- `tee build.log`：標準出力と標準エラーを保存して後から読めるようにする

成否だけでなく, 「何が見つからなかったか」を記録として残すことが大事だった.

生の log をまず眺める

```
./configure: line 2703: 0: command not found
checking for library containing rl_ding... configure: WARNING:
  GNU readline not found - falling back to builtin readline
Warning: Mathematica is not found. ox_math will not be compiled.
** node_up ‘D 加群の制限’ for ‘nk_restriction (option)’ not found
```

同じ `not found` でも, コマンド不足, 依存不足, 任意機能の省略, 文書生成の警告が混ざっている.

build が通ってもすぐには使えない

見た目

- `docker build` は最後まで通った

実際には

- `not found`
- `command not found`
- `cannot find`

といった記録が残っていた

ここで見ている「通った」は最低限であって, 文書生成や描画機能まで含めて十分とは限らない.

log 中の警告は同じ重みではない

例	その場の見方	今回の扱い
`Mathematica is not found`	任意機能の不足	ひとまず保留
`GNU readline not found`	依存パッケージ不足の可能性	後で入れる候補にする
`node_up ... not found`	文書生成側の警告らしい	本体 build とは分けて考える
`0: command not found`	設定スクリプト側の異常かもしれない	優先して確認する

全部を同じ熱量で追うのではなく，「今は保留」「後で直す」「優先して見る」に分けると疲れにくい．

log を大きく 2 種類に分ける

コマンドそのものが無い系

- `md5.sh`
- `uudecode`
- `javac`
- `ptex`
- `pstoimg`
- `makeinfo`

ライブラリの検出に失敗する系

- `libcerf`
- `cairo`, `pango`, `glib-2.0`
- `Qt5Core`, `Qt5Gui`, `Qt5Svg`

まずは「実行ファイルが無いのか」「開発用ライブラリが足りないのか」を分けて読むと見通しがよかった.

log から見えてきたこと

- 数値計算の本体依存だけでなく, 文書生成や画像出力の系統もかなり出てくる
- configure は, 使うかどうか未定の機能まで広く探しに行くことがある
- したがって, パッケージを機械的に足すのではなく, 用途ごとに整理しながら足す必要がある
- 今回は, 本体依存, 文書生成, 描画・GUI の 3 群に分けると理解しやすかった

つまずき 1：md5.sh と PATH

```
WORKDIR /OpenXM/src
ENV PATH="/OpenXM/bin:${PATH}"
RUN make configure
RUN make install
```

- md5.sh は OpenXM が提供するスクリプトで, /OpenXM/bin に置かれる
- パッケージ不足ではなく, 既にあるのに PATH に入っていないため見つからなかった
- not found を見たときは, APT の不足だけでなく PATH も疑う必要がある

つまずき 2：TeX や HTML 生成まわり

代表例

- `texinfo`
- `texi2html`
- `texlive-extra-utils`
- `latex2html`
- `ghostscript`
- `netpbm`

背景と見方

- configure や文書生成が, Texinfo や LaTeX や画像変換の補助ツールまで見に行く
- 「CAS 本体を入れたいだけなのに」と感じやすいが, 本体依存と文書生成依存を分けて説明すると納得しやすい

つまずき 3：描画や GUI まわり

代表例

- `libcairo2-dev`
- `libpango1.0-dev`
- `libglib2.0-dev`
- `qtbase5-dev`
- `qttools5-dev`
- `libqt5svg5-dev`

読み方

- gnuplot の cairo 系出力や Qt 端末に関する依存がここに現れている
- 講習会用途なら全部入りでもよい
- 計算専用なら, 任意依存として切り出す設計も考えられる

不足と修正の対応表

見えた不足	読み方	今回の対応
`md5.sh not found`	PATH の問題	`ENV PATH="/OpenXM/bin:$PATH"`
`Qt5Core`, `Qt5Gui`, `Qt5Svg`	Qt 開発用の不足	`qtbase5-dev`, `libqt5svg5-dev`
`GNU readline not found`	readline 開発用の不足	`libreadline-dev`
`pstoimg`, `makeinfo`	文書生成系の不足	`latex2html`, `texinfo`
`cairo`, `pango`, `glib-2.0`	描画系の `.pc` 不足	`libcairo2-dev`, `libpango1.0-dev`, `libglib2.0-dev`

error を見たらすぐ検索するのではなく, 何の系統の不足かを一度言い換えると修正が早くなる.

完全版 Dockerfile は用途別の束として読む

ビルド本体

- build-essential
- autoconf
- bison
- flex
- file
- git
- pkg-config

本体依存

- libcerf2
- libcerf-dev
- libncurses5-dev
- libncursesw5-dev
- libncurses-dev
- libtinfo-dev

文書生成

- texinfo
- texi2html
- latex2html
- texlive-*
- ghostscript
- netpbm

描画・GUI

- libcairo2-dev
- libpango1.0-dev
- libglib2.0-dev
- qtbase5-dev
- qttools5-dev
- libqt5svg5-dev

長い列挙をそのまま覚えるより, 何のための依存かをコメント付きで読む方が共有しやすい.

用途によって Dockerfile は変わる

講習会や配布用

- 文書生成や描画も含めて全部入りにする
- 「あの機能だけ動かない」を減らしやすい
- build 時間や image サイズはやや重くなる

個人用の計算専用

- CLI 中心なら任意依存を切り出しやすい
- build 時間や image サイズを抑えやすい
- ただし, 後から足したい機能の説明は別に要る

この事例から自分が学んだこと

1. 最初から全部入りを目指さず, 最小版から始める
2. log を, コマンド不足・ライブラリ不足・PATH の問題に分けて読む
3. 依存を用途別に書き分けておく
4. 将来的には Ubuntu の版固定も考える (今回は Ubuntu 24.04 LTS)

将来的にやりたい改善

- `FROM ubuntu:24.04` のように, ベースの版を固定して再現性を上げる
- 本体依存と任意依存をもう少し分けて, 軽い版も作れるようにする
- build 後に `which asir` や `pkg-config` を走らせる確認手順も残す

今回は「まず動かす」が目標だったので, 今後は再現性と説明しやすさをもう少し整えたい.

container の中に入ったあと, まず何を見るか

```
which asir
which md5.sh
echo $PATH
pkg-config --modversion Qt5Core
```

- `which` でコマンドが見えるかを確認する
- `PATH` で「あるのに見えない」問題を切り分ける
- `pkg-config` でライブラリ検出が通るかを確認する

build.log だけでなく, container の中で短い確認コマンドを打つと, 修正の当たりが付けやすい.

最初に試すならこの 3 コマンド

1. イメージを作る

```
docker build -t risa-asir .
```

2. コンテナの中に入る

```
docker run --rm -it risa-asir /bin/bash
```

3. 手元の作業ディレクトリを見せる

```
docker run --rm -it -v "$PWD":/work \
  -w /work risa-asir /bin/bash
```

まとめ

最後に共有したい 3 点

- Docker は, 再現可能な環境を配るための道具として有用
- build が通ったかどうかだけでなく, log の読み取りが大事
- 依存関係は, 用途別に整理して書くと共有しやすい

ありがとうございました