特別研究報告

題目

Docker における複数環境対応のための Dockerfile プリプロセッサの調査

指導教員

楠本 真二 教授

報告者

馬渕 航

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

Docker における複数環境対応のための Dockerfile プリプロセッサの調査

馬渕 航

内容梗概

Docker は可搬性や資源効率の高さからコンテナ仮想化におけるデファクトスタンダードである．Docker コンテナにはベースとなる OS の種類やサービスのバージョン等によって複数の利用形態が存在する．コンテナ配布者は利用形態に合わせて複数の Dockerfile（コンテナ構築手順が記載されたソースコード）を用意することが一般的である．ただし，手動の管理ではなく利用形態に合わせて Dockerfile を複数生成する Dockerfile プリプロセッサ（以降 DPP）を利用し管理している．しかし，Docker 自体は DPP をサポートしておらず，コンテナ配布者は自前の DPP を作成し利用している．その実現方法はプロジェクトによって多種多様であり，手探りでの開発が求められる．そこで，本研究では Dockerfile プリプロセッサの実現方法の体系化を目的として調査を行う．調査の結果，DPP はその構造によって複数の型に分類でき，各型に利点や欠点が存在した．

主な用語

コンテナ仮想化，Docker，Dockerfile，プリプロセッサ，実証的調査

目次

図目次

表目次

1　はじめに

Docker はコンテナ仮想化を実現するプラットフォームである．コンテナ仮想化は一般的なハイパーバイザ仮想と比べて高い資源効率や可搬性を実現する．Docker はコンテナ仮想化におけるデファクトスタンダードであり，IT 企業の 79% 以上が利用しているとの報告がある [1]．また，OSS のようなソフトウェア開発プロジェクトでも広く採用されており [2]，ソフトウェアの配布方法としての利用のみならず，ソフトウェアの開発支援としても活用されている [3][4]．さらには学術分野における実験の再現性確保としての活用も期待されている [5]．Dockerfile は，Docker イメージを生成するためのソースコードである．イメージはコンテナを稼働させるための静的情報であり，Dockerfile から生成されたイメージを起動させることでコンテナが稼働する．また，Dockerfile により生成されたイメージを配布することで，誰がいつ実行しても同様のコンテナを再現可能である [6]．この性質を利用し，Docker Hub では多くの企業や開発者がイメージを公開している．

Docker におけるコンテナ開発においては，複数の環境や利用形態を想定し，サポートすることが一般的である．例えば，コンテナのベースとなる OS（ubuntu や centos）や，提供するサービスのバージョン（1.0 や 2.0）, Docker のホストマシンのアーキテクチャ（AMD364 や i386）など，様々な利用形態の組み合わせが存在する．これらの組み合わせによって Dockerfile 内の記述内容は変化する．特に，ベース OS の違いはコンテナで実行されるコマンドの違いに繋がるため，Dockerfile の記述内容は大きく変化することになる．コンテナ開発者は，これら多種多様な利用環境に応じた複数の Dockerfile を用意する必要がある．そこで，複数の Dockerfile の開発保守を目的として，Dockerfile のプリプロセッサが採用されることがある．DPP では生成対象となる Dockerfile の枠組み，及びベース OS やバージョン等の可変な情報を入力として，複数種類の Dockerfile を自動生成する．

しかしながら，Docker 自体は DPP をサポートしておらず，既存のツールも存在しない [7]．そのため，開発者は独自で DPP を作成しなければならず，その実現方法もプロジェクトによって異なる．DPP の基本的な処理は，テンプレートファイルに対する具体的な値の埋め込み処理で実現可能である．よって sed や awk 等のテキストプロセッサが利用されることが多い．一方で，Python のような高級言語を用いてより高度な DPP 処理を利用している場合もある．よって，DPP 全体の処理をどのように構成するかは様々な選択肢があるといえる．また，置換対象となる可変情報をどのように管理するか，どのような置換情報の組み合わせが存在するかといった点も明らかではない．Docker 開発者は，統一的な実現方法がない状況下で手探りでの DPP 作成が求められる．

そこで，本研究では DPP 実現方法の体系化を目的として，DPP の実態調査を行う．この調査を通じて，DPP で広く採用されている手法やその利点と欠点を明らかにし，DPP 採用を検討している開発者への手がかりを提供できると考える．具体的な調査手法としては，Docker Hub 内の高品質なコンテナ

を対象に，DPP を採用している GitHub プロジェクトを抽出し目視調査を行う．そして，実現方法をパターン化する．さらに各実現方法の利点や欠点を整理し，プロジェクトの性質との相性を加味した分析も行う．以下に示す 4 つの RQ に従って調査を進める．

RQ1: DPP はどの程度利用されているか?

RQ1 では，Dockerfile 管理プロジェクトにおいて，DPP がどの程度採用されているかを調査する．この調査を通じて，DPP の採用率の把握を狙う．

RQ2: DPP はどのように構成されるか?

RQ2 では，RQ1 で発見した DPP 採用プロジェクトを対象として，DPP 全体のファイル構成や各ファイルの役割を調査する．この調査で，DPP を構成要素で分け，後の詳細な調査に必要なファイルの抽出を狙う．

RQ3: DPP における Dockerfile 生成処理はどのように実現されるか?

RQ3 では，RQ2 で整理した DPP 構成ファイルの内，Dockerfile を生成するファイルの中身を精査する．この調査で，Dockerfile 生成の実現方法のパターン化を狙う．

RQ4: DPP における可変情報はどう管理されるか?

RQ4 では，RQ2 で整理した DPP 構成ファイルの内，可変情報を管理するファイルの中身を精査する．この調査を通じて可変情報の種類を明らかにし，管理方法の特徴を整理する．

```
1  FROM alpine
2  ..
3  ENV PYTHON_VERSION 3.11.1
4  RUN set -eux; \
5      apk add --no-cache --virtual .build-deps \
6      gnupg \
7      ..
8      wget python.tar.xz "https://../Python-$PYTHON_VERSION.tar.xz"; \
9  ..
10 CMD ["python3"]
```

図 1　Python 実行環境コンテナを生成する際の Dockerfile

2　準備

2.1　Dockerfile

Dockerfile とは，コンテナ生成手順が記載されたソースコードである．同一ファイル内に，Dockerfile 固有の構文と RUN 命令に内包される Shell Script 構文 [8] を用いて，ベース OS やコンテナ内で実行する命令を記述する．図 1 に Python 実行環境を提供するコンテナを生成する際の Dockerfile の例[*1] を示す．図 1 では，まず FROM 命令でベース OS となる Alpine を選択している．そして，RUN 命令を用いて，パッケージのインストールや URL 先のファイルの取得など，コンテナ内で実行するコマンド列が記載させている．最後に，サービスの実行を行う CMD 命令が記述されている．以上のように，Dockerfile を作成し配布することで，誰でも，どの環境でも Python 実行環境をサービスとするコンテナを再現可能である．

2.2　複数環境対応のための Dockerfile

Dockerfile の作成や配布においては，複数の利用形態を想定しサポートすることが一般的である．利用形態の例としては，コンテナのベースとなる OS やサービスのバージョン等，様々な組み合わせが考えられる．この組み合わせによって Dockerfile で記述する内容は変化する．図 1 で示した Dockerfile は，ベース OS が Alpine，Python のバージョンが 3.11 の場合の Dockerfile であった．ここで，ベース OS を Buster（Debian 系 OS）, Python のバージョンを 3.7 にした場合の Dockerfile の例[*2] を図 2 に示す．ENV で指定する変数の値が異なっているほか，パッケージマネージャが apk から apt-get に変わっていることが読み取れる．さらに，ベース OS が Windows の場合の例[*3] を図 3 に示す．図 3 から Linux 系の OS とは記述が全く異なることが読み取れる．コマンド名が違うことは当然ながら，図 3

[*1] https://github.com/docker-library/python/blob/master/3.11/alpine3.16/Dockerfile

[*2] https://github.com/docker-library/python/blob/master/3.7/buster/Dockerfile

[*3] https://github.com/docker-library/python/blob/master/3.11/windows/windowsservercore-1809/Dockerfile

```
1  FROM buildpack-deps:buster
2  ..
3  ENV PYTHON_VERSION 3.7.16
4  RUN set -eux; \
5      savedAptMark="$(apt-mark showmanual)"; \
6      apt-get update; \
7      apt-get install -y --no-install-recommends \
8      patchelf \
9      ; \
10     wget python.tar.xz "https://../Python-$PYTHON_VERSION.tar.xz"; \
11 ..
12 CMD ["python3"]
```

図 2　図 1 からベース OS を buster, バージョンを 3.7 にした場合の Dockerfile

```
1  FROM mcr.microsoft.com/windows/servercore:1809
2  ..
3  ENV PYTHON_VERSION 3.11.1
4  RUN $url = ('https://../python-{1}-amd64.exe' -f ($env:PYTHON_VERSION -replace '[a
       -z]+[0-9]*$', ''), $env:PYTHON_VERSION); \
5      Write-Host ('Downloading {0} ...' -f $url); \
6      [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::
           Tls12; \
7      Invoke-WebRequest -Uri $url -OutFile 'python.exe'; \
8      Write-Host 'Installing ...'; \
9  ..
10 CMD ["python"]
```

図 3　図 1 からベース OS を Windows にした場合の Dockerfile

の 6 行目に記述されているプロトコル変更文のように，新しく追加される文も存在する．配布者は以上のような記述内容の違いを反映させ，各種組み合わせに対応した Dockerfile を用意する必要がある．

2.3　Dockerfile のためのプリプロセッサ（DPP）

複数の利用形態に応じた Dockerfile の作成にあたっては，Dockerfile プリプロセッサ（DPP）が利用されることがある．開発者が DPP に対して可変情報や Dockerfile のテンプレート情報を指定する．DPP は入力された可変情報が示す利用形態に対応した Dockerfile を自動生成する．

Python コンテナの Dockerfile を生成する DPP の概略を図 4 に示す．図 4 では，Alpine 等のベース OS や 3.11 等のバージョン情報等の可変情報及び Dockerfile のテンプレートとなるテキスト情報を入力としている．DPP は入力情報を基に処理をおこない出力として複数の Dockerfile を生成している．

同様の仕組みとして，C 言語ではマクロ構文を用いた条件付きコンパイル [9] が多用されている．このような仕組みを採用することで，開発者は複数種の Dockerfile を一度に管理でき，手間の軽減やバグの低減に繋がる．

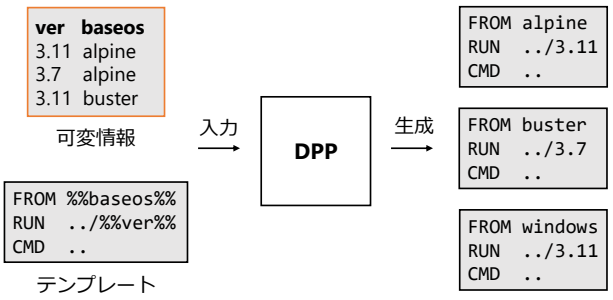

図 4　DPP の概略

2.4　DPP における課題

DPP は複数 Dockerfile の管理において便利であるが，DPP 採用の過程に課題が存在する．まず，Docker 自体は DPP を提供していないため，開発者は自前で DPP を用意しなければならない．また，統一的な方法も存在せず，DPP の実現方法はプロジェクトによって異なる．Dockerfile のテンプレートとなるファイルに可変情報を置換して生成する方法や awk などのスクリプト言語を用いてテンプレートを処理する方法，さらにはテンプレートすら持たない方法まで存在する．多種多様な DPP が存在する状況下で，開発者は手探りでの DPP 開発が求められる．

3 Reserch Questions

本研究では DPP 採用プロジェクトを対象に，DPP の実態を調査する．この調査を通じて様々な開発者が培った DPP 実現方法の体系化を狙う．調査は以下 4 つの問いに従って進める．

RQ1. DPP はどの程度利用されているか?

RQ1 では，Docker Hub の複数のプロジェクトを対象として，DPP を実際に採用しているプロジェクトの割合を調べる．前章で示した，Python や nginx のコンテナの例では DPP が採用されていたが，どの程度 DPP が採用されているかは不明である．本 RQ により研究の前提となる DPP の採用率を確認する．

RQ2. DPP はどのように構成されるか?

RQ2 では，RQ1 で発見した DPP 採用プロジェクトのソースコードを確認し，DPP 全体がどのように．実現されているかを確認する．仮説としては DPP は以下 2 種類の情報で構成されると考えられる．

- Dockerfile 生成処理
- 可変情報の管理

本調査で，上記 2 つの要素の存在含め，DPP を構成するファイルやその役割を明らかにし，RQ3 と RQ4 で調査するファイルを特定する．

RQ3. DPP における Dockerfile 生成処理はどのように実現されるか?

RQ3 では，Dockerfile 生成の実現方法について調査する．複数 Dockerfile の自動生成は DPP の目的に直結する処理であり，体系化において重要な要素となる．また，プロジェクトによって使用言語や生成する Dockerfile が異なるため，開発者によって実現方法のバリエーションが現れる部分であると考える．本 RQ にて，Dockerfile 生成処理の実現方法をパターン化し，各パターンの利点や欠点を整理する．

RQ4. DPP における可変情報はどう管理されるか?

RQ4 では，可変情報の処理に焦点を当てて，調査を実施する．可変情報の種類数によって，DPP が生成する Dockerfile の数は変わる．例えば，コンテナのベースとなる OS（Alpine, Buster）を可変情報とするとき，2 種類の Dockefile が生成される．よって，可変情報の種類や管理方法は，DPP の実現

方法に大きく影響を与えると考える．本 RQ にて，DPP で管理する可変情報を明らかにし，その管理方法を整理する．

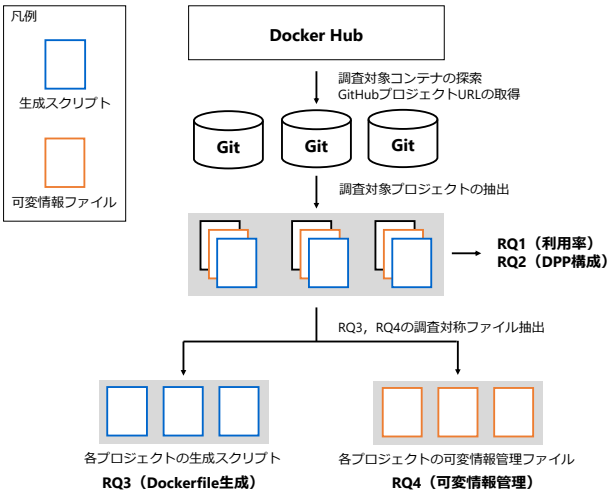

図 5　調査全体の流れ

4　調査手法

4.1　概要

本章では，今回の研究で実施した調査の流れを示す．図 5 に示すのは，全体の大まかな流れである．まず，図 5 の上部に示すのは，DPP の構成ファイル群を抽出するまでの流れである．最初に，Docker Hub から GitHub リポジトリの URL を取得する．次に，GitHub リポジトリ群から DPP を含むプロジェクトを目視で抽出する．図 5 の下部に示すのは，データの調査段階である．抽出した各プロジェクトから，DPP の構成要素となるファイル群を目視で確認し，抽出する．そして，RQ2 の調査として構成要素を整理する．さらに，各構成要素のファイルの内容を目視で調査し，各調査結果を RQ3，RQ4 の回答として整理する．以降の節で，調査の詳細を述べる．

4.2　DPP 構成ファイル群の抽出

調査対象プロジェクトの抽出段階では以下の 3 つの作業を実施する．

- 調査対象コンテナの探索
- コンテナを管理する GitHub プロジェクトの URL を取得
- 調査対象コンテナの抽出

最初に，調査対象のコンテナを Docker Hub から探す．Docker Hub 内には，Docker が高品質と認めたコンテナが存在し，開発者の特性によって，Docker official image，Verified Publisher，Sponsoured OSS の 3 種類からいずれかの公認マークが付けられる．例としては，Python や amazon/aws-cli 等の

コンテナが存在し，利用者も多い．今回の調査では，以上のような公認コンテナの内，32 件を対象とする．

次に，GitHub の URL を Docker Hub から取得する．大抵のコンテナは，Docker Hub 内のコンテナ概要欄ページに Dockerfile のソースコードを示す GitHub の URL を公開している．調査対象のコンテナ概要欄から URL を探し，取得する．ただし，URL を取得できなかったコンテナは調査対象から外し，新たなコンテナを調査対象に追加する．また，一部管理者が同じ複数のコンテナが存在する．この場合管理方法も同様であるため，同管理者の 2 件目以降は調査対象から外し，新たなコンテナを追加する．

4.3 各 RQ の調査

まず，RQ1 の調査を行う．データの抽出段階で計 32 件の GitHub プロジェクトの URL が取得できているため，各プロジェクトのファイル群を目視で調査し，DPP が採用されているプロジェクトの数を調べる．また，DPP 採用プロジェクトは次の RQ2 へ調査を進める．

RQ2 では，DPP の構造を調査する．RQ1 で発見した DPP 採用プロジェクト 32 件の内 15 件を対象とし，DPP を構成するファイルを目視で抽出する．そして，各ファイルの DPP における機能を整理し，共通する機能を見つける．各ファイルの中で，Dockerfile の生成を行うファイルは RQ3 で，可変情報の処理に関わるファイルは RQ4 でさらに詳しく調査する．

RQ3 では Dockerfile 生成の実現方法を調査する．RQ2 で発見した，Dockerfile の生成処理を行うファイルを対象に，実装言語や実現パターンを調査する．

RQ4 では可変情報の処理方法を調査する．RQ3 と同様に，RQ2 の調査で発見した，可変情報を扱うファイルを対象に，実装言語や実現パターンを調査する．

```
1 FROM alpine:%%ALPINE_VERSION%%
2
3 LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"
4
5 ENV NGINX_VERSION %%NGINX_VERSION%%
6 ENV NJS_VERSION   %%NJS_VERSION%%
7 ENV PKG_RELEASE   %%PKG_RELEASE%%
```

図 6　ひな形ファイルの例

5　調査結果

5.1　RQ1: DPP はどの程度利用されているか?

GitHub から収集した 32 件を対象に，DPP の採用割合を調査した結果，87.5%(32 件中 28 件) が DPP を採用していた．DPP はほとんどの Dockerfile 管理プロジェクトで採用されていることが分かる．例としては，Python，nginx，amazonlinux 等あらゆるサービスのプロジェクトが DPP を採用していた．プログラム言語の実行環境やサーバ環境等のサービスの性質に関わらず，DPP は採用されることが分かる．

> DPP はあらゆるサービスのプロジェクトに採用されている．

5.2　RQ2: DPP はどのように構成されるか?

RQ1 で収集したプロジェクトの中から，15 件を対象に構成ファイルの精査を実施した．その結果 DPP は，以下に示す 3 種類の要素で構成されることが分かった．

- ひな形
- 可変情報
- 生成スクリプト

ひな形は，Dockerfile に近い構造を持つテキスト情報であり，docker-BASEOS.template のようなファイル名で存在する．図 6 に nginx プロジェクト[*4] のひな形の一部を示す．ベース OS や，バージョンなどの可変情報が変数として設定されている．

図 7 に nginx コンテナ管理プロジェクトの可変情報の記述例[*5] を示す．nginx のバージョン情報

[*4] https://github.com/nginxinc/docker-nginx/blob/d039609e3a537df4e15a454fdb5a004d519e9a11/Dockerfile-alpine.template

[*5] https://github.com/nginxinc/docker-nginx/blob/d039609e3a537df4e15a454fdb5a004d519e9a11/Dockerfile-alpine.template

```
declare -A nginx=(
    [mainline]='1.21.6'
    [stable]='1.20.2'
)
..
declare -A debian=(
    [mainline]='bullseye'
    [stable]='bullseye'
)
declare -A alpine=(
    [mainline]='3.15'
    [stable]='3.14'
)
```

図 7　可変情報の記述例

```
for variant in \
  alpine{,-perl} \
  debian{,-perl}; do
  ..
  template="Dockerfile-${variant%-perl}.template"
  {
    cat "$template"
  } >"$dir/Dockerfile"
  ..
  sed -i \
    -e 's,%%ALPINE_VERSION%%,'"$alpinever"',' \
    ..
```

図 8　生成スクリプトの例

（1.21.6 や 1.20.2）やベース OS（Bullseye）の情報が変数として定義されている．

生成スクリプトは，実際に Dockerfile の生成を実行するスクリプトである．ひな形のテキスト情報や可変情報をもとに，生成ファイルに Dockerfile の命令を書き込む．図 8 に nginx プロジェクトの生成スクリプトの例[*6] を示す．5 行目にてひな形の内容を生成する Dockerfile に書き込み，ひな形内の変数を 10 行目から sed を用いて置換している．そしてそれらの処理を各ベース OS に対しておこなっていることが 1 行目から 3 行目の記述で分かる．

以上のように，DPP の構成要素は 3 つ存在する．RQ3 では，ひな形と生成スクリプトを対象に，Dockerfile 生成の実現方法を調査する．RQ4 では可変情報に着目し，その管理方法を調査する．

> DPP は生成スクリプト，ひな形，可変情報の 3 種類の要素で構成される．ひな形や可変情報の要素は生成スクリプトに含まれる場合もある．

[*6] https://github.com/nginxinc/docker-nginx/blob/d039609e3a537df4e15a454fdb5a004d519e9a11/update.sh

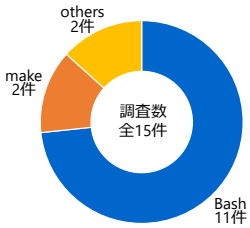

図 9　生成スクリプトの記述言語別採用数とその割合

5.3　RQ3: DPP における Dockerfile 生成処理はどのように実現されるか?

RQ3 では，生成スクリプトやひな形ファイルを対象に，Dockerfile 生成がどのように実現されているかを調査した．その結果を記述言語と実現方法の 2 つの観点から述べる．

5.3.1　記述言語

生成スクリプトの記述言語の種類を調査した結果を図 9 に示す．一番多い言語は Bash であり，15 件中 11 件が採用していた．Linux のデフォルトのログインシェルであることから，高速かつ実装しやすい点が，高い採用率につながっていると考える．その他にも，make や Python 等が採用されていた．ひな形の置換処理を実現しやすい点が採用理由として考えられる．

5.3.2　実現方法

次に，生成スクリプトの記述内容から，Dockerfile 生成の実現方法を調査した．その結果，以下に示す 3 種類の実現パターンの存在を確認した．

- テンプレート型
- メソッド型
- ハイブリッド型

テンプレート型は，ひな形ファイル内の変数を可変情報に応じて置換しながら Dockerfile を生成する方法である．ひな形ファイルと生成スクリプト 2 種のファイルが必ず存在し，生成スクリプトからひな形ファイルの変数を sed 等を用いて置換していく．ひな形ファイル自体が Dockerfile の構造を保っているため，生成対象となる Dockerfile の見通しが立ちやすく，可読性の高い手法と言える．一方で，ひな形ファイルがそのプロジェクトに特化していることが多く，他のプロジェクトへの移植性は低いと言える．また，このテンプレート型はひな形の変数置換方法でさらに分類でき，単純な文字列置換のケースと，スクリプト言語を用いた複雑な置換のケースが存在した．

```
sed -i \
  -e 's,%%ALPINE_VERSION%%,'"$alpinever"',' \
  -e 's,%%DEBIAN_VERSION%%,'"$debianver"',' \
  -e 's,%%NGINX_VERSION%%,'"$nginxver"',' \
  ..
```

図 10　生成スクリプトに含まれる単純な置換処理

```
{{ if is_alpine then ( -}}
FROM alpine:{{ env.variant | ltrimstr("alpine") }}
{{ ) elif is_slim then ( -}}
FROM debian:{{ env.variant | ltrimstr("slim-") }}-slim
{{ ) else ( -}}
FROM buildpack-deps:{{ env.variant }}
{{ ) end -}}
```

図 11　ひな形に含まれる複雑な置換処理

単純な置換の例として，nginx コンテナ管理プロジェクトの生成スクリプトの例[*7] を図 10 に示す．図 6 に示したひな形内の変数（%%ALPINE_VERSION%% 等）を，生成スクリプト内の可変情報データ（$alpinever 等）に sed 等の置換命令を用いて置換しながら Dockerfile を生成する．この方法は実装が容易であり，UNIX 系の命令を用いることから軽量な点が特徴である．

一方で複雑な置換では，awk 等のスクリプト言語を用いて，if 分岐等の制御文を駆使して置換をおこなう．この方法では，ひな形ファイルにスクリプト言語が埋め込まれており，単一ファイルに 2 種類の言語が記述されている．例として，Python プロジェクトでのひな形ファイルの例[*8] を図 11 に示す．図 11 の 2 行目や 4 行目は Dockerfile の構文で書かれている．しかし，1 行目や 3 行目はスクリプト言語である awk の制御文が書かれている．つまり，1 つのひな形ファイルに FROM で始まる Dockerfile の構造を持つ文と，スクリプト言語による制御文が混在している．生成スクリプトで制御文を実行させることで，生成対象の Dockerfile には 3 つの FROM 文の内 1 つが記述されることになる．この方法は sed 等による単純な置換に比べ，複雑な制御が可能となる．しかしながら，ひな形に制御文が混じることで，可読性は低下するといえる．

次に 3 種の生成スクリプト実現パターンの 2 つ目であるメソッド型について説明する．メソッド型は，生成スクリプト内で Dockerfile の生成メソッドを宣言し，そのメソッドの組み合わせによって Dockerfile を生成する方法である．ibmjava プロジェクトでの生成スクリプトの例[*9] を図 12 に示す．図 12 では，FROM 文や，RUN を用いたパッケージマネージャ関連の文等を，Dockerfile の構造で分けて，それぞれの生成メソッドを定義する．そして，可変情報に合わせたメソッドの使い分けや，メ

[*7] https://github.com/nginxinc/docker-nginx/blob/d039609e3a537df4e15a454fdb5a004d519e9a11/update.sh

[*8] https://github.com/docker-library/python/blob/master/Dockerfile-linux.template

[*9] https://github.com/ibmruntimes/ci.docker/blob/master/ibmjava/update.sh

```
print_ubuntu_os() {
  cat >> $1 <<-EOI
  FROM ubuntu:18.04
  EOI
}
..
print_ubuntu_pkg() {
  cat >> $1 <<'EOI'
RUN apt-get update \
    && apt-get install.. wget ca-certificates \
    && rm -rf /var/lib/apt/lists/*
EOI
}
```

図 12　メソッド型を利用した生成スクリプト

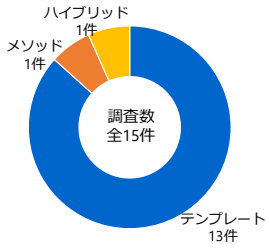

図 13　各型の採用数とその割合

ソッド内での変数処理を用いて，Dockerfile を生成する．ひな形となるファイルが独立していないことから，テンプレート型と比較して生成ファイルの全体像の把握は難しい．一方で，各メソッドがプロジェクトに依存していないため，高い移植性を持つと考える．

ハイブリッド型は，テンプレート型とメソッド型を組み合わせた方法である．Dockerfile をメソッドに分けて生成する方法は残しつつ，各メソッドで Dockerfile に書き込む情報をひな形ファイルとして管理している．したがって，Dockerfile の構造で分けられた複数のひな形ファイルと 1 つの生成スクリプトで構成されている．テンプレート型では，可変情報の違いによる複雑な処理を行うために生成スクリプトとは異なる言語を導入する必要があった．しかし，ハイブリッド型にすることで，生成スクリプトと同じ言語で複雑な実装が可能になる．しかしながら，メソッド型の持つ可読性の低さが目立つことから，万能な方法とは言い難い．

以上の 3 種の Dockerfile 実現パターンが存在した．3 種類の型の採用数を調べた結果を図 13 に示す．テンプレート型が 15 件中 13 件とテンプレート型の採用割合が圧倒的に多かった．テンプレート型が多い理由や，メソッド型やハイブリッド型のプロジェクトがテンプレート型を採用しない理由はまだ分かっておらず，それらの調査は今後の課題である．

DPP での Dockerfile 生成の実現方法は 3 つの型に大別され，その中でもテンプレート型の採用割合が圧倒的に多い．

5.4 RQ4: DPP における可変情報はどう管理されるか?

RQ4 では，可変情報の管理ファイルを対象に調査を実施した．その結果を，可変情報の種類，記述言語，生成スクリプトとの切り分けの有無の 3 つの観点から述べる．

5.4.1 可変情報の種類

調査の結果，以下に示す 3 種類の可変情報の存在を確認した．

- ベース OS
- サービスのバージョン
- ホストマシンのアーキテクチャ

まず 1 つ目はコンテナのベースとなる OS である．複数のベース OS の例としては Alpine，CentOS，Buster や Bullseye 等の Debian 系，Ubuntu，Windows 等が挙げられる．調査した 15 件中 10 件がベース OS を可変情報として扱っていた．残りの 5 件は 1 つのベース OS のみでの Dockerfile 提供を行っていた．

2 つ目は，コンテナの提供するサービスのバージョンである．例えば，Python の実行環境を提供するコンテナであれば，3.10 や 3.9 等によって複数の Dockerfile を作成する．調査した 15 件中 12 件がサービスのバージョンを可変情報として扱っていた．残りの 3 件は最新バージョンのみの Dockerfile を提供していた．

3 つ目は，ホストマシンのアーキテクチャである．Docker Hub で"arch"として扱われる情報であり，AMD364 や i386 等が該当する．アーキテクチャを可変情報として扱うプロジェクトは，15 件中 3 件と少なかった．アーキテクチャの可変情報としての処理は Docker Compose や Docker Buildx が標準でサポートしていることもあり，DPP での処理は少ないと考える．

5.4.2 記述言語

可変情報の処理を記述する言語を調査した結果，プロジェクトによって違いが見られた．可変情報の処理を記述する言語の種類を調査したグラフを図 14 に示す．一番多い言語は Bash であり調査した 15 件中 7 件が採用していた．生成スクリプトの記述言語も Bash が多かったことから，同言語で実装しやすい点が採用理由として挙げられる．

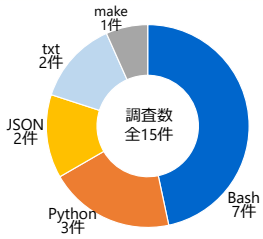

図 14　可変情報管理の言語別採用数とその割合

5.4.3　生成スクリプトとの切り分け

複数ファイルで分けているプロジェクトは 4 件存在．残り 11 件は生成スクリプト内で可変情報の管理も行っていた．

単一ファイルで処理する場合，生成スクリプト内で可変情報を変数として宣言して処理している．また，生成スクリプトと同じ言語で記述される．図 7 で示した変数宣言の例は，生成スクリプトの中に含まれており，生成スクリプトと同じ言語で記述されていた．

複数ファイルで処理する場合は実際の可変情報（alpine と buster や，1.1 と 1.2 等）を具体的に記述したデータファイルが別に存在し，生成スクリプト内でデータファイルを読み取る．別ファイルで分けられたデータの例として，Python イメージの管理プロジェクトでの可変情報ファイル[*10] を図 15 に示す．JSON 形式の深い構造に，バージョンの数字やベース OS 等の可変情報が保存される．

生成スクリプトと同じファイルで可変情報を処理する場合，実装は単純である．しかし，ベース OS やバージョンデータの更新と生成スクリプト自体の更新のどちらが更新されたか判別しづらい点があげられる．一方で複数ファイルに分けると，データの読み取りに多少実装の手間がかかるものの，更新箇所の判別は容易となる．

> 可変情報には，ベース OS，サービスのバージョン，ホストマシンのアーキテクチャの 3 種類が存在する．また，その管理は生成スクリプトと同一ファイル実現される場合もあれば，複数のファイルで実現される場合もある．

[*10] https://github.com/docker-library/python/blob/master/versions.json

```
{
  "3.10": {
    ..
    "variants": [
        "bullseye",
        "buster",
        "alpine",
        "windows/windowsservercore-ltsc2022"
      ],
      "version": "3.10.9"
  },
  "3.11": {
      ..
```

図 15　深い構造を持つ可変情報のデータファイル

6　妥当性の脅威

調査対象に関して考える．今回の調査では，Docker 公認コンテナ 32 件に対して DPP の採用割合を調査し，15 件を対象にファイルの詳細まで精査し，DPP の特徴を整理した．しかし，まだ未調査の 17 件や未収集の Docker コンテナに対しても調査を実施した場合，異なる結果が得られた可能性がある．また，今回の調査対象プロジェクトの選定理由は書けない．コンテナのダウンロード数上位 30 件等で調査対象を選定し再調査を実施した場合，異なる結果が得られた可能性がある．

各実現方法の利点欠点に関して考える．今回の調査で整理した各 DPP 実現パターンの利点や欠点は，各パターンの特徴から考えられる可能性として述べている．そのため，今回述べた利点や欠点が実際に存在するか否かは不明である．コミット履歴や ISSUE の調査，あるいは開発者へのインタビューを実施した場合，今回述べた利点や欠点が実際には存在しない，あるいは新たな点が存在する可能性がある．

プロジェクトの性質と合わせた分析について考える．Dockerfile 生成の実現方法は 3 種類存在したが，採用数としてはテンプレート型が圧倒的に多かった．ただ，テンプレート型の採用率が高い理由や，他の型を採用しているプロジェクトがその型を選ぶ理由は調査していない．また，そもそも DPP を採用していないプロジェクトも調査を実施していない．これらを調査することで，DPP を採用すべきプロジェクトの特徴やプロジェクトの性質に合わせた型の提案等，新たな知見が得られる可能性がある．

7　まとめと今後の課題

本研究では，Dockerfile のプリプロセッサ（DPP）の実現方法の体系化を目的として，GitHub リポジトリ上に存在する Dockerfile 管理プロジェクトの調査を実施した．まず，32 件のプロジェクトを対象に DPP の採用割合を調査した結果，87.5% が DPP を採用していた．次に，15 件のプロジェクトに対して DPP を実現するファイルを調査した結果，DPP はひな形，可変情報，生成スクリプトの 3 種類の要素で構成されることが分かった．そして，生成スクリプトの中身を精査した結果，Dockerfile 生成の実現方法にはテンプレート型，メソッド型，ハイブリッド型の 3 種類の型が存在した．最後に，可変情報の種類と管理方法を調査した結果，可変情報はベース OS，サービスのバージョン，ホストマシンのアーキテクチャの 3 種類が存在し，管理するファイルは生成スクリプトと同一の場合や，異なる場合が存在した．

今後の課題として，まず DPP 構成ファイルのファイル名の調査を検討している．調査時，DPP を構成するファイルのファイル名は，プロジェクトによって違いが見られた．しかし，よく使われるファイル名の存在も確認している．生成スクリプトのファイル名は update.sh が多かった．そこで，頻繁に使われるファイル名やディレクトリ名を調査する．これらの調査を実施することで，DPP を作成する際のファイル名の提案及び DPP 自動検出の実現に役立つと考える．

また，テスト方法の調査も検討している．調査した中には，生成した Dockerfile の妥当性をテストするためのツールが用意されているプロジェクトも存在した．テストを構成するファイル調査し，Dockerfile テスト方法の整理をしていく．さらに過去のソフトウェア工学におけるテストの考え方を取り入れた提案も可能と考える．

謝辞

本研究を行うにあたり，多くの皆様に多大なご支援をいただきました．

楠本真二教授．私の研究や発表に対し，助言をいただきました．NAIST の試験で私の研究に関する面接を控えている時期に，質疑応答の練習のお願いを快く受けてくださり，自身の考えを整理する良い機会となりました．

肥後芳樹教授．中間報告等の機会に，調査結果に対するアプローチや調査の進行に関する助言をいただきました．中には，調査中は出なかった別視点からの助言もあり，非常に参考になりました．

柗本真佑助教．私の指導教員として，研究の進め方や研究に対する姿勢，頭の使い方や整理の仕方，論文やプレゼン等の他者への伝え方を 1 から指導してくださりました．そして，研究に関して調査や考察を手伝ってくださりました．また，普段の休憩時間の雑談は，研究生活を楽しめた 1 つの要素でした．

事務補佐員の橋本 美砂子さん．私が快適に研究生活を送れるような環境づくりをしてくださりました．事務手続きに関して，私の返事が遅れることも多々ございましたが，丁寧に対応してくださりました．

研究室の皆様．初めての研究生活で，分からないことも多々ございましたが，多くの先輩方にサポートしていただきました．研究における注意点を予め伝えてくださり，論文や発表に関して様々な助言をいただきました．また，私が研究や進路に関して先行き不安な時は，励ましの言葉をかけてくださりました．そして，嬉しい出来事や面白い出来事は皆で共有し笑いあう環境であり，とても居心地がよく，楽しい研究生活を送ることができました．

私の友人．分からないことは相談しあう，休暇を楽しむ等，研究生活及び今までの学部生活の支えになってくれました．

私の両親．研究生活の 1 年間だけでなく，これまで私の活動を支援し，温かく見守ってくれました．とても感謝しています．

参考文献

[1] Portworx. Annual container adoption report, 2019. `https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf` (accessed 2023-01-31).

[2] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the docker container ecosystem on github. In *Proc. International Conference on Mining Software Repositories*, 2017.

[3] Mubin Ul Haque, Leonardo Horn Iwaya, and M. Ali Babar. Challenges in docker development: A large-scale study using stack overflow. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] D. Morris, S. Voutsinas, N.C. Hambly, and R.G. Mann. Use of docker for deployment and testing of astronomy software. *Astronomy and Computing*, Vol. 20, pp. 105–119, 2017.

[5] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, Vol. 49, No. 1, p. 71 – 79, jan 2015.

[6] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code - an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 45–55, 2015.

[7] Mohamed A. Oumaziz, Jean-Rémy Falleri, Xavier Blanc, Tegawendé F. Bissyandé, and Jacques Klein. Handling duplicates in dockerfiles families: Learning from experts. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 524–535, 2019.

[8] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. A dataset of dockerfiles. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, p. 528 – 532, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.*, Vol. 28, No. 12, p. 1146 – 1170, dec 2002.