ファイルの更新回数に基づいたファイルのコピー順の変更によるDockerfileのビルド時間の短縮

遠藤 睦実[1]　坂本 一俊[1]　串田 高幸[1]

概要：Dockerfile を用いたビルドは，前回ビルドと比較して変更のあるステップまでキャッシュを利用してビルドが行われる．ビルド時に一度のステップで複数のファイルをコピーする記述をすると，その中の 1 ファイルを編集しただけで該当ステップでコピーする全ファイルのキャッシュが破棄される．そのため，1 つのステップでコピーする際にファイルの更新がある場合，更新があるファイルのみステップを分けてコピーする場合と比較して，ビルド時間が長くなるという課題がある．この課題を解決するため，既にユーザによって記述された Dockerfile に対し，ファイルの更新回数に基づいてファイルのコピー順を修正するソフトウェアの作成を行った．一括でコピーを行う記述がされていた場合，1 回程度しか更新されていないファイルを先にコピーする修正を加えることで，キャッシュの使用率を向上させるのが目的である．本研究の提案に対し実験を行った結果，修正前からキャッシュの使用率は平均 4.13%向上した．

1. はじめに

背景

一般的に，Docker でイメージをビルドする際は，Dockerfile を記述してビルドする手法がとられる．Dockerfile を記述する際は，作成されるイメージのサイズの縮小とイメージのセキュリティの強度の 2 点について意識する必要がある．[*1][*2] Dockerfile を記述する際に意識すべき点を，Dockerfile のベストプラクティスとして公開されている．Dockerfile のベストプラクティスは公式のみならず企業，技術者，研究機関が出している [1][*3][*4][*5]．

ベストプラクティスの例として，更新頻度の低いファイルを先にコピーするという手法が存在する．このベストプラクティスを適用することで，前回ビルドから該当ファイルが更新されていない場合であれば，ビルド時間を短縮することができる．Dockerfile にはキャッシュ機能が実装されており，キャッシュ機能を利用してビルドされたステップは，ほとんど時間を要さずにビルドされるためである．Dockerfile を実行しビルドを行うと各ステップのビルド結果がキャッシュとして保存される．次回ビルドを行う際は前回ビルドと比較され，変更があるステップまではキャッシュを利用し，変更点のあるステップからビルドを開始する．キャッシュを破棄した以降のビルドは，前回との差分が無い場合であっても再びキャッシュを利用することは無く，新しくビルドを行う．

このキャッシュを利用するか破棄するかの判断は，前回のビルドと得られる結果とが異なるステップであるか否かで判断される．ほとんどの Dockerfile コマンドは，各ステップの前回ビルド結果である，子イメージとの単純な比較でキャッシュが利用可能か検証される[*6]．しかし ADD と COPY については，それに加えて，各ファイルのチェックサムを計算し，キャッシュのチェックサムとの比較が行われる．ADD もしくは COPY を行うステップでは，そのステップでコピーするファイル全てのチェックサムを照合し，変更がある場合はこのステップと以降のステップにおいてキャッシュは使用不可能になる．

課題

Dockerfile を利用してビルドを行う際の課題として，次のものが挙げられる．ファイルを 1 つのステップでまとめてコピーする処理を記述すると，そのうちのどれか 1 ファ

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1

[*1] Docker のベストプラクティス
https://maasaablog.com/development/docker/2948/

[*2] 社内の Dockerfile のベストプラクティスを公開します
https://www.forcia.com/blog/002273.html

[*3] Dockerfile リファレンス (Docker 公式)
https://docs.docker.jp/engine/reference/builder.html

[*4] Dockerfile のベストプラクティス Top20
https://sysdig.jp/blog/dockerfile-best-practices/

[*5] Intro Guide to Dockerfile Best Practices
https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/

[*6] Dockerfile を書くベスト・プラクティス (公式)
https://docs.docker.jp/engine/userguide/eng-image/dockerfile_best-practice.html

イルでも変更された場合，そのステップのキャッシュが利用できなくなり，変更の無いファイルをコピーする時間という無駄なビルド時間が発生することである．なぜなら，Dockerfile のキャッシュはステップ単位で管理されていることが原因である．この仕様によって，1 つのステップの中で，変更の無いファイルはキャッシュを利用し，変更されたファイルのみを新しくコピーすることは不可能である．更新の有無でステップを分けてコピーを行っていた場合であれば使用できたキャッシュが，一括でコピーを行った結果破棄されることになる．そのため，ファイルを 1 つのステップでまとめてコピーすると，ファイルを個別にコピーした場合と比較して，そのステップでコピーするファイルに何も更新が無い場合であればビルド時間が変わらないが，更新がある場合はビルド時間は低速化する．

基礎実験

実際にこの課題が存在するのかを検証するため，基礎実験を行った．基礎実験の環境として用意する node.js のプロジェクトと Dockerfile は，Node.js 公式サイトに記載されている，hello world を出力するアプリケーションを Docker 化するチュートリアル[*7]とした．

キャッシュの有無によるビルド時間の差を測るために次の実験を行った．まずは，ソースコード 1 の Dockerfile で 1 度ビルドを行う．これにより，この Dockerfile のキャッシュが生成される．

ソースコード 1 基礎実験の基本となる Dockerfile

```
1 FROM node:18
2 WORKDIR /usr/src/app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 8080
7 CMD [ "node", "server.js" ]
```

先に, キャッシュありのビルドとして何も編集せずにソースコード 1 の Dockerfile をビルドする．その後，キャッシュ無しのビルドとして，package.json を編集し保存した状態で，ソースコード 1 の Dockerfile をビルドする．この 2 つのビルドのビルド時間に差が生まれるかを比較した．実験結果として各ステップのビルド時間と全体のビルド時間を下記に載せる．表 1 がキャッシュあり，表 2 がキャッシュ無しである． キャッシュありの実験では，キャッシュを使用したステップは 0.0s でビルドされ，キャッシュ無しの実験では，キャッシュを利用していないステップは時間をかけてビルドしていることがわかった．

次に，1 つのステップで全てコピーする場合と，ステップを分割してコピーする場合で，同じファイルが更新された際にビルド時間の差が発生するかを確認する実験を行った．準備として，以下の一括でコピーするソースコード 2 の Dockerfile と分割でコピーするソースコード 3 の Dockerfile の 2 つを用意する．

[*7] Node.js Web アプリケーションを Docker 化する
https://nodejs.org/ja/docs/guides/nodejs-docker-webapp/

表 1 キャッシュを利用したビルド

```
Building 1.5s (10/10) FINISHED
CACHED[2/5] WORKDIR /usr/src/app  0.0s
CACHED[3/5] COPY package*.json ./ 0.0s
CACHED[4/5] RUN npm install       0.0s
CACHED[5/5] COPY . .              0.0s
```

表 2 キャッシュを破棄したビルド

```
Building 1.5s (10/10) FINISHED
CACHED[2/5] WORKDIR /usr/src/app  0.0s
[3/5] COPY package*.json ./       0.2s
[4/5] RUN npm install             4.0s
[5/5] COPY . .                    3.0s
```

ソースコード 2 全てのファイルを 1 度のステップでコピーする Dockerfile

```
1 FROM node:18
2 WORKDIR /usr/src/app
3 COPY . .
4 RUN npm install
5 EXPOSE 8080
6 CMD [ "node", "server.js" ]
```

ソースコード 3 package*.json を別途先にコピーする Dockerfile

```
1 FROM node:18
2 WORKDIR /usr/src/app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 8080
7 CMD [ "node", "server.js" ]
```

”COPY . .”は Dockerfile が存在するディレクトリとその下のファイル全てをコピーするコマンドである．このコマンドでは，それよりも前にコピーしたファイルをもう一度コピーすることは行われない．そのため，ソースコード 2 の”COPY . .”では package.json と package-lock.json はコピーされるが，ソースコード 3 ではコピーされない．

ソースコード 2 とソースコード 3 共に Dockerfile のビルドを行って Dockerfile のキャッシュを作成する．その後，server.js を変更した場合に，package.json と package-lock.json を先にコピーしたことで，ソースコード 2 とソースコード 3 の Dockerfile のビルド時間にどれだけ差が生まれるかを実験した．実験結果として各ステップのビルド時間と全体のビルド時間を表 3 と表 4 に載せる．表 3 が

1 つのステップで全てコピーする Dockerfile で，表 4 がステップを分割してコピーする Dockerfile となる． 表 4 では

表 3 1 つのステップで全てコピーする Dockerfile

```
Building 6.0s (10/10) FINISHED
CACHED[2/4] WORKDIR /usr/src/app  0.0s
[3/4] COPY . .                    2.6s
[4/4] RUN npm install             2.2s
```

表 4 ステップを分割してコピーする Dockerfile

```
Building 1.5s (10/10) FINISHED
CACHED[2/5] WORKDIR /usr/src/app  0.0s
CACHED[3/5] COPY package*.json ./ 0.0s
CACHED[4/5] RUN npm install       0.0s
[5/5] COPY . .                    0.5s
```

package.json と package-lock.json をキャッシュを利用してビルドを行っている．ビルドの際に，既にコピーされたファイルが後のステップでもう 1 度コピーされることはない．そのため，表 4 ステップ 5 で行われるその他のファイルのコピーも，同じコマンドを実行する表 3 ステップ 3 と比較すると，後者の分割コピーを行う Dockerfile の方がステップに要する時間が短くなっている．また，”RUN npm install”のステップで必要なファイルである package.json と package-lock.json のキャッシュが破棄される場合，同時に”RUN npm install”のキャッシュが利用できなくなる．実験でも，表 4 ステップ 4 ではキャッシュを利用して RUN できているが，表 3 ステップ 4 ではキャッシュを利用できなくなり，ビルド時間が増加している．

各章の概要

本テクニカルレポートは以下のように構成される．第 1 章では，本稿の背景と課題について述べる．第 2 章では，本稿の関連研究について述べる．第 3 章では，本稿での課題を解決するための提案手法について述べる．第 4 章では，提案した手法の実装について述べる．第 5 章では，提案手法に対しての実験内容と，その評価について述べる．第 6 章では，提案手法の議論を述べる．第 7 章は，本稿のまとめである．

2. 関連研究

Foyzul Hassan らは，Dockerfile とソースコードから文字列変数を抜き出し，依存関係を解析することで，正しい Dockerfile の更新内容を生成するソフトウェアを提案した [2]．Dockerfile のアップデートの自動化について検証し，既存の Dockerfile に対して，78.5%は正しいアップデートを推奨することに成功している．しかし，あくまでも依存関係のアップデートに着目した研究であるため，Dockerfile の更新は行われるが，Dockerfile の最適化までは行われていない．

Jordan Henkel らは，外部環境の変更によって破損し，現行環境では使用できない状態になった Dockerfile を静的に分析し，問題を修正するソフトウェアを提案した [3]．破損した Dockerfile をクラスター化し，蓄積されたデータベースを元に Dockerfile の修正を行う．実験では，ファイルの 73.25%で問題を検出している．しかし，このソフトウェアは一部人の手によって作業する箇所があるために自動化が十分になされておらず，自動修復を提供可能なのはそのうちの 18.9%である．

Yujing Wang らは，Docker イメージのビルドを行う際に，変更のあったレイヤーのみをバイパスし，レイヤー全体の再構築を避けるコードインジェクション手法を提案した [4]．インタープリター言語を使用したイメージの構築では 2 倍から 5 倍の性能が向上した．しかし，複雑なプロジェクトでは効果が薄い，コードインジェクションを使用するため，インタープリター言語を使用していないイメージには使用できないという問題がある．

3. 提案

提案方式

本研究では，図 1 に示す通り，既に作成された Dockerfile に対して，1 つのステップで複数のファイルをコピーする行を，2 つのステップに分割することで，ビルド時間を短縮することを試みる．

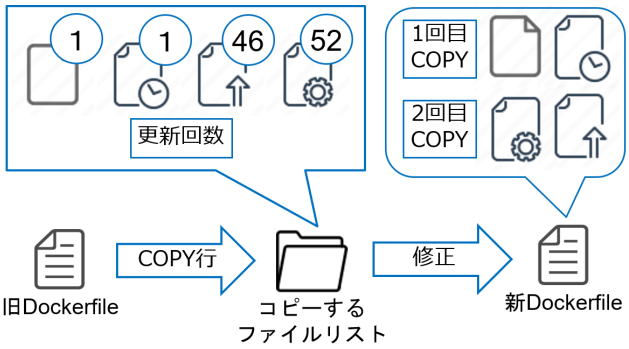

図 1 更新回数でステップごとにコピーするファイルを分ける

ファイル分割の基準にはファイルの更新回数を用いる．更新回数は，Github のコミット履歴から，全コミット中にファイルが登場した回数の累計を更新回数とする．更新回数の少ないファイルはビルドの前半にコピーを行うように既存の Dockerfile を修正する．低頻度の更新回数に分類する回数は 1 回以下，コピーを行う位置は最新の WORKDIR 直後とする．ここで，更新回数が 1 回であれば，そのファイルは Github に送ってからは一度も更新されていないファイルである．Github に該当ファイルを追加する操作で 1 回更新として扱われ，その後追加で修正がされていないた

めにそれ以上更新回数が増加していないからである．

まず，Dockerfile で記述されている全ての COPY コマンドを抽出し，COPY コマンドの引数からコンテナにコピーする全ファイルのリストを作成する．次に，ファイルリストとファイルの更新回数を照らし合わせ，更新回数が 1 回のファイルと 2 回以上の 2 グループに分割する．その後，分割したファイルを COPY する行を，最新の WORKDIR 直後に追加する．WORKDIR コマンドによってコンテナのカレントディレクトリを指定されていない状態では意図したディレクトリにファイルをコピーできないことと，更新される可能性の低いステップほどビルドの早い段階に配置したいことが理由である．

基礎実験では，ベストプラクティス通りに，更新されていないファイルを先にコピーして，更新されたファイルを後でコピーすることで，ビルド時間を短縮することができた．そのため，1 つのステップで一気にファイルをコピーする Dockerfile に対して，普段は更新されないようなファイルをまず先にコピーする Dockerfile に修正することができれば，Dockerfile のビルド時間を短縮することができると考えた．

その普段は更新されないファイルを判断するための基準として，ファイルの更新回数に基づくことにする．何回までを低頻度として分類するかを決定するために，Github 上にある Node.js と Dockerfile を利用したプロジェクトのコミット履歴からファイルの更新回数を調査した．調査対象は Node.js，docker-swarm-visualizer から 31，the-example-app.nodejs から 61，SneakerBot から 42，tribeca から 85，NodeGoat から 24 ファイルとした．結果から，平均で約 1/3 のファイルの更新回数が 1 回であることがわかった．更新回数が 2 回以上のファイルを 1 回のみのファイルに混ぜた場合，数年に数回程度のアップデートに更新する必要のないファイルが巻き込まれ，ビルド時間の無駄が発生する可能性がある．これらの要素を考慮した結果，更新回数が 1 回のファイルを先にコピーすることとした．

更新頻度を更新回数に基づいて決定する場合に発生する問題として，更新回数の増え方はプロジェクトの進行度によって大きく変化するというものが存在する．本提案では，数年以上経っても 1 度しか更新されていないファイル，つまり Github 上にファイルが保管されてから一度も修正されていないファイルが突然更新される可能性は低いという前提で成り立つ．突如更新回数が急激した場合に備え，低頻度ファイルリストに入れるファイルは更新回数が 1 回以下であるとすることで，ファイルが更新されていた場合は低頻度ファイルのリストから除外する形になり，課題が再発する事は避けられる．

問題として，Dockerfile 自身もキャッシュを持つため，Dockerfile を修正すると，修正したステップと以後のステップキャッシュが破棄されてしまうことが挙げられる．図 2 に示す例では，Code ではなく Code2 をコピーするように変更が加えられた結果，Code2 と Data のキャッシュは破棄されることになる．そのため，Dockerfile を実行するた

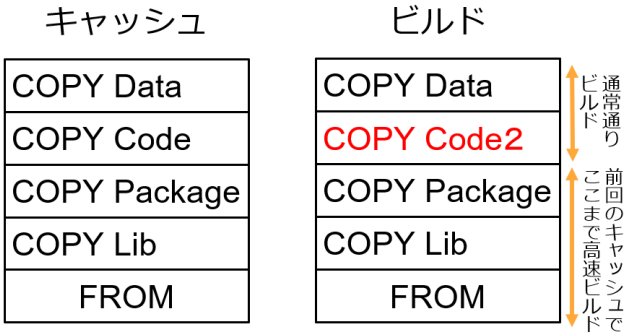

図 2 Dockerfile の変更によるキャッシュの破棄

びに修正してしまうと Dockerfile 自体のキャッシュを利用できなくなる．そのため，Dockerfile に修正を加えるタイミングは必要最小限にするべきである．この問題を解決するために，本ソフトウェアは，結果として先にコピーすべきファイルのリストが前回のビルドから変化している場合のみ，修正を行うようにした．これは，先にコピーするステップとそれ以降のキャッシュが破棄されているため，それと同時に Dockerfile の COPY コマンドを変更して同じくキャッシュを破棄してもビルド時間には影響が出ないからである．

ユースケース・シナリオ

本研究のユースケースシナリオは，Github 上で Node.js を利用した ECWeb サイト開発とする．開発の際は Dockerfile を利用してビルドを行うが，ほとんどのファイルのコピーは一括で行っており，複数人が同じリポジトリのファイルを修正しつつ各自がビルドを繰り返す状況を想定する．

Github では各々の判断でファイルを修正するため，各個人の視点からではどのファイルが更新頻度が低いのか正しく判断を行うのが困難である．その上，ファイル数が増加するほど全体の様相を把握する労力は増加していく．そのため，個人が継続的に人の手で更新されにくいファイルを調べ上げ，Dockerfile の早い段階のステップに移動させる労力は大きく，この行動によりビルド時間を短縮させるよりも，このベストプラクティスを無視してプロジェクトを進める方がプロジェクトの完成が早い可能性も考えられる．この問題に対して，全体のコミット履歴に基づいてファイルのコピー順を変更することで，更新されにくいファイルを先にコピーしてビルド時間の短縮を図る．

ソフトウェアが GitAPI から各ファイルの更新回数を取得し，Dockerfile を修正するのであれば，人がほとんど労力をかけずにビルド時間の短縮を図ることができる．これにより，プロジェクトの関係者や協力者が，変更が加えら

れたことによる変化を実際に確認するまでの時間が短縮されて、スムーズにプロジェクトを進めることの難易度を下げることができる．

実際に Github 上にある Docker を利用した Node.js プロジェクトの Dockerfile を調査した所，ほとんどの Dockerfile が Node.js と Dockerfile を併用する場合のベストプラクティス通りに package.json を先にコピーしているものの，そのほかのファイルは”COPY . .”として一括でコピーしていた．

4. 実装

実装

本提案方式を実現するために，以下の実装を行った．実装は Python を用いて行った．以下図 3 はプログラムのフローチャート図である．

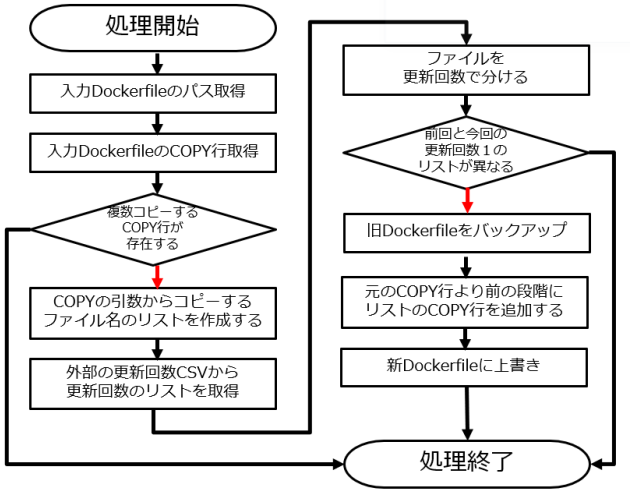

図 3 提案方式を実現するプログラムのフローチャート図

ユーザに対象とする Dockerfile のパスを入力してもらうと，その Dockerfile がビルドを行う際にコピーするファイルのリストを作成する．外部に記録された更新回数の CSV ファイルと比較して，コピーするファイルのリストから更新回数が 1 回以下のファイルのリストを新たに，前回のビルドまでの更新回数と今回のビルドまでの更新回数に基づく 2 つに分けて作成する．作成した 2 つのリストに差異がある場合は，今回のビルドまでの更新回数 1 回のファイルのリストを COPY するコマンドを WORKDIR 直後に追記する．

修正の対象となる Dockerfile から COPY 行を抜きだしてコピーするファイルのリストを作成する際に，ユーザが意図的に他のファイルとは分けてコピーする COPY 行である場合は，その行を変更または移動させると Dockerfile が動かなくなる可能性があるため，これらのファイルには触れない．しかし，同じ分けてコピーするファイルであっても，本プログラムによって追加された COPY 行であれば，元はユーザが後で一括でコピーしても問題無いと判断しているものであるため，変更，移動，削除を行っても Dockerfile のビルドが不可能になることは無い．そのため，この 2 つの個別コピーを区別するために，本プログラムが何を個別にコピーしたのかを外部に記録する．COPY 行からファイルリストを作成する際に，外部記録からコマンドを読み出し，該当するファイルは一括のコピーとして扱う．また，コピーするファイルのリストを作成する際に，.dockerignore に記述されているファイルは除外する．

本プログラムに使用する，該当リポジトリの各ファイルの更新回数が記録された CSV ファイルを作成するためのプログラムも別途作成した．実装は Python を用いて行った．このプログラムは GitAPI を使用して取得したいリポジトリのコミット履歴を順に辿り，そのコミットで変更されたファイル名が登場した回数を，日にちごとに集計するものである．CSV ファイルは縦が日付であり，横がファイル名とする．図 4 にフローチャート図を示す．

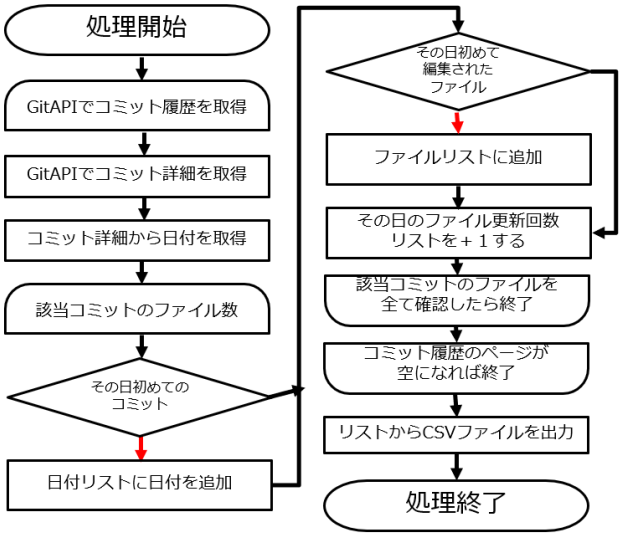

図 4 GitAPI を利用し，リポジトリ内ファイルの更新回数を取得

”https://api.github.com/repos/所有者名/リポジトリ名/commits?page=ページ数”にアクセスすることで，そのリポジトリのコミット履歴を GitAPI から取得することができる．プログラムからアクセスするには，Python の json ライブラリを使用する．コミット履歴を一度に表示できる数は 30 が上限であるため，31 回以上のコミット履歴を追うには，都度 url のページ指定を切り替える必要がある．更新回数を調べるには，更新日時とファイル名の 2 つを取得する必要があるが，全体のコミット履歴からではこの 2 つの情報は分からない．情報を取得するためには，ここから更にコミット履歴の url リンクから該当コミットの詳細情報にアクセスし，commit リストから date を，files リストからすべての filename を閲覧する．

このプログラムは多くのコミット履歴を閲覧するため，GitAPI は同一 IP からのアクセスに時間あたりの上限が定められていることに留意する．ユーザ認証を通し，アクセスするたびに数秒プログラムを待機する必要がある．

5. 実験と分析

評価実験として，以下の実験を行った．リポジトリのコミット初日から順に，更新の無い日を除いた更新履歴の 1 日分の回数の更新をファイルにかけながら，実装したプログラムによる分類を行った．実験対象としたリポジトリは，Github 上に存在する node.js と Dockerfile を利用したサンプルアプリである the-example-app.nodejs[*8]を利用した．この提案手法によって更新回数が 1 回以下のファイルを先にコピーされた場合に，キャッシュを使用できたファイルの数を評価する．

実験環境

実験は, 図 5 のように, Github から入手した the-example-app.nodejs リポジトリが入っているディレクトリと，そのリポジトリの更新回数が記録されている CSV ファイルと実装したプログラムが入っているディレクトリの 2 つをローカル環境に配置して行った．

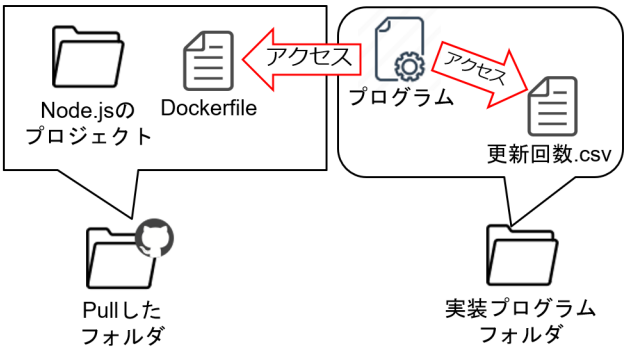

図 5 ファイル配置とアクセス

実験結果と分析

実験した結果のグラフを図 6 に示す．

the-example-app.nodejs では，提案手法適用前の状態ではキャッシュを使用できない日のうち，提案手法によってキャッシュを使用する事が出来た日数の割合は 45.45%，提案手法によりキャッシュを使用可能になった日のキャッシュ使用率平均は 9.09%であった．

キャッシュを使用できている日付のうち，ほとんどの場合において大きくキャッシュの使用率に変化は無いが，37 日目から 40 日目にかけてキャッシュ使用率が落ちていることが実験から明らかとなった．また，開発初期段階では

[*8] the-example-app.nodejs
https://github.com/contentful/the-example-app.nodejs

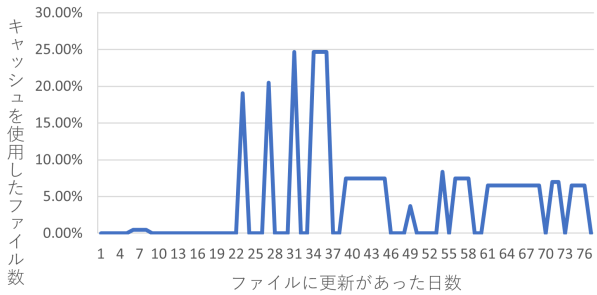

図 6 時間軸に沿ったキャッシュの使用率

ファイルの追加が多く，キャッシュを使用できない状態が続いた．このことから，無期限で更新回数 1 回のファイルを分類するのではなく，ある一定期間経つたびに閾値を更新した方が，キャッシュの使用率が向上する可能性が考えられる．

6. 議論

本提案では，更新回数が 1 回のみのファイルを，その他のファイルよりも早い段階でコピーすることによって，更新されていないファイルのキャッシュ使用率を向上させることができた．しかし，更新回数がリポジトリ全体の数%を占めるようなファイルの存在も確認されている．そのため，新しく更新回数が高頻度のファイルグループを設け，コピーのタイミングを変えることで，更新頻度が中頻度であるファイルのキャッシュの利用率を上げる余地がまだ残されている．今後はこの手法を更新回数の多いファイルにも適用した場合の評価を行う必要がある．

また今後は，実験結果である日を境にキャッシュ使用率が低くなったことや，実際にはプロジェクトの開発中に開発されるコンポーネントは遷移していくことを考慮する必要がある．本提案のように全期間中で更新回数が 1 回とするのではなく，ある一定の期間遡って更新回数が 0 回のファイルを分類することが考えられる．更新回数を一定期間で分けて見ることで，更新頻度の急上昇からの他ファイルのキャッシュ保護，または，更新が一時的に終了したファイルのキャッシュを再活性化することが可能になるかを検証する必要がある．

7. おわりに

Dockerfile の COPY コマンドは，前回のビルドの該当ステップと少しでも相違点があるとキャッシュを全て廃棄する仕様である．この課題を，本稿では更新回数でファイルをグループ分けし COPY を行う順番を変えるという手法で，前回ビルドから更新の無いファイルのキャッシュ利用率がどれだけ向上するかを実験した．評価として，全てのファイルを一括でコピーする場合と提案手法を適用して分割してコピーした場合のキャッシュ利用率を比較した．結

果, 一括でコピーする場合と比べ, 提案手法では平均 4.13% のファイルがキャッシュを利用してビルドを行うことができた.

参考文献

[1] Nüst, D., Sochat, V., Marwick, B., Eglen, S. J., Head, T., Hirst, T. and Evans, B. D.: Ten simple rules for writing Dockerfiles for reproducible data science, *PLOS Computational Biology*, Vol. 16, No. 11, pp. 1–24 (online), DOI: 10.1371/journal.pcbi.1008316 (2020).
[2] Hassan, F., Rodriguez, R. and Wang, X.: RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis, p. 796–801 (online), available from ⟨https://doi.org/10.1145/3238147.3240470⟩ (2018).
[3] Henkel, J., Silva, D., Teixeira, L., Marcelod' Amorim, Reps, T.: Shipwright: A Human-in-the-Loop System for Dockerfile Repair, pp. 1148–1160 (2021).
[4] Wang, Y. and Bao, Q.: A Code Injection Method for Rapid Docker Image Building, (online), DOI: 10.48550/ARXIV.1911.07444 (2019).