Dockerfileにおけるファイル更新回数を利用したCOPY処理の分割によるイメージビルド時間の短縮

遠藤 睦実[1]　高橋 風太[2]　串田 高幸[1]

概要：Docker のビルドはレイヤーの積み重ねによって行われる．レイヤーは RUN と COPY と CMD によって作成される．課題として，一部ファイルの編集により該当の COPY を行うレイヤーがキャッシュできなくなる分イメージのビルド時間が伸びる．これは，ファイルのチェックサムがファイル単位ではなくレイヤー単位で管理している事が原因である．この課題を解決するため，ユーザの用意した Dockerfile に対し，一度にコピーされているファイル群をファイルの更新日時と更新回数でグループ分けし，3 分割してコピーすることで解決を試みる．評価として，提案適用前後でキャッシュを利用してコピーできたファイルの割合とビルドに要した時間の差を比較する．基礎実験として，キャッシュの使用の有無によるビルド時間の変化を調べた．基礎実験は github の doge-unblocker リポジトリを対象に行った．実験結果は，一括コピーで 4.9 秒，編集したファイルを後でコピーすると 3.5 秒であった．編集したファイルのみを隔離して後でコピーすることで，そうしない場合の約 71%のビルド時間となった．

1. はじめに

背景

Docker でイメージのビルドを行う場合，Dockerfile を記述する必要がある [1,2]．Dockerfile を用いてビルドするためには，Dockefile にベースイメージ，コンテナ内部へコピーしたいファイル，コンテナ内へのインストール手順，環境変数やコンテナの挙動を記述する必要がある．実際に Dockerfile を運用する際に，前述した 4 つの要素をどういった順番で配置し，オプションをどう指定するかによって，出力されるイメージのサイズとセキュリティ強度が左右されるという事情がある．そのため Dockerfile を記述する際には，単に出力されるコンテナを動作させるために必要な情報を羅列するだけではなく，イメージのサイズ増加やセキュリティホールの発生を招かないような書き方を心掛ける必要がある [3,4]．こういった Dockerfile の書き方についての経験則をベストプラクティスという名前でエンジニア間で共有されている [5,6] [*1]．

その Dockerfile のベストプラクティスの 1 つに，イメージのレイヤー順を，更新頻度に基づいて可能な範囲で並び替える，というものがある [7]．これは，Docker のビルドにはキャッシュ機能があるためである．初回のビルドでは無い場合，途中まで前回の Dockerfile と同一であるコマンド内容であれば，前回のコマンド実行結果をキャッシュとして利用し，その時点までのコマンド内容を飛ばして処理することができる．

イメージは差分の積み重ねでレイヤーを構築する仕様のため，キャッシュはイメージのレイヤー単位で使用される．差分の積み重ねという特性上，手前のレイヤーで変更が加えられた場合は以後のレイヤーは未変更であっても出力結果が変化する場合がある．そのため，キャッシュを使用できるのは変更の無いレイヤーまでとなり，変更されたレイヤー以後に変更点は無い場合でも，キャッシュは使用されずに再度レイヤー構築が行われる．

キャッシュのうちファイルコピー処理のみは例外で，コピー対象のファイルごとにチェックサムを行うのではなく，該当コマンドでコピーするファイル群に対してまとめてチェックサムをとる．そして，チェックサムの結果とキャッシュに保存されたチェックサムの両者が一致する場合のみキャッシュを使用する．

この仕様により，ファイルのキャッシュはファイル単位ではなく，レイヤー単位で管理されることになる．そのため，ファイルコピーを行う際にコピー対象であるファイル群の一部を編集した場合，チェックサムの結果が合わなくなり，編集していないその他のファイルごとファイル群の

[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
[2] 東京工科大学大学院バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

[*1] Dockerfile リファレンス 20.10(Docker 公式)
https://docs.docker.jp/engine/reference/builder.html

キャッシュ全てが破棄される．

課題

Dockerfile を利用したビルドの仕様の 1 つに，キャッシュシステムがイメージのレイヤー単位で管理されているため，コピーするファイルのキャッシュはファイルごとではなくコマンドごとに管理されていることが挙げられる．その仕様により，本来キャッシュを使用できたファイルを再度コピーするという，スキップすることが可能な処理を実行するため，ビルドに要する時間が長くなるという課題が存在している．

この課題は，複数のファイルを 1 コマンドでコピーするだけで発生する．これは，一度にコピーするファイルの総数が常に 1 つ以下である場合を除く全てのビルドシナリオにおいて，本来はキャッシュを使用可能であるにもかかわらず，キャッシュを破棄されてしまうファイルが出現する可能性があることを意味する．

図 1 は，package.json を使用して yarn でインストールを行う Node コンテナを例とした，js ファイルの編集によって他ファイルのキャッシュが破棄される条件をあらわしたものである．背景が暗くなっている部分がキャッシュが破棄されるレイヤーである．図 1 左のように，3 つのファイルを同時にコピーした場合，そのレイヤーのチェックサムが合わないことでキャッシュが破棄されて，他 2 つのファイルも再度コピーされる．図 1 右のように，js ファイルを他のファイルよりも後にコピーすることで，他 2 ファイルのチェックサムは合致し，キャッシュを使用することができる．

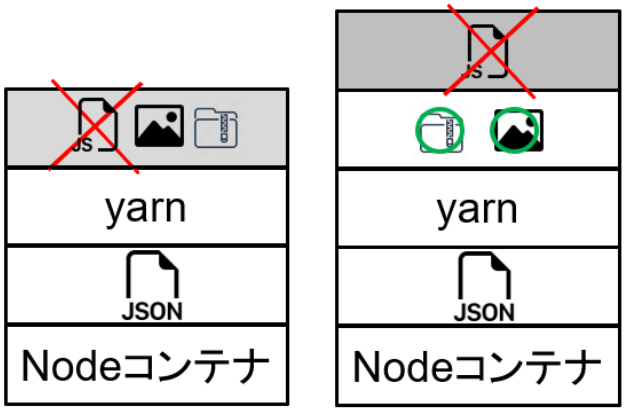

図 1 Build 時におけるファイルキャッシュの問題点

以上の挙動から，ファイルを纏めてコピーした場合，本来キャッシュを使用してコピーすることが出来たファイルを再度コピーし直す手間の分，ビルドに要する時間が長くなるという課題が存在している．

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 2 章では，本稿の関連研究について述べる．第 3 章では，本稿で挙げた課題を解決するための提案について述べる．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

Shipwright という，主に外部環境の変化によってビルドに失敗する状態になった Dockerfile を，キーワードと解決方法をクラスタリングすることで問題を修正するソフトウェアを提案した研究がある [8]．この研究は，ビルドに失敗する Dockerfile を修復する事が目的である．ビルドに失敗する Dockefile の問題分析及び修復により，Dockerfile の再現性を保証しやすくなる．しかしこの研究では，ビルドに成功する Dockefile のビルド高速化は行わない．

RUDSEA という，ソースコードから環境関係のコードを抽出し，Dockerfile の正しい更新内容を生成するソフトウェアを提案した研究がある [9]．この研究は，プロジェクトの更新中に外部環境が更新された際に，Dockefile も自動的に更新後の環境に合わせることが目的である．依存関係の更新及び Dockefile 自身の自動更新によりユーザの負担を減らすことができる．しかしこの研究では，ビルドに成功する Dockefile のビルド高速化は行わない．

インタープリタ言語を扱うイメージにおいて，変更のあるレイヤーのみをバイパスすることで変更の無いレイヤーのリビルドを回避し，キャッシュを使用できるようにする手法を提案する研究がある [10]．この研究は，手前のレイヤーキャッシュが破棄されると以後のレイヤーキャッシュが使用できない問題を回避するのが目的である．インタープリタ言語に限れば，手前のレイヤーキャッシュが破棄されると以後のレイヤーキャッシュが使用できない問題を回避して，ビルド時間を短縮することができる．しかしこの研究では，インタープリタ言語以外を扱うユースケースは対象にできないという問題がある．

DockerMock という，Dockerfile の命令とシェルコマンドの一部をダミーに置き換えることで，Dockerfile のビルド障害をコンテキストベースで検出するソフトウェアを提案した研究がある [11]．この研究は検出精度と再現率を高く保ちながら障害を素早く検知することが目的である．既存研究と比較してより少ないビルド回数でビルド障害となった原因を特定することで，ビルドに失敗する際の修復までの速度が上昇する．しかしこの研究では，ビルドに問題無く成功するが，無駄が多くビルド時間が伸びている場合の対処までは行わない．

3. 提案

提案方式

本稿の課題は，更新されたファイルによって更新のないファイルのキャッシュが破棄されることで発生する．その場合，図 1 左のように 1 回で纏めてコピーせずに，図 1 右のようにまず先に更新されないファイルのキャッシュから使用することで，課題の状況が発生する事を回避できる．

本稿では課題に対して，ユーザの書いた Dockerfile のディレクトリ丸ごとを 1 回で COPY する処理を対象に，更新回数・日時を判断基準として，複数コマンドにわたるコピーに分割した Dockerfile を新しく作成するソフトウェアを提案する．

本提案方式では，ユーザが用意したファイル群に対して，ファイルの更新によって他ファイルのキャッシュを破棄されないように順序良く分類する必要がある．そのための分類基準として，ファイルの更新日時，ファイルの更新回数の 2 つを用いる．この 2 つを用いる理由は，更新頻度が単位時間あたりに何回編集されたかで求められるためである．更新の頻度を求めるために更新回数を，更新回数の増加傾向を時系列で追うために更新日時を使用する．

図 2 に示す通り，ファイルのグループ分けは大きく 3 つに分ける．更新回数が 1 以下のグループ，更新回数上位からファイルをグループに入れる処理をグループが占める更新回数が全体の 20%を超えるまで行ったグループ，どちらにも該当しないグループである．更新回数が 1 以下のグループのみは累計の更新回数を用い，それ以外は直近 30 日の更新回数を用いる．図 2 左のファイルに付随している数字が直近 30 日の更新回数を表す．この例ではファイル A の累計更新回数も 1 とする．

図 2 の例ではまず，更新回数が 1 回という条件を満たすのはファイル A であるため，ファイル A を低頻度にグループする．次に，最も更新回数が多いファイル D から順に，高頻度グループの更新回数が全ファイルの合計更新回数の 20%を超えるまで高頻度グループに分類する．今回の例ではファイル D を追加した時点で高頻度グループは全体の約 80%の更新回数を占めたため，ファイル D を高頻度に分類する．最後に，低頻度と高頻度どちらにも分類されなかったファイル B，C を中頻度に分類する．

図 2 のようにグループ分けを行った後，図 3 に示すようにファイルコピーを 3 段階に分けることにより，編集の無いファイルのキャッシュを使用することができる．図 3 の左の状態では，ファイル D の編集によりファイル A〜C のキャッシュが破棄されている状態である，ここで提案を適用し，図中央のようにファイルをグループ分けする．その後，図右のように頻度の低いグループから順にコピーすることにより，ファイル A とファイル B，C のキャッシュを

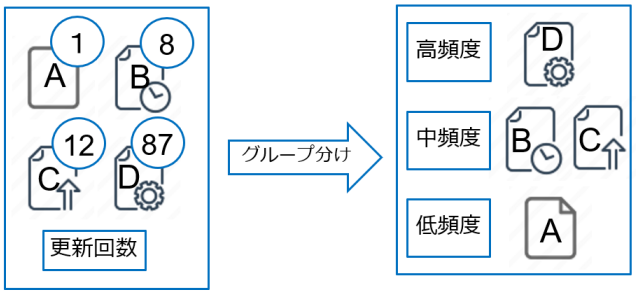

図 2 更新回数に基づくファイルのコピータイミング分割

使用することができるようになる．

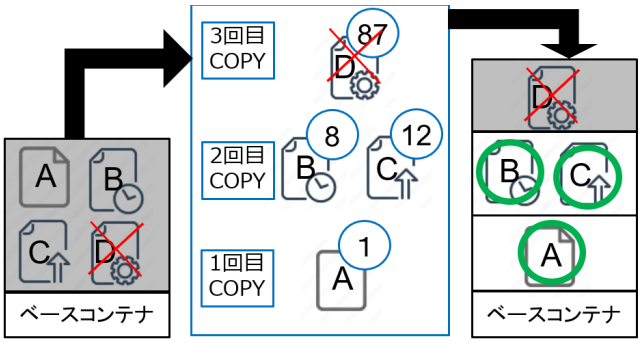

図 3 更新回数に基づくファイルのコピータイミング分割

低頻度グループにファイルを分類する際に，2 回もしくは数回程度の更新回数であるファイルは本稿では分類しないこととする．その理由として，更新回数 1 回のみのファイルを纏めてコピーするケースが最もキャッシュを使用できる確率が高いからである．更新回数が 1 回のみという事は，編集はファイルを用意した時点での 1 回きりで，それ以降は放置されている事を意味する．これが 2 回以上となると最初に用意した状態から編集する必要が出てきた事を意味する．現時点では数回程度でもこれから頻繁に更新し始め，高頻度ファイルに化ける可能性が無いとは言い切れない．もし数回程度のファイルを低頻度に分類していて，かつ低頻度に分類していたファイルが突然高頻度で更新を始めた場合，更新回数が増加して中頻度もしくは高頻度に分類され始めるまでは低頻度に分類されてしまうため，低頻度グループのキャッシュが破棄され続ける事態になる．そのため，更新回数 1 回のみの分類とすることで，キャッシュし続けられるファイルは安定してキャッシュさせる．

COPY 処理の分割という本提案に対して，キャッシュとレイヤーの仕様に注意すべき点が存在する．キャッシュは Dockerfile に記述された命令が前回と一致するかどうかで使用の判断をする仕様である．そのため，既に書かれた命令自体を変更するだけで該当処理のキャッシュは使用できなくなる．提案方式を適用すると COPY 行に変更が入るため，適用直後はファイルキャッシュを使用できず，提案の恩恵を受けるのは次回以降となる問題がある．この問題を避けるため，本提案では以下の対策を行う．キャッシュに

対しては，ビルドごとに提案を適用するのではなく，ビルド時にファイルキャッシュが使用できない場合に適用することで対処する，これは，提案における低頻度グループの中で編集されたファイルが出現した場合，提案で Dockefile に変更を加える箇所よりも前のレイヤーキャッシュが破棄された場合が該当する．イメージのレイヤー上限は 125 までという仕様のため，1COPY につき 1 ファイルのみコピーするといったレイヤー数を無駄に増やす手法は取れない．そのため，本提案のようにファイルは更新される可能性ごとにグループ分けを行い，レイヤーの増加を抑える必要がある．

ユースケース・シナリオ

Python で実装したプログラムを Docker コンテナ内で動かす際に本提案が使用される事を想定している．プロジェクトは Github 上で 1 つリポジトリを作成して管理する．

Dockerfile の中身は，requirements.txt をコピーした後に requirements.txt に記述したライブラリをインストールし，その後ソースコード類をコピーする流れとする．

本提案は，イメージのビルド時に数 MB 以上のファイルをローカルからコンテナへコピーしたい場合を想定している．本提案はキャッシュを使った COPY 処理の時間を短縮するものであるため，ローカルからイメージへコピーするファイルの合計サイズが多ければ多いほど効果的である．

以上を踏まえたユースケースを図 4 に示す．プログラムの内容は Python で開発した Web アプリケーションとし，Web アプリケーションの画面を表示するのにコンテナ内に用意した画像も用いるものとする．ユーザはローカルで開発・用意したファイルを Github でバージョン管理を行う．編集内容を確定したファイルは必ずリポジトリにコミットされ，コミットしていないファイルはローカルにおいても編集されていないものとし，リポジトリとローカルの間で全ファイルの更新内容に齟齬が無い状態を前提とする．ビルドはプログラム 1 に示す Dockerfile を利用して，ローカルで行う．まず先に Requirements.txt に記述したライブラリ群を pip を用いてインストールを行い，その後ソースコードと画像ファイルをイメージへコピーする．ユースケースに画像ファイルのイメージへのコピーを含めた理由は，画像は一度用意した後に更新されることは稀であり，かつファイルサイズが大きくなりやすいという性質を持つからである．Github 上で画像を扱うリポジトリ 3 件を調査した所，画像ファイルはプッシュされて後で編集されるケースが少なく，かつファイルサイズも合計数 MB から数十 MB であった，本提案によってビルド時に先に画像ファイルがコピーされることで，コピーに時間のかかるファイルをキャッシュし，ビルド時間を短縮することができる．

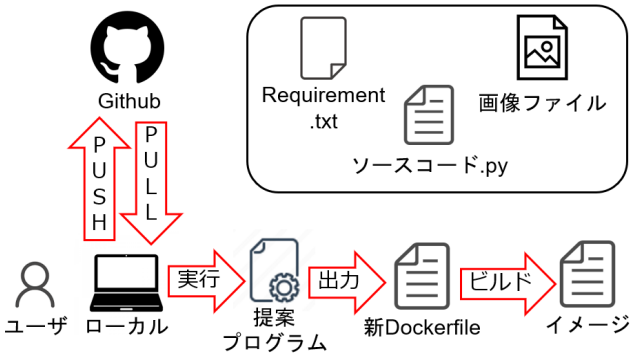

図 4　GitAPI へのアクセスにおける流れ

プログラム 1　requirements.txt を用いて Python 環境を構築する例

```
1 FROM python:3.11.2-slim-bullseye
2 COPY requirements.txt /app/requirements.txt
3 RUN pip install -r /app/requirements.txt
4 COPY . /app
5 WORKDIR /app
6 CMD ["python", "-u", "main.py"]
```

4.　実装

Dockerfile の COPY 処理分割

本提案を実現するために，以下の実装を行った．実装方法については，Python を用いてプログラムを作成した．今回のファイルの更新日時とファイルの更新回数の取得方法については，Github 上のリポジトリが保有するコミット履歴から，該当ファイルのコミット日時の更新日時，該当ファイルの History 数を更新回数とみなし，GitAPI を用いることで取得する．ユーザがイメージへとコピーしたいファイルをプログラムで把握する方法については，Dockerfile 自体はテキストベースである事を活かし，先頭が COPY である行の引数を Python で取得する．Dockerfile で記述されている全ての COPY コマンドを抽出した後に，COPY コマンドの引数からコンテナにコピーするファイルのリストを作成する．

以下がプログラムの流れである．

① 対象にする Dockerfile のパスをユーザから受け取る
② Dockerfile から COPY 行を抽出
③ ファイル単体 COPY のみの処理であれば中断する
④ 抽出した COPY 行と Dockerfile の置かれたディレクトリから，実際にコピーするファイルのリストを作成する
⑤ ファイルの更新日時と更新回数を記した CSV ファイルをロードする．
⑥ ファイルの更新日時と更新回数に基づきファイルをグループ分けする
⑦ 旧 Dockerfile をバックアップ
⑧ ファイルグループごとに COPY 処理を生成する

⑨ 生成した COPY 処理を Dockerfile に追記する

また，図 5 に本実装で行われる入力及び出力を表す．

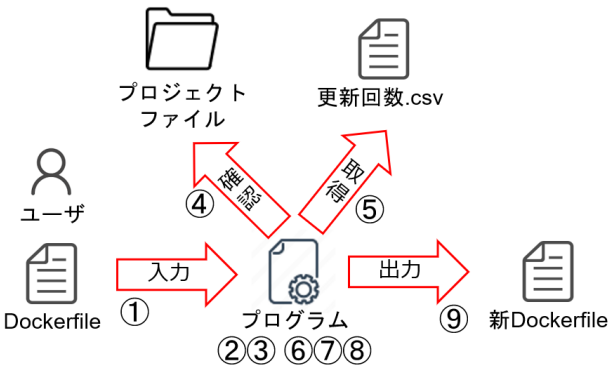

図 5 実装におけるアクセスの流れ

ソフトウェアに対するユーザの入力は，ユーザが提案を適用する対象である，Dockerfile のパスである．入力を元に対象の Dockerfile が置いてあるディレクトリを特定し，イメージのビルドを行う際にコピーする全ファイルをまとめたリストを作成する．名前のみを記録すると，別ディレクトリの同名ファイルが存在した場合は競合するため，ファイル名ではなく Dockerfile が配置してあるディレクトリをルートディレクトリとしたファイルパスを記録する．各ファイルの更新回数を記録した CSV ファイルとプログラム中のファイルを照らし合わせ，ファイルリストから更新回数が 1 回のファイルを抜き出したリストを作成する．これを，前回のビルドまでの更新回数を使ったリストと今回のビルドを含む更新回数を使ったリストの 2 つに分けて作成する．前回までのリストに名前があり今回を含むリストに名前が無いファイルが存在する場合は，更新回数が 1 回のファイルを COPY する処理のキャッシュが使用できない事を意味する．本提案では更新回数が 1 回のファイル群は分割した COPY 処理の中で最も早いタイミングでコピーするため，このケースでは分割したファイル全てのキャッシュが使用不可となる．これは，提案を適用するタイミングである”全ての COPY 処理においてチェックサムが合わなくなり，ビルド時にファイルキャッシュが使用できない場合”に該当するため，提案手法を実際に適用する．この場合は，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 するコマンドを最後に記述された 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，主に前提環境のインストールに使用するなどの理由ユーザが意図的に他のファイルとは分け，直接引数に該当ファイルを記述してコピーする COPY 行が存在する場合がある．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，これらのファイルは本提案の対象から除外する．しかしそれが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記録する．COPY 行からファイルリストを作成する際に，本ソフトウェアが変更もしくは追加したコマンドを記録から読み出し，該当するファイルは一括のコピーとして扱う．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．

GitAPI を用いた更新回数 CSV ファイルの作成

本提案では，各ファイルの更新回数が記録された CSV ファイルを用意する必要がある．そのための，GitAPI を用いて該当リポジトリのコミット履歴を取得する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最新の方から順に辿り，そのコミットで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．手順は以下の通りである．

以下がプログラムの流れである．

(Ⅰ) GitAPI でコミット履歴を取得
(Ⅱ)コミット履歴からコミット詳細を 1 つ取得
(Ⅲ)コミット詳細から日付とファイル名を取得
(Ⅳ)日付・ファイル名共に初出ならば，それぞれリストに追加
(Ⅴ)その日付でファイル名が登場した回数をカウント
(Ⅵ)2〜5 の手順をコミット履歴の数だけ繰り返す
(Ⅶ)最後に日付リストを見出し行，ファイル名リストを見出し列にし，その日のファイル更新回数を記入した CSV ファイルを出力する

また，図 6 にプログラムで行われる入力及び出力を表す．

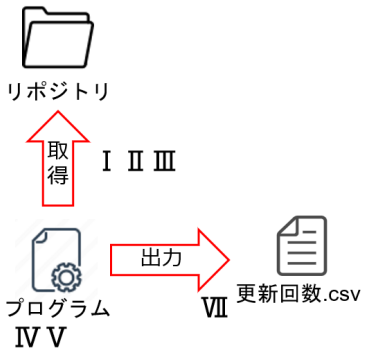

図 6 GitAPI へのアクセスにおける流れ

ブラウザからアクセスする場合，”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．GitAPI で一度に取得できるコミット履歴の数には限りがあり，通常は 30 である．デフォルトでは 1 ページ目，つまり最新の 30 のみ表示されるため，”page=ページ数”を引数として渡し，2 ページ目以降を取得できるようにする．

プログラムからアクセスするために，今回は Python の json ライブラリを使用した．url からリスト化されたテキストとしてコミット情報を取得することができるためである．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 回目以降のコミット履歴を追うには，前述のように都度 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．しかし，全体のコミット履歴からではこの 2 つの情報は表示されていない．そのため 2 つの情報を取得するためには，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する必要がある．詳細情報の，commit リストの date が日付，files リストの filename がファイル名である．

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．このプログラムは多くのコミット履歴を閲覧するため，ユーザ認証を通した上で，GitAPI のアクセス後はアクセス上限に引っ掛からない最短の待機時間である 7 秒プログラムを待機する必要がある．

5. 実験

基礎実験として，キャッシュの有無によるビルド時間の差と，COPY の分割によるキャッシュの活用でビルド時間が短縮されるのかについて実験を行った．

実験環境

実験対象とするリポジトリは，github から doge-unblocker[*2]を clone して用意する．このリポジトリを使用した理由は，コンテナに複数の画像を用意する必要があるため，ビルド時にローカルからイメージにコピーするファイルの合計サイズが約 10MB と他の Github のリポジトリと比較して大きいためである．本提案はキャッシュを使った COPY 処理の時間を短縮するものであるため，ローカルからイメージへコピーするファイルの合計サイズが多ければ多いほど効果的である．そのため，ビルド時にファイルをキャッシュすることによるビルド時間の短縮幅が大きいため，キャッシュの有無によるビルド時間の差を見る実験に適している．

ビルドに使用する Dockerfile は，このリポジトリに同梱されているプログラム 2 を基本とする．

[*2] https://github.com/dogenetwork/doge-unblocker

プログラム 2 実験で使用する Dockerfile

```
1 # syntax=docker/dockerfile:1
2 FROM node:19-bullseye
3 ENV NODE_ENV=production
4 WORKDIR /app
5 COPY ["package.json", "package-lock.json*", "./"]
6 RUN npm install
7 EXPOSE 8080
8 COPY . .
9 CMD [ "npm", "start" ]
```

実験結果と分析

表 1 に，”–no-cache”オプションを使用し，キャッシュを使用しなかった場合のビルド時間を掲載する．

表 1 キャッシュを使用しないビルド

キャッシュ	ステップ	コマンド内容	時間 (秒)
CACHED	[2/5]	WORKDIR /usr/src/app	0.0s
	[3/5]	COPY [package.json,	
		package-lock.json*, ./]	0.2s
	[4/5]	RUN npm install	8.3s
	[5/5]	COPY . .	1.6s
		Building FINISHED	計 13.1s

表 2 に，表 1 の結果をキャッシュとして使用するが，index.js を編集していた場合のビルド時間を掲載する．

表 2 COPY ./ ./でコピーする index.js を編集した場合

キャッシュ	ステップ	コマンド内容	時間 (秒)
CACHED	[2/5]	WORKDIR /usr/src/app	0.0s
CACHED	[3/5]	COPY [package.json,	
		package-lock.json*, ./]	0.0s
CACHED	[4/5]	RUN npm install	0.0s
	[5/5]	COPY . .	1.6s
		Building FINISHED	計 4.9s

表 3 に，表 1 の結果をキャッシュとして使用するが，index.js を編集し，更に index.js のみその他のファイルよりも遅いタイミングでコピーした場合のビルド時間を掲載する．表 3 Stage6 の COPY 処理は index.js を除いた全てのファイルを引数に指定している．しかし，index.js を除きながら”. .”のように一括指定することはできない．そのため全てのファイル名を列挙することになり表が非常に長くなるので，ここでは”. .”と表記している．

図 7 は，表 1〜3 の出力結果から時間をグラフにしたものである．青い棒がビルド時間全体に要した秒数であり，オレンジの棒がビルド時間のうち COPY 処理に要した時間を抜き出した秒数である．

編集のあったファイルのみを後からコピーすることで，実際にファイルをキャッシュできた分だけビルド時間が短縮できることを確認できた．

表 3　index.js を他のファイルよりも後にコピーした場合

キャッシュ	ステップ	コマンド内容	時間 (秒)
CACHED	[2/6]	WORKDIR /usr/src/app	0.0s
CACHED	[3/6]	COPY [package.json, package-lock.json*, ./]	0.0s
CACHED	[4/6]	RUN npm install	0.0s
CACHED	[5/6]	COPY (index.js 以外) .	0.0s
	[6/6]	COPY index.js .	0.2s
		Building FINISHED	計 3.5s

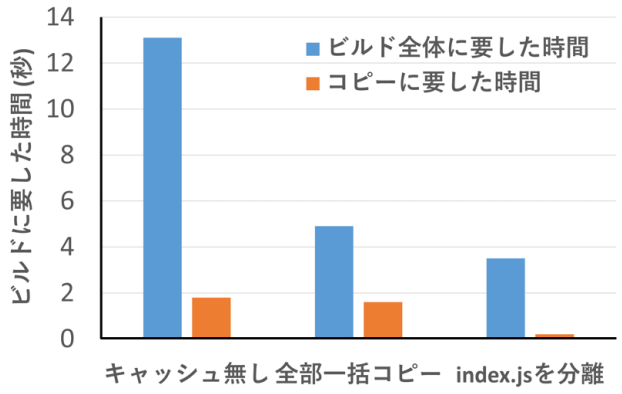

図 7　ビルド時に要した時間 (秒)

6. 議論

今回は提案におけるファイルのグループ分けにおいて，低頻度を更新回数 1 回以下，高頻度を更新回数全体の上位 20%とし，使用する更新回数の期間は直近 30 日とした，しかしこの数字は数件の Python 系リポジトリのコミット傾向から出したものである．そのため，提案の基準となる数字が最適では無い可能性や，サンプル数の少なさから判断を誤っている可能性，Python 以外にもこの基準を適用できるのかという問題がある．今後はより多くのリポジトリに対して実験を行い，最適な分類基準を探る必要がある．

今回プログラムはユーザが手動で実行することになっている．しかし実装したプログラムの実行タイミングをユーザ任せにした場合，ユーザがプログラムの実行を忘れる可能性がある．本提案は普段は更新されないファイルが編集されたビルド時に実行されるのが望ましい．そのため，提案の実行はユーザ任せにはせず，プログラムが普段は更新されないファイルが編集されたかどうかを監視し，編集された時にプログラムが実行される方が確実である．そのための方法として，Python の watchdog モジュールを使用して，対象となる Dockerfile が配置されているディレクトリ内のファイル更新を監視するやり方が考えられる．ファイル更新を検知した際は，更新回数が記録された CSV ファイルから同名のファイルが何回更新されているかを取得し，それが低頻度グループのファイルだった場合に提案を実行することで，提案の適用忘れを防ぐことができると考える．その場合ユーザからの入力は，監視してほしい Dockefile のパスを初回のみ入力してもらう形となる，

7. おわりに

課題は，ファイルをまとめてコピーする場合に，コピーするファイルを 1 つでも編集すると全体のキャッシュが破棄され，ビルドに要する時間が長くなることである．そこでファイルを更新回数の特徴に基づきグループ分けし，コピー処理を複数回に分けることを試みた．基礎実験として，キャッシュの使用の有無によるビルド時間の変化を調べた．実験結果は，一括コピーで 4.9 秒，編集したファイルを後でコピーすると 3.5 秒であった．編集したファイルのみを隔離して後でコピーすることで，そうしない場合の約 71%のビルド時間となった．

参考文献

[1] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S. and Gall, H. C.: An Empirical Analysis of the Docker Container Ecosystem on GitHub, pp. 323–333 (2017).

[2] Lu, Z., Xu, J., Wu, Y., Wang, T. and Huang, T.: An Empirical Case Study on the Temporary File Smell in Dockerfiles, *IEEE Access*, Vol. 7, pp. 63650–63659 (online), DOI: 10.1109/ACCESS.2019.2905424 (2019).

[3] Wu, Y., Zhang, Y., Wang, T. and Wang, H.: Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study, *IEEE Access*, Vol. 8, pp. 34127–34139 (2020).

[4] Xu, J., Wu, Y., Lu, Z. and Wang, T.: Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods, *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1, pp. 185–190 (online), DOI: 10.1109/COMPSAC.2019.00033 (2019).

[5] 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).

[6] Wu, Y., Zhang, Y., Wang, T. and Wang, H.: Dockerfile Changes in Practice: A Large-Scale Empirical Study of 4,110 Projects on GitHub, *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 247–256 (online), DOI: 10.1109/APSEC51365.2020.00033 (2020).

[7] Rosa, G., Scalabrino, S. and Oliveto, R.: Assessing and Improving the Quality of Docker Artifacts, *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 592–596 (online), DOI: 10.1109/ICSME55016.2022.00081 (2022).

[8] Henkel, J., Silva, D., Teixeira, L., Marcelod' Amorim, Reps, T.: Shipwright: A Human-in-the-Loop System for Dockerfile Repair, pp. 1148–1160 (2021).

[9] 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).

[10] Wang, Y. and Bao, Q.: A Code Injection Method for Rapid Docker Image Building, (online), DOI: 10.48550/ARXIV.1911.07444 (2019).

[11] Li, M., Bai, X., Ma, M. and Pei, D.: DockerMock: Pre-Build Detection of Dockerfile Faults through Mocking

Instruction Execution (2021).

