Docker を用いたデプロイ高速化方法の提案と評価

2014SE103 浮須 省　2014SE119 上原 賢二

指導教員：青山 幹雄

1 研究の背景と課題

1.1 研究背景

システム開発において，アプリケーションをユーザに迅速に提供することがビジネス価値向上に重要となっている．アプリケーションの新バージョンを本番環境へ反映させるデプロイ工程では，その規模増大の影響により時間がかり，開発環境と本番環境で差異があると異常が発生する場合がある．また，デプロイ工程では無線ネットワーク経由でソフトウェアをアップデートする手段として，SOTA(Software Updates Over-The-Air)が期待されている．

1.2 研究課題

本研究では，コンテナ型仮想化である Docker を用いて，リアルタイム性の要求が厳しい環境において，デプロイを迅速に行うために以下の研究課題を設定する．

(1) Docker イメージの Build 時間削減

チェーン機能とキャッシュ機能を組合わせ，Build 時間の削減する．

(2) 効果的な Dockerfile の記述方法

Docker イメージのサイズ縮小を図るために，効果的な Dockerfile の記述方法を提案する．さらに，チェーン機能を用いたイメージサイズの縮小を提案する．

(3) 例題を用いてデプロイ時間短縮の効果を評価する．

2 関連研究

2.1 コンテナ型仮想化と Docker

(1) 概要

コンテナ型仮想化は，OS 環境にコンテナと呼ばれる分離された空間を生成し，その空間ごとに OS 環境を構築できる仮想化方法である．コンテナはホスト OS から見ると一つのプロセスとして認識されており，オーバーヘッドが少ない．また，コンテナはホスト OS とカーネルを共有するため，各コンテナに OS は導入されていない．

コンテナ型仮想化を実現するプラットフォームとして，オープンソースで提供されている Docker がある．[2, 3, 8]

以下，図 1 に Docker のアーキテクチャを示す．

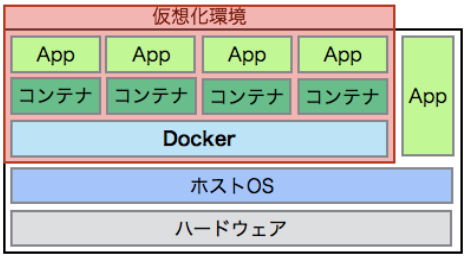

図 1 コンテナ型仮想化(Docker)

(2) Docker コンテナ

Docker コンテナは，Docker における個々の仮想環境を指す．OS のカーネル機能を用いて，複数のファイルシステムをひとまとめにしたものである．Docker イメージを指定して起動することで，仮想環境として生成される．

Docker コンテナの特徴として，異なる環境でも同様に動作可能であることが挙げられる．ホスト OS が Windows や Mac，Ubuntu，CentOS のどれでも同様に動作できる．

(3) Docker イメージ

Docker イメージは，Docker コンテナの実行に必要なファイルシステムのことである． Docker イメージは Read-Only であるため直接書き込むことはできず，コンテナを起動しイメージの上に差分として追加することになる．

Docker イメージの構成を Dockerfile として記述することで，開発環境を共有できる．Dockerfile から Docker イメージ，Docker コンテナへ遷移する手順は，アプローチで述べる．[4]

2.2 Alpine Linux

Alpine Linux は Linux ディストリビューションの一つである．組込システムに適した Linux ユーティリティが元になっており，軽量でベースイメージのサイズが小さいことが特徴である．Alpine Linux 独自のパッケージ管理システムである apk が Dockerfile の記法とマッチしていることから，Docker の環境として有用である．デフォルトのパッケージ数が少ないため，開発に必要となるパッケージをその都度追加することによって不要なパッケージを追加することなく Docker イメージを作成できる．[1, 6, 7]

3 アプローチ

本研究では，Docker イメージを軽量化することで，デプロイ時間の短縮を目的とする．

Docker イメージを用いたアプリケーションのデプロイは，Docker イメージを pull した後，デプロイする方法がある．しかし，本研究ではパッケージの追加などを開発者自身が操作することを考慮し，Dockerfile を記述し Docker イメージを Build する方法を用いてデプロイすることとする．

その Dockerfile にチェーン機能を適用させ効率的な Dockerfile を作成し，軽量化された Docker イメージとキャッシュ機能を用いてデプロイ時間の短縮を行う．

図 2 に，Docker を利用し新機能を追加したアプリケーションが本番環境で運用されるまでのプロセスを示す．

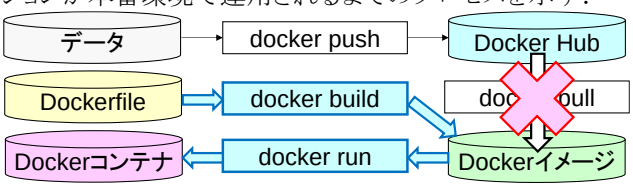

図 2 Docker コンテナ構築の流れ

3.1 提案方法

Docker コンテナを構築するためには Dockerfile と Docker イメージが必要となる. そこで, この 2 点に着目してアプリケーションのデプロイの時間短縮方法を提案する.

Dockerfile は同じ機能を果たすアプリケーションであっても, 記述方法を変更することや, キャッシュ機能を利用することで軽量化を図ることが可能になる. また, ベースイメージに Alpine Linux を利用することでイメージサイズを縮小する.

Docker イメージは差分管理と転送を行うことで軽量化を図る. Docker イメージの変更点のみを転送することで, Docker イメージ全体を転送することに比べ, 短時間で Docker イメージを構築することに繋がる.

4 Docker 導入によるデプロイ高速化

4.1 Docker イメージの差分管理と転送の方法

ファイルシステムの差分管理機能を利用して, 既に構築されている Docker イメージの上に書き込み専用の領域を重ねて構築していく. 各 Docker コンテナは書き込み専用の領域である差分だけを持つことになる. [5]

以下に Docker イメージの差分転送のプロセスを示す.

(1) 開発環境の Docker コンテナを構築している Docker イメージ全体を DockerHub に転送する.
(2) 本番環境の Docker コンテナに(1)で転送した Docker イメージ全体を取得して起動する.
(3) 開発環境で Docker イメージを変更した際は, 差分のみを DockerHub に転送する.
(4) 本番環境で(3)で転送した Docker イメージの差分を取得して起動する.

4.2 Dockerfile のキャッシュ機能

キャッシュ機能は, Dockerfile の 2 回目以降の Build 時, 前回までに変更されていない部分の Build 処理をスキップする機能である.

(1) Build コマンドを実行し, Build の結果を Docker 環境内に保存する.
(2) Dockerfile の全ての行において, 変更された行がある場合は, その行を新規で処理を実行する.
(3) Dockerfile の全ての行において, 変更された行がない場合は, 過去に保存した Build 結果を読み込み処理を実行する.

以下は, キャッシュ機能の適用判定の説明である. 図 3 にアクティビティ図を示す.

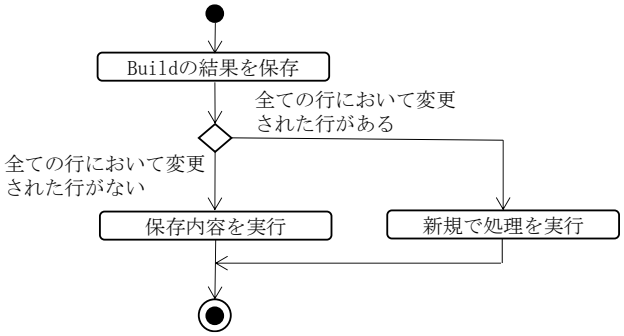

図 3 キャッシュ機能のプロセス

キャッシュ機能を用いて Docker イメージを Build した実行画面を図 4 に示す. Step2, 3, 4, 5, 6, 7 における Using cache と表示された行は, キャッシュ機能を使用して Docker イメージを構築している.

```
$ sudo docker build -t chain -f chain.dockerfile .
Sending build context to Docker daemon  5.12 kB
Step 1 : FROM alpine:latest
 ---> 76da55c8019d
Step 2 : MAINTAINER shokenji
 ---> Using cache
 ---> 829c63678e7c
Step 3 : ENV NGINX_VERSION 1.11.1
 ---> Using cache
 ---> 337877786fa3
Step 4 : RUN apk --update add…
 ---> Using cache
 ---> 76f5beb78f5e
Step 5 : VOLUME /var/cache/nginx
 ---> Using cache
 ---> 76f5beb78f5e
Step 6 : EXPOSE 80 443
 ---> Using cache
 ---> f6e29c834ab6
Step 7 : CMD nginx -g daemon off;
 ---> Using cache
 ---> 8b8de8691df4
Successfully built 8f197c1170be
```

図 4 キャッシュ機能を用いた実行画面

キャッシュ機能を用いることで Docker イメージの構築時間を大幅に短縮できる.

Dockerfile の変更がない行は処理を行わないため, Docker コンテナの構築作業を省略することになり, デプロイ時間を短縮することになる.

また, 変更行が少ない場合は効果が見込めない可能性があるが, 実際のシステムで変更行が多い場合は効果が大きいと予想され, Docker イメージの構築にかかる時間を大幅に短縮できる.

```
入手 →  FROM alpine:3.6                                  初回時
導入 →  MAINTAINER shokenj<sho.ukisu@apps.nise.org>
導入 →  ENV NGINX_VERSION 1.11.1
導入 →  COPY test1.html  /bar/www/html/
導入 →  RUN apk --update add pcre-dev openssl-dev…
実行 →  CMD ["nginx", "-g", "daemon off;"]

Skip →  FROM alpine:3.6                                  2回目以降
Skip →  MAINTAINER shokenji<sho.ukisu@apps.nise.org>
Skip →  ENV NGINX_VERSION 1.11.1
変更 →  COPY test2.html  /bar/www/html/  #この行を変更
導入 →  RUN apk --update add …  #以降キャッシュ不可
実行 →  CMD ["nginx", "-g", "daemon off;"]
```

図 5 キャッシュ機能

図 5 はキャッシュ機能の適用範囲を示した Dockerfile である.

(Step1) 初回時

(1) 1 行目は, Alpine Linux3.6 の Docker イメージを入手する.
(2) 2 行目は, Dockerfile の作成者の情報を入手する.
(3) 3 行目は, 環境変数を設定する.
(4) 4 行目は, Web コンテンツの test.html を Docker イメージの/var/www/html ディレクトリにコピーする.

(5) 5 行目は，FROM で指定したイメージ上で，既存のイメージ上の新しいレイヤで，あらゆるコマンドを実行し，その結果をコミットする．

(6) 6 行目は，指定したコマンドを実行する．

(Step2) 2 回目以降

(1) 1，2，3 行目は，変更が無いためキャッシュ機能により処理を行わない．

(2) 4 行目は，test.html ファイルに test2.html ファイルに変更し，/var/www/html ディレクトリにコピーする．5，6 行目は，4 行目で変更が加わりキャッシュ機能は利用されないため，再び初回時(5)，(6)を行う．

4.3 Dockerfile の記述方法

Docker イメージのサイズを小さくするために以下の記述方法を提案する．

(1) RUN でのコマンドをチェーンさせる

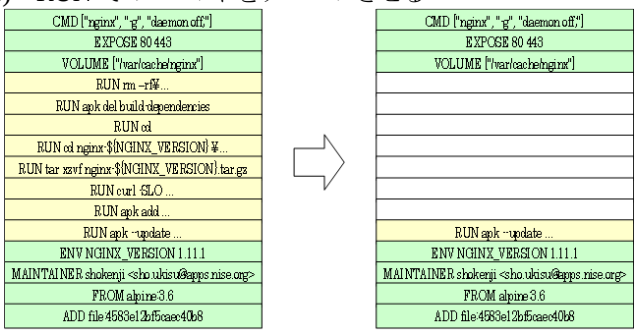

図 6 チェーン機能を用いたレイヤ数の減少

Docker イメージのサイズを縮小する方法の 1 つとして，Docker イメージのレイヤ数を削減する方法がある．Docker イメージは Dockerfile 内のコマンドごとにレイヤが生成され，Docker イメージはレイヤ構造となる．レイヤ自身も Docker イメージのため，図 6 に示すようにレイヤ数を減らすことで Docker イメージのサイズを縮小できる．

ここで，チェーン機能を適用させることで Docker イメージのレイヤ数を削減することができるため，デプロイ時間の短縮が可能になる．

```
RUN apk update
RUN apk upgrade
RUN apk add –update
```

```
RUN apk update ¥
    && apk upgrade ¥
    && apk add –update
```

図 7 チェーンの有無

コマンドの実行やパッケージのインストールなどで使用する場合は RUN コマンドを実行する．実際にユーザに提供されているアプリケーションでは実行しなければならないコマンドが多数存在すると考えられる．

そのような場合は，複数 RUN コマンド図 7 に示すように RUN を 1 つにまとめる．それによって，実行時間の短縮が可能になる．

(2) 変更が多い箇所はできるだけ最後に記述

Build コマンドはキャッシュ機能を利用してレイヤを作成する．ある一行でキャッシュが利用されない場合は，それ以降の全ての行でキャッシュが利用されない．そのため，アプリケーションの修正や改善のために Dockerfile を変更する際は，変更する可能性の多い行をできるだけ最後に記述し，キャッシュ機能を利用することで実行時間を短縮する．

4.4 Alpine Linux を用いた実現方法

Docker イメージは Dockerfile に記述されたベースイメージをもとに構築される．ベースイメージは Docker イメージのサイズに影響するため，セキュアで軽量な Linux ディストリビューションの Alpine Linux を適用する．

実現方法は以下の通りである．

ベースイメージに Alpine Linux を利用する．Alpine Linux サイズは CentOS と比較して 20 分の 1 となる．Docker イメージはベースイメージの影響を大きく受けるため，Docker イメージも縮小可能となる．

また，ベースイメージだけでなく，各アプリケーションにも Alpine Linux を用いる．

5 プロトタイプの実装

(1) 適用対象

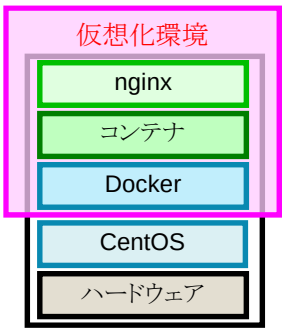

図 8 適用対象のアーキテクチャ

適用対象として図 8 に示すように nginx を用いることとする．nginx は近年の Web サーバにおいて広く用いられているが，限られたリソースの中で開発する必要がある環境にも適応できるよう Docker を用いて開発した．

本研究では，ベースイメージに Alpine Linux を指定した Dockerfile を記述することで nginx の作成をした．

(2) Build 時間測定

Build 時間測定は以下のように行った．

1) Dockerfile 全体にチェーン機能を適用させる．
2) add から tar までの Build 時間測定をする．
3) add から cd までの Build 時間測定をする．
4) add から del までの Build 時間測定をする．
5) add から rm までの Build 時間測定をする．
6) 3)-2)を計算したものを cd nginx から cd までの Build 時間とする．
7) 4)-3)を計算したものを del の Build 時間とする．
8) 5)-4)を計算したものを rm の Build 時間とする．

尚，ここでの各工程番号は図 9 と表 1 で用いた丸囲いの数字に対応する．

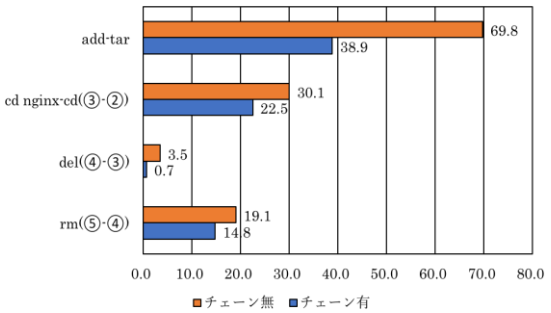

図 9 各操作の Build 時間

表 1 各操作の削減率

各操作	平均削減率[%]	標準偏差[s]
add-tar	44.2	2.7
cd nginx-cd(③-②)	25.0	5.7
del(④-③)	79.3	21.8
rm(⑤-④)	22.2	11.0

本研究のプロトタイプの Build 時間を表 2 に示す.

表 2 プロトタイプの Build 時間

機能		平均値[s]	標準偏差[s]	削減率[%]
チェーン	無	122.41	1.17	37.09
	有	77.00	1.01	
キャッシュ	無	77.00	1.01	99.90
	有	0.07	0.03	

6 評価

(1) チェーン機能の有効性

add-tar における削減率は大きな値となっておりおよそ半減させることができた. 削減率だけに注目した場合, del が最も大きな値となっている. しかし, 全体にチェーンを適用させた場合の Build 時間は 77.00s であるため, del に要した Build 時間は 0.72s に短縮されているが, 全体の Build 時間を考えれば, 大きな影響を与える値ではないことが分かる. そのため, 全体の Build 時間に影響を及ぼしている操作のうちでは, add-tar が最も大きな削減率を示すこととなった.

(2) キャッシュ機能の有効性

Dockerfile 全体にチェーン機能を適用させ, かつキャッシュ機能を適用させた場合, Build 時間は 0.071s を要した.

ただし, Dockerfile に多くの変更を加える可能性がある場合は不向きとなっている. そのため, 変更の可能性が低い部分に関しては, 別の Dockerfile に作成しておき, 事前に Build しておくと効率の良い Dockerfile の作成に繋がる.

もしくは, RUN コマンドの部分にオプションで—no-cache をつけることで, 意図的にキャッシュ機能を適用させないような行の作成をしていくことも一つの方法であると考える.

(3) 組合わせの有効性

チェーン機能とキャッシュ機能の組合わせにより, 最も効果的にデプロイをすることとなった. 単純にチェーン機能やキャッシュ機能を適用させるのではなく, それぞれの機能の特徴を考慮し, 工夫して適用させ Dockerfile を作成することでデプロイ時間の短縮をさせることが可能になった.

7 考察

全体の考察として, Build 時間の短縮を実現するにはチェーン機能の適用によるレイヤ数の削減が大きな割合を占めている. これは各レイヤを作成する際の時間が加味されているためであると考察する.

今回はテストとして nginx のインストール段階について研究を進めたが, 実際の開発環境に適用させた場合, 開発環境の構築にはより多くのパッケージをインストールする必要がある. その際にも効率的に Dockerfile の作成が行えるよう, 新たな知識を取り入れる学習コストがかかることが考察できる.

8 今後の課題

今後の課題を以下に示す.

(1) 差分管理機能の実現
(2) 他のアプリケーションにおける検証
(3) 並行動作可能条件

9 まとめ

本研究では, Docker を用いてデプロイ高速化を実現する方法の提案した.

各レイヤの Build 時間から, 開発環境を準備する箇所である add の項目に割合が集中していることが確認できた. チェーン機能を用いることで該当箇所の Build 時間を半減させることができた.

また, ベースイメージに Alpine Linux を用いることや, チェーン機能を用いてレイヤ数を削減することで Docker イメージ全体のサイズ縮小に繋がった.

そして, チェーン機能とキャッシュ機能の適用条件を明確にすることで, 有効な場合を示すことができ, 効率的な Dockefile の作成を実現することとなった.

10 参考文献

[1] Alpine Linux, https://alpinelinux.org/.
[2] 阿佐 志保, プログラマのための Docker 教科書, 翔泳社, 2015.
[3] 浅井 利文, 森 雅達, 山中 翔太, Docker コンテナを用いたマルチクラウド上の動的アーキテクチャの提案, 南山大学情報理工学部ソフトウェア工学科卒業論文, 2016.
[4] Docker, http://docs.docker.jp/v1.11/engine/understanding-docker.html#id26.
[5] 加藤 耕太, アプリケーションのデプロイへの Docker 導入の提言, 2015, http://www.kobelcosys.co.jp/news/pdf/ronbun_2015_2.pdf.
[6] A. Sakaguchi, Alpine Linux で Docker イメージを劇的に小さくする, https://qiita.com/asakaguchi/items/484ba262965ef3823f61.
[7] A. Yamada, Alpine Linux 入門, https://blog.stormcat.io/post/entry/alpine-entry-apk/.
[8] 吉岡 恒夫, Docker 実践活用ガイド, マイナビ出版, 2016.