

# Dockerfile の開発を支援する インタラクティブツールの提案

---

稲田 司

2023/02/17

鵜林・亀井研究室

# 目次

導入

背景と目的

本ツールの特長

簡単な使い方

まとめ

参考資料 (p.33～)

# 目次

## 導入

## 背景と目的

## 本ツールの特長

## 簡単な使い方

## まとめ

## 参考資料（p.33～）

# Docker とは何か？

ソフトウェアの実行に必要なものすべてをパッケージ化し、それらをプロセスレベルで分離された空間で実行する技術。

## ソフトウェア開発面でのメリット

- ハードや OS の違いを意識せず、開発に専念できる。
- 既存の成果物を活用できる。
- アプリケーションのデプロイ・スケーリングが容易。

# Dockerfile とは何か？

Docker において、

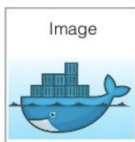
**イメージ** OS やアプリケーションのテンプレート

**コンテナ** イメージを元に生成されるアプリケーション環境

Dockerfile は、イメージ構築を自動化する一連の命令群が記載されたテキストファイル。

```
FROM ubuntu:14.04
MAINTAINER John Doe <john.doe@example.com>
RUN apt-get update && apt-get install -y python3
RUN python3 --help
CMD ["python3", "--help"]
```

build

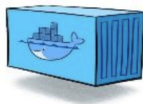


イメージ

run



(commit)



コンテナ

Dockerfile

# 目次

導入

背景と目的

本ツールの特長

簡単な使い方

まとめ

参考資料 (p.33～)

# イメージ・Dockerfile が抱える課題とその解決手法

ビルド時間の短縮

イメージサイズの削減

イメージ分析

サイバー攻撃への対策

同一性の確認

再現性の確保

イメージ開発の効率化

保守性の確保

# イメージ・Dockerfile が抱える課題とその解決手法

- |            |                             |
|------------|-----------------------------|
| ビルド時間の短縮   | → キャッシュの利用, BuildKit        |
| イメージサイズの削減 | → マルチステージビルド, Slim          |
| イメージ分析     | → dive, dlayer              |
| サイバー攻撃への対策 | → Distroless イメージ, BuildKit |
| 同一性の確認     | → ハッシュによる確認, イメージ署名         |
| 再現性の確保     | → ?                         |
| イメージ開発の効率化 | → ?                         |
| 保守性の確保     | → ?                         |



# イメージ・Dockerfile が抱える課題とその解決手法

ビルド時間の短縮	→ キャッシュの利用, BuildKit
イメージサイズの削減	→ マルチステージビルド, Slim
イメージ分析	→ dive, dlayer
サイバー攻撃への対策	→ Distroless イメージ, BuildKit
同一性の確認	→ ハッシュによる確認, イメージ署名
再現性の確保	→ ?
イメージ開発の効率化	→ ?
保守性の確保	→ ?

本ツールで解決したい

# イメージ・Dockerfile が抱える課題とその解決手法

- |            |                             |
|------------|-----------------------------|
| ビルド時間の短縮   | → キャッシュの利用, BuildKit        |
| イメージサイズの削減 | → マルチステージビルド, Slim          |
| イメージ分析     | → dive, dlayer              |
| サイバー攻撃への対策 | → Distroless イメージ, BuildKit |
| 同一性の確認     | → ハッシュによる確認, イメージ署名         |
| 再現性の確保     | → ?                         |
| イメージ開発の効率化 | → インタラクティブツール               |
| 保守性の確保     | → リファクタリングツール               |

# 目次

導入

背景と目的

本ツールの特長

簡単な使い方

まとめ

参考資料 (p.33～)

# なぜ、イメージ開発の効率化が必要なのか？

下のギャップが、開発を非効率にしている。

イメージ開発 Dockerfile を作成する必要がある。

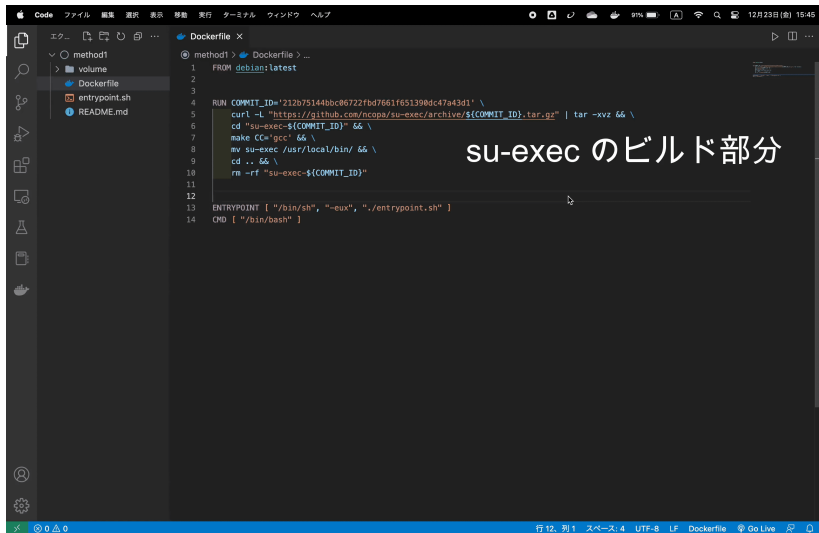
動作確認 コンテナ内で行う必要がある。

現状の動作確認の手法

1. 別で、同じベースイメージから起動したコンテナを使う。
2. 開発途中の Dockerfile から逐一生成するコンテナを使う。

どちらの手法も、問題あり。

# 別で、同じベースイメージから起動したコンテナで動作確認をしながら、**Dockerfile** を開発する例



```
1 FROM debian:latest
2
3
4 RUN COMMIT_ID='212b75144bbc06722fbd7661f651390dc47a43d1' \
5     curl -L "https://github.com/ncopa/su-exec/archive/${COMMIT_ID}.tar.gz" | tar -xvz && \
6     cd "su-exec-${COMMIT_ID}" && \
7     make CC='gcc' && \
8     mv su-exec /usr/local/bin/ && \
9     cd .. && \
10    rm -rf "su-exec-${COMMIT_ID}"
11
12
13 ENTRYPOINT [ "/bin/sh", "-eux", "./entrypoint.sh" ]
14 CMD [ "/bin/bash" ]
```

su-exec のビルド部分

行 12, 列 1 スペース: 4 UTF-8 LF Dockerfile Go Live

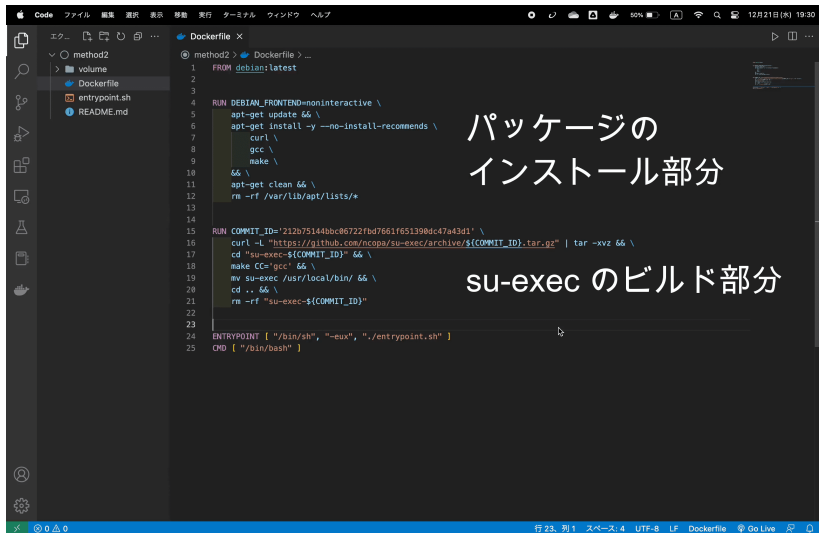
# 別で、同じベースイメージから起動したコンテナで動作確認をしながら、**Dockerfile** を開発する例

```
1 FROM debian:latest
2
3
4 RUN COMMIT_ID='212b75144bbc06722fbd7661f651390dc47a43d1' \
5     curl -L "https://github.com/ncopa/su-exec/archive/${COMMIT_ID}.tar.gz" | tar -xvz && \
6     cd "su-exec-${COMMIT_ID}" && \
7     make CC='gcc' && \
8     mv su-exec /usr/local/bin/ && \
9     cd .. && \
10    rm -rf "su-exec-${COMMIT_ID}"
11
12 ENTRYPOINT [ "/bin/sh", "-eux", "./entrypoint.sh" ]
13
14 CMD [ "/bin/bash" ]
```

su-exec のビルド部分

気にかけることが多い

# 開発途中の Dockerfile から逐一生成するコンテナで動作確認をしながら，Dockerfile を開発する例



The screenshot shows a VS Code editor with a Dockerfile open. The file is named 'Dockerfile' and is located in a project named 'method2'. The file content is as follows:

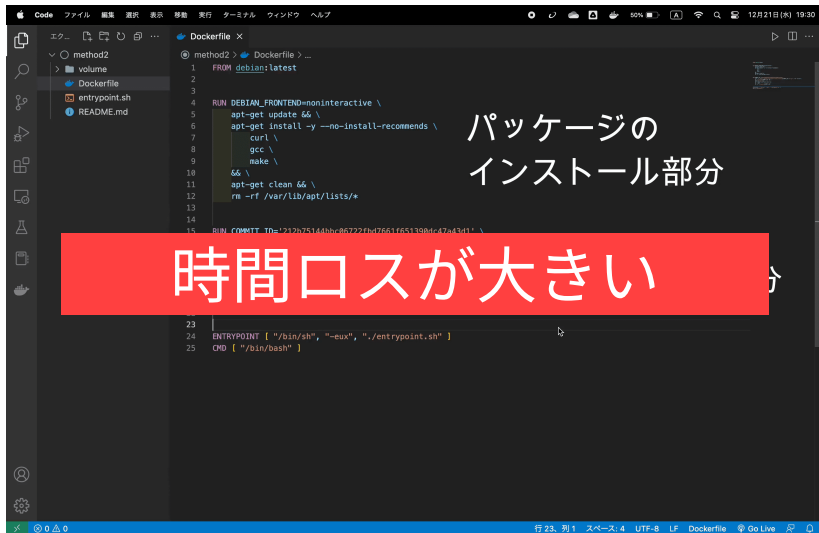
```
1 FROM debian:latest
2
3
4 RUN DEBIAN_FRONTEND=noninteractive \
5     apt-get update && \
6     apt-get install -y --no-install-recommends \
7         curl \
8         gcc \
9         make \
10    && \
11    apt-get clean && \
12    rm -rf /var/lib/apt/lists/*
13
14
15 RUN COMMIT_ID='212b75144bbc06722fbd7661f651390dc47a43d1' \
16     curl -L "https://github.com/ncopa/su-exec/archive/${COMMIT_ID}.tar.gz" | tar -xvz && \
17     cd "su-exec-${COMMIT_ID}" && \
18     make CC="gcc" && \
19     mv su-exec /usr/local/bin/ && \
20     cd .. && \
21     rm -rf "su-exec-${COMMIT_ID}"
22
23
24 ENTRYPOINT [ "/bin/sh", "-euxx", "./entrypoint.sh" ]
25 CMD [ "/bin/bash" ]
```

Annotations in Japanese are overlaid on the right side of the editor:

- パッケージのインストール部分 (Package installation part) - points to lines 4-12.
- su-exec のビルド部分 (su-exec build part) - points to lines 15-21.

The status bar at the bottom shows: 行 23, 列 1 スペース: 4 UTF-8 LF Dockerfile Go Live.

# 開発途中の Dockerfile から逐一生成するコンテナで動作確認をしながら，Dockerfile を開発する例



```
1 FROM debian:latest
2
3
4 RUN DEBIAN_FRONTEND=noninteractive \
5     apt-get update && \
6     apt-get install -y --no-install-recommends \
7         curl \
8         gcc \
9         make \
10    && \
11    apt-get clean && \
12    rm -rf /var/lib/apt/lists/*
13
14
15 RUN COMMIT_ID="212b75144bbr86722fhd7661f651398dc47a43d1"
16
17
18
19
20
21
22
23
24 ENTRYPOINT [ "/bin/sh", "-eux", "./entrypoint.sh" ]
25 CMD [ "/bin/bash" ]
```

パッケージのインストール部分

時間ロスが大きい



## 本ツールを用いて、**Dockerfile** を開発する例

# 本ツールの特長

Dockerfile の効率的な開発手法が必要.



インタラクティブツールとしての側面

ユーザがコンテナ内で動作確認をしながら環境構築するだけで、自動的に Dockerfile を生成してくれる.

これだけでは優れた **Dockerfile** にはならない

前述の方法で自動生成した Dockerfile

これだけでは優れた Dockerfile にはならない

前述の方法で自動生成した Dockerfile

見づらい

# これだけでは優れた Dockerfile にはならない

前述の方法で自動生成した Dockerfile

絶対パス指定の方がいい

集約した方がいい

# Dockerfile のリファクタリング・最適化を行ってみる

リファクタリング・最適化後の Dockerfile

# 本ツールの特長

## インタラクティブツールとしての側面

ユーザがコンテナ内で動作確認をしながら環境構築するだけで、**自動的に** Dockerfile を生成してくれる。

## リファクタリングツールとしての側面

上のようにして生成した Dockerfile に、ベストプラクティスに基づいたリファクタリング・最適化を行うことができる。

# 目次

導入

背景と目的

本ツールの特長

簡単な使い方

まとめ

参考資料 (p.33～)



## 手順 1 / 5

### 「本ツールのリポジトリをクローンする」

本ツールの [GitHub ページ](#) の main ブランチをクローンする.

#### 実行例

```
URL=https://github.com/posl/inada_docker_interactive  
git clone --depth 1 "${URL}"
```

## 手順 2 / 5

### 「開発用のディレクトリを用意する」

開発中の Dockerfile が入るディレクトリを用意する。  
コンテナ内にコピーしたいファイルもここに入れる。

#### 実行例

```
cd ./inada_docker_interactive  
mkdir app
```

## 手順 3 / 5

### 「開発用のコンテナを起動し、開発を開始する」

開発用のディレクトリを設定しながら、`exec.sh` を実行する。  
実行後は、指定したベースイメージに `bash` で入った状態。

#### 実行例

```
sh exec.sh -d ./app -n debian
```

### 「動作確認をしながら、お好みの環境構築を行う」

コンテナ内で動作確認をしながら、環境構築するだけで、それと同時進行で、自動的に Dockerfile が作られていく。

ただし、次の場合は本ツールの機能を使うこと。

- パッケージのインストール → dit package
- ホスト環境からのファイルのコピー → dit copy

## 「Dockerfile のリファクタリング・最適化を行う」

最後に、Dockerfile のリファクタリング・最適化を行う。

```
dit optimize
```

Dockerfile が完成したので、開発を終了する。

```
exit
```

# 目次

導入

背景と目的

本ツールの特長

簡単な使い方

まとめ

参考資料 (p.33～)

## 今後の展望

- 実装途中の部分を完成させる.
- ドキュメントページを作成する.
- Dockerfile に命令を追加する機能を充実させる.
- マルチステージビルドに対応する.
- 標準入力を捕捉できるようにする.

## 背景

- Dockerfile を効率的に開発するためのツールがない。
- Dockerfile のリファクタリング・最適化ツールがない。

## 提案ツール

- コンテナ内の bash 環境で，対話的に作業する。
- 動作確認だけに集中して，Dockerfile が作成できる。
- リファクタリング・最適化後の Dockerfile が得られる。



# 目次

導入

背景と目的

本ツールの特長

簡単な使い方

まとめ

参考資料 (p.33～)

# 基本情報

- ツールの Dockerfile から開発用のコンテナを起動し、  
bash コマンドを用いて、対話的に作業する。
- 開発用のディレクトリには、以下などが入る。
  - Dockerfile (下書き・完成版)
  - 履歴ファイル
  - ユーザがコンテナ内にコピーしたいファイル
- ツールの機能の利用は、dit<sup>1</sup> コマンドから行う。
- /dit<sup>1</sup> ディレクトリにツールが利用するファイルが入るが、不用意にそれらに触れてはいけない。

---

<sup>1</sup>Docker Interactive Tool の略。

# 機能要件

1. シェルで任意のコマンドを実行するたびに，その必要性を判断して，対応する命令を Dockerfile に追加する機能．
2. CUI 上で Dockerfile を編集する機能．
3. ベストプラクティスに基づいて，作成された Dockerfile のリファクタリング・最適化を行う機能．
4. Dockerfile の開発を任意のタイミングで中断・再開できるようにする機能．

# 機能要件

1. シェルで任意のコマンドを実行するたびに，その必要性を判断して，対応する命令を Dockerfile に追加する機能.
2. CUI 上で Dockerfile を編集する機能.
3. ベストプラクティスに基づいて，作成された Dockerfile のリファクタリング・最適化を行う機能.
4. Dockerfile の開発を任意のタイミングで中断・再開できるようにする機能.

## 機能 1

「シェルで任意のコマンドを実行するたびに処理を行う」

bash のシェル変数 `PROMPT_COMMAND` を使用する.

### 使用例

```
# records the latest exit status  
PROMPT_COMMAND='echo "$?" > /tmp/exit-status'
```

# 機能 1

## 「コマンドの反映の可否を判断する」

以下の設定情報（詳細は後述）

- 5つの反映モード
- 反映しないコマンドと条件をまとめた JSON ファイル  
（以下「**ignore ファイル**」と称する）

とコマンドラインの解析結果などを元に判断する。

設定情報はそれぞれ、`'dit config'`・`'dit ignore'` で編集できる。

# 機能 1

## 「対応する命令を Dockerfile に追加する」

以下のような変換規則

- 代入文 → ARG 命令
- cd コマンド → WORKDIR 命令

以下のような反映ルール

- 通常ファイルへのリダイレクトを行う部分は反映する.
- パイプラインの切り詰めは右からのみ行う.

を適用して、追加処理を行う.

このような変換処理は、'dit convert' が担当する.

## 機能1 設定情報 (config)

### 5つの反映モード

**no-reflect** Dockerfile に命令を追加しない.

**strict** できる限り, ignore ファイルの設定に従う.

**normal** 処理のまとまりを考え, 反映の可否を変更する.

**simple** 単純なコマンド以外はそのまま反映する.

**no-ignore** 実行されたコマンドラインは必ず反映する.

### strict と normal の違い

'hoge && piyo' が実行された場合,

**strict** hoge だけを反映する可能性がある

**normal** ひとまとめに, 反映の可否を決定する

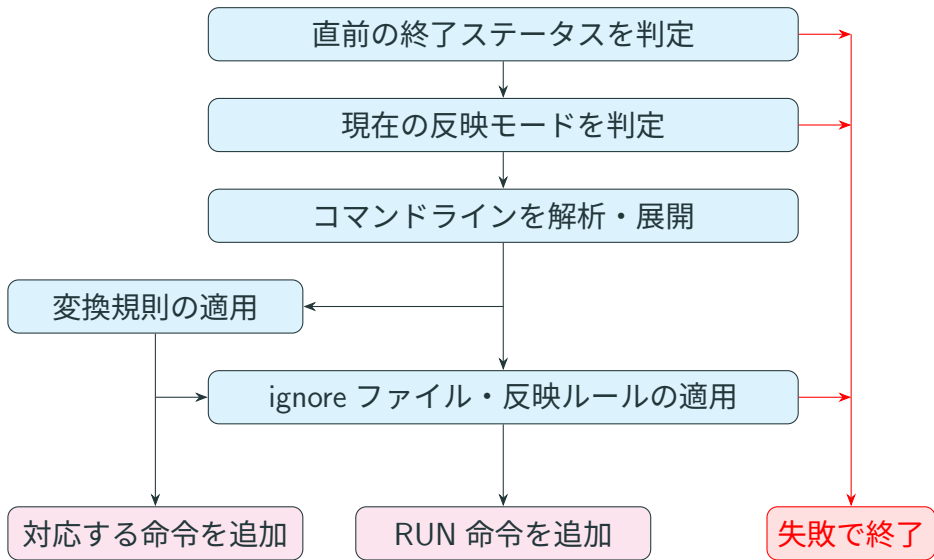


## 機能1 設定情報 (ignore)

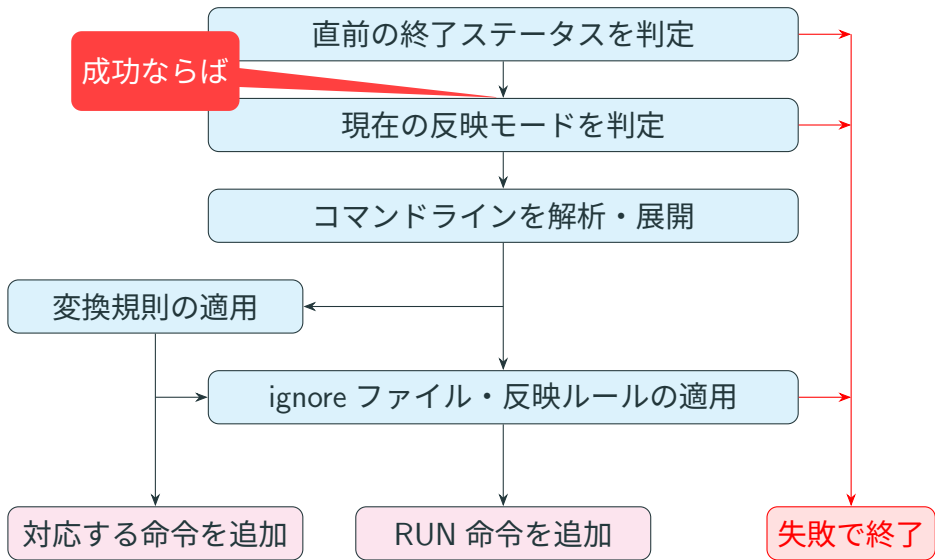
### ignore ファイルの例

```
1 {
2     "ls": null,
3     "dir": "ls",
4     "wget": {
5         "short_opts": "0:",
6         "long_opts": {
7             "output-document": 1
8         },
9         "optargs": {
10             "output-document": "0",
11             "0": [
12                 "_"
13             ]
14         },
15         "detect_anymatch": true
16     }
17 }
```

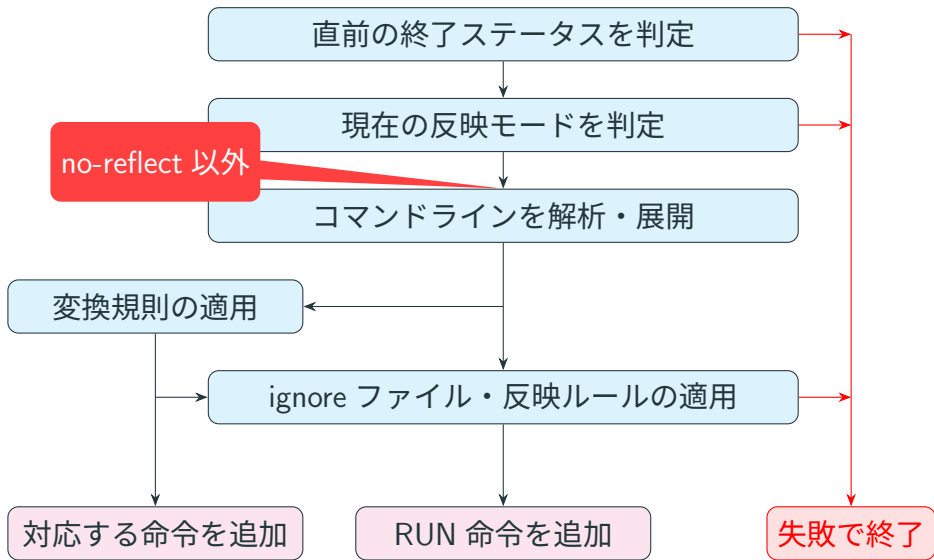
## 機能1 変換処理の流れ (convert)



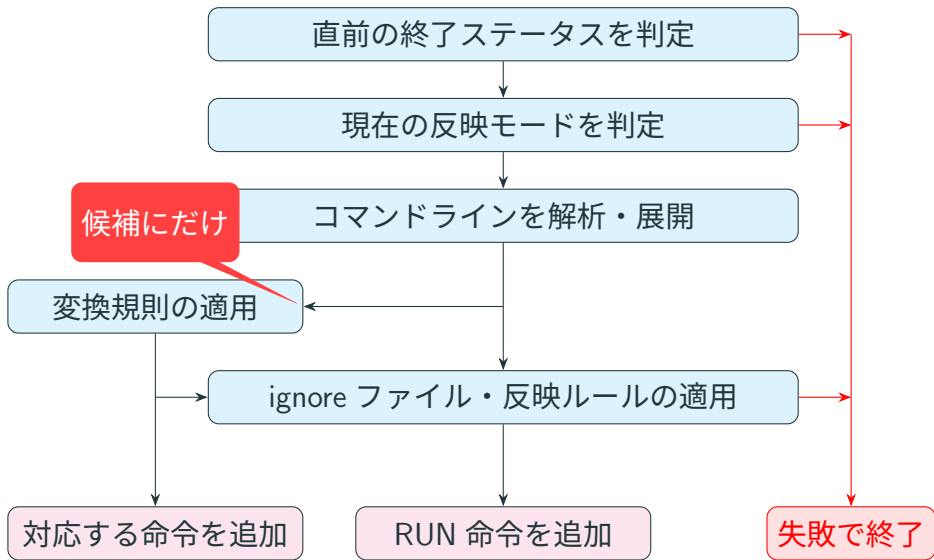
## 機能1 変換処理の流れ (convert)



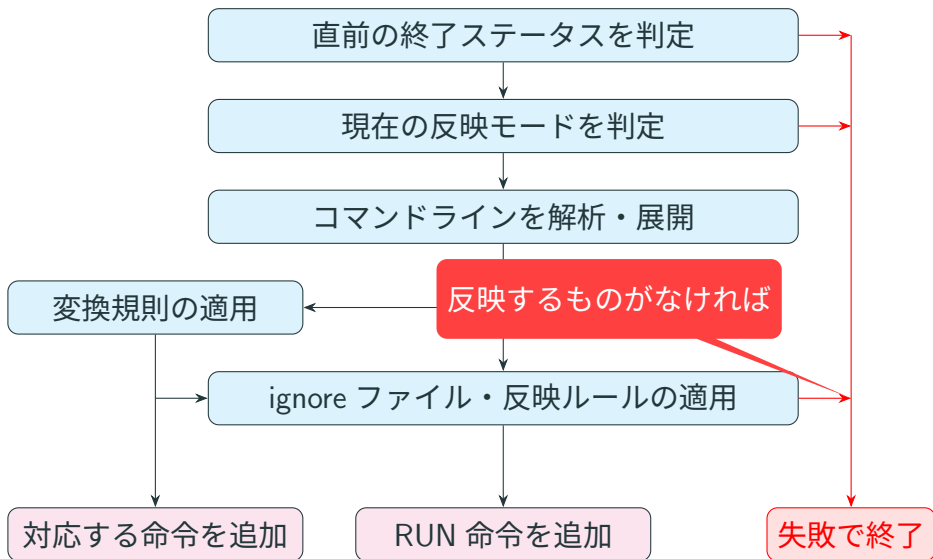
## 機能1 変換処理の流れ (convert)



## 機能1 変換処理の流れ (convert)



## 機能1 変換処理の流れ (convert)



## 機能 1 変換処理の具体例 (convert)

以下の設定で、次のコマンドラインの実行が成功した場合。

設定情報

反映モード strict

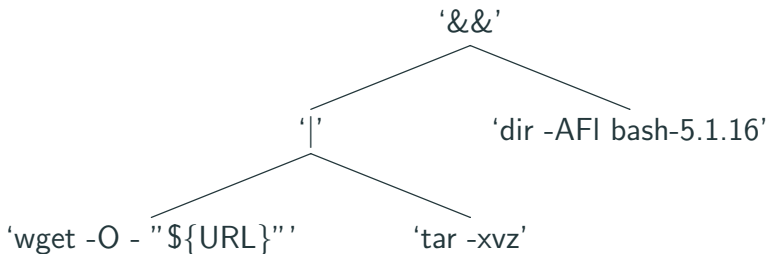
ignore ファイル 先述の内容

コマンドライン

```
# URL=https://ftp.gnu.org/gnu/bash/bash-5.1.16.tar.gz  
wget -O - "${URL}" | tar -xvz && dir -AF1 bash-5.1.16
```

## 機能1 変換処理の具体例 (convert)

コマンドラインの解析・展開後

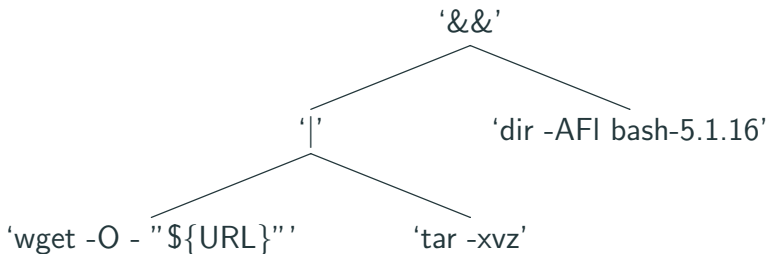


適用	'wget'	'tar'	'dir'
ignore ファイル			
反映ルール			



## 機能1 変換処理の具体例 (convert)

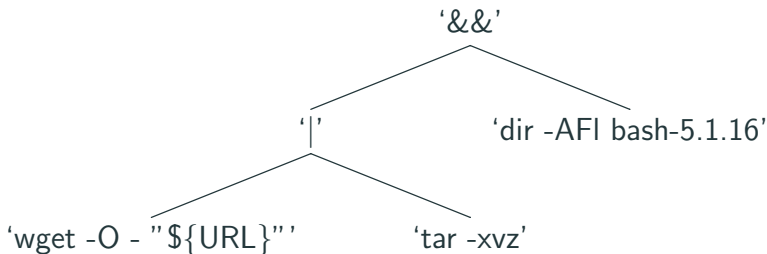
コマンドラインの解析・展開後



適用	'wget'	'tar'	'dir'
ignore ファイル 反映ルール	反映しない	反映する	反映しない

## 機能1 変換処理の具体例 (convert)

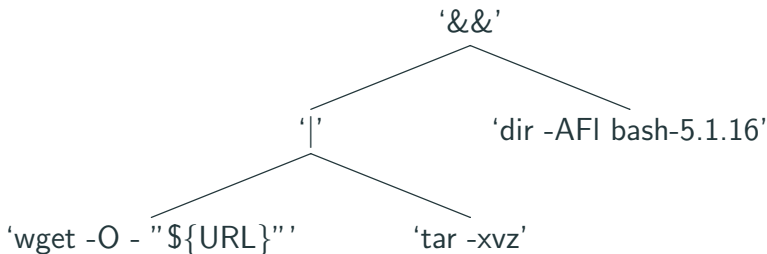
コマンドラインの解析・展開後



適用	'wget'	'tar'	'dir'
ignore ファイル	反映しない	反映する	反映しない
反映ルール	反映する	反映する	反映しない

## 機能1 変換処理の具体例 (convert)

コマンドラインの解析・展開後



適用	'wget'	'tar'	'dir'
ignore ファイル	反映しない	反映する	反映しない
反映ルール	反映する	反映する	反映しない

よって、`'RUN wget -O - "${URL}" | tar -xvz'` を追加する。

# 機能要件

1. シェルで任意のコマンドを実行するたびに，その必要性を判断して，対応する命令を Dockerfile に追加する機能.
2. CUI 上で Dockerfile を編集する機能.
3. ベストプラクティスに基づいて，作成された Dockerfile のリファクタリング・最適化を行う機能.
4. Dockerfile の開発を任意のタイミングで中断・再開できるようにする機能.

## 機能 2

### 「ホスト環境からファイルをコピーする」

#### dit copy

ホスト環境からコンテナ内へのファイルのコピーを行い、その内容を COPY・ADD 命令として Dockerfile に反映する。

このコマンドを使うと、

- COPY 命令と cp コマンドの仕様の違いを意識せず済む。
- ADD 命令の tar 展開機能を利用できる。

## 機能 2

### 「パッケージをインストールする」

#### dit package

最適化された形式で、パッケージのインストールを行い、その内容を RUN 命令として Dockerfile に反映する。

このコマンドを使うと、

- イメージサイズの削減に効果的な方法で実行される。
- パッケージマネージャの違いをあまり意識せず済む。

## 機能 2

### 「Dockerfile に命令を追加する」

#### dit reflect

各種ログを取りながら，Dockerfile に命令を追加する．

#### 実行例

```
# reflects the contents of './instr.txt' in Dockerfile  
dit reflect -d instr.txt
```

```
# reflects the input contents as they are in Dockerfile  
dit reflect -dp -
```

## 機能 2

### 「Dockerfile から指定した行を削除する」

dit erase

条件にマッチする行を，Dockerfile から削除する．

#### 実行例

```
# deletes the lines added just before from Dockerfile
dit erase -d
```

```
# deletes all LABEL instructions from Dockerfile
dit erase -diy -E '^LABEL[[:space:]]'
```



# 機能要件

1. シェルで任意のコマンドを実行するたびに，その必要性を判断して，対応する命令を Dockerfile に追加する機能。
2. CUI 上で Dockerfile を編集する機能。
3. ベストプラクティスに基づいて，作成された Dockerfile のリファクタリング・最適化を行う機能。
4. Dockerfile の開発を任意のタイミングで中断・再開できるようにする機能。

# 「Dockerfile のリファクタリング・最適化を行う」

### dit optimize

下書きの Dockerfile から，完成版の Dockerfile を生成する．

#### 処理内容の例

- WORKDIR 命令のオペランドを絶対パスにする．
- ENV・ARG 命令の重複や無意味な再定義を取り除く．
- 連続する RUN 命令をまとめる．
- 可読性を向上させるための整形を行う．

# 機能要件

1. シェルで任意のコマンドを実行するたびに，その必要性を判断して，対応する命令を Dockerfile に追加する機能．
2. CUI 上で Dockerfile を編集する機能．
3. ベストプラクティスに基づいて，作成された Dockerfile のリファクタリング・最適化を行う機能．
4. Dockerfile の開発を任意のタイミングで中断・再開できるようにする機能．

## 機能 4

### 「Dockerfile の開発を中断・再開する」

履歴ファイルというものを導入することで実現する。

**開発中** 履歴ファイル用の反映モードと ignore ファイルを別で用意し、Dockerfile と同じようにして編集する。

**再開時** source コマンド・history コマンドで履歴ファイルを読み込み、環境を再現する。