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

目次

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ブロイ・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．

build

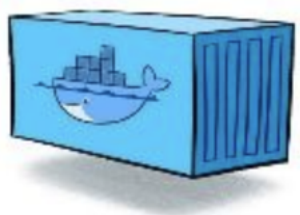
$\longrightarrow$

run

$\longrightarrow$

(commit)

$\longleftarrow$



Dockerfile　　イメージ　　コンテナ

目次

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，DockerSlim
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，DockerSlim
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，DockerSlim
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

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

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

手間が多い

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

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

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

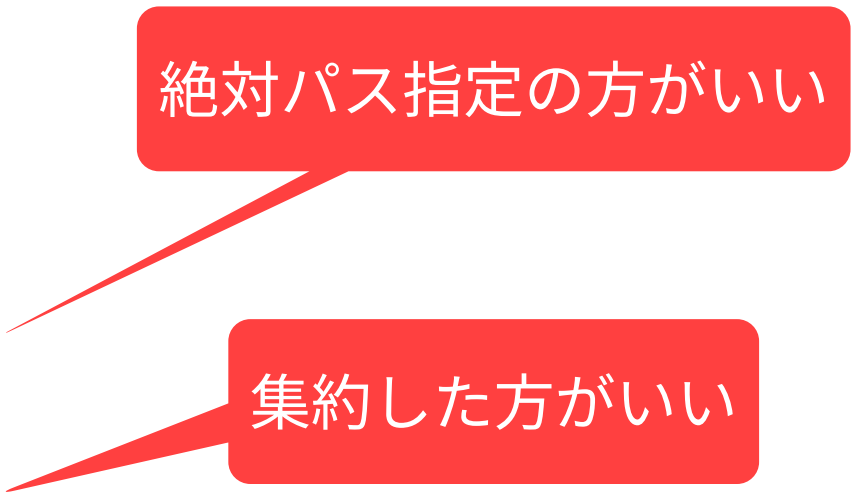
これだけでは優れた Dockerfile にはならない

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



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

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
```

手順 4 / 5
「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 $\longrightarrow$ dit install
- ホスト環境からのファイルのコピー $\longrightarrow$ dit copy

手順 5 / 5
「**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

今後の展望

- ドキュメントページを作る．
- Dockerfile に命令を追加する機能を充実させる．
  - LABEL・EXPOSE 命令
  - CMD・ENTRYPOINT 命令
  - HEALTHCHECK 命令
  - ONBUILD 命令
- 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．
  - ‘apt-get install’ など確認作業があるコマンドに効果大．
  - 上の例なら，-y オプションを付けなくても良くなる．

まとめ

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

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．
- バインドマウントしたディレクトリに以下などが入る．
  - 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・設定の変更は/usr/local/bin にある dit[1] コマンドから行う．（エイリアスあり）
- /dit[1] 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．

[1] 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 から 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 から 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
「コマンドの反映の要否を判断する」

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

- 単独の cd コマンド $\longrightarrow$ WORKDIR 命令
- コマンドを伴わない代入文 $\longrightarrow$ ARG 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.
- パイプラインは前後関係が反映の要否に影響する.

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 反映の要否の決定に処理の流れを考慮しない

normal 処理の流れを汲んで，反映の要否を変更する

simple 単純なコマンド以外はそのまま反映する

no-ignore ignore ファイル・反映ルールを適用しない

strict と normal の違い

‘hoge; piyo’ が実行された場合，

strict 互いに独立して反映するかどうかを決める

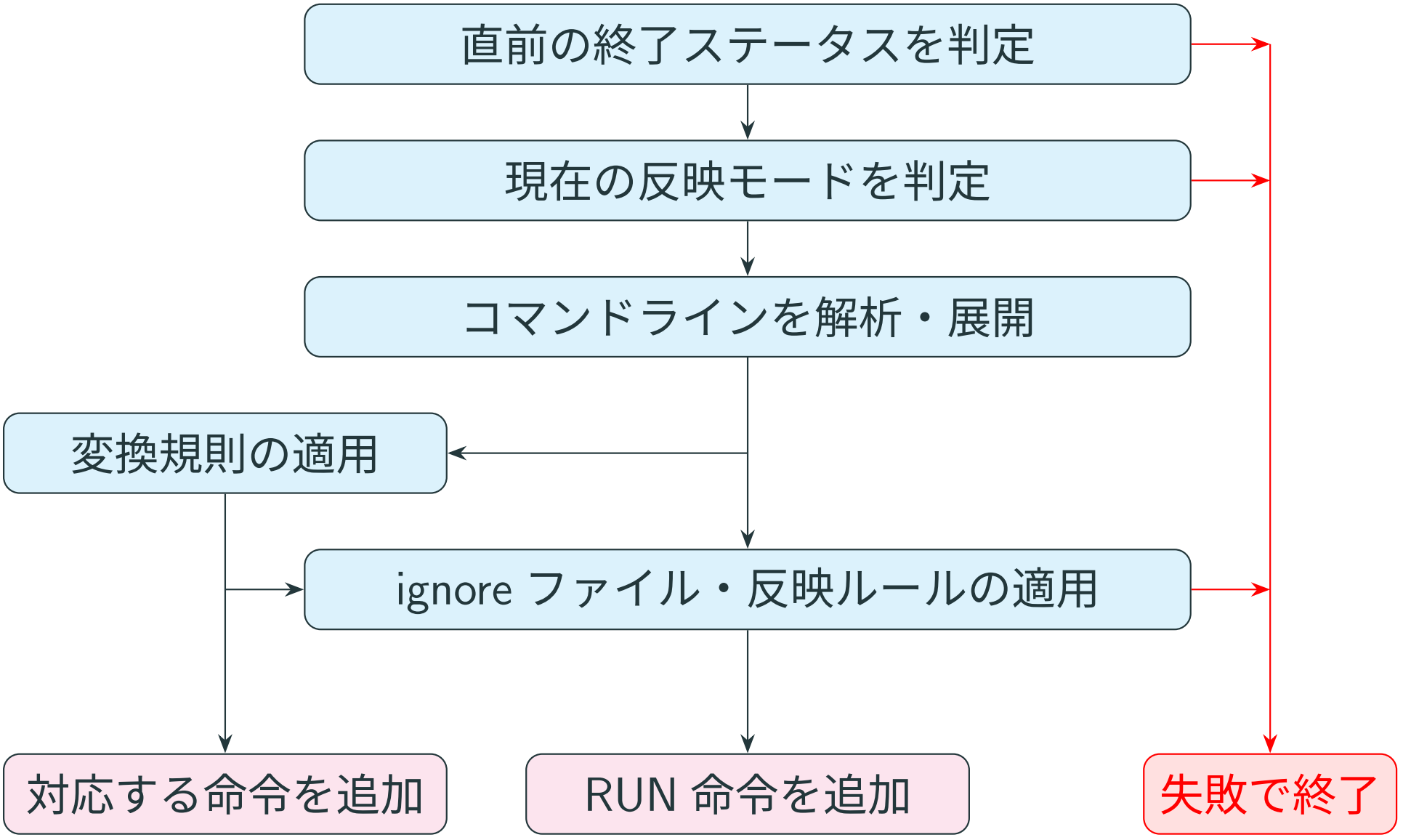
normal piyo を反映するならば，必ず hoge も反映する

機能１　設定情報（ignore）

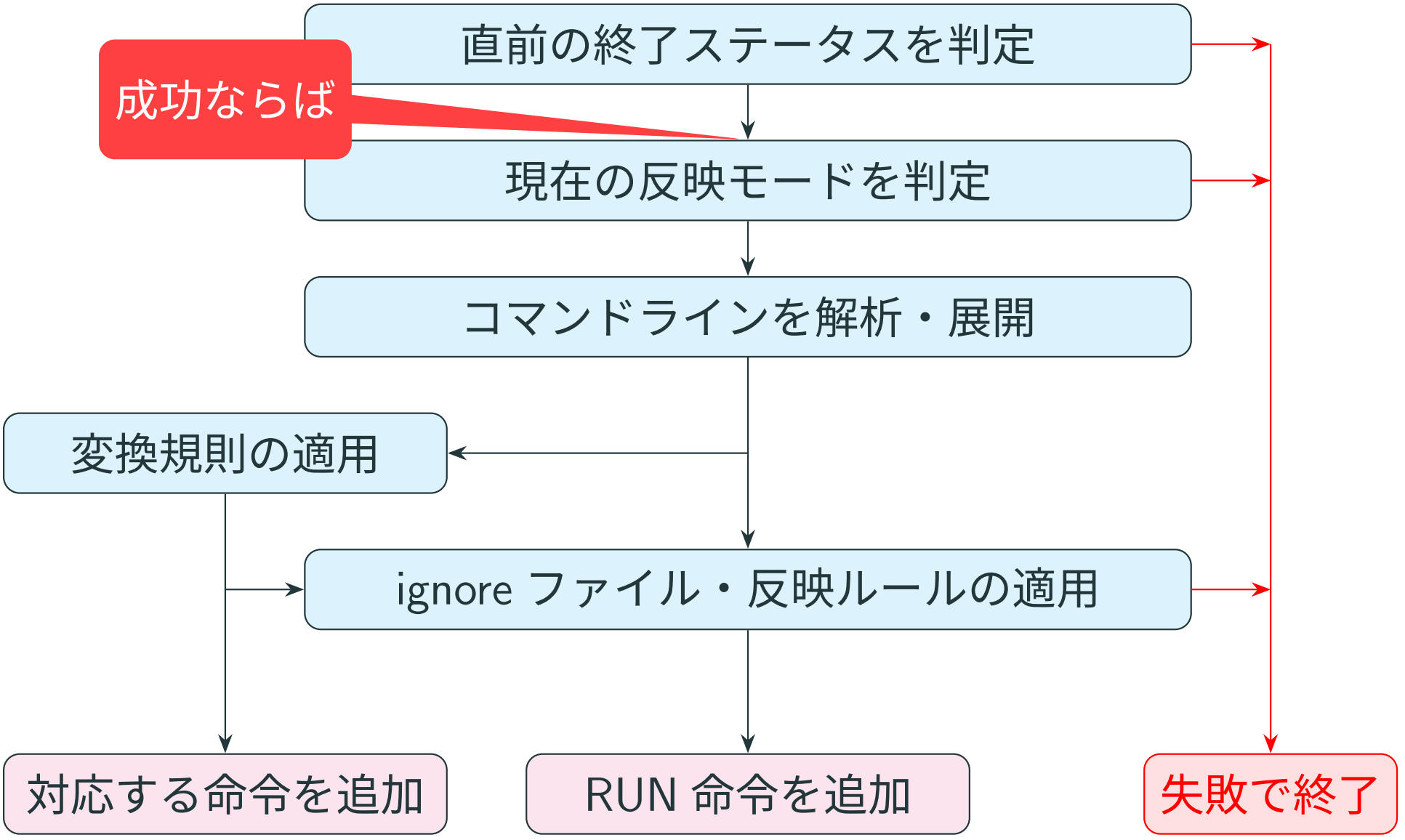
ignore ファイルの例

```
{
    "ls": null,
    "export": {
        "short_opts": "p",
        "cmdargs": [
            null
        ]
    },
    "bash": {
        "short_opts": "nD",
        "long_opts": [
            [ "dump-strings", 0 ],
            [ "dump-po-strings", 0 ]
        ]
    }
}
```

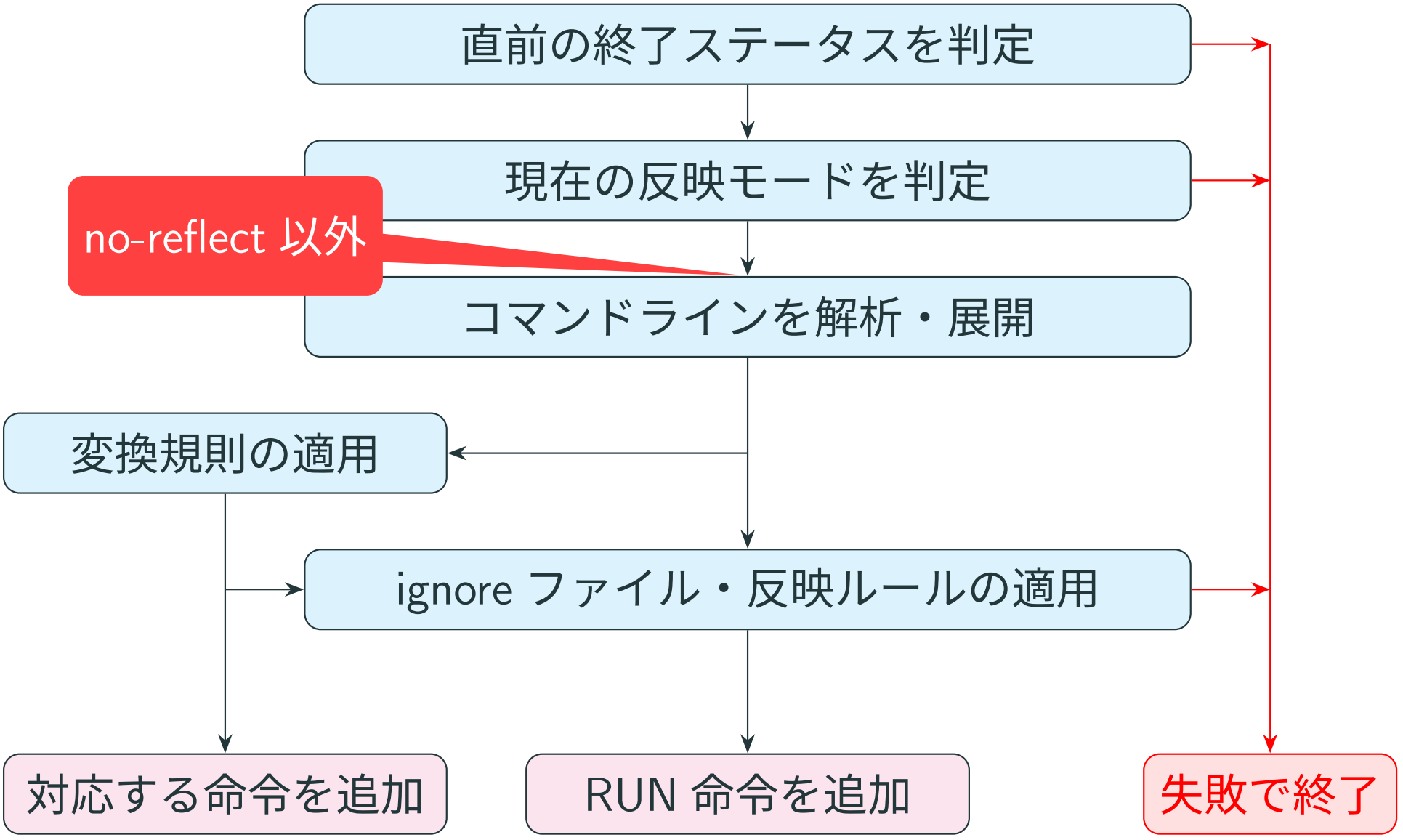
機能１　変換処理の流れ（convert）



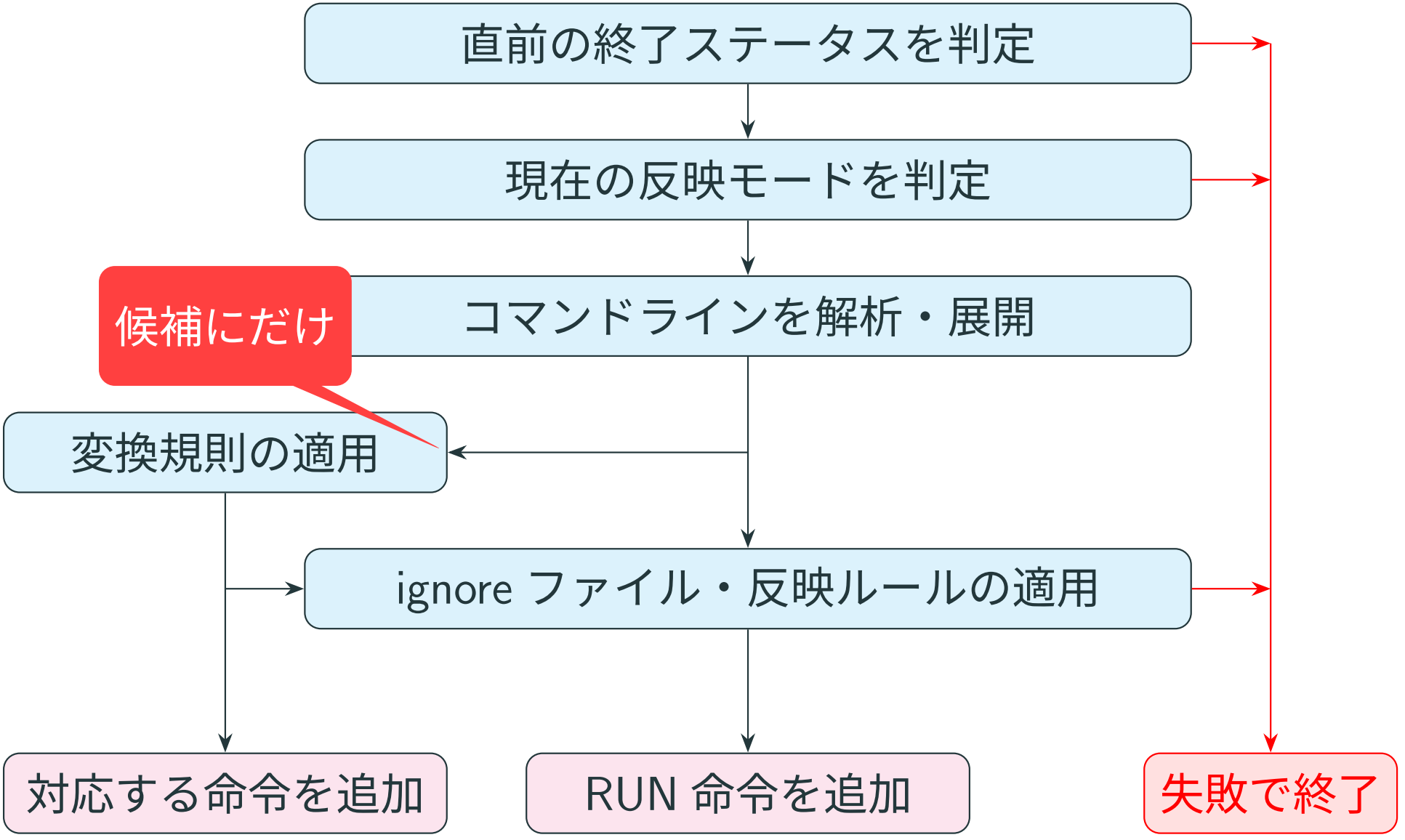
機能１　変換処理の流れ（convert）



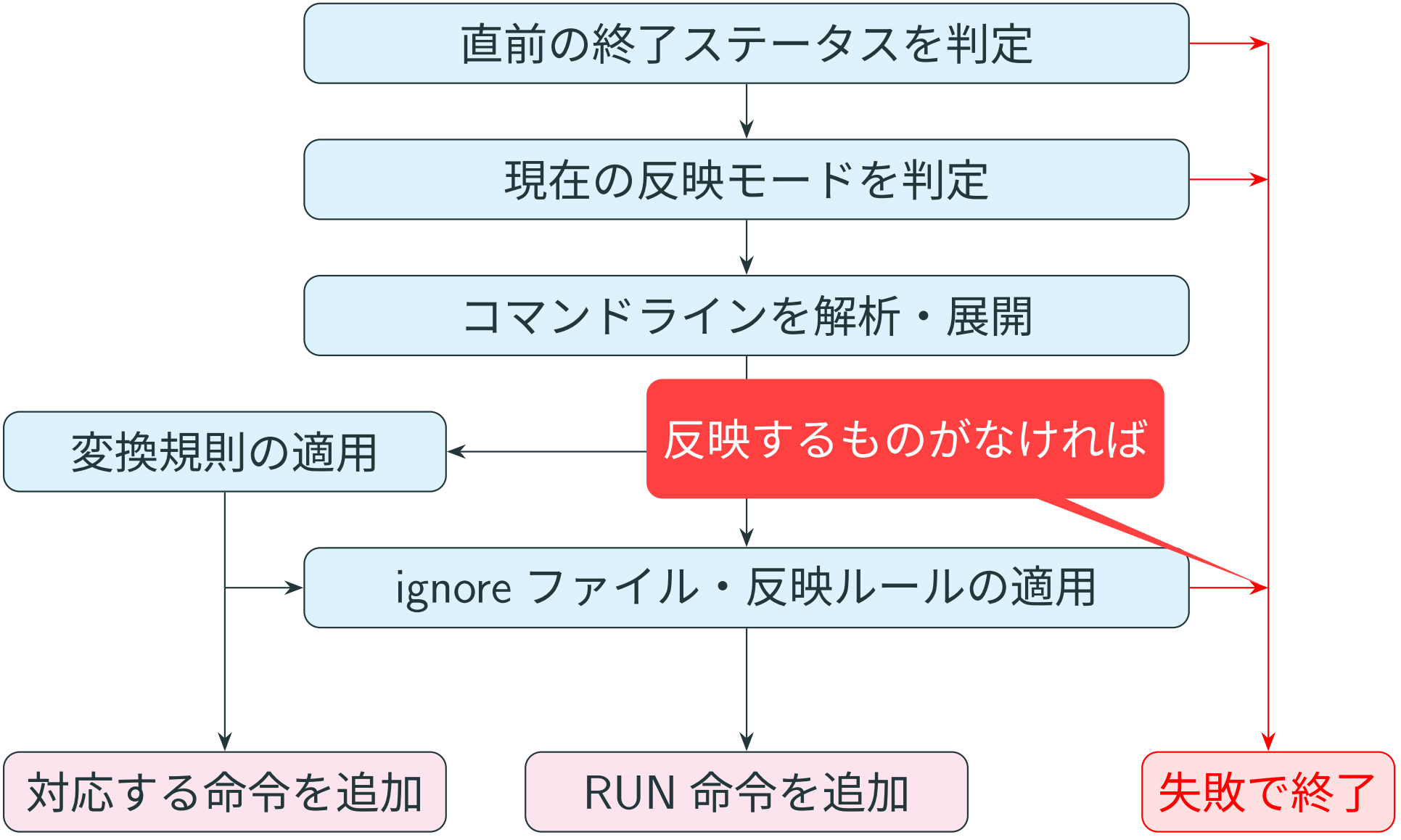
機能１　変換処理の流れ（convert）



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

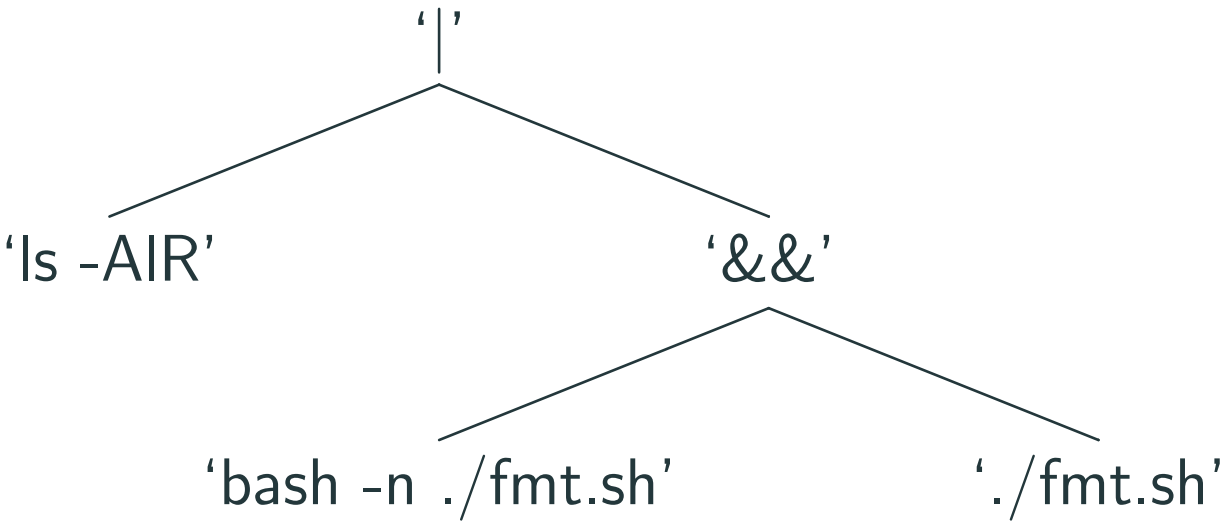
ignore ファイル 先述の内容

コマンドライン

```
# lists information about the current directory
ls -AlR | ( bash -n ./fmt.sh && ./fmt.sh )
```

機能１　変換処理の具体例（convert）

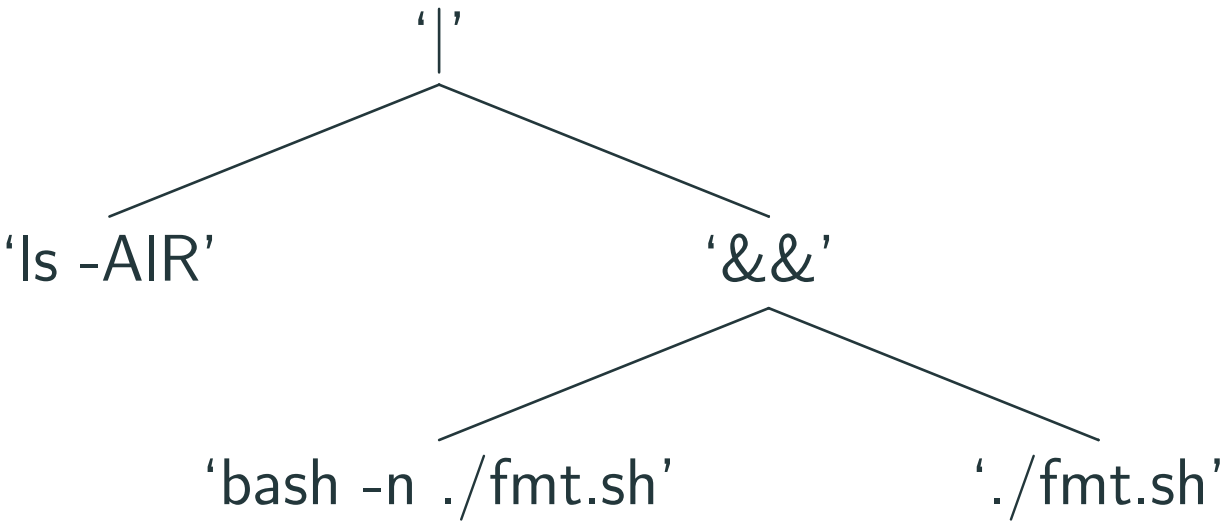
コマンドラインの解析・展開後



適用	‘ls -AIR’	‘bash -n ./fmt.sh’	‘./fmt.sh’
ignore ファイル			
反映ルール			

機能１　変換処理の具体例（convert）

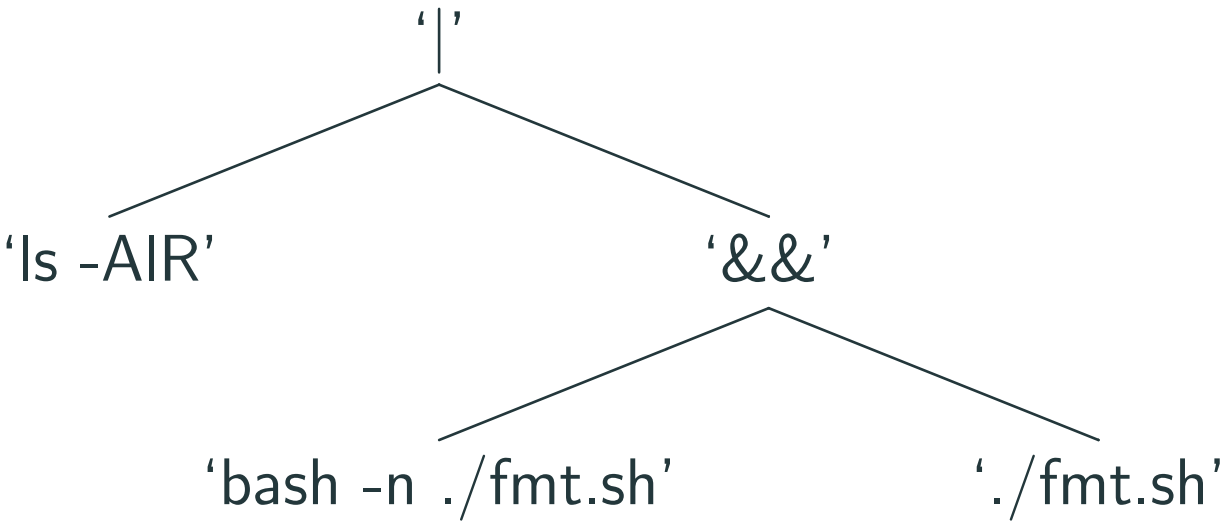
コマンドラインの解析・展開後



適用	‘ls -AIR’	‘bash -n ./fmt.sh’	‘./fmt.sh’
ignore ファイル 反映ルール	反映しない	反映しない	反映する

機能１　変換処理の具体例（convert）

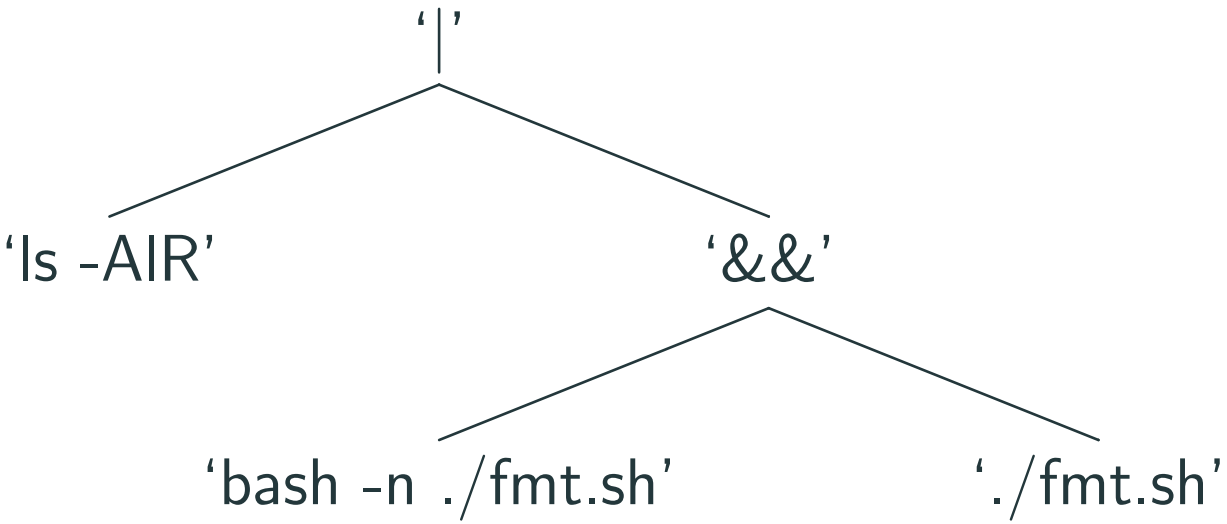
コマンドラインの解析・展開後



適用	‘ls -AIR’	‘bash -n ./fmt.sh’	‘./fmt.sh’
ignore ファイル	反映しない	反映しない	反映する
反映ルール	反映する	反映しない	反映する

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	‘ls -AIR’	‘bash -n ./fmt.sh’	‘./fmt.sh’
ignore ファイル	反映しない	反映しない	反映する
反映ルール	反映する	反映しない	反映する

よって，‘RUN ls -AIR | ./fmt.sh’ 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 から 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
「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 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 から 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３
「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，ファイルをコピーする．

この時，ホスト環境にある.dockerignore の編集も行う．

これにより，ビルド時間が短縮できるようになる．

機能３
「パッケージをインストールする」

dit install

最適化された形式で，パッケージのインストールを行う．

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．
- 最後に，更なるリファクタリング・最適化が行える．

機能３
「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

- 順序に意味がない命令をそれぞれ集めて，並べ替える．
- 可読性を向上させるための整形を行う．
- 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 命令をリスト実行の&&でまとめる．

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 から 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.

中断前 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.