2014年12月18日

Unix/Linux

Makeについて知っておくべき7つのこと

SOURCE

7 Things You Should Know About Make （2014-10-26） by Alexey Shmaiko

本記事は、原著者の許諾のもとに翻訳・掲載しております。

Makeは、様々なタイプのファイルのビルド作業を自動的に行ってくれるシンプルかつ強力なツールです。しかしながら、makefileを書く際に問題にぶち当たるプログラマもいれば、Makeの基

- 翻訳リクエストを送る
- 翻訳フィードバック
  - メール
  - GitHub Issue

最近の投稿
- 私たちはいかにして環状線で"悪さをする列車"を捕まえたか
- ーから学ぶベジェ曲線
- 型クラスはインターフェースとどう違うのか
- どれだけ速く文字列からスペースを削除できるのか
- Node.jsのパフォーマンス最適化を阻むものの見つけ方

タグ
ABテスト
AngularJS AWS

本知識がないことで、既存のものを再発明してしまうプログラマもいます。

Makeの働き

デフォルトでは、Makeは一番目のターゲットから開始します。このターゲットのことをデフォルトゴールと呼びます。

Makeはカレントディレクトリのmakefileを読み込み、一番初めのルールで処理を開始します。しかし、Makeが完全にこのルールを処理する前に、ルールが依存するファイルのためのルールを処理しなければなりません。各ファイルそれぞれは、自身のルールに従って処理されます。

実はこれは、各ターゲットの再帰的アルゴリズムになっています。

1. ターゲットをビルドするルールを見つける。ルールがないようであれば、Makeはうまく動作しません。
2. ターゲットの各必要条件には、その必要条件をターゲットとしてこのアルゴリズムを実行します。
3. ターゲットが存在しない、または、必要条件の更新時間がターゲットの更新時間よりも後である場合は、ターゲットと関連付いているレシピを実行します。レシピが失敗するようであれば、（通常は）Makeはうまく動作しません。

代入のタイプ

Makeでは、makefileを書くのを簡素化するために変数が使われ、 = 、 ?

C++ Clojure CSS
C言語 Docker
D言語 EdTech
Elm Erlang Git
Github Google
Go言語 Haskell
iOS Java
JavaScript jQuery
Lisp MySQL
NodeJS NoSQL
Objective-C
OCaml PHP
podcast
PostgreSQL
Python React
Ruby Rust Scala
SEO SSL
Stack Overflow
Swift TDD
TypeScript
UIデザイン
Unix/Linux
webサーバ
Y Combinator
アジャイル
アルゴリズム
エコシステム
エンジニア採用
オープンソース
キャリアパス
グロースハック
ゲーム開発
コードレビュー
セキュリティ
ソフトウェアアーキテクチャ
データサイエンス

=、 :=、 ::=、 +=、 != から1つの演算子で代入されます。それぞれの演算子の違いは、以下の通りです。

- = は遅延された値を変数に代入します。つまり、変数が使われるたびに変数の値が求められます。シェルコマンドの結果を代入するとき、変数が読み込まれるたびにシェルコマンドが実行されることを忘れないでください。
- := と ::= は、基本的には同じ意味です。このような代入は、変数値を一度だけ処理し、記憶します。簡潔かつ強力であるこのようなタイプの代入は、デフォルトとして選びましょう。
- ?= は、変数が定義されていないときのみ := として機能します。そうでない場合は、何も起きません。
- += は、加算代入演算子です。変数があらかじめ := もしくは ::= に設定されている場合、右辺は即値とみなされます。そうでない場合は、遅延された値とみなされます。
- != は、シェルの代入演算子です。右辺は即座に評価されシェルに渡されます。結果は、左辺にある変数に記憶されます。

パターンルール

同じルールを持つたくさんのファイルがある場合、ターゲットをマッチさせるためにパターンルールを定義することができます。パターンルールは、ターゲットに'%'があることを除いては、通常のルールと同じです。これがあることによって、パターンルールのターゲットは、ファイル名に一致させるパターンと判断され、'%'は空でない部分文字列に一致させることができます。

データベース
デバッグ ネタ
ハッカソン
パフォーマンス
ビジネスモデル
ビジュアライゼーション
ビジュアルデザイン
フリーランス
フレームワーク
プログラミング言語比較
プロジェクト管理
ベストプラクティス
マイクロサービス
まとめ
リファクタリング
入門 型システム
機械学習 生産性
経営組織論 統計
起業アイデア
起業家
関数型プログラミング
顧客開発

アーカイブ
2017年02月
2017年01月
2016年12月
2016年11月
2016年10月
2016年09月
2016年08月
2016年07月
2016年06月
2016年05月
2016年04月
2016年03月
2016年02月

ができます。

私のブログディレクトリには次のMakefileがあります。

```
1. all: \
2.       build/random-advice.html \
3.       build/proactor.html \
4.       build/awesome_skype_fix.html \
5.       build/ide.html \
6.       build/vm.html \
7.       build/make.html \
8.
9.     build/%.html: %.md
10.      Markdown.pl $^ > $@
```

`$@` がターゲットを意味するのに対し、 `$^` は依存関係を意味する自動変数です。つまり、単純にマークダウンファイルをコンバータに渡すというルールです。パターンルールの書き方や自動変数に関する詳細は、マニュアルを参照してください。

デフォルトの暗黙ルール

GNU Makeにはデフォルトのルールがあります。多くの場合明示的なルールを書く必要はありません。デフォルトの暗黙ルールのリストはC、C++、アセンブラプログラムとそれらをリンクすることを含みますが、その限りではありません。完全なリストはMakeのマニュアルで参照できます。

Makefileに何もさせないことは可能です。たとえば、単にhello.cというファイルにプログラムのソースコードを保存して、単に `make hello` を実行できます。Makeはhello.c からhello.oを自動的にコンパイルしてhelloにリンクします。

レシピは `$(CC) $(CPPFLAGS) $(CFLAGS) -c` の形式で定義します。変数を変えることでルールを

- 2016年01月
- 2015年12月
- 2015年11月
- 2015年10月
- 2015年09月
- 2015年08月
- 2015年07月
- 2015年06月
- 2015年05月
- 2015年04月
- 2015年03月
- 2015年02月
- 2015年01月
- 2014年12月
- 2014年11月
- 2014年10月
- 2014年09月
- 2014年08月
- 2014年07月
- 2014年06月

フォロー

- Follow
- @POSTDccさんを
- いいね！ 3,090 シェア

変えられます。ソースファイルをclangでコンパイルするためには、単に `CC := clang` という行を加えるだけです。私は小さなテストプログラムを保存するディレクトリにとても小さなMakefileを置いています。

```
1. CFLAGS := -Wall -Wextra -pedantic -std=c11
2. CXXFLAGS := -Wall -Wextra -pedantic -
   std=c++11
```

ワイルドカードと関数

カレントディレクトリのすべてのCとC++ソースファイルをコンパイルするには、依存関係のために `$(patsubst %.cpp,%.o,$(wildcard *.cpp)) $(patsubst %.c,%.o,$(wildcard *.c))` というコードを使います。

`wildcard` はパターンにマッチするすべてのファイルを検索して、patsubstは妥当なファイル拡張子を `.o` で置き換えます。

Makeにはテキストを変換するためのたくさんの関数があり、 `$(function arguments)` という形式で呼び出します。

関数の完全なリストはマニュアルを参照してください。

なおコンマのあとのスペースは引数の一部とみなされる点に注意してください。スペースがあるといくつかの関数で予期しない結果を引き起こすので、私はコンマのあとにスペースを全く置かないことをお勧めします。

`call` 関数で独自の関数や `eval` 関数でパラメータ化されたテンプレートのようなものを書くこともできます。

検索パス

Makeには特別な変数 VPATH があり、すべての必要条件のための PATH として使われます。またVPATH変数ではディレクトリ名をコロンや空白で区切ります。ディレクトリの並び順はMakeが検索する順序になります。このルールは、すべてのファイルがカレントディレクトリに存在するかのように、必要条件のリストでファイル名を指定できるようにします。

さらにきめ細かな vpath ディレクティブもあります。これはパターンにマッチするファイルごとに検索パスを指定できます。そのため include ディレクトリにすべてのヘッダを保存するなら、以下の行を使えます。

```
1. vpath %.h include
```

しかしながら、Makeはルールの必要条件の部分だけ変えてルール自身を変えないので、ルールでは明示的なファイル名に頼れません。代わりに $^ のような"*自動変数*"を使用しなければなりません。

必要条件のためのディレクトリ検索の詳細はMakeのマニュアルを参照してください。

makefileのデバッグ

makefileをデバッグするためのいくつかのテクニックがあります。

出力

最初のものは単に昔ながらの出力です。以下のMake関数の1つを使って、その表現の値を出力

できます。

```
1. $(info ...) $(warning ...) $(error ...)
```

この行を通過するとMakeはその表現の値を出力します。

出力の使い方はご存知だと思います。

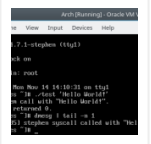
Remake

Makefileをデバッグするために書かれた特別なプログラムもあります。Remakeは指定されたターゲットで止まって、起こったことを調べて、Makeの内部状態を変えることができます。詳細はRemakeによるmakefileのデバッグについての記事を読んでください。

また他の方法に関してmakefileのデバッグについての素晴らしい記事も読んでください。

翻訳に対するフィードバックがございましたら、メールまたはGitHubのIssueよりお寄せください。

関連した投稿



チュートリアル – システムコールの書き方

2017.01.27

もしもディスプレイが壊れたら(ヒマラヤで)

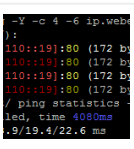
2016.11.18

Linuxシステムコール徹底ガイド

2016.07.28

構造化テキストデータを操作するためのコマンドラインツールリスト

2016.06.02



Pingの発展版 : httping, dnsping, smtpping

2016.06.01

NEXT POST

Angularチームは、どうかしちゃった？

PREVIOUS POST

Git活用法 ー コードはいつも1行ごとにドキュメント化されている