

minus9d's diary

2014-02-03

Makefileの書き方に関する備忘録

linux

12

ブックマーク

※この記事は続き記事になっています

- 第1回:この記事
- 第2回:Makefileの書き方に関する備忘録 その2 - minus9dの日記
- 第3回:Makefileの書き方に関する備忘録 その3 - minus9dの日記

makeの使い方を復習しました。今更makeかよと思われそうですが、曖昧に覚えているところも多く、よい勉強になりました。参照したテキストは、[オライリー](#)がオープンブックとして公開している [Managing Projects with GNU Make, Third Edition](#) (GNU Make 第三版)。今回の記事はほぼ全部これが出典です。

最初の一步

makefileに以下の内容を書いて保存。[コマンドライン](#)でmakeと単に打つと、このmakefileが実行対象となる。

```
hello: hello.c
    gcc hello.c -o hello
```

この内容は以下のようなルールを意味する。

```
target: prerequisites
    command
```

ここで、

- target: 生成されるもの。ここでは実行ファイルhello。
- prerequisites: targetが生成されるために必要なもの。ここでは[ソースコード](#)hello.c。
- command: prerequisitesを使ってtargetを生成するためのシェルコマンド。

makeを実行するとhelloが生成される。makeを再実行した時点で、targetよりもprerequisitesの方が新しければcommandが再実行される。

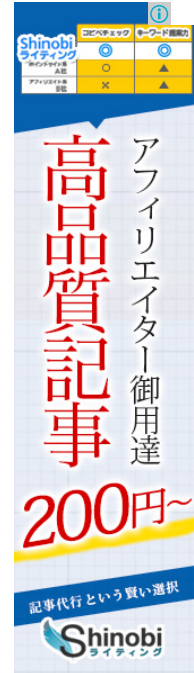
makefile内に複数のtargetがあるとき、単にmakeとすると一番最初のtargetが生成される。

さまざまなルール

Explicit Rules

先に書いたようなもの。依存関係をベタ書きする。

```
hello: hello.c sub.c sub.h
    gcc hello.c sub.c -o hello
```



プロフィール



id:minus9d

プログラミングや画像処理・機械学習の練習帳です。C/C++, Pythonが好きです。

読者になる 21

検索

ブログ内検索

最新記事

[ピタゴラス数を無限に生成する](#)

[PyInstallerでPythonスクリプトをexe化](#)

ワイルドカード

シェルでよく使う*などがmakefileでも使える。

```
prog: *.c
    $(CC) -o $@ $^
```

\$@, \$^は自動変数。ここでは\$@はprog, \$^は*.cを指している。自動変数については後でもう一度出てくる。

Phony Targets

Phonyとは「偽の」という意味。以下のclearのように、ファイルが生成されるわけではないターゲットのことをPhony Targetという。

```
clean:
    rm -f *.o lexer.c
```

cleanというファイルを作りたいわけではなく、cleanという動作をしたいのだ、ということを明示的に表すために、.PHONYを使う。

```
.PHONY: clean
clean:
    rm -f *.o lexer.c
```

自動変数

先ほど出てきた\$@, \$^などを自動変数という。以下のルールを例にとって考える。

```
hello: hello.c sub.c sub.c sub.c
    gcc hello.c sub.c -o hello
```

実行ファイルhelloはhello.c, sub.cに依存している。例のために、prerequisitesの欄にsub.cを何度も書いている。

このとき、自動変数には以下の値が入る。

自動変数*	意味*	今回の例での値
\$@	ターゲット	hello
\$^	prerequisites、ただし重複分は除く	hello.c sub.c
\$?	prerequisitesのうちターゲットより新しいもの	場合による
\$<	prerequisitesのうち最初に書かれたもの1つ	hello.c
\$+	prerequisites、重複分もそのまま	hello.c sub.c sub.c sub.c

自動変数に入る値を確かめるために、@printfを使ってプリントデバッグすることもできる。(記号の正しいエスケープ方法がわからないので、苦肉の策で半角スペースを挿入している)

```
hello: hello.c sub.c sub.c sub.c
    @printf "$$ @ = $@ \n"
    @printf "$$ ^ = $^ \n"
    @printf "$$ ? = $? \n"
    @printf "$$ < = $< \n"
    @printf "$$ + = $+ \n"
    gcc hello.c sub.c -o hello
```

PythonスクリプトをWindowsのexeにする方法 (調査中)

Python3では変数名に日本語が使える

matplotlibで軸の値が小数になったりオフセット表現になったりするのを止める方法

人気記事

B エントリー

pythonでcsvを読む方法 - 標準ライブラリ, pandas, numpy - minus9d's d...
8users

matplotlibをオブジェクト指向スタイルで使う その2 - minus9d's d...
5users

EmacsのflymakeにてC++11の文法を自動チェックさせる - minus9d's di...
6users

matplotlibをオブジェクト指向スタイルで使う - minus9d's diary
6users

Pythonで、文字列の一部の文字を変更する - minus9d's diary
4users

閉形式(closed-form)とは - minus9d's diary
4users

OpenCVを使うならC++かPythonか? - minus9d's diary
3users

C++11のラムダ関数の簡単なまとめ - minus9d's diary
3users

C++11で数字 → 文字列は std::to_string()、文字列 → 数字は std::stoi()...
5users

はてなブログでtex記法を使うときのメモ - minus9d's diary
6users

月別アーカイブ

- ▶ 2016 (12)
- ▶ 2015 (46)
- ▼ 2014 (100)

2014 / 12 (11)

2014 / 11 (8)

2014 / 10 (5)

2014 / 9 (6)

2014 / 8 (9)

2014 / 7 (7)

2014 / 6 (7)

2014 / 5 (8)

2014 / 4 (8)

2014 / 3 (11)

2014 / 2 (15)

2014 / 1 (5)
- ▶ 2013 (75)
- ▶ 2012 (38)
- ▶ 2011 (23)
- ▶ 2010 (14)

他に\$*, %%という自動変数もあるが未調査。

VPATH, vpath

ソースコードの存在する場所を指定するのにVPATHを使う。

```
VPATH = src
```

例えば以下のようなディレクトリ構造のとき、

```
|makefile
|src/
| |main.c
| |include/
| | |header.h
```

makefileを以下のように書ける。

```
VPATH = src
CPPFLAGS = -I include
main: main.c
    $(CC) $(CFLAGS) $^ -o $@
```

vpathを使うと、VPATHを使うよりも柔軟に場所を指定できる。

```
vpath %.h include
vpath %.c src
CPPFLAGS = -I include

main: main.c
    $(CC) $(CPPFLAGS) $^ -o $@
```

上の例では、.hファイルはincludeフォルダに、.cファイルはsrcフォルダにあることを明示的に表している。ただし、GCCにこの情報は伝わらないので、GCCには-Iでヘッダファイルの場所を教える必要は依然として残る。(参考:[c++ - Makefile vpath not working for header files - Stack Overflow](#))

パターンルール

明示的なルールを書かない場合、自動的に暗黙的なルールが適用される。例えば以下の例では、実行ファイルhelloはhello.oに依存することは表しているが、どうやって実行ファイルhelloを生成するのかについては明示的に表していない。

```
hello: hello.o
hello.o: hello.c
```

この場合、makeは、makeが組み込みで用意している暗黙的なルールを呼び出して実行ファイルhelloを生成する。

```
$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

同様に、.cから.oへのコンパイルには以下の暗黙的なルールが使われる。

▶ 2009 (6)

カテゴリー

[math](#) (25)

[programming contest](#) (3)

[python](#) (52)

[emacs](#) (8)

[english](#) (5)

[C](#) (1)

[Windows](#) (12)

[linux](#) (23)

[C++](#) (59)

[zsh](#) (5)

[android](#) (4)

[google](#) (5)

[cygwin](#) (2)

[hatena](#) (4)

[machine learning](#) (4)

[雑文](#) (9)

[others](#) (7)

[visual studio](#) (11)

[git](#) (5)

[opencv](#) (16)

[mac](#) (9)

[octave](#) (5)

[tex](#) (3)

[cv](#) (2)

[matlab](#) (2)

[gadget](#) (1)

[network](#) (4)

[unicode](#) (1)

[programming](#) (10)

[software](#) (3)

[gmail](#) (1)

[ubuntu](#) (1)

[twitter](#) (1)

[book](#) (2)

```
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c -o $@
```

ちなみに、.ccから.oへのコンパイルには以下の暗黙的なルールが使われる。

```
$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c -o $@
```

CPPFLAGSの「CPP」は、C++のためのフラグという意味ではなく、プリプロセッサのためのフラグという意味。普通CPPFLAGSにはインクルードパスを指定する。C++のためのフラグはCPPFLAGSではなくCXXFLAGSなので間違えないこと。

これらの変数に何を入れればよいか、についてはMakefileの書き方に関する備忘録 その3 - minus9dの日記で調査した。

他の暗黙的なルールについては、コマンドラインで

```
$ make --print-data-base (or -p)
```

と打てば調べることができる。(上で書いた暗黙的なルールも、このコマンドで調べた)

suffix rule

suffix ruleとは、暗黙的なルールを定義するときを使うルールのこと。例は以下。

```
.c.o:
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

.oファイルは、同じ語幹を持つ.cファイルに依存するという意味。

これは以下のpattern ruleと等価。

```
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

実行ファイルのように拡張子がないファイルに関するsuffix ruleは以下のように書ける。

```
.p:
    $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

これは以下のpattern ruleと等価。

```
%.p:
    $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

built-inされているsuffix ruleが悪さをする場合は、空のルールを書いて定義を上書きする。

```
%.o: %.l
%.c: %.l
```

suffix ruleは古い書き方で、見た目にも分かりにくい(依存関係の順番がpattern ruleと逆になっている)。GNU makeを使う限りにおいては、suffix ruleではなくpattern ruleを使う方がよいらしい。

[powershell \(2\)](#)

[google code jam \(1\)](#)

[study \(2\)](#)

[google app engine \(1\)](#)

[voa \(5\)](#)

[web \(1\)](#)

[excel \(2\)](#)

[iPhone \(1\)](#)


[subversion \(1\)](#)

[eclipse \(1\)](#)

最近のコメント

 id:minus9d [matplotlibで軸の値が小数になっ...](#) (135日前)

 id:kochory [matplotlibで軸の値が小数になっ...](#) (135日前)

 id:minus9d [matplotlibで軸の値が小数になっ...](#) (136日前)

 id:kochory [matplotlibで軸の値が小数になっ...](#) (136日前)

 id:minus9d [C++11のtupleをvectorに突...](#) (1年前)

Amazon.co.jpアソシエイト

依存関係の自動生成

C言語の場合、ソースファイルは多数のヘッダファイルに依存するのが普通である。その依存関係を手で保守するのは骨が折れる。そこで依存関係を自動生成することを考える。

例えば以下のようなディレクトリ構造を考える。

```
├─makefile
├─src/
│  ├─main.c
│  ├─sub1.c
│  └─sub2.c
└─include/
   ├─header.h
   ├─sub1.h
   └─sub2.h
```

ここで各ソースファイルの依存関係は以下のようにとする。

- main.cはheader.h, sub1.h, sub2.hに依存
- sub1.cはheader.h, sub1.hに依存
- sub2.cはsub2.hに依存

結論から言うと、以下のようなMakefileを書けばよい。

```
vpath %.h include
vpath %.c src
CPPFLAGS = -I include

SOURCES = \
    main.c \
    sub1.c \
    sub2.c

# .cを.oに置換。OBJECTS = main.o sub1.o sub2.o となる
OBJECTS = $(subst .c, .o, $(SOURCES))

# 実行ファイルmainを生成するルール
# 暗黙のルールを利用
main: $(OBJECTS)

# .dファイルを読み込む
-include $(subst .c, .d, $(SOURCES))

# .cファイルを解析して、.cが依存しているヘッダファイルを.dファイルに書き出す
%.d: %.c
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$

clean:
    rm *.exe *.o *.d
```

このMakefileでは、GCCの-Mオプションがキーポイントになる。GCCで-Mオプションを使用してソースファイルを引数にとると、そのソースファイルがincludeしているヘッダファイルが自動的に列挙

される。このMakefileではその機能を利用して、あるソースファイルの依存関係を.dファイルに一時保存する。そして.dファイルをincludeしている。

なお、Makefile中の\$\$\$\$というの、エスケープされて\$\$、つまりシェルのプロセス番号に置換される。このようにして.d(プロセス番号)という一時ファイルが作られる。この一時ファイルはsedによって利用された後消去される。

make実行時の便利オプション

- make --just-print (or -n): makeしたときに実行されるコマンドを表示だけする
- make CPPFLAGS=-DDEBUG: makefile内の変数CPPFLAGSを-DDEBUGで上書きする

続き

Makefileの書き方に関する備忘録 その2 - minus9dの日記

minus9d 2年前



12

ブックマーク

0

シェア

list

ツイート

1

G+1



« Makefileの書き方に関する備忘録
その2

Googleドライブのスプレッドシートの
日付... »

minus9d

Powered by Hatena Blog