

Makefileの関数

 C++ 2349
  Android 6033
  Makefile 79













(/chibi929)chibi929 (/chibi929)が2014/12/24に投稿(2015/03/04に編集) •

編集履歴(3) (/chibi929/items/b8c5f36434d5d3fbfa4a/revisions) • 問題がある投稿を報告する

 84
 ストック

 0
 コメント

 ストック

 (/kiris)
  (/choplin)
  (/masayuko@github)
  (/suzuki)
  (/kimukou)
  (/kaneshin)
  (/shimacpyon)
  (/futoase)
  (/arexsam)
  (/mumoshu)
 ... (/chibi929/items/b8c5f36434d5d3fbfa4a/stockers)

🕒 この記事は最終更新日から1年以上が経過しています。

自分用にずっとまとめようと思って、下書き保存して温めていたMakefile関連です。
 C++用のビルドからAndroid用のビルドまでMakefileを大活用しているが、
 使う機会が少ないのでMakefileの関数はどうも慣れない&上手く活用できない。
 そして毎回調べる。

ということで、
 実用的なものから、今後使うことはないだろう的なものまで、
 リファクタをする時のために調べた結果をまとめておきたい。

英語が読めない自分が英語のドキュメントを読んだりして、
 自分なりの解釈でまとめたので誤りがありましたらごめんなさい。
 一応、ドキュメントを見ながら全部載せしたつもり。
<http://www.gnu.org/software/make/manual/> (<http://www.gnu.org/software/make/manual/>)

ちなみに、makeのバージョンは、

```
$ make --version
GNU Make 3.81
```

4.x系もリリースされているんですね...

文字列関連の関数

filter

TEXT内からPATTERNの文字列に一致する要素を取得する。
.cppファイルのみを取得したいときなどに使える。

使い方

```
$(filter PATTERN...,TEXT)
```

実行例

```
VAR := hoge.h hoge.cpp hogera.h hogera.cpp

.PHONY: all
all:
    @echo "$(filter %.cpp,$(VAR))"
```

結果

```
$ make
hoge.cpp hogera.cpp
```

filter-out

TEXT内からPATTERNの文字列に一致しない要素を取得する。
filterの逆バージョン。

使い方

```
$(filter-out PATTERN...,TEXT)
```

実行例

```
VAR := hoge.h hoge.cpp hogera.h hogera.cpp

.PHONY: all
all:
    @echo "$(filter-out %.cpp,$(VAR))"
```

結果

```
$ make  
hoge.h hogera.h
```

findstring

IN内にFINDの文字列があるかどうかを確認する

使い方

```
$(findstring FIND,IN)
```

実行例

```
VAR := abc def ghi  
  
.PHONY: all  
all:  
    @echo "$(findstring bc,$(VAR))"
```

結果

```
$ make  
bc
```

firstword

NAMES内の最初の要素を取得する。

使い方

```
$(firstword NAMES...)
```

実行例

```
VAR := abc def ghi  
  
.PHONY: all  
all:  
    @echo "$(firstword $(VAR))"
```

結果

```
$ make  
abc
```

lastword

NAMES内の最後の要素を取得する。

使い方

```
$(lastword NAMES...)
```

実行例

```
VAR := abc def ghi  
  
.PHONY: all  
all:  
    @echo "$(lastword $(VAR))"
```

結果

```
$ make  
ghi
```

patsubst

TEXT内からPATTERNにマッチしたものをREPLACEMENTに置き換える。

※substのPATTERNが使えるバージョン。

使い方

```
$(patsubst PATTERN,REPLACEMENT,TEXT)
```

実行例

```
VAR := hoge.cpp hogera.cpp  
  
.PHONY: all  
all:  
    @echo "$(patsubst %.cpp,%.o,$(VAR))"
```

結果

```
$ make  
hoge.o hogera.o
```

sort

LISTの各要素をソートする。

使い方

```
$(sort LIST)
```

実行例

```
VAR := hoge foo hogera fuga  
  
.PHONY: all  
all:  
    @echo "$$(sort $(VAR))"
```

結果

```
$ make  
foo fuga hoge hogera
```

strip

STRINGの無駄な空白を取り除く。

使い方

```
$(strip STRING)
```

実行例

```
VAR := a b    c    d  
  
.PHONY: all  
all:  
    @echo "$$(strip $(VAR))"
```

結果

```
$ make  
a b c d
```

subst

TEXT内のFROMをTOに置き換える。
※patsubstのPATTERNが使えないバージョン。

使い方

```
$(subst FROM,TO,TEXT)
```

実行例

```
VAR := hoge.cpp hogera.cpp  
  
.PHONY: all  
all:  
    @echo "$(subst .cpp, .o, $(VAR))"
```

結果

```
$ make  
hoge.o hogera.o
```

word

TEXT内のN番目の要素を取得する。
※ $0 < N$

使い方

```
$(word N,TEXT)
```

実行例

```
VAR := abc def ghi  
  
.PHONY: all  
all:  
    @echo "$(word 1, $(VAR))"
```

結果

```
$ make
abc
```

wordlist

TEXT内のSからEまでの要素を取得する。
※wordのlistバージョン。

使い方

```
$(wordlist S,E,TEXT)
```

実行例

```
VAR := abc def ghi

.PHONY: all
all:
    @echo "$(wordlist 1,2,$(VAR))"
```

結果

```
$ make
abc def
```

words

TEXTの要素数を取得する

使い方

```
$(words TEXT)
```

実行例

```
VAR := abc def ghi

.PHONY: all
all:
    @echo "$(words $(VAR))"
```

結果

3

ファイル名関連の関数

abspath

NAMESの各要素の絶対パスを取得する。
※ファイルやディレクトリが存在するかどうかはチェックしない。

使い方

```
$(abspath NAMES...)
```

実行例

```
VAR := ../ ../ ../dir Makefile

.PHONY: all
all:
    @echo "$(abspath $(VAR))"
```

結果

```
$ make
/home/chibi /home/chibi/tmp /home/chibi/tmp/dir /home/chibi/tmp/Makefile
```

addprefix

NAMESの各要素の接頭辞としてPREFIXを追加する。
※-Iとかを付けるときなどに使える。

使い方

```
$(addprefix PREFIX, NAMES...)
```

実行例


```
VAR := ./include ./dir/include

.PHONY: all
all:
    @echo "$$(addprefix -I,$(VAR))"
```

結果

```
$ make
-I./include -I./dir/include
```

addsuffix

NAMESの各要素の接尾辞としてSUFFIXを追加する。

※addprefixの接尾バージョン。

※今のところ使う機会がない。

使い方

```
$(addsuffix SUFFIX,NAMES...)
```

実行例

```
VAR := hoge.cpp hogera.cpp

.PHONY: all
all:
    @echo "$$(addsuffix .bak,$(VAR))"
```

結果

```
$ make
hoge.cpp.bak hogera.cpp.bak
```

basename

NAMESの各要素のファイル名を取得する。

(拡張子を排除する)

使い方

```
$(basename NAMES...)
```

実行例

```
VAR := hoge.txt ./dir/hogera.txt

.PHONY: all
all:
    @echo "$$(basename $(VAR))"
```

結果

```
$ make
hoge ./dir/hogera
```

dir

NAMESの各要素のディレクトリ名を取得する。

使い方

```
$(dir NAMES...)
```

実行例

```
VAR := hoge.txt ./dir/hogera.txt

.PHONY: all
all:
    @echo "$$(dir $(VAR))"
```

結果

```
$ make
./ ./dir/
```

join

LIST1とLIST2の先頭から突合せ結合を行う。

LIST1とLIST2のサイズが異なる場合は、
「空文字列+LIST2」や「LIST1+空文字列」といった形で結合される。

使い方

```
$(join LIST1,LIST2)
```

実行例

```
VAR1 := a b c d
VAR2 := 1 2 3 4 5

.PHONY: all
all:
    @echo "$(join $(VAR1),$(VAR2))"
```

結果

```
$ make
a1 b2 c3 d4 5
```

notdir

NAMESの各要素のファイル名を取得する。

使い方

```
$(notdir NAMES...)
```

実行例

```
VAR := hoge.txt ./dir/hogera.txt

.PHONY: all
all:
    @echo "$(notdir $(VAR))"
```

結果

```
$ make
hoge.txt hogera.txt
```

realpath

NAMESの各要素の絶対パスを取得する。

abspathとは異なり、ファイルやディレクトリが存在するもののみ取得する

使い方

```
$(realpath NAMES...)
```

実行例

```
VAR := ../ ./ ../dir Makefile

.PHONY: all
all:
    @echo "$(realpath $(VAR))"
```

結果

```
$ make
/home/chibi /home/chibi/tmp /home/chibi/tmp/Makefile
```

suffix

NAMESの各要素の拡張子を取得する。

※basenameの拡張子バージョン。

使い方

```
$(suffix NAMES...)
```

実行例

```
VAR := hoge.txt ../dir/hogera.txt

.PHONY: all
all:
    @echo "$(suffix $(VAR))"
```

結果

```
.txt .txt
```

wildcard

ワイルドカードを使って実在するファイルを取得する。

PATTERNは複数個書くこともできる。

使い方

```
$(wildcard PATTERN)
```

実行例

```
VAR := ./tmp/*.cpp ./tmp/*.md

.PHONY: all
all:
    @echo "$(wildcard $(VAR))"
```

結果

```
$ make
./tmp/foo.cpp ./tmp/hoge.cpp ./tmp/README.md
```

条件関連の関数

and

全てのCONDITIONが空でないかどうかを取得する(?)

全てのCONDITIONが空ではない場合に最後のCONDITIONが返ってくる。1つでも空があると結果は空。

※こんな関数があったんだ...使う機会なさそう...

使い方

```
$(and CONDITION1[,CONDITION2[,CONDITION...]])
```

実行例

```
VAR1 := abc
VAR2 := def
VAR3 := ghi

.PHONY: all
all:
    @echo "$(and $(VAR1),$(VAR2),$(VAR3))"
    @echo "$(and $(VAR1),$(EMPTY),$(VAR3))"
    @echo "$(and $(VAR3),$(VAR2),$(VAR1))"
```

結果

```
$ make
ghi
abc
```

if

CONDITIONが空文字列だったらELSE_PARTを
空文字列じゃない場合はTHEN_PARTを取得する。(なのかな?)
※and と or の流れから行くと合ってる気はするのだが...
※条件分岐系は挙動の理解に時間かかった...

使い方

```
$(if CONDITION, THEN_PART[, ELSE_PART])
```

実行例

```
VAR1 := abc  
VAR2 := def  
VAR3 := ghi  
  
.PHONY: all  
all:  
    @echo "$(if $(VAR1),$(VAR2),$(VAR3))"  
    @echo "$(if $(EMPTY),$(VAR2),$(VAR3))"
```

結果

```
$ make  
def  
ghi
```

or

CONDITION全てが空文字列の場合は空を返す。
1つでも値がある場合はCONDITION1が返る。
※andを実行してからorをやったからなんとなく挙動の予想がついてた!
※が...どっちにしろ使わなそう...

使い方

```
$(or CONDITION1[, CONDITION2[, CONDITION...]])
```

実行例

```
VAR1 := abc
VAR2 := def
VAR3 := ghi

.PHONY: all
all:
    @echo "$(or $(VAR1),$(VAR2),$(VAR3))"
    @echo "$(or $(VAR1),$(EMPTY),$(VAR3))"
    @echo "$(or $(EMPTY),$(EMPTY),$(EMPTY))"
    @echo "$(or $(VAR3),$(VAR2),$(VAR1))"
```

結果

```
$ make
abc
abc

ghi
```

デバッグ関連の関数

error

MakefileのエラーとしてTEXTが取得され、かつStopされる。

※if関数とかと組み合わせて使うのかな？

使い方

```
$(error TEXT...)
```

実行例

```
.PHONY: all
all:
    @echo "$(error chibi)"
```

結果

```
Makefile:151: *** chibi. Stop.
```

info

TEXTが取得される。

使い方

```
$(info TEXT...)
```

実行例

```
.PHONY: all
all:
    @echo "$(info chibi)"
```

結果

```
chibi
```

warning

行番号とTEXTが取得される。Stopにはならない。
※if関数とかと組み合わせて使うのかな？

使い方

```
$(warning TEXT...)
```

実行例

```
.PHONY: all
all:
    @echo "$(warning chibi)"
```

結果

```
Makefile:159: chibi
```

その他の関数

call

他の変数やマクロを呼び出すときに使用する。
呼び出し先では、PARAMを引数として1,2などで表す。

使い方

```
$(call VARIABLE,PARAM,...)
```

実行例

```
VAR = "var $1 $2"
define MACRO
    @echo "macro $1 $2"
endef

.PHONY: all
all:
    @echo "$(call VAR,a,b)"
    $(call MACRO,a,b)
```

結果

```
$ make
var a b
macro a b
```

eval

変数を展開した上で、Makefileの構文として定義される。
※実行例ではcall関数と組み合わせてみます。

使い方

```
$(eval TEXT)
```

実行例

```
PROGRAMS := server client
server_OBJS = server_a.o server_b.o server_c.o
client_OBJS = client_a.o client_b.o client_c.o

define template
$(1):
    @echo "$1:"
    @echo "$($1_OBJS)"
endef

.PHONY: all
all: $(PROGRAMS)

$(eval $(call template,server))
$(eval $(call template,client))
```

結果

```
$ make
server:
server_a.o server_b.o server_c.o
client:
client_a.o client_b.o client_c.o

$ make server
server:
server_a.o server_b.o server_c.o

$ make client
client:
client_a.o client_b.o client_c.o
```

flavor

変数の形式(?)を返す。

返される値は以下の通り。

- ・undefined: 未定義
- ・simple: Simply expanded variable
- ・recursive: recursively expanded variable

使い方

```
$(flavor VARIABLE)
```

実行例

```
DEFINED := test
RECURSIVE = test

.PHONY: all
all:
    @echo $(flavor UNDEFINED)
    @echo $(flavor DEFINED)
    @echo $(flavor RECURSIVE)
```

結果

```
undefined
simple
recursive
```

foreach

LISTの要素をVARに分解し、TEXTで展開して実行する。

関数名の通りfor系の関数。

※evalの実行例にforeachを追加してみよう。

使い方

```
$(foreach VAR,LIST,TEXT)
```

実行例

```
PROGRAMS := server client
server_OBJS = server_a.o server_b.o server_c.o
client_OBJS = client_a.o client_b.o client_c.o

define template
$(1):
    @echo "$1:"
    @echo "$($1_OBJS)"
endef

.PHONY: all
all: $(PROGRAMS)

$(foreach prog,$(PROGRAMS),$(eval $(call template,$(prog))))
```

結果

```
server:
server_a.o server_b.o server_c.o
client:
client_a.o client_b.o client_c.o

$ make server
server:
server_a.o server_b.o server_c.o

$ make client
client:
client_a.o client_b.o client_c.o
```

origin

どこで定義された変数なのかが取得される。

取得される文字列は以下の通り。

- undefined: 未定義
- default: CC や MAKE など
- environment: PATHなどの環境変数
- environment override:
- file: Makefile内で定義されたもの
- command line: コマンドラインで定義されたもの
- override: Makefile内でオーバーライドされたもの
- automatic: 自動変数

使い方

```
$(origin VARIABLE)
```

実行例

```
DEFINED := test
OVERRIDE := test
override OVERRIDE := test

.PHONY: all
all:
    @echo $(origin UNDEFINED)
    @echo $(origin DEFINED)
    @echo $(origin PATH)
    @echo $(origin OVERRIDE)
    @echo $(origin @)
    @echo $(origin MAKE)
```

結果

```
undefined
file
environment
override
automatic
default
```

shell

shellを叩く。

lsコマンドとかでwildcardの代わりなどもできる。

使い方

```
$(shell COMMAND)
```

実行例

```
.PHONY: all
all:
    @echo "$(shell ls)"
```

結果

```
Makefile tmp
```

value

変数を参照する関数(?)

\$(VARIABLE)と同じ結果が返ってくる。

使い方

```
$(value VARIABLE)
```

実行例

```
VAR := test

.PHONY: all
all:
    @echo "$(VAR)"
    @echo "$(value VAR)"
```

結果

```
test
test
```

4.x系で追加された関数

file

ファイル書き込みを行える。

上書きと追記の2種類のモードがある。

※ついでにこの関数の実験中にわかったことが、

※shell関数を使うと改行コードはスペースに変換されてMakefileで扱う形になるということ。

※ダブルクォーテーションも無くなった。

使い方

```
$(file OP FILENAME[,TEXT])
```

実行例

```
.PHONY: all
all:
    @echo "all"
    $(file > test.txt, "A")
    @echo "$(shell cat test.txt)"
    $(file >> test.txt, "B")
    @echo "$(shell cat test.txt)"
    cat test.txt
```

結果

```
A
A B
cat test.txt
"A"
"B"
```

guile

GNU Guileが統合されたためMakefileからguileが使えるようになっているとのこと。

以下を参考にチャレンジ

http://qiita.com/ka_/items/4aa58cd180e535767977 (http://qiita.com/ka_/items/4aa58cd180e535767977)

GNU Guile自体、
全然わからないのでstring-appendの真似をする。

使い方

```
$(guile SCHEME)
```

実行例

```
all:
    @echo $(guile (string-append "Hello" "World!"))
```

結果

```
HelloWorld!
```

🔗 この記事は以下の記事からリンクされています



makefile > あるフォルダの.cファイル名を取得 > **wildcard**使用 (/7of9/items/38e439e36e5eb0bde714) からリンク 1
年以上前

あなたもコメントしてみませんか :)

ユーザー登録(無料) (/signup?redirect_to=%2Fchibi929%2Fitems%2Fb8c5f36434d5d3fbfa4a%23comments)

すでにアカウントを持っている方はログイン (/login?
redirect_to=%2Fchibi929%2Fitems%2Fb8c5f36434d5d3fbfa4a%23comments)