

Makefile の書き方 [↑]

- [Makefile の書き方](#)
 - [はじめに](#)
 - [make って何？](#)
 - [make 色々](#)
 - [make で Hello World!](#)
 - [Makefile の基本文法：依存関係行](#)
 - [依存関係行の応用その一](#)
 - [依存関係行の応用その二](#)
 - [依存関係行の応用その三](#)
 - [マクロ](#)
 - [内部マクロ](#)
 - [サフィックスルール](#)
 - [さらなる応用](#)
 - [分割 Makefile](#)
 - [C 言語のヘッダーファイルの依存関係の自動解決](#)
 - [その他](#)

[↑](#)

はじめに [↑]

ここでは、Makefile の中でも GNU make にかぎって説明します。

[↑](#)

make って何？ [↑]

ソースファイルを分割して大規模なプログラムを作成していると、コマンドでコンパイルするのが面倒です。また、一部のソースファイルを書き換えただけなのに全部をコンパイルし直すのは時間の無駄です。

そんな問題を解決するのが make です。Makefile と呼ばれるテキストファイルに必要なファイルと各ファイルのコンパイルのコマンド、ファイル間の依存関係を記します。そして、“make”というコマンドを実行するだけで、自動的にコマンドを実行してコンパイルしてくれます。これだけではスクリプトと大差がないのですが、make は Makefile に記された依存関係に基づいて更新されたファイルの内関連のあるものだけを更新することで、コンパイル時間を短くします。

make は特定のプログラミング言語に依存したものではありません。C 言語のソースファイルのコンパイルにも使えますし、Verilog-HDL のシミュレーションにも使えますし、TeX のコンパイルにも使えます。

[↑](#)

make 色々 [↑]

実は make には色々種類があります。主なものをあげると以下の通りです。

- Microsoft nmake (Windows)
- Borland make (Windows)
- GNU make (Windows, UNIX 系)
- Solaris make (Solaris)

“nmake”は Microsoft の Visual C++ に搭載されている make です。Windows 用です。ただし、Visual C++ にはプロジェクト管理機能があるので nmake の世話になることはほぼないでしょう。世話になるのは、公開されているソースファイルをコンパイルする場合だけでしょう。

Borland の“make”は Borland C++ Compiler などに入ってます。Windows 用です。Borland C++ Compiler がフリーとなったことで、公開されているソースファイルに意外と Borland make 用の Makefile がついていたります。

[GNU の“make”](#)は UNIX 系では標準的な make です。Windows 用の GNU make には、[Cygwin](#) に付属のものか、[MinGW](#) のものがあります。ここでは、GNU make の使い方を説明します。

Solaris の“make”は Solaris に付属する make です。Solaris 用のソフトの中には GNU make ではうまくいかないものがあるそうで、そんなときに Solaris make を使うそうです。Solaris make をはじめとする、Solaris 付属の開発環境は、/usr/ccs/bin にあります。

[↑](#)

make で Hello World! [↑]

それではメイクファイルを書いてみましょう。ここではソースファイルとしてコンパイル方法でも使った C 言語の Hello World! を使用します。

```
/* hello.c */
#include <stdio.h>
```

```
int main(int argc, char* argv[]) {  
    printf("Hello World!¥r¥n");  
    return 0;  
}
```

同じディレクトリに“Makefile”というファイルを作成して、以下の内容を記述してください。

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

三行目の先頭は空白文字ではなくてタブ文字(Tab)なので注意してください！ 一行目はコメントです。“#”と書くとその行の“#”以降の文字列はコメントとなります。コマンドから make を実行すると以下のようにコンパイルしてくれます。

```
$ make  
gcc -o hello hello.c
```

このあともう一度 make を実行してみましょう。

```
$ make  
make: `hello' は更新済みです
```

hello.c が更新されていない(hello の方が hello.c よりも日付が新しい)のでコンパイルされません。こんな感じで、必要な部分のみコンパイルしてくれます。

Makefile のファイル名を指定する場合は

```
$ make -f Makefile
```

とします。ファイル名を指定しない場合は、“GNUmakefile”、“makefile”、“Makefile”の順に検索します。

Makefile の基本文法: 依存関係行 [±]

Makefile の基本的な構文は依存関係を表す依存関係行です。依存関係行はこんな感じです。

```
ターゲット名: 依存ファイル名 1 依存ファイル名 2 依存ファイル名 3
    コマンド行 1
    コマンド行 2
    コマンド行 3
```

ターゲット名は一般的に生成されるファイルのファイル名にします(そうでない場合については後述します)。

ターゲット名の後に":"を書いて、その後にスペース区切りで依存するファイルのファイル名を記述します。これらのファイルのうちどれか一つでも更新されるとコマンドが実行されます。

ターゲット名を指定して make を実行する場合は

```
$ make ターゲット名
```

とします。ターゲット名を省略すると、Makefile の中で先頭のターゲットが実行されます。

ターゲット名から始まる行の次の行から実行するコマンドを記述します。コマンドを記述する場合は**必ず先頭にタブ文字を入れる**必要があります。

例として、C 言語の分割コンパイルをしてみましょう。分割コンパイル用に以下のファイルを用意します。

```
/* hello.c */
#include <stdio.h>

void edajima(void);

int main(int argc, char* argv[]) {
    edajima();
    return 0;
}
```

```

/* edajima.c */
#include <stdio.h>

void edajima(void);

void edajima(void) {
    printf("わしが男塾塾長 江田島平八である!!¥r¥n");
}

```

メイクファイルはこんな感じです。

```

# Makefile
hello: hello.c edajima.c
    gcc -o hello hello.c edajima.c

```

hello.c または edajami.c のいずれかを更新するとコンパイルし直してくれます。しかし、このままでは更新されていないファイルもコンパイルし直されてしまうので、少し変更します。

```

# Makefile
hello: hello.o edajima.o
    gcc -o hello hello.o edajima.o

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

edajima.o: edajima.c
    gcc -c edajima.c

```

こうして make を実行すると、

```

$ make
gcc -c hello.c
gcc -c edajima.c
gcc -o hello hello.o edajima.o

```

となります。ここで、edajima.c を書き換えます。

```
/* edajima.c */
#include <stdio.h>

void edajima(void);

void edajima(void) {
    printf("わしが男塾塾長 江田島平八である!!¥r¥n");
    printf("以上!¥r¥n");
}
```

そして make を実行すると、

```
$ make
gcc -c edajima.c
gcc -o hello hello.o edajima.o
```

となって、edajima.o だけが更新されます。



依存関係行の応用その一 [±]

依存関係行を使った応用について説明します。プログラムをコンパイルすると中間ファイルなどができていちいち削除するのが面倒です。そこで、Makefile に以下の行をつけたします。

```
# Makefile
hello: hello.o edajima.o
    gcc -o hello hello.o edajima.o

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

edajima.o: edajima.c
    gcc -c edajima.c
```

```
clean:
    rm -f hello hello.o edajima.o
```

こうしてコマンドで以下のように実行します。

```
$ make clean
rm -f hello hello.o edajima.o
```

不要なファイルをすべて削除してくれます。“clean”は依存するファイルがなく、clean というファイル生成するわけでもなく、コマンドを実行するだけです。このようなターゲットのことを“phony target”と呼びます。phony ターゲットを使用する場合、ターゲット名と同じ名前のファイルがあると変なことになります。

```
$ touch clean
$ make clean
make: `clean' は更新済みです
```

これをさけるためには Makefile を以下のように書き換えます。

```
.PHONY: clean
clean:
    rm -f hello hello.o edajima.o
```

こうすると clean というファイルが存在していても問題ありません。

[↑](#)

依存関係行の応用その二 [↑](#)

もう一つの応用は、複数のプログラムを作成するときに役に立ちます。ここでは以下のソースファイルを追加します。

```
/* raiden.c */
#include <stdio.h>

int main(int argc, char* argv) {
```

```
printf("男の勝負に言葉はいらん¥r¥n");  
printf("ただそれだけのこと……………!!¥r¥n");  
return 0;  
}
```

そして Makefile を以下のようにします。

```
# Makefile  
hello: hello.o edajima.o  
    gcc -o hello hello.o edajima.o  
  
hello.o: hello.c  
    gcc -c hello.c  
  
edajima.o: edajima.c  
    gcc -c edajima.c  
  
raiden: raiden.o  
    gcc -o raiden raiden.o  
  
raiden.o: raiden.c  
    gcc -c raiden.c  
  
.PHONY: clean  
clean:  
    rm -f hello hello.o edajima.o raiden raiden.o
```

これで hello と raiden を作ろうとすると、

```
$ make hello  
$ make raiden
```

となり、面倒です。そこで、ダミーの依存関係行を追加します。

```
# Makefile
```



```
.PHONY: all
all: hello raiden

hello: hello.o edajima.o
    gcc -o hello hello.o edajima.o

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

edajima.o: edajima.c
    gcc -c edajima.c

raiden: raiden.o
    gcc -o raiden raiden.o

raiden.o: raiden.c
    gcc -c raiden.c

.PHONY: clean
clean:
    rm -f hello hello.o edajima.o raiden raiden.o
```

先頭に追加した“all”がミソです。これで make を実行すると、

```
$ make
gcc -c hello.c
gcc -c edajima.c
gcc -o hello hello.o edajima.o
gcc -c raiden.c
gcc -o raiden raiden.o
```

となってめでたく二つのプログラムを一度に作成することができました。

[↑](#)

C 言語ではコンパイルしないけどソースファイルにインクルードされるヘッダーファイルが存在します。ヘッダーファイルが更新されたときにソースファイルをコンパイルし直すにはどうしたらよいのでしょうか？

この問題を解決するには、同じターゲット名の依存関係行を追加します。例えば以下のようなファイルを用意します。

```
/* jaki.h */
char serifu[] = {"聖紆塵 貴様の命 この邪鬼とともにある"};
/* jaki.c */
#include <stdio.h>
#include "jaki.h"

int main(int argc, char* argv[]) {
    printf("%s¥r¥n", serifu);
    return 0;
}

# Makefile
.PHONY: all
all: jaki

jaki: jaki.o
    gcc -o jaki jaki.o

jaki.o: jaki.c
    gcc -c jaki.c

.PHONY: clean
clean:
    rm -rf jaki jaki.o
```

make を実行すると

```
$ make
gcc -c jaki.c
gcc -o jaki jaki.o
```

となります。ここで“jaki.h”を書き換えます。

```
/* jaki.h */  
char serifu[] = {"す すまん 貴様等の期待と信頼を裏切った…………"};
```

そして、make を実行すると

```
$ make  
make: `all' に対して行うべき事はありません。
```

といって更新してくれません。そこで、以下のように“Makefile”を書き換えます。

```
# Makefile  
.PHONY: all  
all: jaki  
  
jaki: jaki.o  
    gcc -o jaki jaki.o  
  
jaki.o: jaki.c  
    gcc -c jaki.c  
  
jaki.o: jaki.h  
  
.PHONY: clean  
clean:  
    rm -rf jaki jaki.o
```

“jaki.o: jaki.h”という行がポイントです。そして、make を実行すると

```
$ make  
gcc -c jaki.c  
gcc -o jaki jaki.o
```

となり、ちゃんと更新されます。

マクロ [↑]

ここから少し難しくなります。これまでは Makefile にファイル名やコマンド名を直接書いていました。しかし、マクロを使うと直接書かなくてすみ、他への流用などが容易となります。マクロを定義するには以下のようにします。

```
マクロ名 = 文字列
```

マクロを参照するには、

```
$(マクロ名)
```

または

```
${マクロ名}
```

とします。実際に使ってみるとこんな感じです。

```
# Makefile
objs = hello.o edajima.o

hello: $(objs)
    gcc -o hello $(objs)

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

edajima.o: edajima.c
    gcc -c edajima.c

.PHONY: clean
clean:
    rm -f hello $(objs)
```

ここでは、オブジェクトファイル名を“objs”というマクロとして定義しています。“\$(objs)”は“hello.o edajima.o”に置換されます。

GNU make では、定義済マクロとして以下のものがあります。

マクロ名	文字列	説明
AR	ar	アーカイブユーティリティ
AS	as	アセンブラ
CC	cc	C コンパイラ
CXX	g++	C++コンパイラ
CO	co	RCS ファイルからリビジョンをチェックアウトする
CPP	\$(CC) -E	C プリプロセッサ
FC	f77	Fortran コンパイラ
GET	get	知らね
LEX	lex	lex
PC	pc	Pascal コンパイラ
YACC	yacc	yacc
YACCR	yacc -r	知らね
MAKEINFO	makeinfo	Texinfo -> Info
TEX	tex	TeX
TEXI2DVI	texi2dvi	Texinfo -> DVI
WEAVE	weave	知らね
CWEAVE	cweave	知らね
TANGLE	tangle	知らね
CTANGLE	ctangle	知らね

RM	rm -f	ファイルの削除
----	-------	---------

上記のプログラムの引数用のマクロもあります。

マクロ名	文字列	説明
ARFLAGS	rv	AR の引数
ASFLAGS		AS の引数
CFLAGS		CC の引数
CXXFLAGS		CXX の引数
COFLAGS		CO の引数
CPPFLAGS		CPP の引数
FFLAGS		FC の引数
GFLAGS		GET の引数
LDFLAGS		リンカ ld の引数
LFLAGS		LEX の引数
PFLAGS		PC の引数
RFLAGS		知らね
YFLAGS		YACC の引数

これらのマクロは再定義可能です。例えば、こんな感じです。

```
# Makefile
objs = hello.o edajima.o
CC = gcc

hello: $(objs)
    $(CC) -o hello $(objs)

hello.o: hello.c
```

```
$(CC) -c hello.c

edajima.o: edajima.c
    $(CC) -c edajima.c

.PHONY: clean
clean:
    $(RM) hello $(objs)
```

ここでは“CC”というマクロを“gcc”という文字列で再定義しています。また、“RM”というマクロをそのまま使用しています。



内部マクロ [±]

前述のマクロは単純に文字列に置換するだけでしたが、内部マクロはもう少し複雑になります。例えば、こんな感じの内部マクロがあります。

```
hello: $(objs)
    $(CC) -o $@ $(objs)
```

ここでは“\$@”という内部マクロを使用しています。これはターゲット名を表すものです。そのため上記の記述は、

```
hello: $(objs)
    $(CC) -o hello $(objs)
```

と解釈されます。

また、以下のものもあります。

```
hello.o: hello.c
    $(CC) -c $<
```

ここでは“\$<”という内部マクロを使用しています。これは依存ファイルの先頭のファイル名を表すものです。そのため上記の記述は、

```
hello.o: hello.c
    $(CC) -c hello.c
```

と解釈されます。依存ファイル名のリストを表す“\$^”という内部マクロもあります。

内部マクロをまとめるとこんな感じです。

内部マクロ名	説明
\$@	ターゲット名
\$%	ターゲットメンバ名(ターゲット名が“edajima.a(momo.o)”の場合、\$@は“edajima.a”で、\$%は“momo.o”)
\$<	依存ファイルの先頭のファイル名
\$?	依存ファイルの内、ターゲットより新しいファイルのリスト
\$^	依存ファイルのリスト
\$+	わかんね
\$*	わかんね

マクロと内部マクロを駆使すると、Makefile はこんな感じになります。

```
# Makefile
program = hello
objs = hello.o edajima.o
CC = gcc
CFLAGS = -g -Wall

$(program): $(objs)
    $(CC) -o $(program) $^
```



```
hello.o: hello.c
    $(CC) $(CFLAGS) -c $<

edajima.o: edajima.c
    $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
    $(RM) $(program) $(objs)
```

[↑](#)

サフィックスルール [±]

サフィックスルールとは、ファイル名の拡張子(サフィックス)ごとにルールを定義するものです。例えばこんな感じです。

```
.SUFFIXES: .o .c

.c.o:
    $(CC) $(CFLAGS) -c $<
```

“.SUFFIXES”は依存関係行と同じ形ですが、意味が違います。サフィックスルールを適用する拡張子のリストを書きます。

“.c.o”がサフィックスルールとなっており、拡張子が“.o”のファイルは拡張子を“.c”変えたファイルに依存していることを表します。変換方法はコマンドで表されています。例えば、ターゲット名が“hoge.o”ならば make はこのサフィックスルールより“hoge.c”に依存していると判断して、コマンドを実行し“hoge.o”を生成します。

サフィックスルールを用いると、こんな感じで書けます。

```
# Makefile

# プログラム名とオブジェクトファイル名
program = hello
objs = hello.o edajima.o
```

```

# 定義済マクロの再定義
CC = gcc
CFLAGS = -g -Wall

# サフィックスルール適用対象の拡張子の定義
.SUFFIXES: .c .o

# プライマリターゲット
$(program): $(objs)
    $(CC) -o $(program) $^

# サフィックスルール
.c.o:
    $(CC) $(CFLAGS) -c $<

# ファイル削除用ターゲット
.PHONY: clean
clean:
    $(RM) $(program) $(objs)

```

ここまでくると、あとは“program”や“objs”を書き換えるだけでいくらかでも流用ができます。ちなみに、ヘッダーファイルの依存関係だけは自分で記述しなければなりません。例えばこんな感じです。

```

# Makefile

# プログラム名とオブジェクトファイル名
program = jaki
objs = jaki.o

# 定義済マクロの再定義
CC = gcc
CFLAGS = -g -Wall

# サフィックスルール適用対象の拡張子の定義

```

```

.SUFFIXES: .c .o

# プライマリターゲット
$(program): $(objs)
    $(CC) -o $(program) $^

# サフィックスルール
.c.o:
    $(CC) $(CFLAGS) -c $<

# ファイル削除用ターゲット
.PHONY: clean
clean:
    $(RM) $(program) $(objs)

# ヘッダーファイルの依存関係
jaki.o: jaki.h

```

[↑](#)

さらなる応用 [↑](#)

[↑](#)

分割 Makefile [↑](#)

プログラムが複雑になって、ディレクトリごとにソースコードを分けるなどしていくと、一つの Makefile で管理するのは面倒になってきます。そんな時には、Makefile を分割することができます。例えば、subdir というサブディレクトリの中に別の Makefile があるとした場合、カレントディレクトリの Makefile で

```

subsystem:
    cd subdir && $(MAKE)

または

subsystem:
    $(MAKE) -C subdir

```

とします。

[↑](#)

C 言語のヘッダーファイルの依存関係の自動解決 [±]

C 言語でプログラミングしている際に、ソースファイルが増えるとヘッダファイルの依存関係をいちいち記述するのは面倒です。色々な解決方法があるみたいですが、とりあえずこんなん考えてみました。

```
# Makefile

# プログラム名とオブジェクトファイル名
program = jaki
objs = jaki.o

# 定義済マクロの再定義
CC = gcc
CFLAGS = -g -Wall

# サフィックスルール適用対象の拡張子の定義
.SUFFIXES: .c .o

# プライマリターゲット
.PHONY: all
all: depend $(program)

# プログラムの生成ルール
$(program): $(objs)
    $(CC) -o $(program) $^

# サフィックスルール
.c.o:
    $(CC) $(CFLAGS) -c $<

# ファイル削除用ターゲット
.PHONY: clean
```

```

clean:
    $(RM) $(program) $(objs) depend.inc

# ヘッダーファイルの依存関係
.PHONY: depend
depend: $(objs:.o=.c)
    -@ $(RM) depend.inc
    -@ for i in $^; do
        cpp -MM $$i | sed "s/[_a-zA-Z0-9][_a-zA-Z0-9]*\.c//g" >>
        depend.inc;
    done

    -include depend.inc

```

gcc のプリプロセッサである cpp と sed を組み合わせています。cpp は指定したソースファイルの依存関係を make の形式で出力してくれるオプションを持っています。それを使って、全ソースファイルの依存関係を“depend.inc”に出力して、それをインクルードしています。“make depend”とコマンドを実行すれば OK です。また、“all: depend \$(program)”とすることで、make する際に毎回“depend.inc”を作成するようにしています。



その他

GNU make には他にも色々な機能があります。詳しくは [Web のマニュアル](#) をご覧ください。また、make を発展させた、[autoconf](#)、[automake](#)、[libtool](#)、などもあります。これは OS 間の差異を吸収するためのツールです。