make

はじめに

make はプログラム構築のための自動化ツールであり、ソフトウェア開発の現場では今でもよく利用されている。一般的にソフトウェア開発はソースコードの編集からコンパイルやリンク、そして実行ファイルの作成の順で進めていくが、このうち煩雑でミスの出やすいコンパイルから実行ファイルの作成の手作業部分を make は代行する。例えば **Makefile** という名前のファイルに関連するソースファイル名や処理の組み合わせを記述し、make を実行すれば実行ファイルが作成される訳である。このとき、ソースファイルの依存関係を make は理解し、処理を自動的に進めてくれる。それゆえソフトウェア開発でのミスを大幅に減らしてくれる訳である。

GUI によるツールが主流になるにつれて make は無用なものとなりつつあったが、近年では再び CUI が注目され、それに伴い make も見直されてきている。本資料では make の基本的な内容を紹介していく。

1 Makefile と make

1.1 Makefile の基本的な書式

Makefile は複数の**ルール**で構成される。ここでルールとは処理内容と入出力ファイルが明記されたもののことで、筆者は関数のようなものをイメージしている。ルールの書き方は以下の通りである。

Listing 1.1 Rule の書き方

```
target : prereg1, prereg2, ...
        command1
        command2
        ...
```

ターゲットはルールによって生成したいファイルなどであり、実行ファイルなどが当てはまる。**依存ファイル** (もしくは必須項目=prereg) は target を生成する上で必要なファイルのことで、メインファイルやヘッダーファイルなどがここに記述される。ルールを書くうえで依存ファイルは複数あってもよく、依存ファイルが無いことも許される。一方でターゲットは必ず 1 つ以上記述されなければならない。

ターゲットと依存ファイルを指定した後、処理コマンドを下に記載していく。ここで処理コマンドは必ずタブで字下げをしなければならない。ただし、コマンドが 1 つだけかつ短ければ、下記のように記述することもできる。

Listing 1.2 Rule の書き方 2

```
target : prereg1, prereg2, ... ; command
```

以下は main.f90 のソースファイルをコンパイルし、target.out という名の実効ファイルを作成する例である。例えばこの Makefile を main.f90 と同じディレクトリに保存し、そこで「make」と実行すれば target.out が作成される。

Listing 1.3 Makefile 例

```
target.out : main.f90
        gfortran -o target.out main.f90
```

Makefile ではコメントアウトと改行を行うことができる。

- **コメント**：「#」を使えば、それ以降の文字から行末まではコメントアウトされる。
- **改行**：Makefile には、C における「;」のような文の終端を表す文字が存在せず、Python のように改行までの 1 行が一つの処理だと判断される。長いコメントを書くときなどに改行が必要となるが、Makefile は Python と同様にバックスラッシュ (Windows では円マーク) を用いて改行する。バックスラッシュの前後は 1 個の空白に置き換えられる。また、折り返し後の字下げは無視される。例えば Listing1.4 は「abc def ghi」と解釈される。

Listing 1.4 Makefile における改行

```
abc\
        def\
ghi
```

上記の通り行の取り扱いは Python と似ているが、空行の行頭におけるタブ文字の扱いには注意が必要である。タブ文字のない空行なら両者とも無視するが、タブ文字が行頭にある空行の場合、Python はそれでも無視する一方で Makefile ではエラーが生じる。

1.2 make の基礎

make のコマンドラインは以下の通りである。

make [-f *makefile*] [*option*] [*target*]

make には多くのオプションが用意されているが、中でも以下の書式を使うことが多い。

- make：カレントディレクトリの Makefile を読み込む。
- make *target*：Makefile 中の特定のターゲットを指定する。
- make -f *makefile*：-f に続くファイルを Makefile として読み込む。
- make -f *makefile* target：指定のファイルを Makefile として読み込み、その中の target を指定。
- make -C subdir：サブディレクトリ subdir の中で make を実行。

make は Makefile に記述されたルールを基に判断し、ターゲットの作成に必要なコマンドを実行する。しかもルールが上手く記述されていれば、不要な処理を省略する機能もついている。本節では make がどのような論理で処理を実行もしくは回避するのか、そしてどういったときにエラーとなってしまうのか見ていく。

1.2.1 make の木構造的考え方

基本的に make の主目的はターゲットの作成である。ターゲットの作成には依存ファイルが必要で、準備されていれば make はターゲットを作成することができる (Listing1.3)。

Makefile には複数のルールを記述することができる。例えば Listing1.3 は 1 つのコマンドで実行ファイルまで作成しているが、それをコンパイルと実行ファイルの 2 段階に分割した場合を考えよう。このとき、下記のような Makefile が考えられる。

Listing 1.5　複数のターゲットがある Makefile

```
1 target.out : main.o
2         gfortran -o target.out main.o
3
4 main.o : main.f90
5         gfortran -c main.f90
```

一般的なプログラミング言語の場合、処理は行の順に施される。make の場合も同様で、ルールの中の処理は行の順に進む。しかしながら、ルールに関してはそうでない。make は最終的に一つのルールのターゲットを作る。これを最終目的のルールと呼ぶことにしよう。最終目的であるルールは、デフォルト設定ならば Makefile の先頭のルールになる。例えば Listing1.5 を下記のように書き直したとする。この場合は main.o しか作成されない。

Listing 1.6　複数のターゲットがある Makefile2

```
1 main.o : main.f90
2         gfortran -c main.f90
3
4 target.out : main.o
5         gfortran -o target.out main.o
```

Listing1.5 と 1.6 の結果の違いは各ルールの依存関係によって生じた。例えば Listing1.5 の場合、make はまずはじめに target.out を作成しようとするが、そこで main.o が無いことに気付く。すると make は main.o をターゲットとしたルールを Makefile 内から探し、もし在るならば先にそのルールを実行する (ない場合はエラーとなる)。こうして最終目的のルールの依存ファイルがすべて用意できたとき、改めて make はそのルールを実行する。従って Listing1.5 では 2 つのルールが実行された訳である。

一方の Listing1.6 の場合、最終目的のルールは main.f90 をターゲットとしている。これは既に存在するファイルなので、ルールの実行も既に可能である。そのため target.out に関するルールは実行されずに終わる。

このように、make はルールを先頭から順に実行するのではなく、各ルールの依存関係を把握して必要なルールのみ適切な順で実行する。もしくは最終目的のルールを根に持つルールの木構造を構築し、補助的なルールを葉ノードから順に実行するとも言える (このような特性は make の**階層構造**と一般的に呼ばれている)。

デフォルト設定の場合、最終目的のルールは先頭のルームになるが、これを変えるには「make target 名」で実行すればよい。例えば Listing1.6 に対して「make target.out」と実行すれば、Listing1.5 のときと同じ結果が得られる。

1.2.2 ルール実行の判断

make は実行時に各ルールの実行要否をチェックし、不要と判断すればそのルール実行を無視する。この機能のおかげで例えばライブラリを不必要に再コンパイルすることを避け、開発時間を短縮できる。実行要否は主にターゲットの有無と関連するファイルのタイムスタンプで判断される。

- 対象のターゲットファイルが存在しない場合、make はそのルールを実行する。
- 対象のターゲットファイルが既に存在する場合、make はまず初めにターゲットファイルと依存ファイルのタイムスタンプを比較する。ターゲットファイルのタイムスタンプよりも後の時間のタイムスタンプである依存ファイルが存在する場合、つまり既存のターゲットファイルが作成された後に依存ファイルに何か変更があったと思われる場合、そのルールは実行される。一方そうでない場合は実行しない。
- 依存ファイルがないルールの場合、タイムスタンプに関わらずターゲットファイルが既に存在するならば実行しない。

1.3 疑似ターゲット

前節で紹介した make 及び Makefile の特徴を利用して様々なテクニックが提案されている。本節では**疑似ターゲット**と呼ばれるいくつかのテクニックを紹介する。疑似ターゲットは特徴的なルールを用いるが、そのルールのターゲットに興味はない。

1.3.1 複数のルールを実行するための疑似ターゲット

例えば下記のような Makefile を考える。

Listing 1.7 複数のターゲットがある Makefile3

```
1 tag1: ; echo "tag1"
2 tag2: ; echo "tag2"
3 tag3: ; echo "tag3"
```

この場合、make は先頭のルールを最終目的とするため、"tag1"のみ出力される。

すべてのターゲットを実行したい場合は、下記のような疑似ターゲットを作成するとよい。

Listing 1.8 疑似ターゲット all

```
1 all : tag1 tag2 tag3
2
3 tag1: ; echo "tag1"
4 tag2: ; echo "tag2"
5 tag3: ; echo "tag3"
```

この場合は all のターゲットを主目的とする訳だが、tag1 や tag2、並びに tag3 が作成されていないので、まずは all 以外の 3 つのルールを実行することになる。このように、複数のルールを最終目的としたい場合は all のような疑似ターゲットを先頭にもってくるとよい (なお、ターゲットの名前は任意だが慣習的に all とすることが多い)。

1.3.2 ファイルを消すための疑似ターゲット

ソフトウェア開発をしているとき、デバッグのために何度も make を実行することがある。途中でエラーが出力された場合、そこまでの中間ファイルだけが残る。場合によっては全ての中間ファイルを消してから再実行しなければならないこともあるし、それ以外の理由でも Makefile に関わるファイルの削除機能は欲しかったりする。make では、こういった目的のために clean という疑似ターゲットを作成することが多い。下記は clean の例である。

Listing 1.9 疑似ターゲット clean

```
1 test1.txt : dep1.txt
2         cat dep1.txt > test1.txt
3 dep1.txt:
4         echo "dep1" > dep1.txt
5 clean:
6         rm -f dep1.txt test1.txt
```

単に「make」と実行した場合、test1.txt と dep1.txt が作成される。これらファイルを消したい場合は「make clean」と実行すればよい。なお all と同様に、疑似ターゲットの名前は任意だが、一般的に clean と名付けることが多い。

1.3.3 疑似ターゲットの宣言

前節の疑似ターゲット all 及び clean は、ディレクトリ内に all や clean といったファイルが存在しない場合にのみ機能する。もしもこういった名前のファイルがあるならば、場合によってはルールの実行が省略されてしまう。もちろん疑似ターゲットの名前は任意なので、他にないターゲット名をつければ済む話だが、大規模なソフトウェア開発になるにつれ困難になってくる。

疑似ターゲットを宣言するには「**.PHONY**」を使えばよい。以下のように書けばいかなる時も clean は疑似ターゲットとして扱われる。

Listing 1.10 .PHONY

```
1 .PHONY : clean
2 clean: ; rm -f dep1.txt test1.txt
```

2 変数

2.1 変数の代入と参照

make には文字列のための**変数**の機能があり、そのおかげで冗長なコマンドラインが見やすくなったり、修正が容易になったりする。

2.1.1 変数への代入

Listing 2.1 変数への代入

```
1 VAR = echo "Hello"
```

変数の代入には「=」を用いる。例えば Makefile 内で上記のように宣言した場合を考えよう。このとき VAR は変数名であり、それに「echo "Hello"」が代入されたことになる。変数名に使える文字は「:、=、#、空白」以外で、大文字小文字は区別される。慣習的には、全て大文字の変数名を定義することが多い。以下は変数への代入に関するルールである。

- 「=」前後の空白は無視できる。例えば「VAR = xxx」と書いても、それは「xxx」と判定される。
- 文字列の末尾の空白は無視されない。例えば「VAR = xxx 」としたとき、それは「xxx 」と判定される。
- 代入する文字列が長いとき、バックスラッシュで改行することができる。バックスラッシュ及びその前後の空白は 1 個の空白に纏めて置き換えられる。折り返し後の行頭のタブや空白は無視される。例えば Listing2.2 の変数は「XXX YYY ZZZ」と判定される。
- シェルの環境変数は Makefile の中でも利用できる。つまり、Makefile 内で環境変数と同じものを定義せず、例えば「$(PATH)」と書いても make には伝わる。

Listing 2.2　変数への代入

```
1 VAR = XXX\
2         YYY\
3         ZZZ
```

2.1.2　変数の参照

変数 VAR に代入された文字列は、$(VAR) もしくは${VAR} により参照できる。以下に変数参照の例を示す。

Listing 2.3　変数の参照

```
1 ECHO = echo "Hello"
2 TARGET = target.txt
3
4 $(TARGET) :
5         $(ECHO) > $(TARGET)
```

以下は変数の参照に関するルールである。

- 変数の代入はルールの実行前に行われる。そのため、Listing2.4 のように書くことも可能である。
- 一つの変数に複数回代入が施された場合、最後に代入されたものが適用される。そのため、Listing2.5 の場合、ECHO=「echo "Bye"」となる。
- ルールのターゲットもしくは依存ファイル部分に変数を用いる場合は、そのルールよりも前の行で変数を定義しなければならない。したがって Listing2.6 はエラーとなる。

Listing 2.4　変数の参照例 1

```
1 target.txt:
2         $(ECHO) > target.txt
3 ECHO = echo "Hello"
```

Listing 2.5　変数の参照例 2

```
1 ECHO = echo "Hello"
2 target.txt:
3         $(ECHO) > target.txt
4 ECHO = echo "Bye"
```

Listing 2.6　変数の参照例 3(エラー例)

```
1 ECHO = echo "Hello"
2 $(TARGET):
3         $(ECHO) > $(TARGET)
4 ECHO = echo "Bye"
```

2.1.3　再帰代入

以下の例は、$(STR) が ABC に展開されそうに見えるが、実際は変数の無限再帰でエラーとなる。

Listing 2.7　無限再帰の例

```
1 STR = A
2 STR = $(STR)BC
```

このようなときは、「STR:=$(STR)BC」と書けばよい。記号「:=」は一回限りの置き換えを意味しており、それゆえ再帰が回避されている。

2.1.4　変数への追加

変数に「+=」で代入すると、既に代入されている文字列の末尾に新たに文字列を追加できる。例えば以下の例の場合、STR は ABC となる。

Listing 2.8　文字列追加の例

```
1 STR = AB
2 STR += C
```

2.2 内部変数

依存関係の記述を簡略化できる内部変数が make には用意されている。以下はよく使われる内部変数である。

- **$@**：依存関係行のターゲットに置き換わる。例えば Listing2.9 では$@は test.txt を意味する。
- **$***：拡張子を除くターゲット名に置き換わる。
- **$**<：先頭の依存ファイルに置き換わる。
- **$^**：依存ファイルに置き換わる。例えば Listing2.10 では$^は test1.o test2.o を意味する。

Listing 2.9　内部変数 1

```
1 test.txt : dep.txt; cat dep.txt > $@
```

Listing 2.10　内部変数 2

```
1 test.out: test1.o test2.o
2         gcc -o $@ $^
```

2.3 拡張子を置換する変数

多くのプログラミング言語において、ソースファイルとオブジェクトファイルに拡張子以外同じ名前を使うことが多い。このようなときに、本節で紹介する変数定義は役に立つ。

「$(変数: 拡張子=変換後拡張子)」は文字列であり、変数に代入された文字列の拡張子を変換後拡張子に変えたものとなっている。例えば下記の場合、OBJS は「test1.o test2.o test3.o」になる。

Listing 2.11　拡張子変換

```
1 SRCS = test1.c test2.c test3.c
2 OBJS = $(SRCS:.c=.o)
3 test: OBJS; gcc -o $@ $(OBJS)
```

置き換えができるのは、区切られた文字列の末尾部分のみである。したがって、「SRCS=test1.ctest2.ctest3.c」と定義されていた場合、「OBJS=test1.ctest2.ctest3.o」となる。

3 make の発展的手法

3.1 コマンド行の制御

3.1.1 コマンド表示の抑止

ルールのコマンド行が実行された場合、デフォルトでは画面上に実行されたコマンド行が表示される。それを抑止したい場合は、コマンド行の先頭に@を付ければよい。例えば下記を実行した場合、2 つ目のコマンド行のみ表示されない。

Listing 3.1　コマンド表示の抑止

```
1 test:
2         echo "shown"
3         @echo "hidden"
4 >>
5 echo "shown"
6 shown
7 hidden
```

3.1.2 エラーでも中断しないコマンド行

コマンド行の実行が正常終了しなかった場合、デフォルトではその時点で終了する。一方でコマンド行の先頭に「-」を付けると、たとえエラーになったとしても実行を続ける。下記はその例である。

Listing 3.2　エラーでも中断しない例

```
1 clean:
2         -rm *.txt
```

なお、「-」と「@」は「-@」のように同時に使うことができる (例「-@rm」)。

3.2 パターンルール

拡張子だけ異なる同名なファイルをターゲットと依存ファイルに指定することは多い。このようなとき、**パターンルール**と呼ばれるものを利用すればルールの記述がシンプルになる。

例えば下記のような Makefile を考える。

Listing 3.3　Makefile(パターンルール前)

```
1 test.out : test1.o test2.o; gcc -o $@ $^
2 test1.o : test1.c; gcc -c $^ -o $@
3 test2.o : test2.c; gcc -c $^ -o $@
```

test1.o と test2.o のルールについて、両者はよく似た形をしていることがわかる (このようなケースは実際によく見られる)。また、パターンルールが利用できる条件の、同名なターゲットと依存ファイルも各ルールは満たす。このとき、「%」を用いて下記のように書き換えることができる。

Listing 3.4　Makefile(パターンルール)

```
1 test.out : test1.o test2.o; gcc -o $@ $^
2 %.o : %.c; gcc -c $^ -o $@
```

2 つ目のルールはディレクトリ内の.c 拡張子を持つ任意のファイルに対し、同名で.o 拡張子のファイルをターゲットとして作成する。従ってディレクトリ内に test1.c と test2.c が既に存在すれば、パターンルールで記述されたルールは test1.o と test2.o を作成する。

パターンルールは各依存ファイル毎にルールの実行要否を判定する。例えば Listing3.6 を実行し、その後 test2.c のみ更新したとする。この状態で再度 Listing3.6 を実行したとき、パターンルールは test1.o のみ再作成する。

3.3　階層ディレクトリにおける make

大規模なソフトウェアを開発する場合、ファイルは階層上のディレクトリで管理する。このとき、Makefile は根ノードに当たるディレクトリだけでなく各階層のディレクトリにも作成する。当然ながら各階層にある Makefile は、そこから下にあるファイルに関するものである。

異なるディレクトリにある Makefile を実行したい場合、1.2 節で紹介したように-C オプションを使用すればよい。しかしながら、この方法だと一度にひとつの Makefile しか実行できない。一度の make 実行で複数の Makefile の処理を行いたいならば、Makefile 内のルールが更に make を実行すればよい。

例えばカレントディレクトリに sub1 と sub2、並びに sub3 というサブディレクトリがあるとする。そしてカレントディレクトリ及び各サブディレクトリに Makefile が置いてある。このとき、カレントディレクトリの Makefile に下記のような疑似ターゲットを記述すれば、全てのサブディレクトリの Makefile が実行できる。

Listing 3.5　階層ディレクトリにおける make

```
1 all:
2         cd sub1 && make
3         cd sub2 && $(MAKE)
4         $(MAKE) -C sub3
```

1 行目のコマンドは sub1 に移動してから make を実行する。2 行目のコマンドも同様だが、make の代わりに$(MAKE) を利用している。make と$(MAKE) は基本的に同じ機能であるが、$(MAKE) はカレントディレクトリにおける make を実行したときのオプションを引き継ぐことができる。例えばカレントディレクトリで「make -n」と実行した場合、$(MAKE) は「make -n」と認識される。なお、2 行目のコマンドは sub1 に移動してから実行されているように見えるので、sub2 が絶対パスでないと機能しないように見える。しかしながらそれは誤解で、実のところ make におけるコマンドは各行で完結する。つまり、1 行目のコマンドで sub1 に移動してから make を実行するが、それが終わればカレントディレクトリに戻る。それゆえ 3 行目のように記述しても問題はない。

3.3.1　変数の引継ぎ

下のディレクトリ階層の Makefile を make するとき、元の Makefile の変数は引き継がれない。引き継ぐためには対象の変数に対して export 処理を施す必要がある。

例えば下記 Makefile の場合、変数 VAR は sub にある Makefile でも使えるようになる。

Listing 3.6　変数の引継ぎ 1

```
1 VAR = ABC
2 export VAR
3 all:; $(MAKE) -C sub
```

export は「export VAR = ABC」のように定義と同時に指定することもできる。また Makefile 中のすべての変数を引き継ぎたい場合は下記のように単に export と書けばよい。

Listing 3.7　変数の引継ぎ 2

```
1 VAR1 = ABC
2 VAR2 = DEF
3 export
4 all:; $(MAKE) -C sub
```

3.4 条件分岐

Makefile には条件分岐の機能が備わっており、下記のような書き方をする。

Listing 3.8 条件分岐

```
1 VAR = OK
2 ifeq ($(VAR),OK)
3         SHOW = OK
4 else
5         SHOW = NG
6 endif
7 target:
8         echo $(SHOW)
```

「ifeq ($(変数), 値)」では変数に代入されている文字列と値を比較し、True であればその後の処理を施す。逆に False であれば else 部分の処理を施す。条件分岐の終了は endif で明記する。else 処理の記述は任意だが、endif は必ず書かなければならない。また ifeq と ($(変数), 値) の間にスペースを入れなければ「missing separator」のエラーが出力される。なお、ルール内の処理にはコマンドしか書けないので、この条件分岐はルール外でのみ利用できる。

「ifeq ($(変数),)」と書かれた場合、変数に値が代入されていれば True と判断し、そうでない場合は False と判断する。同様のことは「ifdef VAR」と書いても可能である。

3.5 include によるファイルの読み込み

Makefile の中に別のファイルの内容を挿入するには include を使えばよい。Makefile 内では「include ファイル名」のように宣言する。ファイルの内容がそのまま挿入されるため、ファイル内のコードは Makefile の書式に従っていなければならない。

例えば下記のようなファイルが「flg.mk」という名前で用意されていたとする。

Listing 3.9 挿入ファイル

```
1 FLG = OK
2 ifeq ($(FLG), OK)
3 SHOW = OK
4 else
5 SHOW = NG
6 endif
```

これを別の Makefile に挿入する場合は、以下のような書き方をする。

Listing 3.10 挿入ファイル 2

```
1 include flg.mk
2 test:; echo $(SHOW)
```

挿入元で SHOW という変数が定義されているため、target に関するルールは問題なく機能する。