Makeの使い方

土屋 英亮 hideaki@strauss.ee.uec.ac.jp

1994年 11月 22日

1 はじめに

C言語等でソフトウェアを開発する場合、特にそれが大規模なものである場合、通常は複数のモジュール (ファイル) に分けて作成します。これらのモジュールは個々にコーディングされコンパイルすることにより、一つの実行可能型ファイルになります。このようなソフトウェアの開発過程において、モジュールごとの依存関係や更新の情報を人間が一つずつ管理することはソフトウェアの規模が大きくなればなるほど、困難になります。これをサポートするプログラムがmake[1] です。

一つ実例を示しましょう。あるソフトウェアhogeを作成しようとしたとします。このhogeは、C言語のソースファイルa.c，b.c，c.c，d.cとインクルードファイルx.h，y.hからなるとします。x.hは、a.c，b.cから参照され、y.hはc.c，d.cから参照されます。このソースファイル群からプログラムhogeを作る一番簡単な方法はコマンドラインから[2]

```
% gcc -g -o hoge a.c b.c c.c d.c
```

図 1: プログラムhogeのコンパイル

とします。毎回これをコマンドラインに打ち込むのは面倒ですので、簡単化のためにシェルスクリプトにするのは良いことかもしれません。しかし、それぞれのモジュールが大きい場合にはコンパイルに時間がかかりますし、部分的に修正した後に全てをコンパイルし直すのは不経済です。そこで、図2のMakefileというファイルを作ります。

[1] 教育系計算機にはGNU makeというGNUによって作られたmakeとSunOS付属のmakeが利用できますが、両方とも利用法は同じです。ただし、GNU makeの方がやや細かい設定を行なうことができます。

[2] 授業で行なわれているようにgccを使ってコンパイルするとします。

```
hoge: a.o b.o c.o d.o
        gcc -o hoge a.o b.o c.o d.o

a.o: x.h
b.o: x.h
c.o: y.h
d.o: y.h
```

図 2: 簡単な Makefile

Makefile を作っておくとコマンドラインから make CC=gcc と入力するだけで、図 3 のように全てが自動的にコンパイルされます。

```
% make CC=gcc
gcc    -c a.c -o a.o
gcc    -c b.c -o b.o
gcc    -c c.c -o c.o
gcc    -c d.c -o d.o
gcc -o hoge a.o b.o c.o d.o
```

図 3: Makefile を使ったコンパイル (1)

次に、a.c を修正したとしましょう。この時、a.c は a.o より新しいファイルになります。ここで、make CC=gcc と入力すると、今度は、図 4 のように a.c がコンパイルされた後で、hoge が作り直されます。

```
% make CC=gcc
gcc    -c a.c -o a.o
gcc -o hoge a.o b.o c.o d.o
```

図 4: Makefile を使ったコンパイル (2)

同様に x.h を修正すると、x.h を参照している a.c と b.c がコンパイルされ、hoge が作り直されます。

このように make を使うと、1) コマンドを毎回入力しないくても良い、2) ソースファイルの更新に対応したコンパイルを行なうことができる、という利点があります。UNIX の場合、大規模なプログラムのコンパイルにはほとんどといっていい程

make が使われています[3] 。皆さんも Makefile の書き方を覚えて、能率良くプログラミング作業を行ないましょう。

2 Makefile の書き方 (基本編)

Makefile は make が実行時に参照するファイルです。基本的な書き方は次のようになっています。

```
ターゲット1: ファイル1　ファイル2 .....
   <TAB>     コマンド
ターゲット2: ファイル3　ファイル4 .....
   <TAB>     コマンド
```

図 5: Makefile の基本的な文法

まず、「ターゲット 1」にはコンパイル後に得られるファイル名を書きます。これを ":" によって区切って、次に「ターゲット 1」を作成するために利用しているファイル名を記述します。これが「ファイル 1」, 「ファイル 2」にあたります。このような関係を「ターゲット 1」は「ファイル 1」, 「ファイル 2, ...」に依存しているといいます。ただし、「ターゲット 1」が a.o、「ファイル 1」が a.c といったような場合、つまりファイルのサフィックス[4] だけが異なる場合は、省略することができます (図 2 の Makefile は省略しています。) Make は自動的に、a.o のソースプログラムは a.c であると認識して作業を行ないます。

<TAB>の後には、「ターゲット 1」を生成するためのコマンドを書きます。C のソースファイルからオブジェクトファイルをコンパイルする場合は暗黙の規則が定義されていて、自動的に "cc -c" というコマンドが利用されます。

Make 初心者の人はひとまず図 2 や図 5 に示した Makefile を書いて利用してみましょう。

3 Makefile の書き方 (中級編 1)

次はマクロを説明します。前節にて、C のソースプログラムからオブジェクトファイルをコンパイルする場合は暗黙の規則が利用されると書きました。しかし実際は次のようなコマンドによって行なわれます。

[3] make のコンパイルはどうするのでしょうね。

[4] ファイル名の末尾のこと、MS-DOS の場合は拡張子という。

```
$(CC) $(CFLAGS) -c
```

図 6: *.c から*.o を生成する規則

CC と CFLAGS はマクロというものです。これは Makefile 内で利用できる文字変数で、利用者はこの値を変更することができます。実際は$(CC) と書かれた全ての部分がマクロ CC の値に置き換わります。デフォルト[5] では CC には “cc” が、マクロ CFLAGS には “” (空文字列) が定義されています。図 3 をもう一度見るとわかるのですが、コマンドラインから make CC=gcc とすることによりマクロ CC への定義を行なうことができます。定義が行なわれると CC=gcc となりコンパイル作業が行なわれます。このマクロへの定義を Makefile 内で行なうことも可能です。

```
CC=gcc
CFLAGS=-g

hoge: a.o b.o c.o d.o
        $(CC) $(CFLAGS) -o hoge a.o b.o c.o d.o

(以下同様)
```

図 7: マクロの定義を行なっている Makefile

このように Makefile の先頭に定義を行ないたいマクロを書きます。ここではマクロ CC に gcc を、CFLAGS に-g[6] を指定しています。マクロの定義の文法は “マクロ名 = 定義したい内容 ” というものです。

では、コマンドラインで定義したマクロの値と、Makefile で定義したマクロの値はどちらが優先されるでしょうか。図 7 の Makefile に対してコマンドラインから make CFLAGS=-O としてみましょう。すると、

```
% make CFLAGS=-O
gcc -O   -c a.c -o a.o
...
```

[5] 何も手を加えないシステムが持っている初期状態のこと。

[6] C コンパイラの “-g” オプションはデバッグオプションというもので、dbx 等のデバッガを利用する時に用います。デバッガとはプログラムを実行しながらその挙動や変数の値を調べることができるツールでプログラムの誤りを発見するのにに非常に有用です。

図 8: コマンドラインからのマクロの定義

となり常にコマンドラインから定義された値が優先されることがわかります。

また、マクロはユーザーが勝手に作ることができます。Make コマンドではデフォルトでは LOADLIBS,FFLAGS,EFLAGS,FC,RFLAGS,RC,CFLAGS,... 等のたくさんのマクロを用意して、その用途に応じて使い分けています。この他にユーザーが定義することもできます。例えば、図 7 で示された Makefile でプログラム hoge のコンパイルに数学ライブラリ[7] が必要になったとします。この時は、

```
CC=gcc
CFLAGS=-g
LIBS=-lm

hoge: a.o b.o c.o d.o
        $(CC) $(CFLAGS) -o hoge a.o b.o c.o d.o $(LIBS)
(以下同様)
```

図 9: 新しいマクロを用いた Makefile

のようにします。マクロを利用する方法は$(LIBS) で、$(LIBS) と書かれた場所は全てその値 “-lm” に置き換わります。

4 Makefile の書き方 (中級編 2)

次は複数のターゲットの記述方法を説明します。一つの Makefile で複数のプログラムを生成したい時はどうするのでしょうか。次のような Makefile を作ってもうまくいきません。

```
....
hoge: a.o b.o c.o d.o
        $(CC) $(CFLAGS) -o hoge a.o b.o c.o d.o

geho: e.o f.o
        $(CC) $(CFLAGS) -o geho e.o f.o
....
```

[7] 浮動小数点演算に利用される関数が入っているライブラリ。C コンパイラのコマンドラインに -lm を追加することにより自動的にリンクされる。

図 10: うまくいかない Makefile

ここで書かれている ":" の前の部分 (hoge と geho) はターゲットと呼ばれます。Make コマンドは Makefile の最初に書かれているターゲットだけを対象にして、作業を行ないます。この場合 hoge を生成するのに必要な a.o,b.o,c.o,d.o のコンパイルが行なわれ、それから hoge がコンパイルされます。しかし、hoge を生成するのにまったく無関係な geho のコンパイルは行なわれません。そこで全てのターゲットの前に all[8] というターゲットを作ります。

```
all: hoge geho


hoge: a.o b.o c.o d.o
....
```

図 11: ターゲットをきちんと書いた Makefile

すると make コマンドと実行した時はターゲット all を生成する作業を開始して、ターゲット all に依存している hoge と geho が生成されます。この時、コマンドラインから

```
% make hoge
```

図 12: ターゲットを指定した make コマンドの利用法

とターゲットを明示的に指定することによりターゲット hoge だけを生成することもできます。

このターゲットの非常に一般的な使い方を紹介しましょう。Makefile の中に次のターゲットを入れておきます。

```
....
clean:
        /bin/rm -f *.o hoge geho
....
```

図 13:ターゲット clean

[8] 慣例的に all が利用されます。

このようにして、コマンドラインから make clean[9] とすると、全てのオブジェクトファイルが消去されます。全てのプログラム作業が終了した時、あるいは最初からコンパイルを全てやり直したい時などに便利でしょう。

また、Makefile のコメント行は、“#” を利用します。この “#” の後ろに書いてある全てのことはコメントとして扱われ、無視されます。

```
....
# make software hoge
hoge: a.o b.o c.o d.o
        $(CC) $(CFLAGS) -o hoge a.o b.o c.o d.o $(LIBS)


....
```

図 14: コメントの入れ方

このコメントの上手な利用法を考えます。ソフトウェアをテストする時にその挙動を知るために、ソースファイルの中で

```
....
#ifdef DEBUG
        fprintf(stderr,"output = %d\n",i);
#endif
....
```

図 15: #ifdef DEBUG の使い方

という行を入れ、変数の内容をわざと出力させてそれを監視することはよく行なわれることです。このとき、Makefile の中で、

```
CFLAGS=-g -DDEBUG
```

図 16: Makefile で DEBUG を定義する。

としておきます。このようにすると C のソースプログラムは “gcc -g -DDEBUG” としてコンパイルされます[10] 。そしてソフトウェアのテストが終ったら

[9] UNIX の世界では make all の次くらいに有名です。

[10] -DDEBUG は#define DEBUG と同じ働きをします。“#define” の働きについては C の教科書を見て下さい。

```
CFLAGS=-g #-DDEBUG
```

図 17: Makefile の DEBUG の定義を無効にする。

として、“-DDEBUG” 以降をコメントとしてして、それが働かないようにします。

最後に makefile について書きます。今までは make コマンドが参照するファイルを Makefile としてきました。しかし、実際は makefile というファイルも利用することができます。もし同一ディレクトリ内に Makefile と makefile がある場合は makefile の方を用いて、Makefile は無視されます。

5 中級編のまとめ

今までに説明してきた make コマンドの利用法は make コマンドの機能のほんの一部分に過ぎません。今回はここまでにしておきますが、しばらくしたらもっと高度な make コマンドの使い方を説明したいと思います。