Makeでヘッダファイルの依存関係に対応する

Unix Make 2012-04-05

CやC++で書かれたプログラムをMakeを使ってビルドする、というのはUnix/Linuxではよく行われていることだが、ちゃんとしたMakefileを書くのは意外と難しい。

例えば以下の3つのファイルからなるプログラムを考える。

- foo.h: 関数fooの宣言がある。
- foo.c: 関数fooの実装がある。
- main.c: 関数fooを呼び出す。

```
/* foo.h */
void foo(int a);
```

```
/* foo.c */
#include "foo.h"
#include <stdio.h>

void foo(int a){
    printf("%d\n", a);
}
```

```
/* main.c */
#include "foo.h"

int main(int argc, char **argv){
    foo(10);
    return 0;
}
```

Makefileは例えば以下のように書ける。

```
PROG := myapp
SRCS := main.c foo.c
OBJS := $(SRCS:%.c=%.o)

CC := gcc

all: $(PROG)

$(PROG): $(OBJS)
     $(CC) -o $@ $^
```

プロフィール

id:wagavulin

読者になる 2

検索

ブログ内検索

リンク

はてなブログ

ブログをはじめる（無料）

お知らせ

はてなブログ グループ

最新記事

Bash on Windows (Windows Subsystem for Linux) でvalgrindを動かす

GCC7ではエラーメッセージが改善されるらしい

valgrindが検出するメモリリークの種類

【翻訳】The Linux Graphics Stack

package.elで自動インストール

月別アーカイブ

▸ 2016 (7)

▸ 2015 (2)

▸ 2013 (1)

▾ 2012 (1)

2012 / 4 (1)

▸ 2011 (10)

▸ 2010 (1)

▸ 2009 (4)

▸ 2008 (2)

▸ 2007 (16)

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

もちろんこれでビルドすることができる。

```
$ make
gcc -c main.c
gcc -c foo.c
gcc -o myapp main.o foo.o
$ ./myapp
10
```

しかし、このMakefileには欠陥がある。例えばmyappができた状態で、関数fooの仕様を変更して引数を追加したとする。

```
/* foo.h */
void foo(int a, int b);
```

```
/* foo.c */
#include "foo.h"
#include <stdio.h>

void foo(int a, int b){
    printf("%d\n", a);
    printf("%d\n", b);
}
```

main.cにある関数fooの呼び出し部分も当然変更する必要があるが、それを忘れてmakeを実行したとしよう。

```
$ make
gcc -c foo.c
gcc -o myapp main.o foo.o
```

このようにfoo.cのみがコンパイルされ、myappが作られる。main.oは古いままだ。なぜならMakefileに記述された依存関係によればmain.oが依存しているのはmain.cのみであり、main.cは書き換えられていないからだ。main.cにあるfooの呼び出しは引数1個のであり、呼ばれた側は2個目の引数も使うことになってしまう。実際、実行すると以下のようにおかしな値が出てくる。

```
$ ./myapp
10
-1218204587
```

こうなってしまった原因は、Makefileに書かれた依存関係が実際の依存関係を反映できていないためだ。main.cはfoo.hをインクルードしている以上、foo.hが変更されたら再コンパイルされるべきである。つまり、main.oはmain.cだけでなくfoo.hにも依存している。しかしMakefileではそれが表現されていない。そこで、Makefileにヘッダファイルの依存関係を書く方法を考えよう。

シンプルな方法

もっともシンプルな方法は手動で書くことだ。Makefileに以下の行を追加しよう。

```
main.o: main.c foo.h
```

これで実行すれば以下のようにmain.cがコンパイルされ、エラーを検出することができる。

```
$ make
gcc -c main.c
main.c: In function ‘main’:
main.c:5: error: too few arguments to function ‘foo’
make: *** [main.o] エラー 1
```

もちろんこんなやり方はすぐに破綻する。すべてのcファイルに対して依存関係を手動で書かなければならないからだ。もう少し良い方法を考えよう。

コンパイラを使う方法

あるcファイルが依存するヘッダファイルを知るにはどうしたらよいか？もちろん#includeを追えばよいわけで、単純なケースならgrepでも使えば分かるだろう。しかしこれは中々に面倒な作業だ。インクルードされたヘッダファイルがインクルードしているものも追いかけなければならないし、include文に見えるものが実はコメントアウトされているかもしれない。さらに、#ifdefによる条件分岐の対応や、ヘッダファイルの実際の場所を知るにはコンパイラのオプション（-Dや-I）を知る必要があり、ソースファイルの解析だけではそもそも不可能だ。

しかし、自力でこのようなことをやらなくても、cファイルがインクルードするヘッダファイルを知っている者がいる。それはコンパイラだ。そして嬉しいことにgccはこの情報を表示するオプションを提供してくれている。-Mから始まるオプションがそれだが、いくつか種類があるためまずはそれらの使い方を覚えよう。

gccの-M?オプション

-M

- Mは指定されたcファイルがインクルードするヘッダファイルを調べ、オブジェクトファイルの依存関係として表示する。前述のfoo.cに対して実行すると以下のようになる。

```
$ gcc -M foo.c
foo.o: foo.c foo.h /usr/include/stdio.h /usr/include/features.h \
/usr/include/bits/predefs.h /usr/include/sys/cdefs.h \
/usr/include/bits/wordsize.h /usr/include/gnu/stubs.h \
/usr/include/gnu/stubs-32.h \
/usr/lib/gcc/i486-linux-gnu/4.4.3/include/stddef.h \
/usr/include/bits/types.h /usr/include/bits/typesizes.h \
/usr/include/libio.h /usr/include/_G_config.h /usr/include/wchar.h \
/usr/lib/gcc/i486-linux-gnu/4.4.3/include/stdarg.h \
/usr/include/bits/stdio_lim.h /usr/include/bits/sys_errlist.h
```

随分とたくさんのファイルが出てきたが、これらはstdio.hからインクルードされているものである。

-MM

- Mと同様だが、システムヘッダディレクトリにあるものは除外される。foo.cに対しては以下のようになり、stdio.hやその先のファイルは表示されなくなる。

```
$ gcc -MM foo.c
foo.o: foo.c foo.h
```

なお-Mと-MMの違いは、gcc-3.0以降で仕様が変更されている。gcc-3.0以前の-MMオプションは、#include "xxx.h" のようにダブルクオートで囲まれたもののみを対象にし、#include のように'<>'で囲まれたものは対象にしない、というものだった。

-MF file

- Mや-MMは結果を標準出力に出力したが、-MFを付けると結果を指定したファイルに保存するようになる。

-MG

- Mや-MMは通常インクルード対象のヘッダファイルが見つからないとエラーを出力するが、-MGを追加するとエラーとしなくなる。ヘッダファイルが自動生成されるようなケースで使うらしい。

-MP

依存するヘッダファイルを偽のターゲットとして追加する。foo.cに対して-MPなし、ありで実行した結果は以下のようになる。

```
> gcc -MM foo.c
foo.o: foo.c foo.h
> gcc -MM -MP foo.c
foo.o: foo.c foo.h

foo.h:
```

これだけでは何の意味があるのか分かりにくいだろう。使い方は後述する。

-MD

- Mや-MMは依存するインクルードファイルを調べるのみでありコンパイルは行わないが、-MDを使うとコンパイルも行われる。-MDを使ったときは-MFを使わなくても結果はファイル（ソースファイルの拡張子を.dにしたもの）に保存される。もちろん-MFでファイル名を指定することもできる。

-MMD

- MDと同様で、-Mと-MMの違いである。

-Mオプションを使ったMakefile

では実際にやってみよう。まずは-MMDを使って.cファイルの依存関係を.dファイルに出力するようにする。-MDでも良いが、/usr/includeや/usr/local/includeにあるようなヘッダに対して非互換のアップデートが行われることはあまりないので-MMDを使う。もちろん変更が多いような環境であれば-MDを使おう。前述のとおり、-MMDではコンパイルも同時に行われる。また、変数DEPSに.dファイルの一覧が入るようにする。

```
PROG := myapp
SRCS := main.c foo.c
OBJS := $(SRCS:%.c=%.o)
DEPS := $(SRCS:%.c=%.d)

CC := gcc

all: $(PROG)
```

```
$(PROG): $(OBJS)
	$(CC) -o $@ $^

%.o: %.c
	$(CC) -c -MMD $<
```

結果は以下のようになる。

```
$ make
gcc -c -MMD main.c
gcc -c -MMD foo.c
gcc -o myapp main.o foo.o
$ ls
Makefile  foo.c  foo.d  foo.h  foo.o  main.c  main.d  main.o  myapp*
$ cat main.d
main.o: main.c foo.h
$ cat foo.d
foo.o: foo.c foo.h
```

今後makeを実行するときはmain.d、foo.dが読み込まれればよい。そのためにinclude文を使う。以下の行を all: のある行より下に追加しよう。all: より上にあるとデフォルトのターゲットが変わってしまう。また、includeの頭に'-'を付けているので.dファイルがなくてもエラーにはならない。こうしておかないと最初の実行時（このときは当然.dファイルはない）にエラーになってしまう。

```
-include $(DEPS)
```

では実際に試してみよう。一度ビルドして、その後foo.hを変更してからもう一度ビルドする。なお、touchコマンドは空ファイルを作るのによく使われるが、既存のファイルを指定すると内容を変えずに更新時刻を現在時刻にする。これにより、foo.hが変更されたとMakeがみなすようになる。Makeの勉強をするときはtouchコマンドが便利なので覚えておこう。

```
$ make                  # 1回目
gcc -c -MMD main.c
gcc -c -MMD foo.c
gcc -o myapp main.o foo.o
$ touch foo.h           # foo.hを変更
$ make                  # 1回目
gcc -c -MMD main.c
gcc -c -MMD foo.c
gcc -o myapp main.o foo.o
```

このように、foo.hをインクルードしているmain.c、foo.cも再コンパイルされている。

ヘッダファイルの削除

このMakefileには実はちょっとした欠陥がある。ヘッダファイルの削除があるとエラーになってしまうのだ。例えばmain.cが関数fooを使わないようになり、そのためfoo.cおよびfoo.hが削除されたとしよう。その状態でmakeを実行すると以下のようになる。

```
$ ls
Makefile  main.c  main.d  main.o  myapp*
$ make
make: *** `main.o' に必要なターゲット `foo.h' を make するルールがありません.  中止.
```

これは、前回のmakeにより生成されたmain.dに

```
main.o: main.c foo.h
```

とあるため、foo.hを探してしまうためだ。

ここで-MPオプションの意味が分かるようになる。あらかじめ-MPを付けておけばmain.dは

```
main.o: main.c foo.h

foo.h:
```

となる。foo.hが空のターゲットとなるため、foo.hがなくなってもエラーにはならない。

ということで、できあがったMakefileは以下のようになった。

```
PROG := myapp
SRCS := main.c foo.c
OBJS := $(SRCS:%.c=%.o)
DEPS := $(SRCS:%.c=%.d)

CC := gcc

all: $(PROG)

-include $(DEPS)

$(PROG): $(OBJS)
        $(CC) -o $@ $^

%.o: %.c
        $(CC) -c -MMD -MP $<

clean:
        rm -f $(PROG) $(OBJS) $(DEPS)
```

さらなる課題

以上でヘッダファイルの依存関係の対応ができた。しかし、このMakefileにはまだ欠点がある。この辺はまたそのうちやろうと思う。

出力ディレクトリの指定

上記のMakefileでは.oや.dをカレントディレクトリに出力するが、生成物は別のディレクトリにしたいこともある。例えばVisual Studioのように、デバッグ版はDebugに、リリース版はRelease以下に出力するような場合だ。大したことのないように思えるが、ディレクトリが複数になると依存関係を正しく記述するのが意外のほど面倒になる。

大きなプロジェクトでの速度

プログラムを書くとき、どのくらいの頻度でコンパイルを行うかは人それぞれだと思うが、完成と思うものができるまで一切コンパイルしないという人は少数派だと思う。ある程度書いた段階でとりあえずビルドして文法的な間違いがないかをチェックする人も多いだろう。

このとき、リンクまでやってしまうと大きなプログラムでは時間がかかるため、コンパイルのみ行うようにすると便利だ。Visual C++では指定したソースファイルのコンパイルのみ行うことができる（Ctrl-F7）。これを使うと短時間で文法のチェックができる。

しかしこのMakefileでは`make foo.o`のようにfoo.cだけをコンパイルするようにした場合でもmain.dまで読んでしまう。それどころか make clean のときも.dファイルを読み込む。.cファイルの数が増えるとそれだけで時間がかかってしまう。場合によっては.dファイルの探索と読み込みだけで10秒くらいかかることもある。

wagavulin 4年前

« Makeでビルドオプションによって出力ディ...
はてなブログへ移行 »