There is the original file(in English) here.
最初,前,次,最後のページ,目次 に移動。

---

## ルールの記述

makefileに現れるルールは、いつ、どうやって特定のファイル――これをターゲットといい、ほとんどはルールひとつにつき1つだけです――を作り直すかを指示します。ルールはターゲットの依存関係である別のファイルや、ターゲットを作成したり更新したりするためのコマンドを列挙します。

ルールの順番に意味はありません。ただし、makeにターゲットを与えるためにあるデフォルトゴールを他の方法で指定しなかった場合を除きます。デフォルトゴールとは一番初めのmakefileの最初のルールのターゲットのことです。最初のルールに複数のターゲットがある場合、一番初めのターゲットだけをデフォルトとみなします。ただしこれには、ピリオド(".")で始まるターゲットは、一つ以上のスラッシュ("/")のようなものを含んでいない限りはデフォルトにはならない、それから型ルールを定義するターゲットはデフォルトゴールとして機能しない、という二つの例外があります。(型ルールの定義と再定義の項を参照。)

このため、makefileでは普通、完全なプログラムをコンパイルするために最初のルールを書くか、(ほとんどは"all"というターゲットを使い、)すべてのプログラムを記述するかのどちらかです。ゴールを指定する引数の項を参照。

## ルールの構文

一般的にルールは大体こんな感じです。

*ターゲット* ： *依存関係*
　　*コマンド*
　　...

またはこんな感じです。

*ターゲット* ： *依存関係* ； *コマンド*
　　*コマンド*
　　...

*ターゲット* とはスペースで区切られたファイル名のことです。ワイルドカード文字を使っても良く(ファイル名にワイルドカードを利用するの項を参照)、フォーム名`*a*（*m*）'はアーカイブ*a*のメンバー*m*を表現するものです(ターゲットとしてのアーカイブメンバーの項を参照)。通常、ルール1つにつき1つのターゲットしかありませんが、場合により複数になる理由があるときもあります(ルールの複数のターゲットの項を参照)。

*コマンド* 行はタブ文字から始まります。第一のコマンドはタブ文字とともに依存関係の後に現れるか、セミコロン(";")とともに依存関係と同じ行に現れるかになります。いずれにせよ、結果は同じです。ルール行でコマンドを記述するの項を参照して下さい。

ドル記号("$")は変数参照の開始に用いられるため、実際にドル記号をルールで使いたくなった場合は`$$'というふうに二つ書かなければなりません(変数の利用法の項を参照)。
新しい行が後に続くバックスラッシュ("¥")を挿入することで長い行を分割することができますが、makeはmakefileの行の長さに制限を設けているわけではないので、別に分割は必要というわけではありません。

一つのルールはmakeに二つの事柄を教えます。すなわち、いつターゲットが古くなるか、という事と、必要になった時にどう更新をするのか、という事です。

古くなる(out of date)事の基準は、*依存関係* という言葉で指定します。依存関係はスペースで区切られたファイル名から成り立っています。(ワイルドカードとアーカイブメンバー(アーカイブファイル更新にmakeを利用する の項を参照)は、ここでもまた利用する事が許されています。)
存在しないターゲットまたは(最終修正時刻を比較して)依存関係のどれかより古いターゲットは、古いターゲットと判断します。これはつまり、ターゲットファイルの中身は依存関係内の情報を基準として処理されるものなので、依存関係のどれかが変更されれば、現存するターゲットの中身はもはや有効なものではないと考えるという事です。

更新方法は*コマンド*で指定します。シェル(普通は"sh")で処理する行がありますが、いくつか余分な機能も付随しています(ルール行でコマンドを書く の項を参照)。

## ファイル名にワイルドカードを利用する

たった一つのファイル名でもワイルドカード文字を使えば複数のファイルを指定できます。ボーンシェル(the Bourne shell)と同じく、makeでのワイルドカード文字は`*'、`?'、`[...]'です。例えば、`*.c'は".c"と最後につく(ワーキングディレク

トリ内の)全てのファイルのリストを指定します。

ファイル名の最初の`~'という文字もまた特別な意味があります。単体、またはスラッシュ("/")が続く場合は、ホームディレクトリを表現しています。例えば、`~/bin'は展開されて、`/home/you/bin'になります。(MS-DOSやMS-Windowsのように)各ユーザーにホームディレクトリを与えないシステムでは、この相関関係は環境変数*HOME* 設定でシミュレートされます。

ワイルドカードによる拡張は(シェルが拡張を行う場所である)ターゲット、依存関係、コマンドでは自動的に展開します。他の状況では、wildcard関数で明示的に要求しなければワイルドカードは展開されません。

ワイルドカード文字の特別な意味はバックスラッシュを先につける事で打ち消す事が可能です。このため、`foo¥*bar'は"foo"、アスタリスク("*")、"bar"という単一の名前のファイルを参照することになります。

## ワイルドカードの利用例

ルールのシェルで拡張されたコマンドでもワイルドカードが使えます。例えば、これは全てのオブジェクトファイルを削除するルールです。

```
clean:
        rm -f *.o
```

また、ワイルドカードはルールの依存関係でも便利です。makefileに下のルールがあると、"make print"は、最後に出力してから変更があった全ての".c"ファイルを出力します。

```
print: *.c
        lpr -p $?
        touch print
```

このルールは"print"を空のターゲットファイルとして使用します。詳しくはイベントを記録するだけの空ターゲットファイルの項を参照して下さい。(変更があったファイルだけを出力するのに自動変数`$?'を使っています、詳しくは自動変数の項を参照して下さい。)

ワイルドカードによる拡張は変数を定義した場合無効です。このため、次のように書くと…

```
objects = *.o
```

こうすると、objects変数の値は実際の`*.o'という文字列を指すことになります。ただし、ターゲットや、依存関係、コマンドにobjectsの値を使うと、この時だけはワイルドカードの拡張が起こります。objectsを拡張させたい場合は、代わりにこうして下さい。

```
objects := $(wildcard *.o)
```

wildcard関数の項を参照して下さい。

## ワイルドカード利用の落とし穴

さて、ここにあるのは、あなたがやりたがらないようなワイルドカード拡張の利用方法の馬鹿正直な一例です。仮に、実行可能ファイル"foo"がディレクトリ内の全てのオブジェクトファイルから成ると命令したいならば、こう書きます。

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

objectsの値は実際の"*.o"という文字列です。ワイルドカードの拡張は"foo"のためのルールでは有効であり、そのため、*存在する*`.o'ファイルのそれぞれが"foo"の依存関係になり、必要であれば再コンパイルされます。

しかし、全ての".o"ファイルを削除した場合はどうなりますか? ワイルドカードにどのファイルもマッチしなければ、そのまま放っておかれて、そしてまた"foo"は"*.o"という全く奇妙なファイル名に依存することになります。こんなファイルなどだいたい存在しないので、makeは"*.o"の作成方法が見つりません、というエラーを出します。

実際にはワイルドカードの拡張で望んだ結果を得ることができますが、wildcard 関数や文字列代用などの、もっと洗練された技術を持つべきです。

これらの技術については後の項で説明します。

MicrosoftのOS(MS-DOSとMS-Windows)はバックスラッシュをパスでディレクトリの区切りに使っています。こんな感じです。

```
c:¥foo¥bar¥baz.c
```

これはUNIX形式の`c:/foo/bar/baz.c'と同じです("c:"部分はいわゆるドライブ文字です)。makeをこれらのOSで動作させる場合、バックスラッシュはUNIX形式のパス名の前につくスラッシュと同じ動作を提供します。ただし、この提供の中にはバックスラッシュが引用文字(quote character)になる場所でのワイルドカードの拡張を含めません。そのため、この場合はUNIX形式のスラッシュを利用しなければなりません。

### wildcardという関数

ワイルドカードはルールで自動拡張されます。しかし、変数をセットしたり関数の引数の中だと、通常はワイルドカードは拡張されません。そういう場所でもワイルドカードを拡張させたいならwildcard関数を使う必要があります。こんな感じです。

```
$(wildcard 型 ...)
```

makefileのいたる所で用いられるこの文字列は、現在存在しているファイル名の中で、所定のファイル名の型(pattern)に一致しているファイル名が空白で区切られリストされたものに置き換わります。現存のどのファイルも型に一致することがなければ、その場合はwildcard関数の出力で型が省略されます。注意:一致しないワイルドカードが無視するより一字一句同じに使おうとするルールではどうなるか、という事はまた別の事です(ワイルドカード利用の落とし穴の項を参照)。

wildcard関数の使い道の一つは、あるディレクトリの全てのCソースファイルのリストの取得に使う事です。こうします。

```
$(wildcard *.c)
```

後ろにつける".c"を".o"に置き換えると、Cソースファイルのリストがは結果としてオブジェクトファイルのリストに変わります。このようにします。

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

(ここではpatsubstという別の関数を使いました。文字列を代用、分析する関数を参照して下さい。)

このように、ディレクトリ内のCソースファイルを全てコンパイルしてそれからまとめてリンクするmakefileは以下のように書けます。

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(objects)
```

(上ではCプログラムをコンパイルするのに暗黙のルールを利用しているため、ファイルをコンパイルするのに明示的にルールを書く必要がありません。"="の変形型である":="の説明については変数の二つの味の項を参照して下さい。)

#### ディレクトリから依存ファイルを探す

大きなプロジェクトでは、バイナリから独立したディレクトリにソースを置くことが望ましい事がよくあります。makeのディレクトリサーチ機能はいくつかのディレクトリをサーチして依存関係にあるファイルを自動的に検出してくれるので、この作業を容易にします。
ディレクトリ間でファイルを再分配する場合、個々のルールを変更する必要がなく、ただサーチパスに目を向けていればいいのです。

### VPATH: 全依存関係をPATH から探す

makeの変数、VPATHの値にはmakeがサーチするディレクトリのリストを指定します。ほとんどの場合、指定するディレクトリは依存関係にあるがカレントディレクトリにはないファイルを含むディレクトリになりますが、そうは言ってもやはり、VPATHにはルールのターゲットであるファイルを含めた全てのmakeが利用するファイルをサーチするリストを指定します。

このため、カレントディレクトリに依存関係やターゲットとして列挙したファイルがない場合、makeは発見できなかったファイルと同じ名前のファイルを探すのにVPATHにあるリストを使います。もしリスト中のディレクトリの一つで同じ名前のファイルが見つかったならば、そのファイルが依存関係になってしまいます(下を見て下さい)。そしてルールは、依存関係リストにあるファイルの名前を、ファイルがカレントディレクトリにあるかのように指定してしまいます。ディレクトリサーチでシェルコマンドを記述する の項を参照して下さい。

VPATH変数では、空白かコロン(":")でディレクトリ名が区切られています。ディレクトリが並べられた順番は、makeがサーチしていく順番になります。(MS-DOSとMS-Windowsでは、コロンはドライブ文字の次の文字としてパス名自身に使われるので、VPATHのディレクトリ名の区切りにはセミコロン(";")が利用されます。)

例として、

```
VPATH = src:../headers
```

は、"src"、"../headers"という二つのディレクトリを指定します。また、makeはこの順番で検 索します。

VPATHの値を前述のようにすると、以下のルール…

```
foo.o : foo.c
```

は、このように書かれたものと解釈します…

```
foo.o : src/foo.c
```

ただし、こうなるのは"foo.c"がカレントディレクトリに存在せず、かつ"src"ディレクトリで見つかったと仮定した場合です。

### vpathディレクティヴ

vpathディレクティヴ(下部の事例に注意しておいて下さい)はVPATH変数に似ていますが、より用途に富みます。vpathディレクティヴを利用すれば、特定の型に一致するファイル名の特定のクラスを探すパスを指定する事ができます。こうすれば、ファイル名群の1クラスを確実に探せるディレクトリや、別のファイル名を確実に探せる(または探せない)ディレクトリを与える事が可能になります。

vpathディレクティヴには3つの形式があります。次のものがその3つです。

`vpath` *型 ディレクトリ*
: *型* に一致するファイル名を検索するのに *ディレクトリ* というパスを指定します。
サーチパスである *ディレクトリ* とは、検索されるディレクトリがコロン(MS-DOSとMS-Windowsではセミコロン)か空白で区切られたリストです。VPATH変数で用いられる検索パスと同じですね。

`vpath` *型*
: *型* に関連するサーチパスを除外します。

`vpath`
: 以前にvpathディレクティヴで指定した全てのサーチパスを消去します。

vpathの型は"%"という文字を含む文字列です。文字列は検索中の依存関係ファイルの名前に必ず一致しなければなりません。"%"という文字は連続する0個以上の文字ならどれにでも一致します(型ルールの場合と同様です。型ルールについては型ルールの定義と再定義の項を参照して下さい)。例えば、%.h は .hで終わるファイルに一致します。("%"がないと、型を依存関係に正確に一致させなければならなくなります。ほとんどの場合不便です。)

vpathディレクティヴの型では、"%"という文字はバックスラッシュ("¥")を前置きすれば引用できます。"%"や別のバックスラッシュを引用するのに使われたバックスラッシュは、ファイル名を比較させる前に型から削除します。"%"という文字を引用するおそれのないバックスラッシュは、この事に悩まされずそのままになります。

依存関係がカレントディレクトリに存在せず、vpathディレクティヴの *型* が依存関係ファイルの名前に一致すると、その場合、vpathディレクティヴの *ディレクトリ* がVPATH変数のそれと同じように(また、それよりも先に)検索されます。

例えば…

```
vpath %.h ../headers
```

は、カレントディレクトリで発見できなければmakeに、`../headers'というディレクトリで`.h'で終わる名前の全ての依存関係を探すように命令します。

もしvpathのいくつかの型が依存関係のファイル名に一致すれば、makeは各ディレクティヴで名前を挙げられた全てのディレクトリを検索して、一致した各vpathディレクティヴを一つずつ処理します。makeは複数のvpathディレクティヴをmakefileで出てきた順番に処理し、同じ型を持つ複数のディレクティヴはお互いを独立したものとして扱います。

このため、

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

は、"foo"内の".c"で終わるファイルを探し、それから"blish"、それから"bar"を探します。
一方、

```
vpath %.c foo:bar
vpath %   blish
```

は、"foo"内の".c"で終わるファイルを探し、それから"bar"、それから"blish"を探します。

## どうやってディレクトリサーチが行われるか

タイプ(一般的なものか選ばれたものか)に関係なく、依存関係をディレクトリサーチで見つけた時、見つかったパス名が依存関係のリストで実際に用意したものに当てはまらない事があります。

makeがディレクトリサーチで見つけたパスを保持するか捨てるかを決めるアルゴリズムは以下にあるようなものです。

1. makefileに指定したパスにターゲットファイルが存在しない場合、ディレクトリサーチを行います。
2. ディレクトリサーチが成功すればパスを保持して、そのファイルを試験的にターゲットとして充ててみます。
3. 同じ方法でターゲットの全依存関係を調べます。
4. 依存関係の処理後、ターゲットに再構築の必要があるかもしれませんし、ないかもしれません。つまり、
    1. ターゲットを再構築する必要がない場合、ディレクトリサーチ中に見つかったファイルへのパスは、このターゲットを含んでいるすべての依存関係リストに利用されます。簡単に言えば、makeがターゲットを再構築する必要がなければディレクトリサーチを通して見つけたパスを使います。
    2. ターゲットを再構築する必要がある場合(ターゲットが古い場合)、ディレクトリサーチ中に見つかったパス名を*見逃して*、makefileで指定されたファイル名を使って再構築します。簡単に言えば、makeがターゲットを再構築しなければならなければ、ディレクトリサーチを通して見つけたディレクトリとは違う場所で再構築します。

このアルゴリズムは複雑に見えるでしょうが、実行してみればほとんどの場合はまったくもって希望通りに動作してくれます。

別のmakeツールではもっと簡単なアルゴリズムを採用しているため、もしもファイルが存在せずにディレクトリサーチを通してファイルが見つかった場合、ターゲットを再構築する必要があってもなくてもパス名が使われます。これだと、ターゲットを再構築すると、ディレクトリサーチ中に発見したパス名で作成されてしまいます。

実際にディレクトリの一部または全部で行わせるのが目的なら、makeにその目的を示すのにGPATHが使えます。

GPATHは構文もフォーマットもVPATHと同じです(つまり、パス名のリストを空白かコロンで区切っています)。もしディレクトリサーチをして、GPATHにも指定してあるディレクトリで古いターゲットが見つかった場合でもパスネームが見逃されません。ターゲットはこの拡張したパスを使って再構築されます。

## ディレクトリサーチでシェルコマンドを記述する

ディレクトリサーチで別のディレクトリにある依存関係を発見した場合、ルールのコマンドは変更されません、つまり、書かれているものそのままの形で実行します。このため、makeが見つけたディレクトリでの依存関係を探させるのに注意をもってコマンドを記述しなければなりません。

`$^' のような自動変数が実行時に利用されます(自動変数の項を参照)。例えば、`$^' の値はルールの全依存関係のリストです。このリストには見つかったディレクトリ名も含みます。それから`$@' はターゲットです。

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

(CFLAGS変数は暗黙のルールをCコンパイルで指定するために存在しています。上では一貫性を持たせるために使ったので、すべてのCコンパイル作業に斉しく影響します。詳しくは暗黙ルールに用いられる変数の項を参照してください。)

コマンドではやって欲しくないというのに、普通、依存関係は極力自分からヘッダーファイルをインクルードしようとします。自動変数`$<' は正真正銘第一の依存関係であることを示します。

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

## ディレクトリサーチと暗黙のルール

暗黙のルールを判断する時にもVPATHやvpathで指定したディレクトリでのサーチが行われます(暗黙のルールの利用の項を参照)。

たとえば`foo.o'で明示的にルールを定めなければ、makeはコンパイル作業に暗黙のルールが充てられるか考えます。具体的には、`foo.c'というファイルが存在すれば、これをビルトインルールでコンパイルする、といった具合です。こういうファイルがカレントディレクトリになければ、存在すると思われるディレクトリを探します。検索先のディレクトリで`foo.c'が存在する(またはmakefileで示した状態である)なら、Cコンパイルの暗黙のルールが適用されます。

通常、暗黙のルールのコマンドは必要性から自動変数を使用します。そのおかげで何の苦労もなくディレクトリサーチで見つかるファイル名を利用できるのです。

## リンクライブラリをディレクトリサーチする

ディレクトリサーチは、リンカで使われるライブラリには特別な方法を適用します。この特別な機能は`-l*name*'という形で名前を書いた場合に実行されます。(これは奇妙な事とも言えます。というのは普通、依存関係はあるファイルの名前であり、ライブラリの*ファイル名*は`-l*name*'のようなものではなくて`lib*name*.a'という感じなのですから。)

`-l*name*'という形で依存関係名を指定すると、makeは、カレントディレクトリvpathサーチパス、VPATHサーチパスから、`lib*name*.a'というファイルを特別に探します。さらに、`/lib'、`/usr/lib'、`*プリフィクス*/lib'ディレクトリも同様に探します。(一番最後のものについては、普通`/usr/local/lib'ですが、MS-DOS/MS-Windows版のmakeでは*プリフィクス*がDJGPPツリーのルートである定義されているかのように動作します)。

例を挙げると、

```
foo : foo.c -lcurses
        cc $^ -o $@
```

は、`foo.c'か`/usr/lib/libcurses.a'よりも`foo'のほうが古ければ`cc foo.c /usr/lib/libcurses.a -o foo'というコマンドを呼び出して実行します。

### 偽りのターゲット(Phony)

本当のファイル名ではないターゲットをフォニーターゲット(Phony target=偽のターゲット)といい、明示的にコマンドを実行させたい時にこの名前を利用します。フォニーターゲットを使うのには、同名のファイルとの衝突の回避、パフォーマンスの向上という二つの理由があります。

ターゲットファイルを作成しないコマンドからなるルールを書くと、再作成処理の度にコマンドが実行される事になります。下がその一例です。

```
clean:
        rm *.o temp
```

rmコマンドは`clean'という名前のファイルを作成しないため、そんな名前のファイルが存在することはおそらくないでしょう。そのため、`make clean'と命令する度にrmコマンドを実行します。

フォニーターゲットがないと、万一なんらかの処理で`clean'という名前のファイルができた場合、その時は作業をやめてしまいます。全く依存関係を持たないため必然的に`clean'というファイルは最新版と判断され、その結果コマンドを実行しなくなってしまいます。この問題を回避するには、下にあるように.PHONYという特別なターゲットを使ってターゲットが偽物である事と明示的に宣言すればいいのです(特別なターゲットについては特別なビルトイン・ターゲット名の項を参照)。

```
.PHONY : clean
```

こうしておけば`make clean'コマンドは、常に`clean'があるかどうかという事を気する事なくコマンドを実行してくれます。

フォニーターゲットが別ファイルから作られうる実際のファイルを指名していないという事をmakeは分かっているので、暗黙のルールでの検索作業でもフォニーターゲットを飛ばします(暗黙のルールの利用の項を参照)。こういう理由から実際にファイルが存在するかを悩む必要がない場合でもターゲットが偽物だと宣言しておくのは、パフォーマンスの面で良い事なのです。

というわけで、下にあるようにまずはじめにcleanがフォニーターゲットだと指定する行を書いて、それからルールを書いて下さい。

```
.PHONY: clean
clean:
        rm *.o temp
```

フォニーターゲットは本物のターゲットファイルの依存関係にすべきではありません。そうしてしまうとmakeがファイル更新をする度に実行することになります。フォニーターゲットが本物のターゲットの依存関係にならない限り、フォニーターゲットのコマンドが実行されるのはフォニーターゲットが特別なゴールになった時だけです(特別なゴールについては特別なゴールのための引数の項を参照)。

フォニーターゲットに依存関係を持たせることもできます。ひとつのディレクトリに複数のプログラムが混在するなら`./Makefile'という一つのmakefileに全プログラムを記述するのが最も手軽な方法です。デフォルトで作り直されるターゲットはmakefileの一番初めのものなので、第一のターゲットを`all'というフォニーターゲットにしておき、個々のプログラム全部を依存関係として書きます。

例:

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

これで`make'とほんの一言命令するだけで三つ全部のプログラムを更新できるようになりました。(`make prog1 prog3'のように)更新するものだけを引数で指定することもできます。

あるフォニーターゲットが別のフォニーターゲットの依存関係にある時、片方がもう片方のサブルーチンと化します。例えば、次にある`make cleanall'では、オブジェクトファイルとdiffファイル(diffは difference のdiff)、それから`program'というファイルを削除することになります。

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

### 依存関係やコマンドのないルール

ルールにコマンドや依存関係がなく、ターゲットとして指定したファイルが存在しないなら、makeがそのルールを実行する時、常にそのターゲットが更新済みであるものと仮定します。この事はそういうターゲットに依存する全ターゲットが毎回コマンドを実行させられることを示唆しています。

下の例がこの事を浮き彫りにしてくれるでしょう。

```
clean: FORCE
        rm $(objects)
FORCE:
```

上に挙げた`FORCE'というターゲットがこの特別な条件を満たしているため、このターゲットに依存する`clean'というターゲットは無理やりコマンドを実行させられる事になります。`FORCE'という名前に関しては特別な意味は全くなく、この方法で一般的に使っている単なる名前でしかありません。(訳注:無理やりコマンドを実行させる意味でforceという単語を慣習的に使っているようです。)

やってみればわかるように、上記の方法で`FORCE'を使えば`.PHONY: clean'を使うのと同じ結果になります。

`.PHONY'を使うほうがより明示的でより能率がいいです。ですが、他のmakeプログラムでは`.PHONY'はサポートされていないために多くのmakefileで`FORCE'が使われています。これについては偽りのターゲット(Phony)の項を参照して下さい。

### イベントを記述するだけの空ターゲットファイル

空っぽのターゲット(empty target)はフォニーターゲットの変形版です。空ターゲットはその時々に応じて明示的に要求した動作をさせるコマンドをしまっておくのに使います。フォニーターゲットと違ってターゲットファイルが本当に存在していても良いのですが、ファイルの中身は関係ないので普通はカラです。

空ターゲットファイルを使う目的は最終更新時刻の記録、つまり最後にルールのコマンドが実行された時間を記録する事にあります。ターゲットファイルを更新するtouchコマンドを他のコマンドに混ぜて使うので、記録作業を行ってくれるのです。

空ターゲットファイルはいくつか依存関係を持たなければなりません。空ターゲットを更新するように要求すると、ターゲットより新しい依存関係があれば——言い換えればターゲットを更新した最後の時刻以降に依存関係に変更があるのなら——コマンドが実行されます。これがその例です。

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

このルールを使うと`make print'は、最後に`make print'を実行した後にいずれかのソースファイルに変更があった場合にのみlprコマンドを実行します。自動変数`$?'は変更があったファイルだけを出力させるために使っています(自動変数の項を参照)。

### 特別なビルトイン・ターゲット名

ターゲットとして書くと特別な意味を持つようになる名前があります。

.PHONY
: .PHONYという特別なターゲットの依存関係は偽のターゲット(Phony Target)になります。フォニーターゲットは処理する時に同名のファイルの存在や最終修正時刻を気にしません。つまり、makeは無条件にコマンドを実行します。これについては偽りのターゲット(Phony)の項を参照してください。

.SUFFIXES
: .SUFFIXESという特別なターゲットの依存関係はサフィクス(接尾語)ルールでチェックされるサフィクスのリストです。これについては古いタイプのサフィクスルール(Suffix Rules)の項を参照してください。

.DEFAULT
: .DEFAULTで指定したコマンドは、(明示ルール、暗示ルールの両方を調べて)発見した、全てのルールなしターゲットに充てられます。これについては最後の手段を定義するデフォルトルールの項を参照して下さい。.DEFAULTコマンドを指定すると、ルールのターゲットとしてではなく依存関係として書かれた全てのファイルは、どれもみな.DEFAULTにあるコマンドを自分のものとして実行します。暗黙のルールを検索するアルゴリズムの項を参照して下さい。

.PRECIOUS
: .PRECIOUSが依存しているターゲットは特別扱いします。具体的には、コマンド実行中にmakeの処理がキャンセルされたり中断されたりした場合にターゲットを削除しません。これについてはmakeを邪魔するか中断させるの項を参照して下さい。それから、ターゲットが中間ファイルであり、不要になれば普通削除されるような場合であっても削除される事がありません。暗黙ルールの連鎖の項を参照して下さい。暗黙のルールに使うターゲットの型を(`%.o'のようにして).PRECIOUSという特別なターゲットの依存関係ファイルとして書き並べることもできます。こうすると、ターゲットの型が目的のファイル名に一致するルールによって作成された中間ファイルの保存が可能になります。

.INTERMEDIATE
: .INTERMEDIATEが依存しているターゲットは中間ファイルとして処理します。暗黙ルールの連鎖の項を参照して下さい。依存関係がない.INTERMEDIATEはmakefileにある全てのターゲットの全てのファイルを中間ファイルとして処理します。

.SECONDARY
: .SECONDARYが依存しているターゲットは、絶対に自動削除されないものを除き、中間ファイルとして扱います。暗黙ルールの連鎖の項を参照してください。依存関係のない.SECONDARYはmakefileにある全てのターゲットの全てのファイルをセカンダリとして処理します。

.IGNORE
: .IGNOREの依存関係を指定すると、makeは特定のファイルを対象とするコマンドの実行中にエラーが発生しても無視します。.IGNOREにはコマンドを書いても無意味です。依存関係のないターゲットとして.IGNOREと書くと、全ファイルに関するコマンド実行中のエラーを無視します。`.IGNORE'とだけ書く利用方法は以前のmakeプログラムとの互換性を持たせる目的でのみサポートしています。makefileのどのコマンドにも影響を与えるということは、さほど便利なことではありませんので、特定のコマンドでのエラーを無視するというずっと便利な方法を利用する事をお勧めします。コマンド内エラー.の項を参照して下さい。

.SILENT
: .SILENTに依存関係を指定すると、makeは特定のファイルを更新するのに使うコマンドを実行前に画面に表示するのをやめます。.SILENTにはコマンドを書いても無意味です。依存関係のないターゲットとして.SILENTと書くと、全コマンドを実行前に画面に表示させないようにします。`.SILENT'とだけ書く利用方法は以前のmakeプログラムとの互換性を持たせる目的でのみサポートしています。特定のコマンドについて黙らせるほうが便利な方法ですのでそちらをお勧めします。コマンドの繰り返しの項を参照して下さい。makeのある特定の実行時にだけ全部のコマンドを黙らせたいなら、`-s'か`--silent'オプションを使ってください(オプション要約の項を参照)。

.EXPORT_ALL_VARIABLES
: ターゲットとして名前を挙げるだけで、容易にmakeに子プロセスに全ての変数をデフォルトとしてエクスポートさせることができます。サブmakeに変数を伝えるの項を参照して下さい。

ターゲットとして書かれていて`.c.o'みたく二つのサフィクスが連結している場合、定義されたどの暗黙ルールのサフィクスでも一つの特別なターゲットとして処理します。こういうターゲットはサフィクスルールになりますが、これは暗黙のルールを定義するのには時代遅れの方法です(ただ、時代遅れではありますが今なお幅広い用途があります)。原理としては、この方法では二つに区切ってその両方ともをサフィクスリストに加えれば、どんなターゲット名も特別になる可能性があります。実際には、サフィクスは普通`.'で始まるため、それぞれの特別なターゲット名もまた`.'から始まるものになります。古いタイプのサフィクスルールの項を参照して下さい。

### ルールの複数のターゲット

一つのルールで複数のターゲットを指定したものは、それぞれのターゲットに対して全く同じルールをいくつも書くのと全く同じ効果を持ちます。同じコマンドをすべてのターゲットに充てますが、コマンドで`$@'を使えばそれぞれ違った動作をさせる事もできます。このルールはすべてのターゲットに同じ依存関係を持たせるのにも役に立ちます。

この機能は二つのケースにおいて便利です。

- コマンドなしの依存関係が欲しい場合。例えば、

```
kbd.o command.o files.o: command.h
```

とすれば、上記の三つのオブジェクトファイルそれぞれに依存関係を一つ追加します。

- 同様に、コマンドは全ターゲットに働きます。自動変数`$@'のおかげでコマンドを完全に同一にする必要がありません。こうすれば、コマンドで更新すべき特定のターゲットを自動変数`$@'で代用できます。(自動変数の項を参照)。例えば、

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

これは下のものと同等です。

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

ここではgenerateという仮想のプログラムが二種類の出力をするもので、一方は`-big'が与えられたとき、もう一方は`-little'が与えられたときに動作すると仮定しています。substの役割については文字列を代用,分析する関数の項で説明しています。

`$@'変数はコマンドでの変化を対象とするものなのですが、変化ターゲットに応じて依存関係を変化させたい場合を考えてみて下さい。複数のターゲットの依存関係を一つのルール上で変化させることは普通はできませんが、スタティックな型ルールを使えばそれも可能になります。詳しくはスタティックな型ルールの項を見て下さい。

### 一つのターゲットのための複数のルール

一つのファイルがいくつかのルールのターゲットになることがあります。そういうターゲットがあると、全ルールで名前を挙げられた全依存関係が合わさって一つの依存関係リストになります。どこかのルールの依存関係のどれかが一つでもターゲットより新しければ、コマンドが実行されます。

一つのファイルのために実行するコマンドは必ず一塊になっていなければなりません。万一、複数のルールで同じファイルのための処理をすれば、makeは最後の組だけを利用してエラーメッセージを表示します。(特例として、それが一つのドットから始まるファイル名である場合のみエラーメッセージは表示されません。この奇妙な動作はmakeの別の完成版との互換性のためだけにあります。)　こうやってmakefileを記述することは全く意味を成さない、という理由でmakeはエラーメッセージを通知します。

依存関係だけしかないエクストラルール(臨時ルール)を使えば、一度で多くのファイルに大量の臨時の依存関係を与えられます。例えば、たいていはobjectsには作業中に作られるコンパイラの出力ファイルの全リストが入ります。そこで、`config.h'が変更されればその全てを再コンパイルさせる、と指示する簡単な方法として以下のように書けます。

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

こうすれば、断続的に追加の依存関係を加えたくなった場合でもちゃんとオブジェクトファイルの作成方法を指定してあるルールを変更せずに、手軽な形式を使って挿入したり取り除いたりする事が可能です。

もう一つの妙策として、makeのコマンド引数でセットした変数を使って追加の依存関係を指定することも出来ます(変数を上書きするの項を参照)。例えば、

```
extradeps=
$(objects) : $(extradeps)
```

とすれば`make extradeps=foo.h'というコマンドでは`foo.h'を各オブジェクトファイルの依存関係として扱わせます。ただし、ただ`make'とするだけでは正常に動作しません。

明示的なルールでコマンドを持つターゲットが何もなければ、その場合makeはコマンドを見つけるために適用可能な暗黙のルールを探します(暗黙のルールの利用の項を参照)。

### 静的(static)な型ルール

複数のターゲットを指定し、ターゲット名を基準として各ターゲットから依存関係名を構成してくれるルールをスタティックな型ルールといいます。複数のターゲットを指定するのには普通のルールよりもこちらのほうが一般的です。というのは、このターゲットでは同一の依存関係を持つ必要がないからです。これらの依存関係は*類似の(analogous)*ものでなければなりませんが、別に*同一の(identical)*ものである必要がないのです。

### 静的な型ルールの構文

ここにあるのが静的な型ルールのシンタックス(構文)です。

*ターゲット* `...`: *ターゲットの型* : *依存関係の型* `...`
*コマンド*
`...`

（訳注：`...`は後に同じ種類…ターゲットならターゲット…を続ける事が出来るという意味）

*ターゲット(targets)*リストでルールが対象とするターゲットを指定します。普通のルールのターゲットと同じようにターゲットにはワイルドカード文字を使えます(ファイル名にワイルドカードを利用するの項を参照)。

*ターゲットの型(the target-pattern)*と*依存関係の型(dep-patterns)*で各ターゲットの依存関係をどう処理するかを命令します。*ターゲットの型(target-pattern)*を各ターゲットと競わせてターゲット名の一部を引き抜く機能として語幹(stem)というものがあります。語幹(stem)は各*依存関係型(dep-patterns)*の代わりに依存関係名(各*依存関係型(dep-pattern)*からできた名前)の作成に使われます。

通常それぞれの型は、たった一つ`%'を含んでいます。*ターゲットの型(target-pattern)*があるターゲットに一致するかを調べる時、`%'はターゲット名のどの部分にも一致します。この、一致した部分のことを語幹(stem)と呼ぶのです。残りの部分は正確に一致しなければなりません。例を挙げると、`foo.o'というターゲットは`%.o'という型に一致し、語幹(stem)は`foo'になります。`foo.c'と`foo.out'というターゲットはこの型には一致しません。

各ターゲットのために用意する依存関係の名前は、それぞれの依存関係の型で`%'で代用された語幹(stem)を基にします。例えば依存関係の型の一つが`%.c'であれば、代用の対象が`foo'なら`foo.c'という依存関係名を定めます。`%'を含まない依存関係の型を書くのも合理的な方法です。そうするとその依存関係は全てのターゲットで同じになります。

型ルールでは`%'という文字を引用するには前置きのバックスラッシュ(`¥')を使います。`%'という文字を引用しないバックスラッシュはさらにバックスラッシュを使うことでそれ自身を引用できます。`%'という文字やバックスラッシュ自身を引用するために用いられるバックスラッシュは、ファイル名と型を比較する前、あるいは型に代用される語幹(stem)を得る前に、型から削除されます。`%'という文字に関係ないバックスラッシュは問題なくそのまま処理されます。例えば`the¥%weird¥¥%pattern¥¥'という型では、`the%weird¥'に有効な`%'文字が続き、その後に`pattern¥¥'が続きます。最後の二つのバックスラッシュは`%'という文字に全く影響しないためにそのまま残ります。

以下の例ではそれぞれに一致する`.c'ファイルから、`foo.o'と`bar.o'をコンパイルさせます。

```
objects = foo.o bar.o

all: $(objects)

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

ここで現れた`$<'というのは依存関係の名前が格納された自動変数であり、`$@'というのはターゲットを格納した自動変数です。これについては自動変数の項を参照して下さい。

指定するターゲットはそれぞれにターゲットの型に一致しなければならず、一致しないものはそれぞれについて警告を出します。ファイルのリストがあり、そのうちいくつかだけが型に一致する場合、`filter`関数を使って一致しないファイルだけを取り除けます(文字列を代用,分析する関数の項を参照)。

```
files = foo.elc bar.o lose.o
```

```
$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

この例において`$(filter %.o,$(files))'の結果は`bar.o lose.o'になり、最初のスタティックな型ルールでは各オブジェクトファイルを相当するCソースファイルからコンパイルして更新させます。`$(filter %.elc,$(files))'の結果は`foo.elc'になり、このファイルは`foo.el'から作成されます。

スタティックな型ルールでの$*の利用方法をもう一つの例で示します。

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

generateコマンド実行時、$*には語幹(stem)が拡張されて、`big'か`little'かどちらかになります。

## 静的な型ルールvs暗黙のルール

スタティック(静的)な型ルールと、型ルールとして定義された暗黙のルールには多くの共通点があります(型ルールの定義と再定義の項を参照)。どちらにもターゲットに合わせる型と依存関係の名前を構築するための型があります。相違点はmakeがいつルールを適用するかを決める方法にあります。

暗黙のルールは自分の持つ型に一致するどんなターゲットでも適用することが *可能*ですが実際に適用 *する*のは、コマンドは省略されているが依存関係は見つけられるターゲットがあった時だけです。当てはまる暗黙のルールが複数現れた場合はルールの順番によって選ばれた一つだけを適用します。

それに比べてスタティックな型ルールでは、ルールで指定したまんまのターゲットのリストを適用します。他のどのターゲットにも適用する事はできず、指定したターゲットのそれぞれで同じように適用されます。二つの矛盾するルールが適用されて、どちらにもコマンドがある場合はエラーになります。

以下の理由から暗黙のルールよりスタティックな型ルールは優れていると言えます。

- 多くのファイルがあり、その名前が統語的に分類できないために普通使われる暗黙のルールがありますが、それさえも一つの明示的なリストで上書きさせる事が出来ます。
- 利用しているディレクトリの中身がはっきりしない場合、他の関係ないファイルがmakeを間違った暗黙のルールに導くかも知れません。暗黙のルールのサーチを行う順番によってこの選択が決まってしまいます。スタティックな型ルールを使えば不確実になることがないのです。つまり、各ルールは正確に指定した通りのターゲットを適用してくれます。

### 二重コロンルール

二重コロンルールとはターゲットの後に`:'を使う代わりに`::'を使って書くルールの事です。このルールは同じターゲットが複数のルールで現れた時、普通のルールとは異なる動作をします。

一つのターゲットが複数のルールで現れると、普通のルールでも二重コロンルールでも、全てのルールは同じタイプでなければなりません。二重コロンルールでは各自のルールが他のルールから独立しています。各二重コロンルールのコマンドは、ターゲットがその依存関係のどれかよりも古い場合に実行されます。二重コロンルールでは、何も実行させなかったり、どれかを実行したり、全てを実行したりさせることが出来ます。

同じターゲットを持った二重コロンルールは実際には完全にお互いに分離されたものになります。それぞれの二重コロンルールは、違うターゲットを持つルールを実行するのと全く同じように個々に実行されます。

一つのターゲットのための二重コロンルールが複数ある場合はmakefileにある順番に実行されていきます。とはいえ二重コロンルールが実際に用を成すのには、中身のコマンドを実行する順番は問題になりません。

二重コロンルールはちょっと分かりにくい上、必ずしも便利ではありません。二重コロンは依存関係ファイルの如何によって更新するのとは違う方法でターゲットを更新させる場合にそのメカニズムを提供してくれますが、こうしたい場合というのは滅多にありません。

二重コロンのそれぞれにコマンドを指定すべきです。もし指定しないと、それを適用した場合に暗黙のルールが利用されることになります。詳しくは暗黙のルールの利用の項を見て下さい。

### 依存関係の自動生成

一つのプログラムのためのmakefileにおいて大体書く必要のあるルールの多くは、ただいくつかのオブジェクトファイルがいくつかのヘッダファイルに依存しているという事だけだったりします。例えば、`main.c'が#includeを使って`defs.h'を利用している場合、こう書く事になるでしょう。

```
main.o: defs.h
```

このルールは、`defs.h'が変更された時は常に`main.o'を更新しなければならない、とmakeに知らせるために必要です。大きなプログラムではmakefileでこんなルールを何十も書かなければならなくなる事は想像に難くないでしょう。それに#includeを加えたり外したりする度に、ちゃんとmakefileを更新させる事に注意深くならなければなりません。

こんな面倒な事を避けるために、現代のCコンパイラの大部分は、ソースの#include行を調べてこういうルールを自分で書いてくれる機能があります。一般的にコンパイラのオプションに`-M'をつけることで機能します。例えば、このコマンドがそうです。

```
cc -M main.c
```

こうすると、以下の内容を出力します。

```
main.o : main.c defs.h
```

こうすれば全部のルールを自分で書く必要はもはやありません。コンパイラがあなたの代わりにやってくれるのですから。

注意:makeが生成してくれる依存関係はmakefileでの`main.o'についての記述を構成してくれているのですが、暗黙のルールが探す中間ファイルを全く考慮していません。つまり利用後の中間ファイルさえ削除されないままになります。これについては暗黙のルールの連鎖の項ををを見て下さい。

古いmakeプログラムでは要求に応じて依存関係を生成させるというコンパイラの機能の利用には`make depend'のようなコマンドを使うことが伝統的な習慣でした。このコマンドでは`depend'というファイルを作成してその中に全ての依存関係の自動生成結果を格納します。そうすることでmakefileが内部で依存関係を読み取るのにincludeを使う事ができました(別のMakefileをインクルードするの項を参照)。

GNU makeではmakefileの更新という機能のおかげでこの習慣は時代遅れのものになりました――古いmakefileはどれも常に依存関係を生成し直してくれるので、もうわざわざmakeに依存関係を生成し直すのに明示的に命令する必要がないのです。どうMakefileが作り直されるのかの項を参照して下さい。

依存関係の自動生成に関しては、各ソースファイルに一致する一つのmakefileを作る習慣を持つことをお勧めします。`*name* .c'という各ソースファイルに対して`*name* .d'という一つのmakefileを用意してその中に`*name* .o'というオブジェクトファイルが依存するものをリストします。この方法をとるのは、ソースファイルに変更があったときに新しい依存関係を産出するのに再スキャン(中身を調べ直す事、を)する必要があるソースファイルだけにします。

下にあるのが`*name* .c'というCソースファイルから`*name* .d'という依存関係のファイル(すなわちmakefile)を生成させる型ルールです。

```
%.d: %.c
        $(SHELL) -ec '$(CC) -M $(CPPFLAGS) $< ¥
                      | sed '¥"s/¥($*¥)¥.o[ :]*/¥1.o $@ : /g'¥" > $@; ¥
                      [ -s $@ ] || rm -f $@'
```

型ルールの定義についての詳しい情報は型ルールの定義と再定義の項を見て下さい。`-e'フラグをシェルに与えると$(CC)コマンドが失敗すれば(0以外のステータスを返して)直ちに終了(exit)させます。シェルは普通、パイプライン(この場合はsed)に最後のコマンドのステータスを返すため、makeはコンパイラからの非ゼロのステータスには気づきません。

GNU Cコンパイラでは、`-M'フラグを使う代わりに`-MM'を使うこともできます。こうすると、システムヘッダファイルでは依存関係を省略します。詳細な情報について知りたい場合は*GNU CCの利用(Using GNU CC)*の`プリプロセッサ制御オプション(Options Controlling the Preprocessor)'の項を見て下さい。

sedコマンドの目的は読み替えさせる(translate)ことにあります。(例示すると、)

```
main.o : main.c defs.h
```

を、以下のように解釈します。

```
main.o main.d : main.c defs.h
```

これはつまりそれぞれの`.o'ファイルが依存している全ヘッダファイルとソースファイルを、相当する`.d'ファイルにも依存させるという事です。こうすることで、ソースファイルやヘッダファイルのどれかに変更があれば常に依存関係の生成をやり直さなければならない、とmakeに知らせます。

`.d'ファイルを更新するルールを一度定義してしまえば、あとは利用したいmakefileの中でincludeディレクディヴを使って中身を全て読ませる事ができます。別のMakefileをインクルードするの項を参照して下さい。例えばこうします。

```
sources = foo.c bar.c

include $(sources:.c=.d)
```

(この例では`foo.c bar.c'というソースファイルのリストを`foo.d bar.d'という依存関係のmakefileのリストとして読み替えるのに代入変数参照を使っています。代入参照についての全情報は代入参照の項にあります。) `.d'ファイルは他にもあるようなmakefileのことなので、さらに指示を与えなくてもmakeは必要であればそれらを更新します。どうMakefileが作り直されるのかの項を参照して下さい。

---

最初,前,次,最後のページ,目次 に移動.