Make

make es una herramienta de generación o automatización de código, muy usada en los sistemas operativos tipo Unix/Linux. Por defecto lee las instrucciones para generar el programa u otra acción del fichero makefile. Las instrucciones escritas en este fichero se llaman dependencias.

La herramienta make se usa para las labores de creación de fichero ejecutable o programa, para su instalación, la limpieza de los archivos temporales en la creación del fichero..., todo ello especificando unos parámetros iniciales (que deben estar en el makefile) al ejecutarlo.

Además de ser éste su objetivo principal, es utilizado para automatización de otras tareas como la creación de documentos del formato docbook, mantenimiento del sistema..., simplemente usando o creando makefiles que hagan estas tareas.

Estructura de un makefile

Los Makefiles son los ficheros de texto que utiliza make para llevar la gestión de la compilación de programas. Se podrían entender como los guiones de la película que quiere hacer make, o la base de datos que informa sobre las dependencias entre las diferentes partes de un proyecto. Todos los Makefiles están ordenados en forma de reglas, especificando qué es lo que hay que hacer para obtener un módulo en concreto. El formato de cada una de esas reglas es el siguiente:

```
objetivo: dependencias
    comandos
```

En “objetivo” definimos el módulo o programa que queremos crear, después de los dos puntos y en la misma línea podemos definir qué otros módulos o programas son necesarios para conseguir el “objetivo”. Por último, en la línea siguiente y sucesivas indicamos los comandos necesarios para llevar esto a cabo. Es muy importante que los comandos estén separados por un tabulador del comienzo de línea. Algunos editores como el mcedit cambian los tabuladores por 8 espacios en blanco, y esto hace que los Makefiles generados así no funcionen. Un ejemplo de regla podría ser el siguiente:

```
juego : ventana.o motor.o bd.o
     gcc –O2 –c juego.c –o juego.o
     gcc –O2 juego.o ventana.o motor.o bd.o –o juego
```

Para crear “juego” es necesario que se hayan creado “ventana.o”, “motor.o” y “bd.o” (típicamente habrá una regla para cada uno de esos ficheros objeto en ese mismo Makefile). En los siguientes apartados analizaremos un poco más a fondo la sintaxis de los Makefiles.

Comentarios en Makefiles

Los ficheros Makefile pueden facilitar su comprensión mediante comentarios. Todo lo que esté escrito desde el carácter “#” hasta el final de la línea será ignorado por make. Las líneas que comiencen por el carácter “#” serán tomadas a todos los efectos como líneas en blanco. Es bastante recomendable hacer uso de comentarios para dotar de mayor claridad a nuestros Makefiles. Podemos incluso añadir siempre una cabecera con la fecha, autor y número de versión del fichero, para llevar un control de versiones más eficiente.

Variables

Es muy habitual que existan variables en los ficheros Makefile, para facilitar su portabilidad a diferentes plataformas y entornos. La forma de definir una variable es muy sencilla, basta con indicar el nombre de la variable (típicamente en mayúsculas) y su valor, de la siguiente forma:

```
   CC = gcc –O2
```

Cuando queramos acceder al contenido de esa variable, lo haremos así:

```
   $(CC) juego.c –o juego
```

Es necesario tener en cuenta que la expansión de variables puede dar lugar a problemas de expansiones recursivas infinitas, por lo que a veces se emplea esta sintaxis:

```
   CC := gcc
   CC := $(CC) –O2
```

Empleando ":=" en lugar de "=" evitamos la expansión recursiva y por lo tanto todos los problemas que pudiera acarrear.

Además de las variables definidas en el propio Makefile, es posible hacer uso de las variables de entorno, accesibles desde el intérprete de comandos. Esto puede dar pie a formulaciones de este estilo:

```
   SRC = $(HOME)/src
   juego:
        gcc $(SCR)/*.c –o juego
```

Un tipo especial de variables lo constituyen las variables automáticas, aquellas que se evalúan en cada regla. Estas variables recuerdan a las variables usadas en los *scripts* de BASH. Los más importantes son:

```
   $@:Se sustituye por el nombre del objetivo de la presente regla.
   $*:Se sustituye por la raíz de un nombre de fichero.
   $<:Se sustituye por la primera dependencia de la presente regla.
   $^:Se sustituye por una lista separada por espacios de cada una de
las dependencias de la presente regla.
   $?:Se sustituye por una lista separada por espacios de cada una de
las dependencias de la presente regla que sean más nuevas que el
objetivo de la regla.
   $(@D):Se sustituye por la parte correspondiente al subdirectorio de
la ruta del fichero correspondiente a un objetivo que se encuentre en
un subdirectorio.
   $(@F):Se sustituye por la parte correspondiente al nombre del
fichero de la ruta del fichero correspondiente a un objetivo que se
encuentre en un subdirectorio.
```

Reglas virtuales

Es relativamente habitual que además de las reglas normales, los ficheros Makefile puedan contener reglas virtuales, que no generen un fichero en concreto, sino que sirvan para realizar una determinada acción dentro de nuestro proyecto software. Normalmente estas reglas suelen tener un objetivo, pero ninguna dependencia. El ejemplo más típico de este tipo de reglas es la regla "clean" que incluyen casi la totalidad de Makefiles, utilizada para "limpiar" de ficheros ejecutables y ficheros objeto los directorios que haga falta, con el propósito de rehacer todo la próxima vez que se llame a "make":

```
clean:
     rm –f juego *.o
```

Esto provocaría que cuando alguien ejecutase “make clean”, el comando asociado se ejecutase y borrase el fichero “juego” y todos los ficheros objeto. Sin embargo, como ya hemos dicho, este tipo de reglas no suelen tener dependencias, por lo que si existiese un fichero que se llamase “clean” dentro del directorio del Makefile, make consideraría que ese objetivo ya está realizado, y no ejecutaría los comandos asociados:

```
$ touch clean
$ make clean
make: `clean' está actualizado.
```

Para evitar este extraño efecto, podemos hacer uso de un objetivo especial de make, .PHONY. Todas las dependencias que incluyamos en este objetivo obviarán la presencia de un fichero que coincida con su nombre, y se ejecutarán los comandos correspondientes. Así, si nuestro anterior Makefile hubiese añadido la siguiente línea:

```
.PHONY : clean
habría evitado el anterior problema de manera limpia y sencilla.
```

Reglas implícitas

No todos los objetivos de un Makefile tienen por qué tener una lista de comandos asociados para poder realizarse. En ocasiones se definen reglas que sólo indican las dependencias necesarias, y es el propio make quien decide cómo se lograrán cada uno de los objetivos. Veámoslo con un ejemplo:

```
juego : juego.o
juego.o : juego.c
```

Con un Makefile como este, make verá que para generar “juego” es preciso generar previamente “juego.o” y para generar “juego.o” no existen comandos que lo puedan realizar, por lo tanto, make presupone que para generar un fichero objeto basta con compilar su fuente, y para generar el ejecutable final, basta con enlazar el fichero objeto. Así pues, implícitamente ejecuta las siguientes reglas:

```
cc –c juego.c –o juego.o
cc juego.o –o juego
```

Generando el ejecutable, mediante llamadas al compilador estándar.

Reglas patrón

Las reglas implícitas que acabamos de ver, tienen su razón de ser debido a una serie de “reglas patrón” que implícitamente se especifican en los Makefiles. Nosotros podemos redefinir esas reglas, e incluso inventar reglas patrón nuevas. He aquí un ejemplo de cómo redefinir la regla implícita anteriormente comentada:

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

Es decir, para todo objetivo que sea un “.o” y que tenga como dependencia un “.c”, ejecutaremos una llamada al compilador de C ($(CC)) con los modificadores que estén definidos en ese momento ($(CFLAGS)), compilando la primera dependencia de la regla ($<, el fichero “.c”) para generar el propio objetivo ($@, el fichero “.o”).

Invocando al comando make

Cuando nosotros invocamos al comando make desde la línea de comandos, lo primero que se busca es un fichero que se llama “GNUmakefile”, si no se encuentra se busca un fichero llamado “makefile” y si por último no se encontrase, se buscaría el fichero “Makefile”. Si no se encuentra en el directorio actual ninguno de esos tres ficheros, se producirá un error y make no continuará:

```
   $make
   make: *** No se especificó ningún objetivo y no se encontró ningún
makefile. Alto.
```

Existen además varias maneras de llamar al comando make con el objeto de hacer una traza o debug del Makefile. Las opciones “-d”, “-n”, y “-W” están expresamente indicadas para ello. Otra opción importante es “-jN”, donde indicaremos a make que puede ejecutar hasta “N” procesos en paralelo, muy útil para máquinas potentes.

Ejemplo de Makefile

La manera más sencilla de entender cómo funciona make es con un Makefile de ejemplo:

```
   CC := gcc
   CFLAGS := -O2
   MODULOS = ventana.o gestion.o bd.o juego
   .PHONY : clean install
   all : $(MODULOS) %.o : %.c
        $(CC) $(CFLAGS) –c $<.c –o $@
   ventana.o : ventana.c bd.o : bd.c gestion.o : gestion.c ventana.o
bd.o
        $(CC) $(CFLAGS) –c $<.c –o $@
        $(CC) $* -o $@
   juego: juego.c ventana.o bd.o gestion.o
        $(CC) $(CFLAGS) –c $<.c –o $@
        $(CC) $* -o $@
   clean:
        rm –f $(MODULOS)
   install:
        cp juego /usr/games/juego
```

Herramientas relacionadas con make

- Gcc
- automake
- GNU Make [1]
- makepp [2]

Bibliografía

- Robert Mecklenburg (Noviembre de 2004). *Managing Projects with GNU Make* [3] (pdf - Bajo licencia GFDL), 3ª edición (en inglés), O'Reilly, pp. 300. ISBN 0-596-00610-1.

Enlaces externos

- Principios básicos de make: El Makefile [4]
- Tutorial de GNU make [5]
- Manual libre de Make en PDF [6], cortesía de Gerardo Aburruzaga (profesor de la Universidad de Cádiz)]

Referencias

[1] http://www.gnu.org/software/make/
[2] http://makepp.sf.net/
[3] http://oreilly.com/catalog/make3/book/index.csp
[4] http://www.chuidiang.com/clinux/herramientas/makefile.php
[5] http://arco.inf-cr.uclm.es/~david.villa/doc/repo/make/make.html
[6] http://www.uca.es/softwarelibre/publicaciones/resolveUid/46ffcd2a383f2c5482645bca9d88e27a

Fuentes y contribuyentes del artículo

Make *Fuente*: http://es.wikipedia.org/w/index.php?oldid=27976120 *Contribuyentes*: .Sergio, Csuarez, DerHexer, Dusan, ElVaka, Emijrp, Gotrek, Hectorct, Kepa.bengoetxea, Peregring Lok0ooo0, Shooke, Willemo, 27 ediciones anónimas

Licencia