Introducción a GNU Make

Índice

1 Introducción a GNU Make

Es frecuente durante el desarrollo del proyecto que haya que compilar dicho proyecto multitud de veces. Además, el proyecto va cambiando, añadiéndose ficheros, bibliotecas con las que se debe enlazar, ejecutables y bibliotecas que se deben generar, etc. Repetir en cada ocasión las llamadas a gcc para compilar cada objetivo del proyecto es ineficiente y propenso a errores. La popular herramienta Make, diseñada para entornos UNIX, tiene como objetivo facilitar y, en cierta medida, automatizar este proceso de compilación del proyecto.

Hay distintas versiones de la herramienta Make, sin embargo todas ellas funcionan de un modo muy parecido. Cuando se invoca, Make interpreta un fichero de texto con una sintaxis especial, generalmente llamado *makefile* o *Makefile*, que le indica los pasos que debe seguir para compilar el proyecto. La sintaxis de este fichero permite expresar los pasos de compilación de forma sencilla y compacta, es sencillo añadir nuevos ficheros, objetivos de compilación, bibliotecas de enlace, etc. Una vez confeccionado el *makefile* repetir la compilación es tan sencillo como ejecutar make desde el terminal.

La estructura de un *makefile* se compone principalmente de tres cosas:

1. **Comentarios**. Los desarrolladores pueden facilitar la comprensión de los `makefiles` mediante la inclusión de 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.
2. **Definiciones de variables**. Una variable es un identificador que será sustituido por una cadena de texto, de forma que podemos utilizar el identificador y make lo sustituirá por el texto de su definición.
3. **Reglas**. Una regla indica los pasos a seguir para construir un objetivo.

Los siguientes apartados describen con algo de detalle la sintaxis de las definiciones de variables y las reglas, así como la invocación del comando make. Para obtener información adicional se recomienda al alumno que consulte la documentación oficial de GNU Make.

Variables

Es muy habitual que los *makefiles* utilicen variables 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 -Wall -pthread
```

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

```
$(CC) partial_sum1.c -o sum1
```

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) -Wall
```

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
sum1:
  gcc $(SRC)/*.c -o sum1
```

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

El formato de una regla es el siguiente:

```
objetivo: prerrequisitos
          comandos
```

Un *objetivo* es generalmente un fichero a crear, normalmente un ejecutable, una biblioteca o un fichero objeto. Los *prerrequisitos* son una serie de ficheros (que pueden ser objetivos de otras reglas) que deben estar presentes y actualizados para que make pueda crear el *objetivo*. Para construir el *objetivo* make ejecutará los *comandos* indicados en el cuerpo de la regla.

Make procesa una regla de la siguiente manera:

1. Primero comprueba los *prerrequisitos*. Cada *prerrequisito* es el nombre de un fichero que debe estar actualizado.
    - Si hay alguna regla para construir el *prerrequisito*, make procesará dicha regla (recursivamente) para actualizar el fichero.
    - Si no hay ninguna regla para crear el *prerrequesito*, pero hay un fichero con ese nombre, el *prerrequisito* queda cumplido.
    - Si no hay ninguna regla para crear el *prerrequisito* y el fichero no existe, make no puede completar la compilación y termina con error.
2. Make determina si debe aplicar la regla o no.
    - Si no existe ningún fichero con el nombre del *objetivo*, o existe pero es más antiguo que alguno de los *prerrequisitos*, entonces la regla debe aplicarse para actualizar el *objetivo*.
    - Si por el contrario el fichero existe y no es más antiguo que ninguno de los *prerrequisitos*, make no necesita aplicar la regla.
3. Si hay que aplicar la regla, make ejecutará los *comandos* escritos en el cuerpo de la regla. Es muy importante que los comandos estén separados por un tabulador del comienzo de línea. Algunos editores cambian los tabuladores por 8 espacios en blanco, y esto hace que los `makefiles` generados así no funcionen.

Un ejemplo sencillo de *makefile* con tres reglas podría ser el siguiente:

```
partial_sum1.o: partial_sum1.c
    gcc -Wall -g -c partial_sum1.c -o partial_sum1.o

partial_sum2.o: partial_sum2.c
    gcc -Wall -g -c partial_sum2.c -o partial_sum2.o

sum : partial_sum1.o partial_sum2.o
    gcc sum -o partial_sum1.o partial_sum2.o
```

En este ejemplo la regla principal es la que nos dice como crear el objetivo sum, que tiene como prerrequisitos partial_sum1.o y partial_sum2.o. Cada uno de estos tiene su propia regla, que indica cómo construirlos a partir de los ficheros fuente partial_sum1.c y partial_sum2.c respectivamente. Al procesar las reglas, y asumiendo que ninguno de los ficheros objetos está actualizado, make ejecutaría las órdenes:

```
   gcc -Wall -g -c partial_sum1.c -o partial_sum1.o
   gcc -Wall -g -c partial_sum2.c -o partial_sum2.o
   gcc sum -o partial_sum1.o partial_sum2.o
```

El orden en que aparecen las reglas en el *makefile* no tiene importancia, salvo para determinar la predeterminada, es decir, el objetivo que `make` construirá si no se le especifica ninguno en la línea de órdenes. Esta regla predeterminada es la primera que aparece en el `makefile` que no empiece por un punto. Por lo tanto, se escribirá el `makefile` de forma que la primera regla que se ponga sea la encargada de compilar el programa entero, o todos los programas que se describan. Es costumbre llamar a tal regla all.

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 sum *.o
```

Esto provocaría que cuando alguien ejecutase make clean, el comando asociado se ejecutase y borrase el fichero Suma y todos los ficheros objeto (la opción -f hace que clean se ejecute sin pedir confirmación de borrado, generar mensajes de diagnóstico o retornar con un error en caso de que los ficheros a borrar no existan). 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' is up to date.
```

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. Si además queremos evitar que errores en la ejecución de un comando paren la ejecución del `Makefile`, incluiremos un guión antes de los mismos:

```
clean:
  -rm sum *.o

$ make clean
rm sum
rm: cannot remove «sum»: No such file or directory
rm: cannot remove «*.o»: No such file or directory
make: [clean] Error 1 (no effect)
```

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:

```
sum1 : partial_sum1.o
partial_sum1.o : partial_sum1.c
```

Con un `Makefile` como este, make verá que para generar Suma1 es preciso generar previamente sumaparcial1.o y para generar sumaparcial1.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 partial_sum1.c -o partial_sum1.o
cc partial_sum1.o -o sum1
```

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 targets specified and no makefile found.  Stop.
```

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. Algunos de los argumentos más habituales empleados con la utilidad `make`, son los mostrados en la siguiente tabla:

ORDEN	EXPLICACIÓN
`make -f fichero`	El fichero de reglas es `fichero` en vez de `Makefile` o `makefile`, que son los nombres por defecto
`make -C directorio`	Cambiar al directorio y aplicar `make` al `makefile` que allí exista. Permite aplicar `make` recursivamente
`make -n`	Mostrar las órdenes que se mandarían ejecutar, pero sin ejecutarlas realmente. Es una ayuda para depuración
`make -d`	Muestra información de depuración además del procesamiento normal. Esta información dice qué ficheros están siendo considerados para ser rehechos, qué tiempos de ficheros están siendo comparados y con qué resultados, qué ficheros necesitan realmente ser rehechos, qué reglas implícitas están siendo tenidas en cuenta y cuáles se están aplicando: o sea, todo lo interesante sobre cómo `make` decide las cosas que tiene que hacer
`make -W fichero`	Pretende que el objetivo *fichero* acaba de ser modificado. Cuando se emplea con la opción -n, esto nos enseña lo que pasaría si fuéramos a modificar ese fichero. Sin -n, es casi lo mismo que ejecutar la orden *touch* en el fichero dado antes de dar la orden `make`, salvo en que el tiempo de modificación se cambia solamente en la imaginación de *fichero*.
`make objetivo`	Comenzar activando el objetivo especificado en vez del primero que se encuentre en `makefile`
`make var=valor`	Incluir la declaración de la variable `var` especificada; tiene preferencia sobre una posible declaración interna en `makefile`