Makefiles

C Prácticas

¿Como hacer un makefile?

En este pequeño tutorial os explico una de las muchas formas de cómo podéis hacer un makefile. Iremos haciéndolo cada vez más complejo. Podréis utilizar la versión que queráis para compilar vuestra práctica, pero os aconsejo que probéis todos, para comprender cómo se organiza el makefile y la sintaxis del mismo.

Para compilar un programa compuesto de múltiples ficheros, podemos hacer uso de la herramienta make, que utiliza un fichero makefile existente en el current directory. Es decir, si exsite un fichero makefile en el directorio, entonces make coge ese fichero, sino, hay que especificarle el nombre del fichero utilizando la opción -f

```
$ make   // coge el fichero makefile por defecto
$ make -f ordenesmake //coge el fichero ordenesmake
```

make se fijará qué fichero de los que componen el proyecto ha sido modificado. En las instrucciones que le damos en el fichero makefile definimos las dependencias que tienen los fichero entre sí, con lo que make podrá determinar que ficheros se ven afectados por la modificación de uno dado. En ese caso, lanzará todas las ordenes que necesite para recompilar y/o linkar sólo aquello que es estrictamente necesario.

Un proyecto de ejemplo muy simple

Supongamos que tenemos un proyecto formado por tres ficheros, uno donde está la función main() y otras funciones, otro fichero donde están una serie de funciones que van a ser usadas desde el main o las otras funciones y por último el fichero de encabezado de las funciones a usar. Veamos:

Fichero principal:

```
#include  <stdio.h>
#include "my_functions.h"

int main() {
  //Llamamos a una función definida en otro fichero, del que hemos puesto su include
  my_hello("Miguel");

  return(0);
}
```

Fichero con el código de las funciones:

```
#include <stdio.h>

void my_hello(char *nombre) {

  printf("Hola %s, Hello world!\n",nombre);

}
```

Fichero de cabecera con los prototipos de las funciones en el fichero de funciones:

```
// Este es el fichero de include que lista los prototipos de las funciones que están
definidas
// en el fichero .c del mismo nombre

void my_hello(char nombre[]);
```

Veamos varias versiones.

Para probar estas versiones, lo que podéis hacer es copiarlas en un directorio donde también tengáis los ficheros del proyecto. Para probar una versión en concreto, o bien hacéis uso de la opción -f del comando make, o copiáis la que os interese con el nombre makefile y usáis make sin parámetros.

make_one

```
helloworld:  helloworld.c my_functions.c
  gcc helloworld.c my_functions.c -o helloworld -I.
```

Esta es una versión muy simple que nos servirá para poder definir varias cosas:

Esta versión sólo tiene una regla (rule), la sintaxis de una regla es la siguiente:

```
target [target...] : [dependent ....]
/t [command ...]<br>
```

Importante: El /t significa que antes del comando debemos usar un tabulador, no valen espacios, es decir, en el ejemplo, los espacios que van antes de gcc son un tabulador. (Los puntos suspensivos significan que puede haber más).

- *target*: es el nombre de la regla, puede haber varios targets que tengan las mismas dependencias y los mismos comandos, es decir, se pueden llamas de distintas formas.
- *dependent*: son los ficheros dependientes de esta regla, es decir, los ficheros que si son modificados, entonces make ejecutará los comandos (command) que tiene la regla.
- *command*: son los comandos que se tienen que ejecutar identificado un cambio en los ficheros dependent.

Vemos pues que en esta primera versión tenemos una única regla llamada helloworld, que depende de modificaciones en los ficheros helloworld.c y my_functions.c.

Otra cosa importante es que make aplicará la primera regla por defecto al empezar a ejecutar a menos que llamemos a make con el nombre de una regla en concreto, en el siguiente ejemplo llamamos a make para que interprete el fichero mifichero y de el ejecute la regla miregla.

```
make miregla -f mifichero
```

Por tanto, la primera versión ejecutará la regla helloworld si los ficheros dependientes han sido modificados. Lo que ejecutará son los comandos que le siguen, en este caso sólo el comando

```
gcc helloworld.c my_functions.c -o helloworld -I.
```

que como vemos es llamar al compilador gcc tal como hemos usado hasta ahora. La única diferencia es el uso del parámetro -I del comando gcc. El parámetro -I determina el directorio donde se encuentran por defecto los ficheros de include que incluimos en nuestros ficheros .c, es decir aquellos que incluimos con #include "". Como vemos la ruta que se indica es . (punto, el current directory). RECORDAR en poner un tabulador antes de gcc

Si ejecutamos esta primera versión vemos que nos vuelca el comando que se ejecuta en consola y genera el fichero ejecutable helloworld. Si volvemos a ejecutar, nos dice que no hay nada que hacer (*helloworld is up to date*) puesto que no se ha cambiado nada.

make_two

En esta versión vamos a definir macros para usarlas en las reglas y comandos. Una macro no es mas que una constante definida al principio del makefile. Las macros tienen valores por defecto, por ejemplo, si utilizamos la macro CC sin haberla redefinido en nuestro makefile, el compilador que se usará por defecto es cc y no gcc.

```
CC=gcc
CFLAGS=-I.

helloworld:  helloworld.c my_functions.c
  $(CC) helloworld.c my_functions.c -o helloworld $(CFLAGS)

clean:
  rm helloworld
```

Aquí hemos definido dos macros,

- CC=gcc define el compilador a ser usado
- CFLAGS=-I. define los flags que se utilizan en el comando de compilación

Al usarlas se tiene que extraer su valor con $(macro) como si fueran variables del shell pero entre paréntesis.

También vemos que hemos creado otra regla, la hemos llamado clean, así si quremos eliminar el ejecutable creado, llamamos a nuestro makefile make_two con el nombre de la regla, por ejemplo así:

```
make clean -f make_two
```

Si copiamos make_two o cambiamos su nombre por makefile podemos llamarlo así para limpiar:

```
make clean
```

Si sólo llamamos con make sin nada mas se ejecuta la primera regla.

make_three

```
CC=gcc
CFLAGS=-I.
INC=my_functions.h
OBJECTS= helloworld.o my_functions.o

helloworld: $(OBJECTS)
  $(CC) $(OBJECTS) $(CFLAGS) -o helloworld

helloworld.o: helloworld.c $(INC)
  $(CC) -c helloworld.c  $(CFLAGS)

my_functions.o: my_functions.c
  $(CC) -c my_functions.c  $(CFLAGS)

clean:
  rm helloworld $(OBJECTS)
```

Aquí vemos que nuestra primera regla ahora hace uso de la macro OBJECTS, que son los ficheros objeto de los ficheros que componen el proyecto.

La diferencia respecto la versión anterior es que aquella, tanto si se modificaba un fichero .c u otro compilaba ambos. Aquí quiero definir reglas independientes para que se compilen sólo los ficheros que proceda. De forma que si modifico my_functions.c no tenga que compilarse helloworld.c si no ha cambiado.

Para ello mi regla helloworld la hago depender de los ficheros objeto, pero tengo que definir reglas para indicar cómo se tienen que compilar dichos ficheros objeto y que dependencias tienen.

Se ha definido otra macro, INC que define el nombre del único fichero de include que tenemos y del que en este caso depende el códigos de helloworld.c

Por tanto la regla helloworld depende de los ficheros helloworld.o y my_functions.o que a su vez hemos definido como reglas, creando así la dependencia entre reglas. La regla helloworld llamará a gcc (vía la macro) con los ficheros objeto, que deben ser por tanto generados en sus reglas respectivas.

La regla helloworld.o depende del fichero helloworld.c y del fichero de include my_functions.h que hemos asignado en la macro. En esta regla añadimos la opción -c al comando de compilación. Esta opción indica que sólo compile y no linke, es decir, que sólo genere el fichero .o. El fichero .o generado tendrá el mismo nombre del fichero .c. Si quisieramos cambiar el nombre del fichero .o podríamos hacerlo usando la opción -o .

La regla my_funcitions.o funciona de la misma manera pero sin la dependencia de my_functions.h.

Finalmente hemos añadido al comando de la regla clean los ficheros objeto vía la macro, de forma que al hacer clean borramos el ejecutable y todos los ficheros objeto.

make_four

```
CC=gcc
CFLAGS=-I.
INC = my_functions.h
OBJECTS= helloworld.o my_functions.o

helloworld: $(OBJECTS)
  $(CC) $^ -o $@ $(CFLAGS)

helloworld.o: helloworld.c $(INC)
  $(CC) -c $< -o $@ $(CFLAGS)
```

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

clean:
  rm helloworld $(OBJECTS)
```

En esta versión utilizamos las macros especiales $@, $^ y $<

- $@ se sustituye por el nombre de la regla.
- $^ se sustituye por los nombre de todas las dependencias separadas por espacios quitando las duplicadas.
- $< se sustituye el nombre del a primera dependencia

Por tanto, la regla helloworld compila todas las dependencias y genera el nombre de la regla como fichero ejecutable.

La regla helloworld.o compila la primera dependencia y genera el nombre de la regla. Esto se puede omitir puesto que el objeto cogerá el nombre de la primera dependencia (en este caso)

La regla my_functions.o funciona de igual forma.

make_five

En esta versión hemos utilizado pattern rules y hemos definido otra macro para fijar el nombre del ejecutable, PROGRAM.

```
CC=gcc
CFLAGS=-I.
INC = my_functions.h
OBJECTS= helloworld.o my_functions.o
PROGRAM= helloworld

$(PROGRAM): $(OBJECTS)
  $(CC) $^ -o $@ $(CFLAGS)

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

clean:
  rm $(PROGRAM) $(OBJECTS)
```

El uso de la macro PROGRAM está claro, le da nombre a la primera regla, además por tanto, como usamos $@ para el comando -o del compilador será finalmente el nombre del ejecutable.

Ahora utilizando reglas de patrón podemos simplificar el fichero makefile, puesto que para todos los ficheros .c queremos generar un .o.

La segunda regla, por tanto, es como si tuviéramos una regla .o para cada fichero .c. Se lanzará por ejemplo, como regla mifichero.o cuando el fichero mifichero.c sea modificado.

Además como usamos $(INC) la regla también se lanza cuando la dependencia my_functions.h es modificada, lanzando en ese caso la regla para todos los .c y generando todos los .o.

Es decir, con usa sola regla generamos todos los .o, para todos nuestros .c que además se indica que dependen del mismo .h. Si un fichero .c no depende realmente de un .h se obvia y no se lanza la regla.

← Entrada anterior

Entrada siguiente →

Deja un comentario

Conectado como alu28. Edita tu perfil. ¿Salir? Los campos obligatorios están marcados con *

Escribe aquí...

Publicar comentario »

Copyright © 2025 Sistemas Operativos
Escuela Politécnica Superior de Elche
Universidad Miguel Hernández
Miguel Onofre Martínez Rach