III Jornadas sobre el sistema operativo Linux: Introducción a Make

Francisco J. 曹 (Tsao) Santín
Grupo de Programadores y Usuarios de Linux- Coruña Linux Users Group
GPUL-CLUG

28 de Abril de 2003

Necesidad de Make
Ejemplo: Programa en un fichero

```
# include <stdio.h>
# include <math.h>
static float pi=3.14159;

main(){
  int i;
  float v[10];

  for(i=0;i<=9;i++){
  v[i]=xsen(0.25*i*pi);
  printf("v[%d]=%f \n",i,v[i]);
  }
}

xsen(x)
float x;
{
  float y;
```

```
  y=10.*sin(x);
  return(y);
    }
```

Generación de binario:

gcc -o unfichero -lm unfichero.c

Ejemplo:Programa principal en un fichero, funciones en biblioteca:

Fichero ampli.c

```
# include <stdio.h>

static float pi=3.14159;
extern float xsen(float);
main(){

   ...

   }
}
```

Fichero funciones.c

```
#include <math.h>
float xsen(float x){
  float y;
  y=10.*sin(x);
  return(y);
    }
```

```
float xcos(float x){
  float y;
  y=10.*cos(x);
  return(y);
    }
```

Generación de binario:

gcc -c funciones.c

gcc -c ampli.c

gcc -o ampli -lm ampli.o funciones.o

Ejemplo: Programa principal en un fichero, funciones en biblioteca, declaración de funciones usando ficheros de cabecera

```
# include <stdio.h>
# include "funciones.h"
static float pi=3.14159;

main(){
  int i;
  float v[10];

  ...

  }
}
```

Archivo de funciones

```
# include <math.h>
```

```
float xsen(float x){
  float y;
  ...
    }

float xcos(float x){
  float y;
  ...
    }
```

Archivo de cabeceras funciones.h

```
extern float xsen(float);
extern float xcos(float);
```

Generación de binario:
gcc -c ampli.c
gcc -c funciones.c

gcc -o ampli -lm ampli.o funciones.o

Utilizando fichero makefile:

```
ampli: ampli.o funciones.o
        gcc -o ampli -lm ampli.o funciones.o
ampli.o: ampli.c funciones.h
        gcc -c ampli.c
funciones.o: funciones.c
        gcc -c funciones.c
clean:
        rm ampli ampli.o funciones.o
```

¿Cómo procesa Make un fichero?

· toma el primer target como objetivo por defecto (excepto especificación de objetivo vg. phony target)

· lee los prerrequisitos

. si los targets de las sucesivas reglas son prerrequisitos, se procesarán

· una regla reconstruirá su target si los prerrequisitos son más recientes que él

Otra versión, usando variables:

```
objetos = ampli.o funciones.o
ampli: $(objetos)
        gcc -o ampli -lm $(objetos)
ampli.o: ampli.c funciones.h
        gcc -c ampli.c
funciones.o: funciones.c
        gcc -c funciones.c
clean:
        rm ampli $(objetos)
```

Tercera versión, con reglas implícitas

```
objetos = ampli.o funciones.o
ampli: $(objetos)
        gcc -o ampli -lm $(objetos)
ampli.o: funciones.h
funciones.o: # este target no tiene\
  prerrequisitos y la linea esta quebrada
clean:
        rm ampli $(objetos)
```

En definitiva ¿Qué encontramos en un makefile?

· reglas explícitas

· reglas implícitas

· definición de variables

· directivas, para: leer otros makefiles, control

. comentarios

Primeros cuidados en la escritura del make-file:

·Nombres: por defecto, GNUmakefile,makefile, Makefile.

Especificación con la opción -f, –file

·Tabuladores y slashes en lineas de comandos

·Espacios, los justos, en definición de variables ·@ para eliminar el echo del comando, - para evitar error

Incluir otros ficheros:

· La directiva include detiene la lectura del makefile y:

- llama a otros makefiles
- llama a ficheros de dependencias generados automáticamente

· ¿Dónde busca?

- el nombre empieza por slash
- directorio actual
- se ha especificado con la opción -I
- directorios por defecto:/usr/local/include,/gnu...

· Variables de entorno asociadas:

- MAKEFILES: ficheros que intenta leer pero si no encuentra no da error
- MAKEFILE_LIST: todos los ficheros incluidos de una u otra manera, el último nombre de la lista es el que está en proceso
- .VARIABLES: lista de todas las variables globales definidas en todos los ficheros procesados hasta el momento

Funcionamiento interno de Make:
· Primera fase: lee todos los makefiles, internaliza variables y reglas, y genera grafo de dependencia de targets y prerrequisitos; las variables que expanden en esta fase se dice que son de expansion inmediata
· Segunda fase: Decide que targets y reglas se usan; las variables aquí expandidas son de expansión diferida

Expansiones en asignaciones:
inmediata = diferida
inmediata ?= diferida
inmediata := inmediata
inmediata += diferida o inmediata
En condicionales: inmediata
En reglas: targets y prerrequisitos inmediata, comandos diferida

Reglas
Sintaxis:

targets:prerrequisitos;comandos

ó:

target:prerrequisitos
comandos

PARTE 1: Prerrequisitos

Tipos:

targets:prerrequisitos normales | prerrequisitos “order-only”

Los prerrequisitos “order-only” no fuerzan la actualización del target

Búsqueda de ficheros en árbol de directorios

Ejemplo 4:

```
progc/ampli.c makefile
progc/cf/funciones.c,funciones2.c makef
progc/hf/funciones.h,funciones2.h
```

Fichero makefile:

```
VPATH = cf hf
objetos = ampli.o funciones.o funciones2.o
ampli: $(objetos)
        gcc -o ampli -lm $(objetos)
include ./cf/makef

ampli.o: ampli.c funciones.h funciones2.h
         gcc -c ampli.c

clean:
          -rm ampli $(objetos)
```

Fichero makef:

```
funciones.o funciones2.o: funciones.c funciones2.f
        gcc -c $^
.PHONY:clean
clean:
        @echo Eliminando $(wildcard *.o)
        -rm *.o
```

Uso de wildcards (*,?,...):

· Asignación literal (comandos, prerrequisitos, targets): expansión por shell

· Usando función wildcard: expande en variables
uso: $ (wildcard patron)

·Variable VPATH
uso: VPATH = directorio 1:directorio 2:...

·Directiva vpath
uso: vpath patrón directorio (busca)
vpath patrón (elimina)
vpath (elimina todos)

Fichero makefile usando vpath:

```
vpath  %.c cf
vpath  %.h hf

objetos = ampli.o funciones.o funciones2.o
ampli: $(objetos)
        gcc -o ampli -lm $(objetos)
include ./cf/makef

ampli.o: ampli.c funciones.h funciones2.h
        gcc -c ampli.c

.PHONY:clean
clean:
        -rm ampli $(objetos)
```

Problema: las reglas no cambian con búsqueda
Usamos variables automáticas:

$ ^ prerrequisitos con el path completo

$ @ targets

$ < primer prerrequisito

Built - in targets:

.PHONY
.DEFAULT
.IGNORE
.SILENT
.EXPORT_ALL_VARIABLES
...

Proceso de reglas con patrones estáticos

uso:
target:patron de target:patron de prerrequisito
comandos

Ejemplo: fichero makef modificado

```
objetos=funciones.o funciones2.o
all: $(objetos)

$(objetos):%.o: %.c
        gcc -c $<
.PHONY: clean
clean:
        @echo Eliminando $(wildcard *.o)
        -rm *.o
```

PARTE 2: Comandos

·Recordar: Tab,;,#

·Ejecución: cada línea de comando abre un proceso independiente, si queremos encadenar los resultados de uno con el siguiente deben separarse con ;

·Ejecución en paralelo: -j n (numero de comandos a ejecutar a la vez)

·Opción -k para continuar aun fallando el anterior comando

· -i para ignorar errores en todos los comandos o guión en línea de comando a ignorar

Uso recursivo de make

Ejemplo: generando bibliotecas de funciones desde makefile

```
VPATH= cf hf
objetos = ampli.o funciones.o funciones2.o
ampli: $(objetos)
        gcc -o ampli -lm $^
biblios:
        cd cf && make -f makef all

ampli.o: ampli.c funciones.h funciones2.h
        gcc -c ampli.c

clean:
        -rm ampli $(objetos)
```

Paso de variables a sub-makes

· directiva export: añade la variable y su valor al entorno para que el sub-make inicialice su tabla de valores

· SHELL y MAKEFLAGS son exportadas siempre, MAKEFILES si contiene algo

· directiva unexport: evita la exportación de variables

· Variable MAKELEVEL: contiene el nivel de submake

· Variable MAKEFLAGS: recoge opciones de línea de comandos tales como -k, -s, pero descarta -C ,-f,-o,-W, caso especial -j

PARTE 3: Variables

· Contienen cadenas de texto

· $(nombre), ${nombre}, $n

Dos tipos

· Recursivamente expandidas: asignan valor mediante = y la directiva define, si contienen referencias a otras variables, su expansión se realiza en el momento de la substitución

· Simplemente expandidas: asignan su valor mediante := y se expanden inmediatamente en el momento de la definición

. El operador ?= asigna solo en caso de que no haya sido definida ya la variable

· Referencias de substitución: variable2=$(variable1:
patron1 = patron2)

Ejemplo:listac=$(listao: %.o = %.c)
Uso de función filter

· Referencias anidadas: x=y; y=z; a := $($(x)) asigna a a z

. Añadir texto a variables: +=
-no usar variable=$(variable) mastexto ¡bucle infinito!
-usar variable=... ;variable += mastexto

· ¡Cuidado con los espacios extra!

· Directiva override: anula asignación
uso: override variable=valor

· Variables de entorno: make las toma por defecto, las puede modificar (internamente), pero para que se mantenga la prioridad del entorno sobre el makefile, se usa -e (poco aconsejable)

· Variables sobre targets: asignaciones sólo válidas para target determinado:

Uso: target : VARIABLE = valor
target : prerrequisitos
También se pueden usar patrones de variables

PARTE 4: Condicionales:

· Estructuras del tipo ifeq(,) ... else ... endif

· También neq,ifdef,ifndef

· ¡No utilizar para controlar comandos en ejecución de shell!

Ejemplo: Enlazando código de C y Fortran:

```
progc/ampli.c ampli.f makefile configuracion
progc/cf/funciones.c funciones2.c makec
progc/ff/funciones.f funciones2.f makef
progc/hf/funciones.h funciones2.h ffunciones.h ffu
```

Fichero amplif.c

```
#include <stdio.h>
#include "./hf/ffunciones.h"
#include "./hf/ffunciones2.h"
static float pi=3.14159;
```

```
main(){
  int i;
  float v[10],z[10],val;

  for(i=0;i<=9;i++){
    val=0.25*i*pi;
  v[i]=xsen_ (&val);
  z[i]=xexp_ (&val);

  printf("v[%d]=%f \n",i,v[i]);
  printf("z[%d]=%f \n",i,z[i]);
  }
}
```

Fichero ffunciones.h

```
 float xsen_ ( float *x );
 float xcos_ (float *x);
```

Fichero funciones.f

```
      function xsen(x)
      real x,xsen

       xsen=10.*sin(x)
      return
      end


      function xcos(x)
      real x,xcos

       xcos=10.*cos(x)
      return
      end
```

Fichero makefile

```
VPATH= ./cf ./hf ./ff
tipo_l=$(shell cat configuracion)
ifeq ($(tipo_l),lenguajec)
objetos = ampli.o funciones.o funciones2.o
prerq = ampli.c funciones.h funciones2.h
```

```
endif
ifeq ($(tipo_l),fortran)
objetos = amplif.o funciones.o funciones2.o
prerq = amplif.c ffunciones.h ffunciones.h
endif

ampli: $(objetos)
        gcc -o ampli -lm $^
cbiblios:
        cd cf && make -f makec all
fbiblios:
        cd ff && make -f makef all

ampli.o: $(prerq)
        gcc -c $<
clean:
        -rm ampli $(objetos)
```

Fichero makef

```
objetos=funciones.o funciones2.o
```

```
all: $(objetos)

$(objetos):%.o: %.f
        g77 -c $<
.PHONY: clean
clean:
        @echo Eliminando $(wildcard *.o)
        -rm *.o
```

Fichero makec

```
objetos=funciones.o funciones2.o
all: $(objetos)

$(objetos):%.o: %.c
         gcc -c $<
.PHONY: clean
clean:
         @echo Eliminando $(wildcard *.o)
         -rm *.o
```

PARTE 5: Funciones para transformación de texto

· Uso: (funcion argumento1,argumento2 ...)

· Para análisis y sustitución de cadenas: subst, patsubst, filter, word, wordlist, words

· Para análisis de nombres: dir, notdir, suffix

· Avanzadas: foreach, if, call, value, eval

· Información sobre variables: origin (undefined,default, environment...)

· Comunicación con el entorno: shell

PARTE 6: Ejecución de Make

· Especificación de objetivos

· Nombres de objetivos standard: clean, all, install...

· Modificando el comportamiento de los comandos: -t,-n,...

· Testeando: -k

PARTE 7: Reglas implícitas

· No hay línea de comandos, make busca como generar el target

· Variables para indicar que comandos y opciones: CC,CFLAGS,FC, FFLAGS

· Uso de patrones vg.: %.o : %.c

Enlaces:

· www.gnu.org

Bibliografía:

· "GNU Make" por RMS, Roland McGrath, Paul Smith

·"Numerical recipes in Fortran/ in C /in Fortran 95"por varios autores