EBook Gratis

APRENDIZAJE makefile

Free unaffiliated eBook created from
Stack Overflow contributors.

#makefile

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: makefile

It is an unofficial and free makefile ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official makefile.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con makefile

Observaciones

Un *makefile* es un archivo de texto que controla el funcionamiento del programa `make` . El programa `make` se usa normalmente para administrar la creación de programas a partir de sus archivos de origen, pero se puede usar más generalmente para manejar cualquier proceso en el que los archivos (o *destinos*) deban regenerarse después de que otros archivos (o *requisitos previos*) hayan sido modificados. El *archivo make* describe la relación entre los objetivos y los requisitos previos, y también especifica los comandos necesarios para actualizar el objetivo cuando uno o más de los requisitos previos han cambiado. La única forma en que `make` determina "fuera de fecha" es comparando la hora de modificación de los archivos de destino y sus requisitos previos.

Los makefiles son algo únicos en algunos aspectos, lo que puede ser confuso inicialmente.

Primero, un makefile consiste en dos lenguajes de programación completamente diferentes en el mismo archivo. La mayor parte del archivo está escrito en un lenguaje que `make` que se pueda entender: esto proporciona asignación y expansión de variables, algunas capacidades de preprocesador (incluidos otros archivos, análisis condicional de secciones del archivo, etc.), así como la definición de objetivos y su prerrequisitos Además, cada objetivo puede tener una *receta* asociada que especifica qué comandos deben invocarse para hacer que ese objetivo se actualice. La receta está escrita como un script de shell (POSIX sh por defecto). El programa `make` no analiza este script: ejecuta un shell y pasa el script al shell que se ejecutará. El hecho de que las recetas no sean analizadas por `make` , sino que son manejadas por un proceso de shell separado, es fundamental para entender los makefiles.

En segundo lugar, un makefile no es un lenguaje de procedimiento como un script: como `make` analiza el makefile, construye un *gráfico dirigido* internamente donde los objetivos son los nodos del gráfico y las relaciones de requisitos previos son los bordes. Sólo después de que todos los archivos make han sido completamente analizada y la gráfica es completa se `make` elegir un nodo (objetivo) y tratar de llevarlo hasta la fecha. Para garantizar que un objetivo esté actualizado, primero debe asegurarse de que cada uno de los requisitos previos de ese objetivo esté actualizado, y así sucesivamente.

Versiones

Nombre	También conocido como	Versión inicial	Versión	Fecha de lanzamiento
Hacer POSIX		1992	IEEE Std 1003.1-2008, Edición 2016	2016-09-30
Hacer NetBSD	bmake	1988	20160926	2016-09-26

Nombre	También conocido como	Versión inicial	Versión	Fecha de lanzamiento
GNU hace	gmake	1988	4.2.1	2016-06-10
Marca SunPro	dmake	2006		2015-07-13
MSVS nmake		2003	2015p3	2016-06-27

Examples

Makefile básico

Considera escribir un "¡Hola mundo!" programa en c. Digamos que nuestro código fuente está en un archivo llamado source.c, ahora para ejecutar nuestro programa necesitamos compilarlo, generalmente en Linux (usando gcc) necesitaríamos escribir `$> gcc source.c -o output` donde salida Es el nombre del ejecutable que se generará. Para un programa básico, esto funciona bien, pero a medida que los programas se vuelven más complejos, nuestro comando de compilación también puede volverse más complejo. Aquí es donde entra un *Makefile* , los makefile nos permiten escribir un conjunto bastante complejo de reglas sobre cómo compilar un programa y luego simplemente compilarlo escribiendo make en la línea de comandos. Por ejemplo, aquí hay un posible ejemplo de Makefile para el ejemplo anterior de hello wold.

Makefile básico

Permite crear un *Makefile* básico y guardarlo en nuestro sistema en el mismo directorio que nuestro código fuente llamado *Makefile* . Tenga en cuenta que este archivo debe llamarse Makefile, sin embargo, el capitolio M es opcional. Dicho esto, es relativamente estándar usar un capitolio M.

```
output: source.c
    gcc source.c -o output
```

Tenga en cuenta que hay exactamente una pestaña antes del comando gcc en la segunda línea (esto es importante en makefiles). Una vez que este Makefile se escribe cada vez que los tipos de usuario make (en el mismo directorio que Makefile) make verificará si se ha modificado source.c (verifica la marca de tiempo) si se ha modificado más recientemente que la salida, se ejecutará La regla de compilación en la siguiente línea.

Variables en Makefiles

Dependiendo del proyecto, es posible que desee introducir algunas variables en su archivo make. Aquí hay un ejemplo de Makefile con variables presentes.

```
CFLAGS = -g -Wall
```

```
output: source.c
    gcc $< $(CFLAGS) -o $@
```

Ahora vamos a explorar lo que pasó aquí. En la primera línea, declaramos una variable llamada CFLAGS que contiene varias banderas comunes que tal vez desee pasar al compilador, tenga en cuenta que puede almacenar tantas banderas como desee en esta variable. Luego tenemos la misma línea que antes de indicar a make que verifique source.c para ver si se ha cambiado más recientemente que la salida, si es así, ejecuta la regla de compilación. Nuestra regla de compilación es casi la misma que antes, pero se acortó mediante el uso de variables, la variable `$<` está integrada en make (conocida como una variable automática, consulte *https://www.gnu.org/software/make/manual/ html_node / Automatic-Variables.html*) y siempre representa la fuente, por lo que en este caso *source.c* . `$(CFLAGS)` es nuestra variable que definimos anteriormente, pero tenga en cuenta que tuvimos que poner la variable entre paréntesis con $ delante como esta `$(someVariable)` . Esta es la sintaxis para decirle a Make que expanda la variable a lo que escribiste antes. Finalmente, tenemos el símbolo $ @, una vez más, esta es una variable integrada en make, y simplemente representa el objetivo del paso de compilación, por lo que en este caso significa *salida* .

Limpiar

Hacer limpio es otro concepto útil para aprender sobre crear archivos. Permite modificar el *Makefile* desde arriba

```
CFLAGS = -g -Wall
TARGETS = output

output: source.c
    gcc $< $(CFLAGS) -o $@

clean:
    rm $(TARGETS)
```

Como puede ver, simplemente agregamos una regla más a nuestro *Makefile* y una variable adicional que contiene todos nuestros objetivos. Esta es una regla un tanto común a tener en makefiles, ya que le permite eliminar rápidamente todos los archivos binarios que produjo simplemente escribiendo `$> make clean` . Al teclear make clean le dices al programa make que ejecute la regla de limpieza y luego make ejecutará el comando rm para eliminar todos tus objetivos.

Espero que esta breve descripción del uso de make le ayude a acelerar su flujo de trabajo. *Makefiles* puede volverse muy complejo, pero con estas ideas debería poder comenzar a usar make y tener una mejor comprensión de lo que está sucediendo en otros *Makefiles de* programadores. Para obtener más información sobre el uso de Make a excellent resource, visite https://www.gnu.org/software/make/manual/ .

Definiendo reglas

Inicio rápido

Una regla describe cuándo y cómo se crean ciertos archivos (**objetivos de** la regla). También puede servir para actualizar un archivo de destino si alguno de los archivos necesarios para su creación (**requisitos previos** del objetivo) es más nuevo que el de destino.

Las reglas siguen la sintaxis a continuación: (Tenga en cuenta que los *comandos* que siguen una regla están sangrados por una **pestaña**)

```
targets: prerequisites
        <commands>
```

donde los *destinos* y *prerrequisitos* son nombres de archivos o nombres reservados especiales y los *comandos* (si están presentes) son ejecutados por un shell para crear / reconstruir los *objetivos* que están desactualizados.

Para ejecutar una regla, simplemente puede ejecutar el comando `make` en el terminal desde el mismo directorio donde reside el *archivo Makefile* . Ejecutar `make` sin especificar el destino, ejecutará la primera regla definida en el *Makefile* . Por convención, la primera regla en el *Makefile a* menudo se llama *todo* o por *defecto* , comúnmente enumera todos los objetivos de compilación válidos como requisitos previos.

`make` solo ejecuta la regla si el objetivo está desactualizado, lo que significa que no existe o que su tiempo de modificación es anterior a cualquiera de sus requisitos previos. Si la lista de requisitos previos está vacía, la regla solo se ejecutará cuando se invoque por primera vez para construir los objetivos. Sin embargo, cuando la regla no crea un archivo y el destino es una variable ficticia, la regla siempre se ejecutará.

GNU hace

Reglas de patrón

Las reglas de patrón se utilizan para especificar múltiples destinos y construir nombres de requisitos previos a partir de nombres de destino. Son más generales y más potentes en comparación con las reglas comunes, ya que cada objetivo puede tener sus propios requisitos previos. En las reglas de patrón, una relación entre un objetivo y un requisito previo se construye en base a los prefijos que incluyen nombres de ruta y sufijos, o ambos.

Imagina que queremos construir los objetivos `foo.o` y `bar.o` , compilando los scripts C, `foo.c` y `bar.c` , respectivamente. Esto podría hacerse usando las reglas ordinarias a continuación:

```
foo.o: foo.c
    cc -c $< -o $@

bar.o: bar.c
    cc -c $< -o $@
```

donde *variable automática* `$<` es el nombre del primer requisito previo y `$@` el nombre del objetivo (aquí se puede encontrar una lista completa de variables automáticas).

Sin embargo, como los objetivos comparten el mismo sufijo, las dos reglas anteriores ahora pueden sustituirse por la siguiente regla de patrón:

```
%.o: %.c
    cc -c $< -o $@
```

Reglas implícitas

Las reglas implícitas dicen `make` cómo utilizar métodos habituales para construir ciertos tipos de archivos de destino, que se utilizan muy a menudo. `make` utiliza el nombre del archivo de destino para determinar qué regla implícita se debe invocar.

El ejemplo de regla de patrón que vimos en la sección anterior, en realidad no necesita ser declarado en un *Makefile* ya que `make` tiene una regla implícita para la compilación de C. Por lo tanto, en la siguiente regla, los requisitos previos `foo.o` y `bar.o` se compilarán utilizando la regla implícita para la compilación de C, antes de construir `foo` .

```
foo : foo.o bar.o
    cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Un catálogo de reglas implícitas y las variables utilizadas por ellas se pueden encontrar aquí .

regla genérica para gzip un archivo

Si un directorio contiene 2 archivos:

```
$ ls
makefile
example.txt
```

y `makefile` contienen el siguiente texto

```
%.gz: %
    gzip $<
```

entonces puede obtener `example.txt.gz` escribiendo en el shell

```
$ make -f makefile example.txt.gz
```

el makefile consisten de una sola regla que instruya *a hacer* cómo crear un archivo cuyo nombre final con *.gz* si existe un archivo con el mismo nombre pero con el sufijo *.gz.*

makefile hola mundo

C: \ makefile:

```
helloWorld :
[TAB]echo hello world
```

ejecutar resultados:

```
C:\>make
echo hello world
hello world
```

Nota: [TAB] debe reemplazarse por una pestaña real, stackoverflow reemplaza las pestañas por espacios, y los espacios no se usan igual que las pestañas en un archivo make.

Lea Empezando con makefile en línea: https://riptutorial.com/es/makefile/topic/1793/empezando-con-makefile

Capítulo 2: .FONY target

Examples

Usando .PHONY para objetivos sin archivos

Use `.PHONY` para especificar los objetivos que no son archivos, por ejemplo, `clean` o `mrproper` .

Buen ejemplo

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

Mal ejemplo

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

En el buen ejemplo, `make` sabe que `clean` no es un archivo, por lo tanto, no buscará si está o no actualizado y ejecutará la receta.

En el mal ejemplo, `make` buscará un archivo llamado `clean` . Si no existe o no está actualizada, ejecutará la receta, pero si existe y está actualizada, la receta no se ejecutará.

Uso de .PHONY para invocaciones recursivas del comando 'make'

El uso recursivo de make significa usar make como comando dentro de un makefile. Esta técnica es útil cuando un proyecto grande contiene subdirectorios, cada uno con sus respectivos makefiles. El siguiente ejemplo ayudará a comprender la ventaja de usar .PHONY con make recursivo.

```
/main
     |_ Makefile
     |_ /foo
            |_ Makefile
            |_ ... // other files
     |_ /bar
            |_ Makefile
            |_ ... // other files
     |_ /koo
            |_ Makefile
            |_ ... // other files
```

Para ejecutar el makefile del subdirectorio desde el makefile de main, el makefile del main debería tener un bucle como se muestra a continuación (hay otras formas en que esto puede lograrse, pero eso está fuera del alcance del tema actual)

```
SUBDIRS = foo bar koo

subdirs:
        for dir in $(SUBDIRS); do \
          $(MAKE) -C $$dir; \
        done
```

Sin embargo, hay trampas con este método.

1. Cualquier error detectado en una sub-marca es ignorado por esta regla, por lo que continuará construyendo el resto de los directorios, incluso cuando uno falla.
2. La capacidad de Make para realizar la ejecución paralela de múltiples objetivos de compilación no se utiliza ya que solo se usa una regla.

Al declarar los subdirectorios como objetivos .PHONY (debe hacer esto, ya que el subdirectorio obviamente siempre existe; de lo contrario, no se generará) estos problemas se pueden superar.

```
SUBDIRS = foo bar koo

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
        $(MAKE) -C $@
```

Lea .FONY target en línea: https://riptutorial.com/es/makefile/topic/5542/-fony-target

Capítulo 3: Makefile avanzado

Examples

Construyendo desde diferentes carpetas de origen a diferentes carpetas de destino

Principales características de este Makefile:

- Detección automática de fuentes C en carpetas especificadas.
- Múltiples carpetas de origen
- Múltiples carpetas de destino correspondientes para archivos de objetos y dependencias
- Generación automática de reglas para cada carpeta de destino.
- Creación de carpetas de destino cuando no existen.
- Gestión de dependencia con `gcc` : construir solo lo necesario
- Funciona en `Unix` y `DOS` .
- Escrito para GNU Make

Este Makefile se puede usar para construir un proyecto con este tipo de estructura:

```
\---Project
    +---Sources
    |   +---Folder0
    |   |       main.c
    |   |
    |   +---Folder1
    |   |       file1_1.c
    |   |       file1_1.h
    |   |
    |   \---Folder2
    |           file2_1.c
    |           file2_1.h
    |           file2_2.c
    |           file2_2.h
    \---Build
        |   Makefile
        |   myApp.exe
        |
        +---Folder0
        |       main.d
        |       main.o
        |
        +---Folder1
        |       file1_1.d
        |       file1_1.o
        |
        \---Folder2
                file2_1.d
                file2_1.o
                file2_2.d
                file2_2.o
```

Makefile

```
# Set project directory one level above of Makefile directory. $(CURDIR) is a GNU make
variable containing the path to the current working directory
PROJDIR := $(realpath $(CURDIR)/..)
SOURCEDIR := $(PROJDIR)/Sources
BUILDDIR := $(PROJDIR)/Build

# Name of the final executable
TARGET = myApp.exe

# Decide whether the commands will be shwon or not
VERBOSE = TRUE

# Create the list of directories
DIRS = Folder0 Folder1 Folder2
SOURCEDIRS = $(foreach dir, $(DIRS), $(addprefix $(SOURCEDIR)/, $(dir)))
TARGETDIRS = $(foreach dir, $(DIRS), $(addprefix $(BUILDDIR)/, $(dir)))

# Generate the GCC includes parameters by adding -I before each source folder
INCLUDES = $(foreach dir, $(SOURCEDIRS), $(addprefix -I, $(dir)))

# Add this list to VPATH, the place make will look for the source files
VPATH = $(SOURCEDIRS)

# Create a list of *.c sources in DIRS
SOURCES = $(foreach dir,$(SOURCEDIRS),$(wildcard $(dir)/*.c))

# Define objects for all sources
OBJS := $(subst $(SOURCEDIR),$(BUILDDIR),$(SOURCES:.c=.o))

# Define dependencies files for all objects
DEPS = $(OBJS:.o=.d)

# Name the compiler
CC = gcc

# OS specific part
ifeq ($(OS),Windows_NT)
    RM = del /F /Q
    RMDIR = -RMDIR /S /Q
    MKDIR = -mkdir
    ERRIGNORE = 2>NUL || true
    SEP=\\
else
    RM = rm -rf
    RMDIR = rm -rf
    MKDIR = mkdir -p
    ERRIGNORE = 2>/dev/null
    SEP=/
endif

# Remove space after separator
PSEP = $(strip $(SEP))

# Hide or not the calls depending of VERBOSE
ifeq ($(VERBOSE),TRUE)
    HIDE =
else
    HIDE = @
endif
```

```
# Define the function that will generate each rule
define generateRules
$(1)/%.o: %.c
    @echo Building $$@
    $(HIDE)$(CC) -c $$(INCLUDES) -o $$(subst /,$$(PSEP),$$@) $$(subst /,$$(PSEP),$$<) -MMD
endef

.PHONY: all clean directories

all: directories $(TARGET)

$(TARGET): $(OBJS)
    $(HIDE)echo Linking $@
    $(HIDE)$(CC) $(OBJS) -o $(TARGET)

# Include dependencies
-include $(DEPS)

# Generate rules
$(foreach targetdir, $(TARGETDIRS), $(eval $(call generateRules, $(targetdir))))

directories:
    $(HIDE)$(MKDIR) $(subst /,$(PSEP),$(TARGETDIRS)) $(ERRIGNORE)

# Remove all objects, dependencies and executable files generated during the build
clean:
    $(HIDE)$(RMDIR) $(subst /,$(PSEP),$(TARGETDIRS)) $(ERRIGNORE)
    $(HIDE)$(RM) $(TARGET) $(ERRIGNORE)
    @echo Cleaning done !
```

Cómo usar este Makefile Para adaptar este Makefile a su proyecto, debe:

1. Cambie la variable `TARGET` para que coincida con su nombre de destino
2. Cambie el nombre de las carpetas de `Sources` y `Build` en `SOURCEDIR` y `BUILDDIR`
3. Cambie el nivel de verbosidad del Makefile en el Makefile mismo o en make call
4. Cambie el nombre de las carpetas en `DIRS` para que coincidan con sus fuentes y construya carpetas
5. Si es necesario, cambia el compilador y las banderas.

Listas de cierre

GNU hace

Esta función de `pairmap` toma tres argumentos:

1. Un nombre de función
2. Primera lista separada por espacios
3. Segunda lista separada por espacios

Para cada tupla comprimida en las listas, llamará a la función con los siguientes argumentos:

1. Tuple elemento de la primera lista.
2. Tuple elemento de la segunda lista.

Se expandirá a una lista separada por espacios de las expansiones de funciones.

```
list-rem = $(wordlist 2,$(words $1),$1)
pairmap = $(and $(strip $2),$(strip $3),$(call \
    $1,$(firstword $2),$(firstword $3)) $(call \
    pairmap,$1,$(call list-rem,$2),$(call list-rem,$3)))
```

Por ejemplo, esto:

```
LIST1 := foo bar baz
LIST2 := 1 2 3

func = $1-$2

all:
    @echo $(call pairmap,func,$(LIST1),$(LIST2))

.PHONY: all
```

Imprimiré `foo-1 bar-2 baz-3` .

Lea Makefile avanzado en línea: https://riptutorial.com/es/makefile/topic/6154/makefile-avanzado

Capítulo 4: Reglas de patrón de GNU

Examples

Regla de patrón básico

Una regla de patrón se indica mediante un único carácter `%` en el objetivo. El `%` coincide con una cadena no vacía llamada **raíz** . El vástago se sustituye por cada `%` que aparece en la lista de requisitos previos.

Por ejemplo, esta regla:

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

Coincidirá con cualquier objetivo que termine en `.o` . Si el objetivo era `foo.o` , el vástago sería `foo` y compilaría `foo.c` a `foo.o` Se puede acceder a los objetivos y requisitos previos mediante variables automáticas.

Objetivos que coinciden con múltiples reglas de patrón

Si un objetivo coincide con varias reglas de patrón, make utilizará aquella cuyos requisitos previos existen o se pueden crear. Por ejemplo:

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@
```

Compilará `foo.c` a `foo.o` o ensamblará `foo.s` a `foo.o` , dependiendo de cuál de `foo.c` o `foo.s` exista.

Si varias reglas tienen requisitos previos que existen o se pueden crear, make utilizará la regla que coincida con la raíz más corta. Por ejemplo:

```
f%r:
    @echo Stem is: $*
fo%r:
    @echo Stem is: $*
```

Utilizará la segunda regla para hacer que el `foo.bar` objetivo, haciendo eco del `Stem is: o.ba`

Si varias reglas coinciden con la raíz más corta, make utilizará la primera en Makefile.

Directorios en reglas de patrón

Si el patrón de destino no contiene barras diagonales, make eliminará la parte del directorio del objetivo que está tratando de construir antes de hacer coincidir. El directorio se colocará delante

del vástago. Cuando se usa el vástago para crear el nombre de destino y los requisitos previos, la parte del directorio se elimina de él, el vástago se sustituye en lugar del `%` y, finalmente, el directorio se coloca delante de la cadena. Por ejemplo:

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

`lib/foobar.o` con `lib/foobar.o` , con:

- Tallo (`$*`): `lib/bar`
- Nombre de destino (`$@`): `lib/foobar.o`
- Prerrequisitos (`$<` , `$^`): `lib/foobar.c`

En este ejemplo, una regla `lib/foo%.o` tendría prioridad sobre la regla `foo%.o` porque coincide con una raíz más corta.

Reglas de patrón con múltiples objetivos

Las reglas de patrón pueden tener múltiples objetivos pero, a diferencia de las reglas normales, la receta es responsable de hacer todos los objetivos. Por ejemplo:

```
debug/%.o release/%.o: %.c
    $(CC) $(CFLAGS_DEBUG) -c $< -o debug/$*.o
    $(CC) $(CFLAGS_RELEASE) -c $< -o release/$*.o
```

Es una regla válida, que construirá objetos tanto de depuración como de liberación cuando uno de ellos deba construirse. Si escribimos algo como:

```
debug/%.o release/%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

Funcionaría cuando solo se `debug/*.o` uno de `debug/*.o` o `release/*.o` , pero solo generaría el primer objetivo (y consideraría que el segundo está actualizado) cuando ambos deben construirse.

Lea Reglas de patrón de GNU en línea: https://riptutorial.com/es/makefile/topic/6860/reglas-de-patron-de-gnu

Capítulo 5: Variables

Examples

Haciendo referencia a una variable

Para usar el valor almacenado en una variable, use el signo de dólar seguido del nombre de la variable entre paréntesis o llaves.

```
x = hello
y = $(x)
# y now contains the value "hello"
y = ${x}
# parentheses and curly braces are treated exactly the same
```

Si el nombre de una variable solo tiene un carácter, los paréntesis / llaves se pueden omitir (por ejemplo, `$x`). Esta práctica se usa para las variables automáticas (ver más abajo), pero no se recomienda para las variables de propósito general.

Variables simplemente expandidas

Las variables simplemente expandidas se comportan como variables de los lenguajes de programación tradicionales. Se evalúa la expresión en el lado derecho y el resultado se almacena en la variable. Si el lado derecho contiene una referencia de variable, esa variable se expande antes de que tenga lugar la asignación.

```
x := hello
y := $(x)
# Both $(x) and $(y) will now yield "hello"
x := world
# $(x) will now yield "world", and $(y) will yield "hello"
```

Una forma alternativa es usar la asignación de dos puntos:

```
x ::= hello
```

Asignación de colon simple y doble son equivalentes. POSIX make standard solo menciona la forma `::=` , por lo que las implementaciones con estándares estrictos pueden no ser compatibles con la versión de una sola coma.

Variables Recursivamente Expandidas

Cuando se define una variable expandida recursivamente, los contenidos del lado derecho se almacenan tal como están. Si una referencia de variable está presente, la referencia en sí se almacena (no el valor de la variable). Haga esperas para expandir las referencias de variables hasta que la variable se use realmente.

```
x = hello
y = $(x)
# Both $(x) and $(y) will now yield "hello"
x = world
# Both $(x) and $(y) will now yield "world"
```

En este ejemplo, la definición de `y` es recursiva. La referencia a `$(x)` no se expande hasta que se expande `$(y)` . Esto significa que siempre que el valor de `x` cambie, el valor de `y` también cambiará.

Las variables expandidas recursivamente son una herramienta poderosa pero fácilmente mal entendida. Se pueden usar para crear construcciones que se asemejan a plantillas o funciones, o incluso para generar automáticamente porciones de un archivo MAKE. También pueden ser la fuente de problemas difíciles de depurar. Tenga cuidado de usar solo las variables expandidas recursivamente cuando sea necesario.

Variables automáticas

En el contexto de una regla individual, Make define automáticamente una serie de variables especiales. Estas variables pueden tener un valor diferente para cada regla en un makefile y están diseñadas para simplificar la escritura de reglas. Estas variables solo se pueden utilizar en la parte de la receta de una regla.

Variable	Descripción
`$@`	Nombre de archivo del objetivo de la regla
`$%`	El nombre del miembro objetivo, si el objetivo de la regla es un archivo
`$<`	Nombre del archivo del primer prerrequisito
`$^`	Lista de todos los requisitos previos
`$?`	Lista de todos los requisitos previos que son más nuevos que el objetivo
`$*`	El "vástago" de una regla implícita o patrón

El siguiente ejemplo utiliza variables automáticas para generar una regla genérica. Estas instrucciones le indican cómo construir un archivo .o a partir de un archivo .c con el mismo nombre. Como no sabemos el nombre específico de los archivos afectados, usamos `$@` como marcador de posición para el nombre del archivo de salida y `$^` como marcador de posición para la lista de requisitos previos (en este caso, la lista de archivos de entrada).

```
%.o: %.c
    cc -Wall $^ -c $@
```

Asignación variable condicional

El operador `?=` Es una extensión que se comporta como `=` , excepto que la asignación *solo* ocurre si la variable aún no está establecida.

```
x = hello
x ?= world
# $(x) will yield "hello"
```

Agregar texto a una variable existente

El operador `+=` es una extensión común que agrega el contenido especificado al final de la variable, separado por un espacio.

```
x = hello
x += world
```

Las referencias de variables en el lado derecho se expandirán solo si la variable original se definió como una variable simplemente expandida.

Lea Variables en línea: https://riptutorial.com/es/makefile/topic/6191/variables

Creditos

S. No	Capítulos	Contributors
1	Empezando con makefile	chen, Community, Dan , Eldar Abusalimov, kdhp, levif, MadScientist, mox, mxenoph, SketchBookGames
2	.FONY target	kdhp, madD7, TimF
3	Makefile avanzado	Andrea Biondo, TimF
4	Reglas de patrón de GNU	Andrea Biondo, kdhp
5	Variables	bta