Compilación con GCC

Sebastián Santisi

2009-12-08

Resumen

Revisión sobre el proceso de compilación con GCC: Fases de compilación, parámetros de compilación, compilación estática, bibliotecas, utilidades, etc.

1. Fases de la compilación

Hacer:

```
$ gcc -o foo foo.c
```

realiza el preprocesamiento, la compilación, el ensamble y el enlace. Es equivalente a llamar a cada etapa independientemente:

```
$ cpp foo.c > foo.i
$ gcc -S foo.i
$ as foo.s -o foo.o
$ ld -lc -o foo foo.o
```

Es más, la primera instrucción no hace más que llamar a cada etapa independientemente. Esto puede comprobarse pidiéndole al GCC que no elimine los archivos temporales generados durante la compilación monolítica:

```
$ gcc -save-temps -o foo foo.c
```

2. Preprocesador

Desde la línea del compilador pueden definirse macros sin necesidad de escribir instrucciones `#define` en los archivos fuente. Para eso se usa el parámetro `-D`:

```
$ gcc -DDEBUG foo.c
$ gcc -DPI="3.1416" foo.c
$ gcc -DRECARGO="((100 + 20) / 100.0)" foo.c
$ gcc -DMSG='"Hola mundo"'
```

declaran una etiqueta `DEBUG` con el valor `1`, una etiqueta `PI` con el valor `3.1416`, una etiqueta `RECARGO` con la expresión `((100 + 20) / 100.0)` (+20 %), y una etiqueta `MSG` con la cadena `"Hola mundo"`, respectivamente.

Muchas veces el reemplazo ciego del preprocesador da lugar a errores no previstos al darles un contexto. Para ver la salida del preprocesador que entra al compilador puede usarse el modificador `-E`; el mismo instruye al GCC para que aborte la compilación luego de llamar al preprocesador y muestre la salida:

```
$ gcc -E foo.c
```

El preprocesador por omisión busca los encabezados de includes en los directorios `/usr/include` y `/usr/local/include` cuando los nombres se encuentran entre `<...>` y como rutas relativas al fuente cuando los mismos están entre `"..."`. Pueden agregarse directorios a la ruta de búsqueda de encabezados de dos formas diferentes, una es mediante parámetros al compilador y la otra mediante variables de entorno:

```
$ gcc -I. -I/home/user/lib foo.c
```

añade el directorio actual y el directorio `/home/user/lib` a la ruta de búsqueda de encabezados. Puede lograrse el mismo efecto mediante la variable `C_INCLUDE_PATH`:

```
$ C_INCLUDE_PATH=".:/home/user/lib" gcc foo.c
```

Si se quisiera conservar el valor de esta variable para toda la sesión, puede hacerse:

```
$ export C_INCLUDE_PATH=".:/home/user/lib"
$ gcc foo.c
```

Notar que múltiples rutas se separan por dos puntos.

3. Compilador

Para interrumpir la compilación luego de terminado el ensamblado y generar un código objeto, puede usarse el modificador `-c`:

```
$ gcc -c foo.c
```

Entre los parámetros referidos a qué compilar, se encuentran los de estándar:

```
$ gcc -pedantic -std=c99 foo.c
```

compila a `foo.c` utilizando el estándar ISO-C99 de forma estricta.

Entre los parámetros referidos a cómo compilar, se encuentran las optimizaciones que el compilador puede realizar (por omisión, no realiza absolutamente ninguna, lo cual genera un código bueno para depurar, pero harto ineficiente para ser ejecutado), y las optimizaciones propias de una plataforma determinada, entre otras. Por ejemplo:

```
$ gcc -O3 -march=pentium4 -mmmx -msse -msse2 foo.c
```

genera un binario orientado a correr en *Pentium4* y haciendo uso de las instrucciones *MMX*, *SSE* y *SSE2*; ejecuciones en procesadores no compatibles no funcionarán y pueden disparar señales de instrucciones no reconocidas (*SIGILL*). Además, esta compilación, activa las optimizaciones de velocidad más agresivas del GCC.

4. Enlazador

El enlazador es el encargado de juntar los diferentes fragmentos que conforman una aplicación y ordenarlos en un único ejecutable. Para ello se nutre de los códigos provenientes de los diferentes archivos de objetos así como de las bibliotecas. Cuando haya versiones dinámicas o estáticas de la misma biblioteca por omisión, el enlazador preferirá enlazar contra la dinámica.

```
$ gcc foo.o /usr/lib/libm.a
```

genera un ejecutable enlazando el código objeto `foo` con la biblioteca matemática de C (estática). Tal vez un resultado similar sería obtenido haciendo:

```
$ gcc foo.o -lm
```

donde la instrucción `-l` le dice al enlazador contra qué biblioteca enlazar. Que el resultado final sea el mismo o no, dependerá de si existe una versión dinámica de la biblioteca `m`; de haberla, se enlazará preferentemente contra ella. Para hacer que todas las bibliotecas se enlacen de forma estática puede hacerse:

```
$ gcc foo.o -lm -static
```

Eso sí, notar que no sólo la `libm` será estática sino también la `libc` (biblioteca estándar de C) y otras bibliotecas utilizadas por `foo.o`.

De manera similar al preprocesador, las bibliotecas se buscan por omisión en los directorios `/usr/lib` y `/usr/local/lib`. Pueden indicarse otros directorios mediante el parámetro `-L` o mediante la variable `LIBRARY_PATH`:

```
$ gcc -L/home/user/lib -lmyutils -lm foo.c -o foo
```

o

```
$ export LIBRARY_PATH="/home/user/lib"
$ gcc -lmyutils -lm foo.c -o foo
```

En el caso de que la compilación se realizara contra una biblioteca estática, entonces el ejecutable será autosuficiente para correr. En cambio, si se compilara contra una biblioteca dinámica, al querer ejecutarlo el sistema operativo buscará dichas bibliotecas en las ubicaciones predefinidas. Siendo que utilizamos un directorio no estándar, cuando queramos ejecutarlo tendremos que especificar la ubicación de `libmyutils.so`, para eso se usa la variable `LD_LIBRARY_PATH`:

```
$ LD_LIBRARY_PATH="/home/user/lib" ./foo
```

Dependiendo de la implementación del *linker*, puede o resolver las dependencias en un determinado orden, o ser capaz de analizar en qué orden debe enlazar los archivos. Si en el ejemplo anterior, la biblioteca `myutils` utilizara funciones implementados en la biblioteca `m`, la compilación:

```
$ gcc -L/home/user/lib -lm -lmyutils foo.c
```

podría no andar, dado que si primero se enlaza la biblioteca matemática, `myutils` aún tendrá dependencias insatisfechas al momento de enlazarla. El orden de enlace debe ser especificado de izquierda a derecha.

5. Bibliotecas

Las bibliotecas estáticas se generan archivando uno o más códigos objeto en un único archivo. La instrucción:

```
$ ar cr libfoo.a foo1.o foo2.o
```

agrega las nuevas versiones de `foo1.o` y `foo2.o` a la biblioteca `libfoo.a` si es que la misma existía previamente, o la crea con el contenido de ella (el parámetro `cr` es *create-replace*). Notar que una biblioteca siempre se nombra empezando con `lib`; cuando el linker busque a la biblioteca `foo`, va a buscar un archivo que se llame `libfoo.a` en el caso de la versión estática.

Puede usarse el archivador, también, para listar el contenido de una biblioteca:

```
$ ar t libfoo.a
```

Las bibliotecas dinámicas, al no embeber su código dentro del ejecutable enlazado, permiten su actualización o reemplazo sin necesidad de recompilarlo. Es requisito que una nueva versión de una biblioteca que reemplace a una precedente mantenga la misma interfaz, si esta se modificara, los ejecutables enlazados con ellas se arruinarían. Para mantener el versionado, cada versión sucesiva de una biblioteca lleva el nombre correspondiente a su versión, pero se añade un un nombre de biblioteca, llamado *soname*, el cual es el nombre de la versión mayor que dicha versión implementa. Dos bibliotecas con el mismo *soname* deberían ser compatibles entre sí. La generación de bibliotecas dinámicas requiere de una compilación especial y de un linkeo especial:

```
$ gcc -c foo.c -fPIC
$ ld -shared -soname libfoo.so.1 -o libfoo.so.1.0.0 -lc foo.o
```

El parámetro `-fPIC` al compilador hace que el mismo genere código independiente de la posición; esto es importante, dado que no hay un enlace en tiempo de compilación que ubique los segmentos de código con respecto a nuestro ejecutable por lo que podrían haber problemas de paginado después. La instrucción al enlazador le pide que genere una biblioteca dinámica (*shared object*), con el *soname* `libfoo.so.1` (esto es, la versión 1 de la biblioteca `foo`, según la misma convención de nombres que se usa para las bibliotecas estáticas), con el nombre real de `libfoo.so.1.0.0` (esto es, la versión 1.0.0), enlazando contra la biblioteca de C, y contra `foo.o`.

Al enlazar un ejecutable contra una biblioteca, el mismo la buscará por su *soname*. Como el nombre del archivo es otro, para que pueda encontrarla, se hace accesible a la biblioteca con los nombres restantes; esto se realiza mediante la creación de enlaces simbólicos de los nombres a usarse contra el archivo real:

```
$ ln -sf libfoo.so.1.0.0 libfoo.so.1
```

genera un enlace simbólico llamado `libfoo.so.1` (*soname*) apuntando al archivo `libfoo.so.1.0.0`.

Para ver qué bibliotecas dinámicas necesita un archivo para funcionar puede utilizarse el programa `ldd`:

```
$ ldd foo
```

lista las dependencias dinámicas del archivo `foo`. Si alguna dependencia no se encuentra en las rutas por omisión, puede utilizarse la variable de entorno `LD_LIBRARY_PATH`.

Para poder ver los símbolos exportados por un programa, una biblioteca o un código objeto se utiliza el comando `nm`:

```
$ nm libfoo.a
```

Con el comando `strip` puede eliminarse dicha tabla de símbolos.

6. Bibliografía

- "An Introduction to GCC", Brian Gough. ISBN: 0-9541617-9-3.
  `http://www.network-theory.co.uk/docs/gccintro/index.html`
- "Shared objects for the object disoriented!", Ashish Bansal.
  `http://www.ibm.com/developerworks/library/l-shobj/`