



Las mejores prácticas con autotools

1. Las mejores prácticas con autotools

El núcleo de la cadena de compilación GNU -- el conjunto de herramientas usadas para construir paquetes de software GNU -- es lo que se conoce como "autotools" un término que alude a los programas autoconf y automake, así como a libtool, autoheader, pkg-config y, a veces, gettext. Estas herramientas permiten compilar software GNU en una amplia variedad de plataformas, y en sistemas operativos Unix o de tipo Unix, proporcionando a los desarrolladores un marco para comprobar la presencia de librerías, funciones y herramientas que quieran usar. Mientras que las autotools son extraordinarias en manos de un desarrollador con experiencia, pueden ser demasiado para quien las usa por primera vez, y no es tan raro encontrar paquetes proporcionados con soporte para autotools que a pesar de funcionar está mal hecho. Este artículo cubre algunos de los errores más comunes que se cometen al usar autotools y algunas formas para alcanzar mejores resultados.

A pesar de la opinión que cualquiera pueda tener acerca de ellas, actualmente no disponemos de ninguna alternativa válida a autotools. Proyectos tales como Scons no son tan válidos en cualquier plataforma como autotools y no envuelven el suficiente conocimiento como para ser útiles de momento. Tenemos montones de comprobaciones automáticas con autotools, y muchas librerías vienen con otra librería m4 para poder comprobar su presencia.

La estructura básica de un proyecto elaborado con autotools es simple. Autoconf toma la ayuda de un archivo `aclocal.m4` (creado por `aclocal` usando las librerías m4 en su ruta de búsqueda y del archivo `acinclude.m4`) para crear el archivo `configure.ac` (anteriormente `configure.in`) y lo transforman en el archivo de comandos "configure". Para cada directorio que se encuentre en el mismo debe existir un `Makefile.am` que `automake` usa para crear las plantillas `Makefile.in`. Las plantillas `Makefile.in` son procesadas por la macro `configure` que las transforma en `Makefiles` reales. Se puede evitar usar `automake` escribiendo nuestros propios archivos `Makefile.in`, pero esto es demasiado complejo y se pierden algunas características de las autotools.

En un archivo `configure.ac` se pueden usar macros que definimos nosotros mismos, aquellas que proporcionan por defecto `autoconf` y `aclocal`, o macros externas proporcionadas por otros paquetes, por ejemplo. En este caso, `aclocal` creará el archivo `aclocal.m4` añadiendo los archivos de librerías que encuentra en las librerías del sistema con las macros definidas; este es un paso crítico para tener un proyecto con autotools que funcione correctamente, como veremos en un momento.

[Imprimir](#)

Actualizado 16 de diciembre, 2005

Sumario: Este artículo cubre algunos de los errores más comunes que se cometen al usar autotools y muestra cómo obtener mejores resultados.

Diego Pettenò
Autor

LinuxBlues
Traductor

Donate to support our development efforts.

[Make a Donation](#)

OSL
OPEN SOURCE LAB

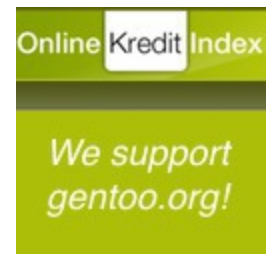
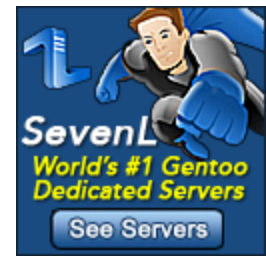
Un `Makefile.am` es principalmente una declaración de intenciones: Se pueden rellenar algunas variables de objetivos con el nombre de los objetivos que pretendemos construir. Estas variables se encuentran estructuradas en un formato del tipo `lugarainstalar TIPODEOBJETIVO`. El lugar es la localización en un sistema de ficheros jerárquico Unix (`bin`, `lib`, `include`, ...), una palabra clave no usada que puede ser definida con una ruta arbitraria (usando la variable `keyworddir`), o la palabra clave especial `noinst` que apunta a los objetivos que no necesitan ser instalados (cabeceras privadas por ejemplo o librerías estáticas usadas durante la construcción). Después de denominar al objetivo, se puede usar el nombre (reemplazando los puntos por caracteres de subrayado) como el prefijo para las variables que afectan a su construcción. De esta forma se pueden proporcionar variables `CFLAGS`, `LD_FLAGS` y `LDADD` especiales usadas durante la construcción de un único objetivo, en lugar de cambiarlas para todos los objetivos. Se pueden usar también variables recogidas en la fase `configure`, si se le pasan a la macro `AC_SUBST` en `configure.ac`, para que puedan ser reemplazadas en los `makefiles`. Además, aunque definir las `CFLAGS` y `LD_FLAGS` en base al objetivo es muy útil, añadir parámetros estáticos en el `Makefile.am` no es adecuado para obtener una mayor portabilidad, dado que no puede saberse de antemano si el compilador que se está usando los soporta, o si realmente se necesitan (`-ldl` indicado en las `LD_FLAGS` es un buen ejemplo de un parámetro necesario en Linux pero no en FreeBSD); en dichos casos, se debe usar `configure.ac` para añadir dichos parámetros.

Las macros más frecuentemente usadas en `configure.ac` son `AC_CHECK_HEADERS`, `AC_CHECK_FUNCS`, y `AC_CHECK_LIB`, que comprueban respectivamente la presencia de, algunos archivos de cabeceras, algunas funciones en librerías, y una librería concreta (que proporciona una función específica). Son importantes para la portabilidad dado que proporcionan una forma de comprobar qué cabeceras están presentes y cuáles no (por ejemplo, cabeceras del sistema que están ubicadas de forma diferente en diferentes sistemas operativos), y para comprobar si una función está presente en una librería del sistema (`asprintf()` no está en OpenBSD por ejemplo, mientras que sí está en la librería GNU C y FreeBSD), y finalmente para comprobar la presencia de una librería de terceros o para ver si un enlace específico a una librería es necesario para obtener determinadas funciones (por ejemplo, la función `dlopen()` está en la librería `libdl` en los sistemas GNU, mientras que se proporciona en la librería C del sistema en FreeBSD).

Además de comprobar la presencia o ausencia de funciones o cabeceras (y a veces de librerías) se suele necesitar habitualmente cambiar la ruta del código (para evitar el uso de funciones que no se encuentran, o para encontrar otras que las reemplacen, por ejemplo). `Autoconf` se encuentra muy a menudo emparejado con otra herramienta, `autoheader`, que crea una plantilla `config.h.in`, usada por el archivo de comandos `configure` para crear una cabecera `config.h` donde se definen macros del preprocesador de la forma `HAVE_funcióndada` o `HAVE_cabeceradada_H` que pueden comprobarse con las directivas `#ifdef/#ifndef` dentro del código fuente de un programa en C o C++ para modificar el código de acuerdo con las características presentes.

He aquí algunas practicas que es necesario tener presentes para crear el código más portable posible con `autotools`.

el archivo de cabecera `config.h` debe considerarse un archivo de cabecera interna, por lo que deberá ser usado únicamente por el paquete que lo ha creado. Se debe evitar editar la plantilla `config.h.in` para añadir nuestro código en ella, dado que ello requerirá actualizarlo manualmente de acuerdo con el `configure.ac` que estamos escribiendo.



Desafortunadamente, algunos proyectos, como Net-SNMP, exportan esta cabecera con las cabeceras de otras librerías, lo cual requiere que cualquier proyecto que use sus librerías deba incluirlas (o proporcionar su propia copia de las estructuras internas de Net-SNMP). Esto es un gran inconveniente, dado que la estructura de la librería de un proyecto con autotools debe ser invisible para el software que la esté utilizando (que puede no usar autotools en absoluto). Además, los cambios en el comportamiento de las autotools no son nada raro, por lo que pueden haber dos comprobaciones idénticas con diferentes resultados dependiendo de la forma en la que se ejecuten. Si se necesitan definir nuestros propios envoltorios o reemplazos en el caso de que algo no se encuentre en el entorno para el que estamos compilando, debe hacerse en cabeceras privadas que no se instalen (declaradas como `noinst_HEADERS` en el archivo `Makefile.am`).

Proporcionar siempre los archivos m4 que se hayan usado. Dado que las autotools se han usado durante años, muchos paquetes (por ejemplo librerías) que pueden ser reutilizados por otros programas, proporcionan un archivo de librería m4 en `/usr/share/aclocal` que hace posible comprobar su presencia (por ejemplo, al usar las macros `-config`) con una simple llamada en una macro. Estos archivos los usa `aclocal` para crear el archivo `aclocal.m4`, y están presentes en los sistemas de los desarrolladores donde `aclocal` se ejecuta para crear la revisión o versión, pero cuando se trata de dependencias opcionales, pueden no estar en los sistemas de los usuarios. Aunque esto no constituye ningún problema, dado que los usuarios raramente ejecutan `aclocal`, es un serio problema en distribuciones basadas en código fuente como Gentoo, donde a veces es necesario crear parches para `Makefile.am` o `configure.ac` y después volver a ejecutar `autoconf` sin tener todas las dependencias opcionales instaladas (o teniendo versiones diferentes, que pueden ser incompatibles, o tener errores, del mismo archivo m4).

Para evitar este problema, debe crearse un subdirectorio m4 en el directorio del paquete y añadir posteriormente los archivos de librería m4 que se estén utilizando. Después es necesario llamar a `aclocal` con las opciones `aclocal -I m4` para buscar en ese directorio antes que en el de librerías del sistema. Entonces podemos elegir si poner ese directorio bajo control de revisión (CVS, SVN, o cualquier otro que se esté usando) o crearlo únicamente para las revisiones. El segundo caso es el mínimo requisito indispensable para un paquete. Con ello minimizamos la cantidad de código que necesita estar bajo control de revisión y asegura que siempre se estará usando la última versión m4, aunque tiene el inconveniente de que cualquiera que compruebe nuestro repositorio no será capaz de ejecutar `autoconf` sin obtener un tarball de la revisión para obtener el m4 del mismo (y con ello podría no funcionar, dado que hemos podido actualizar el `configure.ac` para ajustarse a alguna nueva macro o añadido más dependencias). Por otra parte, poner el directorio m4 bajo control de revisión a veces tienta a los desarrolladores a cambiar las macros para ajustarlas a sus necesidades. Aunque parece muy lógico, dado que los archivos m4 están bajo nuestro control de revisión, causará trastornos a muchos mantenedores de paquetes, dado que a veces las nuevas versiones de los archivos m4 corrigen errores o soportan nuevas opciones y rutas de instalación (por ejemplo, configuraciones multilib), y tener archivos m4 modificados hace imposible reemplazarlos con las nuevas versiones sin más. Lo cual también significa que cuando se actualiza un archivo m4 se deben rehacer todas las modificaciones con respecto al original.

Los archivos m4 son siempre un problema con el que trabajar. Pueden replicar casi el mismo código de librería en librería (dependiendo de la forma en que necesitemos proporcionar las `CFLAGS/LDFLAGS`: con comprobaciones o con una macro `-config`). Para

evitar este problema los proyectos GNOME y FreeDesktop desarrollaron una herramienta denominada pkg-config, que proporciona tanto un binario ejecutable como un archivo m4 para incluirlo en los archivos configure.ac, y permite a los desarrolladores comprobar la presencia de una librería dada (y/o paquete), suponiendo que el paquete haya instalado un archivo de datos .pc de pkg-config. Esta solución hace mucho más simple el trabajo de mantener los archivos de comandos configure.ac, y requiere un tiempo mucho menor de procesamiento mientras se ejecuta la macro configure, dado que usa la información proporcionada por el paquete instalado en lugar de intentarlo si es que se encuentra presente. Por otra parte, esta solución significa que cualquier error que cometa un desarrollador con respecto a una dependencia puede dañar el programa del usuario, dado que únicamente retocan los parámetros del compilador y del enlazador en el archivo de datos y la macro configure no comprueba si la librería funciona. Afortunadamente esto no ocurre muy a menudo.

Para crear el archivo configure, es necesario PKG_CHECK_MODULES, contenido en la librería pkg.m4. Se debe añadir dicho archivo a nuestro directorio m4. Si la dependencia pkg-config es obligatoria (dado que la macro configure ejecuta esta herramienta) no se puede tener la certeza de que nuestro archivo m4 sea el mismo que se encuentre en el sistema de los usuarios, así como tampoco de que no incluya otros errores, dado que puede ser anterior a la nuestra.

Comprobar siempre las librerías con las que se va a enlazar, si se tienen como dependencias obligatorias. Generalmente las macros autoconf o los archivos de datos pkg-config definen los pre-requisitos de librerías que se necesitan enlazar con nuestra librería. Además, algunas funciones que se encuentran presentes en algunas otras librerías extras (como dlopen() en libdl en Linux y Mac OS X) pueden estar en la libc de otros sistemas (esta misma función se encuentra presente en la libc de FreeBSD). En estos casos, se necesita comprobar dónde se encuentra la función sin enlazar a nada por el momento o si se necesita usar una librería específica (para evitar enlazar a una libdl inexistente que fallará cuando no sea necesaria, por ejemplo).

Ser cuidadoso con las extensiones GNU. Una de las cosas que hacen realmente difícil la portabilidad es el uso de funciones de extensión, proporcionadas por la libc de GNU, pero no por otras librerías C de otros sistemas como BSD o uClibc. Cuando se usan dichas funciones, ha de proporcionarse algún tipo de "reemplazo al vuelo", una función que proporcione la misma funcionalidad que la de la librería, probablemente sin el mismo rendimiento o seguridad, que pueda ser usado cuando la función de extensión no esté presente en la librería C del sistema. Estas funciones deben estar protegidas por un bloque #ifdef HAVE_función ... #endif, para que no se dupliquen cuando se encuentren ya presentes. Hay que asegurarse de que estas funciones no son exportadas por la librería a usuarios externos; deben ser declaradas dentro de una cabecera interna, para evitar romper otras librerías que estén haciendo trucos similares.

Evitar compilar código específico de un SO cuando no se necesite. Cuando un programa soporta librerías específicas o un sistema operativo específico, no es raro tener archivos completos de código que son específicos para los mismos. Para evitar compilarlos cuando no se necesiten, se usa la macro AM_CONDITIONAL dentro de un archivo configure.ac. Esta macro automake (únicamente usable si se está usando automake para construir el proyecto) nos permite definir bloques if ... endif dentro de un archivo Makefile.am, dentro del cual se pueden definir variables especiales. Se puede, por ejemplo, añadir una variable "platformsrcs" en la que podemos ajustar el archivo de código

fuente correcto para el tipo de plataforma para la que estamos compilando, para usarla después en una variable `_SOURCES`.

De cualquier modo, hay dos errores muy comunes que los desarrolladores cometen cuando usan `AM_CONDITIONAL`. El primero es el uso de `AM_CONDITIONAL` en una rama que ya es condicional (por ejemplo, en el caso de un conmutador `info` o `case`), lo que conduce a que `automake` se queje de que un condicional se ha definido sólo condicionalmente (`AM_CONDITIONAL` debe ser llamado sólo con un alcance global, fuera de cada bloque `if`, por lo que debe definirse una variable que contenga el estado de las condiciones para después hacer todas las comprobaciones cuando se llame a `AM_CONDITIONAL`). El otro es que no se pueden cambiar las variables de los objetivos directamente, y deben definirse variables de "materia", cuyos resultados estén fuera del condicional, para añadir o eliminar objetivos o archivos de código fuente.

Muchos proyectos, para evitar compilar el código por rutas específicas de código, añaden los archivos completos en condicionales del preprocesador `#ifdef ... #endif`. Mientras que esto suele funcionar, hace el código más feo y más propenso a errores, dado que una sola sentencia fuera del bloque condicional puede ser compilada donde no se necesita en el archivo de código fuente. También engaña a los usuarios a veces, dado que los archivos de código fuente parecen ser compilados en situaciones en las que no tienen ningún sentido.

Tratar de hacer lo mejor posible la búsqueda de sistemas operativos o

plataformas hardware. A veces se necesita buscar un sistema operativo o plataforma hardware específicos. La forma correcta de hacerlo depende de dónde en concreto se necesite saberlo. Si se necesita saberlo para habilitar comprobaciones especiales en `configure`, o si se necesitan añadir objetivos adicionales en los `makefiles`, se debe hacer la comprobación en `configure.ac`. Por otra parte, si debe saberse la diferencia en un archivo de código fuente, por ejemplo, para habilitar una función opcional con código en ensamblador, se debe confiar directamente en el compilador/preprocesador, por lo que se deben usar directivas `#ifdef` con las macros habilitadas por defecto en la plataforma de destino (por ejemplo `__linux__`, `__i386__`, `_ARC_PPC`, `__sparc__`, `_FreeBSD_` y `__APPLE__`).

No ejecutar comandos en `configure.ac`. Si se necesita comprobar el hardware o el sistema operativo en un `configure.ac`, se debe evitar el uso del comando `uname`, a pesar de que esta sea una de las formas más comunes de hacerlo. Actualmente es un error, dado que rompe la compilación cruzada. `Autotools` soporta compilación cruzada de una máquina a otra usando definiciones de anfitriones: cadenas de tipo "hardware-vendor-os" (actualmente, "hardware-vendor-os-libc" cuando se usa GNU libc), tales como `i686-pc-linux-gnu` y `x86_64-unknown-freebsd5.4`. `CHOST` es la definición del anfitrión del sistema para el que se está compilando el software, `CBUILD` es la definición del anfitrión del sistema en el que se está compilando; cuando `CHOST` y `CBUILD` difieren, estamos haciendo compilación cruzada.

En los ejemplos anteriores, la primera definición del anfitrión hace referencia a un sistema con un procesador equivalente a un `pentium2` (o posterior), ejecutando un núcleo Linux con una libc GNU (por lo general se refiere a un sistema GNU/Linux). El segundo se refiere a un sistema AMD64 con un sistema operativo FreeBSD 5.4. (Para un sistema GNU/kFreeBSD, que usa el núcleo FreeBSD y GNU libc, la definición del anfitrión es `hw-unknown-freebsd-gnu`, mientras que para un Gentoo/FreeBSD, usando el núcleo y libc FreeBSD, pero con el marco Gentoo, la definición del anfitrión es `hw-gentoo-freebsd5.4`).

Usando las variables `$host` y `$build` en una macro `configure.ac` se pueden habilitar o deshabilitar características específicas basándose en el sistema operativo o la plataforma hardware en la que se está compilando o para la que se va a compilar.

No abusar de las dependencias "automágicas". Una de las características más útiles de autotools es la comprobación automática de la presencia de librerías, que se usan a menudo para añadir soporte para dependencias adicionales y similares. Sin embargo, abusar de esta característica hace que la construcción de un paquete sea un poco problemática. Mientras que se trata de algo muy útil para quienes la usan por primera vez, y para casi todos los proyectos que tengan dependencias complejas (como con programas multimedia tales como `xine` y `VLC`) usan una capa basada en los módulos que les permite evitar rupturas, las dependencias "automágicas" son un buen dolor de cabeza para los empaquetadores, especialmente para aquellos trabajando en distribuciones basadas en código fuente como Gentoo y capas de tipo ports. Cuando se construye algo con dependencias automáticas se habilitan funciones soportadas por las librerías que se encuentran en el sistema donde se ejecuta la macro `configure`. Lo cual significa que los binarios obtenidos pueden no funcionar en un sistema que comparta los mismos paquetes básicos pero que no disponga de una de las librerías adicionales, por ejemplo. Además, no pueden determinarse las dependencias exactas de un paquete, dado que algunas pueden ser opcionales y no construidas cuando las librerías no se encuentran presentes.

Para evitar esto, `autoconf` nos permite añadir las opciones `--enable/--disable` y `--with/--without` a las macros `configure`. Con dichas opciones se puede habilitar o deshabilitar convincentemente una opción específica (como el soporte para una librería adicional o de una característica específica), y dejar tomar por defecto los valores de las comprobaciones automáticas.

Lamentablemente, muchos desarrolladores no entienden bien el significado de los dos parámetros de las funciones usadas para añadir estas opciones (`AC_ARG_ENABLE` y `AC_ARG_WITH`). Representan el código a ejecutar cuando se les pasa un parámetro o cuando no se les pasa otro. Muchos desarrolladores piensan equivocadamente que ambos parámetros definen el código a ejecutar cuando la característica está habilitada y deshabilitada. Mientras que esto funciona normalmente cuando se le pasa un parámetro sólo para cambiar el comportamiento por defecto, muchas distribuciones basadas en código fuente le pasan parámetros también para confirmar el comportamiento por defecto, lo que conduce a errores (características solicitadas explícitamente no presentes). Ser capaz de deshabilitar características opcionales si no añaden dependencias (pensemos en el soporte de audio OSS en Linux) es siempre una buena cosa para los usuarios, que pueden evitar construir cierta cantidad de código si no piensan usarlo y previene que los desarrolladores añadan sucios trucos para cachear si habilitan o deshabilitan una característica según los usuarios la soliciten.

Mientras que las autotools fueron un gran problema tanto para desarrolladores como para mantenedores debido a que hay versiones diferentes incompatibles que no se llevan demasiado bien juntas (debido a que se instalan en los mismos sitios, con los mismos nombres) y que son usadas en diferentes combinaciones; el uso de autotools hace que los desarrolladores eviten usar todo tipo de juego sucio para compilar software. Si se miran los `ebuild` de portage en Gentoo, los pocos que no usan autotools son los más complejos, dado que deben comprobar variables en muchas configuraciones muy diferentes (podemos o no tener soporte NPTL; podemos estar bajo Linux, FreeBSD o Mac OS X; podemos estar usando GLIBC o cualquier `ptr lib`; y así), mientras que las autotools se ocupan de

esto por sí mismas. También es cierto que muchos parches aplicados por sus mantenedores son para corregir macros defectuosas en el código fuente de origen, pero esto es un pequeño problema comparado con el caos que genera usar sistemas de construcción especiales que no funcionan en absoluto con pequeñas modificaciones del entorno.

Las autotools pueden ser difíciles para los recién llegados, pero cuando se empiezan a usar a diario se encuentra que es mucho más fácil que tener que elaborar makefiles manualmente u otras extrañas herramientas de construcción como imake o qmake, o peor aún, macros de construcción tipo autotools que tratan de reconocer el sistema en el que se están construyendo. Autotools hace sencillo soportar nuevos sistemas operativos y nuevas plataformas de hardware, y evita a los mantenedores y portadores tener que aprender como construir un sistema para resolver adecuadamente la compilación. Escribiendo cuidadosamente una macro, los desarrolladores pueden soportar nuevas plataformas sin el más mínimo cambio.

El contenido de este documento está registrado bajo los términos de la licencia [Creative Commons - Reconocimiento / Compartir Igual](#)

Copyright 2001-2009 Gentoo Foundation, Inc. Questions, Comments? [Contact us.](#)