Scripting práctico rc.d en BSD

Resumen

Los principiantes pueden tener dificultades para relacionar los hechos de la documentación formal del framwork rc.d de BSD con las tareas prácticas de scripting de rc.d. En este artículo, consideramos algunos casos típicos de complejidad creciente, mostramos rc.d características adecuadas para cada caso y comentamos cómo funcionan. Dicho examen debe proporcionar puntos de referencia para un estudio más detallado del diseño y la aplicación eficiente de rc.d.

Tabla de contenidos

1. Introducción . . . 1
2. Delineando la tarea . . . 3
3. Un guión ficticio . . . 3
4. Un script ficticio configurable . . . 5
5. Arranque y parada de un demonio simple . . . 7
6. Arranque y parada de un demonio avanzado . . . 8
7. Conectando un script al framework rc.d . . . 12
8. Dar más flexibilidad a un script rc.d . . . 15
9. Otras lecturas . . . 18

1. Introducción

El BSD histórico tenía un script de arranque monolítico, /etc/rc. Era invocado por init(8) durante el arranque del sistema y realizaba todas las tareas en modo usuario que se requerían para operar en modo multi-usuario: comprobar y montar los sistemas de ficheros, configurar la red, arrancar demonios y demás. La lista precisa de tareas no era la misma en cada sistema; los administradores necesitaban personalizarla. Salvo en unas pocas excepciones, se tenía que modificar /etc/rc, y a los verdaderos hackers les gustaba.

El verdadero problema con el enfoque monolítico era que no proporcionaba control sobre los componentes individuales a partir de /etc/rc. Por ejemplo, /etc/rc no podía reiniciar un solo demonio. El administrador del sistema tenía que encontrar el proceso del demonio a mano, matarlo, esperar hasta que realmente terminara y luego examinar /etc/rc para los flags, y finalmente escribir la línea de comando completa para iniciar el demonio nuevamente. La tarea se volvía aún más difícil y propensa a errores si el servicio a reiniciar consistía en más de un demonio o exigía acciones adicionales. En pocas palabras, el script único no cumplía con el propósito de los scripts: facilitar la vida del administrador del sistema.

Posteriormente hubo un intento de separar algunas partes de /etc/rc para poder arrancar los subsistemas más importantes de forma separada. Un ejemplo importante era /etc/netstart para

levantar la red. Permitía acceder a la red desde el modo de usuario único, pero no se integraba bien con el proceso de arranque automático debido a que algunas partes de su código necesitaban intercalarse con acciones que en esencia no estaban relacionadas con la red. Por eso /etc/netstart se cambió a /etc/rc.network. El segundo ya no era un script ordinario; estaba compuesto de funciones sh(1) grandes y entrelazadas que se llamaban desde /etc/rc en diferentes fases del sistema de arranque. Sin embargo, a medida que las tareas de arranque se hicieron más diversas y sofisticadas, la aproximación "casi modular" se convirtió en un lastre casi más de lo que había sido el /etc/rc monolítico.

Sin un framework limpio y bien diseñado, los scripts de inicio tuvieron que hacer todo lo posible para satisfacer las necesidades de los sistemas operativos basados en BSD en rápido desarrollo. Por fin se hizo evidente que se necesitan más pasos en el camino hacia una sistema rc refinado y extensible. Así nació el rc.d de BSD. Sus padres reconocidos fueron Luke Mewburn y la comunidad NetBSD. Posteriormente se importó a FreeBSD. Su nombre se refiere a la ubicación de los scripts del sistema para servicios individuales, que se encuentra en /etc/rc.d. Pronto conoceremos más componentes del sistema rc.d y veremos cómo se invocan los scripts individuales.

Las ideas básicas detrás del rc.d de BSD son *modularidad fina* y *reutilización de código*. *Modularidad fina* significa que cada "servicio" básico tales como un demonio del sistema o una primitiva de arranque tienen su propio script sh(1) capaz de arrancar el servicio, pararlo, recargarlo y comprobar su estado. Se escoge una acción particular mediante un argumento en la línea de comando del script. El script /etc/rc todavía dirige el sistema de arranque, pero ahora simplemente invoca scripts más pequeños uno a uno con el argumento `start`. También es fácil realizar tareas de parado ejecutando el mismo conjunto de scripts con el argumento `stop`, que es lo que hace /etc/rc.shutdown. Date cuenta de cómo esto sigue de cerca la manera Unix de tener un conjunto pequeño de herramientas especializadas, cada una realizando su tarea lo mejor posible. *Reutilización de código* significa que las operaciones comunes están implementadas como funciones sh(1) y compiladas en /etc/rc.subr. Ahora un script típico puede tener sólo unas pocas líneas de código sh(1). Finalmente, una parte importante del framework rc.d es rcorder(8), que ayuda a /etc/rc a ejecutar los scripts pequeños de forma ordenada respecto a las dependencias entre ellos. También puede ayudar a /etc/rc.shutdown, porque el orden adecuado de apagado es el opuesto al de arranque.

El diseño del rc.d de BSD se describe en el artículo original de Luke Mewburn, y los componentes de rc.d están documentados con gran detalle en las respectivas páginas de manual. Sin embargo, puede que no parezca obvio para un novato de rc.d cómo unir las numerosas partes y piezas para crear un script con estilo para una tarea en particular. Por lo tanto, este artículo intentará un enfoque diferente para describir rc.d. Mostrará qué funciones deben usarse en varios casos típicos y por qué. Ten en cuenta que este no es un documento de instrucciones porque nuestro objetivo no es dar recetas listas para usar, sino mostrar algunas formas fáciles de introducirse en el reino de rc.d. Este artículo tampoco sustituye a las páginas del manual correspondientes. No dudes en consultarlas para obtener documentación más formal y completa mientras lees este artículo.

Hay prerequisitos para entender este artículo. Antes de nada, para dominar rc.d deberías estar familiarizado con el lenguaje de scripting de sh(1). Además deberías conocer cómo el sistema realiza las tareas de arranque y parada en modo usuario, que está descrito en rc(8).

Este artículo se centra en la rama FreeBSD de rc.d. Sin embargo, también puede ser útil para los desarrolladores de NetBSD, porque las dos ramas de rc.d de BSD no solo comparten el mismo

diseño, sino que también son similares en sus aspectos visibles para los creadores de scripts.

2. Delineando la tarea

Un poco de reflexión antes de arrancar `$EDITOR` no dolerá. Para escribir un script rc.d bien hecho para un servicio del sistema, deberíamos poder responder las siguientes preguntas primero:

- ¿El servicio es obligatorio u opcional?
- ¿El script servirá a un solo programa, por ejemplo, un demonio, o realizará acciones más complejas?
- ¿De qué otros servicios dependerá nuestro servicio y viceversa?

De los ejemplos que siguen veremos por qué es importante conocer las respuestas a estas preguntas.

3. Un guión ficticio

El siguiente script simplemente emite un mensaje cada vez que se inicia el sistema:

```
#!/bin/sh ①

. /etc/rc.subr ②

name="dummy" ③
start_cmd="${name}_start" ④
stop_cmd=":" ⑤

dummy_start() ⑥
{
    echo "Nothing started."
}

load_rc_config $name ⑦
run_rc_command "$1" ⑧
```

Las cosas a tener en cuenta son:

① Un script interpretado debería empezar con la línea mágica "shebang". Esa línea especifica el programa intérprete para el script. Gracias a la línea shebang, el script se puede invocar exactamente igual que un programa binario si se ha establecido el bit de ejecución. (Consulta chmod(1).) Por ejemplo, un administrador puede ejecutar nuestro script de forma manual, desde la línea de comando:

```
# /etc/rc.d/dummy start
```



Para que los scripts puedan ser gestionados por el framework rc.d tienen que estar escritos en lenguaje sh(1). Si tienes un servicio o port que usa una utilidad de control binaria o una rutina de arranque escrita en otro lenguaje, instala ese elemento en /usr/sbin (para el sistema) o /usr/local/sbin (para ports) e invócalo desde un script sh(1) en el directorio rc.d apropiado.



Si quieres conocer los detalles acerca de por qué los scripts de rc.d se tienen que escribir en lenguaje sh(1), consulta cómo /etc/rc los invoca mediante `run_rc_script`, luego estudia la implementación de `run_rc_script` en /etc/rc.subr.

 En /etc/rc.subr, se definen un número de funciones sh(1) para que las use el script rc.d. Las funciones están documentadas en rc.subr(8). Aunque es teóricamente posible escribir un script rc.d sin llegar a usar rc.subr(8), sus funciones han demostrado ser extremadamente útiles y hacen el trabajo un orden de magnitud más fácil. Así que no es una sorpresa que todo el mundo recurra a rc.subr(8) en los scripts de rc.d. Nosotros no vamos a ser una excepción.

Un script rc.d debe incluir /etc/rc.subr (utilizando ".") *antes* de llamar a funciones de rc.subr(8) de forma que sh(1) tenga una oportunidad para saber acerca de las funciones. El estilo preferido es hacer "source" de /etc/rc.subr antes de nada.



Algunas funciones útiles relacionadas con redes son proporcionadas por otro archivo de inclusión, /etc/network.subr.

 La variable obligatoria `name` especifica el nombre de nuestro script. Es un requisito de rc.subr(8). Es decir, cada script rc.d *debe establecer* `name` antes de llamar a las funciones de rc.subr(8).

Ahora es el momento adecuado para elegir un nombre único para nuestro script de una vez por todas. Lo usaremos en varios lugares mientras desarrollamos el script. Para empezar, démosle también el mismo nombre al archivo del script.



El estilo actual de los scripts rc.d es englobar los valores asignados a variables entre comillas dobles. Ten en cuenta que esto es sólo una cuestión de estilo y que podría no ser aplicable siempre. Puedes omitir las comillas de forma segura alrededor de palabras sencillas que no contengan metacaracteres de sh(1), mientras que en ciertos casos necesitarás comillas simples para evitar cualquier interpretación del valor por parte de sh(1). Un programador debería ser capaz de distinguir la sintaxis del lenguaje de las convenciones de estilo y aplicar ambas de forma apropiada.

 La idea principal detrás de rc.subr(8) es que un script rc.d proporciona manejadores, o métodos, para que los invoque rc.subr(8). En particular, `start`, `stop`, y otros argumentos pasados a un script rc.d se manejan de esta forma. Un método es una expresión sh(1) que se almacena en una variable llamada `argument_cmd`, donde *argument* corresponde a lo que se puede especificar en la línea de comando del script. Luego veremos cómo rc.subr(8) proporciona métodos por defecto para los argumentos estándar.



Para hacer el código en rc.d más uniforme, es común usar `${name}` donde sea

apropiado. Por tanto, es posible simplemente copiar varias líneas de un script a otro.

 Deberíamos tener en cuenta que rc.subr(8) proporciona métodos por defecto para los argumentos estándar. Consecuentemente, debemos sobrescribir un método con una expresión no-op de sh(1) si queremos que no haga nada.

 El cuerpo de un método sofisticado se puede implementar como una función. Es una buena idea que el nombre de la función tenga un significado.



Se recomienda encarecidamente añadir el prefijo `${name}` a los nombres de todas las funciones definidas en nuestro script de forma que nunca colisionen con funciones de rc.subr(8) o cualquier otro fichero que se incluya de forma habitual.

 Esta llamada a rc.subr(8) carga las variables de rc.conf(5). Nuestro script no las utiliza todavía, pero aún así se recomienda cargar rc.conf(5) porque puede haber variables de rc.conf(5) controlando al propio rc.subr(8).

 Normalmente este es el último comando en un script rc.d. Invoca la maquinaria de rc.subr(8) para realizar la acción solicitada utilizando las variables y métodos que ha proporcionado nuestro script.

4. Un script ficticio configurable

Ahora añadamos algunos controles a nuestro script de prueba. Como sabes, los scripts rc.d están controlados por rc.conf(5). Afortunadamente, rc.subr(8) nos oculta todas las complicaciones. El siguiente script usa rc.conf(5) mediante rc.subr(8) para ver en primer lugar si está habilitado, y para obtener un mensaje que mostrar en el momento del arranque. Estas dos tareas son de hecho independientes. En un lado, un script rc.d puede simplemente soportar la activación y desactivación de su servicio. Por otro lado, un script rc.d obligatorio puede tener las variables de configuración. Sin embargo, nosotros haremos ambas cosas en el mismo script:

```
#!/bin/sh

. /etc/rc.subr

name=dummy
rcvar=dummy_enable ①

start_cmd="${name}_start"
stop_cmd=":"

load_rc_config $name ②
: ${dummy_enable:=no} ③
: ${dummy_msg="Nothing started."} ④

dummy_start()
{
    echo "$dummy_msg" ⑤
```

```
}

run_rc_command "$1"
```

¿Qué cambió en este ejemplo?

 La variable `rcvar` especifica el nombre de la variable ON/OFF.

 Ahora `load_rc_config` es invocado pronto en el script, antes de que se acceda a alguna variable de rc.conf(5).



Cuando examines scripts rc.d, ten en cuenta que sh(1) retrasa la evaluación de expresiones en una función hasta que ésta es invocada. Por lo tanto, no es un error invocar `load_rc_config` tan tarde como justo antes de `run_rc_command` y aún así acceder a variables de rc.conf(5) desde los métodos exportados a `run_rc_command`. Esto es porque los métodos se llaman desde `run_rc_command`, que es invocado *después* de `load_rc_config`.

 Se emitirá un aviso desde `run_rc_command` si `rcvar` está establecida, pero la variable en sí no lo está. Si tu script rc.d es para el sistema base, deberías añadir un valor por defecto para la variable en /etc/defaults/rc.conf y documentarlo en rc.conf(5). De lo contrario tu script debería proporcionar un valor por defecto para la variable. La aproximación canónica para el último caso se muestra en el ejemplo.



Puedes hacer que rc.subr(8) actúe como si la variable estuviera a `ON`, independientemente de su estado actual, poniendo como prefijo del argumento del script `one` o `force`, como en `onestart` o `forcestop`. Ten en cuenta sin embargo que `force` tiene otros efectos peligrosos que mencionaremos abajo, mientras que `one` simplemente tiene preferencia sobre la variable ON/OFF. Por ejemplo, asume que `dummy_enable` es `OFF`. El siguiente comando ejecutará el método `start` a pesar de esa configuración:

```
# /etc/rc.d/dummy onestart
```

 Ahora el mensaje que se mostrará en el arranque ya no está inmutable en el script. Se especifica en una variable de rc.conf(5) llamada `dummy_msg`. Este es un ejemplo trivial de cómo un script de rc.d puede ser controlado por variables de rc.conf(5).



Los nombres de todas las variables rc.conf(5) usadas en exclusiva por nuestro script *deben tener* el mismo prefijo: `${name}_`. Por ejemplo: `dummy_mode`, `dummy_state_file`, y así sucesivamente.



Aunque es posible utilizar internamente un nombre más corto, por ejemplo simplemente `msg`, añadir el prefijo único `${name}_` a todos los nombres globales introducidos por nuestro script nos evitará posibles colisiones en el espacio de nombres de rc.subr(8).

> Como norma, los scripts rc.d del sistema base no necesitan proporcionar valores por defecto para sus variables en rc.conf(5) porque estos deberían establecerse en cambio en /etc/defaults/rc.conf. Por otro lado, los scripts rc.d para ports deberían proporcionar los valores por defecto como se muestra en el ejemplo.

 Aquí utilizamos `dummy_msg` en realidad para controlar nuestro script, es decir, para emitir un mensaje variable. Utilizar una función de shell para esto es demasiado ya que sólo ejecuta un comando. Una alternativa igualmente válida es:

```
start_cmd="echo \"$dummy_msg\""
```

5. Arranque y parada de un demonio simple

Dijimos antes que rc.subr(8) podía proporcionar métodos por defecto. Obviamente, estos no pueden ser muy generales. Se adaptan bien al caso común de arrancar y parar un programa tipo demonio que sea sencillo. Asumamos ahora que necesitamos escribir un script rc.d para dicho demonio llamado `mumbled`. Aquí está:

```
#!/bin/sh

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}" ①

load_rc_config $name
run_rc_command "$1"
```

Agradablemente simple, ¿no? Examinemos nuestro pequeño script. Lo único nuevo a tener en cuenta es lo siguiente:

 La variable `command` tiene sentido para rc.subr(8). Si está establecida, rc.subr(8) actuará en consecuencia con el escenario de servir un demonio convencional. En particular, se proporcionarán los métodos por defecto para los argumentos: `start`, `stop`, `restart`, `poll`, y `status`.

El demonio se arrancará ejecutando `$command` con los flags especificados por `$mumbled_flags`. Por lo tanto todos los datos de entrada para el método `start` por defecto están disponibles en las variables establecidas por nuestro script. A diferencia de `start`, otros métodos podrían requerir información adicional acerca del proceso arrancado. Por ejemplo, `stop` debe saber el PID del proceso para poder terminarlo. En el caso actual, rc.subr(8) escaneará la lista de todos los procesos, buscando un proceso cuyo nombre sea `procname`. Esto último es otra variable que tiene significado para rc.subr(8) y su valor por defecto es el valor de `command`. En otras palabras, cuando establecemos `command`, `procname` se establece al mismo valor. Esto posibilita que nuestro script pueda matar el demonio y así como comprobar en primer lugar si se está ejecutando.



Algunos programas son de hecho scripts ejecutables. El sistema ejecuta dichos scripts iniciando su intérprete y pasándole el nombre del script como un argumento en línea de comandos. Esto se refleja en la lista de procesos, que puede confundir a rc.subr(8). Deberías establecer además `command_interpreter` para que rc.subr(8) sepa cuál es el verdadero nombre del proceso si `$command` es un script.

Para cada script rc.d, hay una variable opcional en rc.conf(5) que tiene preferencia sobre `command`. Su nombre se construye de la siguiente manera: `${name}_program`, donde `name` es la variable obligatoria que discutimos anteriormente. Ejemplo, en este caso será `mumbled_program`. Es rc.subr(8) quien hace que `${name}_program` tenga más prioridad que `command`.

Por supuesto, sh(1) te permitirá establecer `${name}_program` desde rc.conf(5) o incluso desde el propio script si `command` no está establecido. En ese caso, las propiedades especiales de `${name}_program` se pierden y se convierte en una variable ordinaria que tu script puede usar para sus propios fines. Sin embargo, el uso aislado de `${name}_program` está desaconsejado porque usarlo junto con `command` se ha convertido en algo idiomático en los scripts de rc.d.

Para una información más detallada acerca de los métodos por defecto, consulta rc.subr(8).

6. Arranque y parada de un demonio avanzado

Agreguemos un poco de carne a los huesos del guión anterior y hagámoslo más complejo y más rico en características. Los métodos predeterminados pueden hacer un buen trabajo por nosotros, pero es posible que necesitemos ajustar algunos de sus aspectos. Ahora aprenderemos cómo ajustar los métodos predeterminados a nuestras necesidades.

```
#!/bin/sh

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"
command_args="mock arguments > /dev/null 2>&1" ①

pidfile="/var/run/${name}.pid" ②

required_files="/etc/${name}.conf /usr/share/misc/${name}.rules" ③

sig_reload="USR1" ④

start_precmd="${name}_prestart" ⑤
stop_postcmd="echo Bye-bye" ⑥
```

```
extra_commands="reload plugh xyzzy" ⑦

plugh_cmd="mumbled_plugh" ⑧
xyzzy_cmd="echo 'Nothing happens.'"

mumbled_prestart()
{
    if checkyesno mumbled_smart; then ⑨
        rc_flags="-o smart ${rc_flags}" ⑩
    fi
    case "$mumbled_mode" in
    foo)
        rc_flags="-frotz ${rc_flags}"
        ;;
    bar)
        rc_flags="-baz ${rc_flags}"
        ;;
    *)
        warn "Invalid value for mumbled_mode" ⑪
        return 1 ⑫
        ;;
    esac
    run_rc_command xyzzy ⑬
    return 0
}

mumbled_plugh() ⑭
{
    echo 'A hollow voice says "plugh".'
}

load_rc_config $name
run_rc_command "$1"
```

 Se pueden pasar argumentos adicionales a `$command` mediante `command_args`. Se añadirán a la línea de comando después de `$mumbled_flags`. Como la línea de comando final se pasa a `eval` para su ejecución, se pueden especificar redirecciones de entrada y salida en `command_args`.



Nunca incluyas opciones con guiones, como `-X` o `--foo`, en `command_args`. El contenido de `command_args` aparecerá al final de la línea de comando, por lo tanto seguramente aparezcan después que los argumentos presentes en `${name}_flags`; pero la mayoría de los comandos no reconocen opciones con guiones que aparezcan después de los argumentos ordinarios. Una forma mejor de pasar opciones adicionales a `$command` es añadirlas al principio de `${name}_flags`. Otra forma es modificar `rc_flags` como se muestra más adelante.

 Un demonio con buenos modales debería crear un *pidfile* de forma que su proceso se pueda encontrar de forma más fácil y segura. La variable `pidfile`, si está establecida, le dice a rc.subr(8)

dónde puede encontrar el pidfile para que lo usen sus métodos por defecto.



De hecho, rc.subr(8) también usará el pidfile para ver si el demonio está corriendo antes de arrancarlo. Esta comprobación se puede omitir utilizando el argumento `faststart`.

 Si el demonio no puede ejecutarse a menos que exista cierto fichero, tan solo inclúyelos en la lista `required_files`, y rc.subr(8) comprobará que esos ficheros existen antes de arrancar el demonio. También existen `required_dirs` y `required_vars` para directorios y variables de entorno respectivamente. Todas ellas están descritas con detalle en rc.subr(8).



Se puede forzar el método por defecto de rc.subr(8) para que se salte las comprobaciones de prerequisitos utilizando `forcestart` como argumento del script.

 Podemos personalizar qué señales enviar al demonio en caso de que difieran de las que son bien conocidas. En particular, `sig_reload` especifica la señal que hace que el demonio recargue su configuración; es SIGHUP por defecto. Otra señal se envía para parar el proceso del demonio; por defecto es SIGTERM, pero se puede cambiar estableciendo `sig_stop` de forma apropiada.



Los nombres de las señales se tienen que especificar a rc.subr(8) sin el prefijo `SIG`, como se muestra en el ejemplo. La versión de kill(1) de FreeBSD puede reconocer el prefijo `SIG`, pero versiones de otros tipos de OS podrían no hacerlo.

 Realizar tareas adicionales antes o después de los métodos por defecto es fácil. Para cada comando-argumento soportado por nuestro script, podemos definir `argument_precmd` y `argument_postcmd`. Estos comandos sh(1) se invocan antes y después del método respectivo, como es evidente por sus nombres.



Sobrescribir un método por defecto con un `argument_cmd` personalizado no nos impide usar `argument_precmd` o `argument_postcmd` si lo necesitamos. En particular, el primero es bueno para comprobar condiciones sofisticadas, personalizadas que se deberían cumplir antes de ejecutar el comando. Usar `argument_precmd` junto con `argument_cmd` nos permite realizar una separación lógica de las comprobaciones y la acción.

No olvides que puedes poner cualquier expresión sh(1) válida dentro de los métodos, pre-, y post-commands que defines. Invocar simplemente una función que realiza el trabajo es un buen estilo en la mayoría de los casos, pero nunca dejes que el estilo limite tu entendimiento de lo que sucede por debajo.

 Si quieres implementar argumentos personalizados, que también se pueden entender como *comandos* para nuestro script, necesitamos listarlos en `extra_commands` y proporcionar métodos para manejarlos.



El comando `reload` es especial. Por un lados tiene un método preestablecido en rc.subr(8). Por otro, `reload` no se ofrece por defecto. El motivo es que no todos los demonios usan el mismo mecanismo de recarga y algunos no tienen nada que

> recargar. De forma que tenemos que pedir explícitamente que se proporcione la funcionalidad. Podemos hacerlo mediante `extra_commands`.
>
> ¿Qué obtenemos del método por defecto para `reload`? A menudo los demonios recargan su configuración al recibir una señal - típicamente, SIGHUP. Por lo tanto rc.subr(8) intenta recargar el demonio enviándole una señal. La están señal está preestablecida a SIGHUP pero se puede cambiar mediante `sig_reload` si es necesario.

□□ Nuestro script soporta dos comandos no estándar, `plugh` y `xyzzy`. Los hemos visto listados en `extra_commands`, y ahora es momento de proporcionarles métodos. El método para `xyzzy` está entre líneas mientras que el de `plugh` se implementa en la función `mumbled_plugh`.

Los comandos no estándar no se invocan durante el arranque o el apagado. Normalmente están ahí por conveniencia para los administradores. También se pueden usar desde otros subsistemas, por ejemplo, devd(8) si se especifica en devd.conf(5).

Se puede encontrar la lista completa de comandos disponibles en la línea de uso imprimida por rc.subr(8) cuando se invoca el script sin argumentos. Por ejemplo, esta es la línea de uso para el script que estamos estudiando:

```
# /etc/rc.d/mumbled
Usage: /etc/rc.d/mumbled [fast|force|one]
(start|stop|restart|rcvar|reload|plugh|xyzzy|status|poll)
```

□ Un script puede invocar sus comandos estándar y no estándar si lo necesita. Esto parece similar a llamar a funciones, pero sabemos que los comandos y funciones del shell no son siempre la misma cosa. Por ejemplo, `xyzzy` aquí no se implementa como una función. Además, puede haber pre-comandos y post-comandos, que se deberían invocar en orden. De modo que la forma apropiada para que un script ejecute sus propios comandos es mediante rc.subr(8), como se muestra en el ejemplo.

□ rc.subr(8) proporciona una función útil llamada `checkyesno`. Admite una variable como argumento y devuelve cero si y sólo si la variable está establecida a `YES`, o `TRUE`, o `ON`, o `1`, sin considerar mayúsculas y minúsculas; devuelve un valor distinto de cero en caso contrario. En el último caso, la función comprueba que la variable esté establecida a `NO`, `FALSE`, `OFF`, o `0`, sin distinguir entre mayúsculas y minúsculas; imprime un aviso si la variable contiene cualquier otra cosa, es decir, basura.

Ten en cuenta que para sh(1) un código de salida igual a cero significa verdadero y distinto de cero significa falso.



> La función `checkyesno` admite un *nombre de variable*. No pases el *valor* expandido de una variable; no funcionará como se espera.
>
> Lo siguiente es un uso correcto de `checkyesno`:
>
> ```
> if checkyesno mumbled_enable; then
> ```

```
        foo
fi
```

Por el contrario, llamar a `checkyesno` como se muestra abajo no funcionará - al menos no como se espera:

```
if checkyesno "${mumbled_enable}"; then
        foo
fi
```

 Podemos alterar los flags que se pasan a `$command` modificando `rc_flags` en `$start_precmd`.

 En algunos casos podríamos necesitar emitir un mensaje importante que debería ir también a `syslog`. Se puede hacer fácilmente con las siguientes funciones de rc.subr(8): `debug`, `info`, `warn`, y `err`. La última función sale del script con el código especificado.

 Los códigos de salida de los métodos y sus pre-comandos no se ignoran simplemente por defecto. Si `argument_precmd` devuelve un código de salida distinto de cero, el método principal no se ejecutará. Del mismo modo, `argument_postcmd` no será invocado a menos que el método principal devuelva un código de salida igual a cero.



Sin embargo, desde la línea de comando se puede indicar a rc.subr(8) que ignore esos códigos de salida e invoque todos los comandos añadiendo el prefijo `force` a los argumentos, como en `forcestart`.

7. Conectando un script al framework rc.d

Después de que se ha escrito un script, es necesario integrarlo en rc.d. El paso crucial es instalar el script en /etc/rc.d (para el sistema base) o /usr/local/etc/rc.d (para los ports). Tanto bsd.prog.mk como bsd.port.mk proporcionan los hooks necesarios para ello, y normalmente no tienes que preocuparte acerca de los permisos y el propietario. Los scripts de sistema deberían instalarse desde src/libexec/rc/rc.d mediante el Makefile que se encuentra allí. Los scripts de ports se pueden instalar con `USE_RC_SUBR` como se describe en el Porter's Handbook.

Sin embargo, debemos considerar de antemano el lugar de nuestro script en la secuencia de inicio del sistema. Es probable que el servicio manejado por nuestro script dependa de otros servicios. Por ejemplo, un demonio de red no puede funcionar sin las interfaces de red y enrutamiento en funcionamiento. Incluso si un servicio parece no exigir nada, difícilmente puede iniciarse antes de que se hayan verificado y montado los sistemas de archivos básicos.

Ya hemos mencionado rcorder(8). Ahora es momento de mirarlo detenidamente. En pocas palabras, rcorder(8) toma un conjunto de ficheros, examina el contenido e imprime una lista ordenada de dependencias de ficheros del conjunto a `stdout`. El objetivo es mantener la información de dependencia *dentro* de los ficheros de forma que cada uno de ellos sólo habla por sí mismo. Un fichero puede especificar la siguiente información:

- los nombres de las "condiciones" (lo que para nosotros significa servicios) que *proporciona*;

- los nombres de las "condiciones" que *requiere*;
- los nombres de las "condiciones" para las cuales este fichero debería ejecutarse *con anterioridad*;
- *palabras clave* adicionales que se pueden usar para seleccionar un subconjunto de todo el conjunto de ficheros (se puede indicar a rcorder(8) mediante opciones que incluya u omita ficheros que contengan determinadas palabras clave)

No es sorprendente que rcorder(8) pueda manejar sólo ficheros de texto con una sintaxis similar a la de sh(1). Es decir, las líneas especiales entendidas por rcorder(8) se parecen a comentarios de sh(1). La sintaxis de dichas líneas especiales es bastante rígida para simplificar su procesamiento. Lee rcorder(8) para más detalles.

Además de utilizar líneas especiales de rcorder(8), un script puede incidir en sus dependencias de otro servicio forzando su arranque. Esto podría ser necesario cuando el otro servicio es opcional y no arrancará por sí mismo porque el administrador lo ha deshabilitado por error en rc.conf(5).

Con este conocimiento general en mente, consideremos el script demonio simple mejorado con información de dependencia:

```
#!/bin/sh

# PROVIDE: mumbled oldmumble ①
# REQUIRE: DAEMON cleanvar frotz ②
# BEFORE:  LOGIN ③
# KEYWORD: nojail shutdown ④

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"
start_precmd="${name}_prestart"

mumbled_prestart()
{
    if ! checkyesno frotz_enable && \
        ! /etc/rc.d/frotz forcestatus 1>/dev/null 2>&1; then
        force_depend frotz || return 1 ⑤
    fi
    return 0
}

load_rc_config $name
run_rc_command "$1"
```

Como antes, sigue un análisis detallado:

 Esa línea declara los nombres de las "condiciones" que proporciona nuestro script. Ahora otros

scripts pueden registrar una dependencia de nuestro script usando dichos nombres.



Por lo general, un script especifica una sola condición proporcionada. Sin embargo, nada nos impide enumerar varias condiciones allí, por ejemplo, por razones de compatibilidad.

En cualquier caso, el nombre de la condición `PROVIDE:` principal, o única, debería ser el mismo que `${name}`.

Nuestro script indica de qué "condiciones" depende que son proporcionadas por otros scripts. Según esas líneas nuestro script pide a rcorder(8) situarlo después del script (o los scripts) que proporcionan DAEMON y cleanvar, pero antes de los que proporcionan LOGIN.



No se debería abusar de la línea `BEFORE:` para evitar una lista de dependencias incompleta en el otro script. El caso apropiado para usar `BEFORE:` es cuando el otro script no se preocupa por el nuestro, pero nuestro script puede hacer mejor su tarea si se ejecuta antes que el otro. Un ejemplo típico de la vida real son las interfaces de red frente al firewall: si bien las interfaces no dependen del firewall para hacer su trabajo, la seguridad del sistema se beneficiará de que el firewall esté listo antes de que haya tráfico de red.

Además de las condiciones que se corresponden con un solo servicio, hay meta-condiciones y sus scripts tipo "placeholder" para asegurar que ciertos grupos de operaciones se ejecutan antes que otras. Se distinguen por sus nombres en MAYÚSCULAS. Su listado y propósito se puede encontrar en rc(8).

Ten en cuenta que poner un nombre de servicio en la línea `REQUIRE:` no garantiza que el servicio se esté ejecutando cuando nuestro script arranque. El servicio podría fallar al arrancar o estar deshabilitado en rc.conf(5). Obviamente, rcorder(8) no puede controlar esos detalles y rc(8) tampoco lo hará. Por lo tanto, la aplicación arrancada por nuestro script debería ser capaz de lidiar con situaciones en las que algún servicio requerido no esté disponible. En ciertos casos, podemos evitarlo como se discute abajo

Como recordamos del texto anterior, las palabras clave de rcorder(8) se pueden utilizar para seleccionar o excluir algunos scripts. Cualquier consumidor de rcorder(8) puede especificar mediante las opciones `-k` y `-s` qué palabras clave están en la lista "a mantener" y la lista "a omitir", respectivamente. De todos los ficheros que serán ordenados como dependencias, rcorder(8) escogerá sólo aquellos que tengan las palabras clave de la lista "a mantener" (a menos que esté vacía) y que no tenga la palabra clave en la lista "a omitir".

En FreeBSD, /etc/rc y /etc/rc.shutdown usan rcorder(8). Estos dos scripts definen la lista estándar de palabras clave del rc.d de FreeBSD y su significado es el que sigue:

nojail

El servicio no es para un entorno jail(8). Los procedimientos automáticos de arranque y parada ignorarán el script si está dentro de un jail.

nostart

El servicio se tiene que arrancar manualmente o no se arrancará. El procedimiento de arranque automático ignorará el script. Esto se puede usar, junto con la palabra clave shutdown, para escribir scripts que hace algo sólo cuando se para el sistema.

shutdown

Esta palabra clave se especifica *explícitamente* si se necesita parar el servicio antes de la parada del sistema.



Cuando el sistema se va a apagar, se ejecuta /etc/rc.shutdown. Asume que la mayoría de los scripts rc.d no tienen nada que hacer la mayoría del tiempo. Por lo tanto /etc/rc.shutdown invoca los scripts de rc.d de forma selectiva con la palabra clave shutdown, ignorando de forma efectiva el resto de los scripts. Para hacer un apagado incluso más rápido /etc/rc.shutdown pasa el comando faststop a los scripts que ejecuta de forma que se salten las comprobaciones preliminares, por ejemplo la comprobación del pidfile. Como los servicios dependientes se deberían parar antes que sus prerequisitos, /etc/rc.shutdown ejecuta los scripts en orden inverso de dependencia. Si escribes un script rc.d, deberías considerar si es relevante en el momento del apagado. Por ejemplo, si tu script hace su trabajo como respuesta sólo al comando start, entonces no necesitas incluir esta palabra clave. Sin embargo, si tu script gestiona un servicio, probablemente es una buena idea pararlo antes de que el sistema proceda a la fase final de su secuencia de apagado descrito en halt(8). En particular, un servicio se debería parar de forma explícita si necesita un tiempo considerable o acciones especiales para pararse de forma limpia. Un ejemplo típico de dicho servicio es un motor de bases de datos.

 Para empezar, `force_depend` debería usarse con mucho cuidado. Normalmente es mejor revisar la jerarquía de variables de configuración para tu script de rc.d si son interdependientes.

Si aún así no puedes evitar usar `force_depend`, el ejemplo ofrece una forma habitual de cómo invocarlo de forma condicional. En el ejemplo, nuestro demonio `mumbled` requiere que otro, `frotz`, se arranque con antelación. Sin embargo, `frotz` también es opcional; y rcorder(8) no sabe nada acerca de ese detalle. Afortunadamente, nuestro script tiene acceso a todas las variables de rc.conf(5). Si `frotz_enable` es verdadero, esperamos lo mejor y confiamos en que rc.d haya arrancado `frotz`. De lo contrario comprobamos el estado de `frotz`. Finalmente, forzamos nuestra dependencia de `frotz` si se constata que no se está ejecutando. `force_depend` emitirá un mensaje de aviso porque se debería invocar sólo si se ha detectado una mala configuración.

8. Dar más flexibilidad a un script rc.d

Cuando se invoca durante el arranque o la parada, un script rc.d se supone que actúa en todo el subsistema del que es responsable. Por ejemplo, /etc/rc.d/netif debería arrancar o parar todas las interfaces de red descritas en rc.conf(5). Se puede indicar cualquiera de los dos comandos utilizando un argumento como `start` o `stop`. Entre el arranque y la parada, los scripts rc.d ayudan al administrador a controlar el sistema en ejecución, y es cuando surge la necesidad de mayor flexibilidad y precisión. Por ejemplo, el administrador podría querer añadir la configuración de una nueva interfaz de red a rc.conf(5) y luego arrancarla sin interferir con la operación de las

interfaces existentes. La siguiente vez el administrador podría necesitar parar una interfaz de red concreta. Siguiendo el espíritu de la línea de comandos, el script respectivo de rc.d necesita un argumento extra, el nombre de la interfaz.

Afortunadamente, rc.subr(8) permite pasar un número arbitrario de argumentos a los métodos del script (dentro de los límites del sistema). Debido a esto, los cambios en el script pueden ser mínimos.

¿Cómo puede rc.subr(8) obtener acceso a los argumentos extra de la línea de comando? ¿Debería simplemente obtenerlos? De ningún modo. Primero, una función de sh(1) no tiene acceso a los parámetros posicionales del llamante, pero rc.subr(8) es simplemente una pila de dichas funciones. Segundo, las buenas maneras de rc.d dictan que es el script principal el encargado de decidir qué argumentos se pasan a sus métodos.

De modo que la aproximación de rc.subr(8) es como sigue: `run_rc_command` pasa todos sus argumentos salvo el primero de forma literal al método respectivo. El primer argumento, omitido, es el nombre del método en sí: `start`, `stop`, etc. `run_rc_commad` lo desplazará de forma que lo que es `$2` en la línea de comandos original se presentará como `$1` al método y así sucesivamente.

Para ilustrar esta oportunidad, modifiquemos el script ficticio primitivo para que sus mensajes dependan de los argumentos adicionales proporcionados. Aquí vamos:

```
#!/bin/sh

. /etc/rc.subr

name="dummy"
start_cmd="${name}_start"
stop_cmd=":"
kiss_cmd="${name}_kiss"
extra_commands="kiss"

dummy_start()
{
        if [ $# -gt 0 ]; then ①
                echo "Greeting message: $*"
        else
                echo "Nothing started."
        fi
}

dummy_kiss()
{
        echo -n "A ghost gives you a kiss"
        if [ $# -gt 0 ]; then ②
                echo -n " and whispers: $*"
        fi
        case "$*" in
        *[.!?])
                echo
```

```
                ;;
        *)
                echo .
                ;;
        esac
}

load_rc_config $name
run_rc_command "$@" ③
```

¿Qué cambios esenciales podemos notar en el script?

① Todos los argumentos que escribas después de `start` terminan como parámetros posicionales en el método respectivo. Podemos utilizarlos de cualquier forma de acuerdo con nuestra tarea, habilidades e imaginación. En el ejemplo actual, simplemente los pasamos todos a echo(1) como una cadena en la siguiente línea - fíjate en `$*` dentro de las comillas dobles. Aquí se ve cómo se puede invocar ahora el script:

```
# /etc/rc.d/dummy start
Nothing started.

# /etc/rc.d/dummy start Hello world!
Greeting message: Hello world!
```

② Lo mismo aplica a cualquier método proporcionado por nuestro script, no sólo a los estándar. Hemos añadido un método personalizado llamado `kiss` y puede aprovecharse de los argumentos extra del mismo modo que lo hace `start`. Ejemplo:

```
# /etc/rc.d/dummy kiss
A ghost gives you a kiss.

# /etc/rc.d/dummy kiss Once I was Etaoin Shrdlu...
A ghost gives you a kiss and whispers: Once I was Etaoin Shrdlu...
```

③ Si queremos pasar todos los argumentos extra a cualquier método, podemos simplemente sustituir `"$@"` por `"$1"` en la última línea de nuestro script, cuando invocamos `run_rc_command`.



Un programador de sh(1) tiene que entender la sutil diferencia entre `$*` y `$@` como formas de designar todos los parámetros posicionales. Para una discusión en profundidad, consulta un buen manual de script de sh(1). *No* uses las expresiones hasta que las entiendas completamente porque un uso inadecuado puede resultar en scripts defectuosos e inseguros.



Actualmente `run_rc_command` podría tener un bug que impide que mantenga los límites originales entre los argumentos. Es decir, los argumentos con espacios en blanco podrían no procesarse correctamente. El bug nace de un uso inadecuado de

```
$*.
```

9. Otras lecturas

El artículo original de Luke Mewburn ofrece una visión general de rc.d y una explicación detallada de las decisiones de diseño. Proporciona información sobre todo el framework de rc.d y su lugar de un sistema operativo BSD moderno.

Las páginas del manual de rc(8), rc.subr(8), y rcorder(8) documentan los componentes de rc.d en gran detalle. No puedes usar toda la potencia de rc.d sin estudiar las páginas del manual y hacer referencia a ellas mientras escribes tus propios scripts.

La mayor fuente de ejemplos de la vida real en funcionamiento es /etc/rc.d en un sistema real. Su contenido es fácil y agradable de leer porque la mayoría de los asuntos espinosos están escondidos en rc.subr(8). De cualquier forma ten en cuenta que los scripts de /etc/rc.d no han sido escritos por los ángeles, así que podrían tener bugs y decisiones de diseño subóptimas. ¡Ahora puedes mejorarlos!