Depuración de scripts Lua en Redis

Tiempo estimado: 25min

La depuración es algo que todo usuario debe conocer para desenvolverse con soltura con Lua en Redis. Y así resolver problemas de programación fácil y rápidamente. No hay una buena base sin conocer un depurador con el que encontrar fallos en el software. En esta lección, centramos nuestra atención en el depurador de Lua que viene con Redis.

La lección comienza con una introducción a la depuración, proceso mediante el cual se detecta y corrige un fallo de software. También se introduce los componentes o elementos que todo buen depurador debe tener: los puntos de interrupción, la ejecución paso a paso, los observadores y la pila de llamadas. Concluimos, presentando el depurador de Lua de Redis y cómo trabajar con él.

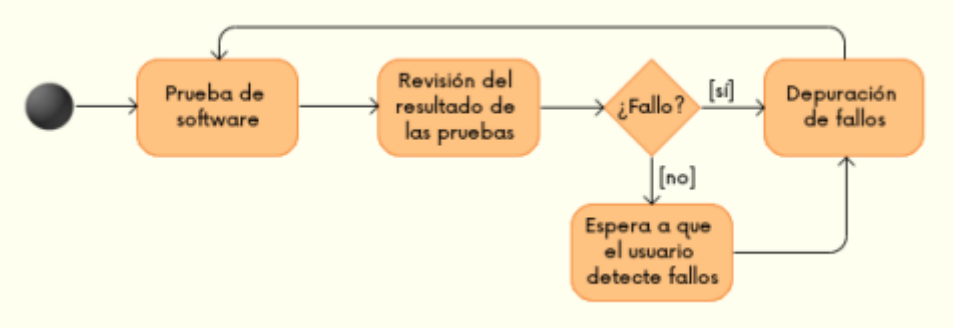
Al finalizar la lección, el estudiante sabrá:

- Qué es la depuración y para qué sirve.
- Qué son los puntos de interrupción y cómo definirlos en el depurador.
- Qué es la ejecución paso a paso y cómo trabajar con ella en el depurador.
- Qué son los observadores.
- Qué es la pila de llamadas y cómo visualizarla con el depurador.

Introducción

Formalmente, la depuración (*debugging*) es el proceso mediante el cual se corrige uno o más defectos o fallos en el software. Por un lado, se busca la causa del problema, es decir, por qué se está dando el error y, una vez encontrado, se resuelve o corrige. Una vez finalizada la depuración, se vuelve a someter el software a la batería de pruebas para validar que realmente se ha resuelto el fallo sin añadir ninguno nuevo, es decir, que la corrección del defecto no tiene efectos colaterales en otras partes del software que no los presentaba antes.

A continuación, se muestra un diagrama de estado que muestra el ciclo de vida del software, en cuanto a fallos se refiere:



Tal como puede observar, nunca se da por finalizada la prueba, más pronto o más tarde, aparecerá un problema. Como buenos desarrolladores, nuestro objetivo es intentar que llegue el menor número de defectos al usuario, para que así no nos saquen los colores y nuestra organización o el cliente no se vea perjudicado por un fallo importante.

Depurador

El depurador (*debugger*) o herramienta de depuración (*debugging tool*) es el programa utilizado para ayudar a depurar software, en nuestro caso particular, *scripts* Lua en Redis.

Los depuradores principalmente proporcionan las siguientes características:

- Ejecución paso a paso (*stepping*). Permite ejecutar una proposición cada vez con objeto de ver

cómo quedan las cosas tras la ejecución de la proposición.

- Puntos de interrupción (*breakpoints*). Permite indicar puntos en los que el depurador detendrá la ejecución.

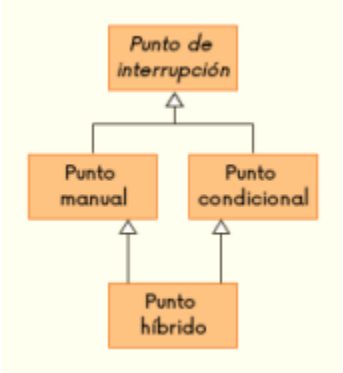
  En ese punto, podremos comprobar los valores de las variables, decidir seguir ejecutando paso a paso o hasta el siguiente punto de interrupción, comprobar la pila de llamadas o finalizar la ejecución.
- Observadores (*watchers*). Permite visualizar el valor de una determinada variable o expresión.
- Consola de consulta (*drawer*). Permite ejecutar expresiones de Lua.
- Pila de llamadas (*call stack*). Permite visualizar qué funciones hay abiertas en el punto bajo depuración.

La idea que se esconde tras un depurador es muy, pero que muy sencilla. Ejecutar un *script* de Lua y, entonces, ejecutarlo proposición a proposición analizando si su flujo de ejecución es el esperado, comprobando los valores de las variables.

Puntos de interrupción

Un punto de interrupción (*breakpoint*) es una línea del programa en el que el depurador debe pausar la ejecución, permitiéndonos así inspeccionar la situación en la que nos encontramos como, por ejemplo, la pila de llamadas y los valores de las variables. Durante una sesión de depuración, podemos fijar tantos puntos de interrupción como sea necesario.

Los puntos de interrupción se clasifican en manuales, condicionales e híbridos.



Un punto de interrupción manual (*manual breakpoint*) es aquel que se asocia a una determinada línea y detiene la ejecución cuando se alcanza ésta. Generalmente, se usa este tipo de punto de interrupción cuando tenemos sospechas de que el problema se encuentra en esa línea o cerca.

En cambio, un punto de interrupción condicional (*conditional breakpoint*) tiene asociada una condición que actúa como interruptor de detención. Cuando se cumple la condición, el punto de interrupción se alcanza, independientemente del punto del programa en el que nos encontramos.

Un punto de interrupción híbrido (*hybrid breakpoint*) está asociado a una determinada línea, al igual que los manuales, pero además tienen asociada una condición. De tal manera que la detención del flujo de ejecución sólo se produce si se alcanza la línea y además se cumple la condición asociada.

Cuantos más tipos de puntos de interrupción soporte nuestro depurador, mejor. Mayor flexibilidad tendremos disponible.

Función redis.breakpoint()

En un *script* Lua de Redis, generalmente los puntos de interrupción se pueden fijar a nivel de línea, a través de la interfaz de usuario proporcionada por el depurador, o bien mediante la función breakpoint() de la biblioteca redis. Cuando se ejecuta esta función, se detendrá la ejecución.

He aquí un ejemplo ilustrativo muy sencillo:

```
if x == 5 then redis.breakpoint() end
```

Esta función es por lo tanto intrusa, pues hay que insertarla en el código fuente del *script*, a diferencia

de otros modos de inserción de puntos de interrupción que no lo son, pues se fijan a nivel del depurador, no del código fuente. Pero aún así, es muy útil saber su existencia y utilizarla cuando es necesario, por ejemplo, en combinación con una sentencia if para simular un punto de interrupción híbrido.

Ejecución paso a paso

La ejecución paso a paso (*stepping*) es la capacidad del depurador de permitir al usuario ejecutar las proposiciones poco a poco, según nuestra necesidad, tras un punto de interrupción. Cuando se detiene la ejecución del *script* en un determinado punto de interrupción, el depurador nos permite continuar la ejecución de varias formas:

- Continuar (*continue* o *resume*). Cuando reanudamos la ejecución con el comando continue o resume, lo que le estamos diciendo al depurador es: *continua la ejecución hasta que te encuentres con un punto de interrupción o el final del programa*.
- Pasar por encima (*step over*). Si usamos step over, lo que hacemos es decirle que ejecute la línea actual y se detenga de nuevo tras ella.

  Así pues, el depurador se detendrá en la siguiente línea de código, ejecutando todo aquello que sea necesario de la línea en curso. Si ésta tiene llamadas a función, las ejecutará sin pararse en ellas.
- Entrar (*step into*). En ocasiones, la línea actual tiene llamadas a funciones. Si deseamos detener el flujo de ejecución en la primera línea del cuerpo de cualquier función invocada en la línea en curso, debemos utilizar el comando step in.

  Si la línea actual no ejecuta ninguna llamada a función, tiene el mismo comportamiento que step over.
- Salir (*step out*). Si estamos dentro de una función y deseamos salir de su cuerpo y detenernos en la línea que provocó su llamada, se puede usar step out.

Observadores

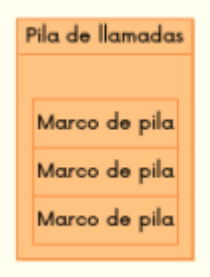
Un observador (*watcher*) es una funcionalidad del depurador mediante la cual se puede solicitar la monitorización de una determinada variable o expresión. Cada vez que se ejecuta una línea del programa, el observador muestra el valor de la variable o expresión bajo observación. Lo que permite visualizar su valor fácilmente.

Consola de consulta

Generalmente, los depuradores proporcionan una consola de consulta (*drawer*) a través de la cual ejecutar expresiones con el contexto de ejecución que hay en el punto en el que nos encontramos detenidos. Estas expresiones pueden ser simples consultas del valor de una variable o de una determinada expresión, similar a un observador. Pero se pueden hacer interactivamente.

Pila de llamadas

Una función puede invocar otra función y ésta a su vez a otra y, así, sucesivamente. La pila de llamadas (*call stack*) es una estructura de datos, que utiliza internamente el motor, en la que deposita las distintas llamadas que están activas actualmente. Cada vez que se invoca una función, se deposita su información en esta pila; a su vez, si la función invoca otra, la información de esta nueva llamada también se deposita en la pila, justo encima de la anterior. Así, es fácil para el motor recurrir a la pila de llamadas para determinar el orden de las invocaciones.



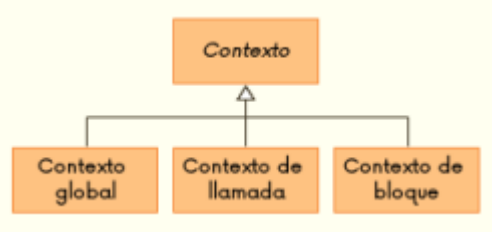
Cada apilamiento o entrada de la pila representa una invocación y se conoce formalmente como marco de pila (*stack frame*). Este marco contiene básicamente tres cosas: el objeto función invocado, la proposición que ha generado la llamada y un contexto de ejecución privado.

El objeto función invocado no es más que la función invocada.

La proposición que ha generado la llamada es eso, la proposición que ha invocado la función. Esta información permitirá al motor que, terminada la ejecución de la función activa, pueda volver atrás y continuar con el flujo de ejecución, es decir, con la proposición que invocó la función.

Finalmente, cada marco de pila contiene un contexto de ejecución privado (*private execution context*), también conocido como tabla de símbolos (*symbol table*), el conjunto de variables locales de la función invocada y los valores particulares que tiene cada una de ellas para esa invocación. Este contexto contiene los parámetros de la función y cada una de las variables locales definidas dentro de ella, junto con sus respectivos valores para esa invocación. Cuando la función finaliza, o lo que es lo mismo, cuando finaliza la invocación, el contexto de ejecución se destruye automáticamente. En algunos lenguajes, el contexto de ejecución se conoce también como ámbito (*scope*).

Básicamente, se puede distinguir tres tipos de contextos de ejecución: el global, el de llamada y el de bloque.



El contexto global (*global context*) o ámbito global (*global scope*) es aquel que contiene las variables globales, accesibles en cualquier función. Por su parte, un contexto de llamada (*call context*) o ámbito de llamada (*call scope*) es aquel que se crea con cada invocación para almacenar las variables específicas de esa llamada. Finalmente, el contexto de bloque (*block context*) o ámbito de bloque (*block scope*) es aquel que se crea cuando se utiliza un bloque para definir variables cuyo ámbito es el bloque que las define.

En todo momento, puede haber un único contexto de ejecución global, pero varios de llamada o de bloque. A la cadena particular de contextos, desde el más local al global, se le conoce como cadena de ámbitos (*scope chain*). Cuando el motor realiza una búsqueda de identificador (*identifier lookup*) para encontrar una variable, consulta la cadena de ámbitos actual, desde la más interna o local a la más externa o global. En caso de no encontrar ninguna variable tras consultar todos los ámbitos de la cadena, propagará un error.

Es importante entender el funcionamiento de la pila. En todo momento, un hilo de ejecución sólo puede tener una función activa, que consistirá en la última función invocada, o sea, la que se encuentra en lo alto de la pila. A esta función, se la conoce formalmente como función activa (*active function*); al resto de funciones que se encuentran en la pila, como funciones en espera (*standby functions*).

Cuando la función activa finaliza, tanto si lo hace normal como anormalmente, el motor desapila o elimina su marco de pila, esto es, el marco que se encuentra en lo alto de la pila, y entonces continúa con la ejecución de la siguiente función, la cual se reconvertirá en la nueva activa. A esta operación de supresión del marco, se la conoce formalmente como limpieza de marco de pila (*stack frame clear*) o desapilamiento de marco de pila (*stack frame pop*) y conlleva, claro está, la reactivación de la función que invocó la función que acaba de terminar, para así continuar con su ejecución.

LDB

El depurador de Lua de Redis, también conocido como LDB, es un depurador de línea de comandos proporcionado por redis-cli. Siendo la interfaz de usuario el propio comando redis-cli, lo que permite la depuración remota.

Las principales características del depurador son:

- Es muy sencillo.
- Permite definir puntos de interrupción manuales.
- Permite la ejecución paso a paso.
- Permite consultar la pila de llamadas.
- Proporciona una consola de consulta.

El depurador *no* proporciona observadores.

Depuración de un script

Para ejecutar un *script* Lua en modo depuración, hay que ejecutarlo con el comando redis-cli y la opción --ldb. Mediante la opción --ldb, le solicitamos a redis-cli que ejecute un *script* dado monitorizándolo con el depurador. La sintaxis básica es:

```
redis-cli --ldb --eval script.lua clave1 clave2… , arg1 arg2…
```

He aquí un ejemplo:

```
$ cat script.lua
local res = ARGV[1] + ARGV[2]
return res
$ redis-cli --ldb --eval ./script.lua , 123 456
Lua debugging session started, please use:
quit    -- End the session.
restart -- Restart the script in debug mode again.
Help    -- Show Lua script debugging commands.

* Stopped at 1, stop reason = step over
-> 1 local res = ARGV[1] + ARGV[2]
lua debugger>
```

Observe que redis-cli entra en modo depuración deteniendo la ejecución del *script* en su primera línea, esperando que le digamos qué debe hacer. Para obtener la lista de comandos del depurador, escriba el comando h o help:

```
lua debugger> h
Redis Lua debugger help:
[h]elp               Show this help.
[s]tep               Run current line and stop again.
[n]ext               Alias for step.
[c]continue          Run till next breakpoint.
[l]list              List source code around current line.
[l]list [line]       List source code around [line].
                     line = 0 means: current position.
[l]list [line] [ctx] In this form [ctx] specifies how many lines
                     to show before/after [line].
[w]hole              List all source code. Alias for 'list 1 1000000'.
[p]rint              Show all the local variables.
[p]rint <var>        Show the value of the specified variable.
                     Can also show global vars KEYS and ARGV.
[b]reak              Show all breakpoints.
[b]reak <line>       Add a breakpoint to the specified line.
[b]reak -<line>      Remove breakpoint from the specified line.
[b]reak 0            Remove all breakpoints.
[t]race              Show a backtrace.
[e]eval <code>       Execute some Lua code (in a different callframe).
[r]edis <cmd>        Execute a Redis command.
[m]axlen [len]       Trim logged Redis replies and Lua var dumps to len.
                     Specifying zero as <len> means unlimited.
[a]bort              Stop the execution of the script. In sync
```



```
                    mode dataset changes will be retained.

Debugger functions you can call from Lua scripts:
redis.debug()        Produce logs in the debugger console.
redis.breakpoint()   Stop execution like if there was a breakpoing.
                     in the next line of code.
lua debugger>
```

Paso de daves y argumentos

redis-cli ejecuta el contenido del archivo mediante un comando EVAL. ¿Recuerda este comando? Se podía pasar parámetros al *script* en forma de claves y argumentos. Éstos se puede pasar a redis-cli si son necesarios para la depuración. Hay que indicarlos tras el archivo en un formato similar al siguiente:

```
clave1 clave2… , arg1 arg2…
```

Observe la coma, es importantísima. Actúa como separador entre las claves y los argumentos. Debe aparecer siempre que se pase argumentos. Si no se pasa claves, la sintaxis a seguir es:

```
, arg1 arg2…
```

No olvide que la coma está delimitada por espacios.

Definición de puntos de interrupción

Recordemos que podemos definir un punto de interrupción mediante redis.breakpoint() o bien hacerlo directamente en el depurador. La primera opción es intrusa porque se encuentra en el código fuente del *script*, mientras que la segunda no lo es.

Definición de puntos de interrupción manuales

Para definir un punto de interrupción manual en el depurador, recordemos, aquellos que están asociados a una determinada línea de código, no hay más que usar el comando b o breakpoint junto con el número de línea al que añadir el punto de interrupción.

Para obtener el listado del *script*, podemos usar el comando w o whole:

```
lua debugger> w
-> 1  local res = ARGV[1] + ARGV[2]
   2  return res
lua debugger>
```

Veamos cómo fijar un punto de interrupción en la línea 2:

```
lua debugger> b 2
-> 1  local res = ARGV[1] + ARGV[2]
  #2  return res
lua debugger>
```

Para suprimir un punto de interrupción, hay que usar también el comando b o breakpoint, pero con el número de línea precedido de un guión. Ejemplo:

```
lua debugger> b -2
Breakpoint removed.
lua debugger>
```

Si deseamos suprimir todos los puntos de interrupción, usaremos b 0.

Ejecución paso a paso

Una vez sabemos definir puntos de interrupción en nuestra sesión de depuración, lo siguiente es explicar cómo ejecutar el código paso a paso. Para ello, disponemos de dos opciones: s o step, también conocido como n o next, para ejecutar la proposición actual y parar; y c o continue, para ejecutar hasta el siguiente punto de interrupción o el final del *script*, lo que primero ocurra.

La ejecución de un *script* en modo depuración comienza con la detención en la primera línea del *script*. Por lo que habrá que hacer uso de los comandos de ejecución paso a paso para comenzar la ejecución.

Ejemplo ilustrativo:

```
$ redis-cli --ldb --eval ./script.lua , 123 456
Lua debugging session started, please use:
quit    -- End the session.
```



```
restart -- Restart the script in debug mode again.
help    -- Show Lua script debugging commands.

* Stopped at 1, stop reason = step over
-> 1  local res = ARGV[1] + ARGV[2]
lua debugger> n
* Stopped at 2, stop reason = step over
-> 2  return res
lua debugger> n

(integer) 579

(Lua debugging session ended -- dataset changes rolled back)

127.0.0.1:6379> quit
$
```

Ejecución de expresiones mediante la consola de consulta

Una característica muy interesante de los depuradores es la consola de consulta, mediante la cual podemos consultar el contexto de ejecución actual.

Por ejemplo, para conocer el contenido de una variable, podemos usar el comando p o print:

```
lua debugger> p ARGV
<value> {"123"; "456"}
lua debugger>
```

Si deseamos mostrar todas las variables locales, podemos usar p o print sin nada más:

```
lua debugger> p
<value> x = "123"
<value> y = "456"
<value> res = 579
lua debugger>
```

Pila de llamadas

La pila de llamadas se puede visualizar mediante el comando t o trace. A continuación, se muestra una pila de llamadas generada para una función recursiva rfibo():

```
lua debugger> t
In rfibo:
-> 5  return rfibo(n - 1) + rfibo(n - 2)
From rfibo:
-> 5  return rfibo(n - 1) + rfibo(n - 2)
From rfibo:
-> 5  return rfibo(n - 1) + rfibo(n - 2)
From top level:
-> 5  return rfibo(n - 1) + rfibo(n - 2)
lua debugger>
```

Modo de ejecución del depurador

Las sesiones de depuración se pueden abrir bajo uno de dos modos: bifurcado o síncrono. Bajo el modo bifurcado (*forked mode*), el predeterminado, cuando finaliza la sesión, los cambios realizados por el *script* se desharán. Al finalizar el *script*, el depurador mostrará un mensaje similar al siguiente:

```
(Lua debugging session ended -- dataset changes rolled back)
```

En cambio, si ejecutamos el *script* en modo síncrono (*synchronous mode*), los cambios se mantendrán. Se consigue invocando el depurador con la opción --ldb-sync-mode en vez de con --ldb.

