Scripts Lua en Redis

Tiempo estimado: 20min

En la lección anterior, presentamos Lua utilizando su propio intérprete. Con una buena base, ya podemos centramos en cómo usarlo en Redis.

Comenzamos la lección introduciendo Lua en Redis. A continuación, mostramos cómo ejecutar *scripts* en Redis y cómo registrarlos para su ejecución posterior con objeto de mejorar el rendimiento. Finalmente, exponemos cómo detener el *script* en ejecución actualmente.

Al finalizar la lección, el estudiante sabrá:

- Cómo ejecutar *scripts* escritos en Lua.
- Cómo registrar *scripts* para mejorar el rendimiento.
- Cómo utilizar la librería redis para ejecutar comandos Redis en los *scripts*.
- Cómo detener el *script* actual en ejecución.

Introducción

Redis viene nativamente con un intérprete de Lua. No podemos hacer uso de Lua en el *shell* de Redis directamente. Tenemos que hacerlo a través de comandos específicos. Dentro de estos comandos, podemos ejecutar *scripts* de Lua, dentro de los cuales podemos acceder a la instancia mediante la librería redis.

Redis sólo permite que haya un único *script* Lua en ejecución en toda la instancia. Si durante su ejecución entran otras peticiones, se encolarán y esperarán. Es muy importante no olvidarlo: la instancia sólo puede ejecutar un único *script* Lua en cada momento. Por esta razón, los *scripts* deben ser cortos.

Para evitar bloqueos *in eternum* de un *script*, se define el tiempo de ejecución (*execution time*). Tiempo máximo que tiene el *script* para llevar a cabo su trabajo. Se configura mediante el parámetro de configuración lua-time-limit y de manera predeterminada es 5 segundos. Aunque es un tiempo bastante alto, se recomienda encarecidamente ejecutar *scripts* cortos, muy cortos. Es más, no se recomienda que el *script* acceda directamente al exterior como, por ejemplo, a disco.

Si se alcanza el tiempo máximo de ejecución, el motor de Redis detendrá la ejecución del *script*. Esta interrupción puede dejar la base de datos en un estado inconsistente. Si hace modificaciones y no ha terminado, la base de datos mantendrá los cambios realizados, pero como no se ha llevado a cabo todos, la base de datos reflejará un estado que no es el esperado tras la ejecución del *script*. Si ha realizado todos sus cambios, pero tenía que devolver algo, no lo devolverá, pero la base de datos permanecerá en un estado consistente. Lo único, que el usuario no recibirá el resultado esperado.

Ejecución de scripts

Para ejecutar una proposición o bloque de código Lua en Redis, principalmente se utiliza el comando EVAL:

```
EVAL código 0
EVAL código 0 argumento argumento argumento…
EVAL código númeroClaves clave clave clave… argumento argumento argumento…
```

El código es algo tan sencillo como una cadena con el bloque de código Lua a ejecutar. A continuación, se indica el número de claves, las cuales se indican justo después. Finalmente, se indica los argumentos.

Las claves y los argumentos no son más que valores que se puede acceder dentro del *script*. Redis distingue entre claves y argumentos. Se utiliza las claves (*keys*) para representar claves de pares clave-valor a las que acceder en el bloque Lua; mientras que los argumentos (*arguments*) se destinan para valores concretos a utilizar o comparar.

Dentro del *script*, las claves y los argumentos se acceden mediante las variables globales KEYS y ARGV, ambas de tipo tabla. La primera contiene la lista de claves; mientras que la segunda, la de argumentos pasados en EVAL. Recordemos que, en Lua, los índices de las listas comienzan en 1 y no en 0.

Al igual que los comandos de Redis, EVAL puede devolver un resultado. El bloque autónomo de Lua debe devolver su resultado mediante una sentencia return, como las funciones. Veámoslo mediante un sencillo ejemplo:

```
127.0.0.1:6379> EVAL "return {KEYS[1], KEYS[2]}" 2 "book:1" "book:2"
1) "book:1"
2) "book:2"
127.0.0.1:6379>
```

Desgranemos el comando de ejemplo. Primero se indica el código Lua del *script*, todo ello mediante una cadena de Redis. A continuación, se indica el número de claves que, a continuación, se indican. Eso es lo que significa el valor 2. Finalmente, se indica las claves, en nuestro caso, book:1 y book:2.

El *script* no hace gran cosa. Simplemente, devuelve las claves proporcionadas como argumentos al comando EVAL.

Para conocer la especificación de Lua soportada por la instancia Redis, puede utilizar el siguiente comando:

```
127.0.0.1:6379> EVAL "return _VERSION" 0
"Lua 5.1"
127.0.0.1:6379>
```

Variables y funciones locales

Recordemos que en Lua cuando se asigna un valor a una variable inexistente, el intérprete lo que hace es crearla globalmente. Para crear una variable local, hay que utilizar la sentencia local. Los *scripts* deben definir sus propias variables y funciones como locales, sino se obtendrá un mensaje de error con el siguiente texto *Script attempted to create global variable*.

No hay más que recordar definir toda variable y función del *script* con la sentencia local.

Biblioteca redis

Redis viene de fábrica con varias bibliotecas Lua cargadas nativamente: base, bitop, cmsgpack, math, redis, string, struct y table. La librería redis es específica de Redis y se usa principalmente para la ejecución de comandos Redis dentro del *script* Lua.

redis.call()

Para ejecutar comandos Redis dentro del *script* Lua, hay que utilizar la función redis.call():

```
function redis.call(comando, ...)
```

La función espera el nombre del comando seguido de sus argumentos. Para ir abriendo boca, he aquí unos ejemplos ilustrativos:

```
redis.call("HGET", "libro:123", "title")
redis.call("HSET", "libro:123", "clicks", 101)
```

Ahora, un ejemplo más completo:

```
127.0.0.1:6379> EVAL "return redis.call('HGETALL', KEYS[1])" 1 "web:2"
1) "url"
2) "/dos"
3) "vistas"
4) "320"
127.0.0.1:6379>
```

Por convenio y buenas prácticas, no se recomienda utilizar claves ni valores en la función redis.call() directamente, sino a través de las variables globales KEYS y ARGV. Por lo que hay que pasarlos en el comando EVAL. Tal como muestra el ejemplo anterior.

Cuando en el *script* se accede a pares clave-valor, los valores se convertirán de tipos Redis a tipos Lua como sigue:

Tipo Redis	Tipo Lua

Cadena	string
Número	number
Conjunto	table
Array asociativo	table
JSON (módulo ReJSON)	string
nil	false

Así pues, si la clave consultada con redis.call() es de tipo cadena, aunque tenga un número, devolverá el valor en una cadena Lua. Si tiene que usar su valor como número, convierta lo recibido a number mediante la función tonumber() de Lua. Esto suele producir errores a los recien llegados.

Pero ojo, si el comando devuelve un número como sucede con el comando SCARD el cual, recordemos, devuelve el número de elementos de un conjunto, redis.call() devolverá un number.

Observe que cuando el comando Redis devuelve un nil, éste se convertirá a false.

redis.pcall()

Cuando redis.call() propaga un error, lo hace mediante un error Lua. Es posible utilizar una segunda función, redis.pcall(), mediante la que no propagar error. Su signatura es como sigue:

```
function redis.pcall(comando, ...)
```

Cuando el motor de Redis propaga un error, la función pcall() lo captura y devuelve como resultado una tabla con el campo err conteniendo el mensaje de error. No se propaga ningún error del *script* al comando EVAL; es más, se continuará con la ejecución del resto del *script*. En cambio, con call() se detendrá el *script* y se propagará el error.

Esto se ve mucho mejor mediante un sencillo ejemplo ilustrativo:

```
127.0.0.1:6379> GE "clave"
(error) ERR unknown command 'GE'
127.0.0.1:6379> EVAL "redis.call('GE'); return 'terminado'" 0
(error) ERR Error running script (call to f_1a165dd11648385d97a56a154182d153d47accee):
@user_script:1: @user_script: 1: Unknown Redis command called from Lua script
127.0.0.1:6379> EVAL "redis.pcall('GE'); return 'terminado'" 0
"terminado"
127.0.0.1:6379>
```

Como puede observar, cuando redis.call() se encuentra con un error, lo propaga al comando EVAL, finalizando el *script* con error. En cambio, redis.pcall() lo captura, lo devuelve como resultado y deja continuar el resto del *script*.

redis.sha1hex()

La función redis.sha1hex() calcula la huella SHA1 de una cadena:

```
function redis.sha1hex(texto)
```

Ejemplo:

```
127.0.0.1:6379> EVAL "return redis.sha1hex('Texto de ejemplo')" 0
"7c7675064211cabbfc00a996f9f613b4ed9d6cae"
127.0.0.1:6379>
```

redis.log()

La función redis.log() envía una entrada al registro de Redis:

```
function redis.log(level, msg)
```

Parámetro	Tipo de datos	Descripción
level	number	Nivel de la entrada. Hay que utilizar las variables: • redis.LOG_DEBUG • redis.LOG_VERBOSE • redis.LOG_NOTICE • redis.LOG_WARNING

msg	string	Texto de la entrada.

Caché de scripts

El comando EVAL ejecuta *scripts* Lua dinámicamente, no mantiene internamente el código para ayudar a que próximas ejecuciones se ejecuten con más rapidez. Cuando somos conscientes de que un *script* se ejecuta con frecuencia, lo que hacemos es añadirlo a la caché de scripts (*script cache*), una estructura de datos interna de la instancia en la que se almacena *scripts* de Lua. Los *scripts* se identifican mediante un nombre único, concretamente su huella SHA1. La idea de esta caché es simular el concepto de procedimiento o función almacenada de otros motores de bases de datos, aunque mediante una manera particular de hacerlo. Muy particular, todo hay que decirlo.

Carga de scripts

La carga de un script (*script load*) es la operación mediante la cual se registra o da de alta un *script* Lua en la caché. Se realiza con el comando SCRIPT LOAD:

```
SCRIPT LOAD código
```

Este comando lee el código, lo analiza y lo almacena internamente en la caché, devolviendo su identificador. De esta manera, las invocaciones del *script* son más rápidas, ya que se analiza una única vez, en el momento de su carga. Esta comando simplemente lo da a conocer a la instancia, no lo ejecuta.

A continuación, se muestra un ejemplo ilustrativo que registra un *script* con el que obtener la versión actual de Redis:

```
127.0.0.1:6379> SCRIPT LOAD "return string.match(redis.call('INFO', 'Server'),
'redis_version[^\\r\\n]+')"
"dab89f0f0693e9c4e99685d227c714df6778ad1c"
127.0.0.1:6379>
```

No olvidemos que el valor devuelto por el comando es el identificador que *debemos* usar para invocar el *script*. Concretamente, su huella SHA1. Ésta es la diferencia con otros motores de bases de datos, donde se definen funciones o procedimientos, usándose su nombre, no su huella, en las invocaciones.

Ejecución de scripts

Para ejecutar un *script* cargado en la caché, hay que utilizar el comando EVALSHA:

```
EVALSHA identificador 0
EVALSHA identificador 0 argumento argumento argumento…
EVALSHA identificador númeroClaves clave clave clave… argumento argumento argumento…
```

El comando es similar a EVAL, salvo que en vez de proporcionar código Lua, se indica el identificador del *script* devuelto por SCRIPT LOAD.

Veamos un ejemplo:

```
127.0.0.1:6379> EVALSHA "dab89f0f0693e9c4e99685d227c714df6778ad1c" 0
"redis_version:3.9.102"
127.0.0.1:6379>
```

Comprobación de existencia

Para saber si un *script* se encuentra dado de alta en la caché, se puede utilizar el comando SCRIPT EXISTS:

```
SCRIPT EXISTS identificador
SCRIPT EXISTS identificador identificador identificador…
```

Devuelve 1 si existe; 0 en otro caso.

Ejemplo:

```
127.0.0.1:6379> SCRIPT EXISTS "dab89f0f0693e9c4e99685d227c714df6778ad1c"
1) (integer) 1
127.0.0.1:6379> SCRIPT EXISTS "0000000000000000000000000000000000000000"
1) (integer) 0
127.0.0.1:6379>
```


Supresión de script

Actualmente, no es posible suprimir un único *script* de la caché. Lo que sí se puede hacer es suprimir *todos* los registrados. Para ello, usar el comando SCRIPT FLUSH:

```
SCRIPT FLUSH
```

Detención abrupta de script

Como indicamos anteriormente, Redis sólo puede tener un único *script* Lua en ejecución en cada momento. Cuando comienza uno, no para hasta que se detiene, ya sea formal o informalmente. En contadas ocasiones, es necesario detener un *script*, esto es, el *script* en ejecución actualmente. Para este fin, disponemos del comando SCRIPT KILL:

```
SCRIPT KILL
```

Observe que no tiene identificador del *script* a finalizar. Como sólo puede haber uno, no hace falta ser más concretos.

