Redis Lua scripts debugger

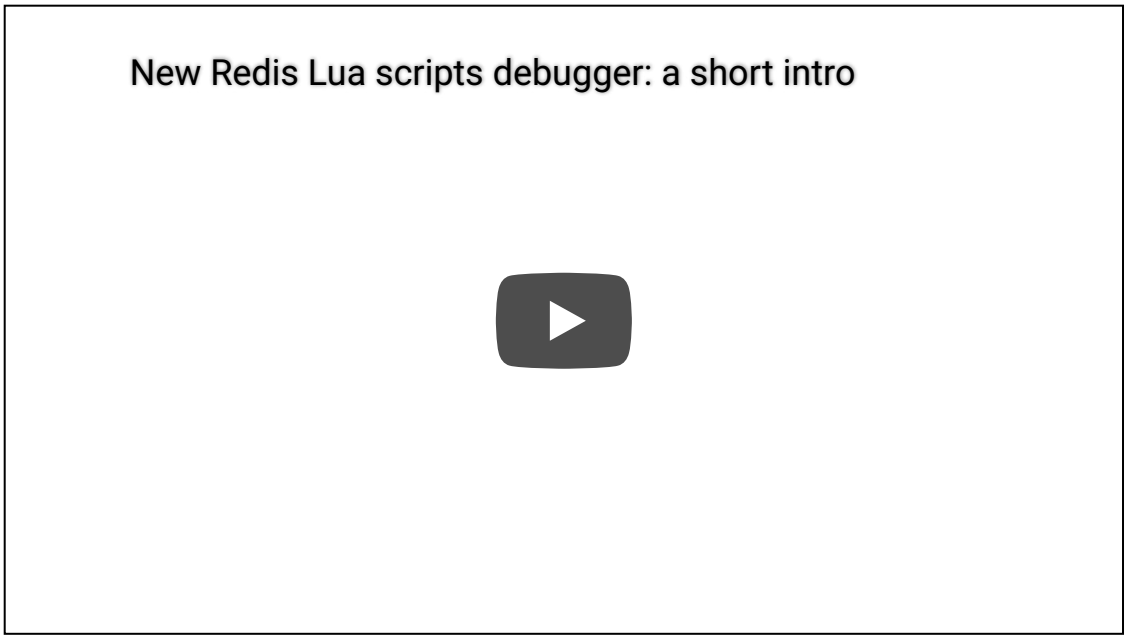
Starting with version 3.2 Redis includes a complete Lua debugger, that can be used in order to make the task of writing complex Redis scripts much simpler.

The Redis Lua debugger, codename LDB, has the following important features:

- It uses a server-client model, so it's a remote debugger. The Redis server acts as the debugging server, while the default client is `redis-cli`. However other clients can be developed by following the simple protocol implemented by the server.
- By default every new debugging session is a forked session. It means that while the Redis Lua script is being debugged, the server does not block and is usable for development or in order to execute multiple debugging sessions in parallel. This also means that changes are **rolled back** after the script debugging session finished, so that's possible to restart a new debugging session again, using exactly the same Redis data set as the previous debugging session.
- An alternative synchronous (non forked) debugging model is available on demand, so that changes to the dataset can be retained. In this mode the server blocks for the time the debugging session is active.
- Support for step by step execution.
- Support for static and dynamic breakpoints.
- Support from logging the debugged script into the debugger console.
- Inspection of Lua variables.
- Tracing of Redis commands executed by the script.
- Pretty printing of Redis and Lua values.
- Infinite loops and long execution detection, which simulates a breakpoint.

Quick start

A simple way to get started with the Lua debugger is to watch this video introduction:



Important note: please make sure to avoid debugging Lua scripts using your Redis production server. Use a development server instead. Also note that using the synchronous debugging mode (which is NOT the default) results into the Redis server blocking for all the time the debugging session lasts.

To start a new debugging session using `redis-cli` do the following steps:

1. Create your script in some file with your preferred editor. Let's assume you are editing your Redis Lua script located at `/tmp/script.lua`.
2. Start a debugging session with:

   ./redis-cli --ldb --eval /tmp/script.lua

Note that with the `--eval` option of `redis-cli` you can pass key names and arguments to the script, separated by a comma, like in the following example:

```
./redis-cli --ldb --eval /tmp/script.lua mykey somekey , arg1 arg2
```

You'll enter a special mode where `redis-cli` no longer accepts its normal commands, but instead prints a help screen and passes the unmodified debugging commands directly to Redis.

The only commands which are not passed to the Redis debugger are:

- `quit` -- this will terminate the debugging session. It's like removing all the breakpoints and using the `continue` debugging command. Moreover the command will exit from `redis-cli`.
- `restart` -- the debugging session will restart from scratch, **reloading the new version of the script from the file**. So a normal debugging cycle involves modifying the script after some debugging, and calling `restart` in order to start debugging again with the new script changes.

- `help` -- this command is passed to the Redis Lua debugger, that will print a list of commands like the following:

```
lua debugger> help
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 l
                      Specifying zero as <len> means unlimited.
[a]abort              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 as if there was a breakpoint in t
                      next line of code.
```

Note that when you start the debugger it will start in **stepping mode**. It will stop at the first line of the script that actually does something before executing it.

From this point you usually call `step` in order to execute the line and go to the next line. While you step Redis will show all the commands executed by the server like in the following example:

```
* Stopped at 1, stop reason = step over
-> 1   redis.call('ping')
lua debugger> step
<redis> ping
<reply> "+PONG"
* Stopped at 2, stop reason = step over
```

The `<redis>` and `<reply>` lines show the command executed by the line just executed, and the reply from the server. Note that this happens only in stepping mode. If you use `continue` in order to execute the script till the next breakpoint, commands will not be dumped on the screen to prevent too much output.

Termination of the debugging session

When the scripts terminates naturally, the debugging session ends and `redis-cli` returns in its normal non-debugging mode. You can restart the session using the `restart` command as usual.

Another way to stop a debugging session is just interrupting `redis-cli` manually by pressing `Ctrl+C`. Note that also any event breaking the connection between `redis-cli` and the `redis-server` will interrupt the debugging session.

All the forked debugging sessions are terminated when the server is shut down.

Abbreviating debugging commands

Debugging can be a very repetitive task. For this reason every Redis debugger command starts with a different character, and you can use the single initial character in order to refer to the command.

So for example instead of typing `step` you can just type `s`.

Breakpoints

Adding and removing breakpoints is trivial as described in the online help. Just use `b 1 2 3 4` to add a breakpoint in line 1, 2, 3, 4. The command `b 0` removes all the breakpoints. Selected breakpoints can be removed using as argument the line where the breakpoint we want to remove is, but prefixed by a minus sign. So for example `b -3` removes the breakpoint from line 3.

Note that adding breakpoints to lines that Lua never executes, like declaration of local variables or comments, will not work. The breakpoint will be added but since this part of the script will never be executed, the program will never stop.

Dynamic breakpoints

Using the `breakpoint` command it is possible to add breakpoints into specific lines. However sometimes we want to stop the execution of the program only when something special happens. In order to do so, you can use the `redis.breakpoint()` function inside your Lua script. When called it simulates a breakpoint in the next line that will be executed.

```
if counter > 10 then redis.breakpoint() end
```

This feature is extremely useful when debugging, so that we can avoid continuing the script execution manually multiple times until a given condition is encountered.

Synchronous mode

As explained previously, but default LDB uses forked sessions with rollback of all the data changes operated by the script while it has being debugged. Determinism is usually a good thing to have during debugging, so that successive debugging sessions can be started without having to reset the database content to its original state.

However for tracking certain bugs, you may want to retain the changes performed to the key space by each debugging session. When this is a good idea you should start the debugger using a special option, `ldb-sync-mode`, in `redis-cli`.

```
./redis-cli --ldb-sync-mode --eval /tmp/script.lua
```

Note that the Redis server will be unreachable during the debugging session in this mode, so use with care.

In this special mode, the `abort` command can stop the script half-way taking the changes operated to the dataset. Note that this is different compared to ending the debugging session normally. If you just interrupt `redis-cli` the script will be fully executed and then the session terminated. Instead with `abort` you can interrupt the script execution in the middle and start a new debugging session if needed.

Logging from scripts

The `redis.debug()` command is a powerful debugging facility that can be called inside the Redis Lua script in order to log things into the debug console:

```
lua debugger> list
-> 1   local a = {1,2,3}
   2   local b = false
   3   redis.debug(a,b)
lua debugger> continue
<debug> line 3: {1; 2; 3}, false
```

If the script is executed outside of a debugging session, `redis.debug()` has no effects at all. Note that the function accepts multiple arguments, that are separated by a comma and a space in the output.

Tables and nested tables are displayed correctly in order to make values simple to observe for the programmer debugging the script.

Inspecting the program state with `print` and `eval`

While the `redis.debug()` function can be used in order to print values directly from within the Lua script, often it is useful to observe the local variables of a program while stepping or when stopped into a breakpoint.

The `print` command does just that, and performs lookup in the call frames starting from the current one back to the previous ones, up to top-level. This means that even if we are into a nested function inside a Lua script, we can still use `print foo` to look at the value of `foo` in the context of the calling function. When called without a variable name, `print` will print all variables and their respective values.

The `eval` command executes small pieces of Lua scripts **outside the context of the current call frame** (evaluating inside the context of the current call frame is not possible with the current Lua internals). However you can use this command in order to test Lua functions.

```
lua debugger> e redis.sha1hex('foo')
<retval> "0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33"
```

Debugging clients

LDB uses the client-server model where the Redis server acts as a debugging server that communicates using RESP. While `redis-cli` is the default debug client, any client can be used for debugging as long as it meets one of the following conditions:

1. The client provides a native interface for setting the debug mode and controlling the debug session.

2. The client provides an interface for sending arbitrary commands over RESP.
3. The client allows sending raw messages to the Redis server.

For example, the Redis plugin for ZeroBrane Studio integrates with LDB using redis-lua. The following Lua code is a simplified example of how the plugin achieves that:

```
local redis = require 'redis'

-- add LDB's Continue command
redis.commands['ldbcontinue'] = redis.command('C')

-- script to be debugged
local script = [[
  local x, y = tonumber(ARGV[1]), tonumber(ARGV[2])
  local result = x * y
  return result
]]

local client = redis.connect('127.0.0.1', 6379)
client:script("DEBUG", "YES")
print(unpack(client:eval(script, 0, 6, 9)))
client:ldbcontinue()
```